Programmering Fortsättning

Föreläsning 8

Joachim von Hacht

Innehåll

1 Instanceof kontra getClass()

Påminner om;

instanceof avgör typkompabilitet för tilldelning och typomvandling, d.v.s. om true så är typomvandlingen säker. OBS! *Ger alltid true för subklasser till aktuell klass.* Ger false för null. Returtyp boolean.

getClass() Ger runtimetypen för klassen, inget annat (returtyp Class).

2 Kanonisk form

(JLS 4.3.2) (Canonical form[1]) Klassen Object innehåller ett antal mycket grundläggande metoder (är som sagt superklass till alla klasser och arrayer).

KOD object

Vid implementation av klasser får man se upp med;

[1] Basic, canonic, canonical: reduced to the simplest and most significant form possible without loss of generality, e.g., "a basic story line"; "a canonical syllable pattern."//Wikipedia

- Override av metoderna: toString(), hashCode(), equals() och clone() (en metod för att skapa kopior)[2].

Motivering;

> "Why is it important to implement these methods correctly? In a small application written, used, and maintained by one individual, it may not be important. However, in large applications, in applications maintained by many people and in libraries intended for use by other people, failing to implement these methods correctly can result in classes that cannot be subclassed easily and that do not work as expected.
>
> It is, for example, possible to write the clone method so that no child classes can be cloned. This will be a problem for users who want to extend the class with the improperly written clone method. For in-house development this mistake can result in excess debug time and rework when the problem is finally discovered. If the class is provided as part of a class library you sell to other programmers, you may find yourself rereleasing your library, handling excess technical support calls, and possibly losing sales as customers discover that your classes can't be extended. "

[2]Finns också en metod finalize i Object. Kommer att köras precis innan objektet skräpsamlas. Inte garanterad att köras. Används inte av oss (strider mot best practices).

2.1 toString()

Kan vara bra att som standard override denna. Används implicit bl.a. +-operator och i Swing-komponenter.

2.2 Equals

Om vi vill ha likhet baserat på värde (inte referens) måste vi override:a denna.

2.2.1 hashCode

Om en klass overrides equals-metoden skall den alltid overrid:a hashCode()[3], hashCode används t.ex. då man sparar objektet i en hash-tabell (Map).

Motiveringen är att objekt som är lika (equals) skall hamna på samma plats i behållaren. Platsen ges av resultatet av hashCode (ett stort heltal). Följande kan t.ex. uppstå om Id implementerar equals() men saknar hashCode(), m är ett behållarobjekt;

```
Person p = new Person();
// Spara person under Id...
m.put( new Id( 6905165058 ), person );
// Hämta samma person...men
p = m.get(new Id( 6905165058 ));
// ...p blir null (ej funnen)!
```

De två instanserna av Id kommer att ha olika hash-kod, så när vi söker efter id (som är equals), så blir det ändå fel. Sökning sker på fel plats!

2.3 Equals och arv...

Kontraktet för equals;

- Skall vara en eqvivalensrelation d.v.s.
  - reflexiv, ∀o; o.equals(o) == true.

[3]Eclipse kan generera metoderna.

  - symmetrisk, x.equals(y) == true => y.equals(x) == true.
  - transitiv, x.equals(y) ∧ y.equals(z) => x.equals(z).

- ∀o; o.equals(null) == false.

Hur skall equals fungera vid arv? Vi måste bestämma om vi skall tillåta jämförelser mellan super och subklass d.v.s.

1. Skall super.equals(sub) alltid vara falskt (same type only comparison)...?
   - Spelar ingern roll om Person eller Student är super/sub-typ till varann, Person("Sven") och Studen("Sven") är aldrig lika.
   - Kan aldrig jämföra objekt i en hieraki, m.m.
   - Enkelt att implementera, använder; this.getClass() != other.getClass().
2. ... eller skall super.equals(sub) kunna ge true (mixed type comparison)?...
   - D.v.s. om Person :> Student så är Person("Sven") och Student("Sven") lika.
   - Innebär att vi kan jämföra olika objekt i en hierarki, kan vara praktiskt.
   - Använder !(obj instanceof MyClass)

2.3.1 ...ger problem[4].

- "Same type" equals (getClass) ger problem med symmetri och transitivitet. Problem med key-värden i tabeller. Spelar ingen roll om man tar hänsyn eller inte till signifikanta fält i subklasser (fält som skall ingå i jämförelsen).

KOD equals.sametype

- "Mixed types" equals (instanceof). Ger problem med symmetri och transitivitet om vi har signifikanta fält i subklasser, annars ok. Symmetri med signifikanta fält kan fixas men då bryter transitiviteten ihop.

KOD equals.mixedtype, equals.mixedtype.fixsymmetry

2.3.2 Design

Deklarera unikt id attribut och equals-metod i basklassen låt subklasser ärva allt (gör metod final).

- Eller använd delegering (se övningar).

KOD equals.mixedtype.nofields

2.4 Djup och grund kopia

Har vi berört förut (men inte namnen).

- Grund kopiering (shallow copy) innebär att vi bara kopierar fält från en klass till en annan rakt av. Fungerar för primitiva typer (kopier skapas) och ick-muterbara fält (dessa kommer att vara delade med det gör inget, de kan inte ändras).
- Djup kopiering (deep copy), Om man har muterbara referenser räcker det inte att göra kopior enligt ovan man måste kopiera de refererade objekten också (och så vidare...), annars kommer orginal och kopia att ha delade objekt.

[4] Se t.ex. webben, Angelica Langer, Secrets of equals() - Part 1, Secrets of equals() - Part 2

BILD

2.5 Clone

Ofta har man behov av att skapa en kopia av ett objekt. Metoden clone i Object var tänkt som en utgångspunkt för detta.

- Metoden clone i Object gör en grund kopia av objektet.
- Kontraktet för clone (kan vara situationsberoende)
  - o.clone() != o
  - o.clone().getClass()== o.getClass()
  - o.clone().equals(o) == true (inget absolut krav)
  - Ingen konstruktor får vara inblandad !!! Väldigt speciellt krav (se kodexempel).
- För att göra objekt "klonbara" måste klassen;
  - Implementera (det tomma) gränssnittet Cloneable
  - Overrida clone() och ändra accessen till public (clone är protected i Object).
- Räcker det med en grund kopia kan man direkt i clone anropa Objects (eller någon superklass) clone i den override:ena metoden (super.clone()).
- Vid djup för man se till att clone gör just detta (skapa kopior på referensvariabler).
- Om man anropar clone på ett objekt som inte implementerar interfacet genereras ett undantag; CloneNotSupportedException (eller InternalError om man gör som nedan).
- Ev. kan clone använda kovariant returtyp (d.v.s. samma typ som klassen istf Object).
- I praktiken fungerar clone som en slags kontruktor, måste uppfylla allt som gäller för kontruktor (kan förstöra invarianter).

2.5.1 Teknik

Tänkbara problem då man implementerar clone;

- Superklassen kanske inte implementerar clone korrekt. Problem...
- Om man har muterbara referenser måste man anropa clone för dessa. Tyvärr saknar många klasser clone (alt. den är felaktig). Problem...
- Finns inte clone för något delobjekt har man problem, ev. kan dessa lösas genom att dra ut all information m.h.a. getNNN() eller kanske med en kopieringskonstruktor, se nedan (i så fall måste man dock veta typen på delobjektet). Inte säker detta går...
- Ett fält kan dessutom hålla en referens till någon subklass...
- Alla fält som är behållare (array) måste klonas "element för element" om elementen är muterbara (behållarnas clone() ger grund kopia).

KOD clone.*

2.6 Förhindra Kloning

Man kan uttryckligen förbjuda cloning, override clone och låt metoden kasta CloneNotSupportedException då den anropas (det enda den gör).

2.7 Klona behållare och arrayer

Finns en hel del varianter. Grundproblemet är om man får en grund eller djup kopia, inte alltid lätt att utläsa ur dokumentationen. Några exempel;

```
// Shallow!
result = (MyClass[]) array.clone();

// API: Copies an array from the
// specified source array (?)
System.arrayCopy( sourceArray,
    srcPos, destArray, destPos );

// API: a copy of the original
// array...(?)
result.array = Arrays.
       copyOf( myArray );
```

3 Kopieringskontruktorer

Ett annat sätt att kopiera objekt är att skapa en s.k. kopieringskonstruktor!

- Fungar bra för final-klasser.
- Ger problem om variabeln är en subklass.

```
// A :> B
// Runtimeclass is B
A a1 = new B();
//Runtimeclass is A :-(
a2 = new A( a1 );
```

KOD copyctor

- En variant med interface med metod som i sin tur anropar kopieringskonstruktorn. Ger rätt rurntimetyp.

KOD copyctor.uno

4 Kopiering med Serializable

> "A common solution to the deep copy problem is to use Java Object Serialization (JOS). The idea is simple: Write the object to an array using JOS's ObjectOutputStream and then use ObjectInputStream to reconsistute a copy of the object. The result will be a completely distinct object, with completely distinct referenced objects. JOS takes care of all of the details: superclass fields, following object graphs, and handling repeated references to the same object within the graph."

5 Jämförelse

Förutom likhet mellan objekt behövs ofta jämförelse. Är objekt a större/mindre än b? Användbart t.ex. vid sortering.

5.1 Comparable

> "This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.[5]"// API

Genom att låta en klass implementera (det generiska) gränssnittet Comparable visar man att klassen har en *naturlig ordning*. Enda metoden är;

```
// Comparable<T>
public int compareTo( T o );
```

[5]The relation that defines the natural ordering on a given class C is:{(x, y) such that x.compareTo(y) <= 0}.

- Sortering sker lätt m.h.a. färdiga metoder i Collections och Arrays om klassen implementerar Comparable (många Java-standardklasser implementerar Comparable t.ex. String). Dessutom kan man söka min och max värden i Collections. Exempel;

```
// Customer-object in list
// implements Comparable
Collections.sort (customerList);
```

- Kontraktet för Comparable (sgn-funktionen ger -1, 0, 1)
  - ∀x,y, sgn(x.compareTo(y))== -sgn(y.compareTo(x)).
  - x.compareTo(y) > 0 ∧ y.compareTo(z) > 0 ⇒ x.compareTo(z) > 0.
  - x.compareTo(y) == 0 ⇒ sgn(x.compareTo(z)) == sgn(y.compareTo(z)).
  - Rekommenderas startkt att; (x.compareTo(y) == 0) == (x.equals(y)). Kallas att det är *consistent* med equals. Om ej skall detta anges mycket tydligt i specificationen.

    > "Curiously, BigDecimal violates this. Look at the Java API documentation for an explanation of the difference"

- Collections och Arrays har metoder för sökning (binärsökning), kräver att man först sorterar.

- Nackdelen med Comparable är att man måste i förväg bestämma objektens naturliga ordning (vilken kanske inte är helt uppenbar). Se comparator.

KOD comparable

5.2 Comparator

En mer flexibel lösning är gränssnittet Comparator som inte implementeras av klassen, utan av en speciell comparator-klass som sedan används vid sorteringen. Kontrakt för Comparable.

```
// Comparator<T>
int compare(T o1, T o2);
boolean equals( Object o );
```

Exempel;

```
// Using a comparator class
Comparator<Customer> comparator = ...
Collections.sort (customerList,
            comparator);
```

- Collections och Arrays sortering kan göras med hjälp av en komparator också.

- Vissa implementationer av behållare, t.ex. TreeSet, kan ta en Comparator som argument till konstruktorn, d.v.s. behållaren ges en total ordning då den skapas.

KOD comparator

5.3 Design

Överväg noga om klassen skall inplementera Comparable! Hur flexibelt ändra sortering (vad skall det sorteras efter).

6 Sammanfattning

- Vid seriös utveckling av större applikationer/bibliotek måste den kanoniska formen beaktas.

- Ev. kan man använda en kopieringskonstruktor med detta följer inte den normala standarden.
- Gränssnitten Comparable och Comparator är användbara bl.a. i sorteringssammanhang.