

Samlingar

TDA548/Joachim von Hacht

Samlingar



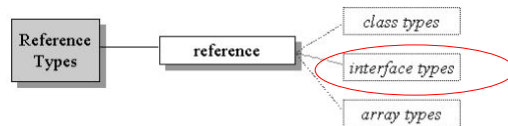
2

I verkligheten och i program är det mycket vanligt att man behöver hantera samlingar av objekt

Typiska operationer:

- Söka, lägga till, ändra, ta bort objektet i/ur samlingen
- Sortera samlingen, hitta olika delmängder, m.m.

Gränssnittstyper



```
public interface MyList ... {  
    boolean isEmpty();  
    boolean add(int i);  
    int get(int index);  
}  
  
// Variable with interface type  
MyList list = ... ;  
list.add(4); // Ok, method in interface type
```

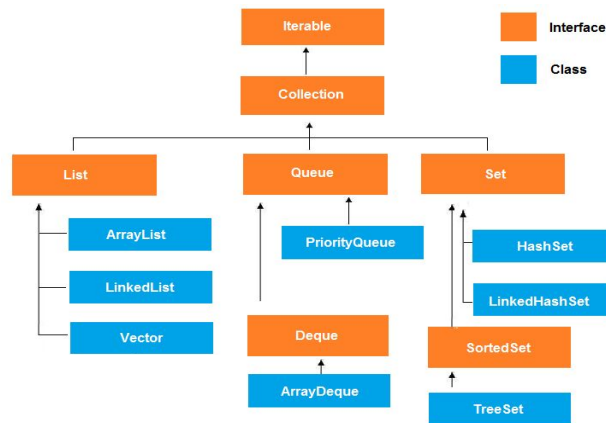
Ett **gränssnitt** ([interface](#)) är en samling av publika abstrakta metoder (public abstract behöver inte anges)

- Kan inte instansiera ett gränssnitt (med new) eftersom metoderna är abstrakta.
- Ett gränssnitt introducerar en referenstyp
 - De operationer som är tillåten för typen är de metoder vi angivit.
 - Innebär att vi kan deklarera en gränssnittstyp för en referensvariabel
 - Vi kan alltså skapa en ny typ utan att ange någon implementation av operationerna (finns inga metoder med körbar kod!)
 - Abstraktion! Vi kan ange att vi vill ha "något" som kan utföra vissa operationer (exakt vad detta är behöver vi inte ange).

I bilden deklareras ett gränssnitt MyList med ett antal metoder.

Vi säger också att ett gränssnitt definierar ett kontrakt.

Samlingar i Java



4

[The Java Collection Framework](#) (JCF) är ett antal färdiga samlingar vi kan använda

- För att använda samlingarna måste vi lägga till `import java.util.*;`
Samlingarna säger inte något om vilken typ av element de kan hantera
 - De är konstruerade för att klara vilken referenstyp som helst, de är **generiska**
 - Vid deklarationen anger man typen samlingen skall hantera, t.ex. `List<Integer>`, en lista med heltal.
- Detta är inget man lär sig utantill. Skulle det behövas får ni lämpliga metoder givna.
 - Alla moderna programmeringsspråk har liknande, huvudsaken ni vet det!
- Samlingarna räknas också som datastrukturer. Det finns en viss struktur på samma sätt som för arrayer och matriser.

JCF specificerar samlingarna med hjälp av ett antal gränssnittstyper. Kontrakten för dessa innebär att (se bild):

- **Iterable**, man måste kunna traversera samlingen
- **Collection**, man måste kunna lägga till/ta bort element, m.m.
- **List**, **Queue** och **Set** anger mer exakt hur elementen skall hanteras (läggas till /tas bort)

- Övriga lämnar vi därhän ...

I bilden

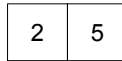
- De blå rutorna visar typer för objekt (klasser) som uppfyller olika kontrakt
- Pilarna visar union d.v.s
 - Klassen ArrayList skall uppfylla kontrakten Iterable, Collection och List
 - Alla metoder som finns i gränssnitten måste finnas implementerade i ArrayList-klassen
- Vi ser att det finns flera olika klasser som uppfyller samma union av kontrakt
 - T.ex. ArrayList och LinkedList
 - Dvs instanser av klasserna är utbytbara, de kan samma saker! Samma kontrakt!
 - Vi kan välja utifrån aktuell situation, smart! ...

List

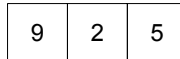
Tom lista



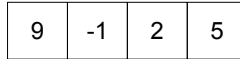
Lägg till 2
Lägg till 5



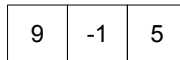
Lägg till 9 först



Lägg till -1 på index 1



Ta bort 2



5

[List](#) är ett kontrakt för en samling som representerar en linjär (ändlig) följd av element

- Påminner mycket om array men är dynamisk och har (många) metoder
 - Tom från början ...
 - .. därefter växer och krymper den dynamiskt
 - OBS! Ett elements index kan alltså ändras då nya element läggs till tas bort!

Metoder i List<T>

```
// Interface type for variable, initiate with class type object
List<Integer> l = new ArrayList<>();

out.println(l.isEmpty()); // True
l.add(100); // Put last in list
l.add(200);
l.add(300);
out.println(l.size() == 3);
out.println(l.get(2)); // Can't use [ ], use get() (0-indexed)
out.println(l.indexOf(300) == 2);

l.set(0, 500); // Will overwrite

Integer i2 = l.remove(1); // Remove and return removed

List<Integer> l2 = l.subList(1, 3);

for (Integer i : l) { // Traversing
    out.print(i); // NOTE Can't remove using this
}

list.add(null); // Very, very BAAAAAD!
```

6

Vi deklarerar referensvariabeln som en interface-typ, objektet däremot skapas utifrån någon klasstyp.

- Finns många fler metoder ...

Array kontra List

När använda vad?

- Array: Fix storlek, elementen skall behålla sina positioner (index)
- List: Alltid annars

7

List ger oss mycket mer färdig

- Dessutom är koden för List mycket grundligt testad, högre kvalitet på vårt program!
- En hel del färdiga Java-metoder returnerar Array:er
 - Isf kan vi fortsätta med array eller omvandla till List, se nedan.

Utskrift av Samlingar

```
List<Integer> list = new ArrayList<>();  
  
// Has own toString method, nice output  
out.println(list);
```

Alla samlingar har egna versioner av toString-metoden, ger läsbara utskrifter.

equals och hashCode

```
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o == null || getClass() != o.getClass()) {
        return false;
    }
    Complex complex = (Complex) o;
    return Double.compare(complex.re, re) == 0 &&
        Double.compare(complex.img, img) == 0;
}

@Override
public int hashCode() { // Let IntelliJ generate
    int result;
    long temp;
    temp = Double.doubleToLongBits(re);
    result = (int) (temp ^ (temp >> 32));
    temp = Double.doubleToLongBits(img);
    result = 31 * result + (int) (temp ^ (temp >> 32));
    return result;
}
```

9

Equals och hashCode-metoderna används (implicit) för elementen i samlingar

- equals() används t.ex. av metoderna indexOf(), contains() m.fl i ArrayList
 - Om felaktig equals()-metod kommer inte objektet att hittas.
 - På samma sätt med hashCode()
- Alla objekt som skall sparas i samlingar måste ha korrekta equals och hashCode.
 - Se bildserie om Klasser

Kort for-loop och Samlingar

```
List<Integer> list = new ArrayList<>();  
  
...  
  
// Bad can't remove in short for loop  
for( Integer i : list){  
    if( i > 200){  
        list.remove(i); // ConcurrentModificationException  
    }  
}
```

10

Fungerar bra ... men

- Man kan inte ta bort ett element i en lista som man traverserar i en kort for loop
- Ger Concurrent ModificationException
- En vanlig for-loop (med index) inte heller bra, eftersom storleken ändras (men inte indexet)
- Forts ...

Ta bort i kort for-loop

```
// Collect all to remove
List<AbstractDrawableObject> hits = new ArrayList<>();

for (AbstractMovableObject m : projectiles) {
    m.move();
    if (m.intersects(ground)) {
        hits.add(m); // Add item to remove
    } else
        ...
}
// After loop, remove all
projectiles.removeAll(hits);
```

11

En enkel teknik för att ta bort i kort for-loop.

- Skapa en tillfällig samling för allt som skall bort
- Lägg till i samlingen i loopen.
- Ta bort allt efter loopen.

Konvertering Array och List

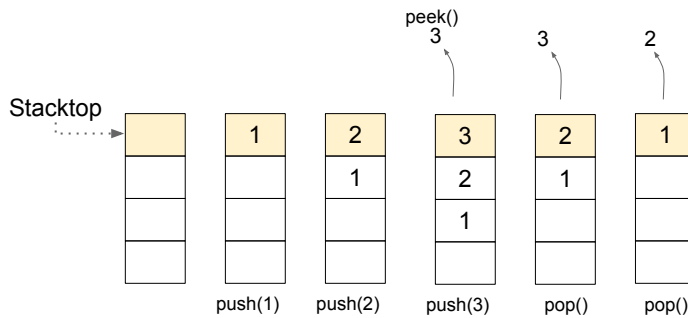
```
// Use helper class Arrays to create unmodifiable  
// list  
List<Integer> iList1 = Arrays.asList(1, 2, 3, 4);  
  
// Create modifiable list out of unmodifiable  
List<Integer> iList2 = new ArrayList(iList1);  
  
// Convert back to array. Must supply an array  
// object as argument  
Integer[] iArr = iList1.toArray(new Integer[]{});
```

12

Tekniskt lite rörigt, men möjligt att byta mellan List och Array

- Inget att kunna utantill

Stack



13

Stack är en datastruktur för en samling som bygger på principen **last-in first-out (LIFO)** (vi har sett anropsstacken tidigare)

- Element kommer ut i omvänd ordning jämfört med hur de stoppades in
 - En stack är användbar om man t.ex. vill "vända" på ordningen.
- Samlingen är linjär.
- Operationer sker på "stacktoppen". En speciell position
 - `push()` lägger till ett element på stacktoppen, "trycker" ner alla befintliga element ett steg
 - `pop()` tar bort elementet på stacktoppen (returnerar det), flyttar övriga ett steg upp
 - `peek()` kan avläsa värdet på toppen utan att ändra stacken (kopierar värdet).
- Dessa operationer är de enda vi behöver i kursen!

Metoder i Deque<T>

```
// Stack has a "strange" type in Java
// Interface is Deque and implementation is ArrayDeque
Deque<Integer> stack = new ArrayDeque<>();

out.println(stack.isEmpty());

stack.push(1);           // [ 1 ] add on top
out.println(stack.peek()); // Just read
out.println(stack.size() == 3); // false
stack.push(2);           // [ 2, 1 ] add on top

stack.pop();             // Remove from top [ 1 ]
out.println(stack.isEmpty()); // false

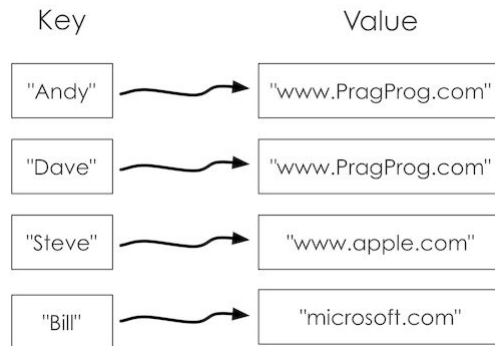
stack.clear();
out.println(stack.isEmpty()); // true
```

14

I Java använd gränssnittet Deque som ett kontrakt för en Stack

- Finns en gammal klass Stack, den skall vi inte använda.
- Inget att kunna utantill!

Avbildningstabell

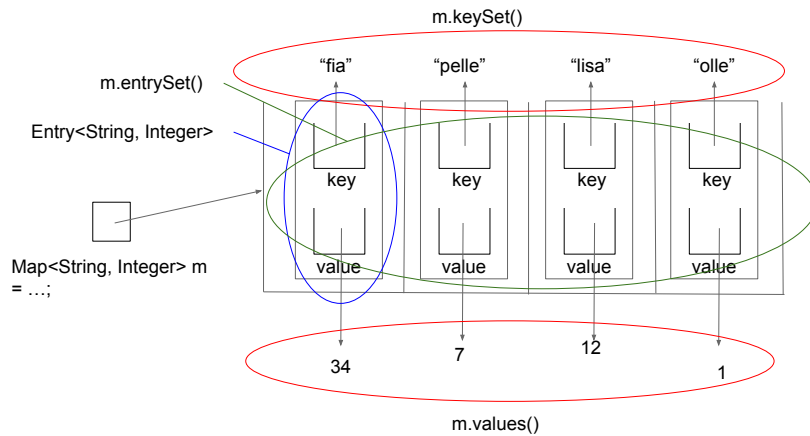


15

En avbildningstabell kopplar en nyckel och ett värde.

- Som en uppslagsbok
- Genom att använda nyckeln (key) kan man slå upp värdet (value)
 - OBS! Åt andra hållet är inte lika lätt.

Avbildningstabell i Java



16

En avbildningstabell i Java har typen `Map<Key, Value>` där båda `Key` och `Value` skall var någon referenstyp (`Map<String, Integer>` i bilden).

- Vanligaste operationerna är:
 - `put(key, value)` sparar värdet under nyckeln i tabellen
 - `get(key)`, avläser och returnerar värdet för en nyckel (ändrar inte i tabellen)

Förutom metoderna `put()` och `get()` kan man i Java behöva följande metoder från `Map`:

- `m.keySet()`, ger en samling med alla nycklar
- `m.values()`, ger en samling med alla värden
- `m.entrySet()`, ger en samling med `Entry` (par av nyckel/värde)

Metoder i Map<K,V>

```
// A Map with key type String and value Integer
// NOTE Variable naming: name + Count (used for names + frequency of name)
Map<String, Integer> nameCount = new HashMap<>();

// Add frequency of names
nameCount.put("fia", 23);
nameCount.put("sven", 31);
nameCount.put("lisa", 29);

int nFia = nameCount.get("fia"); // Look frequency for fia
out.println(nFia == 23);
out.println(nameCount.get("sven") == 31);

// Traversing
for( String s : nameCount.keySet()){ // ALL keys
    out.println(s);
}
for( Integer i : nameCount.values()){ // ALL values
    out.println(i);
}
for( Map.Entry<String, Integer> e : nameCount.entrySet()){ // Both key and value
    out.println(e.getKey() + ":" + e.getValue());
}

nameCount.remove("fia");
out.println(nameCount.size() == 2);
```

17

Gränssnitten/Klasserna ingår inte i JCF (om du saknar dessa i den tidigare bilden, ... spelar ingen roll för oss)