Objektorienterad Programmering DAT043

Föreläsning 9
12/2 -18
Moa Johansson
(delvis baserat på Fredrik Lindblads material)

Metoden clone()

- Skapa kopior av existerande objekt.
- Interface Cloneable
- Deep vs. shallow copy.

```
public class A implements Cloneable{
  private int x;
  private int y;

  @Override
  public Object clone(){
    return super.clone();
  }
}
```

```
A myA = new A();
A myClone = (A) myA.clone(); //cast
```

```
public class B implements Cloneable{
  private A a;
  private int z;

  @Override
  public Object clone(){
    // First: make a shallow copy
    B klon = (B) super.clone();
    // Deep copy instance variable a
    klon.a = (A) klon.a.clone();
    return klon;
  }
}
```

Clone() är en metod som finns definierad i klassen Object, och alltså ärvs. För att kunna kopiera objekt på ett bra sätt måste man dock överskugga clone() om man vill kunna skapa kopior på ett bra och vettigt sätt.

Först av allt kräver Java att klasser vars objekt ska kunna klonas implementerar det tomma interfacet Cloneable. Detta används som en “markör” för att visa att det går att skapa kopior av objekt. Annars kastas ett CloneNotSupportedException.

Om vi vill kopiera ett objekt som bara har instansvariabler av primitiva typer (t.ex. int) går det bra att helt enkelt kalla clone-metoden som ärvts från Object. Den kopierar alla instansvariabler rakt av, vilket är vad vi vill för primitiva typer (se exeplemt ovan klass A).

Om instansvariablerna däremot är av en objekttyp, så kopierar bara clone-metoden från Objekt *referenserna* till dessa (se klass B). Alltså kommer kopians instnasvariabel peka på *samma* objekt som orginalet. Ändrar man då orginalet ändras även kopian, eftersom deras referenser pekar till samma objekt. Detta kallas en *grund kopia* eller shallow copy. Oftast är detta inte vad vi vill, så därför måste man explicit klona även de instansvariabler som är av objekttyp i clone-metoden. Se exemplet för klass B. Denna producerar en s.k. djup kopia eller deep copy.

se t.e.x 10.12.4 i kursboken för mer info.

Repetion: Java Collections Framework
Mängder (Set) och Likhet

- interface Set<E>: mängder, ingen duplicering, ej ordnade.
- Kräver att element-typen överskuggar `equals(Object obj)`, vilken ärvts från Object.

```
public interface Set<E> extends Collection<E>{
 ...
 boolean add(E e);           // Lägg till e om ej redan i mängden.
 boolean contains(Object o); // Finns o i mängden?
 boolean remove(Object o);   // Tar bort o ur mängd, om den finns där.
 ...
}
```

```
@Override
public boolean equals(Object obj) {
   Pair<?, ?> p2 = (Pair<?, ?>)obj; // Type-cast. ? är wildcard för typ
  return fst.equals(p2.fst) && snd.equals(p2.snd);
}
```

•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 referens, 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:
•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.

Repetition: TreeSet och Comparable

- TreeSet<E> implementerar Set<E>. Lagrar element sorterade för effektivitet.
- Kräver implementation av metod för jämförelse för elementtyp, e.g. från interfacet Comparable<T>.

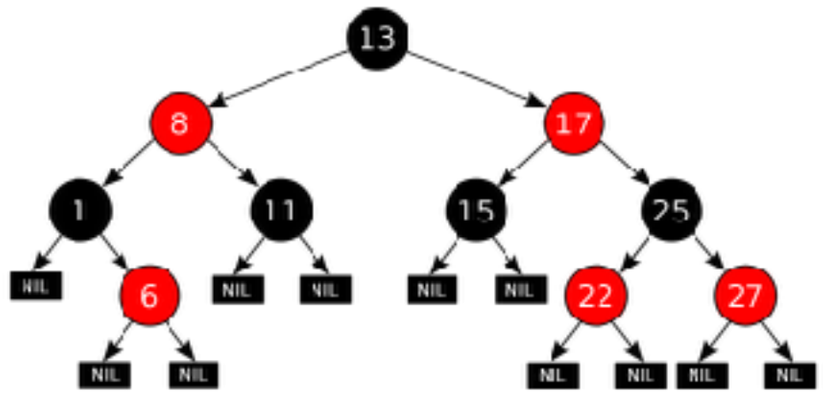
```
interface Comparable<T>{

 int compareTo<T o>
// Returnerar 0 om objekten är lika med varandra
// Returnerar ett positivt tal om större än argumentet o.
// Returnerar ett negativt tal om mindre än argumentet o.
```

•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

Repetition: TreeSet och Comparable

- TreeSet<E> implementerar Set<E>. Lagrar element sorterade för effektivitet.
- Kräver implementation av metod för jämförelse för elementtyp, e.g. från interfacet Comparable<T>.



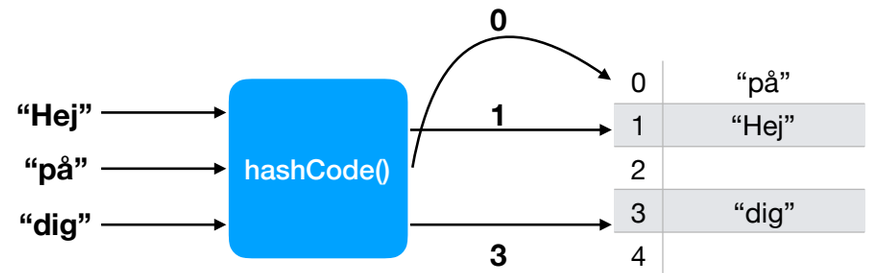
Här är ett exempel som visulaiserar hur ett TreeSet kan se ut (detta är ett s.k. binärt träd, det har alltid två grenar från varje nod). Notera att alla noder till vänster om roten (nod högst upp) är mindre än 13, och att alla noder till höger är större. Det samma gäller för alla interna noder.

Eftersom elementen är sorterade kommer det att gå snabbare att t.ex. leta efter ett element i mängden. Vi kan börja med att jämföra med roten (här 13) och sedan behöver vi bara söka i den halva av trädet som är större/mindre beroende på resultatet.
Det är tydligt att detta är mer effektiv än om vi skulle ha alla element i en lång lista, osorterade och i värsta fall behöva gå igenom hela listan från början till slut.

HashSet och hashCode

- `HashSet<E>` implementerar också `Set<E>`.
- Bygger på en s.k. *hashtabell (hash table)*.
- Objekt lagras i array på plats angivet av dess *hashkod*.
- Elementtypen E bör överskugga metoden `hashCode()` från `Object`.

```
public int 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.
* hashCode (en bra implementation av) ska returnera samma nummer för två objekt x,y ifall x.equals(y). Den ska helst (men det är sällan möjligt att garantera för alla objekt av en typ), inte returnera samma nummer för två objekt x,y som inte är lika.

Set<E>: Samanfattning

- Interfacet Set<E> kan implementeras av helt olika underliggande datastrukturer.
- TreeSet<E> vs. HashSet<E>
- Olika opertationer är olika snabba.
- Val beror på användningsområde.

Mer om detta i kursen om datastrukturer!

(HashSet är ofta snabbare än TreeSet. Dock är elementen i TreeSet sorterade i ordning, vilket kan vara en fördel.)

Interface Map<K,V>

- Nyckel - Värde par (key-value)
- Datastruktur där man “slår upp” med en nyckel och får ett värde.
- Samma underliggande implementation som för mängder:
  - `TreeMap<K,V>`: K implementerar `Comparable`/ `Comparator`
  - `HashMap<K,V>`: K överskuggar `hashCode()`

```
public interface Map<K, V> {
 ...
 V get (Object key);
 V put (K key, V value);
 void putAll(Map<? extends K, ? extends V> m);
 ...
}
```

- Map kallas “Avbildningar” i er bok. är samlingar av nyckel-värde-par (key-value). 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.

Anonyma Klasser

- Lokala klasser som man bara skapar instanser av på ett ställe av kan vara anonyma (utan namn).
  - t.ex. Lyssnare i GUI.
- **Skapas och definieras** där objektet ska användas.

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

•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.
- Här interfacet ActionListener.
•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()
•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.
- Här antar vi att decreaseCheckBox och labelCount är sådana variabler, d.v.s. de pekar alltid till ett och samma objekt.
•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.

Iteratorer

- Standardiserat sätt att genomlöpa alla element i samlingar.
- `Collection<E>` interfacet innehåller metoden `public Iterator<E> iterator().`
- Enhanced for-loop: Kan användas för samlingar som implementerar `Iterable.`

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

Iterator<Integer> i = set.iterator();
while (i.hasNext()){      // Har iteratorn fler element att löpa igenom?
 System.out.println(i.next()); // Returnera nästa element.
}
// detta gör samma sak. Iteratorn används "under ytan" på loopen.
for (Integer n : set) {
  System.out.println(n);
}
```

- 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> (kom ihåg att interfacen i Java Collections framework oftast ärver Collection).
- 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).

Demo: En iterator för länkad lista

- Låt klassen MyLinkedList från förra veckan också implementera interfacet `Iterable<E>`.
  - Detta kräver en metod
    `public Iterator<E> iterator()`
- Implementera iteratorn som en anonym inre klass, som implementerar interfacet `Iterator<E>`.
- Detta görs i metoden `iterator()`.

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

Lambda-uttryck

- Om vi bara vill åt en metod, måste vi då skapa en hel klass?
- **Funktionsinterface:** innehåller bara en abstrakt metod.
  - Kan dock även innehålla default-metoder.
- `ActionListener` är ett exempel på ett funktionsinterface.
  - Enda metod: `public void actionPerformed(ActionEvent e)`

```
buttonCount.addActionListener(
  e -> {    // Kompilatorn kan räkna ut att e har typ ActionEvent
    if (decreaseCheckBox.isSelected()) {
      count--;
    } else {
      count++;
    }
    labelCount.setText(Integer.toString(count));
  }
);
```

- 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.
- Liksom anonyma klasser kan lambda-uttryck referera till lokala variabler.

Lambda-uttryck

- Lambda uttryck har formatet:

```
params -> funktionskropp
```

  - `params` har någon av formerna:

```
()                          // parameterlös funktion
(typ1 arg1, typ2 arg2...) // vanlig parameterlista
(arg1, arg2, ...)          // parameterlista utan typer
arg                         // en parameter utan angiven typ
```

  - `funktionskropp` har någon av formerna:

```
{ ... }                    // vanlig metoddefinition
uttryck                    // kort för "return uttryck"
f(...)                     // anropar metoden f som enda sats
arg                        // en parameter utan angiven typ
```

Lambda-uttryck har formatet:
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

Funktionsinterface

- Se paketet java.util.function
- Samling av många funktionsinterface som kan användas i lambda-uttryck.
- Kom ihåg: Enbart en abstrakt metod per interface.
- **Exempel:** `Predicate<T>` representerar ett predikat, dvs. en funktion som tar något av typ T och returnerar en boolean.

```
@FunctionalInterface
public interface Predicate<T> {
   boolean test (T t);                    // Evaluerar predikatet på t
   default Predicate<T> negate (){...}    // Negation
   default Predicate<T> and (Predicate<? super T> other) //logisk AND
   default Predicate<T> or (Predicate<? super T> other)  //logisk OR
   ...
}
```

* Finns många funktionsinterface i Javas standard-API. Ett antal finns samlade i paketet java.util.function.
* Dessa är ganska abstrakta, men representerar koncept ni känner igen från matematik-kurser och från funktionell programmering.
* Ett exempel är interfacet Predicate.

Demo: LambdaDemo.java

Rekursion

- Inte lika vanligt i Java som i funktionella språk (t.ex. Haskell).
- Loopar kan vara fördelaktiga resursmässigt.
- Rekursion kan dock ge enklare lösningar på vissa problem.

DEMO: Binära Sökträd

•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.

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.

DEMO: BinarySearchTree