

**252-0027**

**Einführung in die Programmierung**  
**Übungen**

**Woche 8: Klassen, Loop-Invarianten**

Timo Baumberger

**Departement Informatik**

**ETH Zürich**

# Organisatorisches



- Mein Name: Timo Baumberger
- Bei Fragen: [tbaumberger@student.ethz.ch](mailto:tbaumberger@student.ethz.ch)  
(*Discord: troxhi*)
  - Mails bitte mit «[EProg25]» im Betreff
- Meine Website: [timobaumberger.com](http://timobaumberger.com)
- Neue Aufgaben: **Dienstag Abend** (im Normalfall)
- Abgabe der Übungen bis **Dienstag Abend (23:59)** Folgewoche
  - Abgabe immer via Git
  - Lösungen in separatem Projekt auf Git

# Programm

- **Bonusaufgabe u07**
- **Loop Invarianten**
- **Java References**
- **Klassen / Objekte (objekt orientiertes Programmieren)**
- **Vorbesprechung**
- **Nachbesprechung**
- **Kahoot**

# **Bonusaufgabe u07**

# The beginning...

```
public static int sumLevel(int[][] matrix, int level) { 26 usages
    if (matrix == null || matrix.length == 0 || !isMatrix(matrix)) return -2;
    int n = matrix.length;
    // PRE: n >= 1
    int maxLevel = (n - 1) / 2;
    // POST: (n % 2 == 1) => (maxLevel >= 0 && maxLevel <= (n - 1) / 2)
    //        (n % 2 == 0) => (maxLevel >= 0 && maxLevel <= n / 2 - 1)

    if (level > maxLevel || level < 0) return -2;
    int levelOffset = maxLevel - level;
    int rim = n - levelOffset;
```

# Ez, gg!

```
int result = 0;
for (int i = levelOffset; i < rim; i++) {
    for (int j = levelOffset; j < rim; j++) {
        if (i == levelOffset || i == rim - 1 || j == levelOffset || j == rim - 1) {
            result += matrix[i][j];
        }
    }
}
return result;
```

# Nice...

```
// O(n^2) but easy to understand :)
int outerSquare = sumSquare(matrix, levelOffset, rim);
int innerSquare = sumSquare(matrix, levelOffset: levelOffset + 1, rim: rim - 1);
int result = outerSquare - innerSquare;
return result;
```

```
private static int sumSquare(int[][] matrix, int levelOffset, int rim) { 2 usages
    int result = 0;
    for (int i = levelOffset; i < rim; i++) {
        for (int j = levelOffset; j < rim; j++) {
            result += matrix[i][j];
        }
    }
    return result;
}
```

# Let's see Paul Allen's code

```
// O(n) solution ;^)
int result = 0;
for (int i = levelOffset; i < rim; i++) {
    result += matrix[levelOffset][i];
    // we have to keep the case where level = 0 in mind
    // otherwise we also add the horizontal border from the bottom
    if (levelOffset != rim - 1) {
        result += matrix[rim - 1][i];
    }
}

for (int i = levelOffset + 1; i < rim - 1; i++) {
    result += matrix[i][levelOffset] + matrix[i][rim - 1]; // left and right border
}

return result;
```



# Enhanced For Loop

- Seit 2004 in Java (<https://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html>)

```
int[] a = new int[] {1, 2, 3, 4};  
  
for (int i = 0; i < a.length; i++) {  
    System.out.println(a[i]);  
}
```

```
for (int elementOfA : a) {  
    System.out.println(elementOfA);  
}
```

# **Loop - Invarianten**

# Loop Invarianten

$$\frac{\{ b \wedge \mathbf{P} \} s \{ \mathbf{P} \}}{\{ \mathbf{P} \} \text{ while } b \text{ do } s \text{ end } \{ \neg b \wedge \mathbf{P} \}} \quad (\text{W}_{\text{H}_{Ax}})$$

```
public int compute(int a, int b) {  
    // Precondition:  a >= 0  
    int x;  
    int res;  
  
    x = a;  
    res = b;  
    // Loop-Invariante: ??  
    while (x > 0) {  
        x = x - 1;  
        res = res + 1;  
    }  
    // Postcondition:  res == a + b  
    return res;  
}
```

## Loop-Invarianten

1. Die Loop-Invariante ist **vor der Schleife** erfüllt.
2. Die Loop-Invariante ist **nach jeder Ausführung** des while – body erfüllt.
3. Die Loop-Invariante ist **nach der Schleife** erfüllt.

```
public int compute(int a, int b) {  
    // Precondition:  a >= 0  
    int x;  
    int res;  
  
    x = a;  
    res = b;  
    // Loop-Invariante: ??  
    while (x > 0) {  
        x = x - 1;  
        res = res + 1;  
    }  
    // Postcondition:  res == a + b  
    return res;  
}
```

## Loop-Invarianten

1. Die Loop-Invariante ist **vor der Schleife** erfüllt.
2. Die Loop-Invariante ist **nach jeder Ausführung** des while – body erfüllt.
3. Die Loop-Invariante ist **nach der Schleife** erfüllt.

**Wir wollen das folgende Hoare Triple beweisen:**

```
{ Precondition }  
  while ( Condition ) { Body };  
{ Postcondition }
```

```

public int compute(int a, int b) {
    // Precondition:  a >= 0

    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

**Wir wollen das folgende Hoare Triple beweisen:**

```

{ Precondition }
  while ( Condition ) { Body };
{ Postcondition }

```

**Dies können wir tun, falls eine Invariante existiert, für welche folgendes gilt:**

1.  $\text{Precondition} \Rightarrow \text{Invariante}$
2.  $\{ \text{Condition} \wedge \text{Invariante} \}$   
 $\text{Body};$   
 $\{ \text{Invariante} \}$  ist ein valides Tripel.
3.  $\neg \text{Condition} \wedge \text{Invariante} \Rightarrow \text{Postcondition}$

}

## Herleitungsbeispiel mit Tabelle

[illegible]

}

## Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a



```

public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1

```

public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2

```

public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3

```

public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3
...	...	...

```

public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3
...	...	...
i	b + i	a - i

```

public int compute(int a, int b) {
    // Precondition:  a >= 0
    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	res	x
0	b	a
1	b + 1	a - 1
2	b + 2	a - 2
3	b + 3	a - 3
...	...	...
i	b + i	a - i

LI:  $\text{res} == b + a - x$

```

public int compute(int a, int b) {
    // Precondition:  a >= 0

    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

1.  $\text{Precondition} \Rightarrow \text{Invariante}$
2.  $\{ \text{Condition} \wedge \text{Invariante} \}$   
 $\text{Body};$   
 $\{ \text{Invariante} \}$  ist ein valides Tripel.
3.  $\neg \text{Condition} \wedge \text{Invariante} \Rightarrow$   
 $\text{Postcondition}$

LI:  $\text{res} == \text{b} + \text{a} - \text{x}$

Reicht das?

```

public int compute(int a, int b) {
    // Precondition:  a >= 0

    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

1. **Precondition**  $\Rightarrow$  **Invariante** ✓

2. { **Condition**  $\wedge$  **Invariante** }  
     **Body**;  
     { **Invariante** } ist ein valides Tripel. ✓

3.  $\neg$  **Condition**  $\wedge$  **Invariante**  $\Rightarrow$   
     **Postcondition** ✗

LI:  $\text{res} == b + a - x$

Reicht das?



```

public int compute(int a, int b) {
    // Precondition:  a >= 0

    int x;
    int res;

    x = a;
    res = b;
    // Loop-Invariante: ??
    while (x > 0) {
        x = x - 1;
        res = res + 1;
    }
    // Postcondition:  res == a + b
    return res;
}

```

1. Precondition  $\Rightarrow$  Invariante ✓

2. { Condition  $\wedge$  Invariante }  
     Body;  
     { Invariante } ist ein valides Tripel. ✓

3.  $\neg$  Condition  $\wedge$  Invariante  $\Rightarrow$   
     Postcondition ✓

res == b + a - x && x >= 0

## **Beispiel 2**



```
public static int factorial(int n) {  
    // Precondition:  $n \geq 0$   
    int counter;  
    int result;  
  
    counter = n;  
    result = 1;  
    // Loop-Invariante: ??  
    while (counter > 0) {  
        result = result * counter;  
        counter = counter - 1;  
    }  
  
    // Postcondition: result =  $n!$   
    return result;  
}
```

## Herleitungsbeispiel mit Tabelle

[illegible]

```

public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n - 1

```

public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n - 1
2	$n * (n - 1)$	n - 2

```

public static int factorial(int n) {
    // Precondition:  $n \geq 0$ 
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result =  $n!$ 
    return result;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	$n$
1	$n$	$n - 1$
2	$n * (n - 1)$	$n - 2$
3	...	$n - 3$

```

public static int factorial(int n) {
    // Precondition:  $n \geq 0$ 
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result =  $n!$ 
    return result;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	$n$
1	$n$	$n - 1$
2	$n * (n - 1)$	$n - 2$
3	...	$n - 3$
...	...	...



```

public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n - 1
2	$n * (n - 1)$	n - 2
3	...	n - 3
...	...	...
i	$n! / (n - i)!$	n - i

```

public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}

```

## Herleitungsbeispiel mit Tabelle

Iteration	result	counter
0	1	n
1	n	n - 1
2	$n * (n - 1)$	n - 2
3	...	n - 3
...	...	...
i	$n! / (n - i)!$	n - i

LI:  $\text{result} == n! / \text{counter}!$

```

public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}

```

1. Precondition  $\Rightarrow$  Invariante ✓

2. { Condition  $\wedge$  Invariante }  
 Body;  
 { Invariante } ist ein valides Tripel. ✓

3.  $\neg$  Condition  $\wedge$  Invariante  $\Rightarrow$   
 Postcondition ✗

LI: result == n! / counter!

Reicht das?

```

public static int factorial(int n) {
    // Precondition: n >= 0
    int counter;
    int result;

    counter = n;
    result = 1;
    // Loop-Invariante: ??
    while (counter > 0) {
        result = result * counter;
        counter = counter - 1;
    }

    // Postcondition: result = n!
    return result;
}

```

1. Precondition  $\Rightarrow$  Invariante ✓

2. { Condition  $\wedge$  Invariante }  
 Body;  
 { Invariante } ist ein valides Tripel. ✓

3.  $\neg$  Condition  $\wedge$  Invariante  $\Rightarrow$   
 Postcondition ✓

LI: result == n! / counter! &&  
 counter >= 0

```

public int compute(int v, int n) {
    // Precondition: n >= 0
    int v = 2;
    int x;
    int tmp;

    x = 1;
    tmp = 1;
    // Loop-Invariante: ??
    while (x <= n) {
        tmp = tmp * v;
        x = x + 1;
    }
    // Postcondition: tmp == 2^n
    return tmp;
}

```

**Wir wollen das folgende Hoare Triple beweisen:**

```

{ Precondition }
  while ( Condition ) { Body };
{ Postcondition }

```

**Dies können wir tun, falls eine Invariante existiert, für welche folgendes gilt:**

1.  $\text{Precondition} \Rightarrow \text{Invariante}$
2.  $\{ \text{Condition} \wedge \text{Invariante} \}$   
 $\text{Body};$   
 $\{ \text{Invariante} \}$  ist ein valides Tripel.
3.  $\neg \text{Condition} \wedge \text{Invariante} \Rightarrow \text{Postcondition}$

```

public int compute(int v, int n) {
    // Precondition: n >= 0
    int v = 2;
    int x;
    int tmp;

    x = 1;
    tmp = 1;
    // Loop-Invariante: ??
    while (x <= n) {
        tmp = tmp * v;
        x = x + 1;
    }
    // Postcondition: tmp == 2^n
    return tmp;
}

```

1.  $\text{Precondition} \Rightarrow \text{Invariante}$
2.  $\{ \text{Condition} \wedge \text{Invariante} \}$   
 $\text{Body};$   
 $\{ \text{Invariante} \}$  ist ein valides Tripel.
3.  $\neg \text{Condition} \wedge \text{Invariante} \Rightarrow$   
 $\text{Postcondition}$

LI:  $v == 2 \ \&\& \ tmp == v^{(x - 1)}$   
 $\&\& \ x \leq n + 1 \ \&\& \ x \geq 1$

# **Rezept für Loop-Invarianten**

# Es gibt keine perfekten Rezepte!



- Loop-Invarianten richtig lösen können ist eine **Frage der Übung**.
- Das folgende Rezept hat sich in der Vergangenheit bei vielen Studenten als nützlich erwiesen.



```
public int compute(int n){  
    // Precondition:  $n \geq 0$   
    int k = 5;  
    int i = 0;  
    int result = 1;  
  
    // Loop-Invariante: ??  
    while ( $i < n$ ) {  
        result = result * k;  
        i = i + 1;  
    }  
    //Postcondition  $result == 5^n$   
    return result;  
}
```

1. Loop Condition und Terminierung kombinieren.

$$\{i \leq n\}$$

2. Postcondition und Loop Body kombinieren.

$$\{result == 5^n\} \rightarrow \{result == 5^i\}$$

3. Conditions wegen benutzter Methoden oder mathematischer Formeln.

$$\{k == 5\}$$

$$\{i \leq n \ \&\& \ result == k^i \ \&\& \ k == 5 \}$$

# Klassen

# Java Access Modifiers

- **Private** (innerhalb der Klasse sichtbar)
- **Public** (überall sichtbar)
- **Default** (in Klassen im gleichen Package sichtbar)
- **Protected** (in Klassen im gleichen Package sichtbar und noch mehr)

# Probleme Lösen: Zeitspanne

In dieser Aufgabe wollen wir eine Klasse Zeitspanne erstellen. Objekte dieser Klasse speichern eine Zeitdauer (z.B. 3 Stunden, 45 Minuten) und geben diese auf Anfrage zurück.

1. Erstellen Sie die Klasse Zeitspanne. Diese Klasse erstellt Objekte welche eine Zeitspanne darstellen (in Stunden und Minuten, Sekunden sind nicht verlangt).
2. Erstellen Sie passende Methoden für diese Klasse. Wir sollten die Zeitspanne verändern können (setzen auf einen beliebigen (sinnvollen) Wert, addieren und subtrahieren), sie in Minuten zurückgeben können und drucken können.
3. Schreiben Sie nun eine Methode 'isMoreThan()', die eine zweite Zeitspanne als Argument nimmt und es ermöglicht zwei Zeitspannen zu vergleichen.

```
public class Zeitspanne {  
    public int minutes;  
    public int hours;  
}
```

1. Erstellen Sie die Klasse Zeitspanne. Diese Klasse erstellt Objekte welche eine Zeitspanne darstellen (in Stunden und Minuten, Sekunden sind nicht verlangt).

```
public class Zeitspanne {  
    public int minutes;  
    public int hours;  
}
```

Ist nicht notwendig...

94 Minuten  $\simeq$  1 Stunde und 34 Minuten

1. Erstellen Sie die Klasse Zeitspanne. Diese Klasse erstellt Objekte welche eine Zeitspanne darstellen (in Stunden und Minuten, Sekunden sind nicht verlangt).



```
public class Zeitspanne {  
    private int minutes;
```

```
    // Getter Methoden
```

```
    public int getTimeInMin() {  
        return this.minutes;  
    }
```



Ein Benutzer würde sicherlich auch die Anzahl  
Stunden wissen wollen.

```
    public int getFullHours() {  
        return this.minutes / 60;  
    }
```

```
}
```



```
public class Zeitspanne {  
    private int minutes;  
  
    // Getter Methoden  
    public int getTimeInMin() {  
        return this.minutes;  
    }  
  
    public int getFullHours() {  
        return this.minutes / 60;  
    }  
  
}
```

(...)

```
public int getFullHours() {  
    return this.minutes / 60;  
}
```

```
// Setter Methoden  
public void setMinutes(int minutes) {  
    this.minutes = minutes;  
}
```



Soll ein Benutzer (indirect) Zugriff auf das minutes Attribut haben?

Hier: Ja

**Dies ist aber oft nicht der Fall!**

}

2. Erstellen Sie passende Methoden für diese Klasse. Wir sollten die Zeitspanne verändern können (setzen auf einen beliebigen (sinnvollen) Wert, addieren und subtrahieren), sie in Minuten zurückgeben können und drucken können. Wenn der user unpassende



(...)

```
// Setter Methoden
```

```
public void setMinutes(int minutes) {  
    this.minutes = minutes;  
}
```

```
public void setTime(int hours, int minutes) {  
    this.minutes = hours * 60 + minutes;  
}
```

```
}
```

Methoden für **jede** Klasse:

❑ Konstruktoren

❑ toString Methode

Später mehr...

2. Erstellen Sie passende Methoden für diese Klasse. Wir sollten die Zeitspanne verändern können (setzen auf einen beliebigen (sinnvollen) Wert, addieren und subtrahieren), sie in Minuten zurückgeben können und drucken können. Wenn der user unpassende

```
public class Zeitspanne {  
    private int minutes;
```

```
  
    public Zeitspanne() {  
        this.minutes = 0;  
    }
```

```
  
    public Zeitspanne(int hours, int minutes) {  
        this.minutes = minutes + hours * 60;  
    }
```

```
  
    (...)   
}
```

## Konstruktoren:

- ❑ Default Konstruktor

- ❑ Parametrisierter Konstruktor

```
public class Zeitspanne {  
    private int minutes;  
  
    (...)   
  
    public Zeitspanne(int hours, int minutes) {  
        this.minutes = minutes + hours * 60;  
    }  
  
    public Zeitspanne(int minutes) {  
        this.minutes = minutes;  
    }  
  
    public Zeitspanne(Zeitspanne other) {  
        this.minutes = other.minutes;  
    }  
  
    (...)   
}
```

## Konstruktoren:

- ❑ Default Konstruktor
- ❑ Parametrisierte Konstruktor(en)
- ❑ Copy Konstruktor

Wieso?

Wir erstellen ein Objekt mit einem anderen Objekt als Vorlage.

```
public class Zeitspanne {  
    private int minutes;  
  
    (...)  
    public Zeitspanne(int hours, int minutes) {  
        this.minutes = minutes + hours * 60;  
    }  
  
    public Zeitspanne(int minutes) {  
        this.minutes = minutes;  
    }  
  
    public Zeitspanne(Zeitspanne other) {  
        this.minutes = other.minutes;  
    }  
  
    (...)  
}
```

Sehr repetitiv...

```
public class Zeitspanne {  
    private int minutes;  
    public Zeitspanne(int minutes) {  
        this.minutes = minutes;  
    }  
    public Zeitspanne() {  
        this(0);  
    }  
    public Zeitspanne(int hours, int minutes) {  
        this(minutes + hours * 60);  
    }  
    public Zeitspanne(Zeitspanne other) {  
        this(other.minutes);  
    }  
    (...)  
}
```



Einen parametrisierten Konstruktor.  
Alle anderen Konstruktoren rufen diesen auf.



```
public class Zeitspanne {  
    private int minutes;  
    public Zeitspanne(int minutes) {  
        this.minutes = minutes;  
    }  
    public Zeitspanne() {  
        this(0);  
    }  
    public Zeitspanne(int hours, int minutes) {  
        this(minutes + hours * 60);  
    }  
    public Zeitspanne(Zeitspanne other) {  
        this(other.minutes);  
    }  
    (...)  
}
```

```
public class Zeitspanne {  
    (...)  
    public void setTime(int hours, int minutes) {  
        this.minutes = hours * 60 + minutes;  
    }  
  
    public String toString() {  
        int hours = this.minutes / 60;  
        int minutes = this.minutes % 60;  
  
        return hours + ":" + minutes;  
    }  
}
```

Methoden für **jede** Klasse:

❑ Konstruktoren

❑ toString Methode



```
public class Zeitspanne {  
    (...)  
    public int[] toHourMin() {  
        int hours = this.minutes / 60;  
        int minutes = this.minutes % 60;  
  
        return new int[] {hours, minutes};  
    }  
  
    public String toString() {  
        int[] split = toHourMin();  
        return split[0] + ":" + split[1];  
    }  
  
}
```

Wir werden diese Rechnungen einige Male machen...

**Wir schreiben eine Methode!**



```
public class Zeitspanne {  
    (...)  
    public int getMinutes(int hours, int minutes) {  
        return hours * 60 + minutes;  
    }  
    public Zeitspanne(int hours, int minutes) {  
        this(minutes + hours * 60);  
    }  
    public void setTime(int hours, int minutes) {  
        this.minutes = hours * 60 + minutes;  
    }  
  
}
```

Wir werden diese Rechnungen einige Male machen...

Wir schreiben eine Methode!

```
public class Zeitspanne {  
    (...)  
    public int getMinutes(int hours, int minutes) {  
        return hours * 60 + minutes;  
    }  
    public Zeitspanne(int hours, int minutes) {  
        this(getMinutes(hours, minutes));  
    }  
    public void setTime(int hours, int minutes) {  
        this.minutes = getMinutes(hours, minutes);  
    }  
  
}
```

Wir werden diese Rechnungen einige Male machen...

Wir schreiben eine Methode!

**Does this work?**

```
public class Zeitspanne {  
    (...)  
    public static int getMinutes(int hours, int minutes) {  
        return hours * 60 + minutes;  
    }  
    public Zeitspanne(int hours, int minutes) {  
        this(getMinutes(hours, minutes));  
    }  
    public void setTime(int hours, int minutes) {  
        this.minutes = getMinutes(hours, minutes);  
    }  
  
}
```

Wir werden diese Rechnungen einige Male machen...

Wir schreiben eine Methode!

**Yes (static)**



```
public class Zeitspanne {  
    (...)  
    public void addTime(int hours, int minutes) {  
        this.minutes += toMinutes(hours, minutes);  
    }  
  
    public void subTime(int hours, int minutes) {  
        this.minutes -= toMinutes(hours, minutes);  
    }  
}
```

2. Erstellen Sie passende Methoden für diese Klasse. Wir sollten die Zeitspanne verändern können (setzen auf einen beliebigen (sinnvollen) Wert, **addieren und subtrahieren**), sie in Minuten zurückgeben können und drucken können. Wenn der user unpassende

```
public class Zeitspanne {  
    (...)  
    public void addTime(int hours, int minutes) {  
        this.minutes += toMinutes(hours, minutes);  
    }  
  
    public void subTime(int hours, int minutes) {  
        this.minutes -= toMinutes(hours, minutes);  
    }  
}
```

← Was wenn die minutes negativ werden würden?

**Jetzt:** boolean Rückgabewert, der sagt, ob die Operation erfolgreich war.

**Später:** Exceptions / Errors

2. Erstellen Sie passende Methoden für diese Klasse. Wir sollten die Zeitspanne verändern können (setzen auf einen beliebigen **(sinnvollen)** Wert, addieren und subtrahieren), sie in Minuten zurückgeben können und drucken können. Wenn der user unpassende

```
public class Zeitspanne {  
    (...)  
    public void addTime(int hours, int minutes) {  
        this.minutes += toMinutes(hours, minutes);  
    }  
    public boolean subTime(int hours, int minutes) {  
        int minutesToSub = toMinutes(hours, minutes);  
        if(this.minutes >= minutesToSub) {  
            this.minutes -= minutesToSub;  
            return true;  
        }  
        return false;  
    }  
}
```

**Jetzt:** boolean Rückgabewert,  
der sagt, ob die Operation  
erfolgreich war.

**Später:** Exceptions / Errors

2. Erstellen Sie passende Methoden für diese Klasse. Wir sollten die Zeitspanne verändern können (setzen auf einen beliebigen **(sinnvollen)** Wert, addieren und subtrahieren), sie in Minuten zurückgeben können und drucken können. Wenn der user unpassende

```
public class Zeitspanne {  
    (...)  
    public boolean isMoreThan(Zeitspanne other) {  
        return this.getTimeInMin() > other.getTimeInMin();  
    }  
  
}
```

3. Schreiben Sie nun eine Methode 'isMoreThan()', die eine zweite Zeitspanne als Argument nimmt und es ermöglicht zwei Zeitspannen zu vergleichen.

# **StackOverflow Error**

```
public class Factorial {  
  
    public static int factorial(int n) {  
        return n * factorial(n - 1);  
    }  
  
}
```

**Szenario:** Du sollst die Methode `factorial(int n)` schreiben, welche  $n!$  berechnen soll.

1. Du kennst die Formel für  $n!$  und implementierst die Methode.
2. Du führst die Methode aus und siehst folgendes:

[illegible]

```
public class Factorial {  
  
    public static int factorial(int n) {  
        return n * factorial(n - 1);  
    }  
  
}
```

**Szenario:** Du sollst die Methode `factorial(int n)` schreiben, welche  $n!$  berechnen soll.

1. **Du kennst die Formel für  $n!$  und implementierst die Methode.**
2. **Du führst die Methode aus und siehst folgendes:** Exception in thread "main"  
`java.lang.StackOverflowError`
3. **Was macht man jetzt?**
  - **Base Case vergessen?**

```
public class Factorial {  
  
    public static int factorial(int n) {  
        if(n == 0) {  
            return 1;  
        }  
        return n * factorial(n - 1);  
    }  
}
```

**Szenario:** Du sollst die Methode `factorial(int n)` schreiben, welche  $n!$  berechnen soll.

1. **Du kennst die Formel für  $n!$  und implementierst die Methode.**
2. **Du führst die Methode aus und siehst folgendes:** Exception in thread "main"  
`java.lang.StackOverflowError`
3. **Was macht man jetzt?**
  - **Base Case vergessen?** Hier ja!
  - **Oft werden aber zu viele Methoden generiert!**  
Dann muss man sich überlegen, wie man gewisse Methodenaufrufe in der Rekursion verhindern kann.



# **Vorbesprechung**

# Aufgabe 1: Close Neighbors

**In den Testcases hat es einen Fehler! (siehe Webseite für die korrigierten Tests)**

Schreiben Sie ein Programm, welches für eine sortierte Folge  $X$  von `int`-Werten  $(x_1, x_2, \dots, x_n)$  und einen `int`-Wert `key` die drei unterschiedlichen Elemente  $x_a$ ,  $x_b$  und  $x_c$  aus  $X$  zurückgibt, die dem Wert `key` am nächsten sind. Für  $x_a$ ,  $x_b$  und  $x_c$  muss gelten, dass  $|\text{key} - x_a| \leq |\text{key} - x_b| \leq |\text{key} - x_c| \leq |\text{key} - x_i|$  für alle  $i \neq a, b, c$  und dass  $x_a \neq x_b \neq x_c \neq x_a$ . Wenn die drei Werte nicht eindeutig bestimmt sind, dann ist jede Lösung zugelassen, die die obige Bedingung erfüllt.

## Beispiele:

Die nächsten Nachbarn für `key == 5` in  $(1, 4, 5, 7, 9, 10)$  sind 5, 4, 7.

Die nächsten Nachbarn für `key == 5` in  $(1, 4, 5, 6, 9, 10)$  sind 5, 4, 6 oder 5, 6, 4.

Die nächsten Nachbarn für `key == 10` in  $(1, 4, 5, 6, 9, 10)$  sind 10, 9, 6.

Implementieren Sie die Berechnung in der Methode `int[] neighbor(int[] sequence, int key)`, welche sich in der Klasse `Neighbor` befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass das Argument `sequence` nicht `null` ist, sortiert ist, nur unterschiedliche Elemente enthält, und mindestens drei Elemente enthält. Denken Sie daran, dass der Wert `key` nicht unbedingt in der Folge  $X$  auftritt. Sie dürfen das Eingabearray `input` nicht ändern.

In der `main` Methode der Klasse `Neighbor` finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `neighbor`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `NeighborTest.java` geben wir die gleichen Tests zusätzlich auch als JUnit Test zur Verfügung. Sie können diese ebenfalls nach belieben ändern. Es wird *nicht* erwartet, dass Sie für diese Aufgabe den JUnit Test verwenden.

## Aufgabe 2: Loop- Invariante

1. Gegeben sind die Precondition und Postcondition für das folgende Programm

```
public int compute(int n) {  
    // Precondition:  n >= 0  
    int x;  
    int res;  
  
    x = 0;  
    res = x;  
  
    // Loop Invariante:  
    while (x <= n) {  
        res = res + x;  
        x = x + 1;  
    }  
    // Postcondition:  res == ((n + 1) * n) / 2  
    return res;  
}
```

Schreiben Sie die Loop Invariante in die Datei "LoopInvariante.txt".

## Aufgabe 2: Loop- Invariante

2. Gegeben sind die Precondition und Postcondition für das folgende Programm.

```
public int compute(int a, int b) {  
    // Precondition:  a >= 0  
    int x;  
    int res;  
  
    x = 0;  
    res = b;  
  
    // Loop Invariante:  
    while (x < a) {  
        res = res - 1;  
        x = x + 1;  
    }  
    // Postcondition:  res == b - a  
    return res;  
}
```

Schreiben Sie die Loop Invariante in die Datei "LoopInvariante.txt".

## Aufgabe 3: Bills

In dieser Aufgabe sollen Sie einen Teil des Systems implementieren, das für den lokalen Stromversorger die Rechnungen erstellt.

Vervollständigen Sie die `process`-Methode in der Klasse `Bills`. Die Methode hat zwei Argumente: einen `Scanner`, von dem Sie den Inhalt der Eingabedatei lesen sollen, und einen `PrintStream`, in welchen Sie die unten beschriebenen Informationen schreiben.

Ihr Programm muss nur korrekt formatierte Eingabedateien unterstützen. Ein Beispiel einer solchen Datei finden Sie im Projekt unter dem Namen `"Data.txt"`. Exceptions im Zusammenhang mit Ein- und Ausgabe können Sie ignorieren.

Eine valide Eingabedatei enthält Zeilen, die entweder einen Tarif oder den Stromverbrauch eines Kunden beschreiben. Der Verbrauch eines Kunden ist niemals grösser als 100000 Kilowattstunden.

Eine Tarifbeschreibung hat folgendes Format:

$$\text{Tarif\_}n\_l_1p_1 \dots l_np_n$$

## Aufgabe 3: Bills

### Files werden noch genauer besprochen!

- Ihr werdet nur mit einem **PrintStream**-Objekt interagieren müssen mittels der Methode **println**.
- Mit **println** könnt ihr analog wie bei **System.out** eine neue Zeile erstellen.
- Hier wird die neue Zeile nicht in der Konsole erstellt, sondern in der entsprechenden Datei.

## Aufgabe 3: Bills

Folgendes gilt für die Parameter:

- **Tarif** (so geschrieben) ist ein Keyword, das angibt, dass diese Zeile einen Tarif beschreibt.
- $n$  ist eine positive ganze Zahl, welche die Anzahl der Kilowattstunden-Intervalle angibt, für die jeweils ein Strompreis festgelegt ist.
- Auf  $n$  folgt eine Folge von  $n$  Paaren von ganzen Zahlen  $(l_1 \sqcup p_1 \dots l_n \sqcup p_n)$ . Die erste Zahl eines Paares gibt die Obergrenze des Intervalls an und die zweite den Preis für den Verbrauch in diesem Intervall; für ein  $i$  mit  $1 \leq i \leq n$ , ist  $l_i$  also der Verbrauch (in Kilowattstunden), bis zu welchem der Strompreis  $p_i$  (in Rappen pro Kilowattstunde) in Rechnung gestellt wird ( $l_i > 0$  und  $p_i \geq 0$ ). Die Paare sind jeweils mit einem Whitespace voneinander getrennt (und  $l_i$  und  $p_i$  jeweils voneinander auch).

Hier sind einige Beispiele für Tarifbeschreibungen:

1. **Tarif 1** 100000 30  
Es gibt ein Intervall. Für jede Kilowattstunde muss 30 Rappen bezahlt werden.
2. **Tarif 2** 1000 10 100000 30  
Es gibt zwei Intervalle. Die ersten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.
3. **Tarif 3** 100 40 1000 10 100000 30  
Es gibt drei Intervalle. Die ersten 100 Kilowattstunden kosten 40 Rappen pro Kilowattstunde. Die nächsten 1000 Kilowattstunden kosten 10 Rappen pro Kilowattstunde. Der Rest kostet 30 Rappen pro Kilowattstunde.

Wenn ein Kunde im Jahr 2000 Kilowattstunden verbraucht, so beträgt die Rechnung für das erste Beispiel 600 Franken, im zweiten Beispiel 400 Franken und 410 Franken im dritten.

## Aufgabe 3: Bills

Hierbei gilt für die Parameter:

- $ID$  ist eine positive ganze Zahl.
- $v_{q_1}$  ist eine ganze Zahl, die den Verbrauch im ersten Quartal in Kilowattstunden angibt ( $v_{q_1} \geq 0$ ).
- $v_{q_2}$  ist eine ganze Zahl, die den Verbrauch im zweiten Quartal in Kilowattstunden angibt ( $v_{q_2} \geq 0$ ).
- $v_{q_3}$  ist eine ganze Zahl, die den Verbrauch im dritten Quartal in Kilowattstunden angibt ( $v_{q_3} \geq 0$ ).
- $v_{q_4}$  ist eine ganze Zahl, die den Verbrauch im vierten Quartal in Kilowattstunden angibt ( $v_{q_4} \geq 0$ ).

Hier ist ein Beispiel für eine Verbrauchsbeschreibung:

115 0 0 0 2000

Der Kunde mit ID 115 hat nur im vierten Quartal Strom verbraucht. Da waren es 2000 Kilowattstunden.

Ein einmal gelesener Tarif wird für alle Kunden angewendet, die nach dieser Tariffinformation in der Eingabedatei erscheinen. Wenn ein neuer Tarif erscheint, dann gilt dieser bis auf Weiteres für die nachfolgenden Kunden. Sie können davon ausgehen, dass eine Kunden-ID nur einmal in der Eingabedatei vorkommen kann und dass die erste Zeile der Eingabedatei eine Tarifbeschreibung ist.



## Aufgabe 3: Bills

Die Methode `process` soll die Eingabedatei verarbeiten und für jeden Kunden eine Zeile

*ID\_b*

in den `PrintStream` schreiben, der der `process`-Methode in `output` übergeben wird. *ID* ist die ID des Kunden (`int`) und *b* ist eine **ganze** Zahl, die die jeweilige Rechnung für den Jahresverbrauch **in Franken** angibt. (Zuerst muss der Jahresverbrauch berechnet werden, dann kann der entsprechende Tarif angewendet werden.) Berechnen Sie den Rechnungsbetrag und runden Sie das Resultat anschliessend (vor der Ausgabe, aber nach den Berechnungen) auf die nächste ganze Zahl. Sie können hierfür die Methode `Math.round(double a)` verwenden. Die Ausgabe darf keine weiteren Zeichen enthalten. Sie können den Betrag so ausgeben, wie er von der `println`-Anweisung herausgegeben wird, d.h. Sie brauchen das Ergebnis nicht zu formatieren.

In der Datei `"BillsTest.java"` finden Sie einen einfachen Test, um das Format Ihres Outputs zu testen.

**Tipp:** Sie können die Aufgabe ohne weitere Vorgaben implementieren. Wir empfehlen, dass Sie sich überlegen, was sinnvolle Klassen sein könnten und was für Teilaufgaben (die dann als Methode implementiert werden können) zweckmässig sind. Sie können, wie bereits gesehen, das `PrintStream`-Objekt genauso wie `System.out` verwenden und mit `println` eine neue Zeile zur Zieldatei hinzufügen.

## Aufgabe 4: Minesweeper (Bonus)

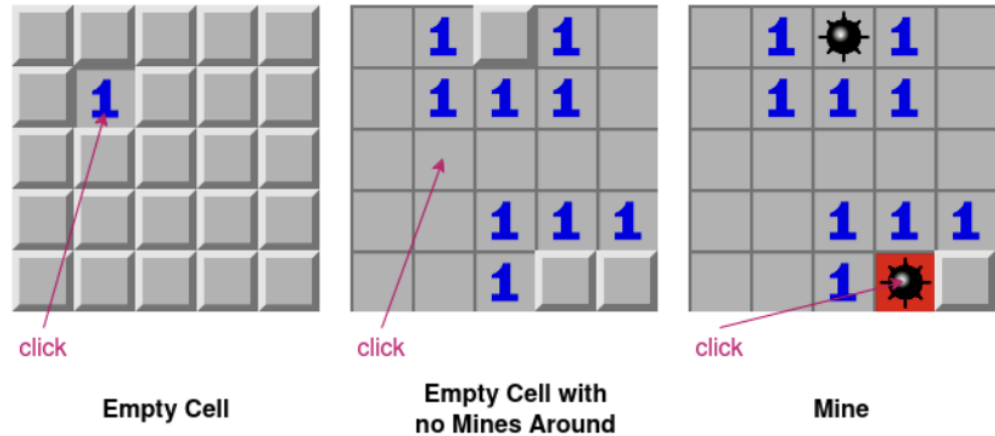


Abbildung 1: Spielbretter nach dem ersten, zweiten und dritten Klick von links nach rechts.

**Achtung:** Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im [www.vvz.ethz.ch](http://www.vvz.ethz.ch)). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich [die allgemeinen Regeln](#).

# **Nachbesprechung**

## Aufgabe 1: Dreiecksmatrix

Die Klasse `Triangle` erlaubt die Darstellung von  $Z \times S$  Dreiecksmatrizen (von `int` Werten).  $Z$  und  $S$  sind immer strikt grösser als 1 (d.h.,  $> 1$ ). Eine  $Z \times S$  Dreiecksmatrix hat  $Z$  Zeilen  $X_0, X_1, \dots, X_{Z-1}$ , wobei Zeile  $X_i$  genau  $(i * (S - 1) / (Z - 1)) + 1$  viele Elemente hat. Dieser Ausdruck wird nach den Regeln für `int` Ausdrücke in Java ausgewertet. Für eine Dreiecksmatrix  $D$  ist  $D_{i,j}$  das  $(j + 1)$ -te Element in der  $(i + 1)$ -ten Zeile.  $D_{0,0}$  ist das erste Element in der ersten Zeile [die immer genau 1 Element hat]. Abbildung 1 zeigt Beispiele von Dreiecksmatrizen. Beachten Sie, dass es möglich ist, dass zwei (aufeinanderfolgende) Zeilen die selbe Anzahl Elemente haben.

0,0		0,0		0,0
1,0 1,1		1,0 1,1		3,0 3,1 3,2 3,3
2,0 2,1 2,2 2,3		2,0 2,1 2,2		0,0
3,0 3,1 3,2 3,3 3,4		3,0 3,1 3,2 3,3		1,0
4,0 4,1 4,2 4,3 4,4 4,5 4,6				2,0 2,1

Abbildung 1: Beispiele von  $5 \times 7$ ,  $4 \times 4$ ,  $2 \times 4$  und  $3 \times 2$  Dreiecksmatrizen.

## Aufgabe 2: Hoare Tripel

Welche dieser Hoare Tripel sind (un)gültig? Bitte geben Sie für ungültige Tripel ein Gegenbeispiel an. Die Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow.

1.  $\{ x \geq 0 \mid y \geq 0 \} \quad z = x * y; \quad \{ z > 0 \}$
2.  $\{ x > 10 \} \quad z = x \% 10; \quad \{ z > 0 \}$
3.  $\{ x > 0 \} \quad y = x * x; z = y / 2; \quad \{ z > 0 \}$
4.  $\{ x > 0 \} \quad y = x * x; \text{sum} = y \% (x + 1); \{ \text{sum} > 1 \}$
5.  $\{ b > c \}$

```
if (x > b) {  
    a = x;  
} else {  
    a = b;  
}
```

```
{ a > c }
```

## Aufgabe 3: Weakest Precondition

Bitte geben Sie für die folgenden Programmsegmente die schwächste Vorbedingung (weakest precondition) an. Bitte verwenden Sie Java-Syntax. Alle Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ `int` und es gibt keinen Overflow.

1.

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

2.

```
P: { ?? }  
S:  if (x != y) {  
    x = y;  
  } else {  
    x = y + 1;  
  }  
Q: { x != y }
```

## Aufgabe 4: Blackbox Testing

Im letzten Übungsblatt haben Sie Testautomatisierung mit JUnit kennengelernt. In dieser Aufgabe sollen Sie nun Tests für eine Methode schreiben, deren Implementierung Sie nicht kennen. Dadurch werden Sie weniger durch möglicherweise falsche Annahmen beeinflusst, die bei einer Implementierung getroffen wurden. Sie müssen sich also überlegen, wie sich *jede* fehlerfreie Implementierung verhalten muss. Diesen Ansatz nennt man auch **Black-Box Testing** da die Details der Implementation verdeckt sind.

In Ihrem “U06”-Projekt befindet sich eine “blackbox.jar”-Datei, welche eine kompilierte Klasse `BlackBox` enthält. Den Code dieser Klasse können Sie nicht sehen, aber sie enthält eine Methode `void rotateArray(int[] values, int steps)`, welche Sie aus einer eigenen Klasse oder einem Unit-Test aufrufen können. Diese Methode “rotiert” ein `int`-Array um eine gegebene Anzahl Schritte.

Vereinfacht macht die Methode `rotateArray()` Folgendes: Eine Rotation mit `steps=1` bedeutet, dass alle Elemente des Arrays um eine Position nach rechts verschoben werden. Das letzte Element wird dabei zum ersten. Mit `steps=2` wird alles um zwei Positionen nach rechts rotiert, usw. Eine Rotation nach links kann mit einer negativen Zahl für `steps` erreicht werden. Das folgende Beispiel ist der erste, einfache Test, den Sie in der Datei “`BlackBoxTest.java`” finden:

```
int[] values = new int[] { 1, 2 };  
int[] expected = new int[] { 2, 1 };  
BlackBox.rotateArray(values, 1);  
assertArrayEquals(expected, values);
```

# Kahoot

<https://create.kahoot.it/details/9d571905-c760-43e3-a69a-6a1b24508269>



## Beispielaufgabe zu Rekursion

Wir wollen eine Person A bei  $(x, y)$  glücklich machen

Falls diese Person A bereits glücklich ist, müssen wir nichts machen.

Falls diese Person A noch nicht glücklich ist, machen wir diese Person glücklich. Danach gehen wir wie folgt vor:

Falls Person A Leute in ihrem Umfeld hat, die nicht glücklich sind, sorgt die Person A dafür dass die Leute in ihrem Umfeld auch glücklich sind. Dieser Schritt wird rekursiv gemacht. Falls Person A aber Leute in ihrem Umfeld hat, die bereits glücklich sind (eine Person reicht) muss Person A nichts unternehmen.

Java Pseudocode folgt auf der nächsten Seite.

```

public static void beHappy(boolean[] world, int coord1, int coord2) {
    // if somebody is already happy we don't have to make that person happy...
    if (isAlreadyHappy(world, coord1, coord2)) {
        return;
    }
    // usually we have more than one termination condition...

    int happyCount = countHappyPeopleAroundMe(world, coord1, coord2);
    world[coord1][coord2] = true; // make the person at (coord1, coord2) happy
    // other people around the person at (coord1, coord2) are already happy.
    if (happyCount != 0) {
        return;
    }
    for (int d = -5; d <= 5; d++) {
        for (int e = -5; e <= 5; e++) {
            if (d == 0 && e == 0) {
                continue; // The person at (coord1, coord2) is already happy :)
            }
            if (!personAtCoordinatesExists(world, coord1+d, coord2+e)) {
                continue; // There is no person at (coord1+d, coord2+e).
            }

            beHappy(world, coord1+d, coord2+e); // spread happiness ;)
        }
    }
}

```