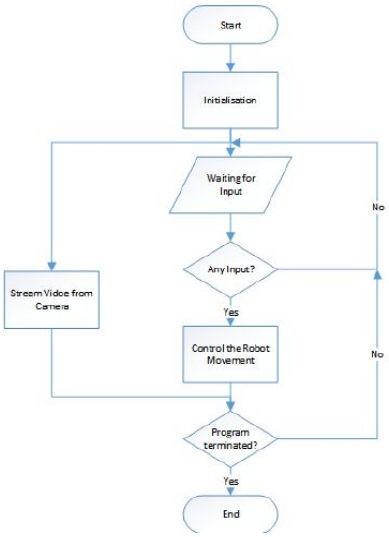
Styrande Satser

DIT012/Joachim von Hacht

Programflöde



2

Ett program byggs upp av satser. Att bara skriva satser en efter en (en **sekvens**) räcker inte för att lösa problem.

Vi behöver två konstruktioner till för att styra i vilken ordning satserna exekveras

- **Selektion** , val (selection). Programmet väljer mellan olika satser.
- **Iteration** , upprepning (iteration). Programmet upprepar ett antal satser
- Selektion och iteration kallas styrande satser, de styr programflödet.. (control flow)
- Dessutom dyker det upp en sats som kan hoppa i flödet (hoppsatser är normalt inte bra, en klassisk artikel)

Finns alltså bara tre olika konstruktioner!

Block med Satser

```
{    // Block start
   out.print("Hello ");
   out.print("world");
   out.println("!");
}   // Block end, no ;
```

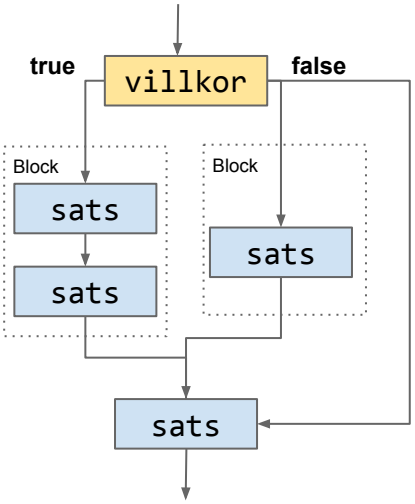
3

En sats kan alltid ersättas med ett block (av eventuellt flera satser)

- Ett block räknas som en sats bestående av 0-n ihopbakade satser
  - Tomma block kan förekomma (undvik)
- Inget ";" efter block *),
  - Behövs inte, det syns var blocket slutar även utan semikolon …
  - .. nämligen vid }
  - Skriver man ; efter så betyder det bara en tom sats efter blocket.

*) Utom vid ett speciellt tillfälle (inget att bekymra sig för i denna kurs).

Selektion



En selektion styrs av ett villkorsuttryck (ett boolesk uttryck)

- Selektion skrivs i Java som: **if**, **if-else**, **if-else if** eller **switch** satser..

if-satsen

```
int i = 4;

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

5

if-satsen

- Villkorsuttrycket skrivs i parentesen efter if
- Uttrycket måste ha typen boolean
- Om sant körs blocket direkt efter villkoret
- Annars fortsätter programmet efter blocket (blocket hoppas över)

Stilen

- Inledande { på samma rad som if
- Indentera satser i blocken (sköts 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

- Även här, se upp men krullparenteser.

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
```

7

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
    - Alltså viktigt i vilken ordning else if skrivs
- Om inget är sant så körs blocket vid else.

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!
...
}
```


Om de booleska uttrycken blir för komplexa, … skapa en boolesk variabel med ett beskrivande namn

- Mycket lättare att förstå

Fallgropar

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


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

Använd alltid block!

9

Fallgropar (pitfalls), saker man får se upp med

"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 detta)

Tomma satsen

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

Använd alltid block för att visa vad som skall köras

- Gäller även om bara en enda sats skall köras!

break-satsen

```
// Must be inside switch-statement or an
// iteration (more to come)
{
    ...
    break;  // Jump from here ...
    ...
}
// ... to here
```


Break-satsen innebär ett hopp ut ur närmsta omslutande block

- Alltså ett hopp i flödet
- Kan bara användas i switch-satsen och i iterationer (mer strax)
    - I iterationer använder vi hopp med försiktighet (så att inte koden blir svårtydd). Vi vill inte hoppa runt “hur som helst”

switch-satsen

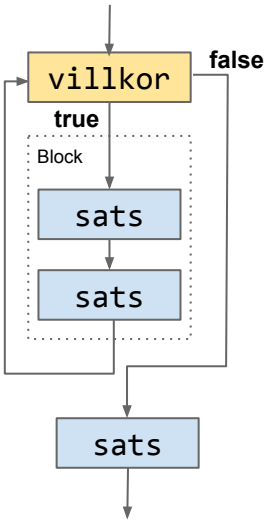
```
// Switch statement
switch (i) {              // i is value to compare for equality
   case 0:                // If match 0 ...
       out.println("match 0"); // ... run this
       break;   // IMPORTANT, else will run "case 1" also
   case 1:
       out.println("match 1");
       break;
   case 2:
       out.println("match 2");
       break;
   default:
      out.println("no match");  // If no match
}
```

11

Switch-satsen är en förenklad selektion där man bara väljer utifrån likhet.

- Villkoret är underförstått likhet.
- Om uttrycket i parentesen efter switch är lika med något av de uppräknade värden i case-”grenarna” så körs satserna i den grenen
- Matchar inget körs default grenen.
- Värdena i grenarna som jämför måste vara konstanta (literaler eller konstant variabler)
- Viktigt med break sist i varje gren, annars kör nästa gren också.
- Switch satsen kan användas för typerna: char, int, String, enum eller omslagstyperna: Character och Integer, m.fl.

Iteration



12

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
  - I detta fall är hopp ok
- Om falskt så hoppas blocket över
- Iterationer kallas ofta **loopar**
- Iteration skriver vi som: while-satser, for-satser, kort for-sats eller forEach.

while-satsen

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

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

13

while-satsen

- Styrs av villkorsuttrycket i parentesen (typen boolean)
- Loopen använder en "räknare" (loop-variabel) för att styra antalet varv i loopen
- Normalt skall loop-variabeln ändras i loopen för att så småningom göra villkorsuttrycket falskt.
    - Ändringen gör normalt sist i loopen.

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

God praxis för loopar

- Använd loop-variabeln enbart till att räkna upp eller ned (behövs något mer, skapa en till (eller flera) andra variabler.

Fallgropar

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

while( ... );{    // No ; after while!
    i++;
}

while( ...  != 0.01){    // double point not exact !
}

while( i >= 0 ){    // Never false!
    i++;
}
```


Händer ibland att man missar och får en loop som inte **terminerar** (körs för evigt, programmet "hänger" sig)

- Man upplever att "inget händer" trots att programmet inte har avslutats

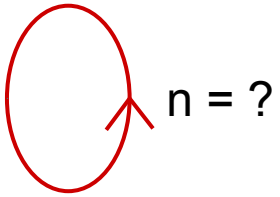
Saker att se upp med

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

OBOE

```
while( ... ){

}
```



15

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

- Känt som “off by one error” (OBOE)
- Ofta orsakat av t.ex. >,< istf >=, <= (eller tvärtom)

Nästlade if och while

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

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

16

Nästlade styrande satser innebär

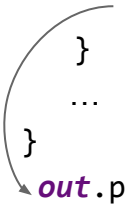
- if- och/eller while-satser inuti if- och/eller while-satser (eller andra satser)

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 är ett måste (IntelliJ sköter det)!

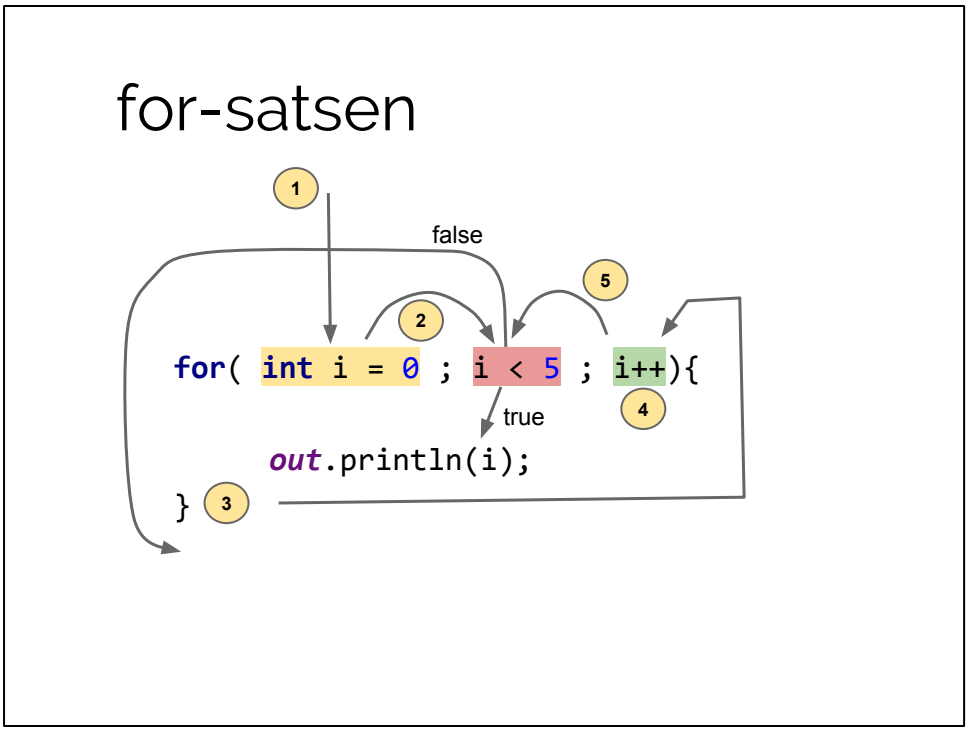
Loop and a Half

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



Ibland vill man köra en del av en loop innan man vet om man skall upprepa

- Om man t.ex. skall läsa ett värde i loopen som påverkar villkoret
- 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++



For-satsen är ett annat sätt att skriva iterationer.

En for-loopen består av 3 delar (åtskilda med ";") inom parentesen och ett efterföljande block

- Första delen av parentesen är en initieringsdel (körs bara en gång, första gången). Använda för att deklarera och initiera loop-variabeln. Variabelns synlighetsområde är bara inom loopen (parentesen och blocket efter)
- Andra delen är villkoret
- Sista delen ändrar loop-variabeln. Denna del körs automatiskt sist i loopen trots att den står på första raden.

Analys av koden i bilden

1. Räknaren (variabeln i) för antal varv deklareras och initieras
2. Villkoret evalueras
    a. Om sant
        - Blocket efter parentesen exekveras
        - När slutparentes i blocket nås sker ett hopp till sista delen av parentesen (3)
        - Räknaren ökas/minskas (4)
        - Därefter evalueras villkoret igen (5)
    b. Om falsk

- Hela blocket hoppas över

OBS!

- Skall vara ";" mellan delarna i parentesen annars konstiga fel
- Som tidigare: Undvik att förändra räknaren i loopen (den skall bara räknas upp/ner i sista delen)

while eller for?

compute the largest power of 2 less than or equal to n	`int power = 1;`
	`while (power <= n/2)`
	`  power = 2*power;`
	`System.out.println(power);`

compute a finite sum (1 + 2 + … + n)	`int sum = 0;`
	`for (int i = 1; i <= n; i++)`
	`  sum += i;`
	`System.out.println(sum);`

19

I Java är while och for logiskt utbytbara, vi kan klara oss med t.ex. bara while dock ...

Vi gör följande:

- while-satsen betecknar konceptet upprepa-tills … vi vet inte på förhand hur många gånger vi skall upprepa
- Ibland vet man på förhand hur många gånger man skall upprepa, i dessa fall använder vi for-satsen (for-loopen)

Nästlade for-satser

```
for (int i = 0; i < arr.length - 1; i++) {
   for (int j = 0; j < arr.length - i - 1; j++) {
       if (arr[j] > arr[j + 1]) {
           int temp = arr[j];
           arr[j] = arr[j + 1];
           arr[j + 1] = temp;
       }
   }
}
```


Nästlade for-satser mycket vanligt.

- Ofta i samband med arrayer, se Arrayer
- Ibland styrs den inre loop-variabeln av den yttre (den inre beror på den yttre).

För att förstå nästlade for-loopar är det vanligen enklast att börja “inifrån”, försök förstå den inre loopen först!

- Gäller även nästlade while

Redundant Kod

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

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

21

Redundant eller duplicerad kod är inte acceptabelt

- Mer (onödig) kod -> mer chanser till fel
- Kod måste hållas i synk, ändringar måste göras på flera ställen.

När ni fått till ett fungerande kodavsnitt …

- .. gör en “code review”!
- Görs något i onödan, eller görs samma sak på flera ställen? Åtgärda!
- Idealet är att allt finns/görs på exakt ett ställe i programmet.

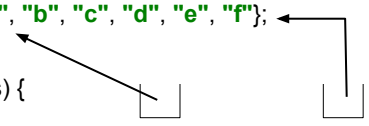
Code smells är tecken på att koden inte är bra!

Kort for-loop

```
int[] is = {1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int i : is) {      // No index just each element, left to right.
   out.print(i);        // i++ sense less
}

String strs[] = {"a", "b", "c", "d", "e", "f"};

for (String s : strs) {
   out.print(s);
   s = "X";  // Senseless
}
```



Tilldelning av s ändrar inte original-array:en

Om man bara vill traversera en array (eller annan samling, mer senare) och inte behöver ett index, finns en enklare for-loop

- Loopen tar ett element i taget från början till slut (index [0-(length-1)]
  - Använd om ni vill … kan förekomma i kodexempel
- Att tilldela loop-variabeln (tex. i och s i koden) är meningslöst.
  - Ändrar inte original array:en
  - Behöver man ändra ett index måste man använda den vanliga for- eller while-loopen.
    - Se även nästa bild
- Kan dessutom ge ConcurrentModificationException i samband med samlingar, see vidare Samlingar

Kort for-loop med Objekt

```
class Point {
      int x, y;
}

Point[] pts = {new Point(3,4), new Point(-1,2), new Point(6,0),};

// Will change object
for (Point p : pts) {
      p.x++;
}
```

Objektet p refererar kommer att ändras

Kort for-loop fungerar bra om man vill modifiera objekt eftersom det inte är själva referensen man ändrar

- Det är ju objektets variabler som ändras.

forEach

```
// Countries is a List<Country>
diceWars
   .getCountries()
   .forEach(this::renderCountry);
```

24

Om man använder en samling finns det en “inbyggd” iteration.

- Påminner om kort for-loop men vi slipper hålla reda på aktuellt element
- Kräver att man skickar en metodreferens som argument (vissa returtyper eller parametrar för metoden krävs).
    - Metodreferensen innebär att varje element i tur och ordning skickas som argument till metoden.
- Se vidare Metoder och Samlingar.