Session: 17

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 key elements that helps to implement functional programming in Java.

- It supplies target type of elements such as lambda expressions and method references.

- Functional interface has one abstract method and zero or more default methods.

Function 1/8

The built-in functional interfaces are included in the `java.util.function` package. `Function<T, R>` is one of them.

`Function<T,R>` can be used:

- As assignment target for a lambda expression.
- To map scenarios such in streams wherein `map()` function of a stream takes an instance of Function to convert the stream of one type to a stream of different type.

Function 2/8

`T → R` is the functional descriptor for `Function<T, R>`

Key Points under Functional Interfaces are:

1) Method `apply ()` → Is the prime abstract functional method of Function interface → Accepts `Type T` as input and yields an output object of `Type R`.

2) `Function <T, R>` contains two default methods: `compose()` and `andThen()`.

Function 3/8

`compose()`

- Combines the function on which it is activated
- Combines with another function that is named `before()`
- Input type will change from `type V` to type `T` if the before function is applied after the combined function

`andThen()`

- Combines the function on which it is activated
- Combines with another function called `after ()`
- `andThen()` combines with `after()`, so when combined function is called, at that point the current function is activated which changes the initial value of type `T` to type `V`

`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

`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()`: Is a function which returns its input argument as is.

Syntax:

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

where,

`T` – Is the type of input and output to the function

Function 7/8

The methods can be used in a chain for creating a function as shown in Code Snippet.

```
Function<integer,String> sampleF = Function.<integer>identity()
.andThen(i → 2*i).andThen(i→ "sampleStr"+i);
```

Resultant function will acquire an integer, multiply it by 2 and finally, prepend "sampleStr" to it.

Complete Program for defining and utilizing `displayDateTime()` method:

```
import java.util.function.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;
import java.time.*;

public class FunctionDemo {
  public static void main(String[] args) {
    Function<LocalDate,LocalDateTime> plusTwoM =
    Function.<LocalDate> identity()
    .andThen(displayDateTime(d → d.plusMonths(2)));
    System.out.println( Stream.iterate( LocalDate.now(), d →
    d.plusDays(1))
    .limit(10)
    .map(plusTwoM)
    .map(Object::toString)
    .collect(joining(", "))
    );
  }
  public static Function<LocalDate,LocalDateTime> displayDateTime(
   final Function<LocalDate,LocalDate> test) {
   return test.andThen(d → d.atTime(2, 2));
  }
}
```

Currying 1/6

Currying process transforms a function having multiple arguments into a function with a single argument.

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

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

Following is the basic pattern of all unit conversions:

- Multiply by the conversion factor

- Adjust the baseline if relevant

```
static double converter(double a, double e, double y) {
return x * e + y;
}
```

Curry-Converter Usage:

- Following code is more flexible and it reuses the existing conversion logic.
- Instead of passing the arguments `a, e,` and `b` all at once to the `converter` method, the arguments `e` and `b` are called to return another function, which when given an argument returns `a * e + b`.
- The method enables to reuse the conversion logic and create different functions with different conversion factors.

```
static DUOcurryConverter(double e, double b){
return (double a) → a * e + b;
}
```

Currying 4/6

Apply converters as follows:

Conversion factor and baseline (`e` and `b`) is passed to the code which returns a function (of `a`) to do exactly what is expected.

For example, the converters can be used as follows:

```
DoubleUnaryOperatorconvert kgtolbs = curriedConverter(0, 2.2046);
DoubleUnaryOperatorconvert GBPtoEUR = curriedConverter(1.268, 0);
DoubleUnaryOperatorconvert CtoF = curriedConverter(9.0/5, 32);
```

Code Snippet shows an example `DoubleUnaryOperator` defines a method `applyAsDouble()` which is used here.

```
double gbp =
convertUSDtoGBP.applyAsDouble(1000);
```

Currying 5/6

Example demonstrates `compose()` in detail.

```
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
```

Currying 6/6

```
    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 1/4

- If the state of an object cannot change after it is constructed, it is considered immutable.
- For example, String is an immutable type in Java.
- Immutable objects are specifically valuable in concurrent applications.
- Example shows the concept of immutability.

```
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
```

Immutable Objects

Objectives to use immutable objects:

- ❖ If it is known that an object's state cannot be changed by another method, then it is easier to reason about how your program works.

- ❖ Immutable objects are thread-safe by default.

- ❖ Immutable objects can be used as keys in a `HashMap` (or similar), as they have the same hash code forever.

Immutable Class

- Immutable class is a class in which the state of its instances does not change while it is constructed.
- Following are some of immutable classes in Java: `java.lang.String`, `java.lang.Integer`, `java.lang.Float`, and `java.math.BigDecimal`.

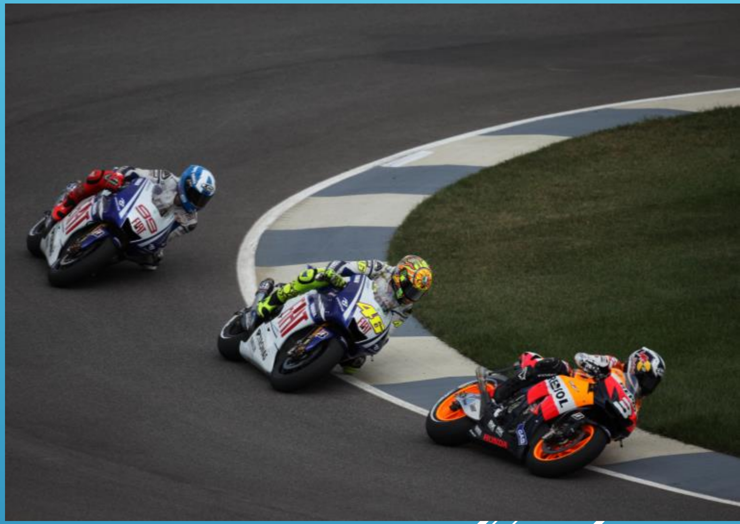
Implementing an Immutable Class

Guidelines to implement an immutable class:

- The class needs to be specified as a final class. `final` classes cannot be extended.
- All fields in the class have to be defined as `private` and `final`.
- Do not define any methods that can alter the state of the immutable object.

Concurrency 1/2

- Concurrency is performing multiple processes that can start, run, and finish in an overlapping time period.
- An alternate way in which Java 8 supports concurrency is through the new `CompletableFuture` class.
- One of its methods is the `supplyAsync()` static method that accepts an instance of the functional interface `Supplier`.



Concurrency 2/2

- Concurrency also has the method `thenAccept()` which accepts a `Consumer` that manages completion of the task.

- The `CompletableFuture` class calls on the specified supplier in a diverse thread and executes the consumer when it is complete.

Recursion 1/5

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

Following example clearly shows how to produce Fibonacci numbers with recursive lambda.

```
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 2/5

Recursion vs. Iteration

- Typical functional programming languages are not bundled with iterative constructs such as while and for loops.
- Following Code Snippet illustrates the usage of `while` loop with an iterator.

```
mangoes { } pass into the Iterator shown here:
Iterator<Mango> th = mangoes.iterator();
while (th.hasNext()) {
       Mango = th.next();
// ...
}
```

Here, the mutations (both changing the state of the Iterator with the `next` method and assigning to the variable `mango` inside the `while` body) are not visible to the caller of the method where the mutations occur.

Recursion 3/5

Recursion vs. Iteration

Using `for-each` loop, such as a search algorithm, is problematic since the loop body is updating a data structure that is shared with the caller as shown in Code Snippet.

```
public void searchForPlatinum(List<String> l, Stats bunch){
    for(String p: l){
    if("platinum".equals(p)){
        bunch.incrementFor("platinum");
     }
     }
}
```

The body of the loop has a side effect that cannot be neglected as functional style; it mutates the state of the `stats` object, which is shared with other parts of the program.

Recursion 4/5

Iterative Factorial

Following Code Snippet demonstrates a standard loop-based form; the variables `k` and `h` are updated and iterated.

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

Recursion 5/5

Recursive Factorial

Following Code Snippet demonstrates a recursive definition.

```
static long factRecur(long i) {//recursive approach

return i == 1 ? 1 :i * factRecur(i-1);

}//result
```

Summary

- Functional interfaces acts as a key element to implement functional programming in Java
- Functional interfaces are defined in `java.util.function` package, which is new to Java 8
- A functional interface has only one abstract method
- Currying is a process that transforms a function having multiple arguments into a function with a single entity that returns a function
- Immutability is the capability of an object to resist or prevent change
- In Java programming, a feature that permits a method to call itself is called recursion