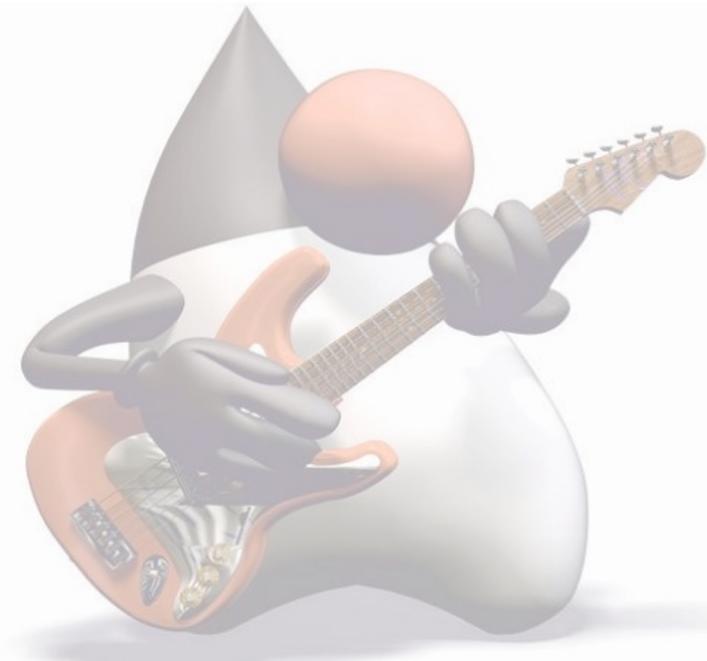


# Java 9-11 New Features



# Trainer

Michael Schaffler

[michael@java.at](mailto:michael@java.at)



# Kurze Vorstellung

- Momentane Tätigkeit
- Angestrebtes Wissen
- Erwartungen an die Schulung



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



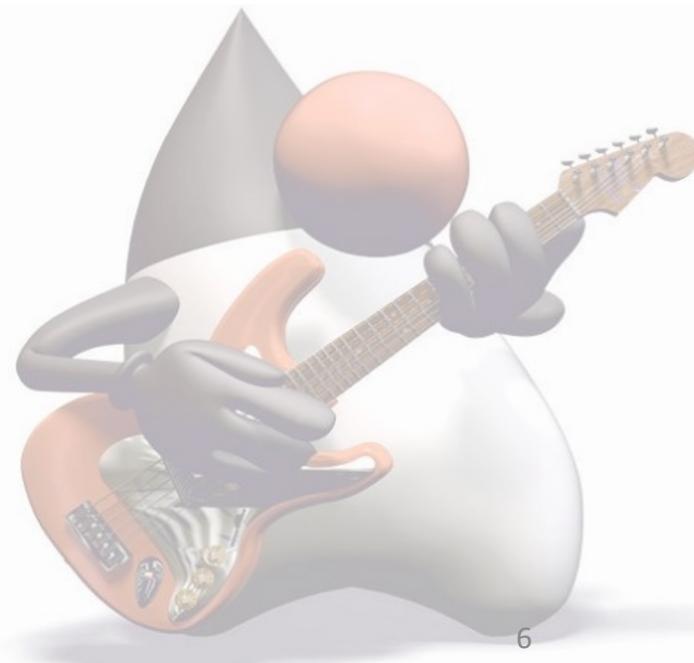
# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - Unmodifiable Lists
  - Erweiterungen Optional<T>
  - Erneute Änderungen Versionierung
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas



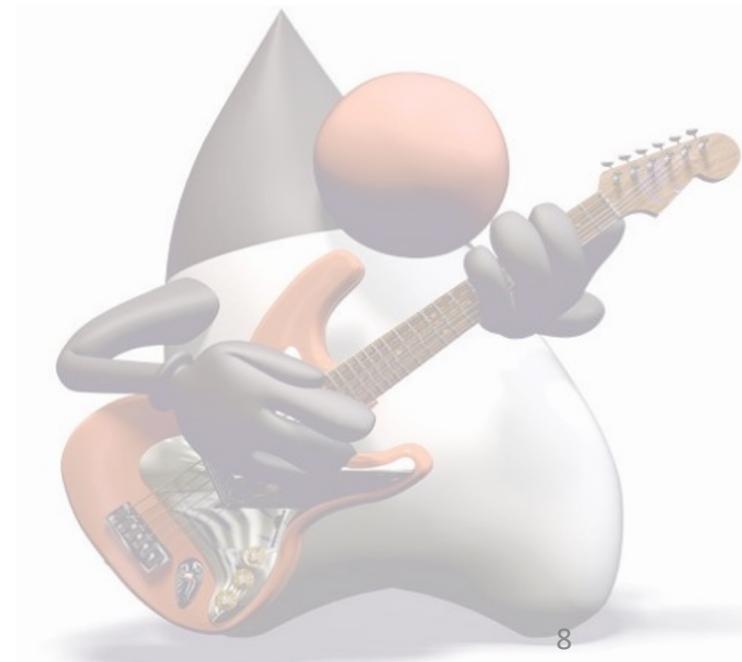
# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



# Übersicht

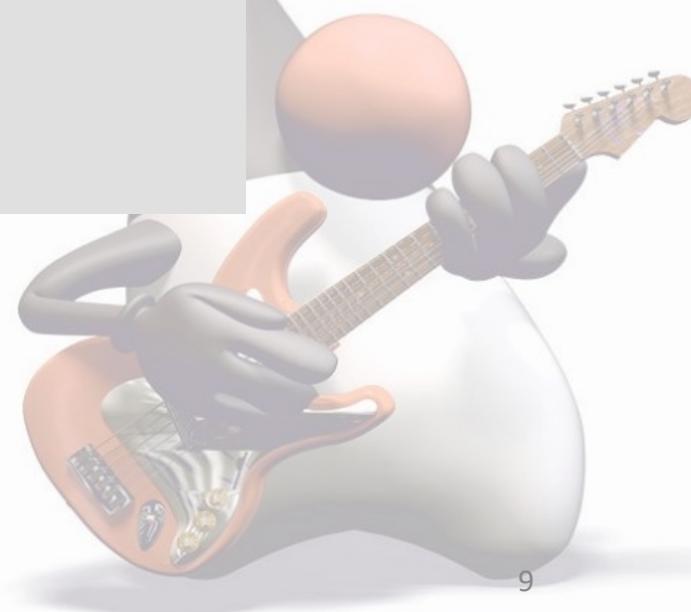
- Modularisierung (Java 9)



# Diamond Operator bei Anonymen Klassen

Java 8:

```
final Comparator<String> byLengthJdk8 = new Comparator<String>()  
{  
    ...  
};
```



# Diamond Operator bei Anonymen Klassen

Java 9:

```
final Comparator<String> byLength = new Comparator<>(){  
    ...  
};
```



# Diamond Operator bei Anonymen Klassen

Zur Erinnerung alternative Deklaration eines Comparator seit Java 8  
auch möglich über:

## 1. Einen Lambda-Ausdruck

```
Comparator<String> byLength = (str1, str2) ->  
    Integer.compare(str1.length(), str2.length());
```

## 2. Die Methode comparing() aus dem Interface Comparator<T>

```
Comparator<String> byLength = Comparator.comparing(String::length);
```

# Übersicht

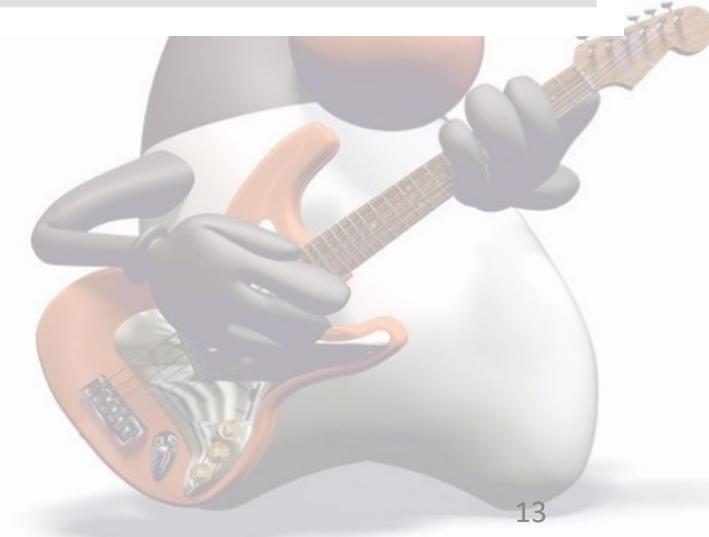
- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Erweiterte @Deprecated Annotation

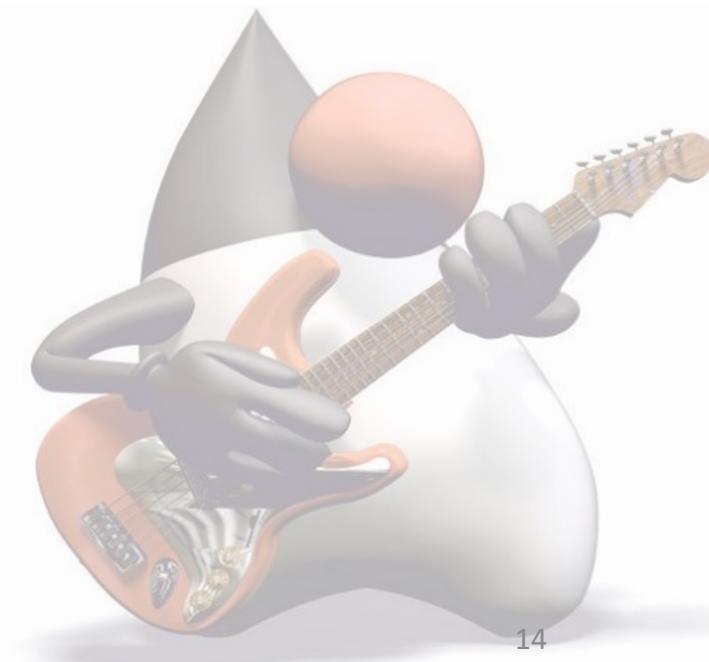
Diese Erweiterung wurde nötig, weil in Zukunft geplant ist, veraltete Funktionalität aus dem JDK zu entfernen, statt sie – wie bislang für Java üblich – aus Rückwärtskompatibilitätsgründen ewig beizubehalten. Das folgende Beispiel zeigt eine Anwendung, wie sie aus dem JDK stammen könnte:

```
@Deprecated(since = "1.5", forRemoval = true)
```



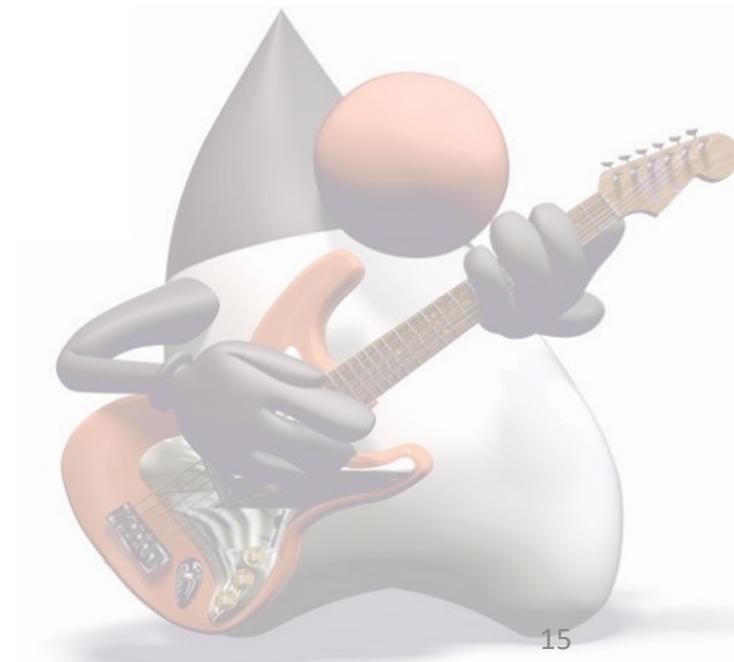
# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - **Private Methoden in Interfaces**
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Private Methoden in Interfaces

```
public interface Ball {  
    default double volume(double radius) {  
        double result = (4./3.) * Math.PI * toThe3(radius);  
        return result;  
    }  
  
    private double toThe3(double radius) {  
        return Math.pow(radius, 3.);  
    }  
}
```



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - **Bezeichner „\_“ ist ein Schlüsselwort**
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Bezeichner „\_“ ist ein Schlüsselwort

bei

```
final String _ = "Underline";
```

produziert der Java-Compiler mit JDK 9 folgende Fehlermeldung:

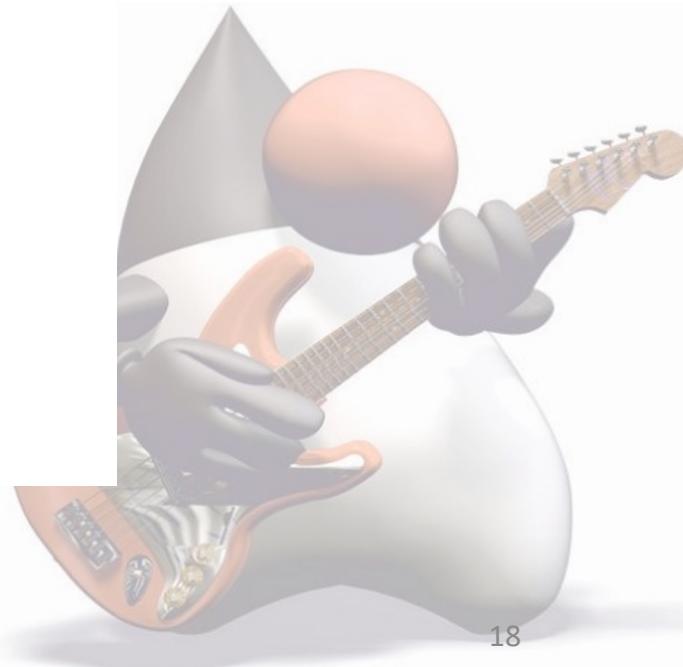
```
as of release 9, '_' is a keyword, and may not be used as an identifier
```



# Bezeichner „\_“ ist ein Schlüsselwort

\$ ist nach wie vor erlaubt;

```
public interface Ball {  
    default double volume(double radius) {  
        double $ = (4./3.) * Math.PI * toThe3(radius);  
        return $;  
    }  
  
    private double toThe3(double radius) {  
        return Math.pow(radius, 3.);  
    }  
}
```



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - **ProcessHandle Interface**
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Process API in JDK 9

## Das Interface ProcessHandle

Neben der PID kann man mithilfe von ProcessHandle noch diverse weitere Informationen zu Prozessen auslesen. Dazu gibt es unter anderem folgende Methoden:

- `current()` – Ermittelt den aktuellen Prozess als ProcessHandle.
- `info()` – Stellt Infos zum Prozess in Form des inneren Interface `ProcessHandle.Info` bereit, etwa zu Benutzer, Kommando usw.
- `info().command()` – Gibt das Kommando als `Optional<String>` aus einem `ProcessHandle.Info` zurück.
- `info().user()` – Liefert den Benutzer als `Optional<String>` aus einem `ProcessHandle.Info`.
- `info().totalCpuDuration()` – Ermittelt aus den Infos die benötigte CPU-Zeit als `Optional<Duration>`. Die Klasse `java.time.Duration` entstammt dem mit JDK 8 neu eingeführten Date and Time API.<sup>1</sup>



# Process API in JDK 8 (ermitteln der PID):

```
private static long getPidJdk8Style() throws InterruptedException,  
    IOException {  
    // $PPID steht für Parent Process ID, also hier derjenigen der JVM  
    final String[] commands = new String[]{"bin/sh", "-c", "echo $PPID"};  
    // Komplexere Windows-Variante  
    final String[] commandsWin = new String[]{"powershell",  
        "gwmi win32_process | ?{$_.ProcessID -eq $pid} | select ParentProcessID | fw -c 2"};  
    // Für Windows: commandsWin nutzen  
    final Process proc = Runtime.getRuntime().exec(commands);  
    if (proc.waitFor() == 0) {  
        try (final InputStream in = proc.getInputStream()) {  
            final int available = in.available();  
            final byte[] outputBytes = new byte[available];  
            in.read(outputBytes);  
            final String pid = new String(outputBytes);  
            // rein theoretisch wäre hier eine NumberFormatException abzufangen  
            return Long.parseLong(pid.trim());  
        }  
    }  
    throw new IllegalStateException("PID is not accessible");  
}
```

# Process API in JDK 9 (ermitteln der PID):

```
private static long getPidJdk9Style() {  
    ProcessHandle processHandle = ProcessHandle.current();  
    return processHandle.pid();  
}
```

## ProcessHandle zu einem Process ermitteln:

```
Process process = Runtime.getRuntime().exec("dir");  
ProcessHandle handle = process.toHandle();
```



# Process API in JDK 9

```
public class TestProcessHandle {  
    public static void main(String...args) {  
        ProcessHandle current = ProcessHandle.current();  
        System.out.println("PID: " + current.pid());  
        System.out.println("Info: " + current.info());  
        System.out.println("Command: " + current.info().command());  
        System.out.println("CPU-Usage: " + current.info().totalCpuDuration());  
    }  
}
```

PID: 13720

Info: [user: Optional[NBMICHAEL\michael], cmd: C:\Program Files\Java\jdk13.0.0.33\bin\java.exe,  
startTime: Optional[2019-10-27T08:50:54.504Z], totalTime: Optional[PT0.28125S]]  
Command: Optional[C:\Program Files\Java\jdk13.0.0.33\bin\java.exe]  
CPU-Usage: Optional[PT0.3125S]

# Process API in JDK 9

```
public class ProcessPrinter {  
    public static void main(String...args) {  
        ProcessHandle.allProcesses().forEach(processHandle ->  
        {  
            final Stream<ProcessHandle> children = processHandle.children();  
            final long count = children.count();  
            if (count > 0) {  
                System.out.println("Info: " + processHandle.info() +  
                    " has " + count + " children");  
            }  
        } );  
    }  
}
```

# Process API in JDK 9

## Prozesse mit **ProcessHandle** kontrollieren

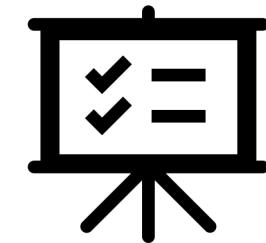
Neben der Bereitstellung und Abfrage von Informationen zu Prozessen existieren auch verschiedene Möglichkeiten, Prozesse zu beenden sowie auf das Ende eines Prozesses zu reagieren. Dazu findet man im Interface **ProcessHandle** folgende Methoden:

- `of(long)` – Liefert ein `Optional<ProcessHandle>` zu einer gegebenen PID.
- `destroy()` – Terminierte einen Prozess, sofern dies erlaubt ist. Ansonsten, etwa für den mit `current()` ermittelten Prozess, wird eine Exception ausgelöst:

```
Exception in thread "main" java.lang.IllegalStateException: destroy of  
current process not allowed
```

- `onExit()` – Liefert ein `CompletableFuture<ProcessHandle>` zurück, das man dazu nutzen kann, verschiedene Aktionen als Reaktion auf das Ende eines Prozesses auszuführen.

# Übungsbeispiel 01



Ermitteln Sie die Prozess-ID und weitere Eigenschaften des aktuellen Prozesses, indem Sie dazu das Interface ProcessHandle und seine Methoden nutzen.

Wie viel CPU-Zeit hat der aktuelle Prozess bislang verbraucht?  
Wie viele Prozesse werden momentan insgesamt ausgeführt?



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - **Collection Factory Methoden**
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Collection Factory Methoden

```
final List<String> names = List.of("Mike", "Tim", "Tom");
```

```
final Set<Integer> primeNumbers = Set.of(2, 3, 5, 7, 11);
```

```
final Map<Integer, String> numberMapping = Map.of(5, "five", 6, "six");
```

Beachte: Überladen definiert bis 10 Entries



# Collection Factory Methoden

```
final Map<Integer, String> mapping = Map.of(5, "five", 6, "six");
```

```
final Map<Integer, String> mapping2 = Map.ofEntries(entry(5, "five"),  
                                         entry(6, "six"));
```

```
mapping.forEach((key, value) -> System.out.println(key + ":" + value));
```

```
mapping2.forEach((key, value) -> System.out.println(key + ":" + value));
```



# Collection Factory Methoden

Beachte:

Alle Collections erzeugt mit Arrays.asList

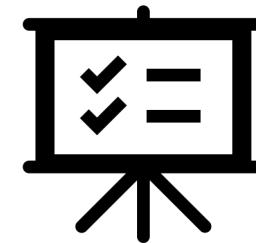
- ... haben eine feste Größe (es können keine zusätzlichen Elemente hinzugefügt werden und auch keine entfernt werden)

Alle Collections erzeugt mit List.of, Set.of, Map.of

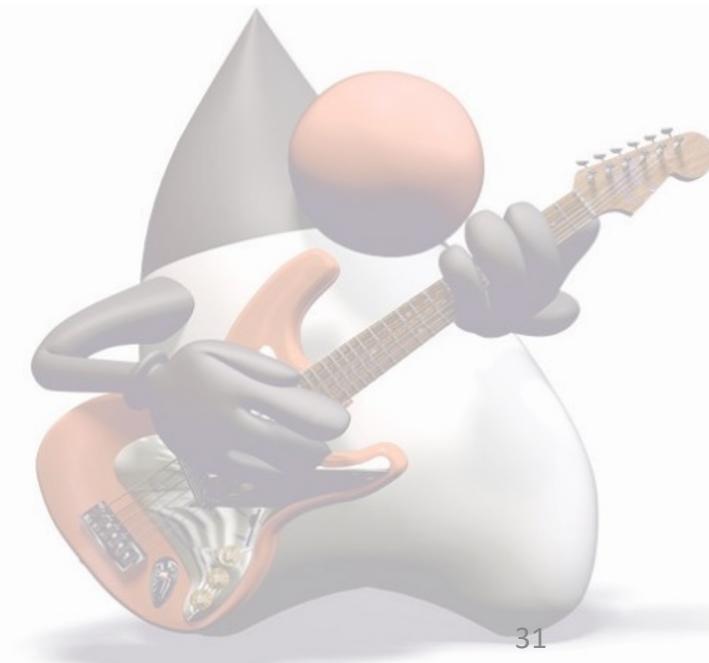
- ... sind immutable Collections
- ... es können daher weder zusätzlichen Elemente hinzugefügt/gelöscht werden, noch können Elemente ausgetauscht werden)
- ... haben bessere Performance als die regulären Collection Implementierungen

Set.of und Map.of führen zu einer Exception, wenn doppelte Einträge hinzugefügt werden

# Übungsbeispiel 02

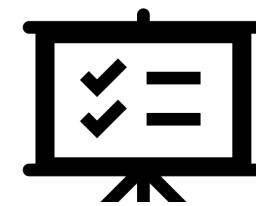


Definieren Sie eine Liste, eine Menge und eine Map mithilfe der in JDK 9 neu eingeführten Collection-FactoryMethoden namens of().



# Übungsbeispiel 02

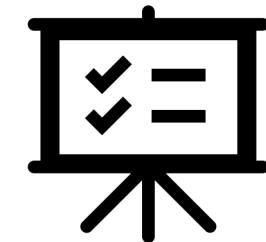
## Ausgangspunkt:



```
public class StartingPoint {  
    public static void main(String... args) {  
        final List<String> names = Arrays.asList("Tim", "Tom", "Mike");  
        System.out.println(names);  
        final Set<Integer> numbers = new TreeSet<>();  
        numbers.add(1);  
        numbers.add(3);  
        numbers.add(4);  
        numbers.add(2);  
        System.out.println(numbers);  
        final Map<Integer, String> mapping = new HashMap<>();  
        mapping.put(5, "five");  
        mapping.put(6, "six");  
        mapping.put(7, "seven");  
        System.out.println(mapping);  
    }  
}
```



# Übungsbeispiel 02



Was beobachtet man nach der Transformation und dem Einsatz der Collection-Factory-Methoden beim Ausführen mit JDK 9?

Nutzen Sie zur Definition der Map alternativ die Methode ofEntries() aus dem Interface Map<K,V>. Wie verändert sich die Lesbarkeit? Was sind die Vorteile dieser Variante?



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - **Reactive Streams**
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Reactive Streams

Themen wie Performance, Skalierbarkeit und Ausfallsicherheit gewinnen heutzutage zunehmend an Bedeutung. Das adressieren Frameworks wie Akka, RxJava, Vert.x und Reactor, indem sie eine asynchrone, eventgetriebene, nicht blockierende Verarbeitung unterstützen – man spricht von Reactive Programming.<sup>6</sup>



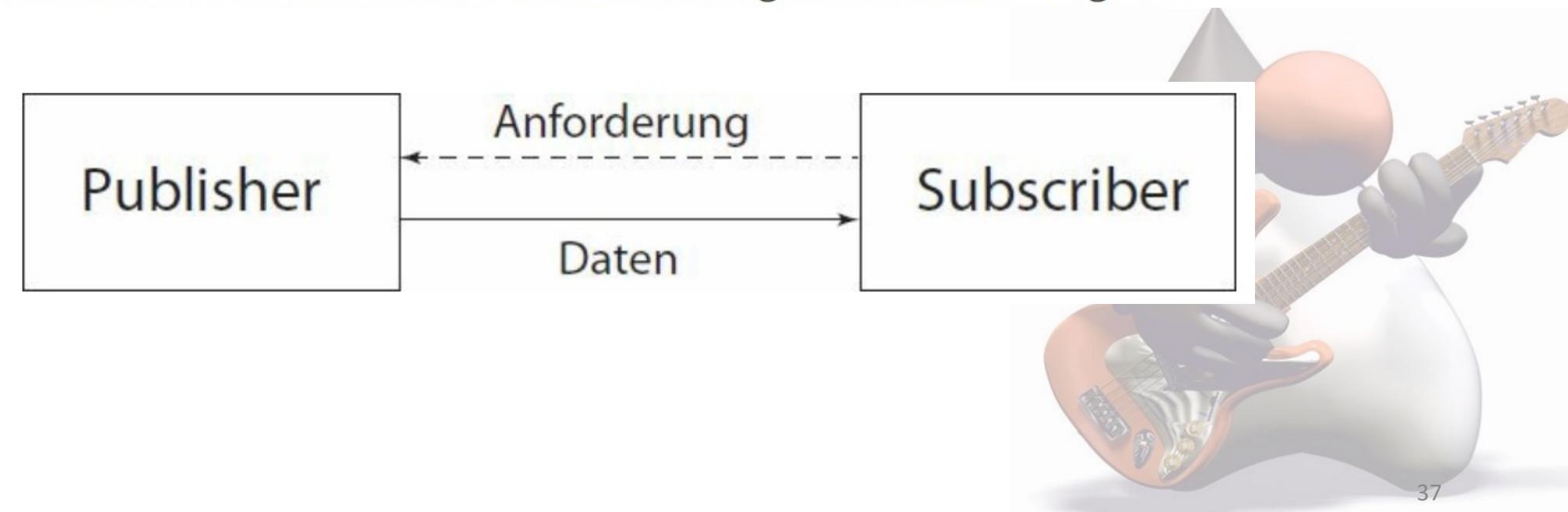
# Reactive Streams

Das Essenzielle ist dabei, dass voneinander unabhängig ausgeführte Verarbeitungseinheiten über Events bzw. Messages miteinander kommunizieren. Dazu registrieren sich Subscriber (Empfänger) bei einem Publisher (Sender). Bei der Kommunikation kann es zu Engpässen kommen, wenn ein Publisher zu langsam oder zu schnell arbeitet. Im ersten Fall werden Subscriber eventuell nur teilweise ausgelastet, im zweiten Fall überlastet. Früher hat man als Abhilfe einen Eingangspuffer aufseiten des Empfängers genutzt. Um Speicherprobleme zu vermeiden, sollte der Eingangspuffer größenbeschränkt sein. Alternativ kann der Sender auch nur eine gewisse Maximalmenge an Daten verschicken. Dadurch verschwendet man aber potenziell Performance, wenn der Empfänger eine schnellere Abarbeitung bewerkstelligen könnte.



# Reactive Streams

Reactive Streams adressieren die beschriebenen Engpässe im Datenfluss. Dazu verfolgen Reactive Streams das Konzept der **Backpressure** (Gegendruck oder Rückstau). Die Idee ist recht simpel: Der Empfänger kann dem Sender mitteilen, wie viele Daten er verarbeiten kann. Der Subscriber fordert aktiv Daten an und der Publisher schickt maximal so viele Daten wie angefordert. Dadurch ermöglichen Reactive Streams eine Selbstregulierung, bei der die Datenverarbeitung an einen möglicherweise temporär langsamen Verarbeiter angepasst wird. Schauen wir uns im Anschluss die mit JDK 9 eingeführte Umsetzung an.



# Reactive Streams

**Das Interface Publisher<T>** Ein Publisher<T> ist ein Veröffentlicher (mitunter auch Erzeuger) von Dingen, die von einem oder mehreren Subscriber<T>-Objekten verarbeitet werden. Dazu können sich diese bei dem Publisher<T> registrieren. Das zugehörige Interface ist minimalistisch wie folgt definiert:

```
@FunctionalInterface  
  
public static interface Publisher<T>  
  
{  
  
    public void subscribe(Subscriber<? super T> subscriber);  
  
}
```

**Das Interface Subscriber<T>** Ein Subscriber<T> ist ein Empfänger von Nachrichten. Die Methoden dieses Interface werden je nach Situation aufgerufen. Dabei wird vom Publisher<T> zunächst eine Subscription an einen Subscriber<T> gesendet und in der Folge werden Daten mit onNext(T) übertragen. Zudem lassen sich Fehlersituationen oder das Ende einer Datenübertragung durch das folgende Interface Subscriber<T> abbilden:

```
public static interface Subscriber<T>
{
    public void onSubscribe(Subscription subscription);

    public void onNext(T item);

    public void onError(Throwable throwable);

    public void onComplete();
}
```



# Reactive Streams

## Das Interface Subscriber<T>

- `onSubscribe(Subscription subscription)` – Dient zur Registrierung und wird vor der eigentlichen Kommunikation aufgerufen.
- `onNext(T item)` – Wird aufgerufen, wenn ein neues Item verfügbar ist.
- `onError(Throwable throwable)` – Für den Fall, dass ein Fehler auftritt, wird diese Methode durch den Publisher<T> aufgerufen, um dies dem jeweiligen Subscriber<T> mitzuteilen.
- `onComplete()` – Falls die Datenübertragung beendet werden soll, kann ein Aufruf dieser Methode durch den Publisher<T> erfolgen.



# Reactive Streams

**Das Interface Subscription** Eine Subscription dient zur Verknüpfung zwischen Publisher<T> und Subscriber<T> und ist wie folgt definiert:

```
public static interface Subscription
{
    public void request(long n);
    public void cancel();
}
```

# Reactive Streams



**Das Interface Processor<T,R>** Der Processor<T,R> kombiniert die beiden Interface Subscriber<T> sowie Publisher<R> und ist folgendermaßen definiert:

```
public static interface Processor<T,R> extends Subscriber<T>, Publisher<R>
{
}
```

# Reactive Streams

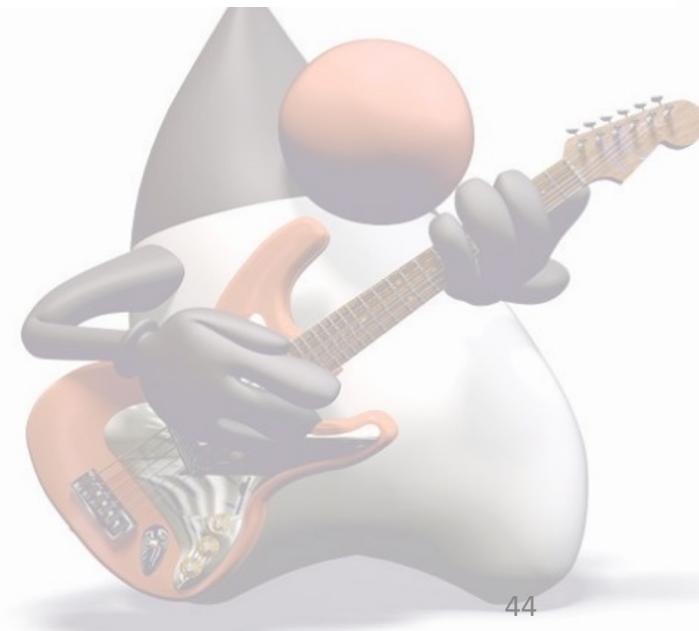


**Das Interface Processor<T,R>** Der Processor<T,R> kombiniert die beiden Interface Subscriber<T> sowie Publisher<R> und ist folgendermaßen definiert:

```
public static interface Processor<T,R> extends Subscriber<T>, Publisher<R>
{
}
```

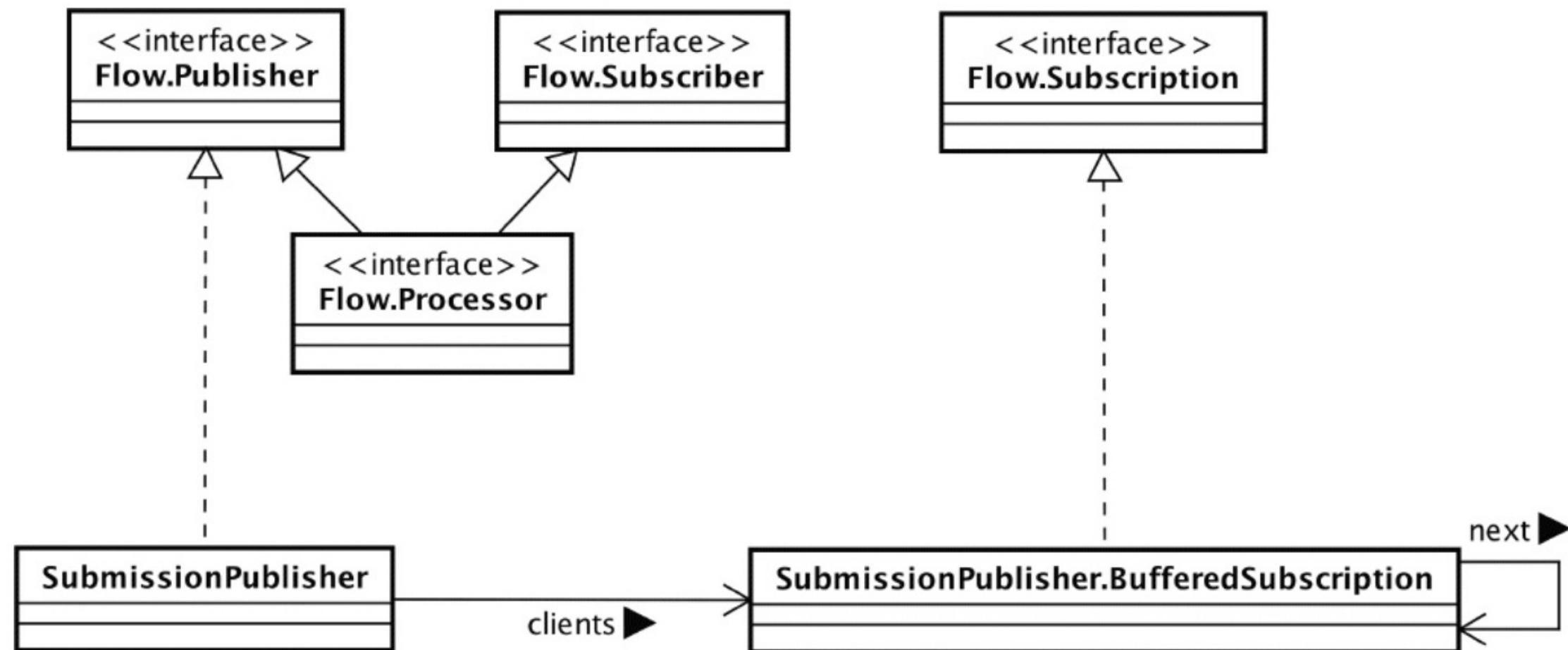
# Reactive Streams

**Die Klassen SubmissionPublisher und BufferedSubscription** Die Klasse `java.util.concurrent.SubmissionPublisher` implementiert das Interface `Publisher<T>` und dient dazu, `Subscriber<T>` zu verwalten, insbesondere, um asynchron Daten an registrierte `Subscriber<T>` publizieren zu können. Die private statische innere Klasse `BufferedSubscription` implementiert das Interface `Subscription` und ermöglicht es, Subscriptions zu verarbeiten. Zudem können registrierte `Subscriber<T>` mithilfe einer `Subscription` neue Daten anfordern oder die Verarbeitung abbrechen.

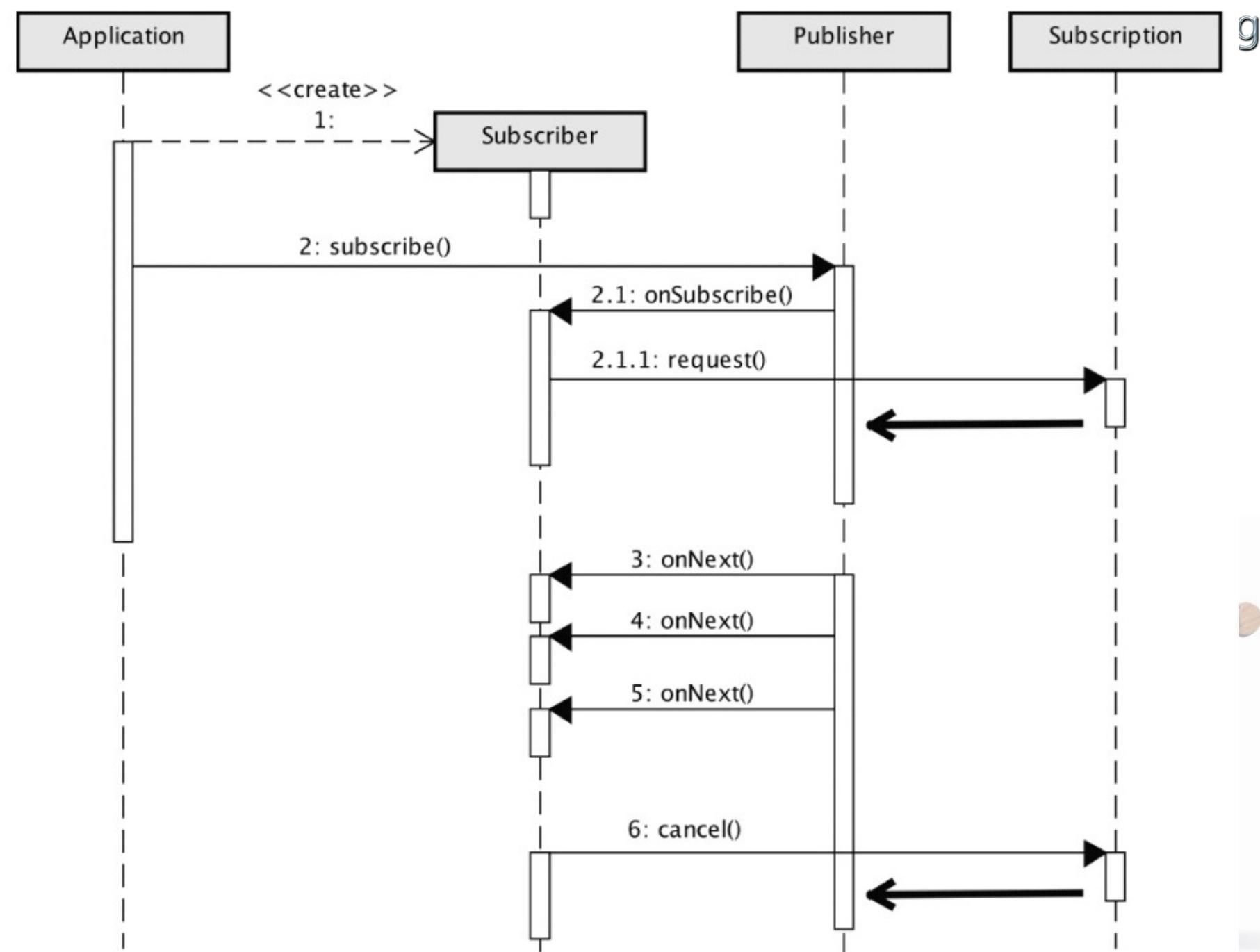


# Reactive Streams

**Die Klassen und Interfaces im Überblick** Abbildung 3-3 zeigt, wie die zuvor beschriebenen Klassen und Interfaces miteinander in Verbindung stehen. Dieses Klassendiagramm sollte das Verständnis für die Zusammenhänge und die im Anschluss vorgestellten Abläufe bei der Kommunikation erleichtern.



# Reactive Streams



# Reactive Streams - Beispiel

```
public class Application {  
    public static void main(final String[] args) throws Exception  
    {  
        final WordPublisher finder = new WordPublisher("private", getInputFiles());  
        finder.subscribe(new WordSubscriber());  
        finder.performSearch();  
        Thread.sleep(2_000); // auf das Ende der Verarbeitung warten  
        finder.terminate();  
    }  
  
    private static List<Path> getInputFiles()  
    {  
        return List.of(  
            Paths.get("src/streams/WordPublisher.java"),  
            Paths.get("src/streams/WordFinderClient.java"));  
    }  
}
```

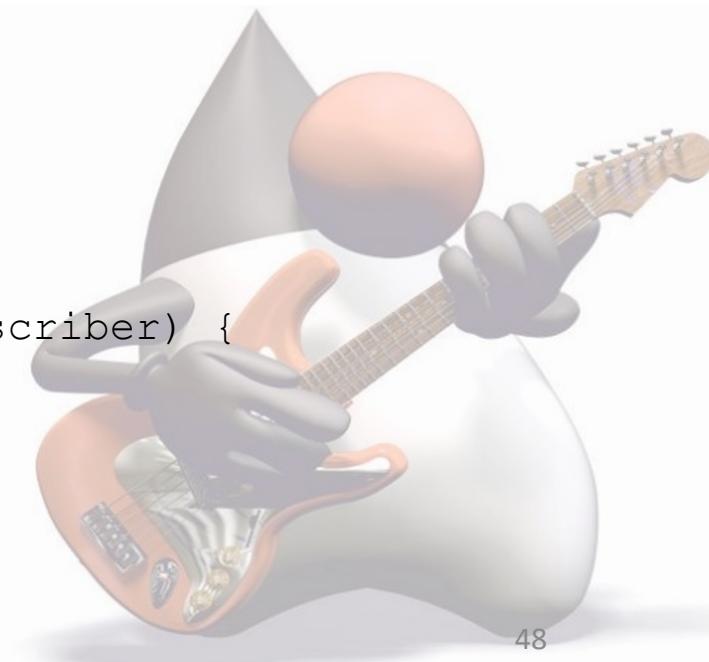


# Reactive Streams - Beispiel

```
public class WordPublisher implements Flow.Publisher<String> {
    private final String word;
    private final List<Path> paths;
    private final SubmissionPublisher<String> publisher;
    private final ExecutorService executor = Executors.newFixedThreadPool(4);

    public WordPublisher(final String word, final List<Path> paths) {
        this.word = word;
        this.paths = Collections.unmodifiableList(paths);
        this.publisher = new SubmissionPublisher<>();
    }

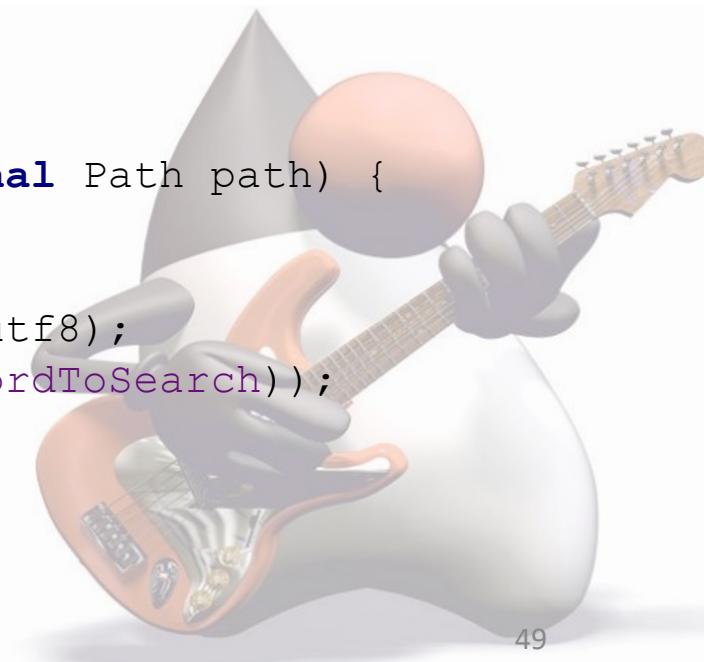
    @Override
    public void subscribe(final Flow.Subscriber<? super String> subscriber) {
        publisher.subscribe(subscriber);
    }
}
```



```
public void performSearch() throws InterruptedException {
    for (final Path path : paths) {
        executor.execute(() -> {
            final Stream<String> occurrences = findWord(word, path);
            occurrences.forEach(line -> publisher.submit("file: " + path +
                " : " + line));
        });
    }
}

public void terminate() throws InterruptedException {
    publisher.close(); // führt zum Aufruf von onComplete()
    executor.shutdown();
}

private Stream<String> findWord(final String wordToSearch, final Path path) {
    try {
        final Charset utf8 = StandardCharsets.UTF_8;
        final List<String> lines = Files.readAllLines(path, utf8);
        return lines.stream().filter(line -> line.contains(wordToSearch));
    } catch (final IOException e) {
        return Stream.of();
    }
}
```



# Reactive Streams - Beispiel

```
public class WordSubscriber implements Flow.Subscriber<String> {
    @Override
    public void onSubscribe(final Flow.Subscription subscription) {
        System.out.println(LocalDateTime.now() + " onSubscribe()");
        subscription.request(Long.MAX_VALUE);
    }

    @Override
    public void onNext(final String item) {
        System.out.println(LocalDateTime.now() + " onNext(): " + item);
    }

    @Override
    public void onComplete() {
        System.out.println(LocalDateTime.now() + " onComplete()");
    }

    @Override
    public void onError(final Throwable throwable) {
        throwable.printStackTrace();
    }
}
```

# Reactive Streams - Beispiel

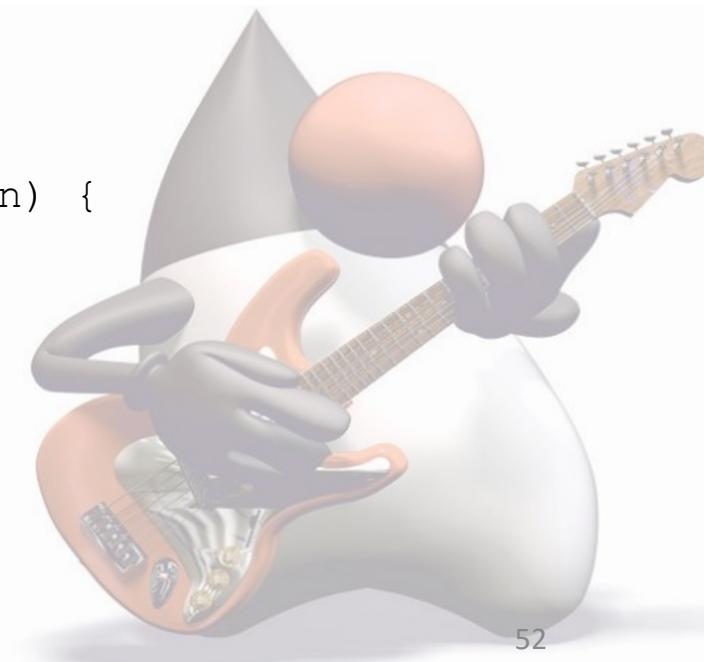
```
2019-10-27T11:10:05.145418800 onSubscribe()
2019-10-27T11:10:05.148407 onNext(): file: src\streams\WordPublisher.java :
2019-10-27T11:10:05.148407 onNext(): file: src\streams\WordPublisher.java :
2019-10-27T11:10:05.148407 onNext(): file: src\streams\WordPublisher.java :
publisher;
2019-10-27T11:10:05.148407 onNext(): file: src\streams\WordPublisher.java :
Executors.newFixedThreadPool(4);
2019-10-27T11:10:05.148407 onNext(): file: src\streams\WordPublisher.java :
wordToSearch,
2019-10-27T11:10:07.131033 onComplete()
```

```
private final String word;
private final List<Path> paths;
private final SubmissionPublisher<String>
private final ExecutorService executor =
private Stream<String> findWord(final String
```



# Reactive Streams – zusätzlicher Subscriber

```
public class First5Subscriber implements Flow.Subscriber<String> {  
    private final Consumer<String> consumer;  
    private Flow.Subscription subscription;  
  
    private int count = 1;  
  
    First5Subscriber(final Consumer<String> consumer) {  
        this.consumer = consumer;  
    }  
  
    @Override  
    public void onSubscribe(Flow.Subscription subscription) {  
        System.out.println("F5 Subscription: " + subscription);  
        this.subscription = subscription;  
        this.subscription.request(5);  
    }  
}
```



# Reactive Streams – zusätzlicher Subscriber

```
@Override  
public void onNext(final String item) {  
    System.err.println("F5 " + count + "x onNext(): " + item);  
    consumer.accept(item);  
    count++;  
    if (count >= 5) {  
        subscription.cancel();  
    }  
}  
  
@Override  
public void onComplete() {  
    System.out.println("onComplete()");  
}  
  
@Override  
public void onError(final Throwable throwable) {  
    throwable.printStackTrace();  
}  
}
```



# Reactive Streams – zusätzlicher Subscriber

```
public class ApplicationMultipleSubscribers {
    public static void main(final String[] args) throws IOException,
        InterruptedException
    {
        final WordPublisher finder = new WordPublisher("private", getInputFiles());
        finder.subscribe(new WordSubscriber());
        finder.subscribe(new First5Subscriber(System.err::println));
        finder.performSearch();
        TimeUnit.SECONDS.sleep(2); // auf das Ende der Verarbeitung warten
        finder.terminate();
    }

    private static List<Path> getInputFiles()
    {
        return List.of(
            Paths.get("src/streams/WordPublisher.java"),
            Paths.get("src/streams/WordFinderClient.java"));
    }
}
```



# Reactive Streams – zusätzlicher Subscriber

```
F5 Subscription: java.util.concurrent.SubmissionPublisher$BufferedSubscription@2c47074a
2019-10-27T11:29:46.751542400 onSubscribe()
2019-10-27T11:29:46.756528800 onNext(): file: src\streams\WordPublisher.java :    private final String word;
2019-10-27T11:29:46.756528800 onNext(): file: src\streams\WordPublisher.java :    private final List<Path> paths;
2019-10-27T11:29:46.756528800 onNext(): file: src\streams\WordPublisher.java :    private final SubmissionPublisher<String> publisher;
2019-10-27T11:29:46.756528800 onNext(): file: src\streams\WordPublisher.java :    private final ExecutorService executor =
Executors.newFixedThreadPool(4);
2019-10-27T11:29:46.757526300 onNext(): file: src\streams\WordPublisher.java :    private Stream<String> findWord(final String wordToSearch,
F5 1x onNext(): file: src\streams\WordPublisher.java :    private final String word;
file: src\streams\WordPublisher.java :    private final String word;
F5 2x onNext(): file: src\streams\WordPublisher.java :    private final List<Path> paths;
file: src\streams\WordPublisher.java :    private final List<Path> paths;
F5 3x onNext(): file: src\streams\WordPublisher.java :    private final SubmissionPublisher<String> publisher;
file: src\streams\WordPublisher.java :    private final SubmissionPublisher<String> publisher;
F5 4x onNext(): file: src\streams\WordPublisher.java :    private final ExecutorService executor = Executors.newFixedThreadPool(4);
file: src\streams\WordPublisher.java :    private final ExecutorService executor = Executors.newFixedThreadPool(4);
2019-10-27T11:29:48.728988300 onComplete()
```

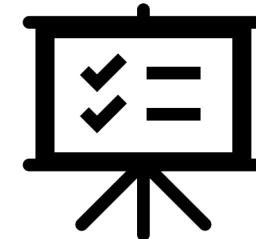
Process finished with exit code 0

# Reactive Streams – Zusammenfassung

Dieses Unterkapitel hat einen Einstieg in Reactive Streams gegeben und die grundlegenden Konzepte kurz vorgestellt. Sie sollten mit diesem Wissen in der Lage sein, erste eigene Experimente zu starten. Richtig spannend wird das Ganze, wenn man größere Verarbeitungsketten mithilfe mehrerer Publisher<T>, Processor<T,R> und Subscriber<T> beschreibt. Das würde jedoch den Rahmen dieses Buchs sprengen. Leider bietet die Implementierung der Reactive Streams im JDK momentan noch kein API zur Verkettung von Operationen. Es bleibt für die Zukunft zu hoffen und zu erwarten, dass hier noch Erweiterungen folgen.

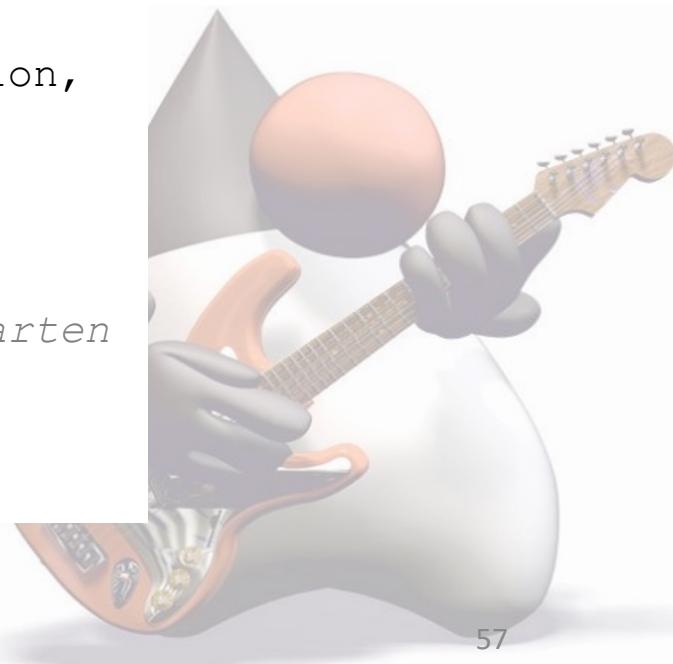


# Übungsbeispiel 03



Gegeben sei ein Programm mit einem Publisher<String>, der Worte aus einer Datei liest und diese an registrierte Subscriber<String> veröffentlicht:

```
public class StartingPoint {
    public static void main(final String[] args) throws IOException,
        InterruptedException {
        final WordPublisher publisher = new WordPublisher();
        publisher.subscribe(new WordSubscriber());
        publisher.doWork();
        Thread.sleep(10_000); // auf das Ende der Verarbeitung warten
    }
}
```



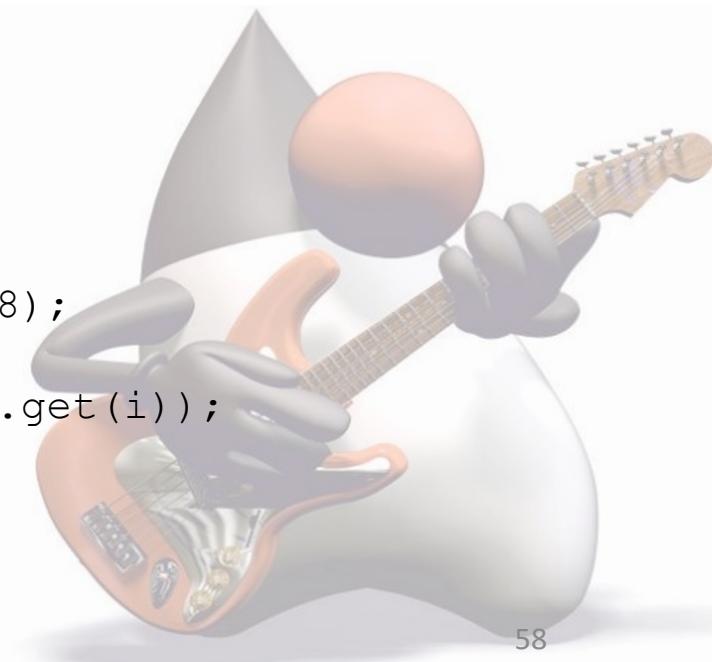
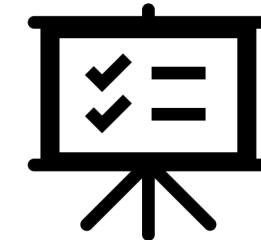
# Übungsbeispiel 03

```
public class WordPublisher implements Flow.Publisher<String> {
    private final SubmissionPublisher<String> publisher;

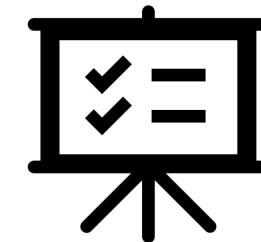
    public WordPublisher() {
        this.publisher = new SubmissionPublisher<>();
    }

    public void subscribe(final Flow.Subscriber<? super String> subscriber) {
        publisher.subscribe(subscriber);
    }

    public void doWork() throws IOException {
        final Charset utf8 = StandardCharsets.UTF_8;
        final Path path = Paths.get("./resources/input.txt");
        final List<String> lines = Files.readAllLines(path, utf8);
        for (int i = 0; i < lines.size(); i++) {
            publisher.submit("line: " + (i + 1) + " : " + lines.get(i));
        }
    }
}
```



# Übungsbeispiel 03

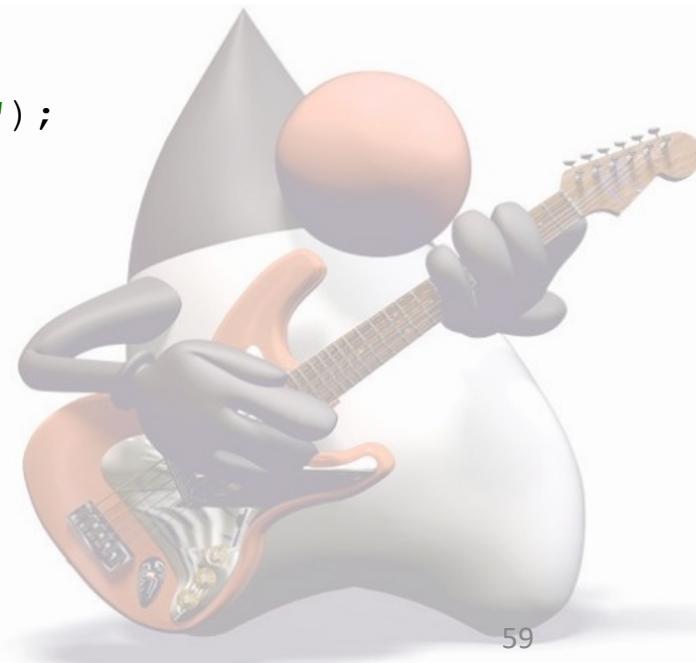


```
class WordSubscriber implements Flow.Subscriber<String> {
    public void onSubscribe(final Flow.Subscription subscription) {
        subscription.request(Long.MAX_VALUE);
    }

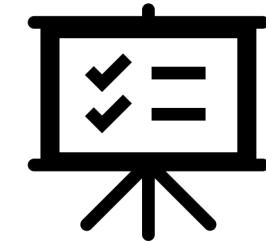
    public void onNext(final String item) {
        System.out.println(LocalDateTime.now() + " onNext(): " + item);
    }

    public void onComplete() {
        System.out.println(LocalDateTime.now() + " onComplete()");
    }

    public void onError(final Throwable throwable) {
        throwable.printStackTrace();
    }
}
```



# Übungsbeispiel 03



Implementieren Sie basierend auf der obigen Klasse WordSubscriber einen eigenen Subscriber<String> namens SkipAndTakeSubscriber, der die ersten n Vorkommen überspringt und danach m Vorkommen ausgibt. Danach soll die Kommunikation gestoppt werden, also der WordPublisher diesem Subscriber<String> keine Daten mehr senden.



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - **Erweiterungen InputStream**
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Erweiterungen InputStream

```
public static void main(final String[] args) throws IOException
```

```
{
```

```
    final byte[] buffer = { 72, 65, 76, 76, 79 };
```

```
    // Liest alle Bytes in einem Rutsch
```

```
    final byte[] result = new ByteArrayInputStream(buffer).readAllBytes();
```

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

```
    // Überträgt Daten direkt aus einem InputStream in einen OutputStream
```

```
    new ByteArrayInputStream(buffer).transferTo(System.out);
```

```
}
```



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Erweiterungen Optional<T>

## Optional<T> Beispielcode (ab Java 8)

```
public class OptionalTest {  
    public static void main(final String[] args) {  
        findCustomer("Tim").ifPresent(System.out::println);  
        findCustomer("UNKNOWN").ifPresent(System.out::println);  
    }  
  
    private static Optional<String> findCustomer(final String customerId) {  
        System.out.println("findCustomer(" + customerId + ")");  
        final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");  
        if (customers.anyMatch(name -> name.contains(customerId))) {  
            return Optional.of(customerId);  
        }  
        return Optional.empty();  
    }  
}
```

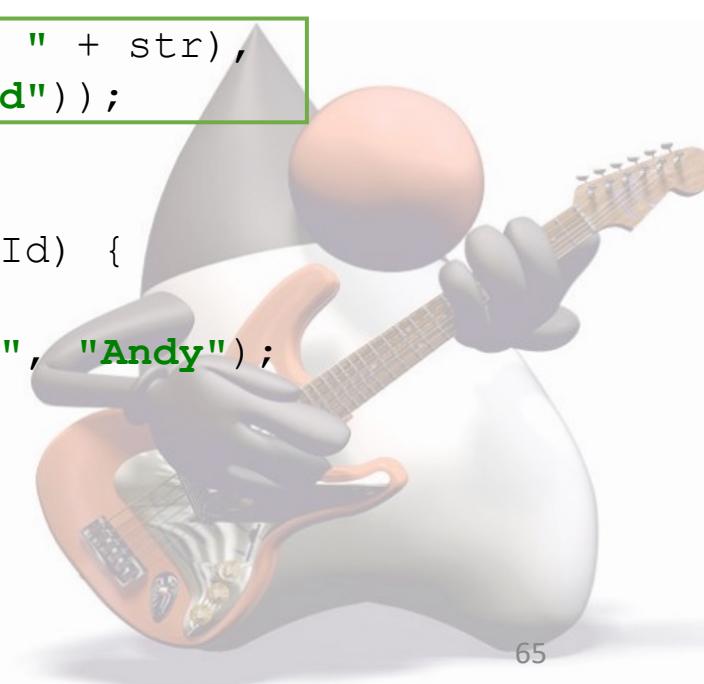
```
findCustomer(Tim)  
Tim  
findCustomer(UNKNOWN)
```



# Erweiterungen Optional<T>

## Optional<T> Beispielcode (ab Java 9)

```
public class OptionalTestJdk9 {  
    public static void main(final String[] args) {  
        final Optional<String> optCustomer1 = findCustomer("Tim");  
        optCustomer1.ifPresentOrElse(str -> System.out.println("found: " + str),  
                                     () -> System.out.println("not found"));  
  
        final Optional<String> optCustomer2 = findCustomer("UNKNOWN");  
  
        optCustomer2.ifPresentOrElse(str -> System.out.println("found: " + str),  
                                     () -> System.out.println("not found"));  
    }  
  
    private static Optional<String> findCustomer(final String customerId) {  
        System.out.println("findCustomer(" + customerId + ")");  
        final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");  
        if (customers.anyMatch(name -> name.contains(customerId))) {  
            return Optional.of(customerId);  
        }  
        return Optional.empty();  
    }  
}
```

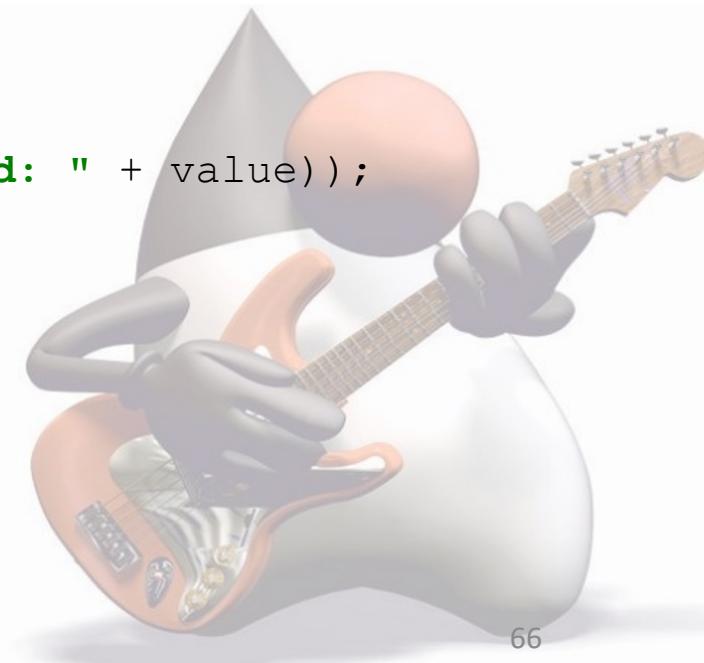


# Erweiterungen Optional<T>

## Optional zu einem Stream umwandeln

```
public class OptionalStream {  
    public static void main(final String[] args) {  
        final Stream<Optional<String>> streamOfOptionalNames = Stream.of(  
            Optional.of("Tim"), Optional.of("Tom"),  
            Optional.empty(), Optional.of("Mike"),  
            Optional.empty(), Optional.of("Andy"));  
  
        final Stream<String> streamOfNames =  
            streamOfOptionalNames.flatMap(Optional::stream);  
  
        streamOfNames.forEach(value -> System.out.println("found: " + value));  
    }  
}
```

```
found: Tim  
found: Tom  
found: Mike  
found: Andy
```

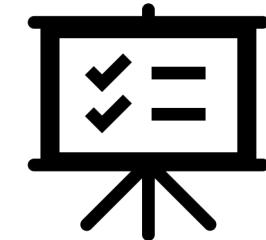


# Erweiterungen Optional<T>

Optional<T>  
or()

```
public class OptionalOr {  
    public static void main(final String[] args) {  
        final Optional<String> optCustomer = multiFindCustomer("Tim");  
        optCustomer.ifPresentOrElse(str -> System.out.println("found: " + str),  
                                     () -> System.out.println("not found"));  
    }  
  
    private static Optional<String> multiFindCustomer(final String customerId) {  
        return findInCache(customerId)  
            .or(() -> findInMemory(customerId))  
            .or(() -> findInDb(customerId));  
    }  
  
    private static Optional<String> findInCache(String customerId) {  
        return null;  
    }  
  
    private static Optional<String> findInMemory(String customerId) {  
        return null;  
    }  
  
    private static Optional<String> findInDb(String customerId) {  
        return null;  
    }  
}
```

# Übungsbeispiel 04



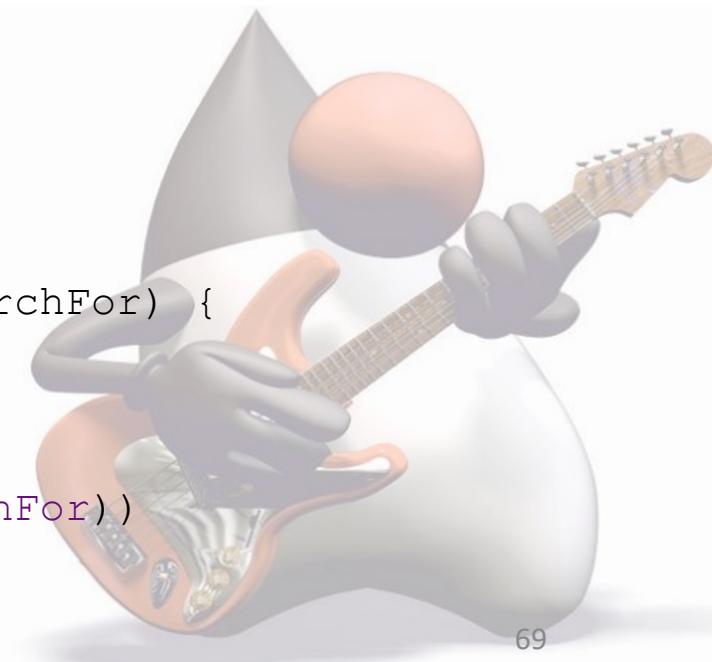
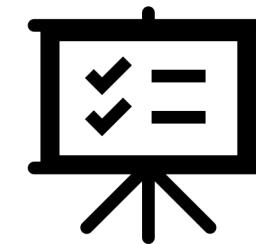
Gegeben sei untenstehendes Programmfragment, das eine Personensuche ausführt und abhängig vom Ergebnis bei einem Treffer die Methode doHappyCase(Person) oder ansonsten doErrorCase() auruft.

Gestalten Sie das Programmfragment mithilfe der neuen Methoden aus der Klasse Optional<T> eleganter.

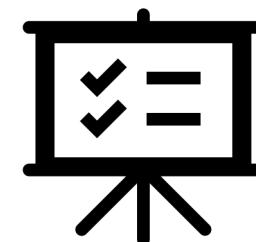


# Übungsbeispiel 04

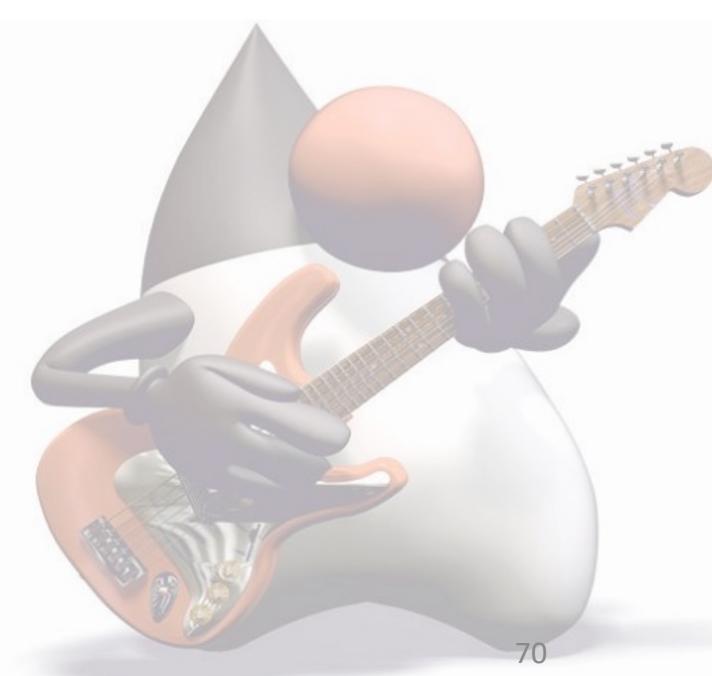
```
public class StartingPoint {  
    private static void findJdk8() {  
        final Optional<Person> opt = findPersonByName("Tim");  
        if (opt.isPresent()) {  
            doHappyCase(opt.get());  
        } else {  
            doErrorCase();  
        }  
        final Optional<Person> opt2 = findPersonByName("UNKNOWN");  
        if (opt2.isPresent()) {  
            doHappyCase(opt2.get());  
        } else {  
            doErrorCase();  
        }  
    }  
  
    private static Optional<Person> findPersonByName(final String searchFor) {  
        final Stream<Person> persons = Stream.of(new Person("Mike"),  
            new Person("Tim"),  
            new Person("Tom"));  
        return persons.filter(person -> person.getName().equals(searchFor))  
            .findFirst();  
    }  
}
```



# Übungsbeispiel 04



```
private static void doHappyCase(final Person person) {  
    System.out.println("Result: " + person);  
}  
  
private static void doErrorCase() {  
    System.out.println("not found");  
}  
  
static class Person {  
    private final String name;  
  
    public Person(final String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```



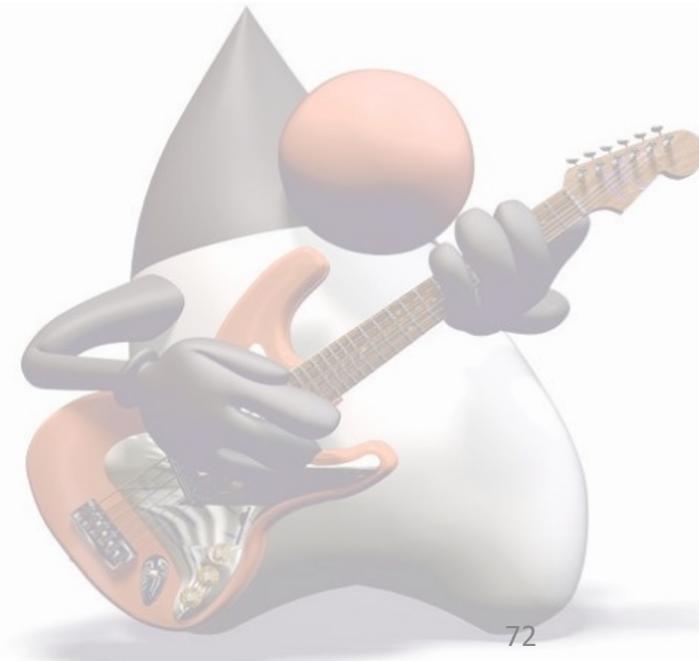
# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



# Erweiterungen java.util.Stream

- `takeWhile(Predicate<T>)` – Verarbeitet Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.
- `dropWhile(Predicate<T>)` – Überspringt Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.



# Erweiterungen java.util.Stream

## takeWhile() & dropwhile()

```
public static void main(final String[] args) {
    // Unendliche Wertefolge erzeugen und alle bis 10 abgreifen
    final IntStream stream1 = IntStream.iterate(1, n -> n + 1);
    System.out.println("takeWhile: " + stream1.takeWhile(n -> n < 10).
        mapToObj(Integer::toString).
        collect(Collectors.joining(", ")));
    // Wertebereich von 7 bis 14 erzeugen und alle kleiner 10 überspringen
    final IntStream stream2 = IntStream.rangeClosed(7, 14);
    System.out.println("dropWhile 1: " + stream2.dropWhile(n -> n < 10).
        mapToObj(Integer::toString).
        collect(Collectors.joining(", ")));
    // Demonstration von dropWhile() bei gemischter Wertefolge
    final IntStream stream3 = IntStream.of(7, 9, 11, 13, 15, 5, 3, 1);
    System.out.println("dropWhile 2: " + stream3.dropWhile(n -> n < 10).
        mapToObj(Integer::toString).
        collect(Collectors.joining(", ")));
}
```



```
takeWhile: 1, 2, 3, 4, 5, 6, 7, 8, 9
dropWhile 1: 10, 11, 12, 13, 14
dropWhile 2: 11, 13, 15, 5, 3, 1
```

# Erweiterungen java.util.Stream

## takeWhile() & dropwhile()

```
public static void main(final String[] args) {  
    Stream<String> words = Stream.of("ab", "bla", "<START>",  
        "Hier", "steht", "der", "Text", "zwischen",  
        "den", "Start-", "und", "Ende-Begrenzern",  
        "<END>", "saas", "bla");  
    Stream<String> content = words.dropWhile(word -> !word.equals("<START>"))  
        .skip(1)  
        .takeWhile(word -> !word.equals("<END>"));  
    content.forEach(System.out::println);  
}
```

Hier  
steht  
der  
Text  
zwischen  
den  
Start-  
und  
Ende-Begrenzern

# Erweiterungen java.util.Stream

## Weitere neue Methoden

Neben den Neuerungen in Form der Methoden `takeWhile()` und `dropWhile()` findet man für Streams folgende neue Methoden:

- `ofNullable(T)` – Liefert einen `Stream<T>` mit einem Element, sofern das übergebene Element ungleich null ist. Ansonsten wird ein leerer Stream erzeugt.
- `iterate(T, Predicate<? super T>, UnaryOperator<T>)` – Es wird ein `Stream<T>` mit dem als ersten Parameter übergebenen Startwert erzeugt. Die folgenden Werte werden durch den `java.util.function.UnaryOperator` berechnet. Im Gegensatz zu der bereits mit JDK 8 existierenden Methode `iterate(T, UnaryOperator<T>)` wird hierbei auch noch das übergebene `java.util.function.Predicate<T>` geprüft und die Erzeugung gestoppt, sobald dieses nicht mehr erfüllt ist.

# Erweiterung java.util.Stream

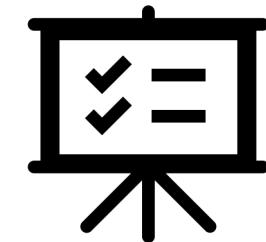
## iterate()

```
public static void main(final String[] args) {  
    System.out.println("iterate with predicate");  
  
    final IntStream stream = IntStream.iterate(1, n -> n < 10, n -> n + 1);  
    System.out.println(stream.mapToObj(num -> "" + num).collect(joining(", ")));  
}
```

```
iterate with predicate  
1, 2, 3, 4, 5, 6, 7, 8, 9
```

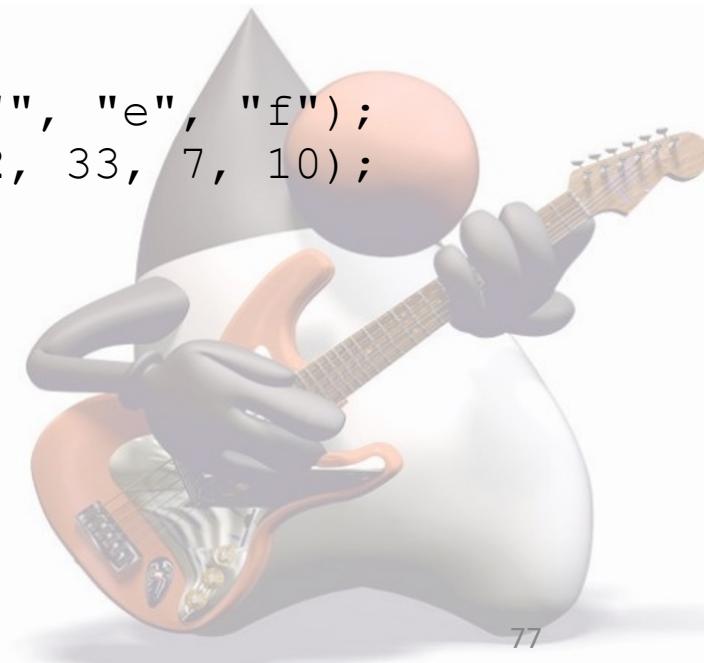


# Übungsbeispiel 05

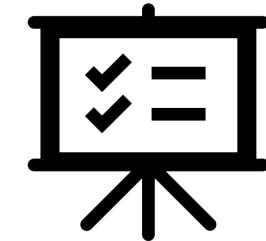


Das Stream-API wurde um Methoden erweitert, die es erlauben, nur so lange Elemente zu lesen, wie eine Bedingung erfüllt ist, bzw. Elemente zu überspringen, solange eine Bedingung erfüllt ist. Als Datenbasis dienen folgende zwei Streams:

```
final Stream<String> values1 = Stream.of("a", "b", "c", "", "e", "f");
final Stream<Integer> values2 = Stream.of(1, 2, 3, 11, 22, 33, 7, 10);
```



# Übungsbeispiel 05



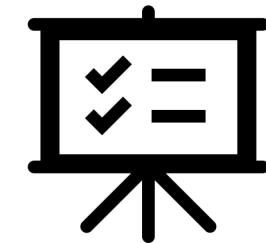
**Aufgabe 5a:** Ermitteln Sie aus dem Stream values1 so lange Werte, bis ein Leerstring gefunden wird. Geben Sie die Werte auf der Konsole aus.

**Aufgabe 5b:** Überspringen Sie in Stream values2 die Werte, so lange der Wert kleiner als 10 ist. Geben Sie die Werte auf der Konsole aus.

**Aufgabe 5c:** Worin besteht der Unterschied zwischen den beiden Methoden dropWhile() und filter()



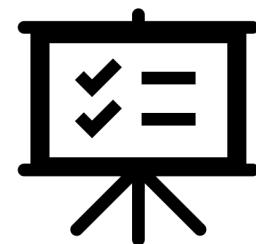
# Übungsbeispiel 06



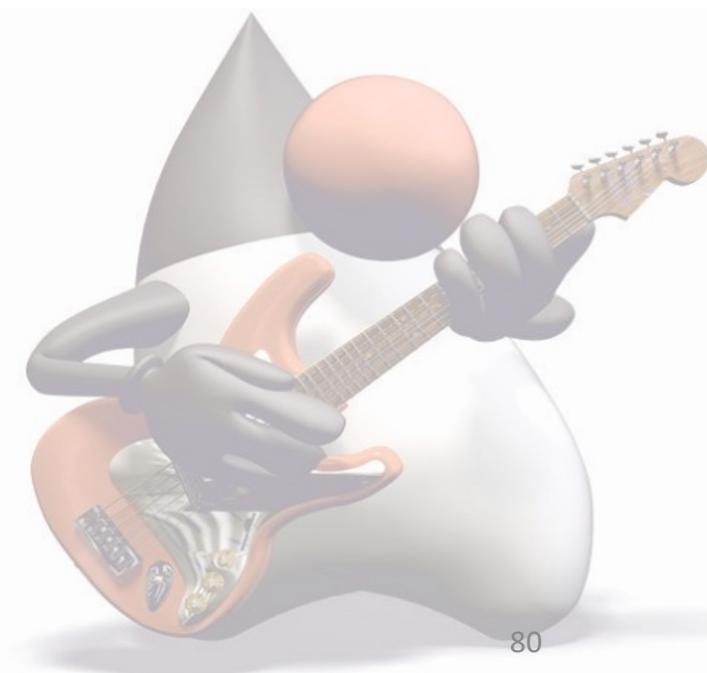
Gegeben sei folgendes Programmfragment, das eine mehrstufige Suche zunächst im Cache, dann im Speicher und schließlich in der Datenbank ausführt. Diese Suchkette ist durch drei find()-Methoden angedeutet und wie folgt implementiert:



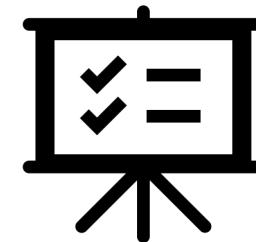
# Übungsbeispiel 06



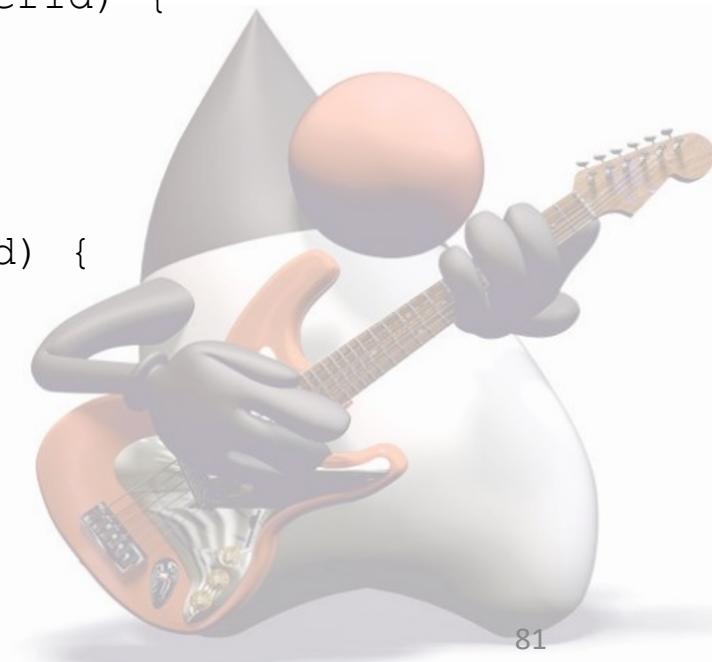
```
public class StartingPoint {  
    public static void main(final String[] args) {  
        final Optional<String> optCustomer = multiFindCustomerJdk8("Tim");  
        optCustomer.ifPresentOrElse(str -> System.out.println("found: " + str),  
            () -> System.out.println("not found"));  
    }  
  
    private static Optional<String> multiFindCustomerJdk8(final String customerId) {  
        final Optional<String> opt1 = findInCache(customerId);  
        if (opt1.isPresent()) {  
            return opt1;  
        } else {  
            final Optional<String> opt2 = findInMemory(customerId);  
            if (opt2.isPresent()) {  
                return opt2;  
            } else {  
                return findInDb(customerId);  
            }  
        }  
    }  
}
```



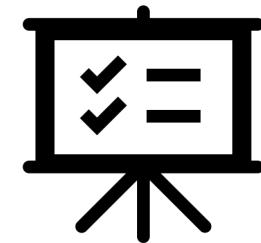
# Übungsbeispiel 06



```
private static Optional<String> findInMemory(final String customerId) {  
    System.out.println("findInMemory");  
    final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");  
    return customers.filter(name -> name.contains(customerId))  
        .findFirst();  
}  
  
private static Optional<String> findInCache(final String customerId) {  
    System.out.println("findInCache");  
    return Optional.empty();  
}  
  
private static Optional<String> findInDb(final String customerId) {  
    System.out.println("findInDb");  
    return Optional.empty();  
}
```



# Übungsbeispiel 06



Vereinfachen Sie die Aufrufkette mithilfe der neuen Methoden aus der Klasse Optional<T>. Sehen Sie, wie das Ganze an Klarheit gewinnt.



# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



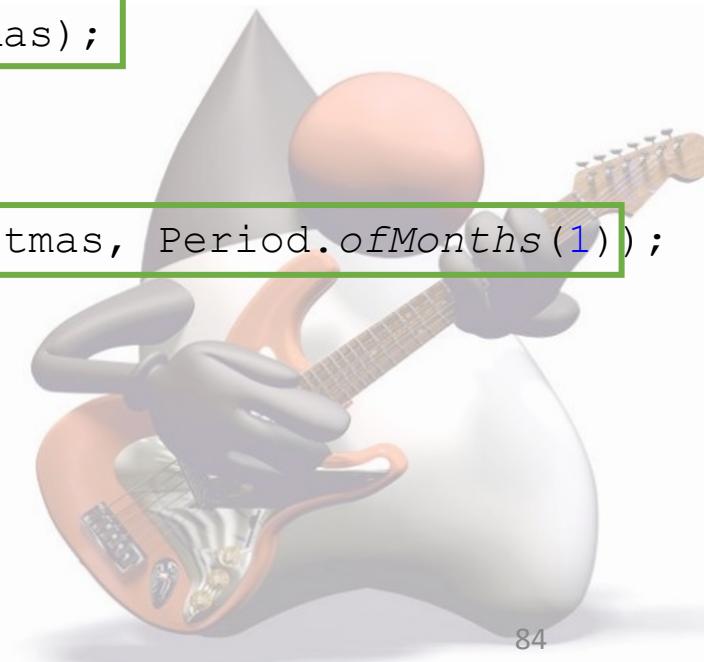
# Erweiterungen LocalDate

## datesUntil()

```
public static void main(final String[] args) {
    final LocalDate myBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate christmas = LocalDate.of(1971, Month.DECEMBER, 24);

    System.out.println("Day-Stream");
    final Stream<LocalDate> daysUntil = myBirthday.datesUntil(christmas);
    daysUntil.skip(150).limit(8).forEach(System.out::println);

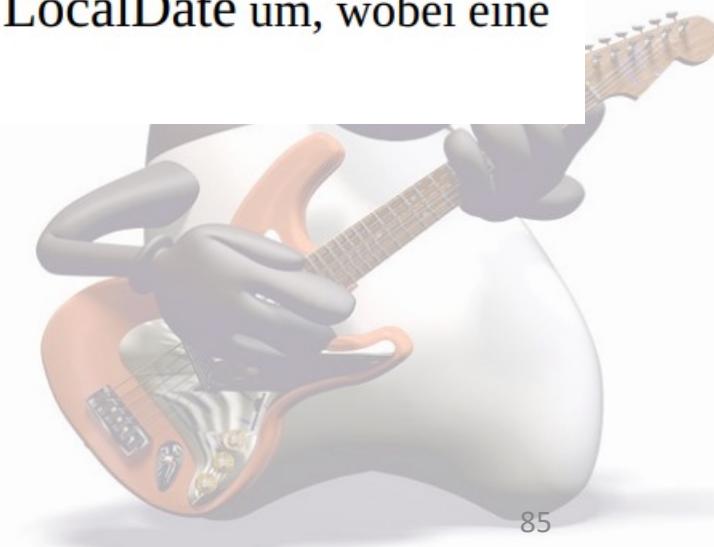
    System.out.println("\nMonth-Stream");
    final Stream<LocalDate> monthsUntil = myBirthday.datesUntil(christmas, Period.ofMonths(1));
    monthsUntil.limit(5).forEach(System.out::println);
}
```



# Erweiterungen LocalDate toEpochSecond() & ofInstant()

Neben dieser Neuerung findet man in der Klasse LocalDate folgende zwei erwähnenswerte Methoden:

- `toEpochSecond(LocalTime, ZoneOffset)` – Konvertiert eine LocalDate-Instanz in einen long, der den seit dem Referenzwert 1970-01-01T00:00:00Z vergangenen Sekunden entspricht. Weil ein LocalDate keine Zeitinformation besitzt, muss hier ein LocalTime- sowie ein java.time.ZoneOffset-Objekt übergeben werden.
- `ofInstant(Instant, ZoneId)` – Wandelt ein java.time.Instant-Objekt in ein LocalDate um, wobei eine Zeitzone in Form einer java.time.ZoneId benötigt wird.



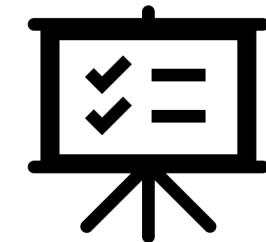
# Erweiterungen LocalDate toEpochSecond() & ofInstant()

```
public static void main(final String[] args) {  
    final LocalDate someday = LocalDate.of(1971, 2, 7);  
    final LocalTime time = LocalTime.of(2, 15);  
    final Instant instant = Instant.ofEpochMilli(0);  
  
    System.out.println("toEpochSecond: " + someday.toEpochSecond(time, ZoneOffset.ofHours(-5)));  
  
    System.out.println("ofInstant: " + LocalDate.ofInstant(instant, ZoneId.of("Europe/Zurich")));  
}
```

```
toEpochSecond: 34758900  
ofInstant: 1970-01-01
```

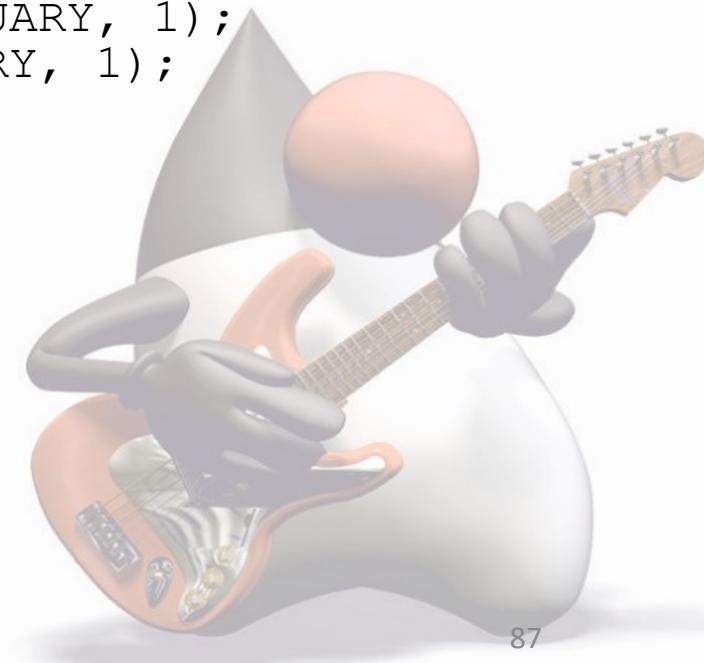


# Übungsbeispiel 07



Berechnen Sie mithilfe der neuen Methoden aus der Klasse LocalDate alle Vorkommen von Freitag, dem 13., in den Jahren 2013 bis einschließlich 2017. Nutzen Sie folgende Zeilen als Ausgangspunkt:

```
final LocalDate startDate = LocalDate.of(2013, Month.JANUARY, 1);
final LocalDate endDate = LocalDate.of(2018, Month.JANUARY, 1);
final Predicate<LocalDate> isFriday = null; // TODO
final Predicate<LocalDate> is13th = null; // TODO
```



# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - **Erweiterungen Arrays**
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# Erweiterungen Arrays

Auch die Utility-Klasse `java.util.Arrays` wurde in JDK 9 erweitert. Das betrifft vor allem folgende Methoden und Funktionalitäten:

- `equals()` – Vergleicht zwei Arrays elementweise per `equals()` und bietet seit JDK 9 eine Bereichseinschränkung.
- `mismatch()` – Prüft, ob zwei Arrays sich unterscheiden, und liefert die Position der ersten Differenz. Durch die Angabe von Indizes können Teilbereiche von Arrays untersucht werden. Wird kein Unterschied gefunden, so ist die Rückgabe `-1`.
- `compare()` – Vergleicht zwei Arrays analog zu Komparatoren. Auch hier kann der Vergleich auf einem Teilbereich der Arrays erfolgen.



# Erweiterungen Arrays

```
public static void main(final String[] args) throws Exception {
    final byte[] a = "Hallo Welt".getBytes();
    final byte[] b = "Hallo JDK 9".getBytes();
    final byte[] c = "JDK 9 Release".getBytes();

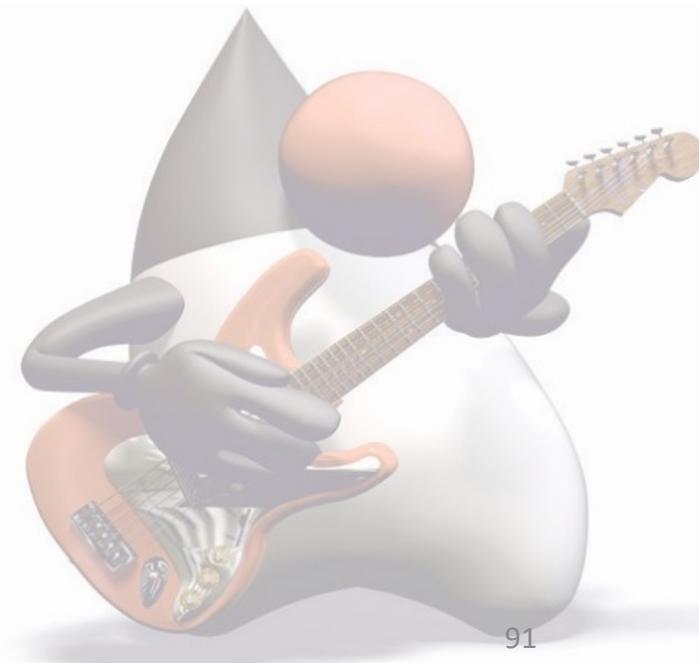
    executeEquals(a, b, c);
    executeMismatch(a, b, c);
    executeCompare(a, b, c);
}

private static void perform(final String info, final byte[] array1, final int start1, final
int end1, final byte[] array2, final int start2, final int end2, final Callable<Object>
action) throws Exception {
    final String value1 = new String(array1, start1, end1 - start1);
    final String value2 = new String(array2, start2, end2 - start2);
    System.out.print(info + "(" + value1 + ", " + value2 + ") => ");
    System.out.println(action.call());
}
```

# Erweiterungen Arrays

```
private static void executeEquals(final byte[] a, final byte[] b, final byte[] c) throws  
Exception {  
    perform("\n>equals", a, 0, 6, b, 0, 6, () -> Arrays.equals(a, 0, 6, b, 0, 6));  
    perform("equals", a, 0, 7, b, 0, 7, () -> Arrays.equals(a, 0, 7, b, 0, 7));  
    perform("equals", c, 0, 5, b, 6, 11, () -> Arrays.equals(c, 0, 5, b, 6, 11));  
}
```

```
equals('Hallo ', 'Hallo ') => true  
equals('Hallo W', 'Hallo J') => false  
equals('JDK 9', 'JDK 9') => true
```



# Erweiterungen Arrays

```
private static void executeMismatch(final byte[] a, final byte[] b, final byte[] c)
throws Exception {
    perform("\nmismatch", a, 0, 6, b, 0, 6, () -> Arrays.mismatch(a, 0, 6, b, 0, 6));
    perform("mismatch", a, 0, 7, b, 0, 7, () -> Arrays.mismatch(a, 0, 7, b, 0, 7));
}
```

```
mismatch('Hallo ', 'Hallo ') => -1
mismatch('Hallo W', 'Hallo J') => 6
```

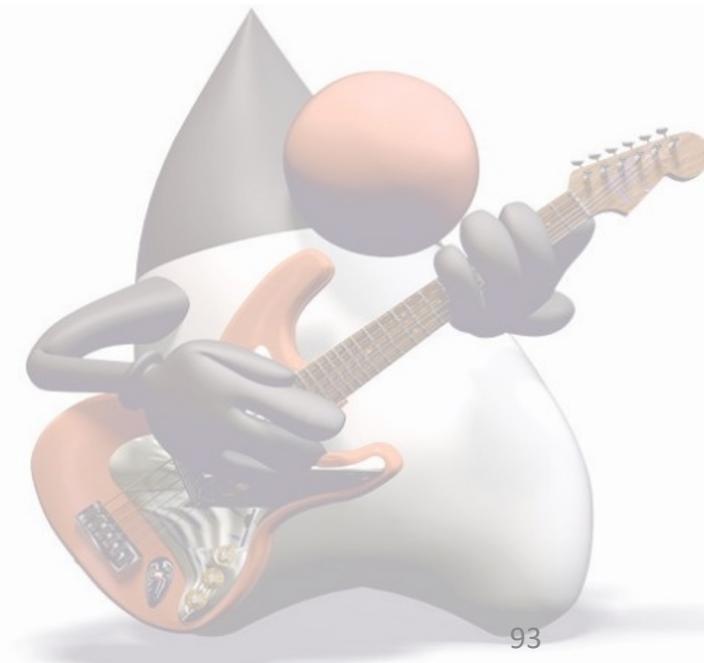
```
private static void executeCompare(final byte[] a, final byte[] b, final byte[] c) throws
Exception {
    perform("\ncompare", a, 0, 6, b, 0, 6, () -> Arrays.compare(a, 0, 6, b, 0, 6));
    perform("compare", a, 0, 7, b, 0, 7, () -> Arrays.compare(a, 0, 7, b, 0, 7));
    perform("compare", b, 0, 5, c, 0, 5, () -> Arrays.compare(b, 0, 5, c, 0, 5));
}
```

```
compare('Hallo ', 'Hallo ') => 0
compare('Hallo W', 'Hallo J') => 13
compare('Hallo', 'JDK 9') => -2
```



# Übersicht

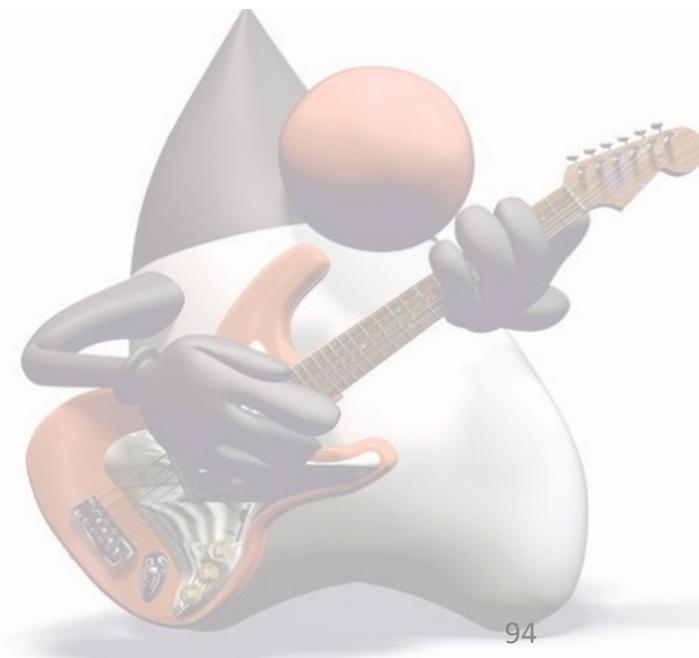
- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - **Erweiterungen Objects**
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# Erweiterungen Klasse Objects

Die Utility-Klasse `java.util.Objects` erlaubt es, durch die Methode `requireNonNull()` eine elegante Prüfung von Preconditions bezüglich null durchzuführen.

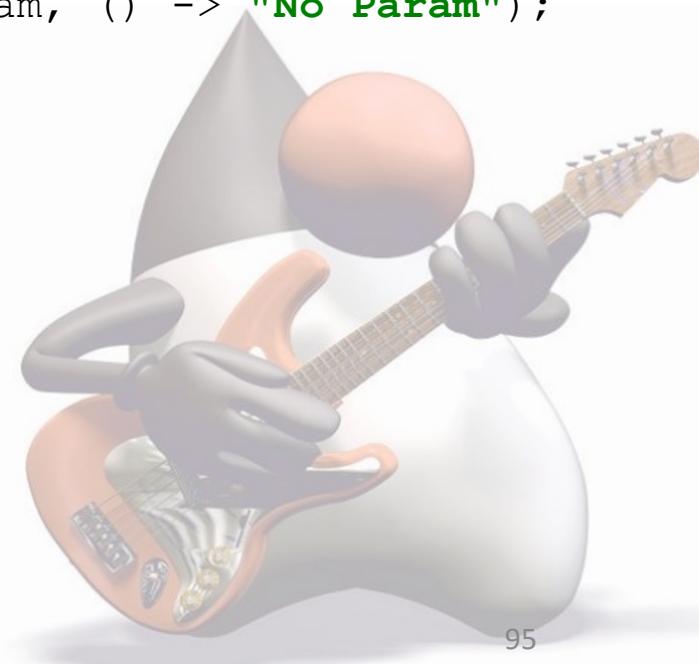
Mit JDK 9 wird das Ganze noch handlicher: Es ist nun analog zu einigen Methoden aus `Optional<T>` möglich, mit `requireNonNullElse()` bzw. `requireNonNullElseGet()` einen Alternativwert im Falle eines null-Werts bereitzustellen bzw. durch einen `Supplier<T>` zu berechnen.



# Erweiterungen Klasse Objects

```
public class ObjectsNonNullExample {  
    public static void main(final String[] args) {  
        System.out.println(generateMsg(null, null));  
    }  
  
    private static String generateMsg(final String msg, final String param) {  
        final String message = Objects.requireNonNullElse(msg, "Default-Msg");  
        final String parameter = Objects.requireNonNullElseGet(param, () -> "No Param");  
  
        return message + " : " + parameter;  
    }  
}
```

Default-Msg : No Param



# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - **Erweiterungen CompletableFuture**
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# Erweiterungen CompletableFuture<T>

In der Klasse CompletableFuture<T> wurden in Java 9 diverse Methoden ergänzt, unter anderem folgende:

- `completeAsync(Supplier<? extends T>)` und `completeAsync(Supplier<? extends T>, Executor)` – Erfüllt das CompletableFuture<T> mit dem vom übergebenen Supplier<T> gelieferten Ergebnis. Dieses wird asynchron von einem Task berechnet, der entweder vom Default Executor oder dem übergebenen ausgeführt wird.
- `orTimeout(long, TimeUnit)` – Sofern das CompletableFuture<T> nicht zuvor erfolgreich ausgeführt wurde, wird es mit einer `java.util.concurrent.TimeoutException` beendet, wenn die angegebene Time-out-Zeit erreicht ist.
- `completeOnTimeout(T, long, TimeUnit)` – Das CompletableFuture<T> wird mit dem übergebenen Wert erfüllt, falls die Berechnungen nicht innerhalb der gegebenen Time-out-Zeit zum Ergebnis führen.
- `failedFuture(Throwable)` – Gibt ein CompletableFuture<T> zurück, das bereits durch die übergebene Exception erfüllt wurde. Dies kann man zum Signalisieren von Fehlerzuständen während einer asynchronen Berechnung nutzen.



# CIIT Erweiterungen CompletableFuture<T>

javatraining

Seit Java 8:

```
public void perform() throws Exception {
    // supplyAsync(): eine einzelne Aktion parallel zum Aufrufer ausführen
    final CompletableFuture<String> supplyAsync =
        CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5));

    // thenAccept(): Aktion nach Abschluss der Ausführung
    supplyAsync.thenAccept(this::notifySubscribers);

    // exceptionally(): Exception-Mapping
    CompletableFuture.supplyAsync(this::failingMsg).exceptionally(ex -> "FAILED")
        .thenAccept(this::notifySubscribers);

    System.out.println(Thread.currentThread() + " perform()");
}

public String longRunningCreateMsg(final int durationInSecs) {
    System.out.println(Thread.currentThread() + " >>> longRunningCreateMsg");
    sleepInSeconds(durationInSecs);
    System.out.println(Thread.currentThread() + " <<< longRunningCreateMsg");

    return "longRunningCreateMsg";
}
```

# Erweiterungen CompletableFuture<T>

Seit Java 8:

```
public String getCurrentThread() {  
    return Thread.currentThread().getName();  
}  
  
public void notifySubscribers(final String msg) {  
    System.out.println(getCurrentThread() + " notifySubscribers: " + msg);  
}  
  
public String failingMsg() {  
    throw new IllegalStateException("ISE");  
}  
  
private void sleepInSeconds(final int durationInSeconds) {  
    try {  
        TimeUnit.SECONDS.sleep(durationInSeconds);  
    } catch (InterruptedException e) {  
        /* not possible here */  
    }  
}
```

# Erweiterungen CompletableFuture<T>

In der Klasse CompletableFuture<T> wurden in Java 9 diverse Methoden ergänzt, unter anderem folgende:

- `completeAsync(Supplier<? extends T>)` und `completeAsync(Supplier<? extends T>, Executor)` – Erfüllt das CompletableFuture<T> mit dem vom übergebenen Supplier<T> gelieferten Ergebnis. Dieses wird asynchron von einem Task berechnet, der entweder vom Default Executor oder dem übergebenen ausgeführt wird.
- `orTimeout(long, TimeUnit)` – Sofern das CompletableFuture<T> nicht zuvor erfolgreich ausgeführt wurde, wird es mit einer `java.util.concurrent.TimeoutException` beendet, wenn die angegebene Time-out-Zeit erreicht ist.
- `completeOnTimeout(T, long, TimeUnit)` – Das CompletableFuture<T> wird mit dem übergebenen Wert erfüllt, falls die Berechnungen nicht innerhalb der gegebenen Time-out-Zeit zum Ergebnis führen.
- `failedFuture(Throwable)` – Gibt ein CompletableFuture<T> zurück, das bereits durch die übergebene Exception erfüllt wurde. Dies kann man zum Signalisieren von Fehlerzuständen während einer asynchronen Berechnung nutzen.



# Erweiterungen CompletableFuture<T>

ab Java 9:

```
public void perform() throws ExecutionException
{
    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
        .completeAsync(() -> "COMPLETE")
        .thenAccept(this::notifySubscribers);

    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
        .orTimeout(3, TimeUnit.SECONDS)
        .exceptionally(ex -> "exception occurred: " + ex)
        .thenAccept(this::notifySubscribers);

    CompletableFuture.supplyAsync(() -> longRunningCreateMsg(5))
        .completeOnTimeout("TIMEOUT-FALLBACK", 2, TimeUnit.SECONDS)
        .thenAccept(this::notifySubscribers);

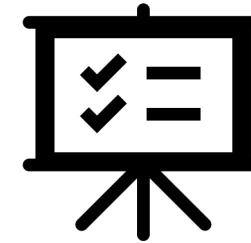
    CompletableFuture.failedFuture(new IllegalStateException())
        .exceptionally(ex -> { System.out.println("ALWAYS FAILING"); return -1; });

    // Give Completablefutures the chance to complete
    sleepInSeconds(10);
}
```

# Übungsbeispiel 07A

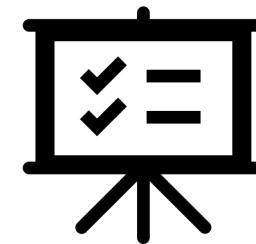
Experimentieren Sie mit untenstehendem Code

```
public class CompletableFutureDemo {  
    public static void main(String[] args) {  
        CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> 2 + 3)  
            .thenApplyAsync(i -> i * 2)  
            .thenApplyAsync(i -> i - 5)  
            .thenApplyAsync(i -> {  
                sleep(5000);  
                System.out.println("Returning");  
                return i;  
            })  
            .thenApply(i->{  
                System.out.println("And then");  
                return i;  
            })  
            .thenAcceptAsync(i -> System.out.println(i));  
  
        System.out.println("Finished");  
        future.join();  
        sleep(10000);  
    }  
}
```



# Übungsbeispiel 07A

Experimentieren Sie mit untenstehendem Code



```
private static void sleep(int i) {  
    try {  
        Thread.sleep(i);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```



# Übersicht

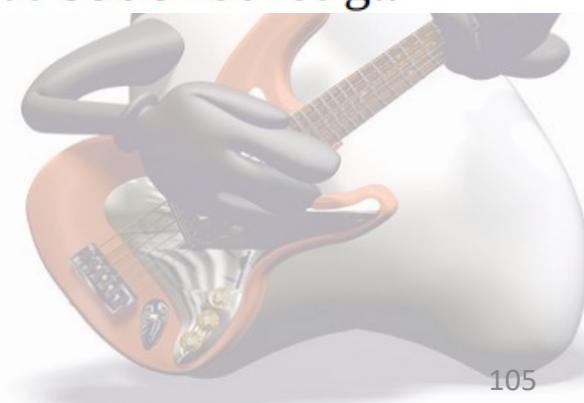
- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - **Optimierungen bei Strings**
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# Optimierungen Strings

Die Klasse `java.lang.String` dient bekanntermaßen der Modellierung von Zeichenketten. Um problemlos verschiedene Zeichensätze bzw. Zeichencodierungen wie ISO-8859-1 oder UTF-8 unterstützen zu können, verwendet die Klasse `String` bis einschließlich JDK 8 als interne Repräsentation der Zeichenkette ein `char[]`. Beobachtungen und Performance-Messungen aus vielen Jahren zeigen aber, dass `Strings` und `char[]` in diversen Java-Applikationen oftmals zu den Topverbrauchern bezüglich Speicher gehören. Zudem werden häufig nur Codierungen wie ASCII oder ISO-8859-1 genutzt, die sich durch lediglich ein Byte beschreiben lassen, wodurch pro Zeichen das andere Byte eines `char` unnötig Speicher belegt. Unter dem Namen »Compact Strings« wurde in JDK 9 eine Erweiterung realisiert, die dieses Problem adressiert.

Die Klasse `String` wurde mit JDK 9 so modifiziert, dass sie die Zeichenkette statt in einem `char[]` in einem `byte[]` namens `value` speichert. Ergänzend wird die Zeichencodierung als Attribut `coder` benötigt.



# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - **@Deprecated in Java 9**
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# @Deprecated in Java 9

## Deprecation der Typen **Observer** und **Observable**

Die Typen `java.util.Observer` und `java.util.Observer`, die ursprünglich zur einfacheren Realisierung des Entwurfsmusters BEOBACHTER dienen sollten, sind nun (endlich) als `@Deprecated` gekennzeichnet, weil sie ein verfehltes Design darstellen. Darauf und auf das Entwurfsmuster BEOBACHTER im Allgemeinen gehe ich in meinem Buch »Der Weg zum Java-Profi« [4] im Detail ein.



# @Deprecated in Java 9

```
@Deprecated(since="9")
```

```
public Boolean(boolean value)
```

```
@Deprecated(since="9")
```

```
public Boolean(String s)
```

Konstruktoren der Wrapper-Klassen Integer, Long, Float und Double

```
@Deprecated(since="9")
```

```
public Integer(int value)
```

```
@Deprecated(since="9")
```

```
public Integer(String s) throws NumberFormatException
```

# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# Änderungen Versionsschema

Bisher war die Versionierung der JDKs mitunter ein wenig verwirrend: Allein anhand der Versionsnummer fiel es extrem schwer, zwischen Minor Releases und Security Updates zu unterscheiden.<sup>2</sup> Auch sind in der Nummerierung die Varianten 8 und 1.8 gebräuchlich.<sup>3</sup> Dies wird nun mit JDK 9 vereinheitlicht und vor allem lassen sich Minor Releases und Critical Patch Updates viel klarer voneinander abgrenzen. Dazu kommt folgendes Versionsschema zum Einsatz: **MAJOR.MINOR.SECURITY**. Insbesondere wird der Ballast der führenden 1 sowie des u für Updates nun aus dem Versionsschema entfernt.

**Tabelle 4-1** Neu eingeführtes Versionsschema bei JDK 9

Version	Bedeutung	Inhalt
9	GA (General Availability)	
9.0.1	CPU (Critical Patch Update)	9 + critical changes
9.1.1	Minor Release	9.0.1 + other changes
9.1.2	CPU	9.1.1 + critical changes
9.2.2	Minor Release	9.1.2 + other changes
9.2.3	CPU	9.2.2 + critical changes

# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - **Multi-Release-Jars**
  - Jshell
  - HTML 5 JavaDoc



# Multi-Release-Jars

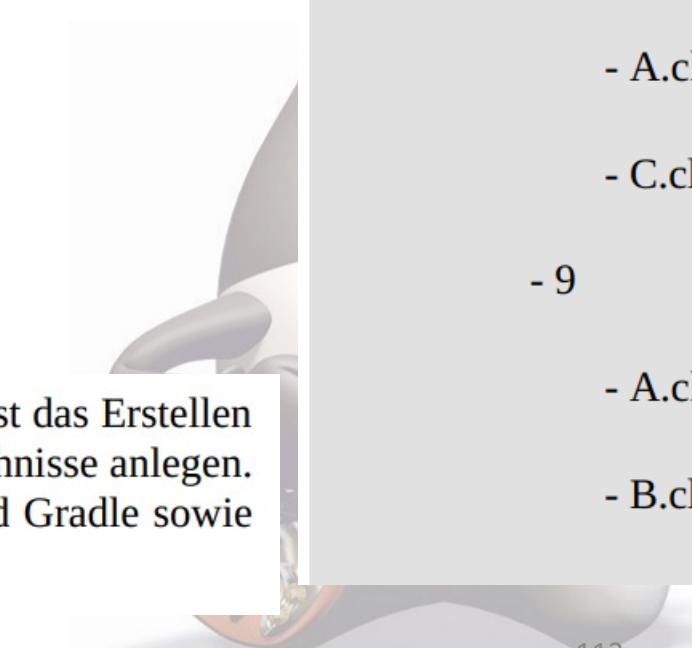
JAR-Dateien dienen bekanntlich dazu, Klassendefinitionen in Form eines ZIP-Archivs zu bündeln und anderen Applikationen bereitstellen zu können. Bisher spiegelt der Aufbau einer solchen Archivdatei den Package- bzw. Verzeichnisaufbau wider und erlaubt es demzufolge, jeweils genau eine Version einer kompilierten Java-Datei zu enthalten. Das ändert sich mit JDK 9: Innerhalb eines JARs kann man nun eine Substruktur aufbauen und .class-Dateien bereitstellen, die für verschiedene, bestimmte Java-Versionen gedacht sind.



# Multi-Release-Jars

Wir sehen drei Varianten der Klasse A. Für Java 9 kommt diejenige aus `versions/9` zum Einsatz. Für Java 8 die aus `versions/8`. Als Fallback wird die auf der Hauptebene hinterlegte Klassendefinition genutzt. Dies ist sinnvoll, weil dadurch eine Rückwärtskompatibilität der JAR-Dateien auch zu älteren JVMs gegeben ist, die das neue Feature nicht unterstützen.

- A.class
- B.class
- C.class
- META-INF
  - versions
    - 8
      - A.class
      - C.class
    - 9
      - A.class
      - B.class



Abgesehen von den zwei (oder mehr, je nach Anzahl unterstützter Versionen) Kompilierphasen ist das Erstellen eines Multi-Release-JARs recht einfach. Allerdings muss man sich auch mehrere Source-Verzeichnisse anlegen. Wünschenswert wäre dafür eine Konvention und der Support durch Build-Tools wie Maven und Gradle sowie durch die populären IDEs, was komfortabel noch nicht der Fall ist.

# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - **JShell**
  - HTML 5 JavaDoc



# JShell

In das JDK wurde mit Java 9 das Tool jshell integriert. Dieses erlaubt einen interaktiven Arbeitsstil und das Ausführen kleinerer Sourcecode-Schnipsel, wie man es bereits aus verschiedenen anderen Programmiersprachen in ähnlicher Form kennt. Man spricht dabei auch von REPL (Read-Eval-Print-Loop). Dadurch wird es möglich, etwas Java-Sourcecode zu schreiben und Dinge schnell auszuprobieren, ohne dafür die IDE starten und ein Projekt anlegen zu müssen.



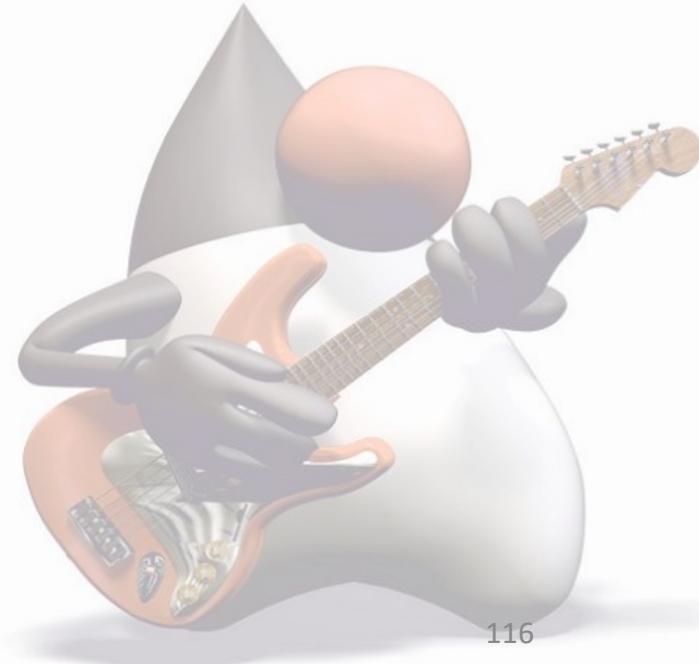
# JShell

```
jshell> int add(int a, int b) {  
...>     return a + b;  
...> }  
| created method add(int,int)
```

```
> jshell  
| Welcome to JShell -- Version 9.0.4  
| For an introduction type: /help intro  
  
jshell> System.out.println("Hello JShell")  
  
Hello JShell
```

```
jshell> add(3, $1)  
$3 ==> 7
```

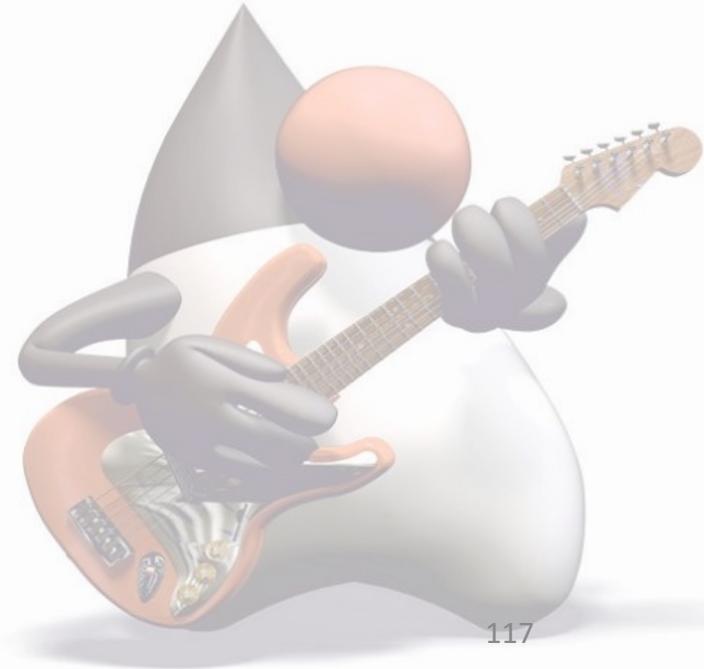
```
jshell> 2 + 2  
$1 ==> 4
```



# JShell

```
jshell> /vars  
| int $1 = 4
```

```
jshell> /methods  
| printf (String,Object...)void  
| add (int,int)int
```



# JShell

Außerdem bietet die jshell eine Historie der Befehle, was nützlich sein kann, um ein vorheriges Kommando wiederholt auszuführen. Mit `! /` lässt sich das letzte Kommando nochmals ausführen. Mit `/list` erhält man eine Übersicht, aus der mit `/<nr>` das `<nr>`te Kommando ausgeführt werden kann:

```
jshell> /list

1 : System.out.println("Hello JShell")

2 : 2+2

3 : int add(int a, int b) {

    return a+b;

}

4 : add(3, $2)
```



# JShell

Folgende Tastaturkürzel erleichtern in der jshell das Editieren und Navigieren:

- Ctrl + A / E – Springt an den Anfang / das Ende einer Zeile.
- ↑ / ↓ – Mit den Cursortasten kann man durch die Historie der Befehle navigieren.
- /reset – Löscht die Befehlshistorie.

Nicht immer ist uns jede mögliche Variante von Aufrufen geläufig, daher ist die Tab-Completion recht praktisch, die ähnlich wie in einer IDE eine Reihe möglicher Vervollständigungen präsentiert:

```
jshell> String.
```

```
CASE_INSENSITIVE_ORDER  class  copyValueOf(  format(  join(
```

```
valueOf(
```

```
jshell> Class.
```

```
class  forName(
```

# JShell

```
jshell> List<Integer> numbers = List.of(1,2,3,4,5,6,7)
```

```
numbers ==> [1, 2, 3, 4, 5, 6, 7]
```

```
jshell> Set<String> names = Set.of("Tim", "Mike", "Max")
```

```
names ==> [Tim, Max, Mike]
```

```
jshell> Map<String, Integer> nameToAge = Map.of("Tim", 41, "Mike", 42)
```

```
nameToAge ==> {Tim=41, Mike=42}
```



# JShell

**Einbinden anderer JDK-Klassen** Standardmäßig sind in der jshell nur Typen aus dem Modul `java.base` zugreifbar,<sup>6</sup> was aber für diverse Java-Gehversuche oftmals ausreichend ist. Möchte man aber Klassen etwa aus Swing oder JavaFX nutzen, so benötigen wir einen Import, beispielsweise `import javax.swing.*` für die Typen wie `JFrame` usw. Ohne diesen erhalten wir folgende Fehlermeldung:

```
jshell> new JFrame("Hello World")
```

```
| Error:
```

```
| cannot find symbol
```

```
| symbol: class JFrame
```

```
| new JFrame("Hello World")
```

```
|     ^----^
```



## JShell

```
import javax.swing.*
```

```
jshell> new JFrame("Hello World")
```

```
$2 ==> javax.swing.JFrame[frame0,0,23,0x0,invalid,hidden,layout=java.awt.BorderLayout,title=Hello World,resizable,normal,defaultCloseOperation=HIDE_ON_CLOSE,rootPane=javax.swing.JRootPane[,0,0,0x0,invalid,layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=16777673,maximumSize=,minimumSize=,preferredSize=],rootPaneCheckingEnabled=true]
```

```
$2.setSize(200, 50)
```

```
$2.show()
```

# JShell

Wenn man weiß, dass ein paar mehr Zeilen zu editieren sind, so kann man den in die jshell integrierten Editor per /edit aufrufen und nutzen. Dieser führt auch eine Syntaxprüfung durch, sobald man die editierten Programmzeilen übernimmt. Ein möglicher Editvorgang, der die obige Klasse Person noch in einer früheren Entwicklungsstufe zeigt, ist in [Abbildung 4-2](#) dargestellt.



The screenshot shows the Java JShell interface. The terminal window title is "jigsaw\_ch6 — java • jshell — 118x39" and the path is "/Users/michaeli/Desktop/PureJava9/quelltext/jigsaw\_ch6 — java • jshell". The terminal output shows:

```
Michaels-MBP-2:jigsaw_ch6 michaeli$ jshell
| Welcome to JShell -- Version 9-ea
| For an introduction type: /help intro

jshell> /edit
| Error:
| missing return statement
| ^
|
```

To the right, there is an "Edit Pad" window titled "JShell Edit Pad" containing the following Java code:

```
class Person
{
    private String name;

    public Person(String name)
    {
```



# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - JShell
  - **HTML 5 JavaDoc**



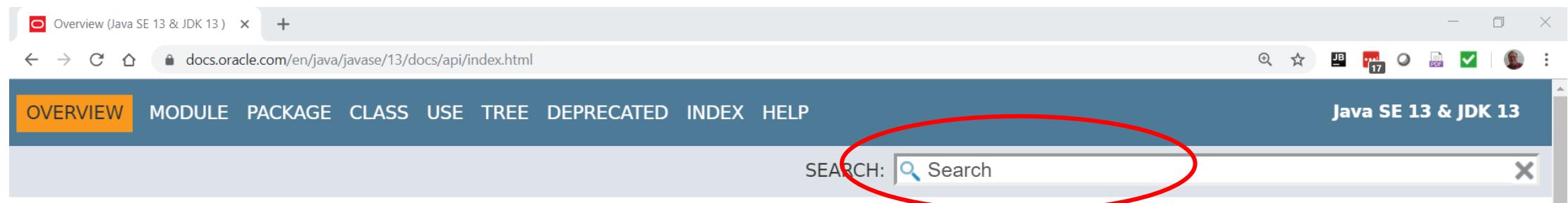
# HTML 5 JavaDoc

```
javadoc -public -sourcepath src -d dochtml5 -html5 -subpackages jdk9example
```

Die sichtbaren Änderungen im angezeigten Javadoc-HTML sind im Vergleich zu der Option `-html4` minimal. Ich habe ein paar leicht abweichende Abstände und Formatierungen gesehen. Auf Ebene des HTML-Sourcecodes konnte ich auch nur marginale Abweichungen feststellen. Insofern bietet der Schalter `-html5` wohl ein eher kosmetisches Update, bringt aber Javadoc wieder auf einen aktuellen Stand und lässt für die Zukunft hoffen, dass die Unterstützung von HTML5 noch erweitert wird.



# HTML 5 JavaDoc - Suche



## Java® Platform, Standard Edition & Java Development Kit Version 13 API Specification

This document is divided into two sections:

### Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

### JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules   Java SE   JDK   Other Modules

Module	Description
--------	-------------

# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - Unmodifiable Lists
  - Erweiterungen Optional<T>
  - Erneute Änderungen Versionierung
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas



# Syntaxerweiterung var

```
var name = "Peter";           // var => String  
  
var chars = name.toCharArray(); // var => char[]  
  
var mike = new Person("Mike", 47); // var => Person  
  
var hash = mike.hashCode();    // var => int
```

```
// var => ArrayList<String>           names.add("Jerry");  
  
var names = new ArrayList<String>();  
names.add("Tim");                     // var => Map<String, Long>  
  
names.add("Tom");  
  
var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L,  
                            "Michael", 47L, "Max", 25L);
```

# Syntaxerweiterung var

Insbesondere wenn die Typangaben mehrere generische Parameter umfassen, kann var den Sourcecode deutlich kürzer und mitunter lesbarer machen. Betrachten wir als Beispiel eine Verschachtelung von Typen analog zu den folgenden:

- `Set<Map.Entry<String, Long>>`
- `Map<Character, Set<Map.Entry<String, Long>>>`

In solchen Fällen spart var einiges an Schreibarbeit – zusätzlich erfolgt hier noch ein statischer Import verschiedener Kollektoren, um die Lesbarkeit zu steigern:

```
// var => Set<Map.Entry<String, Long>>

var entries = personAgeMapping.entrySet();

// var => Map<Character, Set<Map.Entry<String, Long>>>

var filteredPersons = personAgeMapping.entrySet().stream()

    collect(groupingBy(firstChar,
        filtering(isAdult, toSet())));
```



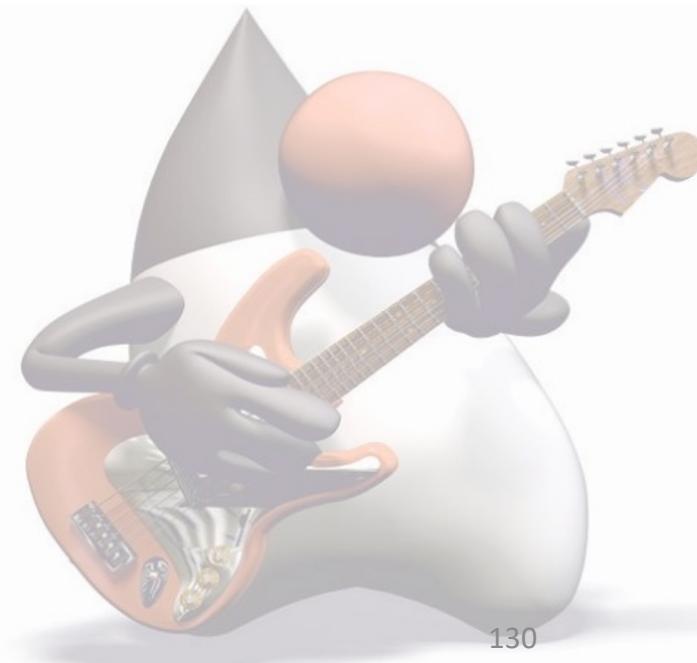
# Syntaxerweiterung var

**Restriktion auf exakten Typ** Beim Einsatz von var sollte man wissen, dass immer der exakte Typ verwendet wird und nicht ein Basistyp, wie man es für Collections getreu dem Paradigma »program against interfaces« sehr gerne macht. Beachten Sie bitte, dass im Folgenden die Variable names deshalb nicht vom Typ List<String> ist, sondern vom Typ ArrayList<String>:

```
// var => ArrayList<String>

var names = new ArrayList<String>();

names = new LinkedList<String>(); // Compile Error
```



# Syntaxerweiterung var

**Weitere syntaktische Besonderheiten** Es gibt weitere Dinge, die zu Kompilierfehlern führen. Das sind eine fehlende Wertangabe und auch eine fehlende Typangabe bei direkten Array-Initialisierungen:

```
var justDeclaration;          // keine Wertangabe / Definition  
  
var numbers = {0, 1, 2};      // fehlende Typangabe
```



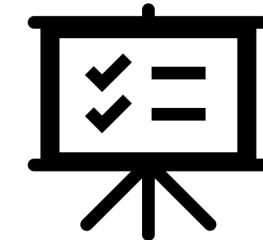
# Syntaxerweiterung var

```
var mixedContent = new ArrayList<>();  
  
mixedContent.add("Strange with var");  
  
mixedContent.add(42);
```

Kompiliert und funktioniert das? Und wenn ja, was ist daran problematisch? Tatsächlich produziert das Ganze keinen Kompilierfehler. Wie kommt das? Aufgrund des Diamond Operators bzw. der nicht vorhandenen Typangabe stehen dem Compiler nicht ausreichend Typinformationen zur Verfügung: Deswegen wird `java.lang.Object` als generischer Parameter genutzt und aus der zuvor auf `String` typisierten Liste wird – wohl eher unerwartet – eine `ArrayList<Object>`!



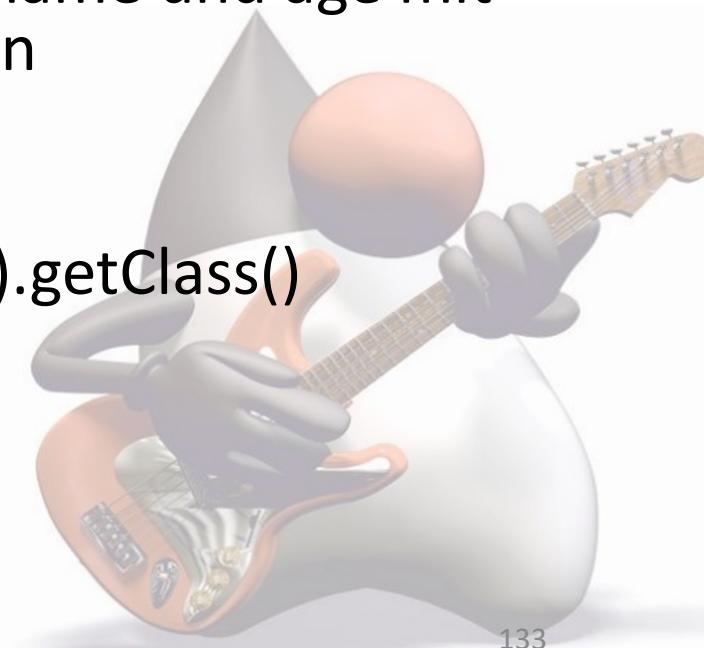
# Übungsbeispiel 08



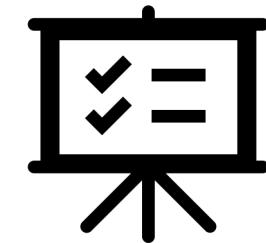
Lernen Sie nachfolgend das neue reservierte Wort var mit seinen Möglichkeiten und Beschränkungen kennen.

**Aufgabe 8a:** Starten Sie die JShell oder eine IDE Ihrer Wahl. Erstellen Sie eine Methode funWithVar(). Definieren Sie dort die Variablen name und age mit den Werten „Mike“ bzw. 47. Geben Sie anschließend deren Typ aus.

**Tipp:** Für die zweite Variable hilft Folgendes: ((Object)age).getClass()



# Übungsbeispiel 08



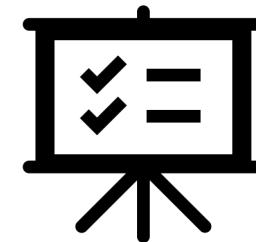
**Aufgabe 8b:** Erweitern Sie Ihr Know-how bezüglich var und Generics.  
Nutzen Sie es für folgende Definition:

```
Map.of ("Tim", 47, "Tom", 7, "Mike", 47);
```

Erzeugen Sie initial zunächst eine lokale Variable personsAndAges und vereinfachen Sie dann mithilfe von var.

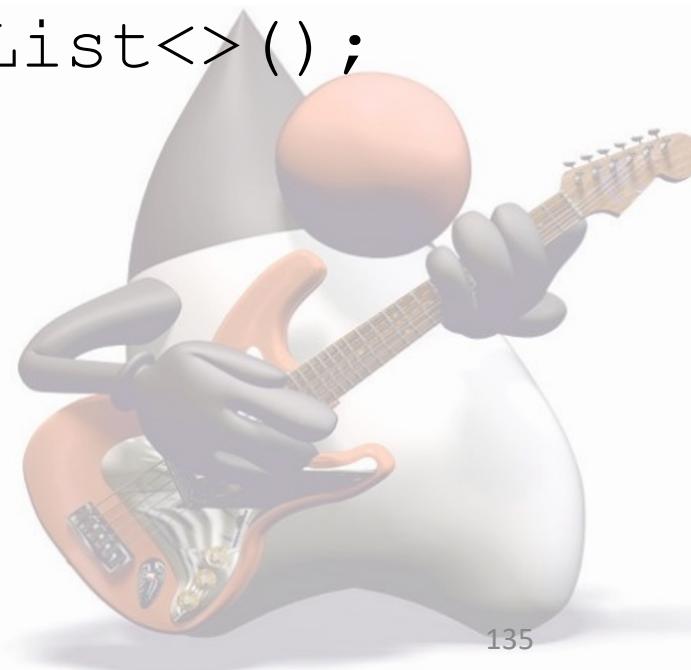


# Übungsbeispiel 08

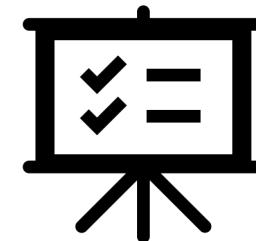


**Aufgabe 8c:** Vereinfachen Sie folgende Definitionen mit var. Was ist zu beachten? Worin liegt der Unterschied?

```
List<String> names = new ArrayList<>();  
ArrayList<String> names2 = new ArrayList<>();
```



# Übungsbeispiel 08



**Aufgabe 8d:** Wieso führen folgende Lambdas zu Problemen? Wie löst man diese?

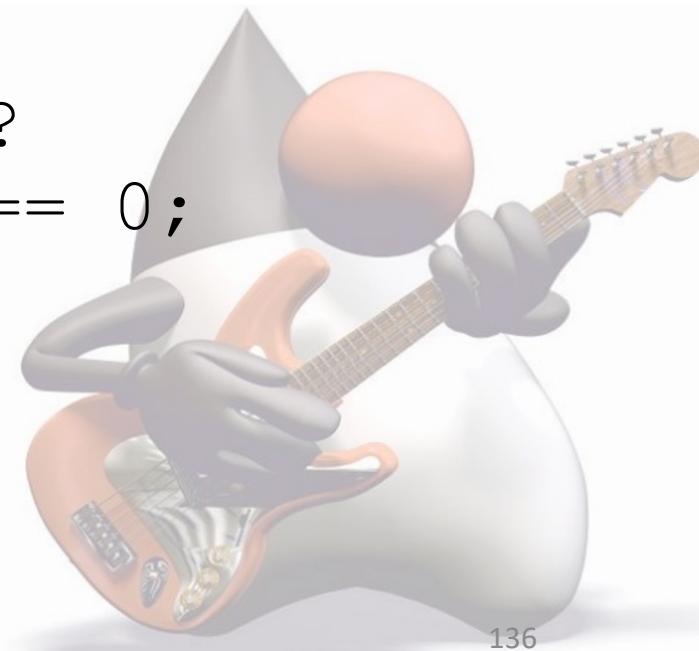
```
var isEven = n -> n % 2 == 0;
```

```
var isEmpty = String::isEmpty;
```

Wieso kompiliert dann aber Folgendes?

```
Predicate<Long> isEven = n -> n % 2 == 0;
```

```
var isOdd = isEven.negate();
```



# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - **Unmodifiable Lists**
  - Erweiterungen Optional<T>
  - Erneute Änderungen Versionierung
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas



# Copy Unmodifiable Lists

```
java.util.List<E>
@Unmodifiable
public static <E> List<E> copyOf(@NotNull java.util.Collection<? extends E> coll)
```

Returns an [unmodifiable List](#) containing the elements of the given Collection, in its iteration order. The given Collection must not be null, and it must not contain any null elements. If the given Collection is subsequently modified, the returned List will not reflect such modifications.



# Copy Unmodifiable Lists

Für den Fall, dass man eine unveränderliche Kopie einer Liste benötigt, macht die in Java 10 neu eingeführte Methode `copyOf()` das Leben deutlich einfacher als noch mit Java 9. Das zeigt folgender Aufruf eindrucksvoll:

```
final List<String> newImmutableCopy = List.copyOf(names);
```

Selbstverständlich gibt es Analogien für Sets und Maps. Das sind die Methoden `Set.copyOf(originalSet)` und `Map.copyOf(originalMap)`.



# Copy Unmodifiable Lists

Basierend auf einer Liste mit teils doppelten Namen wollen wir eine unveränderliche Kopie als Liste bzw. als Set erzeugen. Die oben genannten Methoden haben wir der besseren Lesbarkeit halber statisch importiert:

```
var names = List.of("Tim", "Tom", "Mike", "Peter", "Tom", "Tim");

var immutableNames = names.stream().collect(toUnmodifiableList());

System.out.println("immutableNames type: " + immutableNames.getClass());

var uniqueImmutableNames = names.stream().collect(toUnmodifiableSet());

System.out.println("uniqueImmutableNames content: " +
    uniqueImmutableNames);

System.out.println("uniqueImmutableNames type: " +
    uniqueImmutableNames.getClass());
```



# Copy Unmodifiable Lists

Kommen wir schließlich zur Erzeugung unveränderlicher Maps. Als Beispiel hierzu bereiten wir folgendermaßen aus den vorherigen Daten ein Histogramm auf, das die Häufigkeit der jeweiligen Namen repräsentiert.

```
Function<String, Long> valueMapper = value -> 1L;
```

```
BinaryOperator<Long> mergeFunction = (count, inc) -> count + inc;
```

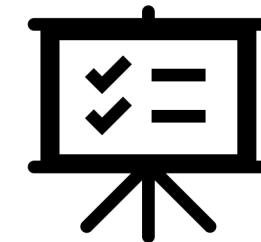
```
var nameCountMap = names.stream().collect(toUnmodifiableMap(Function.identity(),
```

```
        valueMapper,
```

```
        mergeFunction));
```

```
System.out.println("content: " + nameCountMap);
```

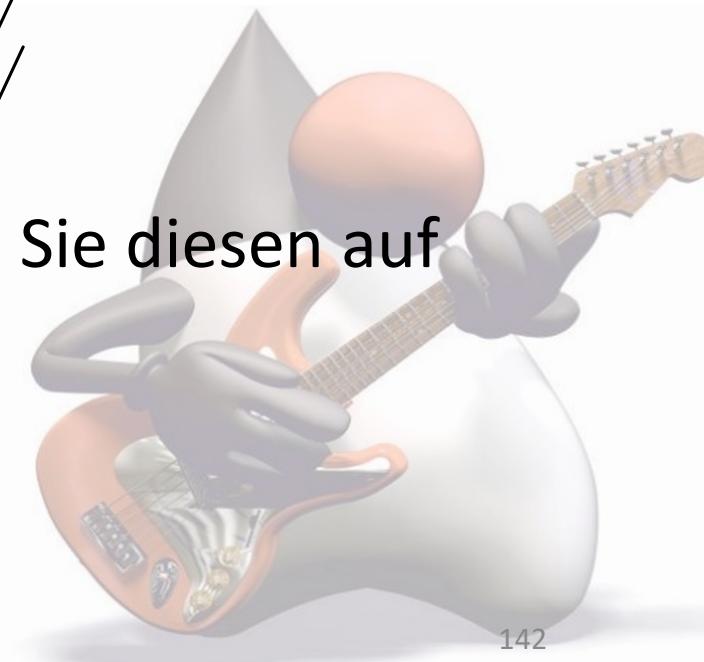
# Übungsbeispiel 09



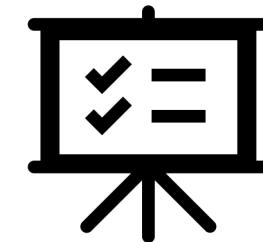
**Aufgabe 9a:** Welche zwei Varianten bietet Java 10, um eine unveränderliche Kopie folgender Liste zu erstellen?

```
List<String> words = List.of("Hello", "World");  
List<String> copy1 = null; /* TODO */  
List<String> copy2 = null; /* TODO */
```

Welchen Typ haben die entstehenden Kopien? Geben Sie diesen auf der Konsole aus



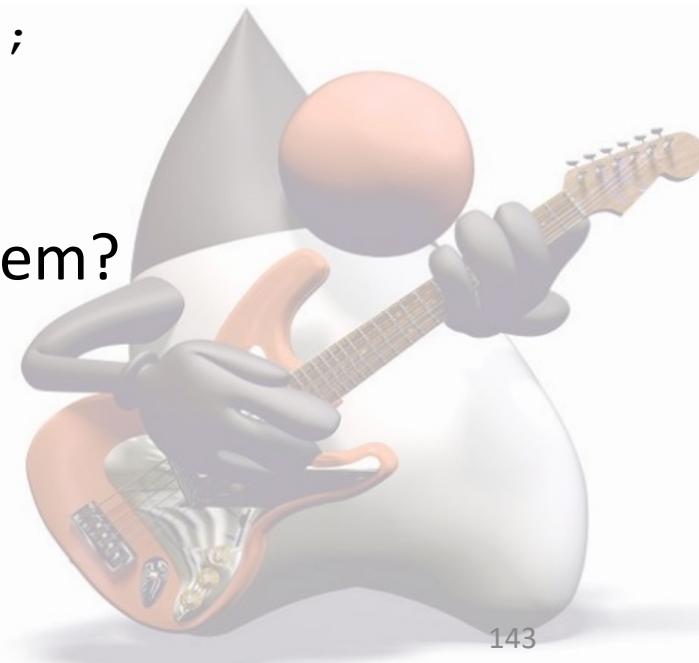
# Übungsbeispiel 09

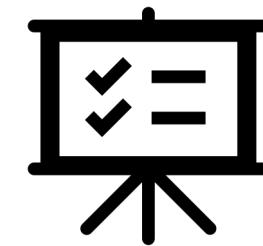


**Aufgabe 9b:** Worin liegt der Unterschied innerhalb der entstehenden unmodifizierbaren Listen, wenn man eine Modifikation im Array vornimmt?

```
final String[] nameArray = { "Tim", "Tom", "Mike" };  
// 3 Varianten von unmodifizierbaren Listen  
final List<String> names1 = Arrays.asList(nameArray);  
final List<String> names2 = List.of(nameArray);  
final List<String> names3 = List.of("Tim", "Tom", "Mike");  
// Modifikation im Array  
nameArray[2] = "Michael";
```

Was geschieht beim Ändern in der Liste analog zu Folgendem?  
`names1.set(1, "XXX");`





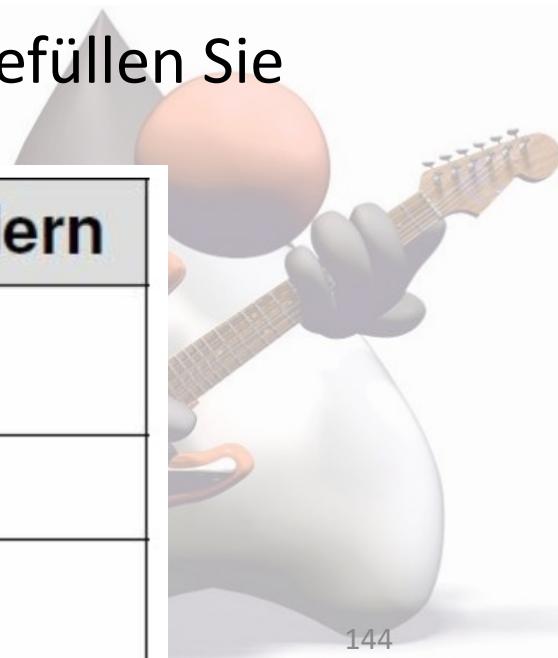
# Übungsbeispiel 09

**Aufgabe 9c:** Was unterscheidet die drei Varianten von Kopien bzw. Wrapper?

```
List<String> names = List.of("Tim", "Tom", "Mike");  
// 3 Varianten, um Kopien von unmodifizierbaren Listen zu erzeugen  
List<String> copy1 = new ArrayList<>(names);  
List<String> copy2 = List.copyOf(names);  
List<String> wrapper = Collections.unmodifiableList(names);
```

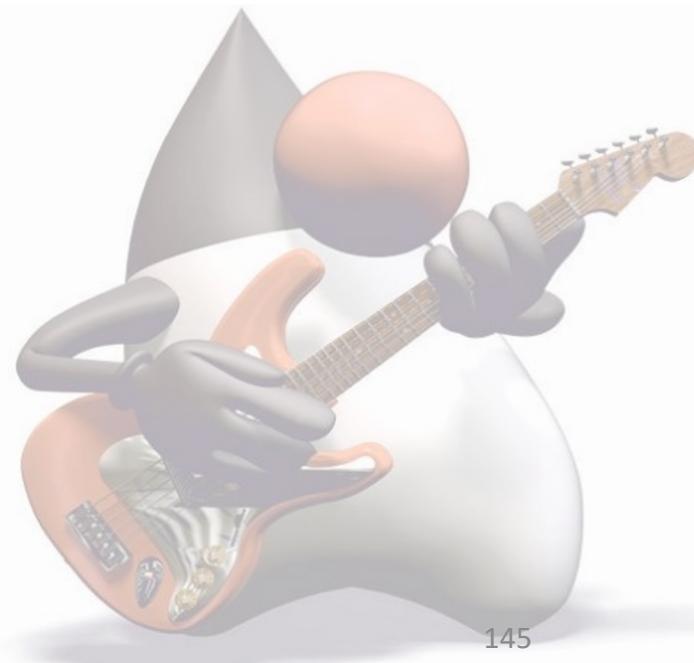
Lassen sich Elemente ändern? Lassen sich Elemente löschen? Befüllen Sie nachfolgende Tabelle.

Variante	Löschen	Ändern
copy1		
copy2		
wrapper		



# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - Unmodifiable Lists
  - Erweiterungen `Optional<T>`
  - Erneute Änderungen Versionierung
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas

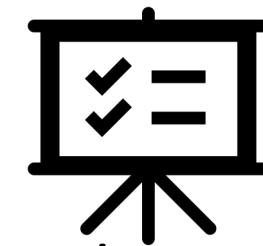


# Erweiterungen Optional<T>

Die Klasse `java.util.Optional<T>` war eine enorme Bereicherung in Java 8: Lange hatte sich die Java-Gemeinde gewünscht, optionale Werte, vor allem für Rückgaben<sup>2</sup>, geeignet modellieren zu können. Mit Java 9 wurde das API durch die drei Methoden `ifPresentOrElse()`, `or()` und `stream()` vervollständigt. Damit schien das API recht komplett. Allerdings hat man sich bei Oracle an einer Unschönheit gestört: an der Methode `get()`. Wieso?

Die Antwort ist einfach: Die Methode `get()` zum Zugriff auf den Wert eines `Optional<T>` sieht zu harmlos aus. Vermutlich dadurch findet man in der Praxis mitunter einen Aufruf von `get()` ohne vorherige Prüfung auf Existenz eines Werts mit `isPresent()`. Das führt dann aber bei einem nicht vorhandenen Wert zu einer `java.util.NoSuchElementException`. Normalerweise erwartet man beim Aufruf einer `get()`-Methode allerdings nicht unbedingt, dass diese eine Exception auslöst. Um diesen Sachverhalt im API direkt ausdrücken zu können, wurde die Methode `orElseThrow()` als Alternative zu `get()` in das JDK aufgenommen, die bei Vorhandensein, den Wert liefert und ansonsten eben eine `NoSuchElementException` auslöst.<sup>3</sup>





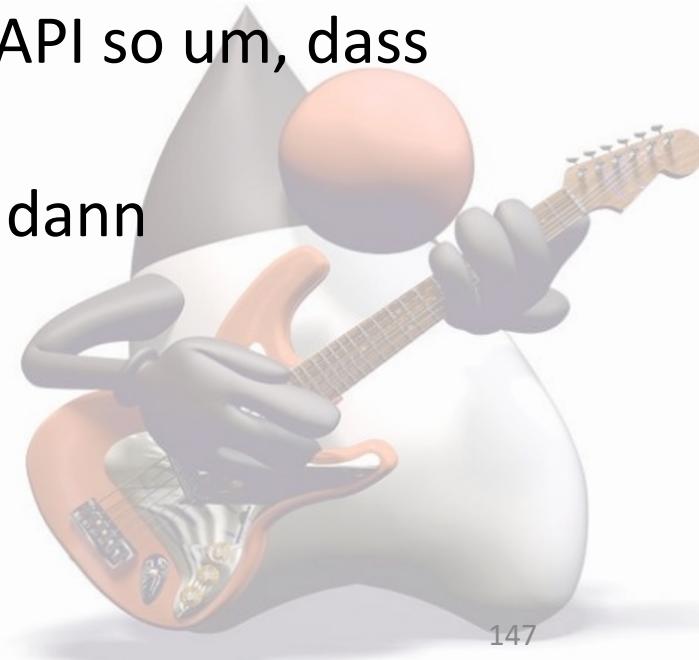
# Übungsbeispiel 10

Gegeben sei eine simple, aber unsichere Wertextraktion aus einem `Optional<T>` wie folgt:

```
static <T> T badStyleExtractValue(final Optional<T> opt) {  
    T value = opt.get();  
    return value;  
}
```

**Aufgabe 10a:** Schreiben Sie diese mit dem neuen Java-10-API so um, dass der potenziell unsichere Zugriff offensichtlich wird.

**Aufgabe 10b:** Wie würde man die Methode oder Aufrufer dann umgestalten?



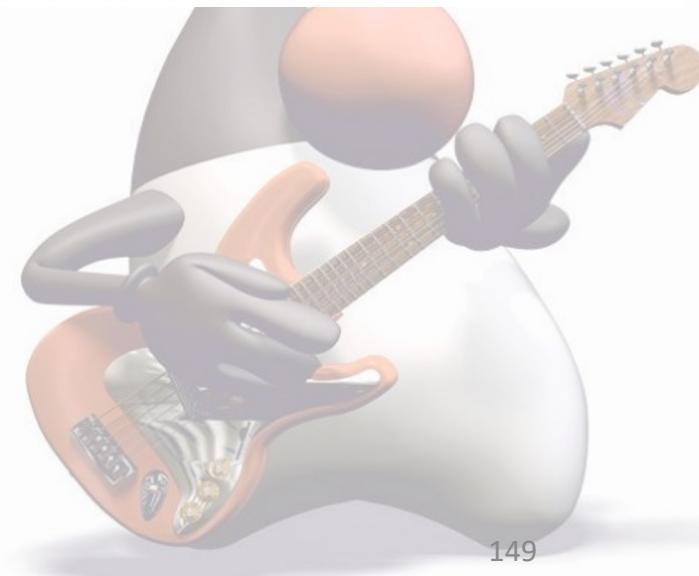
# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - Unmodifiable Lists
  - Erweiterungen Optional<T>
  - **Erneute Änderungen Versionierung**
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas



# Erneute Modifikationen in der Versionierung

Nachdem mit Java 9 endlich ein sinnvolles, nachvollziehbares Versionsschema existiert und zudem die unterstützende Klasse `java.lang.Runtime.Version` in das JDK aufgenommen wurde, ist das Ganze aufgrund einer geänderten Releasepolitik von Oracle leider mit Java 10 bereits teilweise wieder über der Haufen geworfen worden. Man spricht nun nicht mehr von Major- und Minor-Version sowie Security-Patch mit dem Format `MAJOR.MINOR.SECURITY`, sondern von Feature, Interim, Update und Patch mit dem Format `FEATURE.INTERIM.UPDATE.PATCH`. Deswegen sind die Methoden `major()`, `minor()` und `security()` seit Java 10 deprecated und rufen die Methoden `feature()` bzw. `interim()` sowie `update()` auf.



# Erneute Modifikationen in der Versionierung

## Hinweis: Oracles neue Releasepolitik

Bis einschließlich Java 9 wurden neue Java-Versionen immer Feature-basiert veröffentlicht. Das hatte in der Vergangenheit oftmals und mitunter auch beträchtliche Verschiebungen des geplanten Releasetermins zur Folge, wenn ein für die Version wesentliches Feature noch nicht fertig war. Insbesondere deshalb verzögerten sich Java 8 und Java 9 um mehrere Monate bzw. sogar über ein Jahr.

Mit einer zeitbasierten Releasestrategie möchte man dem entgegenwirken. Diese sieht vor, dass jedes halbe Jahr eine neue Java-Version veröffentlicht wird und all jene Features enthält, die bereits fertig sind. Alle drei Jahre ist dann eine sogenannte LTS-Version (Long Term Support) geplant. Eine solche ist in etwa vergleichbar mit den früheren Major-Versionen. Zumindest bezüglich der halbjährlichen Releases hat Oracle schon Wort gehalten, wenn auch Java 9 etwas überraschend nicht als LTS ausgelegt war. Schauen wir gespannt in die Zukunft.



# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - Unmodifiable Lists
  - Erweiterungen Optional<T>
  - Erneute Änderungen Versionierung
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas



# Verschiedenes

- long transferTo(Writer) aus der Klasse java.io.Reader: Es werden alle Zeichen aus dem Reader in den übergebenen Writer übertragen – diese Funktionalität existiert analog in der Klasse java.io.InputStream seit Java 9.

```
var sr = new StringReader("Hello");

var sw = new StringWriter();

sr.transferTo(sw);

System.out.println("SW: " + sw.toString());
```

# CIIT Verschiedenes

- **String toString(Charset)** aus der Klasse `java.io.ByteArrayOutputStream`: Das ist eine überladene Variante der `toString()`-Methode, die das übergebene Charset verwendet. Im nachfolgenden Beispiel definieren wir die vier Buchstaben A, B, C und D als ASCII-Codierung in Form eines Byte-Arrays. Im Anschluss verarbeiten wir sie dann mit den passenden Streams wie folgt:

```
var values = new byte[]{'A', 'B', 'C', 'D'};
```

```
var is = new ByteArrayInputStream(values);
```

```
var bos = new ByteArrayOutputStream();
```

```
is.transferTo(bos);
```

```
System.out.println(bos.toString());
```

```
System.out.println(bos.toString(StandardCharsets.UTF_16));
```



Beim Ausprobieren des obigen Sourcecodes erhält man zunächst die vier Buchstaben als Ausgabe. Wechselt man das Charset in UTF-16, so ändert sich die Ausgabe in chinesische Schriftzeichen. Das ist nachfolgend in [Abbildung 6-2](#) angedeutet:

---

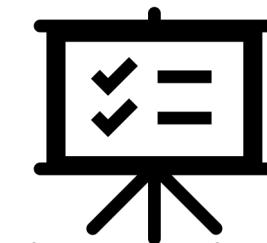
ABCD  
格鈺



# Fazit Java 10

Java 10 ist eher ein kosmetisches Release und hätte meines Erachtens auch genauso gut die Versionsnummer 9.1 oder 9.2 tragen können – über die neue Releasepolitik von Oracle lässt sich kontrovers diskutieren: Änderungen schneller veröffentlichen zu können, ist sicher positiv. Jedoch dafür immer die Major-Version hochzuzählen, gleichzeitig aber kaum relevante Neuerungen zu integrieren, kann man infrage stellen.





# Übungsbeispiel 11

Transferieren Sie den Inhalt aus einem StringReader in eine Datei hello.txt. Lesen Sie diese wieder ein und geben Sie den Inhalt aus. Ergänzen Sie dazu folgende Zeilen:

```
var textFile = new File("hello.txt");
var sr = new StringReader("Hello\nWorld");
try (Writer bfw = null /*TODO*/) {

    // TODO

}

var sw = new StringWriter();

try (Reader bfr = null /*TODO*/) {

    // TODO

}
System.out.println(sw.toString());
```



# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - Unmodifiable Lists
  - Erweiterungen Optional<T>
  - Erneute Änderungen Versionierung
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas



# var Type Interference für Lambdas

Bis einschließlich Java 10 konnte man in einem Lambda-Ausdruck bei den Parametern entweder alle Typen angeben oder alle weglassen. Die Verwendung von var war dort nicht möglich. Das wurde mit Java 11 geändert, sodass folgender Lambda-Ausdruck nun korrekt ist:

```
(var x, var y) -> x.doSomething(y)
```

Blicken wir kurz zurück: In Java 10 waren diese beiden Varianten erlaubt:

```
IntFunction<Integer> doubleItTyped = (final int x) -> x * 2;
```

```
IntFunction<Integer> doubleItNoType = x -> x * 2;
```



# var Type Interference für Lambdas

Nicht unterstützt wurde jedoch die Angabe von var:

```
IntFunction<Integer> doubleItWithVar = (var x) -> x * 2;
```

Warum sollte aber diese Variante überhaupt nützlich sein, wo man doch vollständig auf die Typangabe verzichten kann? Eine berechtigte Frage! Die Antwort ergibt sich aus dem Wunsch, zwar keinen Typ explizit angeben zu müssen, jedoch etwa den Parameter trotzdem final definieren oder Annotations hinzufügen zu können. Das geht natürlich nicht, wenn die Parameterangabe vollständig ohne Typ genutzt wird. Hier erlaubt var, auf die explizite Typangabe zu verzichten und trotzdem weiter gehende Informationen bereitzustellen zu können. Das wird nachfolgend durch eine Annotation @NotNull verdeutlicht:

```
Function<String, String> trimmer = (@NotNull var str) -> str.trim();
```

# var Type Interference für Lambdas

Nicht unterstützt wurde jedoch die Angabe von var:

```
IntFunction<Integer> doubleItWithVar = (var x) -> x * 2;
```

Warum sollte aber diese Variante überhaupt nützlich sein, wo man doch vollständig auf die Typangabe verzichten kann? Eine berechtigte Frage! Die Antwort ergibt sich aus dem Wunsch, zwar keinen Typ explizit angeben zu müssen, jedoch etwa den Parameter trotzdem final definieren oder Annotations hinzufügen zu können. Das geht natürlich nicht, wenn die Parameterangabe vollständig ohne Typ genutzt wird. Hier erlaubt var, auf die explizite Typangabe zu verzichten und trotzdem weiter gehende Informationen bereitzustellen zu können. Das wird nachfolgend durch eine Annotation @NotNull verdeutlicht:

```
Function<String, String> trimmer = (@NotNull var str) -> str.trim();
```

# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



# Neue Hilfsmethoden von String

Die Klasse `java.lang.String` existiert seit JDK 1.0 und hat zwischenzeitlich nur wenige API-Änderungen erfahren. Mit Java 11 ändert sich das. Es wurden folgende Methoden neu eingeführt:

- `isBlank()`
- `lines()`
- `repeat(int)`
- `strip()`
- `stripLeading()`
- `stripTrailing()`



# Neue Hilfsmethoden von String - isBlank

Für Strings war es bisher mühsam oder nur mithilfe von externen Bibliotheken möglich, zu prüfen, ob diese ausschließlich Whitespaces enthalten. Als Abhilfe wurde mit Java 11 die Methode `isBlank()` eingeführt, die sich auf `Character.isWhitespace(int)` stützt. Folgendes Beispiel demonstriert die Prüfung auf einen leeren oder lediglich Whitespaces enthaltenden String:

```
public class BlankTester {  
    public static void main(String... args) {  
        final String exampleText1 = "";  
        final String exampleText2 = " ";  
        final String exampleText3 = " \n \t ";  
        System.out.println(exampleText1.isBlank());  
        System.out.println(exampleText2.isBlank());  
        System.out.println(exampleText3.isBlank());  
    }  
}
```

```
true  
true  
true
```



# Neue Hilfsmethoden von String – lines()

Beim Verarbeiten von Daten aus Dateien müssen des Öfteren Informationen in einzelne Zeilen aufgebrochen werden. Dazu gibt es etwa die Methode Files.lines(Path). Ist die Datenquelle allerdings schon ein String, existierte diese Funktionalität bislang noch nicht. Mit JDK 11 finden wir in der Klasse String die Methode lines(), die einen Stream<String> zurückliefert und wie folgt genutzt werden kann:

```
public class LinesTester {  
    public static void main(String... args) {  
        final String exampleText = "1 This is a\n2 multi line\n" +  
            "3 text with\n4 four lines!";  
        final Stream<String> lines = exampleText.lines();  
        lines.forEach(System.out::println);  
    }  
}
```

```
1 This is a  
2 multi line  
3 text with  
4 four lines!
```



# Neue Hilfsmethoden von String – repeat()

Immer mal wieder steht man vor der Aufgabe, einen String mehrmals aneinanderzureihen, also einen bestehenden String n-mal zu wiederholen. Dazu waren bislang eigene Hilfsmethoden oder solche aus externen Bibliotheken nötig. Mit Java 11 kann man stattdessen die Methode `repeat(int)` nutzen, wie es folgendes Beispiel zeigt:

```
public class RepeatTester {  
    public static void main(String... args) {  
        final String star = "*";  
        System.out.println(star.repeat(30));  
        final String delimiter = " -* - ";  
        System.out.println(delimiter.repeat(6));  
    }  
}
```

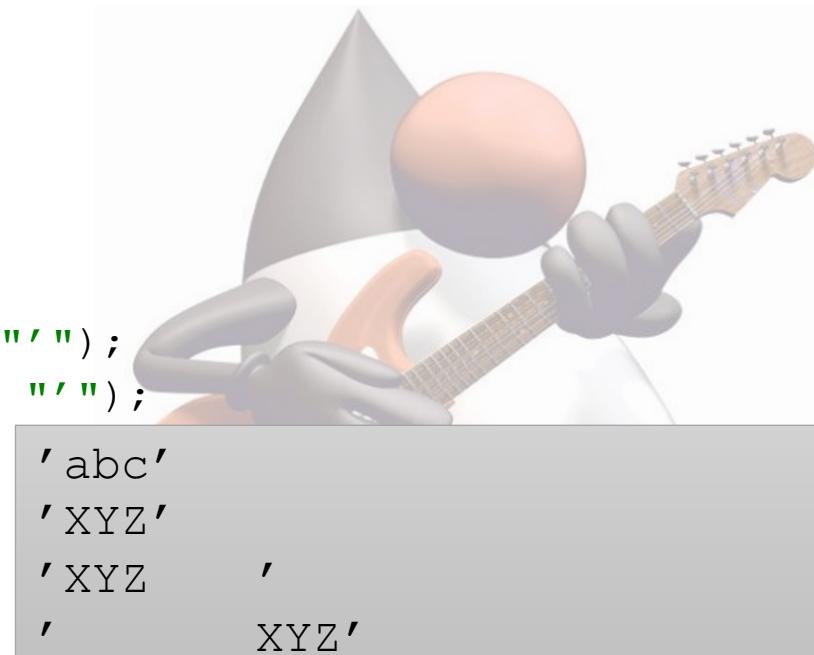
```
*****  
-*_-*-_-*-_-*-_-*
```

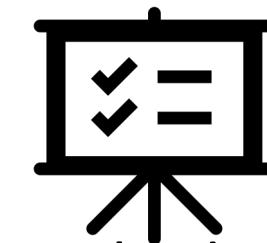


# Neue Hilfsmethoden von String – strip()

Mit der seit JDK 11 neuen Methode `strip()` lassen sich aus einem String führende und nachfolgende Leerzeichen (Whitespaces) entfernen. Gibt es dazu nicht schon die Methode `trim()`? Eigentlich ja, aber `trim()` nutzt eine leicht abweichende Definition von Whitespace. Die Methode `strip()` stützt sich – wie `isBlank()` – auf `Character.isWhitespace(int)` ab. Gleiches gilt auch für die beiden Methoden `stripLeading()` und `stripTrailing()`. Erstere entfernt Whitespace nur am Anfang, letztere nur am Ende eines Strings. Nachfolgendes Beispiel demonstriert diese Methoden:

```
public class StripTester {  
    public static void main(String... args) {  
        final String exampleText1 = " abc ";  
        final String exampleText2 = " \t XYZ \t ";  
        System.out.println("'" + exampleText1.strip() + "'");  
        System.out.println("'" + exampleText2.strip() + "'");  
        System.out.println("'" + exampleText2.stripLeading() + "'");  
        System.out.println("'" + exampleText2.stripTrailing() + "'");  
    }  
}
```





# Übungsbeispiel 12

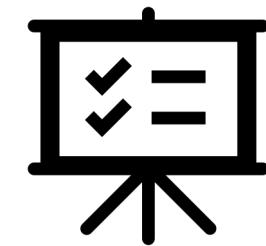
Die Verarbeitung von Strings wurde in Java 11 mit einigen Methoden erleichtert.

**Aufgabe 12a:** Realisieren Sie eine Ausgabe, die fünf Zahlen untereinander ausgibt, jeweils so oft wiederholt, wie die Ziffer, also verkürzt wie folgt:

```
22  
4444  
7777777  
333  
999999999
```

Nutzen Sie folgenden Stream als Eingabe:  
`Stream.of(2, 4, 7, 3, 9)`





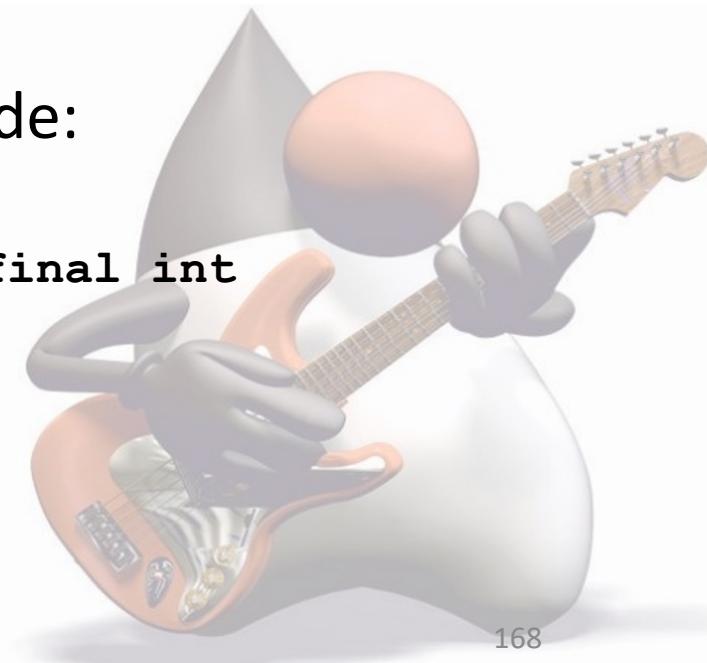
# Übungsbeispiel 12

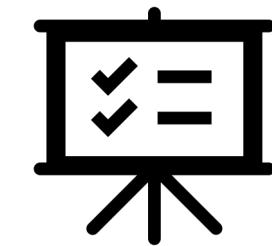
**Aufgabe 12b:** Modifizieren Sie die Ausgabe so, dass die Zahlen rechtsbündig mit maximal 10 Zeichen ausgegeben werden:

```
' 4444'  
' 7777777'  
' 99999999'
```

Implementieren und nutzen Sie dazu folgende Hilfsmethode:

```
private static String formatRightAligned(final int num, final int  
desiredLength) {  
    // TODO  
}
```





# Übungsbeispiel 12

**Aufgabe 12c:** Ändern Sie das Ganze so ab, dass nun statt Leerzeichen führende Nullen ausgegeben werden, etwa wie folgt:

```
'0000004444'  
'0007777777'  
'0999999999'
```



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - **Neue Hilfsmethoden der Klasse File**
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



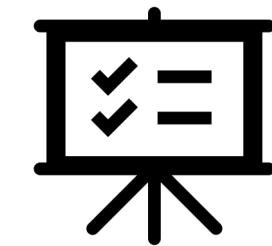
# Neue Methoden der Klasse Files

In Java 11 wurde die Verarbeitung von Strings im Zusammenhang mit Dateien erleichtert. Es ist nun einfach möglich, Strings in eine Datei zu schreiben bzw. daraus zu lesen. Dazu hat die Utility-Klasse `java.nio.file.Files` ein paar Erweiterungen in Form der Methoden `writeString()` und `readString()` erhalten.

```
public static void main(String[] args) throws IOException {
    Path destinationPath = Paths.get("testfile.txt");
    Files.writeString(destinationPath, "1: This is a 'string to file' test\n");
    Files.writeString(destinationPath, "2: Second line", StandardOpenOption.APPEND);
    final String fileContent = Files.readString(destinationPath);
    System.out.println(fileContent);
}
```

1: This is a 'string to file' test  
2: Second line





# Übungsbeispiel 13

Bis Java 11 war es etwas mühsam, Texte direkt in eine Datei zu schreiben bzw. daraus zu lesen. Dazu gibt es nun die Methoden `writeString()` und `readString()` aus der Klasse `Files`. Schreiben Sie mit deren Hilfe folgende Zeile in eine Datei:

- 1: One
- 2: Two
- 3: Three

Lesen Sie diese Zeilen wieder ein und bereiten Sie daraus eine `List<String>` auf.



# Übersicht

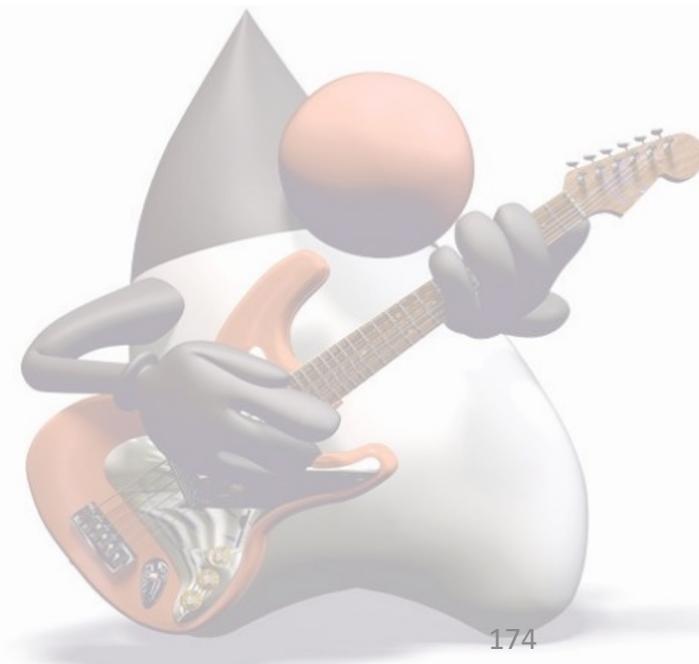
- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen

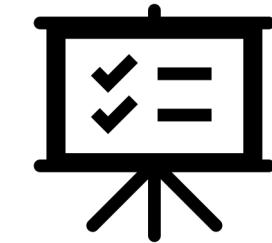


# Erweiterung Optional<T> - isEmpty()

Wie schon im Text zu Java 10 erwähnt, stellt die Klasse `java.util.Optional<T>` eine enorme Bereicherung des JDKs in Java 8 dar. Zur Abrundung wurde das API sowohl in Java 9 als auch in Java 10 noch ein wenig ergänzt. Schließlich bringt Java 11 mit `isEmpty()` noch eine weitere Methode, um für mehr Konsistenz zu anderen APIs wie denjenigen von `Collection` und `String` zu sorgen. Die Methode `isEmpty()` liefert ihrem Namen gemäß `true`, falls kein Wert im `Optional<T>` enthalten ist, und ansonsten `false`.

```
final Optional<String> optEmpty = Optional.empty();
if (!optEmpty.isPresent())
    System.out.println("check for empty JDK 10 style");
if (optEmpty.isEmpty())
    System.out.println("check for empty JDK 11 style");
}
```



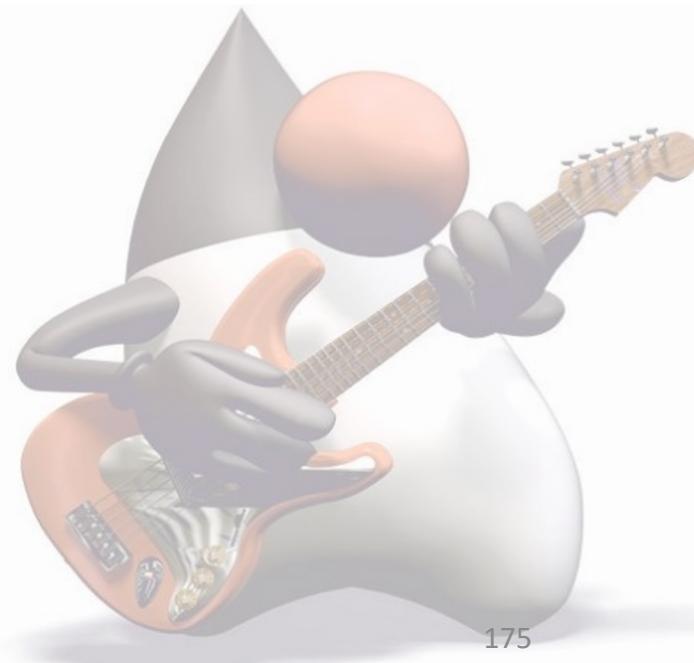


# Übungsbeispiel 14

Vereinfachen Sie folgendes Konstrukt mit der neuen Methode isEmpty() aus der Klasse Optional<T>:

```
static <T> Stream<T> asStream(final Optional<T> opt) {
    if (!opt.isPresent()) {
        return Stream.empty();
    }

    return Stream.of(opt.get());
}
```



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - **Erweiterung Predicate<T>**
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



# Erweiterung Predicate<T> - not()

Mit Java 10 lässt sich eine Negation nur etwas umständlich und schwieriger lesbar, wie im folgenden Listing demonstriert, realisieren:

```
// JDK 10 style

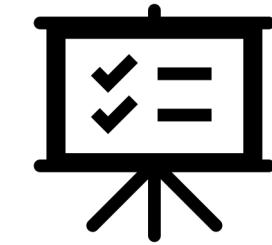
final Predicate<String> isEmpty = String::isEmpty;

final Predicate<String> notEmptyJdk10 = isEmpty.negate();
```

Zur Vereinfachung hätte man sich eine Methode `not()` gewünscht. Java 11 bietet eine solche, die sich wie folgt einsetzen lässt:

```
// JDK 11 style

final Predicate<String> notEmptyJdk11 = Predicate.not(String::isEmpty);
```



# Übungsbeispiel 15

Vereinfachen Sie folgende Prädikate bezüglich der Negation:

```
Predicate<Long> isEven = n -> n % 2 == 0;  
var isOdd = isEven.negate();  
Predicate<String> isBlank = String::isBlank;  
var notIsBlank = isBlank.negate();
```



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - **HTTP/2 API**
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



# HTTP/2 API

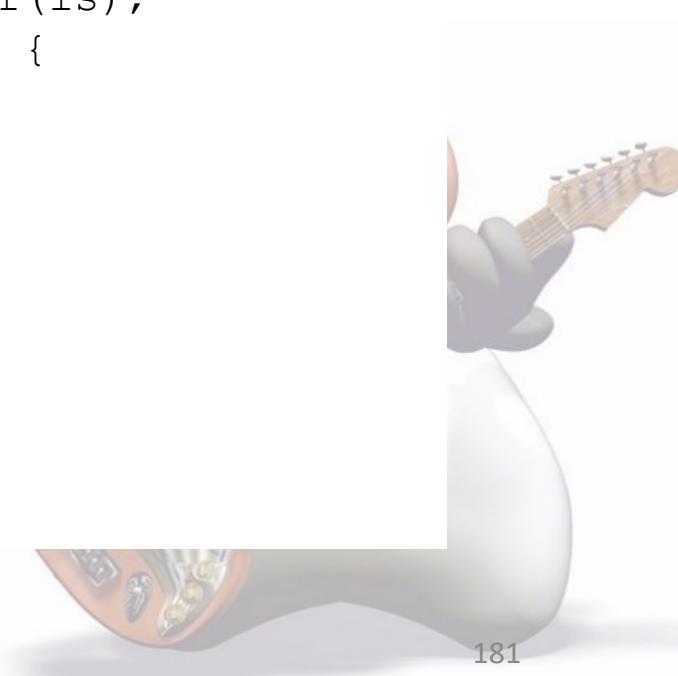
HTTP (Hypertext Transfer Protocol) ist der Standard im Internet zum Transport von Webinhalten, vor allem Webseiten. Bis zum Jahr 2015 galt immer noch der 1997 verabschiedete Standard HTTP 1.1. Mittlerweile nutzen diverse Webseiten aber schon das neue HTTP/2. Dieses basiert auf dem binären, von Google entwickelten SPDY-Protokoll (SPDY steht für speedy). Unter anderem durch diese binäre Datenübertragung statt der textorientierten bei HTTP 1.1 lassen sich deutliche Geschwindigkeitsverbesserungen erzielen.<sup>1</sup> Trotzdem bleibt die Kompatibilität zu HTTP 1.1 gewahrt.



# HTTP/2 API

## http Client Java 10:

```
public class HttpDemo {  
    public static void main(final String[] args) throws Exception {  
        final URL oracleUrl = new URL("https://www.oracle.com/index.html");  
        final String contents = readContent(oracleUrl.openStream());  
        System.out.println(contents);  
    }  
  
    public static String readContent(final InputStream is) throws IOException {  
        try (final InputStreamReader isr = new InputStreamReader(is);  
             final BufferedReader br = new BufferedReader(isr)) {  
            final StringBuilder content = new StringBuilder();  
            String line;  
            while ((line = br.readLine()) != null) {  
                content.append(line + "\n");  
            }  
            return content.toString();  
        }  
    }  
}
```



# HTTP/2 API

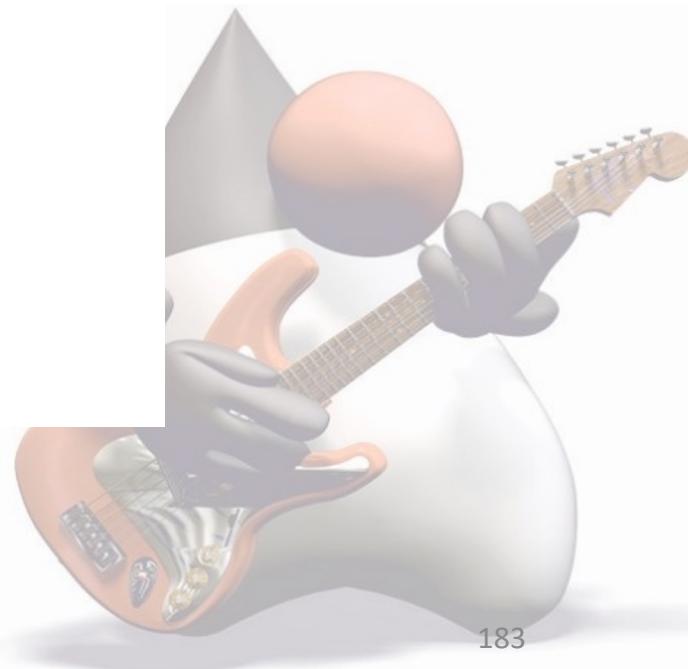
## http Client Java 11:

```
public class HttpDemoNew {  
    public static void main(String...args) throws URISyntaxException,  
        IOException,  
        InterruptedException {  
        final URI uri = new URI("https://www.oracle.com/index.html");  
        final HttpRequest request = HttpRequest.newBuilder(uri).GET().build();  
        final HttpResponse.BodyHandler<String>asString = HttpResponse.BodyHandlers.ofString();  
        final HttpClient httpClient = HttpClient.newHttpClient();  
        final HttpResponse<String> response = httpClient.send(request, asString);  
        printResponseInfo(response);  
    }  
  
    private static void printResponseInfo(final HttpResponse<String> response) {  
        final int responseCode = response.statusCode();  
        final String responseBody = response.body();  
        final HttpHeaders headers = response.headers();  
        System.out.println("Status: " + responseCode);  
        System.out.println("Body: " + responseBody);  
        System.out.println("Headers: " + headers.map());  
    }  
}
```

# HTTP/2 API

## http Client Java 11 – write to file:

```
public class HttpWriteFile {  
    public static void main(final String[] args) throws Exception  
{  
    var uri = new URI("https://www.oracle.com/index.html");  
    var request = HttpRequest.newBuilder(uri).GET().build();  
    var downloadPath = Paths.get("oracle-index.html");  
    var asFile = HttpResponse.BodyHandlers.ofFile(downloadPath);  
    var httpClient = HttpClient.newHttpClient();  
    var response = httpClient.send(request, asFile);  
    if (response.statusCode() == 200)  
    {  
        System.out.println("Content written to file: " +  
            downloadPath.toAbsolutePath());  
    }  
}
```



# HTTP/2 API

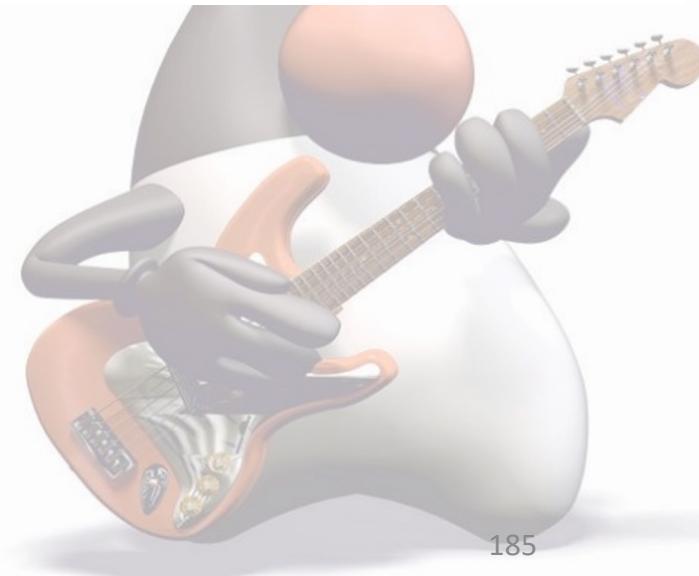
## http Client Java 11 – async:

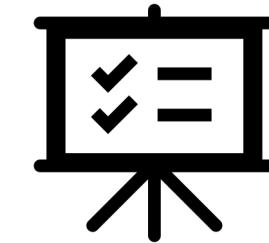
```
public class HttpAsync {  
    public static void main(final String[] args) throws Exception {  
        var uri = new URI("https://www.oracle.com/index.html");  
        var request = HttpRequest.newBuilder(uri).GET().build();  
        var asString = HttpResponse.BodyHandlers.ofString();  
        var httpClient = HttpClient.newHttpClient();  
        final CompletableFuture<HttpResponse<String>> asyncResponse =  
            httpClient.sendAsync(request, asString);  
        // Variante 1: Verarbeitung, sobald die Response eintrifft  
        asyncResponse.thenAccept(response -> printResponseInfo(response, "async"));  
  
        // Variante 2: Verarbeitung mit eigenem Warten, Abfrage auf Erfolg und  
        // bei nicht erfolgreichem Abschluss mit einem Abbruch per cancel(true)  
        final HttpResponse<String> response = asyncResponse.get();  
        printResponseInfo(response, "sync");  
  
        asyncResponse.join(); //warten auf async Verarbeitung  
    }  
}
```

# HTTP/2 API

## http Client Java 11 – async:

```
private static void printResponseInfo(final HttpResponse<String> response, String client) {  
    var responseCode = response.statusCode();  
    String body = response.body();  
    body = body.substring(0, Math.min(10, body.length())) + "...";  
    System.out.println(client + " Status: " + responseCode);  
    System.out.println(client + " Body: " + body);  
}  
}
```



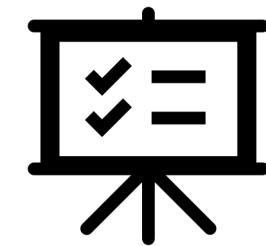


# Übungsbeispiel 16

Gegeben sei untenstehende HTTP-Kommunikation, die auf die Webseite von Oracle zugreift und diese textuell aufbereitet. Wandeln Sie den Sourcecode so um, dass das neue HTTP/2-API aus JDK 11 zum Einsatz kommt.

**Bonus:** Starten Sie die Abfragen asynchron.





# Übungsbeispiel 16

```
public class HttpCommunicationJdk8 {  
    private static void readOraclePageJdk8() throws MalformedURLException,  
        IOException {  
        final URL url = new URL("https://www.oracle.com/index.html");  
        final String contents = readContent(url.openStream());  
        System.out.println(contents);  
    }  
  
    public static String readContent(final InputStream is) throws IOException {  
        try (final InputStreamReader isr = new InputStreamReader(is);  
             final BufferedReader br = new BufferedReader(isr)) {  
            final StringBuilder content = new StringBuilder();  
            String line;  
            while ((line = br.readLine()) != null) {  
                content.append(line + "\n");  
            }  
            return content.toString();  
        }  
    }  
}
```



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - **Epsilon Garbage Collector**
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



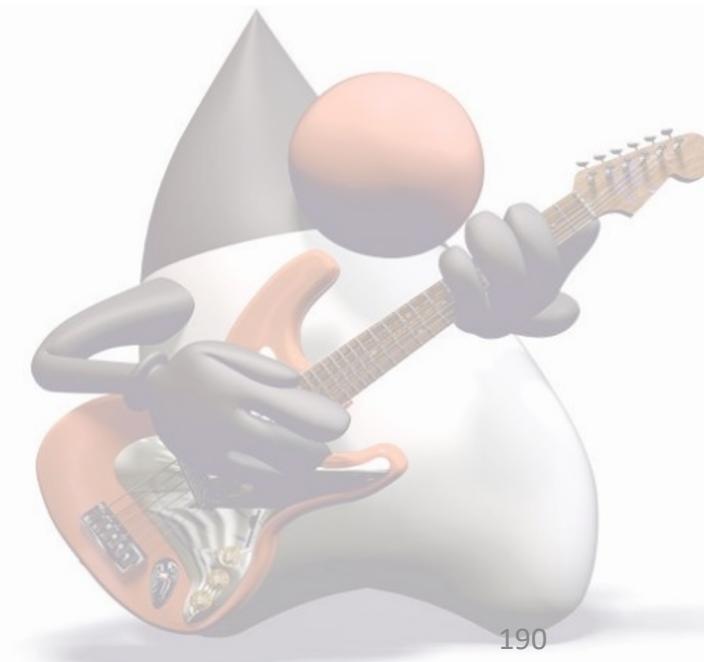
# JVM Änderungen – Epsilon GC

Der Epsilon genannte Garbage Collector aus Java 11 ist etwas speziell: Es werden gar keine Aufräumarbeiten ausgeführt, sondern bei Bedarf einfach immer nur Speicher alloziert – bis irgendwann der Heap aufgebraucht ist. Dann wird die JVM terminiert. Obwohl das Ganze eigentlich ein wenig widersinnig klingt, kann es doch (wenn auch vermutlich seltene) Anwendungsfälle geben, in denen die Garbage Collection eher störend wäre. Das gilt etwa für kleinere Programme, die man mit Java schreibt und die nur einmalig ausgeführt werden, etwa eine Skriptverarbeitung.



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - **Single-File Source-Code Programme**
  - Flightrecorder
  - Deprecations und Entfernungen

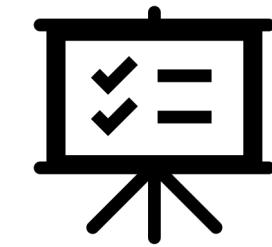


# CIIT JVM Änderungen – Single-File Source-Code

Mit Java 11 wurde ein interessantes Feature eingeführt. Man kann nun Java-Applikationen, die nur aus einer Datei bestehen, direkt in einem Rutsch kompilieren und ausführen. Damit erspart man sich Arbeit und muss auch nichts vom Bytecode und .class-Dateien wissen.

```
java ./HelloWorld.java
```





# Übungsbeispiel 17

Schreiben Sie eine Klasse HelloWorld im Package direct.compilation und speichern Sie diese in einer gleichnamigen Java-Datei. Führen Sie diese direkt mit dem Kommando java aus.



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



# JVM Änderungen – Flightrecorder

Mit Java 11 wurde das Tooling in der JVM um den sogenannten Flight Recorder erweitert.<sup>3</sup> Dies ist ein Tool, mit dem sich Events von Applikationen aufzeichnen lassen. Diese Informationen kann man beispielsweise bei Programmabstürzen oder anderen Problemen als ergänzende Beschreibung bereitstellen. Insbesondere in Kombination mit dem Tool Mission Control wird es möglich, diverse Daten während der Ausführung einer JVM zu sammeln und auszuwerten.



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - **Deprecations und Entfernungen**



# Deprecations & Entfernungen

- **Aufräumarbeiten in der Klasse Thread**

Die Klasse `java.lang.Thread` wurde von Fehlentwicklungen bereinigt: Die Methoden `destroy()` und `stop(Throwable obj)` waren schon seit längerer Zeit deprecated. Um auf die fehlende Funktionalität hinzuweisen, lösten beide Methoden Exceptions aus. Mit Java 11 wurden die Methoden konsequenterweise aus dem JDK entfernt.

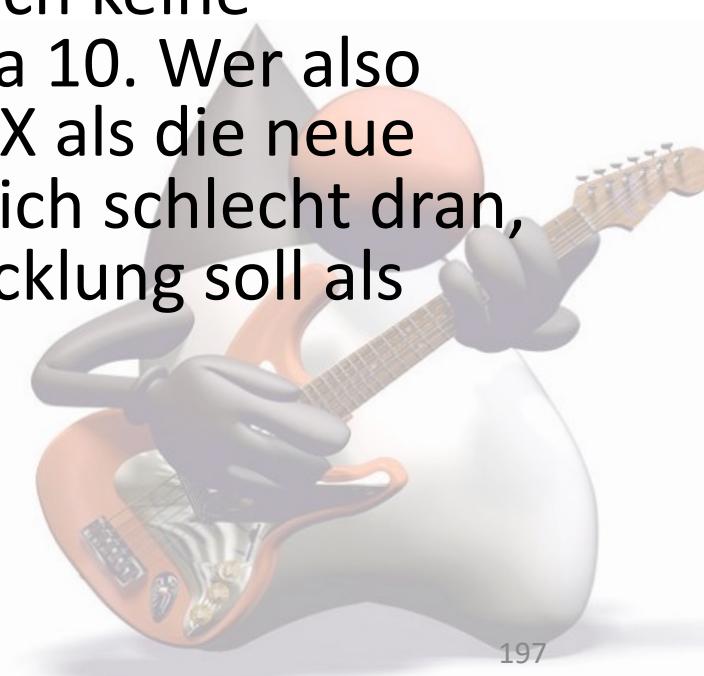
- **Deprecation der JavaScript-Unterstützung**

In Java 8 hat Oracle die in das JDK integrierte JavaScript-Engine erneuert. Leider wurden diese sowie zugehörige APIs in Java 11 als deprecated gekennzeichnet. Eine Entfernung aus dem JDK ist in einem der kommenden Releases wahrscheinlich.

# Deprecations & Entfernungen

- **Ausgliederung von JavaFX**

Während Oracle JavaFX mit Java 8 noch als die neue, zukunftsträchtige Desktop-Technologie propagiert hat und in Java 8 Update 40 sogar diverse Ergänzungen eingefügt wurden, wurde es zunehmend stiller um JavaFX, und in Java 9 finden sich keine signifikanten Neuerungen mehr. Gleiches gilt für Java 10. Wer also damals auf den Zug aufgesprungen ist und auf JavaFX als die neue GUI-Technologie umgestellt hat, ist mit JDK 11 ziemlich schlecht dran, da JavaFX hieraus entfernt wurde. Eine Weiterentwicklung soll als OpenJFX (<https://openjfx.io/>) stattfinden.



# Deprecations & Entfernungen

- **Ausgliederung von Java EE und CORBA**

Mit Java 11 wurden verschiedene Module aus dem JDK entfernt. Dies betrifft die Bereiche Java EE und CORBA. Konkret wurden folgende Module entfernt, die eher zu Java Enterprise gehören:

`java.activation`

`java.corba`

`java.transaction`

`java.xml.bind`

`java.xml.ws`

`java.xml.ws.annotation`



# Übersicht

- Java 9: Syntaxänderungen
  - Diamond Operator bei anonymen Klassen
  - Erweiterte @Deprecated Annotation
  - Private Methoden in Interfaces
  - Bezeichner „\_“ ist ein Schlüsselwort
- Java 9: API Änderungen
  - ProcessHandle Interface
  - Collection Factory Methoden
  - Reactive Streams
  - Erweiterungen InputStream
  - Erweiterungen Optional<T>
  - Erweiterungen java.util.Stream



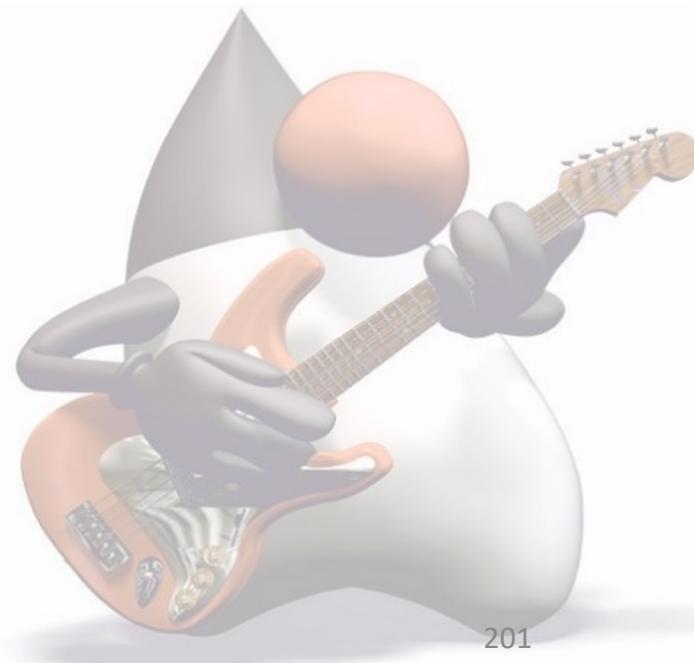
# Übersicht

- Java 9: API Änderungen (Fortsetzung)
  - Erweiterungen LocalDate
  - Erweiterungen Arrays
  - Erweiterungen Objects
  - Erweiterungen CompletableFuture
  - Optimierungen bei Strings
  - @Deprecated in Java 9
- Java 9: JVM Änderungen
  - Änderungen Versionsschema
  - Multi-Release-Jars
  - Jshell
  - HTML 5 JavaDoc



# Übersicht

- Java 10: Syntaxänderungen
  - Syntaxerweiterung var
- Java 10: API Änderungen
  - Unmodifiable Lists
  - Erweiterungen Optional<T>
  - Erneute Änderungen Versionierung
  - Verschiedenes
- Java 11: Syntaxänderungen
  - var Type Interference für Lambdas



# Übersicht

- Java 11: API Änderungen
  - Neue Hilfsmethoden der Klasse String
  - Neue Hilfsmethoden der Klasse File
  - Erneute Erweiterung Optional<T>
  - Erweiterung Predicate<T>
  - HTTP/2 API
- Java 11: JVM Änderungen
  - Epsilon Garbage Collector
  - Single-File Source-Code Programme
  - Flightrecorder
  - Deprecations und Entfernungen



# Übersicht

- Modularisierung (Java 9)

