INF1010 - Eksamensnotater

Eivind, Sjur, Annabelle & Martin

13 June 2010

Innhold

1 Tall

Type	Lovlige verdier
byte	-+127
short	-+32 767
int	-+2 147 483 647
long	-+9 223 372 036 854 775 807L
float	-+3.402 823 47 E+38F
double	-+1.797 693 134 862 315 70 E+308

negative tall er har alltid èn større range pga 0.

1.1 Integere

Integere er heltall, og lite annet. Hvis vi vil konvertere en streng til en int sier vi:

```
int x = Integer.parseInt("123");
```

Hvis vi vil konvertere(caste) et annet tall til int skriver vi:

```
int x = (int) 3.14
```

Operasjon	Beskrivelse	Eksempel
+, -, *, /	De fire regneartene	1+1
++	legge til 1	i++
–	trekke fra 1	i–
%	modulo(rest)	5%2 -> 1
Math.sqrt(...)	Kvadratrot	Math.sqrt(4) -> 2
Math.pow(x, y)	Potens (x^y)	Math.pow(2, 3) -> 8

- Metodekall
  - ++ og –
  - * og /
  - + og -
  - -(som negativt fortegn)

husk at:

```
int x, y=1;
x = y++ + y + ++y;
```

nå er x = 6, fordi stykket blir 1 + 2 + 3. først er y = 1, og så øker den slik at neste gang y brukes er den 2. når vi skriver ++y så vil y være 3 før vi legger det til.

1.2 desimaltall

Jeg velger å bare leke med floats, men reglene er ca det samme som for int.

konvertering fra heltall til desimaltall:

```
float x = (3 + 0.0) / 2;
```

altså vi bare legger til 0.0 som det som skjer først, slik at ett av tallene er et desimaltall, ellers vil vi få heltallsdivisjon. Enkelt og greit, utenom det gjelder de samme operatorene og de samme presidensreglene.

2 Sannhetsverdier (boolean)

- &&
  - Og
  - true && true
  - ||
    - Eller
    - true || false

- ikke b = !false
- < og >
  - mindre enn, større enn
  - b = x<y
- <= og >=
  - mindre eller lik
  - b = x<=y
- ==
  - er lik
  - b = x==y
- !=
  - er ikke lik
  - b = x!=y

2.1 presidensregler

- Metodekall
  - !
  - <, <=, >=, >
  - =, !
  - &&
  - ||

3 Tekst - Strenger og char

vi har strengen s = "kake"

Navn	Forklaring	Eksempel
charAt(...)	tegnet i gitt posisjon(fra 0)	s.charAt(2)=='k'
length()	gir lengden på teksten	s.length()==4
substring(...)	delteksten fra- og tilposisjon gir indeksen og ut	s.substring(1,3)=="ak" s.substring(1)=="ake"
equals(...)	tester likhet mellom strenger (boolean)	s.equals("kake")
indexOf(...)	posisjonen til tegnet/tekst	s.indexOf('a')==1
startsWith(...)	starter teksten med ... (bool)	s.startsWith("ka")
endsWith(...)	ender teksten med ... (bool)	s.endsWith("ke")
compareTo(...)	sammenligning av tekster	s.compareTo("bake")<0
toCharArray()	gjør om strengen til et array av chars	s.toCharArray()

3.1 char

en char-verdi er rett og slett en bokstav, den kan sammenlignes ('a' < 'b') og vil da sammenlignes ut i fra ascii-verdier (alle store bokstaver er mindre enn de små bokstavene). og kan dermed i mange sammenhenger tenkes på som tall.

3.2 String

En string er en rekke med char-verdier, altså ord. Man kan legge ord sammen med pluss-opperatoren ("heisann" + " " + navn), man kan konvertere tall til strenger på denne måten

```
String s = "" + 42;
```

og verdien til s vil være "42".

Strenger kan også deles opp i arrayer ved hjelp av en split-funksjon. eks:

```
string[ ] t = s.split(" ");
```

3.3 Innlesing av tekst

for å lese fra terminal er det lettest å bruke en klasse som finnes i og java.util.*;

Vi bruker også System.in

```
import java.util.*;
Scanner tast = new Scanner(System.in);

String s = tast.nextLine();
```

Fra fil så trenger vi også noen klasser i java.io.* 'programmet' under leser fra en fil og over i terminalen linje for linje.

```
import java.util.*;
import java.io.*;

Scanner f = new Scanner(new File(filnavn));

while (f.hasNextLine()){
  String inn = f.nextLine();
  System.out.println(inn);
}
```

4 Arrayer

Arrayer er en indeksert(fra 0) gruppe av objekter. Man må definere størelsen når man lager objektet.

```
String[] a = new string[3];
```

man kan nå finne lengden på arrayet og bruke det som en int a.length;

Man kan lage arrayer av alle mulige objekter.

4.1 array-metoder

4.1.1 rask sortering

```
import java.util.*;
char[] sorteres = "noesomskalsorteres".toCharArray();
Arrays.sort(sorteres);
for(int i = 0;i<sorteres.length;i++){
    System.out.print(sorteres[i]+" ");
}
```

Arrays er en klasse i java.util som kan sortere for oss "raskt og gæli" bruker antagelig quicksort under panseret og funker med de fleste tall og chars.

5 Løkker og kontrollstrukturer

Navn	**Beskrivelse**	**Eksempel**
for	bestemt antall ganger	for(int i=0;i<3;i++){}
	alle objekter i array	for(String s : a){}
	alle objekter i hash	for(String s : hm.values())
while	i mens test er sann	while(b){}
do-while	utfører løkka før testen	do {} while(b);
switch	Hopper mellom kodeblokker ved input	switch(c){case a: <>; break; default: <>;}

5.1 while

while-løkker er kanskje den enkleste formen for løkker, den gjør en blokk kode så lenge predikatet i parameteret er sant.

Man trenger ikke nødvendigvis å kjøre en kodeblokk, for eksempel hvis man har en boolsk funksjon ("kan()") som parameteret kan man kjøre koden "while(kan());" og dermed si at man skal kjøre helt til "kan()" returnerer false.

5.2 for

for-løkker er en naturlig utvidelse av while da man ofte trenger tellere eller ting som skjer for hver gang kodeblokken skal kjøres, for eksempel en teller når man går igjennom et array.

```
for(int i=0;i<array.length;i++;){
  System.out.println(array[i])
}
```

Men for-løkker kan brukes på andre måter da den har en ganske enkel måte å oppføre seg på "for(initialiseringskode;predikat;postkode)", eneste som det er strengt hva man må ha i en for-løkke er predikatet. initialiseringen, kan være å sette en teller til noe, eller når man går igjennom noder i en graf, så kan man f.eks sette src_{java}{for(Node n = root;n.next!=null;n=n.next)}

5.3 while .. do

Mindre brukt løkketype, egentlig en while-løkke hvor koden kjøres minst en gang før predikatet blir testet.

5.4 switch

Ved bruk av if-setning kan man velge en av to muligheter ved at setningen som evalueres er *true* eller *false*. Med *switch* kan man velge mellom mange inputpåstander som er *integers*, *chars* eller *enums*.

```
switch (<utrrykk>) {
    case verdi1: <settninger>; break;
    etc..
    default: <kjøres ved ingen treff ved <uttrykk> >;
}
```

6 Hashmaps

Hashmaps er en enkel måte å ordne mange objekter med et objekt som indeks.
import java.util.*

HashMap<string,Person> personregister = new HashMap<String,Person>();

Metode	beskrivelse
put(nøkkel, peker)	legge til objekt i HM
get(nøkkel)	hente peker til objekt
remove(nøkkel)	fjerne nøkkel fra HM
containsKey(nøkkel)	bool om nøkkelen er der
containsValue(objekt)	bool om objektet er der
values()	lager en mengde av alle verdiene i HM, brukes til itterering
keySet()	brukes til å lage en mengde av alle nøklene brukes til iterering
isEmpty()	returnerer true hvis tabellen er tom.
size()	Metoden returnerer antall nøkler i tabellen

Hashmaps import java.util.*;

HashMap hashmapnavn = new HashMap(); hashmapnavn.put(nøkkel, verdi);

Hente: Bil b = (Bil) register.get(nøkkel);

Hente alle verdiene: Iterator it = register.values().iterator(); while (it.hasNext()) { Bil b = (Bil) it.next(); }

containsKey(nøkkel) sjekker om objekt med nøkkelen finnes. Gir true/false. containsValue(verdi) sjekker om hashmap inneholder gitte objekt/verdi. size() antall nøkler.

7 UML

Diagrammer av programmene vårt. Tegner selve strukturen til programmet.

-Objekdiagrammer. (UML-bokser med ev. variabler og argumenter etc.) -Klassediagrammer. (bare koblingene etc.)

Tegner selve arkitekturen til programmet vårt.

7.1 Navneklassen

+public -private ˜package

Vi lager en modell av problemområdet vårt (også kalt domenemodell). -Modell av probleme (og modell av databasen).

-Tegner streker mellom klassene. Og navn på forbindelsen.

Vi tegner hvor mange objekter det maksimalt kan være.

Vi har forskjellige skrivemåter for verdier: 1 - en * - null, en eller flere 1..* - minst en 3,4,5 - tre fire eller fem

Tegner koblinger mellom de klassene du tenker programmet skal snakke med.

Vi angir kun de mest sentrale dataene i UML-boksene.. F.eks. viktige variabler og arrayer. F.eks. :klasenavn [atributt-fekt (kan være tomt)].

-Pekere - piler

Tegner flere ved å legge en UML-boks som "skygge" bak.. En representasjon for N antall.

8 Lenkede lister

8.1 Hva er en lenket liste?

En lenket liste er en liste med objekter refererer til neste, eller forrige objekt ved hjelp av pekere.

-Lenket liste. Trenger ikke ta stilling til antall fra start. -Alle objekter er lenket til hverandre, eget objekt som peker på -første objektet. -Må søke i listen forfra for å finne objekter.

```
/* En første skisse av personlista. Først klassen som
 * beskriver objektene (personene) som skal lenkes sammen.
 * Deretter en klasse som beskriver selve lista. */

class Person {
    String navn;
    Person nestePerson ;
    // andre attributter
}

class Personer {
    Person personliste ;
    // . . .
}
// Lage en ny liste med personer:
Personer mineVenner = new Personer ;
```

Hvilke operasjoner trenger vi?

- Sette inn ny person
- Finne en person
- fjerne en person

...

```
class Personer {
    Person personliste ;
```

```
    void setteInnPerson ( Person inn ) { }
    void finnePerson ( Person p ) { } // Finne person og taUtPerson
    void taUtPerson ( Person ut ) { } // Kan også slås sammen om hvert
                                      // objekt bare skal brukes èn gang
    // . . .
}
```

8.2 Innsettingsmetoder

8.2.1 LIFO

Innsetting først (LIFO):

```
void settInnPersonForst ( Person inn ) {
    // Hvis lista er tom, sett inn objektet
    if(personliste == null) personliste = inn ;
    else {
        // minst et objekt i lista
        inn.nestePerson = personliste ;
        personliste = inn;
    }
}
```

8.2.2 FIFO

Innsetting sist (FIFO):

CANNOT INCLUDE FILE code/lenkeeksempel4.java

8.2.3 Innsetting etter et bestemt objekt

```
/** Setter personobjektet i inn etter personobjektet e
* Hverken i eller e er NULL.
* @param e Personobjektet som skal ha i som neste
* @param i Personobjektet som skal inn etter i
*/

void settInnPersonEtter ( Person e , Person i ) {
    i.nestePerson = e.nestePerson ;
    e.nestePerson = i ;
}
```

8.3 Skrive ut en lenket liste

```
// I klassen Personer :
void skrivAlle() {
    Person p = personliste ;
    while (p != null ) {
        p.skrivUt() ;
        p = p. nestePerson ;
    }
}
// I klassen Person :
void skriv ( ) {
System. out . println ( . . . ) ;
}
```

9 Rekursjon

Kaller seg selv. Midlertidig data lagres i en såkalt Call Stack. Når metode kommer til return statement går man "baklengs" tilbake i call stacken og metoden fortsetter etter der den kalte seg selv.

Klassisk eksempel på en rekursjonsmetode er dette fakultetseksempelet.

```
class Factorial {
     int fact(int n) {
         if ( n ==1) return 1;
         return fact (n-1) * n;
     }
}

class Recursion {
     public static void main (String args[]) {
          Factorial f =new Factorial();
          System.out.println(?Factorial of 3 is ? + f.fact(3));
          System.out.println(?Factorial of 4 is ? + f.fact(4));
          System.out.println(?Factorial of 5 is ? + f.fact(5));
     }
}
```

Dersom fact() blir kalt med tallet 1, vil den returnere 1, ellers vil den returnere produktet av fact(n-1)*n. Denne metoden rekurserer inntil n = 1. Når en metode kaller seg selv, vil nye lokale variable og parametere lagres i stack, og koden utføres med disse nye variablene og parameterene fra starten aav. Den lagerr ikke kopi av metoden, kun argumenter/variabler.

9.1 Hanois tårn

10 Binære trær

10.1 Sett inn i binærtre

Under følger et eksempel på en sett-inn-metode for binære trær. Metoden ligger i klassen *BinaryTreeNode*, og vi antar at vi har en metode som sammenligner verdiene til nodene, kalt *compareNodes*.

```
void addNodeToBinaryTree(Node node) {
    if(compareNodes(node) <= 0) {
        if(right == null) {
            right = node;
        } else {
            right.addNodeToBinaryTree(node);
        }
    } else {
        if(right == null) {
            left = node;
        } else {
            left.addNodeToBinaryTree(node);
        }
    }
}
```

10.2 Traversering av treet

10.2.1 Preorder

10.2.2 Inorder

Hvis vi ønsker å traverse binærtreet i synkende rekkefølge bruker vi *inorder traversal*.

```
void printNodes() {
   if(left != null) {
       left.printNodes();
   }
   System.out.print(node.value);
   if (right != null) {
       right.printNodes();
   }
}
```

10.2.3 Postorder

11 Søk

hvis vi har sorterte data vil alltid binærsøk være den raskeste måten å søke på. Binærsøk så halverer vi antall muligheter for hver et eksempel fra den virkelige verden er hvis man skal gjette et tall mellom 1 og 1024 og man har 10 gjetteforsøk, hvor man får vite om man gjettet for stort eller for lite, vil man alltid kunne finne tallet ved at man hele tiden halverer.

eks: vi skal finne tallet (769)

mulige tall	gjett	større / mindre	forsøk
1024	512	større	1
512	768	større	2
256	896	mindre	3
128	832	mindre	4
64	800	mindre	5
32	784	mindre	6
16	776	mindre	7
8	772	mindre	8
4	770	mindre	9
2	769	riktig!	10

ved usorterte mengder er det vanskeligere å gjøre strukturerte søk og det kan ofte være lurt å enten sortere det først, eller dele opp mengden i delmengder og søke hver del for seg selv.

11.1 Eksempel på søking i usortert mengde med tråder

```
public class FinnMinst {
    final int maxVerdiInt = Integer.MAX_VALUE;
    int [ ] tabell; // mengden vi skal søke igjennom
    Oversikt oversikt; // her trådene raporterer

    public static void main(String[ ] args) {
        new FinnMinst();
    }

    public FinnMinst(){
        // vi lager mengden vi skal søke igjennom med random ints
        tabell = new int[64000];
        for (int in = 0; in< 64000; in++)
            tabell[in] = (int)Math.round(Math.random()* maxVerdiInt);

        //vi lager et overvåkningsobjekt som mottar info fra trådene
        oversikt = new Oversikt();
```

```
        for (int i = 0; i< 64; i++)

            // Lager 64 tråder, og hvilken del av arrayet de skal søke på
            new Soketraad(tabell,i*1000,((i+1)*1000)-1,oversikt).start();

        oversikt.vent(); // er ikke ferdig før alt er søkt igjennom
        System.out.println("Minste verdi var: " + oversikt.minste);
    }
}

class Oversikt {
    int minste = Integer.MAX_VALUE, antallFerdigeTrader = 0;

    synchronized void vent() {
        while (antallFerdigeTrader != 64) {
            try {wait();} catch (InterruptedException e){}
        }
    }

    //mottar tall fra en tråd og ser om den er mindre enn den minste
    synchronized void giMinste (int minVerdi) {
        antallFerdigeTrader++;
        if(minste > minVerdi) minste = minVerdi;
        notify();
    }
}

class Soketraad extends Thread {
    int [ ] tab; int start, end; Oversikt ov;

    Soketraad(int [ ] tb, int st, int en, Oversikt o) {
        tab = tb; start = st; end = en; ov = o;
    }

    public void run(){
        int minVerdi = Integer.MAX_VALUE;
        for (int i = start; ind <= end; i++)
            if(tab[i] < minVerdi) minVerdi = tab[i];
        ov.giMinste(minVerdi);
    }
}
```

12 Sortering

Det er ofte en fordel å sortere store mengder data for å gjøre søking i i data effektivt om i det hele tatt mulig.

Vi har gjennomgått flere forskjellige måter å sortere data på jeg vil her fokusere på følgende

- innstikksortering
- quicksort

12.1 Innstikksortering

veldig simpel sorteringsalgoritme som er effektiv på små mengder (i forhold til andre sorteringsalgoritmer), men taper mye når det er større mengder data som skal sorteres baserer seg på at man starter først i mengden og sorterer mens man itererer over mengden

```
class InSort{
    public static void main(String[] args){
        // mengden som skal sorteres
        int[] mengde = {4, 2, 6, 3, 7, 1, 5};
        for(int i=0;i<mengde.length;i++){
            // hvor vi er og hva vi skal stikke inn
            int j=i, temp=mengde[i];
            // går igjennom alt som er sortert frem til punktet der
            // vi enten er på starten av lista, eller at det neste
            // sorterte elementet er mindre enn temp
            while(j>0 && mengde[j-1] > temp){
                // Vi dytter lista fremover for å gjøre plass til innstikk
                mengde[j]=mengde[j-1];
                j--;
            }
            // når vi har funnet riktig sted setter vi inn
            mengde[j] = temp;
        }
    }
}
```

12.2 Quicksort

Mer populær for mellomstore til store datamengder, den tar et objekt fra mengden og forterer listen etter hva som er større og mindre enn det objektet, deretter begynner den å sortere de to delene på samme måte rekursivt (dermed lettere å implementere tråder)

```
public class QSort {
    static int antallBytt = 0;
```

```
    public static void main(String[] args) {
        char [] tegn = "åøæzywvutsrqponmlkjihgfedcba".toCharArray();
        quicksort(tegn,0,tegn.length-1);
        for(int i=0;i<tegn.length;i++){System.out.print(tegn[i]+" "); }
    }
    static void quicksort(char[] a, int fra, int til) {
        //man kan ikke sortere bare en ting
        if (!(til-fra < 1)){
            //vi velger et tall som vi skal
            char pivot = a[til];
            boolean ikkeFerdig = true;
            int forst = fra;     // der vi midlertidig pivoterer rundt,
            int sist = til - 1;  // snevrer oss mot et punkt
            while (ikkeFerdig) {
                while (a[forst] < pivot && forst<a.length-1) { forst++ ;}// vi finner stedet
                while (a[sist] > pivot && sist>0) { sist--;}// der vi vil bytte
                if (forst < sist) { // hvis vi ikke har funnet midten
                    bytt(a, forst, sist);
                    forst++; sist--;
                } else ikkeFerdig = false;
            }
            bytt(a, forst, til);
            quicksort(a, fra, forst-1);
            quicksort(a, forst+1, til);
        }
    }
    static void bytt (char[] a, int fra, int til) {
        char temp = a[til];
        a[til] = a[fra];
        a[fra] = temp;
    }
}
```

12.3 Sortering med tråder

Raskt eksempel med Quicksort som bruker tråder.

```
public class QSortTrad {
    int antallTrader = 1, ferdige = 0;
    char[] a;

    QSortTrad(){
        a = "åøæzywvutsrqponmlkjihgfedcba".toCharArray();
        new Qtrad(0,a.length-1).start();
```

```
        while(antallTrader!=ferdige){
            try { Thread.sleep(0); } catch (Exception e) {};
        }
        for(int i=0;i<a.length;i++){System.out.print(a[i]+" ");}
    }

    public static void main(String[] args) {new QSortTrad();}

    class Qtrad extends Thread {
        int fra, til;
        Qtrad(int i, int j){fra = i; til = j;}

        public void run() {
            if (!(til-fra < 1)){
                char pivot = a[til];
                boolean ikkeFerdig = true;
                int forst = fra;
                int sist = til - 1;

                while (ikkeFerdig) {
                    while (a[forst] < pivot && forst<a.length-1) { forst++ ;}
                    while (a[sist] > pivot && sist>0) { sist--;}
                    if (forst < sist) {
                        bytt(a, forst, sist);
                        forst++; sist--;
                    } else ikkeFerdig = false;
                }

                bytt(a, forst, til);
                antallTrader++;
                new Qtrad(fra, forst-1).start();
                fra = forst+1;
                run();
            } else {
                ferdige++;
            }
        }
        void bytt (char[] a, int fra, int til) {
            char temp = a[til];
            a[til] = a[fra];
            a[fra] = temp;
        }
    }
}
```

13 Packages, interfaces, tilgangsnivåer og abstract

13.1 Packages

Pakker er grupper med relaterte klasser, som hjelper deg med å organisere koden din. Klasser definert i en pakke, er kun tilgjengelig via pakkenavnet. Med pakker har man også muligheten til å gjøre klasser kun tilgjengelige fra innsiden av pakken, noe som kan gi økt kontroll og sikkerhet.

En annen stor fordel med pakker er at når du navngir en pakke, opretter du et nytt *namespace*. Dette får man veldig bruk for når man jobber på større prosjekter. To klasser i Java kan ikke ha samme navn, for da får vi navnkollisjon. Men med namespaces løses dette. Det gjør at vi kan ha to klasser med samme navn, men som ligger i forskjellige pakker, eller namespaces, slik: *package01.CommonName.java* og *package02.CommonName.java*.

Alle klasser i Java hører til en pakke, men når ingen blir spesifisert blir default (global package) benyttet. Denne pakken har heller ikke noe navn.

13.1.1 Opprette en pakke

For å oprette en pakke bruker vi følgende syntax:

```
package MyPackage01;
```

Selve filene i pakken må da legges i en mappe med nøyaktig samme navn, *MyPackage01*. Man kan også ha et hierarki av pakker, f.eks.

```
package MyPackage01.MyPackage02...
```

Og vi definerer mappestrukturen deretter. Det kan være lurt å følge Maven-mappestrukturen, da det er en godt organisert og veletablert standard.

For at Java skal finne pakkenne må de enten ligge i samme mappe som du allerede jobber i, eller så kan man manuelt sett **CLASSPATH** som Java leter etter.

13.1.2 Importering av pakker

Istedenfor å extende hver klasse i en pakke, for å kunne benytte klassene kan man bruke *import*, med eller uten wildcard, slik:

```
import Package.Class // Importerer spesifikk klasse.
import Package.* // Importerer alle klasser i pakken.
```

Hele Java API'et er definert i pakker og importeres på samme måte. Det eneste er *java.lang*, som automatisk importeres, som blant annet inneholder *System*-klassen og *println()*.

13.2 Interfaces

Bruker interface på en klasse hvis vi ønsker egenskaper som flere forskjellige klasser skal arve. Vi beskrivker kun ønsket oppførsel, og ikke hvordan det skal gjøres. Metoder må skrives på nytt for hver gang det implementeres.

Et inteface defineres slik:

```
access interface InterfaceName {
    returntype methodName(parameter);
    type varName = value;
}
```

Og implementeres med,

```
class ClassName implements InterfaceName{ }
```

Vi kan implementere så mange interfaces vi ønsker, og separerer dem med komma.

13.2.1 Variabler i et interface

Variabler i et interface vil implisitt være *public*, *static* og *final*. Det gjør det veldig lett å dele variabler over flere klasser i et program, ved å implementere dette interfacet.

13.2.2 Interfaces kan extende

Vi kan også la et interface extende et annet interface, og hvis en klasse implementerer det ene interfacet, må den også implementere alle metoder som er arvet fra det andre interfacet.

13.3 Tilgangsnivåer

Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Så for å kunne bruke *protected* utenfor en pakke må det være en subklasse av klassen, f.eks. *class Classo1 extends Packageo1.Classo2*.

13.4 Abstract

Vi kan ha abstrakte metoder. Hvis en subklasse arver fra en superklasse med en abstrakt metode, er den nødt til å implementere denne metoden. Gjøres slik:

```
abstract type methodName(parameter);
```

Klassen de abstrakte metodene ligger i må da også defineres som abstrakte, slik:

```
abstract class ClassName{ }
```

Det vil ikke være mulig å lage objekter av abstrakte klasser, men kun av subklasser som extender og implementerer alle de abstrakte metodene.

Abstrakte klasser gjør at vi i objektorientert programmering kan lage en mer virkelighetslik objektmodell.

13.4.1 Bruk av final

Hvis vi ønsker at en metode ikke skal kunne overskrives ved arv, kan vi bruke **final** foran metoden. Det samme gjelder om vi ønsker en klasse som ikke skal kunne arves, eller variabler vi ikke vil skal kunne endres.

13.4.2 Forklaring

- class List
  Her lager vi en klasse som vi sier at inneholder til den generiske typen E.
- class Node
  En intern klasse som ved å være intern overtar egenskapene til E

14 Generiske typer

Objektholdere som kan holde hva som helst, egner seg godt til å lage datastrukturer, siden strukturen blir ikke avhengig av noen spesiell type objekt å kan derfor brukes på hva som helst.

```
class List<E>{
    Node root;

    class Node{
        Node(E e){ lagretObjekt = e; }
        E lagretObjekt;
        Node next;
    }
    void put(E e){
        Node n = new Node(e);
        if(root == null) root = n;
        else{
            n.next = root;
            root = n;
        }
    }
```

```
    E pop(){
        E tmp = root.lagretObjekt;
        root = root.next;
        return tmp;
    }
}
```

Over er et eksempel på en generisk LIFO-liste

15 Subklasser og polyformi

15.1 Arv

Vi kan arve fra en klasse i java, ved å bruke *extends*. Da vil klassen som arver bli kalt *subklassen* og klassen det arves fra bli kalt *superklassen*.

Vi definerer først superklassen, som en vanlig klasse

```
class ClassName {
    int someVar = 10;
}
```

Så kan vi la det som blir subklassen extende denne klassen, og arve fra superklassen, slik:

```
class SubClass extends Classname {
    System.out.print(someVar); // Vil printe ut 10, fra superklassen.
}
```

I Java kan man kun spesifisere en superklasse, men man kan godt ha et hierarki av arv og subklasser.

Det som arv oftest benyttes til er å beskrive ett generelt objekt og la subklasser utvide dette, så kan de igjen spesifisere de elementene som gjør dem spesielle. F.eks. kan man tenke at man lager en superklasse som definerer et *kjøretøy*, også har du subklasser som utvider dette, f.eks. til *varebil* og *personbil*.

15.2 Tilgangskontroll og arv

Selv om du arver metoder og variabler fra en klasse, får du ikke tilgang til elementer som er definert som **private**.

15.3 Arv og konstruktører

Ved kjøring av en subklasse, vil både konstruktøren til subklassen og til superklassen kjøre. Vi kan kalle på konstruktøren til superklassen ved metoden **super(parameter)**.

Vi kan også bruke *super* til å få tilgang til medlemmer av superklassen, f.eks. *super.varName* eller *super.methodName()*.

Konstruktøren til superklassen vil altid eksekveres først, også går det nedover i hierarkiet. Kaller man på super() må dette gjøres øverst i konstruktøren.

15.4 Polymorfi

Vi kan overskrive metoder i subklassene. Java bestemmer hvilken av metodene som kjøres, ved run-time, ved å se på hvilke objekter som refereres til, selv om selve pekeren er av typen Superklasse.

15.5 Object-klassen

Java definerer en Object-klasse, som er en superklasse til alle klasser. Det gjør at man kan ha en referansevariabel av typen Object som kan referere til alle typer klasser.

Object definerer ett sett med metoder man kan benytte, blant annet *Object clone()*, som kloner objektet til et nytt, selvstendig objekt.

16 Unntak (Exceptions)

16.1 Hva er unntak

Et unntak skjer når noe går galt, for eksempel å bruke en metode i et objekt som er null(nullPointerException). Eller prøver å få tak i en verdi i et array som er utenfor arrayets størelse (IndexOutOfBoundsException).

16.2 Fange unntak

Å fange unntak er lett

```
try {
    noe();
} catch (Exception e) {
    gjørDetSomSkalSkjeHvisNoeGårGalt();
}
```

Man kan også bruke exceptions som en sofistikert if. Hvis man vet at koden funker, men at det skal skje noe på et visst punkt eller etter en viss tid, kan man bruke exceptions til å behandle unntaket når det kommer, eller bare gå videre i koden.

```
try {
    while(noe()) { kodeSkjer(); }
} catch (Exception e) {
    løkkaErFerdigFordiNoeKultSkjedde();
}
```

16.3 Definer eget unntak

Vi kan definere våres egene unntak, hvis vi ikke finner et passende unntak i Java sin *Exception*-klasse. Det gjøres slik:

```
class EgendefinertException extends Exception {}
```

Vi kan så la en klasse eller en metode bruke dette, slik:

```
void myMethod() throws EgendefinertException {
    if(a != true) {
        throw new EgendefinertException();
    }
}
```

Unntaket må så kastes vidre til main, eller fanges og behandles.

17 Finalize, garbagecollector etc

finalize() er en metode du kan definere i klasser, som automatisk kjøres før objekter blir tatt av garbage collectoren og slettes. F.eks. hvis du skal dobbeltsjekke at en fil har blitt lukket.

18 Tråder (Threads)

18.1 Oprettelse av en tråd

Kan implemenere *Runnable* eller extende *Thread* classs. Tråder er små prosesser inne i én prosess. Det er raskerkere å skifte fra en tråd til en annen enn fra en prosess til en annen. Å bruke flere prosesser/tråder for å løse en oppgave kalles paralellprogrammering.

For å oprette en tråd gjør vi følgende:

```
MyThreadClass myThreadClass = new MyThreadClass();
Thread newThread = new Thread(myThreadClass);
newThread.start(); // Kaller på run()-metoden.
```

18.2 Nyttige metoder

Hvis vi lurer på om en tråd er fullført eller ikke kan vi kalle på metoden *isAlive()*, som vil returnere en boolean.

En annen måte å vente/få beskjed når en tråd er fullført er å bruke metoden *join()*. Det gjøres på følgende måte:

```
myClass.myThread.join();
```

Vi kan også sette **prioriteten** til en tråd ved å kalle metoden *setPriority(int level);*. Prioritetsnivået er et tall mellom 1 og 10.

18.3 Synkronisering

Vi kan enkelt synkronisere metoder, hvis vi ikke ønsker at flere tråder skal kunne jobbe på de samme variablene, eller metodene, ved å legge til nøkkelordet *synchronized*. Den putter på en lås på metoden, som sier at kun en tråd kan jobbe på metoden av gangen, og andre tråder må vente.

Vi kan også synkronisere en blokk av kode, ved følgende statement:

```
synchronized(object) {
    // Kode til å synkronisere.
}
```

18.4 Kommunikasjon mellom tråder

Hvis vi har flere tråder som skal jobbe etterhverandre på et synkronisert objekt, men det ene objektet er avhengig av at den andre tråden har fullført en prosedyre, kan vi benytte oss av *wait()* og *notify()*-metoden.

18.4.1 wait()

Gjør at tråden sovner, og låser opp låsen på det synkroniserte objektet. Kan kaste *InterruptedException*.

18.4.2 notify()

notify() vekker den andre sovende tråden, hvis den blir kalt inne i det samme, synkroniserte objektet.

18.4.3 notifyAll()

notifyAll() gjør det samme som notify(), men den vekker alle sovende tråder.

18.5 Eksempel på en enkel tråd

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;

class Klokke {
   public static void main(String[] args) throws IOException {
        System.out.println("Trykk [ENTER] for ? starte og stoppe");

      BufferedReader minInn = new BufferedReader
                                  (new InputStreamReader(System.in));

      minInn.readLine ( );

      // Her lages stoppeklokke-objektet:
      Stoppeklokke stoppeklokke = new Stoppeklokke();

      // og her settes den nye tr?den i gang.
      stoppeklokke.start();

      minInn.readLine ( );
      stoppeklokke.avslutt();
   }
}

class Stoppeklokke extends Thread {
   private volatile boolean stopp = false;
   // blir kalt opp av superklassens start-metode.
   public void run() {
      int tid = 0;
      while (!stopp) {
         System.out.println(tid++);
         try {
            Thread.sleep(1 * 1000); // ett sekund
         } catch (InterruptedException e) { }
      }
   }

   public void avslutt() {
      stopp = true;
   }
}
```

Nøkkelordet *volatile* brukes til å indikere at en variabel skal endres av flere enn en tråd.

19 Brukergrensesnitt

```
import javax.swing.*;
import java.awt.*;

class RammeDemo2 extends JFrame {
    RammeDemo2() {
        // En annen måte å sette tittel på rammen:
        super("Første vindu");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300,200);
        setVisible(true);
    }
    public static void main(String[] args) {
        new RammeDemo2();
    }
}
```

Anonym klasse, når klassen ikke har navn, og er skrevet som parameter til en metode.

Bruk *repaint()*-metoden for å tegne vindu på nytt..

20 Shorthands

20.1 If-shorthand

```
if (condition) {
    return x;
}
return y;
```

Burde isteden skrives slik:

```
return (condition ? x : y);
```

21 Prinsipper for større programmer

-KISS

- Moduler
  - Små
    - Definert av grensesnitt
    - Kun en funksjonalitet

        - F.eks. LinkedList
- Dokumentasjon
- Testing
- Versjonkontroll
- Konvensjoner
  - Hvordan kode ser ut
  - Hvordan kode struktureres

Ved over ca 40 linjer i en metode, på tide å splitte.