Datalogisk Institut
Aarhus Universitet

dIntProg-E02

Introduktion til Programmering

Noter

E2002

August 2002

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

`http://www.daimi.au.dk/dIntProg/eksempler/note_...`

eller direkte i directories

`/users/gudmund/dintprog/eksempler/note_...`

hvis du har en UNIX-konto på Datalogisk Institut.

Indhold

1 Java syntaxdiagrammer

Nærværende note giver syntaxdiagrammer for Java.

Kun en delmængde af Java er medtaget. Bl.a. er `switch`- og `for`-sætninger udeladt, ligesom forkortelser af typen `+=`, `++` er udeladt, og de ekstra basale typer `float`, `long` med flere er udeladt. For den fulde Java syntax henvises til Java Language Specification.

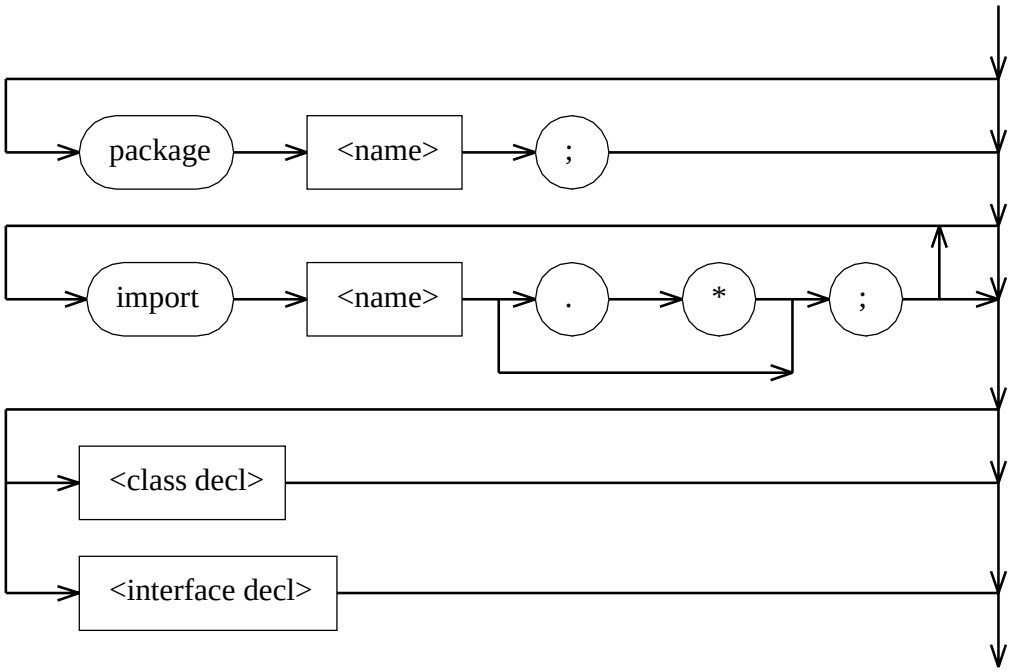
Omvendt er ikke alle konstruktioner der kan laves ifølge syntaksdiagrammerne legale. F.eks. skal udtrykket der angives som betingelse i en `if`- eller `for`-sætning være af type `boolean`, og en metode der erklæres i et interface skal have et `';'` istedet for en rigtig sætningsdel (`<block>`). Der henvises ligeledes til Java Language Specification for de fuldstændige regler

Ikke alle detaljer er angivet, reglerne for konstanter af type `int`, `double`, `char`, `String` (f.eks `-17`, `3.9E-5`, `'z'`, `"java\n"`) er udeladt. For `<identifier>` gælder at det ikke må være et af nøgleordene i Java:

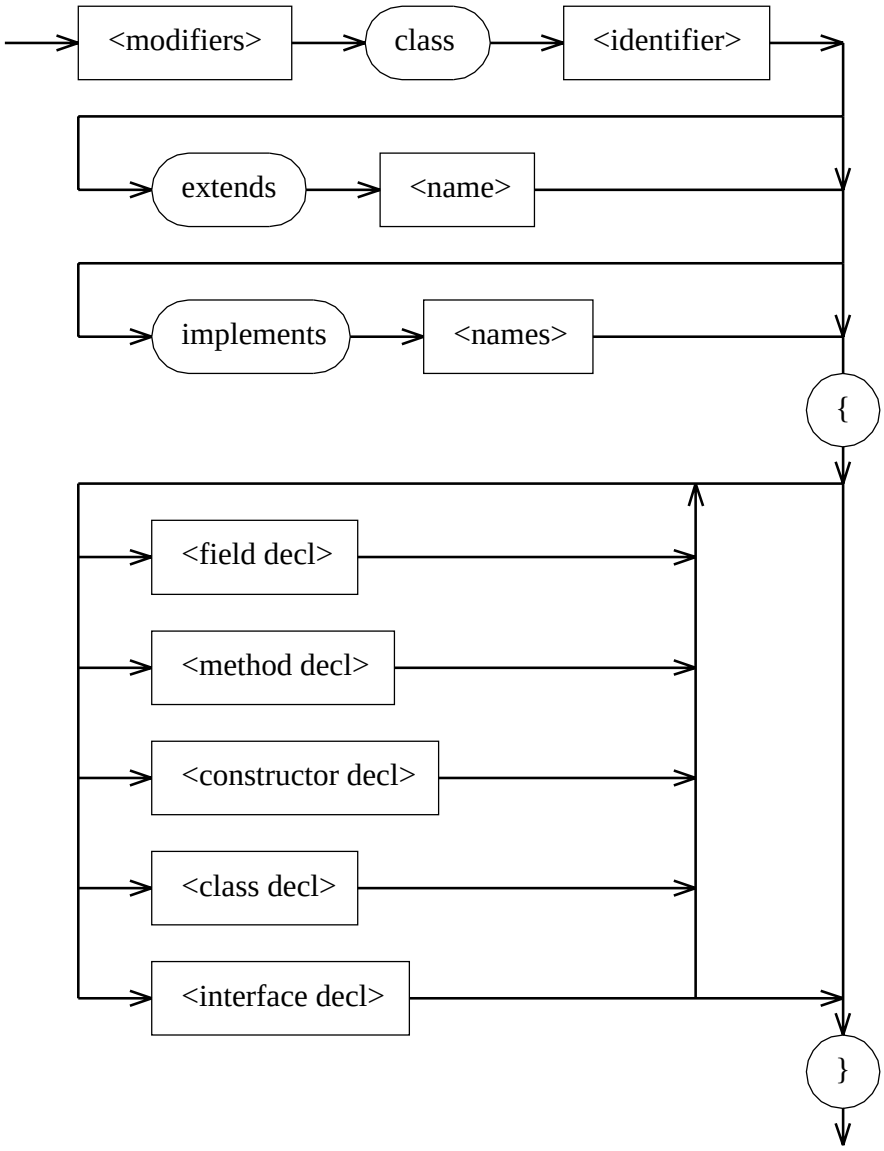
`abstract`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `else`, `extends`, `final`, `finally`, `float`, `for`, `goto`, `if`, `implements`, `import`, `instanceof`, `int`, `interface`, `long`, `native`, `new`, `package`, `private`, `protected`, `public`, `return`, `short`, `static`, `strictfp`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `try`, `void`, `volatile`, `while`.

Gudmund Frandsen og Ole Østerby, 26. nov. 1999.
(rettet aug. 2001)

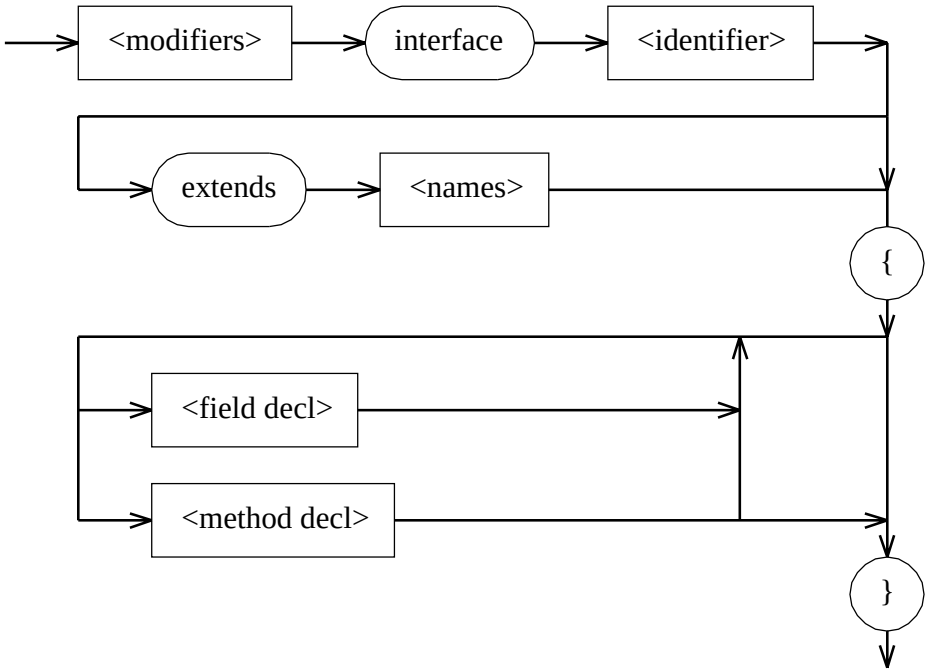
<unit>



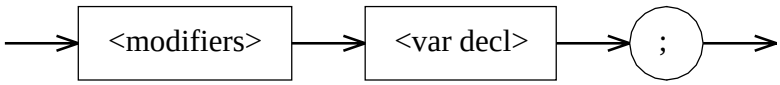
<class decl>


<modifiers>
class
<identifier>
extends
<name>
implements
<names>
{
<field decl>
<method decl>
<constructor decl>
<class decl>
<interface decl>
}

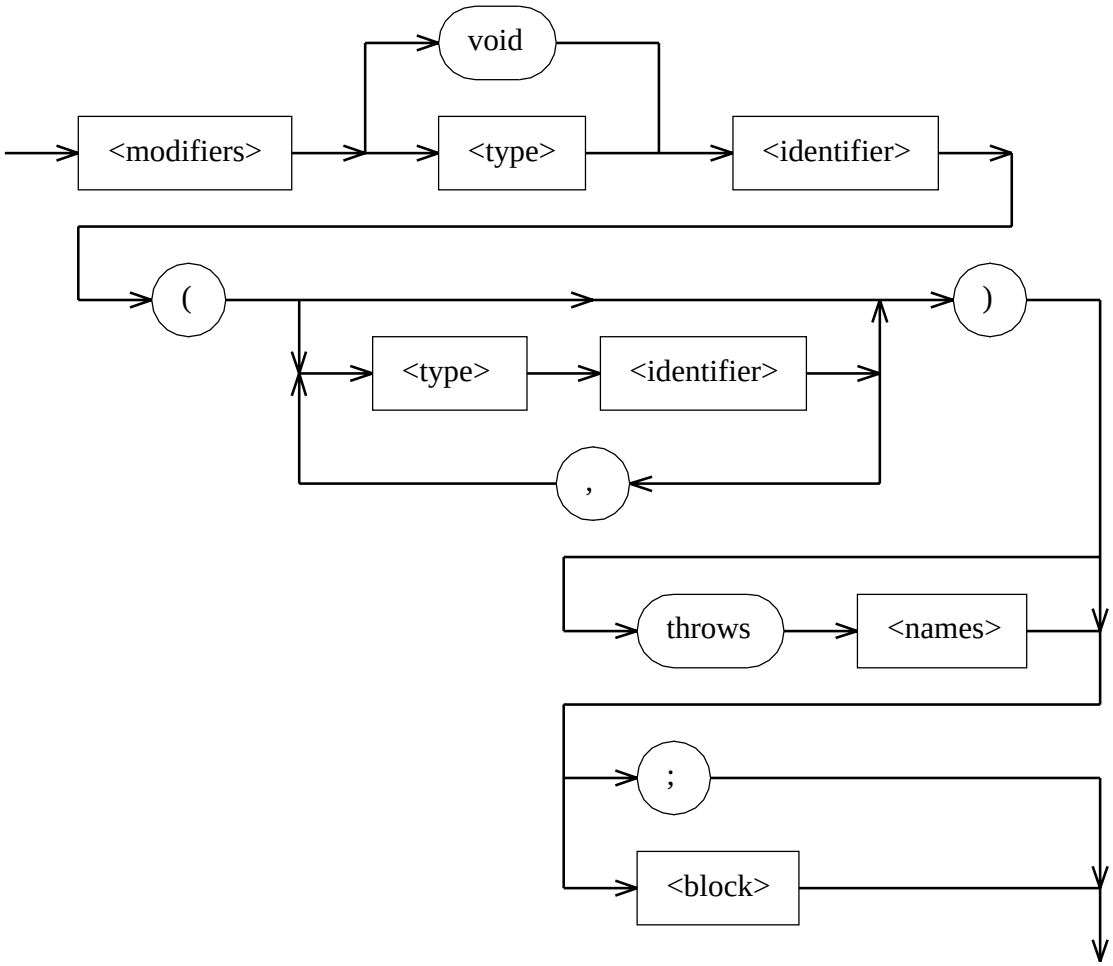
<interface decl>



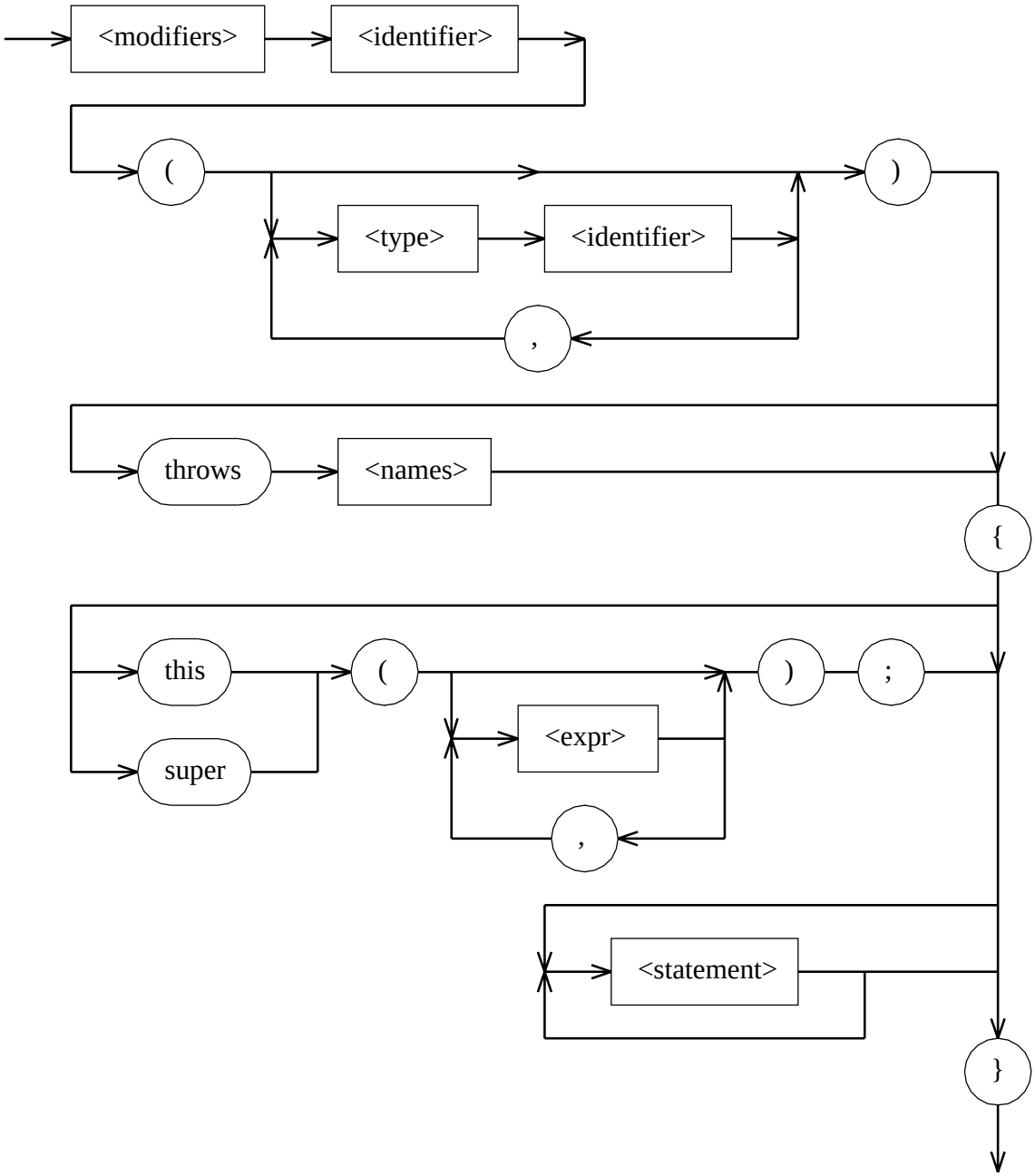
<field decl>



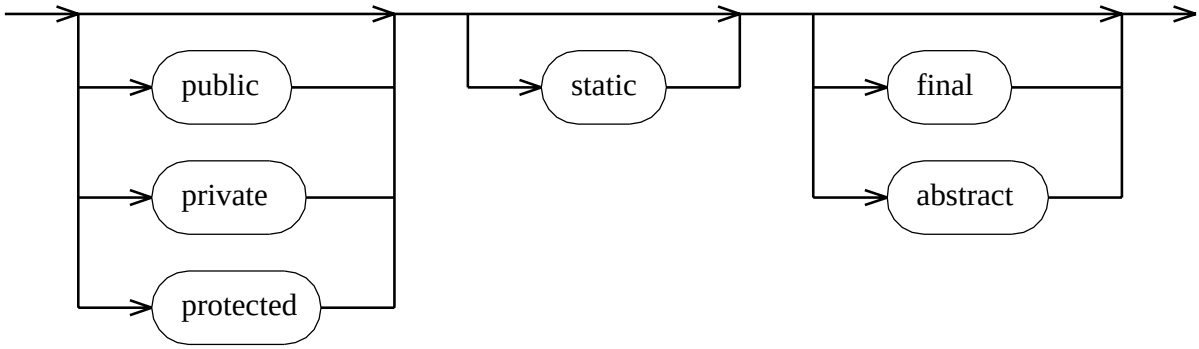
<method decl>


void
<modifiers>
<type>
<identifier>
(
)
<type>
<identifier>
,
throws
<names>
;
<block>

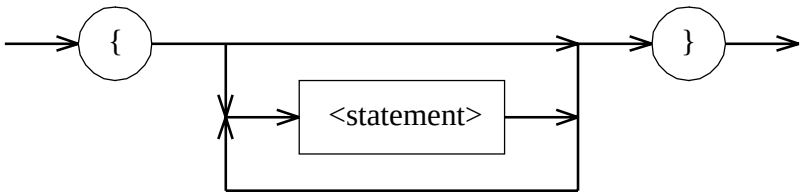
<constructor decl>



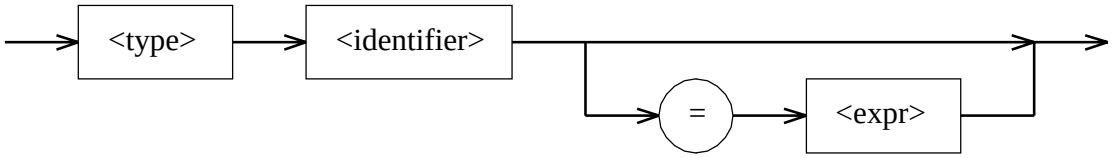
<modifiers>



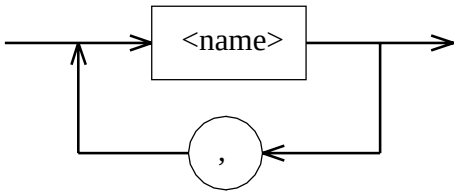
<block>



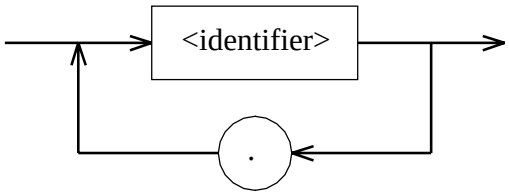
<var decl>



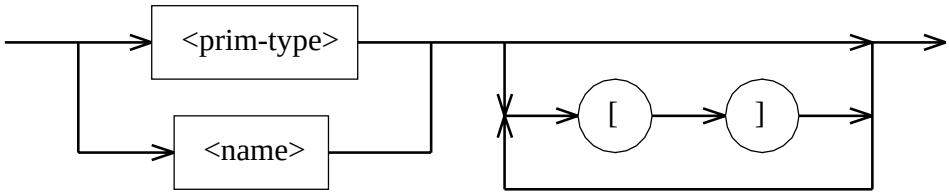
<names>



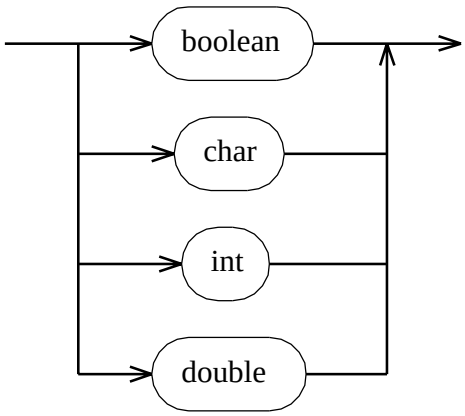
<name>



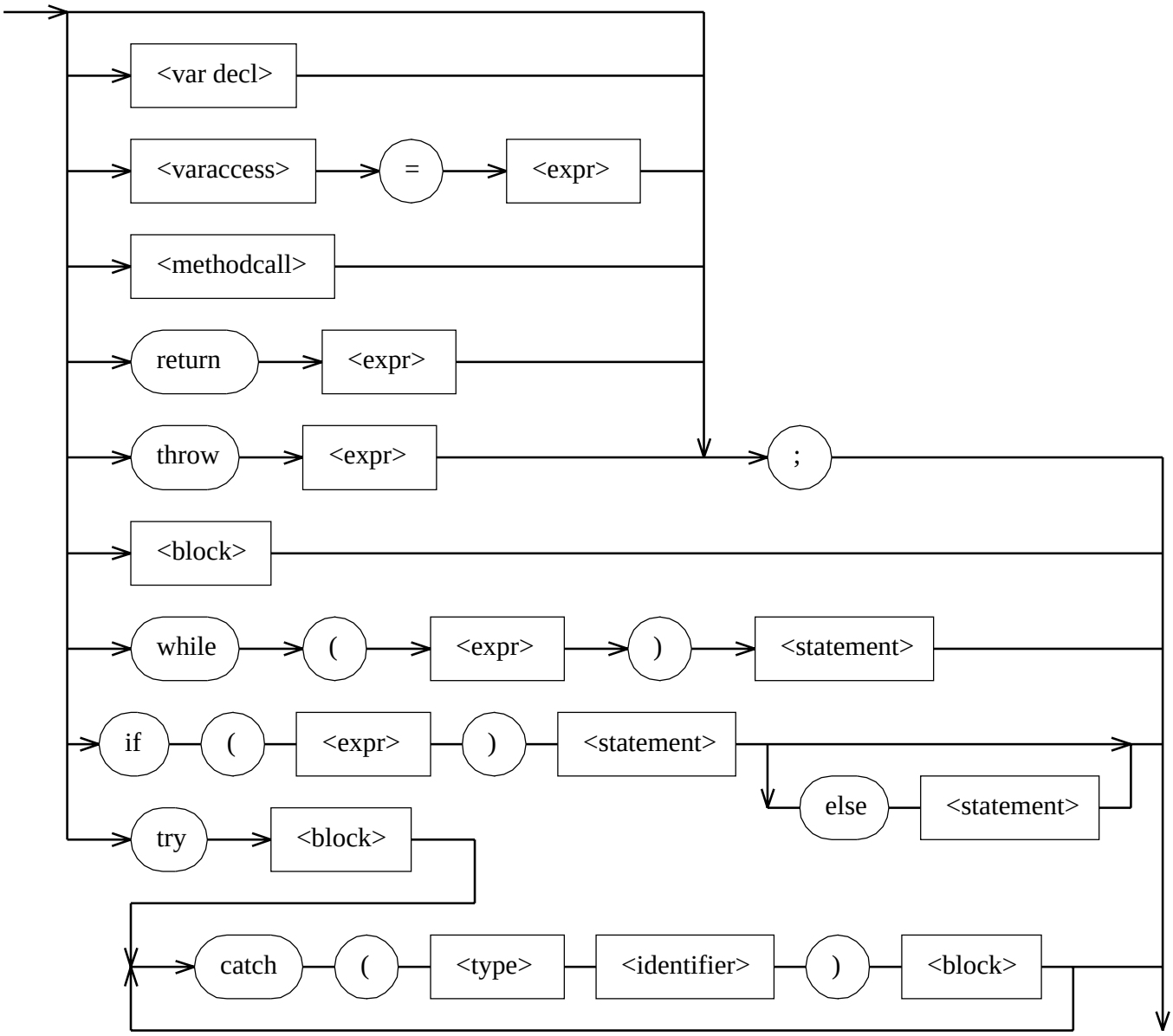
<type>



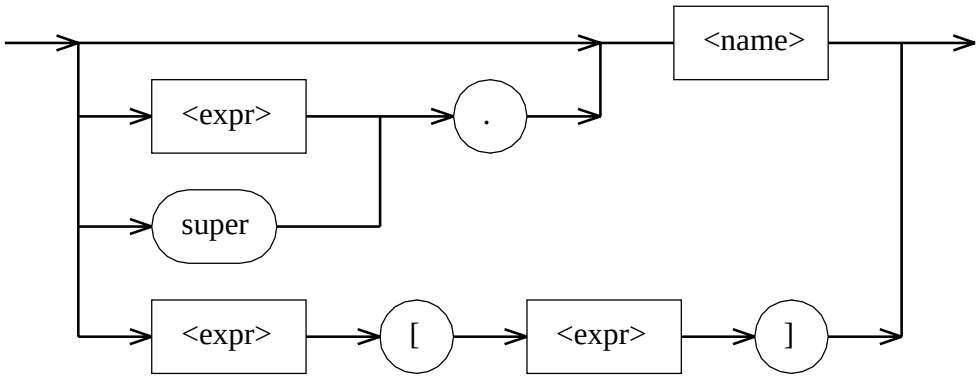
<prim-type>



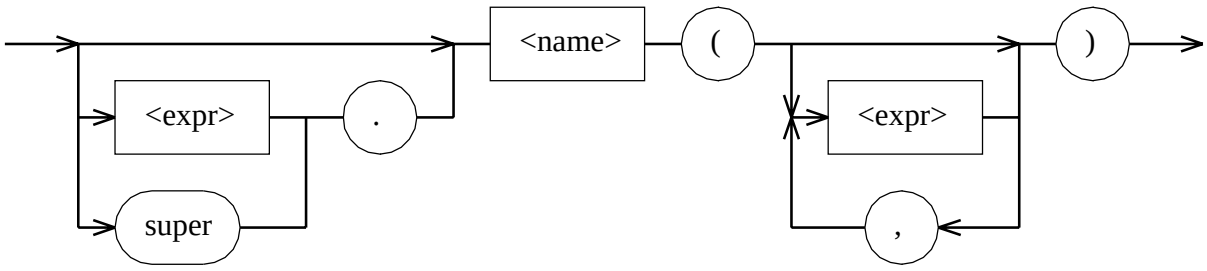
<statement>


<var decl>
<varaccess>
=
<expr>
<methodcall>
return
<expr>
throw
<expr>
;
<block>
while
(
<expr>
)
<statement>
if
(
<expr>
)
<statement>
else
<statement>
try
<block>
catch
(
<type>
<identifier>
)
<block>

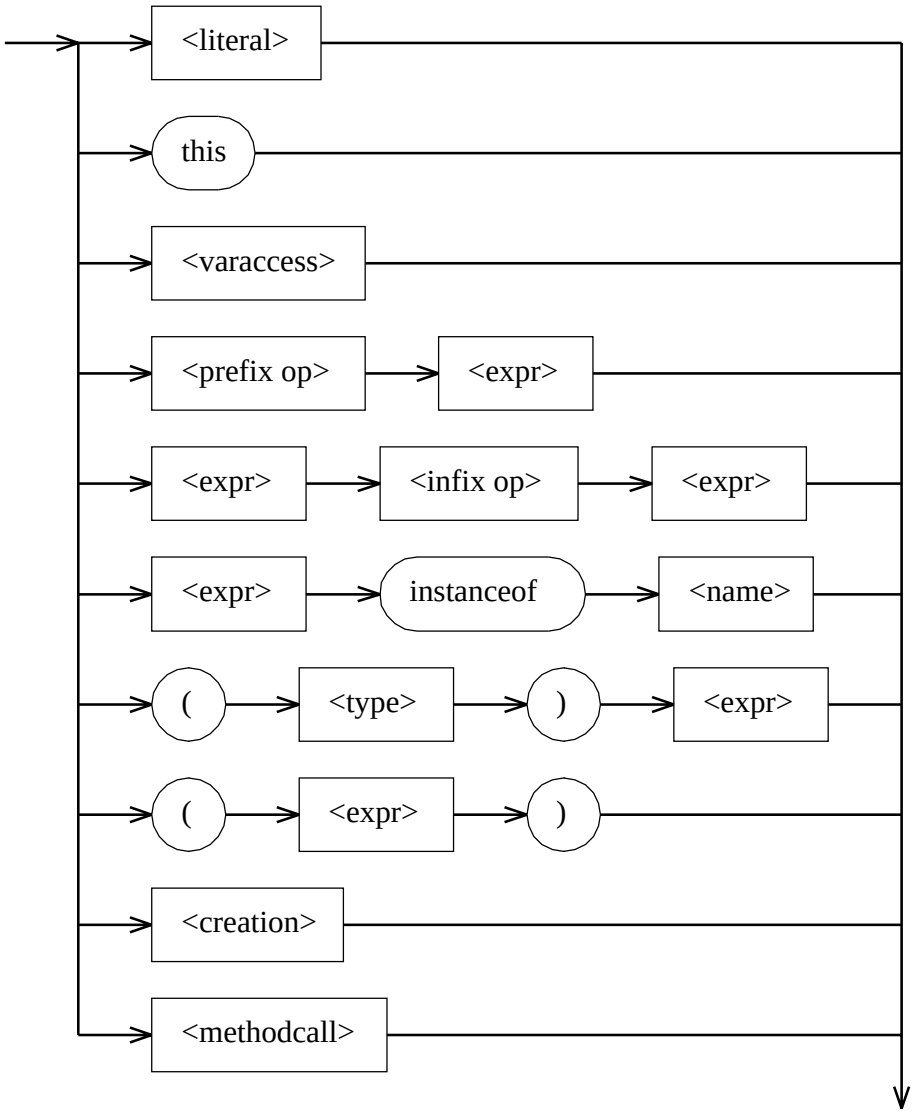
<varaccess>



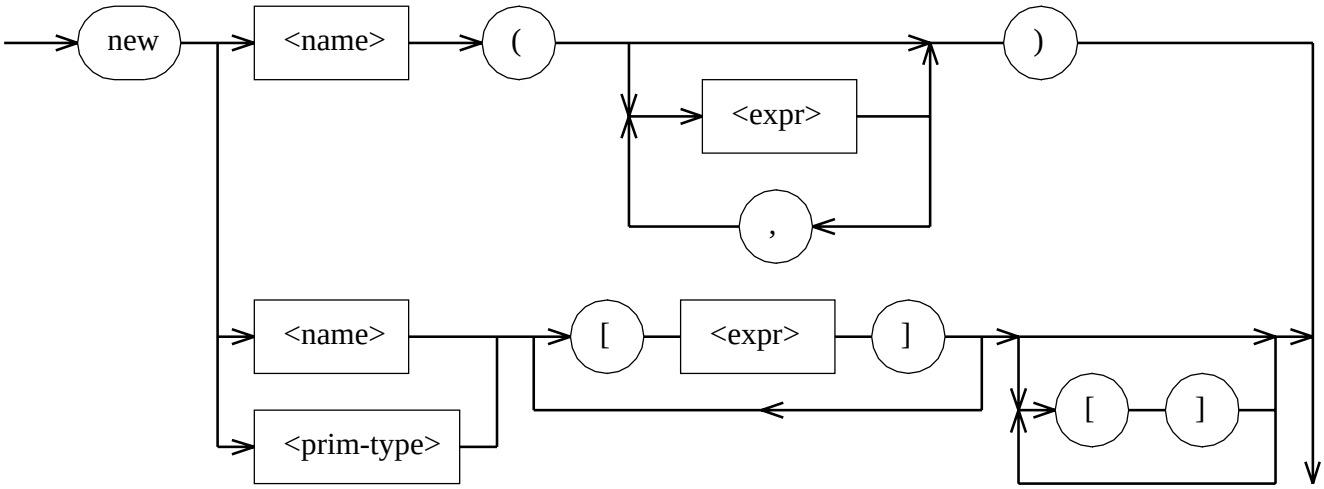
<methodcall>



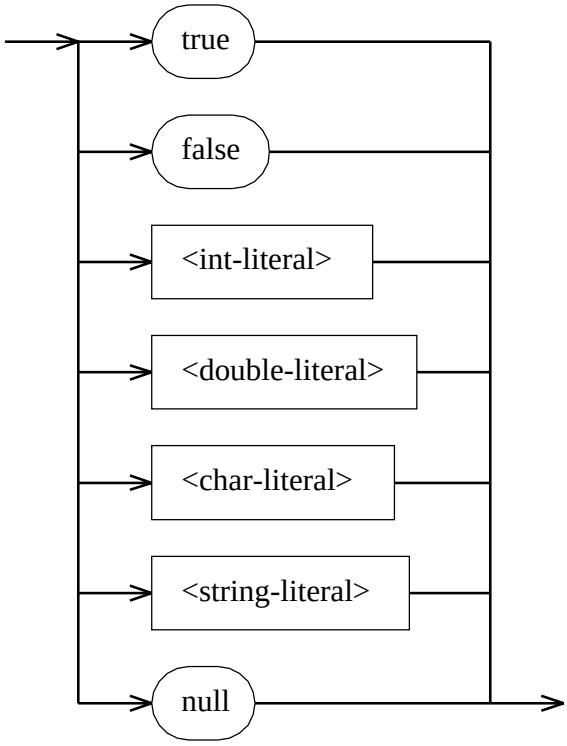
<expr>


<literal>
this
<varaccess>
<prefix op>
<expr>
<expr>
<infix op>
<expr>
<expr>
instanceof
<name>
(
<type>
)
<expr>
(
<expr>
)
<creation>
<methodcall>

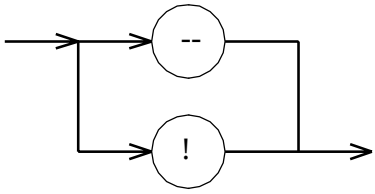
<creation>



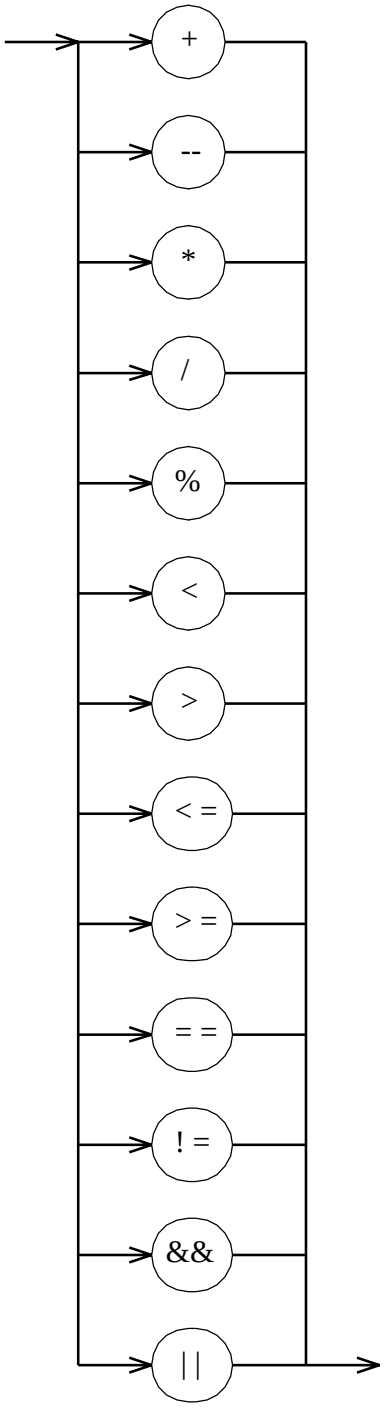
<literal>



<prefix op>



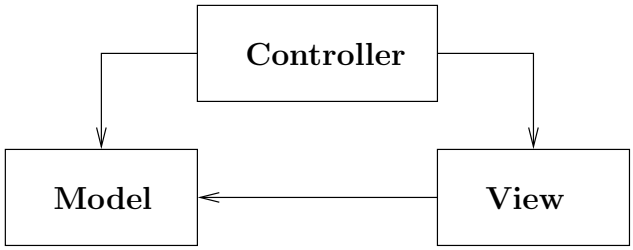
<infix op>



2 Model-View-Controller (MVC)

I kurset forsøges mindre systemer struktureret, så de indgående klasser har en af følgende 3 roller:

- *Modellen* repræsenterer i sin tilstand væsentlige data for hele systemet, og modellen har metoder til aflæsning og ændring af disse data.
- *View* kan præsentere disse data for omverdenen. Oftest som en grafisk præsentation, men den kan også være tekstbaseret. View henter den nødvendige information via modellens access-metoder.
- *Controller* koordinerer systemet ved at læse input, opdatere data i modellen og give besked til view, når ændrede data giver behov for revision i output.



MVC er et eksempel på et design-mønster, dvs en genanvendelig ide/struktur til løsning af forskellige problemer.

Brug af MVC har en række fordele:

- For novicen er det vanskeligt at løse en programmeringsopgave fra bar bund. Anvendelse af MVC tvinger programmøren til at overveje hvorledes specielt modellen ser ud i det konkrete tilfælde, og derved vil man ofte være godt på vej til at fokusere på den væsentlige del af problemet og kan isolere detaljer vedrørende brugergrænsefladen i view-delen.
- Der opnås en modularitet, derved at brugergrænsefladen kan ændres eller udskiftes uafhængigt af modellen, og man kan have flere forskellige (samtidige) brugergrænseflader.
- Der sker en afkobling, derved at hele systemet deles i mindre enheder, som har forholdsvis lidt med hinanden at gøre. Det bliver lettere at forstå koden i view-delen, når den er adskilt fra muligvis komplicerede beregninger i model-delen, ligesom model-delens beregninger bliver lettere at overskue uden sammenblanding med brugergrænsefladen.

- Kontrolstrukturen i systemet fremtræder klart i controller. Man kan se hvorledes bestemte brugerstimuli giver anledning til at modellens opdateringsmetoder kaldes.

2.1 Observer

Opdelingen i M, V og C er uformel, og ikke ganske entydig i praksis. Vi ser nærmere på en variant.

Som ovenfor beskrevet, så er det controller, der efter input fra brugeren og opdatering af modellen giver besked til view om at output skal ændres. På denne måde bliver output gentegnet efter hver potentiel opdatering af modellen, uanset om der er reelle ændringer. Hvis modellen istedet direkte informerer view uden om controller, så kan modellen nøjes med at insistere på opdatering af output i de tilfælde, hvor der er faktiske ændringer. Men en sådan ændret kommunikationstruktur har en ny ulempe, derved at modellen skal kende view.

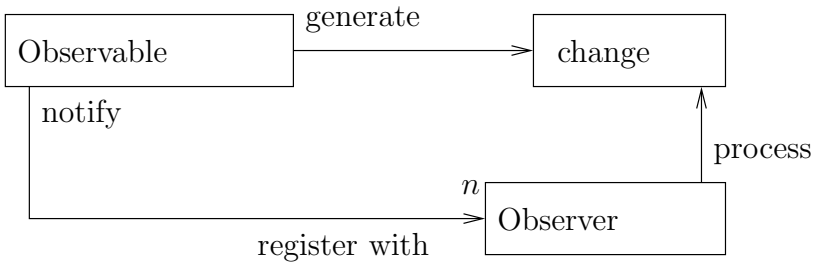
Man kan opnå fordelen og undgå ulempen ved at anvende en analog til *lytter*-begrebet fra hændelsesstyret programmering. Hver view skal registreres som en *observer* hos modellen, der til gengæld lover at fortælle sine observatører, når den undergår ændringer. Dette kan implementeres i en generel klasse, som alle modelklasser udvider. F.eks. således

```
public class Observable {
  private ArrayList observers = new ArrayList();
  private boolean changed;

  public void addObserver(Observer o) {
     observers.add(o);
  }

  public void setChanged() {
    changed = true;
  }

  public void notifyObservers(Object arg) {
    if (changed) {
      for (int i=0; i< observers.size(); i++) {
        Observer o = (Observer)observers.get(i);
        o.update(this,arg);
```

```
      }
      changed = false;
    }
  }
}
```

Klassen benytter interfacet

```
interface java.util.Observer {
  void update(Observable o, Object arg);
}
```

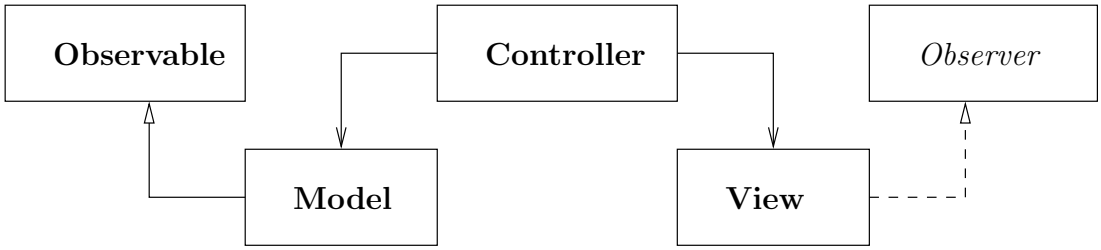


Modellen skal udvide `Observable` og view skal implementere `Observer`. En view registreres som observer ved et kald af modellens `addObserver`. Modellen afslutter update-metoder med et kald til både `setChanged` og `notifyObservers`. Derved får alle view besked gennem kald af deres `update`-metoder. To forhold kræver en kommentar:

- Hvorfor findes metoden `setChanged`? - I komplicerede modeller kan ændringer ske mange steder, og view skal ikke opdateres hver gang. Ved hver faktisk ændring kaldes `setChanged`. Ved afslutning af metoden kaldes `notifyObservers`, og den bevirker opdatering af view, hvis der har været mindst en ændring undervejs.

- Hvorfor er der to parametre til update? - Første parameter giver view mulighed for at kalde accessmetoder i modellen. Samtidigt giver anden parameter modellen mulighed for at give view relevant information direkte i `update`-kaldet. Sidste parameter behøver ikke udnyttes, men den giver fleksibilitet.

Interface `Observer` og en mere sofistikeret udgave af `Observable` findes i Java-pakken `java.util`.



2.2 Eksempel

Dette kan illustreres ved et eksempel. For at gøre eksemplet kort anvendes ikke grafik. Vi ser på faktorisering af heltal. Controller indlæser et tal fra bruger, og leverer det til modellen, der har en metode, der faktoriserer heltal. For hver primfaktor, der findes, bliver brugeren informeret gennem opdatering af view.

Eksempel på input/output:

```
type integer: 87456432875643875

87456432875643875 = 5*...
87456432875643875 = 5*5*...
87456432875643875 = 5*5*5*...
87456432875643875 = 5*5*5*29*...
87456432875643875 = 5*5*5*29*145477*...
87456432875643875 = 5*5*5*29*145477*165840047
```

model:

```
public class Model extends Observable {
  public void factorNumber(long n) {
    long q = n;  long d = 2;  String result = "";
    while (d*d<=q) {
      if (q%d == 0) {
        q = q/d;      result = result + d + "*";
        setChanged();
        notifyObservers(n +" = "+result+"...");
      } else { d = d+1; }
```

```
    }
    setChanged();
    notifyObservers(n +" = "+result+q);
  }
}
```

view:

```
public class View implements Observer {
  public void update(Observable o, Object arg)
  { System.out.println(arg.toString()); }
}
```

controller:

```
public class Controller {
  private Model m;
  public Controller(Model mo) { m = mo; }
  public void go() {
    long n = new LineReader().readLong("type integer: ");
    m.factorNumber(n);
  }
}
```

registrering af observer m.m.:

```
public class ObserverDemo {
  public static void main(String[] args) {
    Model m = new Model(); View v = new View();
    m.addObserver(v);   new Controller(m).go();
  }
}
```

3 Rekursive datastrukturer

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

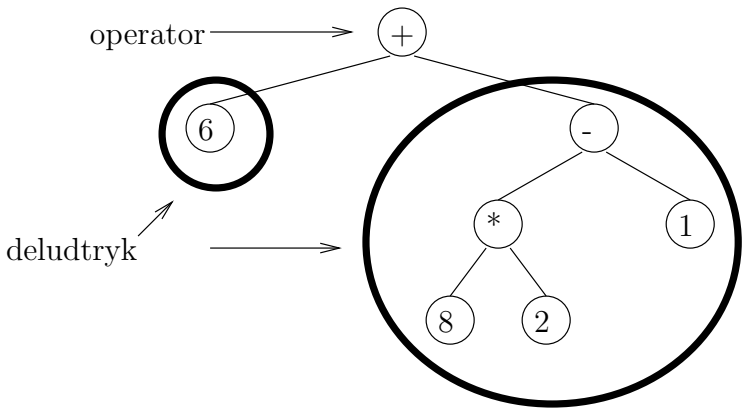
- En *liste* består af et enkelt element efterfulgt af en (kortere) liste eller listen er tom.
- Et *filsystem* består af et directory indeholdende en række mindre filsystemer eller filsystemet består af en enkelt fil

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

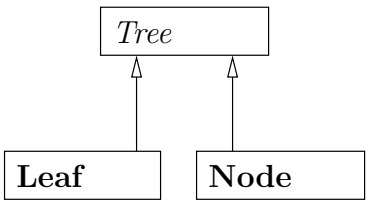
Filsystemet er et eksempel på en rekursiv træstruktur, og denne note vil anvende rekursive binære udtrykstræer som gennemgående eksempel.

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 lille klassehierarki, som kun har to niveauer. På øverste niveau er der en en abstract klasse (den generelle struktur) og på nederste niveau er der en klasse for hvert specialtilfælde. For udtrykstræernes vedkommende får man således følgende hierarki med tre klasser

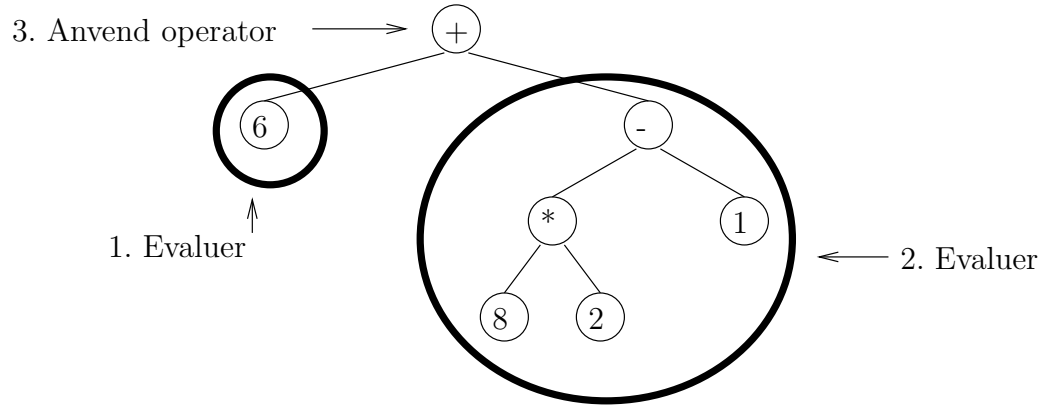



```
public abstract class Tree { }

public class Leaf extends Tree {
    private int num;

    public Leaf(int n) { num = n; }
    public int num() { return num; }
}

public class Node extends Tree {
    private String op;
    private Tree left;
    private Tree right;

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

Det tidligere nævnte udtryk kan repræsenteres ved

```
Tree t =
  new Node(
    new Leaf(6),
    "+",
    new Node(new Node(new Leaf(8),"*",new Leaf(2)), "-",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).num();
  else {
    Tree l = ((Node)w).left();
```

```
    Tree r = ((Node)w).right();
    String o = ((Node)w).op();
    int left_answer = evaluate(l);
    int right_answer = evaluate(r);
    if (o.equals("+"))
        answer = left_answer + right_answer;
    else if (o.equals("*"))
        answer = left_answer * right_answer;
    else throw new RuntimeException("unknown operator:" + o);
  }
  return answer;
}
```

Javakoden 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.

Den abstrakte klasse i toppen af hierarkiet får en abstrakt metode `getValue`, som de to klasser i bunden af hierarkiet udfylder med indhold. På denne måde kan metodekaldet `evaluate(t)` erstattes af kaldet `t.getValue()` for `t` af type `Tree`.

```
abstract class Tree {
    public abstract int getValue() ;
}

class Leaf extends Tree {
    ...
    public int getValue() { return num; }
}

class Node extends Tree {
    ...
    public int getValue() {
        if (op.equals("+"))
            return left.getValue() + right.getValue();
        else if (op.equals("*"))
```

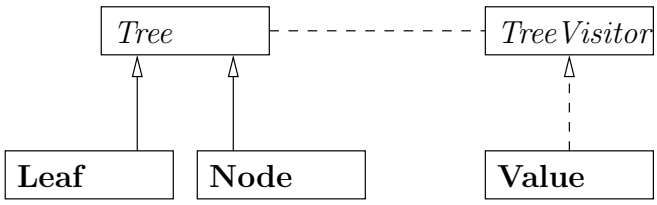
```
            return left.getValue() * right.getValue();
        else throw new RuntimeException
                            ("unknown operator:" + op);
    }
}
```

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 `visitFrom` til hver klasse i hierarkiet. Disse `visitFrom`-metoder sørger for at lave call-back til de anvendelsesspecifikke metoder samlet i den `TreeVisitor`-implementerede klasse.

For beregning af et udtryks værdi benyttes følgende struktur:



De generelle klasser/interface:

```
interface TreeVisitor {
    public Object visitedLeaf(int val);

    public Object visitedNode(Tree left,
                   String op, Tree right);
}

abstract class Tree {
    public abstract Object visitFrom(TreeVisitor v) ;
```

```
}

class Leaf extends Tree {
    ...
    public Object visitFrom(TreeVisitor v) {
        return v.visitedLeaf(num);
    }
}

class Node extends Tree {
    ...
    public Object visitFrom(TreeVisitor v) {
        return v.visitedNode(left,op,right);
    }
}
```

Den anvendelsesspecifikke klasse:

```
class Value implements TreeVisitor {

  public Object visitedLeaf(int val) {
     return new Integer(val);
  }

  public Object visitedNode(Tree left,
                 String op, Tree right) {
    int l = ((Integer)left.visitFrom(this)).intValue();
    int r = ((Integer)right.visitFrom(this)).intValue();
    int answer;
    if (op.equals("+")) answer = l + r;
    else if (op.equals("-")) answer = l - r;
    else if (op.equals("*")) answer = l * r;
    else throw new RuntimeException("unknown operator:" + op);
    return new Integer(answer);
  }
}
```

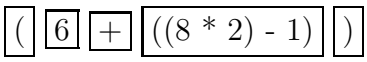
Det tidligere metodekald `t.getValue()` erstattes nu af
`((Integer) t.visitFrom(new Value())).intValue()`

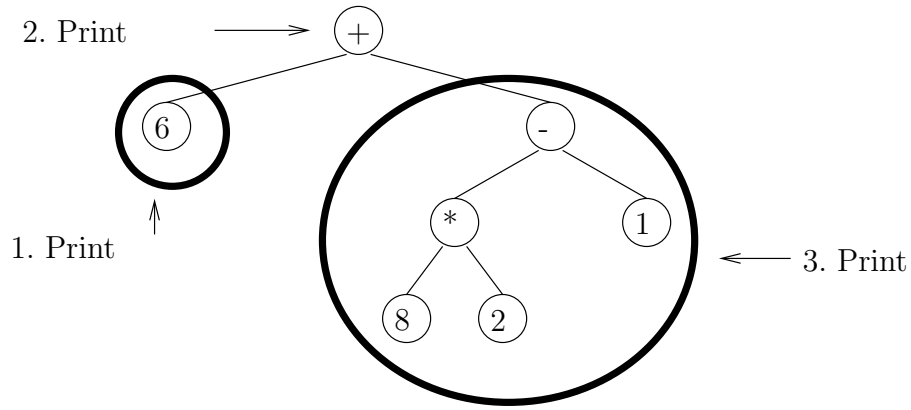
Det anførte cast og brug af wrappertypen `Integer` kan synes omstændelig, men det er altså Java-prisen for at separere den anvendelsesspecifikke

del af gennemløbet (`Value`-klassen) fra den generelle call-back mekanisme i `visitFrom`-metoderne. Mens `t.getValue()` kan returnere en værdi af den primitive type `int`, må den generelle `visitFrom` returnere en objektreference.

For at lave en anden type gennemløb, er det tilstrækkeligt at lave en ny implementation af `TreeVisitor`. Som eksempel betragtes udskrift, der naturligt foregår 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 )
         + ")"
```





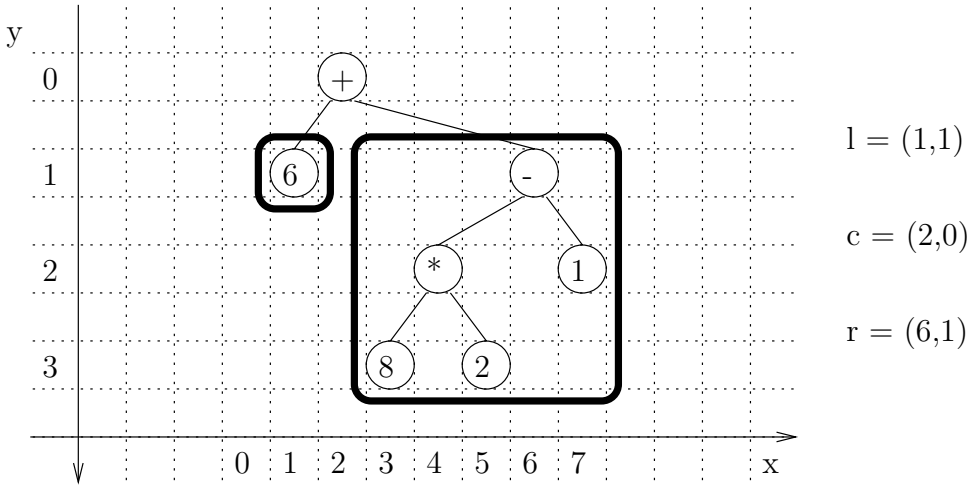
Bemærk at udtrykket ikke udskrives direkte, men den relevante tekst beregnes (uden sideeffekter) og returneres. In-order gennemløbet kan i Java specificeres af følgende klasse

```
class Text implements TreeVisitor {
```

```
  public Object visitedLeaf(int val) {
     return new Integer(val).toString();
  }

  public Object visitedNode(Tree left,
                 String op, Tree right) {
    return ("(" + left.visitFrom(this)
           + op + right.visitFrom(this) + ")");
  }
}
```

Udtrykket `t` kan udskrives med sætningen
`System.out.println(t.visitFrom(new Text()))`.

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 `Draw`, der implementerer `TreeVisitor` og træet `t` kan så vises grafisk med sætningen `new Draw(t)`.

```
class Draw extends JPanel implements TreeVisitor {
  private static final int UNIT = 30;
  private Tree t;
  private Point pen;
  private Graphics2D g;

  public Draw(Tree t) {
    this.t = t;
    JFrame f = new JFrame();
    f.setContentPane(this);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setSize(400,400);
    f.show();
  }

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

  public Object visitedLeaf(int val) {
    return drawSymbol( new Integer(val) );
  }

  public Object visitedNode(Tree left,
                 String op, Tree right) {
    pen.y = pen.y+UNIT;
    Point left_pos = ((Point) left.visitFrom(this));
    pen.y = pen.y-UNIT;
    Point node = drawSymbol(op);
    g.draw(new Line2D.Double(left_pos,node));
    pen.y = pen.y+UNIT;
```

```
    Point right_pos = ((Point) right.visitFrom(this));
    pen.y = pen.y-UNIT;
    g.draw(new Line2D.Double(right_pos,node));
    return node;
  }

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

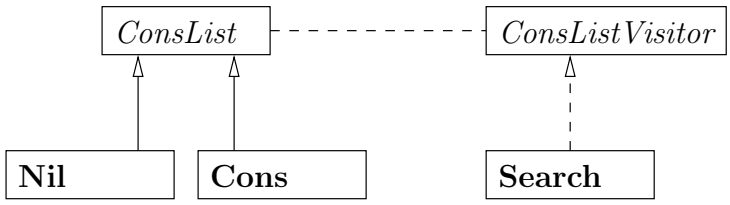
3.3 Rekursiv liste

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 abstract class ConsList {}

public class Nil extends ConsList {}

public class Cons extends 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 `Search`:




```
public abstract class ConsList
{ abstract Object visitFrom(ConsListVisitor v); }

public class Nil extends ConsList {
    Object visitFrom(ConsListVisitor v) { return v.visitedNil(); }
}

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

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

    Object visitFrom(ConsListVisitor v) { return v.visitedCons(hd, tl); }
}

public interface ConsListVisitor {
  public Object visitedNil();
  public Object visitedCons(Object h, ConsList t);
}

public class Search implements ConsListVisitor {
  Object key;

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

  public Object visitedNil() { return new Boolean(false); }

  public Object visitedCons(Object h, ConsList t) {
    if (h.equals(key)) return new Boolean(true);
    else return t.visitFrom(this);
  }
}
```

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

4 Processer, kommunikation og synkronisering

Vi starter med en lille oversigt over kommunikation i JAVA, derefter gennemgås de enkelte konstruktioner med eksempler.

Processer. Vi har indtil nu kun set JAVA-programmer bestående af en enkelt proces. En sådant program kan sagtens omfatte mange objekter, der kalder hinandens metoder, men udførslen foregår strengt sekventielt på een processor. Det er muligt at have flere processer (*threads* i JAVA) der kan udføres samtidigt på forskellige processorer (hvis den datamaskine hvorpå programmet udføres iøvrigt har mere end een processor; imodsat fald sker der en simulering, hvor der på skift udføres en lille bid af de enkelte processer).

Kommunikation. Anvendelsen af flere processer bliver først spændende når de kan *kommunikere* med hinanden. I JAVA sker det ved metodekald på tværs af processerne.

Problem 1: Konflikt. Det kan være uheldigt, hvis to forskellige processer på én gang kalder metoder i et og samme objekt. Derved risikeres eksempelvis, at begge processer samtidigt reviderer værdien af en bestemt variabel, og det er i såfald uklart hvilken tilstand objektet efterlades i.

Synkronisering 1. Dette problem undgås i JAVA ved en sproglig mekanisme, som tillader at man definerer en metode m som værende *synchronized*. Det indebærer at når en proces udfører m hørende til objekt O, så kan ingen andre processer samtidigt udføre Os m metode, eller udføre andre af Os metoder der måtte være definerede som værende *synchronized*.

Problem 2: Deadlock. Det kan være nødvendigt for en proces at vente på at en eller flere andre processer ændrer omgivelserne så en bestemt betingelse b er opfyldt. Eksempel: hvis en laserprinter mangler papir så kan den proces, der styrer printeren sove indtil en anden proces der styrer papirrobotten har fået lagt nyt papir i. Det vil ikke blot være ineffektivt at fordrive ventetiden med en løkke af formen `while (!`b`) do {}`, men hvis den midlertidige standsning sker under kaldet af en synchronized metode m i et objekt O, så kan der blokeres for andre processers brug af O, hvilket i uheldigste fald indebærer at b aldrig bliver opfyldt. Eksempelvis kan printerprocessen have eksklusiv adgang til papirbakken, når den konstaterer at bakken er tom. Hvis printeren ikke frigiver papirbakken til brug for andre processer mens den venter, så vil papirrobotten ikke kunne fylde papirbakken op, og der er opstået en deadlock.

Synkronisering 2. Det problem løses i JAVA ved en sproglig mekanisme (*semaforer*), der giver en proces mulighed for at lade andre processer komme til fadet, mens den venter på at en given betingelse skal blive opfyldt.

4.1 Flere processer

Vi starter med at se på et lille eksempel uden eksplicit kommunikation, hvor to processer begge skriver ud på skærmen.

Man kan danne en selvstændig proces som en udvidelse til klassen `Thread` fra Javas programpakke `java.lang`. I stedet for en `main` metode skal processen have en `run` metode. Hovedprogrammet sætter en proces igang ved at kalde dens `start` metode (der er arvet fra `Thread`). Dette medfører at processens `run` metode bliver udført.

```
class Traade {

  static class Proces extends Thread {
    String name;

    Proces(String name) {
      this.name = name;
    }

    int random(int i, int j) {
      return ( (int) Math.round(i - 0.5 + (j-i)*Math.random()) );
    }

    public void run() {
      for (int i=0; i!=10; i=i+1) {
        System.out.println(name);
        try {sleep(random(0,10)); }
        catch (InterruptedException e) {}
      }
    }
  }

  public static void main(String[] args) {
    Proces a = new Proces("aa");
    Proces b = new Proces("BBBB");
    a.start();
```

```
    b.start();
  }
}
```

I eksemplet udnytter vi yderligere en metode fra superklassen `Thread`, nemlig `sleep`, der standser processen i et antal millisekuner (som angivet i den aktuelle heltalsparameter). Det sikrer at vi får udført de to processer i meget små bidder, selvom vi udfører programmet på en maskine med kun een processor. Ved en faktisk kørsel af programmet viste skærmbilledet følgende output:

```
aa
BBBB
BBBB
BBBB
aa
BBBB
aa
BBBB
aa
aa
BBBB
BBBB
aa
aa
aa
BBBB
aa
BBBB
aa
BBBB
```

I dette eksempel har vi ingen eksplicit kommunikation, men implicit foregår der en *resourcedeling*, idet en metode fra objektet `System.out` kaldes (samtidigt) fra begge processer.

4.2 Resourcedeling og synkronisering

Et andet eksempel på resourcedeling illustreres af følgende program, der omfatter et kontoobjekt og to selvstændige kundeprocesser, der (samtidigt) hæver penge fra kontoen.

```
class Bank {

  static class Account {
    int amount = 0;

    public synchronized void insert(int i) {
      amount = amount + i;
    }

    public synchronized boolean withdraw(int i) {
      if (amount > i) {
        amount = amount - i;
        return true;
      } else { return false; }
    }
  }

  static class Customer extends Thread {
    String name;
    Account a;

    Customer(String name, Account a) {
      this.name = name;
      this.a = a;
    }

    int random(int i, int j) {
      return ( (int) Math.round(i - 0.5 + (j-i)*Math.random()) );
    }

    public void run() {
      for (int i=0; i!=10; i=i+1) {
        int n  = random(0,10);
        boolean succes = a.withdraw(n);
        System.out.println(name+": " + n + " sucess: "+succes);
        try {sleep(random(0,10)); }
        catch (InterruptedException e) {}
      }
    }
  }
```

```
  public static void main(String[] args) {
    Account ac = new Account();
    ac.insert(50);
    Customer a = new Customer("aa",ac);
    Customer b = new Customer("BBBB",ac);
    a.start();
    b.start();
  }
}
```

Læg mærken til kroppen af metoden `withdraw`. Først testes om kontoens saldo er stor nok til at man kan hæve det ønskede beløb, og kun hvis det er tilfældet gennemføres transaktionen. Hvis begge kunder *samtidigt* forsøger at hæve penge fra kontoen, kan der opstå problemer. Antag at saldoen på et tidspunkt er 8 kr, og de to kunder (A og B) derefter forsøger at hæve 5 kr samtidigt. En mulig kronologisk rækkefølge af begivenheder vil være:

1. Under A's kald af `withdraw`: testet `(amount > i)` giver resultat `true`.
2. Under B's kald af `withdraw`: testet `(amount > i)` giver resultat `true`.
3. Under B's kald af `withdraw`: sætningen `(amount = amount-i)` giver ny saldo `3`.
4. Under A's kald af `withdraw`: sætningen `(amount = amount-i)` giver ny saldo `-2`.

Som det ses får begge lov at hæve fra kontoen selvom saldoen er utilstrækkelig.

For at undgå denne eller lignende uheldige forløb, kan man i JAVA anvende *modifier* `synchronized` ved erklæring af en metode og derved beskytte mod flere processers samtidige udførsel af metoden på et og samme objekt.

Mere nøjagtigt beskrevet sker følgende: Hvert objekt har præcis ét *token*. Når en proces A kalder en metode m i et objekt O, hvor m er mærket `synchronized`, så får A først lov at udføre m, når den er kommet i besiddelse af Os token. Efter udførelsen af m tilbageleverer A igen dette token. Da kun én proces ad gangen kan være i besiddelse af Os token, betyder det at kun én proces ad gangen kan være i gang med at udføre en af af Os (synkroniserede) metoder. (I praksis administrerer JAVA systemet ventelister for eventuelle konkurrerende processer).

I det tidligere bankprogram er metoderne `withdraw` og `insert` begge mærket `synchronized`, og det sikrer at hver enkelt kundes transaktion på en bankkonto gøres helt færdig, før andre kunder tillades at indsætte på eller hæve fra samme konto.

4.3 Kommunikationskanaler

I bankkontoeksemplet sker kommunikationen ved at en proces A direkte kalder en metode i et objekt der hører til en anden proces B som proces A ønsker at kommunikere med. Vi skal nu se på et mere abstrakt mønster for kommunikation, hvor der skydes en kommunikationskanal ind imellem de kommunikerende processer.

En kommunikationskanal kan opfattes som et objekt, hvis tilstand beskrives ved:

- en *pakke*: Kanalen kan indeholde netop een pakke (af type `Object`) som er under forsendelse. (Pakken er adresseløs; den indeholder ikke information om hverken afsender eller modtager).
- *optaget/ledig*: Kanalen kan enten være ledig, og så er ingen pakke under forsendelse eller også kan den være optaget og så er netop een pakke under forsendelse.
- en *kø* af ventende processer: Køen omfatter sovende processer der har forsøgt at sende en pakke, mens kanalen var optaget eller forsøgt at modtage en pakke mens kanalen var ledig.

Ved kald af kommunikationskanalens metoder kan man

- *afsende* en pakke: Hvis en proces forsøger at afsende en pakke, når kanalen er ledig, så bliver pakken sendt afsted; hvorefter kanalen er optaget; og alle sovende processer i køen bliver vækket (og det er så op til en vækket proces at gøre et nyt *afsende/modtage*-forsøg). Hvis en proces forsøger at afsende en pakke, når kanalen er optaget, så bliver processen sat til at sove i køen
- *modtage* en pakke: Hvis en proces forsøger at modtage en pakke, når kanalen er optaget, så får den udleveret pakken; hvorefter kanalen er ledig; og alle sovende processer i køen bliver vækket (og det er så op til en vækket proces at gøre et nyt *afsende/modtage*-forsøg). Hvis en

proces forsøger at modtage en pakke, når kanalen er ledig, så bliver processen sat til at sove i køen.

En kommunikationskanal kan i JAVA realiseres ved følgende klasse:

```
class Channel {

  Object p;
  boolean empty = true;

  synchronized public void transmit(Object p) {
    while (!empty) {
      try { wait(); } catch (InterruptedException ie) {}
    }
    this.p = p;
    empty = false;
    notifyAll();
  }

  synchronized public Object receive() {
    while (empty) {
      try { wait(); } catch (InterruptedException ie) {}
    }
    empty = true;
    notifyAll();
    return p;
  }
}
```

Kommunikationskanalens pakke og optaget/ledig status repræsenteres med variablerne `p` og `empty`. Køen af ventende processer håndteres af JAVA fortolkeren, idet metoden `wait()` (som enhver klasse arver fra superklassen `Object`) sætter den kaldende proces i kø, og metoden `notifyAll()` (ligeledes arvet fra Object) har den effekt at alle processer i køen bliver vækket og får kastet en `InterruptedException`.

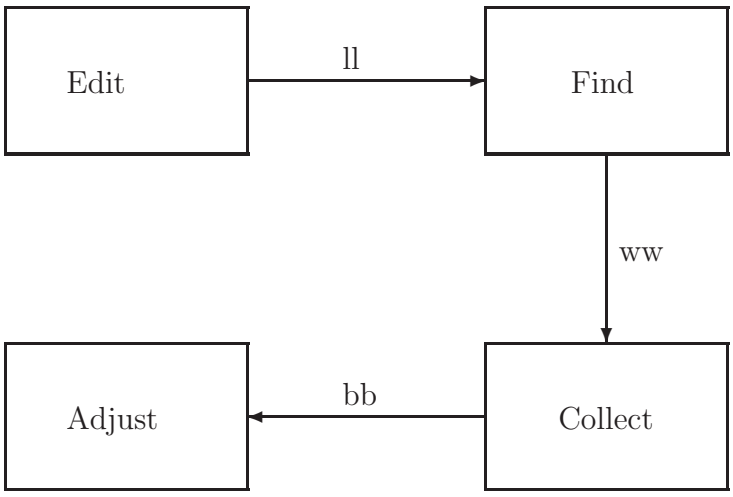
Det skal bemærkes, at når en proces udfører f.eks `transmit` på et `Channel` objekt A, så er A låst så andre processer ikke kan kalde A's metoder (pga `synchronized` modifier), men når processen ved kald af `wait()` sætter sig selv i A's kø, så frigives A, så andre processer kan udføre A's metoder. Når en proces via en metode i A kalder `notifyAll()`, bliver processerne i A's

kø godt nok vækkede, men de får kun een ad gangen lov til at fortsætte udførelsen af A's metoder (efter at den proces der har kaldt `notifyAll()` er færdig med sit kald af en metode fra A).

Vi vil nu anvende Klassen `Channel` i et større eksempel.

4.4 Eksempel: Samlebåndsarbejde

Et klassisk eksempel på et problem, der med fordel kan udnytte flere processer, er produktion af avisspalter. Vi er givet en tekst bestående af en række ord, der skal formatteres som en spalte af en vis bredde med lige højremargen. Man skal simultant finde ordene, finde det rigtige antal der skal ud på samme linje, justere linjen med ekstra blanke og skrive den ud. Man kan meget elegant gøre dette ved at bruge 4 processer. De organiseres som et *samlebånd*, hvor produktet af hver proces sendes videre til yderligere forarbejdning hos den næste.



Den første proces svarer til journalistens skrivemaskine og indlæser teksten en linje ad gangen. Disse *linjer* sendes videre til den anden process, der deler dem op i enkelte ord. Disse *ord* sendes til den tredje process, der bundter dem sammen. Et *bundt* indeholder så mange ord, der kan være på en enkelt linje uden at give overløb, sammen med en angivelse af hvor mange ekstra blanke, der skal sættes ind. Den sidste proces er en printer, der fordeler de ekstra blanke tilfældigt mellem ordene i et bundt og skriver den færdige linje ud.

Udover en klassen `Channel` får vi brug for

- en klasse der kan repræsentere et *bundt*,
- en klasse for hver af de 4 processer,
- et hovedprogram, der konstruerer objekterne og sætter processerne i-gang.

Et bundt repræsenteres ved klassen

```
class Bunch {
  int blanks;
  String[] words;

  Bunch(int blanks, String[] words) {
    this.blanks = blanks;
    this.words = words;
  }
}
```

Den første proces ser således ud

```
class Edit extends Thread {

  BufferedReader in;
  String inname;
  Channel out;

  Edit(String inname, Channel out) {
    this.inname = inname;
    this.out = out;
  }

  public void run() {
    try {
      in = new BufferedReader(new FileReader(inname));
      while(in.ready()) {
        String s = in.readLine();
        out.transmit(s);
      }
      in.close();
    } catch (IOException e) {}
```

```
  }
}
```

Linjerne læses én ad gangen og sendes videre. Man kunne her forestille sig at have et helt system til tekstbehandling, hvor linjerne kan redigeres flere gange, inden de sendes videre til det egentlige spalteprogram. Den anden proces er

```
class Find extends Thread {

  Channel in;
  Channel out;

  Find(Channel in, Channel out) {
    this.in = in;
    this.out = out;
  }

  public void run() {
    while (true) {
      StringTokenizer s = new StringTokenizer((String) in.receive()," ");
      while (s.hasMoreTokens()) {
        out.transmit(s.nextToken());
      }
    }
  }
}
```

der kan ses at bruge en `StringTokenizer` til at opdele linjen i ord. Den tredje proces skrives som

```
class Collect extends Thread {

  Channel in;
  Channel out;
  int col_width;

  Collect(Channel in, Channel out, int col_width) {
    this.in = in;
    this.out = out;
    this.col_width = col_width;
```

```
  }

  public void run() {
    String[] words = new String[col_width];
    int no_chars = 0;
    int no_words = 0;
    while (true) {
      String s = (String) in.receive();
      if (no_chars+no_words+s.length()>col_width) {
        String[] l = new String[no_words];
        for (int i=0; i!=no_words; i=i+1) l[i] = words[i];
        out.transmit(new Bunch(col_width-no_chars-no_words+1,l));
        no_words = 0; no_chars = 0;
      }
      words[no_words] = s;
      ++no_words;
      no_chars = no_chars+s.length();
    }
  }
}
```

Processen opsamler ord, indtil den indlæste spaltebredde overskrides. Den sidste proces ser ud som følger

```
class Adjust extends Thread {

  Channel in;

  Adjust(Channel in) {
    this.in = in;
  }

  int random(int i, int j) {
    return ( (int) Math.round(i - 0.5 + (j-i)*Math.random()) );
  }

  String[] distribute(Bunch b) {
    String[] blanks = new String[b.words.length];
    for (int i=0; i!=blanks.length-1; i=i+1) blanks[i] = " ";
    blanks[blanks.length-1] = "";
    while (b.blanks>0) {
```

```
      int i = random(0,blanks.length-1);
      blanks[i] = blanks[i]+" ";  --b.blanks;
    }
    return blanks;
  }

  public void run() {
    while (true) {
      Bunch b = (Bunch) in.receive();
      String[] blanks = distribute(b);
      for (int i=0; i!= b.words.length; i=i+1) {
        System.out.print(b.words[i]);
        System.out.print(blanks[i]);
      }
      System.out.println();
    }
  }
}
```

hvor metoden distribute fordeler de overskydende blanke tilfældigt mellem ordene. Til slut hovedprogrammet:

```
  public static void main(String[] args) {
    Channel ll = new Channel();
    Channel ww = new Channel();
    Channel bb = new Channel();
    Edit er = new Edit("demo.txt",ll);
    Find fi = new Find(ll,ww);
    Collect co = new Collect(ww,bb,38);
    Adjust ad = new Adjust(bb);
    ad.start();
    co.start();
    fi.start();
    er.start();
  }
```

Bemærk at ved kørsel af programmet skal anvendes `CTL-c` for at standse det, idet de sidste 3 processer venter på at få tilsendt yderligere pakker. Desuden udskrives den sidste spaltelinje ikke, idet den proces som samler ord sammen til hele spaltelinjer står og venter på at få tilsendt flere ord.

Hver af de fire processer er ganske enkel at skrive og kan forbedres og udskiftes, uden at man skal ændre de øvrige. Hvis vi vil tillade *orddeling*, så kan vi (blandt andet) tilføje en ekstra proces. Vi har fået en *faktorisering* af vores beregning. Hvis vi havde flere processorer, så ville spaltningen ovenikøbet gå hurtigere, da alle processerne kan arbejde samtidigt.

4.5 Begrænset buffer

I det netop viste samlebåndseksempel, kan kanalen kun indeholde een pakke. Derved kan f.eks. Edit-processen ikke komme foran Find-processen (med mere end een linje). Der er intet til hinder for at give kanalen lov til at indeholde mange pakker som alle er undervejs på een gang. I spalteopdelingseksemplet er det dog essentielt at pakkerne udtages fra kanalen i samme rækkefølge, som de indsættes i kanalen (ellers kan ordene blive ombyttet i den færdige spalteopdelte tekst).

En sådan mere generel kommunikationskanal med en begrænset buffer kan opfattes som et objekt, hvis tilstand beskrives ved:

- en *liste* af pakker: De pakker der er undervejs lige nu,
- en *kapacitet*: Det højeste antal pakker som kan være undervejs på en gang (kanalen siges at være fyldt, når listen indeholder det højest tilladte antal pakker, og kanalen siges at være tom, når listen er tom),
- en *kø* af ventende processer: Køen omfatter sovende processer der har forsøgt at sende en pakke, mens kanalen var fyldt, samt processer der har forsøgt at modtage en pakke, mens kanalen var tom.

Ved kald af kommunikationskanalens metoder kan man

- *afsende* en pakke: Hvis en proces forsøger at afsende en pakke, når kanalen ikke er fyldt, så bliver pakken sendt afsted; hvis kanalen tidligere var tom, så bliver alle sovende processer i køen vækket. Hvis en proces forsøger at afsende en pakke, når kanalen er fyldt, så bliver processen sat til at sove i køen.
- *modtage* en pakke: Hvis en proces forsøger at modtage en pakke, når kanalen ikke er tom, så får den udleveret pakken; hvis kanalen tidligere var fyldt, så bliver alle sovende processer i køen vækket. Hvis en proces forsøger at modtage en pakke, når kanalen er ledig, så bliver processen sat til at sove i køen.

I JAVA kan kommunikationskanalen med buffer repræsenteres ved at tilføje en fifo-kø (first-in-first-out kø) til vores `Channel` klasse:

```
class BufferChannel {

  Queue buffer;

  BufferChannel(int size) {
    buffer = new Queue(size);
  }

  synchronized void transmit(Object e) {
    while (buffer.isFull())
      try { wait(); } catch (InterruptedException ie) {};
    if (buffer.isEmpty()) {
      buffer.enqueue(e);
      notifyAll();
    } else buffer.enqueue(e);
  }

  synchronized Object receive() {
    while (buffer.isEmpty())
      try { wait(); } catch (InterruptedException ie) {};
    if (buffer.isFull()) {
      Object e = buffer.dequeue();
      notifyAll();
      return e;
    } else return buffer.dequeue();
  }
}

class Queue {

  final int MAX;
  Object[] buffer;
  int first,length;

  Queue(int size) {
    MAX = size;
    buffer = new Object[MAX];
    first = 0; length = 0;
  }

  void enqueue(Object e) {
```

```
    if (isFull()) throw new QueueException("queue is full");
    buffer[(first+length)%MAX] = e;
    length = length+1;
  }

  Object dequeue() {
    if (isEmpty()) throw new QueueException("queue is empty");
    Object e = buffer[first];
    length = length-1;
    first = (first+1)%MAX;
    return e;
  }

  boolean isFull() { return (length==MAX); }
  boolean isEmpty() { return (length==0); }
}

class QueueException extends RuntimeException {
  QueueException(String s) { super(s); }
}
```

Vi kan er erstatte `Channel` med `BufferChannel` i spalteopdelingseksemplet og opnå at producentprocesser kommer en del foran de tilhørende forbrugerprocesser.

Vi skal nu se en anden anvendelse af `BufferChannel`.

4.6 Producent-Forbruger kommunikation

En kommunikation gennem en kanal der har begrænset bufferkapacitet kan opfattes som en producent-forbrugerkommunikation, hvor producenten producerer pakker til forbrugeren i sit eget tempo (og de kan ophobes på kommunikationskanalens buffer), ligesom producenten aftager pakkerne i det tempo, der passer ham. For at illustrere denne fortolkning lader vi producenten være brugeren, der finder på forskellige heltal. De varierende produktionstider kommer fra almindelig ubeslutsomhed. Forbrugerens opgave er at udskrive primtalsfaktoriseringen af tallene. Vi benytter en simpel algoritme, der kan give store variationer i beregningstiden. Det samlede program kan se ud som følger:

```
class Producer extends Thread {

  KeyboardReader in = new KeyboardReader();
  BufferChannel out;

  Producer(BufferChannel out) {
    this.out = out;
  }

  public void run() {
    while (true) {
      try {
        long i = in.readLong("type long integer to be factored: ");
        out.transmit(new Long(i));
      } catch (java.io.IOException e) {}
    }
  }
}

class Consumer extends Thread {

  BufferChannel in;

  Consumer(BufferChannel in) {
    this.in = in;
  }

  public void run() {
    while (true) {
      long n = ((Long)in.receive()).longValue();
      String res = n+"=";
      long p = 2;
      while (n>=p*p) {
        if (n%p==0) {
          res = res + p+"*";
          n = n/p;
        } else p = p+1;
      }
      System.out.println(res+n);
    }
  }
```

```
}

class Prim {
  static BufferChannel b = new BufferChannel(4);
  static Producer p = new Producer(b);
  static Consumer c = new Consumer(b);

  public static void main(String args[]) {
    p.start();
    c.start();
  }
}
```

I det netop viste eksempel, er der kun een producent og een forbruger, men der er ikke noget til hindring for at have flere af hver slags. Specielt med den meget primitive heltalsfaktoriseringsalgoritme, kan det nok være relevant at at have mere end en forbruger. I givet fald er det *kun* nødvendigt at ændre i hovedprogrammet og tilføje en ekstra proces, f.eks

```
class Prim {
  static BufferChannel b = new BufferChannel(4);
  static Producer p = new Producer(b);
  static Consumer c1 = new Consumer(b);
  static Consumer c2 = new Consumer(b);

  public static void main(String args[]) {
    p.start();
    c1.start();
    c2.start();
  }
}
```

4.7 Sammendrag

Sproglige konstruktioner i JAVA:

- En selvstændig proces dannes ved at udvide klassen `java.lang.Thread`, og definere en `run` metode.
- Et proces-objekt sættes igang ved metodekaldet `start()`.
- Modifier `synchronized` i samtlige metodeerklæringer for et objekt angiver at en proces skal have hele objektet for sig selv under udførelsen af en af metoderne.
- En proces der under kald af en synkroniseret metode ønsker at vente på at andre processer skal etablere en eller anden betingelse, kan gå i dvale og frigive objektet til brug for andre processer med metodekaldet `wait()`.
- En proces kan under kald af en synkroniseret metode vække evt processer der ligger i dvale ved det pågældende object med metodekaldet `notifyAll()`.

Programmeringsprincipper:

- Ressourcedeling kan administreres ved synkronisering af processer.
- Kommunikationskanaler er en nyttig abstraktion i forbindelse med multiproces systemer.
- Indviklede kontrolstrukturer kan man fordel beskrives som multiproces systemer.

5 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 "oversættelse" fra Trine til Java af en note skrevet af Erik Meineche Schmidt.

5.1 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

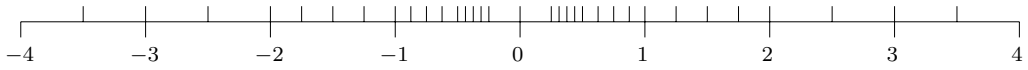
en *base* eller et *grundtal*	$\beta \in \mathcal{N} \backslash \{1\}$
et antal cifre	$s \in \mathcal{N}$
en mindste eksponent	$m \in \mathcal{Z}$
en største eksponent	$M \in \mathcal{Z}$

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 β^e kaldes *eksponentdelen*.

Et flydende-tal-system med parametrene β, 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 β^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

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

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 til-

knyttet 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:

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

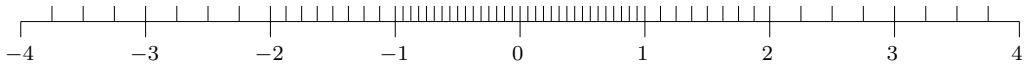
Repræsentationsfejlen ved en enkelt operation er lille, men blot man sammensætter to operationer kan fejlen blive stor:

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

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



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

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 β^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.

5.2 "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`

5.3 "Reelle" algoritmer

I dette afsnit betragtes fire eksempler på numeriske beregninger, som hver på sin måde illustrerer regning med flydende komma tal. For at præcisere hvilke talsystemer, der er på tale, skal vi anvende betegnelsen $F(\beta, s, m, M)$ om et flydende komma talsystem med parametre (β, s, m, M). De "rigtige" reelle tal er således isomorfe med et vilkårligt af systemerne $F(\beta, \infty, -\infty, \infty)$ hvor $\beta \geq 2$, og vi skal derfor for nemheds skyld betegne dem med $F(\infty)$. Vi præsenterer de fire eksempler i to forskellige endelige systemer, dels det decimale legetøjssystem $F(10, 4, -99, 99)$, og dels i JAVA's `double` system $F(2, 53, -1022, 1023)$, hvis præcision er ækvivalent med 15-16 betydende decimale cifre.

Inden gennemgangen af de egentlige eksempler illustrerer vi de to endelige systemer v.hj.a. algoritmer til beregning af grundtallet for den naturlige logaritme

$$e = 2.718281828459045\ldots$$

Vi anvender følgende fra matematikken velkendte faktum

$$\lim_{n\to\infty}(1+\frac{1}{n})^n = e$$

som umiddelbart medfører, at

$$\lim_{n\to\infty}(1+\frac{1}{\beta^n})^{\beta^n} = e \qquad (*)$$

hvor β er et heltal, der er større end 1. Hvis vi for givet β bruger betegnelsen e_n for størrelsen $e_n = (1+\frac{1}{\beta^n})^{\beta^n}$, kan vi beregne følgen $e_1, e_2, \ldots, e_n, \ldots$ v.hj.a. følgende algoritme

Algoritme: Approximation af e

Stimulans :

Respons : $e_1, e_2, \ldots, e_n, \ldots$

Metode :
```
for (n=1; true; n = n+1) {
    r = (1 + 1/β^n)
    for (j=1; j<=n; j = j+1)
        r = r * r * ... * r   (β factors)
    ≪ udskriv (n, 1/β^n, 1 + 1/β^n, r) ≫
}
```

Det er klart, at hvis denne algoritme udføres i $F(\infty)$, "udskriver" den en uendelig række værdier af approximationer til e, som bliver bedre og bedre. Nu vil enhver praktisk udførelse af algoritmen imidlertid ske i et endeligt flydende komma talsystem med nogle parametre (β, s, m, M). I så fald er algoritmens opførsel noget anderledes, hvilket illustreres af følgende resultater af kørsel i vort eksempelsystem $F(10, 4, -99, 99)$ (som altså er et decimalt system, hvor der bruges fire cifre til brøken og to til eksponenten). Vi skal antage, at der approximeres ved afskæring.

Resultatet af en kørsel i $F(10, 4, -99, 99)$ er som følger

n	$\frac{1}{10^n}$	$1 + \frac{1}{10^n}$	$(1 + \frac{1}{10^n})^{10^n}$
1	$1.000 \cdot 10^{-1}$	1.100	2.591
2	$1.000 \cdot 10^{-2}$	1.010	2.591
3	$1.000 \cdot 10^{-3}$	1.001	2.591
4	$1.000 \cdot 10^{-4}$	1.000	1.000
5	$1.000 \cdot 10^{-5}$	1.000	1.000

hvoraf det fremgår, at når n er tilstrækkelig stor, så er $1 + \frac{1}{10^n} \approx 1$, hvorefter resultatet af beregningen er 1. Vi kan yderligere se, at dette indtræffer for $n = 4$, og at det i almindelighed vil ske, når $n = s$, hvor s er antallet af betydende cifre i det flydende komma talsystem der arbejdes med. Vi kan derfor bruge algoritmen til at *illustrere* nøjagtigheden af et konkret system, hvis grundtal vi kender.

Følgende JAVA metode illustrerer nøjagtigheden af JAVA double aritmetik.

```
public static void ApproxE() {
  for (int n=1; n<=55; n=n+1) {
    double h = 1;
    for (int j=1; j<=n; j=j+1) h = 2*h;
    double r = 1+1/h;
    for (int j=1; j<=n; j=j+1) r = r*r;
    System.out.println(n+"   "+1/h+"    "+(1+1/h)+"    "+r);
  }
}
```

En kørsel af programmet resulterer i følgende

n $\frac{1}{2^n}$ $(1+\frac{1}{2^n})$ $(1+\frac{1}{2^n})^{2^n}$

```
1   0.5    1.5    2.25
2   0.25    1.25    2.44140625
3   0.125    1.125    2.565784513950348
4   0.0625    1.0625    2.6379284973666
5   0.03125    1.03125    2.6769901293781824
6   0.015625    1.015625    2.6973449525651
7   0.0078125    1.0078125    2.7077390196880198
8   0.00390625    1.00390625    2.712991624253442
9   0.001953125    1.001953125    2.715632000168995
10   9.765625E-4    1.0009765625    2.7169557294664273
11   4.8828125E-4    1.00048828125    2.7176184823368357
12   2.44140625E-4    1.000244140625    2.7179500811896298
13   1.220703125E-4    1.0001220703125    2.7181159362660465
14   6.103515625E-5    1.00006103515625    2.718198877721643
15   3.0517578125E-5    1.000030517578125    2.718240351930034
16   1.52587890625E-5    1.0000152587890625    2.71826108990388
17   7.62939453125E-6    1.0000076293945312    2.7182714591083794
18   3.814697265625E-6    1.0000038146972656    2.718276643766683
19   1.9073486328125E-6    1.0000019073486328    2.718279236118503
20   9.5367431640625E-7    1.0000009536743164    2.7182805322756565
21   4.76837158203125E-7    1.0000004768371582    2.718281180364026
22   2.384185791015625E-7    1.000000238418579    2.7182815043985844
23   1.1920928955078125E-7    1.0000001192092896    2.718281666420753
24   5.9604644775390625E-8    1.0000000596046448    2.7182817474268006
25   2.9802322387695312E-8    1.0000000298023224    2.7182817879323764
26   1.4901161193847656E-8    1.0000000149011612    2.718281808182473
27   7.450580596923828E-9    1.0000000074505806    2.718281808182473
28   3.725290298461914E-9    1.0000000037252903    2.718281808182473
29   1.862645149230957E-9    1.0000000018626451    2.718281808182473
30   9.313225746154785E-10    1.0000000009313226    2.718281808182473
31   4.6566128730773926E-10    1.0000000004656613    2.718281808182473
32   2.3283064365386963E-10    1.0000000002328306    2.718281808182473
33   1.1641532182693481E-10    1.0000000001164153    2.718281808182473
```

```
34   5.820766091346741E-11    1.0000000000582077    2.718281808182473
35   2.9103830456733704E-11    1.0000000000291038    2.718281808182473
36   1.4551915228366852E-11    1.000000000014552    2.718281808182473
37   7.275957614183426E-12    1.000000000007276    2.718281808182473
38   3.637978807091713E-12    1.000000000003638    2.718281808182473
39   1.8189894035458565E-12    1.000000000001819    2.718281808182473
40   9.094947017729282E-13    1.0000000000009095    2.718281808182473
41   4.547473508864641E-13    1.0000000000004547    2.718281808182473
42   2.2737367544323206E-13    1.0000000000002274    2.718281808182473
43   1.1368683772161603E-13    1.0000000000001137    2.718281808182473
44   5.6843418860808015E-14    1.0000000000000568    2.718281808182473
45   2.8421709430404007E-14    1.0000000000000284    2.718281808182473
46   1.4210854715202004E-14    1.0000000000000142    2.718281808182473
47   7.105427357601002E-15    1.000000000000007    2.718281808182473
48   3.552713678800501E-15    1.0000000000000036    2.718281808182473
49   1.7763568394002505E-15    1.0000000000000018    2.718281808182473
50   8.881784197001252E-16    1.0000000000000009    2.718281808182473
51   4.440892098500626E-16    1.0000000000000004    2.718281808182473
52   2.220446049250313E-16    1.0000000000000002    2.718281808182473
53   1.1102230246251565E-16    1.0    1.0
54   5.551115123125783E-17    1.0    1.0
55   2.7755575615628914E-17    1.0    1.0
```

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* og *baglæns* 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.

Følgende JAVA metode bekræfter dette.

```
public static void harmoni() {
  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);
    }
    System.out.println("\nn= " + n +
                       "\nf: " + (fsum-Math.log(n)) +
                       "\nb: " + (bsum-Math.log(n)));
  }
}
```

Af programmets resultater

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

n= 100
f: 0.5822073316515288
b: 0.5822073316515297

n= 1000
f: 0.5777155815682065
b: 0.5777155815682038

n= 10000
f: 0.5772656640681646
b: 0.5772656640682019

n= 100000
f: 0.5772206648931064
b: 0.5772206648931792

n= 1000000
f: 0.5772161649007153
```

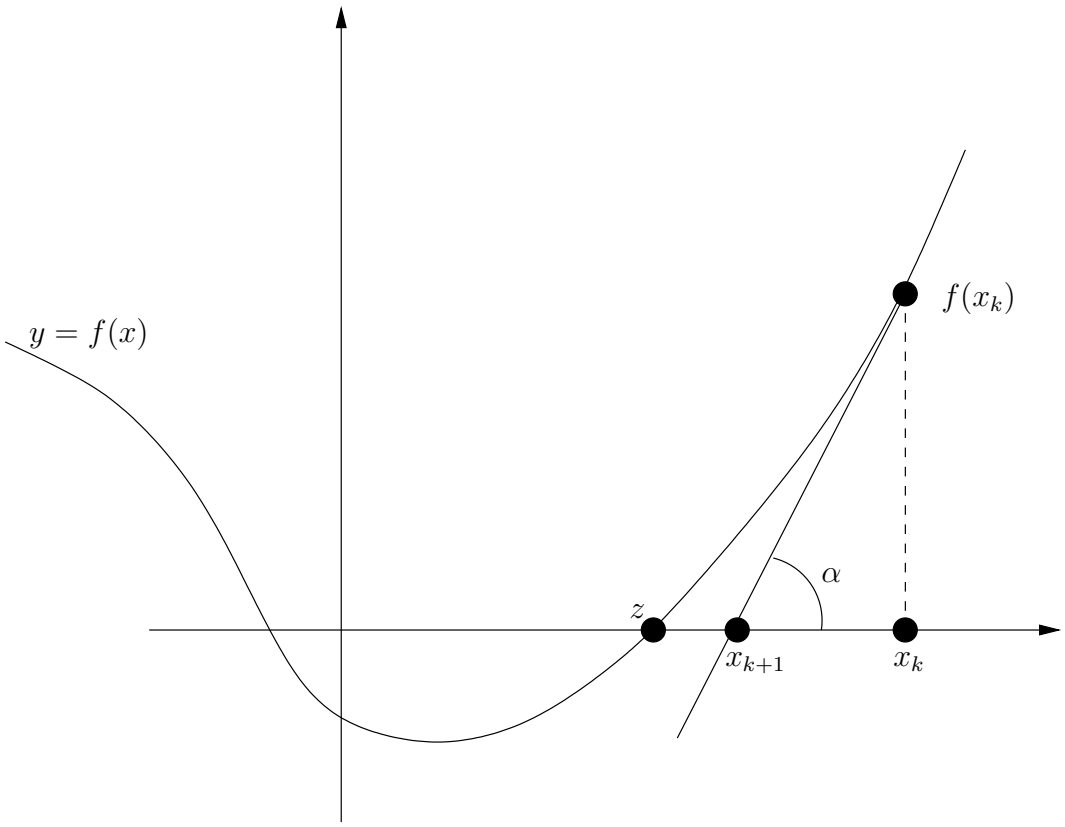
```
b: 0.5772161649014986

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

kan man konkludere, at når der udregnes summer, bør leddene så vidt muligt summeres i *voksende* orden (jfr. Shampine & Allen).

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

Newton Iteration

Som første eksempel betragter vi kvadratrodsuddragning v.hj.a. den såkaldte *Newton-iteration*. Betragt følgende graf $y = f(x)$ 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

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

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 ϵ v.hj.a. følgende algoritme

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 ϵ, dvs. når vi når et k, hvor

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

Det følger af (**), at der da gælder, at

$$\frac{x_k^2 - a}{2x_k} \leq \epsilon x_k$$

hvoraf det følger, at

$$x_k^2 \leq \frac{a}{1-2\epsilon}$$

Da konvergensen er monoton, ved vi også, at $a \leq x_k^2$, hvorfor

$$a \leq x_k^2 \leq \frac{a}{1-2\epsilon}$$

Men så følger det, at[1]

$$\sqrt{a} \leq x_k \leq \frac{\sqrt{a}}{\sqrt{1-2\epsilon}} \approx \frac{\sqrt{a}}{1-\epsilon} \approx \sqrt{a}(1+\epsilon)$$

dvs. at

$$0 \leq x_k - \sqrt{a} \leq \epsilon\sqrt{a}.$$

Sagt med andre ord har vi vist, at når algoritmen standser (fordi den relative afstand mellem x_{k+1} og x_k er mindre end ϵ), så er den relative afstand mellem x_k og $\sqrt{a}$ også mindre end ϵ.

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 ϵ vælges mindre end β^{-s}, så er kravet

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

[1]Her har vi brugt, at når ϵ er så lille, at $\epsilon^2 << \epsilon$, gælder der at $\sqrt{1+\epsilon} \approx 1 + \frac{\epsilon}{2}$ og $\frac{1}{1+\epsilon} \approx 1 - \epsilon$, hvor $\approx$ betegner "næsten lig med".

i $F(\beta, s, m, M)$ ækvivalent med

$$xg - xn > 0$$

fordi β^{-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 ϵ (som for at have mening så skulle være $> \beta^{-s}$). Det er også klart, at valget af et sådant ϵ vil påvirke antallet af iterationer (lille ϵ, 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 ϵ.

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 følgende 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.

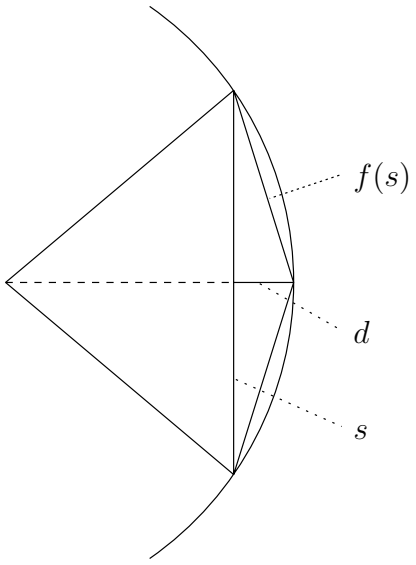
Gode og dårlige formler

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 π, 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 2π (omkredsen af enhedscirklen er som bekendt 2π). Algoritmen ser ud som følger

Algoritme: Beregning af π
Stimulans :
Respons : r: approximation til π
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 π 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
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

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 π 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 π 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
```

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

- rækker skal om muligt summeres i numerisk voksende 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
- matematiske udtryk bør så vidt muligt omskrives med henblik på at minimalisere ciffertab

- fejltolerancer skal vælges med omhu.

Sammenfattende kan det siges, at man skal udvise ekstrem varsomhed ved numeriske beregninger og altid være på vagt over for mulige fejlkilder. Det mest illustrative eksempel på hvor lumske numeriske beregninger kan være, er nok programmet PiWorst, som *konvergerer* (midlertidigt) i *fuld* maskinnøjagtighed – men mod et forkert resultat. At konvergensen er midlertidig ses af, at hvis iterationen fortsættes, bliver resultaterne som følger

```
3.0    12
3.1058285412302498    24
3.132628613281237    48
3.139350203046872    96
3.14103195089053    192
3.1414524722856703    384
3.141557607912925    768
3.141583892148936    1536
3.1415904632367617    3072
3.1415921059596714    6144
3.1415925165881546    12288
3.1415926186407894    24576
3.1415926453212157    49152
3.1415926666655567    98304
3.141592730698579    196608
3.1415929868306565    393216
3.141593669849427    786432
3.1415923038117377    1572864
3.1416086962248038    3145728
3.1415868396550413    6291456
3.1416742650217575    12582912
3.1416742650217575    25165824
3.1430727401700396    50331648
3.137475099502783    100663296
3.092329219213245    201326592
3.3541019662496847    402653184
4.242640687119286    805306368
```

Til sammenligning ses her udviklingen af kørsler af PiBest.

```
3.0    12
3.105828541230249    24
3.1326286132812378    48
3.1393502030468667    96
3.1410319508905093    192
3.1414524722854615    384
3.1415576079118575    768
```

```
3.1415838921483177    1536
3.1415904632280496    3072
3.141592105999271    6144
3.1415925166921563    12288
3.141592619365383    24576
3.1415926450336897    49152
3.1415926514507664    98304
3.141592653055036    196608
3.1415926534561036    393216
3.141592653556371    786432
3.141592653581438    1572864
3.1415926535877046    3145728
3.1415926535892713    6291456
3.141592653589663    12582912
3.141592653589761    25165824
3.1415926535897856    50331648
3.1415926535897913    100663296
3.1415926535897936    201326592
3.141592653589794    402653184
3.141592653589794    805306368
```