Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 3

Lecture 22

Lecture 22

- Method References
- Stream API
- Default Methods

Method References

- Java provides a new feature called method reference in Java 8.
- 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.

1) Reference to a Static Method

You can refer to static method defined in the class.

Syntax

ContainingClass::staticMethodName

Defined a functional interface and referring a static method

```
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();
    }
}
```

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();
    }
}
```

We can also use predefined functional interface to refer methods

```
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);
}
}
```

We can also overload static methods by referring 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);
} }
```

2) Reference to an Instance Method

like static methods, we can also refer instance methods.

Syntax

containingObject::instanceMethodName

```
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
}
```

```
class MyInstanceMethodReference
{
   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();
   }
}
```

Here 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()::print
nMsg);
      t2.start();
   }
}
```

BiFunction interface. It is a predefined interface and contains a functional method apply().

```
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);
} }
```

3) Reference to a Constructor

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

Syntax

ClassName::new

```
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");
    }
}
```

Java 8 Stream

- Java provides a new additional package in Java 8 called java.util.stream.
- This package consists of classes and interfaces allows functional-style operations on the elements.
- We can use stream by importing java.util.stream package.
- 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.

- 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.

Methods	Description
boolean allMatch(Predicate<? super T> predicate)	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
long count()	It returns the count of elements in this stream. This is a special case of a reduction.
Stream<T> distinct()	It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
static <T> Stream<T> empty()	It returns an empty sequential Stream.
Stream<T> filter(Predicate<? super T> predicate)	It returns a stream consisting of the elements of this stream that match the given predicate.

Filtering Collection by using 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 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);
    }
}
```

Java Stream Iterating Example

```
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);
    }
}
```

Filtering and Iterating Collection

```
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));
    }
}
```

Java Stream Example: count() Method

```
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);
    }
}
```

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.

Java Default Method Example

```
1. interface Sayable{
2.     // Default method
3.     default void say(){
4.         System.out.println("Hello, this is default method");
5.     }
6.     // Abstract method
7.     void sayMore(String msg);
8. }
9. public class DefaultMethods implements Sayable{
10.    public void sayMore(String msg){
11. // implementing abstract method
12.        System.out.println(msg);
13.    }
```

```
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


    }
}
```