if- och while-satser, Slumptal, m.m

Vecka 1, Bildserie 3

Innehåll

- Selektion: if, if-else och if-else if-satserna
- Redundant kod
- Iteration: while-satsen
- OBOE
- Terminering
- Nästlade satser
- Loop and a half
- Slumptal
- Felsökning

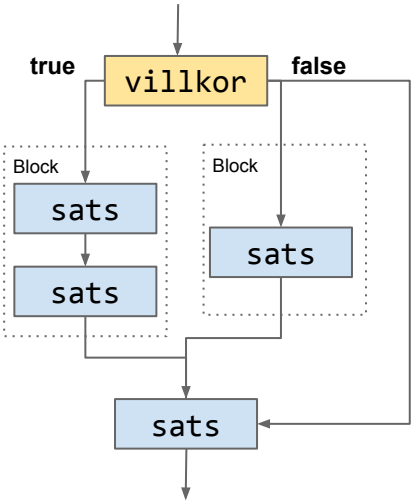
Att Läsa i Boken

- 3.3-3.6
- 3.15-3.16
- 5.1-5.2
- 9.6.2

OBS! Att jag använder lite annorlunda stil

- Inledningsvis behövs aldrig ordet static eller public, bortse från dessa.

Selektion



Våra program har hittills bara bestått av en följd av satser

- En **sekvens** av satser
- Detta räcker inte: Vi behöver kunna styra, mer i detalj, vilka satser som skall exekveras (och i vilken ordning, control flow)
- T.ex. måste programmet kunna göra val (utifrån olika villkor)
  - Om projektilen träffar … , Om kontot är positivt, … Om lösenordet stämmer …

Val kallas **selektion**

- Ett val mellan satser (vi skriver alltid som ett val mellan block)
- Valet styrs av ett villkorsuttryck (ett boolesk uttryck)
- Selektion skrivs i Java som: if, if-else, if-else if eller switch satsen.

if-satsen

```java
int i = 4;

// If expression true ...
if (i % 2 == 0 ) {
    out.println("i is ..."); // .. do this
}
// ... else continue here
```

if-satsen

- Styrs av villkorsuttryck i parentesen
- Uttrycket måste ha typen boolean
- Om sant körs blocket direkt efter villkoret
- Annars fortsätter programmet efter blocket (blocket hoppas över)

OBS! Stilen

- Inledande { på samma rad som if
- Indentera satser i blocken (sköts som sagt av IntelliJ)
- Som sagt: Inget ; efter ett block

if-else satsen

```
int i = 4;

// If expression true ...
if ( 0 <= i && i < 4 ) {
    out.println("i is ..."); // .. do this
} else {
   out.println("i is ...");  // else this
}
// Continue here
```

Som if-satsen men om villkoret är falskt så körs blocket vid else

if-else if-satsen

```
if (j == 3) {                   // if expression true ...
   out.println("j is 3");       //... do this...
} else if (k <= 20) {           // ... else if this true ...
   out.println("k <= 20");      // ... do this ...
} else if ...                   // ... etc.
   ...
} else {                        // ... else ...
   out.println("j != 3 and k > 20");   // ...do this
}
// Continue here
```

Villkoren evalueras ett i taget uppifrån och ner.

- Om något villkor sant så körs blocket direkt efter.
  - Därefter fortsätter programmet efter satsen (efter sista blocket)
  - D.v.s.: om ett villkor sant så exekveras inga andra
- Om inget är sant så körs blocket vid else.

Programmering

```
Input 3 integer (space between) > 3 5 1
Biggest was 5
```

Hur skriva detta program?

- Använd if

Fallgropar

```
int n = ...;
if (n > 0)
   if (n < 5)
       out.println("a");
else
   out.println("b"); // n <= 0 or n >= 5?


if (nCoins <= 0);
   out.println("..."); // Will always run
```

Använd alltid block!

Fallgropar (pitfalls)

"Dangling else"

- Indenteringen ger ett felaktigt intryck av vilka if och else som hör ihop!
- Java tar inte hänsyn till indentering
- else tillhör närmsta if (normalt sköter IntelliJ indenteringen)

Tomma satsen

- Inget ";" efter villkorsparentesen.
- Om så körs den tomma satsen!

Använd alltid block för att visa vad som skall köras (gäller if-sats m.fl)

- Gäller även om bara en enda sats skall köras!
- Så här skall det se ut!

```
if( nCoins <= 0){
    out.println(...);
}
```

Redundant Kod

```
// Bad redundant code!
if ( … ) {
    …
    dices = 1;
} else {
    …
    dices = 1;
}
```

```
// Non redundant
if ( …  ) {
   …
} else {
    …
}
dices = 1;
```

Redundant kod är dåligt, skall inte förekomma!

- Tänk till och organisera om koden!

God Praxis

```
// Hmm, what does it mean? Bad
if( score1 >= 10 || score2 >= 10 ){
    if( score1 != score2) {
        ...
    }
}
// Also bad
if( score1 >= 10 || score2 >= 10 && score1 != score2) {
...
}
// Better
boolean gameOver = score1 >= 10 || score2 >= 10 && score1 !=
score2;
if( gameOver ) {  // Much clearer!
...
}
```

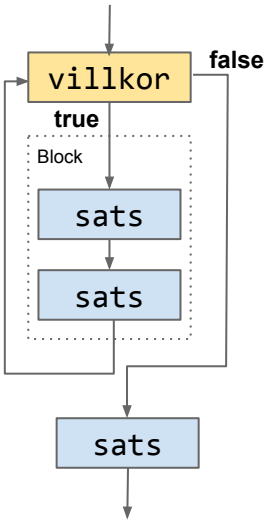
Ofta kan man göra saker på flera olika sätt

- Det som allmänt anses bäst kallas god praxis (best practice)

Exempel på best practices för booleska uttryck och if

- Om uttrycken blir för komplexa, … skapa en boolesk variabel med ett beskrivande namn
- Mycket lättare att förstå

Iteration



Iteration är en upprepning av satser

- Iterationen styrs av ett villkor på samma sätt som selektion (typ boolean)
- Om villkoret sant så körs efterföljande block och programmet "hoppar" upp till villkoret igen
- Om falskt så hoppas blocket över
- Iteration skrivs i Java som while-, for- eller do-satser (vi använder inte do)
- Iterationer kallas ofta "**loopar**"

while-satsen

```
int value = 0;
while (value < 5) {
   out.println(value);
   value++;           // Last in loop
}

out.println(value);  // Value is ?
```

while-satsen

- Styrs av villkorsuttrycket i parentesen (typen boolean)
- Loopen använder en “räknare” (loop variabel) för att styra upprepningen
- Normalt skall loopvariabeln ändras i loopen för att så småningom göra villkorsuttrycket falskt.

Upp eller nedräkning i Java: Börjar normalt på 0 (alltså inte 1), man räknar från 0 och uppåt eller
till 0 (inklusive), nedåt.

Terminering

```
while( i < 5 ){
    …            // Must increment/change i !
}

while( … );{    // No ; !
    i++;
}

while( …  != 0.01){    // Floating point not exact !
}

while( i >= 0 ){    // Bad boolean expression!
    i++;
}
```

Terminering betyder att något avslutas (i detta fall en loop)

- Händer ibland att man missar och får en loop som inte **terminerar** (körs för evigt)
- Man upplever att “inget händer” trots att programmet inte har avslutats

Saker att se upp med

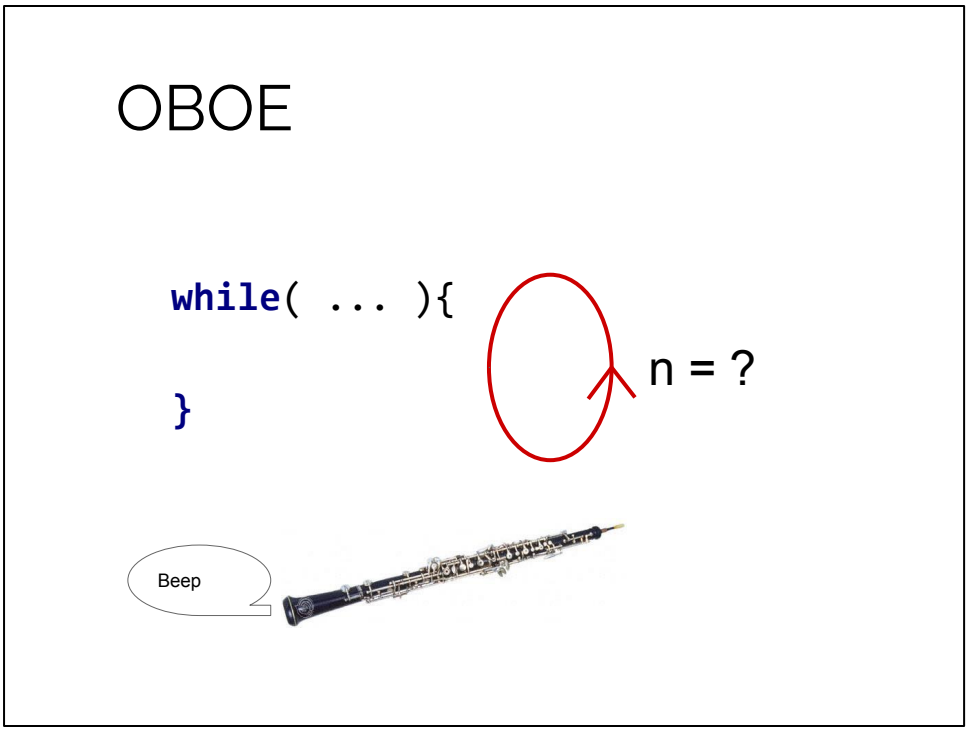
- Villkoret måste påverkas i satsen, det måste bli falskt förr eller senare … (eller använd break, se senare)
- Inget semikolon efter parentesen, innebär att den tomma satsen körs i för evigt
- Flyttal skall inte användas i villkoret
- Felaktigt villkor i uttrycket
    - Ofta bättre att använda <=, >= i stället för == eller != om man skulle missa det exakta värdet.
    - Se upp med || i uttryck, föredra &&!

Programmering

```
Input x > 1
Exp(x) = 2.7182818351251554
```

Hur skriver man programmet?

- Alltså beräkna värdet av e^x för något x
- Måste använda detta (finns färdiga funktioner för exp(), skall inte användas)



Ett mycket vanligt fel är att man kör loopen ett varv för mycket eller för litet

- Känt som “off by one error” (OBOE)
- Måste alltid ha klar för sig exakt hur många var loopen körs!

God praxis för loopar

- Använd loopräknaren enbart till att räkna upp eller ned (behövs något mer, skapa en till (eller flera) andra variabler.
- Ändringen av räknaren gör man normalt sist i blocket.

Nästlade if och while

```
while( ... ){
   …
   if( ... ){
        …
   }else {
   …
   }
   …
}
```

```
if( ... ){
   …
   while( ... ){
        …
   }
   …
}
```

Nästalade styrande satser innebär

- if- och/eller while-satser inuti if- och/eller while-satser (eller andra satser, mer kommer …)
- Den STORA MAGIN …!!!
- Genom att kombinera sekvenser, styrande och nästlade styrande satser skapar vi ett logiskt flöde som utför det programmet är tänkt att utföra
- Kan bli komplext att förstå
  - Mer än tre nästlade nivåer skall undvikas!
  - Noggrann indentering för att underlätta läsning

Loop and a Half

```
while (true) {
    out.print("Input positive int > ");
    int i = scan.nextInt();
    if (i < 0) {
        break;
    }
    // ... Possibly more here
}
out.print("Loop ended");
```

Loop and a half

- Innebär att halva loopen skall köras minst en gång
  - Om man t.ex. skall läsa ett värde i loopen som påverkar villkoret, d.v.s. ev. avbryta loopen
- För att lösa detta används “loop and a half”-mönstret…
  - En variant använder while(true) i kombination med **break**-satsen
    - I princip är loopen “evig”, det finns ingen räknare, typ i++
  - break-satsen innebär att programmet hoppar ur den omedelbart omslutande while-satsen, till första sats efter blocket

Metodik

```
// Once
...
statement;
statement;
statement;
out.println(result);
```

```
// Many
while(...) {
  ...
  statement;
  statement;
  statement;
  out.println(result);
}
```

Om man kan göra något en gång, är det lätt att göra det flera gånger …

- … lägg en loop runt
- D.v.s. vänta med den omslutande loopen, gör klart de satser som skall upprepas först, kontrollera att det fungerar en gång.
- Därefter lägg en loop runt.

Programmering

```
Input an even integer (0 to quit) > 2
Ok! That was even.
Input an even integer (0 to quit) > 1
That wasn't an even number.
Input an even integer (0 to quit) > 3
That wasn't an even number.
Input an even integer (0 to quit) > 0
Yot got 1 correct out of 3
```

Hur skriva programmet?

Slumptal

```
import java.util.Random;

void program() {
   int computer;
   // Create random generator
   Random rand = new Random();
   // Ask it for a random int
   computer = rand.nextInt(3) + 1; // 1-3

}
```

I spelprogram, simuleringar m. m. behövs ofta slumptal

- I Java-program kan man använda en slumptalsgenerator
- Man måste skapa en generator
- När man skapat generatorn kan man be den om olikt typer av slumptal t.ex. slumpmässiga heltal

Analys av kod

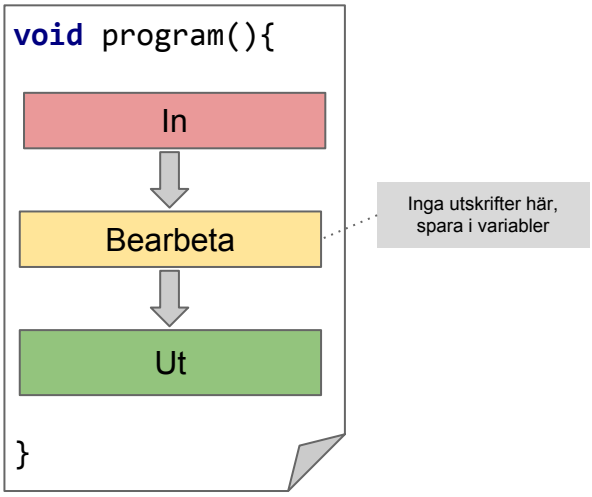
- Vi skapar en slumpgenerator benämnd rand (detaljer kommer senare …)
- Uttrycket rand.nextInt(n) ger ett slumptal i intervallet 0 till (n-1)
  - Vi säger åt slumpgeneratorn rand att generera ett heltal 0-2, intervallet flyttas därefter till 1-3
  - Argumentet n måste vara positivt
- Resultatet tilldelas computer (typkompatibla).

Programmering

```
Welcome to Number Guess
Guess a number [1-100] > 50
To small
Guess a number [1-100] > 75
To small
...
Guess a number [1-100] > 93
To big
Guess a number [1-100] > 92
To big
Guess a number [1-100] > 91
Correct! Number of guesses needed 8
```

Hur skriva programmet?

God Praxis



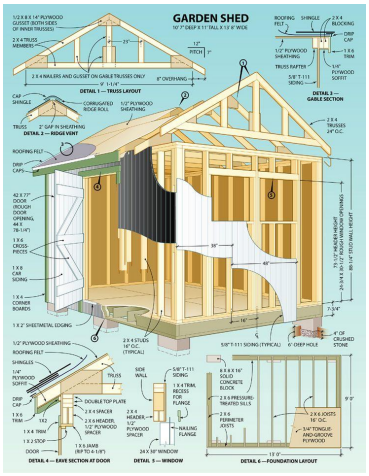
Problem uppstår då vi skriver program, ... kan vara av många olika slag

- Vi kan dela upp problem i olika grupper
  - Syntaxfel, sköta av kompilatorn, vanligen ganska lätta att lösa
  - Logiska fel, vi har helt enkelt tänkt fel någonstans, kan vara (mycket) svåra att hitta
  - Felaktiga utskrifter, logiken är rätt men själva utskriften blir fel
    - Enklare att hitta om logik och IO är separerade (finns på olika platser i programmet)
    - Vi försöker skilja på IO och logik genom att stukturera programmet enligt: In - > bearbeta - > ut (d.v.s läs in data, beräkna (ta fram) resultatet, skriv ut resultat)
    - Vi skriver inte ut resultat direkt utan sparar undan och skriver ut efter bearbetningen

Annan fördel: Vi kanske vill visa programmet på något annat sätt (grafiskt program, app eller web).

- Enklare att anpassa programmet om logiken är separat.

Metodik



Om vi skall bygga ett uthus måste vi ha verktyg, en ritning, material och en arbetsordning (process)

- Vi kan inte börja såga och spika innan vi vet vad/hur och i vilken ordning det skall sågas och spikas!

Om vi skall skriva ett program måste vi ha en skiss och en (övergripande) plan (inte en massa detaljer).

- Vi kan inte börja koda förrän vi har en aning om vad vi vill åstadkomma
- Att skapa en skiss för ett program = att på papper skriv/rita/kladda ner en “ritning”.
- Dessutom försöker vi tänka ut en arbetsordning.
- Därefter använder vi de verktyg vi f.n. känner till: literaler, variabler, initiering, tilldelning, operatorer, in/ut-matning för att implementera programmet …
- …. under tiden vi implementerar (på detaljnivån) får vi ofta prova oss fram, det är ett helt natruligt arbetsätt!
    - Var inte rädd att prova saker!!!!

Inför Övning 1

```
Welcome to Rock, Paper and Scissors
Select 1, 2 or 3 (for R, P or S) > 1
Computer choose: 3
You won
Result 1
Select 1, 2 or 3 (for R, P or S) > 2
Computer choose: 2
A draw
Result 1
Select 1, 2 or 3 (for R, P or S) >
...
Game over!
Draw
```

På övningen kommer ni att jobba med ett program för sten, sax och påse.

- Kommer att ge erfarenheter för att jobba med pågående labben (Nim-spelet).
- Hur tänker man för att skriva ett program …?

OBS! Grupper
Torsdag: A, B
Fredag: C, D