Henrik Bergström
DSV SU/KTH
1999-04-06

Java

Detta kompendium skrevs ursprungligen ht-98 för kursen DSVL2:2 Programspråksteori. Kompendiet gör vissa förutsättningar, som tex att man jobbar i en terminalbaserad miljö, men borde gå att använda utan större problem även om dessa inte är uppfyllda.

Till denna andra version har bara mindre förändringar gjorts, där den viktigaste är tillägget av switch-satsen bland styrstrukturerna. Det finns alltid en uppdaterad version av kompendiet att hämta på: http://www.dsv.su.se/~henrikbe/download.html

Om ni hittar några fel, oklarheter etc rapportera dem gärna till mig via email: henrikbe@dsv.su.se Däremot kommer frågor troligen att ignoreras eftersom jag inte har speciellt mycket tid att ägna åt detta projekt nu för tiden.

1. Grundläggande java

1.1. Ett första exempel

Nedanstående är ungefär det kortaste fungerande program man kan skriva i Java. Det enda det gör är att skriva ut texten ”Hello World!” i ett terminalfönster.

```
public class HelloWorld {
          // Detta är en kommentar till det första enkla exemplet
          // Den gäller till slutet på raden
          public static void main(String[] args) {
                    System.out.println("Hello World!");
          }
}
```

För att köra exemplet skriver ni in det i en fil som heter HelloWorld.java och skriver sedan följande vid terminalprompten:

- javac HelloWorld.java
- java HelloWorld

Observera att Java skiljer på stora och små bokstäver, så man måste se till att skriva av det exakt. Det gäller för övrigt även filnamnet. Det första kommandot kompilerar koden och skapar då en fil som heter HelloWorld.class, och det andra kommandot kör programmet. Observera att Java inte kompilerar till ett direkt körbart program, utan det måste köras av ett annat program, det som i exemplet ovan heter just ”java”. Om man har många Java-filer som ska kompileras samtidigt kan man använda kommandot:

- javac *.java

som kompilerar alla filer som slutar med ”.java”.

Java-kompilatorn är ibland lite för smart när den försöker bestämma vilka filer som behöver kompileras om, vilket kan leda till att ändringar inte träder i kraft. Om man misstänker att det har hänt kan man lösa det genom att ta bort alla filer som slutar på ”.class”.

1.2. Strukturen hos ett Java-program

Allting i Java måste ligga inom en klass, vilket bland annat betyder att ett program även är en klass. I exemplet ovan kallas klassen HelloWorld. För att förvandla en klass till ett körbart program måste det finnas en speciell metod i den som **måste** se ut så här:

```
public static void main(String[] args) {}
```

Det enda som får vara annorlunda är namnet på parametern, och naturligtvis att det får stå programsatser inom hakparenteserna. Det hela läggs i en fil med **samma** namn som klassen följt av ”.java”. Generellt gäller att alla klasser som är ”public” (förklaras i sektion 4.3) **måste** ligga i en fil med samma namn som klassen + ”.java”. Oftast lägger man även andra klasser i egna filer för att underlätta läsbarheten.

2. Datatyper

2.1. Primitiva datatyper

Java har följande primitiva datatyper:

Namn	**Typ**	**Default**
boolean	Logiskt värde, true eller false	false
char	Unicode-tecken	0
byte	Heltal	0
short	Heltal	0
int	Heltal	0
long	Heltal	0
float	Flyttal	0.0
double	Flyttal	0.0

2.2. Operatorer

Java härstammar i grund och botten från C, vilket bland annat betyder att det har en massa operatorer. Alla är inte nödvändiga att kunna så vi nöjer oss med de vanligaste:

Operator	**Vad den gör**
`+`	Plus samt strängkonkatenering
`-`	Minus
`++, --`	Motsvarar ungefär inc och dec i Pascal med tillägget att de har en inbyggd tilldelning, dvs a++ (eller ++a) ökar på a med 1
`!`	Icke, ”not”
`(typ)`	Typomvandling, casting
`<, >, <=, >=`	Mindre än, större än etc.
`==, !=`	Lika med, skilt från
`*, /, %`	Multiplikation, Division, Rest
`=`	Tilldelning
`*=, /=, %=, +=, -=`	Tilldelning och operator
`&&`	Och, ”and”
`||`	Eller, ”or” (exklusivt)

2.2.1. Exempel

Vissa operatorer är lite förvirrande innan man har vant sig vid dem, följande exempel kanske underlättar något. Operatorerna ++ och -- brukar vara speciellt förvirrande, de har nämligen olika betydelse beroende på om de står framför eller bakom den variabel de opererar på. Observera även att tilldelning görs med = medan jämförelse görs med ==.

```
int a,b,c;
double d;

a=5;
b=2;
c=0;

a++; // Öka på a med ett
++a; // Öka på a med ett
--a; // Minska a med ett
a--; // Minska a med ett

b=a++; // Tilldela b a:s värde och öka
       // sedan på a med 1 dvs efteråt:
       // a=6, b=5

c=++a; // Öka på a med 1 och tilldela c
       // det nya värdet, dvs efteråt:
       // a=7, c=7

a+=1;   // Ett annat sätt att öka på a med 1
a/=5;   // Dividera a med 5 och lägg resultatet i a
        // Dvs, samma som a=a/5;

d=13.3;
a=d;   // Detta går inte, a är inte
       // ett flyttal
a=(int)d; // Gör temporärt om d till ett heltal
          // (typomvandling/casting)
          // Dvs, efteråt: a=13, d=13.3
```

2.3. Icke-primitiva datatyper

Förutom de primitiva datatyperna i tabellen ovan har Java egentligen bara två datatyper till, arrayer och objekt. Dessa kallas ibland för refererande datatyper (eng *reference types*) eftersom de hanteras med hjälp av referenser. Dvs, om man tex skickar med en array som parameter till en funktion skickar man egentligen bara en referens till arrayen, och ändrar man någonting i den så ändrar man på det man skickar in, **inte** på en kopia. Objekt är instansieringar av klasser, och kommer att behandlas i sektion 4.1.

2.3.1. Kopiering och tilldelning av icke-primitiva datatyper

Eftersom de icke-primitiva datatyperna hanteras med hjälp av referens blir det vissa problem när man ska kopiera eller jämföra dem. I stort sett stämmer dessa problem överens med de man stöter på med pekare, dvs tex detta kan inträffa:

```
// MinIckePrimitivaDatatyp är en mycket enkel datatyp
// med ett enda attribut (variabel) i den som heter
// "medlem". För tillfället kan ni se den som en
// record i Pascal.

// Deklarera två variabler av typen MinIckePrimitivaDatatyp
MinIckePrimitivaDatatyp V1, V2;


// Skapa det faktiska utrymmet för variablerna
V1=new MinIckePrimitivaDatatyp();
V2=new MinIckePrimitivaDatatyp();

// Tilldela attributet medlem ett värde
V1.medlem=1;
V2.medlem=2;

// Skriv ut värdena på "medlem"
System.out.println(V1.medlem); // Ger 1
System.out.println(V2.medlem); // Ger 2

V2=V1;

V2.medlem=10;

System.out.println(V1.medlem); // Ger 10 eftersom
                               // V1 och V2 refererar
                               // till samma sak
```

För att faktiskt göra en kopiering brukar man därför göra en speciell metod som utför det hela. Tyvärr är det inte som i tex Eiffel att denna är standardiserad[1], utan man får kalla dem vad man vill. Själv brukar jag använda `copy` eller `clone`, men det är som sagt var inget som helst krav.

Allt är dock inte negativt med Javas referenser, tvärtom, de har en mycket stor fördel jämfört med tex pekare i Pascal: de behöver inte tas bort. Java inser automatiskt när ett objekt inte längre behövs och tar då bort det självt. Om man vill vara lite snäll kan man tilldela variabeln värdet **null** (motsvarar nil i Pascal), som betyder inget värde, för att hjälpa Java på traven med att bestämma när ett objekt ska bort.

2.3.2. Arrayer

Syntaxen för att deklarera en array är följande:

```
Typnamn variabelnamn[];
```

eller

```
Typnamn variabelnamn[] = initialisering;
```

Värt att veta är att Javas arrayer **alltid** börjar på index 0 och går till antal-1.

Liksom med övriga variabler kan inte en array deklarerad enligt den översta versionen användas innan den har initialiserats. Om datatypen är en icke-primitiv typ (en klass, se sektion 4.1 nedan) måste även elementen skapas, lämpligen i en for-loop. Följande exempel kanske förklarar en del:

[1] Här ljuger jag lite grann, det finns faktiskt ett ”interface” (se sektion 4.4.1) som heter clonable och som ger tillgång till en metod som heter clone.

```
// Deklarera en array av heltal med 10 positioner
int a1[]=new int[10];

// En flerdimensionell array
int a2[][]=new int[5][7];

// Deklarera och initialisera en array av
// objekt av typen MinIckePrimitivaDatatyp
MinIckePrimitivaDatatyp a3[]=new MinIckePrimitivaDatatyp[5];
for(int i=0; i<5; i++){ a3[i]=new MinIckePrimitivaDatatyp(); }
```

2.3.3. Strängar

En speciell klass som är bra att känna till är String som hanterar strängar. String är lite ful i och med att den inte är en primitiv datatyp, men ändå fungerar tilldelning ungefär som den gör för dessa, och det finns ytterligare en operator +, som lägger ihop två strängar.

Viktigt att känna till om Javas strängar är också att de inte kan ändras, dvs, det finns ingen motsvarighet till att göra str[5]='a' i Pascal. Däremot går det alldeles utmärkt att tilldela en sträng ett nytt värde som i detta exempel:

```
String s="Hej";
s=s+" då!";
```

För att jämföra strängar kan man inte använda =, strängar är inte primitiva datatyper, utan man måste använda metoden `equals`:

```
String s1="Hej";
String s2="Hej";
String s3="då";

s1==s2 // Detta är falskt, s1 och s2 är två olika strängar
s1.equals(s2) // Detta är sant, innehållet i dem är samma
s1.equals(s3) // Detta är falskt, innehållet är olika
```

3. Styrstrukturer

3.1. If-satsen

Syntaxen för en if-sats är:

```
if(villkor){
// Det som ska göras om villkoret är sant
}
```

eller

```
if(villkor){
// Det som ska göras om villkoret är sant
} else {
// Det som ska göras om villkoret inte är sant
}
```

Villkoret kan vara vad som helst som "genererar" ett boolskt värde, tex: en boolsk variabel, ett anrop på en funktion som returnerar en boolean, eller ett vanligt villkor tex "x>1". Däremot är det inte som i C/C++ att man kan sätta in heltal.

3.2. For-loopen

Syntaxen för en for-loop är följande:

```
for(initialisering; loop-villkor; ändring av loop-variabeln){
// Det som ska göras
}
```

Kroppen på for-satsen består alltså av tre delar där den första sker innan man går in i loopen, den andra kollas en gång per varv för att se om man är klar, och den tredje sker precis innan kollen på att man är klar. En lite rolig sak man bör känna till är att vilket som helst av dessa fält kan lämnas blankt.

```
// Skriver ut talen 1 till 10
for(int i=1; i<=10; i++){
        System.out.println(i);
}

// Kortaren version av ovanstående
// Obs att i ökas när kollen på om man ska fortsätta
// görs, vilket leder till att man måste ändra
// hela villkoret.
for(int i=0; i++<10; ){
        System.out.println(i);
}

// En loop som man aldrig kommer att gå in i
for(; false;){
      System.out.println("Detta kommer aldig att skrivas ut!");
}

// En oändlig loop
for(;;){}
```

3.3. While-loopen

While-loopen kan användas både för pretest och posttest-loopar med följande syntax:

```
while(villkor){
// Det som ska göras
}
```

eller

```
do{
// Det som ska göras
} while(villkor);
```

Följande exempel visar skillnaden:

```
// Skriver ut 0 till 9
int i=0;
while(i<10){
   System.out.println(i++);
}
```

```
// Skriver ut 1 till 9
i=0;
do{
   System.out.println(i++);
}while(i<10);
```

3.4. Switch-satsen

Switch-satsen används för att styra programflödet beroende på värdet hos ett uttryck enligt följande syntax:

```
switch (uttryck)
{
     case värde1:   Kod
     case värde2:   Kod
     ...
     case värdeN:   Kod
     default:       Kod
}
```

eller

```
switch (uttryck)
{
     case värde1:   Kod
                    break;
     case värde2:   Kod
                    break;
     ...
     case värdeN:   Kod
                    break;
     default:       Kod
                    break;
}
```

Skillnaden ligger som ni ser i nyckelordet break som kan ses som en kontrollerad version på goto. När programmet träffar på ett break hoppar det omedelbart till slutet på blocket, dvs i det här fallet till }. Uttrycket inom parenteserna måste generera antingen en int, short, byte eller char, och kan vara tex en variabel eller en funktion. Default-satsen är frivillig. Om den finns och värdet av uttrycket inte finns explicit uppräknat på en case-rad så hoppar programflödet dit.

Följande kod visar på lite olika versioner på switch:

```
int i;
i=(int)Math.random()*10;     // Ge i ett slumpmässigt värde mellan
                             // 0 och 10
switch (i)                   // Det hade även gått med
                             // switch((int)Math.random()*10)
{
     case 0:        skriv("Noll");
                    break; // Här avbryts switch-satsen om i är 0
     case 1:
     case 2:
     case 3:        skriv("Litet tal");          // Detta kommer
                             // om i är 1,2 eller 3.
                    break;
```

```
        // Eftersom det inte finns något break mellan nedanstående
        // rader kommer båda att skrivas ut om i är 4, men bara den
        // andra om i är 5
        case 4:         skriv("Om i är 4 kommer både detta");
        case 5:         skriv("och detta att skrivas ut");
                        break;
        default:        skriv("Talet var större än 5");
                        break;    // Detta behövs egentligen inte,
                                  // men gör inget
}
```

4. Objektorientering i Java

4.1. Klasser och Objekt

En av de saker som nybörjare på objektorientering brukar ha det svårast med är relationen mellan klasser och objekt. Den klassiska förklaringen på begreppen är att klasser är en mall för hur objekten ska se ut. Om man tex har klassen ”Person”, så skulle ”Nisse”, ”Kalle” och ”Lisa” kunna utgöra objekt av den klassen. En annan förklaring kan ges med en jämförelse med Pascals records, då skulle klassen utgöras av det man skriver under ”type”-avsnittet, och objekten vara de variabler man skapar. En annan vanlig terminologi är att kalla objekten för **instanser** av en klass.

Syntaxen för att deklarera en klass är ungefär så här:

```
[säkerhetsnivå] class klassnamn{
    [attribut synliga för hela klassen]
    [metoder i klassen]
}
```

För den som har svårt med terminologin är ett attribut en variabel (eller konstant) och en metod en funktion (procedurer finns inte i Java även om de kan simuleras med hjälp av nyckelordet `void`). Variabler har vi deklarerat många i exemplen ovan, så den syntaxen bör vara klar nu, men syntaxen för metod-deklarationer kanske kan behöva en förklaring:

```
[säkerhetsnivå] [static] [returtyp] metodnamn([parametrar]){
          // Programsatser
}
```

De olika säkerhetsnivåerna gås igenom i sektion 4.3, och nyckelordet `static` i sektion 4.7. Returtypen är antingen en primitiv datatyp, en klass, eller `void`, vilket betyder att metoden inte returnerar någonting.

För att sedan skapa instanser (objekt) av klassen skriver man så här:

```
klassnamn variabelnamn = new klassnamn();
```

Eventuellt skall det vara parametrar inom parenteserna, men mer om det i avsnittet om konstruktorer, 4.6.

För den som är van att jobba med pekare kan det kännas lite ovant att man inte har någon riktig koll på när objekt försvinner. Det är upp till Java självt att se till att ta bort dem när de inte längre behövs. För att hjälpa till med detta kan man dock sätta variabeln till `null` för att signalera att man är klar med den.

Om man vill komma åt något i ett objekt använder man samma syntax som när man vill komma åt variabler i en record i Pascal, alltså ”`variabelnamn.attributnamn`” eller ”`variabelnamn.metodnamn(parametrar)`”. Om ett attribut (eller returtypen av en funktion)

har egna metoder, det vill säga den är ett objekt, så kan man lägga på ytterligare nivåer på anropen som tex: ”`variablenamn.attributnamn.metodnamn(parametrar)`”.

4.2. Abstrakta klasser

Ibland vill man inte behöva specificera hela klassen direkt. Då kan man i stället välja att deklarera den som abstract. Det kan även användas när man vill ha en klass som det inte ska gå att skapa instanser av.

```
abstract class AC1{
   public abstract void abstraktMetod();
}

class IC1 extends AC1{
   // IC1 måste implmentera metoden abstraktMetod()
   // eller deklareras som abstrakt själv
   public void abstraktMetod(){
     // Gör något
   }
}

abstract class AC2{
   public void ickeabstraktMetod(){
          // Gör något
   }
}

// IC2 måste inte implementera ickeabstraktMetod()
// eftersom den redan är implementerad i AC2
class IC2 extends AC2{
}

class AbstractExempel {
  public void metod(){
    AC1 ac1; // Det går att deklarera variabler av
             // abstrakta klasser.
    ac1=new IC1();
                    // Men inte att skapa instanser av dem
    ac1=new AC1(); // Detta går alltså inte

    ac1.abstraktMetod(); // Metoder deklarerade i en
                         // abstrakt klass kan dock
                         // anropas, man har ju lovat
                         // att de ska finnas
  }
}
```

4.3. Information hiding

Java har tre nyckelord som man kan använda för att sätta olika ”säkerhetsnivåer” på klasser, metoder och attribut:

- `private`, är den hårdaste, dessa kommer bara klassen själv åt.
- `protected`, är lite snällare, dessa kommer klassens barn åt liksom andra klasser som ligger i samma fil.

- `public` till sist är den snällaste, den kommer alla åt. Observera att alla klasser som är public **måste** ligga i en fil som heter *klassnamnet.java*.

Attribut och metoder som man inte specificerar skyddet på sätts automatiskt till `public`[2]. Värt att veta är att det av vissa teoretiker inom området anses som fult att använda `protected` och `public` på attribut efter som det bryter mot inkapslingsprincipen. Istället anser dessa att man ska göra speciella metoder som har hand om alla ändringar av attribut. På så sätt kan man garantera att ingen klåfingrig programmerare går in och ändrar dem själv, vilket underlättar om man tex vill byta representationsform. För att inte klottra ner kodexemplen för mycket har jag i stort sett undvikit detta utom på något enstaka ställe där det hjälpte till att illustrera ett exempel.

Normalt ska klasser som någon annan ska använda sig av deklareras `public` och följaktligen ligga i en fil med samma namn som klassen, men så länge man inte använder sig av ”packages” för att gruppera klasserna spelar det ingen större roll. För att undvika framtida problem tag dock för vana att skriva ut det lilla ordet.

```
// Klasserna InformationHidingExempel och C2 ligger i samma fil,
// InformationHidingExempel.java
public class InformationHidingExempel {
       private static void metodEtt(){
          // Gör något
       }

       protected static void metodTva(){
         // Gör något
       }

       public static void metodTre(){
         // Gör något
       }

}

class c2 {
      public void c2(){
         InformationHidingExempel.metodEtt(); // Detta går inte
                                              // metoden är private

         InformationHidingExempel.metodTva(); // Detta går eftersom
                                              // klasserna ligger i
                                              // samma fil

         InformationHidingExempel.metodTre(); // Detta går alltid
      }
}

// C1 ligger i filen C1.java
public class c1 extends InformationHidingExempel {
      public void c1(){
         InformationHidingExempel.metodEtt(); // Detta går inte

         InformationHidingExempel.metodTva(); // Detta går eftersom
```

[2] Inte sant, egentligen till ett mellanting mellan public och protected, men för de flesta kommer det att verka som om den är public. För att undvika problem, tag dock för vana att sätta ut säkerhetsnivån.

```
                                                    // c1 ärver från
                                                // InformationHidingExempel
                                                  // se nästa sektion

         InformationHidingExempel.metodTre(); // Detta går alltid
      }
}
```

4.4. Arv

Ett centralt begrepp inom objektorienteringen är arv. Detta är en mycket viktig funktionalitet som bland annat gör att man slipper upprepa kod. Arv implementeras i Java med hjälp av nyckelordet `extends`:

```
public class Parent {
...
}

public class Child extends Parent{
...
}
```

Nu har Child ärvt alla Parents publika metoder och attribut. Child kan även använda sig av de metoder och variabler som är protected i Parent. Dessutom kan man till varje variabel som är av typen Parent (inklusive parametrar till en funktion) skicka en variabel av typen Child. En sak man inte kan göra är dock att via en variabel som är deklarerad som Parent komma åt en av Childs metoder eller attribut, **även** om man vet att det som variabeln refererar till faktiskt är en Child (om man inte typomvandlar variabeln):

```
// Filen Parent.java
public class Parent {
       private void ParentPrivateMetod(){
       // Gör något
       }
       protected void ParentProtectedMetod(){
       // Gör något
       }

       public void ParentPublicMetod(){
          ParentPrivateMetod(); // Ok, en metod i parent
                                // får anropa vilken annan
                                // metod i parent som helst
       }
}

// Filen Child.java
public class Child extends Parent{
       private void ChildPrivateMetod(){
       // Gör något
       }
       protected void ChildProtctedMetod(){
       // Gör något
       }
       public void ChildPublicMetod(){
          ParentPrivateMethod();  // Fungerar inte, metoden
                                  // är privat för Parent
```

```
          ParentProtectedMetod(); // Ok, Child ärver från
                                  // Parent och får då
                                  // använda dess protected
                                  // metoder
          ParentPublicMetod();    // Ok
          ChildPrivateMetod();    // Ok
       }
}

// Filen ArvExempel.java
public class ArvExempel {

       public static void metod1(Parent p){
              // Gör något
       }

       public static void metod2(Child c){
              // Gör något
       }

          public static void main(String[] args) {
               Parent p=new Parent();
               Child c=new Child();

               p.ParentPrivateMetod(); // Går inte
                                       // private
               p.ParentProtectedMetod(); // Går inte
                                         // protected
               p.ParentPublicMetod(); // Ok

               c.ParentPrivateMetod(); // Går inte metoden
                                       // finns inte i Child
               c.ParentProtectedMetod(); // Går inte
                                         // protected
               c.ParentPublicMetod(); // Ok Child har ärvt
                                      // metoden

               c.ChildPrivateMetod(); // Går inte
                                      // private
               c.ChildProtectedMetod(); // Går inte
                                        // protected
               c.ChildPublicMetod(); // Ok

               metod1(p);
               metod1(c); // Går bra, c är ju
                          // även en Parent

               metod2(p); // Går inte, en Parent
                          // är ingen Child
               metod2(c);

               p=c;      // p pekar nu en Child

               p.ChildPublicMetod(); // Går inte, datorn vet
                                     // bara att p är en
```

```
                                        // Parent, inte att den
                                        // egentligen är en
                                        // Child

                // Däremot går det bra om vi typomvandlar om p
                // till en Child
                ((Child)p).ChildPublicMetod();

                // Gör om p till en ren Parent
                p=new Parent();

                // Försök typomvandla p till en Child, detta
                // ska ge ett ClassCastException vilket hanteras
                // av satserna try och catch, se kapitel 5
                try{
                    ((Child)p).ChildPublicMetod();
                }catch(Exception e1){System.out.println(e1);};
            }
}
```

Generellt kan man säga att arv används för tre saker:

1. Lägga till funktionalitet, som i exemplet ovan där Child har en metod mer än Parent (kom ihåg att de andra metoderna inte syns utanför klassen och följaktligen kan ignoreras av den som bara ska använda sig av den)
2. För att ändra på funktionalitet vilket diskuteras i sektionen om överlagring (4.5).
3. För att tillåta så kallad **polymorfism**.

Det sistnämnda är ett fint namn på egenskapen att vid körning av programmet kunna bestämma tex vilken metod som ska anropas. För att konkretisera det hela, antag att man har två klasser K1 och K2 som båda har ärvt från en abstrakt klass AK. I AK finns det en abstrakt metod M deklarerad som både K1 och K2 implementerar på sitt eget sätt. Antag vidare att man har en variabel ”a” av typen AK. Eftersom klassen är abstrakt kan man dock inte skapa ”rena” instanser av den utan måste använda sig av dess subklasser. Om man nu gör ett anrop i stil med ”`a.M()`”, vilken version av M är det som ska användas? Det beror på vad ”a” egentligen är: är ”a” egentligen en K1:a ska K1:s M användas och tvärt om. Det är denna egenskap, att kunna bestämma vilken metod som ska användas i ett visst ögonblick som brukar kallas för polymorfism.

I kod skulle detta exempel kunna se ut så här:

```
AK a1 = new K1();
AK a2 = new K2();
a1.M(); // Nu anropas den version av M som finns i K1
a2.M(); // Nu anropas den version av M som finns i K2
```

4.4.1. Multipelt arv

Vissa objektorienterade språk, tex C++ och Eiffel, tillåter att man ärver från flera föräldrar. Detta är en mycket nyttig egenskap i vissa situationer. Tänk tex att man vill simulera olika typer av fordon, i ett sådant läge skulle en flygbåt kunna ärva både från flygplan och fartyg eftersom den har alla deras egenskaper. Tyvärr ställer det också till en del mycket allvarliga problem. I exemplet ovan har troligen både flygplan och fartyg ärvt från en abstrakt klass fordon som tex kan ha metoden färdas(x,y), som förflyttar fordonet till koordinaterna x och y. Tyvärr är det nu troligen så att både fartyg och flygplan har definierat sin egen version av denna metod, och vilken ska då flygbåt använda sig av? Svaret är naturligtvis att det beror på omständigheterna, ibland den ena, ibland den andra.

För att undvika sådana problem finns det ett antal olika mekanismer som kan användas, men ingen av dem är riktigt snygg. En hel del språk, däribland Java, har därför valt att inte använda sig av multipelt arv, men tyvärr för det också med sig problem, nämligen de gånger när det faktiskt är nyttigt att använda det. I stället använder sig Java av något som många tycker är en snyggare lösning på problemet, interfaces. Att gå in mer noggrant på dessa ligger lite utanför ramen för det här kompendiet, men om man säger att det fungerar som en **rent** abstrakt klass, dvs en klass **helt** utan metodkod, så har vi sagt nog för att de som skulle vilja använda dem ska kunna göra det. Man kan säga att ett interface är ett löfte om vilka metoder och variabler som ska finnas i en klass. Följande exempel får tjäna som fortsatt vägledning:

```
interface int1{
   // Detta deklarera inte en variabel utan
   // snarare en konstant, otydligt, men så
   // är det
   public int i=1;
}

interface int2{
   public int i=2; // Detta kan strula till det lite
                   // om man implementerar både int1
                   // och int2 och tror sig veta vad
                   // i ska ha för värde
   public boolean b=false;

   // Denna metod hade kunnat finnas även i int1, och
   // en klass hade kunnat implementera både int1 och
   // int2 som cla2 nedan gör utan problem efter som
   // det snarare är ett löfte om att det någonstans
   // ska finnas en sådan metod än en riktig metod.
   public void metod();
}

// Denna klass behöver inte göra något eftersom interfacet
// int1 inte innehåller några metoder den måste implementera
class cla1 implements int1{

}

// Denna klass har nu fått attributet ifrån både int1 och int2
// samt från cla1 som ju fick det från int1.
class cla2 extends cla1 implements int1, int2{

      // Eftersom cla2 implementerar int2 måste
      // metod() implementeras eller klassen deklareras
      // som abstrakt
      public void metod(){
            // Gör något
      }
}

class InterfaceExempel {
      // Ett interface kan användas nästan överallt där
      // en klass kan användas
      public void metod(int1 i){
```

```
            // Man kan deklarera variabler från
            // interface
            int2 v=new cla2();

            // och då naturligtvis använda sig av de attribut
            // och metoder som det interfacet deklarerade
            v.metod();
      }
}
```

4.5. Överlagring

Ofta vill en klass som ärver från en annan ändra på vissa beteenden hos sin förälder. Antag till exempel att man vill göra en scrollbar som ändrar färg beroende var på skalan den befinner sig. Scrollbaren har då troligen en metod som anropas varje gång användaren klickar på scrollbaren. Om vi kallar den metoden för `klick(int nypos)` skulle det kunna se ut så här:

```
class ScrollBar {
   private int pos;
   // Fler attribut och metoder

   public void klick(int nypos){
      pos=nypos;
      // Gör övriga saker
   }
}

class FargadScrollBar extends ScrollBar{

   public void klick(int nypos){
       // Detta måste vi göra för att pos ska sättas
       // den är ju privat i ScrollBar
       super.klick(nypos);

       // Egen kod för att ändra färgen etc
       // ...
   }
}
```

Det som nu händer om användaren klickar på en instans av vår nya klass FargadScrollBar så anropas inte klicka i ScrollBar utan i stället anropas den i FargadScrollBar. Den i sin tur anropar klicka i ScrollBar för att vi inte ska behöva skriva om all kod från den, men om vi inte hade behövt någon kod därifrån hade vi kunnat strunta i detta. Därefter gör den de ytterligare saker som behövs för att färgen ska vara riktig, voila, vi har ändrat på beteendet hos ScrollBar. Som man kan se i exemplet finns det ett speciellt nyckelord ”`super`” som man kan använda när man behöver komma åt något i klassen ovanför men inte kan göra det direkt eftersom metoden i den egna klassen överlagrar (döljer) den.

4.6. constructor och finalize

När man skapar nya instanser av en klass använder man som ni kanske kommer ihåg en syntax som den här:

```
klassnamn variabelnamn = new klassnamn();
```

Vad som faktiskt händer är att minne allokeras för objektet och sedan anropas en speciell metod kallad konstruktor (eng *constructor*) som har till uppgift att initialisera objektet, först därefter är objektet tillgängligt för att användas. Typiska uppgifter för konstruktorn är att sätta värden på attribut, öppna filer, kolla att olika resurser är tillgängliga etc. Konstruktorer har alltid följande syntax:

```
[säkerhetsnivå] klassnamn([eventuella parametrar]){
  // Det som ska göras
}
```

Vill man ha flera olika konstruktorer vilket är rätt vanligt heter de alla samma sak men har olika parameteruppsättningar. För att komma åt en konstruktor inifrån en annan konstruktor använder man nyckelordet ”`this`” följt av de parametrar den konstruktor har man vill komma åt. På samma sätt använder man nyckelordet ”`super`” för att komma åt konstruktorer i ovanliggande klasser.

Viktigt att veta är att om man inte själv skriver en konstruktor så finns det alltid en default-konstruktor som inte tar några parametrar, det är den som används när man skapar något med ”`new klassnamn()`”. Om man däremot skapar en konstruktor själv så görs ingen default-konstruktor av systemet. Vill man ändå ha en får man göra den själv. Om man inte har en default-konstruktor i en klass **måste** klasser som ärver från dem deklarera konstruktorer som anropar en av förälderns konstruktorer. Detta är för att **alla** delar av objektet måste bli initialiserade någon gång.

På samma sätt som konstruktorn körs innan objektet egentligen finns, finns det en speciell metod, finalize, som körs när objektet tas bort. Den kan man använda för att tex stänga filer, spara undan information etc. Följande exempel illustrerar båda dessa metoder: I motsats till konstruktorer kan det bara finnas en finalize-metod som inte får ta några parametrar. Till skillnad från tex C++:s destruktorer finns det inget sätt att konstrollera när en finalize-metod kommer att köras vilket gör att de inte används lika mycket i Java som i C++. I stället gör man egna metoder som gör det som är nödvändigt för att avsluta användandet av objektet och anropar dem själv.

```
class PersonExempel {

  public PersonExempel() {
  }

  static public void main(String[] args) {
    Person p1, p2, p3;
    p1=new Person();
    p2=new Person("Henrik");
    p2=new Person("Linda");
    // Dessa kommer troligen inte att leda till att
    // finalize-metoden körs om inte maskinen man
    // sitter på har himla lite minne
    p1=null;
    p2=new Person("Rolf");
    p3=new SpeciellPerson();

    try{
    System.in.read();
    }catch(Exception e){};
    // Här någon gång kommer finalize-metoden att köras
    // för alla fem objekt som har skapats. Åtminstone
    // är det vad som borde ske, men vid kontroll verkar
    // de inte som om det stämmer. Bara de tre första
    // anropades. Vid en kontroll visade det sig att en
    // en boolean måste sättas. Ej gjort här.
```

```
  }
}

class Person {
  String namn;

  public Person() {
     // Anropa den andra konstruktorn med ett
     // default-namn
     this("Ingen aning");
  }

  public Person(String n){
     System.out.println("En person som ska heta "+n+" håller
                         på att skapas");
     namn=n;
  }

  public void finalize(){
     System.out.println(namn+" håller på att tas bort");
  }
}

class SpeciellPerson extends Person{
// Denna klass behöver inte deklarera någon konstruktor
// eftersom Person har en default-konstruktor, men vi gör
// det i alla fall för att visa anropet

   public SpeciellPerson(){
      super("Speciell Person");
   }

}
```

Behöver man då nödvändigtvis ha konstruktorer så fort man vill att variabler i en klass ska få värden direkt när den skapas? Svaret på den frågan är nej, det behövs inte. Om man alltid vill att en variabel ska ha samma värden när man skapar en instans av en klass kan man ge den ett värde direkt vid deklarationen som i det nedanstående exemplet:

```
class InitialiseringExempel{
   // Varje nytt objekt av denna klass kommer
   // att ha variabeln i satt till 1
   public int i=1;
}
```

4.7. Klassmetoder och variabler

I många av exemplen ovan har vi använt oss av nyckelordet `static` utan att säga något mer om det. Det används för att visa att tex en metod kan användas utan tillgång till en instans av klassen. Sådana metoder brukar kallas klassmetoder eller statiska metoder. På ett liknande sätt kallas statiska variabler (attribut) ibland för klassvariabler. Om det är en variabel betyder det att den är gemensam för alla instanser av klassen, dvs om ett objekt ändrar på den så ändras den för alla andra också. Viktigt att veta är också att en klassmetod **inte** kan anropa icke-statiska metoder eller använda sig av icke-statiska attribut i samma klass.

```
class StaticExempel {
   public static void statiskMetod(){
          ickestatiskMetod(); // Går inte, en statisk
                              // metod får inte anropa
                              // en som inte är statisk
                             // i samma klass
   }

   public void ickestatiskMetod(){
          statiskMetod();  // Går bra, en ickestatisk-
                           // metod kan anropa en statisk
   }
}

class AnnanKlass {

public void metod(){
   StaticExempel.statiskMetod(); // Går bra, man behöver inte
                                 // ha ett objekt för att
                                 // kunna anropa en statisk metod
}

}
```

Det finns ett litet problem med statiska variabler och konstruktorer, om man tex har följande klass:

```
class StaticTest{
 static int i;

 public StaticTest(){ i=1; }
}
```

Varje gång ett nytt objekt skapas kommer dess konstruktor att sätta i till 1. För att det ska fungera som man troligen tänkt sig och bara göras en gång ska alltså ovanstående se ut så här:

```
class StaticTest{
 static int i=1;

 public StaticTest(){
        // Gör annat
 }
}
```

Nu kommer bara det första objektet som skapas att sätta i till 1. När vi skapar fler objekt av klassen kommer den att behålla det värde den har blivit tilldelad. Ovanstående förklaring är lite förenklad, egentligen sker tilldelningen innan något objekt av klassen har skapats, så att man kan komma åt variabeln med hjälp av klassnamnet.

5. Undantag

5.1. Vad är undantag

Alla som har programmerat mer än någon enstaka rad vet att ibland inträffar fel som man inte har förutsett. Variabeln man ska dividera med kanske är 0, eller filen man försöker läsa ifrån kanske är lässkyddad. I sådan situationer genererar Java ett så kallat undantag (eng *exception*). Dessa kan man

fånga upp och hantera. Om man inte gör det kommer felet att avbryta körningen av den metod man befinner sig i och skicka undantaget vidare till den metod som anropade den. Om den i sin tur inte tar hand om undantaget kommer den att avbrytas och undantaget skickas vidare ända tills det når huvudprogrammet. Om det inte heller tar hand om undantaget kommer programmet att avslutas med ett felmedelande.

Vidare är det så att vissa klasser av undantag **måste** fångas på någon nivå. Om detta inte görs går det inte att kompilera programmet utan man kommer att få ett fel i stil med ”…Exception must be caught or declared to be thrown”.

5.2. Fånga undantag

För att fånga upp ett undantag använder man följande syntax:

```
try{
   // Kod som kan generera undantag
}catch(exception-typ variabelnamn){
   // Kod för att hantera undantaget
}
```

”exception-typ” är typen av undantag, tex ”java.io.IOException” för fel som har att göra med IO, eller ”java.lang.ArithmeticException” om man tex riskerar att dividera med 0. Om man vill fånga alla undantag använder man klassen ”Exception” som alla andra undantag ärver ifrån. Anledningen till att man måste ha ett variabelnamn är att ett undantag är ett objekt och följaktligen måste refereras till någonstans för att inte bli bortplockat av systemet. För att vänta på att användaren trycker på retur vid terminalen och skriva ut eventuella fel kan tex följande kod användas:

```
try{
    System.in.read();
}catch(Exception e){System.out.println(e);};
```

Ibland vill man kunna hantera flera typer av undantag. Då kan man enkelt lägga till flera catch-satser:

```
try{
    // Gör något
}catch(java.io.IOException variabelnamn1){
    // Kod för att hantera undantag av IO-typ
}catch(Exception variabelnamn2){
    // Kod för att hantera alla andra undantag
}
```

Om det finns kod som **måste** köras, oavsett om det inträffar ett undantag eller inte kan man lägga till ytterligare en sektion som betecknas med nyckelordet `finally`.Om någon del av koden inom try-blocket körs är också koden inom finally-blocket garanterad att köras:

```
try{
    // Gör något
}catch(undantags-typ variabelnamn){
    // Kod för att hantera undantaget
}
finally{
    // Kod som alltid MÅSTE köras, oavsett om
    // det inträffar ett undantag eller inte.
    // Den kommer att köras även om ett undantag
    // vi inte hanterar ovan inträffar.
}
```

5.3. Skicka undantag vidare

Ibland vill man inte fånga undantag precis där de uppkommer utan skicka dem vidare till den som anropade metoden där undantaget inträffade. Detta gäller framför allt när man skriver klasser som andra ska använda. Eftersom man inte vet hur de vill att ett speciellt undantag ska hanteras skickar man det bara vidare till dem, sedan är det deras uppgift att hantera det (eller skicka det vidare i sin tur). För att göra det använder man nyckelordet `throws`.

```
class ThrowsExempel{
  public void vantaPaReturn() throws java.io.IOException{
    System.in.read();
  }
}
```

5.4. Deklarera och kasta egna undantag

Undantag är nyttiga, och något som gör dem ännu nyttigare är att man lätt kan skapa egna och använda för att signalera när något fel som är speciellt för ens egen klass inträffar. För att deklarera ett eget undantag subklassar man helt enkelt klassen ”Exception” (eller någon av dess subklasser), och för att kasta ett undantag använder man nyckelordet `throw`. Följande kod kastar upp undantaget ParameterValueToLowException om den parameter man skickar in till metoden är mindre än 0.

```
class EgetUndantagExempel{
   public void kollaParameter(int p)
     throws ParameterValueToLowException{
      if(p<0){throw new ParameterValueToLowException();}
   }
}

class ParameterValueToLowException extends Exception {
   public ParameterValueToLowException(){
      // Sätt ett felmedelande, i det här fallet helt
      // enkelt namnet på den typen av undantag det
      // gäller
      super("ParameterValueToLowException");
   }
}
```

6. Input och Output från terminal

Alla metoder i Java **måste** ligga inom en klass. Därför finns det en speciell klass, `System`, som kan användas för att göra saker som att läsa in ifrån och skriva till en terminal. I exemplen ovan har vi använt några av de metoder som finns i `System`, men eftersom det är så pass centralt att man kan skriva och läsa till en terminal kan det vara på sin plats med en liten exkursion i strömmarnas underbara värld. I Java läser och skriver man i stort sett aldrig direkt från tex en fil eller en terminal. I stället använder man sig av en abstraktion som kallas strömmar (eng *stream*). Analogin är med en ström av enheter, tex tecken, som kommer till dig (läsning) eller går från dig (skrivning). Fördelen med detta är att man kan använda samma kod för att läsa från en fil som från en terminal.

I klassen System finns det tre strömmar:

- ”in” är den normala ”in-strömmen”, den används när man vill läsa in någonting och är av typen InputStream. Tyvärr är en ren InputStream lite begränsad i vad som man kan läsa från den, därför brukar man använda sig av en mer avancerad variant som i exemplet nedan.
- ”out” är den normala ”ut-strömmen”, den används när man vill skriva något. Viktigt att veta är att den är buffrad, så det är inte säkert att det man skriver på den kommer ut på skärmen förrän det

blir ett uppehåll i kommunikationen eller man explicit säger åt den att skriva ut sig genom att skriva ett radbrytningstecken eller använda metoden ”`flush()`”. ”out” är av typen PrintStream.

- ”err” är felströmmen, den används för att rapportera fel och är precis som ”out” en PrintStream. Till skillnad från ”out” är inte ”err” buffrad, så allt ni skickar till den **ska** komma ut direkt. Använd därför ”err” snarare än ”out” när ni vill ha spårutskrifter.

```
// Nödvändig för att man ska kunna använda
// interfacet DataStream
import java.io.*;

public class IOExempel {

public static void main(String[] args){
  // DataInputStream är kraftfullare än en
  // vanlig InputStream, den kan bland annat direkt
  // läsa in strängar och tal utan att man
  // behöver göra det tecken för tecken
  // DataInput är ett interface som DataInputstream
  // implementerar
  DataInput cin = new DataInputStream(System.in);

  // Detta är buffrat och kommer egentligen inte ut på
  // skärmen innan något speciellt händer. I det här fallet
  // kommer det dock inte att märkas eftersom nästa
  // instruktion är att läsa från terminalen. Då skrivs
  // automatiskt allt ut.
  System.out.print("Skriv in ditt namn:");
  // Låt oss dock vara riktigt j--a säkra på att det
  // kommer ut genom att explicit säga åt System att
  // skriva ut det.
  System.out.flush();

  // Läs in namnet. Alla inläsningar MÅSTE ta hänsyn
  // till att ett IOException kan inträffa
  String namn="";
  try {
      namn=cin.readLine();
  } catch (IOException IOE) {
     // Om något gick fel skicka ett felmedelande
     System.err.println("Något gick fel vid inläsningen");
  }

  // Skriv ut namnet och en hälsning, här behövs ingen
  // flush eftersom det görs automatiskt när man
  // använder println
  System.out.println("Hej "+namn+"!");

  // Vänta på att användaren trycker på return
  try{
    System.in.read();
  }catch(Exception e){System.out.println(e);}
}
}
```

6.1. DataInput

Följande är den intressanta delen av dokumentationen om interfacet DataInput som vi använde ovan, kapat direkt från den officiella Java-dokumentationen. Observera att **alla** dessa metoder måste garderas mot att ett IOException kan inträffa när man använder dem.

- readBoolean()
  Reads a boolean value from the input stream.
- readByte()
  Reads a signed 8-bit value from the input stream.
- readChar()
  Reads a Unicode char value from the input stream.
- readDouble()
  Reads a double value from the input stream.
- readFloat()
  Reads a float value from the input stream.
- readInt()
  Reads an int value from the input stream.
- readLine()
  Reads the next line of text from the input stream.
- readLong()
  Reads a long value from the input stream.
- readShort()
  Reads a 16-bit value from the input stream.
- readUnsignedByte()
  Reads an unsigned 8-bit value from the input stream.
- readUnsignedShort()
  Reads an unsigned 16-bit value from the input stream.

6.2. PrintStream

På samma sätt som ovan är här de intressanta delarna från PrintStream, lite förkortat jämfört med den officiella.

- flush()
  Flush the stream.
- print(anything)
  Print the parameter value.
- println()
  Finish the current line by writing a line separator.
- println(anything)
  Print the parameter value, and then finish the line.

7. Nyttiga klasser

I detta kapitel kommer vi att mycket snabbt gå igenom de viktigaste bitarna av ett antal klasser som det är bra att känna till. Om ni vill ha mer information finns hela API-dokumentationen för Java 1.1 upplagd på Datorsektens hemsida. Den når ni från DSV:s hemsida, via DISK:s. Gå sedan in på Dokumentation och klicka på ”Java Platform Core API”.

7.1. Packages

Javas klasser är uppdelade i något som heter paket (eng *packages*). För att kunna använda en viss klass måste man ibland säga åt kompilatorn vilket av dessa paket som ska användas precis som i IO-exemplet ovan. I genomgången av klasserna nedan kommer detta att visas genom att det direkt under rubriken står ”Paket: paketnamn”. För att det ska fungera måste det överst i filen som använder klassen då stå ”`import paketnamn.*`”.

7.2. StringTokenizer

Paket: java.util

En StringTokenizer är en mycket nyttig klass som används för att dela upp en sträng i delar, tokens. Den har flera konstruktorer:

- StringTokenizer(String)
- StringTokenizer(String, String)
- StringTokenizer(String, String, boolean)

Den första strängen är den som ska delas upp i sina beståndsdelar. Den andra är en lista på de tecken som används för att dela upp strängen. Om den inte är specificerad används space, tab, retur och new-line. Den tredje parametern säger om man vill att skiljetecknen också ska räknas som tokens, om den är false (default) så kastas de bara bort. Dvs om man skapar en StringTokenizer med följande parameter, ”Hej då!” får man två tokens, ”Hej” och ”då!”, medan om man använder parametrarna ”Hej då”, ” !” (obs space före !), och true, så får man fyra, ”Hej”, ” ” (space), ”då”, och ”!”.

Följande metoder finns:

- countTokens()
  Calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception.
- hasMoreElements()
  Returns the same value as the hasMoreTokens method.
- hasMoreTokens()
  Tests if there are more tokens available from this tokenizer's string.
- nextElement()
  Returns the same value as the nextToken method, except that its declared return value is Object rather than String.
- nextToken()
  Returns the next token from this string tokenizer.
- nextToken(String)
  Returns the next token in this string tokenizer's string.

7.3. String

String har vi använd lite överallt ovan, men det finns en del saker som kan vara nyttigt att känna till om den. För det första har Java en del styrtecken som den har ärvt från C. De viktigaste är:

- \n Ger en ny rad
- \t Tab
- \\ Tecknet \
- \’ Tecknet ’

Följande sträng skrivs ut med en radbrytning: ”Nu ska jag\nbyta rad”

Dessutom finns det en del intressanta metoder i klassen String:

- charAt(int)
  Returns the character at the specified index.
- compareTo(String)
  Compares two strings lexicographically.
- concat(String)
  Concatenates the specified string to the end of this string.
- endsWith(String)
  Tests if this string ends with the specified suffix.
- equals(Object)
  Compares this string to the specified object.
- equalsIgnoreCase(String)
  Compares this String to another object.
- indexOf(int)
  Returns the index within this string of the first occurrence of the specified character.
- indexOf(int, int)
  Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
- indexOf(String)
  Returns the index within this string of the first occurrence of the specified substring.
- indexOf(String, int)
  Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
- lastIndexOf(int)
  Returns the index within this string of the last occurrence of the specified character.
- lastIndexOf(int, int)
  Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
- lastIndexOf(String)
  Returns the index within this string of the rightmost occurrence of the specified substring.
- lastIndexOf(String, int)
  Returns the index within this string of the last occurrence of the specified substring.
- length()
  Returns the length of this string.
- regionMatches(boolean, int, String, int, int)
  Tests if two string regions are equal.
- regionMatches(int, String, int, int)
  Tests if two string regions are equal.
- replace(char, char)
  Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

- startsWith(String)
  Tests if this string starts with the specified prefix.
- startsWith(String, int)
  Tests if this string starts with the specified prefix.
- substring(int)
  Returns a new string that is a substring of this string.
- substring(int, int)
  Returns a new string that is a substring of this string.
- toLowerCase()
  Converts this String to lowercase.
- toUpperCase()
  Converts this string to uppercase.
- trim()
  Removes white space from both ends of this string.
- valueOf(boolean)
  Returns the string representation of the boolean argument.
- valueOf(double)
  Returns the string representation of the double argument.

7.4. Integer

Integer är en klass som man kan säga ”omger” en int. Den konstrueras med ”`new Integer(int)`”. Det man framför allt har nytta av är att den kan användas för att konvertera mellan int och String med hjälp av metoderna:

- getInteger(String), en statisk metod som returnerar ett heltal från strängen
- toString(), som tar det värde som finns i objektet och returnerar en sträng med samma värde. Finns även som statisk metod och tar då en int som parameter.

Liknande klasser finns för de andra typerna av tal, de heter: Double, Float och Long. Metoden för att omvandla en sträng till ett tal heter då naturligtvis getDouble etc.

7.5. Math

Math är lite speciellt i det att allt i den är static, alltså måste anropas på formen ”`Math.metodnamn(ev parameter)`”. Den innehåller ett stort antal nyttiga matematiska funktioner till exempel dessa:

- abs(tal)
  Ger absolutvärdet av talet
- max(tal, tal)
  Ger det högsta av talen
- min(tal, tal)
  Ger det minsta av talen
- random()
  Ger ett slumptal mellan 0.0 och 1.0
- round(flyttal)
  Avrundar talet

- sqrt(tal)
  Ger kvadratroten på talet

7.6. Random

Paket: java.util

Även om man kan använda metoden random i klassen Math ovan för att få ett slumptal är det ofta man vill ha lite mer kontroll, då är det lämpligt att använda sig av klassen Random. Den har två konstruktorer, en utan parameter som sätter ”seeden” slumpmässigt, och en där man som användare kan specificera den. En ”seed” är egentligen en siffra som betecknar den följd av slumptal som ska genereras, och ibland kan det speciellt för debugging vara nyttigt att sätta den själv så att man får samma följd av slumptal varje gång man kör programmet.

För att sedan få ett slumptal använder man sig av någon av dessa metoder:

- nextDouble()
- nextFloat()
- nextInt()
- nextLong()