# 252-0027

# Einführung in die Programmierung

## Übungen

# Invarianten

**Henrik Pätzold**
**Departement Informatik**
**ETH Zürich**

# Nachbesprechung Bonusaufgabe

# Nächste Woche eine kleine Theorie-Simulation

# Weakest Precondition

1. WP: {  }

```
k = m * 3;
Q: { k > 0 }
```

2. WP: {  }

```
x = y * 2;
x = x + 1;
Q: {x > 2 }
```

3. WP: {  }

```
if (a > b) {
  c = (-2) * a;
} else {
  c = a + 4;
}
Q: { c > 0 }
```

$$
\begin{aligned}
(b \implies R_1) \wedge (\neg b \implies R_2) &\equiv (\neg b \vee R_1) \wedge (\neg\neg b \vee R_2) \\
&\equiv (\neg b \vee R_1) \wedge (b \vee R_2) \\
&\equiv ((\neg b \vee R_1) \wedge b) \vee ((\neg b \vee R_1) \wedge R_2)) \\
&\equiv ((\neg b \wedge b) \vee (R_1 \wedge b)) \vee ((\neg b \wedge R_2) \vee (R_1 \wedge R_2)) \\
&\equiv (\bot \vee (R_1 \wedge b)) \vee ((\neg b \wedge R_2) \vee (R_1 \wedge R_2)) \\
&\equiv (R_1 \wedge b) \vee (\neg b \wedge R_2) \vee (R_1 \wedge R_2) \\
&\equiv (R_1 \wedge b) \vee (\neg b \wedge R_2) \vee ((R_1 \wedge R_2) \wedge \top) \\
&\equiv (R_1 \wedge b) \vee (\neg b \wedge R_2) \vee ((R_1 \wedge R_2) \wedge (\neg b \vee b)) \\
&\equiv (R_1 \wedge b) \vee (\neg b \wedge R_2) \vee ((R_1 \wedge R_2) \wedge \neg b) \vee ((R_1 \wedge R_2) \wedge b) \\
&\equiv (R_1 \wedge b) \vee (\neg b \wedge R_2) \vee ((\neg b \wedge R_2) \wedge R_1) \vee ((R_1 \wedge b) \wedge R_2) \\
&\equiv (R_1 \wedge b) \vee ((R_1 \wedge b) \wedge R_2) \vee (\neg b \wedge R_2) \vee ((\neg b \wedge R_2) \wedge R_1) \\
&\equiv ((R_1 \wedge b) \wedge (\top \vee R_2)) \vee ((\neg b \wedge R_2) \wedge (\top \wedge R_1)) \\
&\equiv (R_1 \wedge b) \vee (\neg b \wedge R_2) \\
&\equiv (b \wedge R_1) \vee (\neg b \wedge R_2)
\end{aligned}
$$

# Loop - Invarianten

```
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. Precondition ⇒ Invariante

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.

3. ¬ Condition ∧ Invariante ⇒ 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;
}
```

## Herleitungsbeispiel mit Tabelle

| Iteration | res | 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;
}
```

## 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: 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. Precondition ⇒ Invariante
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.
3. ¬ Condition ∧ Invariante ⇒ Postcondition

LI: 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 ⇒ Invariante ✓
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel. ✓
3. ¬ Condition ∧ Invariante ⇒ Postcondition ✗

LI: 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 ⇒ Invariante ✓
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel. ✓
3. ¬ Condition ∧ Invariante ⇒ Postcondition ✓

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

# Beispiel 2

```
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 |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

```java
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 |
| | | |
| | | |
| | | |
| | | |
| | | |

```java
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 |
| | | |
| | | |
| | | |
| | | |

```java
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 >= 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 |
| | | |
| | | |

```java
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 |
| … | … | … |
| | | |

```java
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 |

```java
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: result == n! / counter!

```java
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 ⇒ Invariante ✓

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel. ✓

3. ¬ Condition ∧ Invariante ⇒ 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 ⇒ Invariante ✓
2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel. ✓
3. ¬ Condition ∧ Invariante ⇒ 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. Precondition ⇒ Invariante

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.

3. ¬ Condition ∧ Invariante ⇒ 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. Precondition ⇒ Invariante

2. { Condition ∧ Invariante }
   Body;
   { Invariante } ist ein valides Tripel.

3. ¬ Condition ∧ Invariante ⇒ Postcondition

LI: v == 2 && tmp == 2^(x - 1)
&& x <= n + 1 && x >= 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){
    // Precondtion: n >= 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.**

$$\{i <= n\}$$

2. **Postcondition und Loop Body kombinieren.**

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

3. **Conditions wegen benutzten Methoden oder mathematischen Formeln.**

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

```
{ i <= n && result == 5^i && k == 5 }
```

# Vorbesprechung
# (test-files neu runterladen)

# Aufgabe 1: Close Neighbors

Schreiben Sie ein Programm, welches für eine sortierte Folge $X$ von `int`-Werten $(x_1, x_2, ..., x_n)$ 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 $|\texttt{key} - x_a| \leq |\texttt{key} - x_b| \leq |\texttt{key} - x_c| \leq |\texttt{key} - x_i|$ 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:

$$\texttt{Tarif}\,␣\,n\,␣\,l_1\,␣\,p_1\,\ldots\,l_n\,␣\,p_n$$

# 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.
- $n$ is 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 ($l_1 \sqcup p_1 \ldots l_n \sqcup p_n$). 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 Rechung 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 Verbrauchbeschreibung:

```
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 Tarifinformation 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)

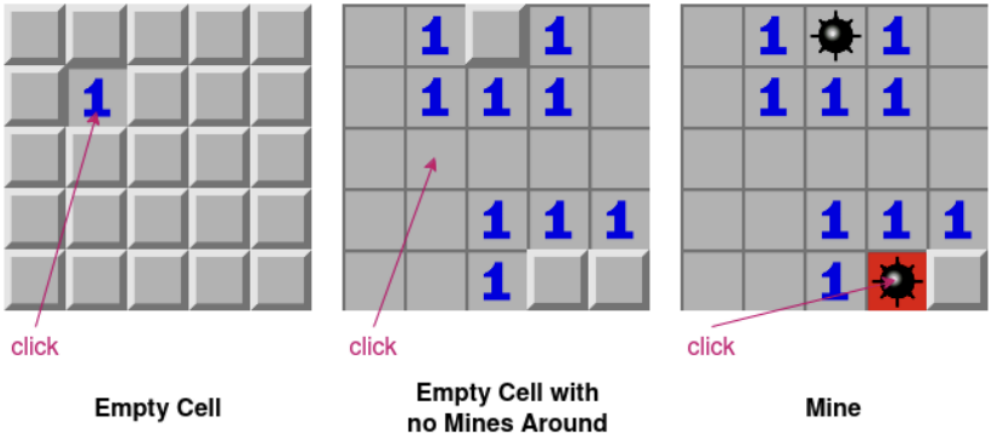


Abbildung 1: Spielbretter nach dem ersten, zweiten und dritten Klick von links nach rechts.

**Achtung:** Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). 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 $Z \times S$ Dreiecksmatrize hat $Z$ Zeilen $X_0, X_1, ..., X_{Z-1}$, 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.

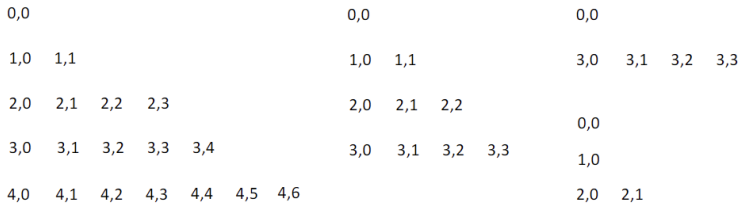


Abbildung 1: Beispiele von $5 \times 7$, $4 \times 4$, $2 \times 4$ und $3 \times 2$ Dreiecksmatrizen.

# 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 >= 0 || y >= 0 }  z = x * y;  { z > 0 }`
2. `{ x > 10 }  z = x % 10;  { z > 0 }`
3. `{ x > 0 }  y = x * x; z = y / 2;  { z > 0 }`
4. `{ x > 0 }  y = x * x; sum = y % (x + 1); { 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);
```