Programmeringsteknik för S, 2004
Grupp IV

Johannes Hjorth
hjorth@nada.kth.se
Rum 4538 på plan 5 i D-huset
08 - 790 69 02

Anteckningar i pdf format:
http://www.nada.kth.se/∼hjorth/teaching/

Dagens övning innehåller ganska mycket, stryk under det du tycker är speciellt viktigt så går det snabbare att ögna igenom senare.

Titta gärna igenom dessa anteckningar igen på tåget när du åker hem.

Idag ska vi gå igenom

Klasser och instanser, hur hänger de samman och vad skiljer dem åt?

Nyckelorden `new`, `public`, `private`, `static` och `this`, vad betyder de?

Metodanrop och lokala kopior av variabler.

Vikten av att skriva snygg kod. Vi ger ett exempel.

Klasser och instanser

Java är ett objektorienterat språk. Det betyder att data och metoder för att manipulera data har samlats ihop till objekt.

En annan viktig fördel med objekt är att de låter oss abstrahera. Vi skiljer användningen av datastrukturer från den underliggande lagringen och implementeringen.

Med objekt kan vi alltså dölja den underliggande strukturen; hur objektet representerar data internt och vilka algoritmer dess metoder använder.

Efter att ett objekts kod är skriven är det bara intressant vilken illusion objektet presenterar för de andra programdelarna. Det vill säga, hur dess gränssnitt ser ut.

Detta betyder att vi kan ändra hur objektet fungerar inuti utan att behöva ändra på hela programmet.

Vad har klasser och instanser med detta att göra?

Vad är en klass?

En klass är vårt sätt att i programkoden beskriva hur objekten ska vara.

```
public class Complex {

    private double real, img;

    // Konstruktorer, skapar en ny instans
    Complex() { real = 0; img = 0; }
    Complex(double real) { this.real = real; img = 0; }

    Complex(double real, double img) {
        this.real = real;
        this.img = img;
    }

    // Metoder för att accessa realdel och imaginärdel
    public double getReal() { return real; }
    public double getImg() { return img; }

    Complex sum(Complex a, Complex b) {
        this.real = a.getReal() + b.getReal();
        this.img = a.getImg() + b.getImg();
        return this;
    }

    // Metod som omvandlar det komplexa talet till en sträng
    public String toString() {
        return real + "+" + img + "i";
    }
}
```

Kort om `public`, `private` och `this`

En variabel deklarerad i en metod existerar endast inom metodens måsvingar, den syns därför inte utifrån.

När en ny instans skapas väljer java den konstruktor som bäst matchar parametrarna vi skickade.

Metoder eller variabler deklarerade `private` är endast synliga inuti klassen.

Däremot är metoder och variabler som är markerade `public` synbara utanför klassen.

Nyckelordet `this` syftar på den instans vars instansmetod vi just nu befinner oss i.

Det kan även användas för att nå klassvariabler och instansvariabler som dolts av lokala variabler, `this.längd` syftar då på klassvariabeln `längd`.

Vad är en instans?

Om klassen är gjutformen så är instansen tennsoldaten, vars utseende är beroende av hur vår klass är definerad i koden.

Ur varje gjutform kan man skapa många liknande tennsoldater, på samma sätt kan man skapa många instanser av samma klass.

Nya instanser av en klass skapas med hjälp av `new`,

```
Complex x = new Complex(3,4);
```

Detta kan ske i till exempel main,

```
import java.io.*;

public class TestComplex {

    public static void main(String args[]) {
        Complex x = new Complex(3,4);
        Complex y = new Complex(5,6);
        Complex z = new Complex();

        z.sum(x,y);

        System.out.println("Summan är: " + z.toString());
    }

}
```

Hur kommer `static` in i bilden?

Både variabler och metoder kan deklareras med eller utan `static`.

Allt som är `static` lagras i ett för alla klassens instanser gemensamt klassobjekt, medan instansvariabler lagras i de olika instansobjekten.

Med andra ord: står det `static` framför finns det en gemensam kopia, står det inte `static` finns det en kopia för varje instans.

I tennsoldatsanalogin har alla samma designer, som deklareras `static`, men de skiljer sig i färg, vilket vi sparar som en instansvariabel.

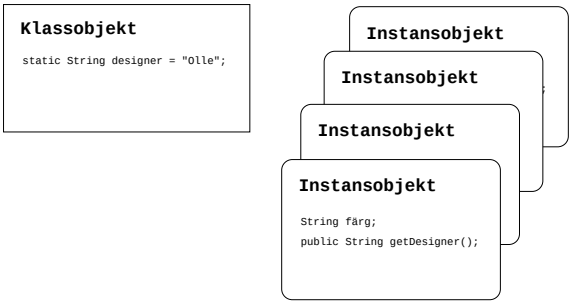
```
public class TennSoldat {

    static String designer = "Olle";
    String färg;

    TennSoldat(String färg) {
        this.färg = färg;
    }

    static String getDesigner() { return designer; }

    public String toString() {
        return designer + "-tennsoldaten är " + färg;
    }
}
```

Skilj på klassobjekt och instansobjekt

Vi repeterar.

Det finns bara ett klassobjekt, i vilken alla klassens `static` metoder och variabler ligger.

Det kan finnas flera instansobjekt, var och en med sin egen kopia av instansvariablerna.



Instansobjekt skapas av en konstruktor som anropas med hjälp av `new TennSoldat()`.

Av detta följer...

Staticmetoder kan bara använda sig av staticvariabler, medan instansmetoder kan använda både instansvariabler och staticvariabler.

Anledning till detta är att en staticmetod inte kan veta vilken av de olika instanserna den ska titta i för att hitta "rätt" färg.

Staticmetoder kan anropas utan att behöva skapa en instans först.

```
import java.io.*;

public class TestTennSoldat {

    public static void main(String[] args) {
        TennSoldat s = new TennSoldat("Blå");
        String d = TennSoldat.getDesigner(); // Anrop av static metod

        System.out.println("getDesigner() är static, retunerar " + d);

        System.out.println(s.toString());
    }

}
```

Notera hur staticmetoden `getDesigner()` anropas i vår kod! Den är inte instansierad.

Hur skickar vi parametrar?

När vi gör ett anrop till en metod med en variabel som argument är det *alltid* en kopia av den variabeln som skickas.

```
import java.io.*;

public class MetodAnrop {
    public static void main(String[] inparametrar) {
        int x = 4, y = 7, summa;

        summa = sum(x, y); // sum anropas och x kopieras till a, y till b
        System.out.println(x + " + " + y + " = " + summa);
    }

    public static int sum(int a, int b) {
        int s = a + b;
        a = 12;
        System.out.println("Lokala variabeln a = " + a);
        return s;
    }
}
```

Det betyder att ändrar vi på variabelns innehåll inuti metoden så påverkar det inte originalet utanför.

```
>javac MetodAnrop.java
>java MetodAnrop
Lokala variabeln a = 12
4 + 7 = 11
```

Vi ser att `a` får värdet 12 i metoden `sum` men att `x` förblir lika med fyra i `main`-metoden.

Likadant med referensvariabelanrop

Tänk på att innehållet i en referensvariabel är just referensen som pekar på ett objekt. Om vi kopierar den pekar den fortfarande på samma objekt.

Nu kommer det viktiga. När vi anropar en metod med ett objekt är det egentligen en referensvariabel pekande till objektet som vi skickar.

Det tas alltså ingen ny kopia av objektet utan bara av referensvariabeln.

Kopian på en referensvariabel pekar på samma objekt som original referensvariabeln!

Vi kan alltså modifera original objektet inifrån metoden genom att arbeta med referensvariabler.

Det är med andra ord samma regler som gäller vid funktionsanrop som vid tilldelning!

Objekten kopieras inte, bara referenserna.

Ett exempel på referensanrop

```
public class Kaka {
    String typ;
    boolean kvar;

    Kaka(String typ) { this.typ = typ; kvar = true; }

    public String toString() {
        if(kvar)
            { return typ + "skaka"; }
        else
            { return "Kaka uppäten..."; }
    }

    void ätaKaka() {
        System.out.println("Mums, mums... God " + typ + "skaka!");
        kvar = false;
    }
}
```

som anropas ifrån `main`-metoden i följande klass

```
import java.io.*;

public class StoraSyster {
    public static void main(String[] inparametrar) {
        Kaka k = new Kaka("morot");

        System.out.println("Låna " + k + " till lillebror.");
        lånaKakaTillLillebror(k);
        System.out.println(k);
    }

    public static void lånaKakaTillLillebror(Kaka lillebrors) {
        lillebrors.ätaKaka();
    }
}
```

Lillebror äter upp kakan!

Ifall det har skapats en lokal kopia av vår `Kaka` skall lillebror kunna äta upp den lånade kakan utan att storasystern förlorar den.

Vi kompilerar och kör vårt exempel.

```
>javac StoraSyster.java
>java StoraSyster
Låna morotskaka till lillebror.
Mums, mums... God morotskaka!
Kaka uppäten..
```

Kakan är uppäten! Lillebror har följt referensen och hittat originalobjektet och ätit upp det.

Vad lär vi oss av detta...

Man kan inte lita på lillebror?
Man kan inte äta kakan och ha den kvar?
eller...
Med referensvariabler kan vi enkelt manipulera våra orignalobjekt direkt!

Varför är detta viktigt?

Anledningen till att inte kopior skickas är effektivitet. Det tar tid att göra kopior, minne måste läggas undan till objektet, konstruktorer köras och instansvariabler kopieras.

Det är också trevligt att kunna manipulera objekten direkt inifrån metoder.

Men varför måste vi ha koll på detta?

Antag att vi av misstag ändrar på ett objekt inuti en metod i tron att det är en lokal kopia. Det kan vara mycket svårt att hitta sådana fel i ett stort program.

Vi spar med andra ord tid.

Nu lämnar vi klasser och instanser

Hur du skriver koden kan ibland vara lika viktigt som vad det är du skriver.

Samma metod kan skrivas på många olika sätt, alla är inte lika bra.

Snygg kod är lättare att förstå.

Felsökning går snabbare.

Lättare att upptäcka fel direkt, före kompilering!

Vad ska vi då tänka på?

Variabelnamn och intendering

Det är viktigt att metoder och variabler får namn som återspeglar deras funktion.

Ostädad kod kan vara mycket svårläst.

Ett skräckexempel,

```
import java.io.*;
public class Obegriplig {
private static int grrrrrrr, grrrrr = 0;
public static void main(String[] a) { int grrrrrr = 4;
for(int grrrr = 1, grrrrr = 1; grrrr <= grrrrrr; grrrr++)
{ grrrrr *= grrrr; }
System.out.println("4! = 1*2*3*4 = " + grrrrr);
while(grrrrr < 27)
System.out.println("Grrrrrrrr!");
grrrrr++; }
}
```

Kommer den här koden att kompilera?

Vad händer när den körs?

Beräknar den fyrfakultet rätt?

Hur många gånger skriver programmet ut texten “Grrrrrrrr!” på vår skärm?

Oändlig loop!

```
>javac Obegriplig.java
>java Obegriplig
4! = 1*2*3*4 = 0
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
Grrrrrrrr!
^C
```

Vi får en oändlig loop... kan ni se varför?

Om vi städar upp koden...

Några intenderingar och väl valda variabelnamn gör felsökningen lättare.

```
import java.io.*;

public class Begriplig {
    private static int k, prod = 0;

    public static void main(String[] a) {

        int sista = 4;

        for(int i = 1, prod = 1; i <= sista; i++) {
            prod *= i;
        }

        System.out.println("4! = 1*2*3*4 = " + prod);

        while(prod < 27)
            System.out.println("Grrrrrrrr!");
        prod++;

    }
}
```

...ser vi felen.

Den lokala variabeln `prod` döljer den privata klassvariabeln `prod` inuti for-loopen. Resultatet av uträkningen sparas i den lokala variabeln, men så snart vi lämnar for-loopen existerar den inte längre utan `prod` svarar då mot klassvariabeln `prod` som är initierad till noll.

Detta leder till att fakulteten beräknas fel!

Den oändliga loopen uppkommer därför att `while`-slingan saknar måsvingar, indenteringen avslöjar att `prod++` ligger utanför loopen.

Utelämnar man måsvingarna är det bara raden direkt efter `while`-loopen som upprepas.

Eftersom loopenvariabeln inte ökar, kommer `while`-slingan aldrig att avslutas.

Kom ihåg!

Var försiktiga med loopvariabler.

Se upp med synbarhet, lokala variabler döljer andra variabler med samma namn!

Med andra ord, var aktsam då du återanvänder variablenamn.

Måsvingar runt `for`- och `while`-loopar.

Indentera din kod, dvs tryck på tab när du står på en rad i emacs. Observera att filens namn måste sluta på `.java` för att det ska fungera.

Välj förklarande variabel och metodnamn.

Det viktigaste idag var dock klasser och instanser.

Skulle du kunna förklara hur de hänger ihop?

Tänk på tennsoldaterna...