DD1389

Föreläsning 9

- Testdriven programutveckling
- Funktionell programmering i Java

TDD

Här kommer att bli en demonstration av utvecklingen.

Funktionell programmering

- Lambda uttryck
- Metodreferens
- Konstruktur referens
- Högre ordning funktioner
- Ren funktionStrömmar
- First-class funktion
- Funktionsreferens
- Funktions komposition
- Flytande gränssnitt
- Functional interface
- Strikt och icke strikt evaluering
- Parallelism
- Beständiga datastrukturer
- Rekursion
- Optional och monader

Funktionell programmerings koncept i Java

- Högre ordningens funktioner
  - Att kunna skicka funktioner som parametrar
  - Att returnera en funktion
- Första klass funktioner
- Renhet (pure functions)

Lambda uttryck

- Anonym funktion
- Kan skickas till och från funktioner
- Oftast ett ”kort” uttryck (tanken bakom lambda-uttryck)
- använder operatorn **->**

Exempel:

- **`n -> n*2`**
- **`(n) -> n*2`**
- **`n -> {int d = 2; return n*d;}`**
- **`(a,b)-> a+b`**

Metodreferens och Konstruktor-referens

Om man vill skicka referens till en metod så kan man använda sig av följande syntax:

```
Math :: round
```

Klassnamnet :: metodnamnet

Hur används?

```
static Function<Float, Integer>  enFunktion(){
      return Math::round;
}
static Integer enAnnanFunktion(Function<Float, Integer> fnk,int n){
      return fnk.apply(n);
}
public static void main(String[] args){
   System.out.println(enAnnanFunktion(Math::round, 10.7));
}
```

Högre ordningens funktioner

- Som man kan ta emot referens till funktioner som parameter
- Som returnerar referens till funktioner

Exempel:

```
import java.util.function.*;
……
public Function <Integer, Integer> dubblera()
{return n -> n*n;}
```

Första klass funktioner (First-class)

- Funktioner som kan tilldelas till en variabel

Exempel:

```
import java.util.function.*;
……
Function <Integer, Integer> powerOf2 = n -> n*n;
System.out.println(powerOf2.apply(10));
```

Renhet (Pure function)

En funktion som **inte** har några sidoeffekter kallas en ”**ren funktion**”.

En funktion har sidoeffekt när koden **inuti** funktionen använder sig av variabler utanför funktionen.

Fördelarna:

- Funktionen returnerar samma värde oavsett hur många gånger den anropas (mad samma indata/parametrar förstås)
- Det finnas inga beroende mellan rena funktioner som annars hade påverkat den ordning de anropas.
- Stödjer **lat evaluering**.

Funktions komposition

- Med metoderna `compose` och `andThen` som finns i Function interfacet kan man anropa flera metoder på samma rad.

Exempel:

```
Function<Integer,Integer> negateThenDecrease=
          ((Function<Integer,Integer>) Math::negateExact)
          .compose(Math::decrementExact);
System.out.println(decreaseThenNegate.apply(10));
System.out.println(decreaseThenNegate.apply(-10));
```

Exempel på andThen

```
Function<Integer,Integer>negateThenDecrease=
          (((Function<Integer, Integer>) Math::negateExact)
           .andThen(Math::decrementExact));

      System.out.println(negateThenDecrease.apply(10));
      System.out.println(negateThenDecrease.apply(-10))
```

Strömmar

```
import java.util.function.*;
import java.util.stream.*;
import java.util.*;
class Streams{
    public static void main(String[] args){
        String[] allaOrd = {"Jag", "heter", "Olle"};
        List<String> list = Arrays.asList(allaOrd);
       Stream<String> stream1 = list.stream();
       stream1.forEach(ord ->System.out.print(ord+" "));
       System.out.println();
       Stream<String> stream2 = list.stream();
       stream2.map(ord->ord.replace('O','a’))
            .forEach(ord - >System.out.print(ord.replace('e','a')+" "));
    }
}
```