252-0027
Einführung in die Programmierung

2.0 Einfache Java Programme

Thomas R. Gross

Department Informatik
ETH Zürich

252-0027
Einführung in die Programmierung

2.4 Verzweigungen

Thomas R. Gross

Department Informatik
ETH Zürich

Übersicht

- **2.4 Verzweigungen**
  - 2.4.1 «if»-Anweisungen
    Vergleichsoperatoren
  - 2.4.2 Typ boolean
  - 2.4.3 Bedingte («short-circuit») Auswertung
  - 2.4.4 Pre- und Postconditions
  - 2.4.5 «Schwächste» Vorbedingung

2.4.5 Schwächste Vorbedingung

Anonymous *03.10. 11:45* *Delete*

was ist der Sinn von { y > 5} Wenn y=20 zugeschrieben wird?


```
// x, y >= 0
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
```

```
v = z / (y / 6) ;
```

2.4.5 Schwächste Vorbedingung

Anonymous *03.10. 11:45* *Delete*

was ist der Sinn von { y > 5} Wenn y=20 zugeschrieben wird?


```
// x, y >= 0
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
// y == 5
v = z / (y / 6) ;
```

2.4.5 Schwächste Vorbedingung

Anonymous *03.10. 11:45* *Delete*

was ist der Sinn von { y > 5} Wenn y=20 zugeschrieben wird?


```
// x, y >= 0
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
// y == 5
v = z / (5 / 6) ;
```

2.4.5 Schwächste Vorbedingung

Anonymous *03.10. 11:45* *Delete*

was ist der Sinn von { y > 5} Wenn y=20 zugeschrieben wird?


```
// x, y >= 0
if (x > 7) { y = x; }
else { y = 20; }
{y > 5}
// y == 5
v = z / 0 ;
```

Was für eine Vorbedingung wollen wir?

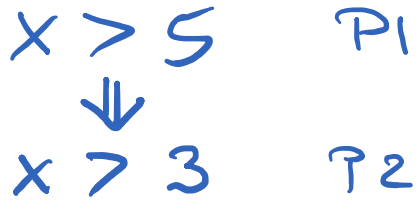
x, y `int`; kein Over/Underflow

```
{x>10}
y = x+1;
{y ≥ 2}
```

```
{x≥1}
y = x+1;
{y ≥ 2}
```

```
{x>5}
y = x+1;
{y ≥ 2}
```

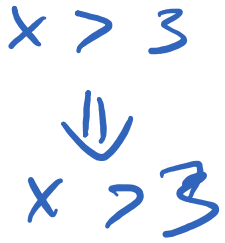
Aussagen über Zustände



Wir haben zwei Aussagen P1 und P2

Wenn P1 ⇒ P2 (also P1 impliziert P2) gilt dann sagen wir:

- P1 ist *stärker* («*stronger*») als P2
- P2 ist *schwächer* («*weaker*») als P1



- **Wenn immer P1 gilt, dann gilt auch P2**
- **Es ist also schwieriger (oder zumindest genauso schwierig) P1 zu erfüllen als (wie) P2 zu erfüllen**
- «stärker als»: «stärker als oder genauso stark wie»

Warum ist das interessant?

- **Stellen wir uns vor:**
  - Es gilt {P} S {Q}, und
  - P1 ist stärker als die Precondition P, und
  - Postcondition Q ist stärker als eine Aussage Q1

- **Dann gilt:**
  - {P1} S {Q}
  - {P} S {Q1}
  - {P1} S {Q1}

- **Stärkere Preconditions (Vorbedingungen) oder schwächere Postconditions sind einfach**
  - Wenn {P} S {Q} gilt
- **Aber was für ein** P **wollen wir wenn** S **und** Q **gegeben sind?**

  {???} S {Q}

- **Am besten wäre es wenn wir zeigen könnten dass** $\{P_s\}$ S {Q} **gilt, wobei** P_s **die *schwächste Precondition* von** Q **für** S **ist**
  - Schwächste heisst: hat die wenigsten Annahmen so dass Q gilt
  - Jede Precondition P so dass {P}S{Q} gültig ist, ist dann stärker als P_s, d.h., P => P_s

- **Ohne Schleifen und Methoden gibt es für jedes Programmsegment S und jede Postcondition Q eine eindeutige *schwächste Precondition* («*weakest precondition*»), abgekürzt wp(S, Q)**

- **Variable sind immer `int` Variable, ohne Over/Underflow**

wp(S, Q)

- **wp(x = e, Q) ist Q in dem jedes x durch e ersetzt wurde**

- **Bespiel:**

  wp(x = y*y, x>4) ist (y*y>4), d.h., |y| > 2

wp(S1;S2,Q)

- **wp(S1;S2, Q) ist wp(S1, wp(S2, Q))**
  - D.h. Sei R die wp(S2,Q) dann ist die schwächste Precondition für die *Folge* S1;S2 die wp(S1, R).

- **Beispiel:**

  wp(y=x+3; z=y+1, z>4) ist (x+3>3), d.h., (x>0)

  warum? R sei die wp(z=y+1, z>4)

  R ist (y+1>4) d.h. (y>3)

  dann ist wp(y=x+3, y>3) doch (x+3>3), d.h., (x>0)

Beispiel

- **Wenn** `S` **die Folge** `y = x+1; z = y-3;` **von Anweisungen ist, und** `Q` **ist** `z == 10`**, dann ist wp(**`S`**,** `Q`**) …**

  wp(`y = x+1; z = y-3, z == 10`**)**

  wp(`y = x+1,` **wp(**`z = y-3, z == 10`**))**

  wp(`y = x+1, y-3 == 10`**)**

  `(x+1)-3 == 10`

  `x == 12`

Beispiel

- **Wenn S die Folge** `y = x+1; z = y-3;` **von Anweisungen ist, und Q ist** `z == 10`, **dann ist wp(S, Q) …**

  wp(`y = x+1; z = y-3,` `z == 10`**)**

  wp(`y = x+1,` **wp(**`z = y-3, z == 10`**))**

  wp(`y = x+1, y-3 == 10`**)**

  `(x+1)-3 == 10`

  `x == 12`

Beispiel (ohne Vereinfachung)

- **Wenn** `S` **die Folge** `y = x+1; z = y-3;` **von Anweisungen ist, und** `Q` **ist** `z == 10`**, dann wp(**`S`**,**`Q`**) …**

  wp(`y = x+1; z = y-3, z == 10`**)**

  wp(`y = x+1,` **wp(**`z = y-3, z == 10`**))**

  wp(`y = x+1, y-3 == 10`**)**

  wp(`y = x+1, y == 13`**)**

  `x+1 == 13`

  `x == 12`

If-Statements

```
{P} if (b) S1 else S2 {Q}
```

- **Tripel ist gültig wenn (und nur wenn) es Aussagen** `Q1, Q2` **gibt so dass**
  1. `{P ∧ b} S1 {Q1}` ist gültig *und*
  2. `{P ∧ !b} S2 {Q2}` ist gültig *und*
  3. Nach dem if-Statement gilt Q, d.h.
     a) Aus Q1 folgt Q
     b) Aus Q2 folgt Q

wp(`if` …, Q)

- **wp(`if b S1 else S2, Q`) – möglich wäre**

  `(b ∧ wp(S1,Q1)) ∨ (!b ∧ wp(S2,Q2))`

- **Aber wir wollen die schwächste Vorbedingung**
- **Da `Q1 ⇒ Q` ist wp(`S1, Q1`) stärker als wp(`S1, Q`) und wg.**

  `Q2 ⇒ Q` ist wp(`S2, Q2`) stärker als wp(`S2, Q`)
  - «stärker als» lässt auch «genauso stark wie» zu

wp(`if` …, Q)

- **Da `Q1` ⇒ `Q` ist wp(`S1, Q1`) stärker als wp(`S1, Q`) und wg.**

  `Q2` ⇒ `Q` **ist wp(`S2, Q2`) stärker als wp(`S2, Q`)**

- **wp(`if b S1 else S2, Q`) ist:**

  `(b ∧ wp(S1,Q)) ∨ (!b ∧ wp(S2,Q))`

- **Bemerkungen**
  - Egal welchen Zustand die Ausführung erreicht hat, b ist entweder wahr (`true`) oder falsch (`false`)
  - Oft kann dieser Ausdruck dann weiter vereinfacht werden

Beispiel

```
S ist if (x < 5) {
        y = x*x;
      } else {
        y = x+1;
      }
Q ist y >= 9
```

wp(S, y >= 9**)**

= (x < 5 ∧ wp(y = x*x, y >= 9))

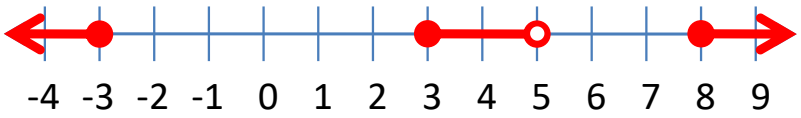
∨ (x >= 5 ∧ wp(y = x+1, y >= 9))

= (x < 5 ∧ x*x >= 9)

∨ (x >= 5 ∧ x+1 >= 9)

= (x <= -3) ∨ (x >= 3 ∧ x < 5)

∨ (x >= 8)



wp(S, Q) = `true`

- **Wenn wp(`S`,`Q`) = `true` dann heisst das, dass die Ausführung von `S` immer einen Zustand produziert in dem `Q` gilt.**
  - `true` gilt für jeden Zustand
  - Keine Annahmen

Die schwächste Vorbedingung

- **Stellen wir uns vor:**
  - Wir haben S (Programm oder Statement) und Aussage Q, die nach Ausführung von S gelten soll.
  - Jetzt suchen wir P so dass $\{P\}S\{Q\}$ gültig ist
- **Nehmen wir es gibt verschiedene Aussagen P_i so dass $P_1 \Rightarrow P_2 \Rightarrow P_3 \Rightarrow P_4 \Rightarrow \ldots$**
  - P_i ist schwächer als Aussage P_{i-1}
- **Nehmen wir weiter an dass $\{P_1\}S\{Q\}, \{P_2\}S\{Q\}, \ldots,$ $\{P_4\}S\{Q\}$ gilt, aber *nicht* $\{P_5\}S\{Q\}$**
  - P_5 reicht nicht aus, um zu zeigen, dass Q nach S wahr ist

Beispiel

P1: a==4

P2: a>3

P3: a>=3

P4: a>=2

P5: a>=–1

{P_i}	**{P_1} S {Q}**	**{P_2} S {Q}**
S: `x = 5+a;`	`x = 5+a;`	`x = 5+a;`
Q: { x >= 5}	**x >= 5**	**x >= 5**
	gilt	*gilt*

Beispiel

P1: a==4

P2: a>3

P3: a>=3

P4: a>=2

P5: a>=–1

	$\{P_i\}$	**$\{P_4\}$ S $\{Q\}$**	**$\{P_5\}$ S $\{Q\}$**
S:	`x = 5+a;`	`x = 5+a;`	`x = 5+a;`
Q:	**{ x >= 5}**	**x >= 5**	**x >= 5**
		gilt	*gilt nicht*

Die schwächste Vorbedingung

- **Stellen wir uns vor:**
  - Wir haben S (Program oder Statement) und Aussage Q, die nach Ausführung von S gelten soll.
  - Jetzt suchen wir P so dass $\{P\}S\{Q\}$ gültig ist
- **Nehmen wir es gibt verschiedene Aussagen P_i so dass $P_1 \Rightarrow P_2 \Rightarrow P_3 \Rightarrow P_4 \Rightarrow \ldots$**
  - P_i ist schwächer als Aussage P_{i-1}
- **Vielleicht ist es einfacher zu zeigen, dass $\{P_1\}S\{Q\}$ gilt als $\{P_2\}S\{Q\}$**
  - Denn $P_1 \Rightarrow P_2$ usw.

Die schwächste Vorbedingung

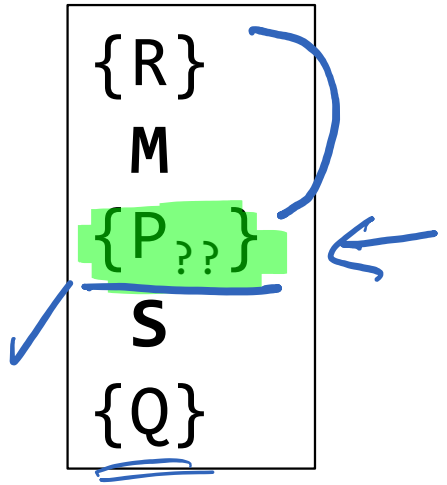
- **Vielleicht ist es einfacher zu zeigen, dass $\{P_1\}S\{Q\}$ gilt als $\{P_2\}S\{Q\}$**
  - Denn $P_1 \Rightarrow P_2$ usw.
- ***Aber* wir müssen vielleicht irgendwann zeigen, dass P_1 gilt, nachdem M ausgeführt wurde**

$\{R\}$
M
$\{P_{??}\}$
S
$\{Q\}$

  - Daher sind wir an der *schwächsten* Aussage P" interessiert, die gerade noch ausreicht so das {P"}S{Q} gültig ist
  - Also P" ist die Aussage P an der wir interessiert sind.

Die schwächste Vorbedingung

- **Vielleicht ist es einfacher zu zeigen, dass $\{P_1\}S\{Q\}$ gilt als $\{P_2\}S\{Q\}$**
  - Denn $P_1 \Rightarrow P_2$ usw.
- ***Aber* wir müssen vielleicht irgendwann zeigen, dass P_1 gilt, nachdem M ausgeführt wurde**



  - Daher sind wir an der *schwächsten* Aussage P'' interessiert, die gerade noch ausreicht so das $\{P''\}S\{Q\}$ gültig ist
  - Also P'' ist die Aussage P an der wir interessiert sind.

Also …

- **Wenn wir rückwärts schliessen und wir wollen das {P}S{Q} gültig ist, dann zeigen wir dass {P"}S{Q} gilt, wobei P" die *schwächste Precondition* von Q für S ist .**
  - Schwächste heisst: hat die wenigsten Annahmen/Einschränkungen so dass Q gilt
  - Jede Precondition P so dass {P}S{Q} gültig ist, ist dann stärker als P", d.h., P $\Rightarrow$ P"

Eine kleine Komplikation

- **Wenn wir vorwärts schliessen, dann gibt es ein Problem mit Zuweisungen:**
  - Änderungen des Wertes einer Variablen können andere Annahmen/Aussagen beeinflussen.

Beispiel

```
{true}
w=x+y;   //S1
{w == x + y;}
x=4;     //S2
{w == x + y ∧ x == 4}
y=3;     //S4
{w == x + y ∧ x == 4 ∧ y == 3}
```

x==3 und y==4
Dann hätten wir
`w == 7`

Beispiel mit Problem

```
{true}
w=x+y;
{w == x + y;}
x=4;
{w == x + y ∧ x == 4}
y=3;
{w == x + y ∧ x == 4 ∧ y == 3}
```

Aber halt: wissen wir wirklich `w == 7` ?

Eine kleine Komplikation

- **Wenn wir vorwärts schliessen, dann gibt es ein Problem mit Zuweisungen:**
  - Änderungen des Wertes einer Variablen können andere Annahmen/Aussagen beeinflussen.

- **Lösung: Wenn wir einer Variablen einen Wert zuweisen dann müssen wir in der Postcondition den Namen dieser Variablen durch einen anderen (neuen) Namen ersetzen**
  - Dann können wir uns auf den «alten» Wert beziehen

Beispiel

```
{true}
w=x+y;
{w == x + y;}
x=4;
{w == x1 + y ∧ x == 4}
y=3;
{w == x1 + y1 ∧ x == 4 ∧ y == 3}
```

Beispiel (alternativ)

```
{true}
w=x+y;
{w == x + y;}
x=4;
{w == x_old + y ∧ x == 4}
y=3;
{w == x_old + y_old ∧ x == 4 ∧ y == 3}
```

Praktisches Beispiel

- **Austausch («swap»)**

```
tmp = x;
x = y;
y = tmp;
```

- **Ansatz:**
  - Geben Sie dem ursprünglichen Inhalt (Wert) einer Variable einen Namen so dass Sie ihn in der Postcondition verwenden können.
  - Diese Namen existieren nur für die logischen Ausdrücke; es gibt nicht entsprechende Variablen im Programm.
  - Diese zusätzlichen Namen verhindern dass wir Abhängigkeiten «vergessen»

```
{x == x_old ∧ y == y_old}
tmp = x;
{x == x_old ∧ y == y_old ∧ tmp == x_old}
x = y;
{x == y ∧ y == y_old ∧ tmp == x_old}
y = tmp;
{x == y_old ∧ y == tmp ∧ tmp == x_old}
```

```
{x == x_old ∧ y == y_old}
tmp = x;
{x == x_old ∧ y == y_old ∧ tmp == x_old}
x = y;
{x == y ∧ y == y_old ∧ tmp == x_old}
y = tmp;
{x == y_old ∧ y == tmp ∧ tmp == x_old}
{x == y_old ∧ y == x_old}
```

252-0027
Einführung in die Programmierung

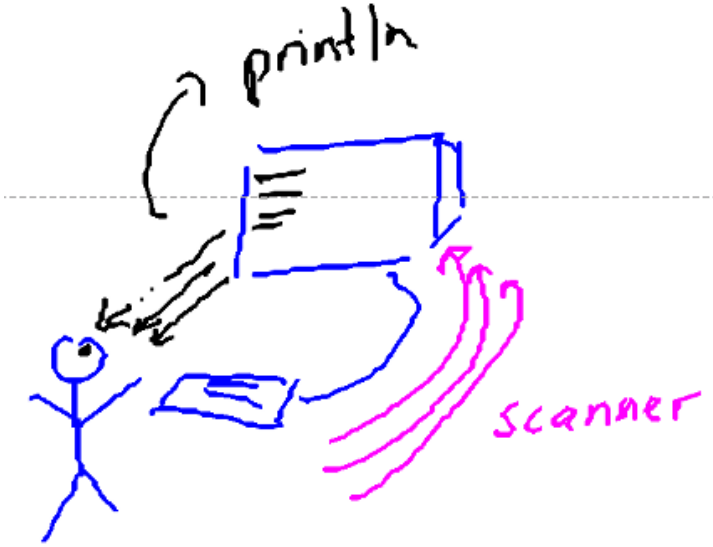
2. X Input

Thomas R. Gross

Department Informatik
ETH Zürich

Interaktive Programme mit `Scanner`

- **Einfache interaktive Programme ...**
- **Program liest Benutzer Input**
  - Text – Eingabe durch Tastatur
  - Lesen mit «Scanner»
  - (Oft mit) Ausgabe von Text auf Monitor/Bildschirm mit `println(..)`
- **Interaktiv: Programm kann auf Benutzer Input *reagieren***



Einfache Eingabe (Input)

- **Interaktives Programm: Liest Input von der Konsole («console») oder Terminal.**
  - Während das Programm ausgeführt wird fragt das Programm den Benutzer
    - Benutzer tippt Input ein
  - Der vom Benutzer eingegebene Input wird durch Anweisungen im Programm in Variablen gespeichert
- **Später auch andere Arten des Inputs (von anderen Fenstern)**

Konsole und `System.in`

- **Konsole («console»): früher ein Gerät, das mit Computer verbunden war**
  - Erlaubte Eingabe (manchmal auch Ausgabe)

Konsole und `System.in`

- **Konsole («console»): früher ein Gerät das mit Computer verbunden war und Start/Management des Computers zulies**
- **Heute: (meist) Fenster im (graphischen) Benutzerinterface**
  - `System.in` (Standard Input): Ein (vordefiniertes) Fenster für Input
  - `System.out` (Standard Output): Ein (vordefiniertes) Fenster für Output.
- `System.in` und `System.out` können dasselbe Fenster sein.

Input und Output

- **Input komplizierter als Output**
  - Benutzer können sich auf unvorgesehene Weise verhalten
  - Benutzer können Fehler machen
  - Benutzer können unpassenden Input liefern
  - Aber interaktive Programme sind viel interessanter …..
- **Eclipse verwendet ein Fenster um `System.in` und `System.out` zu zeigen:**
  - Output zu `System.out`
  - Keyboard Input von `System.in`

Input für Programm

- **Müssen Input lesen**
  - Umwandlung von Darstellung der Konsole/des Terminals in Format, das vom Java Programm verarbeitet werden kann
  - Idealerweise wird diese (Basis)Software zur Verfügung gestellt
- `Scanner` **bietet solche Services an**
  - Es gibt auch andere Möglichkeiten aber Scanner ist (relativ) einfach und erleichtert unsere Arbeit.

- **Output hat das selbe Problem (Umwandlung der Darstellung) – auch dafür gibt es bereits Software (`println(..)`)**

Input und `System.in`

- **Idee: Benutzer tippt Eingabe (via Tastatur) und Programm liest diese von `System.in` mittels eines `Scanner`s**
- `Scanner`: **erlaubt es Input von unterschiedlichen Quellen zu lesen**
  - Kann von verschiedenen Stellen lesen
    - Auch von `System.in`
    - Datenquelle wird angegeben wenn `Scanner` konstruiert wird
  - Später auch von Dateien, Web Seiten, Datenbanken, … lesen

`Scanner` **Syntax**

- `Scanner` **sind in der Bibliothek** `java.util` **definiert**
  - Muss erst bekannt gegeben werden
  - `import java.util.Scanner;` // so dass wir Scanner benutzen können
- **Programm braucht ein** `Scanner` **Objekt um von der Konsole zu lesen:**
  - Dieses muss konstruiert werden

    `Scanner` name = `new Scanner(System.in);`
- **Beispiel:**

```
Scanner myConsole = new Scanner(System.in);
```

`Scanner` **Methoden**

Method	Description
`nextInt()`	reads an `int` from the user and returns it
`nextDouble()`	reads a `double` from the user
`next()`	reads a one-word `String` from the user
`nextLine()`	reads a one-*line* `String` from the user

Aufruf einer Methode braucht einen Scanner

```
Scanner myConsole = new Scanner(System.in);
```

und dann Aufruf in Punktnotation («dot notation»):

```
int alter = myConsole.nextInt();
```

`Scanner` **Methoden**

Method	Description
`nextInt()`	reads an `int` from the user and returns it
`nextDouble()`	reads a `double` from the user
`next()`	reads a one-word `String` from the user
`nextLine()`	reads a one-*line* `String` from the user

- Der eingegebene Wert kann weiter verarbeitet werden.

```
int alter = myConsole.nextInt();
System.out.println("Ihre Eingabe " + alter);
```

- **Aber *wie* weiss der Benutzer dass eine Eingabe erwartet wird?**

Scanner **Eingabe**

- Das Programm fordert den Benutzer auf, Wert(e) einzugeben
  - «prompt» (Aufforderung): Text der angibt das/welche Eingabe erwartet wird
- Der Prompt erscheint im Konsolenfenster
  - Jetzt kann die Eingabe erfolgen – erscheint auch im selben Fenster
  - `System.in` und `System.out` werden von Eclipse besonders behandelt
- Jede Scanner Methode wartet bis der Benutzer die Eingabe mit der «ENTER» (oder «RETURN») Taste abschliesst.
- Beispiel:

```
System.out.print("Wie alt sind Sie? ");  // prompt
int alter = myConsole.nextInt();
System.out.println("Ihre Eingabe " + alter);
```

Scanner **Beispiel**

```
import java.util.Scanner;    // so dass Scanner verwendet werden kann

public class UserInputExample {
    public static void main(String[] args) {
        Scanner myConsole = new Scanner(System.in);


        System.out.print("Wie alt sind Sie? ");
        int alter = myConsole.nextInt();


        int jahre = 67 - alter;
        System.out.println(jahre  + " Jahre bis zur Pensionierung!");
    }
}
```

alter	19
jahre	48

Konsole (Eingabe des Benutzers unterstrichen):

```
Wie alt sind Sie?       19
48 Jahre bis zur Pension!
```

Scanner **Beispiel 2**

```
import java.util.*;
public class ScannerMultiply {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Please type two numbers: ");
        int num1 = console.nextInt();
        int num2 = console.nextInt();

        int product = num1 * num2;
        System.out.println("The product is " + product);
    }
}
```

Output (Benutzereingabe unterstrichen):

```
Please type two numbers: 8 6
The product is 48
```

- Der `Scanner` kann mehrere Zahlen in einer Zeile (endet mit RETURN) lesen.

Wo finde ich Informationen

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Scanner.html

- **Dort finden wir mehr Informationen**
  - Z.Zt. noch nicht alle Konzepte erklärt
  - Darum «ignorieren» wir die andere Aspekte (ausser denen, die wir für das Beispiel brauchen)

nextInt

```
public int nextInt()
```

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

`the int scanned from the input`

Eingabe Elemente

- `Scanner` **liest (für** `nextInt()`**) optionales Vorzeichen und Folge von Ziffern und wandelt diese in eine Zahl (**`int`**) um**
  - Braucht Beschreibung von legalen `int`s
  - Z.B. EBNF Beschreibung
  - Muss wissen wo eine `int` Beschreibung aufhört (anfängt)
  - Beschreibung von Leerzeichen/Trennzeichen
    - Auch wieder mit EBNF möglich
- **Folge von Zeichen die der** `Scanner` **liest: Token**
  - Input Element
  - Erwartete Zeichen hängen von Methode (z.B. `nextDouble()`) ab

Eingabe Elemente

- **Folge von Zeichen die der Scanner liest: Token**
  - Token werden durch *Zwischenraum* («*whitespace*») getrennt
    - Auf Deutsch: Leerzeichen, auf Englisch: «space», «blank»
    - Tabulator Zeichen («tab»),
    - Zeilenvorschub («new line»).
  - Wie viele Token sind in dieser Zeile?

```
23  John Smith   42.0  $2.50
```

Eingabe Elemente

Wenn das Token nicht den richtigen Typ hat gibt es (zur Laufzeit) eine Fehlermeldung.

```
System.out.print("Was ist Ihr Alter? ");
int alter = myConsole.nextInt();
```

Output:

```
Was ist Ihr Alter? Timmy
java.util.InputMismatchException
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        ...
```

Noch ein Beispiel

```
import java.util.Scanner;   // for Scanner
public class IncNumber {
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("Type a number: ");
    int numberIn = console.nextInt();
    System.out.println("Number + 1: " + (1 + numberIn));
    }
}
```

- `Scanner`**'s** `next` **Methode liest ein Wort (d.h. keine Zwischenräume) als** `String`**.**

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
name = name.toUpperCase();
System.out.println(name + " has " + name.length() +
    " letters and starts with " + name.substring(0, 1));
```

- **Beispiele:** (Input unterstrichen)

```
What is your name? Chamillionaire
CHAMILLIONAIRE has 14 letters and starts with C
What is your name? Donald Duck
DONALD has 6 letters and starts with D
What is your name? "Mickey Mouse"
"MICKEY has 7 letters and starts with "
```

`Scanner` **Methoden**

Method	Description
`nextInt()`	reads an `int` from the user and returns it
`nextDouble()`	reads a `double` from the user
`next()`	reads a one-word `String` from the user
`nextLine()`	reads a one-*line* `String` from the user

- Jede Methode wartet bis der Benutzer die Eingabe mit der "ENTER" (oder "RETURN") Taste abschliesst.
- Die `next()` Methode liest ein Wort (d.h. keine Zwischenräume) als `String`
  - Keine Anführungszeichen (")!