Grunderna

Vecka 1, Bildserie 2

Innehåll

- Kompilering och exekvering
- Literaler
- Primitiva typer
- Typsystem
- Uttryck
- Operatorer
- Lat evaluering
- Överlagrad operator
- Implicita typomvandlingar
- Likhet
- Variabler och initiering
- Synlighetsområde
- Tilldelning
- In- och dekrementering
- Uttryck med sidoeffekter
- Inmatning
- Undantag

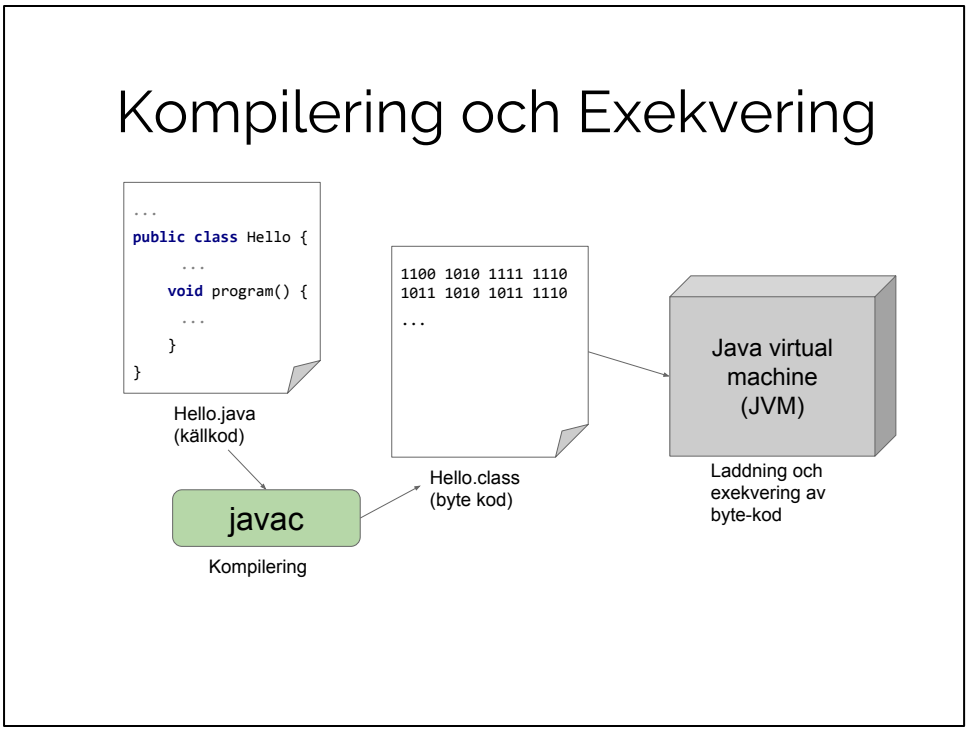
Ooop, en hel del att lära sig ...

Att Läsa i Boken

- 2.3-2.6
- 2.9-2.10
- 2.7 (ej Cast operator)
- 2.10.1-2.10.2
- 2.11
- 2.13-2.14
- 2.18
- 3.1-3.2, 3.10

OBS! Att jag använder lite annorlunda stil

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



Kompilering

- Innan ett Java-program kan exekveras måste det **kompileras**
- Ett program kallat javac (= java compiler, en kompilator) översätter källkoden till en binärfil med bytekod
- Bytekoden 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
- 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.
- Kompileringen sköts automatiskt av IntelliJ (i bakgrunden) så fort källkoden ändras.

Ett kompilerat program (byte-koden) körs på en virtuell maskin (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 operativ)
- Dock måste din dator måste ha en JVM för att det skall fungera.

Literaler

En heltalsliteral (integer literal)

```
143567
```

En flyttalsliteral (floating point literal)

```
25.345
```

En teckenliteral (character literal)

```
'Z'
```

En strängliteral (string literal)

```
"En halv groda dansar aldrig ensam"
```

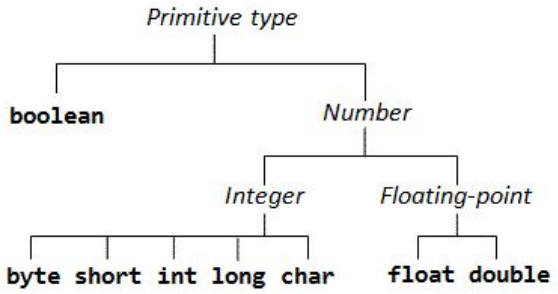
En boolesk literal (boolean literal)

```
true
```

Literaler (literals) är fixa värden i koden (hårdkodade värden, värden som aldrig ändras, bakas in i själva programkoden)

- En heltalsliteral består (oftast enbart) av siffror (ev. tecken, m.m.)
- En flyttalsliteral har en decimalpunkt i övrigt siffror (ev. tecken, exponent, m.m.)
- En teckenliteral består av ett enda tecken med enkla citat runt
  - Ett blankslag skrivs: ' ' (ett blankslag 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 så som mellanslag (dock inte nyrad).
  - Tomma strängen skrivs : “” (följd av 0 tecken)
- En boolesk literal betecknar ett sanningvärde 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

Primitiva Typer



Primitiva typer bestämmer:

- Hur mycket minne som behövs för värdet samt min och maxvärden
  - Om min/max-värdet under/överskrids, kan det leda till felaktiga resultat. Kallas underflow respektive overflow.
- Hur bitarna i minnet skall tolkas (heltal eller potensform)
- Namn på primitiva typer är reserverade ord

Bilden visar de 8 **primitiva** (inbyggda) **typerna** (mängderna) i Java. Dessa finns färdiga i språket och kan inte ändras (eller lägga till nya).

- **boolean**, är typen för sanningvärden
  - Finns bara två värden i typen: true och false
  - … som alltså ges typen boolean.
- **int**, är typen för heltal (integer)
  - Heltalsliteraler ges denna typ
  - -2147483648 till 2147483647 (32 bitar)
- **char**, är typen för enstaka tecken (character). Egentligen en teckenkod, därför räknas den som ett heltal.
  - Alla teckenliteraler ges denna typ
  - 0 till 65,535 (16 bitar).
- **double**, är typen för reella närmevärden (double precision, ca 15 decimaler)
  - Flyttalsliteraler får denna typ

    - 4.94065645841246544e-324d till 1.79769313486231570e+308d (64 bitar)
- byte, short, long och float hoppar vi över så länge (kommer troligen inte att användas i kursen)

Typen för strängar?

- Strängar är inte en primitiv typ (storlekan kan ju variera … m.m.)
- Strängliteraler ges typen **String**, mer senare …

Varför Typer?

```
out.println(123 + true); //?!?
```

Förutom att bestämma minnesbehov så används typer för att förhindra oss att använda data på ett felaktigt sätt, att blanda ihop saker och ting

- I denna kurs är detta den viktigaste aspekten på typer
- Man kan se typsystemet som en mängd tillsammans med tillåtna operationerna på elementen

Java är ett statiskt typat språk

- Språket är konstruerat för att hindra oss från att av misstag blanda ihop olika "sorter" och utifrån detta göra meningslösa eller felaktiga operationer
- Java har ett **typsystem** som skall eliminera sådana typer av fel redan vid kompileringen (innan vi kört programmet)
- I kompilatorn finns en "type checker" som sköter typkontrollen i samband med att koden kompileras.
- För att typsystemet skall fungera måste alla värden i Java ha en känd typ vid kompileringen d.v.s.
  - Alla värden måste tillhör en viss mängd (int , double, o.s.v.)
  - Java vet vilken typ av operander olika operatorer accepterar, inbyggt i språket (mer strax)

Exempel:

Värdet 123 kommer att ges typen int av typsystemet

Värdet true kommer att ges typen boolean
Operationen 123 + true är inte meningfull!

- Typsystemet vet att operationen addition inte kan utföras med typerna.

IntelliJ kommer att visa ett felmeddelande, ett **typfel** (typsäkerhet)
I detta exempel är det lätt att se felet, … men så är det inte alltid!

Ibland ges värden automatiskt en typ (t.ex. literaler), ibland måste vi ange typen

- När vi anger typ skriver vi helt enkelt int, boolean, o.s.v. i koden
- …, t.ex. vid deklarationer, mer strax…

Uttryck

Exempel på uttryck

```
123
```

```
value < 100
```

```
(count + 1) / max
```

En sats med ett uttryck i

```
out.println(constraint == true);
```

Ett uttryck

- Uttryck ingår som delar av satser
- Representerar ett värde (står för ett värde)
- Byggas upp av literaler, variabler, operatorer, m. m.
- Uttrycket **evalueras** (beräknas) och får ett slutgiltigt värde under körningen.
- Ingen speciell slutmarkering används för uttryck
- Ett uttryck är ett värde. Alla värden måste ha en typ -> Ett uttryck har en typ!

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

Lista med de flesta (men inte alla) operatorer i Java

- Associativtet: Om flera operatorer med samma **prioritet** (precedence)? Vilken görs först?
  - Mestadels: börja från vänster
- **Unär operator** tar en operand
- **Binär operator** tar två
- Operatorer ++, ++, +, -, *,/, %, , <,<=, >, >= kräver numeriska typer
  - +, -, *,/ är de vanliga binära 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 referens-typer (mer senare) som operatorer
  - OBS! == (två likhetstecken för likhet, vanligt nybörjarfel att använda ett!)
- &&, || och ! används för logiskt och, logiskt eller samt logiskt icke, kräver booleska operander

- =, tilldelningsoperatorn är mycket speciell, mer strax

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

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

Några uttryck

```
10 - 4 - 2            // 4
1 < 2                 // true
5 / 2                 // 2
4 == 4                // true  NOTE!!! ==
1 >= 1                // true
35 % 2 == 0           // false
true != false         // true
1 < 2 && 1 == 2       // false
1 < 2 || i == j       // true
!true                 // false
```

Vilket värde respektive typ har uttrycken?

Lat Evaluering

```
int i = 4;

1 < 2 || i != 4;

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

Behöver inte evalueras

Operatorerna || och && använder lat evakuering

- Om vänster operand till || är sann, evalueras inte högersidan (eftersom uttrycket är sant om någon av operanderna är sanna)
- Om vänster operand till && är falsk evalueras inte högersidan (eftersom uttrycket är falskt om någon av operanderna är falska)

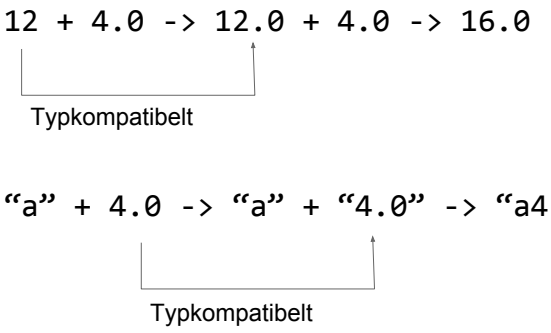
Överlagrad +-operator

```
12 + 4 -> 16

12.0 + 4.0 -> 16.0

'0' + '0' -> 96

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

+-operatorn fungerar olika beroende på operandernas typ

- För numeriska typer sker vanlig addition
  - Om typen är char används teckenkoden i additionen (char räknas ju 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

Implicita typomvandlingar



Typsystemet skall förhindra att vi blandar ihop saker.

- Men för vissa operatorer är det rimligt att tillåta olika typer t.ex. addition med heltal och reella tal.
  - Rimligt = Inga typfel kan uppstå
- I sådana fall omvandlas operanderna till samma typ enligt vissa regler för **typkompabilitet** därefter utförs operationen.
  - Operanderna omvandlas till den “större” typen (d.v.s. mer minne används)
  - Tvärtom sker normalt inte men …
    - … det kan ske t.ex. int -> char, om det innebär att ingen information försvinner (om en heltalsliteral får plats i en char-variabel)

Implicita typomvandlingar innebär att:

- **Kompatibla** typer automatiskt typomvandlas.

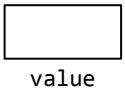
Omvandling till strängar

- Då + operatorn har en operand av typen String omvandlas den andra operanden till String, därefter sker konkatenering.

Variabler

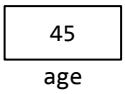
Deklaration

```
int value;
```



Deklaration och initiering

```
int age = 45;
```




```
// Some more declarations and initializations
double d = 25.345;
char ch = 'Z';
boolean b = true;
String s = "Hello";
```

Variabler i Java (i imperativa språk) är ändringbara behållare för värden

- Variabler är ett sätt för programmet att komma ihåg saker

En variabel måste **deklareras** innan den kan användas.

- Deklarationen gör variabeln känd för programmet.
- Vid deklaration anges typ och namn
  - Typen anger vilken typ av värden som kan lagras i variabeln t. ex. int eller double.
    - Typsystemet måste veta
  - Namn kallas **identifierare**
    - Vi kan själva välja namn men det finns regler för hur dessa får skrivas
      - I princip bokstäver, siffror och "_" .
      - Får inte inledas med siffra, inte innehålla osynliga (vita) tecken.
    - Namngivning av variabler är svårt, namnet skall
      - Inledas med liten bokstav
      - Sammasatta namn skrivs camelCase (som ovan)
      - 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 block i programmet kan ha korta namn, typiskt i, j m, n.
    - Variabeldeklarationer är satser (avslutas med ";")

När vi senare i programmet använder namnet behöver vi inte ange någon typ (vi har ju redan sagt vilken typen är, variabeln och typen är känd eftersom vi deklarerat den)

Initiering av variabler

- En variabel måste ges ett värde innan den kan användas.
- Kan göras med en initiering i samband med deklarationen.
- Initieringvärdet måste vara typkompatibelt med variabelns typ!

Synlighetsområde

```
{
 int i = 0;

}


{
 int i = 2;
}
```

```
{

 int i = 0;

    {
        int i = 2; // Bad
        int j = 6;
    }

}
```

Synlighetområdet (scope) anger var i programmet det är möjligt att använda en variabel

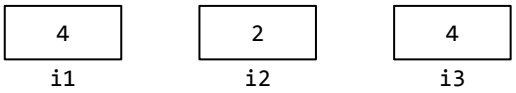
- I Java sammanfaller synlighetsområde och block (förenklat)

Följande gäller

- Variabler är bara synliga (kan bara användas) i det synlighetsområde de är deklarerade (fr.o.m. deklarationen och vidare), förutom …
    - … nästlade synlighetsområden
        - Ett inre block kommer åt variabler i ett yttre
        - Yttre block kommer inte åt variabler i ett inre
- I Java gäller att variabler inte får ha samma namn inom samma synlighetsområde
- Inom olika synlighetsområden kan samma namn användas
    - Mycket praktiskt: Slipper hitta på nya namn hela tiden!

Likhet för Variabler

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



Likehet för variabler innebär att innehållet i respektive variabel jämförs

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

Likhet för Flyttal

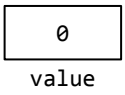
```
double d1 = 1.0;
double d2 = d1 - 0.6 - 0.4;
double d3 = d1 - 0.9 - 0.1;

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

Flyttal är närmevärden

- Skall aldrig jämföras med ==
- Finns färdig abs() function, vi använder abs(d1 - d2) < eps, mer senare...

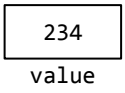
Tilldelningsatsen

```
int value = 0;
```



Tilldelning

```
value = 234;
```



Tilldelning innebär att man skriver över en variabels värde med ett nytt värde

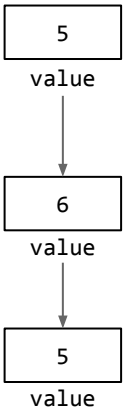
- Skrivs m.h.a. **tilldelningsoperatorn** = (likhetstecken)
- Innebär att ett värdet från höger sida om =-operatorn kopieras till en variabel på vänster sida.
  - OBS! Helt annan betydelse än i matematik!!!
  - På vänstersidan om = skall det alltid stå en variabelnamn (variabeln dit värdet skall kopieras)
    - Variabelnamn kan var "konstiga" t.ex.: c.n, a[0], c.get()[1] och liknande …
  - Om det står ett uttryck på höger sida beräknas detta först
  - Typen på uttrycket till höger och variabeln måste vara kompatibla, annars typfel

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

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

In- och Dekrementering

```
int value = 5;
```



Inkrementering

```
value++:
```

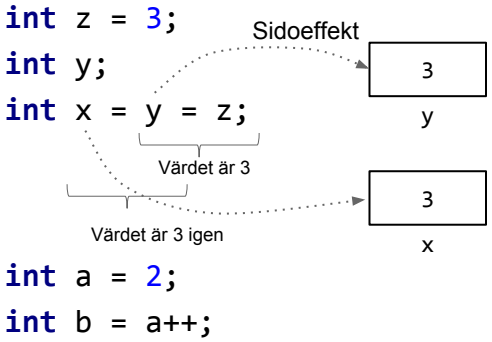
Dekrementering

```
value--;
```

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

- Vi använder bara operatorerna tillsammans med heltalsvariabler
- 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 skriva både före och efter variabelnamnet, vi skriver bara efter

Uttryck med Sidoeffekter



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

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

In/de-krementering är uttryck

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

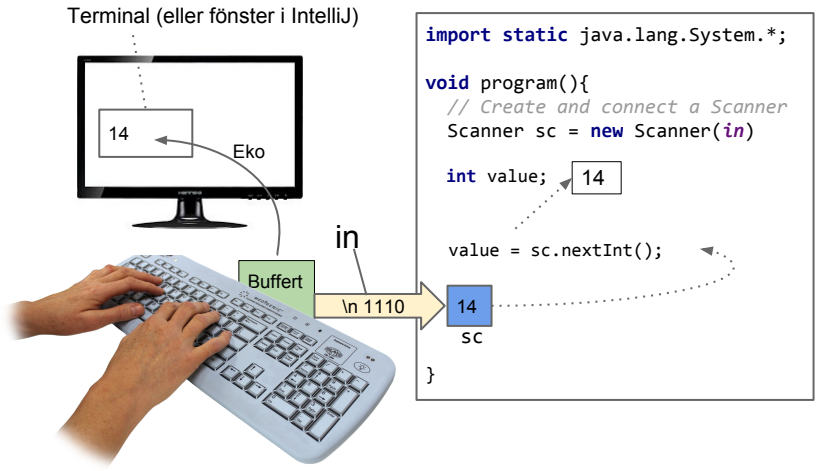
Sammansatta Tilldelningsoperatorer

```
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;
```

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

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

Inmatningssatser



Alla Java program har automatiskt tillgång till en byteström, in.

- in är i vårt fall kopplad till tangentbordet.
- Genom att använda namnet "in" i koden får programmet tillgång till strömmen
- Vi använder inte inströmmen direkt utan kopplar den till en Scanner
  - Vi måste själva skapa en Scanner och koppla strömmen.
  - En Scanner kan ta emot ett antal bytes och översätta dessa till numeriska värden eller strängar
- 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)
- Vi måste skriva import static java.lang.System.* för att kunna använda in.

Följande sker (se bild)

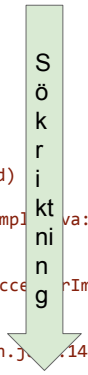
- Programmet kommer till inmatningsatsen, sc.nextInt(), där det stannar och väntar på en inmatning
- Vi skriver på tangentbordet
  - Det vi skriver sparas i en buffert (inget skickas till programmet, alltså ingen inmatning än)
  - Innehållet i bufferten visas på skärmen (ett eko, så vi ser vad

    - 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 numersikt 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 kopiera till variabeln value.

Undantag

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

```
123a
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)
```



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

- Om vi matar in 123a (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
        - …det tar tid att lära sig förstå vad felmeddelandet försöker säga… viktigt att börja NU!
        - Man börjar alltid att leta uppifrån!
    - Meddelandet förklarar var undantaget uppstod (vilken fil och rad i filen)
        - Meddelandet räknar upp en massa olika filer och rader, …
        - det mesta är färdig Java kod (kod vi inte skrivit), där finns inte felet …
        - … vi måste leta efter en fil vi känner igen (som vi skrivit)
            - I den filen, på angiven rad, uppstår undantaget
- Tills vidare accepterar vi detta beteende vid inmatningar och åtgärdar inte. Fortsättning följer ….

Inmatning och Användare

Normalt kan vi aldrig lite 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)!

Programmering

```
Please, enter your weight (kg) > 80
Please, enter your height (m) > 1.78
BMI = 25.24933720489837
```

Hur skriver man ett dylikt program (övningen fanns med på intro-lab-passet)?

- Grön skrift är det vi matar in till programmet.

Att Göra

- Quiz 2
- Börja med laboration 1