Selektion

Sidst ændret: 07/19/2018 14:11:09
Opgaver

"…"

- …

Først ser vi på boolske udtryk og den boolske variabel boolean. Dernæst ser vi på if-sætningen med else, og nestede if-sætninger. Vi eksemplificerer switch-sætningen som et specialtilfælde af visse nested if-sætninger. Til slut berøres den tertiære operator og vi opstiller en opdateret præcedenstabel.

Forudsætninger:

Have et kendskab til konstruktion af simple programmer, der svarer til kapitlet "Simpel programmering" og kendskab til kontrolstrukturer svarende til kapitlet "Algoritmer".

Selektion er valg

I kapitlet om algoritmer så vi hvilken opfattelse af programmering, der lå bag ingeniør/datalog-historien. Det viste sig at være idéen om et **sekventielt forløb**, med kommandoer der fulgte efter hinanden som perler på en snor. En sådan indskrænket opfattelse af programmering ville stærkt begrænse hvad vi kunne gøre. Til at afhjælpe dette problem blev der introduceret nogle simple **kontrolstrukturer**, som styrede kommandoer og på den måde selv blev kommandoer, der kunne træffe valg og foretage gentagelser. Vi skal i dette kapitel se på hvilke kontrolstrukturer der findes i Java, når det drejer sig om **selektion**, der er den faglige betegnelse for "valg".

Selektion gør et program mere dynamisk

Selektion gør et program i stand til at tilpasse sig de situationer der kan opstå mens det udføres. Programmøren giver et program flere alternative måder at arbejde videre med et problem, alt efter hvordan situationen aktuelt ser ud. Man taler om at programmet dynamisk tilpasser sig. "Dynamisk" er et af de universale plus-ord, som kan dække over mangt og meget, men i programmering har det en præcis betydning. Noget der er dynamisk, er noget der tager form under programudførelsen. Med selektion taler vi derfor om dynamisk tilpasning, fordi programmet tilpasser sig, træffer valg, mens det kører.

Når vi senere skal se på objektorienteret programmering, og de muligheder der i den forbindelse findes mht. at gøre programmer dynamiske, vil vi sikkert tænke tilbage på selektion som en beskeden repræsentant for de muligheder der findes.

1. Boolske udtryk

Programmets tilstand er variablenes værdier

For at arbejde med selektion, skal vi kunne beskrive forskellige situationer, så vi kan skelne mellem hvornår vi vil gøre det ene eller det andet. Situationer er mere præcist **tilstande**. En tilstand er givet ved en række variables værdier. Man taler samlet om programmets tilstand som alle variables værdier under ét (mere præcist er det samtlige variables værdier *og* det sted man er kommet til i udførelsen, men det sidste vil vi se bort fra her). I praksis ser man altid kun på en lille del af variablene i et program, når man

skal tage stilling til "situationen". Man siger at programmets tilstand er afgørende for hvad der sker, da selv få variable er en del af programmets tilstand.

En tilstandsbeskrivelse er et **udsagn** der siger noget om variables værdier. Et sådant udsagn kan enten være sand eller falsk afhængig af de variables værdi, som indgår i udtrykket. F.eks.:

```
antal > 10
```

hvor sandhedsværdien af udtrykket, eller udsagnet, afhænger af variabelen **`antal`**. Først når programmet når til at skulle **evaluere** dette udtryk ser det på **`antal`**'s værdi, og først da tages der stilling til om programmets tilstand opfylder betingelsen for at gøre dette eller hint.

Boolsk udtryk

Sådanne udtryk der evalueres til enten sand eller falsk, kaldes **boolske udtryk** (eller logiske udtryk). Ligesom aritmetiske udtryk kan boolske udtryk indeholde beregninger (der sammenlignes med andre værdier). F.eks. kunne eksemplet ovenfor udvides til:

```
2 * antal > 10
```

Sammenlignings-operatorerne har lavere præcedens end de aritmetiske

Man får derved blandet aritmetiske operatorer (+, -, *, osv.) med sammenlignings-operatorer. Det betyder at der skal tages stilling til præcedens mellem endnu flere operatorer end da vi først så på præcendens i aritmetiske udtryk. Der gælder for alle **sammenlignings-operatorerne** at de binder svagere end samtlige aritmetiske. Det betyder at alle beregninger i boolske udtryk udføres før der foretages sammenligninger.

Lad os se hvilke sammenlignings-operatorer der findes i Java:

Operator	Betydning
<	Mindre end
<=	Mindre end eller lig med
>	Større end
>=	Større end eller lig med
==	Lig med
!=	Forskellig fra

Tabel 1: *Sammenlignings-operatorerne i Java*

Man bemærker at der i fire tilfælde anvendes en kombination med =. Det er let at huske placeringen af dette lighedstegn, da det altid skal stå sidst.

Forveksling af == og = er uundgåelig fejl

En af de hyppigst forekomne **logiske fejl** i programmer, kommer af Java's anvendelse af dobbelt lighedstegn for lighed. Altså ==, i stedet for = som forbeholdes assignment. Fejlen består i, at man bruger enkelt lighedstegn i sammenligninger. Det er en fejl som alle kommer til at lave (mange gange) og man kan godt glemme alt om at undgå det. Det vil følge enhver programmør, dog i aftagende grad, til den dag han slipper tastaturet.

Inden vi går videre med logiske udtryk, skal vi se på den sidste af de **simple typer** i Java, nemlig en type, hvis værdimængde er sandhedsværdierne sand og falsk.

1.1 `boolean`

`boolean` er Java's **boolske type**, der kan indeholde værdierne **`true`** og **`false`**. Man kan erklære variable og lave tildelinger fuldstændig som for de numeriske typer, blot skal

man her anvende boolske udtryk på højresiden af assignment. F.eks.:

Source 1:
Anvendelse af **`boolean`**

```
int indtægt=10000, udgift=7000;
boolean erOverskud;

erOverskud = indtægt > udgift;
```

Her erklæres en boolsk variabel **`erOverskud`**, der skal indeholder oplysning om der er et overskud; hvis man har haft en given indtægt og udgift.

I eksemplet ovenfor er brugt en sammenligningsoperator, en operator der tager to numeriske operander og giver en boolsk værdi. Der findes også specielle boolske operatorer der tager boolske operander og giver et boolsk resultat.

1.2 Boolske operatorer

Lad os se nogle af de boolske operatorer i Java:

Tabel 2:
Boolske operatorer i Java

Operator	Betydning
`&`	AND
`|`	OR
`^`	XOR

Man bruger normalt de engelske betegnelser på tryk, men veksler frit mellem de engelske og danske betegnelser ("og", "eller" og "eksklusiv eller") i daglig tale.

Sandhedstabel

Når man skal beskrive de boolske operatorers virkemåde bruger man normalt det der kaldes **sandhedstabeller**. En sandhedstabel beskriver hvilke værdier man får ved at anvende operatoren på de mulige værdier som operanderne kan antage.

1.2.1 AND

Lad os først se sandhedstabellen for AND:

Tabel 3:
Sandhedstabellen for AND

`&`	**`false`**	**`true`**
`false`	**`false`**	**`false`**
`true`	**`false`**	**`true`**

Man ser at AND kun giver værdien **`true`** hvis begge operander er sande, ellers bliver resultatet **`false`**.

Tilhøre interval

Et godt eksempel på en situation hvor man anvender AND er når man vil teste på **intervaller**.

Hvis man er lidt matematisk minded kunne man måske ønske at bruge et boolsk udtryk som

```
2 < antal < 10
```

men det kan man ikke i Java. Hvis man forsøgte ville det nemlig blive fortolket forkert i forhold til det tilsigtede.

Evaluere fra venstre mod højre

Java vil se på udtrykket som bestående af tre operander med to operatorer. Da ingen af de to operatorer har præcedens frem for den anden vil hele udtrykket blive evalueret fra venstre mod højre. Lad os for eksemplets skyld antage at **`antal`** har værdien 5. I så fald vil deludtrykket **`2 < antal`** blive evalueret til **`true`**, og det samlede udtryk ville være reduceret til:

```
true < 10
```

og nu er problemet tydeligt, idet der er en oplagt typefejl (sammenlignings-operatorerne kan nemlig ikke operere på en blanding af forskellige typer)

Sammensat udtryk

Hvad gør man så? Man bruger AND som vi netop har introduceret, og formulerer den samme tanke ved et **sammensat udtryk**:

```
2 < antal & antal < 10
```

Sammenlignings-operatorerne har højere præcedens end de rent boolske

Nu vil udtrykket igen blive evalueret fra venstre mod højre, men fordi sammenlignings-operatorerne har højere præcendens end de rent boolske operatorer, vil udtrykket blive reduceret til (hvis vi stadig antager at **`antal`** har værdien 5):

```
true & true
```

og dernæst endelig til værdien **`true`**.

1.2.2 OR

Lad se sandhedstabellen for OR:

Tabel 4: *Sandhedstabellen for OR*

|	false	true
false	false	true
true	true	true

Man ser at OR kun giver værdien **`false`** hvis begge operander er falske, ellers bliver resultatet **`true`**.

Ikke tilhøre interval

Hvis man i modsætning til eksemplet ovenfor ønsker at teste om en værdi *ikke* tilhører et interval kan man bekvemt anvende OR. Hvis vi ser på det samme interval, og vil teste om **`antal`** *ikke* ligger mellem 2 og 10, men udenfor intervallet kun man bruge:

```
antal <= 2 | 10 <= antal
```

Man ser nogle gange begyndere lave den banale fejl at bruge et &; hvilket naturligvis giver et udtryk der altid er falsk, da **`antal`** ikke kan være på begge sider af intervallet samtidig.

1.2.3 XOR

Lad os se den sidste af de tre boolske operatorer fra tabel 2, XOR:

Tabel 5:
Sandhedstabellen for XOR

^	false	true
false	false	true
true	true	false

Hvor sandhedstabellerne for AND og OR nok ikke var så overraskende, er det formodentlig de færreste der, uden at have hørt om den i forvejen, har haft nogen idé om hvad "eksklusiv eller" dækkede over.

Kræsen OR

XOR er en mere kræsen udgave af OR. En af operanderne skal være sand, men kun én af dem.

Vende sandhedsværdi

XOR er speciel, og den bruges da også kun i meget sjældne tilfælde. Lad os se et eksempel, der viser hvordan man kan bruge XOR til at "vende" sandhedsværdien af et udtryk (den anvendes dog aldrig i praksis til dette formål).

Hvis man XOR hvad som helst (**`true`** eller **`false`**) med **`true`**, får man det modsatte. Vi kunne f.eks. udtrykke at **`antal`** ikke tilhører intervallet fra 2 til 10, ved at bruge udtrykket med interval-test og kombinere det med XOR og **`true`**:

```
true ^ ( 2 < antal & antal < 10 )
```

XOR har højere præcendens end AND og OR

Her er udtrykket med AND placeret i paranteser da XOR har højere præcendens end AND. [Dette gælder kun for de tunede versioner - jeg skal have fundet ud af hvordan jeg præcist vil formulere teksten på det her sted]

1.2.4 NOT

Unær operator

Det er naturligvis en meget klodset måde at udtrykke en såkaldt **negation**, og man har da også en speciel unær operator til dette formål. Operatoren kaldes NOT og man bruger et udråbstegn til at repræsentere den. Lad os for et syns skyld se sandhedstabellen for NOT, selv om den er ganske triviel:

Tabel 6:
Sandhedstabellen for NOT

!	
false	true
true	false

NOT er forøvrigt den eneste unære boolske operator, men med kun to mulige værdier, **`true`** og **`false`**, i vores værdimængde, kan man vel dårligt forestille sig der kunne være andre!

Udtrykket ovenfor kan nu formuleres som

```
!( 2 < antal & antal < 10 )
```

men man vil naturligvis vælge at bruge udgaven med OR, da den må formodes at være lettest at læse for de fleste.

NOT har højere præcendes end

Bemærk i øvrigt, at det er nødvendigt at bruge paranteser ovenfor. Det skyldes at NOT har højere præcendens end både **<** og AND. NOT har samme præcendens som de unære aritmetiske operatorer (fortegn).

AND, OR og XOR

1.2.5 Tunede operatorer

Operatorerne `&&` og `||`

Der findes to andre boolske operatorer, der i virkeligheden er de mest anvendt. Det er **`&&`** og **`||`**. De virker umiddelbart som de tilsvarende med ét **`&`** og ét **`|`**, men de først nævnte har en ekstra effekt på evalueringen af de udtryk de indgår i. Man kan sige at de er "tunede". Lad os igen se udtrykket med interval-test; denne gang med tunet operator:

```
2 < antal && antal < 10
```

Lad os nu, i modsætning til tidligere, antage at **`antal`** har værdien 0. Efter det første deludtryk er blevet evalueret, vil det blive:

```
false && antal < 10
```

Tunede operatorer

her kan man allerede konkludere at det samlede udtryk må være falsk, da **`false`** AND'et med hvad som helst giver **`false`**. Det er denne tunede effekt **`&&`** og **`||`** har. De springer direkte til konklusionen, hvis de ud fra deres første operand (venstre-siden) kan drage den endelige slutning uden at evaluere højre-siden. For AND er det hvis venstre-siden er **`false`**, mens det for OR er hvis den er **`true`**.

Evaluering med side-effekt

Som sagt er **`&&`** og **`||`** de normalt anvendte operatorer for AND og OR, eftersom de har denne tunede effekt på udtryk. Men spørgsmålet står tilbage: Hvorfor findes de andre? De andre findes fordi man en sjælden gang kan være interesseret i, at det samlede udtryk evalueres færdigt. Hvorfor dog det? Det er man interesseret i, hvis udtrykket har en såkaldt **side-effekt**. Et temlig intetsigende eksempel på dette kunne være:

```
2 < antal & antal++ < 10
```

Side-effekter er rodede

hvor man gerne vil have inkrementeret **`antal`**, nu man alligevel har fat i den ved evalueringen af udtrykket. Man kunne godt komme med lidt bedre eksempler, men jeg har alligevel valgt et dårligt, da man ikke bør lave udtryk med side-effekter. Det er ganske enkelt rodet programmering, der giver nedsat læsbarhed.

2. if-sætningen

2.1 Den simple if-sætning

if-sætningen svarer til konstruktionen med "hvis", der blev brugt i kapitlet "Algoritmer". **Syntaksen** for en if-sætning er som følger:

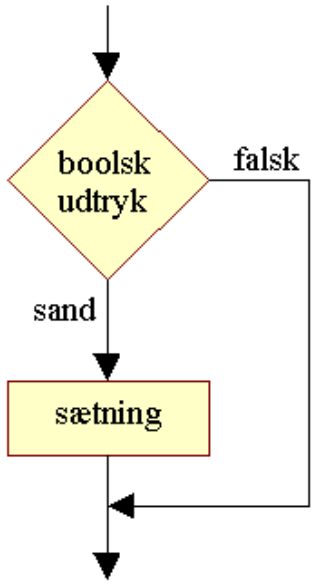
Syntax 1:
Simpel if-sætning

```
if ( boolsk udtryk )
  sætning;
```

Man skal bemærke paranteserne. De er ikke kun med af hensyn til læsbarheden, de er en del af syntaksen og *skal* altid være der.

Semantikken er: Hvis det boolske udtryk er sandt udføres sætningen, ellers ikke.

Man kan bruge et **dataflow diagram** til at illustrere forløbet i udførelsen af en if-sætning:



Figur 1:
Simpel if-sætning

Vi kommer ind fra oven. Afhængig af om det boolske udtryk er sand eller falsk følger vi pilene rundt. Hvis udtrykket er sand, skal vi en tur via sætningen (udføre den), i modsat fald slipper vi udenom.

Et eksempel på anvendelsen kunne være:

Source 2:
Anvendelse af simpel if-sætning

```
int indtægt=10000, udgift=7000;

if ( indtægt > udgift )
  System.out.println( "Der er overskud i kassen" );
```

Her vil den anførte besked kun blive udskrevet, hvis **`indtægt`** er større end **`udgift`**.

2.2 Den sammensatte sætning

Én sætning

Som man bemærker, er if-sætningen kun i stand til at håndtere **én sætning**. Den kan dermed kun betinge udførelsen af en enkelt sætning i forhold til det boolske udtryk. Det er naturligvis utilstrækkeligt, og man har derfor en syntaktisk konstruktion, der kaldes en **sammensat sætning** (compound statement). **Syntaksen** for en sammensat sætning er:

Syntax 2:
Sammensat sætning

```
{ sætning 1;
  sætning 2;
    ...
  sætning n; }
```

Her kan man altså samle et vilkårligt antal sætninger, og på den måde gør dem til en samlet enhed - én sætning.

Tuborg paranteser

De lidt underlige paranteser har for øvrigt en speciel dansk betegnelse, man kalder dem **tuborg paranteser**. Hvis man ser bag på en ølvogn (fra tuborg formodes) er der øverst på udsmykningen en bue, der ligner en sådan parantes. Hvis man så forestiller sig en

væltet ølvogn, der ligger på siden, så er analogien fuldendt, idet ølvognen kan ligge på den ene eller den anden side alt efter om det skal være en højre- eller venstre-parantes.

Lad os se hvorledes det kunne anvendes til at udvide vores eksempel fra før:

Source 3:
Anvendelse af simpel if-sætning, med sammensat sætning

```
int indtægt=10000, udgift=7000, overskud;

if ( indtægt > udgift ) {
  overskud = indtægt - udgift;
  System.out.println( "Der er et overskud på " + overskud + " kr." );
}
```

Nu kan der foregå flere ting, men stadig betinget af, at det boolske udtryk er sandt. Man kunne måske ønske sig at der alternativt skete noget andet hvis udtrykket var falsk. Til dette formål har man en udvidet version af if-sætningen, den såkaldte if-else-sætning.

2.3 if-else-sætningen

Syntaksen er en udbygning af den simple if-sætning:

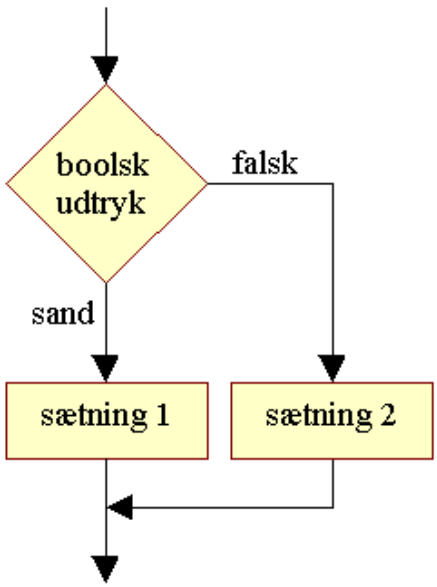
Syntax 3:
if-else-sætning

```
if ( boolsk udtryk )
   sætning 1;
else
   sætning 2;
```

Semantikken er: Hvis det boolske udtryk er sandt udføres sætningen 1; hvis det boolske udtryk er falsk udføres sætning 2.

Lad os ligeledes se et **dataflow diagram**, der illustrerer forløbet i udførelsen af en if-else-sætning:

Figur 2:
if-else-sætning



Vi kunne udvide vores eksempel fra tidligere, til også at reagere på et evt. underskud:

Source 4:
Anvendelse af if-else-sætning

```
int indtægt=10000, udgift=7000, overskud, underskud;

if ( indtægt >= udgift ) {
```

```
  overskud = indtægt - udgift;
  System.out.println(
    "Der er et overskud på " + overskud + " kr." );
} else {
  underskud = udgift - indtægt;
  System.out.println(
    "Der er et underskud på " + underskud + " kr." );
}
```

Bemærk at det boolske udtryk er ændret, da det nok er en anelse mere naturligt at tale om et overskud på 0 kr. end et tilsvarende underskud på 0 kr.

2.4 Nestede if-sætninger

Problemet med at tale om et overskud på 0 kr. kunne vi håndtere ved at dele op i tre situationer i stedet for to:

Source 5:
Nestede if-sætninger

```
int indtægt=10000, udgift=7000, overskud, underskud;

if ( indtægt > udgift ) {
  overskud = indtægt - udgift;
  System.out.println(
    "Der er et overskud på " + overskud + " kr." );
} else if ( indtægt < udgift ) {
  underskud = udgift - indtægt;
  System.out.println(
    "Der er et underskud på " + underskud + " kr." );
} else
  System.out.println( "Det kunne være det samme!" );
```

Det vi gør her, med at kombinere flere if-sætninger inden i hinanden, kalder man **nestede** if-sætninger.

Lad os analysere den nestede struktur ved at lave **konventionelle indrykninger** der afspejler sætningsstrukturen:

Pseudo 1:
Nestede if-sætninger

```
if ( boolsk udtryk )
  sætning;
else
  if ( boolsk udtryk )
     sætning;
  else
    sætning;
```

Hvis vi ser på den første if-else-sætning, er dens sætning 2 endnu en if-else-sætning. Den sidste er bygget ind i den anden. Dette er tilladt da en if-else-sætning netop selv er en sætning.

Selv om nestede if-sætninger ikke er noget problem, hvis man holder tungen lige i munden, er der en klassisk situation som kan give lidt problemer. Hvis vi i vores eksempel ovenfor lader sætning 1 være en simpel if-sætning og sætning 2 være en vilkårlig sætning, får vi billedet:

Pseudo 2:
Uden indrykning

```
if ( boolsk udtryk )
if ( boolsk udtryk )
sætning;
else
sætning;
```

Her er der med vilje ikke lavet nogen indrykninger, for spørgsmålet er: Hvilken if hører

Hvem skal have else?

else'en til? I princippet er det et ubesvaret spørgsmål når man generelt taler programmeringssprog med denne konstruktion. Fra tidernes morgen fik man ikke slået fast hvordan det hang sammen, og det blev overladt compiler-konstruktørerne at svare på spørgsmålet. Nu er problemet i virkeligheden ikke så stort, for længe inden Java kom til verden, havde der dannet sig **koncensus** om spørgsmålet, men f.eks. Pascal findes i versioner der er indbyrdes modstridende på dette punkt (*tidlige* Pascal compilere!). Java og alle andre sprog, i vore dage, er enige om at det er den nærmeste foregående if der får else! Dermed vil de korrekte indrykninger være:

Pseudo 3:
Indrykning svarende til korrekt sammenhæng

```
if ( boolsk udtryk )
  if ( boolsk udtryk )
    sætning;
  else
    sætning;
```

Nu kunne det måske være, at man i virkeligheden ønskede at det skulle være anderledes. I så fald realiserer man det vha. tuborg-paranteser:

Pseudo 4:
Tuborg paranteser gennemtvinger anden sammenhæng

```
if ( boolsk udtryk ) {
  if ( boolsk udtryk )
    sætning;
} else
  sætning;
```

Man ser at den inderste if-sætning bliver lukket inde, og at den første if-sætning bliver det nærmest foregående if, på samme **sætningsniveau** som else.

Præcedens betragtning

Reglen om, at **`else`** hører til den nærmeste foregående **`if`**, kan betragtes som en slags præcedens-regel. Set fra den synsvinkel, kan man ligeledes se på tuborg-paranteser som svarende til de almindelige paranteser i præcedens-sammenhæng - tuborg-paranteserne kan nemlig bruges til at ændre sammenhængen mellem **`if`** og **`else`**.

3. switch-sætningen

Hvis der er mange valgmuligheder kan et program visse steder blive én lang **`if-else`**-sætning der kan være vanskelig, og ikke mindst træls, at lave.

Betragt følgende menu fra de "gode gamle dage" før det **grafiske brugerinterface**:

Figur 3:
Gammeldags menu-billede

Til implementationen af denne menu kunne man anvende følgende skitserede del-program, der foretager en selektion baseret på den talværdi som brugeren har indtastet:

Pseudo 5:
Mange nestede if-else-sætninger

```
if ( valg == 0 ) {
  // Afslut programmet
  System.out.println( "Programmet afsluttes" );
  ...
} else if ( valg == 1 ) {
  // Valgmulighed1
  ...
} else if ( valg == 2 ) {
  // Valgmulighed2
  ...
} else if ( valg == 3 ) {
  // Valgmulighed3
  ...
} else if ( valg == 4 ) {
  // Valgmulighed4
  ...
} else {
  // Forkert valg
  System.out.println( "Forkert valg" );
}
```

I disse situationer har Java en speciel kontrolstruktur, der til dels kan betegnes som **syntaktisk sukker**. Det er switch-sætningen. Med switch-sætningen vil det skitserede del-program blive:

Pseudo 6:
Tilsvarende switch-sætning

```
switch ( valg ) {
  case 0:
           // Afslut programmet
           System.out.println( "Programmet afsluttes" );
           ...
           break;

  case 1:
           // Valgmulighed1
           ...
           break;

  case 2:
           // Valgmulighed2
           ...
           break;

  case 3:
           // Valgmulighed3
           ...
           break;

  case 4:
           // Valgmulighed4
           ...
           break;

  default:
           // Forkert valg
           System.out.println( "Forkert valg" );
}
```

Indgang for hver værdi

Man placerer det udtryk, hvis værdi man vil basere selektionen på, i parantes efter **`switch`**. Dernæst laver man en **indgang** for hver af de mulige værdier, der starter med **`case`** efterfulgt af værdien og et kolon. Efter hver af disse indgange placerer man de sætninger der skal udføres hvis udtrykket har den til indgangen hørende værdi, og afslutter med et **`break`**. Man kan evt. anføre en **default indgang**. Hvis ingen af de anførte værdier ved indgangene passer, vil default-indgangen blive anvendt.

Lad os generelt eksemplificere syntaksen for switch-sætningen:

Syntax 4:
switch-sætningen

```
switch ( udtryk ) {
  case værdi:
              sætning;
              ...
              sætning;
  case værdi:
  case værdi:
              sætning;
              ...
              sætning;
  default:
              sætning;
              ...
              sætning;
}
```

Man kan beskrive syntaksen mere nøjagtig med BNF, men det gør den vanskeligere at læse. Det betyder dog, at ovenstående skal have et par afklarende ord med på vejen.

De værdier, og dermed den type som udtrykket evauleres til, skal være enten **`char`**, **`byte`**, **`short`** eller **`int`**. Det må derfor ikke være **`boolean`** eller **`long`**.

Literaler

Værdierne skal være **literaler**, det må ikke være udtryk.

`break`

Når sætningen **`break`** udføres, bevirker det at der springes ud af hele switch-sætningen.

Er man først sluppet ind et sted, fortsætter man til man møder et `break`

Hvis man ikke bruger et **`break`** som den sidste sætning i rækken af sætninger for en indgang, vil udførelsen fortsætte med de sætninger der hører til den næste indgang. Det sker uden skelen til om den anførte værdi for næste indgang er lig med udtrykket. Med andre ord: er man sluppet ind ét sted kan man bare kører det hele igennem, med mindre man møder et **`break`**. Det er sjældent at man bruger denne effekt, og man ser stort set altid at den sidste sætning for hver indgang er **`break`**. Dog er der en situation hvor man undlader **`break`**. Det er når man ønsker at flere forskellige værdier skal give anledning til at de samme sætninger udføres.

Lad os se et simpelt eksempel på dette:

Source 6:
Det samme for forskellige værdier

```
switch ( valg ) {
  case 3:  System.out.println( "3" );
           break;

  case 5:
  case 7:  System.out.println( "5 eller 7" );
           break;
}
```

Her vil det samme blive udført uanset om tal er 5 eller 7.

Default-indgang ikke nødvendig

Man behøver ikke anføre en default-indgang, som det ses i eksemplet ovenfor. Bemærk at default-indgangen ikke behøver et afsluttende **`break`**, da den er den sidste sætning. Man ser dog alligevel tit at **`break`** er brugt som den sidste sætning i default-indgangen, fordi programmører er bange for at komme ud af **vane** med at sætte **`break`**'s de andre steder (det kan være en *meget* frustrerende fejl at lede efter).

Kan ikke angive intervaller

Man kan ikke angive intervaller i switch-sætninger, andet end at man gør det ved at opremse værdierne med en række indgange. Dette er en mangel ved Java's switch-sætning, som f.eks. Pascal ikke lider af.

4. ?: operatoren

?: er den eneste **tertiære operator** i Java. Den tager, som navnet siger, tre operander og evaluerer dem til en værdi. **Syntaksen** er:

Syntax 5:
Den tertiære operator

```
boolsk udtryk  ?  udtryk 1  :  udtryk 2
```

Semantikken er: Hvis det boolske udtryk er sandt evalueres det samlede udtryk til værdien af udtryk 1, ellers til værdien af udtryk 2.

Man observerer det nære **slægskab** med if-else-sætningen og følgende to stykker kode er da også **ækvivalente**:

Source 7:
Slægtskab med if-else-sætningen

```
int værdi=3;

if ( boolsk udtryk )
  værdi = værdi+5;
else
  værdi = 2*værdi;
```

```
int værdi=3;

værdi = boolsk udtryk ? værdi+5 : 2*værdi;
```

?: er ikke særlig læsbar

I almindelighed regner man ikke ?:-operatoren for nogen "pæn" operator. Det skyldes at de fleste ikke mener at den er så nem at læse. Nu skriver man naturligvis ikke Java programmer til folk der ikke kender Java (så skulle man jo nok have valgt noget andet), men selvom man har brugt operatoren igennem længere tid, bliver den aldrig "pæn". Den er **syntaktisk** så meget anderledes end alt andet i Java, at den altid vil være **den grimme ælding** blandt operatorerne.

God i en snæver vending

Der er dog visse situationer hvor den kommer til sin ret, f.eks, når man i en snæver vending skal vælge mellem to muligheder og en if-sætning blot vil gøre et lille problem til en stor løsning. Det kunne f.eks. være i en udskrift, med mulighed for både entals og flertals angivelse:

Source 8:
Ental eller flertal

```
int tilbage=5;

System.out.println( "Du skal have " + tilbage +
                    (tilbage==1 ? " krone" : " kroner") + " tilbage" );
```

Kun assignments har lavere præcedens end ?:

?: har en meget lav præcendens, og det er naturligvis af hensyn til de tre udtryk der indgår, ellers ville disse altid skulle i parantes. Det betyder at man, som i eksemplet her, må pakke det ind i paranteser, hvis det optræder som et deludtryk.

5. Opdateret Præcedens Tabel

I dette kapitel er der blevet føjet en hel del nye operatorer til vores samling, lad os derfor se en præcedenstabel, hvor de optræder sammen med de operatorer vi har fra tidligere. De nye operatorer er fremhævet med en mørkere gul baggrund.

Tabel 7:

Operatorer	Evalueringsretning

Ny præcedens-tabel

`( )`	`->`
`++ -- + - !` (typecast)	`<-`
`* / %`	`->`
`+ -`	`->`
`< <= > >=`	`->`
`== !=`	`->`
`&`	`->`
`^`	`->`
`|`	`->`
`&&`	`->`
`||`	`->`
`?:`	`<-`
`= += -= *= /= %=`	`<-`

Repetitionsspørgsmål

1 Hvad er selektion?

2 Hvad betyder dynamisk, i programmering?

3 Hvad er programmets tilstand?

4 Hvad er et boolsk udtryk?

5 Hvad har højest præcedens: sammenlignings-operatorerne eller de aritmetiske operatorer?

6 Hvad er operatoren for lighed, og hvilket problem giver det?

7 Hvad er operatoren for ulighed?

8 Står = altid først eller sidst, når der bruges to tegn til én operator?

9 Hvad er en boolean?

10 Hvad kræves der for at AND evalueres til `true`?

11 Hvorfor kan man ikke skrive: `2 < antal < 10`?

12 Hvad har højest præcedens: sammenlignings-operatorerne eller de rent boolske?

13 Hvad kræves der for at OR evalueres til `false`?

14 Hvad er XOR?

15 Hvilken præcedens har XOR i forhold til de tunede AND og OR?

16 Hvad bruger man NOT til?

17 I hvilken præcendens-gruppe befinder NOT sig?

18 Hvad vil det sige at `&&` og `||` er tunede?

19 Hvad er en evaluering med side-effekt?

20 Hvordan løser man problemet med at if-sætningen kun kan håndtere en/to sætninger?

21 Hvorfor hedder det tuborg-paranteser?

22 Hvad er nestede if-sætninger?

23 Hvem får else?

24 Hvilke krav stilles der til en række nestede if-sætninger, for at de kan erstattes med en switch-sætning?

25 Skal man i switch-sætningen nødvendigvis anføre en default?

26 Kan man angive intervaller i switch-sætningen?

27 Hvad vil det sige at ?: er en tertiær operator?

28 Hvilken præcedens har ?: operatoren?

Svar på repetitionsspørgsmål

1 Selektion er valg

2 Noget der er dynamisk - er noget der tager form under programudførelsen

3 Samtlige variables værdier (og det sted man er kommet til i programudførelsen)

4 Et udtryk der evalueres til enten **`true`** eller **`false`**

5 De aritmetiske har højest præcedens

6 ==. Problemet er, at man intuitivt vil bruge = i stedet for, hvilket er en fejl

7 !=

8 = står altid sidst

9 Java boolske type, der kan antage værdierne **`true`** og **`false`**.

10 At begge operander er sande.

11 Fordi udtrykket, som ethvert andet, evalueres fra venstre mod højre

12 Sammenlignings-operatorerne har højest præcedens

13 At begge operander er `false`

14 En kræsen udgave af OR, der ikke vil have at begge operander er `true`

15 XOR har højere præcedens

16 Til at vende sandhedsværdier

17 I gruppe med de unære aritmetiske operatorer (fortegn)

18 Hvis de på grundlag af en evaluering af venstre-siden kan fastslå resultatet uden at evaluere højre-siden, dropper de højre-siden

19 Evaluering af et udtryk, der ændrer en eller flere af de variable der optræder i udtrykket

20 Man anvender en sammensat sætning, der samler flere sætninger til én

21 På grund af ligheden med den bue man kan se bag på ølvogne

22 if-sætninger der er inde i hinanden

23 Den nærmeste foregående if

24 Det samme udtryk skal sammenlignes med en række literaler

25 Nej

26 Nej, kun ved at opremse alle værdierne i intervallet

27 At den tager tre operander

28 Den er meget lav. Kun gruppen med assignments er lavere