



**Peter Backlund** är Javautvecklare på Citerus, och har bland mycket annat utvecklat en portning av Java Pet Store till Grails - Grails Pet Store.

[peter.backlund@citerus.se](mailto:peter.backlund@citerus.se)

## Från Java till Groovy

**Kan du Java och är nyfiken på Groovy? Följ med när Peter Backlund steg för steg portar kod från Java till Groovy, och lär dig om hur Groovys dynamiska egenskaper kan vara till nytta för dig.**

Att använda dynamiska språk i Javas virtuella maskin och tillsammans med Java är väldigt populärt just nu, eftersom det gör det möjligt att kombinera de dynamiska språkens kärnfulla och flexibla syntax med den Javas prestanda, spridning och mognadsgrad.

Groovy är ett dynamiskt språk som har likheter med bland annat Ruby men som från början skapades för att användas just tillsammans med Java. Groovy har tre stora fördelar som vi ska titta närmare på: det är syntaktiskt väldigt likt Java, det är helt naturligt att använda befintliga Javabibliotek och det är dessutom inga problem att anropa Groovy-kod från Java.

Vi kommer att titta på en hyfsat enkel metod som är implementerad och testad i Java, där vi successivt skriver om både implementation och test i Groovy och slutligen tittar på hur tester och implementationer kan användas i vilken kombination som helst.

Vår metod ordnar en lista strängar efter fallande frekvens, och så här ser den ut i Java:

```
import org.apache.commons.collections.HashBag;

public class JavaOrderer implements Orderer {

    public List<String> orderByFreq(List<String> in) {
        final Bag bag = new HashBag(in);
        List<String> out = new
        ArrayList<String>(bag.uniqueSet());

        Collections.sort(out, new Comparator<String>() {
            public int compare(String a, String b) {
                int freq = Integer.valueOf(
                    bag.getCount(b)).compareTo(bag.getCount(a)
                );
            }
        });
    }
}
```



```

        return freq != 0 ? freq : a.compareTo(b);
    }
});

return out;
}
}

```

Här är ett enhetstest som visar hur metoden fungerar. Listan ordnas så att strängen med flest förekomster är först ("b" nedan, med tre förekomster) och därefter i fallande ordning. Om två strängar har lika antal förekomster ("c" och "d" nedan) ordnas de alfabetiskt.

```

public class OrdererTestUsingJava extends TestCase {

    public void testOrderByFreq() {
        Orderer orderer = new JavaOrderer();
        List<String> unordered = Arrays.asList("b", "d", "b", "b", "a", "c", "a");
        List<String> ordered = orderer.orderByFreq(unordered);
        assertEquals(Arrays.asList("b","a","c","d"), ordered);
    }

}

```

Nu ska vi göra en stegvis omskrivning från Java till Groovy. Vi börjar med att skapa en Groovy-implementation av Orderer-interfacet, vårt första exempel på integration mellan Java och Groovy. Samtidigt passar vi på att plocka bort några saker som är överflödiga i Groovy, nämligen public-modifieraren som är underförstådd (A), och semikolon som inte behövs alls. Dessutom byter vi namn på parametern in eftersom det är ett reserverat ord i Groovy (B).

```

(A) class GroovyOrderer implements Orderer {
    (A) List<String> orderByFreq(List<String> (B) inList) {
        final Bag bag = new HashBag(inList)
        List<String> out = new ArrayList<String>(bag.uniqueSet())

        Collections.sort(out, new Comparator<String>() {
            public int compare(String a, String b) {
                int freq = Integer.valueOf(bag.getCount(b)).compareTo(bag.getCount(a))
                return freq != 0 ? freq : a.compareTo(b)
            }
        })
        return out
    }
}

```

(Siffrorna inom parentes är inte del av koden, utan visar var förändringar har skett med föregående version, med hänvisning från texten.)

Därefter kan vi byta statiska typdeklarationer mot nyckelordet `def` (C), och använda en mekanism som kallas coercion, en slags blandning mellan castning och typkonvertering (D):

```
(C) def orderByFreq(inList) {
  (C) def bag = new HashBag(inList)
  (C) def out = bag.uniqueSet() (D) as List

  Collections.sort(out, new Comparator<String>() {
    public int compare(String a, String b) {
      (C) def freq = Integer.valueOf(bag.getCount(b)).compareTo(bag.getCount(a))
      return freq != 0 ? freq : a.compareTo(b)
    }
  })

  return out
}
```

En av de få språkliga konstruktioner i Java som inte stöds i Groovy är anonyma inre klasser (de beräknas tillkomma i Groovy 1.7), så i nuläget kompilerar inte vår Groovy-implementation. Istället kan vi använda en closure för att implementera `Comparator`-interfacet (E). En closure är, enkelt uttryckt, ett anonymt block kod, eller en implementation av ett interface med en enda metod men utan klass- eller methodsignatur. Med hjälp av coercion kan vi få en closure att fungera som `Comparator` (F):

```
def orderByFreq(inList) {
  def bag = new HashBag(inList)
  def out = bag.uniqueSet() as List

  Collections.sort(out, (E) { a, b ->
    def freq = Integer.valueOf(bag.getCount(b)).compareTo(bag.getCount(a))
    return freq != 0 ? freq : a.compareTo(b)
  } (F) as Comparator)

  return out
}
```

En väldigt användbar sak i Groovy är att man har använt metaprogrammering, att programmatiskt modifiera befintliga klasser i körtid, för att väva in mycket av det Jakarta Commons-biblioteken och vissa statiska standardmetoder gör i Javas standardklasser. Exempelvis har man vävt på metoden `sort()` på `java.util.Collection` (H), så istället för att skriva `Collections.sort(out)`; kan man skriva `out.sort()`.

Dessutom är API:t utökat med att acceptera en vanlig closure som Comparator, man behöver inte ens explicit coercion (I):

```
def orderByFreq(inList) {
    def bag = new HashBag(inList)
    def out = bag.uniqueSet() as List

    (H) out.sort { a, b ->
        def freq = bag.getCount(b).compareTo(bag.getCount(a))
        return freq != 0 ? freq : a.compareTo(b)
    } (I)

    return out
}
```

Slutligen kan vi stryka explicita return-satser (J), eftersom Groovy alltid returnerar evalueringen av den sista raden (sort-metoden returnerar den sorterade listan), samt använda operatorn <=> för compareTo (K):

```
def orderByFreq(inList) {
    def bag = new HashBag(inList)
    def out = bag.uniqueSet() as List

    out.sort { a, b ->
        def freq = bag.getCount(b) (K) <=> bag.getCount(a)
        freq != 0 ? freq : a (K) <=> b
    }
    (J)
}
```

Då är vi framme vid idiomatisk Groovy. Nu har vi sett att Groovys syntax ibland är nästan identisk med Javas och ibland rätt mycket elegantare och mer kortfattad. Dessutom har vi sett hur vi utan problem kan använda saker som HashBag från Jakarta Commons och hur Javas standardklasser är utökade och förbättrade när man använder dem från Groovy.

Slutligen ska vi se hur vi kan anropa Groovy-kod från Java. Vi modifierar vårt enhetstest en smula:

```
public class OrdererTestUsingJava extends TestCase {

    private void doTestOrderByFreq(Orderer orderer) {
        List<String> unordered = Arrays.asList("b", "d", "b", "b", "a", "c", "a");
        List<String> ordered = orderer.orderByFreq(unordered);
        assertEquals(Arrays.asList("b", "a", "c", "d"), ordered);
    }
}
```

```
public void testJavaImpl() {
    doTestOrderByFreq(new JavaOrderer());
}

public void testGroovyImpl() {
    doTestOrderByFreq(new GroovyOrderer());
}

}
```

När man kör testet ser man att båda implementationerna fungerar. Vi anropar Groovy-kod från Java, och det är helt sömlöst integrerat. Nu vänder vi på steken och skriver om vårt enhetstest i Groovy, och anropar både vår Java-implementation och vår Groovy-implementation:

```
class OrdererTestUsingGroovy extends GroovyTestCase {

    private def doTestOrderByFreq(orderer) {
        def unordered = ["b", "d", "b", "b", "a", "c", "a"] (L)
        def ordered = orderer.orderByFreq(unordered)
        assert (N) ["b","a","c","d"] == (M) ordered
    }

    void testJavaImpl() {
        doTestOrderByFreq(new JavaOrderer())
    }

    void testGroovyImpl() {
        doTestOrderByFreq(new GroovyOrderer())
    }

}
```

En del saker har vi sett tidigare, men i testklassen ser vi några ytterligare Groovy-specifika detaljer: byggd list-syntax (L), att ==-operatorn fungerar som equals() (för objektidentitet används tre likhetstecken, ===) (M) och att assert-nyckelordet är aktiverat och användbart i testsammanhang (N). Det går naturligtvis även bra att använda JUnits vanliga metoder, till exempel assertTrue och assertEquals om man föredrar det.

Slutligen några ord om prestanda. Jag skrev ett litet mikrobenchmark som ordnade en lista enligt metoden i artikeln med 100 element 10 000 gånger i följd (med uppvärmning), och Java visade sig vara runt sex gånger snabbare, med 150 millisekunder mot Groovys 900. Sådana här resultat ska man förstås alltid ta med en rejäl nypa salt, men man kan ändå notera två intressanta saker. Till att börja med är Groovy i det här fallet ändå så pass snabbt att man kan köra den aktuella rutinen fler än tio tusen gånger på en sekund, så det är redan idag tillräckligt snabbt för en mängd olika typer av uppgifter. Men än

# PNEHM!

Citerus nyhetsbrev för dig som vill lyckas med mjukvaruutveckling

C I T E R U S  
Utvecklar människor och mjukvara

viktigare är att man väldigt lätt kan använda Java för att eliminera flaskhalsar i en Groovy-baserad applikation.

Ett litet projekt med koden från artikeln inklusive benchmark finns att ladda ner från Google Code: <http://code.google.com/p/pnehm-java-to-groovy/>.

## Lär dig mer om Groovy

- Groovy: <http://groovy.codehaus.org/>
- Groovy JDK: <http://groovy.codehaus.org/groovy-jdk/>
- Webbramverket Grails: <http://grails.org/>
- Swingramverket Griffon: <http://griffon.codehaus.org/>
- Introduktion till Grails (av artikelförfattaren):  
<http://pnehm.citerus.se/kunskap/pnehm/pnehmartiklar/vivalaevolucion>

