Programmering Fortsättning

Föreläsning 9

Joachim von Hacht

Innehåll

1 Mer om generiska klasser

En mycket bra referens till generiska typer är http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html

1.1 Generiska klasser och implementationsarv

Generiska klasser kan ingå i arvshierakier (behållare skall aldrig ärvas!). Inte blanda råa och generiska typer m.m.

KOD generics.inherit

1.2 Generiska klasser som inre klasser

Går bra.

KOD generics.inner

1.3 Generiska klasser och kanonisk form

Hur implementeras equal, clone, ... för generiska klasser, left as excercise to the reader, see Angelica Langer...(kommer ej på tenta)

1.4 Begränsade typparametrar

Genom att vid deklarationen ange begränsingar för typparametern får kompilatorn mer typinformation. Exempel; T är en subtyp till Number!

```
// Always extend (also for interfaces)
public class A <T extends Number> {
  T data;
  :
  // Compiler knows this is ok
  data.doubleValue();
```

Förhindrar felaktiga instansieringar;

```
//String doesn't extend number!
A<String> a = new A<String>();
```

Generall iakttagelse: Man försöker "stämma av" typsystemet så att man kan göra så mycket som möjligt men ändå bibehålla typsäkerheten.

KOD generic.boundtype

1.4.1 Multipla typparametrar och multipelt begränsade typparametrar

Kan man ha. Exempel;

```
class Pair<A extends Comparable<A>
    & Cloneable,
    B extends Comparable<B>
    & Cloneable>
    implements Comparable<Pair<A,B>>,
    Cloneable
{...}
```

- Samtliga begränsningar måste gälla (unionen av...) .

2 Generiska typer

Klasser och gränssnitt introducerar typer, så också generiska klasser och gränssnitt. Vad gäller?

- Om S :> T och C är någon klass så är **inte** C<S> :> C<T>, t.ex. en behållare med element S är inte supertyp till en behållare med element T!

```
List<String> ls = ...
List<Object> lo = ls; //FEL!
```

Motivering: Om så skulle vi kunna göra;

```
lo.add(new Object());
// Assign Object to String
String s = ls.get(0);
```

- Invariant subtypning gäller för generiska typer!!! (jämför arrayer och kovarians)
- En parametriserad typ är alltid subtyp till motsvarande råa typ t.ex.

```
List :> List<String>

List l;
List<String> ls = ...;
l = ls; // Ok.
ls = l; // Warning!
```

2.1 Jokrar (Wildcards)

- Används bara vid instansieringar d.v.s. inte vid typdeklarationer (wildcard instantiations -> wildcard parameterized type).
- Hur skriva en metod som kan ta vilken sorts lista som helst? Vilken är typen för "vad som helst"? Följande går inte;

```
List<String> ls = ...

public myMethod(
        List<Object> lo ){
   :
}
// Bad call ls not subtype
o.myMethod( ls );
```

- Lösning: Supertypen för alla behållare är ;

```
// Wildcard instantiations
List<?> lu;
```

? kallas joker (wildcard) och typen "List of unknown". Löser problemet!

```
// Print any Collection
void printCollection(
           Collection<?> c){
   for (Object e : c) {
       System.out.println(e);
   }
}
```

- Dock, det enda vi vet om elementen i List<?> är att de åtminstonde är av typen Object! Detta får följder för vad man kan göra med List<?>! *Allt som kräver mer typinformation än så är otillåtet*[1]*!*
  - Kan inte instansiera "unknown"-behållare (new ArrayList<?>() FEL).
  - Kan aldrig stoppa in något i en unknown-behållare. Vet inte om det vi stoppar in är subtyp (eftersom typen är okänd).
- Naturligtvis är; List<?> != List<Object>
- ? kallas obegränsad joker (unbounded wildcard)
- Begränsade jokrar ger typsäker kovarians, se nedan.

KOD generics.wildcard

[1] Finns även något som kallas "wildcard capture" ett specialfall då man kan använda t.ex. en typ av Set<?> där en typ av Set<T> krävs (supertypen istf. subtypen).

2.2 Begränsade jokrar (bounded wildcards)

Generiska typer är invarianta, begränsade jokrar ger ökad flexibilitet. Jokrar kan bara ha en begränsning, alltså inte multipla som ovan vid 1.4.1.

- Detta gäller bara parametertyper![2]

2.2.1 Uppåt begränsade jokrar

(wildcards with upper bounds)

- Antag att vi har A :> B :> C samtliga med metoden doIt() (overridden). Vi vill ha en metod som kan ta listor av typ A, B eller C *och anropa doIt()* . Följande går inte;

```
// Works for List<A> only invariance
void doItAny(List<A> la) {
   for( .... ){
    la.doIt();
   }
}
```

List<?> som parameter går inte heller eftersom vi då bara har typen Object att arbeta med.

- Lösning: Begränsade jokrar (bounded wildcards).

```
// Works for subclasses
void doItAny(
      List<? extends A> la){
   for( .... ){
    la.doIt(); //Ok!!
   }
}
```

```
List<B> lb = ...
doIt( lb ); // Ok
```

- I exemplet ovan är A övre *gräns för typparametern (upper bound).*

- <T extends Number> går inte, alltså T istf ?, ger ingen vettig infomation, se nedan...

KOD generics.wildcard.upperbound

2.2.2 Nedåt begränsade jokrar

Antag A :> B och att vi vill ge en mängd en total ordning t.ex. MySet<B>. Kan göras genom att skicka in en komparator till konstruktorn (konstruktorn använder denna för att på något sätt sortera elementen). Hur skall konstruktorn se ut?

```
public class MySet<E> {...
    public MySet(Comparator<E> c){...
```

Konstruktorn kräver nu att att vi skickar in en Comparator<B>, samma E på båda ställena!!, ...men det skulle kunna gå lika bra med en Comparator<A> (ev. en annan ordning men fungerar). Lösning;

```
public class TreeSet<E> {...
    public TreeSet(
       Comparator<? super E> c){...
```

- *E är lower bound.*

KOD generics.wildcard.lowerbound2

2.2.3 Typer med jokrar och begränsade parametrar

Vilka super/sub-typs relationer finns det mellan instansierade generiska typer? Kan

[2] "No wildcard type for return value! Wouldn't make the API any more flexible. Would force user to deal with wildcard types explicitly. User should not have to think about wildcards to use your API" //Josh Bloch

bli mycket komplicerat (Exempel se Angelica Langer). Vilken typ är super respektive subtyp av ...??? (kommer ej på tentan)

```
Collection<
    ? extends Serializable>
List<? extends Number>
```

eller

```
Pair<? extends Serializable,? super Long>
Pair<? extends Number,Object>
```

3 Generiska metoder

- Förutom typer kan också metoder (och konstruktorer) vara generiska (både instans och statiska[3]).
- Anges med en typvariabel i vinkelparenteser innan returtypen, typvariabeln används i parametrar och/eller returtyper och i metodkropp.
- Behöver vanligen inte instansieras, kompilatorn härleder typen (typerna), se nedan.
- Kan inte använda jokrar som typvariabler (parametrar kan använda jokrar (returtyper ska inte använda jokrar)).

KOD generics.methods

3.1 Typhärledningar

(Type inference) Extremt komplicerat se JLS 15.12.2.7. Mycket förenklat finns...

[3]Typvariabeln har ju inget med klassen att göra!

3.1.1 Typhärledning från parametertyper

- Hur blir det med overloading av generiska metoder? Metoderna kan få flera olika signaturer!
- Kompilatorn härleder typargumenten baserat på de aktuella argumentens typer (vilka typer skall stoppas in istället för typargumenten).
  - Normalt härleds de mest specifika typer som möjliggör ett typriktigt anrop.

KOD generic.methods.infer

- Kan leda till konstigheter då kompilatorn härleder en gemensam supertyp. Skydda mot detta genom begränsningar.

KOD generic.methods.counterintuitive

3.1.2 Typhärledningar vid tilldelningar

(assignment contexts) Om kompilatorn inte kan få fram typerna m.h.a. parametrarna kan den ändå i vissa fall härleda typen;

- Om resultattypen finns i returtypen och det sker en tilldelning till en parametriserad typ. Exempel;

```
// No params!
public <T> Trap<T> makeTrap(){
  return new Trap<T>();
}

// Can infer T to Mouse
Trap<Mouse> mouseTrap =
        makeTrap();
// Can infer T to Bear
Trap<Bear> bearTrap =
        makeTrap();
```

- I icke-generisk kod är följande ekvivalent;

```
g( f(x) );

// Same as above (if non-generic)
tmp = f(x);
g( tmp );
```

  Men i generisk är det inte säkert g(f(x)) kompilerar. Ingen tilldelning finns, kan ev. inte härleda typer!

3.2 Explicit instansiering

I vissa fall kan inte kompilatorn härleda instansieringen av typparametrarna alls. Man måste då explicit ange den aktuella typen. Sker enlig;

```
// Explicit
o.<String>myMethod(...);
```

KOD generic.methods.explicit

3.3 Overload och override av generiska metoder

Båda kan ge överraskande resultat. Se Typradering nedan.

4 Typradering

(Se också föreläsning 6) Generiska typer har av kompabilitetskäl implementerats m.h.a. s.k. typradering (type erasure).

4.1 Teknik

- Generiska typer: Vid typradering ersätts typvariabeln med sin längst till vänster angivna begränsning (den första) eller Object om sådan saknas.
- Parametriserade typer; Ersätts med den icke-paramereriserade typen List<String> blir List, o.s.v.
- Generiska metoder; Alla förekomster av typvariabeln ersätt som vid typer.
- Kompilatorn lägger in explicita typomvandlingar för att få korrekta typer.
- Typraderingen kan ändra signaturtypen, kompilatorn genererar då specialla metoder för att återställa detta (bridge methods[4]).

KOD generics.erasure

4.2 Konsekvenser

4.2.1 Typinformation och klassliteraler

Instanceof fungerar bara med reifiable typer (typer som har kvar runtime information). Exempel;

```
Object o = new LinkedList<Long>();
o instanceof List      //Ok
o instanceof List<?>   //Ok
o instanceof List<Long> //Error
o instanceof List<
     ? extends Number> //Error
o instanceof List<
     ? super Number> //Error
```

4.2.2 Overload

Följande är ok.

```
class SomeClass {
   // Object param atfer erasure
   public <T> void method(T arg){...}
   // Number after
```

[4] Vilka normalt inte kan anropas direkt, kollas av kompilator.

```
    public <T extends Number> void method( T arg){...}
    // No erasure
    public void method( Long arg){...}
}
```

Metod ett och två kommer att ha olika signatur eftersom generiska metoder inkluderar parametrarnas "bounds" (Object och Number). Följande blir fel (får samma signatur efter radering (Number));

```
class SomeClass {
  public <T extends Number> void method( T arg){...}
  public void method( Number arg){...}
}
```

KOD generics.overload

4.2.3 Override

Fungerar.

KOD generics.override

4.2.4 Static

Typparametrar får inte förekomma i samband med static. Alla parametriserade typer delar samma klassdeklaration, d.v.s. List<String> och List<Integer> skulle dela samma klassvariabel??!!

```
public final class X <T> {
  private static T field; // error
  public static T getField() {
    return field; } // error
  public static void setField( T t) {
    field = t; } // error
}
```

En generisk klass kan ha *vanliga statiska medlemmar*. Parameteriserade typen med jokrar är också tillåtet;

```
    private static List<T> field; // No!
    private static List<?> field; // Ok!
```

4.2.5 Arrayer

Kan inte ha typvariabler för arrayer (finns ingen runtime information om vad T är).

```
// T gone after compiling
<T> T[] makeArray(T t) {
    return new T[100]; // error
}
```

Elementtypen för ett array-objekt får inte vara en typvariabel eller en parametriserad typ. Runtime kontrollen av att man sparar rätt typ i arrayen kräver typinformation men denna typinformationen har ju raderats vid typraderingen.

> JLS 10.10 "If the element type of an array were not reifiable (§4.7), the virtual machine could not perform the store check described in the preceding paragraph. This is why creation of arrays of non-reifiable types is forbidden. One may declare variables of array types whose element type is not reifiable, but any attempt to assign them a value will give rise to an unchecked warning (§5.1.9)."

KOD generics.array

5 Generiska undantag

Javas catch-gren fungerar inte med generiska klasser, d.v.s. vi kan int ha generiska undantagsklasser.

> "It is a compile-time error if a generic class is a direct or indirect subclass of Throwable. This restriction is needed since the catch mechanism of the Java virtual machine works only with non-generic classes."

6 Reflection

Wikipedias definition;

> "In computer science, reflection is the process by which a computer program can observe and modify its own structure and behavior. The programming paradigm driven by reflection is called reflective programming."

Mer konkret: Vi kan ta fram data om objekt (inte objektets data utan data om objektet = data om datan = metadata).

Reflection är inget man tar till direkt. Det används vid avancerad (speciell) utveckling t.ex. ramverk[5] och pluginbaserade applikationer.

Fördelar Ger helt nya möjligheter.

Nackdelar Långsamt, ej säkert (ofta mycket casting), kan bryta mot inkapsling, måste matcha strängar exakt (vad händer då vi flyttar mellan paket o.dyl), m.m.

> "There is no easy way (other than File Search) even in modern IDE's to know which attribute is referenced and where. This makes Refactorings much more complex (tiresome!) and error prone."

[5] Ett ramverk är en halvfärdig applikation. Tanken är att användare (programmerare) skall kunna anpassa ramverket till olika behov.

6.1 Reflection i Java

- I Java har alla objekt möjlighet att komma åt sitt klassobjekt m.h.a. getClass() eller bara .class (får ett objekt av typen Class<T>). Objektet innehåller infomation om klassen t.ex. konstruktorer, metoder, fält (vi kan även komma åt private!!). Som tidigare sagt;
  - För att använda .class måste man veta klassnamnet
  - getClass() räcker med ett objekt.

KOD reflection.basic

6.1.1 Instansiering med reflection

Man kan utifrån en sträng (klassens kvalificerade namn) instansiera nya objekt. Man kan även skapa objekt som implementerar ett känt interface.

KOD reflection.creation

Att man kan skapa implementationer av interface gör att man t.ex. dynamiskt kan ladda ner kod och köra klassen behöver inte vara känd bara interfacet (jmf RMI). Dessutom säkerhetsproblem

KOD reflection.plugin

Finns även speciella fall t.ex. databaser då kombinationen reflection/Class<T> kan ge mer specifika klasser (än Object, men ändå typsäkert).

6.2 Reflection och generiska typer

Fungerar. Vi kommer åt mycket dock ej exakta runtimtyper.

KOD reflection.generics

7 Sammanfattning

- Generiska typer ger oss stor frihet med bibehållen typsäkerhet (många sätt att stämma av typsystemet, jokrar, begränsningar..).
- Invariant subtypning gäller för generiska typer.
- Generiska metoder finns också.
- Generiska typer är implementerat med "typradering", ger många följdverkningar.
- M.h.a. reflection kan vi få fram meta data om objekt. Kan även användas för att skapa objekt givet det kvalificerade namnet.