

Best Practices, Debugging & Regex

Programmieren Tutorium Nr.17

Aleksandr Zakharov | 4. Februar 2026

Wiederholung

Wiederholung
●○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

Wiederholung - Rekursion

Welche Aussagen sind korrekt?

1. In einem rekursiven Ansatz braucht man keine Zählvariable. **Nein**
2. Wenn rekursiv eine Endlosschleife entsteht, dann wird diese in einem Timeout enden. **Nein**

Wiederholung
○●○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

Wiederholung - Java API

Welche Datenstruktur (List, Set, Map) verwendet ihr in den folgenden Kontexten?

1. Es soll eine Kundenliste angelegt werden. Jeder Kunde ist über eine einzigartige Nummer definiert, es sollen allerdings noch mehr Daten gespeichert werden können. **Set, Map**
2. Studenten sollen für eine Klausur in Hörsäle verteilt werden. Die Verteilung soll nach Nachname sortiert sein. **List, (Sorted)Set**

Wiederholung - Designprinzipien

Welche Aussagen sind korrekt?

1. Jede Klasse sollte genau eine Verantwortung haben. **Ja**
2. Wird eine Klasse mit Komposition modelliert, lässt sich diese Klasse wie bei Vererbung als Elternklasse ersetzen. **Nein**
3. Code sollte so allgemein wie möglich sein. **Ja**
4. Das ‘Liskov Substitution Principle’ besagt, dass Code offen für Erweiterung, aber geschlossen für Änderung sein sollte. **Nein**

Wiederholung
○○○●

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

Best Practices

Wiederholung
○○○○

Best Practices
●○○○○○○○○

Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

Best Practices

■ Java-spezifische Tipps

- instanceof
- == und equals()
 - equals() überschreiben

Wiederholung
○○○○

Best Practices
○●○○○○○○○○

Debugging & Testing
○○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

instanceof

- Oft falsch verwendet

- “Wenn Klasse A, dann tue a(), wenn Klasse B, tue b().”

```
Object o;  
  
if (o instanceof ClassA) {  
    doSomething();  
  
} else if (o instanceof ClassB) {  
    doDifferentThing();  
  
} else {  
    doDefaultThing();  
}
```

```
abstract class Better { do(); }  
// oder interface  
class ClassA extends Better { ... }  
class ClassB extends Better { ... }
```

```
Better o;  
o.do();
```

Besser

Unterschiede zwischen == und equals()

	primitive Datentypen	Objekte
<code>==</code>	Vergleicht den Wert in der Speicherzelle Vergleicht den Wert	Vergleicht die Speicheradresse
<code>.equals()</code>	existiert nicht	kann implementiert werden

equals() überschreiben

equals

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

Parameters:

`obj` - the reference object with which to compare.

Returns:

`true` if this object is the same as the `obj` argument; `false` otherwise.

See Also:

`hashCode()`, `HashMap`

Wiederholung
○○○○

Best Practices
○○○○●○○○○○

Debugging & Testing
○○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

equals() überschreiben

- 1. `x.equals(x)` → true
- 2. `x.equals(y)` → `y.equals(x)`
- 3. `x.equals(y), y.equals(z)` → `x.equals(z)`
- 4. `x.equals(y), x,y gleich verändern` → `x.equals(y)`
- 5. `x.equals(null)` → false

■ Zusätzlich: Wenn `x.equals(y)`, dann auch `x.hashCode() == y.hashCode()`

equals() überschreiben

```
@Override  
public boolean equals(Object otherObj) {  
    if (this == otherObj) { return true; }  
    if (otherObj == null) { return false; }  
    if (getClass() != obj.getClass()) { return false; } // oder instanceof  
  
    MyClass other = (MyClass) otherObj;  
    // Vergleiche Eigenschaften  
    boolean isEqualA = this.a == other.a; // primitiver Datentyp  
    boolean isEqualB = this.b == other.b; // Enum  
    boolean isEqualC = this.c.equals(other.c); // Objekte  
    boolean isEqualD = this.d.compareTo(other.d) == 0; // String-Buffer  
    return isEqualA && isEqualB && isEqualC && ...;  
}
```

equals() überschreiben - Vererbung

- Mit Vererbung wirds messy
 - Prüfen der Attribute der Superklasse: `super.equals()`
- Entweder
 - `x.equals(y) == y.equals(x)` → Symmetrie (nutzt `getClass()`)
oder
 - `parent.equals(child) == true` → Liskov Substitution Principle (nutzt `instanceof`)
- Lösung: equals-Methode der Elternklasse nicht überschreiben lassen (→ final)

Wiederholung
○○○○

Best Practices
○○○○○○●○○

Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

hashCode() überschreiben

```
@Override  
public int hashCode() {  
    int result = 0;  
    // Ganzzahl  
    result = 31 * result + i;  
    // Float oder Double  
    result = 31 * result + (f != 0.0f ? Float.floatToIntBits(f) : 0);  
    // Objekte oder Enums  
    result = 31 * result + (name != null ? name.hashCode() : 0);  
  
    return result;  
}
```

Wiederholung
○○○○

Best Practices
○○○○○○○○●○

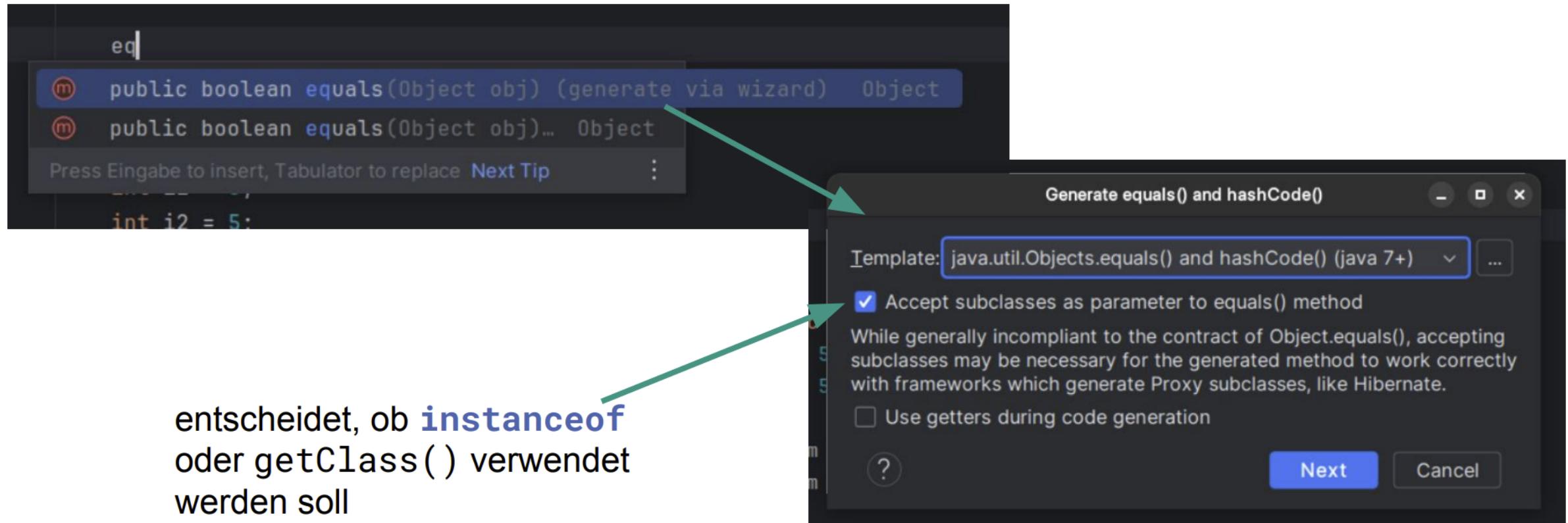
Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

equals() und hashCode() überschreiben

... oder automatisch generieren lassen.



Wiederholung
○○○○

Best Practices
○○○○○○○○●

Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

Debugging & Testing

Wiederholung
○○○○

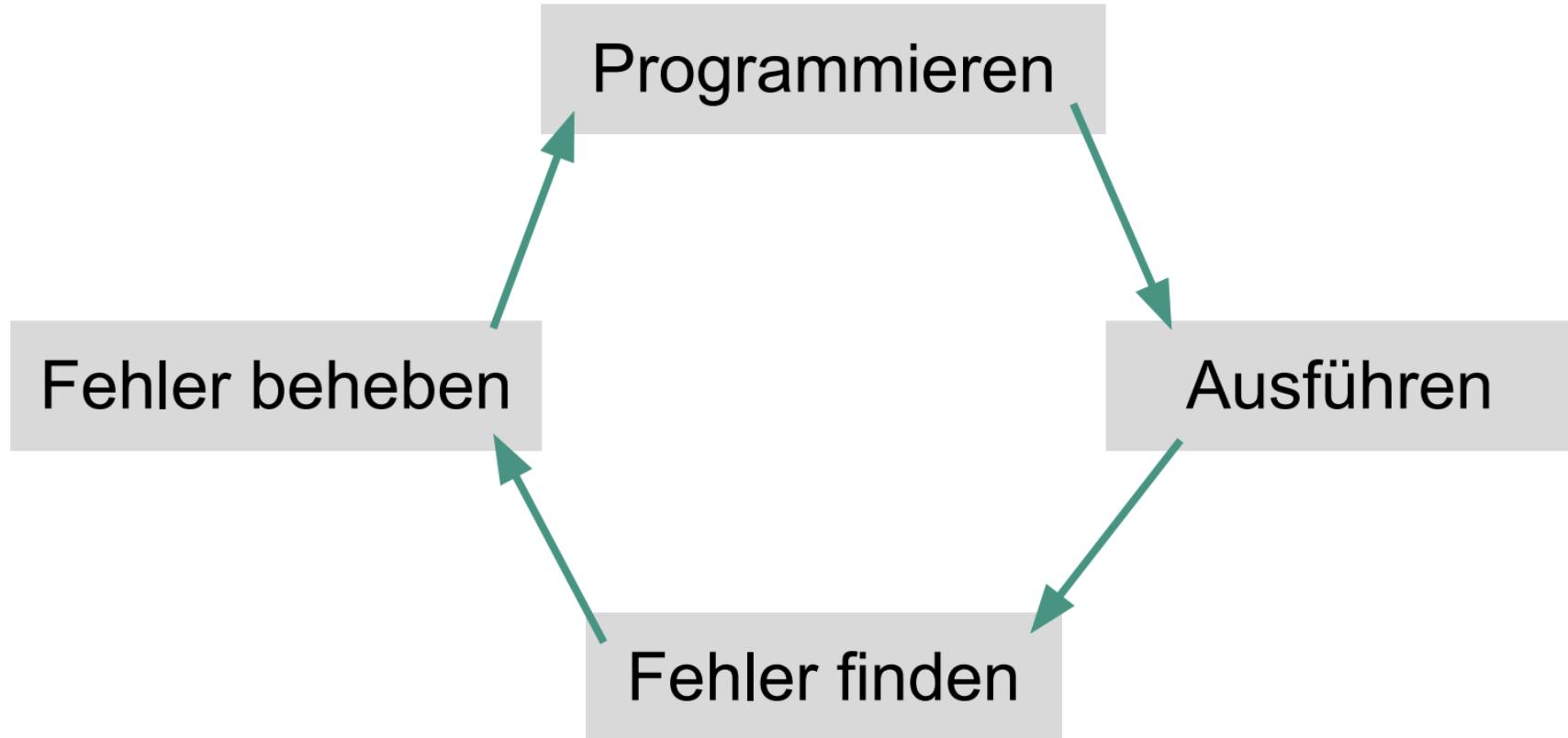
Best Practices
○○○○○○○○○○

Debugging & Testing
●○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

Typischer Arbeitsablauf



Wiederholung
oooo

Best Practices
oooooooooo

Debugging & Testing
○●oooo

RegEx
oooo

RegEx-Aufgaben
oooo

(Laufzeit-) Fehler finden

- Statische Codeanalyse → sinnvoll, aber nicht vollständig (findet vor allem Compiler-Fehler)
- Beim End-User → schlecht
- Manuell → sehr aufwendig
- Automatisiert (wie bei den Übungsblättern) → beste Möglichkeit

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○●○○○

RegEx
○○○○○

Regex-Aufgaben
○○○○

Fehler beheben / Debugging

- Sobald ein Fehler gefunden wurde, muss die Fehlerquelle identifiziert werden.

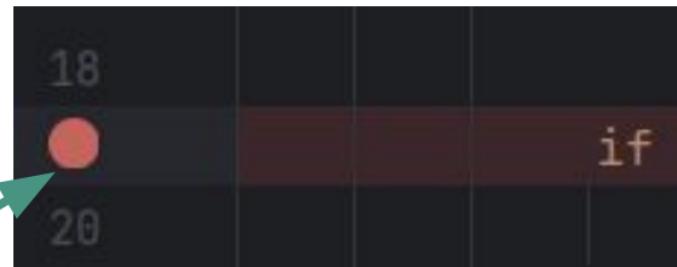
```
List<Integer> myList = new ArrayList<>();  
System.out.println(myList.toString());  
myList.add(5);  
System.out.println(myList.toString());  
myList.addAll(otherList);  
System.out.println(myList.toString());  
System.out.println(myList.size());  
myList.remove(5);  
System.out.println(myList.toString());  
System.out.println(myList.size());
```



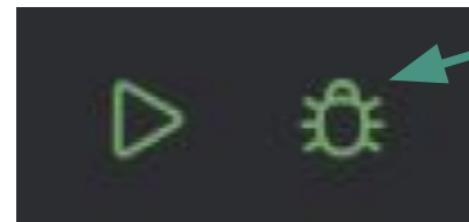
- Umständlich
- Aufwendig
- Fehleranfällig

Fehler beheben / Debugging

- Besser: Interaktiver Debugger / Breakpoints



Setzen von Breakpoints
(die Zeilenummer klicken)



Debug-Modus starten

Wiederholung
oooo

Best Practices
oooooooooo

Debugging & Testing
oooo●○

RegEx
oooo

Regex-Aufgaben
oooo

Allgemeine Tipps

- Ähnliche Eingaben probieren und schauen, wann das Ergebnis wieder richtig ist.
- Nur kleine Teile des Codes prüfen, nicht alles zusammen.
- Eine dritte Person fragen.
- Eine Pause machen!
- Auf kleine Tippfehler achten (z.B. string1, string2)

Wiederholung
oooo

Best Practices
oooooooooo

Debugging & Testing
oooo●●

RegEx
ooooo

Regex-Aufgaben
oooo

Reguläre Ausdrücke

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○

RegEx
●○○○○

Regex-Aufgaben
○○○○

GBI Ausschnitt

Definition regulärer Ausdrücke (1)



RA kleinste Menge mit:

- $\emptyset \in RA$
- $R_1, R_2 \in RA \implies (R_1 | R_2), (R_1 R_2) \in RA$
- $R \in RA \implies (R^*) \in RA$
- Alphabet $Z = \{ |, (,), *, \emptyset \}$ von «Hilfssymbolen»
- Alphabet A enthalte kein Zeichen aus Z
- *regulärer Ausdruck* über A ist eine Zeichenfolge über dem Alphabet $A \cup Z$, die gewissen Vorschriften genügt.
- Menge der regulären Ausdrücke (RA) ist wie folgt festgelegt:
 - \emptyset ist ein RA
 - für jedes $x \in A$ ist x RA
 - wenn R_1 und R_2 RA sind, dann auch $(R_1 | R_2)$ und $(R_1 R_2)$
 - wenn R ein RA ist, dann auch (R^*)
 - Nichts anderes sind reguläre Ausdrücke.
(vgl. Definition von Formeln in Aussagen- und Prädikatenlogik)

Reguläre Ausdrücke im Programmieren

- “RegEx” (**Regular Expressions**)
 - Mächtig zum Prüfen von Strings
 - z.B. Nutzerinteraktionen
 - In vielen Programmiersprachen implementiert
- Cheatsheet, Interaktive Tests etc.: [RegExr](#)

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○○

RegEx
○○●○○

Regex-Aufgaben
○○○○

Syntax-Übersicht

Character classes

.	any character except newline
\w \d \s	word, digit, whitespace
\W \D \S	not word, digit, whitespace
[abc]	any of a, b, or c
[^abc]	not a, b, or c
[a-g]	character between a & g

Anchors

^abc\$	start / end of the string
\b \B	word, not-word boundary

Escaped characters

\. * \\	escaped special characters
\t \n \r	tab, linefeed, carriage return

Groups & Lookaround

(abc)	capture group
\1	backreference to group #1
(?:abc)	non-capturing group
(?=abc)	positive lookahead
(?!abc)	negative lookahead

Quantifiers & Alternation

a*	0 or more, 1 or more, 0 or 1
a{5}	exactly five, two or more
a{1,3}	between one & three
a+?	match as few as possible
a{2,}?	match ab or cd

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○

RegEx
○○○●○

Regex-Aufgaben
○○○○

RegEx in Java

■ `java.util.regex` oder `String::matches(String regex)`

Beispiele:

```
String input = scanner.next();
String regex = "^(add )([A-Z][a-z] ){2}$";
input.matches(regex);
```

```
String input = scanner.next();
String regex = "^(\\d+)$";

if (! input.matches(regex)) {
    System.out.println("Integer needed");
}
int i = Integer.parseInt(input);
```

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○○

RegEx
○○○○●

Regex-Aufgaben
○○○○

Aufgabe 1

Testfälle	Akzeptanz -kriterium	[^a-z]{3}\-[^a-z]{2}\s\d*	[\w]+-\[\w]+\ \ \d+	[A-Z]+(- \s)[\w\]{1,8}
LIP-PE 123	X			
lip-pe 123	-			
BI ER 9876	X			
B-U 56789	X			
PB-Z 11	X			
HF-ABC 654	-			
MI-LA 54321	-			

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○○

RegEx
○○○○○

Regex-Aufgaben
●○○○

Lösung 1

Testfälle	Akzeptanz -kriterium	<code>[^a-z]{3}\-[^a-z]{2}\s\d*</code>	<code>\w+\-\w+\ \d+</code>	<code>[A-Z]+(- \s)\w\]{1,8}</code>
LIP-PE 123	X	X	X	X
lip-pe 123	-	-	X	-
BI ER 9876	X	-	-	X
B-U 56789	X	-	X	X
PB-Z 11	X	-	X	X
HF-ABC 654	-	-	X	X
MI-LA 54321	-	-	X	X

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○●○○

Aufgabe 2

Aufgabenstellung

Schreiben Sie einen regulären Ausdruck plus einen e Schreiben Sie einen regulären Ausdruck plus einen ersetzen den Ausdruck, der ersetzen den Ausdruck, der unterstrichene Texte (...) durch kursive Texte (*...*) ersetzt. (*...*) ersetzt.

Lösung

Suchausdruck: <u>(.*)</u>

Ersetzender Ausdruck: <i>\1</i>

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○●○

bis nächste Woche :)

Wiederholung
○○○○

Best Practices
○○○○○○○○○○

Debugging & Testing
○○○○○

RegEx
○○○○○

Regex-Aufgaben
○○○●