Learning Java - A Foundational Journey

Session: 14

More on Functional Programming

Objectives

- Explain functional interfaces
- Describe immutability in Java
- Define and explain concurrency in Java
- Explain Recursion in Java

Introduction to Functional Interfaces

- Functional interfaces are one of the key elements to implement functional programming in Java.
- The `java.util.function` package in Java 8 onwards defines functional interfaces that can supply target types for elements such as lambda expressions and method references.
- A functional interface has only one abstract method (which is called as a functional method) and zero or more default methods.

Function 1-3

Function<T, R> is one of these in-built functional interfaces included in the package.

Function<T, R> is a functional interface, it can be applied as an assignment target for a lambda expression or a method reference.

It is a term that describes the signature of the abstract method of a functional interface.

Function 2-3

Default methods:

andThen(Function):

Returns a composed function that initially applies this function to its input and later applies the specified function to the output.

Syntax:

```
default <V> Function<T,V> andThen(Function<? super R,? extends V>
after)
```

where,

T – Is the type of input to the function

R – Is the type of result of the function

V – Is the type of output of the after function and of the composed function

after – Is the function to apply after this function is applied

Function 3-3

compose(Function):

Comparable to andThen(), however in reversed sequence

Syntax:

```
default <V> Function<V,R> compose(Function<? super V,? extends T>
before)
```

where,

R – Is the type of input to the function

V – Is the type of output of the before function and input to the function

T – Is the output of the composed function

before – Is the function to apply before this function is applied

identity():

Returns a function which constantly returns its input argument.

Syntax:

```
static <T> Function<T,T> identity()
```

where,

T – is the type of input and output to the function

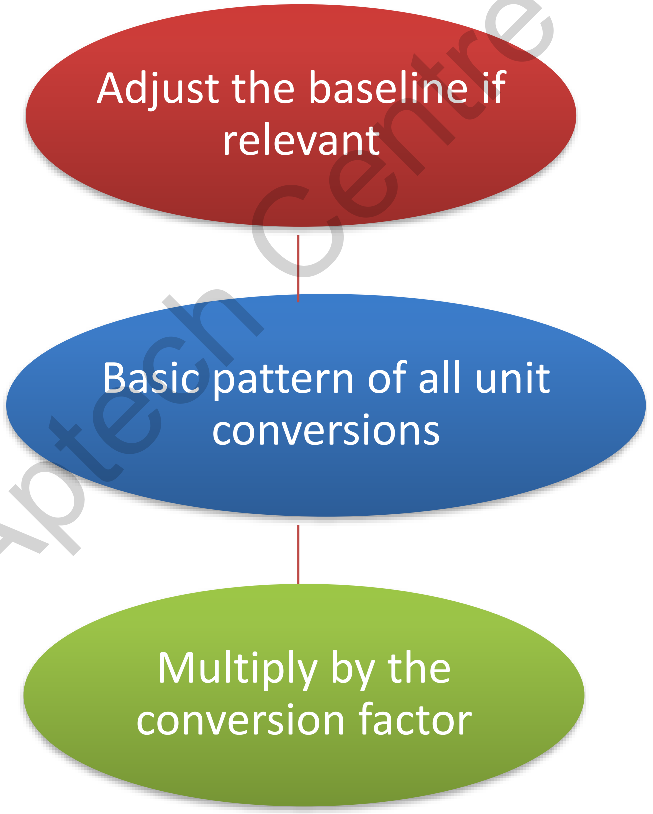
Currying 1-2

Currying is the process of transforming a function having multiple arguments into a function with a single argument.

Syntax:

```
f(a, b) = (g(a)) (b)
```

Here, f is a function, a and b are arguments.


Adjust the baseline if relevant
Basic pattern of all unit conversions
Multiply by the conversion factor

Currying 2-2

```
import java.util.function.BiFunction;
import java.util.function.Function;
public class JavaCurry {
     public void curryfunction() {
          // Create a function that adds 2 integers
          BiFunction<Integer,Integer,Integer> adder = (x, y) -> x + y;

          Function<Integer,Function<Integer,Integer>> currier = x -> y
          -> adder.apply(x,y);

          Function<Integer,Integer> curried = currier.apply(5);

          // To display Results
          System.out.printf("Curry : %d\n", curried.apply(2));
     }
    public void compose() {
         //Function to display the result with + 4
         Function<Integer,Integer> addFour = (x) -> x + 4;
         // function to display the result with * 5
         Function<Integer,Integer> timesFive = (x) -> x * 5;
         // to display the result with n number of times using compose
         Function<Integer,Integer> compose1 = addFour.compose(timesFive);
         //to display the result with add
         Function<Integer,Integer> compose2 = timesFive.compose(addFour);
         // TO display the end Result
         System.out.printf("Times then add: %d\n", compose1.apply(7));
         // (7 * 4) + 5
         System.out.printf("Add then times: %d\n", compose2.apply(7));
         // (7 + 5) * 4
    }
    public static void main(String[] args) {
         new JavaCurry().curryfunction();
         new JavaCurry().compose();
    }
}
```

Output:

```
Curry: 7
Result as Times then add: 39
Result as Add then times: 55
```

Immutability 1-3

Immutability is the capability of an object to resist or prevent change.

It is one of the core concepts in functional programming.

If the state of an object cannot change after it is constructed, it is considered immutable.

```
public class Main {
    public static void main(String[] args) {
        String sample = "immutable";
        System.out.println(sample); // immutable
        change(sample);
        System.out.println(sample); // immutable
    }
    public static void change(String str) {
        str = "mutable";
    }
}
```

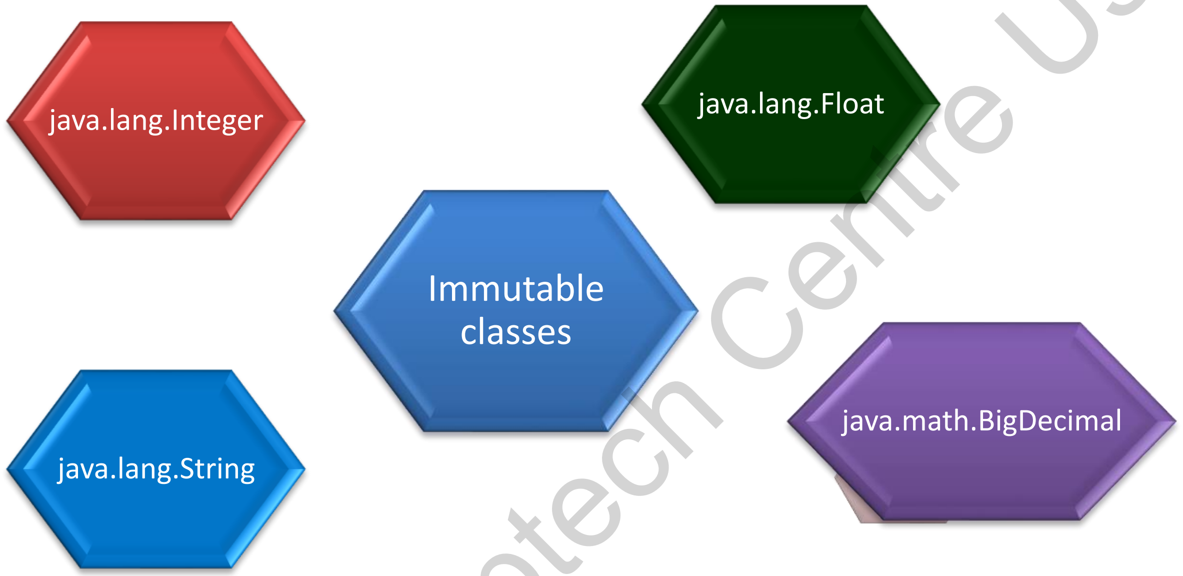
Output:

```
immutable
immutable
```

Immutability 2-3

Immutable class

Immutable class is a class in which the state of its instances does not change while it is constructed.



- It eradicates the prospect of data turning inaccessible when used as keys in Map and Set
- Immutable object must not change its state, when it is in the collection

Immutability 3-3

Implementing an immutable class

Following are some guidelines on implementing an immutable class correctly:

The class requires to be specified as a final class. final classes cannot be extended.

The class requires to be specified as a final class. These final classes cannot be extended.

Do not define any methods that can alter the state of the immutable object.

Concurrency

Concurrency is multiple processes can start, run, and finish in overlapping time periods

Functional programming paves a strong base for concurrent programming

One among those ways is the `parallelStream()` method on Collection

Should be used cautiously as excessive concurrency can slow down the application

Recursion 1-2

Recursion is a programming language feature supported by various languages including Java

```
import java.lang.Math;
import java.util.Locale;
import java.text.NumberFormat;
import java.text.DecimalFormat;
import java.util.stream.IntStream;
import java.util.stream.DoubleStream;
import java.util.function.IntUnaryOperator;

 /*Aptech Java FPJ
 *Recursion Sample*/

public class FibboRecursion{
     static IntUnaryOperatorfibonacci;
      public static void main(String[] args) {
            System.out.println("Fibonacci Number Sequence:");//output
            IntStream.range(0,15)// to display how many fibonacci numbers

            .map(fibonacci = f -> {

            return f == 0 || f == 1
                                       ? 1
            : fibonacci.applyAsInt(f - 2) + fibonacci.applyAsInt(f - 1);
            })
               .parallel()
              .forEachOrdered(g ->System.out.printf("%s ", g));
          }
}
```

Output:

```
Fibonacci Number Sequence:
1  1  2  3  5  8  13  21  34  55  89  144  233
377  610
```

Recursion 2-2

Iterative factorial

```
static int factIter(int j) {//iterative approach
int k=1;
for (int h=1; h<=j; h++) {
k *=h;
}
return k;

}//result
```

Here, the code demonstrates a standard loop-based form; the variables k and h are updated and iterated.

Recursive factorial

```
static long factRecur(long i) {//recursive approach
return i ==1 ? 1 :i * factRecur(i-1);
}//result
```

Here, the code demonstrates a recursive definition (the function calls itself) in a simple form.

In general, making a recursive function call is much more expensive than the single machine-level branch instruction required to iterate.

Summary

- Functional interfaces are one of the key elements to implement functional programming in Java.
- Functional interfaces are defined in java.util.function package.
- A functional interface has only one abstract method (which is called as a functional method) and zero or more default methods.
- Currying is a concept in which, a function f of two arguments (consider a and b) is realized created as an alternative for a function g of one argument that returns a function.
- State cannot always be determined safely and for that reason, it is avoided whenever possible, in functional programming.
- Immutability is the capability of an object to resist or prevent change. It is one of the core concepts in functional programming.
- In Java programming, recursion is the feature that permits a method to call itself. A method that calls itself is known as recursive.