DAT043 - föreläsning 9

Mängder, avbildningar, iteratorer, anonyma klasser, lambda-uttryck, sökträd, rekursion

2017-02-13

Java collections framework, forts.

Mängder, likhet

- `Set<E>` är ett interface för samlingar av typen mängder, d.v.s. där elementen inte är ordnade och inga av elementen är lika varandra.
- Huvudmetoderna är `add`, `remove` och `contains`.
- För att mängder ska fungera korrekt krävs det att man överskuggar metoden `equals` från `Object` för den klass som utgör typen för elementen. Implementeringen i `Object` jämför bara om de två objekten är samma, d.v.s. samma koll som `==` gör på referensvärden.
- Detta är också viktigt för metoder i andra samlingar som är beroende av att jämför objekt.
- Om vi t.ex. vill ha en mängd av par enligt exemplet tidigare så får vi först överskugga `equals`:

```
@Override
public boolean equals(Object obj) {
  Pair<?, ?> p2 = (Pair<?, ?>)obj;
  return fst.equals(p2.fst) && snd.equals(p2.snd);
}
```

- Kom ihåg detta, för eftersom det finns en default-implementering i `Object` får du inget kompileringsfel (till skillnad från Haskell där du måste skicka en instans av `Eq`).
- `equals` är korrekt implementerad för alla relevanta klasser i Javas API, t.ex. `String`, `Integer`.

`TreeSet`, `Comparable`

- `TreeSet<E>` är en implementation av `Set<E>`. Denna bygger på att lagra elementen sorterat för att det ska gå snabbt att hitta dem.
- Därför måste man förutom att överskugga `equals` även implementera `Comparable` eller `Comparator`.
- `Comparable<T>` motsvarar klassen `Ord` i Haskell och deklarerar metoden
  `int compareTo(T o)`
  som ska returnera 0, ett negativt tal eller ett positivt tal beroende på om aktuellt objekt är lika med, mindre än eller större än o (motsvarande `LT`, `EQ`, `GT` i Haskell)
- Alla relevanta klasser i Javas API, t.ex. `String` och `Integer`, implementerar `Comparable`, så de kan användas i `TreeSet`.

HashSet, hashCode

- `HashSet<E>` är en annan implementation av `Set<E>` som bygger på en hashtabell, d.v.s. en array där objekt lagras på den plats som deras hashkod anger.
- Detta är också ett sätt att snabbt avgöra om ett element är medlem i mängden.
- Vill man använda denna för en viss typ av element bör klassen ha en välfungerande implementation av metoden `hashCode` från `Object`.
- Avsaknad av detta får man heller ingen varning om.
- `hashCode` har en implementering för t.ex. `Integer` och `String`.

Avbildningar (Maps)

- Avbildningar är samlingar av nyckel-värde-par. Man lägger in par av objekt där det ena är av nyckeltyp och det andra av värdetyp. Man kan sedan slå upp en nyckel och får ett värde.
- Interfacet är `Map<K, V>`
- De implementeras på samma sätt som mängder. Motsvarande implementeringar heter `TreeMap<K, V>` och `HashMap<K, V>`.
- Det är typen `K` som för `TreeMap` behöver implementera `Comparable` eller `Comparator` och för `HashMap` överskugga `hashCode`.

Iteratorer

- Det finns ett standardiserat sätt att genomlöpa alla element i en samling – iteratorer.
- I huvudinterfacet `Collection<E>` finns en metod `iterator()` som returnerar ett objekt av typen `Iterator<E>`.
- `Iterator<E>` är ett interface med huvudmetoderna `boolean hasNext()` och `E next()`. Dessa kan användas för att besöka alla element i godtycklig samling (lista, mängd, etc.)
- Ett iterator-objekt håller reda på var den befinner sig i representationen och `next` returnerar varje element exakt en gång.
- Med `hasNext` kan man avgöra om man besökt alla element eller om det finns fler.
- Man kan använda enhanced for loop för att utföra genomlöpning. Istället för en array anger man en samling (mer specifikt ett objekt vars klass implementerar gränssnittet `Iterable`).

```
Set<Integer> set =
  new TreeSet<>();
set.add(5);
set.add(8);
set.add(2);

for (Iterator<Integer> i =
       set.iterator();
     i.hasNext();) {
  System.out.println(
    i.next());
}
// detta gör samma sak
for (Integer n : set) {
  System.out.println(n);
}
```

Iterator

Som exempel på implementation av iterator, låt oss göra klassen som implementerar länkad lista på förra föreläsningen (se tillhörande kod) till Iterable.

- Ändra klassens deklaration till
```
public class LinkedList<E>
   implements List<E>,
              Iterable<E>
```
- Lägg till lokala klassen till höger här på sidan.

```
private class
    LLIterator implements
      Iterator<E> {

  Link next;

  public LLIterator() {
    next = first;
  }

  @Override
  public boolean hasNext() {
    return next != null;
  }

  @Override
  public E next() {
    E elt = next.elt;
    next = next.next;
    return elt;
  }
}
```

Anonyma klasser och lambda-uttryck

Anonyma klasser

- Lokala klasser som bara man bara skapar en instans av kan man definiera på ett smidigt sätt direkt i koden där instansen skapas.
- Detta kallas anonyma klasser. En anonym klass både definierar en klass och skapar en instans.
- I ett uttryck som ska vara en viss klass eller interface `T` kan man skriva `new T(`*constrarg*`) {`*klassdef*`}`
- Det innebär att man definierar en anonym lokal klass (en klass man inte ger något namn till) som ärver eller implementerar `T`.
- *klassdef* utgör klassens definition som vanligt. Men den kan inte definierar någon konstruerare.
- Ett objekt av denna anonyma klass skapas och konstrueraren i `T` som matchar *constrarg* anropas. Om det är interface skriver man `T()`

Anonyma klasser

- I definitionen av en anonym klass kan man referera till lokala variabler i metoden där klassen skapas, men bara om de är final eller effectively final.
- Effectively final är variabler som inte är deklarerade som final men aldrig ändras, d.v.s. skulle kunna deklareras final utan att kompileringsfel skulle uppstå.
- Som exempel, se koden för GUI-exempel från tidigare föreläsning. Istället för att definiera klassen ButtonCountListener kan vi använda anonym klass-konstruktion:

```
buttonCount.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    if (decreaseCheckBox.isSelected()) {
      count--;
    } else {
      count++;
    }
    labelCount.setText(Integer.toString(count));
  }
});
```

Lambda-uttryck

- I Java 8 kan man definiera en anonym klass och skapa en instans av den med lambda-uttryck.
- Det gäller klasser som enbart implementerar ett interface och att detta är av typen funktionsinterface, d.v.s. innehåller enbart en metod.
- För sådana kan man skapa en anonym klass genom att skriva
  paramlista `->` *fknskropp*
- *paramlista* har en av formerna
  - `()` – parameterlös funktion
  - `(typ1 var1, typ2 var2, …)` – vanlig parameterlista
  - `(var1, var2, …)` – parameternamn utan typer
  - `var` – funktion med en parameter vars typ inte anges
- *fknskropp* har en av formerna
  - `{ vanlig metoddef }`
  - `uttryck` – betyder `return uttryck;`
  - `f(…)` – anropar metoden `f` som enda sats

Lambda-uttryck

- Liksom anynoma klasser kan lambda-uttryck referera till lokala variabler.
- Som exempel, se koden för GUI-exempel från tidigare föreläsning. Istället för att definiera klassen ButtonCountListener kan vi använda lambda-uttryck:

```
buttonCount.addActionListener(
  e -> {
    if (decreaseCheckBox.isSelected()) {
      count--;
    } else {
      count++;
    }
    labelCount.setText(Integer.toString(count));
  }
);
```

Sökträd, rekursion

Rekursion

- Att skriva rekursiva metoder är inte lika vanligt i imperativa programspråk som i funktionella. Förekomsten av loopar gör att man ofta använder dessa istället.
- Det kan också vara en fördel resursmässigt att använda loopar om motsvarande rekursiva lösning skulle göra en stor mängd nästlade anrop till sig själv. Detta eftersom utrymmet som krävs i stacken blir stor.
- Men i vissa situationer är det betydligt enklare att lösa problemet med rekursion, nämligen då metoden behöver anropa sig själv på flera ställen.

Rekursion, Sökträd

- Ett exempel på när rekursion är sämre än att använda en loop är när man ska genomlöpa en lista som kan vara mycket stor.
- För att ge ett par exempel på när valet mellan rekursion och loop inte spelar någon större roll och när rekursion är överlägset ska vi titta på s.k. binära sökträd.
- Binära sökträd är en annat sätt att lagra element sorterat. Elementen lagras i noderna i ett binärt träd. Alla element i vänster delträd är mindre än elementet självt och alla i höger delträd är större.
- `TreeSet` och `TreeMap` bygger på sökträd.
- Se föreläsningens kod för implementation av ett par rekursiva och icke-rekursiva metoder kopplade till binära sökträd.