

Have you ever considered  
just not writing bugs?

Tagir Valeev



**Lou Creemers** 

@lovelacecoding



Have you all ever considered just not writing bugs into your software?

8:50 AM · Feb 23, 2024 · **46.8K** Views



159



66



417



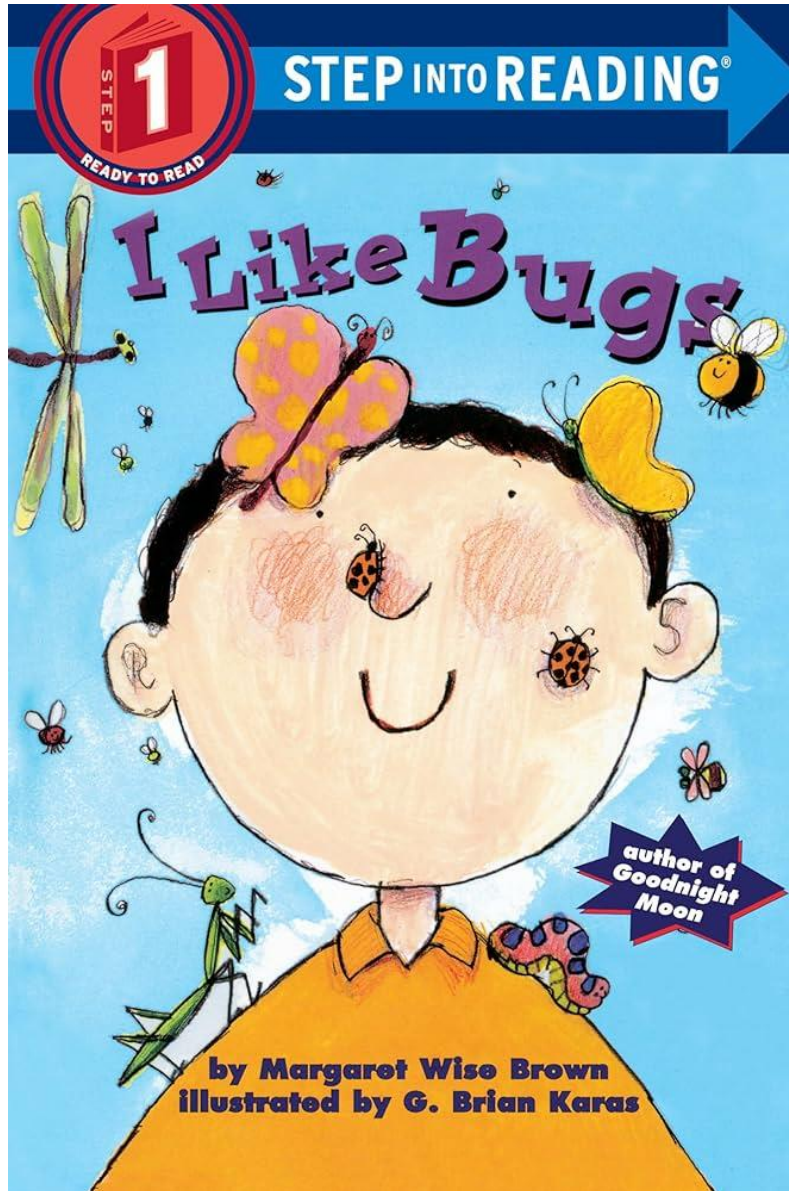
10

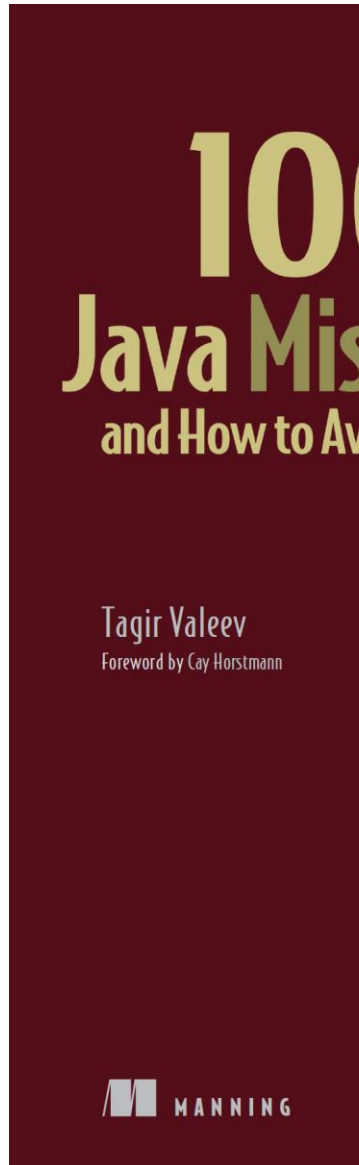


# About me

- ✓ 15 years of Java programming experience
- ✓ 7 years in JetBrains (IntelliJ IDEA Java team technical lead)
- ✓ Contributed to FindBugs static analyzer (~2014)
- ✓ OpenJDK committer
- ✓ Java Champion
- ✓ Wrote a book about Java mistakes



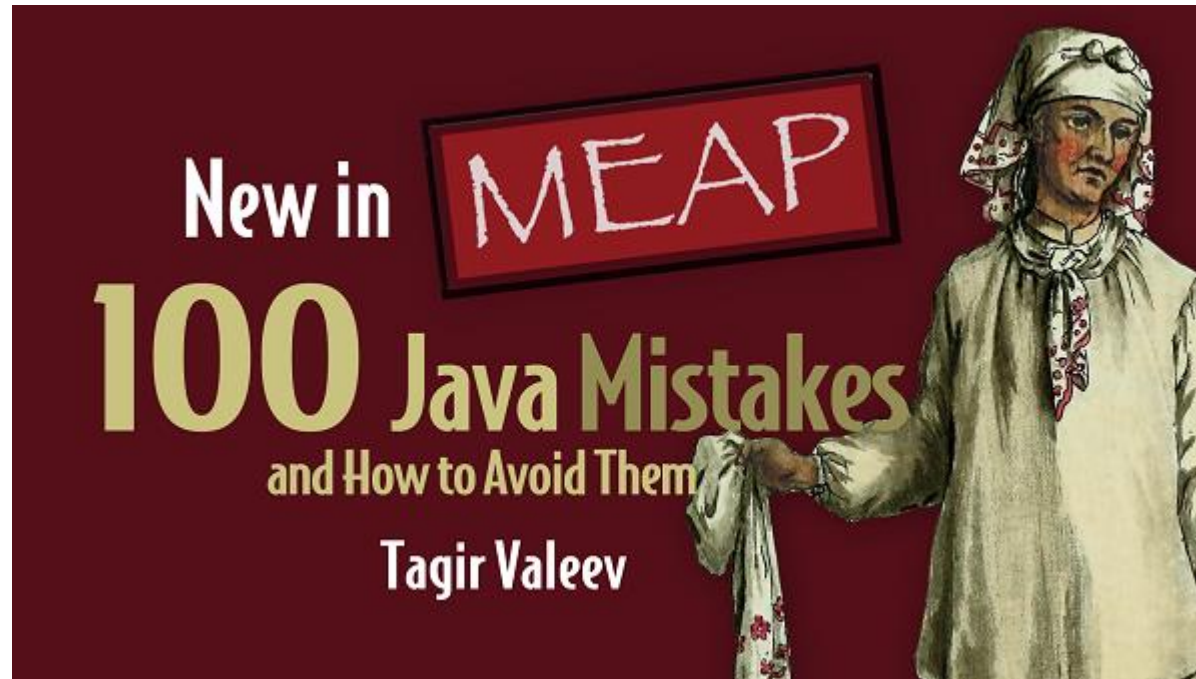




<https://www.manning.com/books/100-java-mistakes-and-how-to-avoid-them>



<https://www.amazon.com/dp/1633437965/>



Promotion Code (45% off)

100 Java Mistakes (both e-book and paper)

<https://www.manning.com/books/100-java-mistakes-and-how-to-avoid-them>

**valeevmu2**



# Bugs

Huge and complex



Tiny and local



```
- return Stream.of(streams).flatMap(s → (Stream<String>) s);
```



**Tagir Valeev** 12:57 Unresolved ✕

Was the stream of a single `Set<Stream>` element instead of multiple `Stream` elements. Btw mistake#70 from my "100 Java Mistakes" book 🤔

Reply

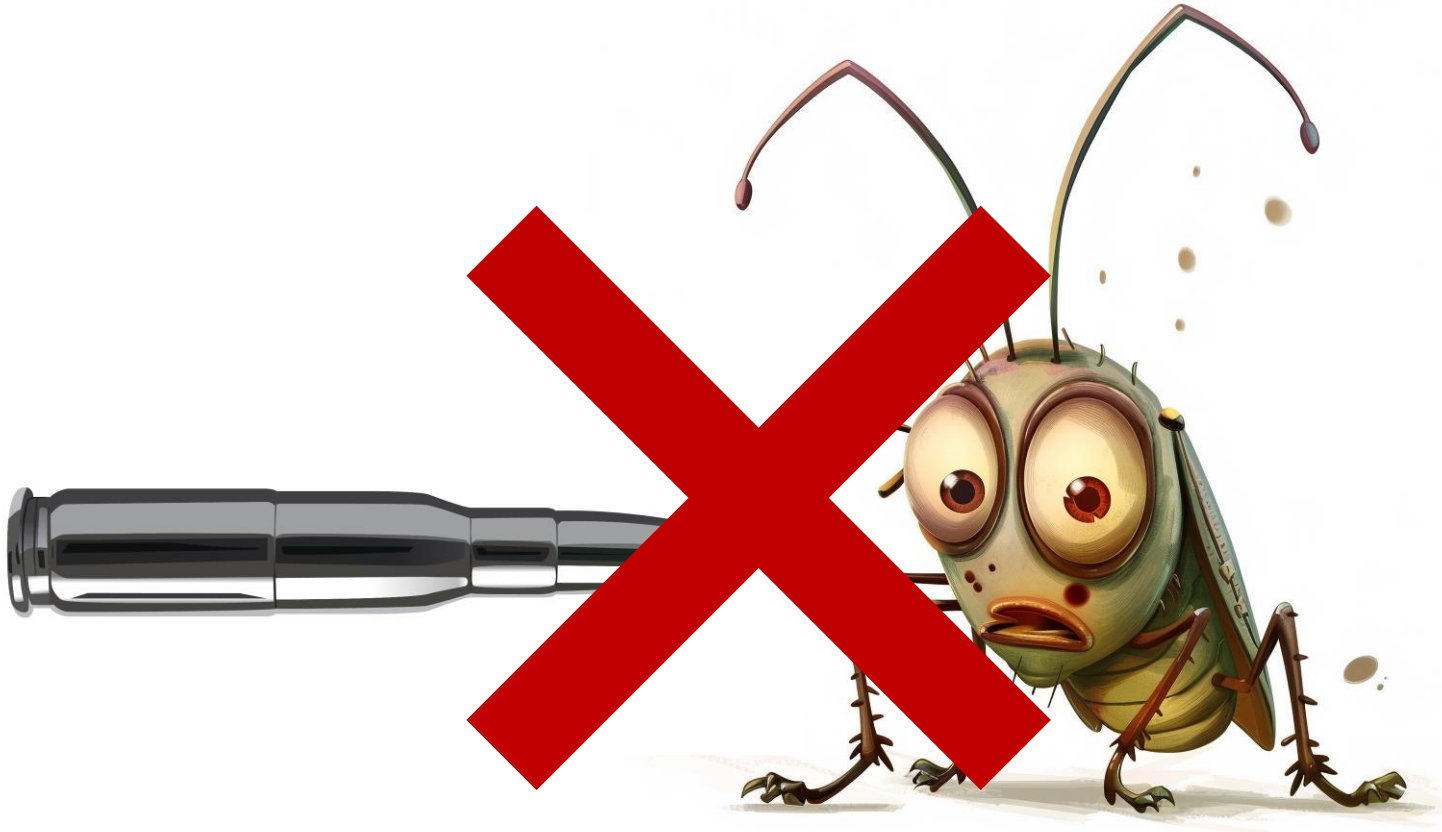
```
+ return streams.stream().flatMap(s → s);
```



# How to avoid bugs?

- ✓ Code style
- ✓ Idiomatic code
- ✓ Avoid repetitions
- ✓ Code review
- ✓ Pair programming
- ✓ Assertions
- ✓ AI!
- ✓ Static analysis
- ✓ Dynamic analysis (Pathfinder)
- ✓ Testing
  - ✓ Unit tests
    - ✓ + Coverage control (JaCoCo)
    - ✓ + Mutation coverage control (Pitest)
  - ✓ Smoke tests
  - ✓ Functional tests
  - ✓ Integration tests
  - ✓ Property tests
  - ✓ ...

Bad news: there's no silver bullet



COVERALLS



Repo on GitHub

AMAEMBO / STREAMEX

100%



DEFAULT BRANCH: MASTER

REPO ADDED  
08 SEP 2015 10:23AM

FILES  
44

BADGE  
coverage 100%

EMBED

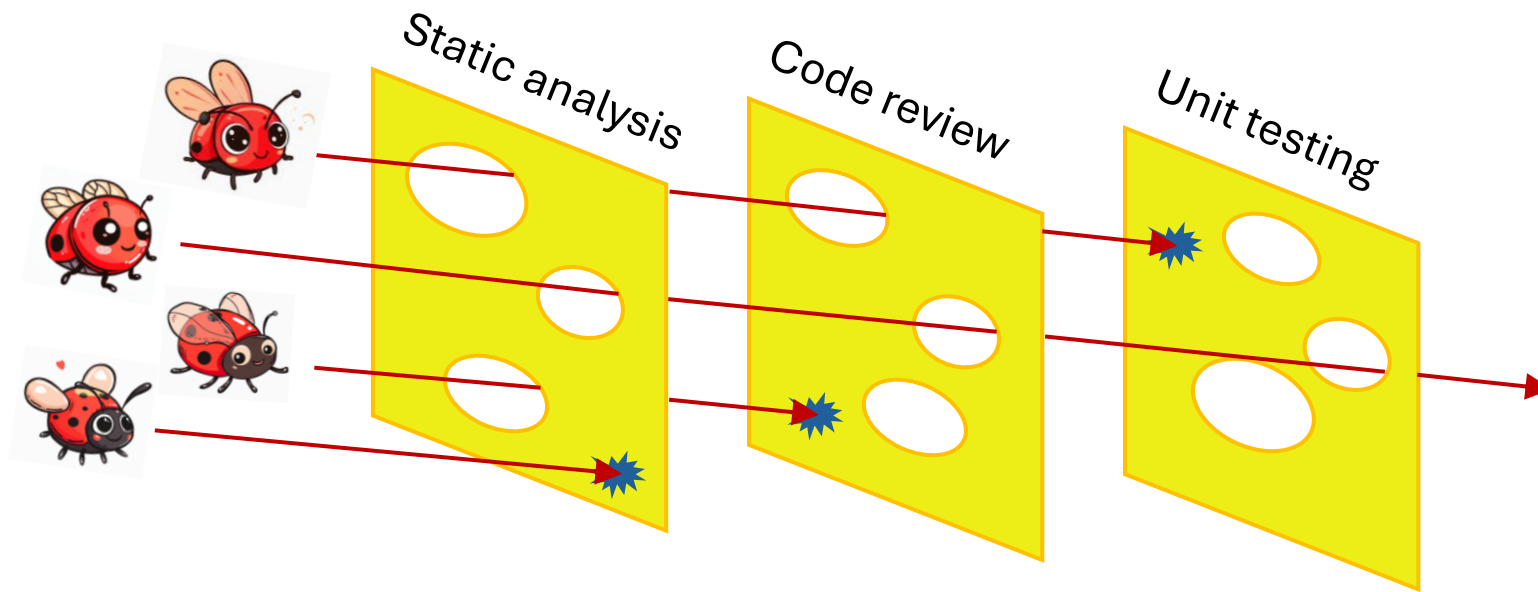
LAST BUILD ON BRANCH MASTER

BRANCH: MASTER

COMMITTED 29 NOV 2023 09:51AM				COVERAGE: 99.69%. REMAINED THE SAME
BUILD #	BUILD TYPE	COMMITTED BY	COMMIT MESSAGE	RUN DETAILS
#673	push	web-flow	Merge pull request #274 from Kivanval/fix/desc  Correcting a comment in StreamEx.java	5787 of 5805 relevant lines covered (99.69%) 1.0 hits per line

RELEVANT LINES COVERED

# Good news: Swiss cheese model



James Reason, 1990

[https://en.wikipedia.org/wiki/Swiss\\_cheese\\_model](https://en.wikipedia.org/wiki/Swiss_cheese_model)

# Initial capacity (StringBuilder)

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + indent < 0 ? 0 : indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```

# Initial capacity

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + indent < 0 ? 0 : indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```



# Initial capacity

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + indent < 0 ? 0 : indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```

1. indent >= 0

→ capacity = indent >= 0



2. -str.length() <= indent < 0 → capacity = indent < 0



3. indent < -str.length() → capacity = 0



# Let's ask AI

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + indent < 0 ? 0 : indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```

  Extract Surround //  

Explain Code

Suggest Refactoring

Find Problems

New Chat Using Selection

Suggest refactoring

## AI Assistant



The provided Java code is generally accurate but there can be some potential issues depending on how it is used:

1. **Null String Handling:** The method does not handle the case where the input string (`str`) is null. This will result in a `NullPointerException` when calling `str.length()`. To handle this, you could set the default value for `str` to an empty string if it is null.

Here is a modified code snippet for this:

Java



```
static String indentString(String str, int indent) {  
    if (str == null) {  
        str = "";  
    }  
  
    int capacity = str.length() + indent < 0 ? 0 : indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```



2. **Negative Indent Handling:** The method allows for negative indent values. Depending on your expected behavior, you would want to handle this properly. Currently, a negative `indent` value causes the for loop to be skipped entirely, resulting in the method returning the input string without the indentation spaces. If a negative indent is considered invalid, then you could throw an exception or default to zero.

Here's a code snippet for handling negative indent:

```
Java
static String indentString(String str, int indent) {
    if (str == null) {
        str = "";
    }

    if (indent < 0) {
        throw new IllegalArgumentException("Indent must not be negative.");
    }

    int capacity = str.length() + indent;
    StringBuilder sb = new StringBuilder(capacity);
    for (int i = 0; i < indent; i++) {
        sb.append(' ');
    }
    sb.append(str);
    return sb.toString();
}
```

3. **StringBuilder Capacity:** The current method calculates capacity of StringBuilder based on string length and indent. However, the calculation (`str.length() + indent < 0 ? 0 : indent;`) might be giving incorrect capacity in case if `str.length() + indent` is negative. Capacity of StringBuilder should be sum of length of String and the indent, to accommodate the initial string plus additional spaces.

```
Java
static String indentString(String str, int indent) {
    if (str == null) {
        str = "";
    }

    if (indent < 0) {
        throw new IllegalArgumentException("Indent must not be negative.");
    }

    int capacity = str.length() + indent;
    StringBuilder sb = new StringBuilder(capacity);
    for (int i = 0; i < indent; i++) {
        sb.append(' ');
    }
    sb.append(str);
    return sb.toString();
}
```

This will help to correct and optimize the capacity of StringBuilder and prevent unnecessary resizing and data copying.

# What's wrong here?

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + (indent < 0 ? 0 : indent);  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```



# What's wrong here?

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + indent < 0 ? 0 : indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```

- ✓ Initial capacity is not easily testable!
- ✓ Manual algorithm implementation
- ✓ Action at a distance
- ✓ Premature optimization?

# Asserts to the rescue?

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + indent < 0 ? 0 : indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    assert capacity == sb.length();  
    return sb.toString();  
}
```

- ✓ Initial capacity is not easily testable!
- ✓ Manual algorithm implementation
- ✓ Action at a distance
- ✓ Premature optimization?

# How to fix?

```
static String indentString(String str, int indent) {  
    int capacity = str.length() + Math.max(0, indent);  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```

- ✓ Initial capacity is not easily testable!
- ✓ Manual algorithm implementation
- ✓ Action at a distance
- ✓ Premature optimization?

# What's wrong here?

```
static String indentStringAndLineBreak(String str, int indent) {  
    int capacity = str.length() + Math.max(0, indent) + 1;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    sb.append('\n');  
    return sb.toString();  
}
```

- ✓ Initial capacity is not easily testable!
- ✓ Manual algorithm implementation
- ✓ Action at a distance
- ✓ Premature optimization?

# How to fix?

```
static String indentString(String str, int indent) {  
    if (indent <= 0) return str;  
    int capacity = str.length() + indent;  
    StringBuilder sb = new StringBuilder(capacity);  
    for (int i = 0; i < indent; i++) {  
        sb.append(' ');  
    }  
    sb.append(str);  
    return sb.toString();  
}
```

- ✓ Initial capacity is not easily testable!
- ✓ Manual algorithm implementation
- ✓ Action at a distance
- ✓ Premature optimization?

# Or probably this way?

```
static String indentString(String str, int indent) {  
    if (indent <= 0) return str;  
    return " ".repeat(indent) + str;  
}
```

- ✓ Initial capacity is not easily testable!
- ✓ Manual algorithm implementation
- ✓ Action at a distance
- ✓ Premature optimization?



# Initial capacity (ArrayList)

```
List<String> trimAndAdd(List<String> input, String newItem) {  
    List<String> result = new ArrayList<>(input.size() + newItem == null ? 0 : 1);  
    for (String s : input) {  
        result.add(s.trim());  
    }  
    if (newItem != null) {  
        result.add(newItem.trim());  
    }  
    return result;  
}
```

# Precedence again

```
List<String> trimAndAdd(List<String> input, String newItem) {  
    List<String> result = new ArrayList<>(input.size() + newItem == null ? 0 : 1);  
    for (String s : input) {  
        result.add(s.trim());  
    }  
    if (newItem != null) {  
        result.add(newItem.trim());  
    }  
    return result;  
}
```

# Static analysis cheese slice helps

```
List<String> trimAndAdd(List<String> input, String newItem) {  
    List<String> result = new ArrayList<>(input.size() + newItem == null ? 0 : 1);  
    for (String s : input) {  
        result.add(s.trim());  
    }  
    if (newItem != null) {  
        result.add(newItem.trim());  
    }  
    return result;  
}
```

Condition 'input.size() + newItem == null' is always 'false'

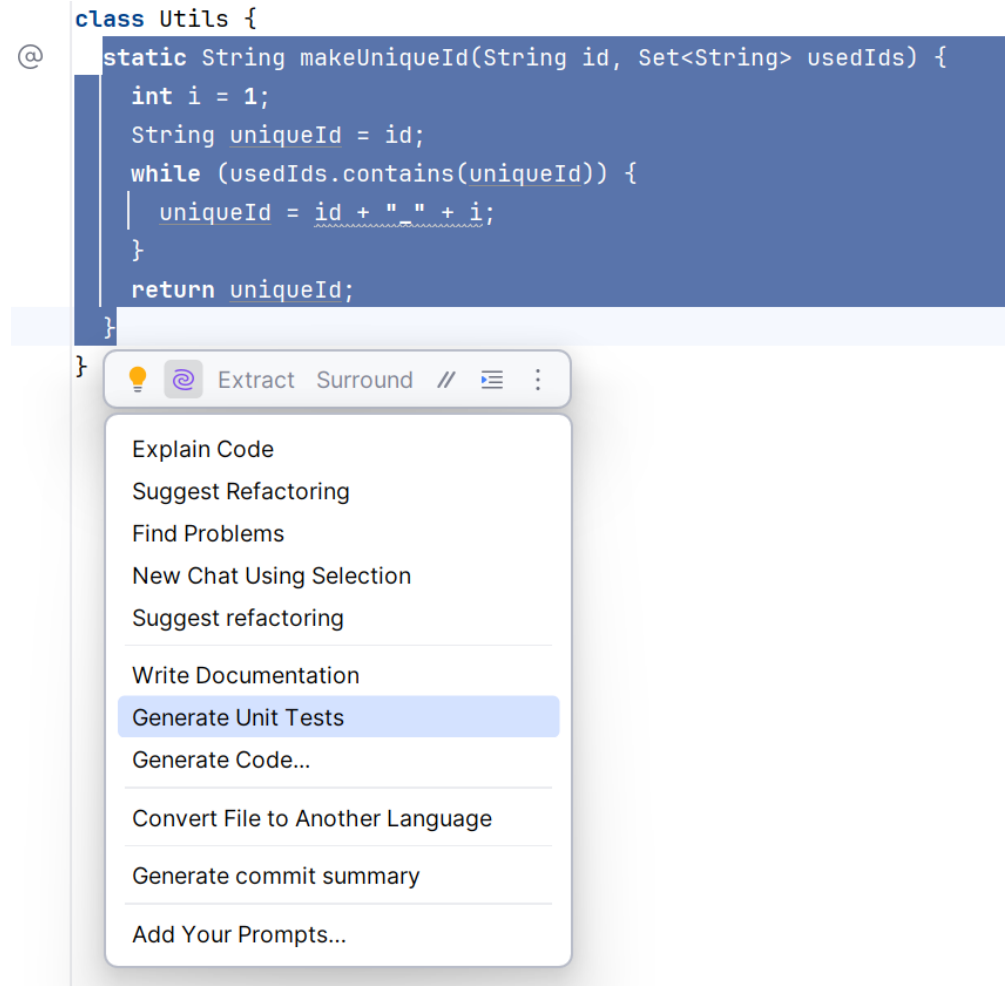
Simplify 'input.size() + newItem == null' to false Alt+Shift+Enter

# Write tests!

```
import java.util.Set;

class Utils {
    static String makeUniqueId(String id, Set<String> usedIds) {
        int i = 1;
        String uniqueId = id;
        while (usedIds.contains(uniqueId)) {
            uniqueId = id + "_" + i;
        }
        return uniqueId;
    }
}
```

# AI can do this!



```
public final class UtilsTest {

    @Test
    public void testMakeUniqueId() {
        String id = "Test";
        Set<String> usedIds = new HashSet<>();
        String newId = Utils.makeUniqueId(id, usedIds);

        assertEquals(id, newId);
    }

    @Test
    public void testMakeUniqueIdWithPresentId() {
        String id = "Test";
        Set<String> usedIds = new HashSet<>();
        usedIds.add(id);
        String newId = Utils.makeUniqueId(id, usedIds);

        assertNotEquals(id, newId);
    }
}
```



# Check coverage

```
7  class Utils {
8  @   static String makeUniqueId(String id, Set<String> usedIds) {
9      int i = 1;
10     String uniqueId = id;
11     while (usedIds.contains(uniqueId)) {
12         uniqueId = id + "_" + i;
13     }
14     return uniqueId;
15 }
16 }
```

Coverage    UtilsTest ×				
⌵ ⬆ ⬇ ↗ ↘ 🔍				
Element ^	Class, %	Method, %	Line, %	Branch, %
✓ 📁 com.example	100% (1/1)	100% (1/1)	100% (5/5)	100% (2/2)
🕒 Utils	100% (1/1)	100% (1/1)	100% (5/5)	100% (2/2)

# Oops!

```
@Test
public void testMakeUniqueIdWithTwoConflicts() {
    String newId = Utils.makeUniqueId("Test", Set.of("Test", "Test_1"));
    assertEquals("Test_2", newId);
}
```

# Oops!

```
@Test
public void testMakeUniqueIdWithTwoConflicts() {
    String newId = Utils.makeUniqueId("Test", Set.of("Test", "Test_1"));
    assertEquals("Test_2", newId);
}
```



```
@Test
public void testMakeUniqueIdWithTwoConflicts() {
    String newId = Utils.makeUniqueId("Test", Set.of("Test", "Test_1"));
    assertEquals("Test_2", newId);
}
```

```
import java.util.Set;
```

```
class Utils {
    static String makeUniqueId(String id, Set<String> usedIds) {
        int i = 1;
        String uniqueId = id;
        while (usedIds.contains(uniqueId)) {
            uniqueId = id + "_" + i;
        }
        return uniqueId;
    }
}
```

# Idempotent loop body

```
import java.util.Set;

class Utils {
    static String makeUniqueId(String id, Set<String> usedIds) {
        int i = 1;
        String uniqueId = id;
        while (usedIds.contains(uniqueId)) {
            uniqueId = id + "_" + i;
        }
        return uniqueId;
    }
}
```

1. *The condition is false initially.* The loop is not executed at all.
2. *The condition is true initially, but the body makes it false.* The loop is executed only once.
3. *The condition is true initially, and the body doesn't change it.* The loop is infinite, and the program hangs.

# Static analysis helps here!

```
static String makeUniqueId(String id, Set<String> usedIds) {  
    int i = 1;  
    String uniqueId = id;  
    while (usedIds.contains(uniqueId)) {  
        Idempotent loop body    : ;  
    }  
    return uniqueId;  
}
```

Rule: If you have a loop, your tests should test at least cases with 0, 1, and 2+ iterations.

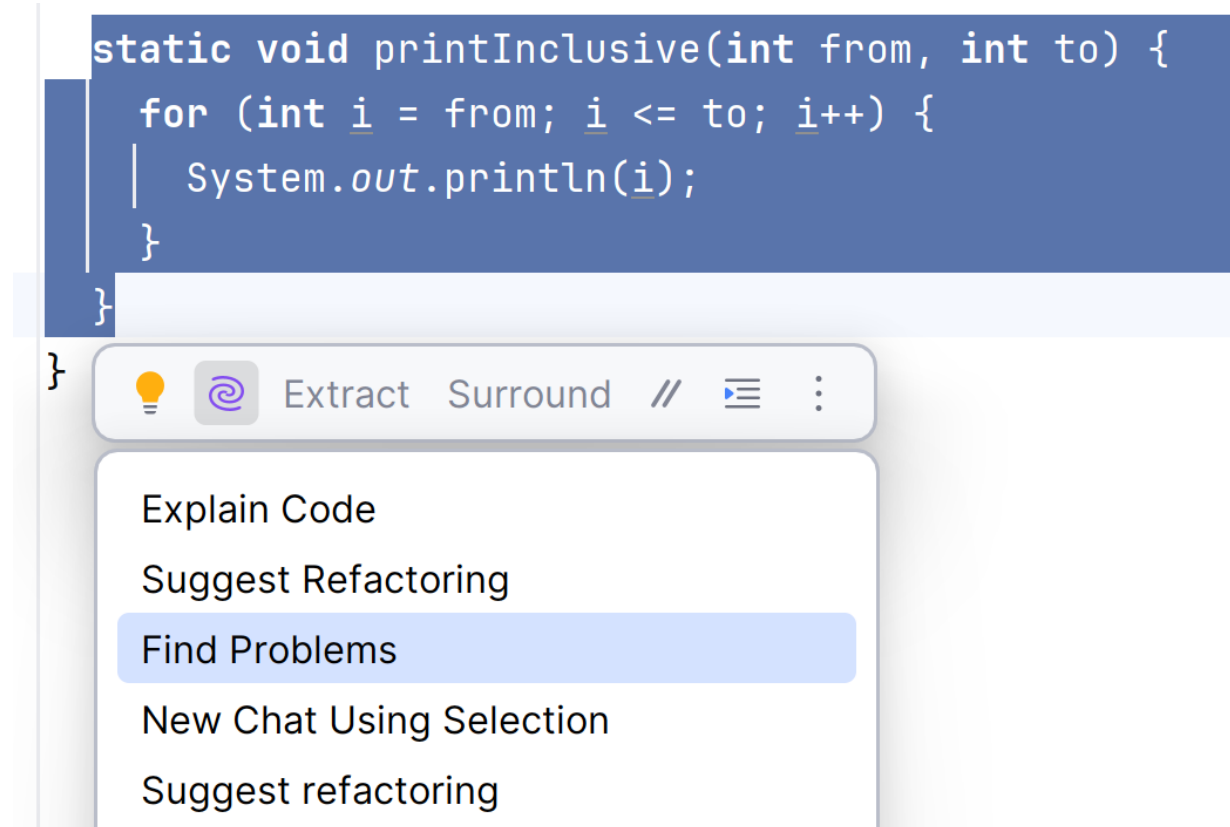
```
static String makeUniqueId(String id, Set<String> usedIds) {  
    int i = 1;  
    String uniqueId = id;  
    while (usedIds.contains(uniqueId)) {  
        Idempotent loop body    ;  
    }  
    return uniqueId;  
}
```

# Corner cases

```
static void printInclusive(int from, int to) {  
    for (int i = from; i <= to; i++) {  
        System.out.println(i);  
    }  
}
```



# AI to the rescue?



# AI to the rescue?

```
static void printInclusive(int from, int to) {  
    for (int i = from; i <= to; i++) {  
        System.out.println(i);  
    }  
}
```

## AI Assistant



### Potential Problems

1. **Invalid range:** There is no check to ensure from is less than or equals to. If from is greater than to, then the loop will not execute, which might not be the expected behavior.
2. **Arguments Validation:** There are no checks for the validity of the parameters. If from or to is a very large positive or negative number, it might lead to unexpected behavior.
3. **Consistent execution time:** Due to the nature of the loop, the execution time of this method will vary depending on the size of the range (i.e., the difference between from and to). This could potentially impact performance if the range is large.

# Let's protect ourselves

```
static void printInclusive(int from, int to) {  
    if (from > to) {  
        throw new IllegalArgumentException("from > to");  
    }  
    if (to - from < 0 || to - from > 1000) {  
        throw new IllegalArgumentException("too many numbers to process");  
    }  
    for (int i = from; i <= to; i++) {  
        System.out.println(i);  
    }  
}
```

# Are there still mistakes?

```
static void printInclusive(int from, int to) {  
    if (from > to) {  
        throw new IllegalArgumentException("from > to");  
    }  
    if (to - from < 0 || to - from > 1000) {  
        throw new IllegalArgumentException("too many numbers to process");  
    }  
    for (int i = from; i <= to; i++) {  
        System.out.println(i);  
    }  
}
```

I see the following potential issues:

1. Redundant condition check: The condition `to - from < 0` is redundant because if it were true, it would have already been caught by the `from > to` check.
2. Lack of null check: Although your function does not include any object parameters, if you plan to pass or let users pass object parameters it's always recommended to add proper null checks.
3. Error message clarity: The error message `too many numbers to process` could be a bit clearer. Something like `the difference between 'to' and 'from' exceeds the limit of 1000` would more directly explain the error.
4. Method accessibility: The `printInclusive` method is declared static, which is fine if this method should be accessible without needing an instance of the `Utils` class. But if you intend to use this method in a more object-oriented context, you might want to consider making it an instance method.
5. Unused class warning: The `Utils` class is currently marked as unused. This could just be because the method hasn't been called yet in any running code. If the method is used, the `SuppressWarnings("unused")` line can be removed.
6. Throwing unchecked exceptions: `IllegalArgumentException` is an unchecked exception. Use of unchecked exceptions is part of Java programming convention, but it's good to know that they could potentially cause issues in larger systems if not caught, as they do not have to be declared in method signatures.

Do remember that some of these issues are not necessarily problems, but more considerations to keep in mind based on your specific use case and programming style preferences.

# Hm...

```
static void printInclusive(int from, int to) {  
    if (from > to) {  
        throw new IllegalArgumentException("from > to");  
    }  
    if (to - from < 0 || to - from > 1000) {  
        throw new IllegalArgumentException("too many numbers to process");  
    }  
    for (int i = from; i <= to; i++) {  
        System.out.println(i);  
    }  
}
```

Bullshit: `printInclusive(-2_000_000_000, 2_000_000_000);`

More null-checks for the  
null-check god

Doubtful but okaaaay

Are you crazy?

C'mon it's just a sample

OMG

I see the following potential issues:

1. Redundant condition check: The condition `to - from < 0` is redundant because if it were true, it would have already been caught by the `from > to` check.
2. Lack of null check: Although your function does not include any object parameters, if you plan to pass or let users pass object parameters it's always recommended to add proper null checks.
3. Error message clarity: The error message `too many numbers to process` could be a bit clearer. Something like the difference between 'to' and 'from' exceeds the limit of 1000 would more directly explain the error.
4. Method accessibility: The `printInclusive` method is declared static, which is fine if this method should be accessible without needing an instance of the `Utils` class. But if you intend to use this method in a more object-oriented context, you might want to consider making it an instance method.
5. Unused class warning: The `Utils` class is currently marked as unused. This could just be because the method hasn't been called yet in any running code. If the method is used, the `SuppressWarnings("unused")` line can be removed.
6. Throwing unchecked exceptions: `IllegalArgumentException` is an unchecked exception. Use of unchecked exceptions is part of Java programming convention, but it's good to know that they could potentially cause issues in larger systems if not caught, as they do not have to be declared in method signatures.

Do remember that some of these issues are not necessarily problems, but more considerations to keep in mind based on your specific use case and programming style preferences.

# Oops

```
static void printInclusive(int from, int to) {  
    if (from > to) {  
        throw new IllegalArgumentException("from > to");  
    }  
    if (to - from < 0 || to - from > 1000) {  
        throw new IllegalArgumentException("too many numbers to process");  
    }  
    for (int i = from; i <= to; i++) {  
        System.out.println(i);  
    }  
}  
  
public static void main(String[] args) {  
    printInclusive(Integer.MAX_VALUE - 10, Integer.MAX_VALUE);  
}
```

-2144389300  
-2144389299  
-2144389298  
-2144389297  
-2144389296  
-2144389295  
-2144389294  
-2144389293  
-2144389292  
-2144389291  
-2144389290  
-2144389289  
-2144389288  
-2144389287  
-2144389286  
-2144389285  
-2144389284  
-2144389283  
-2144389282

```
static void printInclusive(int from, int to) {  
    if (from > to) {  
        throw new IllegalArgumentException("from > to");  
    }  
    for (int i = from; i <= to; i++) {  
        System.out.println(i);  
    }  
}  
  
public static void main(String[] args) {  
    printInclusive(Integer.MAX_VALUE, Integer.MAX_VALUE);  
}
```

-2143740797  
-2143740796  
-2143740795  
-2143740794  
-2143740793  
-2143740792  
-2143740791  
-2143740790  
-2143740789  
-2143740788  
-2143740787  
-2143740786  
-2143740785  
-2143740784  
-2143740783  
-2143740782

-2144033077  
-2144033076  
-2144033075  
-2144033074  
-2144033073  
-2144033072  
-2144033071  
-2144033070  
-2144033069  
-2144033068  
-2144033067  
-2144033066  
-2144033065  
-2144033064  
-2144033063  
-2144033062  
-2144033061  
-2144033060

# How to fix it?

```
static void printInclusive(int from, int to) {  
    if (from > to) {  
        throw new IllegalArgumentException("from > to");  
    }  
    if (to - from < 0 || to - from > 1000) {  
        throw new IllegalArgumentException("too many numbers to process");  
    }  
    for (int i = from; i >= from && i <= to; i++) {  
        System.out.println(i);  
    }  
}  
  
public static void main(String[] args) {  
    printInclusive(Integer.MAX_VALUE - 10, Integer.MAX_VALUE);  
}
```



# Real life

▼ ↕ 3 ■■■■ java/java-analysis-impl/src/com/intellij/codeInspection/dataFlow/rangeSet/LongRangeSet.java

⋮		@@ -763,7 +763,8 @@ public static LongRangeSet modRange(long from, long to, long mod, long bits) {
763	763	for (int newMod = (int)length; newMod <= intMod / 2; newMod++) {
764	764	if (intMod % newMod == 0) {
765	765	long newBits = 0;
766	-	for (long i = from; i <= to; i++) {
	766	+ // `to` could be Long.MAX_VALUE; so `i >= from` condition is important to react on possible overflow
	767	+ for (long i = from; i >= from && i <= to; i++) {
767	768	if (isSet(bits, remainder(i, intMod))) {
768	769	newBits = setBit(newBits, remainder(i, newMod));
769	770	}
⋮		

<https://github.com/JetBrains/intellij-community/commit/b56a1fc16d446c92528994e0d333cbfa9ade21cd>

# Use Stream API!

```
static void printInclusive(int from, int to) {  
    if (from > to) {  
        throw new IllegalArgumentException("from > to");  
    }  
    if (to - from < 0 || to - from > 1000) {  
        throw new IllegalArgumentException("too many numbers to process");  
    }  
    for (int i = from; i <= to; i++) {  
        // ...  
    }  
}
```

Collapse loop with stream 'forEach()'

Collapse loop with stream 'forEachOrdered()'

Remove braces from 'for' statement

```
1 import java.util.stream.IntStream;
```

```
10 IntStream.rangeClosed(from, to).forEach(System.out::println);
```

# Convert to Kotlin! (Ctrl+Alt+Shift+K)

```
fun printInclusive(from: Int, to: Int) {  
    require(from <= to) { "from > to" }  
    require(to - from < 0 || to - from <= 1000) { "too many numbers to process" }  
    for (i in from..to) {  
        println(i)  
    }  
}
```

```
@JvmStatic  
fun main(args: Array<String>) {  
    printInclusive(Int.MAX_VALUE - 10, Int.MAX_VALUE)  
}
```


Output:


```
2147483637  
2147483638  
2147483639  
2147483640  
2147483641  
2147483642  
2147483643  
2147483644  
2147483645  
2147483646  
2147483647
```

Rule: You should have a language construct or library method for every useful and repeating task

Task: iterate over closed range of numbers

`IntStream.rangeClosed(from, to)` 

`for (i in from..to) { ... }` 

`for (int i = from; i <= to; i++) { ... }` 

# Clamping values

```
static void printProgress(int percent) {  
    if (percent > 100) {  
        percent = 100;  
    }  
    if (percent < 0) {  
        percent = 0;  
    }  
    System.out.println("Progress: " + percent + "%");  
}
```

# Clamping values

```
static void printProgress(int percent) {  
    percent = percent < 0 ? 0 : percent > 100 ? 100 : percent;  
    System.out.println("Progress: " + percent + "%");  
}
```

# Clamping values

```
static void printProgress(int percent) {  
    percent = Math.max(Math.min(percent, 0), 100);  
    System.out.println("Progress: " + percent + "%");  
}
```

# Clamping values

```
static void printProgress(int percent) {  
    percent = Math.max(Math.min(percent, 0), 100);  
    System.out.println("Progress: " + percent + "%");  
}
```

```
public static void main(String[] args) {  
    printProgress(-10);  
    printProgress(10);  
    printProgress(50);  
    printProgress(90);  
    printProgress(130);  
}
```

## Output:

Progress: 100%  
Progress: 100%  
Progress: 100%  
Progress: 100%  
Progress: 100%



# Clamping values

```
static void printProgress(int percent) {  
    percent = Math.max(Math.min(percent, 0), 100);  
    System.out.print  
}
```

Result of 'max' is the same as the second argument making the call meaningless

[Replace with '100'](#) Alt+Shift+Enter    [More actions...](#) Alt+Enter

# AI to the rescue!

## AI Assistant



The code you have posted contains a logical error. The purpose of the `Math.max` and `Math.min` expressions appears to be to ensure that `percent` is clamped between 0 and 100. However, the code isn't doing what it is expected to do.

The inner call to `Math.min(percent, 0)` will always return 0 if `percent` is greater than 0. Passing that result to `Math.max` along with 100 will always return 100. So, regardless of the initial value of `percent`, the output will always be 100%.

Java code:

Java



```
class Utils {  
    static void printProgress(int percent) {  
        percent = Math.max(Math.min(percent, 100), 0);  
        System.out.println("Progress: " + percent + "%");  
    }  
}
```

# In the wild

## Fixed min/max operations in ComputerRetentionWork

As previously implemented, nextCheck would always happen in 1 minute, regardless of what the RetentionStrategy.check() returned.

🔑 master (#1514)

📁 untagged-refs/heads/master ... 2.332.1-rc-2

Bernie Schelberg committed on Jan 6, 2015

```
core/src/main/java/hudson/slaves/ComputerRetentionWork.java

@@ -63,7 +63,7 @@ protected void doRun() {
63      63          if (!nextCheck.containsKey(c) || startRun > nextCheck.get(c)) {
64      64              // at the moment I don't trust strategies to wait more than 60 minutes
65      65              // strategies need to wait at least one minute
66      -      final long waitInMins = Math.min(1, Math.max(60, c.getRetentionStrategy().check(c)));
66      +      final long waitInMins = Math.max(1, Math.min(60, c.getRetentionStrategy().check(c)));
67      67              nextCheck.put(c, startRun + waitInMins*1000*60 /*MINS->MILLIS*/);
68      68          }
69      69      }
```

<https://github.com/jenkinsci/jenkins/commit/e00f99251e0b>

# Real solution (Java 21)



JDK / JDK-8301226

Add clamp() methods to java.lang.Math and to StrictMath

Resolved ▾

Export ▾

## Details

Type:	🚀 Enhancement	Resolution:	Fixed
Priority:	🔥 P4	Fix Version/s:	21
Affects Version/s:	21		
Component/s:	core-libs		
Labels:	release-note=yes		
Subcomponent:	java.lang		
Resolved In Build:	b10		

## Description

Quite often it's necessary to clamp a numerical value to a given range, using the algorithm like this:

```
int clampedValue = value > max ? max : value < min ? min : value;
```

or probably

```
int clampedValue = Math.max(min, Math.min(max, value));
```

Some examples in wild: [1] [2]

## People

Assignee:	Tagir Valeev ⓘ
Reporter:	Tagir Valeev ⓘ
Votes:	1 Vote for this issue
Watchers:	3 Start watching this issue

## Dates

Created:	2023-01-27 02:28
Updated:	2023-08-30 09:52
Resolved:	2023-02-13 21:40

# Real solution (Java 21)

```
static void printProgress(int percent) {  
    percent = Math.clamp(percent, 0, 100);  
    System.out.println("Progress: " + percent + "%");  
}
```

```
public static void main(String[] args) {  
    printProgress(-10);  
    printProgress(10);  
    printProgress(50);  
    printProgress(90);  
    printProgress(130);  
}
```

# Poor man template

```
private static final String TEMPLATE = "Hello USER_NAME!";

static void greetUser(String user) {
    String greeting = TEMPLATE.replaceAll("USER_NAME", user);
    System.out.println(greeting);
}
```

# Poor man template

```
private static final String TEMPLATE = "Hello USER_NAME!";

static void greetUser(String user) {
    String greeting = TEMPLATE.replaceAll("USER_NAME", user);
    System.out.println(greeting);
}

greetUser("John");

Hello John!
```

# Poor man template

```
private static final String TEMPLATE = "Hello USER_NAME!";

static void greetUser(String user) {
    String greeting = TEMPLATE.replaceAll("USER_NAME", user);
    System.out.println(greeting);
}

greetUser("$1lly name");
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: No group 1
    at java.base/java.util.regex.Matcher.checkGroup(Matcher.java:1818)
    at java.base/java.util.regex.Matcher.start(Matcher.java:496)
    at java.base/java.util.regex.Matcher.appendExpandedReplacement(Matcher.java:1107)
    at java.base/java.util.regex.Matcher.appendReplacement(Matcher.java:1014)
    at java.base/java.util.regex.Matcher.replaceAll(Matcher.java:1200)
    at java.base/java.lang.String.replaceAll(String.java:3065)
    at com.example.Utills.greetUser(Utills.java:8)
    at com.example.Utills.main(Utills.java:13)
```



# Convenience comes at a cost

`String.replace`: replaces all substrings

`String.replaceAll`: replaces all regular expression matches

```
public String replaceAll(String regex, String replacement) {  
    return Pattern.compile(regex).matcher(this).replaceAll(replacement);  
}
```

Rules: when designing your API, note that too many “convenient” methods may make things confusing.

Avoid stringly-typed code.

# Stringly-typed code

```
static int countPathComponents(String fileName) {  
    String[] components = fileName.split(File.separator);  
    return (int) Stream.of(components)  
        .filter(Predicate.not(String::isEmpty)).count();  
}
```

*countPathComponents("/etc/passwd") → 2*

# On Windows machine

```
static int countPathComponents(String fileName) {  
    String[] components = fileName.split(File.separator);  
    return (int) Stream.of(components)  
        .filter(Predicate.not(String::isEmpty)).count();  
}
```

```
countPathComponents("C:\\tmp\\file.txt")
```

Exception in thread "main" java.util.regex.PatternSyntaxException: Unescaped trailing backslash near index 1  
\  
^

```
at java.base/java.util.regex.Pattern.error(Pattern.java:2204)  
at java.base/java.util.regex.Pattern.compile(Pattern.java:1951)  
at java.base/java.util.regex.Pattern.<init>(Pattern.java:1576)  
at java.base/java.util.regex.Pattern.compile(Pattern.java:1101)  
at java.base/java.lang.String.split(String.java:3352)  
at java.base/java.lang.String.split(String.java:3443)  
at com.example.Utills.countPathComponents(Utills.java:12)  
at com.example.Utills.main(Utills.java:21)
```

# Fix?

```
static int countPathComponents(String fileName) {  
    return (int) Pattern.compile(File.separator, Pattern.LITERAL)  
        .splitAsStream(fileName)  
        .filter(Predicate.not(String::isEmpty)).count();  
}
```

```
countPathComponents("C:\\tmp\\file.txt")
```

# Idiomatic?

```
static int countPathComponents(String fileName) {  
    return Path.of(fileName).getNameCount();  
}
```

# Idiomatic!

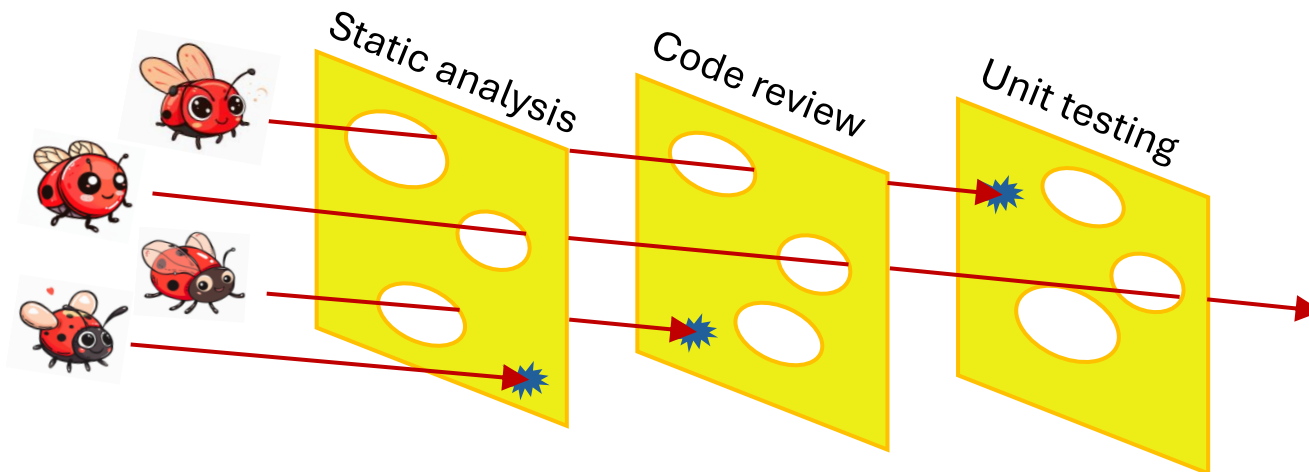
```
static int countPathComponents(String fileName) {  
    return Path.of(fileName).getNameCount();  
}
```



Rule (again): use proper types for your entities, and strong typing will help you to avoid bugs

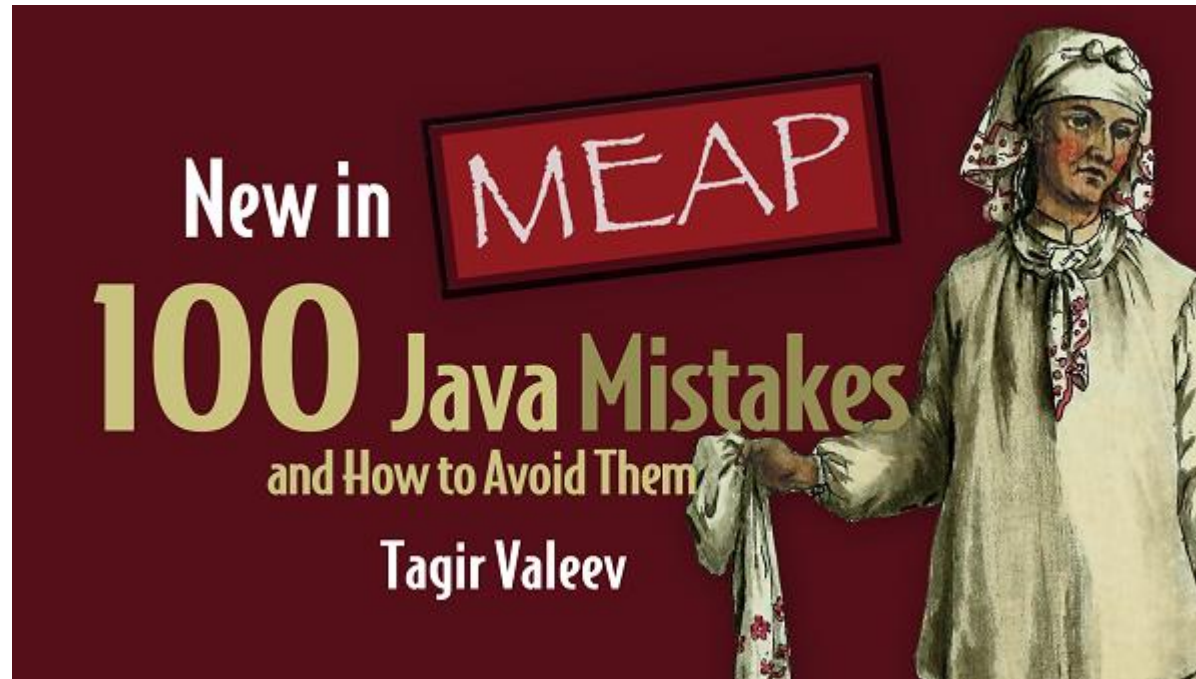
# Conclusion

- ✓ Use static analysis
- ✓ Write unit tests
- ✓ Clear constructs for every idiom
- ✓ Avoid repetitions
- ✓ Consult AI when in doubt but don't rely on it too much
- ✓ No premature optimizations
- ✓ Educate yourself, read books 😊



# Thank you





Promotion Code (45% off)  
100 Java Mistakes (both e-book and paper)  
<https://www.manning.com/books/100-java-mistakes-and-how-to-avoid-them>

**valeevmu2**