Lambda-uttryck

Tema: Att formulera och använda lambdauttryck och funktionella interface i Java.

Bakgrund

I den här övningen ska vi inledningsvis utgå ifrån en given klass `Person`, ett givet interface `PersonCondition`, och ett huvudprogram.

Klassen `Person` har följande specifikation:

```
/** Skapar ett personobjekt, med namn och ålder. */
Person(String name, int age);

/** Hämtar personens namn. */
String getName();

/** Hämtar personens ålder. */
int getAge();

/** Hämtar en sträng med personens namn och ålder. */
String toString();
```

Även följande interface, `PersonCondition`, är givet. Metoden `test` beskriver villkor, som ett `Person`-objekt kan uppfylla.

```
/** Beskriver ett villkor som Person-objekt kan uppfylla. */
public interface PersonCondition {

    /** Returnerar true om villkoret är uppfyllt, annars false. */
    boolean test(Person p);
}
```

Slutligen har vi ett huvudprogram, där en lista skapas och skrivs ut. Metoden `print` skriver ut de personer som uppfyller ett givet villkor. Som exempel på ett sådant villkor finns klassen `AgeCondition`, och i huvudprogrammet skapas ett sådant objekt.

```
public class PersonExample {

    /** Skriver ut alla personer i listan som uppfyller villkoret condition. */
    public static void print(List<Person> persons, PersonCondition condition) {
        for (Person p : persons) {
            if (condition.test(p)) {
                System.out.println(p.toString());
            }
        }
    }

    /** Villkor för att skilja ut personer som uppnått myndighetsålder. */
    private static class AgeCondition implements PersonCondition {
        public boolean test(Person p) {
            return p.getAge() >= 18;
        }
    }

    public static void main(String[] args) {
        List<Person> persons = new ArrayList<Person>();
        persons.add(new Person("Nilsson, Sten", 13));
        persons.add(new Person("Jonsson, Camilla", 23));
```

```
        persons.add(new Person("Hermansson, Lena", 38));
        persons.add(new Person("Fransson, Anneli", 15));
        persons.add(new Person("Lundström, David", 21));
        persons.add(new Person("Björk, Stefan", 20));
        persons.add(new Person("Andersson, Gun", 55));
        persons.add(new Person("Lundgren, Carina", 12));
        persons.add(new Person("Svensson, Ulf", 47));

        // (#)

        PersonCondition cond = new AgeCondition();   // (*)
        print(persons, cond);                        // (*)
    }
}
```

U 1. Bekanta dig med kodexemplet ovan. Hur många rader skrivs ut? (Varje `Person`-objekt antas skrivas ut som en rad.)

U 2. Som du ser har vi skrivit en hel del för att få in villkoret som parameter. Med lambda-uttryck kan detta göras enklare. Det är det som är deras stora poäng.

Skriv en ny rad, som ersätter raderna märkta `(*)` ovan. Vi ska fortfarande anropa `print`, men nu istället använda ett lambda-uttryck för villkoret.

Notera att vi nu helt kan ta bort den nästlade klassen `AgeCondition` – varför då?

U 3. Skriv ett anrop till `print` som skriver ut alla personer vars namn börjar på ”Lund”. Din lösning ska bestå av **en** rad Java-kod. Använd metoden `startsWith` i `String`:

`String`

```
/** Tests if this string starts with the specified prefix. */
boolean startsWith(String suffix);
```

U 4. Våra `PersonCondition` och `AgeCondition` beskriver funktioner som tar något (en `Person`) som parameter och returnerar `true` eller `false`. Sådana funktioner kallas inom den matematiska logiken för *predikat*.

Bland Javas standardklasser finns ett generiskt interface `Predicate` för sådana funktioner, och vårt interface `PersonCondition` är väsentligen ekvivalent med `Predicate<Person>`. Det ser ut så här:

`Predicate<T>`

```
/** Evaluates this predicate on the given argument. */
boolean test(T t);
```

I interfacet `List` finns följande metod:[1]

```
/** Removes all of the elements of this collection that
    satisfy the given predicate. */
boolean removeIf(Predicate<E> filter)
```

Skriv en rad som tar bort alla personer vars efternamn börjar på ”B” ur listan `persons`. Använd `removeIf` och ett lambdauttryck.

[1] Noga räknat finns metoden i superinterfacet `Collection`: `https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#removeIf-java.util.function.Predicate-`. Där har parametern till metoden `removeIf` en lite annan typ än ovan, nämligen `Predicate<? super E>`. Distinktionen spelar emellertid ingen roll när vi använder lambdauttryck, eftersom Java då ändå själv räknar ut lambdauttryckets parametertyp. Se även lösningsförslaget.

Sortering med `Comparator`

U 5. I Java finns ett standardinterface som heter `Comparator`, som anger i vilken ordning två värden ska vara, exempelvis vid sortering. Metoden `compare` jämför två värden, här kallade `o1` och `o2`. Den returnerar ett heltalsvärde som är < 0 om `o1` ska komma före `o2` i ordning, > 0 om `o1` istället ska komma efter `o2`, och exakt 0 om de båda värdena anses lika.

Värdet är alltså samma slags jämförelsevärde som i `String`-metoden `compareTo`, som du träffat på tidigare. Där är det alfabetisk ordning som jämförs; i en `Comparator` kan vi beskriva en godtycklig ordning själva.

```
public interface Comparator<T> {
    /** Returnerar ett värde <0, >0, eller ==0
        beroende på hur vi vill ordna o1 och o2. */
    int compare(T o1, T o2);
}
```

`Comparator` är ett exempel på ett *funktionellt interface*. Hur kan man se det? Och vad har funktionella interface med lambdauttryck att göra?

U 6. I standardinterfacet `List`, som vi bland annat använt för våra `Person`-objekt ovan, finns följande metod:

```
List<E>
```

```
/** Sorts this list according to the order
    induced by the specified Comparator. */
void sort(Comparator<E> c);
```

Skriv Java-kod för att sortera listan med avseende på namn (alfabetisk, stigande ordning). Lösningen ska bestå av **en** Java-rad, som passar på platsen märkt `(#)` i det inledande kodexemplets `main`-metod. Använd ett lambdauttryck.

U 7. Skriv en motsvarande Java-rad för att sortera listan med avseende på personernas ålder.

Tips: subtraktion kan vara användbart.

Att formulera och använda funktionella interface

För numerisk-matematiska beräkningar är det ibland användbart att beräkna ett approximativt värde på en funktions derivata i en given punkt. Vi kan beräkna en sådan approximation med hjälp av derivatans definition:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

(När vi räknar numeriskt så här kan vi inte låta h gå mot exakt noll, utan måste välja ett ”lagom” litet värde, exempelvis 10^{-6}. Vi återkommer till det.)

Vi vill nu ha en metod `deriv`, som beräknar en sådan approximativ derivata. Den som anropar metoden ska kunna använda ett lambdauttryck för att beskriva $f(x)$, så här:

```
double yp = deriv(x -> Math.sin(x),  0.3);
```

Här approximeras derivatan $f'(0.3)$, där $f(x) = \sin(x)$. Svaret `yp` $\approx \cos(0.3) \approx 0.9553$.

Anm. Om du läst Matlab som en del av kursen i Programmeringsteknik känner du kanske igen detta exempel från Matlab-bokens kapitel 1.3 och den första Matlab-laborationen. Javas lambdauttryck kan ses som en motsvarighet till Matlabs anonyma funktioner.

U 8. Skriv ett funktionellt interface `Func` som kan användas för den första parametern till `deriv` ovan. Detta funktionella interface ska alltså beskriva en funktion som tar ett double-värde som parameter, och returnerar ett annat double-värde som resultat.

U 9. Implementera metoden `deriv`:

```
/** Beräknar ett approximativt derivatavärde f'(x)
    för den givna funktionen f och det givna x-värdet. */
static double deriv(Func f, double x);
```

Använd derivatans definition ovan. För h kan du använda följande:

```
final double h = 1e-6;  // 10 upphöjt till -6
```

U 10. Läs om standardinterfacet `Function` i Java-dokumentationen.[2] Skriv en implementation av `deriv` där du använder standardinterfacet `Function` istället för `Func`.

U 11. (Avancerad uppgift.) Derivatan av en funktion är ju en annan funktion. Vi skulle därför vilja ha en *funktion* som resultat från `deriv`, så att vi kan skriva så här:

```
Func f = x -> Math.sin(x);    // f(x) = sin(x)
Func g = deriv(f);            // g(x) = f'(x)
double yp = g.compute(0.3);   // yp = g(0.3) = f'(0.3)
```

(Namnet `compute` ovan beror på ditt val i uppgift (a), och kan alltså skilja i din lösning.)

Implementera en ny version av metoden `deriv`:

```
/** Returnerar en funktion som approximerar derivatan för f. */
static Func deriv(Func f);
```

[2] `https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html`