Grunderna

TDA548/Joachim von Hacht

Språket Java

Språket Java beskrivs i en specifikation

- Vi använder version 8 av språket, kallas även version 1.8
- Specifikationen beskriver hur vi skall skriva språket Java (syntax) och …
- … betydelsen av det vi skriver (semantik).

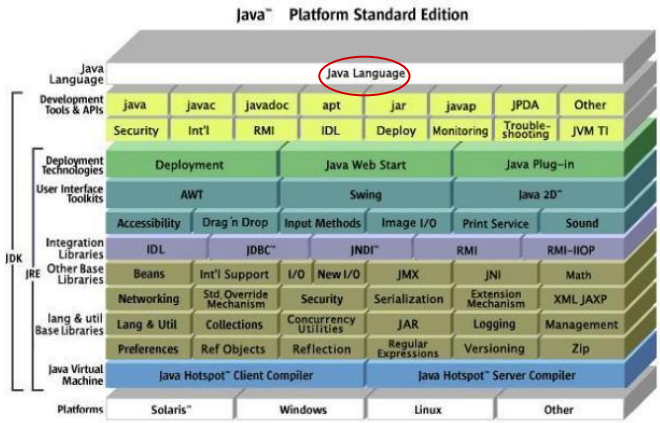
OBS! Att det vi går igenom i kursen gäller för många (de flesta) imperativa och/eller objektorienterade språken.

- Kursen är alltså inte en Java kurs.

För att kunna utveckla program i Java på din dator måste du installera Java Development Kit, JDK 1.8 (Java Runtime Environment, JRE räcker inte)

- Popularitet för olika språk

Java-Plattformen

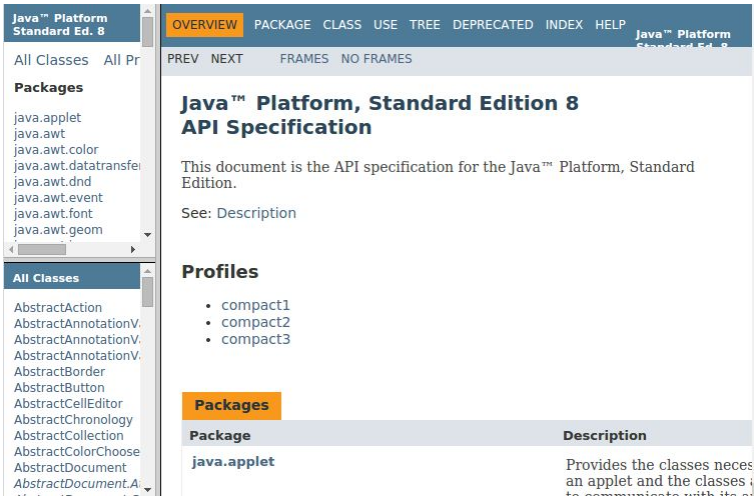


3

Java är en **plattform**

- Dels finns själva språket Java som är en ganska liten del
- Utöver detta finns bl.a en mängd **standardbiblioteket (libraries)**
  - Biblioteken kallas ofta för API:n (**Application Programming Interface**, programmeringsgränssnitt)
  - Ett API innehåller körbar kod som vi kan använda i vårt program (så slipper vi skriva samma sak om och om igen)
    - I vår kod betyder det att vi använder färdiga objekt, mer kommer ...
  - Standardbiblioteket är organiserat i olika grupper utefter ändamål
  - För att använda standard API:er skriver man t.ex. i programmet : import java.utils.List …. mer kommer ...
- Java-program körs i en speciell exekveringsmiljö, en “virtuell” dator, mer senare ...
- Java-plattformen = språket + standardbiblioteken + exekveringsmiljön

Java-Dokumentation



4

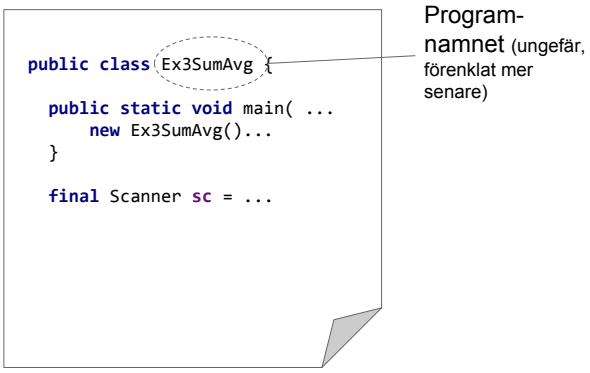
En svårighet med moderna språk är att de innehåller enorma standardbibliotek

- Förutom själva språket måste man lära sig hitta bland allt färdigt.
- I denna kurs används bara en mycket liten del … (inga större problem, ABSOLUT inget man behöver lära sig utantill)

Modern programutveckling innebär bl.a.

- Att man söker igenom Web:en efter bibliotek (standard eller andra s.k. tredjepartsbibliotek) med lämplig kod för uppgiften.
- Att man därefter sammanfogar den funna koden med “så lite som möjligt” av egen kod.
    - Mer om detta i senare kurser
    - I denna kurs skriver vi mycket själva för att lära oss.

Program och Fil



Ex3SumAvg.java
(en textfil)

5

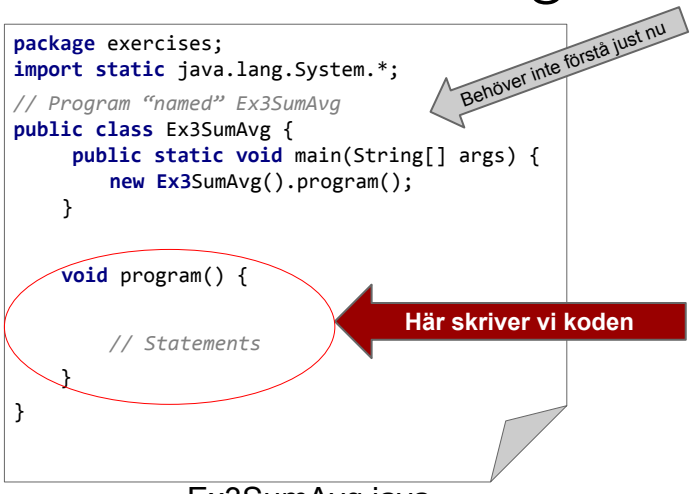
Ett Java-program skrivs och sparas i en textfil

- En textfil är fil uppbyggd av rader, med tecken läsliga för människor, Se vidare bildserie Undantag och Filhantering
- Filen kallas källkodsfil (eller **källkoden, source code**)
- Java kräver teckenkodningen UTF-8 för källkodsfiler.
- Källkodsfiler använder suffixet .java, t.ex. SumAvg.java
    - Om programmet (filen) har ett sammansatt namn skrivs det CamelCase

Varje program (koden för programmet) finns i en enda fil (tills vidare…)

- I bilden: public class SumAvg, anger ungefär att programmet "heter" SumAvg, det ligger i filen SumAvg.java

En Mall för Java Program



Ex3SumAvg.java

6

För att förenkla för nybörjare använder vi en "mall" för våra program.

- OBS! All källkod skrivs på engelska.

Analys av mallen, se bilden

- Filen med programmet måste ligga i en speciell mapp, anges med **package** överst i filen. Filer kan alltså inte flyttas hur som helst!
  - Var programfilen finns i filsystemet och package måste stämma!
  - package exercises betyder mappen src/exercises i IntelliJ projektet.

- Vid **import** anger vi vilka färdiga Java-APIer programmet behöver.
  - Ibland får vi lägga till någon rad här, utifrån behov.
  - I mallen anger vi att vi behöver allt (= asterisken) från API:et "java.lang.System" (bl. a. resurser för att hantera skärm och tangentbord)

- I programkoden kan man lägga in **kommentarer**.
  - Inleds med `//`, gäller en rad
  - eller omsluts av `/*` … `*/` för flera rader.
  - Kommentarer har ingen påverkan på programmet..
    - Kommentarer är till för människor, skall underlätta

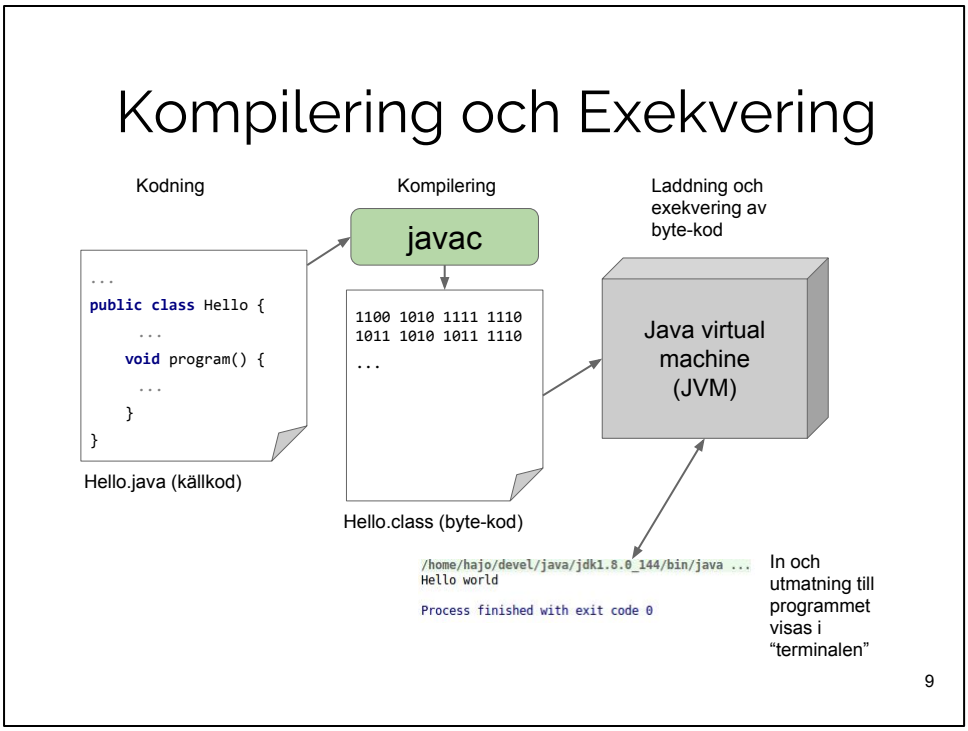
        - förståelsen.
        - Kommentarer skrivs också på engelska.
        - IntelliJ visar kommentarer med grå kursiv stil.

- De matchande krullparenteserna {...}, kallas ett **block**
    - Ett block är en avgränsad del av programmet.
    - Block kan ligga inuti block, kallas **nästlade** block.
    - Vänster marginal är indragen för att visa den nästlade strukturen på blocken, kallas **indentering**
        - Indentering är mycket viktigt för förståelsen av ett program, vi använder alltid!
        - IntelliJ kan hjälpa till med detta (kallas även formatering, se instruktion i övningar)

- public static void main(...) och blocket direkt efter måste stå där!
    - Exakt vad det betyder återkommer vi till, tills vidare får vi acceptera detta utan förklaring.
    - Raden med new Ex3SumAvg().program(), betyder ungefär "skapa och starta" programmet.

- Vid void program() börjar vårt program
    - void bekymrar vi oss inte för just nu, återkommer…
    - I blocket efter program() skriver vi **satser (statements)** mer strax …
    - När vi når sista krullparentesen i blocket är programmet slut

- Lite grundläggande syntax och semantik för språket Java:
    - Skillnad på liten/stor bokstav (string och String tolkas som två olika saker)
    - Ett eller flera "vita" tecken (blanksteg, nyrad, etc.) spelar ingen roll (räknas som ett tecken).
    - Tomma rader spelar ingen roll.
    - Parenteser skall alltid matcha (...), {....}, [...], <...>, {{ … }}, … och vara korrekt nästlade.
    - Vissa ord är reserverade för språket Java, reserverade ord **(keywords).**
        - "import" är ett reserverat ord, inget vi namnger får heta "import", ett program får inte heta import.
        - IntelliJ visar reserverade ord i mörkblått med fet stil.
    - Bryter vi mot syntaxreglerna för vi ett syntaxfel (syntax error)
        - Visas med röd understrykning i IntelliJ (peka på markering så kanske något tips visas)



Kompilering

- Innan ett Java-program kan exekveras (köras) måste det **kompileras**
- Ett program kallat javac (= java compiler, en kompilator) översätter källkoden till en binärfil med byte-kod (byte)
- Byte-koden sparas i en ny fil, en class-fil som heter samma som källkoden men med suffixet .class istf .java.
  - Vi har efter kompileringen två filer
- Det är class-filen som kommer att exekveras
  - Har vi källkoden kan vi alltid skapa en ny class-fil, det är källkoden som är viktig!
- Om programmet av någon anledning inte går att kompilera får man ett **kompileringsfel**
  - Ett program med kompileringsfel går inte att köra (eftersom det inte går att kompilera).
  - Syntaxfel leder som sagt till kompileringsfel.

Ett kompilerat program (byte-koden) körs på en virtuell dator (Java Virtual Machine, JVM).

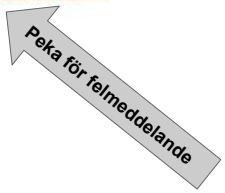
- Detta gör att Java-program utan någon modifikation kan köras på flera olika operativsystem (eftersom JVM:en eliminerar skillnader mellan olika operativsystem
  - Man säger att Java är **plattformsoberoende** (många andra

- språk måste kompileras om för varje operativsystem)
- Dock måste din dator ha en JRE för att det skall fungera.

OBS! Kompileringen sköts automatiskt av IntelliJ (i bakgrunden) så fort källkoden ändras.

Kompileringsfel

```
// Syntax error

void program() {
out.println(Hello world");
}
```



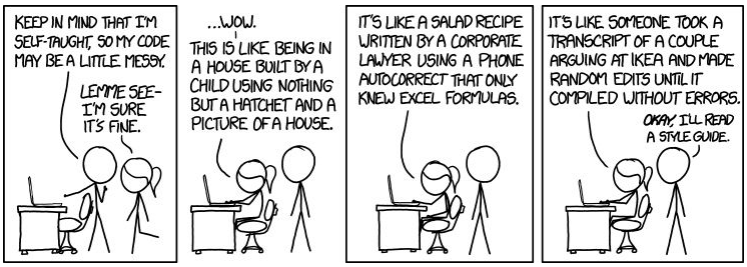
8

Om programmet vi skrivit är felaktigt kommer kompilatorn inte att kompilera det.
Istället visas röda felmarkeringar och ett felmeddelande (om man pekar på markeringen).

Ett fel kan generera följdfel!

- Rätta upp/vänster till ner/höger

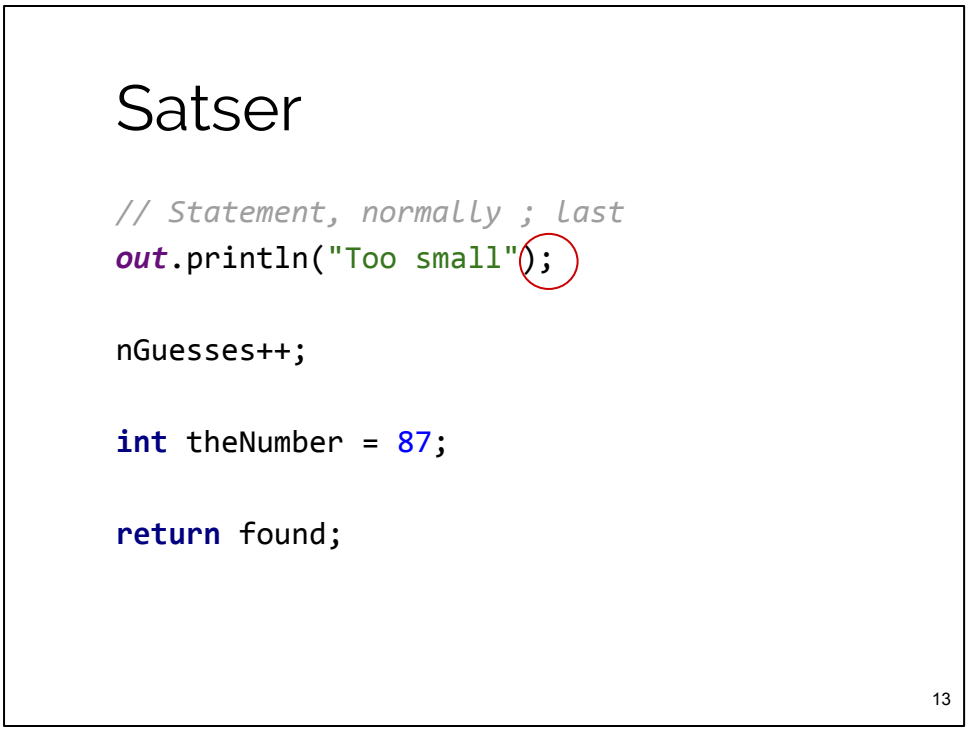
Kodstil



9

Java-program skall skrivas med en viss kodstil (t.ex. var krullparenteser skall stå. liten/stor bokstav, m.m.)

- Stil är viktigt! Rörig kod är svår att felsöka!
- Vi följer i princip Googles stil (… för mycket detaljer för denna kurs …)
    - Använd samma stil som exempelkoden!
    - Att alla använder samma stil handlar om kommunikation, samma stil underlättar kommunikation mellan programmerare.



Imperativ programmering innebär att, i kod, steg för steg beskriva för datorn hur en uppgift skall utföras.

- Ett steg kan vara att flytta ett värde i från en plats i minnet till en annan.
- Stegen är alltså väldigt "små" och oftast väldigt många.

Ett "steg" i ett Java-program kallas för en **sats** (statements).

- Ett Java program körs sats för sats tills det inte finns fler satser att exekvera.

En sats i Java

- Är den minsta fristående enheten (atomen).
- Är ett **imperativ,** en uppmaning till datorn att göra något (därav imperativ programmering)
- Måste avslutans med semikolon, ";" visar var satsen är slut (några undantag senare).
  - Jämför: en mening på svenska, avslutas med punkt (eller ?, !)
  - En sats kan sträckas sig över flera rader (ofta skriver vi dock en sats per rad)!
- Den tomma satsen skrivs ";" (inget utförs)
- Ordningen på satserna är mycket viktig!
  - Datorn exekverar satserna i den ordning vi skrivit dem (vanlig

    - läsordning vänster höger, uppifrån och ner)! Datorn gör som vi skriver (… inte som vi vill…)

Satser byggs i sin tur upp av

- literaler, variabler, operatorer, metodanrop, …
- Dessa kan inte köras fristående, de måste ingå i en sats.

Literaler

```
// Integer literal
143567
// Real literal (floating point literal)
25.345
// Boolean literal
true
// Character literal
'Z'
// String literal
"Hello world!"
```

15

Literaler (literals) är ett sätt att skriva (representera) fixa värden i koden (hårdkodade värden, värden som aldrig kan ändras)

- En heltalsliteral består (oftast enbart) av siffror (ev. tecken, m.m.)
- En flyttalsliteral (reellt tal) har en decimalpunkt i övrigt siffror (ev. tecken, exponent, m.m.)
- En teckenliteral består av ett enda tecken med enkla citat runt
  - Ett blanksteg skrivs: ‘ ‘ (ett blanksteg mellan enkla citationstecken)
- En strängliteral (**sträng** säger man vanligen) är en följd av tecken (0-n) som omges av dubbla citationstecken.
  - Kan innehålla blanka (osynliga) tecken såsom blanksteg..
  - Tomma strängen skrivs : “” (följd av 0 tecken, tomt inom citatet)
- En boolesk literal betecknar ett sanningsvärde och skrivs som **true** eller **false**
  - true och false är reserverade ord
- IntelliJ visar numeriska och booleska literaler i blått, tecken och strängliteraler i grönt

OBS! Att t.ex. heltalsvärden skulle kunna representeras med romerska siffror eller annat … (men såklart är den naturligaste representationen att använda vanliga siffror).

- Ordet “ **representera**” är vanligt, försök få grepp om det ….

Literaler och Typer

```
143567          // Type is: int
25.345          // double
true            // boolean
'Z'             // char
"Hello world!" // Type is: String
```

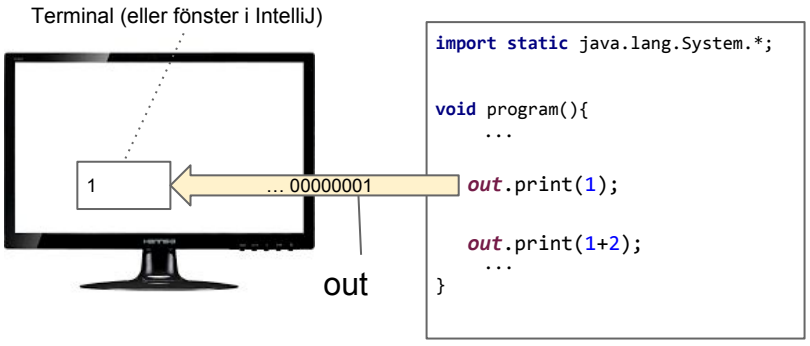
12

Alla värden i Java har en typ (tillhör en viss mängd). Literaler representerar värden och klassificeras automatiskt som tillhörande en viss typ. De typer vi behöver just nu.

- int, (integer), typen för heltal
- double, typen för reella tal, kallas också flyttal (floating point numbers)
- boolean, typen för sanningsvärden (finns bara true och false)
- char (character), typen för enstaka tecken.
- String, typen för följder av 0-n tecken.

- Alla utom String kallas för primitiva typer, String är en referenstyp, mer senare..
- Se vidare bildserie om Typer

Utmatningssatser



18

Utmatning (output)

- Java program har automatiskt tillgång till en byte-ström, men namnet **out**.
  - Byte-ström är en kanal för att skicka bytes.
- out är i vårt fall kopplad till datorns skärm.
- Genom att använda namnet "out" i koden får programmet tillgång till strömmen
- För att skriva ut något använder vi en utmatningssats t.ex.: out.print(...);
  - Betyder att värdet i parentesen skall skickas till strömmen för att slutligen hamna på skärmen
    - Om värdet måste beräknas så sker detta först, innan det skickas till strömmen.
  - Vill vi ha en **nyrad** efter utskriften skriver vi: out.println(…), ln = new line
- Vi kommer att se utskrifterna i ett fönster i IntelliJ (= terminalen).
- Vi måste skriva import static java.lang.System.* överst för att kunna använda out.

Anm: En hel del kodexempel visar System.out.println(...) … för jobbigt att skriva, vårt sätt är mer ekonomiskt.

Deklarationer

a är 15. Dividera a med b! Skriv ut resultatet!

a är 15, b är en banan. Dividera a med b! Skriv ut resultatet!

a är heltalet 15, b är heltalet 5. Dividera a med b! Skriv ut resultatet!

```
// Variable declarations in Java
int a;
int b;
```

14

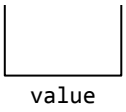
En deklaration innebär att vi förklarar vad ett namn står för.

- I Java måste vi t.ex. deklarera variabler, metoder och klasser

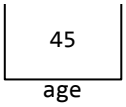
I bilden: På samma sätt som vi måste veta vad saker står för så måste datorn veta det.
En deklaration berättar vad något är.

Variabler

```
// Variable declaration
int value;
```




```
// Declaration and initialization
int age = 45;
```




```
// More declarations and
// initializations
double d = 25.345;
char ch = '?';
boolean b = false;
String s = "Hello";
```

20

En variabel i Java (i imperativa språk) är en ändringsbar behållare för en viss typ av värden

- Vi ritar variabler som öppna lådor med namnet under och värdet i.

En variabel måste deklareras innan den kan användas.

- Vid deklaration anges typ och namn
  - Typen anger vilken typ av värden som kan lagras i variabeln t. ex. int eller double.
  - Namn kallas **identifierare**
    - Vi kan själva välja namn utifrån vissa regler
      - I princip får vi använda bokstäver, siffror och "_" .
      - Får inte inledas med siffra, inte innehålla osynliga (vita) tecken.
    - Namngivning av variabler är svårt, namnet skall
      - Inledas med liten bokstav
      - Sammansatta namn skrivs camelCase
      - Namnet skall förklara syftet med variabeln, nPlayers, maxScore, width, … (skilj på singularis och pluralis, d.v.s. "s" på slutet)
      - Var lagom långt (4-10) tecken ungefär
      - Undvik korta namn som x, y …
        - … dock: Variabler som används i ett litet

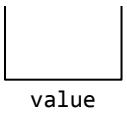
        - avsnitt programmet kan ha korta namn, typiskt i, j m, n (i loopar, mer senare …)
    - Variabeldeklarationer är satser (avslutas med “;”)
- OBS! En variabeldeklaration ger en variabel

När vi senare i programmet vill använda variabeln skriver vi bara namnet
- Vi behöver inte ange typ igen, programmet vet från deklarationen.

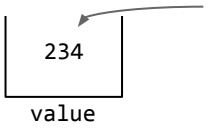
Initiering av variabler
- En variabel måste ofta ges ett värde innan den kan användas.
- Kan göras med en initiering i samband med deklarationen (eller senare).
- Initieringsvärdet måste “stämma” (vara kompatibelt) med variabelns typ! Se bildserie Typer

Tilldelning

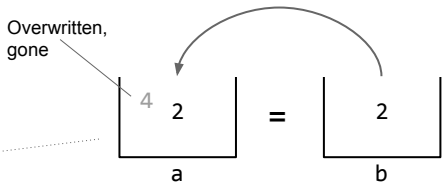
```
// Declaration
int value;
```




```
// Assignment
value = 234;
```




```
int a = 4;
int b = 2;
a = b;
```



22

Tilldelning (assignment)

- Skrivs m.h.a. **tilldelningsoperatorn** "=" (likhetstecken)
  - Helt annan betydelse än i matematik
- Innebär att ett värdet från höger sida om operatorn kopieras till en variabel på vänster sida.
  - På vänstersidan om operatorn skall det alltid stå något som betecknar en variabel
    - Variabeln dit värdet skall kopieras (oftast ett variabelnamn)
  - Om det står ett uttryck på höger sida beräknas detta först
  - Typen på uttrycket till höger och variabeln måste vara kompatibla, annars typfel
    - Se bildserie om typer.
  - Efter tilldelningen är det "gamla" värdet på vänstersidan borta (överskrivet).

Att använda ett variabelnamn vid en tilldelningssats innebär två olika saker.

- Om namnet står till vänster om =-operatorn, betyder det att värdet skall läggas i variabeln namnet syftar på
- Om namnet står till höger betyder det värdet som finns i variabeln skall avläsas..
  - Exempel: `x = x + 1; // Läs x på höger sida, öka med 1, stoppa`

- tillbaks i samma x (vänster sida)

Operatorer

Prioritet			Associativitet

1	`[]`
	`()`
	`.`
		array index
		method call (anrop)
		member access
			Left -> Right

2	`++`
	`--`
	`+ -`
		pre or postfix increment
		pre or postfix decrement
		unary plus, minus
			Right -> Left

3	`(type)` `new`	type cast (typomvandling) object creation	Right -> Left

4	`*`
	`/`
	`%`
		multiplication
		division
		modulus (remainder)
			Left -> Right

5	`+ -` `+`	addition, subtraction string concatenation	Left -> Right

7	`< <=`
	`> >=`
	`instanceof`
		less than, less than equal
		greater than, greater than equal
		reference test
			Left -> Right

8	`==` `!=`	equal to not equal to	Left -> Right
12	`&&`	logical AND	Left -> Right
13	`||`	logical OR	Left -> Right
14	`? :`	conditional (ternary)	Right -> Left
15	`=` `+= -= *= /= %=`	assignment (tilldelning) compound assignment	Right -> Left

24

Lista med de flesta operatorer i Java (flera är samma som i vanlig matematik)

- **Prioritet** (precedence) visar vilken operator som skall göras förs (om flera olika)
- Associativitet: Om flera operatorer med samma prioritet? Vilken görs först? Mestadels: börja från vänster
- **Unär operator** tar en operand
- **Binär operator** tar två
- Operatorer ++, ++, +, -, *,/, %, , <,<=, >, >= kräver numeriska typer (int, double,...) för operanderna
  - +, -, *,/ är de vanliga aritmetiska operatorerna, % är modulo.
  - Unärt "-" ger ett negativt värde
  - OBS! /, division utförs som heltalsdivision om båda operander av heltalstyp
  - Vid / se upp med vad som hamnar ovanför och under!
  - <,<=, >, >= är jämförelseoperatorer (relational), större än o.s.v., returnerar booleska värden
- Operatorerna == och != används för likhet respektive olikhet kan ta numeriska, booleska eller referenstyper (mer senare) som operander
  - OBS! == (två likhetstecken för likhet, vanligt nybörjarfel att använda ett!)
- &&, || och ! används för logiskt och, logiskt eller samt logiskt icke,

- kräver booleska operander

Finns en del speciella saker i samband med beräkningar
- NaN, Not a Number
- +/- Infinity
- … m.m. kan dyka upp men inget vi behöver sätta oss in i

Division med 0 är som vanligt inte tillåtet, kommer att ge
- Ett undantag (programmet kraschar) för heltal
- Värdet Infinity för flyttal

Sammansatta Tilldelningsoperatorer

```
x += 1;        // x = x + 1;
x -= 2         // x = x - 2;
x *= 3;        // x = 3 * x;
x /= 2;        // x = x / 2;
x %= 2;        // x = x % 2;
```

18

Operatorerna `+=`, `-=`, `*=`, `/=` är förkortningar.

- Utför en operation och tilldelar resultatet (till samma variabel)
- Använd om du vill …. kan dyka upp i exempelkod

Uttryck

```
123

value < 100

(count + 1) / max

// A statement to print value
// of an expression
out.println(constraint == true);
```

19

Ett uttryck:

- Byggs upp av literaler, variabler, operatorer, m. m.
- Representerar ett värden (står för ett värde) *)
- **Evalueras** (beräknas) och får ett slutlig värde under körningen.
  - Hur beräkningen sker beror på prioritet och associativitet.
- Har ingen speciell slutmarkering
- Måste ingå i en sats (kan inte exekveras fristående)
- Alla värden måste ha en typ d.v.s. ett uttryck har en typ!

*) Finns mer att säga om detta i Java, men det gör vi inte.

Lat Evaluering

```
int i = 4;

1 < 2 || i != 4;

1 > 2 && i == 4;
```

Behöver inte evalueras

Vid beräkning av uttryck med operatorerna || och && används lat evakuering

- Om vänster operand (uttryck) är sant, evalueras inte högersidan (eftersom hela uttrycket är sant om någon av operanderna är sanna)
- Om vänster operand (uttryck) är falskt evalueras inte högersidan (eftersom hela uttrycket är falskt om någon av operanderna är falska)

Överlagrad +-operator

```
12 + 4 -> 16

12.0 + 4.0 -> 16.0

'0' + '0' -> 96

"123" + "4" -> "1234"
```

21

+-operatorn fungerar olika beroende på operandernas typ

- För numeriska typer t.ex. sker vanlig addition
- Om typen är char används teckenkoden i additionen (char räknas som en numerisk typ)
  - Kod för t. ex. tecknet ‘0’ är 48.
- För strängar sker sammanslagning, **konkatenering**.
- Att operatorn beter sig olika utifrån operandernas typer kallas att den är **överlagrad**
  - Finns bara en sådan operator i Java

Likhet för Flyttal

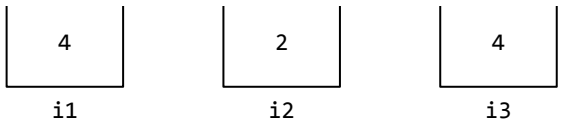
```
double d1 = 1.0;
double d2 = d1 - 0.6 - 0.4;
double d3 = d1 - 0.9 - 0.1;

out.println(d1 == d2);   // True!
out.println(d1 == d3);   // False!
```


Flyttal är närmevärden

- Skall aldrig jämföras med ==
- Man får använda lite matematik eller färdiga metoder i Java (t.ex. Double.compareTo())
- Mer senare...

Likhet för Variabler

```
int i1 = 4;
int i2 = 2;
int i3 = 4;
out.println(i1 == i2);   // False
out.println(i1 == i3);   // True
```



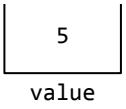
23

Likhet för variabler innebär att innehållet i respektive variabel jämförs

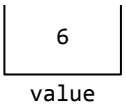
- Använder == -operatorn
- Så är det alltid, alltid innehållet i variabeln!
- Om samma innehåll så ger uttrycket värdet true annars false

In- och Dekrementering

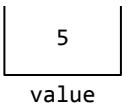
```
int value = 5;
```




```
// Incrementation
value++:
```




```
// Decrementation
value--;
```

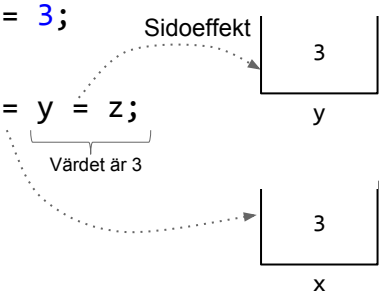


Inkrementering eller dekrementering ökar eller minskar en numerisk variabels värde med 1.

- Inkrementeringsoperatorn `++` ökar variabelns värde med 1
- Dekrementeringsoperatorn `--` minskar variabelns värde med 1
- Kan inte användas för literaler: `5++` går inte (literaler är fixa värden)!
- `++/--` kan skrivas både före och efter variabelnamnet, vi skriver bara efter, mer senare.
- Vi använder detta bara för heltalsvariabler

Uttryck med Sidoeffekter

```
int z = 3;
int y;
int x = y = z;
```




```
int a = 2;
out.println(a++);  // Print 2
out.println(a);    // Print 3
```

25

Ett uttryck står för ett värde, ett värde beräknas men …

- … ibland sker något mer, kallas en **sidoeffekt**
  - Innebär vanligen att minnet ändras
- Tilldelning och in/de-krementering är uttryck med sidoeffekter.

Tilldelning är ett uttryck med sidoeffekt

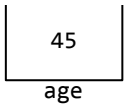
- Värdet av uttrycket är "det tilldelade"
- Sidoeffekten är att variabelns värde ändras.

In/de-krementering är uttryck med sidoeffekter

- Värdet av uttrycket då ++/-- står efter variabeln är det aktuella värdet (innan ändring)
- Sidoeffekten är att variabelns värde ökar/minskar med 1 (efter att värdet avlästs)

Konstanta Variabler

```
// Declaration and initialization
final int age = 45;

age = 48;   // Compile error
age++;      // Compile error
```



26

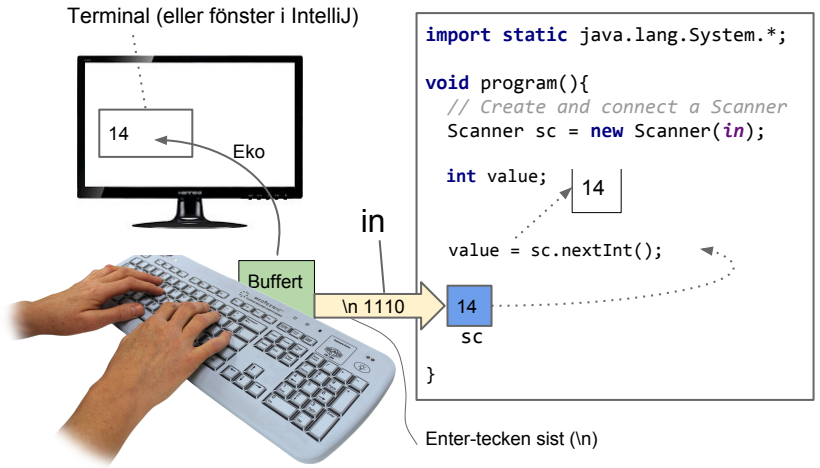
En konstant variabel kan inte ändras

- Att variabeln är konstant anges med **final** framför typen
- Måste ges ett värde vid deklarationen (eftersom den inte kan ändras)

Konstanta variabler kan användas för att slippa “magic numbers” i koden.

- Vi vill normalt inte ha värden (literaler) rakt i koden.
  - Om ett värde hårdkodas på flera ställen blir det jobbigt och riskabelt att ändra.
- Ett värde rakt av kan vara svårt att förstå. Vad syftar värdet på?
  - Bättre att ge värdet “ett namn” genom att skapa en konstant variabel.

Inläsningssatser



35

Alla Java program har automatiskt tillgång till en byte-ström med namnet **in**.

- in är i vårt fall kopplad till tangentbordet.
- Vi måste skriva import static java.lang.System.* för att kunna använda strömmen.
- Genom att använda namnet "in" i koden får programmet tillgång till strömmen
- Vi använder inte inströmmen direkt utan kopplar den till en Scanner
  - En Scanner kan ta emot ett antal bytes från strömmen och översätta dessa till numeriska värden eller strängar
  - Vi måste själva skapa en Scanner och koppla strömmen.
  - Vi använder Scannern i in inläsningssats: sc.nextInt() för att läsa in ett heltal
  - sc.nextLine(), sc.nextDouble() och sc.nextBoolean() finns också (nextLine() ger en sträng)

Följande sker (se bild)

- Programmet kommer till inmatningsatsen, sc.nextInt(), där det stannar och väntar på en inmatning
- Vi skriver på tangentbordet
  - Det vi skriver sparas i en buffert (inget skickas till programmet, alltså ingen inmatning än)

    - Innehållet i bufferten visas på skärmen (ett eko, så vi ser vad vi skriver)
    - Om vi vill kan vi radera i bufferten m.h.a. backspace
- När vi är klara skickar vi allt i bufferten till programmet genom att trycka Enter
    - Hela buffertinnehållet skickas då till Scanner:n som försöker omvandla innehållet till t.ex. ett heltal.
    - Det kan hända att Scanner:n bara kan omvandla en del till t.ex. ett numeriskt värde
    - … om så kan det ligga kvar tecken i inströmmen (det som inte gick att använda).
- Tilldelningen gör att det omvandlade värdet i Scanner:n kopieras till variabeln value.

Undantag

```
value = scan.nextInt();
```

```
12a34
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at samples.old.IO.program(IO.java:31)
    at samples.old.IO.main(IO.java:13)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62
)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl
.java:43)
    at java.lang.reflect.Method.invoke(Method.java:483)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
```

Sökriktning

28

Om vi skickar något som Scanner:n inte kan omvandla för vi ett **undantag** (exception)

- Om vi matar in 12a34 (i bilden) kan inte detta omvandlas till ett heltal
- Programmet vet inte vad det skall göra och ett undantag uppstår, programmet avbryts (kraschar).
    - I samband med detta skrivs ett felmeddelande ut, Java försöker berätta vad undantaget berodde på
        - I detta fall InputMismatchException (alltså det vi skrev matchade inte vad Scanner:n förväntade sig).
    - Meddelandet förklarar var undantaget uppstod (vilken fil och rad i filen)
        - Meddelandet räknar upp en massa olika filer och rader, …
        - … det mesta är kod från API:er, där finns inte felet …
        - … vi måste leta efter en fil vi känner igen (som vi skrivit)
            - I den filen, på angiven rad, uppstår undantaget
        - Man börjar alltid att läsa uppifrån!
    - Det tar tid att lära sig förstå vad felmeddelandet försöker säga… viktigt att börja nu!!

Tills vidare accepterar vi detta beteende vid inläsningar och åtgärdar inte. Fortsättning följer ….

Inmatning och Användare

29

Normalt kan vi aldrig lita på att användaren gör som vi vill t.ex.

- … matar in korrekt data på korrekt sätt o.s.v …
- Men, tills vidare antar vi att om inget annat anges så sköts all inmatning korrekt …
- … d.v.s. våra program kontrollerar inte vad användaren skriver in
- Om kontroll skall göras anges detta särskilt (t.ex. i uppgiften)!

Numeriska Operationer

```
import static java.lang.Math.*; // Must have

double d = sqrt(25); // 5.0
d = pow(5, 2);       // 25.0
d = floor(4.6);      // 4.0
d = ceil(4.4);       // 5.0
int i = floor(4.4);  //Type error
int j = ...
d = pow(6, 2*j);  // Ok, to use int as arg.
...
```


För numeriska operationen (funktioner) finns ett API som heter Math.

- Måste ange: import static java.lang.Math.*;
- Vi får då tillgång till metoder för kvadratrot, logaritm, trigonometri, m.fl.
    - Alla metoder tar double värden som argument och returnerar ett värde av typen double
- Finns även färdiga konstanter för **π** (skrivs PI) och e (skrivs E)

Slumptal

```
import java.util.Random;

// Create random generator
final Random rand = new Random();

void program() {
   int value;
   // Ask it for a random int
   value = rand.nextInt(3) + 1; // 1-3
}
```

31

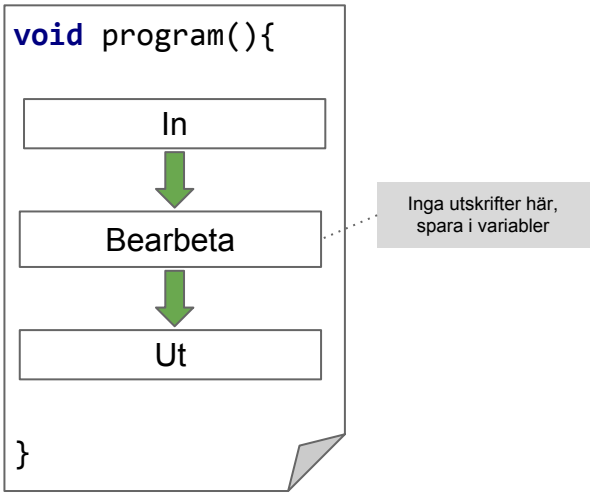
I spelprogram, simuleringar m. m. behövs ofta slumptal

- I Java-program kan man använda en slumptalsgenerator
- Man måste först skapa en generator
- Därefter kan man be den om olika typer av slumptal t.ex. slumpmässiga heltal

Analys av kod

- final-raden betyder (ungefär) att vi skapat en slumptalsgenerator som heter rand.
  - Detta sker utanför (före) program(): Ett specialfall!
    - Ni skall undvika detta!
    - Deklarera allt i program() tills vidare ...
- I program() deklarerar vi en heltalsvariabel value
- Uttrycket rand.nextInt(3) säger åt slumptalsgeneratorn att skap ett slumptal i intervallet [0, 2] (d.v.s. givet talet n för vi max ut n-1)
  - Genom att addera 1 flyttas intervallet därefter till [1,3]
  - Resultatet tilldelas value.

Grundläggande Struktur



32

Vi försöker alltid att strukturera ett program enligt: In - > bearbeta (logik) - > ut.

- D.v.s läs in data, bearbeta, visa resultat
- Vi skriver inte ut resultat direkt utan sparar undan och skriver ut efter bearbetningen

Fördelar.

- Lättare att hitta i koden. Olika delar av koden ansvarar för olika saker.
  - Vid ev. fel, lättare att hitta utmatningen och kontrollera denna.
  - Enklare att skriva tester, logiken separerad, mer senare ...
- Vi kanske vill visa programmet på något annat sätt (en App eller Webbsida).
  - Enklare att anpassa programmet om logiken är separat.