Programmering, minneshjälp för den glömske

Philip Larsson

2013–02–07

1 Programmering - bra att veta

1.1 Inmatning

Inmatning sker med klassen `Scanner`. `Scanner` hör till paketet `java.util` så detta måste importeras.

```
// Read/scan from keyboard
Scanner s = new Scanner(System.in);

// Read next int to nand next string to str
int n = s.nextInt();
String str = s.nextLine();
```

Vi kan koppla Scanner objektet till att läsa in från textfiler istället

```
Scanner sf = new Scanner(new File("in data.txt"));
```

Men, då vi använder anropet `new File` måste vi samtidigt använda Javas felhanteringsmekanism.

```
Scanner sf = null;
try{
        sf = new Scanner(new File("in data.txt"));
} catch (FileNotFoundException e) {
        System.err.println("Could not open file!");
        System.exit(1);
}
```

1.2 Random

För att använda klassen `Random` måste vi importera:

```
import java.util.Random;
```

Vi använder random på följande sätt:

```
// skapar en slumptalsgenerator rng
Random rng = new Random();

// slumpar tal 0...5 + 1
```

```
int dice = rng.nextInt(6) + 1;
```

1.3 Formatera utskrifter med printf

Om man ska skriva ut blandat text och värden kan det vara smidigare att använda printf.

```
int a = 13;
int b = 37;
System.out.printf("The value of a is %d and the value of b is %d", a, b);
```

%d är en styrkod i exemplet ovan. Men det finns flera:

- %[bredd]d - heltal, där bredd är antal positioner som talet ska skrivas i. Utelämnas bredd skrivs talet ut i så många positioner som behövs
- %[bredd].[antal decimaler]f - reella tal (eng floating number)
- %[bredd]s - strängar

```
double pi = Math.PI;
System.out.printf("Pi = %.3f ", pi);

String firstName = "Leonardo";
System.out.printf("First name : %s", firstName);
```

1.4 Foreach loop

```
ArrayList<Person> pList = new ArrayList<Person>();
for (Person p : pList) {
       System.out.println(p.getName());
}
```

Läses som *för varje Person p i pList.*

1.5 compareTo

```
int compareTo(String other)
denna.compareTo(other);
```

Jämför strängen med en annan sträng, returnerar:

- ett negativt heltal om denna sträng kommer före den andra i alfabetisk ordning
- 0 om då båda strängarna är lika
- ett positivt heltal om denna sträng kommer efter den andra i alfabetisk ordning

`other`: den sträng vi jämför med.

1.6 Stringbuilder

Objekt av klassen `String` är oföränderliga - det innebär att det saknas operationer för exempelvis ändring, insättning och borttagning av tecken inuti strängen. Dessa operationer finns istället i klassen StringBuilder, som alltså beskriver en sträng som vi kan ändra i.
Nedan visas ett par metoder i StringBuilder:

```
class StringBuilder:
/* Creates a new, empty StringBuilder */
       String Builder()

/*Creates a new StringBuilder with the same content as a given String.
s: the string this StringBuilder shall include from the beginning. * /
       StringBuilder(String s)

/ * Extends the string with a string.
str: the string to be added * /
       StringBuilder append(String str)

/ * Removes the characters inside the string.
start: index of the first character to be deleted,
end: index of the character after the last character to be deleted. * /
       StringBuilder delete(int start, int end)
```

Exempel:

```
String firstName = "Bosse";
StringBuilder sb = new StringBuilder(firstName);
sb.append(" Bus");
String name = sb.toString();
```

1.7 StringTokenizer

StringTokenizer används för att plocka ut delar av en sträng. För att använda StringTokenizer måste vi importera `java.util`.

```
class StringTokenizer:

/ * Creates a StringTokenizer that pick out the parts of a given string.
* The parts in the string is assumed separated by spaces and tabs.
* s: the string that we must dismantle. * /
       StringTokenizer(String s)

/ * Creates a StringTokenizer that pick out the parts of a given string.
* s: the string that we will dismantle,
* delimiters: the character that separates the string parts. * /
       StringTokenizer(String s, String delimiters)

/ * Counts how many parts are left in the current string. * /
       int countTokens()
```

```
/ * Determine if there are more elements in the current string. * /
       boolean hasMoreTokens()

 /* Returns the next part of the current string, provides execution error if there are
     no more parts in the string. * /
       String nextToken()
```

Exempel:

```
/*Plockar ut orden ur en String */
String s = "I'm gonna pop som tags";
StringTokenizer st = new StringTokenizer(s);
System.out.println("There is " + st.countTokens() + " words at this line.");
while (st.hasMoreTokens()) {
       System.out.println(st.nextToken());
}
```

1.8 Skyddsnivåer

- `public`: Attribut och metoder som vi skriver `public` framför kan anropas och användas varifrån som helst.
- `Ingen markering`: Attribut och metoder som vi inte skriver något framför kan bara anropas och användas av andra klasser i samma katalog.
- `private`: Attribut och metoder som vi skriver `private` framför kan bara anropas och användas inifrån den egna klassen.
- `protected`: Klassen själv, andra klasser i samma paket, och this i subklasser har tillgång till dem.

1.9 Vektorer

En vektor är ett slags objekt med ett bestämt antal värde, där varje värde har ett givet index. En vektor har alltid en bestämd storlek, och denna anges när vi skapar den.

```
String[] names;
names = new String[5];
```

Listor är i allmänhet betydligt enklare att arbeta med, men vi har dock själva större kontroll på vad som händer i datorn när vi använder vektorer.
Om vi i förväg vet vilka värden som skall ligga i en vektor så kan vi ge den sitt startvärde direkt.

```
String[] workDays = {"mon", "tue", "wed", "thu", "fri"};
```

1.10 Vektorer med flera index - matriser

Vi kan ha vektorer med mer än ett index, en matris!

```
double[][] value = new double[3][3];
```

1.11 Överskuggning och Överlagring

- Överskuggning innebär att vi ersätter en gammal (ärvd) operation med en ny som har samma signatur - vi får alltså ingen ny operation.
- Överlagring innebär att vi lägger till en ny operation som har samma namn som en ärvd operation, men annan signatur. (t.ex olika antal parametrar eller olika typer på sina parametrar)

Vi kan även använda den överskuggade operationen när vi överskuggar med hjälp av `super`.

```
public void forward(int n){
        super.forward(n);
        count += n;
}
```

Vi skriver `super.forward(n)` för att få kompilatorn att förstå att det är operationen `forward(int)` i superklassen som skall anropas. Om vi hade skrivit `forward(int)` hade vi anropat oss själva - sådana anrop kallas rekursiva och är väldigt användbara i många sammanhang, men här hade det bara inneburit at operationen hade anropat sig själv tills minnet tar slut (och vi får `java.lang.StackOverflowError`).

1.12 Abstrakta klasser

En klass kan ärva från en abstrakt klass precis på samma sätt som man ärver från andra klasser, men en abstrakt klass skiljer sig lite:

- Man kan inte skapa ett objekt av den abstrakta klassen, om någon försöker så får denna kompileringsfel.
- En abstrakt klass kan ha abstrakta metoder[1]

Observera att bara kan ärva (*extends*) ett objekt [2].
Det finns flera poänger med abstrakta klasser:

- Vi behöver inte längre skriva någon kod i de operationer som vi ändå inte kan implementera på något vettigt sätt.
- Den som ärver klassen upptäcker om någon av de abstrakta operationerna har glömts bort eller deklarerats felaktigt - även detta skulle ge upphov till kompileringsfel.

```
abstract class Bank {
        // Variabler
        private double cash;
```

[1] En abstrakt metod är en metod utan implementation, och de måste implementeras av subklasser.
[2] Medans man kan implementera flera interface

```
        // Abstrakta metoder
        public abstract void setup();
        public abstract int getNbrOfAccounts();
        public abstract string getBankName();

        // Vanliga metoder
        public double getCash() {
                return cash;
        }
}
```

1.13 Exceptions

Det finns två slag av Exceptions:

- Unchecked Exceptions Ð används då felet beror på programmeraren
  Ex: NullPointerException eller ArrayIndexOutOfBoundsException
- Checked Exceptions Ð används då felet inte beror på programmeraren
  Ex: FileNotFoundException om man försöker öppna en fil som inte finns

När man anropar en metod som genererar en checked exception måste man ta hand om det. Normalt fångar man det i en try-catch-sats:

```
Scanner scan = null;
try {
   // open file with file name fileName
   scan = new Scanner(new File(fileName));
} catch (FileNotFoundException e) {
   System.err.println("Couldn't open file " + fileName);
   System.exit(1);
}
 ... use scan ...
```

Om exception inträffar, avbryts exekveringen av satserna i try-blocket och satserna i catch-blocket exekveras.

I satsen `catch(Exception e)` kan t.ex. följande metoder användas för att få mer information:

- `e.printStackTrace();` som skriver ut information om raden där felet inträffat och den/de metodanrop som lett till denna rad.
- `e.getMessage();` som returnerar en sträng med meddelande om felets art.

Generera egna exceptions:

```
throw new IllegalArgumentException("amount to deposit < 0");
```

Mönstret är:

```
  throw new ExceptionClass();
  throw new ExceptionClass(message);
```

Detta gör att exception-objekt skapas, exekvering av metoden avbryts. Javasystemet letar efter fångade catch-block.

Exempel:

```
/**
 * Deposits the specified amount.
 * pre: The specified amount is >= 0
 * post: The specified amount is added to balance
 * @param n The amount to deposit
 * @throws IllegalArgumentException if the specified amount is < 0
 */
public void deposit(int n) {
   if (n < 0) {
      throw new IllegalArgumentException("amount to deposit < 0");
   }
   balance = balance + n;
}
```

2 Sökning

2.1 Linjärsökning

```
int pos;
for (pos = 0; pos < n; pos++) {
       if (v[pos] == value) {
              break;
       }
}
```

Denna sökmetod kallas linjärsökning eftersom vi söker stegvis genom vektorn, element för element. I värsta fall, dvs då det sökta elementet antingen inte finns, eller finns sist i vektorn, måste vi söka igenom hela vektorn. Skulle det sökta värdet finnas behöver vi göra i genomsnitt n/2 jämförelser. Söktiden är linjär, eller $O(n)$.

2.2 Binärsökning

Om den vektor vi söker i är sorterad kan vi skriva betydligt effektivare algoritmer än den ovan. Ett exempel är att ”gissa ett tal” och sedan gaffla sig fram. Vi börjar i mitten och delar successivt in återstående alternativ i så lika delar som möjligt. Vi inför två variabler, `min` och `max` för att hålla reda på inom vilka indexgränser det sökta värdet kan finnas. Därefter bestämmer vi mittpunkten i intervaller, `mid`, och undersöker det värdet som ligger där. Vi får tre fall:

- Värdet i mittpunkten är större än det största värdet. Vi kan då konstatera att det sökta värdet måste finnas mellan `min` och `min` -1, så vi ändrar `max` till `mid` -1 och gör ett nytt försök.
- Värdet i mittpunkten är mindre än det sökta värdet. Vi vet då att värdet måste finnas mellan `mid` +1 och `max`, så vi ändrar `min` till `mid` +1 och gör ett nytt försök.
- Vi hittar det sökta värdet, vilket gör att vi kan avbryta sökningen och konstatera att värdet finns i index `mid`.

Exempel:

```
public static int binarySearch(int[] a, int n, int value){
       int min = 0;
       int max = n-1;
       while (min <= max) {
              int mid = (min+max)/2;
              if (a[mid] > value){
                     max = mid-1;
              } else if (a[mid] < value) {
                     min = mid + 1;
              } else {
                     return mid;
              }
       }
return -1;
}
```

Om vi i varje steg eliminerar hälften av de återstående elementen och från början har $n \approx 2^k$ element i vektorn behövs k varv för att vi bara skall ha en möjlig kandidat kvar. Det största antalet varv i loppen blir därför proportionellt mot $log_2 n$ (man brukar skriva $O(logn)$), en avsevärd förbättring jämfört med linjärsökning.
Skillnaden är inte så stor vid sökningar i mindre vektorer, men om vi skulle söka efter en person i en vektor med hela svenska befolkningen skulle linjärsökning kräva i snitt 4.500.000 jämförelser, medan det med binärsökning skulle räcka med mindre än 25 jämförelser.

3 Sortering

I Java finns det ett smidigt sätt att sortera vektorer:

```
Arrays.sort(v);
```

om vi skall sortera hela heltalsvektorn v.

3.1 Urvalssortering

Den kanske enklaste metoden att sortera är att börja med att leta upp det minsta värdet i vektorn och sätta in det först. Därefter letar vi upp det näst minsta elementet (som är det minsta av de återstående elementen) och sätter in det näst först, och så fortsätter vi att leta upp och sätta in det tredje minsta, fjärde minsta, etc, tills alla element är i rätt ordning.

```
public static void selectionSort(int[] a, int n) {
	for (int i = 0; i < n; i++) {
		int smallest = i;
		for (int k = i+1; k < n; k++) {
			if (a[k] < a[smallest]) {
				smallest = k;
			}
		}
		int tmp = a[i];
		a[i] = a[smallest];
		a[smallest] = tmp;
	}
}
```

För att få ett mått på hur effektiv denna metod är kan vi räkna antalet jämförelser som behövs för att sortera en vektor med n tal. I första varvet i den yttre for-loopen gör vi $n-1$ jämförelser (de görs inuti den inre for-loopen), i det andra varvet i den yttre for-loopen gör vi $n-2$ jämförelser, etc. Totalt får vi:

$$T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

jämförelser. dvs om vi dubblar antalet element i vektorn så krävs det ungefär fyra gånger fler jämförelser för att sortera elementen.

3.2 Bubbelsortering

En annan sorteringsmetod är bubbelsortering. Den är inte särskilt effektiv men den är ganska lätt att förstå och tillhör allmänbildningen.
Metoden bygger på att vi upprepade gånger går igenom vektorn bakifrån och låter tal som är mindre än sina föregångare byta plats med dessa. När vi i något var inte längre behöver byta plats på två element är vi färdiga. I varje varv kan vi gå ett steg kortare eftersom de minsta talen efterhand samlas i vektorns början.

4 Lista, stack och kö

4.1 Listor

En lista är en ordnad följd av element.

- Det finns en före-efter-relation mellan elementen
- Begrepp som första element i listan", "efterföljaren till visst element i listan"är meningsfulla. Det finns alltså ett positionsbegrepp.

Vi kan skapa en Lista med interfacet list:

```
List<String> names;
```

och på så vis vänta med att välja vilken typ av lista vi vill ha. Två exempel på olika klasser som implementerar List.

- `ArrayList`, som är särskilt bra på att snabbt hämta värden med get-operationen. Insättningar bör man bara göra sist i listan, då insättningar i andra positioner orsaker flyttningar av element.
- `LinkedList`, som är särskilt bra på att sätta in och ta bort element i början av listan.

Vi skapar en lista såhär:

```
List<String> names = new ArrayList<String>();
```

Observera:

```
// works but considered ugly:
ArrayList<String> names = new ArrayList<String>();

// better! :
List<String> names = new ArrayList<String>();
```

4.2 Kö

En kö kallas även FIFO-lista. First In First Out.
Insättningar göras alltid sist i följden, och bortagning avser alltid första elementet.
Analogi: En (vanlig) kö när man handlar mat på ICA...

4.3 Stack

En stack kallas även LIFO-lista. Last In First Out.
Operationer sker på toppen av stacken, dvs. nya element läggs till längst upp, och element tas bort längst upp.
Analogi: En tallriks dispenser, en hög med papper.

5 Bra att veta

5.1 Klassen Object

Det finns en klass `Object`, som är ”alla klassers moder”, om vi inte uttryckligen ärver någon annan klass så ärver vi automatiskt klassen `Object`.
Klassen `Object` innehåller inte särskilt mycket av intresse, men följande kan vara bra att känna till:

- toString

  `public String toString()`

  ger en sträng som beskriver objektet. Denna operation anropas automatiskt om vi skulle försöka skriva ut ett objekt, och implementationen i klassen `Object` skriver ut klassnamnet och ett heltal (objektets hash-kod)

- Metoden `equals` används för att jämföra om två objekt är lika.

  ```
  public boolean equals(Object obj);
  ```

  Metoden returnerar true om och endast om de jämföra objekten är identiska. Om man istället vill att innehållet inuti objekten ska jämföras måste man skugga `equals`.

  ```
  public boolean equals(Object obj){
         if (obj instanceof Person){
                return idNbr == ((Person) obj).idNbr;
         }else{
                return false;
         }
  }
  ```

  - Observera att parametern till `equals` måste vara av typen `Object`, annars blir det inte skuggning. Därför måste också typomvandlig till Person ske när man ska använda obj:s idNbr.
  - Uttrycket `obj instanceof Person` returnerar true om obj:s typ är av Person eller någon subklass till Person.
  - Uttrycket `obj instanceof Person` returnerar false om obj har värdet null.

- Förutom operationerna ovan innehåller även `Object` en operation

  ```
  int hasCode()
  ```

  som ger en så kallad hash-kod, dvs ett slags numeriskt fingeravtryck för ett objekt. Detta innebär att alla objekt har en hashkod.

5.2 Interfaces

Ett interface(svenska: gränssnitt) är som ett skal med metoder som den som implementerar måste använda. En klass kan implementera flera interface.

```
public interface InterfaceName{
       //here is public abstract methods
}
```

```
public class ClassName implements InterfaceName{
       // attributs, methods, constructor as usual, but all methods in InterfaceName
           must be included
}
```

Om det skiljer mycket mellan metoderna i klasserna så använd interface istället för en superklass.

5.3 Kodningskonventioner

Enligt Javas kodningskonvention skall man skriva klasser med stor begynnelsebokstav, ex namnet `Account` istället för `account`.

Av historiska skäl väljer man i allmänhet att skriva konstanter med stora bokstäver.

```
public final int GOLDEN_NUMBER = 42;
```

5.4 iterator

```
ArrayList a1 = new ArrayList();
Iterator itr = a1.iterator();
```

5.5 Principer

Klasser och objekt gör det möjligt för oss att skriva våra program så att de blir lättare att förstå än om vi bara använder ett huvudprogram och underprogram - detta är särskilt viktigt i större programsystem. att utnyttja klasser och objekt på rätt sätt är dock långtifrån trivialt, och mycket har skrivits om hur man gör.

- Cohesion, som talar om hur väl en klass eller ett paket (ofta använder man namnet 'modul') hänger ihop - en klass eller ett paket som gör precis en sak har hög kohesion, medan en klass eller ett paket som har flera olika relaterade uppgifter har låg cohesion.
  Våra program blir betydligt enklare att förstå, modifiera och återanvända om de ingående klasserna och paketen har hög kohesion, dvs om de gör en eller några få saker vardera.

- Coupling, som anger hur en klass beror av andra klasser. I ett system med låg coupling kommunicerar objekt med hjälp av enkla protokoll (specifikationer), de varken kan eller behöver känna till det inre tillståndet hos varandra.

  Ett system med låg coupling gör det möjligt för oss att ändra i en klass utan att andra klasser påverkas i någon större utsträckning, i ett system med hög coupling kan däremot en ändring i en klass ge upphov till en lång kedja av ändringar i andra klasser. Andra problem med låg coupling är att det är svårt att förstå vare enskild klass utan att förstå alla de klasser som den interagerar med, och dessutom att det är svårt att testa individuella klasser.

När vi skriver lite större program bör vi sträva efter att ha hög kohesion och låg coupling - ofta får vi automatiskt låg coupling om vi har hög kohesion. Att program ibland blir lite effektivare om vi ökar graden av coupling motiverar sällan den ökade komplexiteten som det ger upphov till.
Det finns ett antal principer som kan hjälpa oss att ge våra program bättre struktur:

- Separation-of-concerns: När vi designar våra system bör vi försöka dela ner dem i mindre delar (klasser), där de olika delarna har så lite gemensam funktionalitet som möjligt - detta är ett sätt att öka de ingående delarnas cohesion.

- Single-responsibility: En klass skall helst bars ha en enda, väldefinierad uppgift, och alla dess operationer bör ha med denna uppgift att göra. För varje ny uppgift en klass får ökar risken att klassen behöver ändras - genom att hålla antalet ansvarsområden för en klass nere ökar vi dess cohesion.

- Information-hiding: Om vi gör det omöjligt att se det inre av våra klasser (genom att vi `private`-deklarerar attribut och lämpliga operationer) så riskerar ingen annan att koppla sin klass för hårt till det inre tillståndet i vår klass - vi tvingar alltså på användaren lägre coupling.

- Open-closed principle: En klass bör vara öppen för utvidgningar, men stängd för ändringar. Detta innebär bland annat att vi inte bör lägga till nya operationer i en klass som redan används, utan att vi stället utvidgar den till nya subklasser. Ofta designar vi våra grundläggande klasser för att våra så allmänna som möjligt (gärna abstrakta), och använder överskuggning för att få objekt av subklasser att bete sig annorlunda än objekt av den ursprungliga klassen.