# Noter

Oktober 2013

Dette er en samling af noter skrevet til kurset Programmering 2 som supplement til den anvendte lærebog. Java-kildeteksten for de anvendte programeksempler kan nås via kursuswebsiden

# Indhold

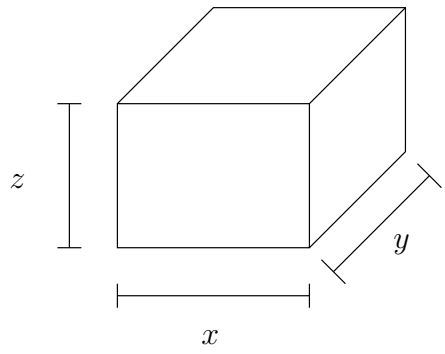
# 1 Klassedesign og invarianter

Dette kapitel er et supplement til kapitel 3, *Guidelines for Class Design* i Horstmanns bog *Object oriented Design & Patterns*. Vi viser eksempel på brug af en speciel form for kontrakter nemlig implementations- og løkkeinvarianter ved implementation af hele klasser og enkelte metoder. En implementationsinvariant kan simplificere implementationen af en klasse og dermed gøre det lettere at indse at den er implementeret korrekt, ligesom en løkkeinvariant kan gøre det lettere at sikre rigtig start og slut af sweep og iterationer. Vi vil kun anvende løkkeinvarianter i en forenklet udgave, som vi kalder visuelle løkkeinvarianter. En sådan illustreres med en figur fremfor i matematisk notation.

I kurset *Algoritmer og datastrukturer 1* gives en mere grundig introduktion til brugen af invarianter.

## 1.1 Implementationsinvariant for en kasse

Vi ønsker at modellere en papkasse repræsenteret ved de tre sidelængder




På papkasser kan man oplagt definere en ordning, således at $k_1 \leq k_2$, hvis $k_1$ kan være inde i $k_2$ idet vi kræver, at siderne holdes parallelle. Vi ser bort fra væggenes tykkelse, så en papkasse kan f.eks. være inde i sig selv.

Den offentlige interface skal tilbyde følgende operationer:

- En konstruktør der givet tre tal laver en kasse med de angivne dimensioner
- en metode der beregner rumfanget af kassen
- en metode der beregner om kassen kan være inde i en anden givet kasse

- en metode der beregner den mindste kasse, som både kassen selv og en anden kasse kan være inden i.

Dvs. klassen skal have følgende form

```
public class Box {

    public Box(int w, int h, int d) { ... }

    public int volume() { ... }

    public boolean fitsIn(Box b) { ... }

    public Box combine(Box b) { ... }
}
```

Når vi overvejer, hvorledes f.eks. `fitsIn` skal skrives, så ser vi hurtigt en komplikation. Vi er ikke interesseret i papkasser, der står "på skrå", men alligevel kan en papkasse vendes på 6 essentielt forskellige måder, så det ser ud som om, vi bliver nødt til at undersøge en hel del tilfælde. Det kan vi dog undgå hvis vi påtvinger klassen `Box` en implementationsinvariant som instans variablerne skal overholde:

```
    /**
     * IMPLEMENTATION INVARIANT:   x <= y <= z
     */
    private int x,y,z;
```

Implementationsinvarianten kan ses som en kontrakt mellem implementørerne af konstruktøren og de forskellige metoder, idet

- Inden udførelsen af en public metode kan man gå ud fra at invarianten er overholdt.
- Inden afslutning af konstruktøren eller en af de offentlige metoder skal det sikres at invarianten er overholdt

Bemærk også at de tre instansvariabler har visibility modifier `private` og derfor kun kan ændres af kode inde i klassen. Det er derfor let at se at invarianten altid vil blive overholdt, uanset hvordan klassen benyttes.

Vi kommer til følgende implementation

```
public class Box {

    /**
```

```
     * IMPLEMENTATION INVARIANT:   x <= y <= z
     */
    private int x,y,z;

    public Box(int w, int h, int d) {
        x = w; y = h; z = d;
        while (x>y || y>z) {
            if (x>y) {int temp = x; x = y; y = temp; }
            if (y>z) {int temp = y; y = z; z = temp; }
        }
    }

    public int volume() {
        return (x*y*z);
    }

    public boolean fitsIn(Box b) {
        return ( x<=b.x && y<=b.y && z<=b.z );
    }

    public Box combine(Box b) {
        return new Box(Math.max(x,b.x),
                       Math.max(y,b.y),
                       Math.max(z,b.z));
    }
}
```

I metoderne `fitsIn` og `combine` udnytter vi, at invarianten gælder. I konstruktøren sorterer vi `x`, `y` og `z`, således at invarianten bliver etableret.

Hvis vi ikke opretholdt invarianten, så blev vi nødt til at tage stilling til alle de 6 forskellige måder, man kan orientere en papkasse på. F.eks. ville metoden `fitsIn` blive til

```
public boolean fitsIn(Box b) {
    return ( (x<=b.x && y<=b.y && z<=b.z) ||
             (x<=b.x && z<=b.y && y<=b.z) ||
             (y<=b.x && x<=b.y && z<=b.z) ||
             (y<=b.x && z<=b.y && x<=b.z) ||
             (z<=b.x && x<=b.y && y<=b.z) ||
             (z<=b.x && y<=b.y && x<=b.z) );
}
```

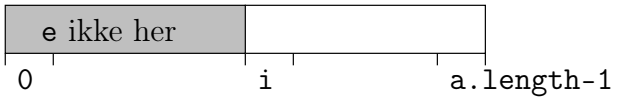
og operationen `combine` kunne blive helt uoverskuelig.

## 1.2 Visuel løkkeinvariant ved søgning

Det første eksempel på anvendelse af en løkkeinvariant er en simpel lineær søgning. Der skal implementeres en metode, som givet et array og et tal returnerer tallets position i arrayet, hvis det findes i arrayet og ellers returnerer `-1`. Metodens signatur er således

```
public int linearSearch(int[] a, int e) { ... }
```

Implementationen klares let ved et enkelt sweep gennem arrayet. Et typisk øjebliksbillede under dette sweep er følgende:




Vi kan samtidig opfatte tegningen som en kontrakt (den visuelle løkkeinvariant) der skal overholdes af den der implementerer metoden, dvs.

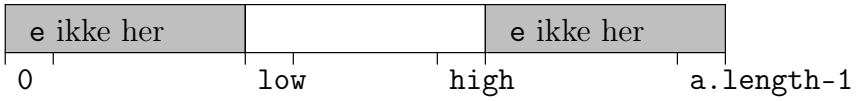
- tegningen kan antages at være korrekt før hvert iterationsskridt
- det skal sikres at tegningen stadig er korrekt efter hvert skridt, og efter initialiseringen af løkkevariablen `i`.

På baggrund af tegningen er det let at skrive sweep-løkken, så løkkevariablen `i` initialeres rigtigt og vi anvender det rigtige stopkriterium. Ved start af sweep er det skraverede område tomt og `i` skal initialiseres til 0. Undervejs må vi kun tælle `i` op, når det første element i det hvide område ikke er identisk med det element, vi leder efter. Hvis det første element i det hvide område skulle vise sig at være det element, vi leder efter kan søgning umiddelbart standses. Eneste anden grund til at standse er at det hvide område bliver tomt, dvs. søgningen fortsætter så længe `i<a.length`:

```
public int linearSearch(int[] a, int e) {
    int i=0;
    while (i<a.length) {
        if (a[i]==e) return i;
        else i++;
    }
    return -1;
}
```

Hvis man søger i en sorteret liste er binær søgning et effektivt alternativ til lineær søgning. Antag at vi leder efter "Johansen" i telebogen. Hvis vi slår op ca. midt i telefonbogen og finder en side med "Mikkelsen" kan vi eliminere den halvdel af telefonbogen der følger efter "Mikkelsen" fra den videre søgning. Hvis vi dernæst slår op ca. midt i den tilbageværende halvdel af bogen og finder en side med "Hedegård" kan vi tilsvarende eliminere den del af bogen der kommer før "Hedegård". På denne vis fortsætter vi med at slå op ca. midt i den tilbageværende del af telefonbogen indtil vi enten har fundet en side med "Johansen" eller vi har elimeneret hele bogen og må konkludere at "Johansen" ikke forekommer.

Hvis denne strategi anvendes på søgning i et sorteret heltalsarray kan et typisk øjebliksbillede se ud som følgende:




Som for lineær søgning er det forholdsvis let at implementere metoden, når man hele tiden har tanke for at overholde den visuelle løkkeinvariant.

```
public int binarySearch(int[] a, int e) {
    int low = 0;
    int high = a.length-1;
    while (low<=high) {
        int mid = (low+high)/2;
        if (a[mid]==e) return mid;
        else if (a[mid]<e) low = mid+1;
        else high = mid-1;
    }
    return -1;
}
```

For en fornuftig søgning er det væsentlig at det "hvide" ikke-undersøgte område på tegningen mindskes (så meget som muligt) i hvert skridt. Ved lineær søgning, der består af et enkelt sweep, sikres det ved at tælle løkkevariablen een frem i hvert skridt. Ved binær søgning er progressionen mindre gennemskuelig. Hvis man er uforsigtig kan man lave en implementation, hvor det hvide område i nogle tilfælde ikke mindskes, og dermed får vi en evig løkke, f.eks.:

```
public int loopingBinarySearch(int[] a, int e) {
    int low = 0;
```

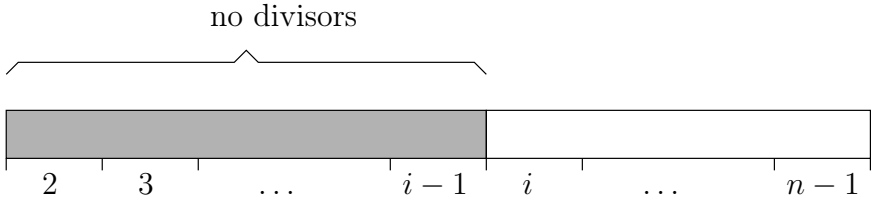
```
    int high = a.length-1;
    while (low<=high) {
        int mid = (low+high)/2;
        if (a[mid]==e) return mid;
        else if (a[mid]<e) low = mid;
        else high = mid;
    }
    return -1;
}
```

Java stiller en polymorf binær søgning til rådighed for brugeren i metoden `Collections.binarySearch`.

## 1.3 Visuel løkkeinvariant ved primtalsfaktorisering

Først betragtes problemet at afgøre om et tal er et primtal. Primtallene er $2, 3, 5, 7, 11, 13, \ldots$, og der gælder at $n \geq 2$ er et primtal hvis og kun hvis $n$ ikke har en divisor $d \in \{2, 3, \ldots, n-1\}$.

For at afgøre om $n$ er et primtal, kan man lave et sweep over de mulige divisorer, og for hver divisor checke om den faktisk dividerer $n$. Vi anvender en løkkevariabel $i$ til at angive hvor langt vi er kommet med sweepet, og som tidligere kan vi tegne et øjebliksbillede

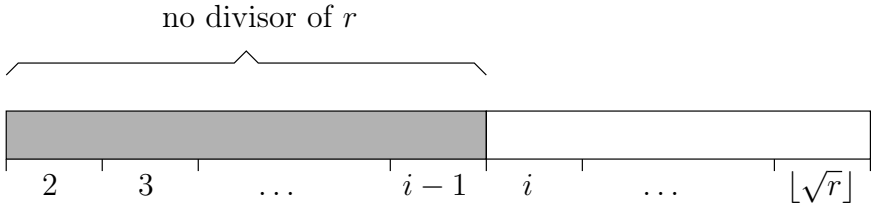



Baseret herpå kommer man let frem til følgende metode med precondition $\texttt{n} \geq 2$.

```
public boolean isPrime(int n) {
    int i = 2;
    while (i<n) {
        if (n%i==0) return false;
        else i++;
    }
    return true;
}
```

Næstefter løses det udvidede problem, hvor man finder den fuldstændige primfaktorisering af $n$. F.eks ønsker vi at metodekaldet `factor(75)` returnerer teksten `"3*5*5"`.

Man kan søge gennem de mulige divisorer $d \in \{2, 3, \ldots, n-1\}$ ved et sweep. Vi ønsker at en divisor der forekommer med multiplicitet $> 1$ skrives et tilsvarende antal gange. Tages et øjebliksbillede under sweepet, vil vi have udskrevet nogle faktorer, og der vil restere $r$, den ufaktoriserede del af $n$. Hvis $r$ kan faktoriseres yderligere, vil den have en divisor $d \leq \sqrt{r}$. Denne indsigt leder til følgende visuelle invariant




Herudfra implementeres en metode, der som precondition har $n \geq 2$:

```
public String factor(int n) {
    int r = n;
    int i = 2;
    String s = "";
    while (i*i<=r)   {
        if (r%i == 0) {
            s = s + i + "*";
            r = r/i;
        }
        else i++;
    }
    return s + r;
}
```

Den angivne metode vil aldrig lede efter divisorer der er større end $\sqrt{n}$. En tilsvarende optimering kan foretages på implementationen af `isPrime`. Med eller uden denne optimering er begge algoritmer alt for langsomme til praktisk brug ved faktorisering (primalitetstest) af tal med f.eks. tredive eller flere cifre. Java tilbyder en randomiseret primalitetstest i metoden `BigInteger.isProbablePrime`, der er hurtig for mangecifrede tal, men til gengæld kan returnere et fejlagtigt svar. Den eneste kendte effektive (randomiserede) algoritme til at finde primtalsfaktoriseringen af mangecifrede tal

forudsætter adgang til en kvantecomputer af en størrelse, som det endnu ikke er lykkedes at konstruere.

# 2 Rekursive metoder

En *rekursiv* metode indeholder *kald af sig selv*. Dette begreb er ikke så fremmed, som det måske synes ved første øjekast. Et velkendt eksempel på en rekursivt defineret udregningsmetode udgøres af de sædvanlige regler for differentiation af rationale funktioner (det vil sige funktioner som er summer, produkter eller kvotienter mellem polynomier). Reglerne er som følger; $f(x)$ og $g(x)$ er rationale funktioner og $c$ er en reel konstant

1) $c' = 0$

2) $(x^n)' = nx^{n-1}$

3) $(cf(x))' = cf'(x)$

4) $(f(x) + g(x))' = f'(x) + g'(x)$

5) $(f(x)g(x))' = f(x)g'(x) + f'(x)g(x)$

6) $\left(\dfrac{f(x)}{g(x)}\right)' = \dfrac{g(x)f'(x) - f(x)g'(x)}{g^2(x)}$

Reglerne 1) og 2) angiver direkte resultatet af at differentiere simple udtryk, mens reglerne 3)–6) angiver, at et sammensat udtryk differentieres ved, at det opbrydes i deludtryk, som hver især differentieres ved (rekursiv) anvendelse af reglerne.

I det følgende vises et antal eksempler på rekursive metoder.

## 2.1 Rekursiv funktion: Fibonacci tal

Fibonacci tallene er opkaldt efter matematikeren *Leonardo Fibonacci af Pisa*, som i 1202 udgav værket *Liber Abaci* (bogen om beregninger), hvori han blandt andet stillede spørgsmålet

> Hvor mange kaninpar kan der blive af et nyfødt kaninpar på et år, hvis hvert par føder et nyt par hver måned og et nyfødt par føder første gang efter to måneder?

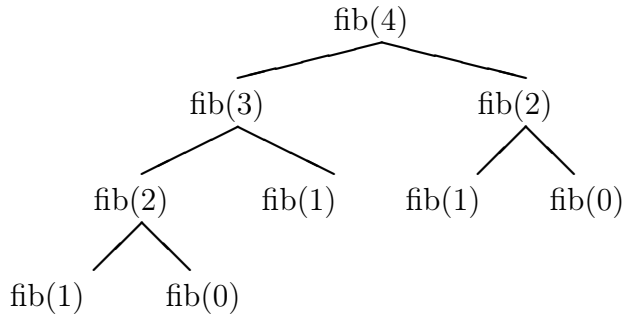
Hvis det antages, at kaniner aldrig dør, er det ikke svært at se, at der spørges efter det 12. element i følgen $\{F_n\}_{n\geq 0}$ defineret ved

$$F_n = \begin{cases} 1 & n = 0, 1, \\ F_{n-1} + F_{n-2} & n > 1. \end{cases}$$

Denne definition er rekursiv, da formlen for det $n$'te Fibonacci-tal henviser til mængden af (mindre) Fibonacci-tal. Vi kan skrive en tilsvarende simpel funktion, der udregner det $n$'te element

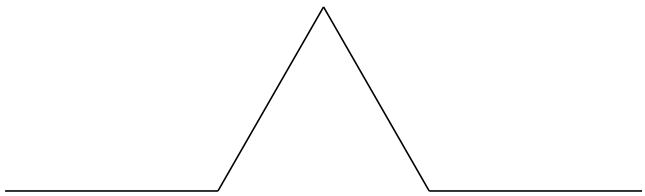
```
public long fib(int n) {
    if (n <= 1) return 1;
    else return fib(n - 1) + fib(n - 2);
}
```

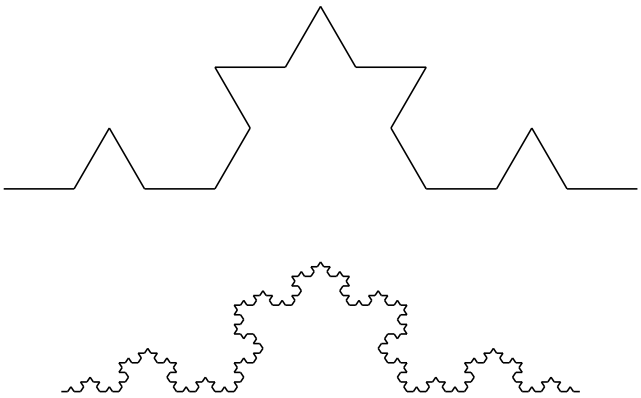
Et kald af fib(4) vil give anledning til følgende træstruktur af procedurekald




## 2.2 Rekursiv grafik: Fraktaler

En *fraktal* er en kurve, som intetsteds er glat nok til at kunne approximeres med linjestykker. Et simpelt eksempel på en fraktal er den såkaldte *Koch-kurve*, der opstår som grænseværdien (for $n$ gående mod uendelig) af *Koch-linjen* af orden $n$. En Koch-linje af orden $n$ opnås ud fra en Koch-linje af orden $n-1$ ved at give alle linjestykker i den et trekantet hak. De følgende er Koch-linjer af orden 1, 2 og 4





Vi kan skrive en meget simpel rekursiv procedure, der tegner en Koch-linje af en given orden. Vi anvender et `Crayon` objekt `c`, hvor metoden `turn` ændrer blyantens tegneretning med et specificeret antal grader og metoden `move` trækker blyanten den specificerede længde i den aktuelle tegneretning så et linjestykke tegnes.

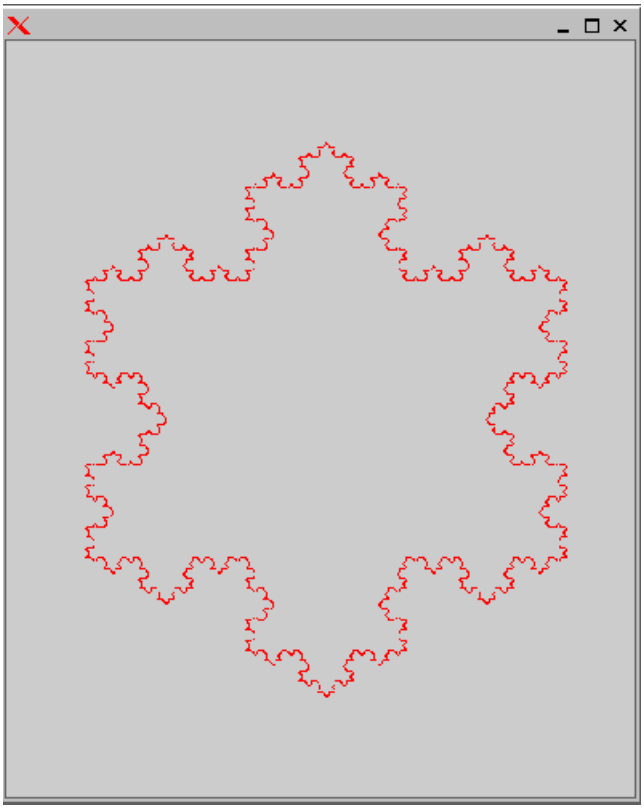
```
private void kochLine(int order, double len) {
    if (order == 0) c.move(len);
    else if (order > 0) {
        kochLine(order-1,len/3); c.turn(-60);
        kochLine(order-1,len/3); c.turn(120);
        kochLine(order-1,len/3); c.turn(-60);
        kochLine(order-1,len/3);
    }
}
```

Bemærk, at vi foretager 4 rekursive kald, og at vi her har et eksempel på en rekursiv procedure, hvor en ikke-rekursiv formulering forekommer at være endog meget besværlig.

Følgende program tegner en snefnug-lignende figur bestående af tre Koch-linjer (kildekoden til `Crayon`-klassen er udeladt men kan findes i eksempelkataloget).

```
import java.awt.*;
public class KochLineTest {
  private static Crayon c;

  private static void kochLine(int order, double len) { ... }
```

```
  public static void main(String[] args) {
    c = new Crayon(Color.red,1);
    c.jumpto(50,150);
    kochLine(4,300); c.turn(120);
    kochLine(4,300); c.turn(120);
    kochLine(4,300); c.turn(120);
  }
}
```



## 2.3 Gensidig rekursion

Det er tilladt at definere to metoder som gensidigt rekursive, f.eks.:

```
public void ned(int n) {
    while(n%2==0) n = n/2;
    op(n);
}

public void op(int n) {
```

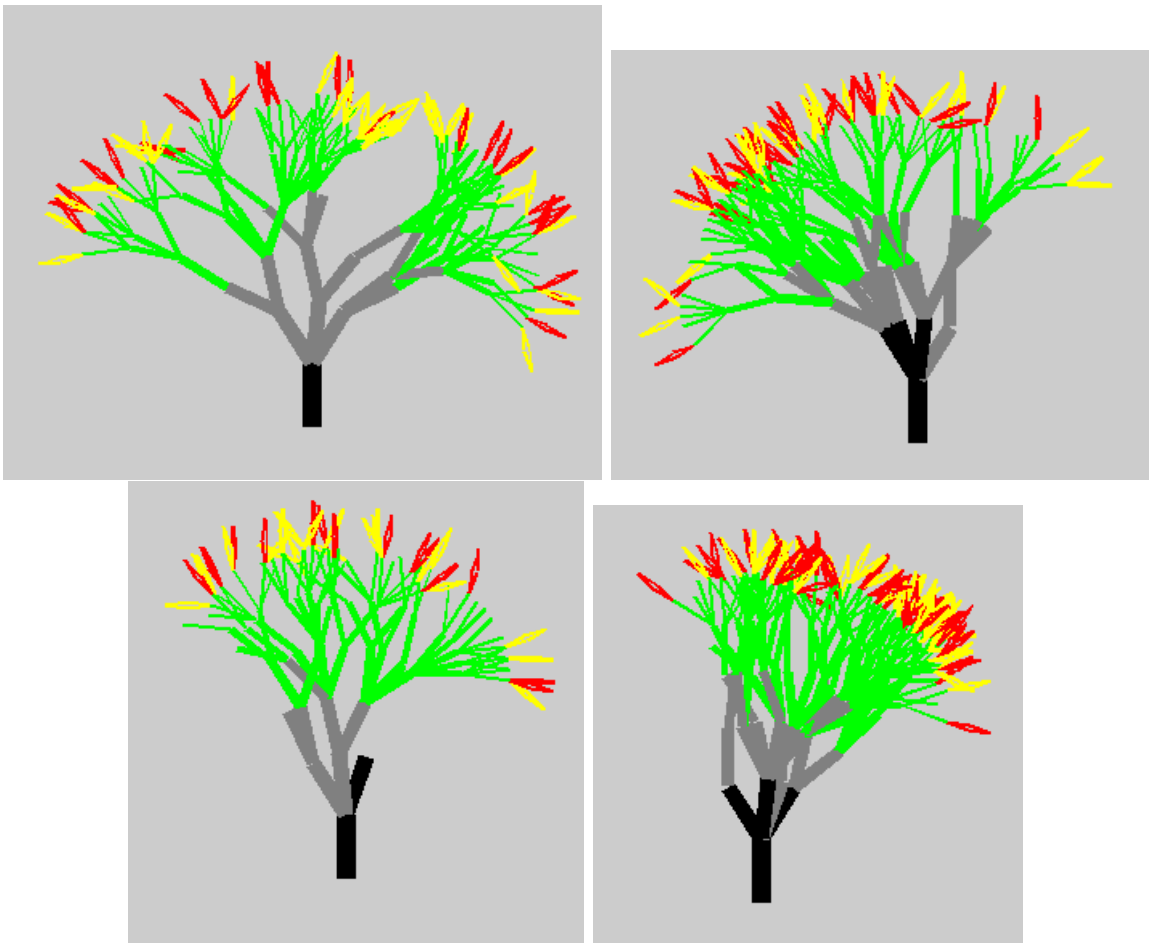
```
    if (n>1) { n = 3*n+1; ned(n); }
}
```

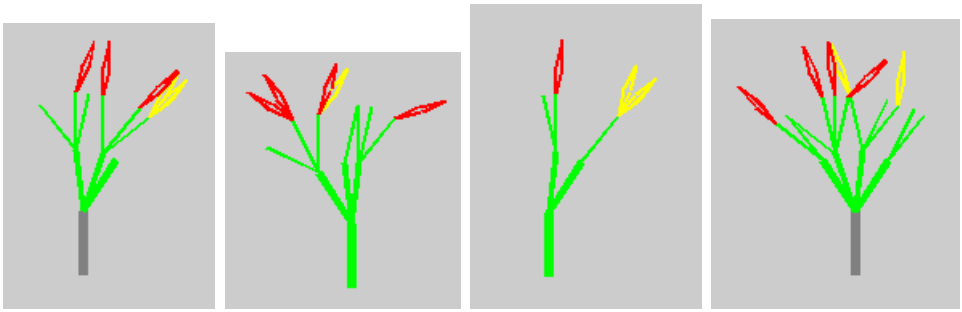
Det er ukendt om kald af metoden `ned` standser for alle input (aktuelle parametre)!

I afsnittet om rekursive strukturer, skal vi se flere eksempler på gensidigt rekursive metoder.

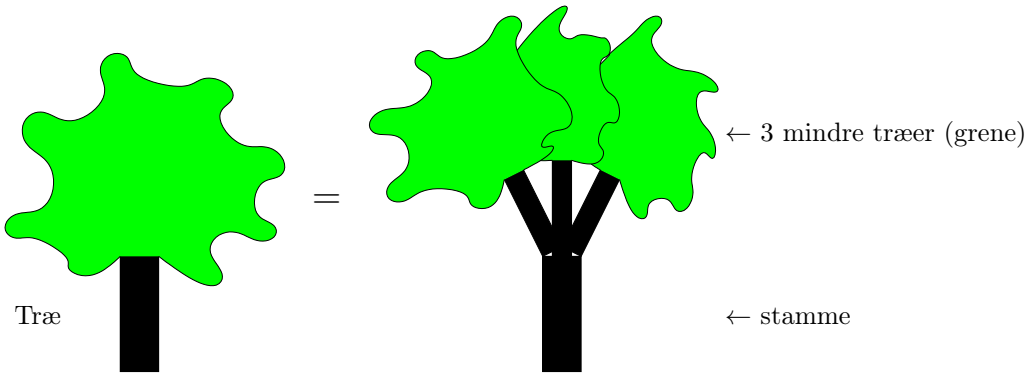
## 2.4 Randomisering og rekursion

Her er nogle eksempler på træer (blomster) der er genereret ved en simpel rekursiv teknik kombineret med randomisering.





Ved genereringen af et træ opfattes det enten som bestående af en stamme med flere mindre træer (grene) på toppen, eller blot som værende et enkelt blad.




De foranstående figurer er genereret ved at lade et træ bestå af en stamme med 6 grene (deltræer). Vinklen mellem grenene er 13 grader, og et deltræ tegnes ikke nødvendigvis. Faktisk er sandsynligheden for at et deltræ tegnes kun 40%.

På figurerne er den første stamme tegnet med bredde 6 og bredden aftager derpå en for hvert rekursionsskridt. Når bredden når ned på 0 tegnes et blad i stedet for et træ. Dvs. bladet tegnes ikke nødvendigvis men kun med sandsynlighed 30%. Stammer tegnes i en tilfældig farve, der delvist afhænger af bredden. For at gøre udtegningen relativt simpel anvendes en udgave af `Crayon` klassen, som tillader ændring af farve og bredde på en farveblyant (metoderne `setColor`, `setWidth`).

```
import java.awt.*;
public class TreeTest {
  private static CCrayon c;

  /** tree draws a random tree
   * @param n - order of tree
```

```
   * @param len - length of trunk  */
  public static void tree(int n, int len) { ... }

  /** leaf draws maybe(!) a leaf of a random color
   * @param len - side length of leaf  */
  private static void leaf(int len) { ... }

  private final static int BRANCH_MAX = 6;
  private final static int BRANCH_ANGLE = 13;
  private final static double BRANCH_PROB = 0.4;
  private final static double LEAF_PROB = 0.3;

  public static void main(String[] args) {
    c = new CCrayon();
    c.jumpto(200,400);
    c.turn(-90);
    tree(6,40);
  }
}
```

Den basale rekursive metode til udtegning af et træ ser således ud:

```
  private static void tree(int n, int len) {
    if (n==0) leaf(len/2);
    else {
      int bias = (int) (2*Math.random());
      if (n+bias >=6) c.setColor(Color.black);
      else if (n+bias >=4) c.setColor(Color.gray);
      else c.setColor(Color.green);
      c.setWidth(2*n);
      c.move(len);
      c.turn((BRANCH_ANGLE*(BRANCH_MAX-1))/2.0);
      for (int i = 1 ; i<=BRANCH_MAX ; i = i+1 ) {
        if (Math.random()<BRANCH_PROB)  tree(n-1, len-2);
        c.turn(-BRANCH_ANGLE);
      }
      c.turn((BRANCH_ANGLE*(BRANCH_MAX+1))/2.0);
      c.turn(180); c.jump(len); c.turn(-180); //return pen to base of tree
    }
  }
```

Og et blad udtegnes med denne metode

```
  private static void leaf(int len) {
    if (Math.random()<LEAF_PROB) {
      if (Math.random()<0.5) c.setColor(Color.red);
      else c.setColor(Color.yellow);
      c.setWidth(2);
      c.turn(-BRANCH_ANGLE/2.0);   c.move(len);
      c.turn(BRANCH_ANGLE);        c.move(len);
      c.turn(180-BRANCH_ANGLE);    c.move(len);
      c.turn(BRANCH_ANGLE);        c.move(len);
      c.turn(180-BRANCH_ANGLE/2.0);
    }
  }
```

Ved at ændre på konstanterne kan man frembringe træer (planter, blomster) med andre karakteristika.

## 2.5 Generelt om rekursion

Ligesom while-sætninger kan kald af rekursive metoder give sætninger, hvis udførelse aldrig terminerer. Det simpleste eksempel er

```
public void loopingRecursiveMethod()
{  loopingRecursiveMethod();  }
```

Et kald af metoden `loopingRecursiveMethod` fører til et nyt, rekursivt kald af `loopingRecursiveMethod`, der fører til endnu et rekursivt kald af `loopingRecursiveMethod` og så videre. Hvis enhver instans af en rekursiv metode udfører endnu et rekursivt kald, så vil udførelsen naturligvis aldrig stoppe. Det er derfor vigtigt, at enhver rekursiv metode indeholder en *rekursionsbetingelse*, der på et tidspunkt tillader proceduren at undgå flere rekursive kald. Hvis man kigger tilbage på de forskellige eksempler i dette afsnit vil man se at de alle indeholder en sådan rekursionsbetingelse.

En rekursiv løsning kan være betydeligt lettere at forstå (og programmere) end en iterativ løsning. F.eks. vil det være kompliceret (omend bestemt muligt) at lave en iterativ algoritme til at tegne Koch kurver. Til gengæld er en rekursiv løsning i nogle tilfælde meget ineffektiv. F.eks. vil den rekursive Fibonacci metode tage meget længere tid at udføre end en tilsvarende iterativ metode (hvorfor?)

```
public long iterativeFib(int n) {
    if (n <= 1) return 1;
```

```
    long fold = 1;
    long fold2 = 1;
    long fnew = 1;
    for (int i = 2; i <= n; i++) {
        fnew = fold + fold2;
        fold2 = fold;
        fold = fnew;
    }
    return fnew;
}
```

# 3 Rekursive datastrukturer

I kurset er tidligere introduceret rekursive metoder, dvs. algoritmer, hvis beskrivelse er selvrefererende. Tilsvarende kan man tale om rekursive strukturer. Eksempler:

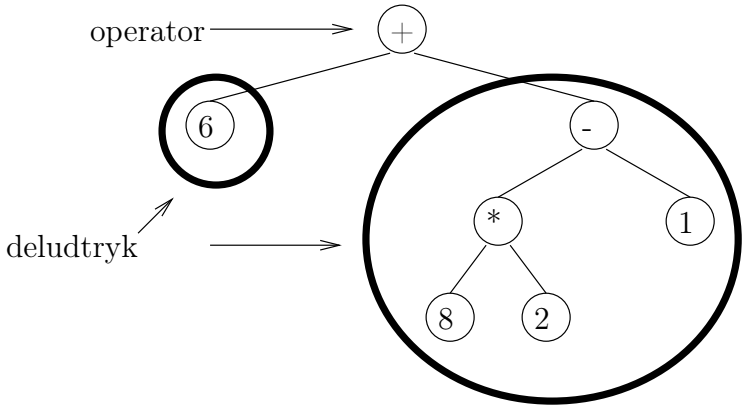
- En *liste* består af et enkelt element efterfulgt af en (kortere) liste eller listen er tom.
- Et *filsystem* består af et directory indeholdende en række mindre filsystemer eller filsystemet består af en enkelt fil

Ved definitionen af en rekursiv metode er der vigtigt at give metoden mulighed for at standse. Tilsvarende er der for både listen og filsystemet lagt en smutvej ud af rekursionen, listen kan være tom og filsystemet kan bestå af en enkelt fil.

## 3.1 Rekursive træer

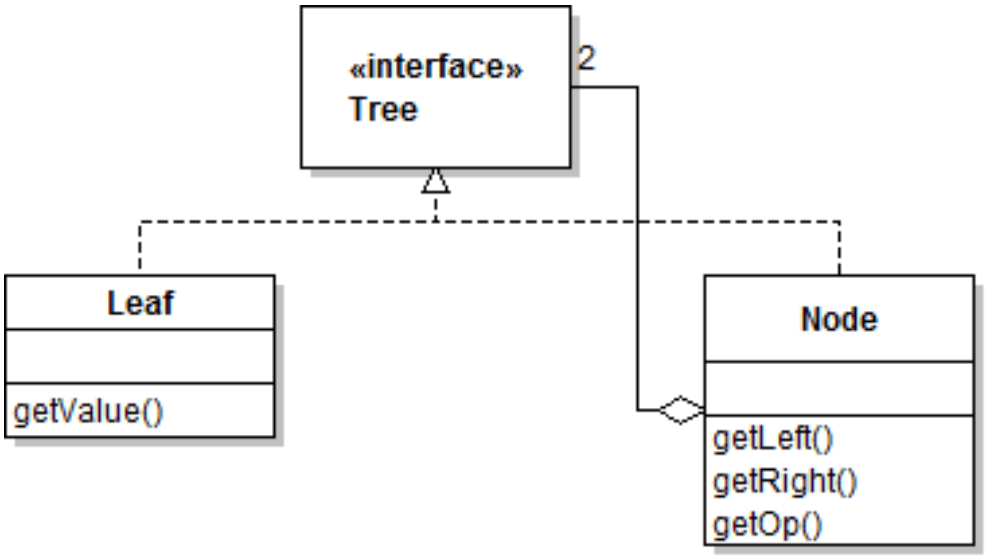
Filsystemet er et eksempel på en rekursiv træstruktur, og denne note vil anvende rekursive binære udtrykstræer som gennemgående eksempel.

Udtrykket $(6+((8\cdot 2)-1))$ kan opfattes som bestående af en operator "+"og to (mindre) udtryk "6"og "$((8\cdot 2)-1)$".




Man kan faktiske beskrive alle mulige aritmetiske udtryk (over $+,\cdot$) rekursivt. Enten består udtrykket af en operator og to deludtryk eller også består udtrykket af et tal alene.

En sådan rekursiv struktur repræsenteres naturligt ved et interface og implementerende klasser. For udtrykstræernes vedkommende får man




```
public interface Tree { }

public class Leaf implements Tree {
    private int value;
    public Leaf(int n) { value = n; }
    public int getValue() { return value; }
}

public class Node implements Tree {
  private Operator op;
  private Tree left;
  private Tree right;

  public Node(Tree l, Operator o, Tree r)
  { op = o; left = l; right = r; }

  public Operator getOp() { return op; }
  public Tree getLeft() { return left; }
  public Tree getRight() { return right; }
}
```

hvor `Operator` er en Enumerated Type som beskrevet i Horstmann side 267–268 (2.udgave).

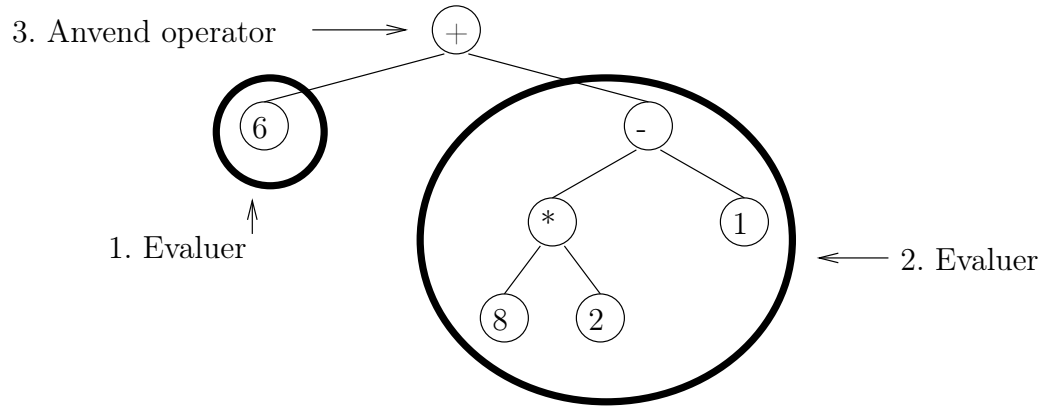
```
public enum Operator {
  PLUS("+"), MINUS("-"), MULT("*"), DIV("/");
```

```
  private String name;
  private Operator(String name) { this.name = name; }
  public String toString() { return name; }
  public static Operator parseOp(String s) {
      for (Operator op : Operator.values())
  if (op.name.equals(s)) return op;
      throw new UnsupportedOperationException(s);
  }
  public int apply(int left, int right) {
      switch (this) {
          case PLUS:  return left + right;
          case MINUS:  return left - right;
          case MULT:  return left * right;
          case DIV:  return left / right;
          default: throw new UnsupportedOperationException(this.toString());
      }
  }
}
```

Det tidligere nævnte udtryk kan repræsenteres ved

```
Tree t =
  new Node(
    new Leaf(6),
    Operator.PLUS,
    new Node(new Node(
      new Leaf(8),Operator.MULT,new Leaf(2)),
      Operator.MINUS,
      new Leaf(1)));
```

Næsten alle former for anvendelse af et udtrykstræ involverer et gennemløb af træet. Hvis værdien af udtrykket skal beregnes, må man nødvendigvis kende alle operatorer og alle tal i hele træet. Denne information kan opsamles ved et gennemløb. Et gennemløb kan ske på flere måder, men et rekursivt post-order gennemløb vil opsamle informationen i en rækkefølge der er belejlig for beregning af udtrykkets værdi. Hvis et udtryk er sammensat, vil et post-order gennemløb først gennemløbe de to deludtryk (rekursivt) og til sidst læse operatoren.




```
evaluate(v):
    if v is a leaf:
         return number stored at v
    else
         x = evaluate(left subexpression stored at v)
         y = evaluate(right subexpression stored at v)
         return x o y  (where o is operator stored at v)
```

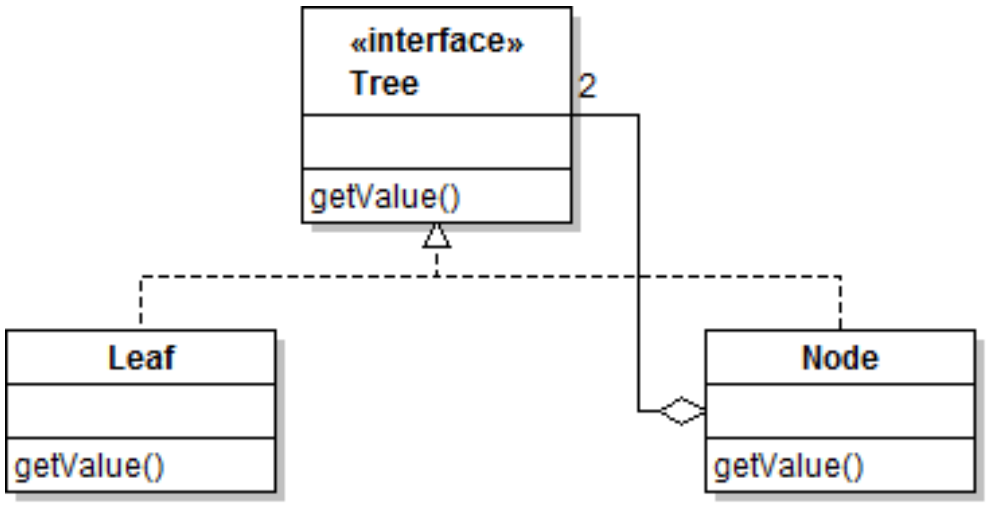
I Java kan et sådant rekursivt gennemløb repræsenteres på flere måder. På klassisk vis kan man lave en enkelt rekursiv metode, som svarer ret nøje til pseudokoden.

```
public int evaluate(Tree w) {
    int answer;
    if ( w instanceof Leaf )
        answer = ((Leaf)w).getValue();
    else {
        Tree l = ((Node)w).getLeft();
        Tree r = ((Node)w).getRight();
        Operator o = ((Node)w).getOp();
        int left_answer = evaluate(l);
        int right_answer = evaluate(r);
        answer = o.apply(left_answer,right_answer);
    }
    return answer;
}
```

Java-koden er omstændelig, specielt fordi det er nødvendigt at teste om et `Tree` objekt faktisk er et blad eller en indre træknude, og derefter lave casts afhængig af testets udfald. Casts er påtvunget af Java-sproget, men

`instanceof`-testet sikrer at den rekursive metode kan standse fremfor at kalde sig selv i en uendelighed, og dette test kan synes uomgængeligt. Men det er en forkert betragtning. Ved at forlægge evalueringen til metoder i de tre klasser kan man undgå casts og test.

Interfacet tilføjes en metode `getValue`, som klasserne skal implementere. På denne måde kan metodekaldet `evaluate(t)` erstattes af kaldet `t.getValue()` for `t` af type `Tree`.




```
public interface Tree {
    public int getValue() ;
}

public class Leaf implements Tree {
    ...
    public int getValue() { return value; }
}

public class Node implements Tree {
    ...
    public int getValue() {
        return op.apply(left.getValue(),right.getValue());
    }
}
```

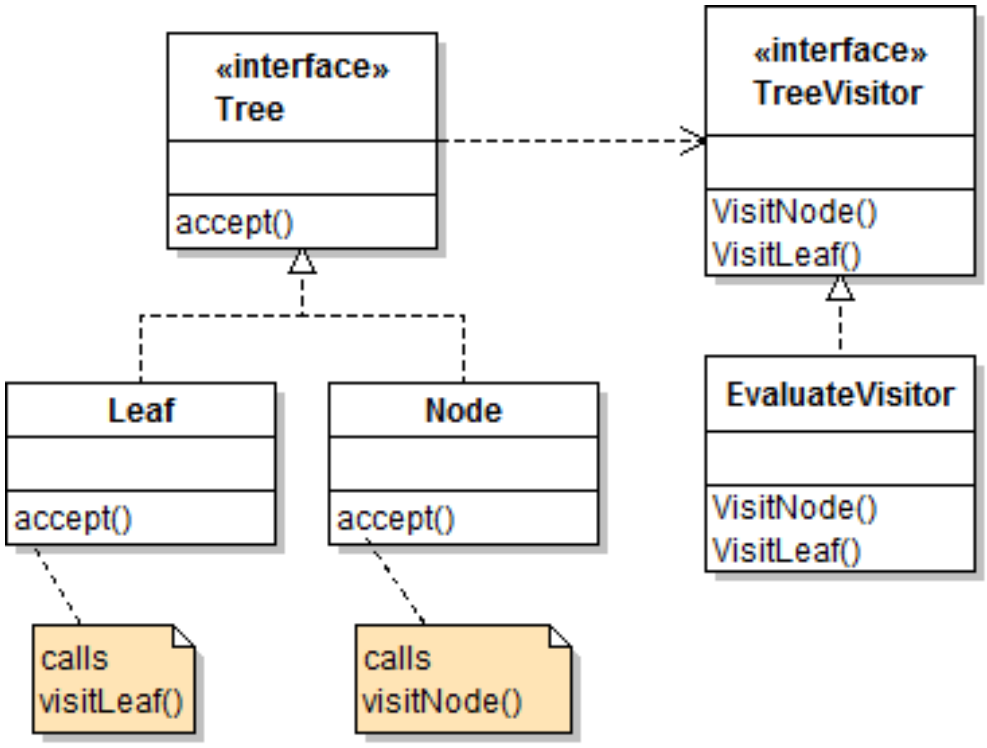
Den resulterende javakode er simplere, men der er en ny ulempe. Det har været nødvendigt at ændre de oprindelige tre klasser. Hvis man ønsker at gennemløbe et udtrykstræ med et nyt formål, f.eks. udskrift af udtrykket, så

vil det være nødvendigt at tilføje klasserne en metode skræddersyet til netop udskrift, og dermed igen ændre de oprindelige klasser.

## 3.2 Visitor

Visitor designmønstret undgår at føje en ekstra metode til alle klasser i hierarkiet for hver type gennemløb. Metoderne samles i stedet i en ny klasse ifølge en skabelon der er angivet i et `TreeVisitor`-interface. Derudover skal der en gang for alle tilføjes en enkelt metode `accept` til hver klasse i hierarkiet. Disse `accept`-metoder sørger for at lave call-back til de anvendelsesspecifikke metoder samlet i den `TreeVisitor`-implementerende klasse.

For beregning af et udtryks værdi benyttes følgende struktur:




De generelle klasser/interface:

```
public interface TreeVisitor<T> {
    public T visitLeaf(Leaf l);

    public T visitNode(Node n);
}
```

```
public interface Tree {
    public <T> T accept(TreeVisitor<T> v) ;

    public int getValue() ;
}

public class Leaf implements Tree {
    ...
    public <T> T accept(TreeVisitor<T> v) {
        return v.visitLeaf(this);
    }
}

public class Node implements Tree {
    ...
    public <T> T accept(TreeVisitor<T> v) {
        return v.visitNode(this);
    }
}
```

Den anvendelsesspecifikke klasse:

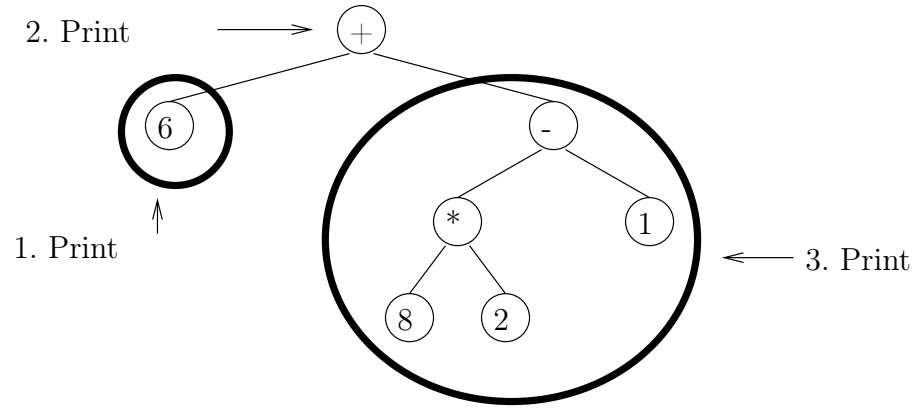
```
public class EvaluateVisitor implements TreeVisitor<Integer> {

    public Integer visitLeaf(Leaf l) { return l.getValue(); }

    public Integer visitNode(Node n) {
        int l = n.getLeft().accept(this);
        int r = n.getRight().accept(this);
        return n.getOp().apply(l,r);
    }
}
```

F.eks. kan værdien af udtrykket `t` udskrives med
`System.out.println(t.accept(new EvaluateVisitor()))`.

For at lave en anden type gennemløb, er det tilstrækkeligt at lave en ny implementation af `TreeVisitor`. Som eksempel betragtes udskrift af selve udtrykket. Det foregår naturligt ved et rekursivt in-order gennemløb, dvs. først udskrives venstre deludtryk, dernæst operatoren, og til sidst højre deludtryk.

```
text(v):
  if v is a leaf:
    return number
  else
    return "("
         + text( left subexpression)
         + operator
         + text( right subexpression )
         + ")"
```

( | 6 | + | ((8 * 2) - 1) | )



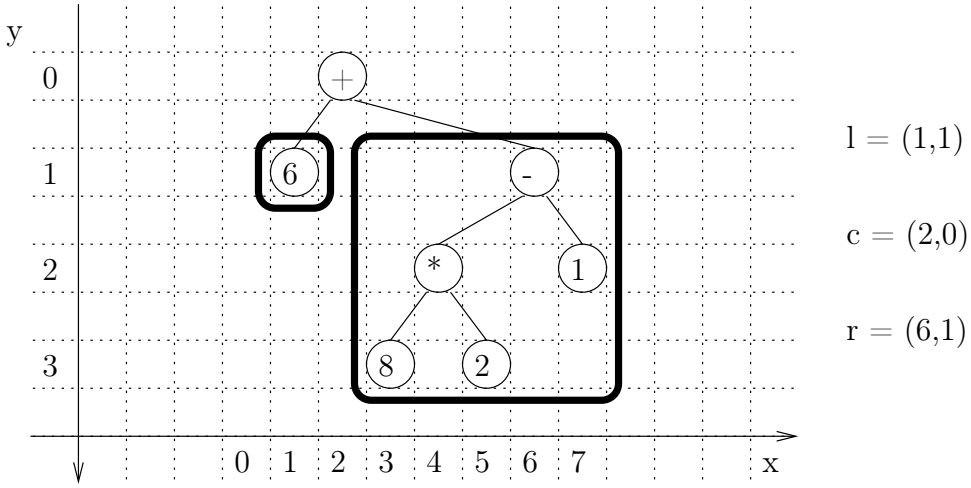

Bemærk at udtrykket ikke udskrives direkte, men den relevante tekst beregnes (uden sideeffekter) og returneres. In-order gennemløbet kan i Java specificeres af følgende klasse

```
public class PrintVisitor implements TreeVisitor<String> {

    public String visitLeaf(Leaf l) {
        return new Integer(l.getValue()).toString();
    }

    public String visitNode(Node n) {
        return ("(" + n.getLeft().accept(this)
                + n.getOp() + n.getRight().accept(this) + ")");
    }
}
```

Udtrykket `t` kan udskrives med sætningen

`System.out.println(t.accept(new PrintVisitor())).`

I stedet for at skrive udtrykket som tekst, kunne man ønske at tegne det grafisk som et træ




Ved et in-order gennemløb kan man tegne træet fra venstre mod højre, således at hvert symbol (operator eller tal), der udskrives, tegnes en enhed længere mod højre end det forrige, samtidigt skal roden i et deludtryk tegnes en enhed længere nede end udtrykkets operator.

Hvis stregerne, der repræsenterer grenene udelades kan tallene og operatorerne tegnes korrekt af følgende pseudokode:

```
state : a current drawing position (x,y)
        initially (x,y) = (0,0)

drawSymbol(s):
  increment x and draw s;

draw(v):
  if v is a leaf:
    drawSymbol( number );
  else
    increment y;
    draw( left subexpression );
    decrement y;
    drawSymbol( operator );
    increment y;
    draw( right subexpression );
```

```
    decrement y;
```

For at tilføje gren-stregerne til tegningen er det nødvendigt at gemme koordinaterne for en linjes første endepunkt indtil positionen af linjens andet endepunkt er kendt (og linjen kan tegnes). I den modificerede pseudokode returnerer tegnemetoderne positionerne for det symbol (roden af det udtryk) de netop har tegnet.

```
state : a current drawing position (x,y)
        initially (x,y) = (0,0)

drawSymbol(s): // returns position where s is drawn
  increment x and draw s;
  return (x,y)

draw(v): // returns where root of expression is drawn
  if v is a leaf:
    return drawSymbol( number );
  else
    increment y;
    l = draw( left subexpression );
    decrement y;
    c = drawSymbol( operator );
    draw line from l to c;
    increment y;
    r = draw( right subexpression );
    decrement y;
    draw line from r to c;
    return c;
```

På basis af den udvidede pseudokode laves en klasse `DrawVisitor`, der implementerer `TreeVisitor` og træet `t` kan så vises grafisk med sætningen `new DrawVisitor(t)`.

```
public class DrawVisitor extends JComponent
                         implements TreeVisitor<Point> {

    private static final int UNIT = 30;
    private Tree t;
    private Point pen_pos;
    private Graphics2D g;

    public DrawVisitor(Tree t) {
```

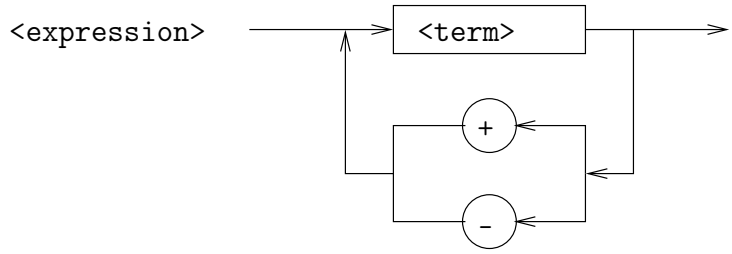
```
        this.t = t;
        JFrame f = new JFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(this);
        f.setSize(400,400);
        f.setVisible(true);
    }

    private Point drawSymbol(Object ob) {
        pen_pos.x += UNIT;
        g.drawString(ob.toString(),pen_pos.x,pen_pos.y-4);
        return (Point) pen_pos.clone();
    }

    public Point visitLeaf(Leaf l) {
        return drawSymbol( new Integer(l.getValue()) );
    }

    public Point visitNode(Node n) {
        pen_pos.y += UNIT;
        Point left_pos = n.getLeft().accept(this);
        pen_pos.y -= UNIT;
        Point node = drawSymbol(n.getOp());
        g.draw(new Line2D.Double(left_pos,node));
        pen_pos.y += UNIT;
        Point right_pos = n.getRight().accept(this);
        pen_pos.y -= UNIT;
        g.draw(new Line2D.Double(right_pos,node));
        return node;
    }

    public void paintComponent(Graphics g) {
        this.g = (Graphics2D)g;
        pen_pos = new Point(UNIT,UNIT);
        t.accept(this);
    }
}
```

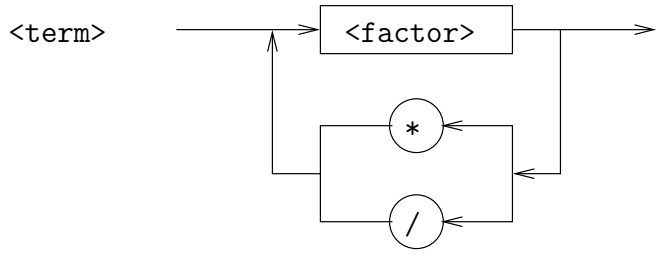
## 3.3 Fra tekst til rekursiv struktur: gensidigt rekursive metoder

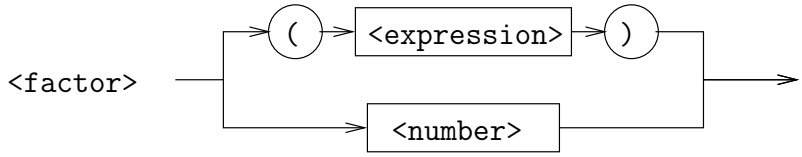
Som et simpelt men realistisk eksempel på gensidig rekursion vil vi se hvordan man kan opbygge den rekursive struktur for et aritmetisk udtryk udfra en tekst, der beskriver udtrykket. F.eks. følgende

$$3 + 4 * 5$$
$$(3 + 4) * 5$$
$$(1 - (2 - (3 - (4 - 5))))$$

Tolkningen af teksten kompliceres af at `*` og `/` binder tættere end `+` og `-` ligesom man kan gruppere deludtryk med parenteser. Men dette indfanges af de følgende syntax diagrammer, som samtidig afgrænser hvilke tekster, vore metoder vil være i stand til at tolke som aritmetiske udtryk:

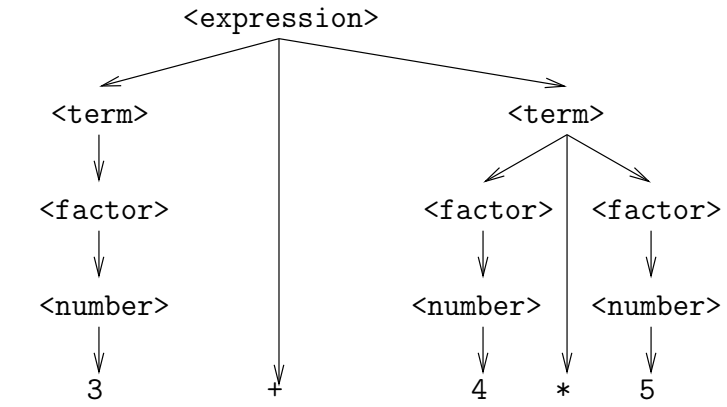


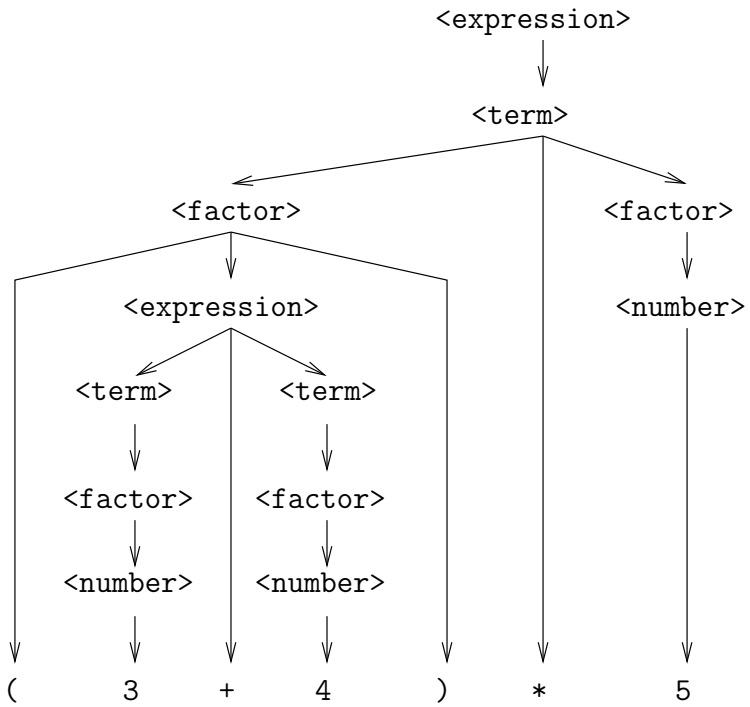




Ved at anvende syntaxdiagrammerne vil man se at `<expression>` kan danne `<term>+<term>`, hvor den første `<term>` via `<factor>` genererer tallet `3`, mens den anden `<term>` først danner `<factor>*<factor>` og derefter danner de to `<factor>` henholdsvis tallene `4` og `5`.

Denne derivation af tekststrengen `3+4*5` kan opskrives i et syntaxtræ.




Det er her tillige gjort for `(3+4)*5`.




På baggrund af syntaxdiagrammerne er der naturligt at skrive 3 gensidigt rekursive metoder `parseExpression`, `parseTerm`, `parseFactor`, der kan konstruere `Tree` objekter, der repræsenterer henholdsvis et udtryk, en term og en faktor.

Da hele udtrykket bliver leveret som en tekst er det praktisk at have en hjælpeklasse `ExpressionTokenizer` til at levere leksikalske symboler et ad gangen, hvor et leksikalsk symbol er en tekst bestående af cifre, eller en

af "+", -", "*", /", "(", ")". En ExpressionTokenizer har to metoder nextToken() og peekToken(), som begge returnerer det næste leksikalske symbol. Forskellen mellem de to metoder er at nextToken() er en mutator der skifter ExpressionTokenizer objektets tilstand et (leksikalsk) symbol frem, mens peekToken() efterlader objektets tilstand uændret. Detaljerne i implementationen af klassen kan findes i eksempelkataloget.

De 3 gensidigt rekursive metoder defineres i klassen ExpressionParser:

```
public class ExpressionParser {

   public ExpressionParser(String anExpression)  {
      tokenizer = new ExpressionTokenizer(anExpression);
   }

    public Tree parseExpression() { ... }

    public Tree parseTerm() { ... }

    public Tree parseFactor() { ... }

    private ExpressionTokenizer tokenizer;
}
```

Metoden til opbygning af en Expression indeholder i overensstemmelse med syntaxdigrammet et ubetinget kald af metoden til opbygning af en Term

```
   public Tree parseExpression() {
      Tree t = parseTerm();
      while ("+".equals(tokenizer.peekToken())
             || "-".equals(tokenizer.peekToken())) {
         Operator op = Operator.parseOp(tokenizer.nextToken());
         Tree t2 = parseTerm();
         t = new Node(t,op,t2);
      }
      return t;
   }
```

Metoden til opbygning af en Term indeholder i overensstemmelse med syntaxdigrammet et ubetinget kald af metoden til opbygning af en Factor

```
   public Tree parseTerm() {
      Tree t = parseFactor();
      while ("*".equals(tokenizer.peekToken())
```

```
            || "/".equals(tokenizer.peekToken())) {
         Operator op = Operator.parseOp(tokenizer.nextToken());
         Tree t2 = parseFactor();
         t = new Node(t,op,t2);
      }
      return t;
   }
```

Metoden til opbygning af en `Factor` har en rekursionsbetingelse der sikrer at et rekursivt kald af metoden til opbygning af en `Expression` kun foretages under visse betingelser, så der er mulighed for at rekursionen kan stoppe

```
   public Tree parseFactor() {
      Tree t;
      if ("(".equals(tokenizer.peekToken())) {
         tokenizer.nextToken();
         t = parseExpression();
         tokenizer.nextToken(); // read ")"
      } else t = new Leaf(Integer.parseInt(tokenizer.nextToken()));
      return t;
   }
```

Vi behøver nu ikke anvende den omstændelige skrivemåde

```
Tree t =
  new Node(
    new Leaf(6),
    Operator.PLUS,
    new Node(new Node(
      new Leaf(8),Operator.MULT,new Leaf(2)),
      Operator.MINUS,
      new Leaf(1)));
```

men kan benytte den kortere

```
Tree t =  new ExpressionParser("6+(8*2-1)").parseExpression();
```

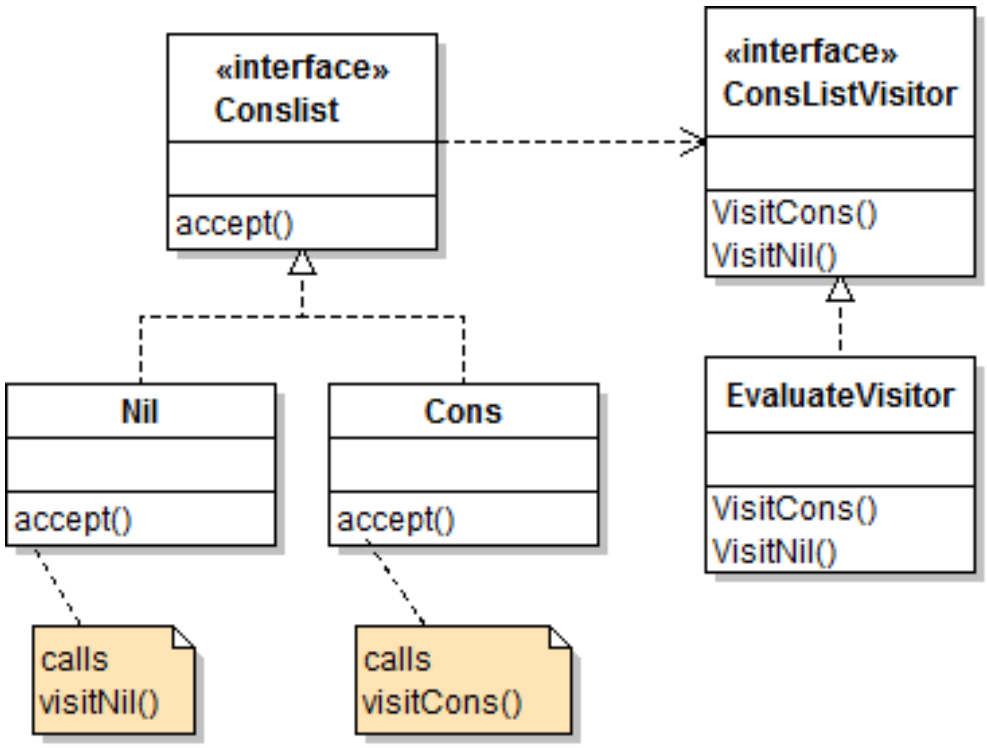
## 3.4 Rekursiv liste

Der er tradition for at bruge typenavnet `ConsList` for en rekursiv liste. Den kan enten være tom (`Nil`) eller være sammensat (`Cons`), dvs. bestå af et element efterfulgt af en (mindre) liste. En sådan liste kan modelleres ved tre klasser:

```
public interface ConsList {}

public class Nil implements ConsList {}

public class Cons implements ConsList {
    private Object hd;
    private ConsList tl;

    public Cons(Object x, ConsList y) { hd = x; tl = y; }
}
```

Listen `["Easter","Xmas"]` kan f.eks. repræsenteres som
`new Cons("Easter",new Cons("Xmas",new Nil()))`.

Et gennemløb af en liste kan ske ved brug af visitormønstret i analogi med et trægennemløb. Som eksempel betragtes visitormønstret anvendt til lineær søgning, hvor de anvendelsesspecifikke metoder er samlet i klassen `SearchVisitor`:




```
public interface ConsList {
    public <T> T accept(ConsListVisitor<T> v);
}
```

```
public class Nil implements ConsList {
    public <T> T accept(ConsListVisitor<T> v)
    { return v.visitNil(this); }
}

public class Cons implements ConsList {
    private Object hd;
    private ConsList tl;
    public Cons(Object x, ConsList y) { hd = x; tl = y; }
    public Object head() { return hd; }
    public ConsList tail() { return tl; }
    public <T> T accept(ConsListVisitor<T> v)
      { return v.visitCons(this); }
}

public interface ConsListVisitor<T> {
  public T visitNil(Nil n);
  public T visitCons(Cons c);
}

public class SearchVisitor implements ConsListVisitor<Boolean> {
  private Object key;

  public SearchVisitor(Object k) { key = k; }

  public Boolean visitNil(Nil n) { return false; }

  public Boolean visitCons(Cons c) {
    return  c.head().equals(key)  ||  c.tail().accept(this);
  }
}
```

Søgning i den rekursive liste `l` efter teksten `"Xmas"` kan ske med sætningen `l.accept(new SearchVisitor("Xmas"))`.

## 3.5 Sammenligning af gennemløbsteknikker

Hvis det vides at den rekursive struktur er statisk (ingen nye klasser der implementerer interfacet vil blive tilføjet) så er visitormønstret en fleksibel

teknik, der tillader nye slags gennemløb uden at den eksisterende programkode behøver ændres.

Hvis det modsat vides at der ikke vil opstå behov for nye slags gennemløb, så er lokale rekursive metoder en god teknik (eksemplificeret ved `getValue{}` metoderne), der tillader tilføjelse af nye klasser der implementerer interfacet uden at den eksisterende programkode behøver ændres.

# 4 Algoritmer over "reelle" tal

Denne note består af tre afsnit. Første afsnit omhandler de basale lovmæssigheder – eller snarere mangel på samme – bag datamaters approximationer til reelle tal og den dertil hørende aritmetik. I andet afsnit beskrives Javas indbyggede `double`-type. Tredje afsnit indeholder et antal eksempler, der illustrerer nogle af de oftest forekommende problemer med denne såkaldte *flydende-komma-aritmetik*. Eksemplerne er konstrueret med henblik på at illustrere nogle algoritmer, hvor de underliggende primitiver (især addition) ikke har de egenskaber, man normalt ville forvente. En egentlig fremstilling af, hvordan man faktisk håndterer disse problemer, dvs. konstruerer gode numeriske algoritmer, hører til emneområdet *numerisk analyse* og er uden for rammen af denne note.

Første afsnit er et redigeret uddrag af en note skrevet af Ole Østerby, mens sidste afsnit er en redigeret "oversættelse" fra Trine til Java af en note skrevet af Erik Meineche Schmidt.

## 4.1 Flydende komma aritmetik

### Flydende-komma-tal

Der er som bekendt uendeligt mange reelle tal, men en sædvanlig computer (eller regnemaskine) kan kun håndtere en endelig del af disse direkte i hardware. Disse *maskintal* udgør normalt et *flydende-komma-tal-system*.

Et flydende-komma-tal-system er karakteriseret ved

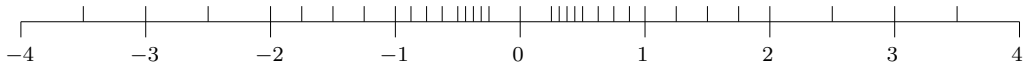
| | |
|---|---|
| en *base* eller et *grundtal* | $\beta \in \mathcal{N} \backslash \{1\}$ |
| et antal cifre | $s \in \mathcal{N}$ |
| en mindste eksponent | $m \in \mathcal{Z}$ |
| en største eksponent | $M \in \mathcal{Z}$ |

Hvert flydende-komma-tal har formen

$$y = \pm d_1.d_2 \ldots d_s \cdot \beta^e,$$

hvor exponenten $e$ ligger i intervallet $m \leq e \leq M$. Tallet er normaliseret, så det første ciffer $d_1$ opfylder $1 \leq d_1 \leq \beta - 1$. De resterende cifre opfylder naturligvis $0 \leq d_k \leq \beta - 1$.

Tallet $0 = 0.0 \ldots 0 \cdot \beta^e$ er desuden med i systemet.




Figur 1: $F(2, 3, -2, 1)$

Tallet $d_1.d_2 \ldots d_s$ fortolkes som $d_1 + d_2 \cdot \beta^{-1} + \cdots + d_s \cdot \beta^{-s+1}$ og kaldes *mantissen* eller *taldelen*, mens $\beta^e$ kaldes *eksponentdelen*.

Et flydende-tal-system med parametrene $\beta, s, m$ og $M$ betegnes $F(\beta, s, m, M)$.

Figur 1 viser tallene i $F(2, 3, -2, 1)$, hvilket svarer til, hvad man kan repræsentere med 6 bits. Bemærk hvordan afstanden mellem flydende tal varierer hen over intervallet. Bemærk endvidere det relativt store interval fra 0 til det mindste positive flydende tal $\beta^m$.

På en sædvanlig datamat er det nærliggende at vælge $\beta = 2$, hvorefter de $s$ cifre kan lagres i $s$ bits. Hertil kommer en bit til fortegn. Eksponenten $e$ kan lagres som et heltal, f. eks. i intervallet $[-2^t, +2^t - 1]$, hvilket kræver $t + 1$ bits, d.v.s. at tallet ialt kræver $s+t+2$ bits. For $\beta = 2$ sikrer normaliseringen at $d_1 = 1$, og denne redundante information kan udelades, så kun $s + t + 1$ bits anvendes.

### Beregningsfejl

Ved aritmetik på en endelig mængde af tal vil der opstå en repræsentationsfejl, når resultatet ikke er repræsenterbart.

Vi vil give eksempler på repræsentationsfejl i det fiktive system $F(10, 4, -99, 99)$, dvs. et sædvanligt titalssystem, hvor tal skrives med 4 betydende cifre. Eksponentgrænserne er valgt så store $(-99 \ldots 99)$, at de ikke giver restriktioner i praksis.

Tallene 1.573 og 0.1824 er således gyldige maskintal, mens

$$\begin{array}{lcll} 1.573 & + & 0.1824 & = & 1.7554 \\ 1.573 & - & 0.1824 & = & 1.3906 \\ 1.573 & \times & 0.1824 & = & 0.2869152 \\ 1.573 & / & 0.1824 & = & 8.6239035\ldots \end{array}$$

ikke kan repræsenteres i $F(10, 4, -99, 99)$. Vi må konstatere, at *mængden af maskintal er ikke lukket over for de fire regningsarter.*

Hvorledes foregår aritmetik i så fald på maskintallene? Vi vil antage, at der til en mængde af maskintal $\mathcal{M}$ – i vores tilfælde $F(10, 4, -99, 99)$ – er tilknyttet en funktion $fl : \mathcal{R} \mapsto \mathcal{M}$, som til ethvert reelt tal tilknytter et (nærtliggende) repræsentenbart tal, f.eks. ved afrunding eller afskæring af cifre. Hvis vi betegner de aritmetiske operationer på maskintallene med symboler $\oplus, \ominus, \otimes, \oslash$, så vil vi antage at $x \oplus y = fl(x + y)$, og tilsvarende for de øvrige operationer. Denne antagelse er ganske realistisk. De fleste datamater har interne regneregistre af længde $2s$. De kan således internt lagre et produkt af to $s$–cifrede tal, eller en sum af to tal, hvis eksponenter afviger med højst $s$. Men også i de tilfælde – typisk ved division – hvor et eksakt mellemresultat ikke kan repræsenteres, har vi tilstrækkelig information til at kunne finde $fl(a/b)$ korrekt.

For at illustrere repræsentationsfejl, må vi definere $fl$ for eksempelsystemet $F(10, 4, -99, 99)$. Vi vælger simpel afskæring, dvs. $fl(.213599) = fl(.213501) = .2135$, og de aritmetiske operationer i systemet betegnes med symboler $\oplus, \ominus, \otimes, \oslash$. Det tidligere eksempel kan nu elaboreres:

$$
\begin{array}{lllllllllll}
1.573 & \oplus & 0.1824 & = & fl(1.573 & + & 0.1824) & = & fl(1.7554) & = & 1.755 \\
1.573 & \ominus & 0.1824 & = & fl(1.573 & - & 0.1824) & = & fl(1.3906) & = & 1.390 \\
1.573 & \otimes & 0.1824 & = & fl(1.573 & \times & 0.1824) & = & fl(0.2869152) & = & .2869 \\
1.573 & \oslash & 0.1824 & = & fl(1.573 & / & 0.1824) & = & fl(8.6239035\ldots) & = & 8.623
\end{array}
$$

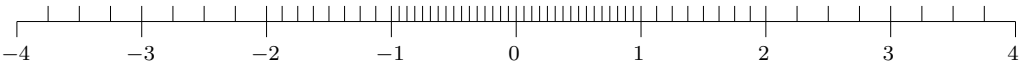
Repræsentationsfejlen ved en enkelt operation er lille, men blot man sammensætter to operationer kan fejlen blive stor:

$$
\begin{array}{rcll}
(1.418 \oplus 2937) \ominus 2936 & = & 2938 \ominus 2936 & = 2.000 \\
1.418 \oplus (2937 \ominus 2936) & = & 1.418 \oplus 1.000 & = 2.418 \\
1.418 \otimes (2001 \ominus 2000) & = & 1.418 \otimes 1.000 & = 1.418 \\
(1.418 \otimes 2001) \ominus (1.418 \otimes 2000) & = & 2837 \ominus 2836 & = 1.000
\end{array}
$$

Vi observerer to af de væsentligste konsekvenser af, at maskintallene ikke er lukkede over for de sædvanlige regningsarter, nemlig at *den associative og den distributive lov ikke holder for maskintal*.

### IEEE Standard for Binary Floating Point Arithmetic

I 1985 udkom på initiativ af IEEE (Institute of Electrical and Electronics Engineers, USA) en standard for binære flydende tal, og denne standard




Figur 2: $F(2, 4, -1, 1)^*$

efterleves i store træk af de fleste moderne processorer. IEEE omfatter to formater, som i vor notation (nogenlunde) svarer til

| | |
|---|---|
| Single precision | $F(2, 24, -126, 127)$ |
| Double precision | $F(2, 53, -1022, 1023)$ |

IEEE standarden foreskriver desuden korrekt afrunding med den ekstra finesse, at hvis $x$ ligger midt imellem to maskintal, så afrundes til det tal, hvis mindst betydende bit er 0. Standarden kræver imidlertid også tre muligheder for retningsbestemt afrunding nemlig mod 0, mod $+\infty$ og mod $-\infty$.

En optælling af antal bits giver henh. 33 og 65 bits til de to repræsentationer, hvoraf vi kan slutte, at $d_1$ ikke forudsættes lagret eksplicit.

Det ses også, at to mulige værdier for eksponenten er blevet udtaget til specielle formål. Den høje værdi er blevet reserveret til repræsentationer for overløb $\pm\infty$ og `NaN` (= Not a Number, f. eks. $0/0$ eller $\infty - \infty$). Der er endvidere indført regneregler for disse generaliserede tal, således at et program ikke behøver stoppe p.g.a. en division med 0 eller anden form for overløb.

Endvidere råder man også delvis bod på det "store hul" mellem 0 og $\beta^m$ ved at tillade mantissen ved netop denne eksponent at være unormaliseret.

Figur 2 viser tallene i et IEEE–lignende talsystem $F(2, 4, -1, 1)^*$, hvor den ledende bit i et normaliseret tal er underforstået og hvor den mindste eksponent er reserveret unormaliserede tal i nærheden af 0. Vi har dog *ikke* friholdt den største eksponent til specielle formål. Som i den tidligere figur svarer talsystemet til, hvad man kan repræsentere med 6 bits.

## 4.2 "Reel" aritmetik i Java

Java benytter IEEE Standarden (*IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*). Det betyder at de repræsenterede værdier af typen `double` er på formen

$$\pm b_1.b_2 \ldots b_{53} \cdot 2^e, \qquad \text{hvor } b_i \in \{0, 1\} \text{ og } -1022 \leq e \leq 1023$$

$\pm\infty$ og `NaN`

Java double aritmetik er således tilnærmelsesvis identisk med talsystemet $F(2, 53, -1022, 1023)$.

Der gælder desuden

- Hvis den underliggende computer tilbyder et andet system, er det i et vist omfang tilladt Java fortolkeren at anvende dette. Hvis man vil være helt sikkert på at der benyttes IEEE Standard 754 for 64 bits flydende tal ved repræsentation af `double`-værdier, kan man anvende modifier `strictfp` på den omgivende metode eller klasse.
- $\pm\infty$ og `NaN` benævnes i Java syntax: `Double.NEGATIVE_INFINITY`, `Double.POSITIVE_INFINITY` og `Double.NaN`

## 4.3 "Reelle" algoritmer

I dette afsnit betragtes fire eksempler på numeriske beregninger, som hver på sin måde illustrerer regning med flydende komma tal. Vi præsenterer eksemplerne i to forskellige endelige systemer, dels det decimale legetøjssystem $F(10, 4, -99, 99)$, hvor der approximeres ved afskæring, og dels i JAVA's `double` system $F(2, 53, -1022, 1023)$. Endvidere vil vi bruge notationen $F(\infty)$ for de "rigtige" reelle tal.

### Summationsrækkefølge

Som det første eksempel på et af standardproblemerne i forbindelse med endelige talsystemer skal vi betragte summation af den *harmoniske række* dvs. rækken

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \cdots$$

Det vides fra matematikken, at talfølgen $H_1, H_2, \ldots, H_n, \ldots$, hvor

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

er divergent, at den divergerer meget langsomt, samt at $H_n$ næsten er lig med den naturlige logaritme. Mere præcist gælder der, at

$$\lim_{n\to\infty} (H_n - ln(n)) = \gamma$$

hvor $\gamma = 0.577215664901533$ er den såkaldte *Eulers konstant*.

Det er klart, at $H_n$ kan udregnes på flere forskellige måder, vi skal se på hhv. *forlæns*, *baglæns* og *balanceret* summation. Hvis rækken udregnes forfra, må der forventes ciffertab, når et lille led adderes til et stort led efter forskriften

$$H_n = H_{n-1} + \frac{1}{n}$$

Hvis den derimod summeres bagfra, er det i højere grad led af nogenlunde samme størrelsesorden, der adderes, hvorfor man ville forvente et bedre resultat. Endelig vil man forvente et endnu bedre resultat, hvis alle additioner sker mellem to næsten lige store tal. I tilfældet med den harmoniske række og lille $n$ kan man tilnærmelsesvis opnå dette ved rekursivt at beregne første og anden halvdel af summen for sig og til sidst addere de to dele.

JAVA metoden `harmonic` anvender alle tre principper.

```
public static void harmonic() {
  for (int n=10; n<=10000000; n=n*10) {
    double fsum = 0;
    double bsum = 0;
    for (int j=1; j<=n; j=j+1) {
      fsum = fsum + 1.0/j;
      bsum = bsum + 1.0/(n-j+1);
    }
    double rsum = recursiveHarmonic(1,n);
    System.out.println("\nn= " + n +
                       "\nf: " + (fsum-Math.log(n)) +
                       "\nb: " + (bsum-Math.log(n)) +
                       "\nr: " + (rsum-Math.log(n)));
  }
}
```

hvor

```
private static double recursiveHarmonic(int first, int last) {
  if (first==last) return 1.0/first;
  int med = (first+last)/2;
  return recursiveHarmonic(first,med)
       + recursiveHarmonic(med+1,last);
}
```

Udførsel giver:

```
n= 10
f: 0.6263831609742079
```

```
b: 0.6263831609742079
r: 0.6263831609742079

n= 100
f: 0.5822073316515288
b: 0.5822073316515297
r: 0.5822073316515279

n= 1000
f: 0.5777155815682065
b: 0.5777155815682038
r: 0.5777155815682065

n= 10000
f: 0.5772656640681646
b: 0.5772656640682019
r: 0.5772656640681983

n= 100000
f: 0.5772206648931064
b: 0.5772206648931792
r: 0.5772206648932023

n= 1000000
f: 0.5772161649007153
b: 0.5772161649014986
r: 0.5772161649014489

n= 10000000
f: 0.5772157148989514
b: 0.5772157149016444
r: 0.5772157149015342
```

Når metoden som vist udføres i Java skal man have ret store $n$-værdier for at se forskel på de 3 summationsrækkefølger, men hvis man udfører metoden med $F(10, 4, -99, 99)$-aritmetik bliver forskellen meget tydelig, idet kun den balancerede (rekursive) teknik giver tilnærmelsesvist korrekt resultat for de anvendte $n$-værdier:

```
n= 10
f: 0.6250
b: 0.6260
r: 0.6260

n= 100
f: 0.5370
b: 0.5650
r: 0.5780
```

```
n= 1000
f: 0.1620
b: 0.3770
r: 0.5700

n= 10000
f: -2.141
b: -1.141
r: 0.5680

n= 100000
f: -4.441
b: -3.441
r: 0.5600

n= 1000000
f: -6.741
b: -5.741
r: 0.5500
```

Vi kan konkludere, at når summer udregnes, bør man tilstræbe at additioner kun foregår mellem delresultater der er næsten lige store. Det er bedre at summere leddene i voksende orden end i aftagende orden.

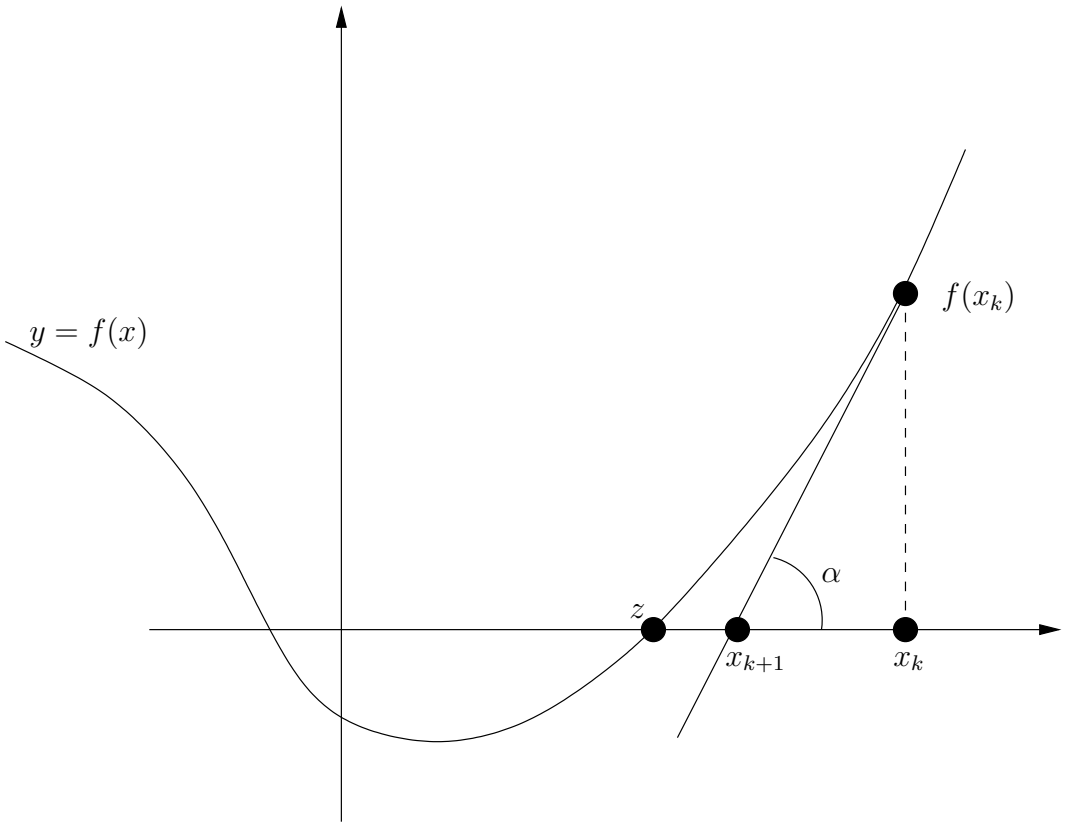
### Newton Iteration

De næste eksempler handler om iterative processer, som er konvergente med eksakt aritmetik i $F(\infty)$, men hvis konvergensegenskaber i endelige flydende komma talsystemer er mere problematiske.

Som første eksempel betragter vi kvadratrodsuddragning v.hj.a. den såkaldte *Newton-iteration*. Figur 3 viser grafen $y = f(x)$ for en funktion, som har et nulpunkt $z$, dvs. hvor $f(z) = 0$.

En iterativ metode til beregning af et sådant nulpunkt er en metode, hvor $z$ beregnes som grænseværdien af en følge af formen $x_0, x_1, \ldots, x_k, x_{k+1}, \ldots$ hvor $x_{k+1}$ beregnes ud fra en eller flere af de foregående $x$-værdier. I Newton-iterationen beregnes $x_{k+1}$ som det punkt, hvor $x$-aksen skæres af *tangenten* til kurven $y = f(x)$ i punktet $(x_k, f(x_k))$. Sammenhængen mellem $x_k$ og $x_{k+1}$ er som følger

$$\tan(\alpha) = \frac{f(x_k)}{x_k - x_{k+1}} = f'(x_k)$$




Figur 3: Newton iteration

hvoraf fås at

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \qquad (*)$$

Såfremt $z$ er et *simpelt* nulpunkt for funktionen $f$ (hvilket betyder at $f'(z) \neq 0$) og såfremt startværdien $x_0$ vælges passende, kan man bevise, at følgen $x_0, x_1, \ldots, x_k, \ldots$ er konvergent og at den konvergerer mod $z$, dvs. at

$$f(\lim_{k\to\infty} x_k) = 0$$

Vi kan bruge Newton-iteration til at finde kvadratroden af et positivt reelt tal $a$, idet $\sqrt{a}$ er det positive nulpunkt for funktionen

$$f(x) = x^2 - a$$

(*) får i dette tilfælde følgende udseende

$$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k} \qquad (**)$$

som med startværdien

$$x_0 = \frac{1}{2}(a+1)$$

kan vises at konvergere *monotont* mod $\sqrt{a}$, dvs. der gælder $x_0 > x_1 > \ldots > x_k > x_{k+1} > \ldots \to \sqrt{a}$.

Baseret på ovenstående kan vi nu beregne kvadratroden af et reelt tal $a > 0$ med en relativ nøjagtighed på $\epsilon$ v.hj.a. følgende algoritme for $F(\infty)$.

Algoritme: Kvadratrod i $F(\infty)$
Stimulans : $a, \epsilon \in F(\infty)$, $a > 0$, $1 > \epsilon > 0$
Respons : $r \in F(\infty) : 0 \leq r - \sqrt{a} \leq \epsilon\sqrt{a}$
Metode :

```
xn = (a+1)/2
xg = xn/(1-ϵ)+1
while (xg-xn > ϵ*xg) {
  xg = xn;
  xn = xg-(xg*xg-a)/xg/2;
}
r = xg;
```

hvor vi bruger to variable $xg$ og $xn$ til at indeholde hhv. $x_k$ og $x_{k+1}$ (initialiseringen af $xg$ skal blot sikre, at den kontrollerende betingelse i `while`-sætningen er opfyldt første gang). Algoritmen standser, når den relative forskel mellem $xg$ og $xn$ er mindre end $\epsilon$, dvs. når vi når et $k$, hvor

$$x_k - x_{k+1} \leq \epsilon x_k$$

Man kan vise at når algoritmen standser er $x_k$ en god approximation til $\sqrt{a}$, idet der "næsten" gælder at $x_k - \sqrt{a} \leq \epsilon\sqrt{a}$, hvis $\epsilon << 1$.

Det er klart, at da $\epsilon > 0$, er ovenstående algoritme meningsfuld som algoritme over de reelle tal. Det interessante er nu, at den også er meningsfuld for $\epsilon = 0$ over et endeligt system $F(\beta, s, m, M)$. Dette skyldes, at hvis $\epsilon$ vælges mindre end $\beta^{-s}$, så er kravet

$$xg - xn > \epsilon * xg$$

i $F(\beta, s, m, M)$ ækvivalent med

$$xg - xn > 0$$

fordi $\beta^{-s}$ er den mindste relative forskel talsystemet kan skelne. Følgende udgave af algoritmen vil derfor udregne kvadratroden af et vilkårligt positivt flydende komma tal med den største nøjagtighed, talsystemet tillader.

Algoritme: Kvadratrod i $F(\beta, s, m, M)$

Stimulans : $a : a > 0$
Respons : $r : r \approx \sqrt{a}$
Metode :

```
xn = (a+1)/2
xg = xn+1
while (xg-xn > 0) {
  xg = xn;
  xn = xg-(xg*xg-a)/xg/2;
}
r = xg;
```

Hvis algoritmen køres i $F(10, 4, -99, 99)$ med stimulans $a = 8.519$, fås følgende beregning

| $xg$ | $xn$ |
|---|---|
| 4.859 | 4.759 |
| 4.759 | 3.276 |
| 3.276 | 2.938 |
| 2.938 | 2.918 |
| 2.918 | 2.918 |

Følgende program er en JAVA version af kvadratrodsalgoritmen.

```
public static double newton(double a) {
  if (a>0) {
    double x0 = a+1;
    double xn = (a+1)/2;
    int k = 0;
    while (x0-xn>0) {
      x0 = xn;
      xn = x0 - (x0*x0-a)/x0/2;
      k = k+1;
    System.out.println(a +"    "+ k +"    "+ x0 +"    "+ xn);
    }
    return xn;
  } else return 0;
}
```

Det skulle være klart, at man også i denne sidste algoritme kunne medtage en relativ fejltolerance $\epsilon$ (som for at have mening så skulle være $> \beta^{-s}$). Det er også klart, at valget af et sådant $\epsilon$ vil påvirke antallet af iterationer

(lille $\epsilon$, mange iterationer og omvendt), og såfremt tidsforbruget er en kritisk faktor, kan en sådan tolerance bruges til at betale for hastighed med nøjagtighed. Hvis man imidlertid er villig til at betale for maksimalt pålidelige resultater, bør man så vidt muligt altid regne i fuld nøjagtighed, dvs. i tilfælde som ovenfor fortsætte iterationen indtil følgen af iterander ikke længere aftager. Begrundelsen herfor er, at en algoritme, der regner med fuld nøjagtighed, er *universel* i den forstand, at den er optimal uanset hvilket system $F(\beta, s, m, M)$ den bruger, dvs. uanset hvilken maskine den køres på. Såfremt man bruger relative fejltolerancer, hvis meningsfuldhed som nævnt afhænger af den konkrete maskine, risikerer man at stille sig tilfreds med mindre end en given maskine kan yde.

Det er imidlertid ikke altid muligt at konvertere en konvergent $F(\infty)$-algoritme til en konvergent $F(\beta, s, m, M)$-algoritme, der regner med fuld nøjagtighed. Det næste eksempel viser en algoritme, der konvergerer i $F(\infty)$ for ethvert valg af fejltolerance $\epsilon$.

### Fixpunkter

Vi betragter iterative processer af formen

$$\begin{aligned} x_0 &= \text{startværdi} \qquad (*) \\ x_{k+1} &= g(x_k) \end{aligned}$$

hvor $g$ er en differentiabel reel funktion. Såfremt $x_0$ vælges på en sådan måde at mængden af iterander er indeholdt i et interval, hvor $|g'(x)| < 1$, så er iterationen konvergent, og følgen $x_0, x_1, \ldots, x_k, \ldots$ konvergerer mod et *fixpunkt* for funktionen $g$, dvs. et punkt $x$ for hvilket $x = g(x)$. En sådan afbildning $g$ kaldes også en *kontraktion*.

Betragt nu 2. gradsligningen $3x^2 + 8x - 10 = 0$, som ved omskrivning er ækvivalent med

$$x = (10 - 3x^2)/8$$

Hvis vi sætter $g(x) = (10 - 3x^2)/8$, er spørgsmålet om at løse ligningen ækvivalent med at finde et fixpunkt for $g$, dvs. at vi med en passende valgt startværdi kan bruge iterationen (*) til at løse 2. gradsligningen.

Vi får følgende algoritme

Algoritme: Iterativ løsning af $x = g(x)$ i $F(\infty)$

Stimulans : $\epsilon : \epsilon \in F(\infty), \epsilon > 0$
Respons : $x : |x - g(x)| \leq \epsilon * |x|$
Metode : `xg = 0;`

```
xn = 0.92;
while (|xg-xn| > ϵ*|xn|) {
  xg = xn;
  xn = (10-3*xg*xg)/8;
}
x = xn;
```

Hvis algoritmen udføres med $\epsilon = 10^{-7}$, giver den som resultat $x = 0.9274433$.

Den tilsvarende "fuld-nøjagtigheds"-algoritme ser ud som følger

Algoritme: Iterativ løsning af $x = g(x)$ i $F(\beta, s, m, M)$

Stimulans :
Respons : $x : x \approx g(x)$
Metode :
```
xg = 0;
xn = 0.92;
while (|xg-xn| > 0) {
  xg = xn;
  xn = (10-3*xg*xg)/8;
}
x = xn;
```

Hvis den udføres i $F(10, 4, -99, 99)$, giver den følgende resultat

| $xg$ | $xn$ |
|---|---|
| 0 | 0.9200 |
| 0.9200 | 0.9326 |
| 0.9326 | 0.9238 |
| · | 0.9300 |
| · | 0.9257 |
| · | 0.9287 |
| · | 0.9266 |
| · | 0.9281 |
| · | 0.9271 |
| · | 0.9277 |
| 0.9277 | 0.9273 |
| 0.9273 | 0.9276 |
| 0.9276 | 0.9273 |
| 0.9273 | 0.9276 |
| ⋮ | ⋮ |

dvs. beregningen går i en uendelig løkke. Problemet med algoritmen er den kontrollerende betingelse $|xn - xg| > 0$, som kræver at to på hinanden føl-

gende iterander skal være *ens*, og som derfor ikke tager højde for de afskæringsfejl, der opstår i det endelige talsystem.

Bemærk, at situationen var en anden i Newton-iterationen, hvor vi ikke krævede, at to iterander var ens, men derimod at en følges monotoni-egenskab blev brudt. Det kan vi ikke gøre her, fordi der er tale om alternerende konvergens, dvs. følgen af iterander skiftevis vokser og aftager. Såfremt en sådan alternerende beregning skal udføres i et endeligt talsystem, må der anvendes en passende relativ fejltolerance, dvs. en tolerance, som er stor nok til at beregningen standser og lille nok til at resultatet er brugbart. Det er i det hele taget sjældent (for ikke at sige aldrig) anbefalelsesværdigt at spørge, om to flydende komma tal er *ens*, man skal altid spørge, om de er *tilstrækkeligt* ens.

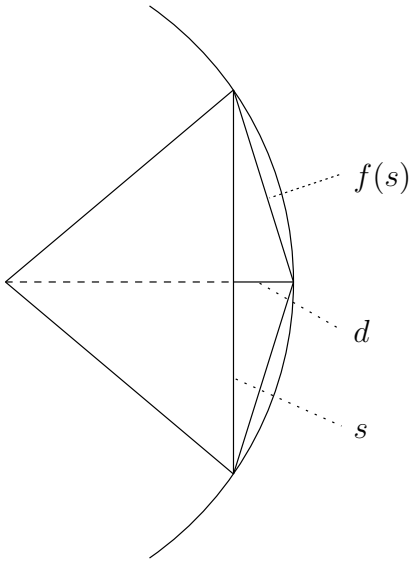
### Gode og dårlige formler

Som det sidste eksempel i dette afsnit skal vi vise, hvordan matematisk set ækvivalente formler kan have vidt forskellige numeriske egenskaber. Det konkrete eksempel er en algoritme til beregning af $\pi$, som for voksende værdier af $k$ beregner omkredsen af en regulær $k$-kant, der er indskrevet i enhedscirklen. Algoritmen starter med en 6-kant, hvis kantlængde er 1, og den fordobler antallet af kanter, indtil omkredsen af en $k$-kant er lig omkredsen af en $2k$-kant. Denne omkreds er den beregnede værdi af $2\pi$ (omkredsen af enhedscirklen er som bekendt $2\pi$). Algoritmen ser ud som følger

Algoritme: Beregning af $\pi$
Stimulans :
Respons : $r$: approximation til $\pi$
Metode :

```
sg = 0;
pg = 0;
sn = 1;
pn = 6;
k = 6;
while (pn>pg) {
  sg = sn;
  pg = pn;
  sn = f(sg);
  k = 2*k;
  pn = k*sn;
}
r = pn/2;
```

Den funktion $f(s)$, der optræder i algoritmen, beregner sidelængden i en indskreven $2k$-polygon, når sidelængden i en $k$-polygon er $s$. Vi kan finde et udtryk for $f(s)$ v.hj.a. følgende tegning




hvor et par anvendelser af Pythagoras giver

$$\begin{aligned} f(s)^2 &= d^2 + (\frac{s}{2})^2 \\ &= (1 - \sqrt{1 - (\frac{s}{2})^2})^2 + (\frac{s}{2})^2 \\ &= 1 + (1 - (\frac{s}{2})^2 - 2\sqrt{1 - (\frac{s}{2})^2}) + (\frac{s}{2})^2 \\ &= 2(1 - \sqrt{1 - (\frac{s}{2})^2}) \end{aligned}$$

Omend matematisk korrekt, bør et sådant udtryk ikke anvendes til regning med flydende komma tal af følgende årsager. Når $s$ er lille, er $1-(\frac{s}{2})^2$ næsten lig med 1, dvs. $\sqrt{1-(\frac{s}{2})^2}$ er endnu mere lig med 1, hvorfor $1-\sqrt{1-(\frac{s}{2})^2}$ er tæt på 0, og så må der forventes at ske et stort ciffertab ved subtraktionen $1-\sqrt{}$. I denne situation bør man lede efter en ækvivalent formel, hvor ciffertabet begrænses mest muligt. I det aktuelle tilfælde kan vi erstatte den kritiske subtraktion med en addition v.hj.a. identiteten

$$1 - \sqrt{x} = \frac{1 - x}{1 + \sqrt{x}}$$

hvilket giver følgende alternative udtryk for $f(s)^2$

$$\begin{aligned} f^*(s)^2 &= 2\,\frac{1-(1-(\frac{s}{2})^2)}{1+\sqrt{1-(\frac{s}{2})^2}} \\ &= \frac{s^2}{2(1+\sqrt{1-(\frac{s}{2})^2})} \end{aligned}$$

som er befriet for ciffertab så langt som det er praktisk muligt.

Følgende tabel viser værdierne af de centrale variable i algoritmen ved en udførelse i $F(10,4,-99,99)$ for de to forskellige formler for $f$

$$\begin{aligned} f(s) &= \sqrt{2-\sqrt{(2+s)(2-s)}} \\ f^*(s) &= \frac{s}{\sqrt{2+\sqrt{(2+s)(2-s)}}} \end{aligned}$$

Hvis den "gode" formel $f^*(s)$ anvendes, ser beregningen ud som følger

| $k$ | $\sqrt{(2+sg)(2-sg)}$ | $2+\sqrt{(2+sg)(2-sg)}$ | $sn$ | $pn$ |
|---|---|---|---|---|
| 6 | – | – | 1.000 | 6.000 |
| 12 | 1.732 | 3.732 | 0.5176 | 6.211 |
| 24 | 1.937 | 3.937 | 0.2611 | 6.266 |
| 48 | 1.982 | 3.982 | 0.1308 | 6.278 |
| 96 | 1.995 | 3.995 | 0.06546 | 6.284 |
| 192 | 1.998 | 3.998 | 0.03273 | 6.284 |

dvs. den beregnede approximation af $\pi$ er

$$\pi \approx 3.142$$

Anvendes derimod udtrykket $f(s)$ fås følgende beregning

| $k$ | $\sqrt{(2+sg)(2-sg)}$ | $2-\sqrt{(2+sg)(2-sg)}$ | $sn$ | $pn$ |
|---|---|---|---|---|
| 6 | – | – | 1.000 | 6.000 |
| 12 | 1.732 | 0.2670 | 0.5176 | 6.211 |
| 24 | 1.937 | 0.06200 | 0.2626 | 6.302 |
| 48 | 1.983 | 0.01600 | 0.1341 | 6.436 |
| 96 | 1.994 | 0.006000 | 0.07745 | 7.435 |
| 192 | 1.997 | 0.003000 | 0.05477 | 10.51 |
| 384 | 1.998 | 0.002000 | 0.04472 | 17.17 |
| 768 | 1.998 | 0.002000 | 0.04472 | 34.34 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

hvoraf det følger, at beregningen overhovedet ikke konvergerer. Ved inspektion af søjlen med værdierne af $2 - \sqrt{(2+sg)(2-sg)}$ er det også nemt at se, at det er her den manglende nøjagtighed slår igennem, idet den hurtigt kommer ned på ét betydende ciffer.

Følgende to JAVA programmer beregner ligeledes $\pi$ v.hj.a. ovenstående algoritme og med de samme to formler for $f(s)$, og viser, at nøjagtighedsproblemerne ikke skyldtes et for lille talsystem. Når flere cifre medtages, kan vi måske udskyde problemet, men vi fjerner ikke årsagen, nemlig en uhensigtsmæssig beregningsformel, som i ethvert flydende komma tals system vil bevirke, at vi ikke opnår den nøjagtighed, som det var rimeligt at forvente. Til sammenligning skal det anføres, at den korrekte værdi af $\pi$ er

$$\pi = 3.141592653589793\ldots$$

```
public static void piBest() {
  double p0 = 0;  double pn = 6;
  double s0 = 0;  double sn = 1;
  int k = 6;
  while (pn>p0) {
    s0 = sn;    sn = s0/Math.sqrt(2+Math.sqrt((2+s0)*(2-s0)));
    k = 2*k;
    p0 = pn;    pn = k*sn;
  }
  System.out.println(p0/2 +"    "+k);
}
```

```
3.141592653589794 402653184
```

```
public static void piWorst() {
  double p0 = 0;  double pn = 6;
  double s0 = 0;  double sn = 1;
  int k = 6;
  while (pn>p0) {
    s0 = sn;    sn = Math.sqrt(2-Math.sqrt((2+s0)*(2-s0)));
    k = 2*k;
    p0 = pn;    pn = k*sn;
  }
  System.out.println(p0/2 +"    "+k);
}
```

```
3.141593669849427 786432
```

### Sammenfatning

Vi har i dette afsnit forsøgt at illustrere nogle af de principielle problemstillinger i forbindelse med regning med "endelige reelle tal" og har herunder konkret peget på følgende

- Ved summation af rækker bør man tilstræbe at additioner foregår mellem delresultater der er næsten lige store. Summation i numerisk voksende orden er bedre end summation i numerisk aftagende orden.
- Man bør levere sine resultater med fuld nøjagtighed hvis man kan.
- Man skal ikke spørge om to flydende komma tal er *ens*, men derimod om de er tilstrækkeligt ens. Fejltolerancer skal vælges med omhu.
- Matematiske udtryk bør så vidt muligt omskrives med henblik på at minimere ciffertab. Vi har i et eksempel omskrevet $x - y$ til $\frac{x^2-y^2}{x+y}$.