© 1999-2003, Flemming Koch Jensen
Alle rettigheder forbeholdt

Simpel programmering

Sidst ændret: 07/19/2018 14:11:06
Opgaver

"Der er ingen grund til, at noget menneske skulle have en computer i hjemmet"

- Ken Olson dir., DEC, 1977

Vi starter naturligvis med "Hello world", men går hurtigt videre til et andet eksempel. Kommentarer og deres vigtighed introduceres. Dernæst ser vi lidt på lagerforståelse i forbindelse med assignment. Simpel input og output eksemplificeres. De tre betegnelser identifiers, reserverede ord og literaler introduceres. De tre forskellige fejltyper efterfølges af en række formaninger om gode kodevaner.

Det har efterhånden længe været en tradition blandt lærebøger i programmering at det første eksempel, har skrevet teksten: "Hello World", på skærmen. Lad os derfor ikke bryde traditionen men ile med dette eksempel i Java:

Source 1:
Traditionel Hello World

```
class HelloWorld {
  public static void main( String[] argv ) {
    System.out.println( "Hello World" );
  }
}
```

```
Hello World
```

Som sagt, er det en tradition, og mere er det ikke værd, for det er i virkeligheden et temlig ringe eksempel at starte med. Lad os derfor i stedet indlede med et andet:

Source 2:
Censor program

```
/* Censor program
Det første rigtige
eksempel */

class Tillykke {
  public static void main( String[] argv ) {
    int karakter;

    // karakteren er altid 12
    karakter = 12;
    System.out.println( "Tillykke, De bestod med " + karakter );
  }
}
```

```
Tillykke, De bestod med 12
```

Se, det var jo straks et mere opmuntrende eksempel.

Suffix `java`

Når man vil lave et program, som censor-programmet ovenfor, skal man skrive kildeteksten i en tekst-file og gemme den med det navn som man får ved først at tage det ord der står efter **`class`** (i dette tilfælde **`Tillykke`**) og give det suffix **`java`**, altså **`Tillykke.java`**.

Overse visse

For at lave selv det mindste program er man nød til at skrive en del tekst. Vi vil i første omgang

detaljer indtil videre

ignorere en del af denne obligatoriske tekst i kildeteksten, da en forståelse af dens betydning kræver kendskab til flere forskellige detaljer, der ikke er hensigtmæssige at fordybe sig i på nuværende tidspunkt - der er nok nyt der skal læres!

Det vi vil "læse", når vi ser kildeteksten er derfor kun:

```
/* Censor program
Det første rigtige
eksempel */

int karakter;

// karakteren er altid 12
karakter = 12;
println( "Tillykke, De bestod med " + karakter );
```

Vores program består altså i første række af syv linier som vi i det følgende vil studere nærmere.

1. Kommentarer

Kode er selv-dokumenterende

Der er nogle der mener at kildetekst er selvdokumenterende. Selvdokumenterende vil sige, at man ikke behøver at forklare hvad der sker i kildeteksten. Det er bare et spørgsmål om at læse den, så giver det sig selv. I virkeligheden har de ret - kode er selvdokumenterende mht. hvad det gør. Hvis det ikke var tilfældet ville det ikke være en algoritme.

To modtagere af kildeteksten

Der er blot et "men". At kildeteksten i en vis forstand er selvdokumeneterende er ikke nødvendigvis af større praktisk værdi. Et program skrives i virkeligheden til to modtagere. Den ene er compileren, der indirekte skal udføre programmet. Den anden er programmørerne - os selv, og alle andre der måtte komme til at arbejde med kildeteksten.

Notater er guld værd

Compileren behøver kun kildeteksten, der algoritmisk beskriver programmet i det pågældende sprog. Selv om en programmør er veluddannet og erfaren, er der et problem med at bruge kildeteksten alene som "læsestof" - tid! At læse et program igennem for at udlede dets virkemåde er tidskrævende. Lidt notater med kommentarer til hvorfor og hvordan ting er lavet i kildeteksten vil være guld værd i en sådan situation.

Ignoreres af compileren

I Java, og andre sprog, er det muligt at skrive kommentarer direkte i kildeteksten. Kommentarerne er specielle. De er kun rettet mod den menneskelige læser, ikke compileren. Compileren vil ganske enkelt ignorere alle kommentarer i kildeteksten.

1.1 Kommentarer i Java

Der er to måder at skrive kommentarer på, alt efter hvor omfattende de skal være.

Enkelt-linie kommentar

Hvis man blot ønsker at skrive en enkelt linie, eventuelt efter et stykke kildetekst bruges **//**. I consor-programmet optræder der en enkelt af disse. **//** betyder at alt fra disse tegn og linie ud skal ignoreres af compileren.

Fler-linie kommentar

Vil man skrive flere linier kan man blot starte hver af dem med **//**. Man har dog en anden mulighed. I stedet kan man starte sin kommentar med de to tegn **/*** og afslutte den med den omvendte kombination ***/**. I censor-programmet er der øverst et eksempel på dette. Måden, **/*** og ***/** er anvendt i eksemplet, er med vilje lavet lidt primitiv, for at sammenhængen mellem **/*** og ***/** skal være tydelig. Der findes nemlig mange sindrige måder at lave ascii-grafiske opstilling, som kan sløre sammenhængen. Den mest almindelige er nok:

```
/*
 * Censor program
 *
 *   Det første rigtige eksempel
 */
```

Her har man startet hver af de mellemliggende linier med `*` for at få et grafisk mønster, der vækker øjets opmærksomhed.

Mådehold med grafik-show

I starten vil man sikkert more sig med at lave forskellige grafiske boxe og lignende, men i længden bør man holde sig til enkle og ukomplicerede opstillinger, ellers kommer det hurtigt til at virke trættende.

Selv om de to slags kommentarer er temlig forskellige i deres natur, kan man godt lave en direkte sammenligning, hvis man betragter linieskift som terminator i forhold til `//`.

Linier	Start	Slut
Enkelt	`//`	linieskift
Flere	`/*`	`*/`

1.2 Gode/dårlige kommentarer

Der findes generelt to slags gode kommentarer: Overskrifts-kommentarer og forklarings-kommentarer.

Overskrifts-kommentarer

Overskrifts-kommentarer skal hjælpe læseren til at navigerer rundt i kildeteksten - til hurtigt at kunne se hvor man er og hvad det drejer sig om. De har naturligvis deres navn fra overskifter i almindelig tekst, der tjener samme formål. Overskifts-kommentarer er mere monstrøse end forklarings-kommentarer. De skal tiltrække læserens opmærksomhed, så et blik på en side let fanger øjet. Man lader et par linier frigøre kommentaren fra den øvrige kildetekst, og bruger større grafiske virkemidler for at fremhæve tilstedeværelsen.

Forklarings-kommentarer

En forklarings-kommentar er anderledes. Den er tilknyttet en mindre del af kildeteksten og har til formål at forklare hvad der sker, og eventuelt hvorfor. En forklarings-kommentar skal ikke tiltrække opmærksomhed - den skal supplere kildeteksten, ikke lede ind til den. Når man læser en forklarings-kommentar vil det derfor være som følge af, at man allerede er blevet fanget af den pågældende kildetekst, men har brug for en forklaring.

Man kan ikke sige det generelt, men der er nok en tendens til at `//` bliver brugt til forklarings-kommentarer, mens `/* */` mest bliver brugt til overskrifts-kommentarer, men det varierer en del.

Der findes mange dårlige kommentarer. F.eks. kommentarer der fortæller det åbenlyse - overflødige forklaringskommentarer!

I censor-programmet er der en overflødig forklarings-kommentar:

```
// karakteren er altid 12
karakter = 12;
```

Overflødige

Kommentaren er overflødig fordi den blot gentager den efterfølgende sætning. En sætning der må betegnes som letlæselig for enhver der kan programmere Java. Hvis den derimod havde fortalt noget om, hvorfor karakter altid sættes til 12, havde den sikkert været bedre.

Uklare eller indforståede

En kommentar skal naturligvis oplyse mere end den forvirrer. En kommentar der er uklar eller indforstået, har andre ikke meget glæde af. Man kan godt bruge kommentarer i kildeteksten til notater mens man arbejder med den, men efterhånden som kildeteksten nærmer sig sin endelige form, bør der kun være kommentarer som tjener vedligeholdelsesformål.

2. Lagerforståelse

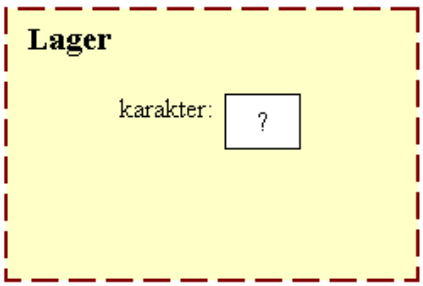
2.1 Erklæring

Sætningen:

```
int karakter;
```

Plads i lagret

kaldes en **erklæring** fordi den erklærer hvad **`karakter`** er. Den bevirker at der reserveres en plads i lageret til et heltal, som kaldes **`karakter`**:



Der er normalt talrige sådanne pladser reserveret, men vores beskedne program reserverer kun én.

Allokering

At der vil være tale om et heltal ses af angivelsen: **`int`**, for **integer** (dk.: heltal). Det at der reserveres en plads i lagret kalder man **allokering**. Der allokeres plads til en integer i lagret. Pladsen får navnet **`karakter`**. Dvs. at man ved at anvende dette navn kan referere til pladsen. **`Karakter`** kaldes en **variabel** fordi der på dens plads i lagret kan være en værdi, som man har mulighed for at ændre - at variere. Det tal der til enhver tid måtte befinde sig på **`karakter`**'s lagerplads kalder man **variablens værdi**, **`karakter`**'s værdi.

Udefineret

I figuren er der anført et spørgsmålstegn på pladsen i stedet for en værdi. Det skyldes at en variabel som udgangspunkt ingen værdi har - den er udefineret. Det betyder *ikke* at værdien er ukendt - der er ganske enkelt ingen!

Lad os råde bod på det - lad os give **`karakter`** en værdi.

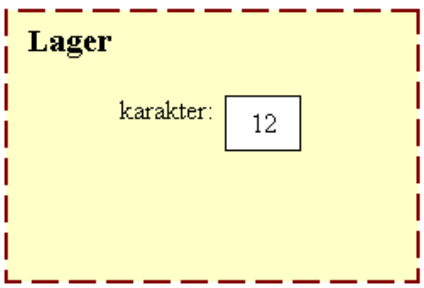
2.2 Assignment

I vores program sker det ved udførelsen af sætning:

```
karakter = 12;
```

Tildeling af værdi

Sætningen, der kaldes et **assignment** eller en **tildelings-sætning**, bevirker at **`karakter`**'s værdi sættes til 12.



Et assignment er altid opbygget af tre elementer. I midten er der lighedstegnet, der adskiller en variabel på venstresiden fra et udtryk på højresiden:

<variabel> = <udtryk>

Men hvad er et udtryk? Vi vil i første omgang se på aritmetiske udtryk.

2.3 Aritmetiske udtryk

Nok var vores censor-program et bedre eksempel end "Hello World"-eksemplet, men det mangler en meget grundlæggende proces som findes i ethvert program, nemlig beregninger.

Udtryk

Beregninger formuleres i form af udtryk. Vi kalder dem aritmetiske udtryk fordi de arbejder med tal. Udtryk kan indgå mange steder i et program, men den mest elementære anvendelse er i forbindelse med et assignment:

```
karakter = 6 + 6;
```

Evaluering af udtryk

Her optræder der et udtryk på højresiden, og resultatet af beregningen assignes til **`karakter`**. Det at man beregner et udtryk, kalder man at udtrykket **evaluere**s.

Eftersom udtrykket i dette tilfælde altid evalueres til det samme, er det ikke specielt interessant. I eksemplet er der direkte anført to tal, men man kan også anvende variabel-navne i et udtryk:

```
karakter = 12;
karakter = karakter + 1;
```

Her har vi to assignments. Det første skal sikre os, at **`karakter`** har en værdi før det andet assignment udføres. Det andet assignment er nemlig det interessante! Udtrykket består nu af en variabels værdi, samt et tal. Først slår man **`karakter`**'s værdi op, dernæst lægger man **`1`** til, og resultatet assignes til **`karakter`**.

Variable i udtryk ændres ikke

Man skal være opmærksom på en væsentlig ting i denne forbindelse. Før resultatet assignes til **`karakter`** sker der ingen ændring af dens værdi. At **`karakter`** optræder i udtrykket på højre side ændrer *ikke* dens værdi! Det eneste der sker med en variabel, når den indgår i et udtryk, er at dens værdi bliver slået op - *aldrig* at den ændres!

Ud over + har vi naturligvis også de andre almindelige regnearter:

+, -, /, *, %

Modulus

Skråstreg er division, men hvad er %? Den kaldes modulus og er rest ved division! Lad os se nogle eksempler:

5 % 3 = 2

10 % 3 = 1

9 % 3 = 0

1 % 3 = 1

Modulus bruges normalt i forbindelse med mere eller mindre trick-betonede anvendelser. F.eks. kan man bruge den til at se om et tal går op i et andet, i så fald er modulus 0. Ovenfor er det tilfældet med 9 og 3, da 3 går op i 9.

En anden ting man observerer, er at modulus aldrig kan blive større end én mindre end det man dividerer med. I eksemplet ovenfor betyder det, at uanset hvad man måtte tage modulus 3 på, kan det aldrig give mere end 2. Det skyldes naturligvis, at hvis det blev 3 eller mere, ville 3 gå op i det endnu engang!

Heltals-division

Division kræver i virkeligheden en bemærkning. Hvad er 5/3? Det er 1, og ikke 1.666..., som man ellers kunne tro. Det skyldes at / er heltals-division, når man dividerer to heltal. Resultatet bliver

ikke afrundet, men er antallet af gange 3 gå op i 5, nemlig 1 gang. Vi skal senere i kapitlet om typer, se hvordan man kan arbejde med komma-tal.

Som det sikkert er bekendt er * og / "stærkere" end + og -. Dette at nogle regnearter er stærkere end andre, gør at de bestemmer i hvilken rækkefølge del-udtryk evalueres. F.eks.

```
5 + 3 * 2
```

Hvis de alle var lige stærke ville resultatet blive 16, men da * er stærkere end +, bliver 3*2 evalueret først, og resultatet bliver 11.

Præcedens

Det at * er stærkere end +, betegner man, at * har præcedens over +. Man kan opstille det i en præcedens-tabel; hvor de stærkeste er øverst, og de svageste er nederst:

*, /, %
+, -

/ og % danner "par"

Alle på samme præcedens-niveau er lige stærke, og vi ser her at modulus har samme styrke som division; hvilket er meget naturligt da de danner "par".

3. Simpel input/output

Når man starter med at lave mindre programmer bliver det hurtigt et ønske at kunne udskrive og indlæse værdier. Det er desværre ikke så enkelt at gøre det, hvilket skyldes flere faktorer.

Java er grafisk

Java er grafisk orienteret. Det betyder, at man i "rigtige" programmer gør et større arbejde for at få en grafisk dialog med brugeren. Man skal specificere og forstå en masse, for at opnå dette. Når man i starten har brug for lidt input og output, er det derfor en uforholdsmæssig stor indlæringsbyrde man pålægges for så lidt.

Indlæsning er kompliceret

Man har også mulighed for at udskrive og indlæse mere primitivt. Det er på traditionel form, hvor man skriver linie efter linie, som på en gammeldags EDB-skærm. Udskrift er i den forbindelse rimelig enkelt, men indlæsning er stadig krævende. Det skyldes at indlæsning er en mere kompliceret proces. Hvad er det man indlæser? Er det et tal eller en tekst? Hvad skal der gøres hvis man indtaster en tekst når programmet forventer et tal? Mao. indlæsning foregår i en mere uforudsigelig proces fordi der kommunikeres med brugeren, mens udskift foregår uden at brugeren er aktiv.

Heldigvis er det ikke udskrivning, men indlæsning der er kompliceret. Det er nemlig problemfrit at lære programmering uden indlæsning meget langt hen ad vejen, mens udskrivning er et must.

3.1 Output til skærmen

Man kan udskrive tekst, tal og variables værdier. Hvis vi f.eks. vil udskrive værdien af variablen **`karakter`** uden yderligere oplysninger, kan vi bruge sætningen:

```
System.out.println( karakter );
```

```
12
```

Hvad er **`System.out.println`**? Glem det indtil videre og brug det bare! Det er nemlig en lang forklaring, der bliver betydelig kortere, hvis vi venter til senere.

Hvis vi derimod ikke ønskede at oplyse karakteren, men blot meddele, at man har bestået, kunne

følgende sætning bruges:

```
System.out.println( "Du bestod" );
```

```
Du bestod
```

Tekst mellem anførselstegn

Man bemærker her at tekst i et program skal placeres mellem to anførselstegn. Dette gælder hvis det er tekst som programmet skal behandle som data, og dermed ikke skal forsøge at "forstå". Man kalder en tekst, der skal opfattes som data, for en **tekststreng**.

I censor-programmet havde vi et eksempel på en tekststreng, der bliver sat sammen med en variabel:

```
System.out.println( "Tillykke, De bestod med " + karakter );
```

```
Tillykke, De bestod med 12
```

Klistre-plus

Effekten af dette blev at tekststrengen og værdien af **`karakter`** blev udskrevet i forlængelse af hinanden. Man har mulighed for at "klistre" ting sammen vha. + i udskrifter, og det kan bruges flere gange efter hinanden. F.eks.:

```
System.out.println( "Du fik " + karakter + ", så du bestod" );
```

```
Du fik 12, så du bestod
```

Bemærk det ekstra mellemrum som afslutter den første tekststreng. Den er nødvendig for at værdien af **`karakter`** ikke bliver mast sammen med den første tekststreng. Hvis den ikke var der, ville resultatet blive:

```
System.out.println( "Du fik" + karakter + ", så du bestod" );
```

```
Du fik12, så du bestod
```

Uden linieskift

Man bemærker måske at der altid foretages et efterfølgende linieskift i forbindelse med udskrift-sætningen. Der findes en alternativ udskrivnings-sætning, som ikke foretager et sådant linieskift. F.eks.:

```
System.out.print( "Du fik " + karakter );
System.out.println( ", så du bestod" );
```

```
Du fik 12, så du bestod
```

Muligheden for at udskrive uden linieskift bruges f.eks. ofte i forbindelse med indlæsning.

3.2 Klistre-plus og aritmetisk plus

Samme præcedens

I Java har man som bekendt valgt også at bruge + til at samle ting ved udskrift (mere præcist at sætte tekst sammen, men mere om det senere). Det kan være lidt forvirrende. Klistre-plus og aritmetisk plus har nemlig samme præcedens:

```
```

```
System.out.println( "Sum er " + 3 + 5 );
```

```
Sum er 35
```

Evalueres fra venstre mod højre

Her vil udtrykket mellem paranteserne blive evalueret. Man skal ikke lade sig forvirre af at der indgår tekst i dette "regnestykke" for i programmering begrænser udtryk sig ikke til regneudtryk. Udtrykket evalueres fra venstre mod højre, og det forløber derfor som:

```
"Sum er " + 3 + 5
 "Sum er 3" + 5
   "Sum er 35"
```

Grunden til at dette sker er, at hvis en af de to ting der skal pluses er en tekststreng vil + blive et klistre-plus. Da teksten står forrest, får den hele tiden trumfet igennem at plusserne bliver klistre-plusser. Lad os i stedet placere 3+5 forrest:

```
System.out.println( 3 + 5 + " er summen" );
```

```
8 er summen
```

Igen evalueres det hele fra venstre mod højre, men nu bliver det første plus et aritmetisk plus fordi det står mellem to tal. Evalueringsforløbet bliver:

```
3 + 5 + " er summen"
  8 + " er summen"
    "8 er summen"
```

Det er først til sidst teksten kommer ind i billedet og gør det sidste plus til et klistre-plus.

3.3 Input fra tastaturet

Lad det være sagt med det samme: Indlæsning fra tastaturet er rædselsfuldt! Se blot flg. eksempel:

```
import java.io.*;

class simpelInput {
  public static void main( String[] argv ) throws IOException {

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

    System.out.print( "Indtast et tal: " );

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

    System.out.println( "Du indtastede " + tal );
  }
}
```

```
Indtast et tal: 12
Du indtastede 12
```

Senere, senere ...

Programmet indlæser et tal og udskriver det igen. Igen vil en forklaring af, hvad de forskellige dele betyder kræve en uforholdsmæssig lang forklaring i forhold til hvad en sådan forklaring på et

senere tidspunkt vil kræve. Derfor, bare gør det - tænk ikke for meget over det - det vender vi tilbage til senere!

Men hvordan bruger man det så?

De første tre linier (der kunne stå på én; hvis den ikke blev så forfærdelig lang!), skal kun stå i starten af programmet:

```
BufferedReader indlæser =
  new BufferedReader(
    new InputStreamReader( System.in ) );
```

Uanset hvor mange tal vi måtte ønske at indlæse efterfølgende, skal det kun stå én gang. Det er det der giver os **`indlæser`** som vi kan bruge gentagne gange til indlæsning.

Ledetekst

Når man skal indtaste data, er det rart med en ledetekst, der fortæller *hvad* programmet forventer. Samtidig vil ledetekstens tilstedeværelse på skærmen indikere *hvornår* input forventes.

Man plejer derfor at have følgende to linier umiddelbart efter hinanden:

```
System.out.print( "Indtast et tal: " );

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

Først er der sætningen der udskriver ledeteksten (bemærk at der ikke bruges linieskift). Dernæst sætningen der indlæser tallet. Her har vi valgt at kalde variablen **`tal`** men resten af linien skal altid være den samme. Man bemærker at **`indlæser`** fra starten af programmet bliver brugt i denne sammenhæng.

Lad os se et eksempel, hvor vi indlæser tre tal og udskriver dem på en linie:

```
import java.io.*;

class simpelInput {
  public static void main( String[] argv ) throws IOException {

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

    System.out.print( "Indtast første tal: " );
    int første = Integer.parseInt( indlæser.readLine() );

    System.out.print( "Indtast andet tal: " );
    int andet = Integer.parseInt( indlæser.readLine() );

    System.out.print( "Indtast tredie tal: " );
    int tredie = Integer.parseInt( indlæser.readLine() );

    System.out.println( "Du indtastede: " + første + ", " +
                        andet + ", " + tredie );
  }
}
```

Vi vil undgå indlæsning

Vi vil i videst muligt omfang undgå indlæsning. Det er reelt sjældent nødvendigt i vores programeksempler, så længe vi kun laver relativ enkle programmer. Der er dog nogle opgaver sidst i kapitlet, med indlæsning, som skal sikre at man kan gøre det, såfremt indlæsning bliver nødvendigt.

4. Identifiers, reserverede ord og literaler

Kategorier

I kildeteksten optræder der mange forskellige elementer som tilsammen beskriver programmet. Når man mere systematisk skal studere programmering - det er det vi gør - er det en fordel at

kunne placere de forskellige elementer i kategorier. Identifiers, reserverede ord og literaler er tre sådanne kategorier.

4.1 Identifiers

Identifiers er navne eller ord der optræder i kildeteksten. De udemærker sig ved at være navne vi selv fastlægger. Det er altså ikke navne fra programmeringssproget Java som sådan (se evt. reserverede ord nedenfor).

I censor-programmet optræder der flere identifiers, her markeret med rødt og blåt:

```
/*
 *   Censor program
 *   Det første rigtige eksempel
 */

class Tillykke {
  public static void main( String[] argv ) {
    int karakter;

    // karakteren er altid 12
    karakter = 12;
    System.out.println( "Tillykke, De bestod med " + karakter );
  }
}
```

`argv` er en identifier, men vi vil i denne sammenhæng vælge at se bort fra den, da vi ikke her vil se på dens anvendelse.

Som nævnt tidligere er **`Tillykke`** det navn kildeteksten skal gemmes under (mere præcist **`Tillykke.java`**). Det er et navn vi selv har valgt, og når det står lige efter **`class`** vil vi indtil videre betragte det som programmets navn.

Identifieren **`karakter`** er navnet på vores variabel som vi tildeler værdien 12. Ligesom navnet **`Tillykke`** kunne vi have valgt at kalde den noget andet, og programmet ville være aldeles upåvirket - det ville gøre nøjagtig det samme.

Når vi vælger identifiers er der nogle simple regler vi skal overholde. En identifier må kun bestå af tegnene a-z, A-Z, 0-9, $ og _. Den sidste af disse kalder man normalt underscore. Man må også bruge de danske bogstaver æ, ø og å. Yderligere må en identifier ikke starte med et ciffer 0-9. Man bemærker at en identifier ikke må indeholde mellemrum.

Case-sensitive

Hvad med store og små bogstaver, er der forskel på dem? Java skelner mellem store og små bogstaver. Man siger at Java er case-sensitiv. Betragt følgende to identifiers:

```
DetteErEnIdentifier

detteerenidentifier
```

For Java er disse to identifiers ikke ens. For den, er der lige så stor forskel på a og A, som på a og b.

Sætninger som identifiers

Bemærk iøvrigt måden hvorpå den øverste identifier er skrevet som en sætning, hvor man indikerer starten af hvert ord ved at første bogstav er stort. Det er en meget anvendt teknik når man laver identifiers. Den lidt ældre teknik med at adskille ordene med underscore er gået lidt af mode.

```
dette_er_en_identifier
```

I forbindelse med "program-navne" bør man undgå at bruge de danske bogstaver æ, ø og å. Der kan opstå problemer med compilere, der ikke kan håndtere de deraf følgende file-navne.

4.2 Reserverede ord

Nok kan vi selv vælge at navngive ting med identifiers, men da man lavede Java valgte man nogle ord til selve sproget. Disse ord kan vi ikke selv bruge som identifiers. De reserverede ord er:

abstract	default	goto	operator	synchronized
boolean	do	if	outer	this
break	double	implements	package	throw
byte	else	import	private	throws
byvalue	extends	inner	protected	transient
case	false	instanceof	public	true
cast	final	int	rest	try
catch	finally	interface	return	var
char	float	long	short	void
class	for	native	static	volatile
const	future	new	super	while
continue	generic	null	switch	

Tabel 1:
Reserverede ord

Ovenfor er en række af de reserverede ord skrevet med blåt. Disse er ikke i anvendelse i sproget, men er reserverede af hensyn til evt. fremtidig brug.

*[21. juni 2003: To af mine studerende (Stefan Lyager Jensen og Brian Nørremark) har prøvet at anvende de blå ord, og konstateret at kun **const** og **goto** stadig er reserverede, mens de øvrige kan bruges som identifiers. Siden er **assert** og **strictfp** blevet reserverede - jeg vil opdatere listen ved lejlighed]*

4.3 Literaler

Bogstavelige

Literaler (eng. bogstavelige) er data vi skriver direkte i kildeteksten. Det er ting som compileren skal tage "bogstaveligt". Den skal ikke begynde at fortolke eller oversætte noget - den skal tage det som det er!

I censor-programmet er der to literaler:

```
/*
 *   Censor program
 *   Det første rigtige eksempel
 */

class Tillykke {
  public static void main( String[] argv ) {
    int karakter;

    // karakteren er altid 12
    karakter = 12;
    System.out.println( "Tillykke, De bestod med " + karakter );
  }
}
```

Det ene er en talværdi, et numerisk literale, mens det andet er en tekststreng, et tekst literale. Vi vil

senere i kapitlet om typer, se hvordan man kan angive andre slags literaler.

5. Fejltyper

Debugging

Fejl er uundgåelige. Man vil lave dem i ethvert program man kommer i berøring med. Hver gang man laver et stykke kode, laver man også en række fejl. At fjerne disse fejl er en naturlig del af processen, og ofte den mest tidskrævende. Man kalder det debugging (aflusning) og en stor del af ens tid vil gå med at trække kammen igennem ens kildetekst.

Man kan inddele fejlene i tre grupper.

5.1 Compile-time errors

Syntaks-fejl

Denne type fejl er de nemmeste at finde og normalt også de nemmeste at rette. De opstår under compileringen, hvor compileren finder syntaktiske fejl i kildeteksten. De er nemme at finde, fordi compileren gør arbejdet for os. Det eneste problem i den forbindelse, ligger i at compilerens angivelse af fejlen i enkelte tilfælde kan være mangelfuld eller vildledende. Hvis man først har fundet fejlen er det normalt enkelt at rette den. De fleste af disse fejl er tastefejl - typo's.

Semi-kolon

Den hyppigst forekommende fejl blandt begyndere er manglende semikolon. Hvis jeg havde en krone for hver gang jeg har sagt semikolon til en studerende ville jeg være en rig mand.

5.2 Run-time errors

Programmet går ned

Disse fejl er karakteriseret ved at de får programmet til at terminere. De opstår under udførelsen og er af en sådan art at programmet ikke er i stand til at håndtere situationen selv[1]. Det er en fejltype der kan være meget vanskelig at finde.

5.3 Logiske fejl

Det var ik' det den sku'

Hvad hjælper det at programmet kan oversættes og kører uden at gå ned; hvis det ikke gør det det skal? Logiske fejl er "Det var ikke det den sku'"-fejl

Der optræder en sådan fejl i censor-programmet. Karakteren 12 findes ikke, så selvom det er en ganske imponerende karakter, er der nok noget galt. Programmet compilerer uden fejl og det kører uden problemer, men det gør ikke hvad det skulle (hvad end så det præcist skulle være).

6 Gode kode vaner

Orden er godt

Vi ved det jo godt! Orden i tingene gør det hele meget nemmere at arbejde med, og hvad der er nemt det kan vi godt lide. Vi kan så godt lide det, at vi ikke gider at holde orden. At holde orden kræver en indsats, det gør ondt der hvor dovenskaben sidder. Vi lader os derfor friste, orden er jo først noget vi får glæde af senere, og måske skal vi ikke se mere til det vi laver - hvem ved?

Vaner er ikke trælse

Der er bare et problem. Dovenskab straffes med problemer! Så enten skal vi gøre noget der er træls, eller også får vi (måske) problemer. Heldigvis er der en smart ting, det er vaner. Vaner er ting man gør uden at tænke så meget over det, de er ikke træls at udføre.

Bliv ikke manisk

Nu kan ordens-vaner godt blive så maniske at man begynder at rette kuglepenne, så de flugter med bordets geometri. Det drejer sig derfor om at få fornuftige vaner. Mange fornuftige ting er kedelige, så over i vanerne med dem, så skal man ikke tænke så meget på dem.

At indlære

For at noget bliver en vane skal man vænne sig til at gøre det. Man skal i starten være bevidst og

vaner er træls

omhyggelig med altid at gøre det, så det til sidst bliver indgroet. At opbygge en vane er derfor træls, men belønningen kommer senere, og er vedvarende.

Hvad er det så man skal gøre sig til en vane?

6.1 Indrykninger

Fremhæver syntaksen

At lave indrykninger der fremhæver syntaksen er godt. Når man læser programmet er det nemmere at forstå den strukturelle opbygning fordi indrykninger understreger sammenhænge.

Lære ved eksemplets magt

Hvordan lærer man så at lave passende indrykninger? Der er to muligheder. Enten kan man studere en formel beskrivelse af hvad man bør gøre, eller man kan lære det hen ad vejen ved at se hvordan andre gør. Jeg foretrækker den sidste vinkel, da den første er kedelig! Sun har lavet en formel kode-konvension, men man bør nok primært anvende den til afklaring, hvis man har brug for noget at støtte sig til.

Kværulantiske diskussioner om små variationer

Der er en ulempe ved at lære det ud fra eksempler. De er ikke enige! Mindre forskelle i hvordan man laver indrykninger er ikke væsentlige, men man kan føre lange kværulantiske diskussioner om emnet. Det kan være en sand fornøjelse hvis man er i det humør, men ellers er det idiotisk.

Godt med fælles konvension i projekt

I forbindelse med projekter hvor flere mennesker arbejder sammen over længere tid, kan det være en fordel at bruge nøjagtig samme kode-konvension, da det trods alt gør koden mere behagelig at læse. Små forskelle i individuelle konvensioner kan være et irritationsmoment og i implementationsfasen er man ofte under tidspres, så der skal ikke så meget til.

6.2 Kommentarer

Selve udformingen af kommentarer hører ind under kode-konvensionen, men det at huske dem er en god vane.

Nemmest at lave kommentarer når man laver koden

En kommentar er et klassisk eksempel på noget man får glæde af senere, men som synes overflødig når den laves. Lav kommentarer - straffen er *meget* hård! Hvis man ikke har lavet indrykninger kan de nemt laves senere, men det kan tage lang tid at finde ud af hvad et program laver hvis kommentarerne mangler. Det nemmeste tidspunkt at lave kommentarer er mens man laver koden, det sværeste tidspunkt er senere.

6.3 Parvise paranteser

Træls når paranteser ikke passer

En af de mere trælse fejl at lede efter, er når paranteserne ikke passer. Visse IDE'er afhjælper dette problem, men de er få. Når man i f.eks. Emacs sætter en højreparantes flytter markøren et kort øjeblik tilbage til den tilhørende venstreparantes, så man kan se om sammenhængen er som man regnede med. Som sagt er det en sjældenhed, så hvad gør man så?

Sætte paranteser parvist

Jeg har gjort det til en vane at sætte paranteser parvis. Hver gang jeg sætter en venstreparantes sætter jeg også den tilhørende højreparantes, og først derefter skriver jeg indholdet. De eneste situationer hvor paranteserne går i kage for mig, er når jeg begynder at rette i noget jeg allerede har skrevet, ellers sker det aldrig!

6.4 Andre vaner

De dumme har problemer - de kloge har vaner

Vaner er en stærk ting, som man bør udnytte. Der er mange andre situationer i et udviklingsforløb, også flere i implementationsfasen, hvor gode vaner belønner sin ejermand. Husk at dovenskab koster - i morgen. De dumme har problemer - de kloge har vaner!

fodnoter

1 Vi skal senere under Exception Handling, se hvordan man kan lave fejlhåndtering.

Repetitionsspørgsmål

1 Hvad skal den tekstfile hedde som man gemmer sit program i?

2 Hvorfor ser vi bort fra nogen af detaljerne i starten?

3 I hvilken forstand er kode selv-dokumenterende?

4 Hvilke to modtagere skrives kildeteksten til?

5 Hvorfor skriver man kommentarer i et program?

6 Hvilke to former for kommentarer findes der?

7 Hvad er dårlige kommentarer?

8 Hvad betyder allokering?

9 Hvad vil det sige at en variabel er udefineret?

10 Hvad sker der ved et assignment?

11 Hvad er et aritmetisk udtryk?

12 Hvad betyder det at et udtryk evalueres?

13 Kan en variabel der indgår i et udtryk ændre værdi?

14 Hvad er modulus?

15 Hvad er præcedens?

16 Hvad er modulus' præcedens?

17 Hvorfor nøjes vi med simpel input/output?

18 Hvornår er et + et klistreplus?

19 Hvor mange gange skal "`BufferedReader indlæser = ...`" stå i et program?

20 Hvad er en identifier?

21 Hvad vil det sige at Java er case-sensitiv?

22 Kan man bruge æ, ø og å?

23 Hvad er reserverede ord?

24 Hvad er et literale?

25 Hvad er en compile-time error?

26 Hvad er en run-time error?

27 Hvad er en logisk fejl?

28 Hvorfor er det en god vane at lave indrykninger?

29 Hvorfor er det en god vane at lave kommentarer samtidig med man laver koden?

30 Hvorfor er det en god vane at skrive paranteser parvis før man skriver indholdet?

Svar på repetitionsspørgsmål

1 Den skal hedder programmets navn (som er det der står lige efter **`class`**) efterfulgt af **`.java`**.

2 Fordi der er så mange ting der skal læres, og vi derfor vil koncentere os om lidt mindre af gangen.

3 I forhold til compileren er koden selvdokumenterende, men sjældent i forhold til en menneskelig læser.

4 Compileren og menneskelige læsere, f.eks. andre programmører.

5 For at gøre det lettere for et menneske at læse kildeteksten.

6 Overskriftskommentarer og forklaringskommentarer.

7 Overflødige, uklare eller indforståede kommentarer.

8 At der gøres plads til noget i lagret, f.eks. en variabel.

9 At den ikke har nogen værdi.

10 Ved assignment tildeles en variabel en ny værdi.

11 Populært sagt: et regnestykke.

12 At det beregnes, at man finde dets værdi.

13 Nej.

14 Rest ved division

15 Præcedens beskriver styrkeforholdet mellem regnearter.

16 Modulus har samme præcedens som gange og dividere.

17 Fordi man i et virkeligt program vil bruge meget tid på at lave input/output på en grafisk brugergrænseflade. Vi er i første omgang kun interesseret i input/output for at kunne bruge det i simple programmer, derfor vil vi ikke bruge for meget tid på at lære om idet.

18 Hvis der står en tekststreng på mindst en af siderne.

19 Kun en gang. Derefter kan man bruge **`indlæser`** gentagne gange.

20 Et ord som indgår i vores program, som vi selv har fundet på.

21 At Java gør forskel på store og små bogstaver i identifiers.

22 Ja, men undgå det i program-navne, da de deraf følgende filenavne kan give problemer.

23 Det er identifiers som sproget har reserveret til sig selv.

24 Det er et stykke data der optræder direkte i kildeteksten. Man kunne også kalde det en konstant.

25 En fejl der opstår under compileringen.

26 En fejl der opstår under udførelse af programmet.

27 At programmet ikke gør det det skulle.

28 Fordi det gør programmet lettere at læse, og man dermed sparer tid senere hen.

29 Fordi det er mens man laver koden at man har bedst mulighed for at skrive dem, senere kan det være tidskrævende at finde ud af hvad programmet gør.

30 Så skal man sjældent lede efter paranteser der ikke passer sammen.