Java 1.8 new features

1) Lambda expressions

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

2) Method References

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

3) Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

4) Stream API

Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.

Stream is functional in nature. Operations performed on a stream does not modify it's source.

For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

Stream is lazy and evaluates code only when required.

The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

5) Stream Filter

Java stream provides a method filter() to filter stream elements on the basis of given predicate.

Suppose you want to get only even elements of your list then you can do this easily with the help of filter method.

This method takes predicate as an argument and returns a stream of consisting of resulted elements.

6) Date and Time API

The java.time, java.util, java.sql and java.text packages contains classes for representing date and time.

7) Default methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

8) Java forEach loop

forEach() is use to iterate the elements.

It is defined in Iterable and Stream interface.

It is a default method defined in the Iterable interface.

Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface.

So, you can pass lambda expression as an argument.

9) Java Collectors

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

1) Lambda Expressions

Why use Lambda Expression

To provide the implementation of Functional interface.

Less coding.

Java Lambda Expression Syntax

(argument-list) -> {body}

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
```

One Parameter Syntax

```
(p1) -> {
//Body of single parameter lambda
}
```

Two Parameter Syntax

```
(p1,p2) -> {
//Body of multiple parameter lambda
}
```

Example-1

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Test it Now

Output:Drawing 10

A lambda expression can have zero or any number of arguments.

Example-2

Java Lambda Expression Example: No Parameter

```
interface Sayable
{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

Test it Now

Output: I have nothing to say.

Example-3

Java Lambda Expression Example: Single Parameter

```
interface Sayable
{
    public String say(String name);
}

public class LambdaExpressionExample4{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses
        Sayable s2= name ->{
            return "Hello, "+name;
        };
        System.out.println(s2.say("Sonoo"));
    }
}
```

Test it Now

Output:

```
Hello, Sonoo
Hello, Sonoo
```

Example-4

Java Lambda Expression Example: Multiple Parameters

```
interface Addable
{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

Test it Now

Output:

```
30
300
```

Example-5

Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

```
interface Addable{
    int add(int a,int b);
}


public class LambdaExpressionExample6 {
    public static void main(String[] args) {


        // Lambda expression without return keyword.
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));


        // Lambda expression with return keyword.
        Addable ad2=(int a,int b)->{
                            return (a+b);
                            };
        System.out.println(ad2.add(100,200));
    }
}
```

Test it Now

Output:

```
30
300
```

Example:-6

Foreach Loop

```
import java.util.*;
public class LambdaExpressionExample7
{
    public static void main(String[] args)
{

        List<String> list=new ArrayList<String>();
        list.add("ankit");
        list.add("mayank");
        list.add("irfan");
        list.add("jai");

        list.forEach((n)->System.out.println(n));
    }
}
```

Test it Now

Output:

```
ankit
mayank
```

```
irfan
jai
```

Java Lambda Expression Example: Multiple Statements

Example 7

```
@FunctionalInterface
interface Sayable{
    String say(String message);
}

public class LambdaExpressionExample8{
    public static void main(String[] args) {

        // You can pass multiple statements in lambda expression
        Sayable person = (message)-> {
            String str1 = "I would like to say, ";
            String str2 = str1 + message;
            return str2;
        };
            System.out.println(person.say("time is precious."));
    }
}
```

Test it Now

Output:I would like to say, time is precious.

Java Lambda Expression Example: Creating Thread

Example 8

```
public class LambdaExpressionExample9
{
    public static void main(String[] args)
{
       //Thread Example without lambda
        Runnable r1=new Runnable(){
            public void run(){
                System.out.println("Thread1 is running...");
            }
        };
        Thread t1=new Thread(r1);
        t1.start();
        //Thread Example with lambda
        Runnable r2=()->{
                System.out.println("Thread2 is running...");
        };
        Thread t2=new Thread(r2);
        t2.start();
    }
}
```

Test it Now

Output:

```
Thread1 is running...
Thread2 is running...
```

Java lambda expression can be used in the collection framework.

It provides efficient and concise way to iterate, filter and fetch data.

Example 9

Java Lambda Expression Example: Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class LambdaExpressionExample10
{
    public static void main(String[] args)
{
        List<Product> list=new ArrayList<Product>();
        //Adding Products
        list.add(new Product(1,"HP Laptop",25000f));
        list.add(new Product(3,"Keyboard",300f));
        list.add(new Product(2,"Dell Mouse",150f));
        System.out.println("Sorting on the basis of name...");

        // implementing lambda expression
        Collections.sort(list,(p1,p2)->{
        return p1.name.compareTo(p2.name);
        });
        for(Product p:list){
            System.out.println(p.id+" "+p.name+" "+p.price);
        }      }}
```

Test it Now

Output:

```
Sorting on the basis of name...
2       Dell Mouse  150.0
1       HP Laptop   25000.0
3       Keyboard    300.0
```

Java Lambda Expression Example: Filter Collection Data

Example 10

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class LambdaExpressionExample11{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();
        list.add(new Product(1,"Samsung A5",17000f));
        list.add(new Product(3,"Iphone 6S",65000f));
        list.add(new Product(2,"Sony Xperia",25000f));
        list.add(new Product(4,"Nokia Lumia",15000f));
        list.add(new Product(5,"Redmi4 ",26000f));
        list.add(new Product(6,"Lenevo Vibe",19000f));

        // using lambda to filter data
        Stream<Product> filtered_data = list.stream().filter(p -> p.price > 20000);
        // using lambda to iterate through collection
        filtered_data.forEach(
                product -> System.out.println(product.name+": "+product.price)
        );
    }
}
```

Test it Now

Output:

```

Iphone 6S: 65000.0
Sony Xperia: 25000.0
Redmi4 : 26000.0
```

Example 11

Java Lambda Event Handling Example

```
public class Program1
{
public static void main(String[] args)
{
Runnable r1=new Runnable()
{
        @Override
        public void run() {
                System.out.println("This is a run method");
                        }
};
Thread t1=new Thread(r1);
t1.start();

Runnable r2=()->
{
	System.out.println("My name is Sandip");
};

Thread t2=new Thread(r2);
t2.start();
}
}
```

Date and Time API

java.time.LocalDate class

java.time.LocalTime class

java.time.LocalDateTime class

java.time.MonthDay class

java.time.OffsetTime class

java.time.OffsetDateTime class

java.time.Clock class

java.time.ZonedDateTime class

java.time.ZoneId class

java.time.ZoneOffset class

java.time.Year class

java.time.YearMonth class

java.time.Period class

java.time.Duration class

java.time.Instant class

java.time.DayOfWeek enum

java.time.Month enum

Example 1

```
import java.time.LocalDate;
public class LocalDateExample1
{
  public static void main(String[] args)
{
    LocalDate date = LocalDate.now();
    LocalDate yesterday = date.minusDays(1);
    LocalDate tomorrow = yesterday.plusDays(2);
    System.out.println("Today date: "+date);
    System.out.println("Yesterday date: "+yesterday);
    System.out.println("Tomorrow date: "+tomorrow);
    LocalDate date1 = LocalDate.of(2017, 1, 13);
    System.out.println(date1.isLeapYear());
    LocalDate date2 = LocalDate.of(2016, 9, 23);
    System.out.println(date2.isLeapYear());
    LocalDate date = LocalDate.of(2017, 1, 13);
    LocalDateTime datetime = date.atTime(1,50,9);
    System.out.println(datetime);
LocalDate d1 = LocalDate.now();
        String d1Str = d1.format(DateTimeFormatter.ISO_DATE);
        System.out.println("Date1 in string :  " + d1Str);
        // Example 2
        LocalDate d2 = LocalDate.of(2002, 05, 01);
        String d2Str = d2.format(DateTimeFormatter.ISO_DATE);
        System.out.println("Date2 in string :  " + d2Str);
        // Example 3
        LocalDate d3 = LocalDate.of(2016, 11, 01);
        String d3Str = d3.format(DateTimeFormatter.ISO_DATE);
        System.out.println("Date3 in string :  " + d3Str);
String dInStr = "2011-09-01";
        LocalDate d1 = LocalDate.parse(dInStr);
        System.out.println("String to LocalDate : " + d1);
        // Example 2
        String dInStr2 = "2015-11-20";
        LocalDate d2 = LocalDate.parse(dInStr2);
        System.out.println("String to LocalDate : " + d2);


  }
}
```

Example 2

```
import java.time.LocalTime;
public class LocalTimeExample1 {
  public static void main(String[] args) {
    LocalTime time = LocalTime.now();
    System.out.println(time);
LocalTime time = LocalTime.of(10,43,12);
    System.out.println(time);
LocalTime time1 = LocalTime.of(10,43,12);
    System.out.println(time1);
    LocalTime time2=time1.minusHours(2);
    LocalTime time3=time2.minusMinutes(34);
    System.out.println(time3);
 LocalTime time1 = LocalTime.of(10,43,12);
    System.out.println(time1);
    LocalTime time2=time1.plusHours(4);
    LocalTime time3=time2.plusMinutes(18);
    System.out.println(time3);
 ZoneId zone1 = ZoneId.of("Asia/Kolkata");
    ZoneId zone2 = ZoneId.of("Asia/Tokyo");
    LocalTime time1 = LocalTime.now(zone1);
    System.out.println("India Time Zone: "+time1);
    LocalTime time2 = LocalTime.now(zone2);
    System.out.println("Japan Time Zone: "+time2);
    long hours = ChronoUnit.HOURS.between(time1, time2);
    System.out.println("Hours between two Time Zone: "+hours);
    long minutes = ChronoUnit.MINUTES.between(time1, time2);
    System.out.println("Minutes between two time zone: "+minutes);
  }  }
```

Example 3

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class LocalDateTimeExample1
{
    public static void main(String[] args)
 {
        LocalDateTime now = LocalDateTime.now();
        System.out.println("Before Formatting: " + now);
        DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy
HH:mm:ss");
        String formatDateTime = now.format(format);
        System.out.println("After Formatting: " + formatDateTime);
LocalDateTime datetime1 = LocalDateTime.now();
    DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy
HH:mm:ss");
    String formatDateTime = datetime1.format(format);
    System.out.println(formatDateTime);
 LocalDateTime a = LocalDateTime.of(2017, 2, 13, 15, 56);
    System.out.println(a.get(ChronoField.DAY_OF_WEEK));
    System.out.println(a.get(ChronoField.DAY_OF_YEAR));
    System.out.println(a.get(ChronoField.DAY_OF_MONTH));
    System.out.println(a.get(ChronoField.HOUR_OF_DAY));
    System.out.println(a.get(ChronoField.MINUTE_OF_DAY));
LocalDateTime datetime1 = LocalDateTime.of(2017, 1, 14, 10, 34);
  LocalDateTime datetime2 = datetime1.minusDays(100);
  System.out.println("Before Formatting: " + datetime2);
  DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");
  String formatDateTime = datetime2.format(format);
  System.out.println("After Formatting: " + formatDateTime );
LocalDateTime datetime1 = LocalDateTime.of(2017, 1, 14, 10, 34);
  LocalDateTime datetime2 = datetime1.plusDays(120);
  System.out.println("Before Formatting: " + datetime2);
  DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");
  String formatDateTime = datetime2.format(format);
  System.out.println("After Formatting: " + formatDateTime );


    }
}
```

Example 4

```
import java.time.*;
public class MonthDayExample1
 {
  public static void main(String[] args)
 {
    MonthDay month = MonthDay.now();
    LocalDate date = month.atYear(1994);
    System.out.println(date);
MonthDay month = MonthDay.now();
    boolean b = month.isValidYear(2012);
    System.out.println(b);
MonthDay month = MonthDay.now();
    long n = month.get(ChronoField.MONTH_OF_YEAR);
    System.out.println(n);
MonthDay month = MonthDay.now();
    ValueRange r1 = month.range(ChronoField.MONTH_OF_YEAR);
    System.out.println(r1);
    ValueRange r2 = month.range(ChronoField.DAY_OF_MONTH);
    System.out.println(r2);


  }
}
```

Example 5

```
import java.time.OffsetTime;
import java.time.temporal.ChronoField;
public class OffsetTimeExample1
 {
  public static void main(String[] args)
{
    OffsetTime offset = OffsetTime.now();
    int h = offset.get(ChronoField.HOUR_OF_DAY);
    System.out.println(h);
    int m = offset.get(ChronoField.MINUTE_OF_DAY);
    System.out.println(m);
    int s = offset.get(ChronoField.SECOND_OF_DAY);
    System.out.println(s);
OffsetTime offset = OffsetTime.now();
    int h = offset.getHour();
    System.out.println(h+ " hour");
OffsetTime offset = OffsetTime.now();
    int s = offset.getSecond();
    System.out.println(s+ " second");
  }
}
```

Example 6

```
import java.time.OffsetDateTime;
public class OffsetDateTimeExample1
 {
    public static void main(String[] args)
{
        OffsetDateTime offsetDT = OffsetDateTime.now();
        System.out.println(offsetDT.getDayOfMonth());
  OffsetDateTime offsetDT = OffsetDateTime.now();
        System.out.println(offsetDT.getDayOfYear());
OffsetDateTime offsetDT = OffsetDateTime.now();
        System.out.println(offsetDT.getDayOfWeek());
OffsetDateTime offsetDT = OffsetDateTime.now();
        System.out.println(offsetDT.toLocalDate());
OffsetDateTime offset = OffsetDateTime.now();
    OffsetDateTime value =  offset.minusDays(240);
    System.out.println(value);
OffsetDateTime offset = OffsetDateTime.now();
    OffsetDateTime value =  offset.plusDays(240);
    System.out.println(value);

    }
}
```

Java Stream Example

Price less than 30000

Example 1

```
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
```

```
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        List<Float> productPriceList = new ArrayList<Float>();
        for(Product product: productsList){

            // filtering data of list
            if(product.price<30000){
                productPriceList.add(product.price);    // adding price to a
productPriceList
            }
        }
        System.out.println(productPriceList);   // displaying data
    }
}
```

Output:

```
[25000.0, 28000.0, 28000.0]
```

Java Stream Example: Filtering Collection by using Stream

Here, we are filtering data by using stream.

Example 2

```
import java.util.*;
import java.util.stream.Collectors;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        List<Float> productPriceList2 =productsList.stream()
```

```
                              .filter(p -> p.price > 30000)// filtering data
                              .map(p->p.price)       // fetching price
                              .collect(Collectors.toList()); // collecting as list
        System.out.println(productPriceList2);
    }
}
```

Output:

```
[90000.0]
```

Example 3

Java Stream Iterating Example

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

```
import java.util.stream.*;
public class JavaStreamExample {
    public static void main(String[] args){
        Stream.iterate(1, element->element+1)
        .filter(element->element%5==0)
        .limit(5)
        .forEach(System.out::println);
    }
}
```

Output:

```
5
10
15
20
25
```

Example 4

Java Stream Example: Filtering and Iterating Collection

```
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // This is more compact approach for filtering data
        productsList.stream()
                        .filter(product -> product.price == 30000)
                        .forEach(product -> System.out.println(product.name));
    }
}
```

Output:

```
Dell Laptop
```

Example 5

Java Stream Example : reduce() Method in Collection

This method takes a sequence of input elements and combines them into a single summary result by repeated operation. For example, finding the sum of numbers, or accumulating elements into a list.

In the following example, we are using reduce() method, which is used to sum of all the product prices.

```
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // This is more compact approach for filtering data
        Float totalPrice = productsList.stream()
                    .map(product->product.price)
                    .reduce(0.0f,(sum, price)->sum+price);   // accumulating price
        System.out.println(totalPrice);
        // More precise code
        float totalPrice2 = productsList.stream()
                .map(product->product.price)
                .reduce(0.0f,Float::sum);   // accumulating price, by referring method of
Float class
        System.out.println(totalPrice2);
    }
}
```

Output:

```
201000.0
201000.0
```

Example 6

Java Stream Example: Sum by using Collectors Methods

We can also use collectors to compute sum of numeric values. In the following example, we are using Collectors class and it?s specified methods to compute sum of all the product prices.

```
import java.util.*;
import java.util.stream.Collectors;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // Using Collectors's method to sum the prices.
        double totalPrice3 = productsList.stream()
                        .collect(Collectors.summingDouble(product->product.price));
        System.out.println(totalPrice3);

    }
}
```

Output:

```
201000.0
```

Example 7

Java Stream Example: Find Max and Min Product Price

Following example finds min and max product price by using stream. It provides convenient way to find values without using imperative approach.

```
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // max() method to get max Product price
        Product productA = productsList.stream().max((product1, product2)-
>product1.price > product2.price ? 1: -1).get();
        System.out.println(productA.price);
        // min() method to get min Product price
        Product productB = productsList.stream().min((product1, product2)-
>product1.price > product2.price ? 1: -1).get();
        System.out.println(productB.price);
           }
}
```

Output:

```
90000.0
25000.0
```

Example 8

Java Stream Example: count() Method in Collection

```
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // count number of products based on the filter
        long count = productsList.stream()
                    .filter(product->product.price<30000)
                    .count();
        System.out.println(count);
    }
}
```

Output:

```
3
```

Example 9

stream allows you to collect your result in any various forms. You can get you result as set, list or map and can perform manipulation on the elements.

Java Stream Example : Convert List into Set

```
import java.util.*;
import java.util.stream.Collectors;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}

public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));

        // Converting product List into Set
        Set<Float> productPriceList =
            productsList.stream()
            .filter(product->product.price < 30000)   // filter product on the base of
price
            .map(product->product.price)
            .collect(Collectors.toSet());   // collect it as Set(remove duplicate elements)
        System.out.println(productPriceList);
    }
}
```

Output:

```
[25000.0, 28000.0]
```

Example 10

Java Stream Example : Convert List into Map

```
import java.util.*;
import java.util.stream.Collectors;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}

public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        // Converting Product List into a Map
        Map<Integer,String> productPriceMap =
            productsList.stream()
                        .collect(Collectors.toMap(p->p.id, p->p.name));

        System.out.println(productPriceMap);
    }
}
```

Output:

```
{1=HP Laptop, 2=Dell Laptop, 3=Lenevo Laptop, 4=Sony Laptop, 5=Apple
Laptop}
```

Example 11

Method Reference in stream

```
import java.util.*;
import java.util.stream.Collectors;

class Product{
    int id;
    String name;
    float price;

    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public float getPrice() {
        return price;
    }
}

public class JavaStreamExample {

    public static void main(String[] args) {

        List<Product> productsList = new ArrayList<Product>();

        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));


        List<Float> productPriceList =
                productsList.stream()
                            .filter(p -> p.price > 30000) // filtering data
                            .map(Product::getPrice)         // fetching price by referring
getPrice method
                            .collect(Collectors.toList());  // collecting as list
        System.out.println(productPriceList);
    }
}
```

Output:

```
[90000.0]
```

Example 12

Java Stream filter() example

In the following example, we are fetching and iterating filtered data.

```
import java.util.*;
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        productsList.stream()
                    .filter(p ->p.price> 30000)   // filtering price
                    .map(pm ->pm.price)          // fetching price
                    .forEach(System.out::println);  // iterating price
    }
}
```

Output:90000.0

Example 13

Java Stream filter() example 2

In the following example, we are fetching filtered data as a list.

```
import java.util.*;
import java.util.stream.Collectors;
```

```
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }  }
public class JavaStreamExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        List<Float> pricesList =  productsList.stream()
                    .filter(p ->p.price> 30000)   // filtering price
                    .map(pm ->pm.price)          // fetching price
                    .collect(Collectors.toList());
        System.out.println(pricesList);
    }
}
```

Output:[90000.0]

Java Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

```
interface Sayable{
    // Default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){        // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();   // calling default method
        dm.sayMore("Work is worship");  // calling abstract method
    }
}
```

Output:Hello, this is default method

Work is worship

Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

```
interface Sayable{
    // default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
    // static method
    static void sayLouder(String msg){
        System.out.println(msg);
    }
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){     // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();                    // calling default method
        dm.sayMore("Work is worship");      // calling abstract method
        Sayable.sayLouder("Helloooo...");   // calling static method
    }
}
```

Output:

```
Hello there
Work is worship
Helloooo...
```

Abstract Class vs Java 8 Interface

After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class is almost similar except that you can create constructor in the abstract class whereas you can't do this in interface.

```
abstract class AbstractClass{
    public AbstractClass() {        // constructor
        System.out.println("You can create constructor in abstract class");
    }
    abstract int add(int a, int b); // abstract method
    int sub(int a, int b){      // non-abstract method
        return a-b;
    }
    static int multiply(int a, int b){  // static method
        return a*b;
    }
}
public class AbstractTest extends AbstractClass{
    public int add(int a, int b){        // implementing abstract method
        return a+b;
    }
    public static void main(String[] args) {
        AbstractTest a = new AbstractTest();
        int result1 = a.add(20, 10);    // calling abstract method
        int result2 = a.sub(20, 10);    // calling non-abstract method
        int result3 = AbstractClass.multiply(20, 10); // calling static method
        System.out.println("Addition: "+result1);
        System.out.println("Substraction: "+result2);
        System.out.println("Multiplication: "+result3);
    }
}
```

Output:

```
You can create constructor in abstract class
Addition: 30
Substraction: 10
Multiplication: 200
```

Java forEach loop

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

forEach() Signature in Iterable Interface

```
default void forEach(Consumer<super T>action)
```

Java 8 forEach() example 1

```
import java.util.ArrayList;
import java.util.List;
public class ForEachExample {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hocky");
        System.out.println("------------Iterating by passing lambda expression--------------");
        gamesList.forEach(games -> System.out.println(games));

    }
}
```

Output:

```
------------Iterating by passing lambda expression--------------
Football
Cricket
Chess
Hocky
```

Java 8 forEach() example 2

```
import java.util.ArrayList;
import java.util.List;
public class ForEachExample {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hocky");
        System.out.println("------------Iterating by passing method reference---------------");
        gamesList.forEach(System.out::println);
    }
}
```

Output:

```
------------Iterating by passing method reference---------------
Football
```

```
Cricket
Chess
Hocky
```

Java Stream forEachOrdered() Method

Along with forEach() method, Java provides one more method forEachOrdered(). It is used to iterate elements in the order specified by the stream.

Singnature:

```
void forEachOrdered(Consumer<? super T> action)
```

Java Stream forEachOrdered() Method Example

```
import java.util.ArrayList;
import java.util.List;
public class ForEachOrderedExample {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hocky");
        System.out.println("------------Iterating by passing lambda expression---------------");
        gamesList.stream().forEachOrdered(games -> System.out.println(games));
        System.out.println("------------Iterating by passing method reference---------------");
        gamesList.stream().forEachOrdered(System.out::println);
    }
}
```

Output:

```
------------Iterating by passing lambda expression---------------
Football
Cricket
Chess
Hocky
------------Iterating by passing method reference---------------
Football
Cricket
Chess
Hocky
```

Java Collectors Example: Fetching data as a List

```
import java.util.stream.Collectors;
import java.util.List;
import java.util.ArrayList;
class Product{
    int id;
    String name;
    float price;

    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class CollectorsExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        List<Float> productPriceList =
                productsList.stream()
                            .map(x->x.price)         // fetching price
                            .collect(Collectors.toList());  // collecting as list
        System.out.println(productPriceList);
    }
}
```

Output:

```
[25000.0, 30000.0, 28000.0, 28000.0, 90000.0]
```

Java Collectors Example: Converting Data as a Set

```
import java.util.stream.Collectors;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;
classProduct{
    intid;
    String name;
    floatprice;

    public Product(intid, String name, floatprice) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
publicclass CollectorsExample {
    publicstaticvoid main(String[] args) {
        List<Product>productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(newProduct(1,"HP Laptop",25000f));
        productsList.add(newProduct(2,"Dell Laptop",30000f));
        productsList.add(newProduct(3,"Lenevo Laptop",28000f));
        productsList.add(newProduct(4,"Sony Laptop",28000f));
        productsList.add(newProduct(5,"Apple Laptop",90000f));
        Set<Float>productPriceList =
            productsList.stream()
                        .map(x->x.price)        // fetching price
                        .collect(Collectors.toSet());   // collecting as list
        System.out.println(productPriceList);
    }
}
Output:[25000.0, 30000.0, 28000.0, 90000.0]
```

Java Collectors Example: using sum method

```
import java.util.stream.Collectors;
import java.util.List;
import java.util.ArrayList;
class Product{
    int id;
    String name;
    float price;

    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class CollectorsExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        Double sumPrices =
                productsList.stream()
                            .collect(Collectors.summingDouble(x->x.price));  // collecting as list
        System.out.println("Sum of prices: "+sumPrices);
        Integer sumId =
                productsList.stream().collect(Collectors.summingInt(x->x.id));
        System.out.println("Sum of id's: "+sumId);
    }
}
```

Output:

```
Sum of prices: 201000.0
Sum of id's: 15
```

Java Collectors Example: Getting Product Average Price

```
import java.util.stream.Collectors;
import java.util.List;
import java.util.ArrayList;
class Product{
    int id;
    String name;
    float price;

    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class CollectorsExample {
    public static void main(String[] args) {
        List<Product> productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        Double average = productsList.stream()
                    .collect(Collectors.averagingDouble(p->p.price));
        System.out.println("Average price is: "+average);
    }
}
```

Output:

```
Average price is: 40200.0
```

Java Collectors Example: Counting Elements

```
import java.util.stream.Collectors;
import java.util.List;
import java.util.ArrayList;
class Product{
    intid;
    String name;
    floatprice;

    public Product(intid, String name, floatprice) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
    publicint getId() {
        returnid;
    }
    public String getName() {
        returnname;
    }
    publicfloat getPrice() {
        returnprice;
    }
}
publicclass CollectorsExample {
    publicstaticvoid main(String[] args) {
        List<Product>productsList = new ArrayList<Product>();
        //Adding Products
        productsList.add(new Product(1,"HP Laptop",25000f));
        productsList.add(new Product(2,"Dell Laptop",30000f));
        productsList.add(new Product(3,"Lenevo Laptop",28000f));
        productsList.add(new Product(4,"Sony Laptop",28000f));
        productsList.add(new Product(5,"Apple Laptop",90000f));
        Long noOfElements = productsList.stream()
                              .collect(Collectors.counting());
        System.out.println("Total elements : "+noOfElements);
    }
}
```

Output:Total elements : 5

Java Method References

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Types of Method References

There are following types of method references in java:

Reference to a static method.

Reference to an instance method.

Reference to a constructor.

Types of Java Method References

1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

```
ContainingClass::staticMethodName
```

Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```
interface Sayable{
    void say();
}
public class MethodReference
 {
    public static void saySomething()
{
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args)
{
        // Referring static method
        Sayable sayable = MethodReference::saySomething;
        // Calling interface method
        sayable.say();
    }
}
```

Output: Hello, this is static method.

Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

```
public class MethodReference2 {
    public static void ThreadStatus(){
        System.out.println("Thread is running...");
    }
    public static void main(String[] args) {
        Thread t2=new Thread(MethodReference2::ThreadStatus);
        t2.start();
    }
}
```

Output:Thread is running...

Example 3

You can also use predefined functional interface to refer methods. In the following example, we are using BiFunction interface and using it's apply() method.

```
import java.util.function.BiFunction;
class Arithmetic
{
public static int add(int a, int b)
{
return a+b;
}
}
public class MethodReference3 {
```

```
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
int result = adder.apply(10, 20);
System.out.println(result);
}
}
```

Output:30

Example 4

You can also override static methods by referring methods. In the following example, we have defined and overloaded three add methods.

```
import java.util.function.BiFunction;
class Arithmetic{
public static int add(int a, int b){
return a+b;
}
public static float add(int a, float b){
return a+b;
}
public static float add(float a, float b){
return a+b;
}
}
public class MethodReference4
{
public static void main(String[] args) {
```

```
BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;
BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;
BiFunction<Float, Float, Float>adder3 = Arithmetic::add;
int result1 = adder1.apply(10, 20);
float result2 = adder2.apply(10, 20.0f);
float result3 = adder3.apply(10.0f, 20.0f);
System.out.println(result1);
System.out.println(result2);
System.out.println(result3);
}
}
```

Output:

```
30
30.0
30.0
```

2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

```
containingObject::instanceMethodName
```

Example 1

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

```
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args) {
        InstanceMethodReference methodReference = new
InstanceMethodReference(); // Creating object
        // Referring non-static method using reference
            Sayable sayable = methodReference::saySomething;
        // Calling interface method
            sayable.say();
            // Referring non-static method using anonymous object
            Sayable sayable2 = new InstanceMethodReference()::saySomething; // You
can use anonymous object also
            // Calling interface method
            sayable2.say();
    }
}
```

Output:

```
Hello, this is non-static method.
Hello, this is non-static method.
```

Example 2

In the following example, we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

```
public class InstanceMethodReference2 {
    public void printnMsg(){
        System.out.println("Hello, this is instance method");
    }
    public static void main(String[] args) {
    Thread t2=new Thread(new InstanceMethodReference2()::printnMsg);
        t2.start();
    } }
```

Output:Hello, this is instance method

Example 3

In the following example, we are using BiFunction interface. It is a predefined interface and contains a functional method apply(). Here, we are referring add method to apply method.

```
import java.util.function.BiFunction;
class Arithmetic{
public int add(int a, int b){
return a+b;
}}
public class InstanceMethodReference3 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder = new Arithmetic()::add;
int result = adder.apply(10, 20);
System.out.println(result);
}  }
```

Output:30

3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

ClassName::new

Example

```
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

Output: Hello