

ex1basics

Joachim von Hacht

Program och Fil

```
public class Ex3SumAvg {  
    public static void main( ...  
        new Ex3SumAvg()...  
    }  
  
    final Scanner sc = ...  
}
```

Program-
namnet (ungefär,
förenklat mer
senare)

Ex3SumAvg.java
(en textfil)

2

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**)
- OBS! All källkod skrivs på engelska.
- Java kräver teckenkodningen [UTF-8](#) för källkodsfiler.
- Källkodsfiler använder suffixet .java, t.ex. SumAvg.java (namnet på filen inleds med stor bokstav)
 - 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 Ex3SumAvg, anger ungefär att programmet "heter" Ex3 SumAvg, det ligger i filen Ex3SumAvg.java

En Mall för Java Program

```
package exercises;
import static java.lang.System.*;
// Program "named" Ex3SumAvg
public class Ex3SumAvg {
    public static void main(String[] args) {
        new Ex3SumAvg().program();
    }

    void program() {
        // Statements
    }
}
```

Behöver inte förstå just nu

Här skriver vi koden

Ex3SumAvg.java

3

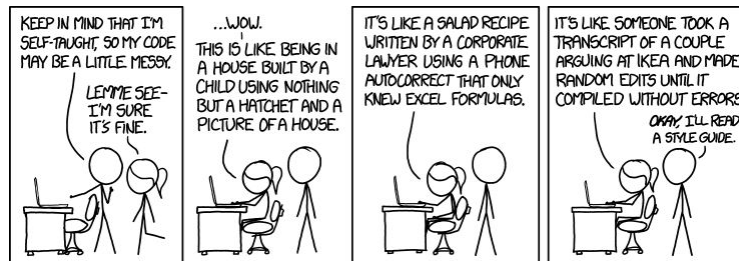
För att förenkla för nybörjare använder vi en "mall" för våra program.

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 ett IntelliJ projektet.
- Vid **import** anger vi vilka färdiga Java-resurser 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 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.

Kodstil

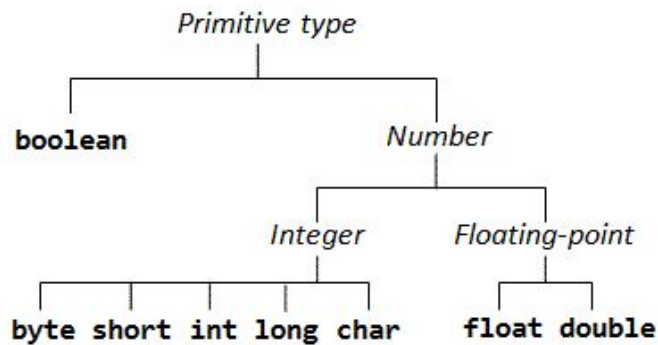


4

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

- Att alla använder samma stil handlar om kommunikation, samma stil underlättar kommunikation mellan programmerare.
- Dessutom: Stillös (rörig) kod är svår att felsöka!
- Använd samma stil som exempelkoden!
 - Allt är genomtänkt, att det skrivs på ett visst sätt har en orsak!

Primitiva Typer i Java



5

Alla värden i Java har en typ (tillhör en viss mängd).

Följande typer finns färdiga i Java.

- int, (integer), typen för (ändliga) heltal
- double, typen för reella (ändliga) tal, kallas också flyttal ([floating point numbers](#))
- boolean, typen för sanningsvärden (finns bara true och false)
- char (character), typen för enstaka tecken.
- m.fl. se bild.
- De ovan kallas primitiva (ungefär som atomära)

Literaler

```
143567      // Integer Literal

25.345      // Real Literal (floating point Literal)

true        // Boolean Literal

'Z'         // Character Literal

"Hello world!" // String Literal
```

6

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 [flyttal](#)sliteral (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 (o-n) som omges av dubbla citationstecken.
 - Kan innehålla blanka (osynliga) tecken såsom blanksteg..
 - Tomma strängen skrivs : "" (följd av o 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

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
```

7

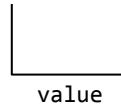
Alla värden i Java har en typ. Literaler representerar värden och klassificeras automatiskt som tillhörande en viss typ, se bild.

- Alla utom String är primitiva typer, String är en referenstyp, mer senare..

Variabler

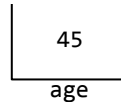
// Variable declaration

```
int value;
```



// Declaration and initialization

```
int age = 45;
```



// More declarations and

// initializations

```
double d = 25.345;
```

```
char ch = '?';
```

```
boolean b = false;
```

```
String s = "Hello";
```

8

En variabel i Java (i imperativa språk) är en ändringsbar behållare för värden.

- Eftersom variabeln representerar ett värde måste den ha en typ.
- För att kunna hänvisa till variabeln senare i programmet måste den ha ett namn.

En variabel deklaration innebär att vi anger typ och namn för variabeln

- Efter deklarationen kan vi använda variabeln för att spara eller avläsa värden.
- Vi ritar variabler som öppna lådor med namnet under och värdet i.

När vi senare i programmet vill använda variabeln skriver vi bara namnet

- Namnet syftar på namnet vi angav vid deklarationen

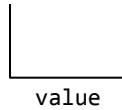
Initiering av variabler

- En variabel kan (i vissa fall måste) 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!

Tilldelningssatser

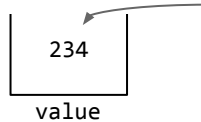
// Declaration

int value;



// Assignment

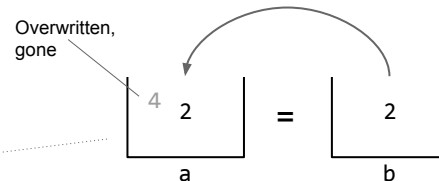
value = 234;



int a = 4;

int b = 2;

a = b;



10

Tilldelning ([assignment](#))

- Skrivs m.h.a. **tilldelningsoperatoren** "=" (likhetstecken)
 - Helt annan betydelse än i matematik
- Innebär att ett värde från höger sida om operatoren kopieras till en variabel på vänster sida.
 - På vänstersidan om operatoren 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 ==-operatoren, 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)

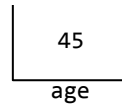
För att förstå programkod läser man som vanlig text d.v.s. uppifrån ned och vänster till höger men ..

- ... i vissa fall måste man läsa höger till vänster t.ex. vid initiering och tilldelning.

Konstanta Variabler

// Declaration and initialization

final int age = 45;



age = 48; *// Compile error*

10

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 (literals) 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.

Operatorer

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

13

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

Aritmetik

```
// +, -, * as usual but / tricky

out.println(5 / 2);           // Integer division
out.println(5.0 / 2);        // Real division
out.println((double) 5 / 2); // Force real division
out.println(4 + 6 / 3 * 2);   // Is 2 below / ??
out.println((4 + 6) / (3 * 2));

// This is *NOT* exponentiation (no such operator)
out.println(3 ^ 4);
```

12

Allt vi behöver finns och fungerar som vi är vana vid (addition, subtraktion, multiplikation, division).

- Dock! Se upp med division!
 - Om operanderna är heltal blir det heltalsdivision!
 - Vad står över/under divisionstecknet?!

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);

out.println(PI);
```

13

Finns färdiga matematiska funktioner (kallas metoder i Java).

- Måste ange: `import static java.lang.Math.*;`
- Vi får då tillgång till metoder för kvadratroten, logaritm, trigonometri, m.fl.
- Finns även färdiga konstanter för π (skrivs `PI`) och e (skrivs `E`)

Överlagrad +-operator

12 + 4 -> 16

12.0 + 4 -> 16.0

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

"123" + 4 + 5 -> "12345"

1 + 2 + "345" -> "3345"

14

+-operatorn fungerar olika beroende på operandernas typ

- För numeriska typer t.ex. sker vanlig addition
- För strängar sker sammanslagning, **konkatenering** om minst en av operanderna är en sträng.
- Att operatorn beter sig olika utifrån operandernas typer kallas att den är **överlagrad**
 - Finns bara en sådan operator i Java

Sammanfatta Tilldelningsoperatorer

<code>x += 1;</code>	<code>// x = x + 1;</code>
<code>x -= 2</code>	<code>// x = x - 2;</code>
<code>x *= 3;</code>	<code>// x = 3 * x;</code>
<code>x /= 2;</code>	<code>// x = x / 2;</code>
<code>x %= 2;</code>	<code>// x = x % 2;</code>

15

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

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

Uttryck

123

value + 10.0

(count + 1) / max

constraint = true

16

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, mer strax)
- Alla värden måste ha en typ d.v.s. ett uttryck har en typ!
 - Typen för uttrycket beräknas också

Satser

```
// Statement, normally ; last  
out.println("Too small");  
  
nGuesses++;  
  
int theNumber = 87;  
  
return found;
```

20

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

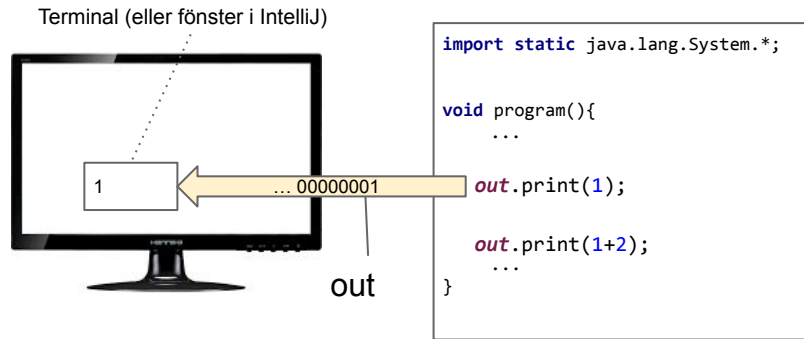
- Är den minsta fristående enheten (atomen).
- Är ett **imperativ**, en uppmaning till datorn att göra något (därav imperativ programmering)
- Måste avslutas med semikolon, ";" visar var satsen är slut (några undantag senare).
 - Jämför: en mening på svenska, avslutas med punkt (eller ?, !)
 - En sats 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, uppfifrån och ner)! Datorn gör som vi skriver (... inte som vi vill...)

Satser byggs i sin tur upp av

- literaler, variabler, operatorer, metodanrop, ... (d.v.s. uttryck)
- Dessa kan inte köras fristående, de måste ingå i en sats.

Utmatningssatser

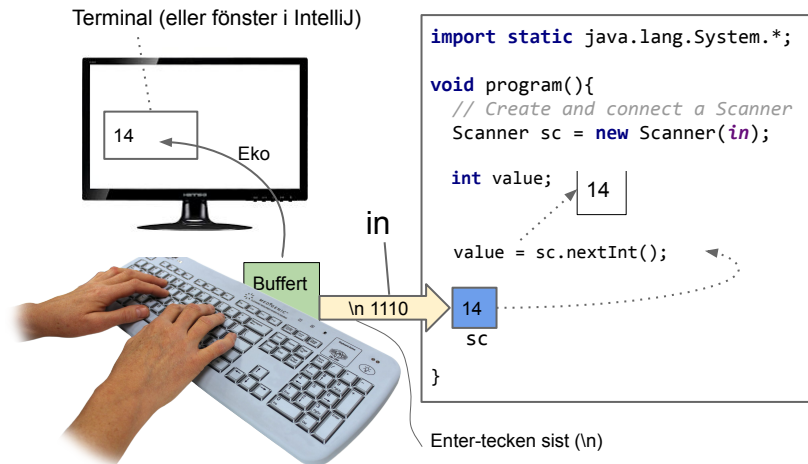


18

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 "bara" out.
 - En hel del kodexempel visar `System.out.println(...)` ... för jobbigt att skriva, vårt sätt är mer ekonomiskt.

Inläsningssatser



23

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 ihop den med inströmmen.
 - Vi använder Scannern i in inläsningssats t.ex.: `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,

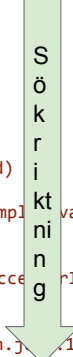
- alltså ingen inmatning än)
- Innehållet i bufferten visas på skärmen (ett eko, så vi ser vad vi skriver)
- Om vi vill kan vi radera i bufferten m.h.a. backspace
- När vi är klara skickar vi allt i bufferten till programmet genom att trycka Enter
 - Hela buffertinnehållet skickas då till Scanner:n som 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)
```



20

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 åtgärddar inte. Fortsättning följer

Inmatning och Användare

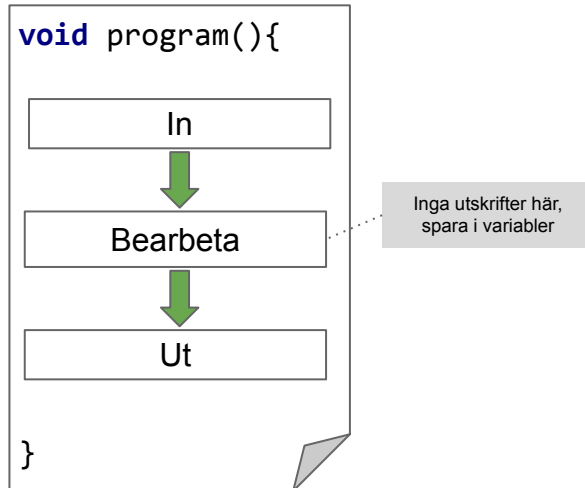


21

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)!

Grundläggande Struktur



22

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 IO är separat.