

Java 12-17 New Features

michael.schaffler@java.at



Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - Klasse CompactNumberFormat
 - Erweiterungen Klasse Files
 - Der Teeing Kollektor
- Java 13 & 14 Änderungen
 - Switch Expressions
 - Verbesserungen bei NullpointerExceptions



Übersicht

- Java 15/16/17 Änderungen
 - Text Blocks
 - Records
 - Pattern Matching bei instanceof
 - Sealed Classes
 - Hidden Classes



Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - Klasse CompactNumberFormat
 - Erweiterungen Klasse Files
 - Der Teeing Kollektor
- Java 13 & 14 Änderungen
 - Switch Expressions
 - Verbesserungen bei NullpointerExceptions



Microbenchmark Suite

- Eigene Benchmarks sehen oft so aus:

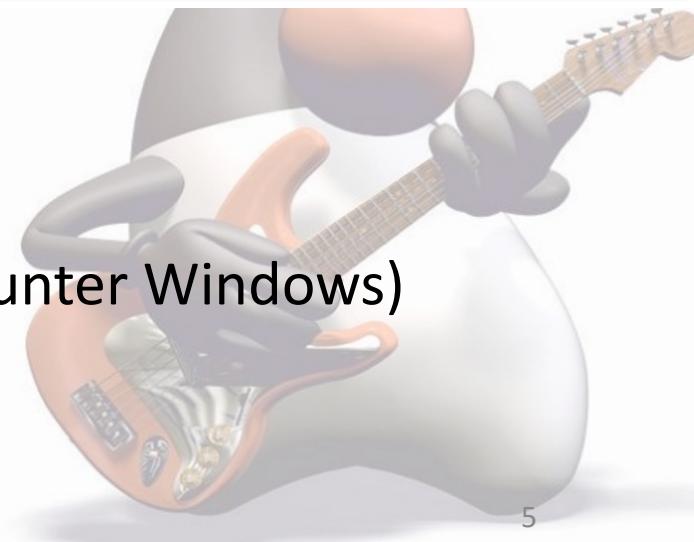
```
// ACHTUNG: keine gute Variante
final long startTimeMs = System.currentTimeMillis();

someCodeToMeasure();

final long stopTimeMs = System.currentTimeMillis();
final long duration = stopTimeMs - startTimeMs;
```

Probleme:

- Kein Warmlaufen
- Ungenauigkeit von System.currentTimeMillis (zumindest unter Windows)



Microbenchmark Suite

Bessere Variante mit Verwendung von System.nanoTime

```
// bessere Variante
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
    someCodeToMeasure();
}

final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```



Microbenchmark Suite

Noch besser mit Warmläufen des Codes:

```
// noch bessere Variante der Messung durch Warm-up
for (int i = 0; i < MAX_WARMUP_ITERATIONS; i++)
{
    someCodeToMeasure();
}

// eigentliche Messung nach Warm-up
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
    someCodeToMeasure();
}
final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

Microbenchmark Suite

Beste Lösung: JMH Framework mit Java 12

JMH arbeitet mit Annotations und integriert basierend darauf verschiedene Messungen. Hierzu bietet sich die Nutzung eines Maven-Archetypes an. Folgender Aufruf erzeugt ein neues Verzeichnis mit einem vollständig initialisierten JMH-Benchmark:

```
mvn archetype:generate \  
  -DinteractiveMode=false \  
  -DarchetypeGroupId=org.openjdk.jmh \  
  -DarchetypeArtifactId=jmh-java-benchmark-archetype \  
  -DgroupId=org.sample \  
  -DartifactId=jmh-test \  
  -Dversion=1.0-SNAPSHOT
```



Microbenchmark Suite

Beste Lösung: JMH Framework mit Java 12

```
public class MyBenchmark
{
    @Benchmark
    public void testMethod()
    {
        // This is a demo/sample template for building your JMH benchmarks.
        // Edit as needed. Put your benchmark code here.
    }
}
```

Um dann den Benchmark auszuführen, muss man in das durch den obigen Aufruf erzeugte Verzeichnis wechseln und dort – wie von Maven gewohnt – kompilieren und paketieren:

```
mvn package
```

```
java -jar target/benchmarks.jar
```

Microbenchmark Suite

Beste Lösung: JMH Framework mit Java 12

```
public class FirstIntegerBenchmark
{
    @Benchmark
    public String numberAsHexString()
    {
        final int number = 1_234_567;
        return Integer.toHexString(number);
    }

    @Benchmark
    public String numberAsBinaryString()
    {
        final int number = 123_456_789;
        return Integer.toBinaryString(number);
    }
}
```

Microbenchmark Suite

Beste Lösung: JMH Framework mit Java 12

```
public class FirstIntegerBenchmark
{
    @Benchmark
    public String numberAsHexString()
    {
        final int number = 1_234_567;
        return Integer.toHexString(number);
    }

    @Benchmark
    public String numberAsBinaryString()
    {
        final int number = 123_456_789;
        return Integer.toBinaryString(number);
    }
}
```

Microbenchmark Suite

Ausgabe

```
# JMH version: 1.23
# VM version: JDK 14, OpenJDK 64-Bit Server VM, 14+36-1461
# VM invoker: /Library/Java/JavaVirtualMachines/jdk-14.jdk/Contents/Home/bin/
#           java
# VM options: <none>
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.FirstIntegerBenchmark.numberAsBinaryString

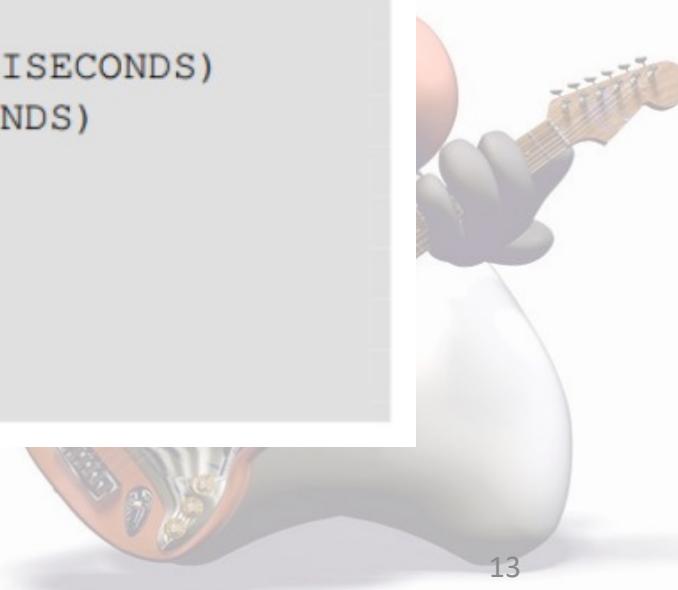
# Run progress: 0.00% complete, ETA 00:16:40
# Fork: 1 of 5
# Warmup Iteration  1: 56326650.457 ops/s
# Warmup Iteration  2: 59780600.908 ops/s
# Warmup Iteration  3: 64051264.973 ops/s
# Warmup Iteration  4: 64008812.168 ops/s
# Warmup Iteration  5: 64186426.089 ops/s
Iteration   1: 61304141.790 ops/s
Iteration   2: 61953075.846 ops/s
Iteration   3: 63036185.662 ops/s
Iteration   4: 64322247.337 ops/s
Iteration   5: 64872604.082 ops/s
...
...
```

Microbenchmark Suite

Möchte man den Benchmark noch ein wenig feintunen, beispielsweise die Anzahl an Iterationen oder das Warm-up und die Anzahl an Messdurchgängen konfigurieren, so könnte man dies folgendermaßen mithilfe der Annotations `@Warmup` sowie `@Measurement` tun:

```
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.annotations.Fork;

@Measurement(iterations = 3, time = 1000, timeUnit = TimeUnit.MILLISECONDS)
@Warmup(iterations = 7, time = 1000, timeUnit = TimeUnit.MICROSECONDS)
@Fork(4)
public class MyBenchmark
{
    ....
}
```



Microbenchmark Suite

- Weitere Annotationen:

`@State` – Man kann spezifizieren, wie der Zustand zwischen den Threads des Benchmarks geteilt werden soll. Im Thread-Scope erhält jeder Thread seinen unabhängigen Zustand. Hier nutzen wir den Zustand pro Benchmark: Damit ist dieser für alle Threads gleich.

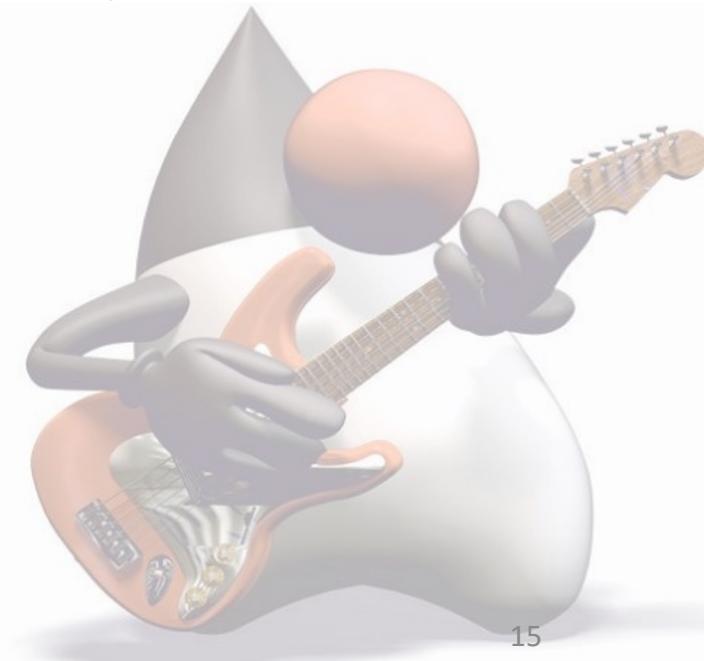
`@BenchmarkMode` – Oftmals ist zur Performance-Beurteilung die Anzahl an Operationen pro Zeiteinheit (`ops/time` = Default) oder die durchschnittliche Ausführungszeit (`time/ops`) von Interesse. Beides verwenden wir im diesem Beispiel.

`@OutputTimeUnit` – Mit JMH werden die Laufzeiten normalerweise in Nanosekunden berechnet. Mitunter ist man aber an anderen Zeitskalen interessiert, wie oben an Millisekunden für die Potenzierung.



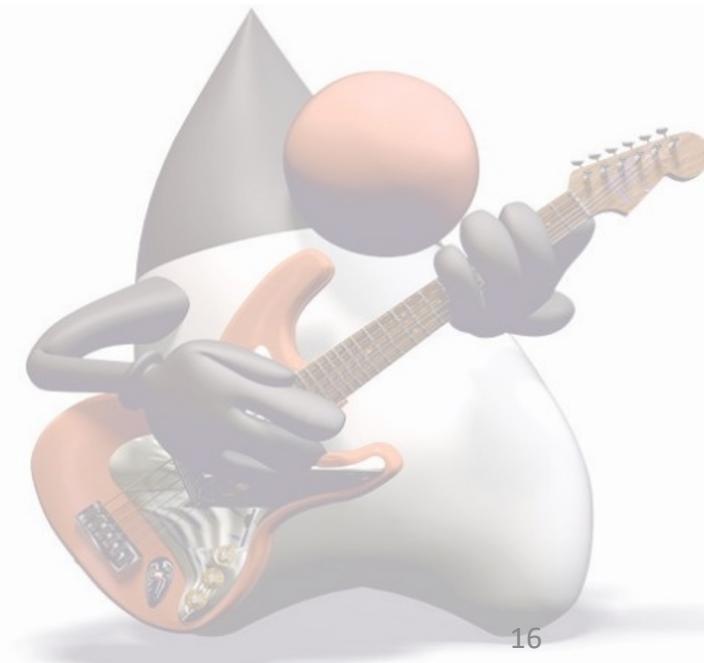
Übung 18

- Öffnen Sie die Microbenchmark Demo und experimentieren Sie mit den Beispielen
- Erstellen Sie eigene Benchmarks in dem Sie die Performance von ArrayList und LinkedList hinsichtlich unterschiedlicher Operationen untersuchen



Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - Klasse CompactNumberFormat
 - Erweiterungen Klasse Files
 - Der Teeing Kollektor
- Java 13 & 14 Änderungen
 - Switch Expressions
 - Verbesserungen bei NullpointerExceptions



Erweiterungen String

- Methode indent

```
private static void indentExample()
{
    var test = "Test".indent(4);
    printTextAndLength(test);

    var removeTest = "      6789".indent(-5);
    printTextAndLength(removeTest);
}

private static void printTextAndLength(final String removeTest)
{
    System.out.println("'" + removeTest + "'");
    System.out.println(removeTest.length());
}
```

Diese Methode gibt Folgendes aus:

```
'      Test
'
9
' 6789
'
5
```

← Linefeed wird eingefügt

Erweiterungen String

- Methode indent

Eine gleichmäßige Einrückung kann man für mehrzeilige Strings problemlos folgendermaßen realisieren:

```
private static void indentHowItShouldBeUsed()
{
    var header = "Report";
    var infoMessage = "This is a message\nThis is line 2\nLine3".indent(4);

    System.out.println(header);
    System.out.println(infoMessage);
}
```

Diese Methode gibt Folgendes aus:

```
Report
    This is a message
    This is line 2
    Line3
```

Erweiterungen String

- Methode transform

Nehmen wir an, wir wollten zunächst alle Zeichen eines Strings in Großbuchstaben wandeln, dann alle Zeichen T entfernen und schließlich den String in einzelne Bestandteile aufspalten. Bisher realisiert man das beispielsweise in mehreren Schritten mit jeweils neu entstehenden temporären Stringobjekten für die Zwischenschritte wie folgt:

```
private static void transformationOldStyle()
{
    var text = "This is a Test";

    var upperCase = text.toUpperCase();
    var noTs = upperCase.replaceAll("T", " ");
    var result = noTs.split(" ");

    System.out.println(Arrays.toString(result));
}
```

Erweiterungen String

- Methode transform

Mit Java 12 kann man diese Umwandlung etwas eleganter durch die Hintereinaderschaltung von `transform()` folgendermaßen schreiben:

```
private static void transformationJdk12Style()
{
    var text = "This is a Test";

    // chaining of operations
    var result = text.transform(String::toUpperCase).
                transform(str -> str.replaceAll("T", "")).
                transform(str -> str.split(" "));

    System.out.println(Arrays.toString(result));
}
```

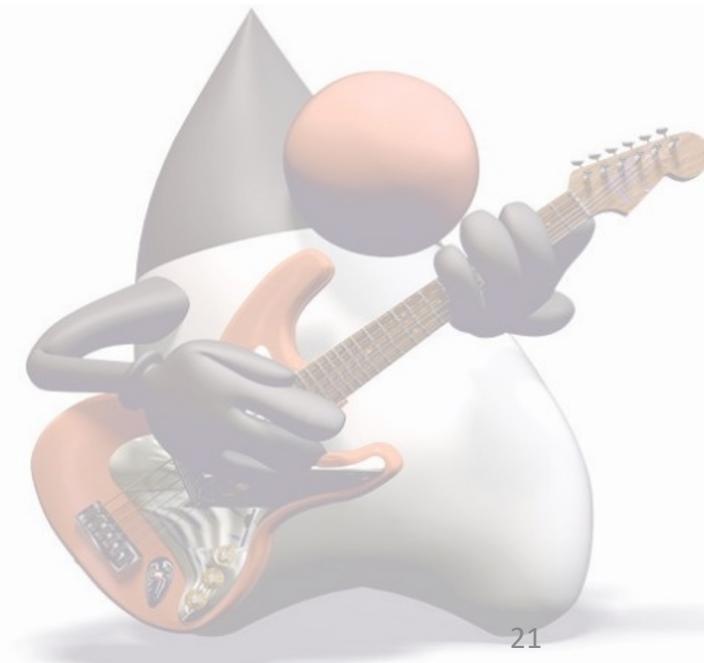


Beide Methoden geben Folgendes aus:

```
[HIS, IS, A, ES]
```

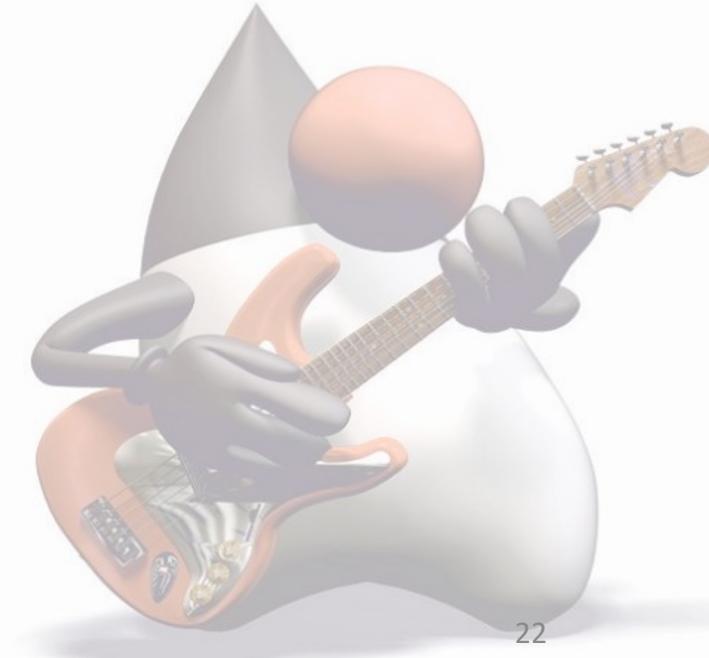
Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - **Klasse CompactNumberFormat**
 - Erweiterungen Klasse Files
 - Der Teeing Kollektor
- Java 13 & 14 Änderungen
 - Switch Expressions
 - Verbesserungen bei NullpointerExceptions



Neue Klasse CompactNumberFormat

In Java 12 wurde die Verarbeitung, Darstellung und Formatierung von Zahlen durch die Klasse `java.text.CompactNumberFormat` erweitert. Ziel dieser neuen Formatierungsklasse ist es, kürzere Zahlendarstellungen zu generieren: aus 100.000 wird dann etwa 100K.



Neue Klasse CompactNumberFormat

- Formatierung mit CompactNumberInstance

```
public static void main(final String args[])
{
    var shortFormat = getUsCompactNumberFormat(NumberFormat.Style.SHORT);
    formatNumbers("SHORT", shortFormat);

    var longFormat = getUsCompactNumberFormat(NumberFormat.Style.LONG);
    formatNumbers("LONG", longFormat);
}

private static NumberFormat getUsCompactNumberFormat(NumberFormat.Style style)
{
    return NumberFormat.getCompactNumberInstance(Locale.US, style);
}

private static void formatNumbers(final String style,
                                 final NumberFormat shortFormat)
{
    System.out.println("\nNumberFormat " + style);
    System.out.println("Result: " + shortFormat.format(10_000));
    System.out.println("Result: " + shortFormat.format(123_456));
    System.out.println("Result: " + shortFormat.format(1_234_567));
    System.out.println("Result: " + shortFormat.format(1_950_000_000));
}
```



NumberFormat SHORT
Result: 10K
Result: 123K
Result: 1M
Result: 2B
NumberFormat LONG
Result: 10 thousand
Result: 123 thousand
Result: 1 million
Result: 2 billion

Neue Klasse CompactNumberFormat

- Formatierung mit CompactNumberInstance

Diese Funktionalität kann in einigen Anwendungsfällen sicherlich praktisch sein. Ebenfalls nützlich ist, dass es entsprechende Methoden zum Parsing gibt:

```
public static void main(final String[] args) throws ParseException
{
    System.out.println("US/SHORT parsing:");
    var shortFormat = NumberFormat.getCompactNumberInstance(Locale.US,
                                                               Style.SHORT);
    System.out.println(shortFormat.parse("1 K")); // ACHTUNG: wird als 1 erkannt
    System.out.println(shortFormat.parse("1K"));
    System.out.println(shortFormat.parse("1M"));
    System.out.println(shortFormat.parse("1B"));

    System.out.println("\nUS/LONG parsing:");
    var longFormat = NumberFormat.getCompactNumberInstance(Locale.US,
                                                               Style.LONG);
    System.out.println(longFormat.parse("1 thousand"));
    System.out.println(longFormat.parse("1 million"));
    System.out.println(longFormat.parse("1 billion"));
}
```

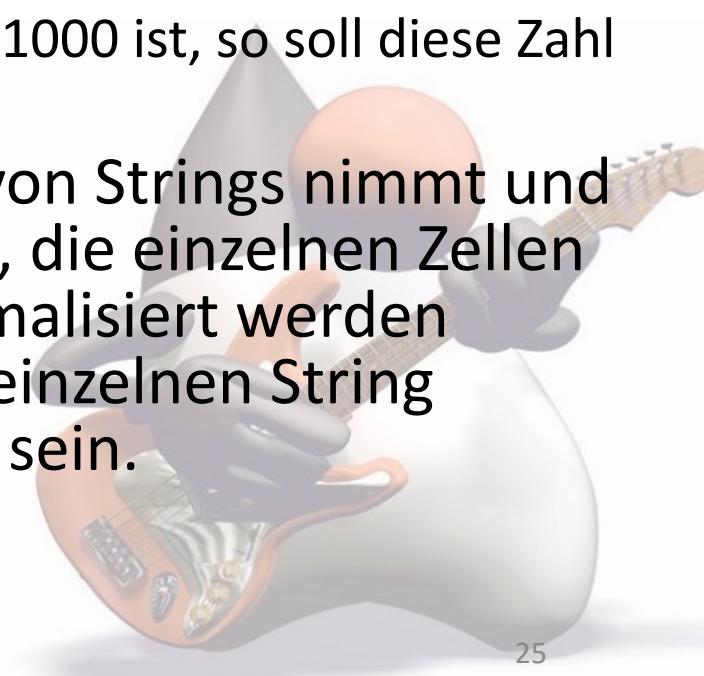


```
US/SHORT parsing:
1
1000
1000000
1000000000

US/LONG parsing:
1000
1000000
1000000000
```

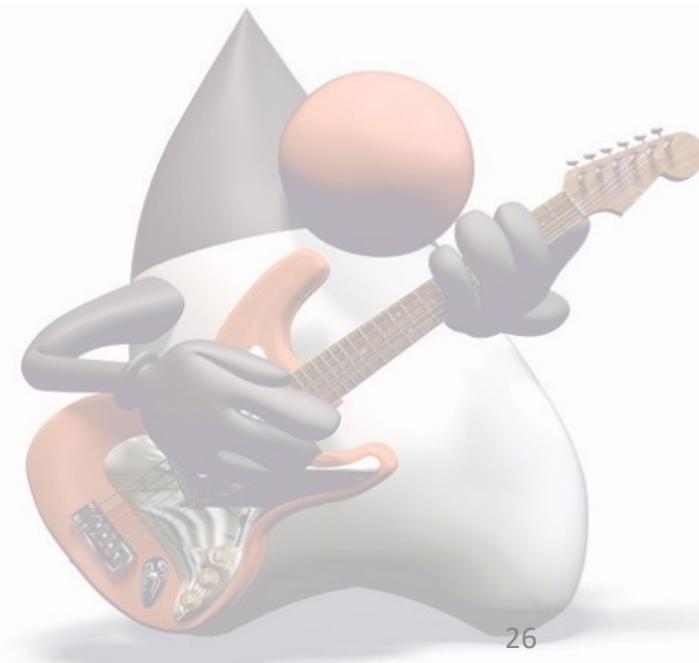
Übung 19

- Schreibe eine Methode „String normalizeCsvCell(String s)“ unter Verwendung der String.transform Methode, die
 - führende und nachfolgende Whitespaces entfernt,
 - nullwerte in Leerstrings umwandelt,
 - alle Semikolons in Kommas umwandelt.
 - Falls ein String nur eine ganzzahlige Zahl beinhaltet, die ≥ 1000 ist, so soll diese Zahl in ein Kurzformat konvertiert werden.
- Schreibe eine Methode toCsvLine, die einen Stream von Strings nimmt und einen String zurückliefert. Im Stream von Strings sind, die einzelnen Zellen der csv Zeile, die zuerst mittels normalizeCsvCell normalisiert werden sollen. Danach sollen die einzelnen Strings zu einem einzelnen String zusammengehängt werden, das Trennzeichen soll „;“ sein.



Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - Klasse CompactNumberFormat
 - **Erweiterungen Klasse Files**
 - Der Teeing Kollektor
- Java 13 & 14 Änderungen
 - Switch Expressions
 - Verbesserungen bei NullpointerExceptions



Erweiterungen Klasse Files

- Methode mismatch

Betrachten wir eine kleine Beispielapplikation zum besseren Verständnis: Nachfolgend wollen wir mit Funktionalitäten aus Java 11 textuelle Informationen in temporäre Dateien schreiben und deren Inhalte dann mithilfe von `mismatch()` vergleichen:

```
private static void compareSameContent() throws IOException
{
    Path filePath1 = Files.createTempFile("test1", ".txt");
    Path filePath2 = Files.createTempFile("test2", ".txt");

    Files.writeString(filePath1, "Same Content");
    Files.writeString(filePath2, "Same Content");

    long pos = Files.mismatch(filePath1, filePath2);
    System.out.println("Same content: mismatch(File1, File2) = " + pos);
}
```

Beim Ausführen der obigen Programmzeilen kommt es zu folgenden Ausgaben:

```
Same content: mismatch(File1, File2) = -1
Different content: mismatch(File1, File2) = 5
```

Erweiterungen Klasse Files

- Methode mismatch mit unterschiedlichen Encodings

```
private static void compareDifferentEncodings() throws IOException
{
    var fileLatin1 = Files.createTempFile("encl", ".latin1");
    var fileUtf8 = Files.createTempFile("enc2", ".utf8");

    var msg = "Zürich is beautiful. Mainz too";
    Files.writeString(fileEncl, msg, StandardCharsets.ISO_8859_1);
    Files.writeString(fileUtf8, msg, StandardCharsets.UTF_8);

    var pos = Files.mismatch(fileLatin1, fileUtf8);
    System.out.println("Different encoding: mismatch(File1, File2) = " + pos);
}
```

Die beiden Texte sind bezogen auf die Buchstaben identisch, sie weichen jedoch durch die Kodierung des Umlauts ü an Position 1 auf Byte-Ebene voneinander ab. Daher erhalten wir folgende Ausgabe (unter Windows erhält man eine 2 – aufgrund der zu Unix-Systemen abweichenden Byte-Reihenfolge):

```
Different encoding: mismatch(File1, File2) = 1
```

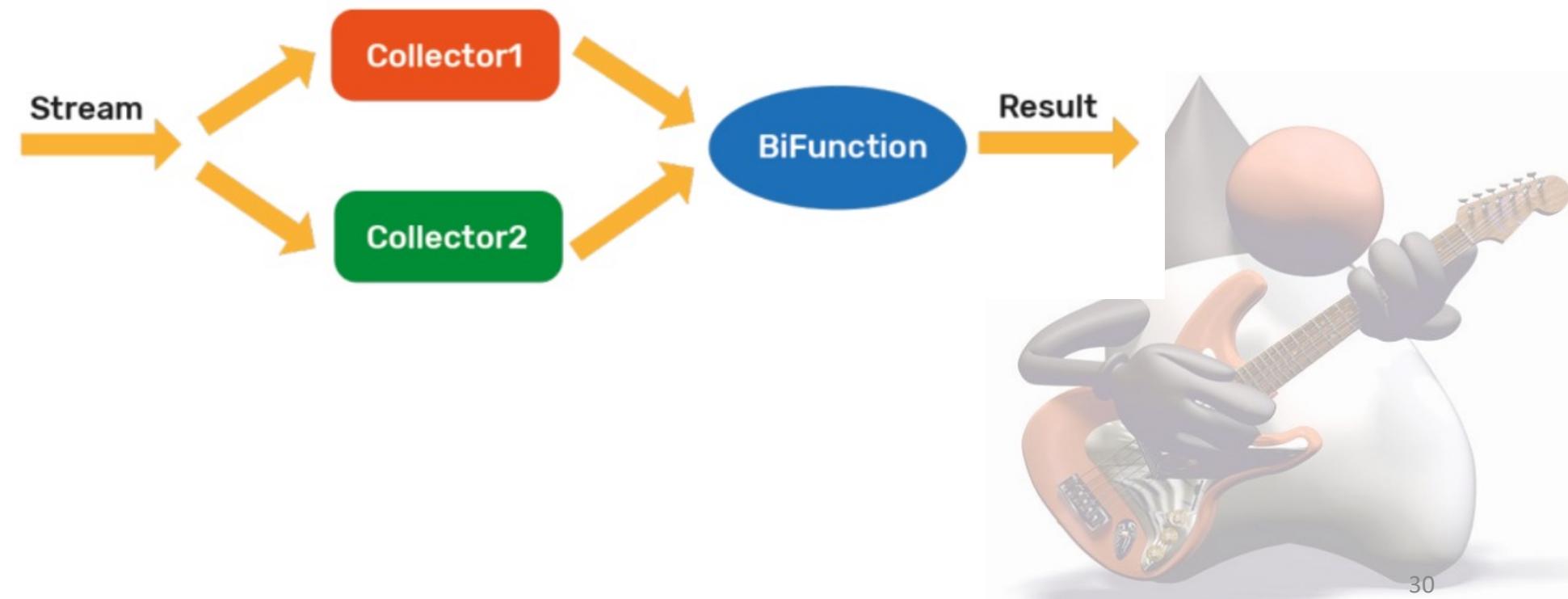
Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - Klasse CompactNumberFormat
 - Erweiterungen Klasse Files
 - **Der Teeing Kollektor**
- Java 13 & 14 Änderungen
 - Switch Expressions
 - Verbesserungen bei NullpointerExceptions



Teeing Kollektor

- „Teeing“ stammt vom T-Stück einer Rohrverbindung
- Es können dabei mehrere Kollektoren eines Streams zusammengeführt werden



Teeing Kollektor

Nehmen wir an, es sollen die Zahlenwerte eines Streams in einem Rutsch sowohl summiert als auch deren Anzahl ermittelt werden. Das lässt sich mithilfe des Teeing-Kollektors und zweier weiterer Kollektoren recht einfach formulieren. Als Ausgangsbasis dienen zwei Streams aus Zahlen. Die gerade beschriebene Funktionalität wird in der Methode `calcCountAndSum()` realisiert – deren Lesbarkeit erhöht sich durch den statischen Import der fett markierten Kollektoren:

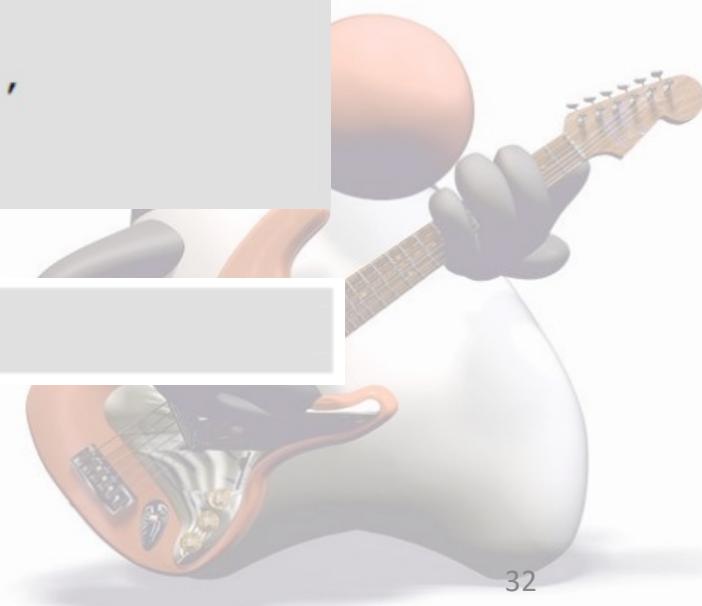
```
public static void main(final String[] args)
{
    System.out.println(calcCountAndSum(Stream.of(1, 2, 3, 4, 5, 6)));
    System.out.println(calcCountAndSum( Stream.of(1, 3, 5, 7, 11, 13, 17)));
}

private static Pair<Long> calcCountAndSum(Stream<Integer> numbers)
{
    return numbers.collect(teeing(
        counting(),
        summingLong(n -> n),
        (count, sum) -> new Pair<Long>(count, sum)));
}
```

Teeing Kollektor

- Weiteres Beispiel: Zusammenführen zweier Listen

```
import static java.util.stream.Collectors.*  
  
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd", "Carsten");  
  
final Predicate<String> startsWithMi = text -> text.startsWith("Mi");  
final Predicate<String> endsWithM = text -> text.endsWith("m");  
  
// Wäre var hier nicht wünschenswert? => siehe Text  
final BiFunction<List<String>, List<String>, List<List<String>>> combineLists =  
    (list1, list2) -> List.of(list1, list2);  
  
var result = names.collect(teeing(filtering(startsWithMi, toList()),  
                                filtering(endsWithM, toList()),  
                                combineLists));  
  
result ==> [[Michael, Mike], [Tim, Tom]]
```

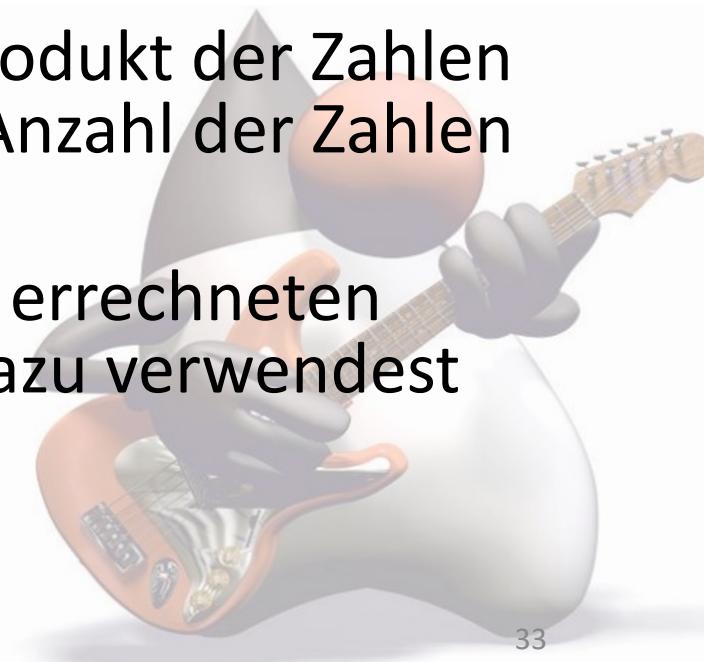


Übung 20

- Generiere eine Reihe von 10 Zufallszahlen vom Typ double zwischen 1 und 10.
- Berechne das geometrische Mittel.

Tipps

- Verwende dazu einen Reducing Collector, der das Produkt der Zahlen berechnet, sowie einen Counting Collector, der die Anzahl der Zahlen feststellt.
- Verwende eine Teeing Collector, der aus den beiden errechneten Werten dann das geometrische Mittel berechnet. Dazu verwendest Du am Besten die Klasse Math.



Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - Klasse CompactNumberFormat
 - Erweiterungen Klasse Files
 - Der Teeing Kollektor
- Java 13 & 14 Änderungen
 - **Switch Expressions**
 - Verbesserungen bei NullpointerExceptions



Erweiterung Switch Statement

Klassisches Switch

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters;
switch (day)
{
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numOfLetters = 6;
        break;
    case TUESDAY:
        numOfLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numOfLetters = 8;
        break;
    case WEDNESDAY:
        numOfLetters = 9;
        break;
    default:
        numOfLetters = -1;
}
```

Switch Expressions mit Java 14

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                    -> 7;
    case THURSDAY, SATURDAY         -> 8;
    case WEDNESDAY                  -> 9;
};
```



Erweiterung Switch Statement

- Der Compiler erkennt nun, dass nicht alle Werte abgedeckt sind
- Der folgende Code führt zu einem Compile-Error „A Switch expression should cover all possible values“

```
String numericString = switch (value)
{
    case 1 -> "one";
    case 2 -> "two";
    case 3 -> "three";
    // default -> "N/A";
};
```



Erweiterung Switch Statement

- Switch Expression mit enum

```
static String monthToName(final Month month)
{
    return switch (month)
    {
        case JANUARY -> "January";
        default -> "N/A"; // hier KEIN Fall Through
        case FEBRUARY -> "February";
        case MARCH -> "March";
    };
}
```

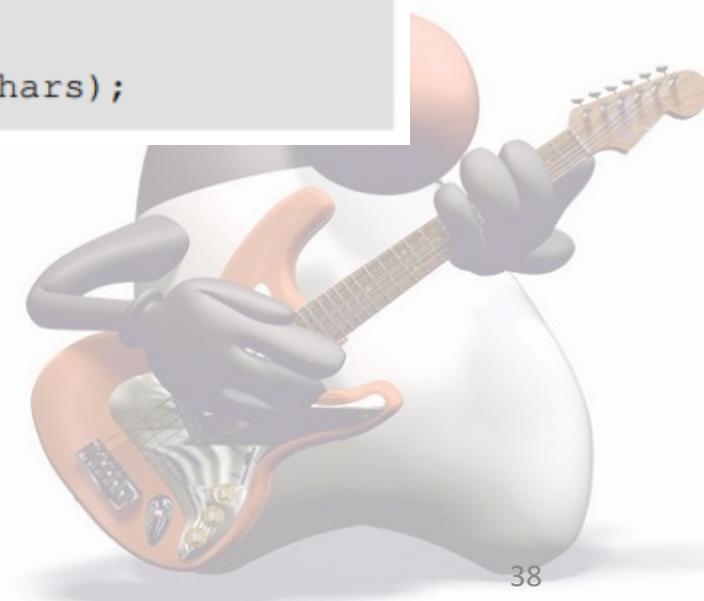


Erweiterung Switch Statement

- Switch Expression mit yield
- Syntax ähnelt der vorherigen Syntax mit ->

```
int numOfChars = switch (color)
{
    case RED: yield 3;
    case BLUE: yield 4;
    case GREEN: yield 5;
    case YELLOW, ORANGE: yield 6;
};

System.out.println("Color " + color.name() + " chars: " + numOfChars);
```



Erweiterung Switch Statement

- Yield muss verwendet werden, wenn ein case einer Switch-Expression mehrere Anweisungen enthält

```
DayOfWeek day = DayOfWeek.FRIDAY;

int numOfLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> {
        if (day == DayOfWeek.SUNDAY)
            System.out.println("SUNDAY is FUN DAY");

        yield 6;
    }
    case TUESDAY                  -> 7;
    case THURSDAY, SATURDAY       -> 8;
    case WEDNESDAY                -> 9;
};

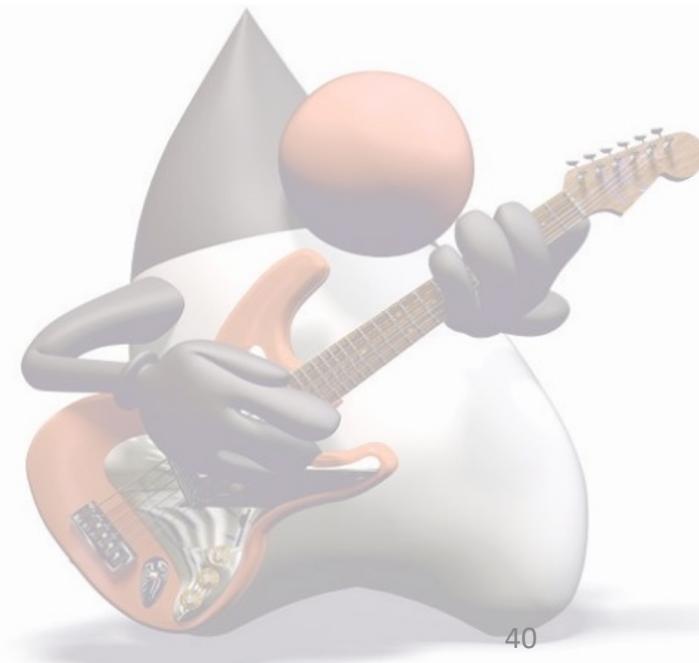
System.out.println(numOfLetters);
```



Erweiterung Switch Statement

Switch ohne return Wert

```
String s = "Hallo";
switch (s) {
    case "Hallo" -> {
        System.out.println("Auch hallo");
    }
    case "Wie geht es Dir?" -> {
        System.out.println("Gut!");
    }
    //no default case necessary
}
```



Übung 21

- Schreibe ein Switch Statement, dass als Eingang die Anzahl der Kinder nimmt und daraus die Höhe des Kindergeldes zurückliefert.
- Die Berechnung ist wie folgt:
 - 1 Kind 400
 - 2 Kinder 820
 - 3 Kinder 860
 - Ab 4tem Kind: $860 + ((\text{kinder} - 3) * 400)$



Übersicht

- Java 12 Änderungen
 - Microbenchmark Suite
 - Erweiterungen Klasse String
 - Klasse CompactNumberFormat
 - Erweiterungen Klasse Files
 - Der Teeing Kollektor
- Java 13 & 14 Änderungen
 - Switch Expressions
 - Verbesserungen bei NullpointerExceptions



Verbesserungen bei NullpointerExceptions

- Beispielcode mit NullPointerException

```
public class NPE_Example
{
    public static void main(final String[] args)
    {
        A a = null;
        a.value = "ERROR";
    }
}
```

- Bisherige Fehlermeldung

```
Exception in thread "main" java.lang.NullPointerException
at java14.NPE_Example.main(NPE_Example.java:8)
```

- Fehlermeldung ab Java 14

```
Exception in thread "main" java.lang.NullPointerException:
    Cannot assign field 'value' because 'a' is null.
at java14.NPE_Example.main(NPE_Example.java:8)
```



Damit diese Informationen aufbereitet werden, bedarf es der Aktivierung durch den JVM-Parameter `-XX:+ShowCodeDetailsInExceptionMessages`.

Verbesserungen bei NullpointerExceptions

- Beispielcode: getWindow(5) liefert null

```
int width = getWindowManager().getWindow(5).size().width();
```

- Fehlermeldung ab Java 14

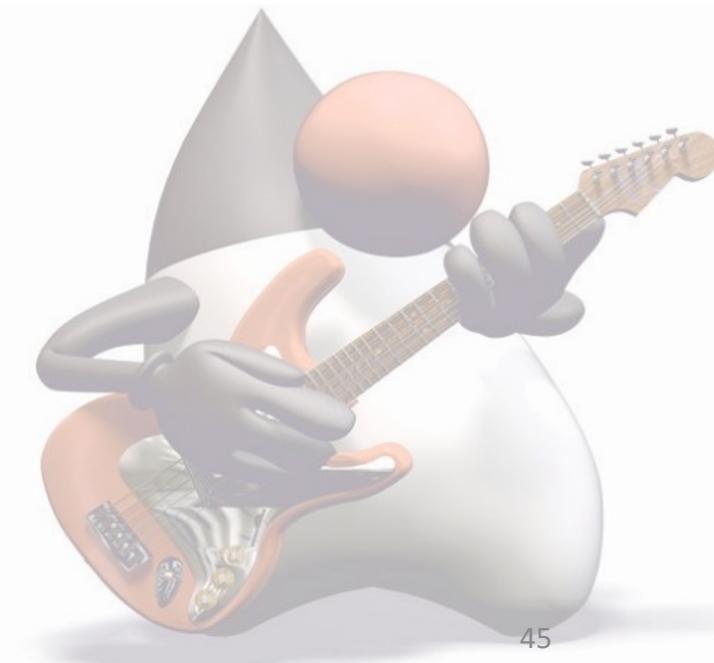
```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "java14
.NPE_Third_Example$Window.size()" because the return value of "java14.
NPE_Third_Example$WindowManager.getWindow(int)" is null
at java14.NPE_Third_Example.main(NPE_Third_Example.java:24)
```



Damit diese Informationen aufbereitet werden, bedarf es der Aktivierung durch den JVM-Parameter `-XX:+ShowCodeDetailsInExceptionMessages`.

Übersicht

- Java 15/16/17 Änderungen
 - Text Blocks
 - Records
 - Pattern Matching bei instanceof
 - Sealed Classes
 - Hidden Classes



Textblocks

Text Blocks beschreiben mehrzeilige Strings. Diese beginnen und enden mit je drei Anführungszeichen, wobei der eigentliche Inhalt auf einer neuen Zeile beginnen muss:

```
System.out.println("""  
    -Ich bin ein-  
    -Text Block-  
""");
```

Bei der Ausgabe kommt es zu einer Einrückung, die sich durch die Einrückung der abschließenden “““ ergibt.

```
-Ich bin ein-  
-Text Block-
```

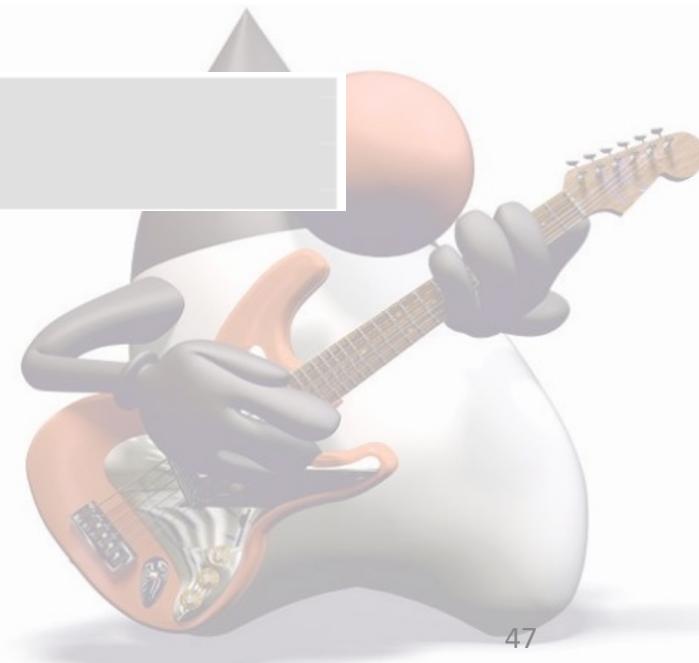


Textblocks

Sofern keine Einrückung benötigt wird, ist es oftmals besser, den Abschluss des Text Blocks mit in die letzte Zeile aufzunehmen:

```
var firstTextBlockNoIndentation = """  
    -Ich bin ein-  
    -Text Block""";  
System.out.println(firstTextBlockNoIndentation);
```

```
-Ich bin ein-  
-Text Block-
```



Textblocks

Escaping von “““:

```
System.out.println("""  
    First 'line' simple quotes  
    Second "line" double quotes  
    Third line \""" three quotes  
    Fourth line no quotes, just \\ :-) """);
```

Die Ausgabe sieht folgendermaßen aus:

```
First 'line' simple quotes  
Second "line" double quotes  
Third line """ three quotes  
Fourth line no quotes, just \\ :-)
```



Textblocks

Platzhalter mit Methode `formatted`:

```
String placeholders = """
    Michael %s hat am "%tF"
    %d Bücher in '%s' gekauft.
    """ .formatted("Inden", LocalDate.of(2020, 1, 20), 7, "Bremen");
```

Durch diese Anweisungen wird folgende Ausgabe produziert:

```
Michael Inden hat am "2020-01-20"
7 Bücher in 'Bremen' gekauft.
```



Textblocks

Erweiterung mit Java 14

```
String text = """  
    This is a string splitted \  
    in several smaller \  
    strings.\ \  
""";
```

Damit wird folgender einzeiliger String erzeugt:

```
This is a string splitted in several smaller strings.
```

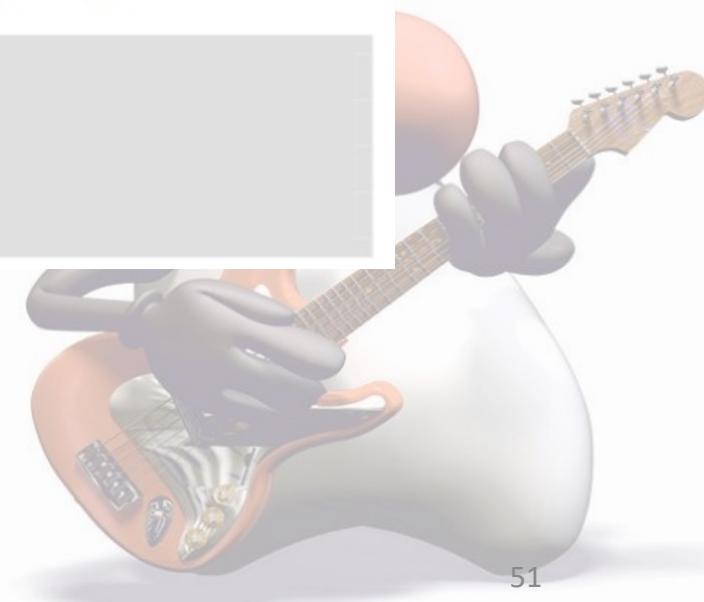


Textblocks

Erweiterung mit Java 14

Escape-Sequenz \s Die zweite Escape-Sequenz \s wird einfach in ein Leerzeichen umgewandelt und verhindert, dass Leerzeichen in einem Text Block am Ende abgeschnitten werden. Das Standardverhalten entfernt überschüssige Leerzeichen am Ende eines Strings und fügt direkt den Zeilenumbruch hinzu. Mit der neuen Escape-Sequenz kann man beispielsweise erreichen, dass alle Zeichenketten die gleiche Länge besitzen. Nehmen wir dazu an, wir wollten die Namen alle gleich lang begrenzen:

```
System.out.println("""  
    Tim \s  
    Peter\s  
    Mike \s""");
```



Übersicht

- Java 15/16/17 Änderungen
 - Text Blocks
 - Records
 - Pattern Matching bei instanceof
 - Sealed Classes
 - Hidden Classes



Records

- ... sind Datentypen, die von Haus aus unmodifiable sind (siehe auch: <https://ciit.at/immutable-objects-in-java-14-neu-gemacht/>)
- ... haben automatisch für jedes Feld eine Accessor Methode
- ... implementieren automatisch equals und hashCode
- ... implementieren automatisch toString
- ... implementieren automatisch einen Konstruktor



Records

- Beispiele

```
record MyPoint(int x, int y) { }
```

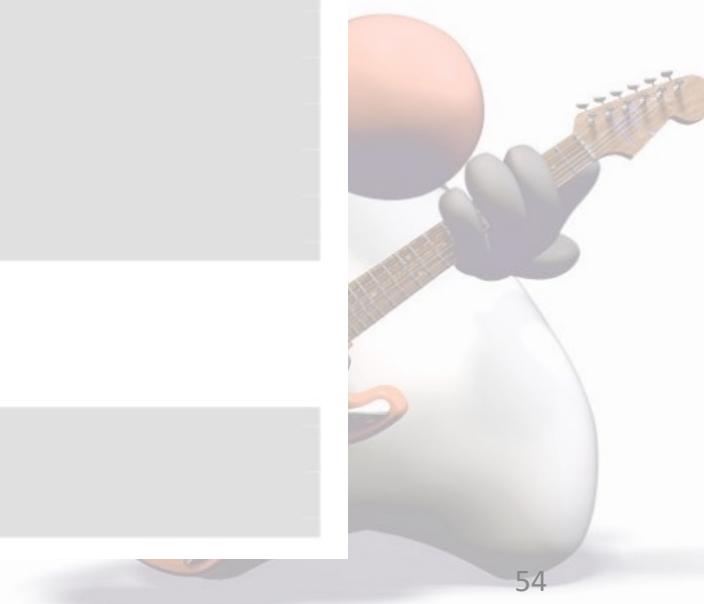
```
public class MyPointClient
{
    public static void main(final String[] args)
    {
        final MyPoint myPoint = new MyPoint(47, 11);

        System.out.println(myPoint);
        System.out.println(myPoint.x());
        // System.out.println(myPoint.x);    // nicht zugreifbar
    }
}
```

Diese Zeilen führen zu folgender Ausgabe:

```
MyPoint [x=47, y=11]
```

```
47
```



Records

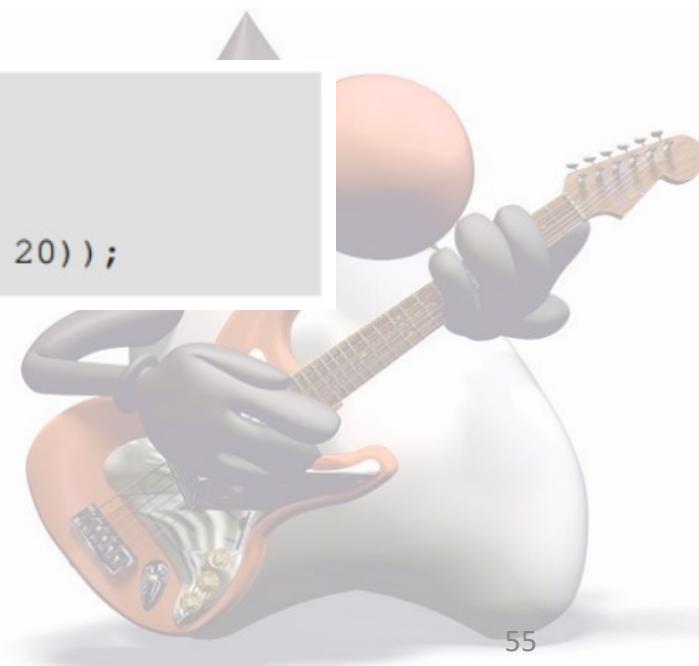
- Weitere Beispiele

```
record Point(int x, int y) { }
record TopLeftWidthAndHeight(Point topLeft, int width, int height) { }

record ColorAndRgbDTO(String name, int red, int green, int blue) { }
record PersonDTO(String firstname, String lastname, LocalDate birthday) { }
```

```
var oldStyle = new Rectangle(10, 10, 70, 30);

var topLeft = new Point(10, 10);
var newStyle = new Rectangle(new TopLeftWidthAndHeight(topLeft, 60, 20));
```

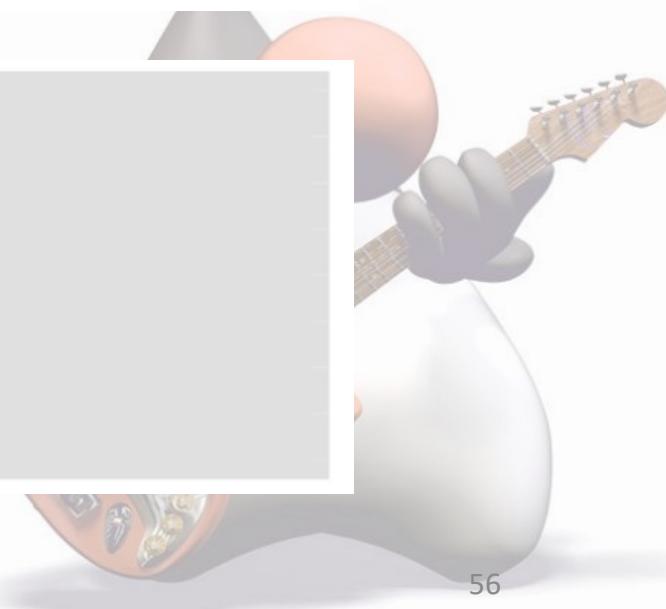


Records

- Records können auch eigene Methoden haben:

```
record PersonDTO(String firstname, String lastname, LocalDate birthday)
{
    public String asFullname()
    {
        return firstname + " " + lastname;
    }
}
```

```
record MyPoint(int x, int y)
{
    @Override
    public String toString()
    {
        return String.format("(x: %d,y: %d)", x, y);
    }
}
```



Records

- Es können eigene Konstruktoren geschrieben werden

```
record MyPoint(int x, int y)
{
    public MyPoint(String values)
    {
        this(Integer.parseInt(values.split(",")[0]),
              Integer.parseInt(values.split(",")[1]));
    }
}
```

Das erlaubt dann folgende Konstruktion:

```
MyPoint myPoint = new MyPoint("72,71");
```



Records

- Es können eigene Konstruktoren geschrieben werden

Interessanterweise gibt es eine syntaktische Besonderheit: Man kann anstelle eines Konstruktors auch folgendes Konstrukt verwenden – dadurch erfolgen die gleichen Prüfungen, aber man muss keine Zuweisungen der Variablen von Hand vornehmen:

```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval
    {
        if (lower > upper)
        {
            String errorMsg = String.format("invalid: %d (lower) >= %d (upper)", lower, upper);
            throw new IllegalArgumentException(errorMsg);
        }
    }
}
```

Übersicht

- Java 15/16/17 Änderungen
 - Text Blocks
 - Records
 - **Pattern Matching bei instanceof**
 - Sealed Classes
 - Hidden Classes



Pattern Matching bei instanceof

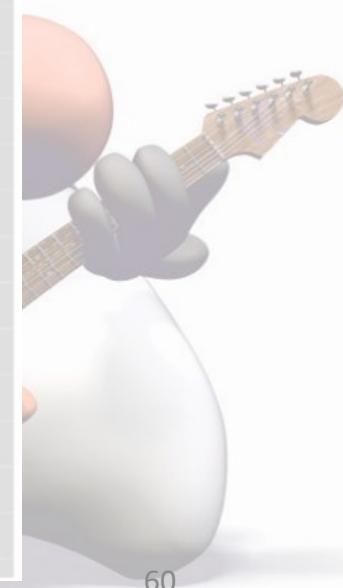
- Bisher:

```
if (obj instanceof Person)
{
    final Person person = (Person) obj;
    // ... Zugriff auf person...
}
```

- Ab Java 14 gibt es einen direkten Zugriff auf den geprüften Objekttyp:

```
Object obj = "Hallo Java 14";

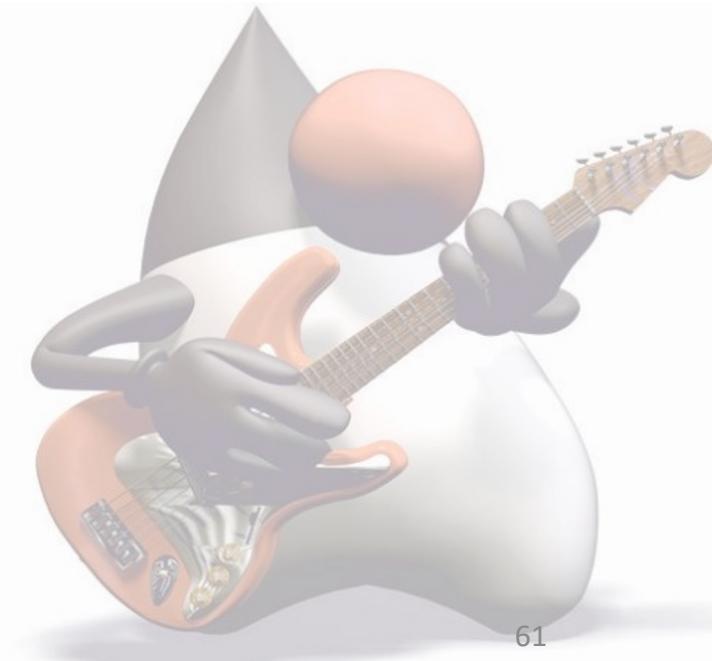
if (obj instanceof String str)
{
    // Hier kann man str nutzen
    System.out.println("Länge: " + str.length());
}
else
{
    // Hier kein Zugriff auf str
    System.out.println(obj.getClass());
}
```



Pattern Matching bei instanceof

- Auch möglich:

```
if (obj instanceof String str2 && str2.length() > 5)
{
    System.out.println("Länge: " + str2.length());
}
```



Übersicht

- Java 15/16/17 Änderungen
 - Text Blocks
 - Records
 - Pattern Matching bei instanceof
 - **Sealed Classes**
 - Hidden Classes



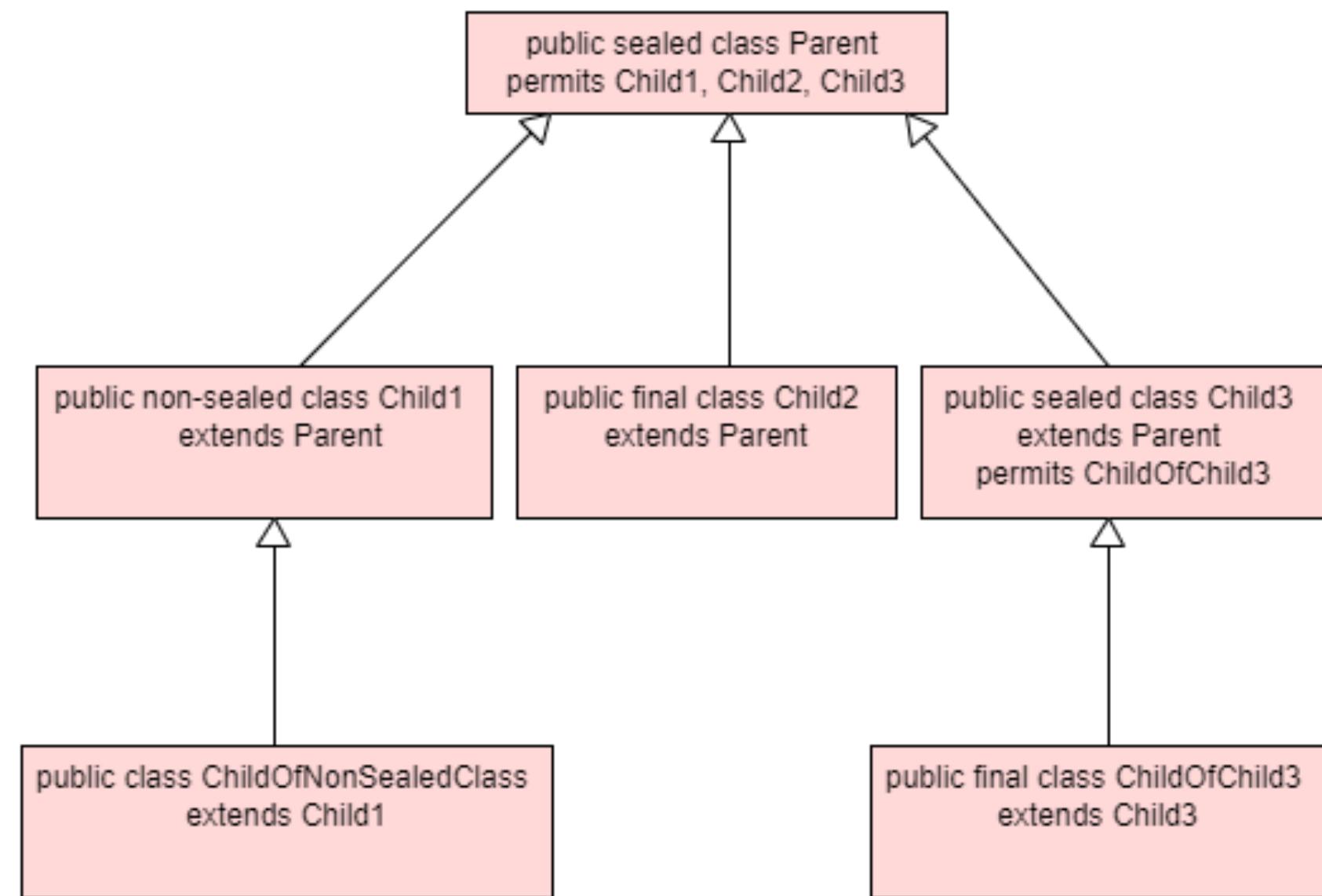
Sealed Classes & Interfaces

- Sealed Classes sind Klassen, die nur von bestimmten Klassen erweitert werden dürfen.
- Welche Klassen diese erweitern dürfen, muss mittels des neuen Schlüsselworts „permits“ aufgelistet werden.
- Eine Sealed Class muss Subklassen haben.
- Eine Subklasse einer Sealed Class muss selbst entweder als final, non-sealed oder sealed deklariert sein.

```
public sealed class Parent permits Child1, Child2, Child3{  
}
```



Sealed Classes



Sealed Interfaces

- Auch interfaces können Sealed sein.

```
public sealed interface Car permits ElectricCar, PetrolCar,  
DieselCar, HydrogenCar{
```

```
}
```

```
final class ElectricCar implements Car{
```

```
}
```

```
non-sealed class PetrolCar implements Car{
```

```
}
```

```
final class DieselCar implements Car{
```

```
}
```

```
final class HydrogenCar implements Car{
```

```
}
```

Sealed Classes & Interfaces

- Preview Feature in Java 17 (compiler switch --enable-preview):
 - Pattern Matching in switch

```
Car car = new PetrolCar();
String statement = switch (car) {
    case PetrolCar pc-> "Too much CO2";
    case DieselCar dc-> "Too much soot";
    case HydrogenCar hc-> "Too little efficiency";
    case ElectricCar ec-> "Too resource intensive";
};
```

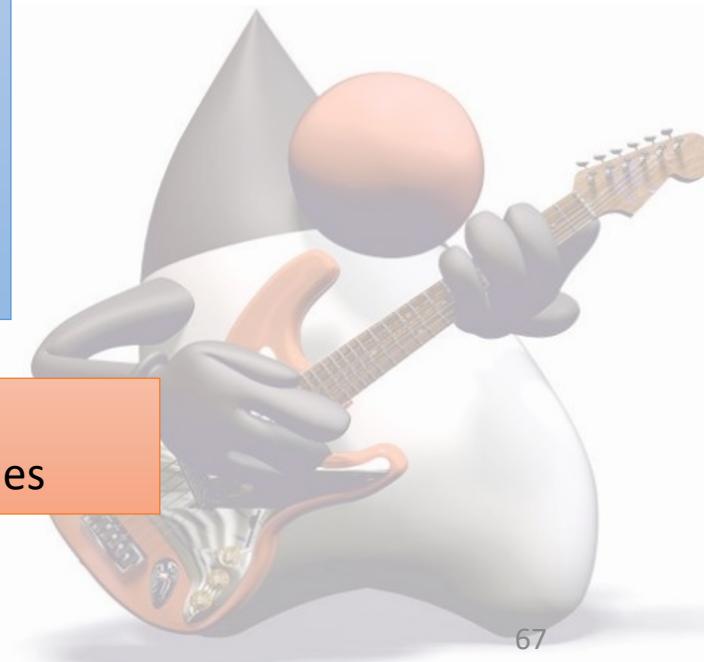


Sealed Classes & Interfaces

- Preview Feature in Java 17 (compiler switch --enable-preview):
 - Mit Sealed Classes können bei switch die möglichen Werte hergeleitet werden

```
Car car = new PetrolCar();
String statement = switch (car) {
    case PetrolCar pc-> "Too much CO2";
    case DieselCar dc-> "Too much soot";
    case HydrogenCar hc-> "Too little efficiency";
    //case ElectricCar ec-> "Too resource intensive";
};
```

COMPILATION ERROR :
the switch expression does not cover all possible input values



Übersicht

- Java 15/16/17 Änderungen
 - Text Blocks
 - Records
 - Pattern Matching bei instanceof
 - Sealed Classes
 - **Hidden Classes**



Hidden classes

- Hidden Classes sind während der Laufzeit dynamisch generierte Klasse
- Sie können durch andere Klassen nicht über den Classloader „entdeckt“ werden.
- Sie werden von Frameworks verwendet, um während der Laufzeit Klassen zu generieren, die nur durch das Framework verwendet werden sollen.
- Die bisher existierenden Möglichkeiten dynamische Klasse zu generieren (ClassLoader::defineClass und Lookup::defineClass) haben den Nachteil, dass die so generierten Klassen für andere sichtbar sind.

Hidden classes

```
public static void main(String[] args) throws IllegalAccessException {  
    MethodHandles.Lookup lookup = MethodHandles.lookup();  
    byte[] classInByteCodeFormat= ...;  
    final Class<?> hiddenClass = lookup.defineHiddenClass(classInByteCodeFormat, true).lookupClass();  
    final Constructor<?>[] constructors = hiddenClass.getConstructors();  
    final Object newInstance = constructors[0].newInstance(null);  
    //Assume hiddenclass is implementing the Runnable interface  
    Runnable runnable = (Runnable)newInstance;  
    runnable.run();  
}
```



Java 12-17 New Features

michael.schaffler@java.at

