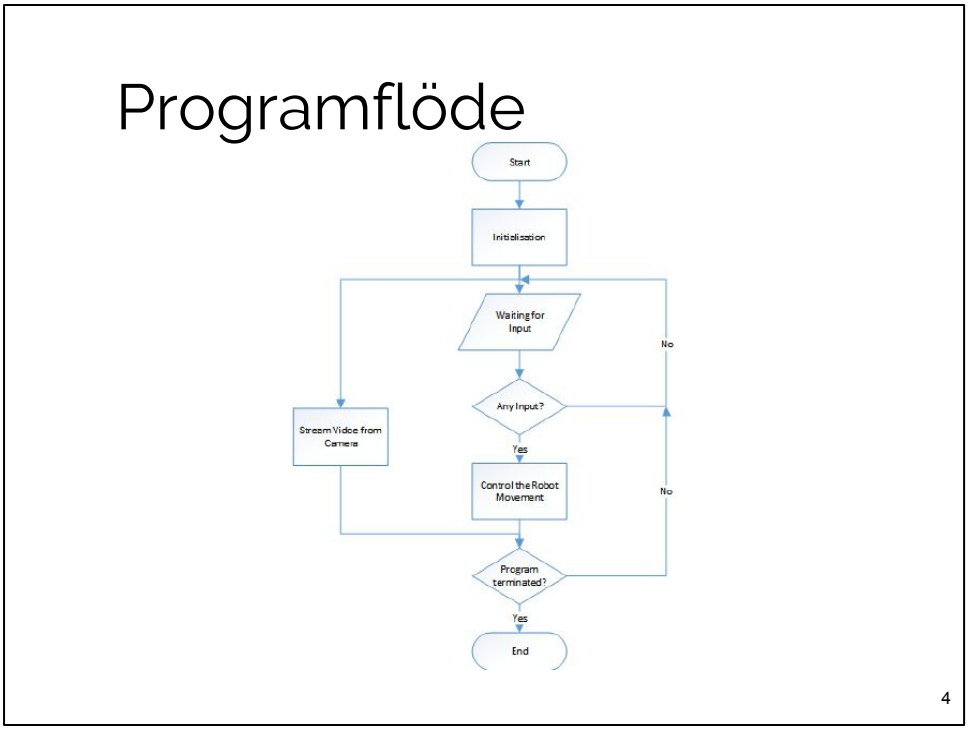
Bilder Vecka 1

TDA548/Joachim von Hacht

Grunderna

(se bildserie Grunderna … självklart kan du inte smälta allt, gå igenom och repetera vartefter under kursens gång)

Styrande Satser



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. Programmet väljer mellan olika satser.
- **Iteration** , upprepning. 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, sekvens, selektion och iteration!

Block med Satser

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

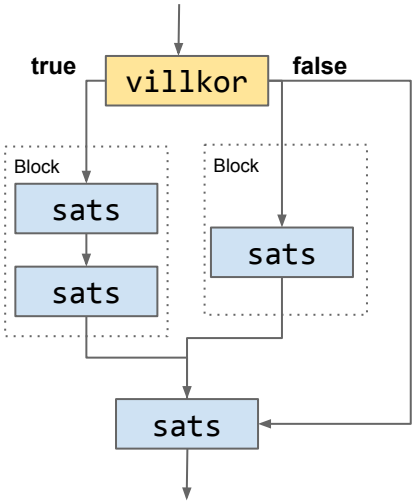
5

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
```


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
```

9

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.

Fallgropar if-satsen

```
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!

10

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!

<u>Använd alltid block för att visa vad som skall köras</u>

- Gäller även om bara en enda sats skall köras!

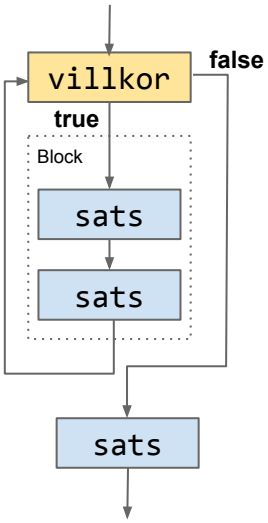
break-satsen

```
// 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”

Iteration



12

Iterationen (upprepning) 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
- 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 (med något undantag, se loop and a half).
- Räknaren 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-satsen

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

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

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

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

14

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

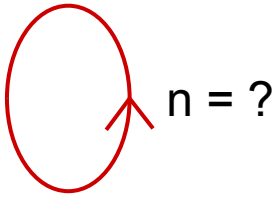
- Man upplever att "inget händer" trots att programmet inte har avslutats

Saker att se upp med

- Villkoret måste påverkas i loopen, det måste bli falskt förr eller senare (undantag: Loop and a half se nedan)
- Inget semikolon efter parentesen, innebär att den tomma satsen körs i för evigt
  Likhet för 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 Styrande Satser

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

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

16

Nästlade styrande satser innebär

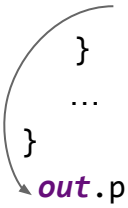
- if- och/eller while-satser inuti if- och/eller while-satser (eller andra styrande satser)

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)!

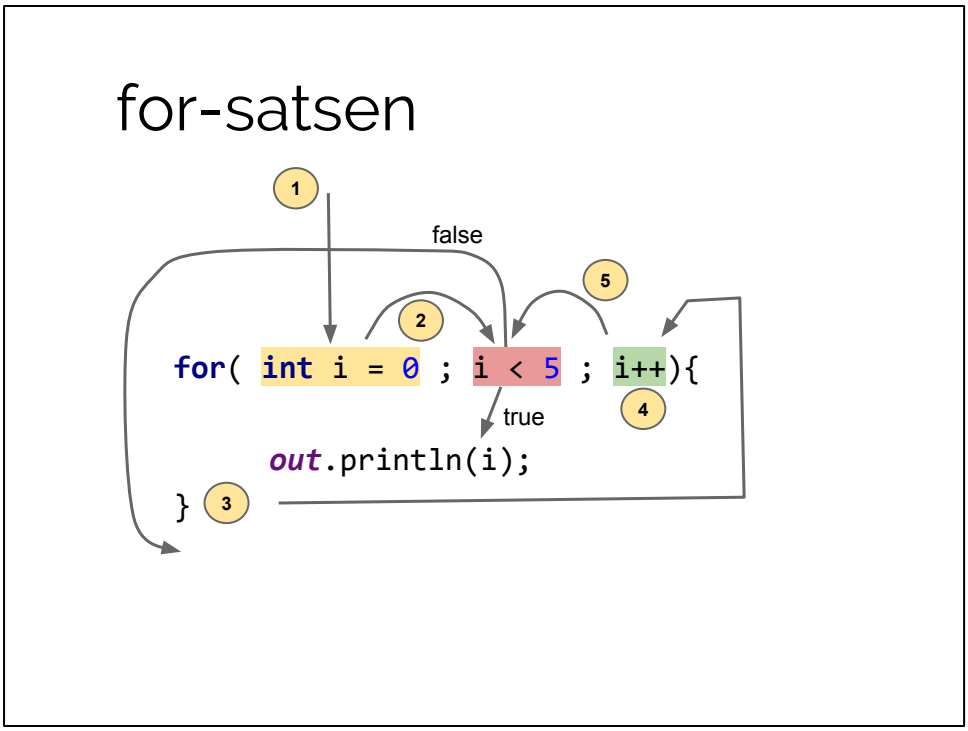
Loop and a Half

```
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

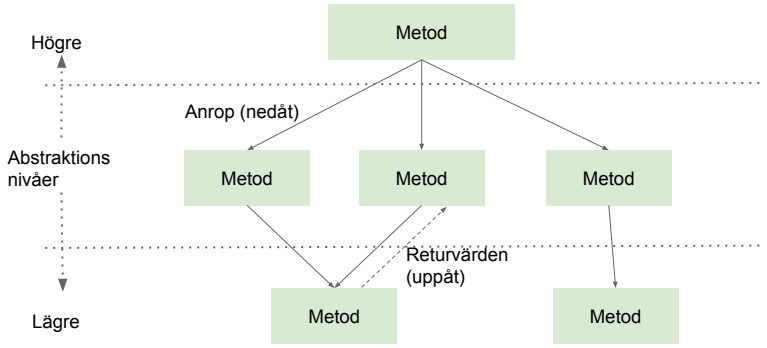
I Java är while och for logiskt utbytbara, vi kan klara oss med t.ex. bara while dock ...

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)

Metoder

Konceptuell bild



22

En metod är en avgränsad del av ett program som utför en viss uppgift

- Ger ofta ett visst utvärde/resultat (motsvarar matematisk funktion) …
- … men inte alltid.

Att använda metoder i ett program ger många fördelar

- Programmet får en struktur, programmet blir greppbart
  - Om ett program överstiger ett par hundra rader börjar det bli ohanterligt ..
  - … genom att strukturera programmet m.h.a. metoder kan vi behärska komplexiteten.
- Vi kan bygga upp programmen bit för bit
  - Man skriver och testar en metod i taget.
- Metoder har namn!
  - Programmet blir lättare att förstå om vi inför begrepp på en högre nivå (säger mer vad det handlar om)
- Metoder kan återanvändas, innebär att vi undviker upprepad kod (redundans)!!
  - Vill vi köra samma kod på flera ställen, anropar vi samma metod
- Innehållet i en metod kan ändras utan att resten av programmet behöver ändras.

I bilden: Metoder arbetar på olika abstraktionsnivåer:

- Metoderna på låg abstraktionsnivå är mycket detaljerade (använder ofta primitiva typer)
- På nästa nivå är de mer övergripande och åstadkommer mer (arbetar med en större del av problemet).
- OBS! Metoder på högre nivå anropar de på lägre. Returvärden skickas från lägre nivå till högre.
- Kallas en skiktad design, mer i senare kurser.
- För att åstadkomma diagrammet, se: Bildserie Arbetssätt och Programkonstruktion

Metoddeklaration

```
// Declaration of method named add
// Parameters int a and int b
// Return type int
int add( int a, int b ){   // Head
  return a + b;            // Body
}
```

24

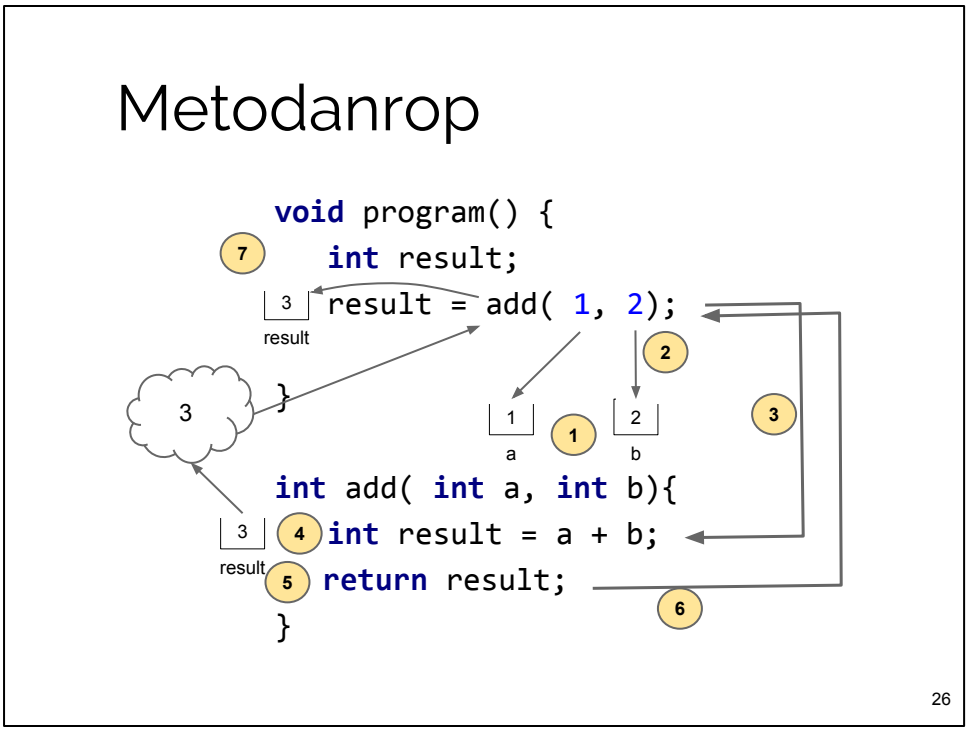
Man skapar en metod m.h.a. en **metoddeklaration**

- Första raden i deklarationen kallas metoden **huvud** (head)
  - I huvudet anges (vänster -> höger): Returtyp, namn och en **parameterlista** inom parentes
    - Returtypen anger typen på värdet som metoden returnerar (om något, mer senare …)
    - Namnet väljer vi själva (efter de regler som finns för namn)
      - Namnet skall vara lagom långt och beskriva vad metoden skall göra, ofta är ett verb inblandat
      - Metodnamn inleds med liten bokstav därefter camelCase.
    - I parameterlistan deklareras ett 0-n variabler (parametrarna).
      - Dessa används för att ta emot inkommande data vid anropet (under körning).
      - För varje parameter anges typ och namn (även här väljer vi namn)
- Koden i blocket efter huvudet kallas metodens **kropp**. I kroppen skrivs den kod som skall köras då metoden anropas
- En **return**-sats i kroppen används för att avsluta metoden och skicka tillbaka värdet som står efter return.

- Om det står ett uttryck efter beräknas detta först
- Returtypen i huvudet och typen på uttrycket som returneras måste vara kompatibla, annars typfel
- Om vi anger en returtyp måste vi returnera ett värde annars kompileringsfel
    - Kompilatorn kontrollerar att det garanterat finns en return-sats som kommer att köras
    - Om man t.ex. har en if-sats i metoden måste man ev. ha flera return-satser
- En metod kan bara returnera ett värde!
    - Värdet kan dock var sammansatt t.ex. en array

Program innehåller normalt många metoddeklarationer

- I vilken ordning deklarationerna skrivs i programmet spelar ingen roll (de får inte vara nästlade)
- Vi lägger inledningsvis alla metoddeklarationer i slutet av programmet



Att exekvera metoden, att **anropa** den, görs genom att skriva metodens namn och aktuell indata inom en parentes.

- Indatan kallas **argument** (fast i JLS kallas de formella parametrar)
- Argumenten måste matcha parametrarna (antal, position, typkompatibla) annars kompileringsfel.
- Om argumenten är uttryck som behöver beräknas gör detta först (ev. implicita typomvandlingar görs också). Beräknings sker v->h.

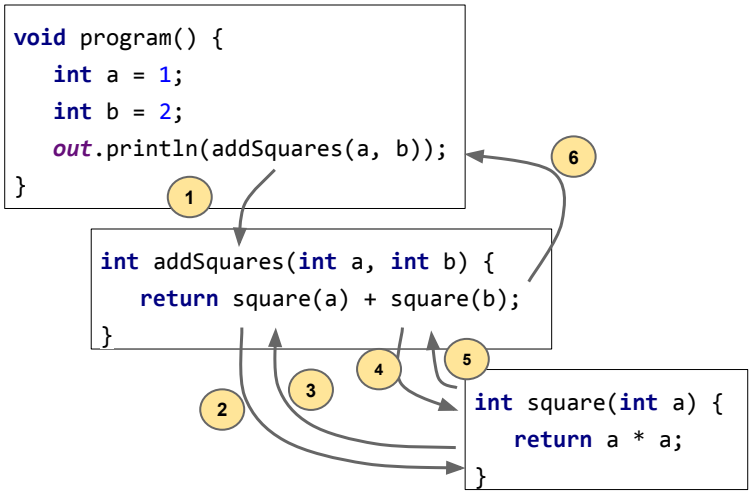
Följande sker vid anropet av metoden (detta skall ni kunna utantill!)

1. Variablerna i metoden (inkl. parametrar) skapas i en del av minnet kallat programmet **anropsstack (call stack).**
2. Argumentenvärden kopieras till metodens parametrar (utifrån position).
    - Kallas **värdeanrop (call by value).** Så sker alltid i Java
3. Därefter sker ett hopp, från aktuell (påbörjad) sats till den första satsen i metoden
4. Metoden exekveras sats för sats till en return-sats påträffas
5. Vid return-satsen kopieras returvärdet till en tillfällig lagringsplats, därefter frigörs minnet på anropsstacken, d.v.s. variablerna i metoden försvinner!!!
6. Programmet hoppar tillbaks till den påbörjade satsen (återhopp)
7. Om vi inte tilldelar returvärdet kommer det att försvinna då nästa

1. sats körs
2. I koden i bilden gör vi en tilldelning, vi sparar värdet. Typen på returnerat värde och variabeln måste vara kompatibla, annars typfel

OBS! Att metoder alltid jobbar med kopior av värden..

Metodanrop från Metod



En metod hoppar alltid tillbaks till satsen där den anropades

- Kan ske i flera steg, om en metod anropar en annan
- Anropsstacken används då till att hålla reda på i vilken metod programmet är och vart det skall hoppa då metoden är klar
- I IntelliJ kan man inspektera exakt vad som händer men anropsstacken (i debuggern)

Numeriska Parametrar och Returtyper

```
// Convert Fahrenheit to Celsius
double f2c(double fahrenheit){
   return (fahrenheit - 32) * 5 / 9;
}

// Absolute value (NOTE: 2 return statements)
int abs(int n) {
   if (n < 0) {
       return -n;
   }
   return n;
}
```

25

Används för olika typer av beräkningar.

Nästlade metodanrop

```
out.println(sqrt(pow(3, 2) + pow(4, 2)));
```


Det går att skicka returvärdet från en metod direkt som argument till en annan metod

- Kallas **nästlade anrop** (nested calls)
- Ibland användbart … slipper en del "onödiga" variabler för returvärden
  - Speciellt vid utskrifter
- … dock svårt att felsöka, allt sker i ett enda svep

Parametrar evalueras vänster till höger

- Men koda aldrig så att man blir beroende av detta

Boolesk Returtyp

```
// Boolean method (NOTE: No if statement needed)
boolean isGameOver(int score1, int score2){
    return score1 >= 10 || score2 >= 10
                        && score1 != score2;
}
```


Booleska metoder används för att svara på ja/nej frågor

- Ofta bara en rad med ett (komplext) boolesk uttryck
- Namnges som en ja/nej-fråga, hasWinner(), isEven(), isLeapYear()
- Användbara, höjer abstraktionsnivån, döljer rörig kod

Sträng som Returtyp

```
final Scanner sc = new Scanner(in);

// Get user name (NOTE: No variable used)
String getName() {
   out.print("Please enter your name > ");
   return sc.nextLine();  // Return result at once
}
```


Här i kombination med IO.

- Mer senare.

void-metoder

```
void roundMsg(int result, int computer, int statistic) {
   out.println("Computer choose: " + computer);
   if (result == draw) {
       out.println("A draw");
   } else if (result == humanWin) {
       out.println("You won");
   } else {
       out.println("Computer won");
   }
   out.println("Result " + statistic); // No return!
} // Jump back
```

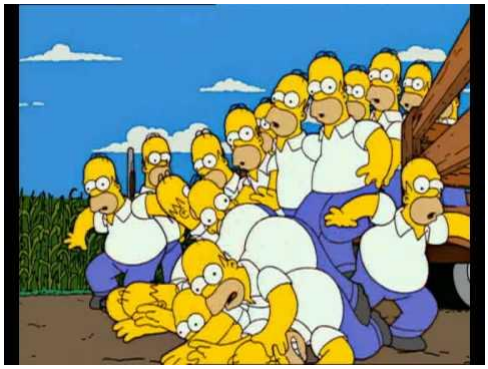
29

void-metoder

- Man kan ange att en metod inte returnerar ett resultat …
- … görs genom att ange **void** istället för returtyp
- Typiskt funktioner som "bara gör något", skriver ut till skärmen t.ex.
- I övrigt fungerar void-metoder som andra metoder
  - Återhopp sker då sista krullparentesen nås.
- Metoden får inte innehålla en return-sats med ett värde efter
  - Däremot bara return går bra, innebär att man avslutar metoden
    - Man kan alltså avsluta "mitt i" en metod
    - Gäller även för icke-void metoder , undvik (men ok vid vissa tillfällen)!
- Eftersom void-metoder inte returnerar något, representerar de inte något värde
  - … de är inte uttryck (de är satser)
  - Kan inte stå t.ex. vid tilldelning

Arrayer

Många av Samma



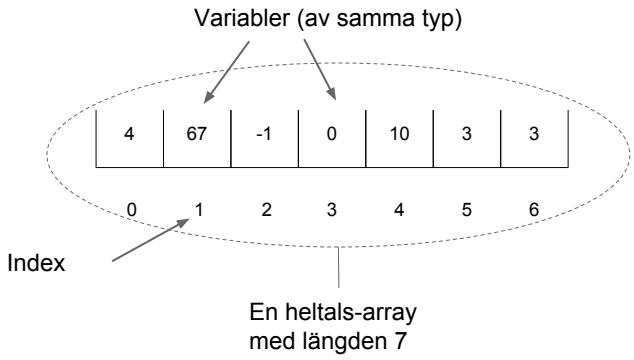
Literaler och variabler kan användas för enstaka värden.

- Men vad händer om vi behöver många värden …
- … 10 heltalsvariabler, eller 100, eller 100000 …. ?
- Orimligt att deklarera dessa en och en.
- Lösning: Om vi behöver <u>många variabler av samma typ</u> kan vi använda en array …

Finns tyvärr inget bra svenskt namn, “vektor” och “fält” förekommer.

- Vi använder det engelska namnet array.

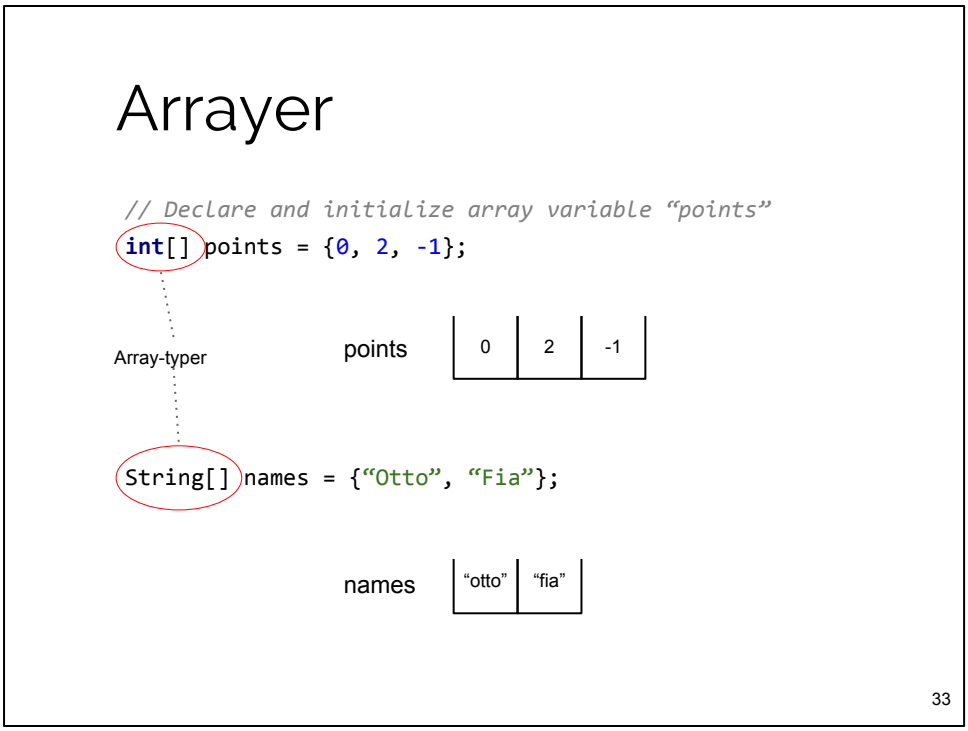
Konceptuell bild



32

En array är en konsekutiv (inga tomrum) följd av variabler.

- Antal variabler bestäms en gång för alla, förändras aldrig!
- Längden för arrayen är antalet variabler.
- Varje variabel har ett index.
- Index börjar på 0 och slutar på längden-1.



För att använda en array måste vi deklarera och initiera en variabel för den.

Deklaration görs som vanligt med: Typ, namn och ev. initiering

- Typ anges med typen för enskilda variabler + [ ] (man utgår från en "grundtyp")
  - Vi säger t.ex. int-array för typen int[]
- Namnet gäller för hela arrayen (de enskilda variablerna i array:en har inga namn).
- Man initierar genom att ange värdena för de enskilda variablerna i en lista då vi deklarerar arrayen.
  - Enskilda variablerna skapas och initieras med värden från listan i skriven ordning (från vänster till höger)
  - Array:en kommer att innehålla lika många variabler som värden vi angav

Mer om detta följer ...

Utskrift av Array:er

```
int[] points = { 1, 2, 3, 4, 5 };

// Will print like [I@330bedb4
out.println(points);

// Better, use "built in" helper
String s = Arrays.toString(points);
out.println(s);  // Will print [1, 2, 3, 4, 5]
```


Att direkt använda out.println() ger en (normalt) oanvändbar/obegriplig utskrift typ: [I@7f31245a

- Arrays.toString(), omvandlar arrayen till en sträng som sedan kan skrivas ut.

Indexering

```
int[] points = {0, 0, 0};

points[0] = 4;                  // { 4, 0, 0 }
points[2] = 1;                  // { 4, 0, 1 }
points[1] = points[2];          // { 4, 1, 1 }
points[0] == points[2];         // false
int i = 1;
points[0] > points[i];          // true

points[6] = 3;                  // Exception
points[-1] = 3;                 // Exception
```

35

Indexering innebär att komma åt de enskilda variablerna, **elementen**, i en array

- Indexering skrivs: *array-namn* [*index*]
  - [ ] (hakparenteser) kallas indexeringsoperatorn
- Index måste var ett heltalsuttryck (en variabel går bra).
- Vi måste själva se till att index inte hamnar utanför array:en, …
  - … om utanför så, undantag (exception) vid körning, ett **exekveringsfel**

Längden av en Array

```
int[] points = { 0, 5, 2, 0, 9 };

out.println(points.length);   // 5

// Last value!
out.println(points[points.length-1]); // 9
```


Man kan komma åt längden av en array på ett fördefinierat sätt (inbyggt i språket)

- Genom att skriva: *array-namn*.length

Traversering av Array:er

```
int[] arr = { 1, 2, 3, 4, 5, 6 };


for( int i = 0 ; i < arr.length ; i++){
   out.print(arr[i]);    //123456
}


for( int i = arr.length - 1 ; i >= 0 ; i--){
   out.print(arr[i]);   //654321
}
```

37

Traversering

- Ofta vill man **traversera** (genomlöpa) en array d.v.s. komma åt alla variabler i tur och ordning
  - Från vänster till höger eller tvärtom.
- Traversering görs vanligen med en for-sats (eftersom vi vet längden = antal varv)
- Som villkor använder man: i < array.length (vänster till höger)
  - D.v.s. i skall vara strikt mindre än längden eftersom sista index är ett mindre än längden
- eller i >= 0 (höger till vänster), större eller lika med eftersom första index är 0.

Inläsning till Array

```
String[] names = new String[3];

out.print("Input 3 names (enter after each) > ");
for (int i = 0; i < names.length; i++) {
   names[i] = sc.nextLine();
}
```


Måste använda en loop.

- Finns ingen genväg

Mer om Initiering

```
int[] points1;    // Declare arrays variables
int[] points2;

// Use variable and initialize array, must use new-operator
// because initialization not done at declaration
points1 = new int[]{2, 9, 0, 1, -4};

// No value list, must supply length (3), default values 0
points2 = new int[3];  // [0, 0, 0]
points2 = new int[points1.length]; // Or same length

// Same for strings
String[] names;
names = new String[]{“Otto”, “Fia”,“Pelle”, “Siv”};
```

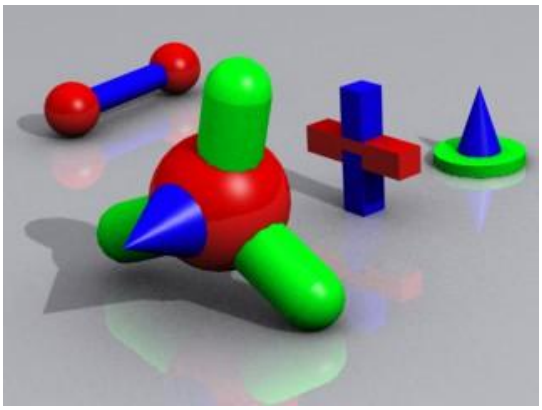

Man kan skapa och initiera arrayer på flera platser i programmet (inte bara vid variabeldeklarationen)

För att skapa och initiera en array på någon annan plats än vid deklarationen

- Använd operatorn **new** + array-typen + [ ] + en lista med värden.
- Om ingen lista med värden anges måste man ange ett värde för arrayens längd inom hakparenteserna
    - Variablerna får då förbestämda värden beroende på typ, int ger t.ex. 0.

Klasser

Flera men Olika



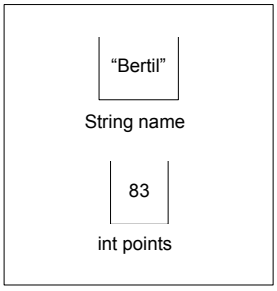
41

Arrayer används när vi behöver många variabler av samma typ.

Ibland behöver vi flera variabler men ev. av olika typ (vissa kan vara av samma typ)

- Om vi t.ex. vill beskriva en spelare i ett spelprogram så kanske vi behöver en variabel för spelarnamn och en för poäng
  - Krångligt t.ex. vid metodanrop att behöva skicka många variabler
- Istället för att ha enskilda variabler kan vi skapa ett **objekt** med flera variabler
  - Vi kan då skicka objektet, d.v.s. “alla” variablerna, på en gång till en metod.
- Dessutom hör ju de enskilda variablerna konceptuellt ihop, de används för att beskriva en spelare
  - Programmet blir lättare att förstå

Konceptuell bild



Objekt med två variabler av olika typ

42

Ett objekt är en "förpackning" av variabler som hör ihop (mer senare)

- I bilden har vi samlat flera variabler som hör ihop med en spelare.

Vi kan inte skapa objekt direkt (förutom vissa specialfall, en sträng är t.ex. ett objekt)

- För att skapa objekt behöver vi en **klass**

OBS Bilden! Att värdet "Bertil" egentligen inte ligger i variabeln (det är en referensvariabel, se Referenser).

Klassdeklaration

```
// Very basic class
class Player {
   String name;  // Instance variables
   int points;
}
```

43

För att skapa en klass i Java gör man en klassdeklaration

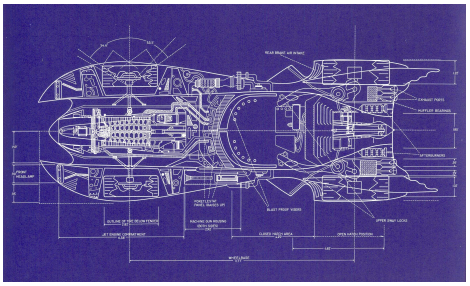
- Vi kan lägga klassdeklarationer var som helst i programmet.
- Vi lägger dem vanligen mot slutet i programmet (efter metoderna, inte i någon metod).
    - Eller i egna filer, se senare.

Klassdeklaration

- Inleds med det reserverade ordet **class** + namnet på klassen.
    - Namn inleds med stor bokstav (CamelCase vid behov)
    - Namnet brukar vara ett substantiv och skall förklara vad objekten skall representera.
- Därefter ett block
    - I blocket deklarerar vi de variabler vi vill skall finnas i objekten, dessa kallas **instansvariabler** (alt. **attribut**)
    - I vilken ordning instansvariabler deklareras spelar ingen roll (lite förenklat).

Klass och Instans

Klass



Objekt (Instanser)

44

Vi kan skapa ett godtyckligt (ändligt) antal objekt utifrån en och samma klass

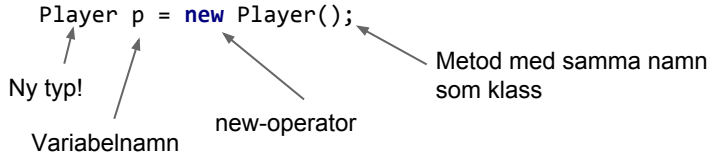
- Man kan se en klass som en ritning
- Vi säger att ett objekt är en **instans** av en klass
- Alla instanser får sin egen uppsättning av variabler
    - Vi kanske behöver två spelarobjekt, ett för “Pelle” och ett för “Fia”, båda innehåller egna variabler name och points.

Analogier:

- Pannkaksrecept (klass) och pannkaka (objekt)
- Pepparkaksform (klass) och pepparkaka (objekt)

Variabler för Objekt

```
// Declare, instantiate and initialize
Player p = new Player();
```




```
Player p1;   // Declare ...
// ... later instantiate and assign
p1 = new Player();
```

45

En klass introducerar en ny referenstyp

- Typsystemet är utbyggbart för klasser, se Typer
- Om vi skapar en ny klass skapar vi en ny typ! Vi kan vi deklarera variabler av denna typ!
- När vi deklarerat variabeln kan vi initiera den
  - För att initiera variabeln måste vi instansiera (skapa) ett objekt.
  - Instansieringen sker genom att använda **new**-operatorn tillsammans med en metod som heter som klassen.
  - Vi kan inte göra som med arrayer t.ex. { “olle”, 87 } eller liknande går inte.

Analys av kod (vänster till höger)

- Player är den nya typen som klassen introducerat
- p är namnet på variabeln
- new skapar ett objekt i minnet (inklusive de variabler som finns i objektet)
- Därefter körs metoden Player() för objektet, en sådan metod finns alltid men syns inte i koden, mer senare.
- Därefter initieras variabeln p med objektet (mer senare).

Punktnotation

```
// Declare instantiate and initialize
Player p1 = new Player();

// Dot notation to access variables in object
p1.name = "pelle";
p1.points = 2;

out.println( p1.name );  // “pelle”
out.println( p1.points ); // 2
```


Ett namn på en array-variabel betecknade hela arrayen (lite förenklat, mer senare)

- För att komma åt enskilda variabler använde vi indexering

Ett namn på en objektvariabel (p1 i bilden) betecknar hela objektet (lite förenklat, mer senare)

- För att komma åt de enskilda variablerna används punktnotation d.v.s. objektvariabelnamn (p1), punkt, instansvariabelnamn (name)

Arrayer med Objekt

```
// Declare and initialize array AND instantiate
// objects (each new .. creates an object)
Player[] players = { new Player(), new Player() };

// First indexing then dot notation
players[0].name = "pelle";
players[0].points = 23;
players[1].name = "fia";
players[1].points = 45;

out.println( players[0].name );  // “pelle”
out.println( players[1].points ); // 45
```

47

Om vi har en typ kan vi skapa arrayer utifrån typen

- Så också med klasstyper!
- För att komma åt enskilda variabler används …
- … först indexering eftersom vi började med en array (ger ett helt objekt) …
- … därefter punktnotation för att komma åt variabeln i objektet.

OBS! Bara Player[] players; skapar inte några player objekt, … bara array-objektet!

Mer om Metoder

Array som Parameter

```
// Find max value in array
int max(int[] arr) {
   int m = arr[0];
   for (int i = 1; i < arr.length; i++) {
       if (arr[i] > m) {
           m = arr[i];
       }
   }
   return m;
}
```

Objekt som Parameter

```
// Print complete dog
void printDog(Dog dog) {
   out.print("Name: " + dog.name);
   out.println(" Age:" + dog.age);
}

// Class declaration
class Dog {
   String name;
   int age;
}
```

Array med Objekt som Parameter

```
Dog findOldest(Dog[] dogs){
   int index = 0;
   int maxAge = dogs[index].age;
   for( int i = 0 ; i < dogs.length ; i++){
       if( dogs[i].age > maxAge){
           index = i;
           maxAge = dogs[i].age;
       }
   }
   return dogs[index];
}
```


Kan skicka komplexa argument

- Här en parameter för en array med Dog-objekt

Array med Objekt som Returtyp

```
Dog[] getDogs(){
    Dog[] dogs = { new Dog(), new Dog() };
    dog[0].name = sc.nextLine();
    dog[1].name = sc.nextLine();
    return dogs;
}
```


Komplex returtyp

- Här en array med Dog-objekt som returvärde.

Arbetssätt

Arbetssätt

54

Ett genomtänkt arbetssätt skall hjälpa er att klara kursens laborationer (...och följande kursers)

- Genom att arbeta på ett visst sätt ökar vi vår förmåga att lösa problem.
- Generellt: Behärska komplexitet.

Minsta Steget

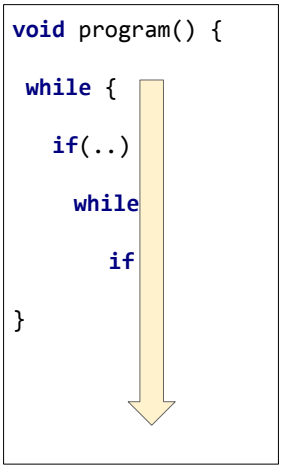
```
void program() {
   // Very small basic step
   out.println("Program started");

}
```

55

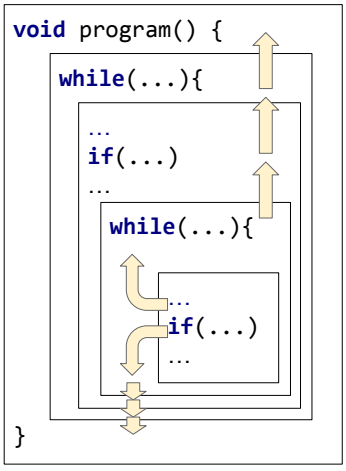
När vi skriver ett program (eller håller på med en del av) kan man inte skriva allt på en gång!

- Man börjar med att försöka hitta ett minsta steg som för oss närmare lösningen
- Kodar detta och kontrollerar om det fungerar … d.v.s. kör (eller testar) programmet.
- När det fungerar söker vi nästa minsta steg på samma sätt
- Vi ser till att alltid ha en körbar kärna som vi bygger på steg för steg.
- Bortse inledningsvis från alla specialfall, skriv “normalkoden” först
    - … MEN, notera specialfallen, … ofta där felen (buggarna) finns.

Inifrån Ut



Uppifrån och ner

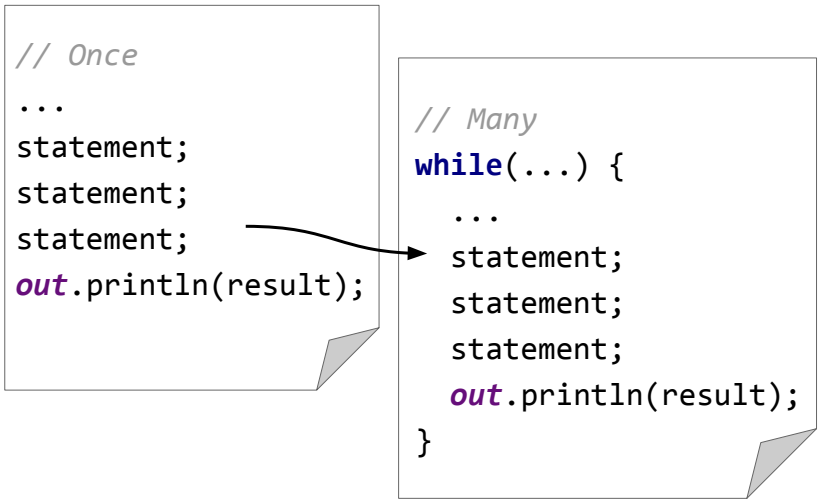


Inifrån ut

När man jobbar med minsta steget tekniken arbetar man ofta “inifrån-ut”

- Inifrån-ut innebär att man inte skriver programmet “rad för rad” uppifrån och ned.
- Istället börjar man “inifrån”
- När man fått till något minst steg utökar man vid behov programmet genom att lägga till ev. if/while etc. “runt”
- När detta fungerar utökar man med ev. nästa “lager” o.s.v.

En Gång, flera Gånger



När man använder minsta steget och inifrån-ut:

- Om man kan göra något en gång, … är det lätt att göra det flera gånger … lägg en loop runt!
- Börja med att göra något en gång först!

Hårdkodad data

```
// Hard coded (for now)
players = new Player[2];
Player p1 = new Player();
p1.name = "Olle";
Player p2 = new Player();
p2.name = "Fia";
players[0] = p1;
players[1] = p2;

// Use hard coded
...
...
```

```
// Later ... a method
players = getPlayers();


Player[] getPlayers() {

   // Read in player data

   return players;
}
```

58

Vi vissa steg kan det behövas data som man inte har än. Om så:

- Hårdkoda data, d.v.s. skriv dit något tillfälligt (något enkelt). Bara så att vi kan fortsätta med nästa steg.
- Senare ersätts det hårdkodade med inläst och/eller beräknad data.

Kommentera och *//TODO*

```
/*coin = new ImageIcon(ImageIO
       .read(new
File("src/exercises/optional/gold_coin_single.png")))
       .getImage();*/
coin = new ImageIcon(this.getClass() // TODO better use Image?

       .getResource("gold_coin_single.png"))
       .getImage();
// getAudioInputStream() also accepts a File or InputStream
//AudioInputStream ais = AudioSystem.
// getAudioInputStream(new
File("src/exercises/optional/atari.wav"));
AudioInputStream ais = AudioSystem
     .getAudioInputStream(this.getClass()
     .getResourceAsStream("atari.wav"));
```

59

Under utvecklingen undviker man att ta bort kod, .. kanske koden inte var så dum ändå …!

- Använd kommentarer istället för att ta bort kod!
- Lätt att få tillbaka den gamla koden om den visade sig vara bra!
- När programmet är färdig: STÄDA UPP! Ta bort kommenterade stycken, indentera, fixa bättre namn,,...

Ofta stöter man på saker som man borde fixa till (när man egentligen håller på med en annan sak).

- Ta för vana att lägga in TODO noteringar (sökbara i IntelliJ)

I bilden

- Gammal kod bortkommenterad (tills vi är säkra på att den nya är bättre, annars, kommentera ut den nya och avkommentera den gamla)
- TODO-notering för att senare kolla om Image är bättre än ImageIcon