DAT043 - föreläsning 8

Paket, generics, Java collections framework

2017-02-07

Paket och tillgänglighet

- Ovanför klasser finns en hierarkisk namespace med paket.
- Filer som inte deklareras i något paket finns på toppnivån i hierarkin.
- Man deklarerar en fil att finnas i ett paket genom att i början skriva `package x.y.z;`
- En klass i ett paket nås genom att skriv `x.y.z.A`
- Man kan göra klasser tillgängliga utan kvalificering genom `import x.y.z.A;`
- Man kan importera importera alla klasser i ett paket (dock i dess underpaket) genom att skriva `import x.y.z.*;`
- Javas API är definierat i en hierarki, t.ex. `java.util.Scanner`
- Klassmedlemmars tillgänglighet skiljer sig ibland inom och utanför det egna paketet, se
  http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
- Klasser kan också ha olika tillgänglighet. Se samma sida.

Generics

Generics – bakgrund

- Precis som i Haskell vill man i Java kunna skriva kod som inte beror av (exakt) vilken typ den hanterar bara en gång, t.ex.
  `reverse :: [a] -> [a]`
- Från början fanns inte generics i Java. Sättet man gjorde det på då var att använda `Object`. Alla objekt är av typen `Object`, men ej primitiva värden.

```
public static void reverse(Object[] arr) {
  for (int i = 0; i < arr.length / 2; i++) {
    Object temp = arr[i];
    arr[i] = arr[arr.length - 1 – i];
    arr[arr.length - 1 - i] = temp;
  }
}
String[] a = {"A", "B", "C"};
reverse(a);
```

Generics – bakgrund

- Men detta fungerar inte alltid så bra. Som exempel, antag att vi vill definiera en klass som motsvarar ett par i Haskell (en tuple med två element).

```
public class Pair {
  Object fst, snd;
  public Pair(Object fst, Object snd) {
    this.fst = fst; this.snd = snd;
  }
}
Pair p = new Pair(new Integer(1),
                  new String("2"));
int x = (Integer)p.fst;
String y = (String)p.snd;
```

  Vi måste explicit typomvandla elementen när vi vill komma åt dem med dess specifika typer.

- I senare versioner av Java finns generics-mekanismen. Detta innebär att man liksom i Haskell kan använda typparametrar för att definiera metoder.

Generiska metoder

- Metoden `reverse` kan skrivas så generiskt så här:

```
public static <T> void reverse(T[] arr) {
  for (int i = 0; i < arr.length / 2; i++) {
    T temp = arr[i];
    arr[i] = arr[arr.length - 1 - i];
    arr[arr.length - 1 - i] = temp;
  }
}
```

- Man deklarerar typparametrar före metodens resultattyp genom `<T1, T2, …, Tn>`. Dessa parametrar kan sedan förekomma i resten av metoddeklarationen och i definitionen.
- Man kan också definiera generiska klasser och interface.

Generiska klasser

- Klassen `Pair` kan skrivas så här:

```
public class Pair<T, U> {
  T fst;
  U snd;
  public Pair(T fst, U snd) {
    this.fst = fst; this.snd = snd;
  }
}
Pair<Integer, String> p =
  new Pair<Integer, String>(new Integer(1),
                            new String("2"));
int x = p.fst;
String y = p.snd;
```

  Här behöver vi inte typomvandla och vi vet att vi har något av rätt typ.

- Man kan använda diamant-operatorn, `<>`, för att slippa skriva ut typen när man skapar en instans:

```
Pair<Integer, String> p = new Pair<>(new Integer(1),
                                     new String("2"));
```

Polymorfa arrayer

- I Haskell kan man begränsa de godtyckliga typerna om koden är beroende av att de uppfyller någon egenskap (implementerar en Haskell-klass):
```
minimum :: Ord a => [a] -> a
```
- I Java kan man åstadkomma vissa liknande saker utan generics. Antag att vi vill skriva en metod som ritar om alla Swing-komponenter i en array.
```
public static void repaintAll(JComponent[] arr) {
  for (JComponent comp : arr) {
    comp.repaint();
  }
}
```
- Denna kan vi anropa så här:
```
JButton[] buttons = new JButton[10];
// initiera elementen i arrayen
repaintAll(buttons);
```
- Detta fungerar för att `Jbutton[]` betraktas som en subklass till `JComponent[]`.

Polymorf användning av generiska klasser

- Detta är inte självklart, för det gör att vi kan skriva:
```
Object[] objs = buttons;
objs[0] = new String("");
```
- I Java hanteras detta genom att run-time kontrollera om arrayen innehåller element av den typ som man försöker tilldela den.
- Av den anledningen lagrar varje array information (när programmet körs) om vilken typ av element den innehåller.
- Programmet ovan stoppas med `ArrayStoreException` på andra raden.
- Säg nu att vi vill skriva en metod motsvarande repaintAll fast för ett par av komponenter.
```
public static void repaintBoth(Pair<JComponent, JComponent> p) {
  p.fst.repaint();
  p.snd.repaint();
}
```
- Om man försöker använda denna metod med ett par av Jbuttons får man kompileringsfel.
```
Pair<JButton, JLabel> p = new Pair<>(new JButton(), new JLabel());
// repaintBoth(p); – går ej
```

Type erasure

- Att vi inte kan göra motsvarande sak som för arrayer beror på att `Pair<JButton, JLabel>` inte ses som subklass till `Pair<JComponent, Jcomponent>`.
- Skälet är att det missbruk av typer som upptäcks vid exekvering när det gäller arrayer istället fångas upp vid kompilering när det gäller typuttryck (`GenType<ConcType1, ConcType2>`).
- Information av vilka konkreta typer som en instans av en generisk klass innehåller finns inte tillgänglig vid exekvering. Detta kallas type erasure.
- En begränsning i Java, som är en konsekvens av skillnaden i hantering av typer i arrayerr (run-time) och parametriserade typer (compile-time), är att man inte kan skapa arrayer där elementtypen är en typparameter eller ett uttryck med generiska typer.
  `new T[10]` – går ej
  `new Pair<Integer, String>` – går ej

Begränsade typparametrar

- För att kunna definiera en metod som `repaintBoth` så den går och tillämpa på det sätt vi vill använder man begränsade typparametrar, `T extends C`.
- Metoden `repaintBoth` kan skrivas så här:

```
public static <T extends JComponent,
               U extends JComponent>
    void repaintBoth(Pair<T, U> p) {
  p.fst.repaint();
  p.snd.repaint();
}
```

- Nu kan man anropa den med ett par av objekt där varje objekt tillhör en subklass till `JComponent`:

```
Pair<JButton, JLabel> p =
  new Pair<>(new JButton(), new JLabel());
repaintBoth(p);
```

Wildcards

- Wildcards kan ibland användas istället för typparametrar för att deklarera generiska metoder.
- `?` Betyder vilken typ som helst.
- `? extends C` betyder vilken typ som helst som är en subklass till klassen C eller till en klass som implementerar interfacet C eller en utökningen av detta.
- Metoden repaintBoth kan deklareras så här:

```
public static void repaintBoth(
  Pair<? extends Jcomponent,
       ? extends JComponent> p)
```

- Wildcards kan inte användas t.ex. då man måste kunna uttrycka att två variabler ska ha samma, okända typ.
- De kan inte heller ersätta typparametrar som inte är parameter till en generisk klass.

Exempel: länkad lista

- Vi definierade ett gränssnitt för listor av heltal på föreläsning 5.
- Denna skulle egentligen kunna vara generisk. Implementeringen utnyttjar inte att elementen är just heltal. Alla förekomster av `int` som refererar till elementtypen kan ersättas med en typparameter `E`.
- Vi ska utgå från ett generiskt list-gränssnitt och skriva en annan implementation av lista, s.k. länkad lista.

Java collections framework

Samlingar (Collections)

- I Javas API finns en välanvänd del som kallas Java collections framework.
- Det är en grupp interface och klasser som används till samlingar (collections) av objekt, alltså för att gruppera objekt lite på samma sätt som arrayer gör.
- Java collections framework använder flitigt generics för att samtidigt definiera klasser som samlar objekt av alla möjliga typer.
- Huvudinterfacet är `Collection<E>`. I denna deklareras generella metoder för samlingar såsom `add(E e)`, `size()` etc.

Listor

- Ett interface som utökar `Collection<E>` är `List<E>`. Detta interface definierar samlingar av typen lista, d.v.s. en samling av objekt som är ordnade. `List<E>` deklarar metoder såsom `add(int index, E e)` och `get(int index)` som lägger till och hämtar ut ett element på en viss plats i listan.
- För att kunna skapa listor måste det förstås finnas implementerande klasser (som för alla interface).
- `ArrayList<E>` implementerar `List<E>` med en dynamisk array, likt det icke-generiska exemplet som visades under föreläsning 5.
- `LinkedList<E>` är en annan implementering av `List<E>` som använder en länkad lista likt exemplet vi såg nyss.
- Det är oftast bättre att använda ett list-objekt än en array. Listor kan växa och krympa utan att man behöver implementera det själv. Man kan ta bort och skjuta in element mitt i listan. Dessutom fungerar, som vi sett, typkontrollen bättre för generiska klasser än arrayer.

```
List<String> xs =
  new ArrayList<>();

xs.add("a");
xs.add("b");
xs.add("c");
xs.add(2, "bc");
xs.remove(0);
for (int i = 0;
     i < xs.size(); i++)
  System.out.println(
    xs.get(i));
}

Resultat:
b
bc
c
```

Stackar och köer

- `Stack<E>` är en klass som implementerar en stack (FILO – först in, sist ut). Huvudmetoderna är `push` och `pop`.
- `Queue<E>` är ett interface för köer (FIFO – först in, först ut). Implementeras av t.ex. `LinkedList`.
- `Dequeue<E>` är ett interface för double ended queues, alltså köer där man kan stoppa in och ta ut element i båda ändarna. Detta interface implementeras t.ex. av `ArrayDequeue` och `LinkedList`.

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`.