10 Metaprogrammering i Java

(*Litteratur: se sista sidan*)

- Reflektion, introspektion, metaprogrammering
- Reflektion i Java
- Klassladdning
- Tillämpningar

Flexibilitet
Utbyggbarhet
"Plugginbarhet"

Vad handlar det om?

- *Så här får man väl inte göra?*

```
String MinTyp = "int";
MinTyp i;
```

... eller får man det?
… och vad skulle det vara bra för?

Svar: 1) Ja i princip. 2) Tveksamt i det här fallet, men …

Detta!

- *Så här får man göra (något förenklat):*

```
String MinTyp = "Klassnamn";
Class cl = laddaKlass(MinTyp);
Object obj = cl.görObekt();
obj.metod();
```

- komponentbaserad programmering
- en applikation kan använda nya klasser som tillkommer efter kompileringen av applikationen
- minimal koppling mellan applikationer och komponenter

Reflektiv programmering

- **Reflektion** i programmering är förmågan hos ett program att använda eller modifiera sin egen kod eller kod i omgivningen
  - *synonym*: metaprogrammering, introspektion
  - använda en sträng under exekveringen som om innehållet vore källkod
  - konvertera en **sträng** som matchar det **symboliska namnet** på en klass eller metod till en referens till klassen, eller anrop av metoden
- När ett programmeringsspråk stöder reflektiv programmering sparas typinformation i källkoden tillsammans med den kompilerade koden (.class-filen i Java)
- I Java representeras (reflekteras) sådan information under exekveringen i objekt av speciella metaklasser
  - Ex. klassen `Game` reflekteras av ett objekt av metaklassen `Class`

Användningsområden

Enterprise Computing (storskaligt)
- Java Beans, RMI (Remote Method Invokation)
- manipulering av objekt med mjukvaruverktyg
  - verktyget använder reflektion för att analysera eller ändra egenskaper hos dynamiskt laddade komponenter

Designmönster (småskaligt)
- generella factory-klasser
- generella lyssnare
- m.m.

Hantering av typinformation i Java

I Java finns flera metoder för att hantera typinformation under exekveringen
- `instanceof`
- Klassliteraler
  - Ex. `int.TYPE, Command.class` ger ett klassobjekt av typen `Class` som reflekterar typen
- Reflektion

Hantering av typinformation i Java

- Med `instanceof` kan man testa den dynamiska typen hos ett objekt
  - ```
    if ( cmd instanceof GoCommand )
        ((GoCommand)cmd).execute();
    else if ( puh! ... många fall blir det ...
    ```
- Kräver att klassen är känd vid kompileringen
  - eftersom klassen måste namnges
- Klasshierarkin avspeglas strukturellt i koden
  - Förändringar i hierarkin kräver motsvarande förändringar i koden

Fallanalyssyndromet
`switch`-satser …

Hantering av typinformation i Java

- Ex.

```
void printClassName(Object obj) {
    System.out.println("The class of " + obj + " is "
        obj.getClass().getName());
}
```

- Ex.

```
System.out.println("The name of class Foo is:" +
    Foo.class.getName());
```


Reflektion i Java

- den flexibla metoden – reflektion!
- reflekterade klasser kan
  - vara okända vid kompileringen
  - laddas dynamiskt under exekveringen
- typkontroll sker även här, men vid run-time

- hur kan man använda en okänd klass?

Hur kan man använda en okänd klass?

Utanför applikationen (komponenter):

1. Definiera ett gränssnitt, t.ex.

```
public interface Command {
    public void execute();
}
```

2. Definiera subklasser till `Command`

- antag att gränssnittet `Command` är känt i applikationen, men *inte* subklasserna
- med Javas reflektionsmekanism kan subklasserna användas i applikationen, trots att de ej är kända där vid kompileringen
- **Fördel:** Programmet kan *frikopplas* från klasshierarkin
  - nya subklasser kräver **ej** omkompilering av applikationen

Hur kan man använda en okänd klass?

Inuti applikationen:

3. Ladda någon av de okända subklasserna

```
String klassnamn;
   ...
laddade_subklassen = laddaKlass(klassnamn);
```

4. Skapa ett objekt av den laddade klassen

```
laddade_subklassen objektet;
```

5. Typomvandla till `Command`:

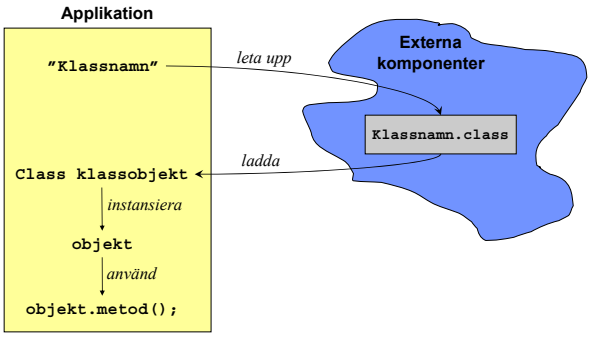
```
Command obj = (Command)objektet;
obj.execute();
```

- **Alltså:** *Den laddade klassens namn måste förstås anges vid laddningen*, men namnet behöver inte vara känt vid kompileringen, **det räcker om det är känt vid run-time!**

Klassladdning

- En klass laddas under exekveringen av JVM (Java Virtual Machine) från en `.class`-fil när ett objekt av klassen används första gången.
- För varje klass som laddas av JVM skapas ett motsvarande objekt av metaklassen `Class` som beskriver (reflekterar) den laddade klassen. Objektet innehåller bl.a. klassens statiska variabler.
```

Dynamisk klassladdning



Javas metaklasser

- `Class`
  - objekt av denna klass reflekterar klasser och interface
- `Constructor`
  - ... reflekterar konstruktorer
- `Method`
  - ... reflekterar metoder
- `Field`
  - … reflekterar attribut
- `Modifier`
  - … reflekterar accessmodifierare, `public`, `static`, m.m.
- `Array`
  - … reflekterar fält

`java.lang.Class`

Metoder (i urval)	Förklaring
`static Class forName(String className)`	laddar klassen eller gränssnittet med namnet `className` och returnerar dess klassobjekt
`Constructor getConstructor( Class ... parameterTypes)`	ger ett konstruktorobjekt med angivna parametertyper
`Method getMethod(String name, Class ... parameterTypes)`	ger en metodobjekt med angivna parametertyper
`Field getField(String name)`	ger ett variabelobjekt med angivet namn
`Object newInstance()`	skapar ett objekt av klassen som reflekteras av detta klassobjekt

Varargs = variabla argument

- I Java finns en möjlighet att skriva metoder som tar ett variabelt antal argument.

Ex

```
int f(int ... a) {
    int sum = 0;
    for ( int x : a )
        sum += x;
    return sum;
}
```

Ex. på anrop:

```
f(1); f(1,2); f(45,-17,99)
```

Ekvivalent kod

```
int f(int[] a) {
    int sum = 0;
    for ( int x : a )
        sum += x;
    return sum;
}
```


`java.lang.reflect.Constructor`

Metoder (i urval)	Förklaring
`Class getDeclaringClass()`	ger klassobjektet för klassen som deklarerar konstruktorn som reflekteras av detta konstruktorobjekt
`Class[] getParametertypes()`	ger ett fält av klassobjekt som reflekterar de formella parametrarna till konstruktorn som …
`Class[] getExceptionTypes()`	ger ett fält med alla undantag som kan kastas av konstruktorn
`String getName()`	ger konstruktorns namn

`java.lang.reflect.Method`

Metoder (i urval)	Förklaring
`Object invoke(Object o,Object ...args)`	anropar metoden som representeras av detta metodobjekt för objektet `o` och med parametrarna `args`
`Class[] getParametertypes()`	ger ett fält av klassobjekt som represen-terar de formella parametrarna
`Class getReturntype()`	ger metodens returtyp
`Class[] getExceptionTypes()`	ger ett fält med alla undantag som kan kastas av metoden
`String getName()`	ger metodens namn

Exempel

```
// Utan reflektion
Klassnamn objekt = new Klassnamn();
objekt.metod();
```

- Namn- och metodnamn är låsta
- Typkontroll vid kompileringen
- Hög läsbarhet

- Namn- och metodnamn kan variera under exekveringen
- Typkontroll under exekveringen (exceptions)
- Lägre läsbarhet

```
// Med reflektion
Class klassobjekt = Class.forName("Klassnamn");
Method metod = klassobjekt.getMethod("metod",parametertyper);
Object objekt = klassobjekt.newInstance();
metod.invoke(objekt,parametrar);
```


Exempel 1: commandFactory

```
public interface Command {
    public void execute();
}

public class Take implements Command {
    public void execute() {
        // do whatever
    }
}

+ Go, Stop, Turn, Quit, etc.
```

Ett något mer utvecklat exempel finns i `gamecharacters`

commandFactory

```
public void processCommand(String commandName) {
   Command command = getFactoryCommand(commandName);
   command.execute();
}

public static Command getFactoryCommand(String s) {
   Command command = null;
   try {
       // Load the class with name s and instantiate it
       command = (Command) Class.forName(s).newInstance();
   } catch (Exception e) {
       e.printStackTrace();
   }
   return command;
}


Ex.
String cmd;
...
processCommand(cmd);
```


Exempel2: Metoder med parametrar

```
public class C {
    public void f(Integer i) { ... }
    public void f(String s) { ... }
}


...
Class cls = Class.forName("C");
Method m = cls.getMethod("f",String.class);
Object obj = cls.newInstance();
m.invoke(obj,"Hej!");
```


Exempel 3: Generell lyssnare

```
public class Main extends JFrame implements ActionListener {
   public Main() {
      ...
      JButton plingButton = new JButton("pling");
      plingButton.setActionCommand("pling");
      plingButton.addActionListener(this);
      add(plingButton);

      JButton plongButton = new JButton("plong");
      plongButton.setActionCommand("plong");
      plongButton.addActionListener(this);
      add(plongButton);

      JButton klonkButton = new JButton("klonk");
      klonkButton.setActionCommand("klonk");
      klonkButton.addActionListener(this);
      add(klonkButton);
   }
```

... forts.

Generell lyssnare

```
   ...
  public void pling() { ... }
  public void plong() { ... }
  public void klonk() { ... }

  // General listener using reflection
  public void actionPerformed(ActionEvent ev) {
      try {
            this.getClass().
                  getMethod(ev.getActionCommand()).
                        invoke(this);
      }
      catch (Exception e) {
            e.printStackTrace();
      }
   }
}
```

Generell lyssnare

Litteratur

- McCluskey, Glen, Using Java Reflection, http://java.sun.com/developer/technicalArticles/ALT/Reflection *En relativt lättsmält introduktion till javas reflektions-API.*
- Portwood, Michael, Using Java Reflection Technology to Improve Design, *Den finns på kurshemsidan (… men var kom originalet ifrån?)* *En samling OH-bilder som ger en bra översikt.*
- Neward, Ted, Understanding Class.forname(), http://www.idevelopment.info/data/Programming/java/reflection/Understanding_ClassForName.pdf *En grundlig genomkörare för den som ämnar ge sig på avancerad komponentbaserad programmering. Tar bl.a. upp problemet med olika klassladdare och CLASSPATH i samband med Extensions m.m.*