Java 8 New Features

michael.schaffler@java.at

Agenda

- Lambda Expressions
  - Einstieg in Lambdas
  - Default Methoden in Interfaces
  - Methodenreferenzen

Agenda

- Collections & Lambdas
  - externe vs. Interne Iteration
  - Collections Erweiterungen
  - Streams
  - Filter-Map-Reduce
  - Fallstricke bei Lambdas

Agenda

- Java Date & Time API
  - Datumsverarbeitung vor JSR-310
  - Überblick über die neu eingeführten Klassen

Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

Agenda

- Weitere Änderungen in JDK 8
  - Keine Permanent Generation mehr
  - Erweiterungen im Bereich Reflection
  - Base64-Codierungen
  - Änderungen bei Annotations

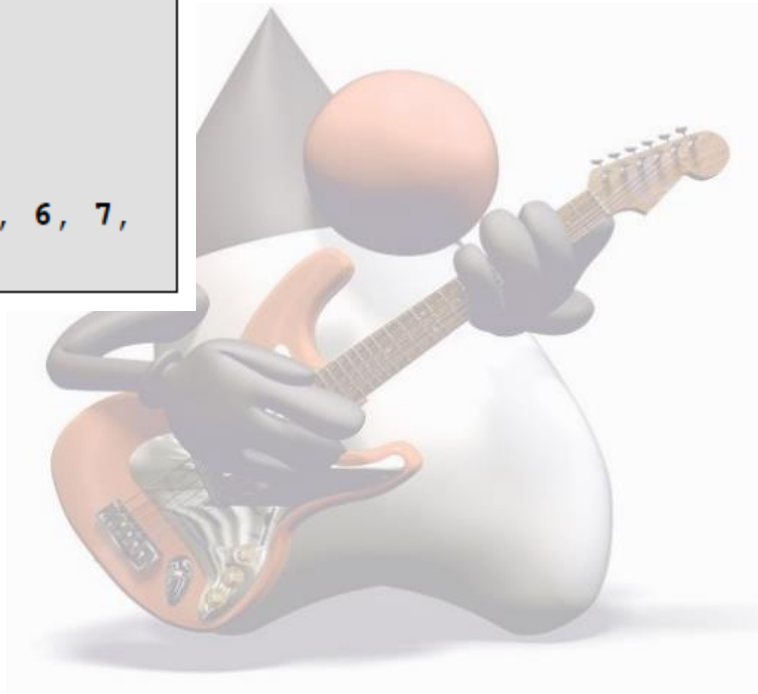
Agenda

- Lambda Expressions
  - Einstieg in Lambdas
  - Default Methoden in Interfaces
  - Methodenreferenzen

Beispiel Sortierung

```
// Sortierung mit Comparator
final List<String> names = Arrays.asList( "Andy", "Michael", "Max"
, "Stefan");
Collections.sort( names, new Comparator<String>( )
{
    @Override
    public int compare( final String str1, final String str2)
    {
        return Integer.compare( str1.length( ), str2.length( ));
    }
});

// Iteration und Ausgabe
final Iterator<String> it = names.iterator( );
while ( it.hasNext( ))
{
    System.out.print( it.next( ).length( ) + ", "); //  3, 4, 6, 7,
}
```

Geht das kürzer?

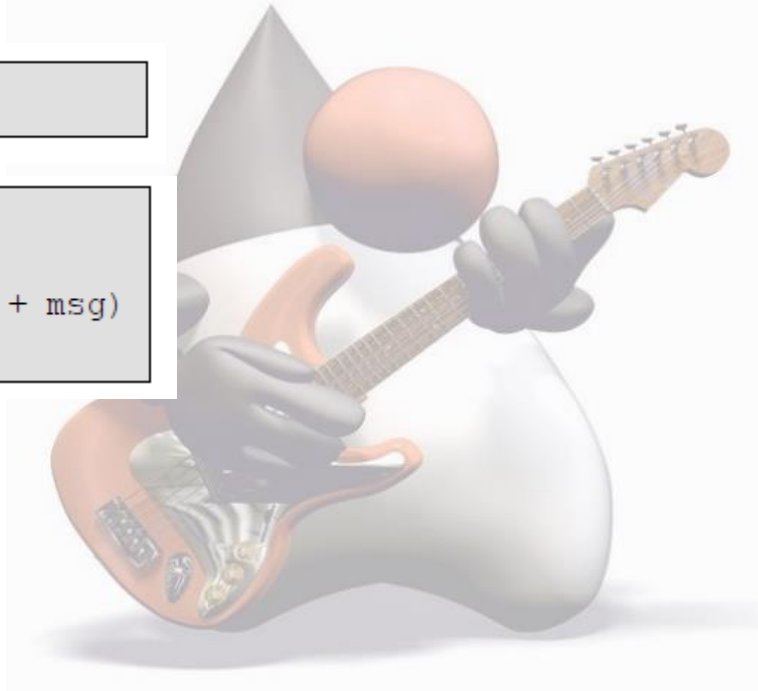


Einstieg in Lambdas

Ein Lambda ist ein Behälter für Sourcecode ähnlich einer Methode, allerdings ohne Namen und ohne explizite Angabe eines Rückgabetyps oder ausgelöster Exception.

```
( Parameter-Liste) -> { Ausdruck oder Anweisungen }
```

```
( int x, int y) -> { return x + y; }
( long x) -> { return x * 2; }
( ) -> { String msg = "Lambda"; System.out.println( "Hello " + msg)
; }
```

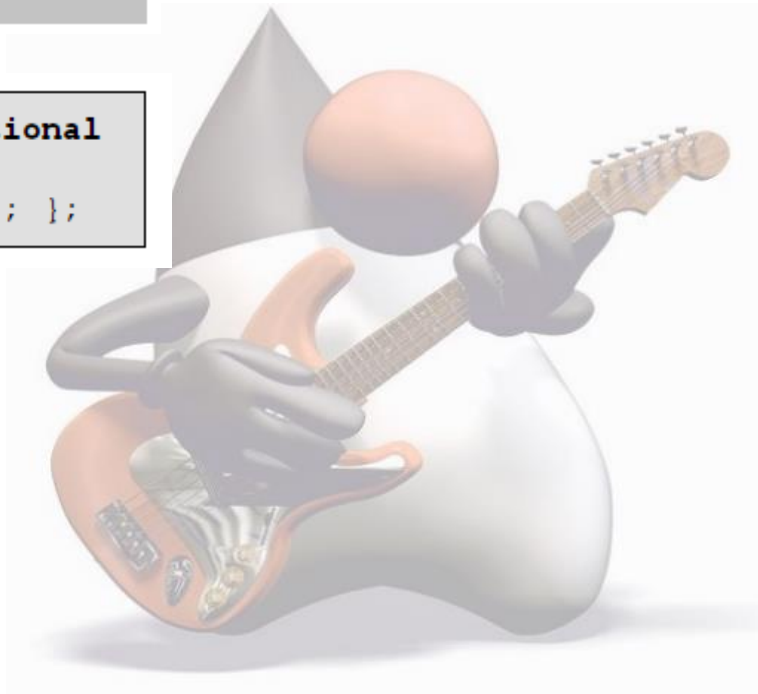


Einstieg in Lambdas

Besonderheit: Lambdas im Java-Typsystem

Bis JDK 8 konnte in Java jede Referenz auf den Basistyp `Object` abgebildet werden. Mit Lambdas existiert nun ein Sprachelement, das nicht direkt dem Basistyp `Object` zugewiesen werden kann, sondern nur an Functional Interfaces.

```
// Compile-Error: incompatible types: Object is not a functional
interface
Object greeter = () -> { System.out.println("Hello Lambda"); };
```

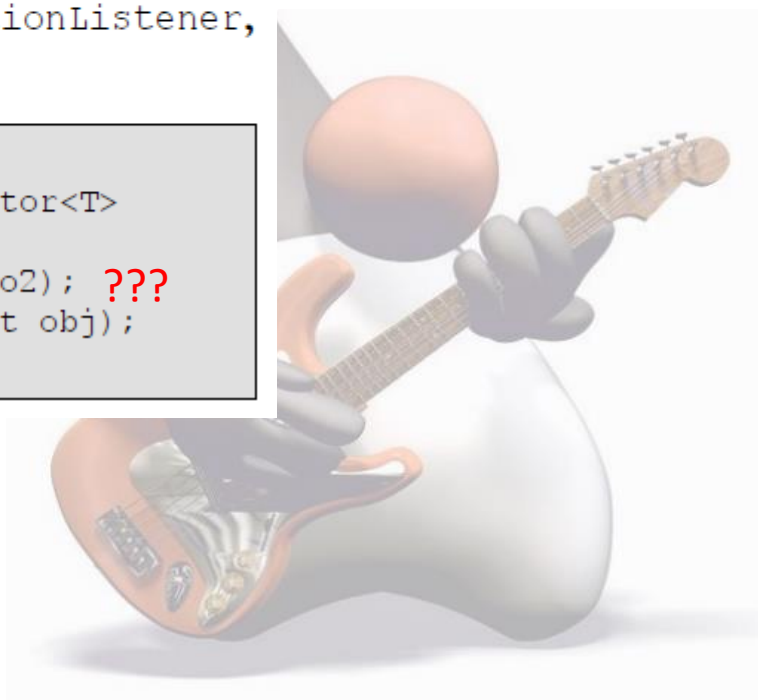


Functional Interfaces / SAM Typen

Ein ***Functional Interface*** ist eine neue Art von Typ, die mit JDK 8 eingeführt wurde, und repräsentiert ein Interface mit genau einer abstrakten Methode. Ein solches wird auch ***SAM-Typ*** genannt, wobei SAM für Single Abstract Method steht. Diese Art von Interfaces gibt es nicht erst seit Java 8 im JDK, sondern schon seit Langem und vielfach – wobei es früher für sie aber keine Bezeichnung gab. Bekannte Vertreter der SAM-Typen und Functional Interfaces sind etwa `Runnable`, `Callable`, `Comparator`, `FileFilter`, `FilenameFilter`, `ActionListener`, `EventHandler` usw.

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run( );
}
```

```
@FunctionalInterface
public interface Comparator<T>
{
    int compare( T o1, T o2);  ???
    boolean equals( Object obj);
}
```



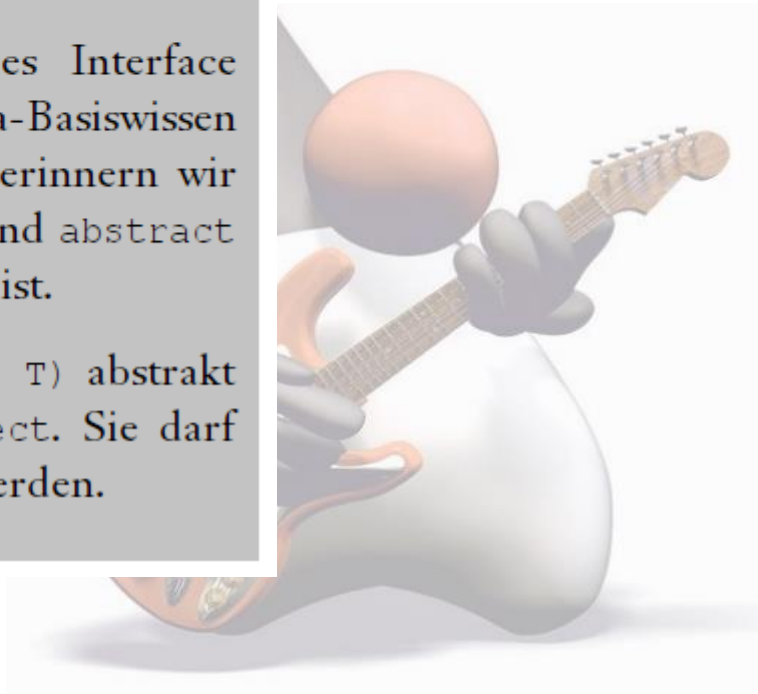
Functional Interfaces SAM Typen

Tipp: Besondere Methoden in Functional Interfaces

Wenn wir im obigen Listing genauer hinsehen, könnten wir uns fragen, wieso denn `java.util.Comparator<T>` ein Functional Interface ist, wo es doch zwei Methoden enthält und keine davon abstrakt ist, oder? Als Besonderheit gilt in Functional Interfaces folgende Ausnahme für die Definition von abstrakten Methoden: Alle im Typ `Object` definierten Methoden können zusätzlich zu der abstrakten Methode in einem Functional Interface angegeben werden.

Verbleibt noch die Frage, warum wir in der Definition des Interface `Comparator<T>` keine abstrakte Methode sehen. Mit ein wenig Java-Basiswissen oder nach einem Blick in die Java Language Specification (JLS) erinnern wir uns daran, dass alle Methoden in Interfaces automatisch `public` und `abstract` sind, auch wenn dies nicht explizit über Schlüsselwörter angegeben ist.

Basierend auf den Argumentationen ist die Methode `compare(T, T)` abstrakt und die Methode `equals(Object)` entstammt dem Basistyp `Object`. Sie darf damit zusätzlich im Interface zur abstrakten Methode aufgeführt werden.



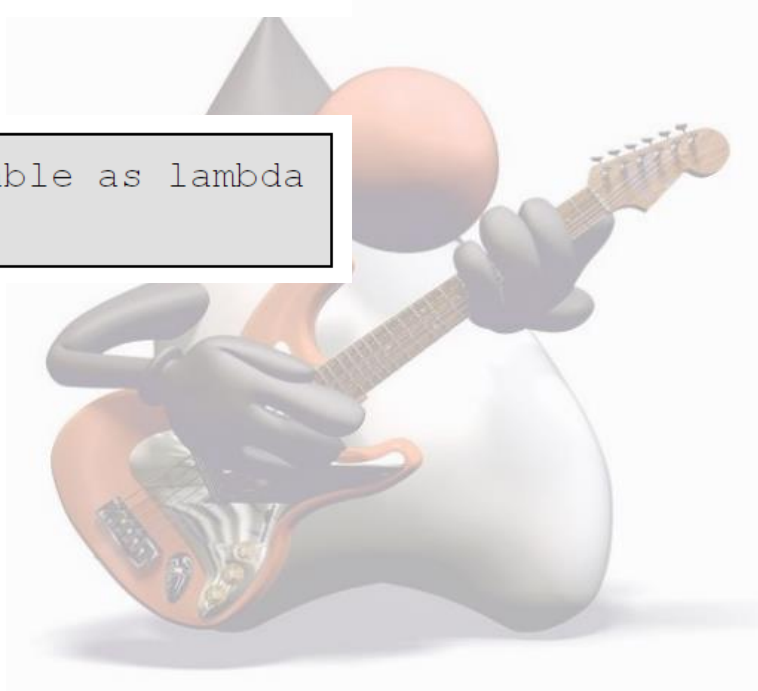
Functional Interfaces / SAM Typen

```
// SAM-Typ als anonyme innere Klasse
new SAMTypeAnonymousClass( )
{
    public void samTypeMethod( METHOD-PARAMETERS )
    {
        METHOD-BODY
    }
}

// SAM-Typ als Lambda
( METHOD-PARAMETERS ) -> { METHOD-BODY }
```

Functional Interfaces / SAM Typen

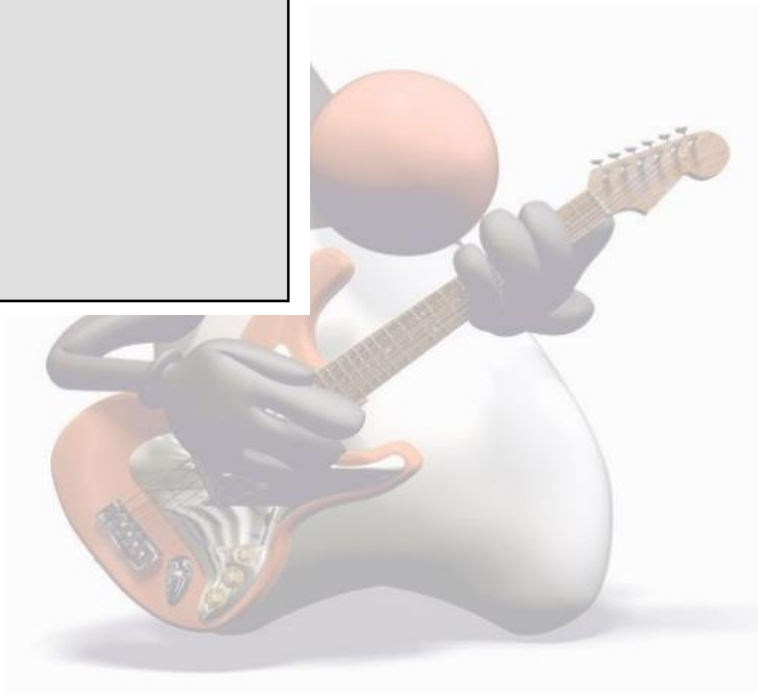
```
Runnable runnableAsNormalMethod = new Runnable()
{
    @Override
    public void run()
    {
        System.out.println("runnable as normal method");
    }
}
```

```
Runnable runnableAsLambda = () -> System.out.println("runnable as lambda
");
```



Functional Interfaces / SAM Typen

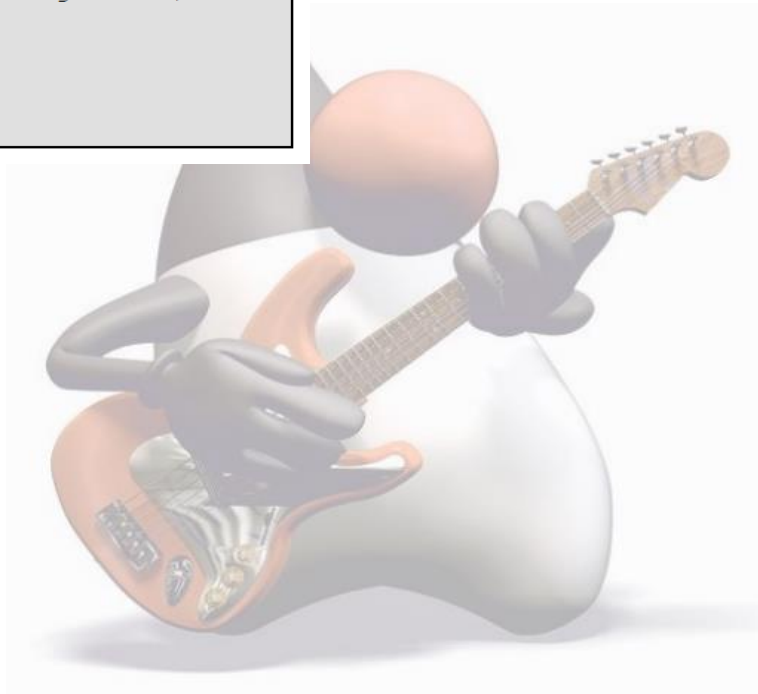
```
// Hinweis: Diamond Operator ist nicht für anonyme innere Klassen möglich
Comparator<String> compareByLength = new Comparator<String>()
{
    @Override
    public int compare( final String str1, final String str2)
    {
        final int length1 = str1.length();
        final int length2 = str2.length();

        if ( length1 < length2)
            return 1;
        if ( length1 > length2)
            return -1;

        return 0;
    }
};
```



Functional Interfaces / SAM Typen

```
Comparator<String> compareByLength = new Comparator<String>()
{
    @Override
    public int compare( final String str1, final String str2)
    {
        return Integer.compare( str1.length(), str2.length());
    }
};
```

```
Comparator<String> compareByLength = ( final String str1, final String str2) ->
{
    return Integer.compare( str1.length(), str2.length());
};
```



Functional Interfaces / SAM Typen

Type Inference: Ähnlich wie beim Diamond Operator bei der Definition generischer Klassen ist es für Lambdas möglich, auf die Typangaben für die Parameter im Sourcecode zu verzichten. Dazu ermittelt der Compiler die passenden Typen aus dem Einsatzkontext. Den vorherigen Komparator schreibt man ohne Typangabe wie folgt:

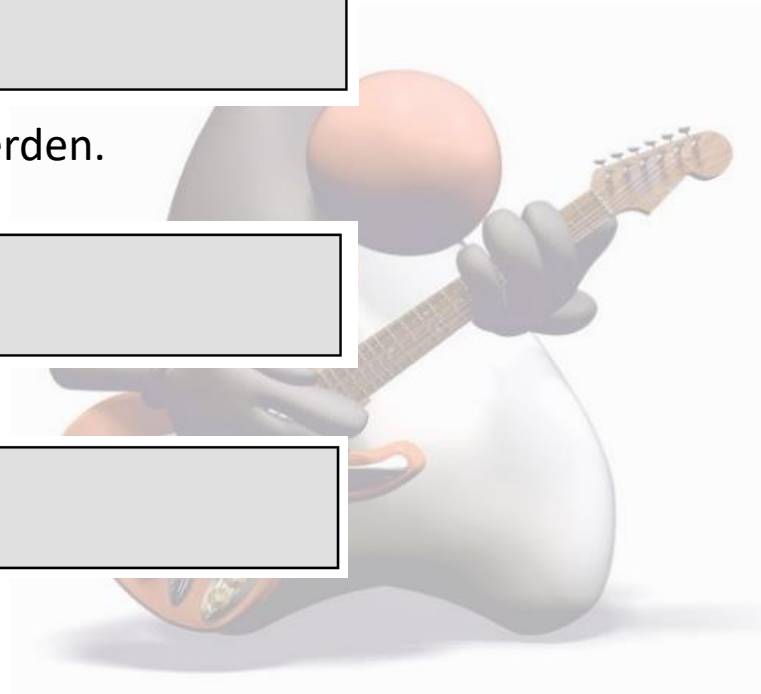
```
Comparator<String> compareByLength = ( str1, str2) ->
{
    return Integer. compare( str1. length( ), str2. length( ));
};
```

Wenn nur ein Ausdruck existiert, kann auch das return weggelassen werden.
Damit kann

```
( int x, int y) -> { return x + y; }
( long x) -> { return x * 2; }
```

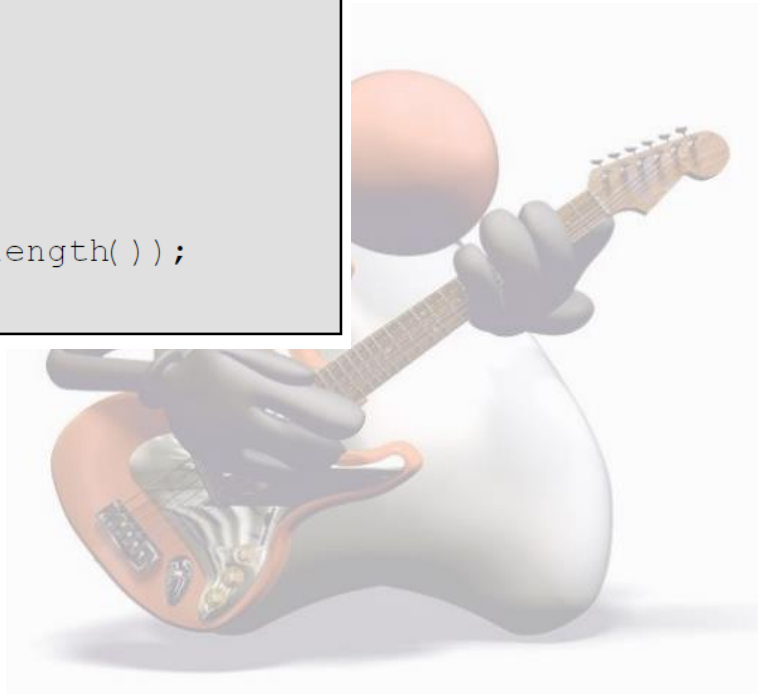
Verkürzt werden zu

```
( x, y) -> x + y
x -> x * 2
```



Functional Interfaces / SAM Typen

```
public static void main( final String[ ] args)
{
    final List<String> names = Arrays. asList( "Andy", "Michael", "Max", "Stefan"
);

    // Lambda als Methodenparameter
    Collections. sort( names, ( str1, str2) -> Integer. compare( str1. length( ),
                                                                  str2. length( )));

   // Alternative mit Lambda als Rückgabe einer Methode
   names. sort( compareByLength( ));
}

public static Comparator<String> compareByLength( )
{
     return ( str1, str2) -> Integer. compare( str1. length( ), str2. length( ));
}
```



Functional Interfaces / SAM Typen

this in Lambdas

- Lambdas haben keine Beziehung zu einem Objekt
- this referenziert das außenliegende Objekt in dem die Expression deklariert wird.
- **Beachte:** das ist unterschiedlich zu inneren (anonymen) Klassen, bei denen this auf das innere Objekt referenziert.

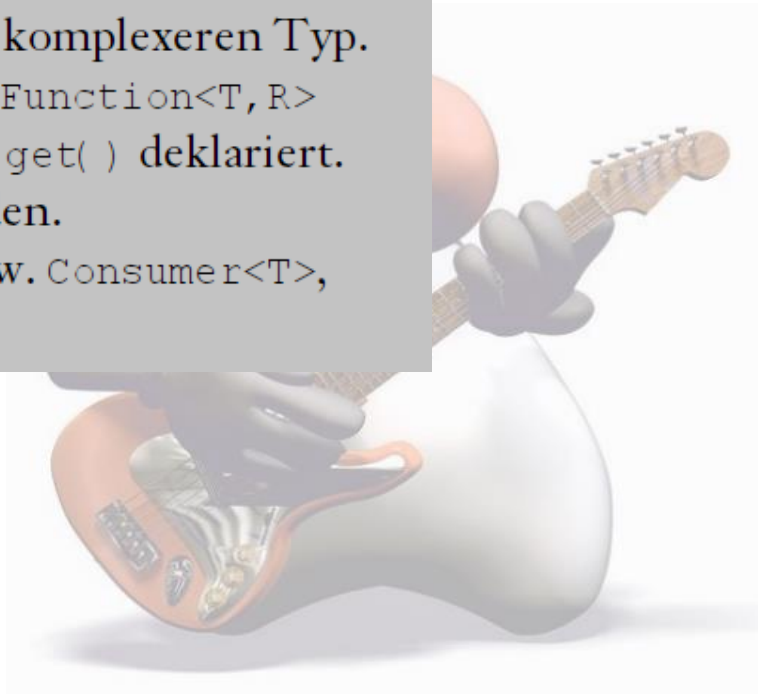
Functional Interfaces / SAM Typen

Zugriff auf Variablen:

- Lambdas können auf die variablen des äußeren Objekts zugreifen.
- Lokale Variablen müssen „effectively“ final sein, um in der Lambda Expression verwendet werden zu können (checked by compiler).

Vordefinierte Functional Interfaces

- `Consumer<T>` – Beschreibt eine Aktion auf einem Element vom Typ `T`. Dazu ist eine Methode `void accept( T)` definiert.
- `Predicate<T>` – Definiert eine Methode `boolean test( T)`. Diese berechnet für eine Eingabe vom Typ `T` einen booleschen Rückgabewert (z. B. `olderThan( )`). Damit lassen sich sehr gut Filterbedingungen ausdrücken.
- `Function<T,R>` – Definiert eine Abbildungsfunktion in Form der Methode `R apply( T)`. Damit wird ein allgemeines Konzept von Transformationen beschrieben. Recht gebräuchlich ist beispielsweise die Extraktion eines Attributs aus einem komplexeren Typ.
- `Supplier<T>` – Stellt ein Ergebnis vom Typ `T` bereit. Im Gegensatz zu `Function<T,R>` erhält ein `Supplier<T>` keine Eingabe. Im Interface ist die Methode `T get( )` deklariert. Damit lassen sich Objekterzeugungen auf verschiedene Weise nachbilden.
- `BiFunction<T,U,R>` und `BiConsumer<T,U>` – Wie `Function<T,R>` bzw. `Consumer<T>`, jedoch mit jeweils zwei Eingabewerten vom Typ `T` und `U`.



Übung 1 – Command Pattern

Sie haben folgende Klasse:

```
public class CalculatorEngine {
    public static Integer execute (Function<CalculatorEngine,Integer> function){
        return function.apply(new CalculatorEngine());
    }

    public int plus(int a, int b){
        return a+b;
    }

    public int minus(int a, int b){
        return a - b;
    }

    public int multipliedBy(int a, int b){
        return a * b;
    }

    public int devidedBy(int a,int b){
        return a/b;
    }
}
```

Schreiben Sie einen Invoker, der die execute Methode mit Lambda und mit inneren anonymen Klassen nutzt um folgendes zu berechnen:

1+1

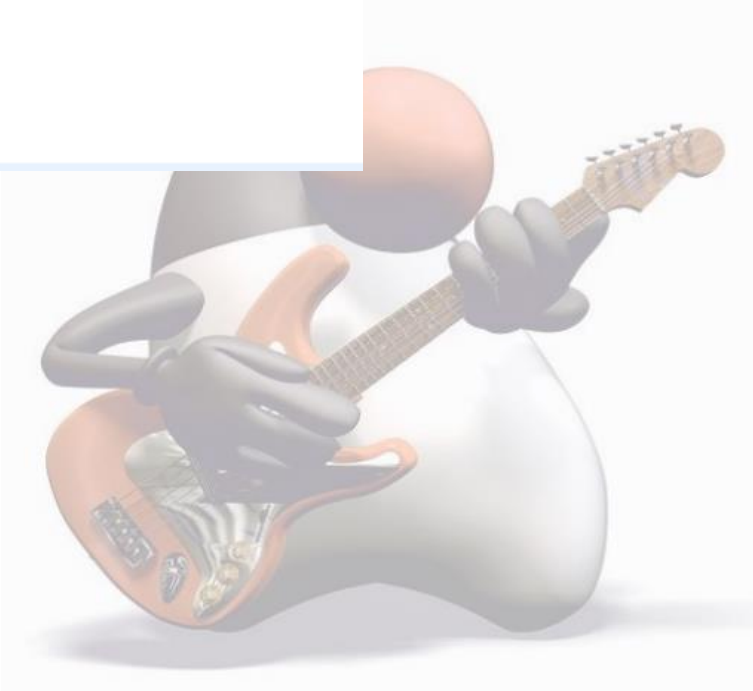
(10+10)/20

3*8

Übung 2

- Schreiben Sie eine Klasse Programm sodass folgender Code ausführbar ist:

```
public class Invoker {

    public static void main(String[] args) {
        Programm programm = new Programm();
        programm.add(()->System.out.println("Hello"))
                .add(()->{
                    System.out.println("This example does not really make sense.");
                    System.out.println("But it works!");
                })
                .execute();
    }
}
```



Agenda

- Lambda Expressions
  - Einstieg in Lambdas
  - Default Methoden in Interfaces
  - Methodenreferenzen

Default Methoden in Interfaces

- Bis JDK 7 sind Java Interfaces sind abstrakte Klassen, die nur abstrakte Methoden haben dürfen.
- Ab JDK 8 dürfen Interfaces auch konkrete „default“ Methoden haben.
- Der Implementierer eines Interfaces erbt diese Implementierung, kann sie aber auch überschreiben.

Default Methoden in Interfaces

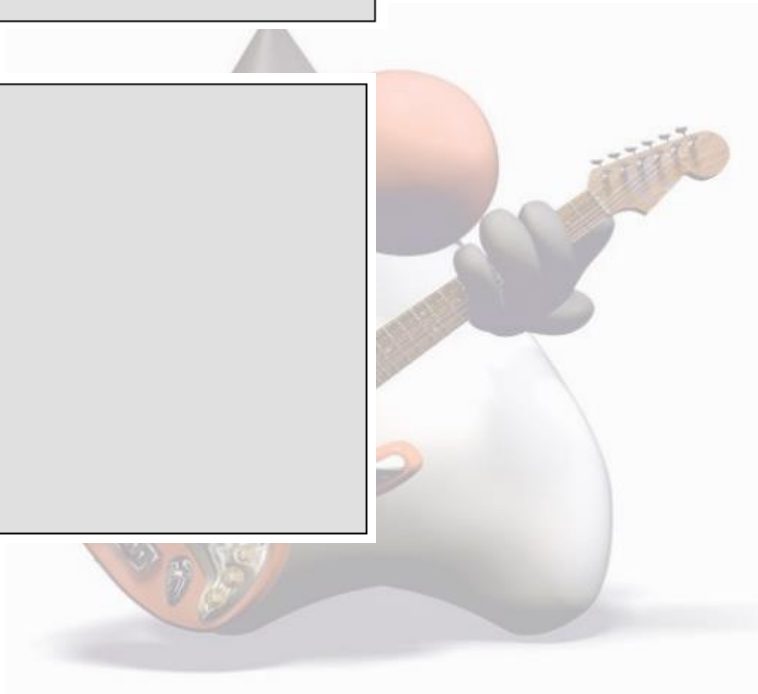
```
@FunctionalInterface
public interface Consumer<T>
{
    void accept( T t);

    default Consumer<T> andThen( Consumer<? super T> after)
    {
        Objects. requireNonNull( after);
        return ( T t) -> { accept( t); after. accept( t); };
    }
}
```

Default Methoden in Interfaces

Beispiele aus dem JDK

```
public interface List<E> extends Collection<E>
{
    default void sort( Comparator<? super E> c)
    {
        Collections. sort( this, c);
    }
}
```

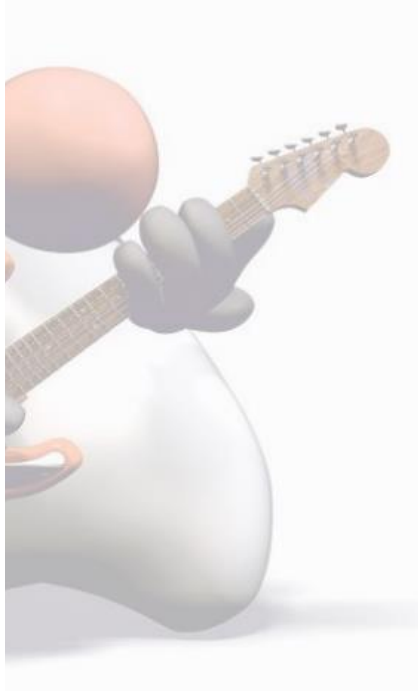
```
public interface Iterable<T>
{
    default void forEach( Consumer<? super T> action)
    {
        Objects. requireNonNull( action);
        for ( T t : this)
        {
            action. accept( t);
        }
    }
}
```



Statische Methoden in Interfaces

mit JDK 8 können Interfaces auch statische (konkrete) Methoden besitzen

```
@FunctionalInterface
public interface Comparator<T>
{
    // ...

    public static <T extends Comparable<? super T>> Comparator<T> reverseOrder( )
    {
        return Collections. reverseOrder( );
    }

    public static <T extends Comparable<? super T>> Comparator<T> naturalOrder( )
    {
        return ( Comparator<T>) Comparators. NaturalOrderComparator. INSTANCE;
    }

    public static <T> Comparator<T> nullsFirst( Comparator<? super T> comparator)
    {
        return new Comparators. NullComparator<>( true, comparator);
    }

    public static <T> Comparator<T> nullsLast( Comparator<? super T> comparator)
    {
        return new Comparators. NullComparator<>( false, comparator);
    }

    // ...
}
```



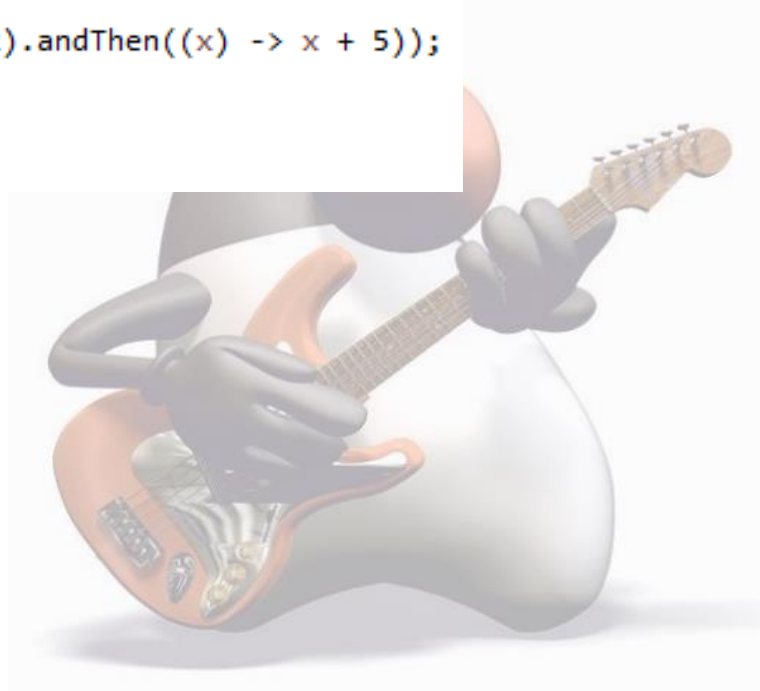
Übung 3

- Analysieren Sie die Methode Function.andThen in der Java 8 API Dokumentation
- Was ist Function, was ist andThen?

Übung 3

- Schreiben Sie eine Klasse Calculation, sodass der folgende Code exekutiert:

```
public class Invoker {
    public static void main(String[] args) {
        System.out.println(Calculation.start(3.).evaluate((x)->x*x)); // 9.0
        Function<Double, Double> function = (x) -> x + 1;
        Double result = Calculation.start(0.).evaluate(function.andThen((x) -> x + 2).andThen((x) -> x + 5));
        System.out.println(result); // 8.0
    }
}
```



Übung 4

- Sie haben folgende Klasse:

```
public class ProgrammableCalculator {
    Function<Double[],Double> function;

    private ProgrammableCalculator(Function<Double[],Double> function){
        this.function = function;
    }

    public static ProgrammableCalculator program(Function<Double[],Double> function){
        ProgrammableCalculator result = new ProgrammableCalculator(function);
        return result;
    }

    public Double calculate(Double ...numbers){
        return function.apply(numbers);
    }

}
```

Berechnen Sie mit Hilfe des ProgrammableCalculators Folgendes:

- Summe zweier Zahlen
- Durchschnitt einer beliebigen Zahlenfolge

Agenda

- Lambda Expressions
  - Einstieg in Lambdas
  - Default Methoden in Interfaces
  - Methodenreferenzen

Methodenreferenzen

Methodenreferenzen besitzen die Syntax:

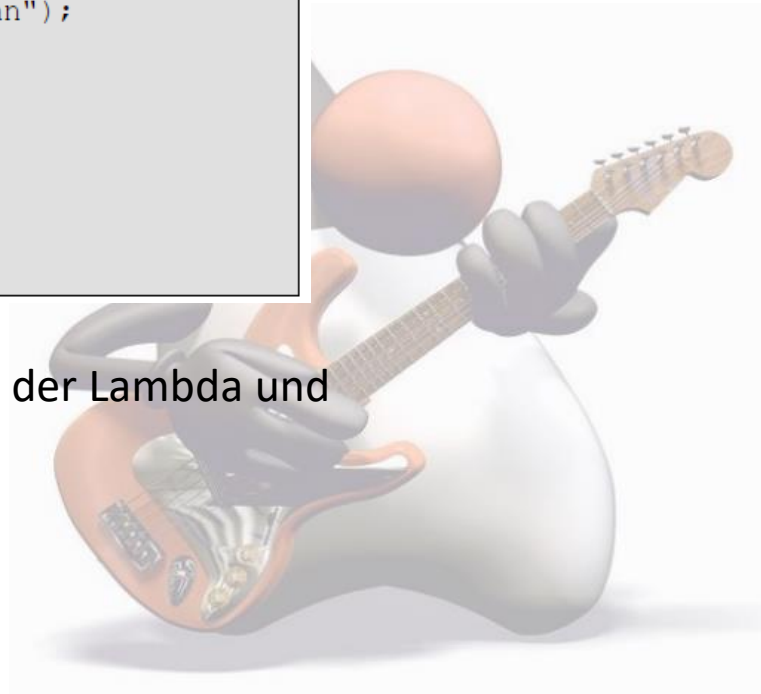
`Klasse::Methodenname` und verweist auf ...

- eine Methode – `System.out::println`, `Person::getName`, ...
- einen Konstruktor – `ArrayList::new`, `Person[]::new`, ...

Beispiel:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");

   // Lambda
   names.forEach( it -> System.out.println(it) );

   // Methodenreferenz
   names.forEach( System.out::println );
}
```

Die Übergabe der Parameter erfolgt automatisch über die Reihenfolge der Lambda und Methoden Paramter.

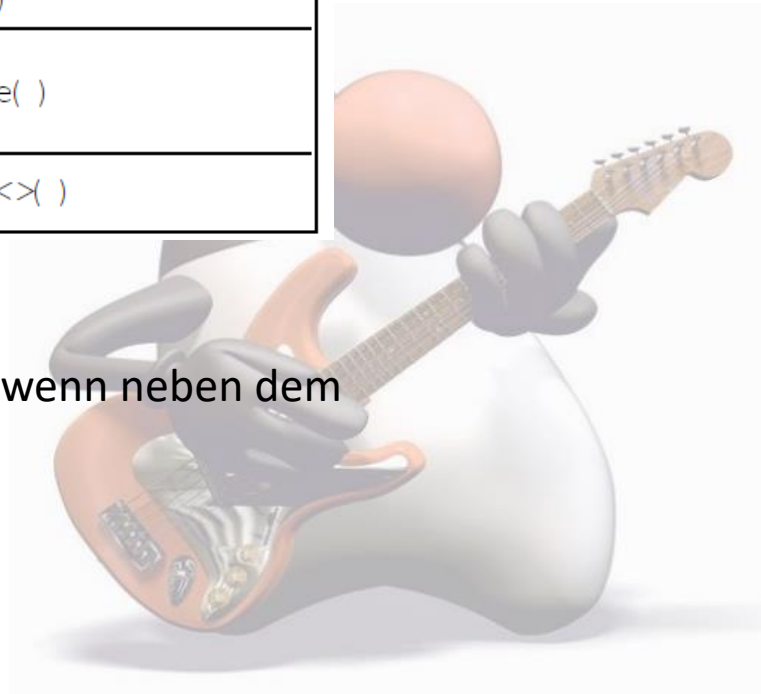


Methodenreferenzen

Weitere Beispiele:

Referenz auf ...	Als Methodenreferenz	Als Lambda
Statische Methode	`String::valueOf`	`obj -> String.valueOf(obj)`
Instanzmethode eines Typs	`Object::toString` `String::compareTo`	`obj -> obj.toString()` `(str1, str2) -> str1.compareTo(str2)`
Instanzmethode eines Objekts	`person::getName`	`() -> person.getName()`
Konstruktor	`ArrayList::new`	`() -> new ArrayList<>()`

Eine Lambda ist immer dann durch eine Methodenreferenz ersetzbar, wenn neben dem Methodenaufruf keine weiteren Aktionen in dem Lambda erfolgen.



Übung 5

- Sie haben die folgenden Klassen:

```
public class Extractor {
    public static <Person,R> void extractAndPrint(Function<Person, R> function, Person person){
        System.out.println(function.apply(person));
    }

    public static Person set(BiFunction<Person, String, Person> consumer, Person person, String value){
        return consumer.apply(person, value);
    }
}
```

```
public class Invoker {
    public static void main(String[] args) {
        Person person = new Person().setName("Michael").setEmail("michael@java.at");
        Extractor.extractAndPrint(Person::getName, person);
        Extractor.set(Person::setName,person,"Thomas");
        Extractor.extractAndPrint(Person::getName, person);
        Extractor.extractAndPrint(Person::getEmail, person);
        Extractor.extractAndPrint((p)->p.getEmail(),person);
    }
}
```

Schreiben Sie die Klasse Person und experimentieren Sie.

Agenda

- Collections & Lambdas
  - externe vs. Interne Iteration
  - Collections Erweiterungen
  - Streams
  - Filter-Map-Reduce
  - Fallstricke bei Lambdas

Externe versus interne Iteration

- Externe Iteration: Das Durchlaufen der Collection wird durch den Aufrufer kontrolliert. Z.B. über Index- oder Iteratorzugriff.
- Interne Iteration: Das Durchlaufen der Collection wird durch die Collectionklasse gekapselt und durch diese kontrolliert.

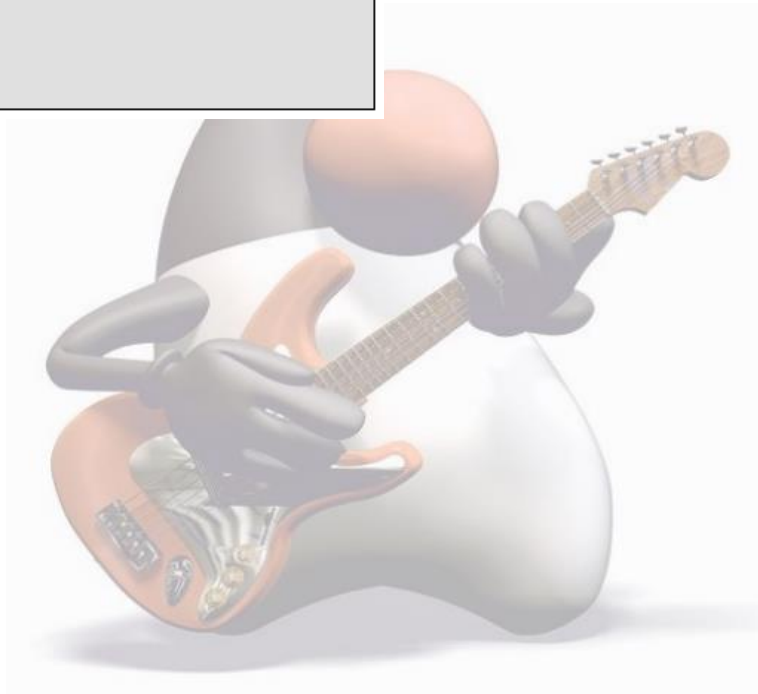
Interne Iteration in JDK 8

Mit JDK 8 bieten Collections die Methoden:

```
sort( Comparator<? super T>)
forEach( Consumer<? super T>)
```

Beispiele:

```
// Interne Iteration in drei Varianten
names. forEach( ( String name) -> { System. out. println( name); });
names. forEach( name -> System. out. println( name) );
names. forEach( System. out:: println);
```



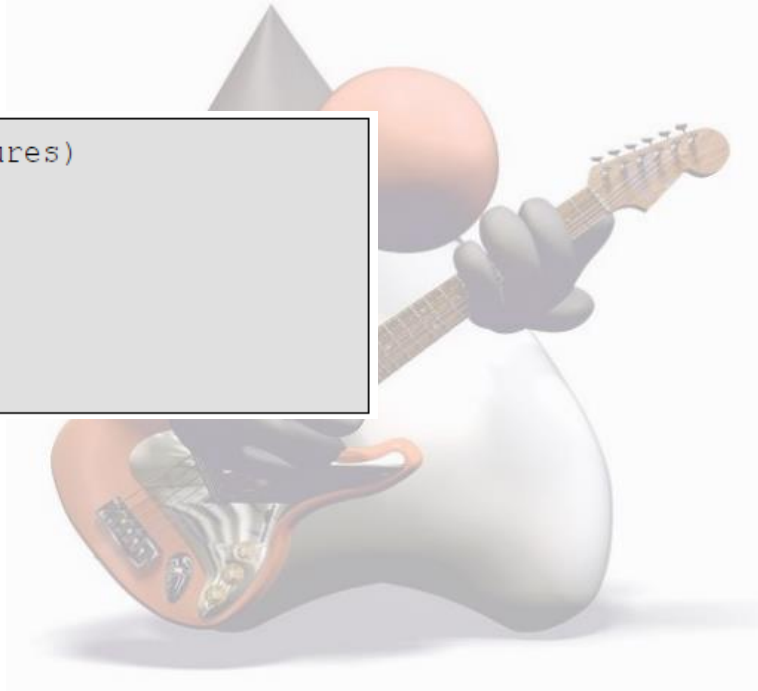
Externe versus interne Iteration

Externe Iteration:

```
public static void brightenExtern( final List<GraphicsFigure> selectedFigures)
{
    for ( final GraphicsFigure figure :  selectedFigures)
    {
        brighten( figure);
    }
}
```

Interne Iteration:

```
public static void brightenIntern( final List<GraphicsFigure> selectedFigures)
{
   selectedFigures. forEach( figure ->
   {
       brighten( figure);
   });
}
```



Externe versus interne Iteration

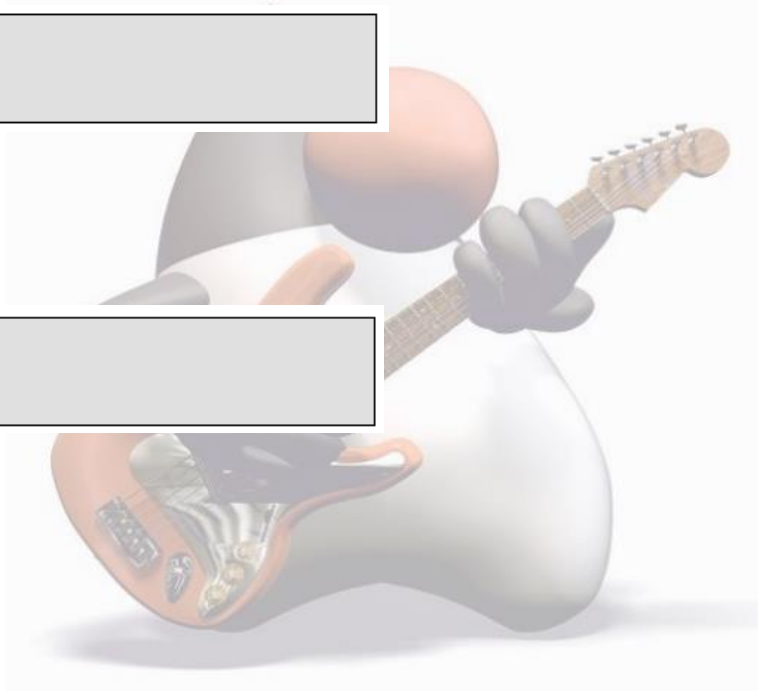
Weiteres Beispiel:

```
public static void process( final Collection<GraphicsFigure> figures,
                            final Consumer<GraphicsFigure> consumer)
{
    figures. forEach( consumer);
}
```

```
final Consumer<GraphicsFigure> brighten = figure -> brighten( figure);
process( figures, brighten) ;
```

Oder auch:

```
final Consumer<GraphicsFigure> brighten = figure -> brighten( figure);
figures. forEach( brighten) ;
```

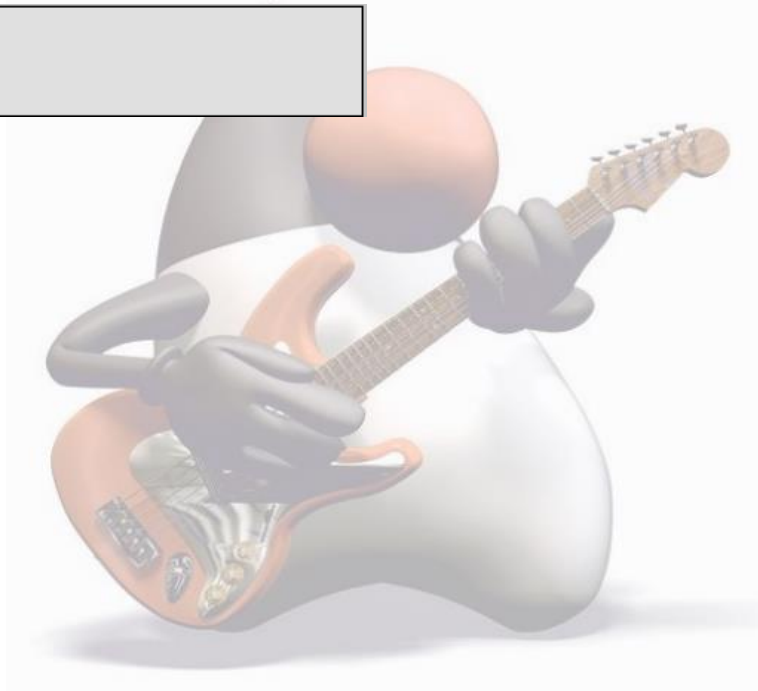


Externe versus interne Iteration

Weiteres Beispiel:

```
// Prüflogik mit externer Iteration
for ( final Person person : persons)
{
    Objects. requireNonNull( person);
}
```

```
// Prüflogik mit interner Iteration und Methodenreferenz
persons. forEach( Objects: : requireNonNull)
```



Übung 6

- Sie haben folgende Klasse:

```java
public class Person {
    private long id;
    private String name;
    public Person(long id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
    public long getId() {
        return id;
    }
    public Person setId(long id) {
        this.id = id;
        return this;
    }
    public String getName() {
        return name;
    }
    public Person setName(String name) {
        this.name = name;
        return this;
    }
}
```

Konstruieren Sie Liste von Persons.
Konstruieren Sie aus der Liste eine Map, key: id der Person, value: Person
Lösen Sie die Aufgabe einmal mit interner und einmal mit externer Iteration.

Agenda

- Collections & Lambdas
  - externe vs. Interne Iteration
  - Collections Erweiterungen
  - Streams
  - Filter-Map-Reduce
  - Fallstricke bei Lambdas

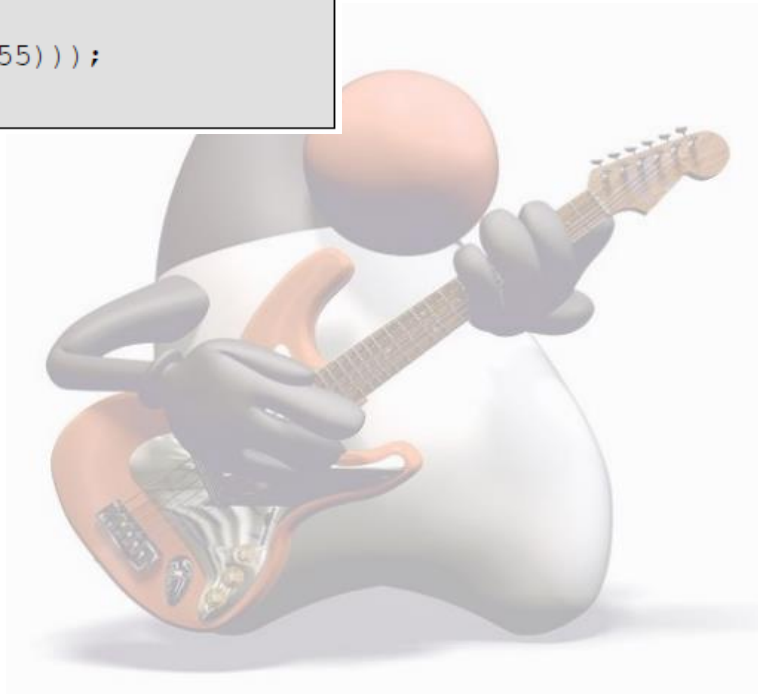
Das Interface Predicate<T>

```
@FunctionalInterface
public interface Predicate<T>
{
   boolean test( T t);

   default Predicate<T> and( Predicate<? super T> other) { ... }
   default Predicate<T> negate( ) { ... }
   default Predicate<T> or( Predicate<? super T> other) { ... }
}
```

Das Interface Predicate<T>

Beispiel:

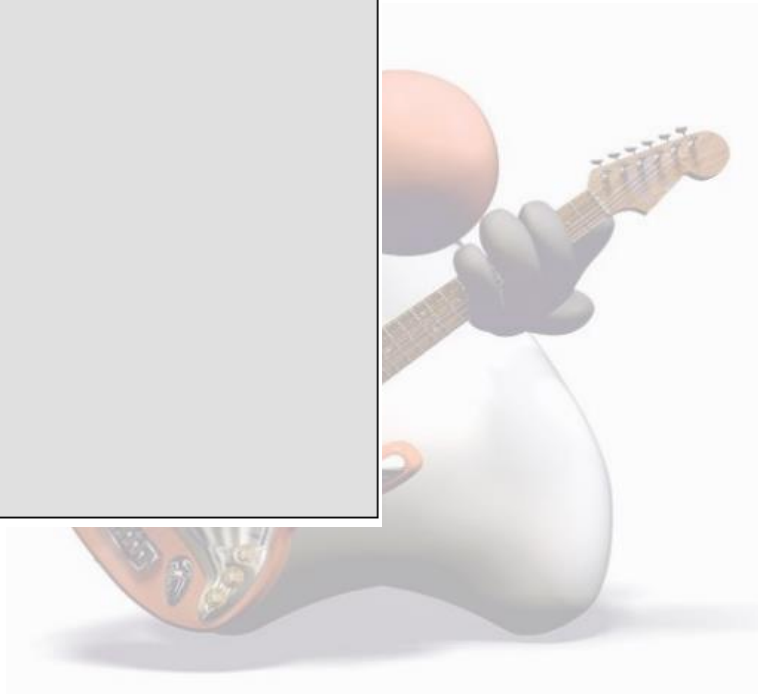
```
public static void main( final String[ ] args)
{
   // Prädikate formulieren
   final Predicate<String> isNull = str -> str == null;
   final Predicate<String> isEmpty = String::isEmpty;
   final Predicate<Person> isAdult = person -> person.getAge( ) >= 18;

   System.out.println( "isNull( ''):      " + isNull.test( "")); 
   System.out.println( "isEmpty( ''):     " + isEmpty.test( "")); 
   System.out.println( "isEmpty( 'Pia'):  " + isEmpty.test( "Pia")); 
   System.out.println( "isAdult( Pia):    " + isAdult.test( new Person( "Pia", 55))); 
}
```



Das Interface `Predicate<T>`

Beispiele mit `negate, and, or`:

```
public static void main( final String[ ] args)
{
   final List<Person> persons = createDemoData( );

   // Einfache Prädikate formulieren
   final Predicate<Person> isAdult = person -> person. getAge( ) >= 18;
   final Predicate<Person> isMale = person -> person. getGender( ) == Gender. MALE;

   // Negation einzelner Prädikate
   final Predicate<Person> isYoung = isAdult. negate( );
   final Predicate<Person> isFemale = isMale. negate( );

   // Kombination von Prädikaten mit AND
   final Predicate<Person> boys = isMale. and( isYoung);
   final Predicate<Person> women = isFemale. and( isAdult);

   // Kombination von Prädikaten mit OR
   final Predicate<Person> boysOrWomen = boys. or( women);

   removeAll( persons, boysOrWomen);
   System. out. println( persons);
}
```



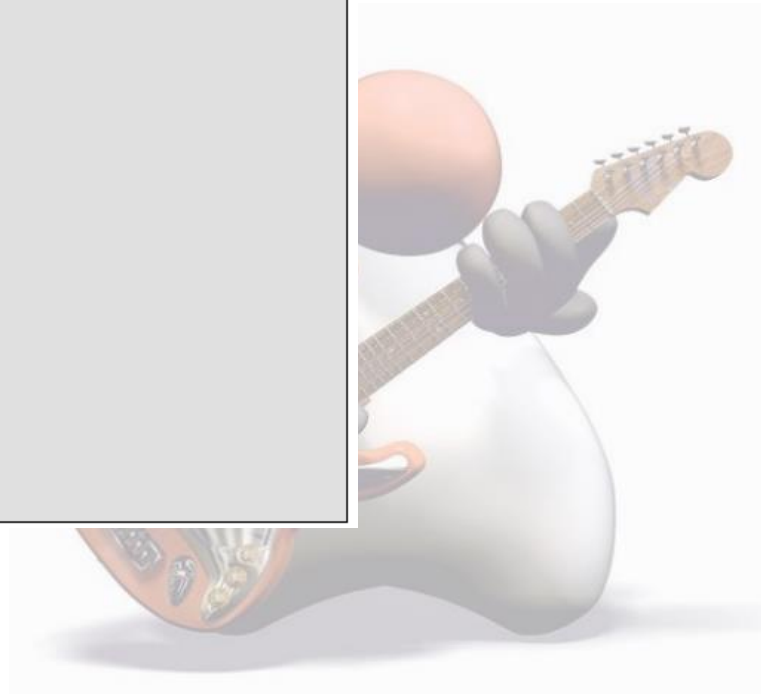
removeIf Methode in Collection

default boolean **removeIf(Predicate<? super E> filter)**

Removes all of the elements of this collection that satisfy the given predicate.

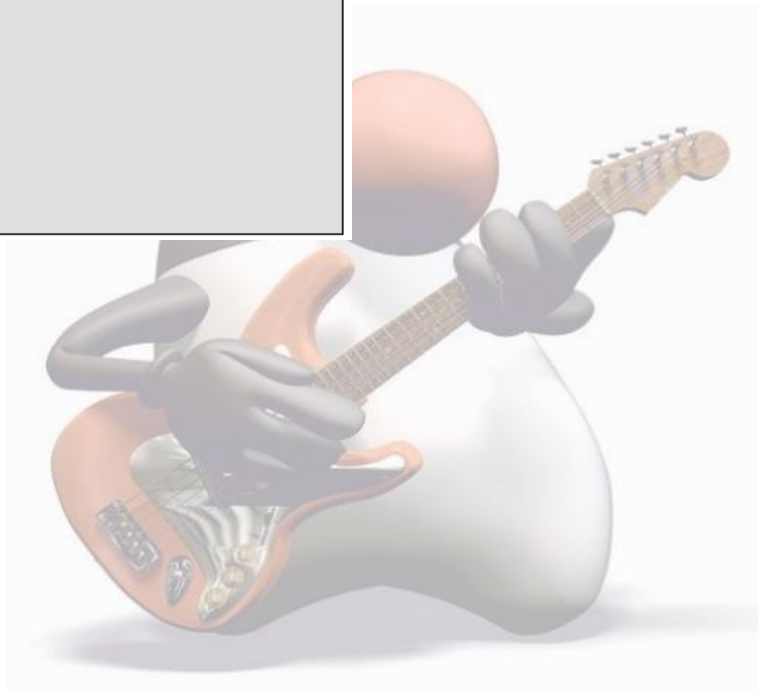
Beispiel:

```
public static void main( final String[ ] args)
{
   final List<String> names = createDemoNames( );

   // Löschaktionen ausführen
   names. removeIf( String: : isEmpty)
   System. out. println( names);
}

private static List<String> createDemoNames( )
{
   final List<String> names = new ArrayList<>( );
   names. add( "Max");
   names. add( "");                    // Leereintrag
   names. add( "Andy");
   names. add( "Michael");
   names. add( " ");                   // potenziell auch ein "Leereintrag"
   names. add( "Stefan");
   return names;
}
```



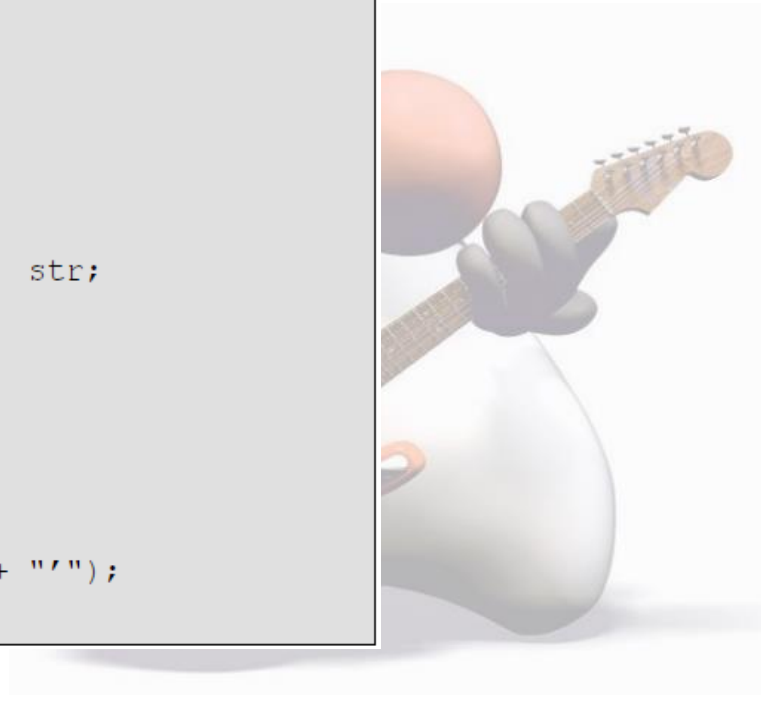
Interface Function<T,R> und UnaryOperator<T>

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T>
{
    // Statische Methoden seit JDK 8 in Interfaces erlaubt
    static <T> UnaryOperator<T> identity( )
    {
        return t -> t;
    }
}

@FunctionalInterface
public interface Function<T, R>
{
    R apply( T t);
    // ...
}
```



Interface Function<T,R> und UnaryOperator<T>

```
public static void main( final String[ ] args)
{
   // Mark
   final UnaryOperator<String> markTextWithM = str -> str.startsWith( "M") ?
                                        ">>" + str.toUpperCase( ) + "<<" : str;

   printResult( "Mark 1", "unchanged", markTextWithM);
   printResult( "Mark 2", "Michael", markTextWithM);

   // Trim
   final UnaryOperator<String> trimmer = String::trim;
   printResult( "Trim 1", "no_trim", trimmer);
   printResult( "Trim 2", " trim me ", trimmer);

   // Map
   final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" : str;
   printResult( "Map same", "same", mapNullToEmpty);
   printResult( "Map null", null, mapNullToEmpty);
}

private static void printResult( final String text, final String value,
                                 final UnaryOperator<String> op)
{
   System.out.println( text + ": '" + value + "' -> '" + op.apply( value) + "'");
}
```

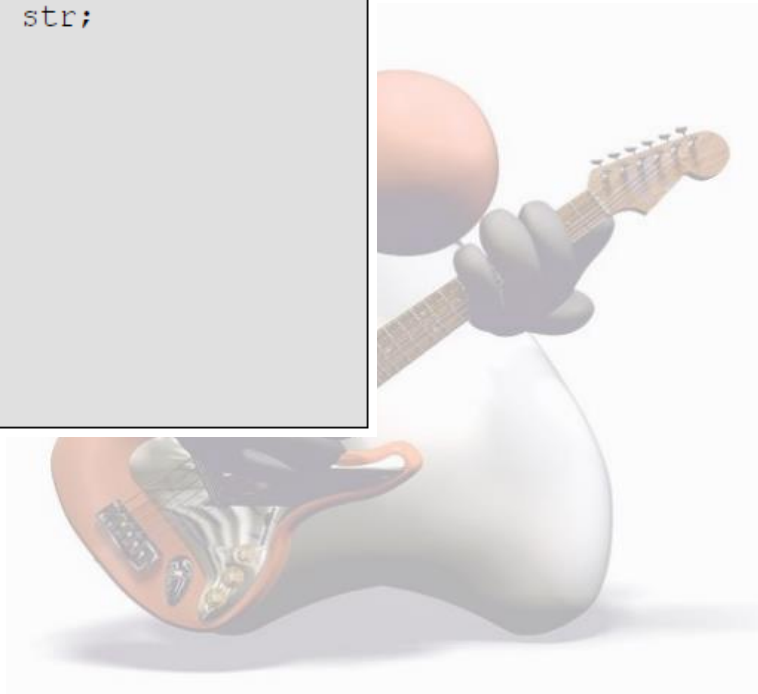


replaceAll Methode in List

default void	**replaceAll(UnaryOperator<E> operator)** Replaces each element of this list with the result of applying the operator to that element.

Beispiel:

```
public static void main( final String[ ] args)
{
   final List<String> names = createDemoNames( );

   // Spezialbehandlung von null-Werten
   final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" : str;
   names. replaceAll( mapNullToEmpty);

   // Leerzeichen abschneiden
   names. replaceAll( String:: trim);

   // Leereinträge herausfiltern
   names. removeIf( String:: isEmpty);

   System. out. println( inputs);
}
```



Übung 7

- Erstellen Sie eine Liste von 1000 Zufallszahlen zwischen 0 und 100
- Löschen Sie alle Elemente, die > 0 und <100
- Verwenden Sie dazu das FunctionalInterface Predicate

Agenda

- Collections & Lambdas
  - externe vs. Interne Iteration
  - Collections Erweiterungen
  - Streams
  - Filter-Map-Reduce
  - Fallstricke bei Lambdas

Streams

java.util.stream.Stream != java.io.XyzStream

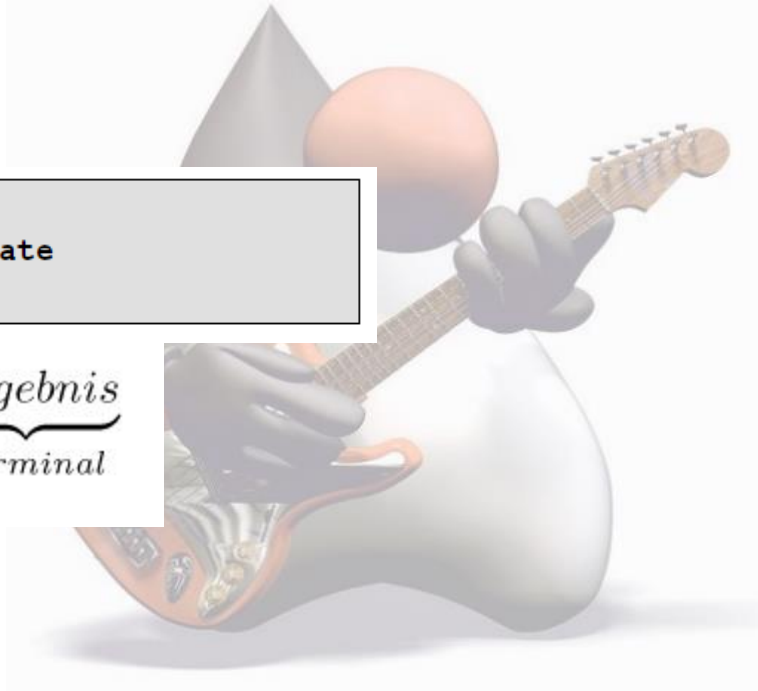
```
public interface Stream<T>
extends BaseStream<T,Stream<T>>
```

A sequence of elements supporting sequential and parallel aggregate operations.

Beispiel:

```
List<Person> adults = persons.stream().                    // Create
                              filter(Person::isAdult).     // Intermediate
                              collect(Collectors.toList()); // Terminal
```

$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow \ldots \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$



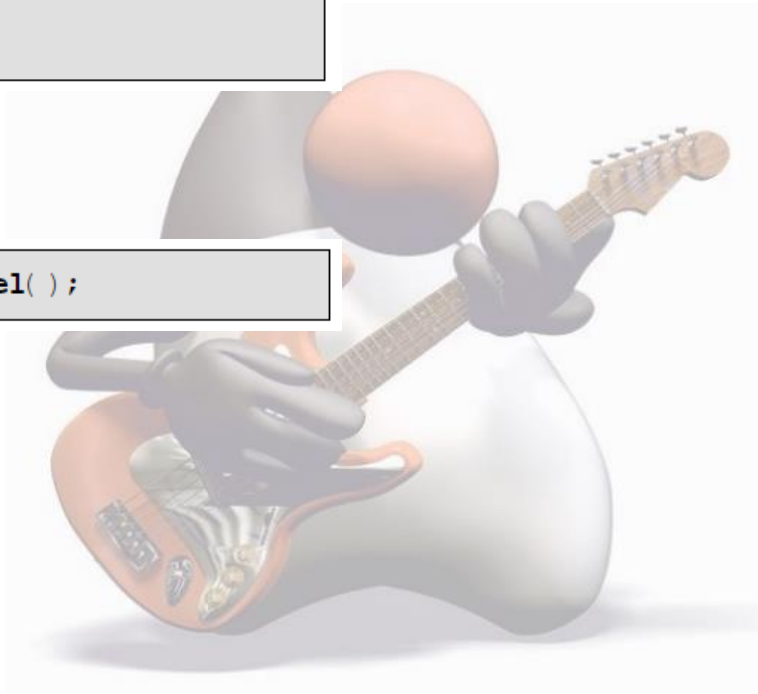
Erzeugen von Streams

```
final String[ ] namesData = {"Karl", "Ralph", "Andi", "Andy", "Mike" };
final List<String> names = Arrays.asList( namesData);

final Stream<String> streamFromArray = Arrays.stream( namesData);
final Stream<String> streamFromList = names.stream( );
```

Als Besonderheit können Collections eine sequenzielle sowie eine parallele Variante eines Streams liefern:

```
final Stream<String> sequentialStream = names.stream( );
final Stream<String> parallelStream = names.parallelStream( );
```

```
final Stream<String> parallelArrayStream = Arrays.stream( namesData).parallel( );
```



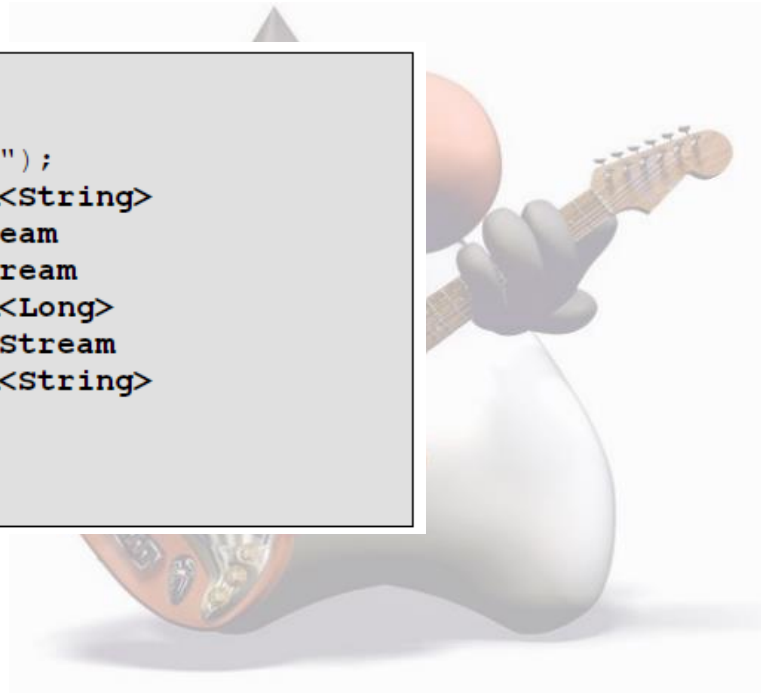
Erzeugen von Streams

```
final Stream<String> names = Stream.of( "Tim", "Andy", "Mike");     // String
final Stream<Integer> integers = Stream.of( 1, 4, 7, 7, 9, 7, 2);   // Integer

final IntStream values = IntStream.range( 0, 100);                  // int
final IntStream chars = "This is a test".chars( );                  // int
```

Stream von primitiven int Werten

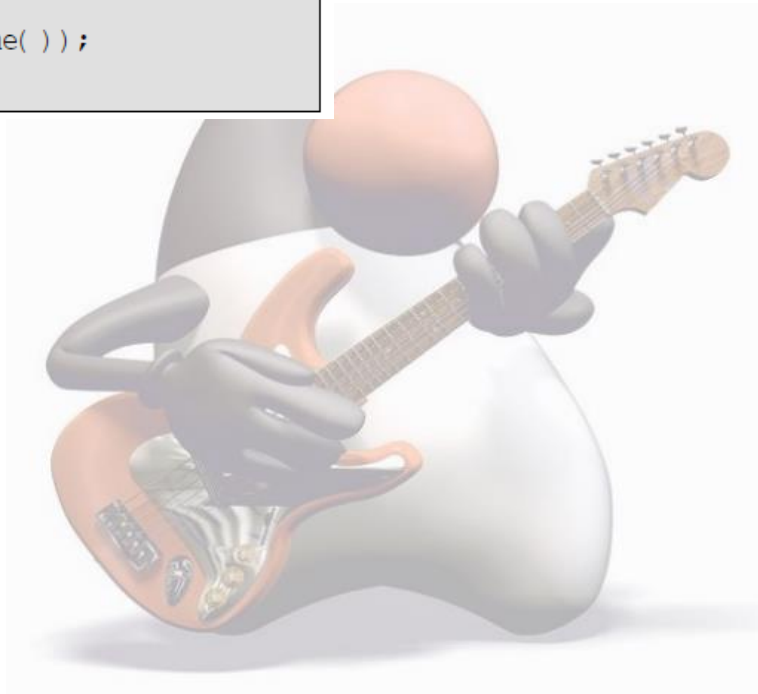
Umwandlung in Stream<Long>

```
public static void main( final String[ ] args)
{
   final List<String> names = Arrays.asList( "Mike", "Stefan", "Nikolaos");
   Stream<String> values = names.stream( ).                    // -> Stream<String>
                           mapToInt( String::length).          // -> IntStream
                           asLongStream( ).                    // -> LongStream
                           boxed( ).                           // -> Stream<Long>
                           mapToDouble( x -> x * .75).         // -> DoubleStream
                           mapToObj( val -> "Val:  " + val);   // -> Stream<String>

   values.forEach( System.out::println);
}
```



Unendliche Streams

```
public static void main( final String[ ] args)
{
   final IntStream iteratingValues = IntStream. iterate( 0, x -> x + 1);

   final AtomicInteger ai = new AtomicInteger( 0);
   final Stream<Integer> generatedValues = Stream. generate( ai: : getAndIncrement);

   final int[ ] firstTen = iteratingValues. limit( 10). toArray( );
   final Object[ ] secondTen = generatedValues. limit( 10). toArray( );

   System. out. println( Arrays. toString( firstTen));
   System. out. println( Arrays. toString( secondTen));
   System. out. println( "Element type: " + secondTen[ 0]. getClass( ). getTypeName( ));
}
```



$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow \ldots \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$

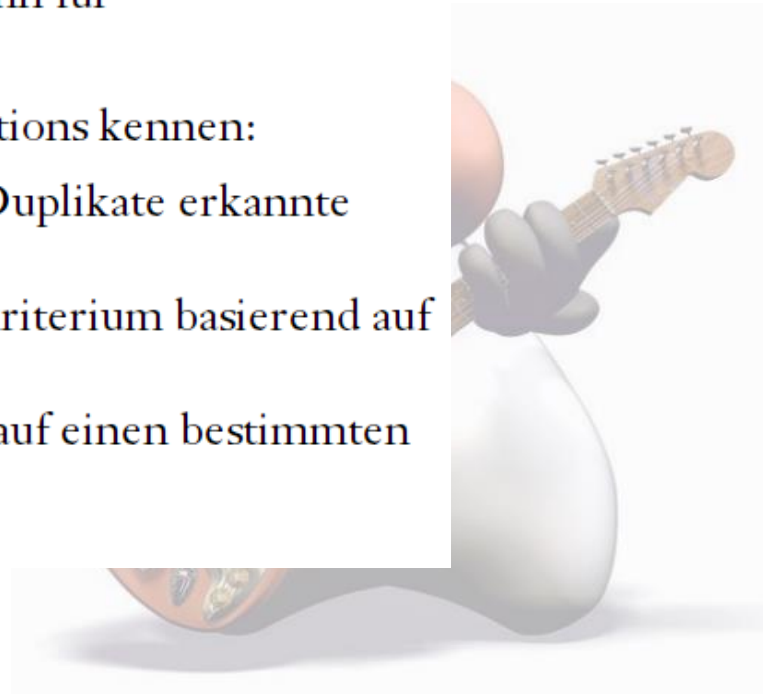
Intermediate Operations

Bei den zustandslosen Intermediate Operations sind folgende wichtig:

- `filter( )` – Filtert alle Elemente aus dem Stream heraus, die nicht dem übergebenen `Predicate<T>` genügen.
- `map( )` – Transformiert Elemente mithilfe einer `Function<T,R>` vom Typ `T` auf solche mit dem Typ `R`. Im Speziellen können die Typen auch gleich sein.
- `flatMap( )` – Bildet verschachtelte Streams als einen flachen Stream ab.
- `peek( )` – Führt eine Aktion für jedes Element des Streams aus. Dies kann für Debuggingzwecke sehr nützlich sein.

Darüber hinaus sollte man folgende zustandsbehaftete Intermediate Operations kennen:

- `distinct( )` – Entfernt alle gemäß der Methode `equals( Object)` als Duplikate erkannte Elemente aus einem Stream.
- `sorted( )` – Sortiert die Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- *`limit( )`* – Begrenzt die maximale Anzahl der Elemente eines Streams auf einen bestimmten Wert. Dies ist eine Short-circuiting Operation.
- `skip( )` – Überspringt die ersten *n* Elemente eines Streams.

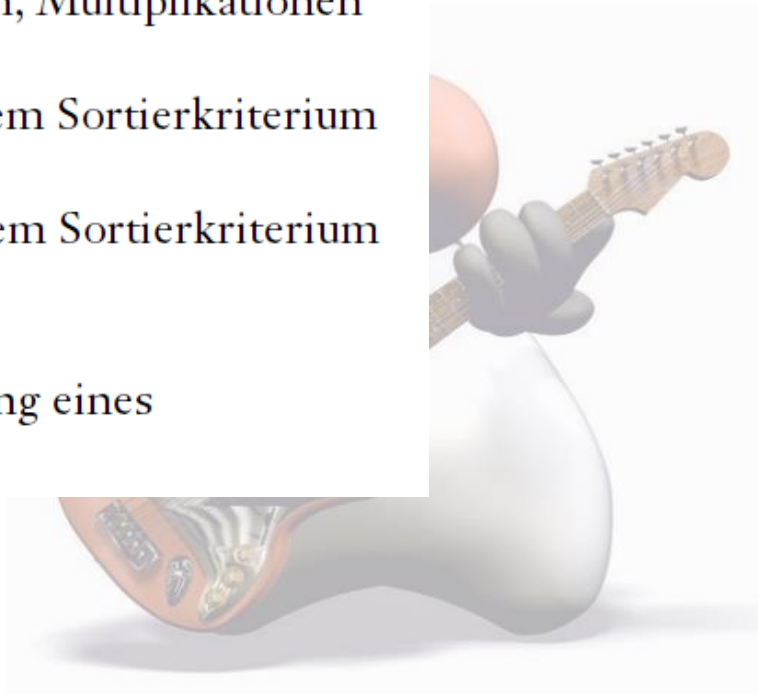


$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow \ldots \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$

Terminal Operations

Neben den umfangreichen Intermediate Operations wird eine noch imposantere Zahl an Terminal Operations geboten, unter anderem folgende:

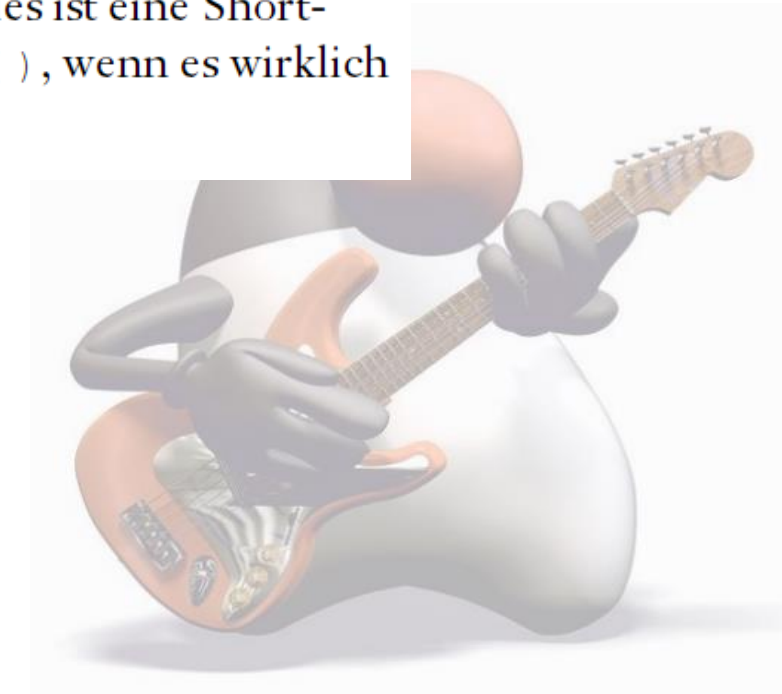
- `forEach( )` – Führt eine Aktion für jedes Element des Streams aus.
- `toArray( )` – Überträgt die Elemente aus dem Stream in ein Array.
- `collect( )` – Überträgt die Elemente aus dem Stream in eine Collection.
- `reduce( )` – Verbindet die Elemente eines Streams. Ein Beispiel ist die kommaseparierte Konkatenation von Strings. Alternativ kann man aber auch Summationen, Multiplikationen usw. ausführen, um einen Ergebniswert zu berechnen.
- `min( )` – Ermittelt das Minimum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `max( )` – Ermittelt das Maximum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `count( )` – Zählt die Anzahl an Elementen in einem Stream.
- *`anyMatch( )`* – Prüft, ob es mindestens ein Element gibt, das die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.



$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow \ldots \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$

Terminal Operations

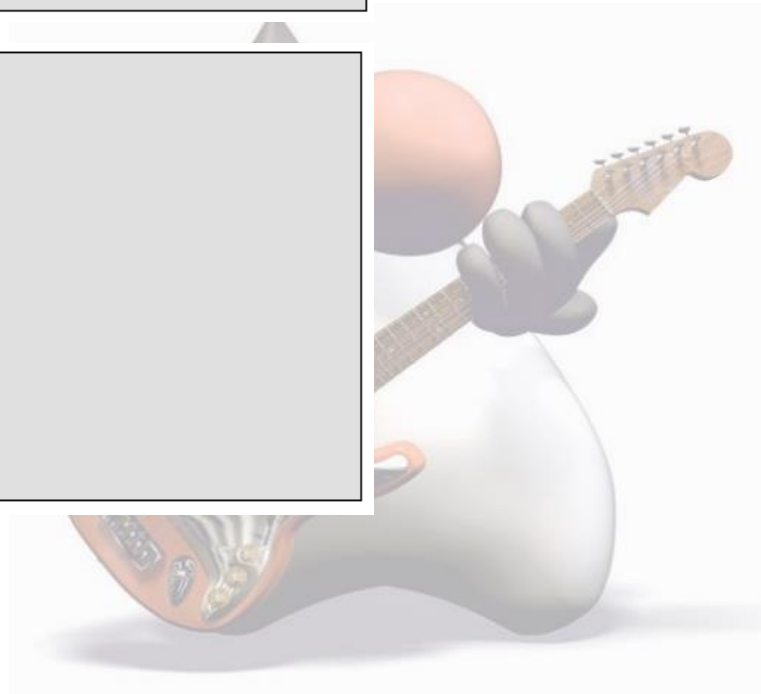
- `allMatch()` – Prüft, ob alle Elemente die Bedingung eines `Predicate<T>` erfüllen. Dies ist eine Short-circuiting Operation, die allerdings abbricht, wenn sie das erste Gegenbeispiel gefunden hat.
- `noneMatch()` – Prüft, ob keines der Elemente die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.
- `findFirst()` – Liefert das erste Element des Streams, falls es ein solches gibt. Dies ist eine Short-circuiting Operation.
- `findAny()` – Liefert ein beliebiges Element, falls es ein solches gibt. Dies ist eine Short-circuiting Operation und kann manchmal günstiger sein als `findFirst()`, wenn es wirklich nur darum geht, einen beliebigen Treffer zu erhalten.



filter

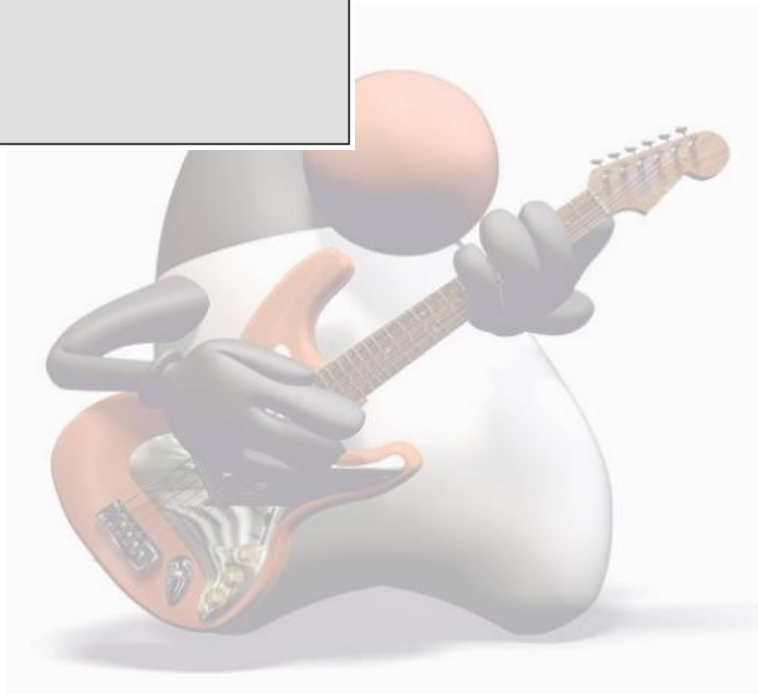
```
public static void main( final String[ ] args)
{
   final List<Person> persons = new ArrayList<>( );
   persons. add( new Person( "Micha", 43));
   persons. add( new Person( "Barbara", 40));
   persons. add( new Person( "Yannis", 5));

   // final Predicate<Person> isAdult = person -> person. getAge( ) >= 18;
   final Stream<Person> adults = persons. stream( ). filter( Person: : isAdult) ;

   adults. forEach( System. out: : println);
}
```

```
public class Person
{
   private int age;

   // ...

   public boolean isAdult( )
   {
       return age >= 18;
   }
}
```



map

```
public static void main( final String[ ] args)
{
   final List<Person> persons = new ArrayList<>( );
   persons. add( new Person( "Barbara", 40));
   persons. add( new Person( "Yannis", 5));

   // Mapping auf Name mit Lambda
   final Stream<Person> adults = persons. stream( ). filter( Person: : isAdult);
   final Stream<String> namesStream = adults. map( person -> person. getName( ));
   // Mapping auf Alter mit Methodenreferenz
   final Stream<Integer> agesStream = persons. stream( ). map( Person: : getAge).
                                                filter( age -> age >= 18);
   namesStream. forEach( System. out: : println);
   agesStream. forEach( System. out: : println);
}
```



flatMap

Aufgabe: Stream mit allen Wörtern aus mehreren Sätzen erzeugen:

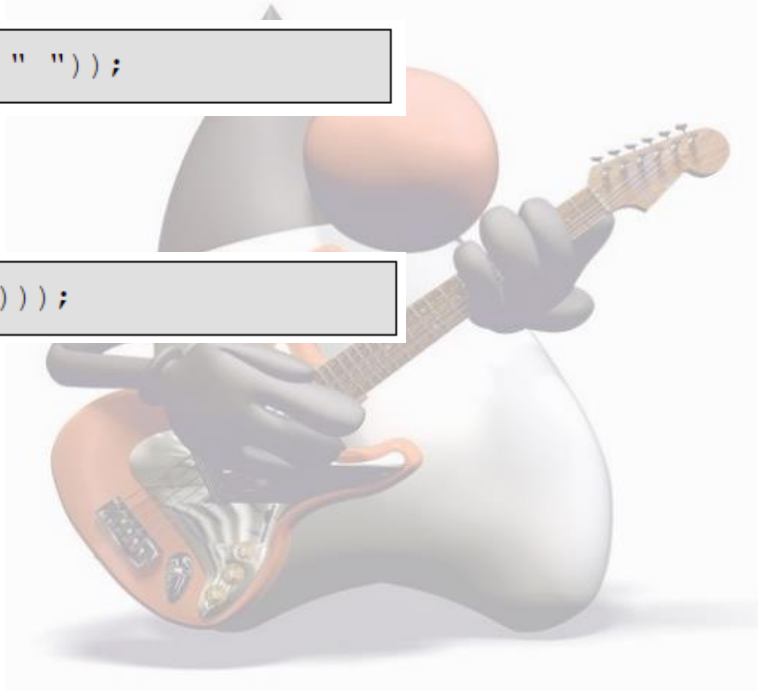
```
final List<String> sentences = Arrays.asList("This is the first line.",
                                             "The second line of this text.",
                                             "Third line contains some text.",
                                             "Last line and goodbye:",
                                             "End of text!");

final Stream<String> asStream = sentences.stream();
```

Folgendes führt zu ineinander verschachtelten Streams:

```
Stream<Stream<String>> words = asStream.map(line -> Stream.of(line.split(" ")));
```

Verschachtelte Stream können mittels flatMap „flachgeklopft“ werden:

```
Stream<String> words = asStream.flatMap(line -> Stream.of(line.split(" ")));
```

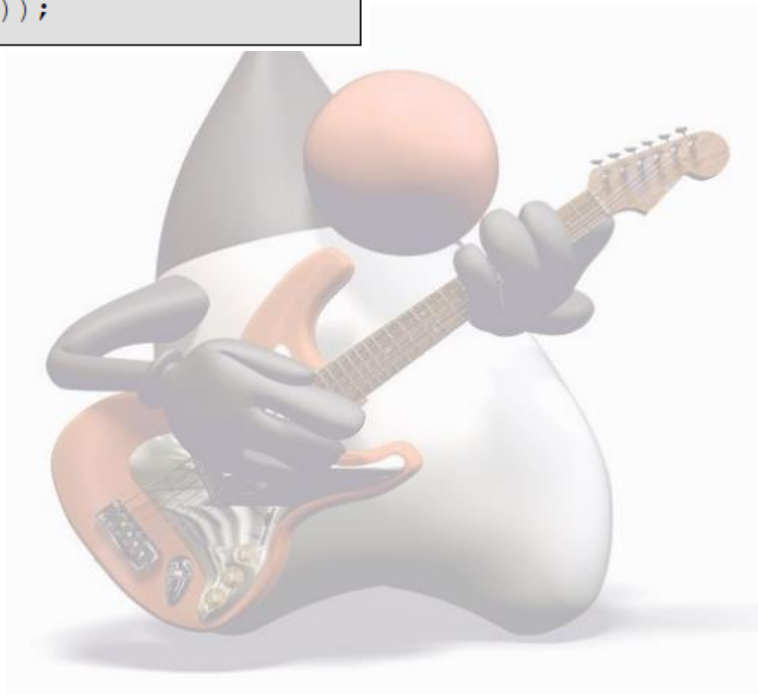


```java
public static void main( final String[ ] args) throws IOException
{
    final Path exampleFile = Paths. get( "src/main/resources/" +
                                         "jdk8/streams/Example. txt");

    // Datei einlesen neu in JDK 8:  Siehe dazu Kapitel 6
    final List<String> contents = Files. readAllLines( exampleFile);

   // Daraus einen Stream von Worten machen
   final Stream<String> words = contents. stream( ).
                                flatMap( line -> Stream. of( line. split( " ")));

   // Prädikate für kurze Wörter
   final Predicate<String> isShortWord = word -> word. length( ) <= 3;
   final Predicate<String> notIsShortWord = isShortWord. negate( );

   // Prädikate für spezielle und zu ignorierende Wörter
   final List<String> ignoreableWords = Arrays. asList( "this", "these", "them");
   final Predicate<String> isIgnorableWord = word ->
   {
       return ignoreableWords. contains( word. toLowerCase( ));
   };
   final Predicate<String> isSignificantWord = isIgnorableWord. negate( );

   // Filterung basierend auf den Prädikaten
   final Stream<String> filteredContents = words. map( String: : trim).
                                           filter( notIsShortWord).
                                           filter( isSignificantWord);

   filteredContents. forEach( it -> System. out. print( it + ", "));
}
```

peek

peek – auslesen und verarbeiten des aktuellen Wertes als Intermediate Operation

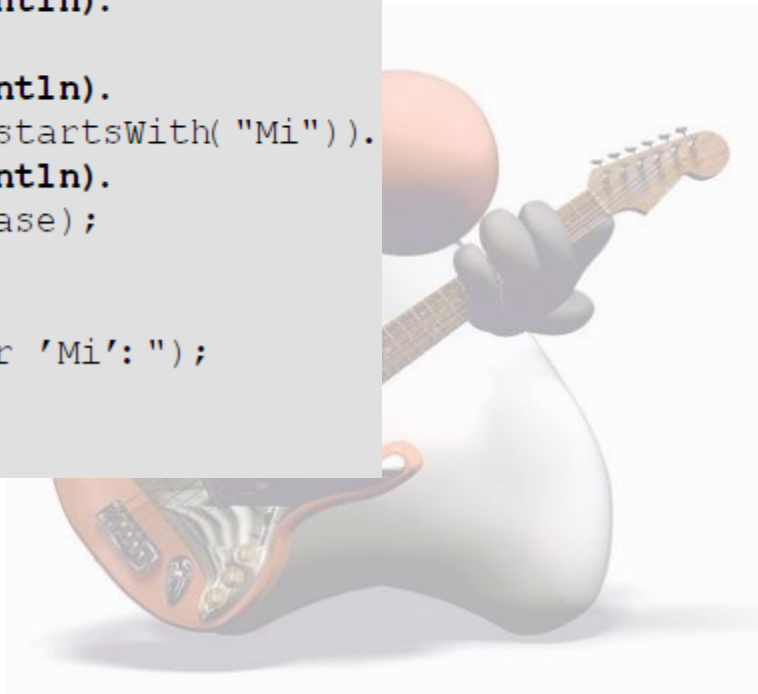
```
final Stream<Person> adults = persons.stream().filter(Person::isAdult);

// Ausgabe, um die Filterung zu überprüfen
adults.forEach(System.out::println);

// Weitere Filterung auf dem Stream vornehmen
final Stream<Person> mikes = adults.filter(person ->
                                           person.getName().equals("Mike"));
```



peek

peek – auslesen und verarbeiten des aktuellen Wertes als Intermediate Operation

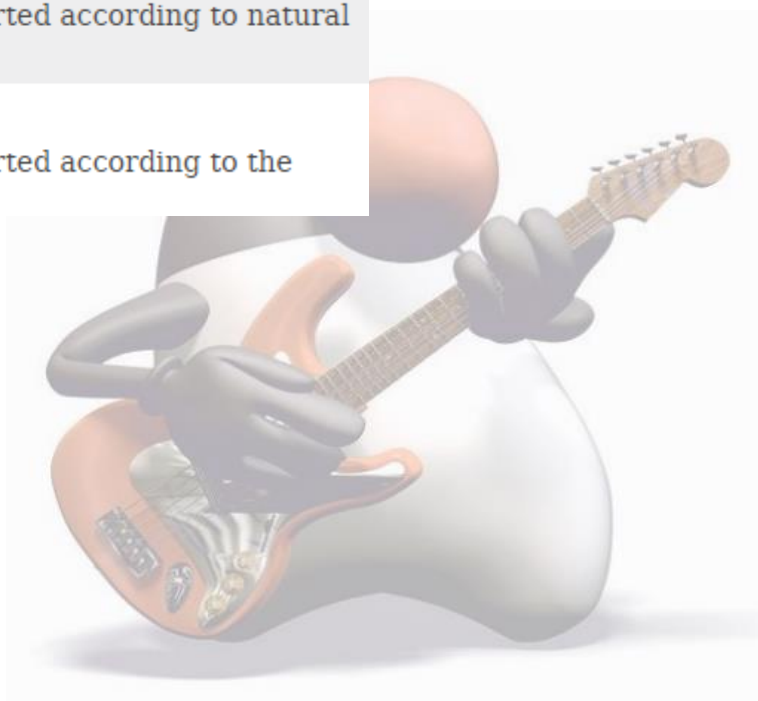
```
public static void main( final String[ ] args)
{
   final List<Person> persons = createDemoData( );

   final Stream<Person> stream = persons. stream( );
   final Stream<String> allMikes = stream. peek( System. out: : println).
                                           filter( Person: : isAdult).
                                           peek( System. out: : println).
                                           map( Person: : getName).
                                           peek( System. out: : println).
                                           filter( name -> name. startsWith( "Mi")).
                                           peek( System. out: : println).
                                           map( String: : toUpperCase);

   // Löst die Verarbeitung aus
   System. out. println( "Protokollierung jedes Schritts -- Filter 'Mi': ");
   allMikes. forEach( System. out: : println);
}
```



distinct und sorted

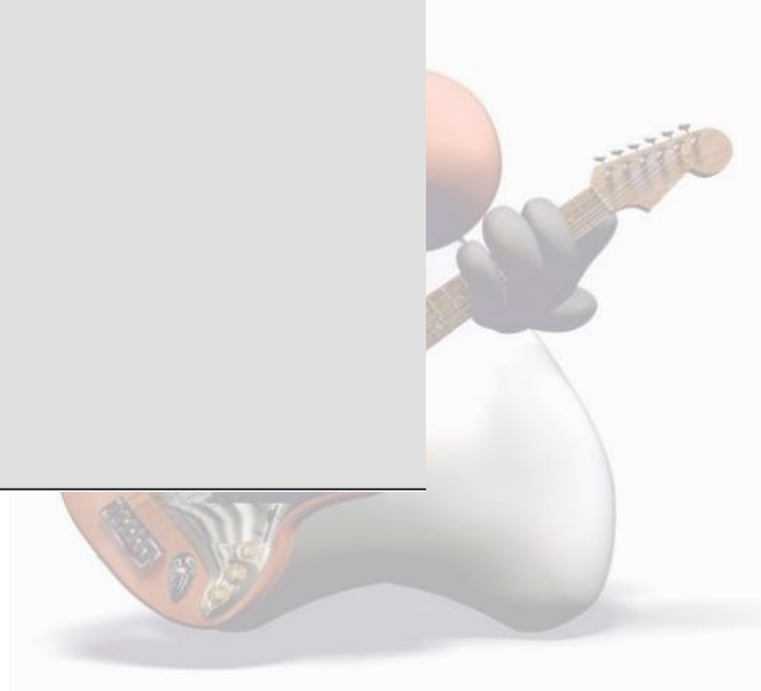
`Stream<T>`	`distinct()` Returns a stream consisting of the distinct elements (according to `Object.equals(Object)`) of this stream.

`Stream<T>`	`sorted()` Returns a stream consisting of the elements of this stream, sorted according to natural order.
`Stream<T>`	`sorted(Comparator<? super T> comparator)` Returns a stream consisting of the elements of this stream, sorted according to the provided `Comparator`.



distinct und sorted

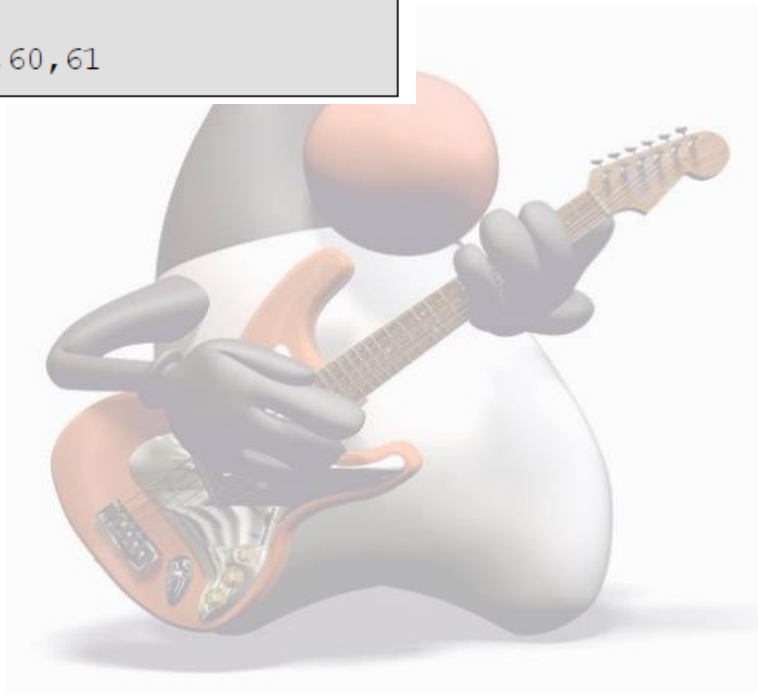
```
public static void main( final String[ ] args)
{
    final Stream<Integer> distinct = createIntStream( ). distinct( );
    final Stream<Integer> sorted= createIntStream( ). sorted( );
    final Stream<Integer> sortedAndDistinct = createIntStream( ). sorted( ).
                                                                  distinct( );
    printResult( "distinct:           ", distinct);
    printResult( "sorted:             ", sorted);
    printResult( "sortedAndDistinct:  ", sortedAndDistinct);
}

private static Stream<Integer> createIntStream( )
{
    return Stream. of( 7, 1, 4, 3, 7, 2, 6, 5, 7, 9, 8);
}

private static void printResult( final String hint,
                                 final Stream<Integer> stream)
{
    final List<Integer> result = stream. collect( Collectors. toList( ));
    System. out. println( hint + result);
}
```



limit und skip

`Stream<T>`	**`limit`**`(long maxSize)` Returns a stream consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.
`Stream<T>`	**`skip`**`(long n)` Returns a stream consisting of the remaining elements of this stream after discarding the first `n` elements of the stream.

```
final IntStream iteratingValues = IntStream.iterate( 0, x -> x + 1);

iteratingValues.skip( 50).limit( 12); // => 50,51,52,53,54,55,56,57,58,59,60,61
```

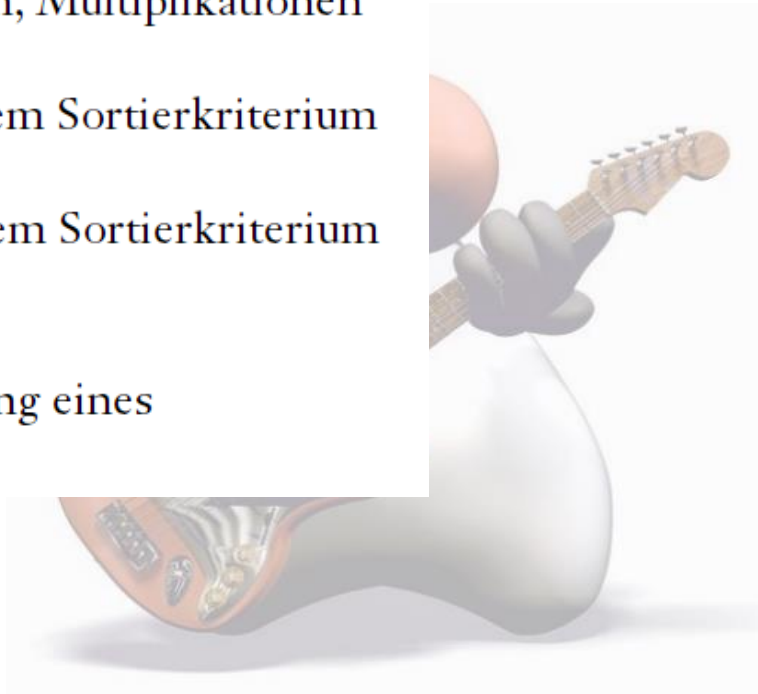


$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow \ldots \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$

Terminal Operations

Neben den umfangreichen Intermediate Operations wird eine noch imposantere Zahl an Terminal Operations geboten, unter anderem folgende:

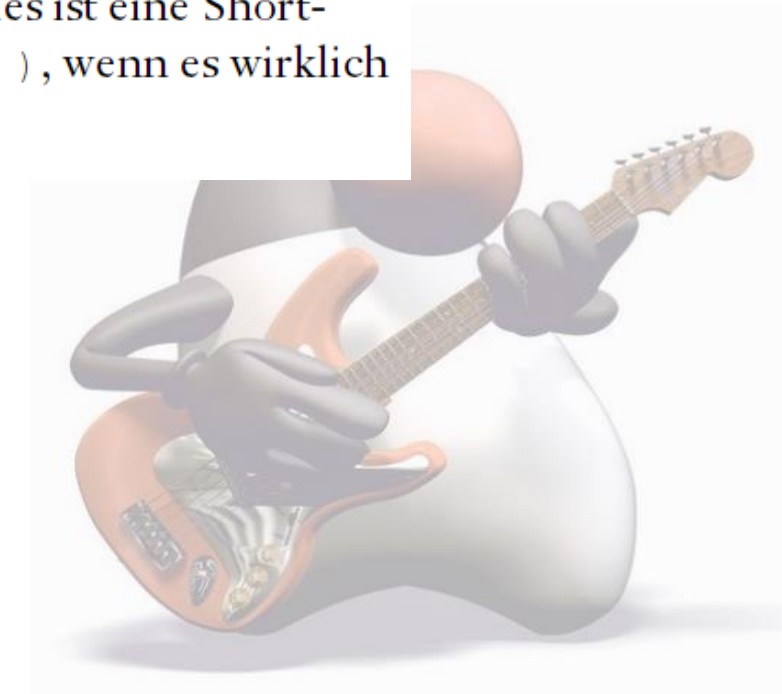
- `forEach( )` – Führt eine Aktion für jedes Element des Streams aus.
- `toArray( )` – Überträgt die Elemente aus dem Stream in ein Array.
- `collect( )` – Überträgt die Elemente aus dem Stream in eine Collection.
- `reduce( )` – Verbindet die Elemente eines Streams. Ein Beispiel ist die kommaseparierte Konkatenation von Strings. Alternativ kann man aber auch Summationen, Multiplikationen usw. ausführen, um einen Ergebniswert zu berechnen.
- `min( )` – Ermittelt das Minimum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `max( )` – Ermittelt das Maximum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `count( )` – Zählt die Anzahl an Elementen in einem Stream.
- *`anyMatch( )`* – Prüft, ob es mindestens ein Element gibt, das die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.



$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow \ldots \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$

Terminal Operations

- `allMatch()` – Prüft, ob alle Elemente die Bedingung eines `Predicate<T>` erfüllen. Dies ist eine Short-circuiting Operation, die allerdings abbricht, wenn sie das erste Gegenbeispiel gefunden hat.
- `noneMatch()` – Prüft, ob keines der Elemente die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.
- `findFirst()` – Liefert das erste Element des Streams, falls es ein solches gibt. Dies ist eine Short-circuiting Operation.
- `findAny()` – Liefert ein beliebiges Element, falls es ein solches gibt. Dies ist eine Short-circuiting Operation und kann manchmal günstiger sein als `findFirst()`, wenn es wirklich nur darum geht, einen beliebigen Treffer zu erhalten.



forEach

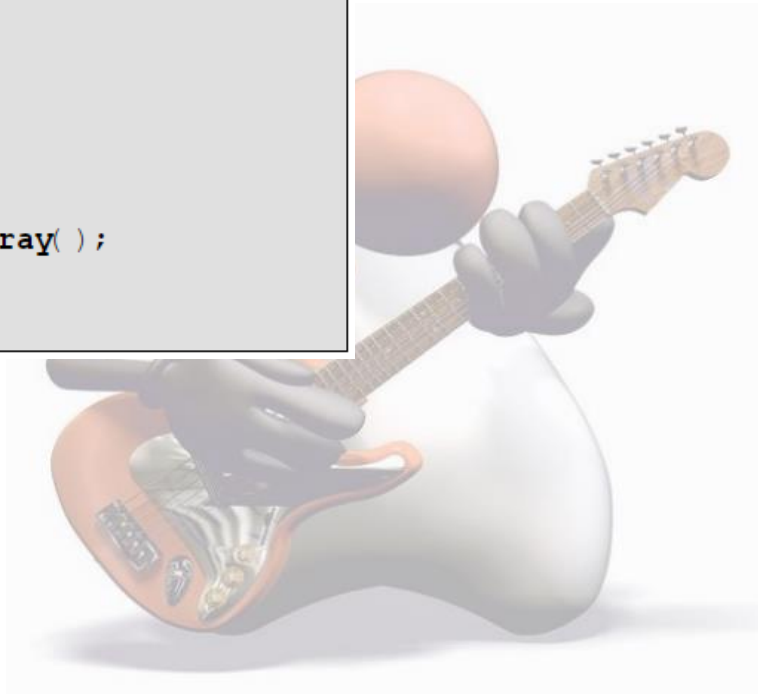
void	**forEach**(**Consumer**<? super **T**> action) Performs an action for each element of this stream.

```
streamFromArray.forEach(System.out::println);
streamFromValues.sorted().distinct().forEach(System.out::println);
```

toArray

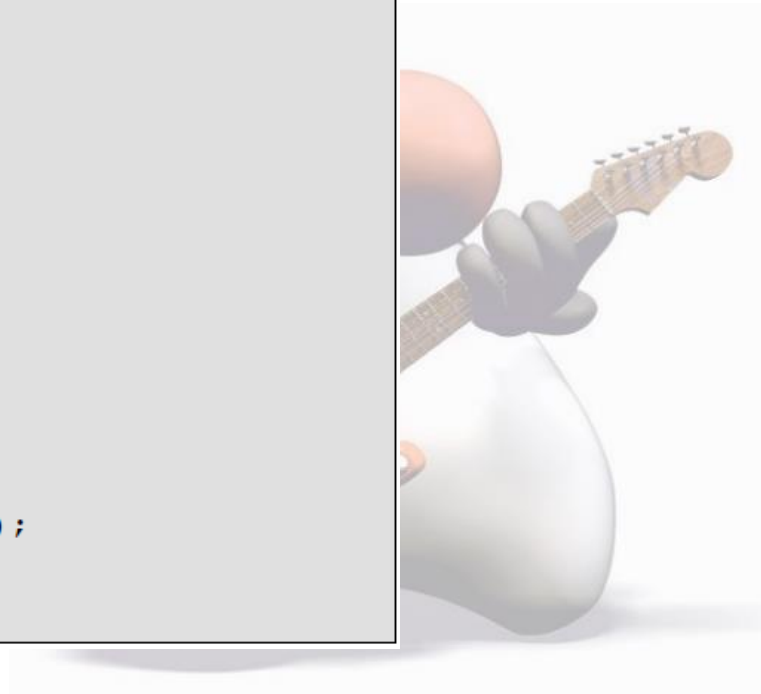
`Object[]`	`toArray()` Returns an array containing the elements of this stream.

```
public static void main( final String[ ] args)
{
    // Zufallszahlen von 0 bis 100
    final Random random = new Random( );
    final Supplier<Float> randomSupplier = ( ) -> random. nextFloat( ) * 100;

    final Object[ ] randomNumbers = Stream. generate( randomSupplier).
                                        limit( 7). toArray( );
    System. out. println( Arrays. toString( randomNumbers));
    System. out. println( "Element type: " + randomNumbers[ 0]. getClass( ));

    final int[ ] intRandoms = Stream. generate( randomSupplier).
                              limit( 7). mapToInt( val -> val. intValue( )). toArray( );
   System. out. println( Arrays. toString( intRandoms));
}
```



count,sum,min,max,average

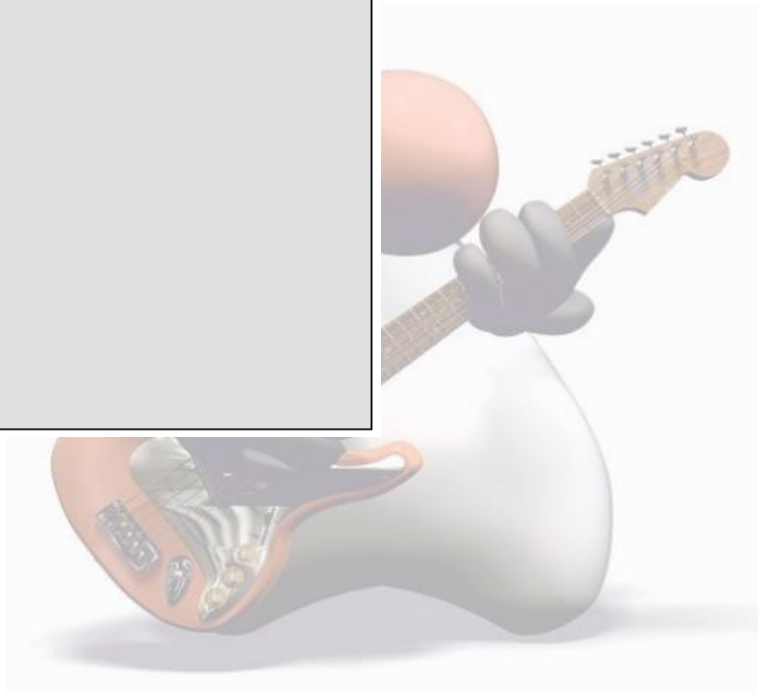
```
public static void main( final String[ ] args)
{
    final List<Person> persons = new ArrayList<>( );
    persons. add( new Person( "Ten", 10));
    persons. add( new Person( "Twenty", 20));
    persons. add( new Person( "Thirty", 30));
    persons. add( new Person( "Forty", 40));

    // Anzahl an Personen, deren Name mit 'T' startet
    final int count = persons. stream( ). filter( person -> person. getName( ).
                                        startsWith( "T")). count( );
    System. out. println( "count:  " + count);

    // Summe berechnen
    final int sum = persons. stream( ). filter( person -> person. getName( ).
                                       startsWith( "T")).
                                       mapToInt( Person: : getAge). sum( );
    System. out. println( "sum:  " + sum);

    // Durchschnitt berechnen
    final OptionalDouble avg = persons. stream( ).
                                        filter( person -> person. getName( ).
                                        contains( "X")).
                                        mapToInt( Person: : getAge). average( );
    System. out. println( "avg:  " + avg);
}
```



allMatch, anyMatch, noneMatch

boolean	**allMatch**(**Predicate**<? super T> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	**anyMatch**(**Predicate**<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
boolean	**noneMatch**(**Predicate**<? super T> predicate) Returns whether no elements of this stream match the provided predicate.

```
public static void main( final String[ ] args)
{
   final List<String> names = Arrays. asList( "Tim", "Tom", "Micha");

   final Predicate<String> startWithT = str -> str. startsWith( "T");

   final boolean allStartWithT = names. stream( ). allMatch( startWithT);
   final boolean anyStartWithT = names. stream( ). anyMatch( startWithT);
   final boolean noneStartWithT = names. stream( ). noneMatch( startWithT);

   System. out. println( "allStartWithT:   " + allStartWithT);
   System. out. println( "anyStartWithT:   " + anyStartWithT);
   System. out. println( "noneStartWithT:  " + noneStartWithT);
}
```



findFirst, findAny

`Optional<T>`	**`findAny()`** Returns an **`Optional`** describing some element of the stream, or an empty `Optional` if the stream is empty.
`Optional<T>`	**`findFirst()`** Returns an **`Optional`** describing the first element of this stream, or an empty `Optional` if the stream is empty.

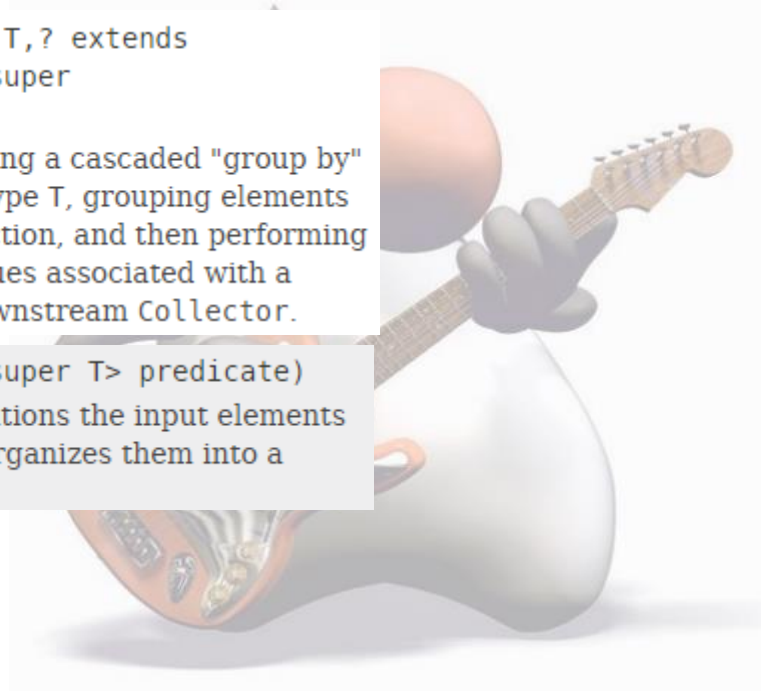
```
final Optional<Person> optionalFirst = filteredPersons.findFirst();
final Optional<Person> optionalAny = filteredPersons.findAny();
```

anyMatch, findFirst

```
// Namensfilter definieren
final Predicate<Person> nameFilter = person -> person.getName().equals(desired);

// containsPersonWithName()
final boolean personFound = persons.stream().anyMatch(nameFilter);

// findPersonByName()
final Optional<Person> searchedPerson = persons.stream().filter(nameFilter).
                                                         findFirst();
```

Collectors: joining, groupingBy, partitioningBy

static Collector<CharSequence,?,String>	joining() Returns a Collector that concatenates the input elements into a String, in encounter order.
static <T,K> Collector<T,?,Map<K,List<T>>>	groupingBy(Function<? super T,? extends K> classifier) Returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.
static <T,K,A,D> Collector<T,?,Map<K,D>>	groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream) Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
static <T> Collector<T,?,Map<Boolean,List<T>>>	partitioningBy(Predicate<? super T> predicate) Returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>.



Collectors: joining, groupingBy, partitioningBy

```
import static java.util.stream.Collectors.counting;
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.joining;
import static java.util.stream.Collectors.partitioningBy;

// ...

final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",
                                         "Florian", "Michael", "Sebastian");

final String joined = names.stream().sorted().collect(joining(", "));

Map<Integer, List<String>> grouped =
                           names.stream().collect(groupingBy(String::length));

Map<Integer, Long> counting =
                   names.stream().collect(groupingBy(String::length,
                                          counting()()));

Map<Boolean, List<String>> partitions =
                           names.stream().filter(str -> str.contains("i")).
                           collect(partitioningBy(str -> str.length() > 4));

// ...
```

Parallelverarbeitung

Umschalten zwischen paralleler und sequentieller Verarbeitung:

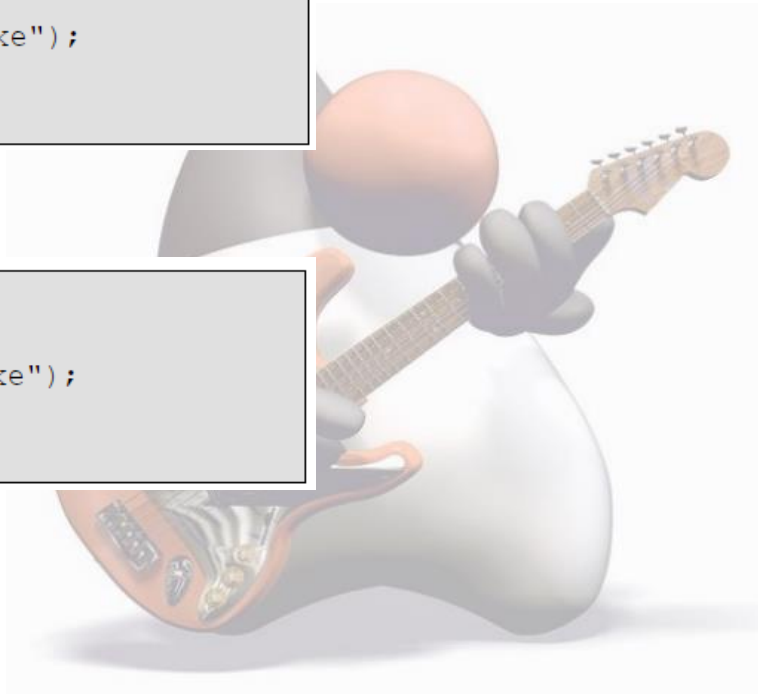
```
final String adults = persons.parallelStream().filter(Person::isAdult).
                              sequential().map(Person::getName).
                              collect(Collectors.joining(", "));
```

Folgender Code erzeugt trotz `sorted()` ggf. unsortierte Ergebnisse:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike");
    names.parallelStream().sorted().forEach(System.out::println);
}
```

Das kann gelöst werden mittels:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike");
    names.parallelStream().sorted().forEachOrdered(System.out::println);
}
```



Übung 8

- Sie haben eine Klasse Person

```
public class Person {
    private String name;
    private String email;

    public String getName() {
        return name;
    }
    public Person setName(String name) {
        this.name = name;
        return this;
    }
    public String getEmail() {
        return email;
    }
    public Person setEmail(String email) {
        this.email = email;
        return this;
    }


}
```

- Konstruieren Sie einen Array von Person mit und ohne Email Adressen
- Erstellen Sie eine Liste jener Personen, die eine nicht-leere Email Adressen haben

Übung 9

- Konstruieren Sie eine Liste, die 100.000 Persons hat. Setzen Sie den Namen einer Person auf „Schwarzer Peter“.
- Durchsuchen Sie den Array parallel nach dem Schwarzen Peter und extrahieren Sie dessen Email Adresse.
- Machen Sie die gleiche Suche sequentiell und vergleichen Sie die Zeiten.

Agenda

- Collections & Lambdas
  - externe vs. Interne Iteration
  - Collections Erweiterungen
  - Streams
  - Filter-Map-Reduce
  - Fallstricke bei Lambdas

Filter-Map-Reduce

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an `Optional` describing the reduced value, if any. This is equivalent to:

```
boolean foundAny = false;
T result = null;
for (T element : this stream) {
    if (!foundAny) {
        foundAny = true;
        result = element;
    }
    else
        result = accumulator.apply(result, element);
}
return foundAny ? Optional.of(result) : Optional.empty();
```

Filter-Map-Reduce

```
public static void main( final String[ ] args)
{
  final Stream<String> names = Stream. of( "Mike", "Tom", "Peter", "Chris");
  final Stream<Integer> integers = Stream. of( 1, 2, 3, 4, 5);
  final Stream<Integer> empty = Stream. of( );

  final Optional<String> stringConcat = names. reduce( ( s1,s2) -> s1 + ", " + s2);
  final Optional<Integer> multiplication = integers. reduce( ( s1,s2) -> s1 * s2);
  final Optional<Integer> addition = empty. reduce( ( s1,s2) -> s1 + s2);

  System. out. println( "stringConcat:    " + stringConcat);
  System. out. println( "multiplication:  " + multiplication);
  System. out. println( "addition:        " + addition);
}
```

Filter-Map-Reduce

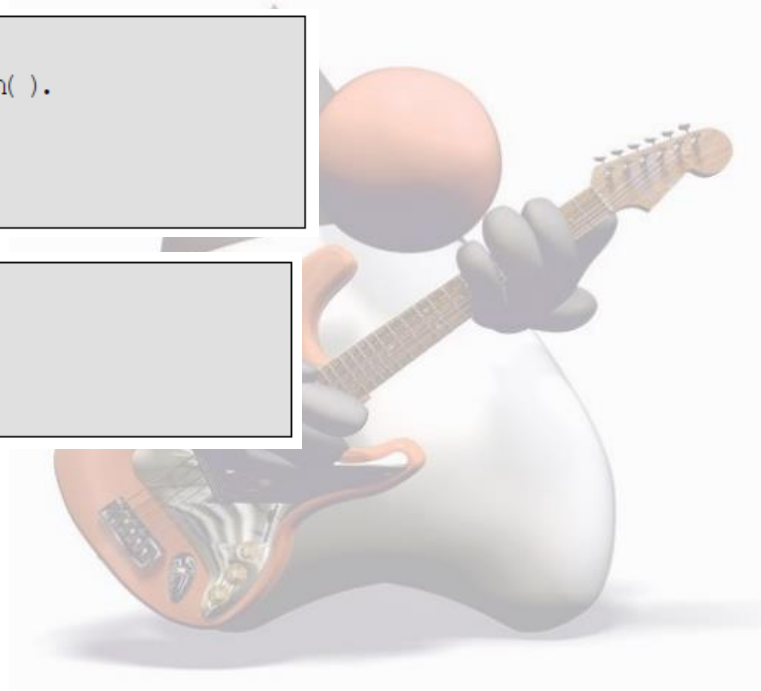
```
T reduce(T identity,
         BinaryOperator<T> accumulator)
```

Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. This is equivalent to:

```
T result = identity;
for (T element : this stream)
    result = accumulator.apply(result, element)
return result;
```

```
final String bornInJuly = persons.stream().
                              filter(person -> person.getBirthday().getMonth().
                                                      equals(Month.JULY)).
                              map(Person::getName).
                              reduce("", stringCombiner);
```

```
final BinaryOperator<String> stringCombiner = (str1, str2) ->
{
    return str1.isEmpty() ? str2 : str1 + ", " + str2;
};
```



Filter-Map-Reduce

Hinweis: Die Methode `String.join()`

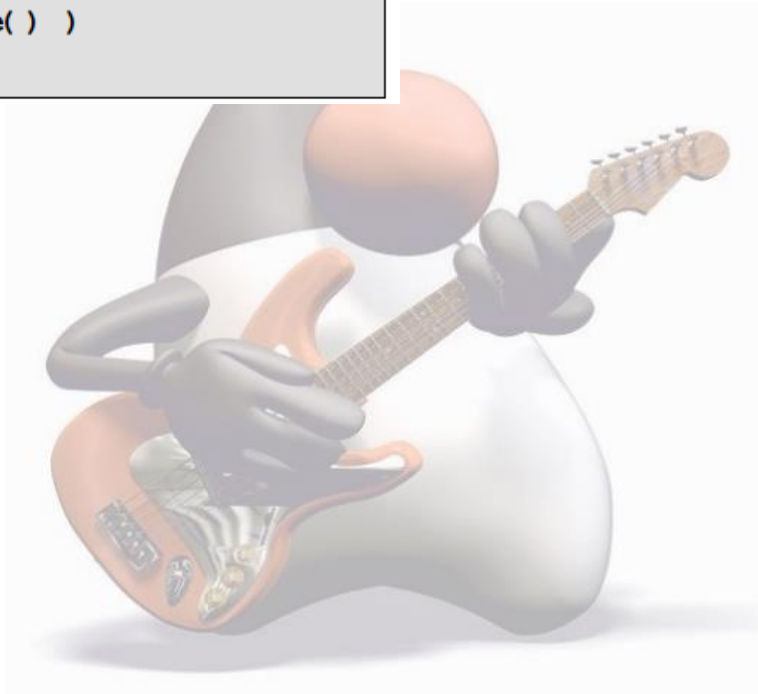
Wenn tatsächlich nur textuelle Werte miteinander verknüpft werden sollen, dann kann man die in der Klasse `String` mit JDK 8 neu eingeführte Methode `join(CharSequence delimiter, CharSequence... elements)` nutzen:[a]

```
final String stringConcat = String.join(", ", names);
```

Filter-Map-Reduce

Ein weiteres Beispiel:

```
// Explizite Umwandlung / Mapping mit toString
final String joined1 = persons.stream().mapToInt(Person::getAge)
                                        .mapToObj(Integer::toString)
                                        .collect(joining(", "));

// Implizite Umwandlung / Mapping durch "" + value
final String joined2 = persons.stream().map( person -> "" + person.getAge() )
                                        .collect(joining(", "));
```



Übung 10

- Erstellen Sie eine Liste mit 100.000 Integer Zufallszahlen im Wertebereich zwischen 1 und 1000.
- Erstellen Sie einen String mit einer Komma separierten Liste, die jene Zahlen beinhaltet, die kleiner als 10 sind. Die Zahlen dürfen nicht doppelt vorkommen, sie sollen sortiert sein.
- Berechnen Sie während der Verarbeitung die Summe der gefilterten Zahlen.
- Implementieren Sie die gleiche Logik mit Schleifen und vergleichen Sie die Performance.

Agenda

- Collections & Lambdas
  - externe vs. Interne Iteration
  - Collections Erweiterungen
  - Streams
  - Filter-Map-Reduce
  - Fallstricke bei Lambdas

Imperative Lösung mit Streams

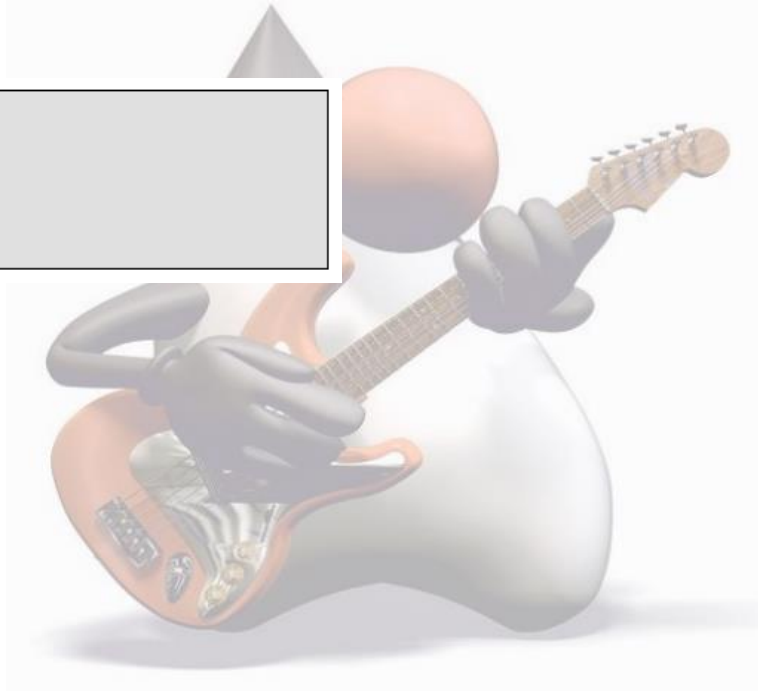
Imperative Lösung:

```
for ( int i = 0; i < persons. size( ); i++)
{
    final Person person = persons. get( i);
    if ( person. getName( ). equals( "TheOne"))
    {
        action( person);
    }
}
```

Schlechte (nachgebaute imperative) Lösung mit Streams:

```
IntStream. range( 0, persons. size( )).
           mapToObj( persons: : get).
           filter( person -> person. getName( ). equals( "TheOne")).
           forEach( person -> action( person));
```

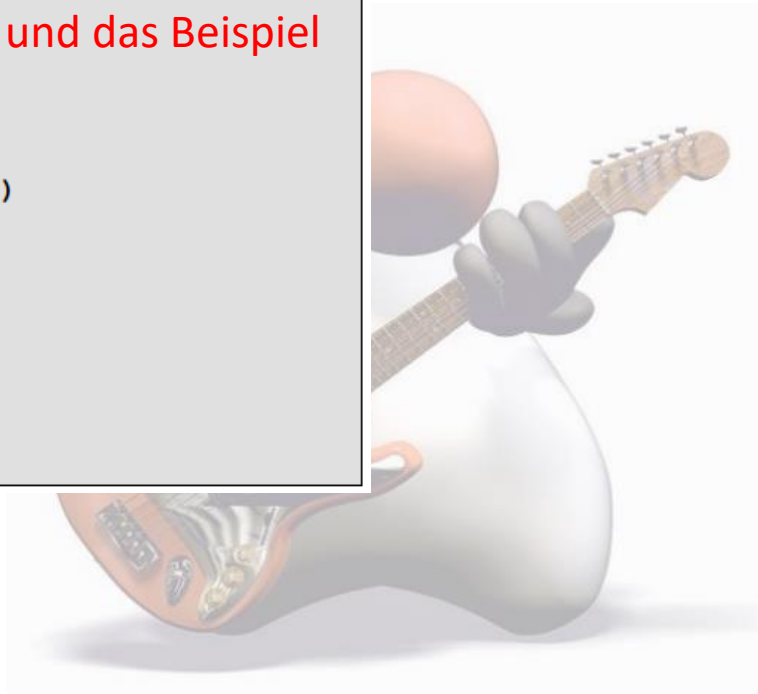
Übung: Wie sieht eine gute Umsetzung mit Streams aus?



Fallstrick Komplexität

```
final List<Person> persons = new ArrayList<>( );

// ...

// Mapping auf Alter
final Stream<Integer> agesStream = persons. stream( ).
                                   map( new Function<Person, Integer>( )
{
    @Override
    public Integer apply( final Person person)
    {
        return person. getAge( );
    }
});

// Durchschnittsberechnung
final int averageAge = agesStream. collect( Collectors.
                                averagingInt( new ToIntFunction<Double>( )
{
    @Override
    public int applyAsInt( final Double value)
    {
        return value. intValue( );
    }
});
```

Wo ist der Fehler?
Übung: Versuchen Sie die Fehlermeldung des Compilers zu verstehen und das Beispiel zu korrigieren.



Agenda

- Java Date & Time API
  - Datumsverarbeitung vor JSR-310
  - Überblick über die neu eingeführten Klassen

Alte Datumsverarbeitung

```
public static void main( final String[ ] args)
{
    // Geburtstag des Autors:  7. 2. 1971
    final int year = 1971;
    final int month = 2;
    final int day = 7;

    System. out. println( new Date( year, month, day));
    System. out. println( new Date( year - 1900, month - 1, day));
}
```

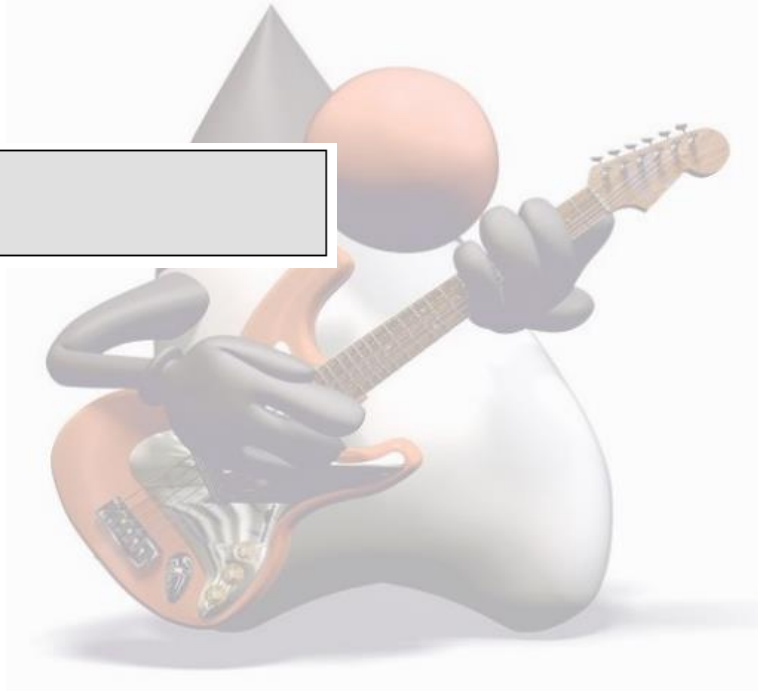
Was ist das Ergebnis?

Alte Datumsverarbeitung

```
public static void main( final String[ ] args)
{
    // Geburtstag des Autors:  7. 2. 1971
    final int year = 1971;
    final int month = 2;
    final int day = 7;

    System. out. println( new Date( year, month, day));
    System. out. println( new Date( year - 1900, month - 1, day));
}
```

Was ist das Ergebnis?

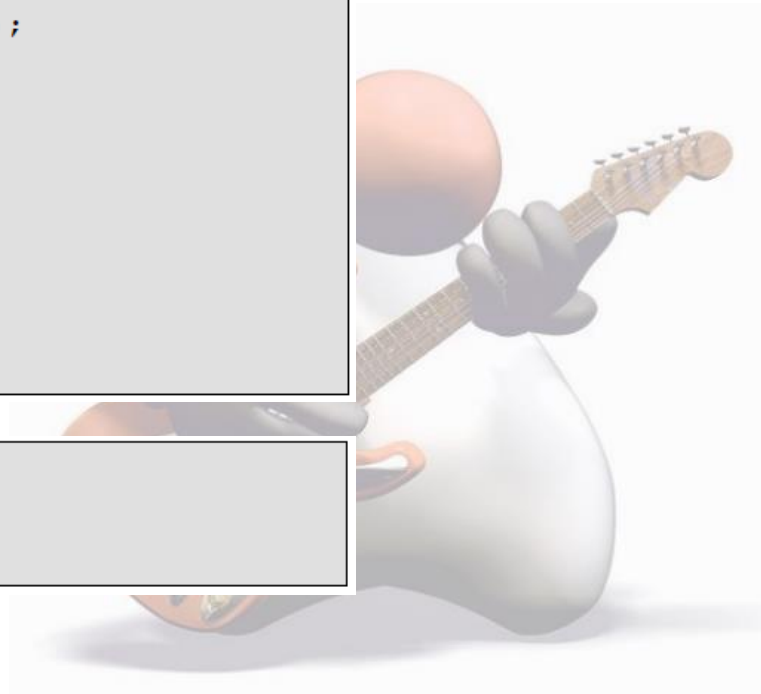
```
Tue Mar 07 00: 00: 00 CET 3871
Sun Feb 07 00: 00: 00 CET 1971
```



Alte Datumsverarbeitung

```
public static void main( final String[ ] args) throws ParseException
{
    // Unterschied 1 Stunde, 10 Minuten und 20 Sekunden
    final String startTimeAsString = "10: 10: 10";
    final String endTimeAsString = "11: 20: 30";

    // Umwandlung in Date-Objekte
    final SimpleDateFormat dateFormat = new SimpleDateFormat( "HH: mm: ss");
    final Date startTime = dateFormat. parse( startTimeAsString);
    final Date endTime = dateFormat. parse( endTimeAsString);
    // Berechne Differenz basierend auf Millisekunden
    final long durationInMs = endTime. getTime( ) - startTime. getTime( );
    System. out. println( "duration in seconds = " + TimeUnit. MILLISECONDS.
                                                    toSeconds( durationInMs));

    final String duration1 = dateFormat. format( new Date( durationInMs));
    System. out. println( "duration 1 = " + duration1);

    // DateFormat muss Zeitzone gesetzt bekommen
    dateFormat. setTimeZone( TimeZone. getTimeZone( "GMT"));
    final String duration2 = dateFormat. format( new Date( durationInMs));
    System. out. println( "duration 2 = " + duration2);
}
```

```
duration in seconds = 4220
duration 1 = 02: 10: 20
duration 2 = 01: 10: 20
```



Agenda

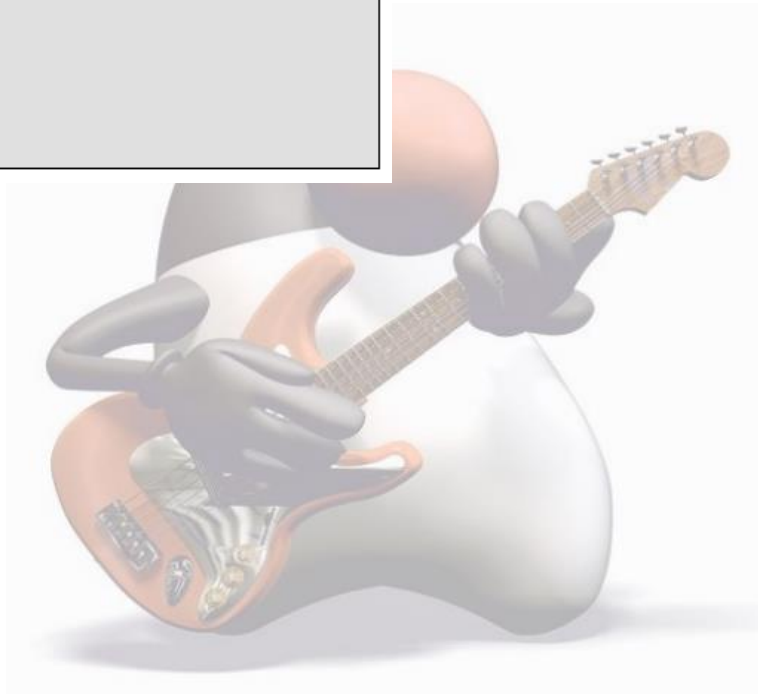
- Java Date & Time API
  - Datumsverarbeitung vor JSR-310
  - Überblick über die neu eingeführten Klassen

java.time.Instant

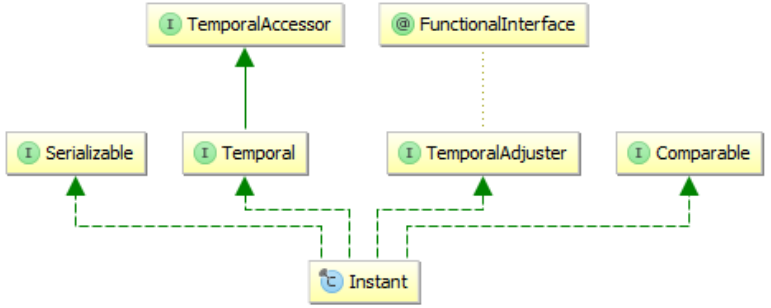
- Vergleichbar mit der Klasse Date
- Nanosekunden seit 01.01.1970 00:00:00 UTC
- Zeitstempel in der Zeitzone UTC
- Zeitstempel intern gespeichert als Sekunden (long) + Nanosekunden (int)

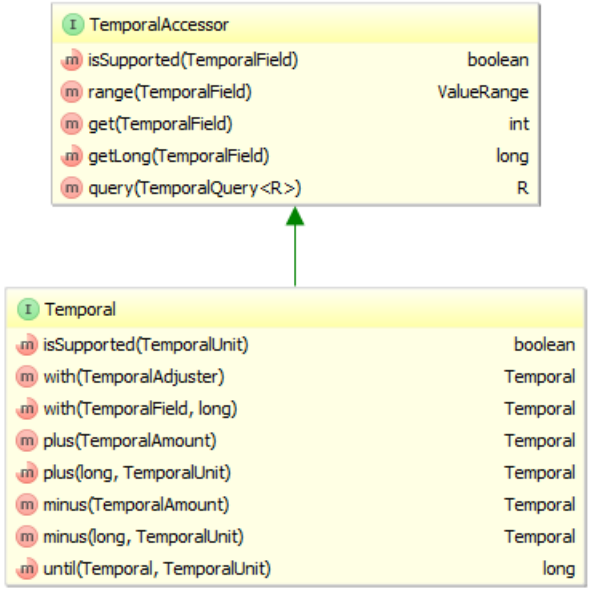
java.time.Instant

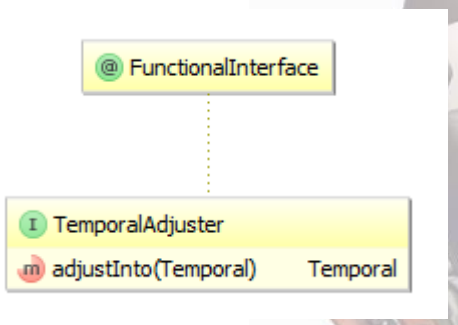
```
public static void main( final String[ ] args)
{
   // Abfahrt jetzt und Reisedauer 5 Stunden
   final Instant departureTime = Instant. now( );
   final Instant expectedArrivalTime = departureTime. plus( 5, ChronoUnit. HOURS);

   // Verspätung von 7 Minuten auf zwei Arten berechnen
   final Instant realArrival = expectedArrivalTime. plus( 7, ChronoUnit. MINUTES);
   final Instant realArrival2 = expectedArrivalTime. plus( Duration. ofMinutes( 7));

   System. out. println( departureTime);          // 2014-03-22T13: 54: 50. 818Z
   System. out. println( expectedArrivalTime);    // 2014-03-22T18: 54: 50. 818Z
   System. out. println( realArrival);            // 2014-03-22T19: 01: 50. 818Z
   System. out. println( realArrival2);           // 2014-03-22T19: 01: 50. 818Z
}
```



java.time.Instant



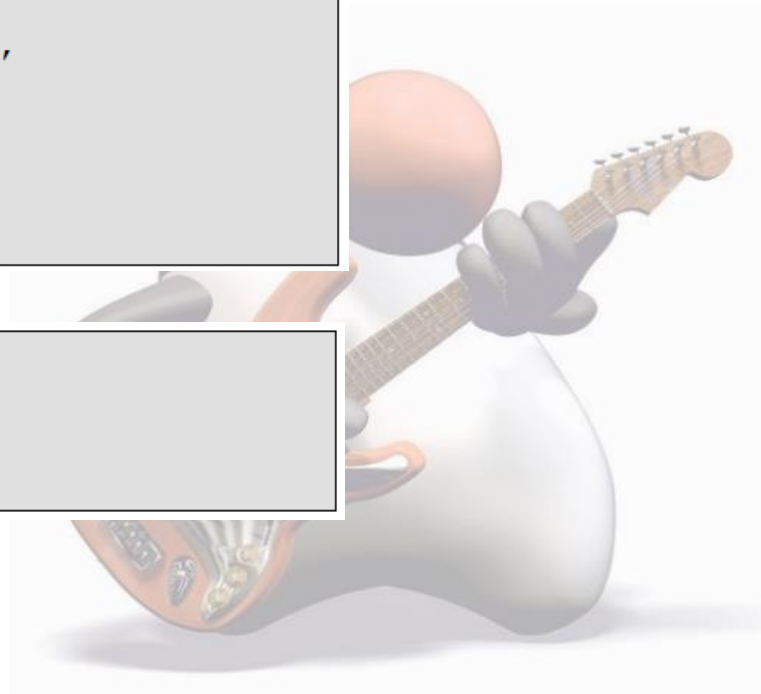




java.time.Instant

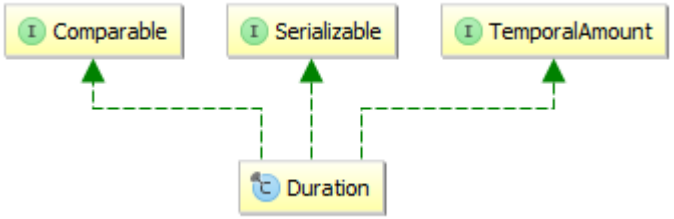
```
public static void main( final String[ ] args)
{
    // Abfahrt jetzt und Reisedauer 5 Stunden
    final Instant departureTime = Instant. now( );
    final Instant arrivalTime = departureTime. plus( 5, ChronoUnit. HOURS);

    System. out. println( "departure now:  " + departureTime);
    System. out. println( "arrival now + 5h:  " + arrivalTime);

    // Berechnungen durchführen: Differenz bilden
    final long inBetweenHours = ChronoUnit. HOURS. between( departureTime,
                                                            arrivalTime);
    final long inBetweenMinutes = ChronoUnit. MINUTES. between( departureTime,
                                                                arrivalTime);

    System. out. println( "inBetweenHours:    " + inBetweenHours);
    System. out. println( "inBetweenMinutes:  " + inBetweenMinutes);
}
```

```
departure now:     2014-02-19T22: 13: 50. 691Z
arrival now + 5h:  2014-02-20T03: 13: 50. 691Z
inBetweenHours:    5
inBetweenMinutes:  300
```

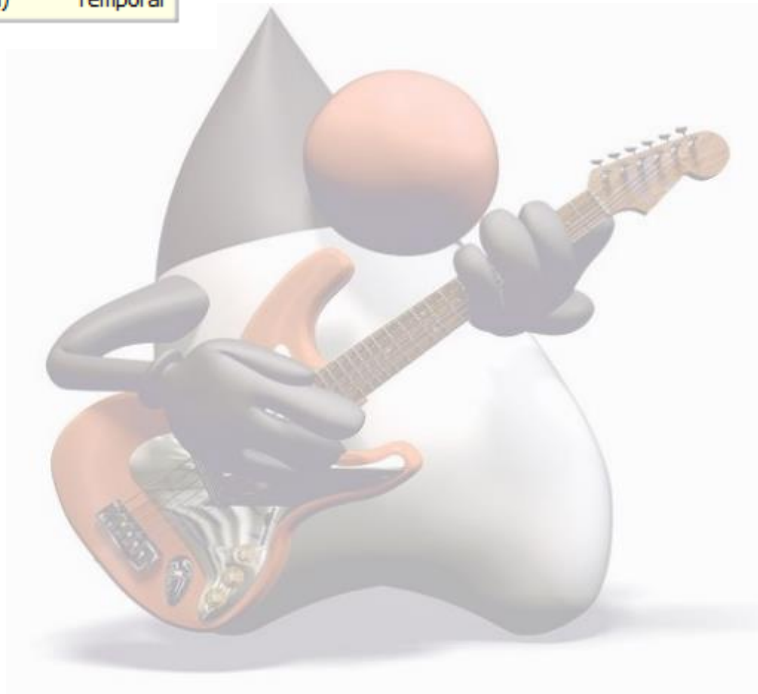


java.time.Duration

- Repräsentiert eine Zeitdauer

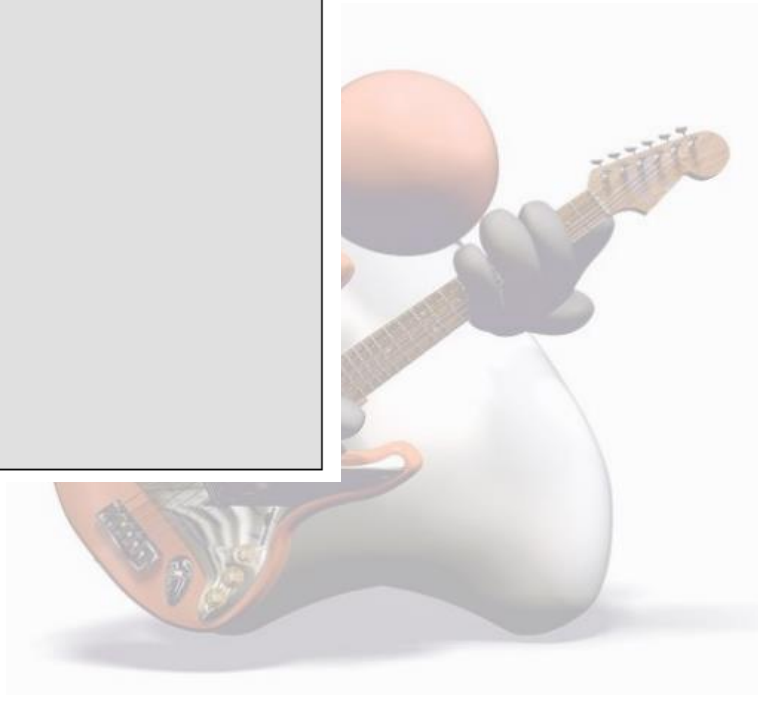


TemporalAmount	
get(TemporalUnit)	long
getUnits()	List<TemporalUnit>
addTo(Temporal)	Temporal
subtractFrom(Temporal)	Temporal



java.time.Duration

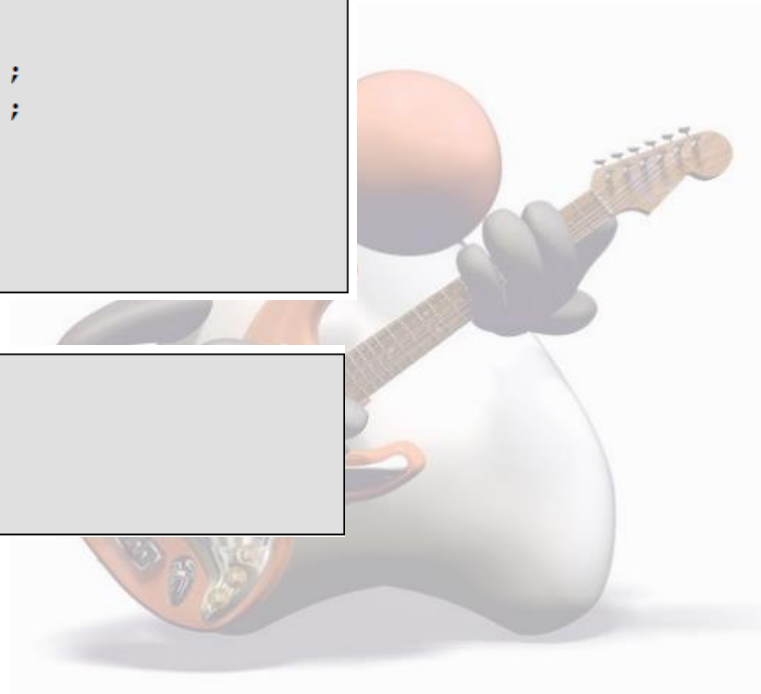
```
public static void main( final String[ ] args)
{
    // Erzeugung
    final Duration durationFromSecs = Duration. ofSeconds( 15);
    final Duration durationFromMinutes = Duration. ofMinutes( 30);
    final Duration durationFromHours = Duration. ofHours( 45);
    final Duration durationFromDays = Duration. ofDays( 60);

    System. out. println( "From Secs:     " + durationFromSecs);
    System. out. println( "From Minutes:  " + durationFromMinutes);
    System. out. println( "From Hours:    " + durationFromHours);
    System. out. println( "From Days:     " + durationFromDays);

    // Berechnungen
    final Instant now = Instant. now( );
    final Instant silvester2013 = Instant. parse( "2013-12-31T00: 00: 00Z");
    final Instant myBirthday2015 = Instant. parse( "2015-02-07T00: 00: 00Z");
    final Duration duration1 = Duration. between( now, silvester2013);
    final Duration duration2 = Duration. between( now, myBirthday2015);

    System. out. println( now + " -- " + silvester2013 + ":  " + duration1);
    System. out. println( now + " -- " + myBirthday2015 + ":  " + duration2);
}
```



java.time.Duration

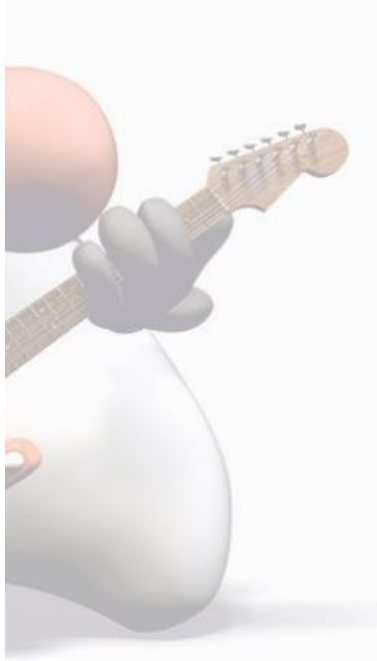
```
public static void main( final String[ ] args)
{
    // Erzeugung
    final Instant christmas2013 = Instant. parse( "2013-12-24T00: 00: 00Z");
    final Instant silvester2013 = Instant. parse( "2013-12-31T00: 00: 00Z");
    final Instant jdk8Release = Instant. parse( "2014-03-18T00: 00: 00Z");

    // Vergleichswerte errechnen
    System. out. println( "Christmas -> Silvester:      " +
                          Duration. between( christmas2013, silvester2013));
    System. out. println( "Silvester -> JDK 8 Release:  " +
                          Duration. between( silvester2013, jdk8Release));

    // Berechnungen
    final Instant calcSilvester_1 = christmas2013. plus( Duration. ofDays( 7));
    final Instant calcSilvester_2 = christmas2013. plus( 7, ChronoUnit. DAYS);

    System. out. println( calcSilvester_1);
    System. out. println( calcSilvester_2);
}
```

```
Christmas -> Silvester:      PT168H
Silvester -> JDK 8 Release:  PT1848H
2013-12-31T00: 00: 00Z
2013-12-31T00: 00: 00Z
```



```
java.time.LocalDate
java.time.LocalTime
java.time.LocalDateTime
```

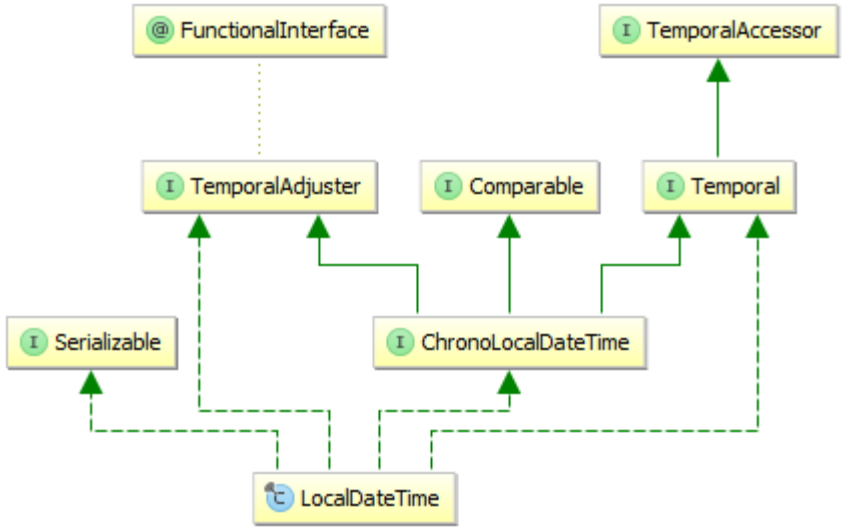
- LocalDate … Datum ohne Zeit
- LocalTime … Zeit ohne Datum
- Kombination aus Datum und Zeit
- LocalDateTime.now() bezieht sich auf die Zeitzone der Maschine
- Die Zeitzone wird im Objekt allerdings nicht gespeichert!
- Um LocalDateTime in Instant umzuwandeln wird zusätzlich die Zeitzone benötigt

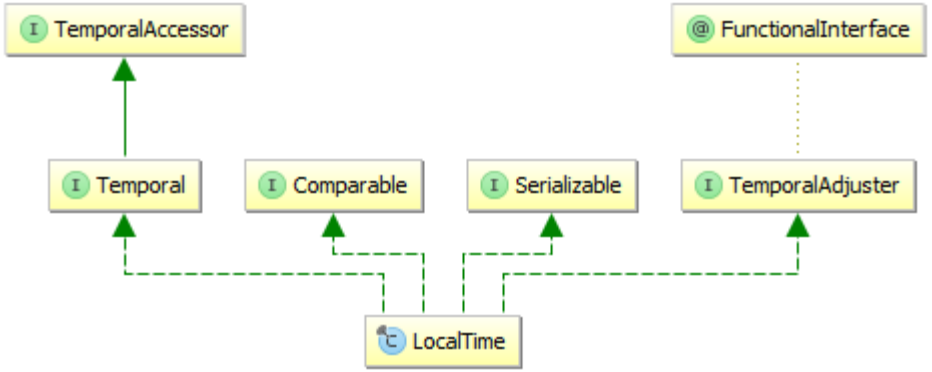


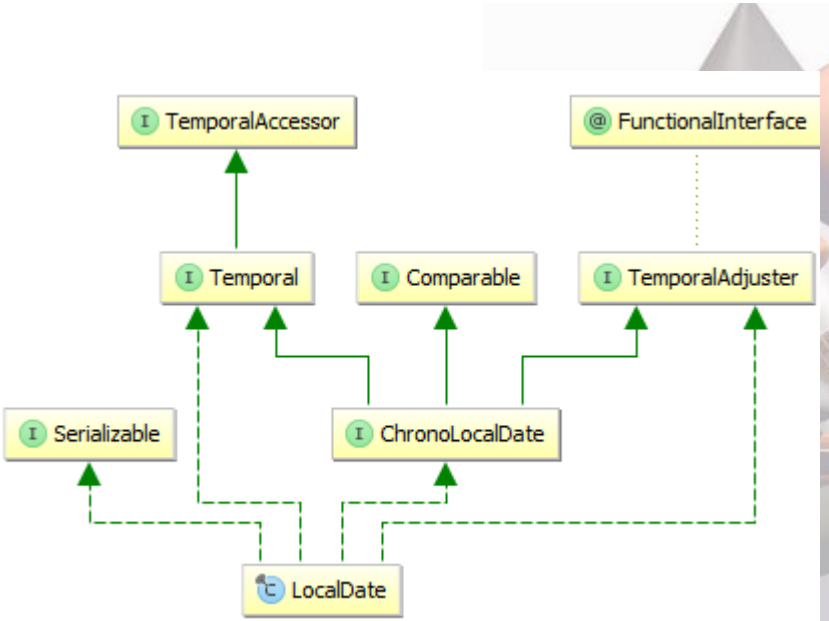
java.time.LocalDate
java.time.LocalTime
java.time.LocalDateTime

```
public class TestDate {
    public static void main(String[] args) {
        //Maschinenzeitzone: Österreich UTC+1 DTS
        Instant instant = Instant.now();
        System.out.println(instant); // -> 2015-05-07T12:21:20.153Z

        LocalDateTime ldt = LocalDateTime.now();
        System.out.println(ldt); //2015-05-07T14:21:20.226
    }
}
```

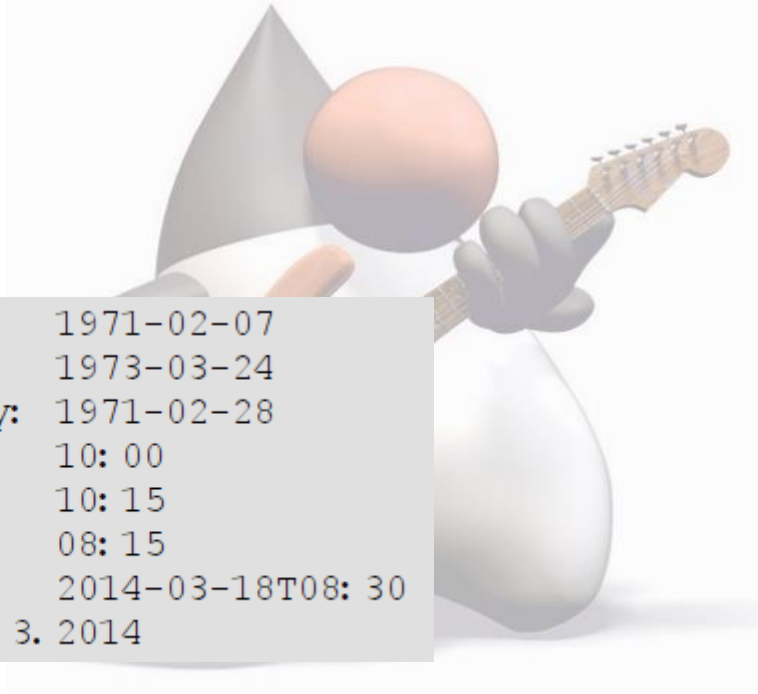
```
java.time.LocalDate
java.time.LocalTime
java.time.LocalDateTime
```







```java
public static void main( final String[ ] args)
{
    final LocalDate michasBirthday = LocalDate. of( 1971, Month. FEBRUARY, 7);
    final LocalDate barbarasBirthday = michasBirthday. plusYears( 2).
                                                       plusMonths( 1).
                                                       plusDays( 17);
    final LocalDate lastDayInFebruary = michasBirthday. with( TemporalAdjusters.
                                                              lastDayOfMonth( ));

    System. out. println( "michasBirthday:     " + michasBirthday);
    System. out. println( "barbarasBirthday:   " + barbarasBirthday);
    System. out. println( "lastDayInFebruary:  " + lastDayInFebruary);

    final LocalTime atTen = LocalTime. of( 10,00,00);
    final LocalTime tenFifteen = atTen. plusMinutes( 15);
    final LocalTime breakfastTime = tenFifteen. minusHours( 2);

    System. out. println( "atTen:              " + atTen);
    System. out. println( "tenFifteen:         " + tenFifteen);
    System. out. println( "breakfastTime:      " + breakfastTime);

    final LocalDateTime jdk8Release = LocalDateTime. of( 2014, 3, 18, 8, 30);
    System. out. println( "jdk8Release:        " + jdk8Release);
    System. out. printf( "jdk8Release:   %s. %s. %s\n", jdk8Release. getDayOfMonth( ),
                                                        jdk8Release. getMonthValue( ),
                                                        jdk8Release. getYear( ));
}
```



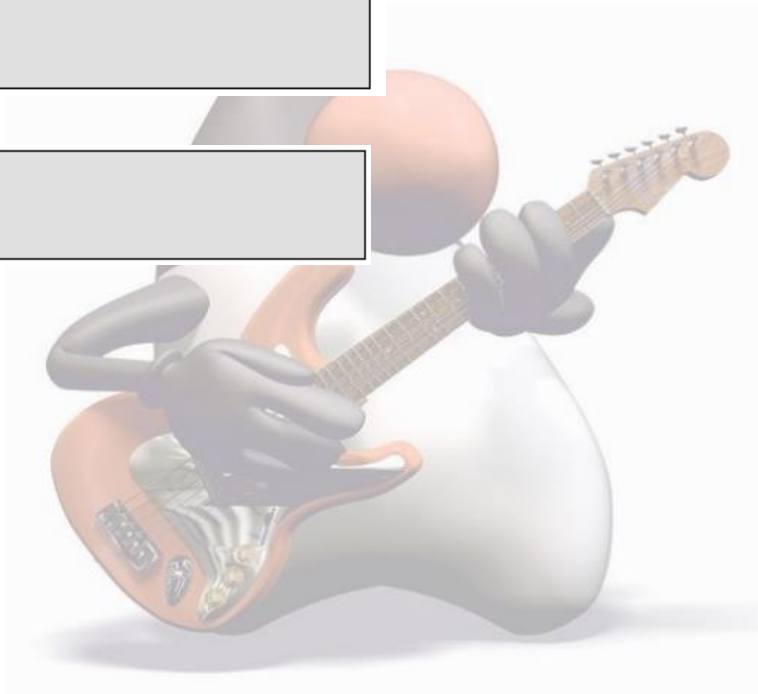
```
michasBirthday:     1971-02-07
barbarasBirthday:   1973-03-24
lastDayInFebruary:  1971-02-28
atTen:              10: 00
tenFifteen:         10: 15
breakfastTime:      08: 15
jdk8Release:        2014-03-18T08: 30
jdk8Release:   18. 3. 2014
```

java.time.DayOfWeek
java.time.Month

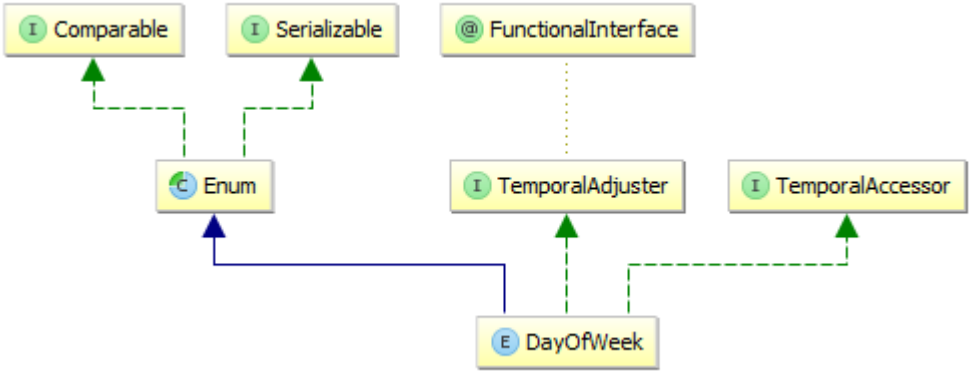
- Enums für Wochentage und Monate

```
public static void main( final String[ ] args)
{
    final DayOfWeek sunday = DayOfWeek. SUNDAY;
    final Month february = Month. FEBRUARY;

    System. out. println( sunday. plus( 5) );
    System. out. println( february. plus( 13) );
}
```

```
FRIDAY
MARCH
```



java.time.DayOfWeek
java.time.Month



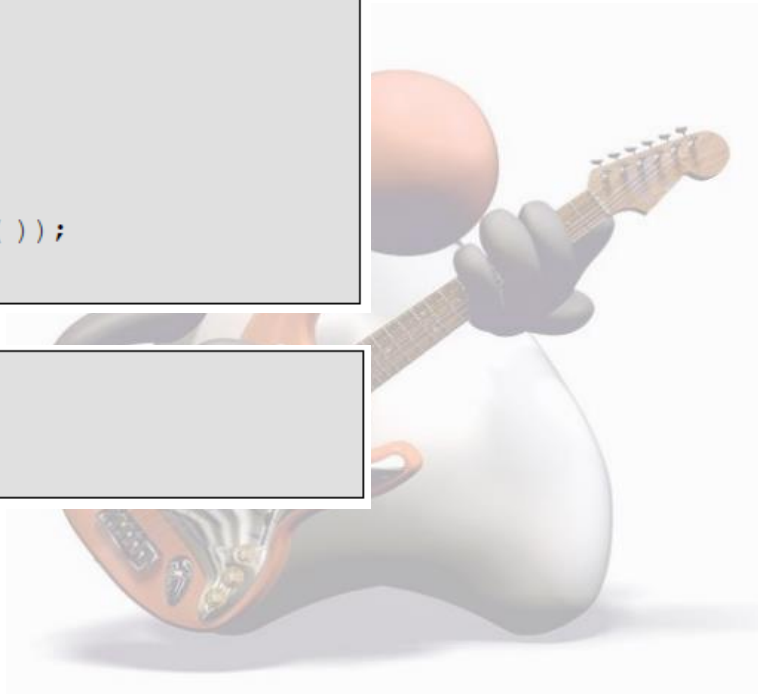
```
java.time.YearMonth
java.time.MonthDay
java.time.Year
```

- YearMonth … beinhaltet Jahr und Monat
- MonthDay … beinhaltet Monat und Tag
- Year … beinhaltet Jahr

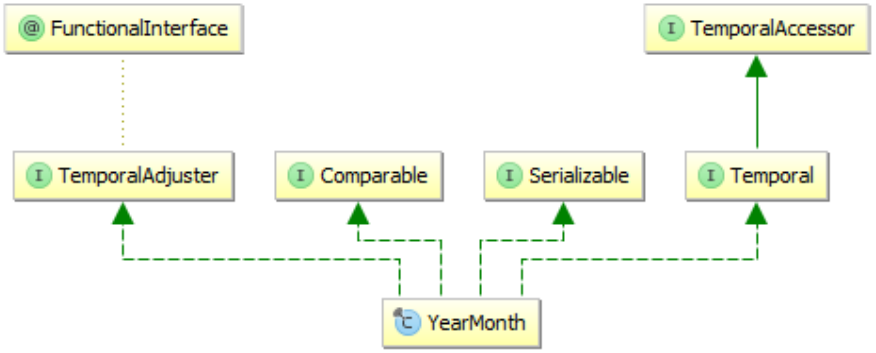
java.time.YearMonth
java.time.MonthDay
java.time.Year

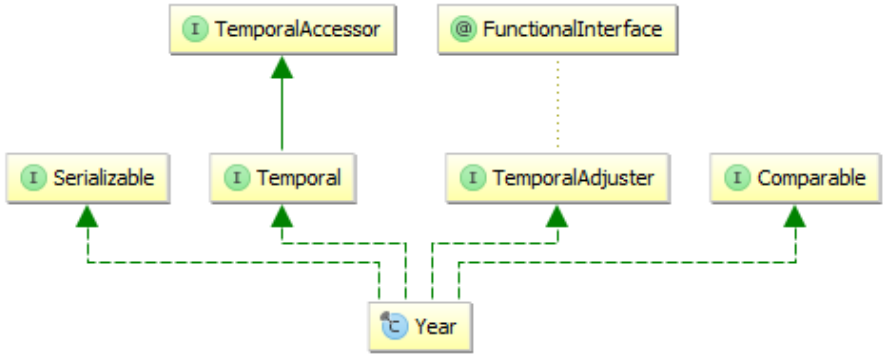
```
public static void main( final String[ ] args)
{
    // YearMonth: Demonstration jeweils mit und ohne Konstanten
    final YearMonth yearMonth = YearMonth. of( 2014, 2);
    final YearMonth february2014 = YearMonth. of( 2014, Month. FEBRUARY);

    // MonthDay: Achtung, ISO-Format mit der Reihenfolge: Monat, Tag
    final int dayOfBirthday = 7;
    final MonthDay monthDay1 = MonthDay. of( 2, dayOfBirthday);
    final MonthDay monthDay2 = MonthDay. of( Month. FEBRUARY, dayOfBirthday);

    // Year
    final Year year = Year. of( 2012);

    System. out. println( "YearMonth:  " + february2014);
    System. out. println( "MonthDay:   " + monthDay2);
    System. out. println( "Year:       " + year + " / isLeap? " + year. isLeap( ));
}
```

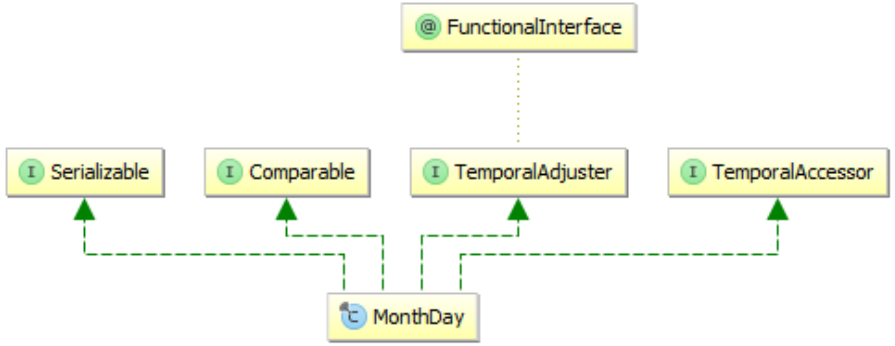
```
YearMonth:  2014-02
MonthDay:   --02-07
Year:       2012 / isLeap? true
```

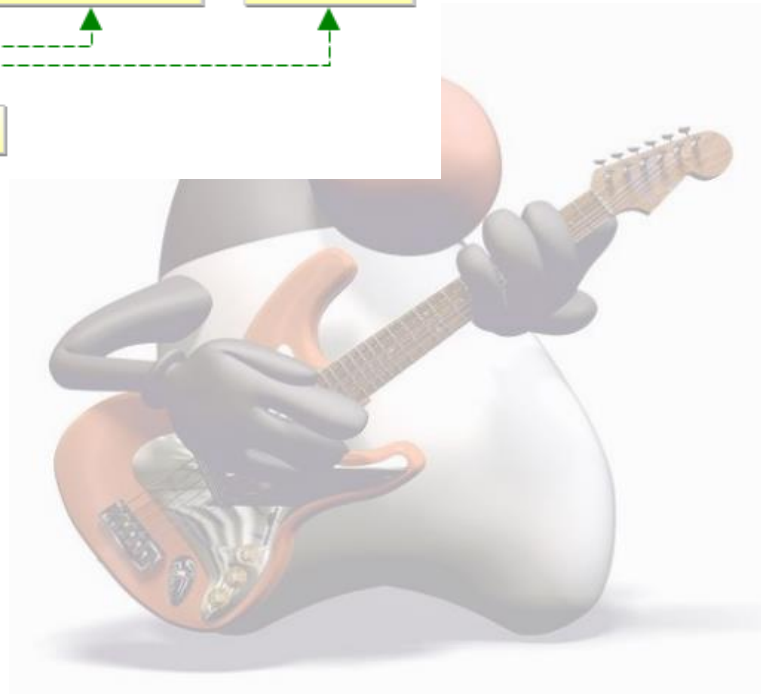


java.time.YearMonth
java.time.MonthDay
java.time.Year









java.time.Period

- Repräsentiert einen Zeitabschnitt
- ???
- Was ist der Unterschied zu java.time.Duration?

Hintergrundwissen: Warum gibt es `Duration` und `Period`?

Zunächst verwundert die Definition von Zeitabschnitten durch zweierlei Klassen – doch die Intention der beiden Klassen ist unterschiedlich. Während `Duration` einen Zeitabschnitt in Form von Sekunden modelliert, ist die Klasse `Period` eher zur Modellierung von Zeitabschnitten im Bereich von Tagen, Monaten oder gar Jahren gedacht.

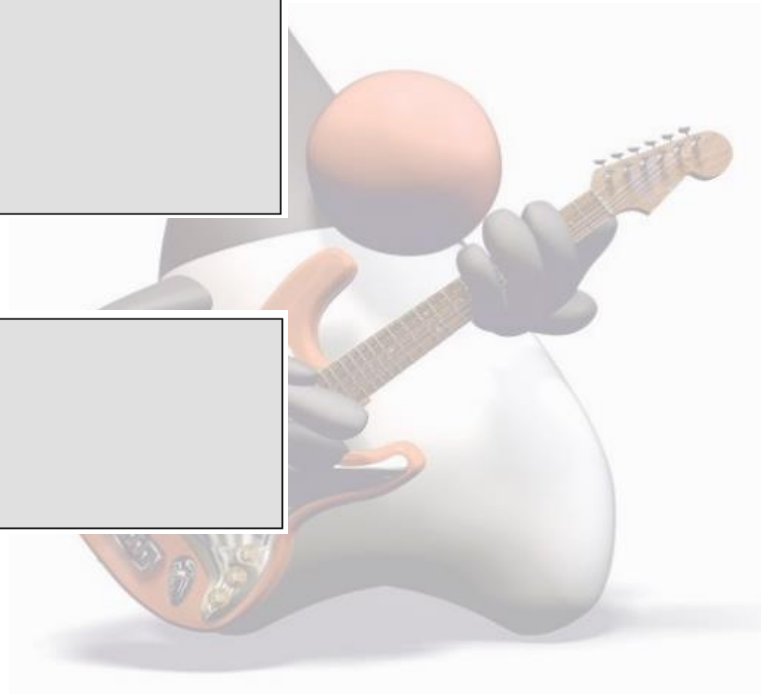
Aufgrund ihrer Ausrichtung auf Sekunden modelliert die Klasse `Duration` etwa einen Tag als exakt 24 Stunden, also 24 * 60 * 60 = 86.400 Sekunden. Die Klasse `Period` arbeitet dagegen auf eher konzeptioneller Ebene mit Tagen und Monaten unabhängig von der exakten Länge in Sekunden.

Den daraus entstehenden fundamentalen Unterschied zwischen beiden Modellierungen kann man sich am besten im Zusammenhang mit Winter- und Sommerzeit klarmachen: Es gibt Tage, die 23 Stunden lang sind, und solche, die eine Dauer von 25 Stunden besitzen. Wird die Länge eines Tags jedoch fix als 24 Stunden angenommen und dieser Wert wiederum in Form einer Zeitspanne in Sekunden repräsentiert, so kommt es zu Berechnungsfehlern, wenn an kürzeren oder längeren Tagen ein Tag in die Zukunft oder Vergangenheit »gesprungen« werden soll: Man bewegt sich somit entweder eine Stunde zu wenig oder zu viel in die Zukunft. Nutzt man die Klasse `Period`, muss man sich um diese Details nicht kümmern, da das »Konzept Tag« und nicht dessen Pendant in Sekunden (eine simple ganze Zahl) zum Einsatz kommt.

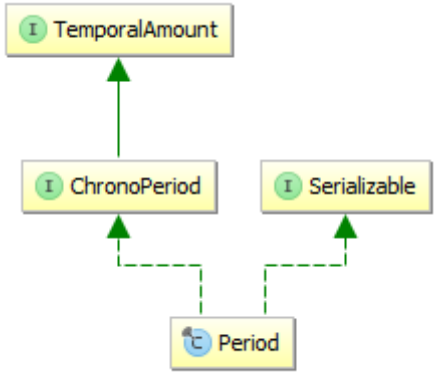
java.time.Period

```
public static void main( final String[ ] args)
{
    final LocalDateTime start = LocalDateTime. of( 1971, 2, 7, 10, 11);

    final Period thirtyOneDays = Period. ofDays( 31);
    System. out. println( "7. 2. 1971 + 31 Tage:  " + start. plus( thirtyOneDays));

    final Period oneMonth = Period. ofMonths( 1);
    System. out. println( "7. 2. 1971 + 1 Monat:  " + start. plus( oneMonth));

    final LocalTime now = LocalTime. now( );
    System. out. println( "now:              " + now);

    final LocalTime fiveMinutesLater = now. plus( 5, ChronoUnit. MINUTES);
    System. out. println( "now + 5 min:    " + fiveMinutesLater);

    final LocalTime sevenHoursLater = now. plusHours( 7);
    System. out. println( "now + 7 hours:  " + sevenHoursLater);
}
```

```
7. 2. 1971 + 31 Tage:  1971-03-10T10: 11
7. 2. 1971 + 1 Monat:  1971-03-07T10: 11
now:              20: 38: 29. 937
now + 5 min:    20: 43: 29. 937
now + 7 hours:  03: 38: 29. 937
```



java.time.Period



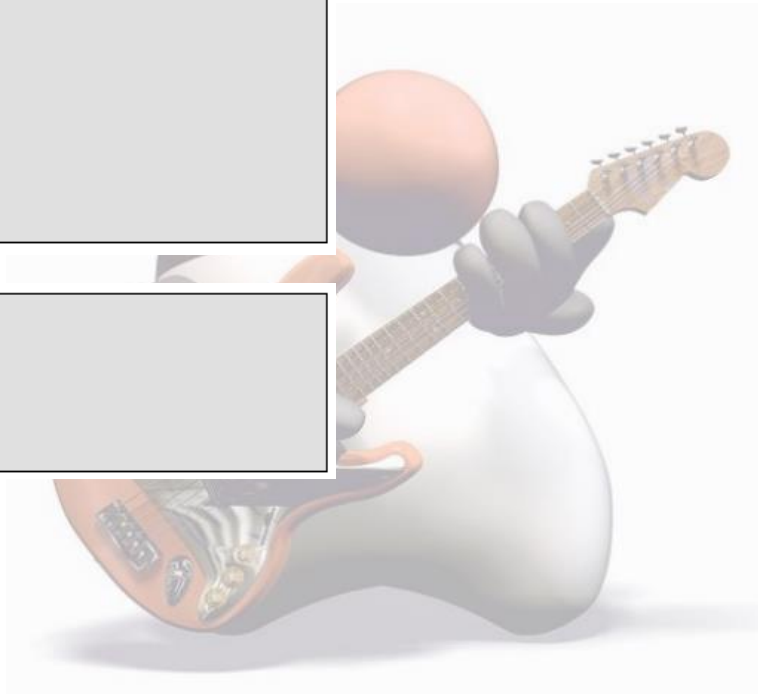
java.time.Clock

- Eine Clock Instanz bezieht sich auf eine Zeitzone
- Wird verwendet um Instant Objekte und die Zeit in Millisekunden in einer Zeitzone zu ermittlen.

java.time.Clock

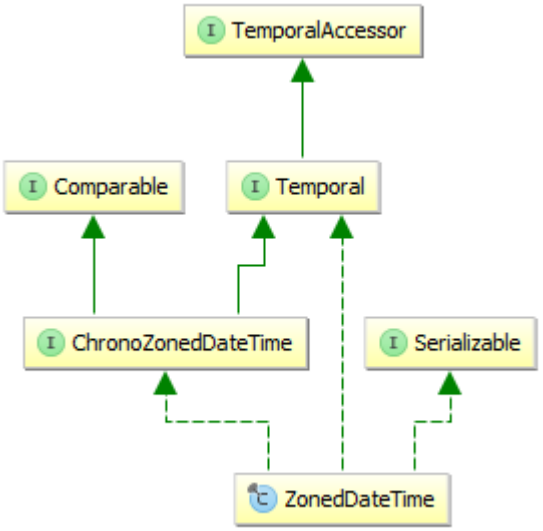
```
public static void main( final String[ ] args)
{
    // Basis UTC
    final Clock clockUTC = Clock. systemUTC( );
    System. out. println( clockUTC);
    printClockMillis( clockUTC);

    // Basis Default-Zeitzone
    final Clock clockDefaultZone = Clock. systemDefaultZone( );
    System. out. println( clockDefaultZone);
    printClockMillis( clockDefaultZone);
}

private static void printClockMillis( final Clock clock)
{
    final long currentTime = clock. millis( );
    System. out. println( currentTime);
}
```

```
SystemClock[ Z]
1395495448297
SystemClock[ Europe/Berlin]
1395495448365
```



java.time.ZonedDateTime

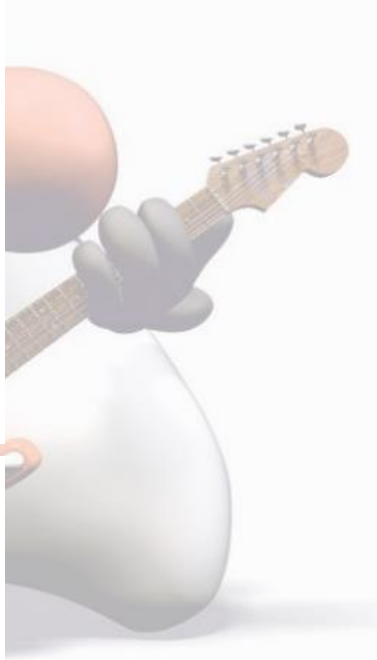
- Datum und Zeit in einer bestimmten Zeitzone



java.time.ZonedDateTime

```
public static void main( final String[ ] args)
{
    // Aktuelle Zeit als ZonedDateTime-Objekt ermitteln
    final ZonedDateTime now = ZonedDateTime. now( );

    // Die Uhrzeit ändern und in neuem Objekt speichern
    final ZonedDateTime nowButChangedTime = now. withHour( 11). withMinute( 44);

    // Neues Objekt mit verändertem Datum erzeugen
    final ZonedDateTime dateAndTime = nowButChangedTime. withYear( 2008).
                                                         withMonth( 9).
                                                         withDayOfMonth( 29);

    // Einsatz einer Monatskonstanten
    final ZonedDateTime dateAndTime2 = nowButChangedTime. withYear( 2008).
                                       withMonth( Month. SEPTEMBER. getValue( )).
                                       withDayOfMonth( 29);

    System. out. println( "now:            " + now);
    System. out. println( "-> 11: 44:      " + nowButChangedTime);
    System. out. println( "-> 29. 9. 2008: " + dateAndTime);
    System. out. println( "-> 29. 9. 2008: " + dateAndTime2);
}
```

```
now:            2014-03-20T23: 15: 01. 488+01: 00[ Europe/Berlin]
-> 11: 44:      2014-03-20T11: 44: 01. 488+01: 00[ Europe/Berlin]
-> 29. 9. 2008: 2008-09-29T11: 44: 01. 488+02: 00[ Europe/Berlin]
-> 29. 9. 2008: 2008-09-29T11: 44: 01. 488+02: 00[ Europe/Berlin]
```



`java.time.DateTimeFormatter`

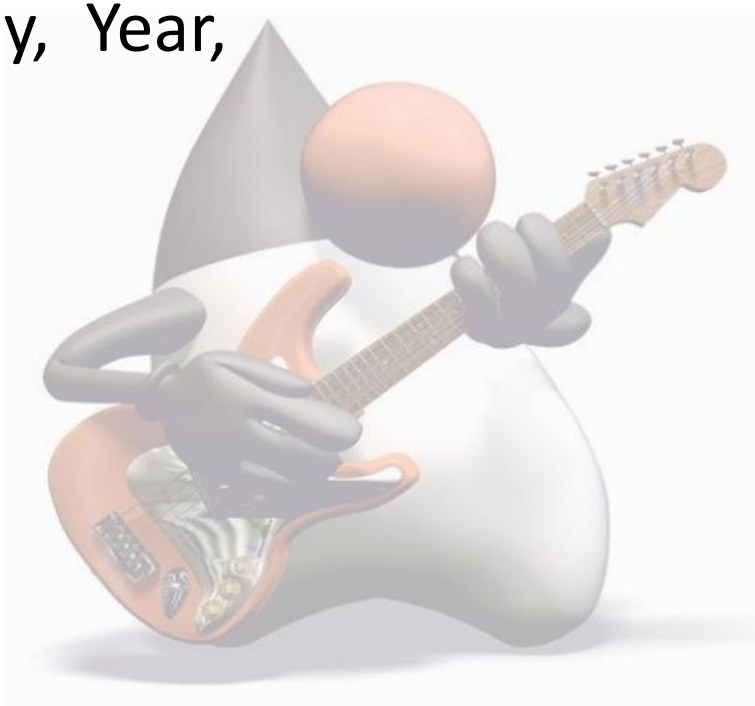
- Formatiert und parst `TemporalAccessors`
- TemporalAccessors sind: `DayOfWeek, Instant, LocalDate, LocalDateTime, LocalTime, Month, Year, YearMonth, ZonedTimeDate` …

java.time.DateTimeFormatter

- Viele TemporalAccessors implementieren die Methoden

`String`	`format(DateTimeFormatter formatter)` Formats this date-time using the specified formatter.
`static` Xyz	`parse(CharSequence text, DateTimeFormatter formatter)` Obtains an instance of Year from a text string using a specific formatter.

- z.B. LocalDate, LocalDateTime, LocalTime, MonthDay, Year, YearMonth, ZonedDateTime



java.time.DateTimeFormatter

```
String input = ...;
try {
    DateTimeFormatter formatter =
                        DateTimeFormatter.ofPattern("MMM d yyyy");
    LocalDate date = LocalDate.parse(input, formatter);
    System.out.printf("%s%n", date);
}
catch (DateTimeParseException exc) {
    System.out.printf("%s is not parsable!%n", input);
    throw exc;      // Rethrow the exception.
}
// 'date' has been successfully parsed
```

```java
public class Flight {
    public static void main(String[] args) {
        DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM d yyyy  hh:mm a");

        // Leaving from San Francisco on July 20, 2013, at 7:30 p.m.
        LocalDateTime leaving = LocalDateTime.of(2013, Month.JULY, 20, 19, 30);
        ZoneId leavingZone = ZoneId.of("America/Los_Angeles");
        ZonedDateTime departure = ZonedDateTime.of(leaving, leavingZone);

        try {
            String out1 = departure.format(format);
            System.out.printf("LEAVING:  %s (%s)%n", out1, leavingZone);
        } catch (DateTimeException exc) {
            System.out.printf("%s can't be formatted!%n", departure);
            throw exc;
        }

        // Flight is 10 hours and 50 minutes, or 650 minutes
        ZoneId arrivingZone = ZoneId.of("Asia/Tokyo");
        ZonedDateTime arrival = departure.withZoneSameInstant(arrivingZone)
                                         .plusMinutes(650);

        try {
            String out2 = arrival.format(format);
            System.out.printf("ARRIVING: %s (%s)%n", out2, arrivingZone);
        } catch (DateTimeException exc) {
            System.out.printf("%s can't be formatted!%n", arrival);
            throw exc;
        }

        if (arrivingZone.getRules().isDaylightSavings(arrival.toInstant()))
            System.out.printf("  (%s daylight saving time will be in effect.)%n",
                              arrivingZone);
        else
            System.out.printf("  (%s standard time will be in effect.)%n",
                              arrivingZone);
    }
}
```

```
LEAVING:  Jul 20 2013  07:30 PM (America/Los_Angeles)
ARRIVING: Jul 21 2013  10:20 PM (Asia/Tokyo)
```

Interoperabilität

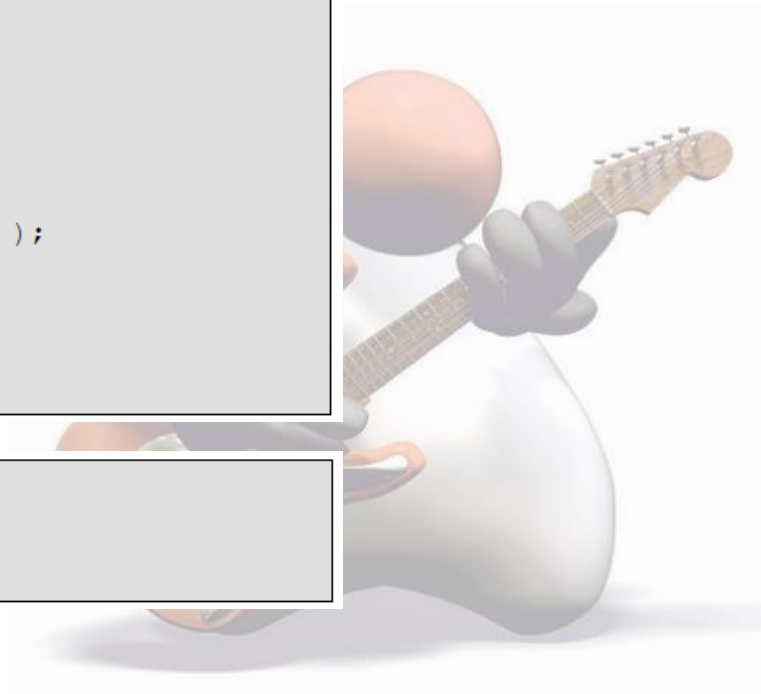
- Es wurden Konvertierungsmethoden eingeführt:

  - `Date. from( Instant)`
  - `Date. toInstant( )`
  - `Calendar. toInstant( )`
  - `GregorianCalendar. toZonedDateTime( )`
  - `GregorianCalendar. from( ZonedDateTime)`

Interoperabilität

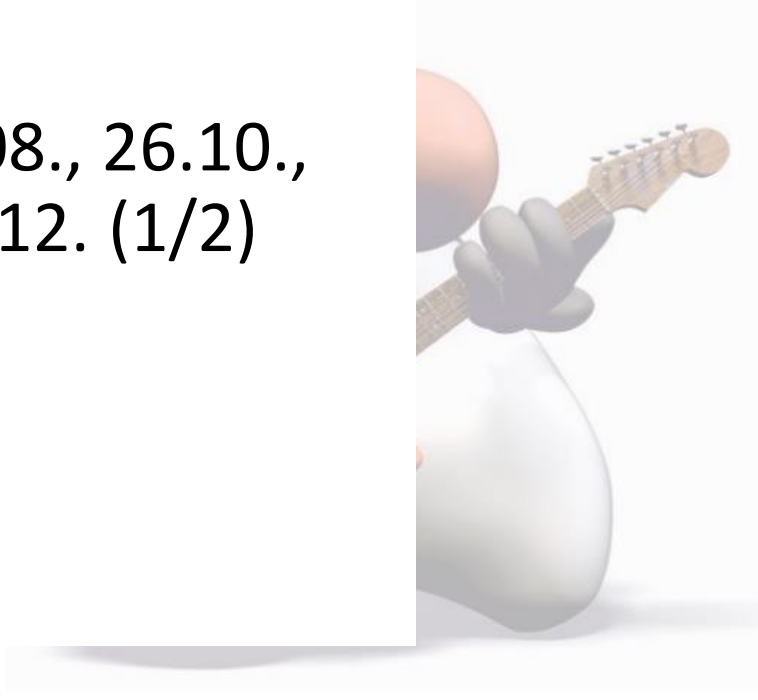
```
public static void main( final String[ ] args)
{
    // Berechnungen basierend auf Date
    final Date now = new Date( );
    final Instant nowAsInstant = now. toInstant( );

    final ZoneId systemZone = ZoneId. systemDefault( );
    final LocalDateTime localDateTime = LocalDateTime. ofInstant( nowAsInstant,
                                                                  systemZone);
    final ZoneId zoneCalifornia = ZoneId. of( "America/Los_Angeles");
    final ZonedDateTime zonedDateTime = ZonedDateTime. ofInstant( nowAsInstant,
                                                                  zoneCalifornia);

    System. out. println( "LocalDateTime:  " + localDateTime);
    System. out. println( "ZonedDateTime:  " + zonedDateTime);

    // Berechnungen basierend auf Calender
    final GregorianCalendar nowAsCalendar = new GregorianCalendar( );
    final ZonedDateTime nowAszonedDateTime = nowAsCalendar. toZonedDateTime( );

    final Instant instant = nowAszonedDateTime. toInstant( );
    System. out. println( "Instant:        " + instant);
}
```

```
LocalDateTime:  2014-05-22T21: 08: 36. 089
ZonedDateTime:  2014-05-22T12: 08: 36. 089-07: 00[ America/Los_Angeles]
Instant:        2014-05-22T19: 08: 36. 172Z
```



Übung 11

- Sie haben folgende Methode gegeben:
  ```
  public static LocalDate getOstersonntag(int jahr) ;
  ```
- Ostermontag: Ostersonntag + 1 Tag
- Christi Himmelfahrt: Ostersonntag + 39 Tage
- Pfingstmontag: Ostersonntag + 50 Tage
- Fronleichnam: Ostersonntag + 60 Tage
- Weitere Feiertage: 01.01., 06.01., 01.05., 15.08., 26.10., 01.11., 08.12., 24.12. (1/2), 25.12., 26.12.,31.12. (1/2)



Übung 11

Feiertage in Österreich:

- Implementieren Sie eine Methode: isFeiertag, die zu einem gegebenen Datum ermittelt, ob es sich um einen Feiertag handelt
- Implementieren Sie eine Methode: isWerktag, die zu einem gegebenen Datum ermittelt, ob es sich um einen Werktag, also keinen Feiertag, keinen Samstag oder Sonntag handelt.

Übung 11

Feiertage in Österreich:

- Schreiben Sie eine Methode, die alle Feiertage für das Jahr 2015 ausgibt.
- Schreiben Sie eine Methode, die die Anzahl der Werktage für das Jahr 2015 ausgibt.

Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

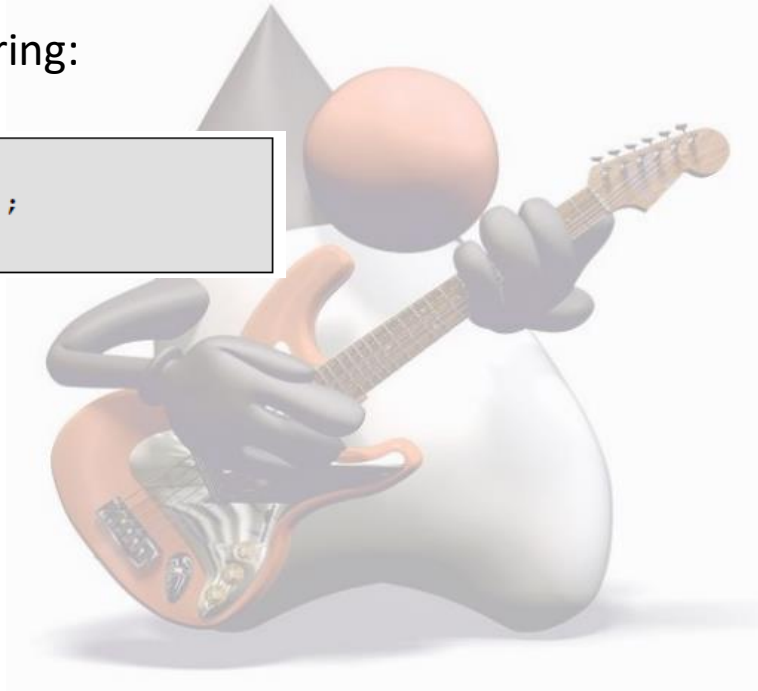
Erweiterungen Comparator<T>

Definition eines Comparators mittels einer Lambda Expression:

```
final Comparator<Person> compareByName = (person1, person2) ->
{
    return person1.getName().compareTo(person2.getName());
};
```

Definition eines Comparators mittels der Factory Methode comparing:

```
// Varianten mit Comparator.comparing
Comparator<Person> byName1 = Comparator.comparing(person -> person.getName());
Comparator<Person> byName2 = Comparator.comparing(Person::getName);
```



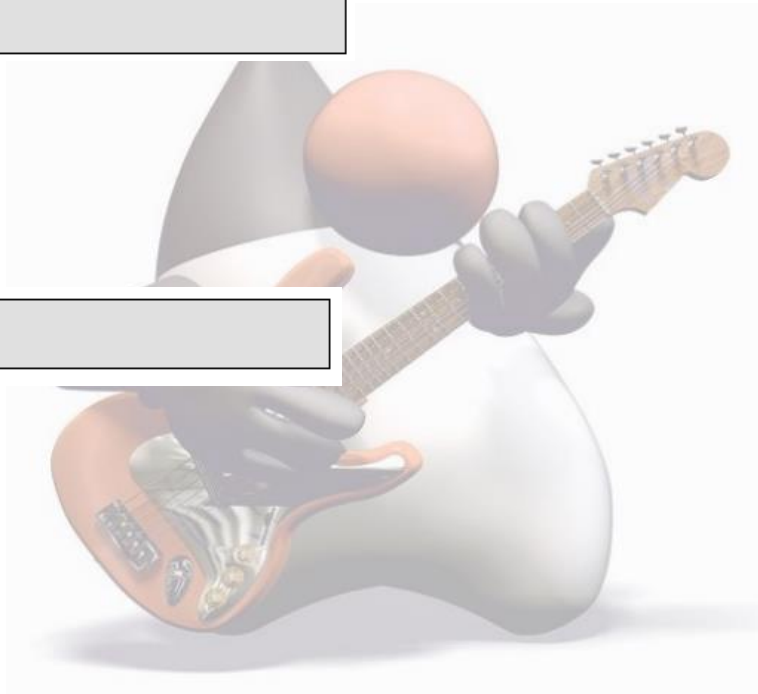
Erweiterungen Comparator<T>

Aneinanderreihung von Comperators:

```
// Komparatoren für ein spezielles Attribut
Comparator<Person> byFirstname = Comparator.comparing(Person::getFirstname);
Comparator<Person> byName = Comparator.comparing(Person::getName);
Comparator<Person> byAge = Comparator.comparing(Person::getAge);

// Kombination von Komparatoren
Comparator<Person> byNameAndFirstname = byName.
                                        thenComparing(byFirstname);
Comparator<Person> byNameAndAge = byName.thenComparing(byAge);
```

Verarbeitung primitiver Datentypen:

```
Comparator<Person> byAge = Comparator.comparingInt(Person::getAge);
```

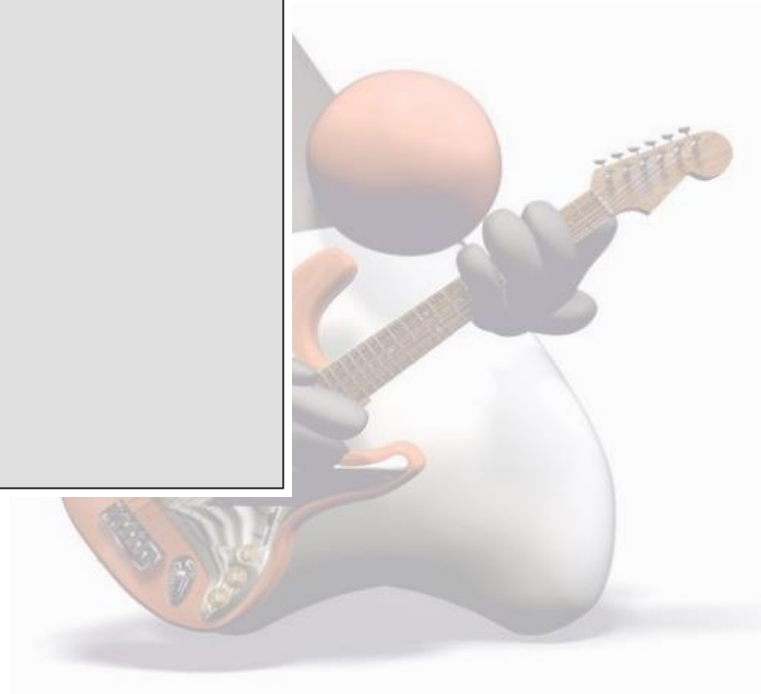


Erweiterungen Comparator<T>

Spezielle Ordnungen:

```
final Comparator<Person> byName = Comparator.comparing( Person::getName);
final Comparator<Person> byNameDescending = byName.reversed( );
```

```
public static void main( final String[ ] args)
{
    final Integer[ ] primes = { 1, 7, 3, 13, 11, 5, 17, 19 };

    // aufsteigend
    final Comparator<Integer> naturalOrder = Comparator.naturalOrder( );
    // absteigend
    final Comparator<Integer> reverseOrder = Comparator.reverseOrder( );
    // aufsteigend
    final Comparator<Integer> naturalOrderAgain = reverseOrder.reversed( );

    sortAndPrint( "naturalOrder     ", primes, naturalOrder);
    sortAndPrint( "reverseOrder     ", primes, reverseOrder);
    sortAndPrint( "naturalOrderAgain", primes, naturalOrderAgain);
}
private static void sortAndPrint( final String name, final Integer[ ] primes,
                                  final Comparator<Integer> sortOrder)
{
    Arrays.sort( primes, sortOrder);
    System.out.println( name + ": " + Arrays.toString( primes));
}
```



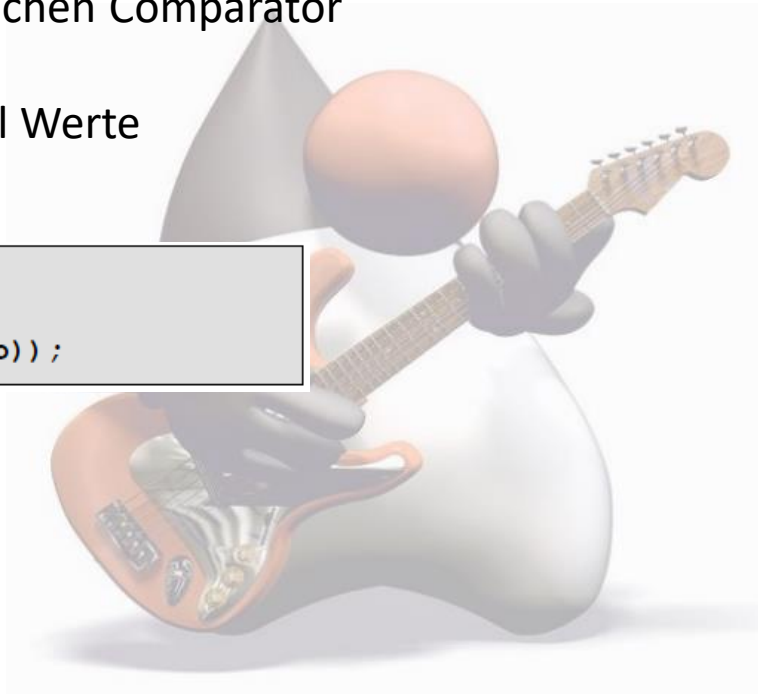
Erweiterungen Comparator<T>

Um einen Comparator robust zu machen, muss er normalerweise mit Null Werten umgehen können.
Der zuvor definierte Comparator würde eine NullPointerException auslösen, falls Person.getName() null liefert.

```
// Varianten mit Comparator.comparing
Comparator<Person> byName1 = Comparator.comparing( person -> person.getName());
Comparator<Person> byName2 = Comparator.comparing( Person::getName);
```

Um mit Nullwerten umgehen zu können, dekoriert man den eigentlichen Comparator mit einem Comparator, der die null values entsprechend behandelt.
Untenstehend eine Kombination eines Key-Extractors und eines Null Werte sortierenden Comparators:

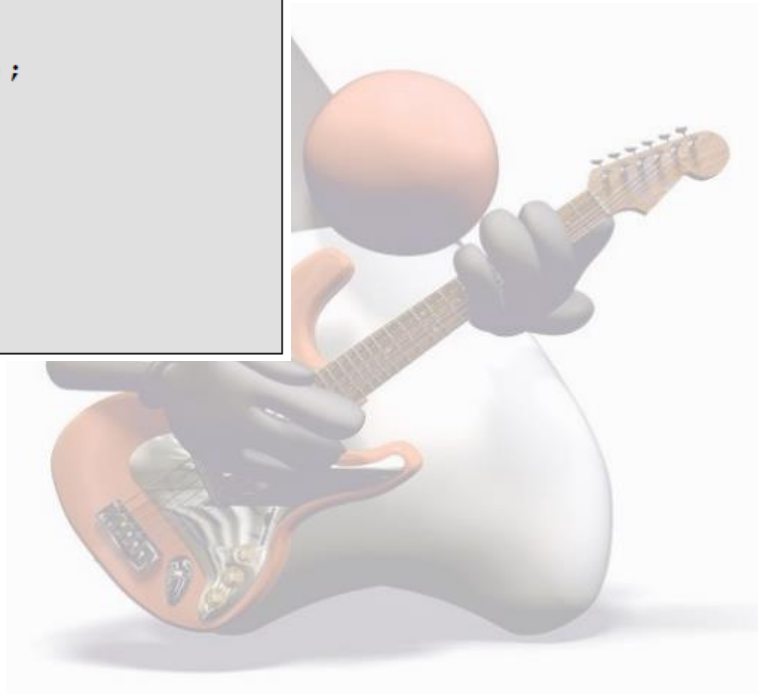
```
Comparator<Person> byFavoriteColor = Comparator.comparing(
                                     Person::getFavoriteColor,
                                     Comparator.nullsFirst( String::compareTo));
```



Erweiterungen Comparator<T>

Um mit Nullwerten umgehen zu können, dekoriert man den eigentlichen Comparator mit einem Comparator, der die null values entsprechend behandelt:

```
public static void main( final String[ ] args)
{
    final List<String> names = Arrays. asList( "A", null, "B", "C", null, "D");

    // Null-sichere Komparatoren zur Dekoration bestehender Komparatoren
    final Comparator<String> naturalOrder = Comparator. naturalOrder( );
    final Comparator<String> nullsFirst = Comparator. nullsFirst( naturalOrder);
    final Comparator<String> nullsLast = Comparator. nullsLast( naturalOrder);

    names. sort( nullsFirst);
    System. out. println( "nullsFirst:  " + names);
    names. sort( nullsLast);
    System. out. println( "nullsLast:  " + names);
}
```

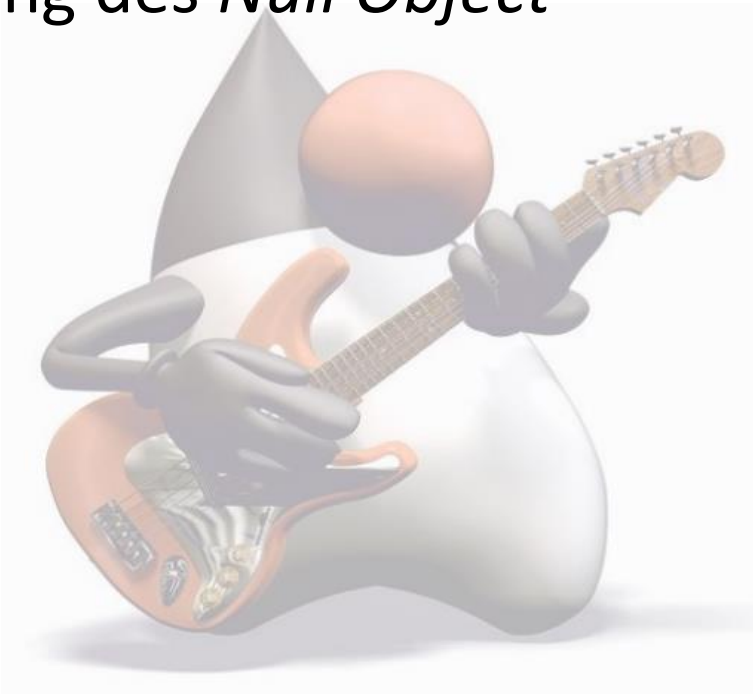


Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

Optional<T>

- Optional ist ein Wrapper um Rückgabewerte, der es erlaubt eine leere Rückgabe ohne Null Werte zu implementieren.
- Er dient damit auch der Vermeidung von NullPointerExceptions.
- Optional<T> ist gewissermaßen eine Implementierung des *Null Object Patterns*



Opt

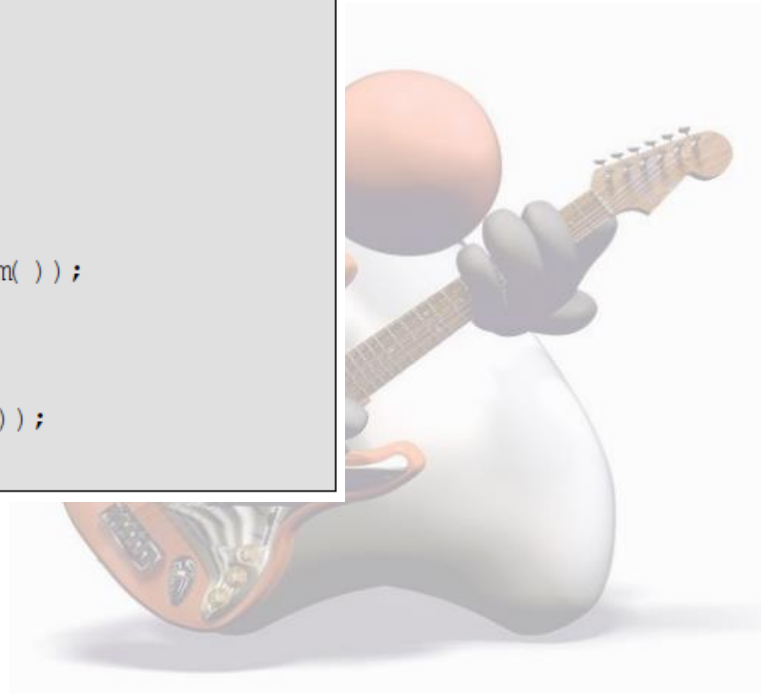
```
public static void main( final String[ ] args)
{
    final Integer[ ] sampleValues = {1,3,5,7,11,13,17,19};
    final Integer[ ] noValues = {};

    // Minimum und Maximum berechnen
    final Comparator<Integer> naturalOrder = Comparator. naturalOrder( );
    final Optional<Integer> max = Arrays. stream( sampleValues). max( naturalOrder);
    final Optional<Integer> min = Arrays. stream( noValues). min( naturalOrder);

    // Minimum und Maximum ausgeben
    System. out. println( "max:          " + max);
    System. out. println( "min:          " + min);

   // Prüfe, ob es einen Wert gibt
   System. out. println( "isPresent?:   " + min. isPresent( ));

   // Zugriff auf den Wert
   final Integer maxValue = max. get( );
   System. out. println( "maxValue:     " + maxValue);

   // Konstruktionsmethoden
   final Optional<Integer> optionalFromValue = Optional. of( 4711);
   final Optional<Double> optionalFromNull = Optional. ofNullable( null);
   System. out. println( "from Value:  " + optionalFromValue);
   System. out. println( "from null:   "+ optionalFromNull);
}
```

```
max:          Optional[ 19]
min:          Optional. empty
isPresent?:   false
maxValue:     19
from Value:   Optional[ 4711]
from null:    Optional. empty
```

Optional<T>

Behandlung von Alternativen mittels `Optional<T>`:

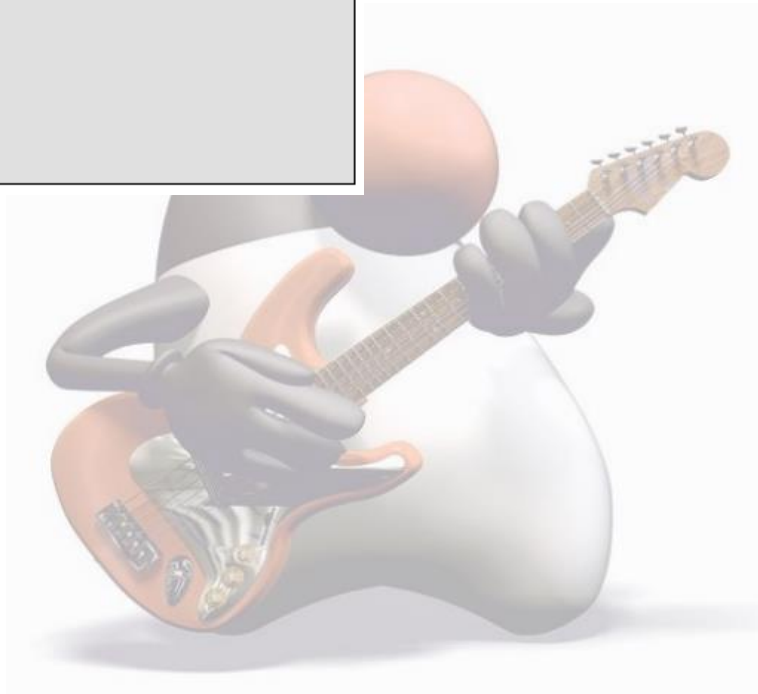
```
public static void main( final String[ ] args)
{
    final Integer[ ] noValues = {};

    final Optional<Integer> min = Arrays. stream( noValues).
                                          min( Comparator. naturalOrder( ));

    // Führe Aktion aus, wenn vorhanden
    min. ifPresent( System. out: : println);

    // Alternativen Wert liefern, wenn nicht vorhanden
    System. out. println( min. orElse( -1));

    // Berechne Ersatzwert, wenn nicht vorhanden
    final Supplier<Integer> randomGenerator = ( ) -> ( int)( 100 * Math. random( ));
    System. out. println( min. orElseGet( randomGenerator));

    // Löse eine Exception aus, wenn nicht vorhanden
    min. orElseThrow( ( ) -> new NoSuchElementException( "there is no minimum"));
}
```



Optional<T>

Verarbeitung von primitiven Werten:

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

    final int[ ] sampleValues = {1,3,5,7,11,13,17,19};

    final OptionalInt min = Arrays. stream( sampleValues). min( );
    final OptionalInt max = Arrays. stream( sampleValues). max( );
    final OptionalDouble avg = Arrays. stream( sampleValues). average( );

    System. out. println( "min:  " + min);
    System. out. println( "max:  " + max);
    System. out. println( "avg:  " + avg);
}
```



Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

Parallele Operationen auf Arrays

Paralleles sortieren:

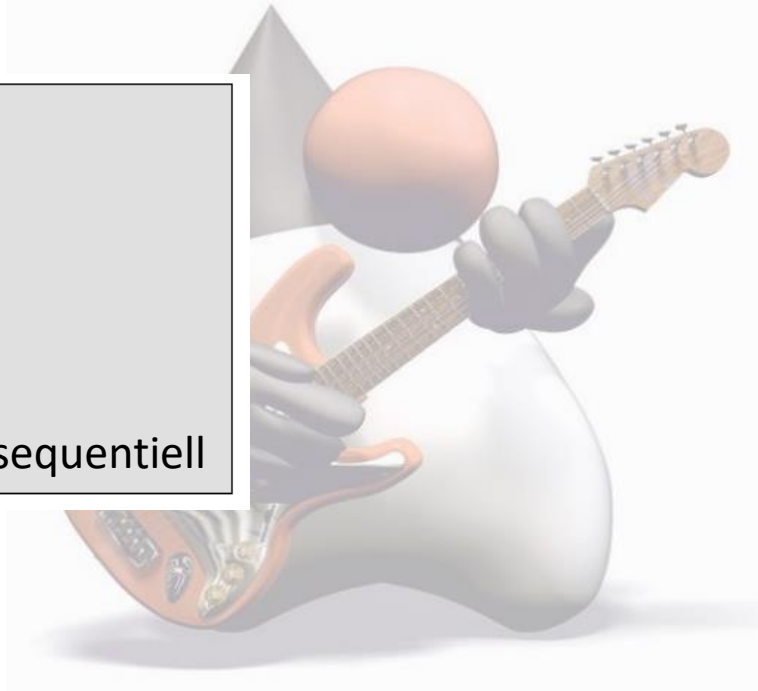
```
public static void main( final String[ ] args)
{
    final int[ ] numbers = { 1, 9, 2, 8, 7, 3, 5, 6, 4, 10 };

    System. out. println( "Initial:  " + Arrays. toString( numbers));
    Arrays. parallelSort( numbers);
    System. out. println( "Sorted:  " + Arrays. toString( numbers));
}
```

Vergleich Performance:

```
Current limit:          10000
parallel : sequential:  2. 138%

Current limit:          100000
parallel : sequential:  272%

Current limit:          1000000
parallel : sequential:  23%

Current limit:          10000000
parallel : sequential:  28%
```

Hotspot Compiler Effekt

Parallel ca. 4x schneller als sequentiell

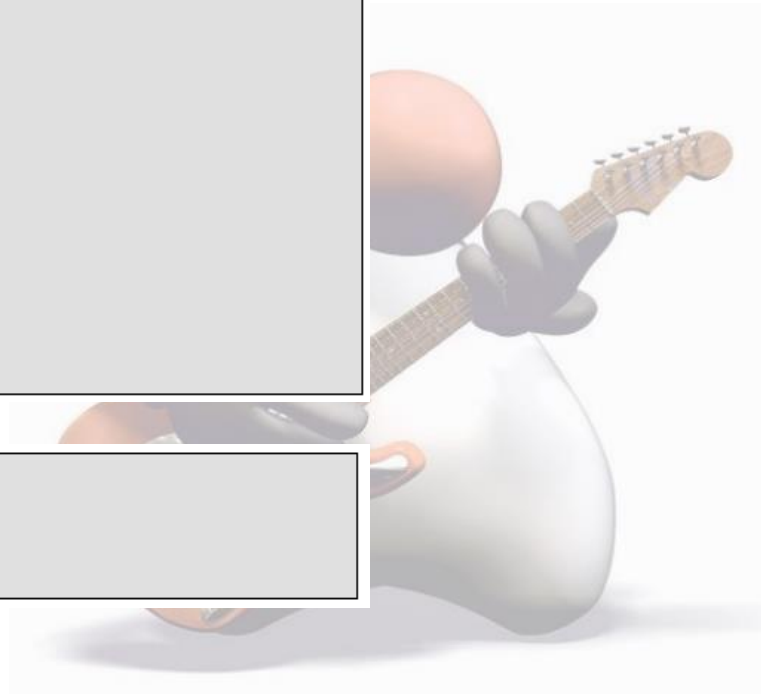


Parallele Operationen auf Arrays

Paralleles setzen von Werten:

```
public static void main( final String[ ] args)
{
    final int[ ] numbers = { 1, 3, 7, 15, 31, 63 };
    System. out. println( "Initial:  " + Arrays. toString( numbers));

    // Inkrement - Achtung hier wird der Index übergeben, nicht der Wert
    final IntUnaryOperator increment = i -> { return numbers[ i] + 1; };

    Arrays. parallelSetAll( numbers, increment);
    System. out. println( "Increment:  " + Arrays. toString( numbers));

    // Alle Werte < 10 durch 2 teilen, alle anderen mit 2 multiplizieren
    final IntUnaryOperator specialMapping = i ->
    {
        final int value = numbers[ i];
        return value < 10 ? value/2 :  value * 2;
    };

    Arrays. parallelSetAll( numbers, specialMapping);
    System. out. println( "Converted:  " + Arrays. toString( numbers));
}
```

```
Initial:    [1, 3, 7, 15, 31, 63]
Increment:  [2, 4, 8, 16, 32, 64]
Converted:  [1, 2, 4, 32, 64, 128]
```

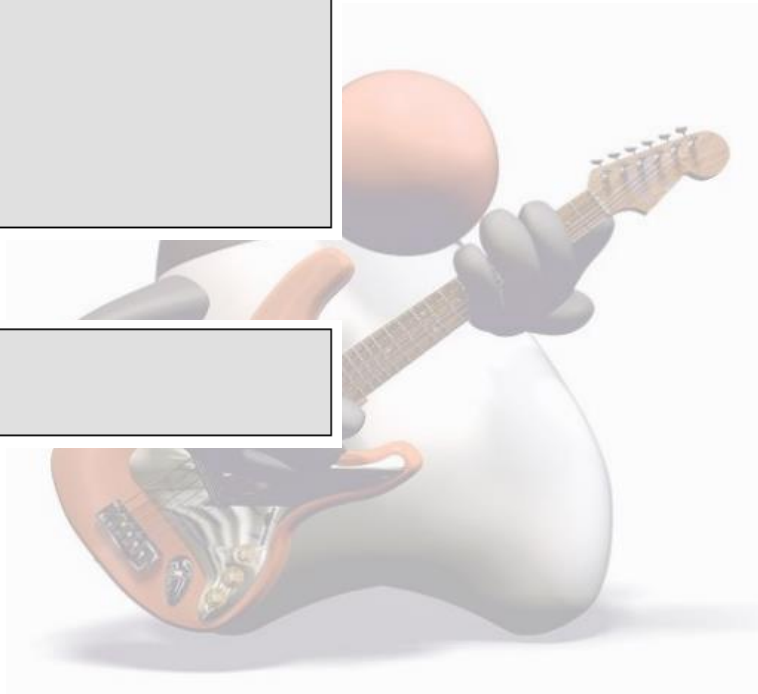


Parallele Operationen auf Arrays

Paralleles setzen von Werten, weiteres Beispiel:

```
public static void main( final String[ ] args)
{
    final String[ ] names = { "Andy", " Trim ", null, " Trim ", "Ralph" };
    System. out. println( "Initial:  " + Arrays. toString( names));

    // Spezielle Nachbearbeitung von Strings
    final IntFunction<? super String> trimAndMapNullToNA = i ->
    {
        final String value = names[ i];
        return value == null ? "-n/a-" :  value. trim( );
    };

    Arrays. parallelSetAll( names, trimAndMapNullToNA);
    System. out. println( "Converted:  " + Arrays. toString( names));
}
```

```
Initial:   [ Andy, Trim, null, Trim , Ralph]
Converted: [ Andy, Trim, -n/a-, Trim, Ralph]
```

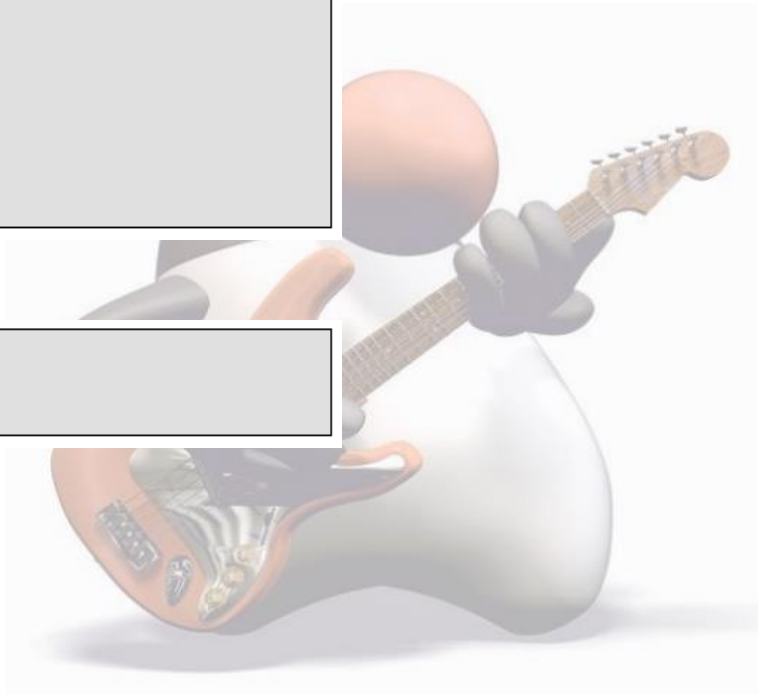


Parallele Operationen auf Arrays

Paralleles setzen von Werten, weiteres Beispiel:

```
public static void main( final String[ ] args)
{
    final String[ ] names = { "Andy", " Trim ", null, " Trim ", "Ralph" };
    System. out. println( "Initial:  " + Arrays. toString( names));

    // Spezielle Nachbearbeitung von Strings
    final IntFunction<? super String> trimAndMapNullToNA = i ->
    {
        final String value = names[ i];
        return value == null ? "-n/a-" :  value. trim( );
    };

    Arrays. parallelSetAll( names, trimAndMapNullToNA);
    System. out. println( "Converted:  " + Arrays. toString( names));
}
```

```
Initial:   [ Andy, Trim, null, Trim , Ralph]
Converted: [ Andy, Trim, -n/a-, Trim, Ralph]
```

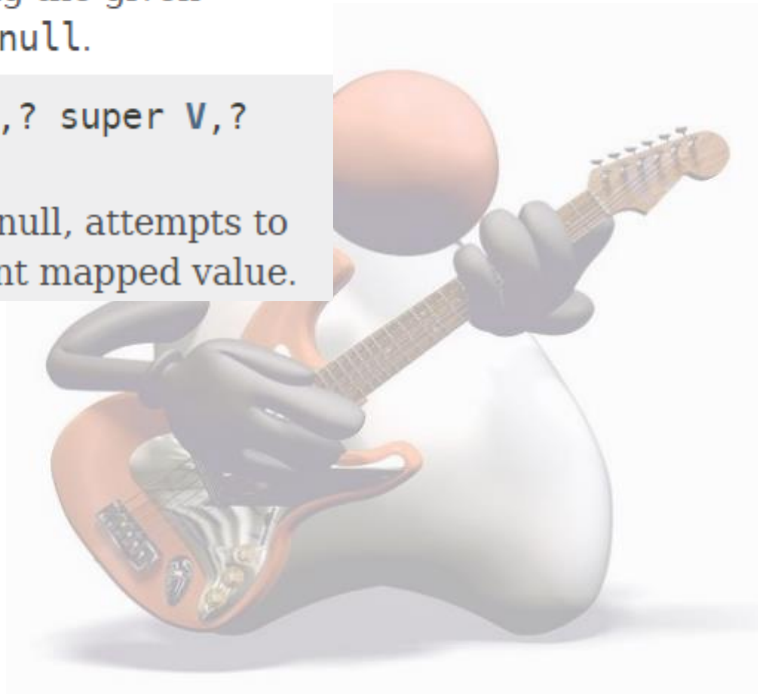


Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

Erweiterungen Map<K,V>

default V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
default V	computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
default V	computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.



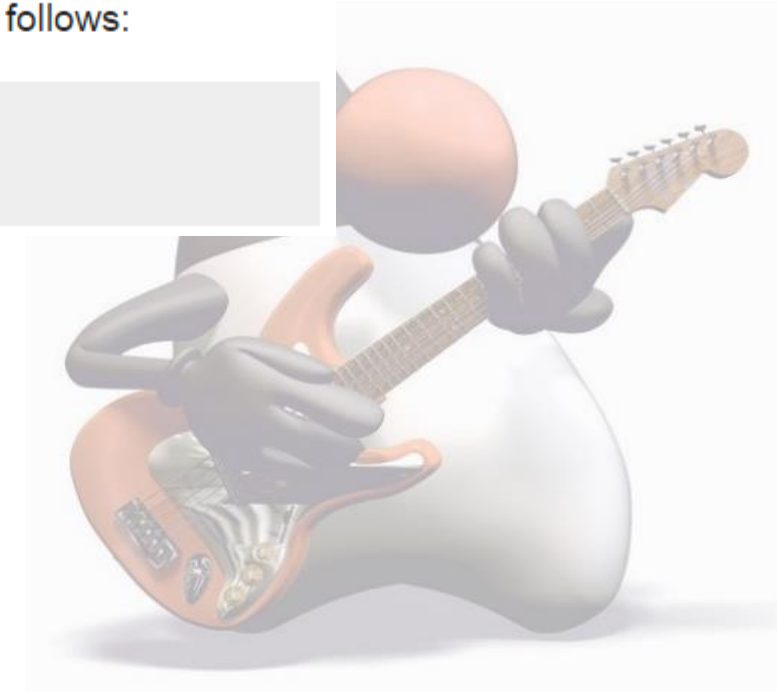
Erweiterungen Map<K,V>

Berechnung der Fibonaccizahlen in einer Map:

```
private static Map<Integer,Long> memo = new HashMap<>();
static {
   memo.put(0,0L); //fibonacci(0)
   memo.put(1,1L); //fibonacci(1)
}
```

And for the inductive step all we have to do is redefine our Fibonacci function as follows:

```
public static long fibonacci(int x) {
   return memo.computeIfAbsent(x, n -> fibonacci(n-1) + fibonacci(n-2));
}
```

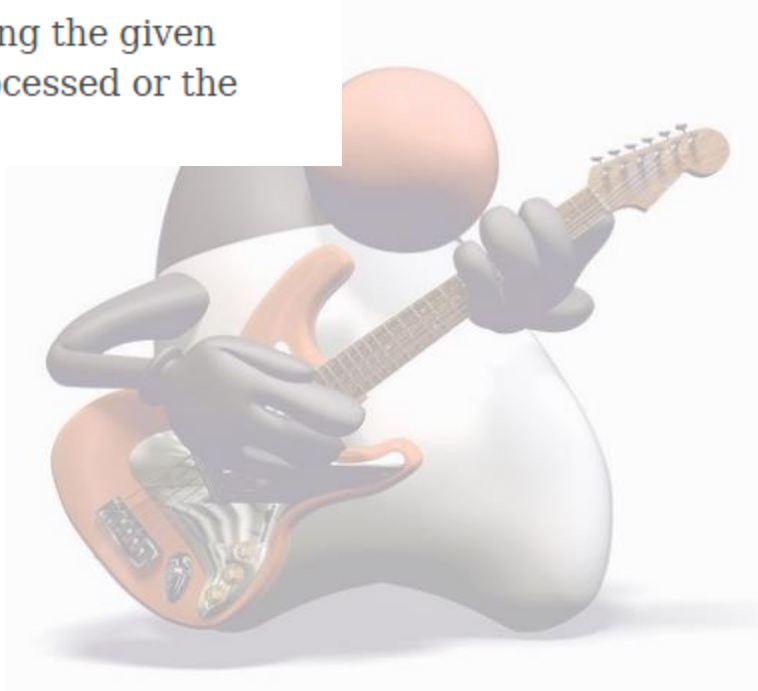


Erweiterungen Map<K,V>

default V	**getOrDefault(Object key, V defaultValue)** Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
default V	**putIfAbsent(K key, V value)** If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

Erweiterungen Map<K,V>

default V	replace(K key, V value) Replaces the entry for the specified key only if it is currently mapped to some value.
default boolean	replace(K key, V oldValue, V newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
default void	replaceAll(BiFunction<? super K,? super V,? extends V> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.



Erweiterungen Map<K,V>

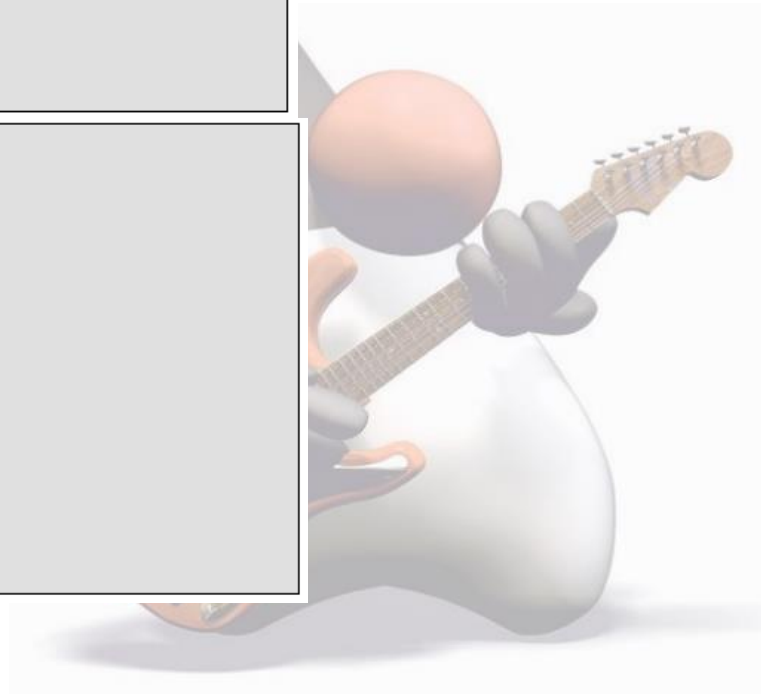
Initialer Wert

```
default V          merge(K key, V value, BiFunction<? super V,? super V,?
                   extends V> remappingFunction)
```

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

```
private static List<String> createTestData( )
{
    final List<String> wordList = Arrays. asList( "Dies", "ist", "eine", "Liste",
                        "Eine", "Liste", "kann", "Worte", "enthalten",
                        "Dies", "ist", "das", "Ende", "der", "Liste");
    return wordList;
}
```

```
public static void main( final String[ ] args)
{
    final List<String> wordList = createTestData( );

    final Map<String, Integer> wordCounts = new TreeMap<>( );
    for ( final String word : wordList)
    {
        wordCounts. merge( word, 1, Integer:: sum);
    }

    System. out. println( wordCounts);
}
```



Was macht das Programm?

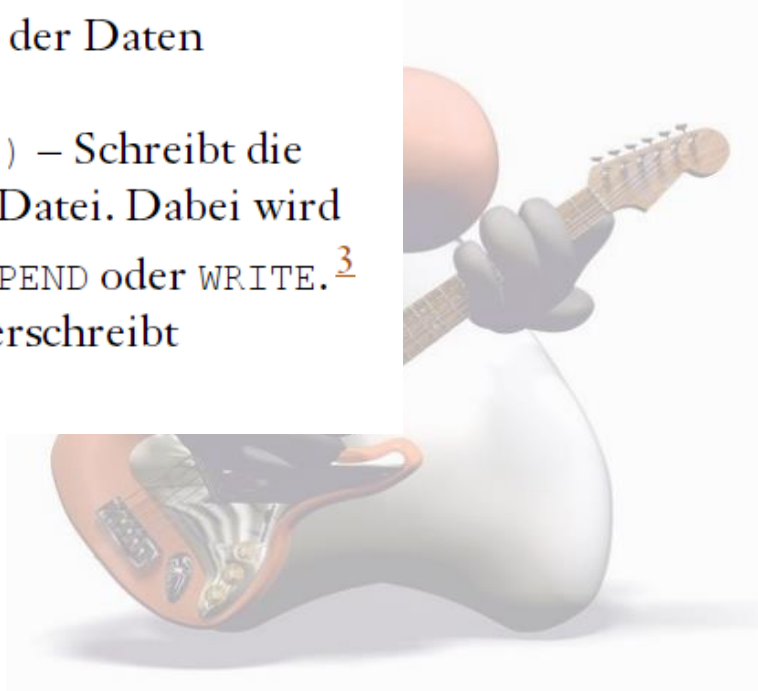
Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

Erweiterungen `nio` und `Files`

Mit JDK 8 gibt es auch im Bereich NIO (New Input Output[2]) einige Neuerungen Stellvertretend dafür betrachten wir die Utility-Klasse `java.nio.file.Files`. Diese wurde um verschiedene Hilfsmethoden erweitert, unter anderem um folgende:

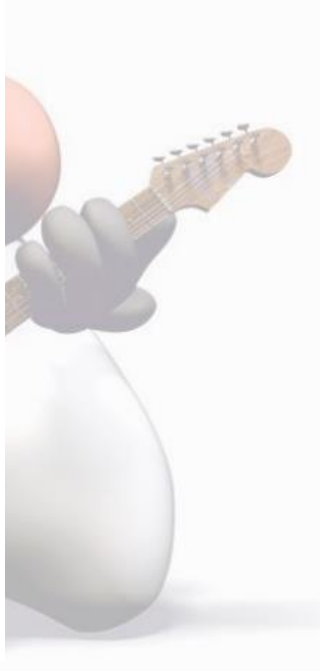
- `lines(Path)` – Stellt eine Datei zeilenweise in Form eines `Stream<String>` bereit.
- `readAllLines(Path)` – Liest eine Datei zeilenweise ein und gibt die Zeile als `List<String>` zurück.
- `list(Path)` – Liefert den Inhalt eines Verzeichnisses als `Stream<Path>`. Das Besondere dabei ist, dass der Inhalt sukzessive bei Bedarf ermittelt wird und nicht direkt von vornherein. Es wird, wie im Abschnitt über Streams beschrieben, immer nur ein Teil der Daten angefordert, wenn eine Terminal Operation ausgeführt wird.
- `write(Path, Iterable<? extends CharSequence>, OpenOption...)` – Schreibt die übergebenen Textzeilen in die durch den `Path`-Parameter referenzierte Datei. Dabei wird der Schreibmodus durch die angegebene `OpenOption` bestimmt, etwa `APPEND` oder `WRITE`.[3] Ersteres fügt an die Datei an, Letzteres schreibt von Anfang an – und überschreibt gegebenenfalls vorhandene Informationen.



Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

- `ConcurrentHashMap<K,V>` – In der Klasse `ConcurrentHashMap<K,V>` wurde eine Vielzahl an Methoden ergänzt. Das sind unter anderem `computeIfAbsent()`, `forEach()`, `forEachEntry()`, `forEachKey()`, `forEachValue()`, `merge()`, `reduce()` und `search()`. Einige davon haben wir schon bei der Betrachtung der Neuerungen im Basisinterface `Map<K,V>` besprochen. Hier möchte ich nur kurz nochmals stellvertretend auf die Methode `putIfAbsent()` eingehen: Insbesondere bei Multithreading vermisst man eine Funktionalität, einen Wert für einen Schlüssel in der Map zu speichern, falls dieser dort noch nicht existiert. Obwohl das einfach klingt, ist es eine Mehrschrittoperation. Eine solche kann zu nahezu beliebigen Zeitpunkten unterbrochen werden, wodurch Inkonsistenzen und Berechnungsfehler durch konkurrierende Zugriffe entstehen können. Herkömmlicherweise musste man entweder mit `synchronized` oder Locks arbeiten, um einen kritischen Abschnitt zu realisieren, in dem zuerst ein Lesezugriff und danach gegebenenfalls ein Schreibzugriff erfolgte. Mithilfe der Methode `putIfAbsent()` kann man sich derartige »Verrenkungen« ersparen.
- `StampedLock` – Die Klasse `java.util.concurrent.locks.StampedLock` ist eine spezielle Variante eines Locks. Diese ist in ihrer Intention ähnlich einem `ReadWriteLock`. Ein `StampedLock` arbeitet zunächst mit optimistischen Sperren und erlaubt damit performantere Verarbeitungen, wenn viel Parallelität in Form gleichzeitiger Lesezugriffe mit seltenen Schreibzugriffen erfolgt. Führt das optimistische Sperren jedoch zu Konflikten, so muss man auf eine konventionelle Sperrung zurückgreifen. Die Hoffnung besteht darin, dass solche Situationen eher selten auftreten und man im Normalfall von den Performance-Vorteilen der optimistischen Verfahren profitieren kann.

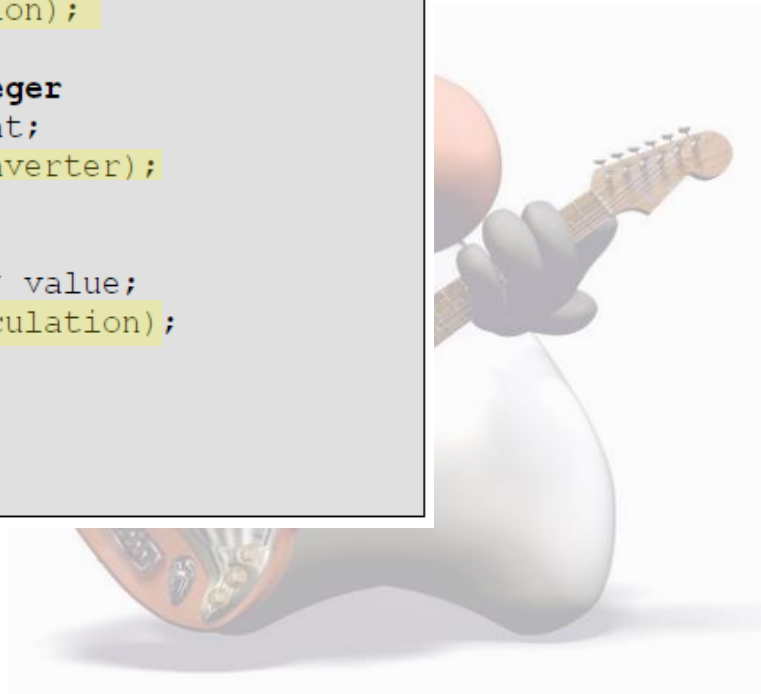


CompletableFuture<T>

- **Erweiterung von** `Future<T>`
- Nahtlose Weiterverarbeitung nach asynchroner Verarbeitung.
- Asynchrone Verarbeitungskette

CompletableFuture<T>

```
public static void main( final String[ ] args) throws InterruptedException,
                                                      ExecutionException
{
    // Schritt 1: Aufwendige Berechnung, hier nur Rückgabe von einem String
    final Supplier<String> longRunningAction = ( ) ->
    {
        System. out. println( "Current thread: " + Thread. currentThread( ));
        return "101";
    };
    final CompletableFuture<String> step1 =
                        CompletableFuture. supplyAsync( longRunningAction);

    // Schritt 2: Konvertierung, hier nur Abbildung von String auf Integer
    final Function<String, Integer> complexConverter = Integer::parseInt;
    final CompletableFuture<Integer> step2 = step1. thenApply( complexConverter);

    // Schritt 3: Konvertierung, hier nur Multiplikation mit .75
    final Function<Integer, Double> complexCalculation = value -> .75 * value;
    final CompletableFuture<Double> step3 = step2. thenApply( complexCalculation);

    // Explizites Auslesen per get( ) löst die Verarbeitung aus
    System. out. println( step3. get( ));
}
```



Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

Nashorn JavaScript Engine

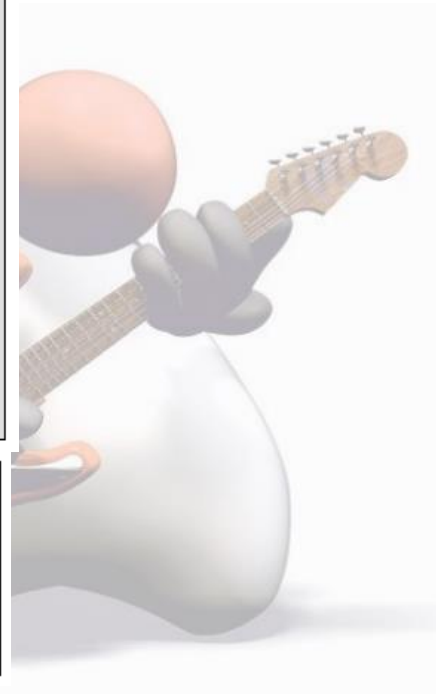
- In JDK 8 eingebaute JavaScript Engine
- Ermöglicht Dynamische Ausführung von JavaScript Code

Nashorn JavaScript Engine

- In JDK 8 eingebaute JavaScript Engine
- Ermöglicht Dynamische Ausführung von JavaScript Code

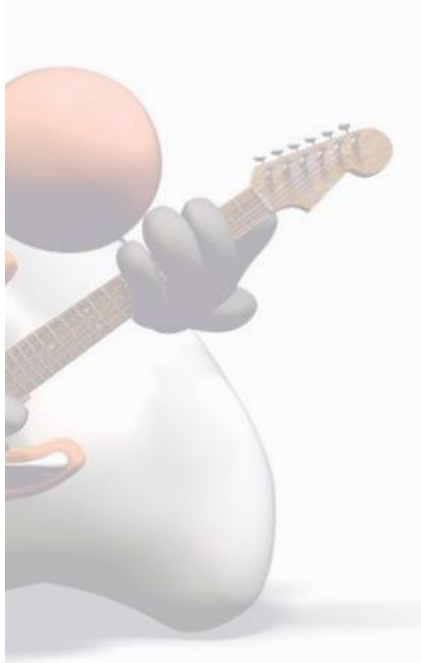
```
public static void main( final String args[ ])
{
   final ScriptEngineManager manager = new ScriptEngineManager( );

   for ( final ScriptEngineFactory factory : manager. getEngineFactories( ))
   {
      System. out. println( factory. getEngineName( ));
      System. out. println( factory. getEngineVersion( ));
      System. out. println( factory. getLanguageName( ));
      System. out. println( factory. getLanguageVersion( ));
      System. out. println( factory. getExtensions( ));
   }
}
```

```
Oracle Nashorn
1. 8. 0
ECMAScript
ECMA - 262 Edition 5. 1
[ js]
```



Na

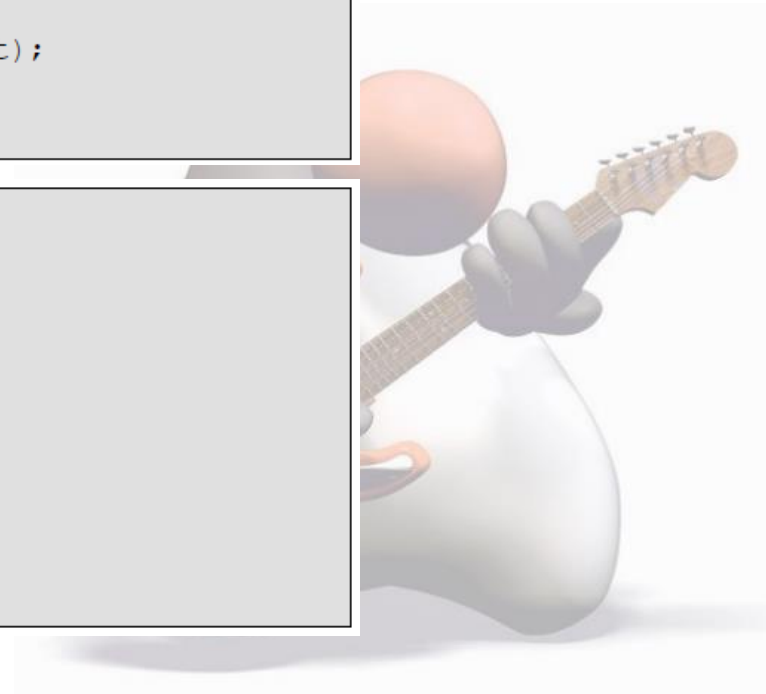

```
// Nur für dieses Beispiel throws Exception - in realen Programmen behandeln
public static void main( final String[ ] args) throws Exception
{
   final ScriptEngineManager manager = new ScriptEngineManager( );
   final ScriptEngine engine = manager. getEngineByName( "js");

   // Kommando println( ) ist mit JDK 7 noch erlaubt; JDK 8: nur noch print( )
   engine. eval( "print( ' Hello! JavaScript executed from a Java program. ' )");

   // Data Binding
   engine. put( "a", 2);
   engine. put( "b", 7);

   final Bindings bindings = engine. getBindings( ScriptContext. ENGINE_SCOPE);
   final Object a = bindings. get( "a");
   final Object b = bindings. get( "b");
   System. out. println( "a = " + a);
   System. out. println( "b = " + b);

   // Berechnung ausführen
   final Object result = engine. eval( "a + b;");
   System. out. println( "a + b = " + result);

   // Ergebnis der Berechnung wird einer JavaScript-Variablen zugewiesen
   final String script = "var ergebnis = Math. max( a, b)";
   engine. eval( script);

   // Wert der Variablen von Engine ermitteln
   final Object result2 = engine. get( "ergebnis");
   System. out. println( "Math. max( a, b) = " + result2);

   // Typen der Variablen ermitteln
   System. out. println( "typeof a = " + engine. eval( "typeof a"));
   System. out. println( "typeof ergebnis = " + engine. eval( "typeof ergebnis"));
}
```



Nashorn JavaScript Engine

```
public static void main( final String[ ] args) throws Exception
{
    final ScriptEngineManager manager = new ScriptEngineManager( );
    final ScriptEngine engine = manager. getEngineByName( "js");

    final String calculation = "7 * ( x * x) + ( 3 - x) * ( x + 3) / 10";
    System. out. println( "f( x) = " + calculation);

    for ( int x = -10; x <= 10; x++)
    {
        engine. put( "x", x);

        final Object calculationResult = engine. eval( calculation);
        System. out. println( "x = " + x + "\t / f( x) = " + calculationResult);
    }
}
```

```
x = -10  / f( x) = 690. 9
x = -9   / f( x) = 559. 8
...
x = -2   / f( x) = 28. 5
x = -1   / f( x) = 7. 8
x = 0    / f( x) = 0. 9
x = 1    / f( x) = 7. 8
x = 2    / f( x) = 28. 5
...
x = 9    / f( x) = 559. 8
x = 10   / f( x) = 690. 9
```



Agenda

- Weitere Änderungen in JDK 8
  - Keine Permanent Generation mehr
  - Erweiterungen im Bereich Reflection
  - Base64-Codierungen
  - Änderungen bei Annotations

Permanent Generation vor JDK 8

- Permanent Generation: Teil des Heaps in dem Klassen und Konstanten gespeichert werden.
- java.lang.OutOfMemoryError: PermGen space
- Problem: viele Bibliotheken, dynamisches Laden und Entladen von Klassen

Keine Permanent Generation

- Permanent Generation Heap wurde nun durch einen größenveränderlichen Metaspace ersetzt.
- Kein Setzen der PermGenSize mehr möglich (wird ignoriert)
- Es kann nicht mehr zu OutOfMemory Exceptions wegen zu kleiner PerGenSize kommen

java.lang.OutOfMemoryError: PermGen space

Agenda

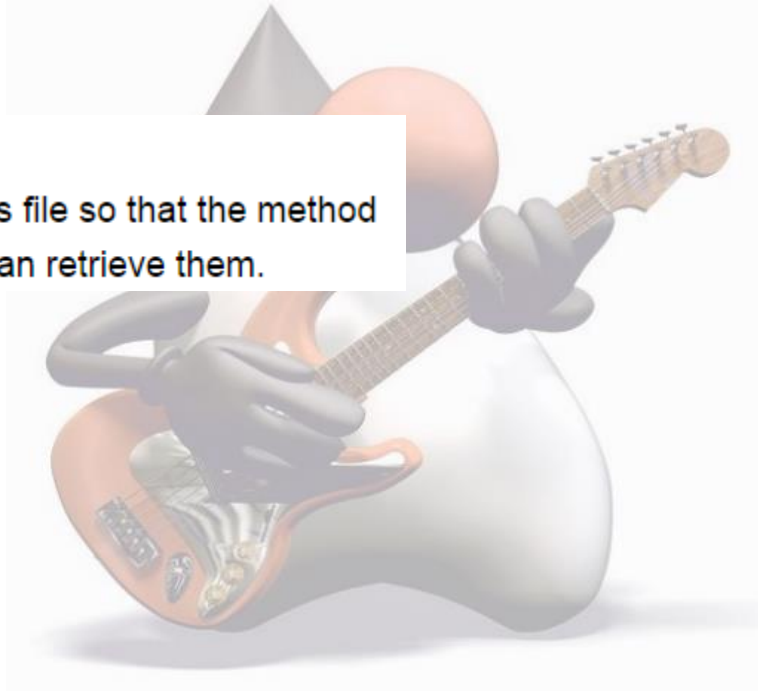
- Weitere Änderungen in JDK 8
  - Keine Permanent Generation mehr
  - Erweiterungen im Bereich Reflection
  - Base64-Codierungen
  - Änderungen bei Annotations

Erweiterungen Reflections API

- Abfragen der Parameternamen von Methoden jetzt möglich
- Dazu muss ein Compiler Flag beim Aufruf von javac gesetzt werden:

-parameters

Stores formal parameter names of constructors and methods in the generated class file so that the method `java.lang.reflect.Executable.getParameters` from the Reflection API can retrieve them.



Agenda

- Weitere Änderungen in JDK 8
  - Keine Permanent Generation mehr
  - Erweiterungen im Bereich Reflection
  - Base64-Codierungen
  - Änderungen bei Annotations

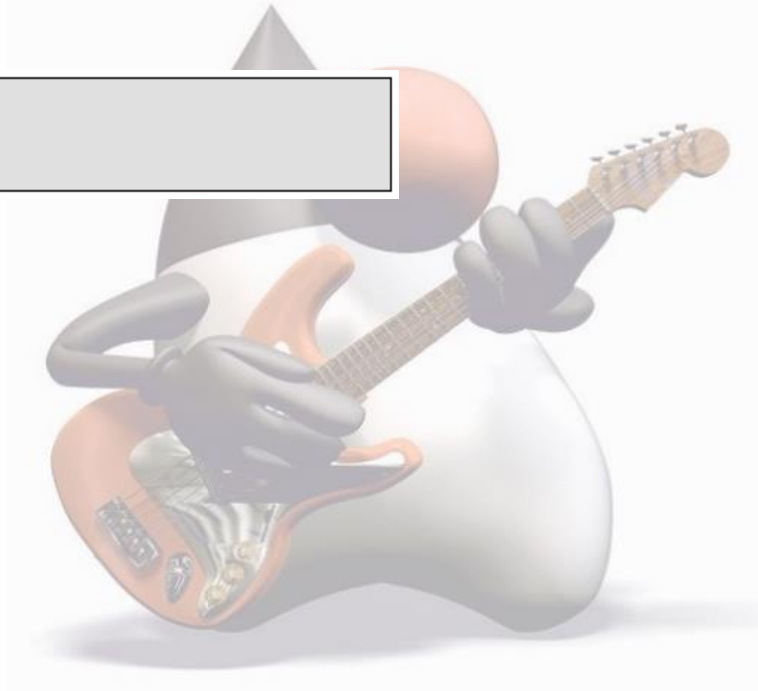
Base64 Ecoding

- Base64 Encoding wurde bisher nur in der Sun Implementierung durch die proprietäre Klasse sun.misc.Base64Encoder / Decoder unterstützt
- Mit java.util.Base64 gibt es jetzt eine im Java Standard verfügbare Klasse.

Base64 Ecoding

```
public static void main( final String[ ] args)
{
    final byte[ ] bytes = "This is the Base64 Test". getBytes( );

    final String encoded = Base64. getEncoder( ). encodeToString( bytes);
    System. out. println( "Base64 encoded = " + encoded);

    final byte[ ] decoded = Base64. getDecoder( ). decode( encoded);
    System. out. println( "Base64 decoded = " + new String( decoded));
}
```

```
Base64 encoded = VGhpcyBpcyB0aGUgQmFzZTY0IFRlc3Q=
Base64 decoded = This is the Base64 Test
```



Agenda

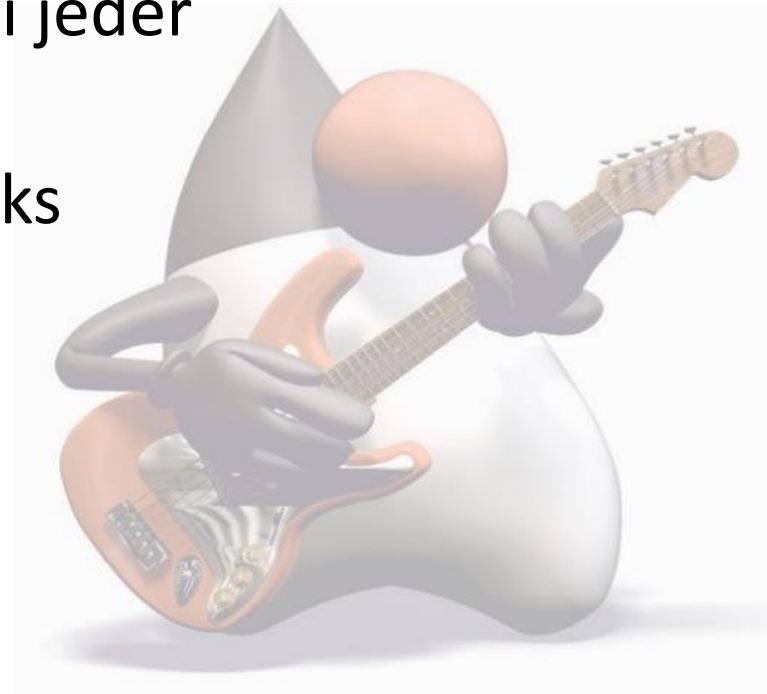
- Weitere Änderungen in JDK 8
  - Keine Permanent Generation mehr
  - Erweiterungen im Bereich Reflection
  - Base64-Codierungen
  - Änderungen bei Annotations

Änderungen bei Annotations

Die möglichen Targets von Annotations (enum ElementType) wurden erweiter um:

- **TYPE_PARAMETER**: Annotations können auf Typ Parametern bei Generics angewendet werden
- **TYPE_USE:** Verwendung von Annotation bei jeder Verwendung eines Typs

Bedeutet: Künftige Nutzung durch Frameworks



Änderungen bei Annotations

Simple type definitions with type annotations look like this:

```
1. @NotNull String str1 = ...
2. @Email String str2 = ...
3. @NotNull @NotBlank String str3 = ...
```

Type annotations can also be applied to nested types

```
1. Map.@NonNull Entry = ...
```

Constructors with type annotations:

```
1. new @Interned MyObject()
2. new @NonEmpty @Readonly List<String>(myNonEmptyStringSet)
```

They work with nested (non static) class constructors too:

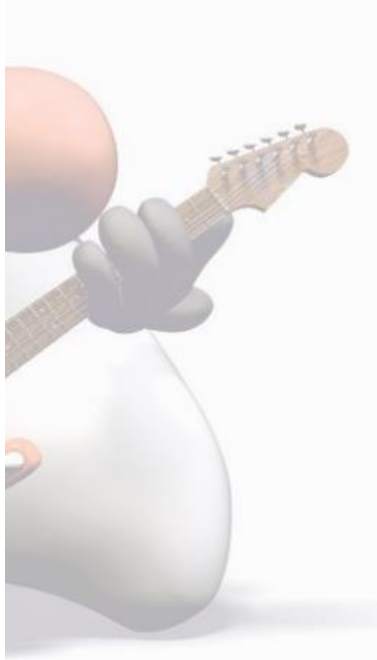
```
1. myObject.new @Readonly NestedClass()
```

Type casts:

```
1. myString = (@NonNull String) myObject;
2. query = (@Untainted String) str;
```

Inheritance:

```
1. class UnmodifiableList<T> implements @Readonly List<T> {
```



Änd

We can use type Annotations with generic type arguments:

```
1. List<@Email String> emails = ...
2. List<@ReadOnly @Localized Message> messages = ...
3. Graph<@Directional Node> directedGraph = ...
```

Of course we can nest them:

```
1. Map<@NonNull String, @NonEmpty List<@Readonly Document>> c
```

Or apply them to intersection Types:

```
1. public <E extends @ReadOnly Composable<E> & @Localized Mess
   void foo(...) { ... }
```

Including parameter bounds and wildcard bounds:

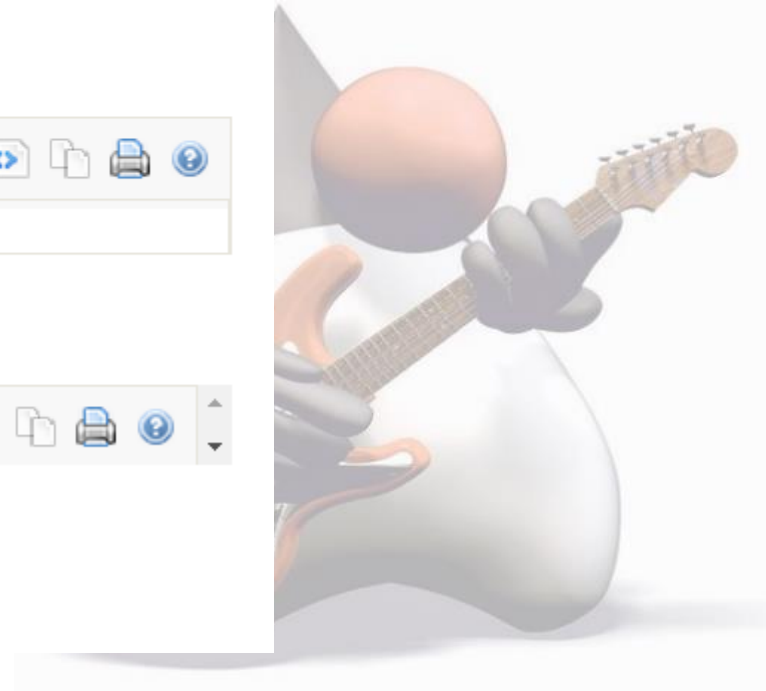
```
1. class Folder<F extends @Existing File> { ... }
2. Collection<? super @Existing File> c = ...
3. List<@Immutable ? extends Comparable<T>> unchangeable = ...
```

Generic method invocation with type annotations looks like this:

```
1. myObject.<@NotBlank String>myMethod(...);
```

For generic constructors, the annotation follows the explicit type arguments:

```
1 new <String> @Interned MyObject()
```



Änderungen bei Annotations

Throwing exceptions:

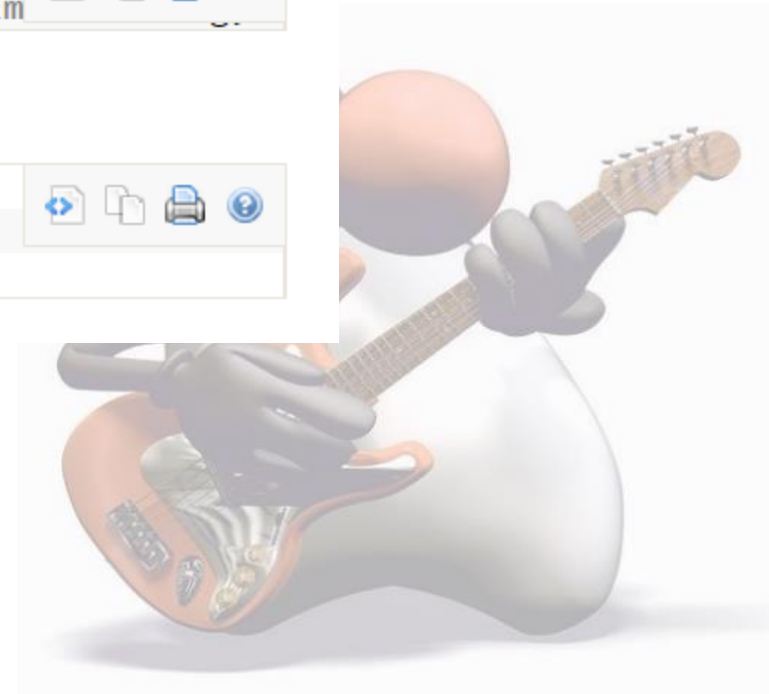
```
1. void monitorTemperature() throws @Critical TemperatureExcep
2. void authenticate() throws @Fatal @Logged AccessDeniedExcep
```

Type annotations in instanceof statements:

```
1. boolean isNonNull = myString instanceof @NonNull String;
2. boolean isNonBlankEmail = myString instanceof @NotBlank @Em
```

And finally Java 8 method and constructor references:

```
1. @Vernal Date::getDay
2. List<@English String>::size
3. Arrays::<@NonNegative Integer>sort
```



Java 8 New Features

michael.schaffler@java.at

Agenda

- Lambda Expressions
  - Einstieg in Lambdas
  - Default Methoden in Interfaces
  - Methodenreferenzen

Agenda

- Collections & Lambdas
  - externe vs. Interne Iteration
  - Collections Erweiterungen
  - Streams
  - Filter-Map-Reduce
  - Fallstricke bei Lambdas

Agenda

- Java Date & Time API
  - Datumsverarbeitung vor JSR-310
  - Überblick über die neu eingeführten Klassen

Agenda

- Weitere Änderungen in JDK 8
  - Erweiterungen im Interface Comparator<T>
  - Die Klasse Optional<T>
  - Parallele Operationen auf Arrays
  - Erweiterungen im Interface Map<K,V>
  - Erweiterungen im NIO und der Klasse Files
  - Erweiterungen im Bereich Concurrency
  - „Nashorn“ – die neue JavaScript Engine

Agenda

- Weitere Änderungen in JDK 8
  - Keine Permanent Generation mehr
  - Erweiterungen im Bereich Reflection
  - Base64-Codierungen
  - Änderungen bei Annotations