Generiska klasser och funktioner

Algoritmer och datastrukturer
Obligatorisk Laboration nr 2

Syfte

Att ge träning i definition av generiska klasser och funktioner i Java. Problemen som belyses är viktiga att förstå i allmänhet, och för resten av kursen i synnerhet.

Övningen kräver ej avancerade typbegränsningsuttryck med wild-card och liknande men det rekommenderas ändå att studera dessa eftersom de dyker upp senare i kursen. Föreläsning 3 sammanfattar det viktigaste om generiska typuttryck. Ett exempel på en generisk mängdklass finns i OH från föreläsning 1.

Litteratur

Weiss: 4.6-8
Litt. Weiss 4.7, Skansholm 17.1.
OH-kopior från föreläsningarna: 1 (s. 22-) om en generisk mängdklass, 2 (s. 22-) om standardklasserna i Java, samt 3 om generiska klasser.

Färdig programkod

Given programkod som behövs i uppgifterna finns på kurshemsidan under fliken **Laborationer**.

Genomförande och redovisning

Uppgifterna är obligatoriska.

- Redovisning sker i grupper om två personer – inte färre, inte fler.
- Varje grupp skall självständigt utarbeta sin egen lösning.
  Samarbete mellan grupperna om övergripande principer är tillåtet, men inte plagiering.

Gå till kurshemsidan och vidare till **Laborationer->Hur man använder Fire-systemet**.

- Om du inte registrerat labbgrupp i Fire än: Gör laboration 0.0 innan du fortsätter!
- Redovisa filerna `SingleBuffer.java`, `TestSingleBuffer.java`, `CollectionOps.java` samt `Main.java`. med dina lösningar, samt `README.txt` med allmänna kommentarer om lösningen.
- Andra halvan av **Redovisningsproceduren** beskriver hur du postar materialet i Fire.
- Senaste redovisningsdag, se Fire.

Uppgift 1 En generisk enplatsbuffert `TestSingleBuffer.java`

Klassen `SingleBuffer` har metoderna `put` och `get` och kan lagra ett dataelement. Operationen `put` sätter in ett nytt dataelement i en tom buffert, och `get` tar bort och returnerar innehållet i en full buffert. Metoden `put` returnerar ett booleskt returvärde som indikerar om operationen lyckades eller ej; `put` misslyckas om bufferten är full, och `get` om den är tom och i så fall returerar `get` **`null`**, annars det borttagna elementet. Initialt är en buffert tom. Klassen skall vara generisk med avseende på det lagrade dataelementet. Definiera klassen! Exempel:

```
SingleBuffer<Integer> b = new SingleBuffer<Integer>();
Integer x,y;
boolean result;
result = b.put(123);      // result == true
result = b.put(456);      // result == false
x = b.get();              // x == 123
result = b.put(456);      // result == true
x = b.get();              // x == 456
y = b.get();              // y == null (bufferten var tom)
```

Uppgift 2 En egen generisk utskriftsmetod `CollectionOps.java,Main.java`

Litt. Weiss 4.7.3, Skansholm 17.1.2
Skriv en generisk (statisk) klassmetod `print` som skriver ut elementen i en objektsamling:

```
public static <T> void print(Collection<T> l)
```

Om samlingen innehåller elementen a, b och c så skall utskriften ha formen [a,b,c] och [] om samlingen är tom. Placera metoden i klassen `CollectionOps` och skriv testfall för den i `main`. Du får anta att samlingens elementtyp har en `toString`-metod, men *inte* att hela samlingen har en sådan. Studera först typen `java.util.Collection` i Java API. *Tips*: Samlingar är itererbara.

Uppgift 3 En generisk metod som vänder en lista `CollectionOps.java,Main.java`

Litt. Weiss 4.7.3, Skansholm 17.1.2
Skriv en generisk klassmetod som vänder elementen i en lista:

```
public static <T> List<T> reverse(List<T> l)
```

Exempel: Givet följande tre listor, med innehåll inom parentes

```
List<Integer> heltal              [1, 2, 3, 4, 5]
List<Double> flyttal              [1.25, 3.14, 9.7]
List<String> campusLindholmen     ["Saga", "Svea", "Jupiter"]
```

så skall metodanropen

```
CollectionOps.print(CollectionOps.reverse(heltal));
CollectionOps.print(CollectionOps.reverse(flyttal));
CollectionOps.print(CollectionOps.reverse(campusLindholmen));
```

resultera i att listornas innehåll förändras till [5,4,3,2,1], [9.7,3.14,1.25], resp. [”Jupiter”,”Svea”,”Saga”], ingen ny lista skall skapas, men en referens till den förändrade listan skall returneras, så att man t.ex. kan anropa metoden som ovan. Algoritmen får ej skapa temporära listor, enkla variabler räcker.

Tips: En viss metod som presenterades i den tredje föreläsningen kan visa sig användbar i lösningen.

Uppgift 4 En generisk metod som jämför två samlingar

`CollectionOps.java`, `Main.java`

Litt. Weiss 4.7.3, Skansholm 17.1.2

Skriv en generisk klassmetod som avgör om alla elementen i en samling är strikt mindre än (<) alla elementen i en annan samling. Alla jämförelser skall göras med ett komparatorobjekt som skickas med som parameter (jfr uppgift 2 i lab 1, samt förel. 3).

```
public static <T> boolean less(Collection<T> c1,
                               Collection<T> c2,
                               Comparator<T> comp)
```

Exempel: Givet följande komparatorobjekt och listor, med listornas innehåll inom parentes

```
IntegerComparator intcomp = new IntegerComparator();
StringComparator stringcomp = new StringComparator();
List<Integer> li1                [4,2,5,1,3]
List<Integer> li2                [8,6,7,9]
List<Integer> li3                [97,5,123,18]
List<String> johanneberg         ["HC2", "ED", "HC3"]
List<String> lindholmen          ["Saga", "Svea", "Jupiter"]
```

så skall anropen

```
CollectionOps.less(li1,li2,intcomp);
CollectionOps.less(li1,li3,intcomp);
CollectionOps.less(johanneberg,lindholmen,stringcomp);
```

Returnera **`true`**, **`false`**, resp. **`true`**. Definiera först `IntegerComparator` enligt samma mönster som `StringComparator` (se förel. 3).

Tips: I standardklassen `java.util.Collections` finns ett par metoder som kan göra din lösning till en "enradare".

Högre ordningens funktioner

Inom området funktionell programmering används s.k. högre ordningens funktioner. Sådana funktioner kan ta andra funktioner som parametrar och även ge funktioner som resultat. Man kan "simulera" något liknande i C med hjälp av funktionspekare, och i java med speciella objekt, s.k. *funktionsobjekt*, och från och med Java 8, med *λ-uttryck.* En högre ordningens funktion som brukar finnas i funktionella programspråk som ML eller Haskell, är *map* som applicerar en funktion på alla elementen i en lista. Översatt till pseudojava skulle det t.ex. kunna se ut så här:

```
Integer sign(Double x) {
    if ( x > 0.0d ) return 1;
    else if ( x == 0.0 ) return 0;
    else return -1;
}

List<Double> l1 = new ArrayList<Double>();
// lägg in talen 23.4, -19.0, 377.62, 0.0, 18.9, -32.12 i listan
...
List<Integer>l2 = map(sign,l1);
print(l2);                          → [1,-1,1,0,1,-1]
```

Ovanstående syntax har tidigare inte varit möjlig i Java. Vi visar först hur man använder funktionsobjekt, och i nästa avsnitt λ-uttryck. Först definierar vi ett generiskt gränssnitt för funktioner som tar ett argument:

```
public interface Function<T,R> {
    R apply(T x);
}
```

Vi använder två typvariabler för att göra `apply` så generell som möjligt. Den kan då returnera ett resultat av en annan typ (`R`) än argumentets typ (`T`). Ett gränssnitt som innehåller exakt en funktion kallas *funktionsgränssnitt.* Nu låter vi *klassen* `Sign` implementera `Function` genom att överskugga metoden `apply`:

```
public class Sign implements Function<Double,Integer> {
    public Integer apply(Double x) {
        return x.compareTo(0.0d);
    }
}
```

En instans av `Sign` är ett exempel på ett funktionsobjekt.

Den generiska klassmetoden `map` kan nu definieras

```
public static <T,R> List<R>
map(Function<T,R> f,List<T> l) {
    List<R> result = new ArrayList<R>(l.size());
    for ( T x : l )
        result.add(f.apply(x));
    return result;
}
```

Metoden fungerar, men det finns nackdelar: För att göra den generell använder vi `List` som parameter- och resultattyp, men returvärdet konstrueras med en `ArrayList`. Om metoden anropas med en annan listtyp som argument förväntar sig troligen användaren ett returvärde av samma typ. Dessutom vill vi göra metoden mer generell så att den kan hantera alla samlingar av typen `Collection`, inte bara listor. Vi kan utnyttja Javas möjligheter för att hantera dynamisk typinformation på följande sätt:

```
public static <T,R> Collection<R>
map(Function<T,R> f,Collection<T> c) {
    // Determine the dynamic type of the collection
    // so the same kind can be returned
    Class<? extends Collection> cls = c.getClass();
    try {
        // Create a result object of the same dynamic type as c
        Collection<R> result = (Collection<R>)cls.newInstance();
        for ( T x : c )
            result.add(f.apply(x));
        return result;
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Nu kan vi skriva om de två sista raderna i pseudokoden på förra sidan till typkorrekt fungerande javakod:

```
Collection<Integer> l2 = map(new Sign(),l1);
print(l2);
```

Uttrycket `new Sign()` skapar ett funktionsobjekt som används i `map`.

Uppgift 5 En generisk filtermetod

`CollectionOps.java`, `Main.java`

Med ledning av ovanstående exempel med `map` är nu uppgiften att på analogt sätt konstruera en generisk filtermetod. Filter tar ett *predikat* (en boolesk funktion med en parameter) samt en objektsamling som parametrar och returnerar en ny samling med alla element som uppfyller predikatet. Ett exempel i pseudokod:

```
boolean isEven(Integer x) {
    return x % 2 == 0;
}

List<Integer> l1 = new ArrayList<Integer>();
// lägg in talen 3, -42, 88, 19, -37, 0, 18 i listan
...
List<Integer>l2 = filter(isEven,l1);
print(l2); // [-42,88,0,18]
```

Först definierar vi det generiska funktionsgränssnittet

```
public interface Predicate<T> {
    boolean test(T x);
}
```

Definiera nu *klassen* `IsEven` i Java så att den implementerar `Predicate`. Definiera sedan den generiska klassmetoden `filter`. Filter blir ganska lik slutversionen av `map`. Tänk först efter vilka typvariabler den bör ha. Placera `filter` i `CollectionOps`. Skriv testkod i `Main`! Hitta gärna på ytterligare liknande exempel! Klassen `CollectionOps` skall alltså innehålla följande metoder när allt är klart:

```
public class CollectionOps {
    public static <T> void print(Collection<T> l) { ... }

    public static <T> List<T> reverse(List<T> l) { ... }
    public static <T> boolean less(Collection<T> c1,
                                   Collection<T> c2,
                                   Comparator<T> comp) { ... }
    // given
    public static <T,R> Collection<R>
    map(Function<T,R> f,Collection<T> l) { ... }
    ...
    // define filter!
}
```

Lambdauttryck i Java

I Java 8 har samlingsklasserna i `java.util` omarbetats för att möjliggöra parallell bearbetning i framtida datorarkitekturer. Ett led i denna förändring är att Java nu tillåter användning av λ-uttryck (lambdauttryck). I korthet är ett lambdauttryck en namnlös (anonym) funktion. Idén kommer från λ-kalkyl som är en teori för beräkningsbarhet inom området rekursionsteori. Kalkylen uppfanns redan på 1930-talet (!) av Alonzo Church.

Ex. Ett enkelt λ-uttryck för addition av två tal: λ x.λ y.x+y
Uttrycket kan appliceras på argument och reduceras till sitt värde:

(λ x.λ y.x+y) 1 2 … (λ y.1+y) 2 … 1+2 … 3

I Javas syntax skrivs ovanstående uttryck `(x,y) -> x+y`, punkten har alltså blivit en pil. Javas λ-uttryck kan inte existera isolerat utan måste placeras i ett sammanhang där de kan typkontrolleras mot ett funktionsgränssnitt. Om vi definerar

```
public interface BiFunction<T,U,R> {
    R apply(T x,U y);
}
```

kan vi t.ex. skriva

```
BiFunction<Integer,Integer,Integer> plus = (x,y) -> x+y;
System.out.println(plus.apply(1,2));
```

Vi kan också skicka λ-uttryck som parametrar till funktioner:

```
Function<Integer,Integer> sign = x -> x.compareTo(0);
map(sign,someList);
```

eller enklare:

```
map(x -> x.compareTo(0),someList);
```

Javakompilatorn översätter automatiskt λ-uttryck till funktionsobjekt av en lämplig subtyp till det matchande funktionsgränssnittet. Man kan säga att ett λ-uttryck definieras som en *funktion*, men anropas som en *metod*.

- Java 8 är installerat på datorerna i datasalarna - och kanske även på din dator?
- Paketet `java.util.function` innehåller en mängd olika funktionsgränssnitt, bl.a. `Function` och `Predicate`.
- Studera referenserna nedan för fler exempel.
- Prova gärna att byta funktionsobjekten av typerna `IsEven` och `Sign` i testkoden i `main` mot λ-uttryck.

Referenser

Angelika Langer & Klaus Kreft, *Lambda expressions in Java*,
http://www.angelikalanger.com/Lambdas/LambdaReference.pre-release.pdf

Jan Skansholm, *Java direkt*, Studentlitteratur 2014, 8:e upplagan.

Uppgift 6 Programmera med λ-uttryck

`CollectionOps.java`, `Person.java`, `Main.java`

Klassen `Person` definieras:

```
public class Person {
    private String name;
    private String email;
    private String gender;
    private int age;

    public Person(String name,String email,String gender,int age) {
        this.name = name;
        this.email = email;
        this.gender = gender;
        this.age = age;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }
    public String getGender() { return gender; }
    public int getAge() { return age; }
}
```

Sist i `main` finns kod som skapar en lista av personer. Lägg till kod som skriver ut epost-adresserna till alla kvinnor äldre än 65 år. Kombinera `print`, `map`, `filter` och lämpliga λ-uttryck, det blir högst ett par rader kod!