DD1385
Programutvecklingsteknik
Några bilder till föreläsning 6
30/10 2018

Innehåll

- Ramverk
- Javas objektsamlingar
- Generiska typer i Java
- Interfacet **Comparable**
- Omslagsklasserna
- Mönstret Iterator

Ramverk är en uppsättning

- Återanvändbara
- Generella
- Samarbetande

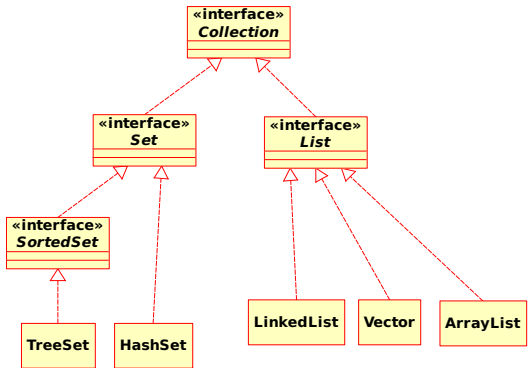
klasser för en kategori av tillämpningar

Ramverk i Java-API:n (exempel)

`awt` `swing` `Collections` `Maps`

Collection

Javas bibliotecksklasser för objektsamlingar,
här visas *en del av* interface- och klasshierarkin:



Några av metoderna i `Collection`.

Alla implementerande klasser har dessa

```
add(obj)
addAll(coll)
iterator()
remove(obj)
removeAll(coll)
retainAll(coll)
toArray()
isEmpty()
size()
```

List

- Numrerad samling objekt
- Dynamisk
  (kan växa och krympa, jfr []-array)
- Insättning och borttagning görs var som helst i listan

Ett objekt kan finnas på flera ställen i en lista

Det gäller []-arrayer också!

ArrayList, Vector

- Lagras m.h.a. []-arrayer
- Bäst för snabb åtkomst/ändring av listelement.

LinkedList

- Lagras länkat
- Bäst för snabb insättning/borttagning av listelement
- Välj den listtyp som passar bäst för aktuell tillämpning!

Generiska typer

Antag att vi vill ha en klass som kan hantera objekt av tre andra klasser vars typer varierar mellan tillämpningar:

```
class Handler {
  Object item1, item2, item3;

  Handler(Object i1, Object i2, Object i3){
    item1 = i1; item2 = i2; item3 = i3;
  }
  :
}
```

Objekt av `Handler` kan skapas

```
Handler handler =
  new Handler(new Counter(),
              ''BANANA'',
              new Color(100,230,120));
```

Ge `Handler` typ-parametrar så blir klassen säkrare (*)

```
class Handler<T1,T2,T3> {
  T1 item1; T2 item2; T3 item3;

  Handler(T1 i1, T2 i2, T3 i3){
    item1 = i1; item2  = i2; item3 = i3;
  }
  :
}
```

Skapa ett objekt:

```
Handler<Counter,String,Color> handler =
  new Handler(new Counter(),
              ''BANANA'',
              new Color(100,230,120));
```

(*) Användning av fel typ upptäcks redan vid kompilering och inte först när objektet används.

Generiska typer, forts.

En objektsamling kan skapas utan att elementtypen anges:

```
ArrayList alist = new ArrayList();
```

Element av typ `Object` antas men varningsmeddelande ges då listan används.

Collection-klasserna kan ges typparameter

```
ClassName<ElementType>
```

Generiska typer, forts.

Större säkerhet och enklare hantering med

```
CollType<ElementTyp> coll =
        new CollType<ElementTyp>();
```

Endast `ElementTyp` -objekt tillåts i samlingen

Till exempel:

```
ArrayList<String> alist =
        new ArrayList<String>();
```

Endast `String`-objekt kan sättas in i listan.

Några metoder i `List`.

add(obj)	sätt in sist
add(k,obj)	sätt in på plats k
get(k)	referens till objektet på plats k returtyp Object eller T T = typparametern till objektsamlingen
remove(k)	tag bort elementet på plats k

Set

- Ett objekt kan bara finnas med en gång.
- Använder `equals()` från `Object`

HashSet

- Snabb åtkomst
- Använder `hashCode()` från `Object`

SortedSet

- Hålls sorterad
- Elementen måste implementera interfacet `Comparable<T>` eller `Comparable`

interfacet `Comparable`

- Bestämmer en naturlig ordning – *natural ordering* – mellan element av en klass
- Gör generell sortering möjlig
- T.ex biblioteksmetoden `Collections.sort(alist)` sorterar en `List` med objekt, förutsatt att objektens klass implementerar `Comparable`

`Comparable` utan typparameter

```
public interface Comparable {
    public int compareTo (Object obj);
}
```

`Comparable` med typparameter

```
public interface Comparable<T> {
    public int compareTo (T obj);
}
```

`compareTo()` ska returnera <0, 0 eller >0 beroende på ordningen mellan elementen.

Om man väljer att göra en klass `Comparable` så måste man skriva metoden `compareTo()` enligt:

`x.compareTo(y)` $<$ 0 om `x` kommer före `y`

`x.compareTo(y)` $>$ 0 om `y` kommer före `x`

`x.compareTo(y)` $=$ 0 om `x` och `y` är lika
`x.equals(y)` $=$ **true** bör gälla då

`compareTo()` ska vara **konsistent** med `equals()`

Vi gör strax klassen `Spelkort` `Comparable`.

Klassen Collections

innehåller många statiska metoder som påverkar eller returnerar objektsamlingar.

Exempel:

`Collections.sort(aList);` sorterar aList

`Collections.shuffle(aList);` blandar aList

`Collections.reverse(aList);` vänder aList

Omslagsklasser

- Ibland behövs objekt av primitiva data
- T.ex. för en objektsamling av primitiva data
- I Javabiblioteket finns en klass för varje primitiv typ

```
Byte byte     Boolean boolean
Short short   Character char
Integer int   Float float
Long long     Double double
```

Omslagsklasser, forts

- Omslagsklasserna innehåller många metoder, t.ex. för omvandling mellan `String` och aktuell typ. Se dokumentationen!
- Autoboxing = automatisk konvertering mellan primitiv typ och dess omslagstyp.
- Unboxing = automatisk konvertering från omslagstyp till primitiv typ.

Exempel

`Integer.MAXVALUE` är största möjliga `int` – värde.

`Integer.parseInt(astring)` omvandlar textsträng till `int`, t.ex. vid textinmatning.

```
int finatalet = Integer.parseInt("1729");
Integer x = new Integer(27);
Integer y = 38;     // autoboxing
int z = x*y;        // unboxing
```

Gör inte autoboxing och unboxing i onödan!

Mönstret Iterator

- Ger elementen ur en (ev. komplicerad) struktur i sekvens
- Avslöjar ej detaljer om lagringen
- Implementerar ett `Iterator` – gränssnitt, t.ex.

```
interface Iterator {
  Object first();
  Object next();
  boolean isDone();
  Object current();
}
```

Iteratorn känner till den underliggande strukturen
och "vet" hur långt den kommit i iterationen

Iteratorn i Java-API:n

```
interface Iterator<T> {
  T next();
  boolean hasNext();
  void remove();
}
```

Alla objektsamlingar har Iterator !!

Hur används en iterator?

Antag att vi har en samling objekt av klassen `Book`:

```
Iterator<Book> bookIter = bookColl.iterator();
while (bookIter.hasNext())
  System.out.println(bookIter.next());
```

Nedan används iteratorn fast det inte syns explicit

```
for (Book b: bookColl)
  System.out.println(b);
```

Iterator erbjuder säker borttagning ur samlingen:

```
Iterator<Book> bookIter = bookColl.iterator();
while (bookIter.hasNext())
  Book b = bookIter.next();
  if (...)      //condition for removal
    bookIter.remove(); //current object removed
```

Exempel med Set

Generera
goal st olika tal från intervallet `1..range`.

tries räknar antalet försök.

range $\geq$ goal nödvändigt.

Använd subklassen HashSet.

- Slumpa tal, lägg i en mängd
- När mängden har goal st tal är det klart (vi vet ju att det är olika tal)

```
HashSet<Integer> hset =
  new HashSet<Integer>();

int tries = 0;

while (hset.size() < goal){
  int next =
    (int)(Math.random()*range+1);
  hset.add(next);    // autoboxing
  tries++;
}
```

Skriv ut talen ur hset

Med iterator explicit:

```
Iterator<Integer> iter = hset.iterator();

while (iter.hasNext())
    System.out.print(" " + iter.next());
System.out.println();
```

Med for-sats (iteratorn används):

```
for (int i : hset)
    System.out.print(" " + i);
System.out.println();
```

Provkör `Generate.java`

Varning för osäker operation:

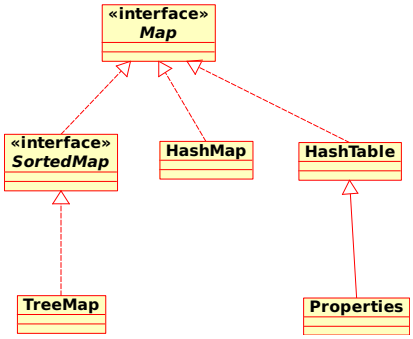
Att ta bort element ur en lista medan man loopar över listan.
Via Java-API:ns `iterator()` går det bra:

Tag bort alla 5-multipler ur hset via iter:

```
while (iter.hasNext()){
  int x = iter.next(); // autoboxing
  if (x%5 == 0)
    iter.remove();     // allowed and secure
                       // operation
}
```

Map

Bibliotheksklasser för avbildningstabeller, en del av klasshierarkin



Map

Objekt lagras tillsammans med en nyckel som också är ett objekt.

Map liknar dictionaries i Python

Några metoder i Map

`put(key,val)`	lägg in val med nyckel key
`remove(key)`	tag bort
`get(key)`	ger referens till objektet
`keySet()`	alla nycklar
`size()`	antal element