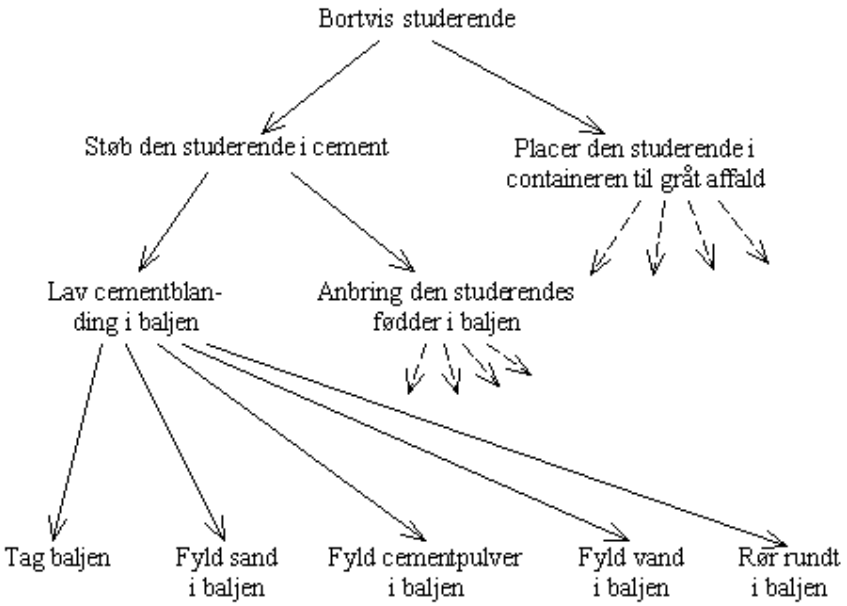
Metoder

Sidst ændret: 07/19/2018 14:11:05

Opgaver

I kapitlet om algoritmer blev det i pedel-eksemplet illustreret, hvordan man kan nedbryde en algoritme i mindre del-algoritmer. Hidtil har vi brugt denne teknik ved at betragte det nederste niveau som vores program og kun brugt nedbrydningen til at styre sammenhængen.

Hvis vi tegner pedel-eksemplet op i følgende struktur



Figur 1: *Nedbrydning i pedel-eksemplet*

genkender man først nederst, det sekventielle forløb på mål-niveauet. Hvor mål-niveauet er det niveau, hvor vi har nået veldefinerede skridt for den der skal udføre algoritmen.

Under selve udviklingen af algoritmen bruger vi strukturen ovenfor, men i programmet optræder den ikke. Dér, er det kun mål-niveauet vi anfører.

Man skal ikke lave særlig store programmer før den sekventielle linie af kode bliver en uhåndterlig og uoverskuelig masse.

Problemet med manglende overskuelighed kunne måske afhjælpes ved at give hver delalgoritme en overskrift, en kommentar, der stammer fra niveauet over det laveste niveau. Dette kunne man gentage for hvert niveau videre op, og med passende indrykninger få det til at se nogenlunde ud. Pseudo-kode der skitserer nedbrydningen fra figur 1 ville i så fald få følgende udseende:

Figur 2: *Struktur med*

```
/*
 *   Bortvis studerende
 */
```

overskrifts-kommentarer

```
   /*
    *   Støb den studerende i cement
    */

      /*
       *   Lav cementblanding i baljen
       */

         Tag baljen;
         Fyld sand i baljen;
         Fyld cementpulver i baljen;
         Fyld vand i baljen;
         Rør rundt i baljen;

      /*
       *   Anbring den studerendes fødder i baljen
       */

         ...
         ...

   /*
    *   Placer den studerende i containeren til gråt affald
    */

      ...
      ...
```

Hvis algoritmen er meget stor, bliver det stadig uoverskueligt, da del-algoritmer på samme niveau vil blive spredt over store afstande. Vi har brug for andre teknikker, som bedre løser problemet.

Copy/Paste

I pedel-eksemplet er det ikke særlig tydeligt, men i programmer er man ofte ude for at skulle bruge den samme del-algoritme flere gange, evt. med mindre ændringer. Dette skulle i så fald ske ved at den samme kode blev kopieret rundt til de steder hvor den bruges, med en teksteditor (såkaldt copy-paste-programmering - se evt. copy-paste anti-pattern).

Ét sted

Løsningen er, at disse stykker kode kun skal findes ét sted. Og at de delalgoritmer i træet der nedbryder, ved at bruge dette stykke kode, gør det ved at henvise til det, og ikke ved at gentage det.

Funktions-struktur

Denne fremgangsmåde kræver en støtte fra programmerings-sproget. Java, og mange andre sprog, har en funktionsstruktur der bruges til at indkapsle en delalgoritme, og som ved reference fra et andet sted i programmet kan udføres.

1. Simpel metode

Lad os se et eksempel:

Source 1: *Simpel metode*

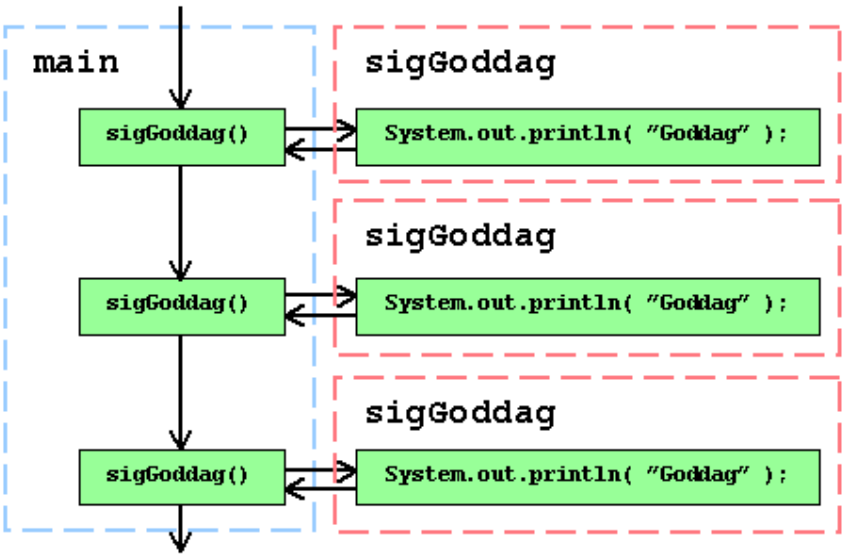
```
class MetodeEksempel_1 {

  static void sigGoddag() {
    System.out.println( "Goddag" );
  }

  public static void main( String[] argv ) {
    sigGoddag();
    sigGoddag();
    sigGoddag();
  }
}
```

```
Goddag
Goddag
Goddag
```

`static` og `void`

Lad os starte med en række kommentarer til hvad der står. Laver man en metode er der en lang række muligheder - mange ting man kan angive og "indstille". Nogle af disse skal anføres selv om de blot er en angivelser af, at der ikke rigtig skal være "noget". Sådan to stykker "dødt væv" er der i eksemplet ovenfor, og dem vil vi i første omgang vælge at overse for ikke at komplicere det hele. Det drejer sig om ordet **`static`**, der har at gøre med de objektorienterede egenskaber ved metoden. Vi vil ikke under den grundlæggende programmering arbejde med disse muligheder, så skriv blot **`static`** på dette sted i enhver metode indtil videre. Dernæst er der ordet **`void`** som vi kommer ind på senere i dette kapitel.

For bedre at forstå hvad der sker i eksemplet ovenfor, kan man tegne forløbet i et diagram:



Figur 3: *Simpel metode*

Metodekald

I **`main`** står der tre ens sætninger. De består hver af et navn på en metode ("**`sigGoddag`**") efterfulgt af en tom parantes. Dette er metode-kald. Et metodekald angiver, at det der er specificeret i den til navnet hørende metode skal udføres på dette sted i algoritmen. I figuren ovenfor kan vi se, at de tre kald realiserer tre **`System.out.println`**-sætninger. Man skal se på det, som om der bliver lavet *tre kopier* af metoden **`sigGoddag`**, og at disse tre kopier bliver udført efter hinanden, pga. de tre metodekald i **`main`**.

Om man fra starten af får den forståelse, at det er en kopi af metoden der udføres, har stor betydning for, hvor let eller vanskeligt man får ved at forstå metode-begrebet (Dette gør sig i særdeleshed gældende, når vi skal se på rekursive metoder).

2. Lokale variable

I eksemplet ovenfor var der tale om en gentagelse af det samme tre gange. Lad os forfølge denne iteration lidt, og se hvordan man kunne lave den lidt smartere.

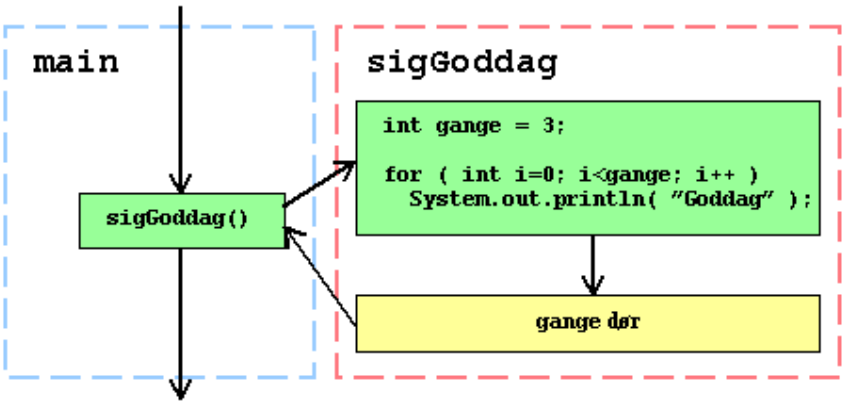
Source 2: *En lokal*

```
class MetodeEksempel_2 {

  static void sigGoddag() {
```

variabel

```
    int gange = 3;

    for ( int i=0; i<gange; i++ )
      System.out.println( "Goddag" );
  }

  public static void main( String[] argv ) {
    sigGoddag();
  }
}
```

Her erklæres der en variabel **`gange`** inde i metoden, men inden vi ser på betydningen af en sådan variabel, vil vi igen tegne et diagram der illustrerer hvad der sker:



Figur 4: *En lokal variabel*

Når vi bruger en gul farve, er det fordi der ikke er tale om en del af kildeteksten. Grunden til at vi indfører et sådant gult felt er at der sker noget specielt med en variabel i en metode når metoden er færdig.

En variabel der erklæres i en metode kaldes en lokal variabel. Den tilhører metoden og kan ikke bruges af andre "udefra". "Udefra" vil sige alle andre steder end i metode-kroppen. Det er f.eks. ikke muligt i **`main`** at tildele **`gange`** en værdi eller se hvad dens værdi er. Det er i virkeligheden meget logisk at det forholder sig sådan, fordi selve metoden ikke eksisterer før den bliver kaldt, og ej heller efter kaldet. Husk på, at man skulle tænke på et metodekald som om der blev lavet en kopi af metoden, som så blev udført, og kopien eksisterer netop kun ved udførelsen og hverken før eller efter.

Metodens hoved og krop

Der tales ovenfor om metodens "egen kode". Den kode der er omkranset af tuborg-paranteserne kaldes metodens **krop** (eng. *body*). Tilsvarende kalder man den første linie med navnet og paranteserne, lige før metode-kroppen, for **metode-hovedet**.

Hvad menes der med at **`gange`** dør? Det betyder at den lokale variabel lever inde i kopien af metoden, og at dens eksistens hænger nøje sammen med metode-kopiens eksistens. Derfor forsvinder den lokale variabel sammen med metoden når kaldet afsluttes - når metoden terminerer.

3. Parametre

En af fordelene ved en metode er at man kan få udført det samme flere gange ved at kalde den samme metode. Vi så det, i det første eksempel, hvor der blev sagt "Goddag" tre gange, fordi metoden blev kaldt tre gange. I det andet eksempel udvidede vi i realiteten blot metoden var at flytte noget af funktionaliteten fra **`main`** over i **`sigGoddag`**. Men hvad hvis man ikke vil have der skal ske *præcist* det samme hver gang man kalder metoden? hvad hvis man, f.eks. vil variere hvor mange gange der siges "Goddag" for hvert af kaldene?

Lad os se hvordan det kunne gøres:

Source 3:
En parameter

```
class MetodeEksempel_3 {

  static void sigGoddag( int gange ) {
    for ( int i=0; i<gange; i++ )
      System.out.println( "Goddag" );
  }

  public static void main( String[] argv ) {
    sigGoddag( 3 );
  }
}
```

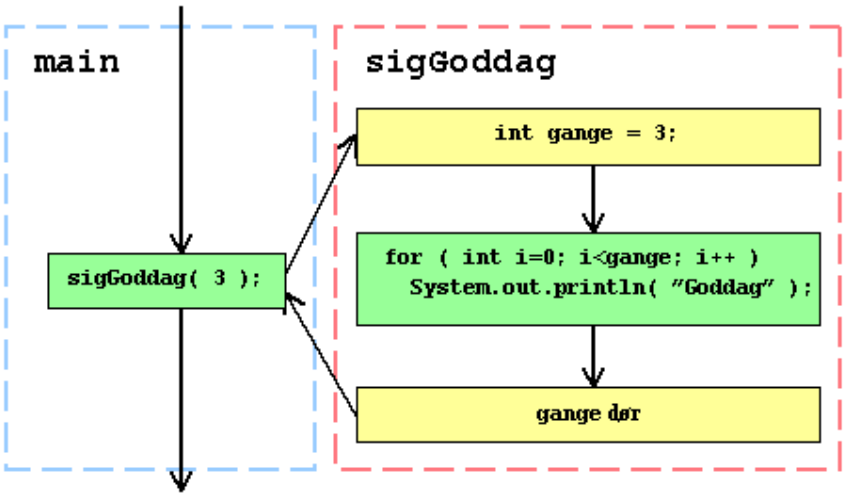
Her anvendes der en såkaldt **parameter** til at overføre en værdi til metoden. Erklæringen af **`gange`** optræder nu oppe mellem paranteserne i metode-hovedet, men anvendes stadig i selve metode-kroppen. Værdien **`3`** er her placeret i selve kaldet, hvor den optræder mellem paranteserne.

Formelle og aktuelle parametre

Alt hvad der optræder mellem paranteserne i kaldet og paranteserne i metode-hovedet kaldes parametre, men for at kunne skelne mellem parametre, der står disse to steder, kalder man de parametre der findes i metode-hoveder: **formelle parametre**, og dem der optræder i kaldene: **aktuelle parametre**. Her er heltals-literalet 3 en aktuel parameter - den har en konkret værdi, mens **`gange`** er en formel parameter - dens værdi afhænger af den aktuelle parameter.

Lad os tegne hvad der sker i eksemplet:

Figur 5:
En parameter



Parameter-overførsel

Her dør **`gange`** på samme måde som en lokal variabel. Der gælder, at parametre er "ligesom" lokale variable, men de kan lidt mere. Parametre kan bruges til at kommunikere med metoden. Værdien **`3`** der optræder i kaldet bliver assignet **`gange`** før metoden går i gang - det sker i selve metode-kaldet. Dette assignment kalder man **parameter-overførslen**. Overførslen er på ingen måde mystisk, den er fuldstændig som ethvert andet assignment, blot ser den anderledes ud - ikke andet!

Med parametre har vi nu mulighed for at overføre værdier fra den kode der kalder, til den kode, der er i metode-kroppen. Vi har en kommunikationskanal. Men hvad hvis vi vil overføre mere end en værdi?

3.1 Mere end én parameter

Lad os se et eksempel, hvor vi overfører to ting på en gang:

Source 4:
To parametre

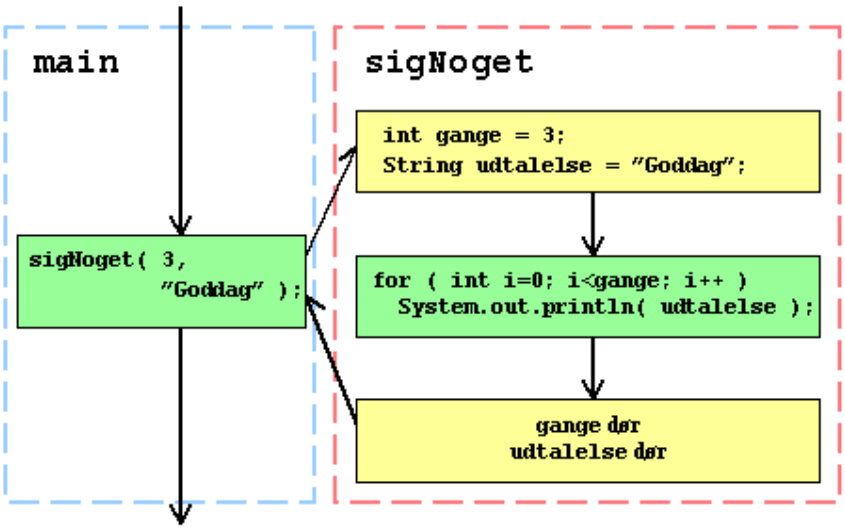
```
class MetodeEksempel_4 {

  static void sigNoget( int gange, String udtalelse ) {
    for ( int i=0; i<gange; i++ )
      System.out.println( udtalelse );
  }

  public static void main( String[] argv ) {
    sigNoget( 3, "Goddag" );
  }
}
```

Først bemærker man at metoden har skiftet navn. Nu er det ikke længere sikkert der siges "Goddag", så et mere generelt navn er på sin plads.

Vi har nu to formelle parametre **`gange`** og **`udtalelse`**, og i kaldet tilsvarende to aktuelle parametre **`3`** og **`"Goddag"`**. Man observer at de typemæssigt passer sammen i samme rækkefølge som de står, og parameter-overførslen sker da også efter denne rækkefølge.

Lad os endnu engang illustrere forløbet:

Figur 6:
To parametre



Her bliver parameter-overførslen til to på hinanden følgende assignments.

3.2 Arrays som parametre

Hvordan overfører man et array som parameter? Det er meget enkelt. Som for alle andre parametre foregår det med et assignment fra den aktuelle til den formelle parameter. Lad os se et eksempel:

Source 5:
Parameter-overført array

```
class MetodeEksempel_5 {

  static void udskrivArray( int[] array ) {
    for ( int i=0; i<array.length; i++ )
      System.out.print( array[i] + " " );
    System.out.println();
  }

  public static void main( String[] argv ) {
    int[] primtal = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
```

```
    udskrivArray( primtal );
  }
}
```

```
2 3 5 7 11 13 17 19 23
```

Det assignment der udføres ved parameter-overførslen er:

```
int[] array = primtal
```

hvorefter metode-kroppen kan arbejde med arrayet som med ethvert andet array.

3.3 Gentaget typeangivelse

Hvis man kender lidt til andre programmeringssprog, er der en lille syntaktisk detalje man måske gerne vil have afklaret: Skal man nødvendigvis gentage typen hvis to, eller flere, på hinanden følgende formelle parametre har samme type?

Lad os først se et eksempel hvor typen er angivet to gange:

Source 6: *dobbel type-angivelse*

```
static void gørNoget( int antal, int sum ) {
  ...
}
```

Spørgsmålet er, om man i stedet kunne skrive:

Source 7: *enkel type-angivelse*

```
static void gørNoget( int antal, sum ) {
  ...
}
```

Svaret er ganske enkelt nej - det kan man ikke!

4. Returnering

I de eksempler vi hidtil har set, har den ønskede "effekt" af metode-kaldet været, at noget er "sket". Resultatet har været en udskrift på skærmen.

Hvad hvis man i stedet ønskede at resultatet skulle være en værdi - et resultat som den der foretager kaldet efterfølgende kunne bruge til noget? Lad os se et eksempel:

Source 8: *returnering af værdi*

```
class MetodeEksempel_6 {

  static int minimum( int tal1, int tal2 ) {
    int mindst;

    if ( tal1 < tal2 )
      mindst = tal1;
    else
      mindst = tal2;

    return mindst;
```

```
  }

  public static void main( String[] argv ) {
    int mindst = minimum( 3, 5 );
    System.out.println( "Den mindste værdi er " + mindst );
  }
}
```

```
Den mindste værdi er 3
```

Lad os først se på det som vi allerede kan forstå i dette eksemepel, og dernæst studere det nye.

Metoden hedder **`minimum`** og har to formelle parametre **`tal1`** og **`tal2`**, der er kan antage almindelige tal-værdier. I **`main`** kaldes der med de aktuelle parametre **`3`** og **`5`**. Der er en lokal variabel **`mindst`** som bruges til at finde den mindste værdi af **`tal1`** og **`tal2`**. En if-sætning i metode-kroppen afgør om det er **`tal1`** eller **`tal2`** der er mindst, og værdien af den mindste placeres i den lokale variabel.

Så er der det nye! I metode-hovedet er **`void`** nu erstattet med **`int`**. Sidste i metode-kroppen er der en sætning med **`return`**, og endelig er kaldet i **`main`** placeret som om det var et udtryk der blev evalueret til en **`int`**.

En værdi tilbage fra metoden

Det hele drejer sig om at få en værdi ud fra metoden og tilbage til det sted hvor metoden bliver kaldt. I metode-hovedet angives det med en type-angivelse før metode-navnet, hvad det er for noget der returneres. I dette tilfælde er det en **`int`**. return-sætningen der afslutter metode-kroppen bevirker at metoden terminerer og returnerer dét som fås ved at evaluere udtrykket, der står lige efter ordet **`return`**. I dette tilfælde er udtrykket næsten så simpel som det kunne være, det er kun et variabel-navn og resultatet er blot dens værdi.

Vi kan altså med en return-sætning bestemme hvad der skal returneres fra en metode, vi skal bare huske at anføre typen på det vi returnerer lige før metode-navnet. Men hvad sker der når værdien kommer tilbage til det sted hvor der blev kaldt - hvem tager imod den?

Det fungerer på den måde at selve kaldet bliver behandlet som et udtryk der evalueres til den værdi der returneres. På den måde vil kald kunne optræde alle steder hvor et udtryk kan stå - og det er mange! I vores eksempel er det som højresiden i et assignment, hvor vi "griber" den returnerede værdi i en variabel.

Lad os se et lidt større eksempel, der anvender returnering i forbindelse med metoder:

Source 9:
Eksempel der anvender returnering

```
import java.io.*;

class MetodeEksempel_7 {

  static int indlæsTal( String prompt ) throws IOException {
    BufferedReader indlæser =
      new BufferedReader(
        new InputStreamReader( System.in ) );

    System.out.print( prompt + ": " );

    int tal = Integer.parseInt( indlæser.readLine() );

    return tal;
  }

  static int minimum( int tal1, int tal2 ) {
    int mindst;

    if ( tal1 < tal2 )
      mindst = tal1;
```

```
    else
      mindst = tal2;

    return mindst;
  }

  public static void main( String[] argv ) throws IOException {
    int tal1 = indlæsTal( "Indtast det første tal" );
    int tal2 = indlæsTal( "Indtast det andet tal" );

    int mindst = minimum( tal1, tal2 );

    System.out.println( "Det mindste tal er: " + mindst );
  }
}
```

```
Indtast det første tal: 9
Indtast det andet tal: 5
Det mindste tal er: 5
```

Her har vi en metode **`indlæsTal`**, der kan bruges til at indlæse tal fra brugeren. Når vi har fået de to tal-værdier vi ønsker, bruger vi vores **`minimum`**-metode fra før, til at finde den mindste værdi, som vi til slut udskriver på skærmen.

Man kan kun returnere én værdi

Det bliver ofte efterlyst hvordan man kan returnere mere end én værdi. Det kan man ganske enkelt ikke! Vi skal senere se med objekter, at det er muligt at wrappe flere værdier sammen, så man kan omgå denne begrænsning, men det er i realiteten ikke så tit, det giver de store problemer med begrænsningen på én værdi. Man kan dog bemærke, at det giver en ulighed i forbindelse med kommunikationen med metoden. Det er muligt at send dén flere værdier, men man kan kun modtage én tilbage!

5. Overloading

Samme navn, men "forskellige"

Overloading er, når to metoder har samme navn, men man alligevel kan kende forskel på dem, og i kaldet vælger hvilken der skal bruges.

Umiddelbart lyder det mærkeligt, for er det ikke netop navnet vi bruger til at identificere metoden? Jo, delvist!

Der er også noget andet i kaldet som identificerer metoden, nemlig antal og typen af parametre. Lad os se et eksempel, hvor vi laver en fejl i kaldet:

Source 10:
Fejl-kald

```
class MetodeEksempel_8 {

  static int minimum( int tal1, int tal2 ) {
    int mindst;

    if ( tal1 < tal2 )
      mindst = tal1;
    else
      mindst = tal2;

    return mindst;
  }

  public static void main( String[] argv ) {
    int mindstAf3 = minimum( 6, 2, 5 );
  }
}
```

```
---------------------------- Compiler Output ----------------------------
test.java:15: Wrong number of arguments in method.
    int mindstAf3 = minimum( 6, 2, 5 );

1 error
```

Igen har vi vores **`minimum`**-metode fra før, men nu kalder vi den med tre aktuelle parametre, og compileren vil brokke sig højlydt, da **`minimum`** kun har to formelle parametre. Med andre ord, det er ikke kun navnet der kan bruges til at identificere metoden - parametrene skal også passe!

Det er netop parametrene der giver os mulighed for at have flere metoder med samme navn. Eftersom **`main`** gerne vil finde minimum af tre tal, så lad os lave endnu en metode med samme navn, men med tre parametre:

Source 11: **`minimum`** *overloaded*

```
class MetodeEksempel_9 {

  static int minimum( int tal1, int tal2 ) {
    int mindst;

    if ( tal1 < tal2 )
      mindst = tal1;
    else
      mindst = tal2;

    return mindst;
  }

  static int minimum( int tal1, int tal2, int tal3 ) {
    int mindst;

    mindst = minimum( tal1, tal2 );
    mindst = minimum( mindst, tal3 );

    return mindst;
  }

  public static void main( String[] argv ) {
    int mindstAf2 = minimum( 3, 5 );
    System.out.println( "Den mindste af de to er " + mindstAf2 );

    int mindstAf3 = minimum( 6, 2, 5 );
    System.out.println( "Den mindste af de tre er " + mindstAf3 );
  }
}
```

```
Den mindste af de to er 3
Den mindste af de tre er 2
```

Compileren skal finde ud af det

Her har vi angivet to kald i **`main`**. Det første kald vil compileren knytte til den *første* forekomst af **`minimum`**-metoden, da den passer med navn og parametre. Det andet kald vil i stedet blive knyttet til den *anden* forekomst af **`minimum`**-metoden, da den passer med de tre parametre. Med andre ord, det hele afhænger af at compileren kan finde ud af præcist hvilken metode der kaldes.

Retur-typen?

Når vi nu er så godt i gang med at finde ting der kan bruges til at identificere hvilken metode der kaldes; hvad så med retur-typen - den type vi angiver lige før metode-navnet? I vores eksempel gemmes værdien der returneres fra kaldet i **`int`**-variable, så spørgsmålet er, om man ikke kan skelne metoder fra hinanden, selv om de hedder det samme og har samme formelle parametre, blot de har forskellig retur-type?

I vores eksempel er det rimelig ligetil at gøre det, men situationen kunne nemt være en anden. Hvad hvis vi valgte at ignorere det der blev returneret? Hidtil har vi pænt modtaget det returnerede med en variabel med passende type. Lad os se et eksempel hvor vi ignorerer det returnerede:

Source 12: *Kald der ignorerer returnering*

```
class MetodeEksempel_10 {

  static int minimum( int tal1, int tal2 ) {
    ...
  }

  public static void main( String[] argv ) {
    minimum( 3, 5 );
  }
}
```

Retur-typen kan ikke bruges

Her kalder vi **`minimum`**, men af uvis hvilken grund ønsker vi blot at metoden udføres, vi er ikke interesseret i den værdi der returneres! Nu er det ikke muligt for compileren at skelne metoden fra en eventuel anden, med samme navn og formelle parametre, men forskellig returtype. Bla. derfor har man vedtaget at returtypen ikke kan bruges til overloading.

5.1 Mindste tal i array

Bygge på simpel løsning

Ovenfor så vi, hvordan man kunne anvende **`minimum`**-metoden med to parametre til at implementere den overloadede metode med tre parametre. Dette at lave en løsning på et simpelt problem, og anvende det til at løse et mere kompliceret problem, kan være kraftfuldt i sin enkelthed; hvilket vi vil se endnu et eksempel på.

Vi vil overloade med endnu en **`minimum`**-metode: én der finder det mindste tal i et array!

Idéen er inspireret af løsningen med de tre parametre: at arbejde med et foreløbigt minimum:

Source 13: *Iterativ anvendelse af* **`minimum`**-*metode*

```
static int minimum( int[] tabel ) {
// PRE: tabel.length > 0

  int mindst = tabel[0];

  for ( int i=1; i<tabel.length; i++ )
    mindst = minimum( mindst, tabel[i] );

  return mindst;
}

public static void main( String[] argv ) {
  int[] t = { 4, 8, 6, 6, 4, 9, 8, 3, 6, 4, 5, 2 ,9 };

  System.out.println( "mindste tal: " + minimum( t ) );
}
```

Vores udgangspunkt er array'ets første element. Vi arbejder os dernæst fremad ved hele tiden at finde minimum af vores foreløbige minimum og det næste element i array'et.

6. Faktorisering af kode

Matematik

Man bruger ofte metoder til at faktorisere kode. Betegnelsen stammer fra matematikken; hvor man tit ser følgende omformning foretaget i forbindelse med matematiske udtryk:

```
A*B + A*C = A*(B + C)
```

Kortere udtryk

Vi sætter **A** udenfor parantes - vi faktoriserer **A**. Da man sætter **A** udenfor parantes, taler man også om at **udfaktorisere A**. Formålet med at udfaktorisere **A**, er at **A** nu kun optræder én gang i udtrykket på højre side. I matematikken kan faktorisering, kombineret med andre teknikker, bidrage til at gøre udtryk kortere og simplere - såkaldt "formel-gymnastik".

Kortere kildetekst

I forbindelse med metoder kan faktorisering bidrage til at gøre kildeteksten kortere og lettere at overskue. Jeg ser med jævne mellemrum projekter, hvor kildeteksten kunne forkortes med flere sider; hvis de studerende ellers havde været opmærksom på det.

Hvad vil det sige at udfaktorisere kode? Lad os se et generelt eksempel:

Source 14: *Før faktorisering*

```
static void f() {
  aaa;
  bbb;
  ccc;
}

static void g() {
  ddd;
  bbb;
  ccc;
  eee;
}
```

De enkelte linier: **aaa**, **bbb** osv, står for sætninger. Deres konkrete indhold er uvæsentligt i denne sammenhæng - de repræsenterer blot hver deres sætning. Man bemærker at sætningerne **bbb** og **ccc** optræder i både **f** og **g**. Vi kan udfaktorisere disse sætninger ved at indføre en ny metode:

Source 15: *Efter faktorisering*

```
static void f() {
  aaa;
  h();
}

static void g() {
  ddd;
  h();
  eee;
}

static h() {
  bbb;
  ccc;
}
```

Metoden **h** indeholder nu de fælles sætninger: **bbb** og **ccc**. Hvor disse sætninger tidligere stod i **f** og **g**, er der nu et kald af **h**. Man kan lidt abtrakt sige, at vi har sat sætningerne "udenfor parantes" - udenfor **f** og **g**.

Hvor meget er sparet?

Ovenstående eksempel er naturligvis noget abstrakt. Antallet af linier man kan faktorisere i virkelige situationer vil ofte være betydelig større end to, og virkningen vil derfor også blive større. I eksemplet sparer vi rent faktisk ingen linier, da de indførte metode-kald opvejer det sparede. Havde der i stedet været tale om flere linier, ville besparelsen nærme sig 50%. Havde linierne fra **h** stået tre steder ville besparelsen nærme sig 66%, fire steder 75% osv.

Kode-redundans

I forbindelse med faktorisering af kode, taler man om, at man fjerner redundans - **kode-redundans**. Det er et emne vi vil vende tilbage til i kapitlet "*Klasser*" under "*Objektorienteret Programmering*".

7. Virkefelter

[Generelt om virkefelter (har allerede nævnt det i forbindelse med for-sætning + lokale variable)]

8. Signaturen

Inden vi går videre, er der et begreb der vil være på sin plads at få defineret. Det er **signatur-begrebet**! Hvad er en signatur? Alle metoder har en signatur og den er givet ved:

Definition: Signatur

En metodes signatur er givet ved:

- Retur-typen
- Navnet
- Parametrenes type

Disse tre indeholder til sammen alle nødvendige oplysninger for at kunne foretage et kald. Selve kaldet baserer sig på navnet og parametrenes type, mens returneringen - hvis man er interesseret i den - skal passe med returtypen.

Hvad med metode-hovedet - er "signatur" ikke det samme som metode-hovedet? Næsten, men ikke helt. Ud over `static` som vi jo lader ligge til den objektorienterede programmering, er der parametrenes navne. Disse navne er nemlig evig ligegyldige når vi foretager kaldet! De er ikke kendetegnende for metoden, da man kan ændre dem uden at det betyder noget for nogen af de kald man eventuelt allerede måtte have lavet rundt omkring.

Repetitionsspørgsmål

1 Hvad er mål-niveauet for nedbrydning?

2 Hvordan kan man med overskriftskommentarer vise nedbrydning i kildeteksten?

3 Hvad er copy-paste programmering?

4 Hvad er lokale variable?

5 Hvornår terminerer en metode?

6 Hvad menes der med at en lokal variabel dør?

7 Hvad er et metode-hoved og en metode-krop?

8 Hvad er forskellen på en parameter og en lokal variabel?

9 Hvad er formelle og aktuelle parametre?

10 Parameteroverførsel kan sammenlignes med en anden operation i Java, hvilken?

11 Hvis man har flere parametre, hvad bruger man så til at adskille dem med?

12 Hvordan fungerer parameteroverførsel af arrays?

13 Skal man nødvendigvis gentage typen hvis to, eller flere, på hinanden følgende, formelle parametre har samme type?

14 Hvad angiver returtypen og hvor skal den placeres?

15 Kan man returnere mere end én værdi fra en metode?

16 Hvad sker der når en return-sætning udføres?

17 Hvad bruger compileren til at skelne mellem metoder, der har samme navn?

18 Hvorfor kan returtypen ikke bruges i forbindelse med overloading?

Svar på repetitionsspørgsmål

1 Det niveau hvor hvert skridt er veldefineret for den der skal udføre algoritmen.

2 Man laver en overskriftskommentar for hver delalgoritme, og laver en indrykning for hvert niveau, så delalgoritmer på samme nedbrydningsniveau har samme indrykning.

3 Når man kopierer kildetekst rundt i programmet, til steder hvor der skal gøres det samme, eller næsten det samme.

4 Det er variable der er erklæret i metode-kroppen og som kun findes der. De kan ikke tilgås fra andre steder end metode-kroppen.

5 En metode terminerer når sidste linie i metode-kroppen udføres, eller når en return-sætning udføres.

6 De lever og dør sammen med den kopi af metoden som udføres i forbindelse med et kald.

7 Metode-kroppen er den kildetekst der står mellem metodens tuborg-paranteser. Metode-hovedet er det der står før metode-kroppen.

8 En parameter kan bruges til at sende oplysninger til metoden. Bortset fra dette er parametre og lokale variable det samme.

9 Formelle parametre optræder i metode-hovedet, de aktuelle i metode-kaldet.

10 Assignment.

11 Komma

12 Fuldstændig som enhver anden form for parameteroverførsel - som et assignment.

13 Ja.

14 Returtypen angiver typen på det der returneres og skal placeres lige før metode-navnet.

15 Nej

16 Metoden terminerer og resultatet af at evaluere det, der står efter ordet **`return`**, returneres til det sted hvor der blev kaldt.

17 Parametrenes type og dermed implicit også deres antal.

18 Fordi man kan ignorere det der returneres. Derfor kan compileren ikke altid være sikker på, at der er informationer nok til at skelne mellem metoderne.