
JavaOne™
Java SE 8 for
Java EE devs
@Delabassee @JosePaumard

Agenda

Java SE 8 has been published ~1 year ago

Java EE 7 application servers are supporting it now (Weblogic)

Questions?

#SE84EE

José PAUMARD

MCF Um. Paris 13

PhD Appl C.S.

Open source dev.

Independant

@JosePaumard

José PAUMARD

pluralsight
hardcore dev and IT training

Java™ magazine
By and for the Java community

Parleys

Microsoft Virtual Academy

@JosePaumard

Date

Date: Instant

An Instant is a point on the time line

```
Instant start = Instant.now() ;

Instant end = Instant.now() ;
```

Date: Duration

A « duration » is the amount of time between instants

```
Instant start = Instant.now() ;

Instant end = Instant.now() ;
```

```
Duration elapsed = Duration.between(start, end) ;
long millis = elapsed.toMillis() ;
```

Date: Duration

There is an arithmetic on durations

```
Instant start = Instant.now() ;

Instant end = Instant.now() ;
```

```
Duration elapsed = Duration.between(start, end) ;
long millis = elapsed.toMillis() ;
```

```
elapsed.plus(2L, ChronoUnit.SECONDS) ;
```

Date: Duration

The precision of the time line is 10^{-9} s (nanosecond)

Date: LocalDate

A LocalDate is an « every day » date

```
LocalDate now = LocalDate.now() ;

LocalDate shakespeareDoB =
   LocaleDate.of(1564, Month.APRIL, 23) ;
```

Date: Period

A « period » is the amount of time between local dates

```
LocalDate now = LocalDate.now() ;

LocalDate shakespeareDoB =
   LocaleDate.of(1564, Month.APRIL, 23) ;

Period p = shakespeareDoB.until(now) ;
System.out.println("# years = " + p.getYears()) ;
```

Date: Period

A « period » is the amount of time between local dates

```
LocalDate now = LocalDate.now() ;

LocalDate shakespeareDoB =
   LocaleDate.of(1564, Month.APRIL, 23) ;

Period p = shakespeareDoB.until(now) ;
System.out.println("# years = " + p.getYears()) ;
```

```
long days = shakespeareDoB.until(now, ChronoUnit.DAYS) ;
System.out.println("# days = " + days) ;
```

Date: TemporalAdjuster

Arithmetic with local dates

```
LocalDate now = LocalDate.now() ;

LocalDate nextSunday =
   now.with(TemporalAdjusters.next(DayOfWeek.SUNDAY)) ;
```

Date: TemporalAdjuster

Arithmetic with local dates

```
LocalDate now = LocalDate.now() ;

LocalDate nextSunday =
   now.with(TemporalAdjusters.next(DayOfWeek.SUNDAY)) ;
```

A toolbox of 14 static methods

`firstDayOfMonth()`, `lastDayOfYear()`

`firstDayOfNextMonth()`

Date: TemporalAdjuster

Arithmetic with local dates

```
LocalDate now = LocalDate.now() ;

LocalDate nextSunday =
   now.with(TemporalAdjusters.next(DayOfWeek.SUNDAY)) ;
```

A toolbox of 14 static methods

`firstInMonth(DayOfWeek.MONDAY)`

`next(DayOfWeek.FRIDAY)`

Date: LocalTime

A LocalTime is an everyday time: ex. 10:20

```
LocalTime now = LocalTime.now() ;

LocalTime time = LocalTime.of(10, 20) ; // 10:20
```

Date: LocalTime

A LocalTime is an everyday time: ex. 10:20

```
LocalTime now = LocalTime.now() ;

LocalTime time = LocalTime.of(10, 20) ; // 10:20
```

```
LocalTime lunchTime = LocalTime.of(12, 30) ;

LocalTime coffeeTime = lunchTime.plusHours(2) ; // 14:20
```

Date: ZonedTime

Useful for localized times

```
Set<String> allZonesIds = ZoneId.getAvailableZoneIds() ;

String ukTZ = ZoneId.of("Europe/London") ;
```

Date: ZonedTime

Useful for localized times

```
System.out.println(
   ZonedDateTime.of(
      1564, Month.APRIL.getValue(), 23, // year / month / day
      10, 0, 0, 0,                      // h / mn / s / nanos
      ZoneId.of("Europe/London"))
); // prints 1564-04-23T10:00-00:01:15[Europe/London]
```

Date: ZonedTime

Arithmetic on localized time

```
ZonedDateTime currentMeeting =
   ZonedDateTime.of(
      LocalDate.of(2014, Month.APRIL, 18), // LocalDate
      LocalTime.of(9, 30),                 // LocalTime
      ZoneId.of("Europe/London")
);

ZonedDateTime nextMeeting =
   currentMeeting.plus(Period.ofMonth(1));
```

Date: ZonedTime

Arithmetic on localized time

```
ZonedDateTime currentMeeting =
   ZonedDateTime.of(
      LocalDate.of(2014, Month.APRIL, 18), // LocalDate
      LocalTime.of(9, 30),                 // LocalTime
      ZoneId.of("Europe/London")
);

ZonedDateTime nextMeeting =
   currentMeeting.plus(Period.ofMonth(1)) ;
ZonedDateTime nextMeetingUS =
   nextMeeting.withZoneSameInstant(ZoneId.of("US/Central")) ;
```

Date: Formatter

Factory class: DateTimeFormatter

```
ZonedDateTime nextMeetingUS =
   nextMeeting.withZoneSameInstant(ZoneId.of("US/Central"));

System.out.println(
   DateTimeFormatter.ISO_DATE_TIME.format(nextMeetingUS)
);
// prints 2014-04-12T03:30:00-05:00[US/Central]

System.out.println(
   DateTimeFormatter.RFC_1123_DATE_TIME.format(nextMeetingUS)
);
// prints Sat, 12 Apr 2014 03:30:00 -0500
```

Date: bridges with java.util.Date

API to go from legacy Date to Date & Time

```
Date date = Date.from(instant);              // API -> legacy
Instant instant = date.toInstant();          // legacy -> new API
```

Date: bridges with java.util.Date

API to go from legacy Date to Date & Time

```
Date date = Date.from(instant);              // API -> legacy
Instant instant = date.toInstant();          // legacy -> new API
```

```
TimeStamp time = TimeStamp.from(instant); // API -> legacy
Instant instant = time.toInstant();       // legacy -> new API
```

Date: bridges with java.util.Date

API to go from legacy Date to Date & Time

```
Date date = Date.from(instant);               // API -> legacy
Instant instant = date.toInstant();           // legacy -> new API
```

```
TimeStamp time = TimeStamp.from(instant); // API -> legacy
Instant instant = time.toInstant();       // legacy -> new API
```

```
Date date = Date.from(localDate);          // API -> legacy
LocalDate localDate = date.toLocalDate(); // legacy -> new API
```

Date: bridges with java.util.Date

API to go from legacy Date to Date & Time

```
Date date = Date.from(instant);              // API -> legacy
Instant instant = date.toInstant();          // legacy -> new API
```

```
TimeStamp time = TimeStamp.from(instant); // API -> legacy
Instant instant = time.toInstant();       // legacy -> new API
```

```
Date date = Date.from(localDate);          // API -> legacy
LocalDate localDate = date.toLocalDate(); // legacy -> new API
```

```
Time time = Time.from(localTime);          // API -> legacy
LocalTime localTime = time.toLocalTime(); // legacy -> new API
```

Usable in JPA

JPA does not support this Date & Time API (yet)

But with converters, we can still use it

```
@Entity
public abstract class AbstractPersistent {

   @Convert(converter=DateConverter.class)
   @Temporal(TemporalType.TIME)
   private Instant instance;
}
```

Usable in JPA

The converter is a classical object

```
public class DateConverter
implements AttributeConverter<Instant, Date> {

   public Date convertToDatabaseColumn(Instant instant) {
      return Date.from(instant);
   }

   public Instant convertToEntityAttribute(Date date) {
      return date.toInstant();
   }
}
```

Usable in JPA

The converter is a classical object

```
public class DateFormatConverter
implements AttributeConverter<ZonedDateTime, String> {

   public String convertToDatabaseColumn(ZonedDateTime time) {
      return DateTimeFormatter.ISO_DATE_TIME.format(time);
   }

   public ZonedDateTime convertToEntityAttribute(String formated) {
      return DateTimeFormatter.ISO_DATE_TIME
                  .parse(formated, ZonedDateTime::from);
   }
}
```

Annotations

Annotations in Java 7

Wrapping annotations

```
@TestCases({
   @TestCase(param=1, expected=false),
   @TestCase(param=2, expected=true)
})
public boolean even(int param) {
   return param % 2 == 0;
}
```

An annotation cannot be applied more than once

Annotations in Java 8

Java 8 makes it possible!

```
@TestCase(param=1, expected=false),
@TestCase(param=2, expected=true)
public boolean even(int param) {
   return param % 2 == 0;
}
```

How does it work?

It’s a trick!

How does it work?

First: create the annotations as usual

```
@interface TestCase {
   int param();
   boolean expected();
}
```

```
@interface TestCases {
   TestCase[] value();
}
```

How does it work?

Second: make the annotation repeatable

```
@Repeatable(TestCases.class)
@interface TestCase {
    int param();
    boolean expected();
}
```

```
@interface TestCases {
    TestCase[] value();
}
```

Type annotations

Annotations can be now put on types

Example 1: tell that a variable should not be null

```
private @NonNull List<Person> persons = ... ;
```

Type annotations

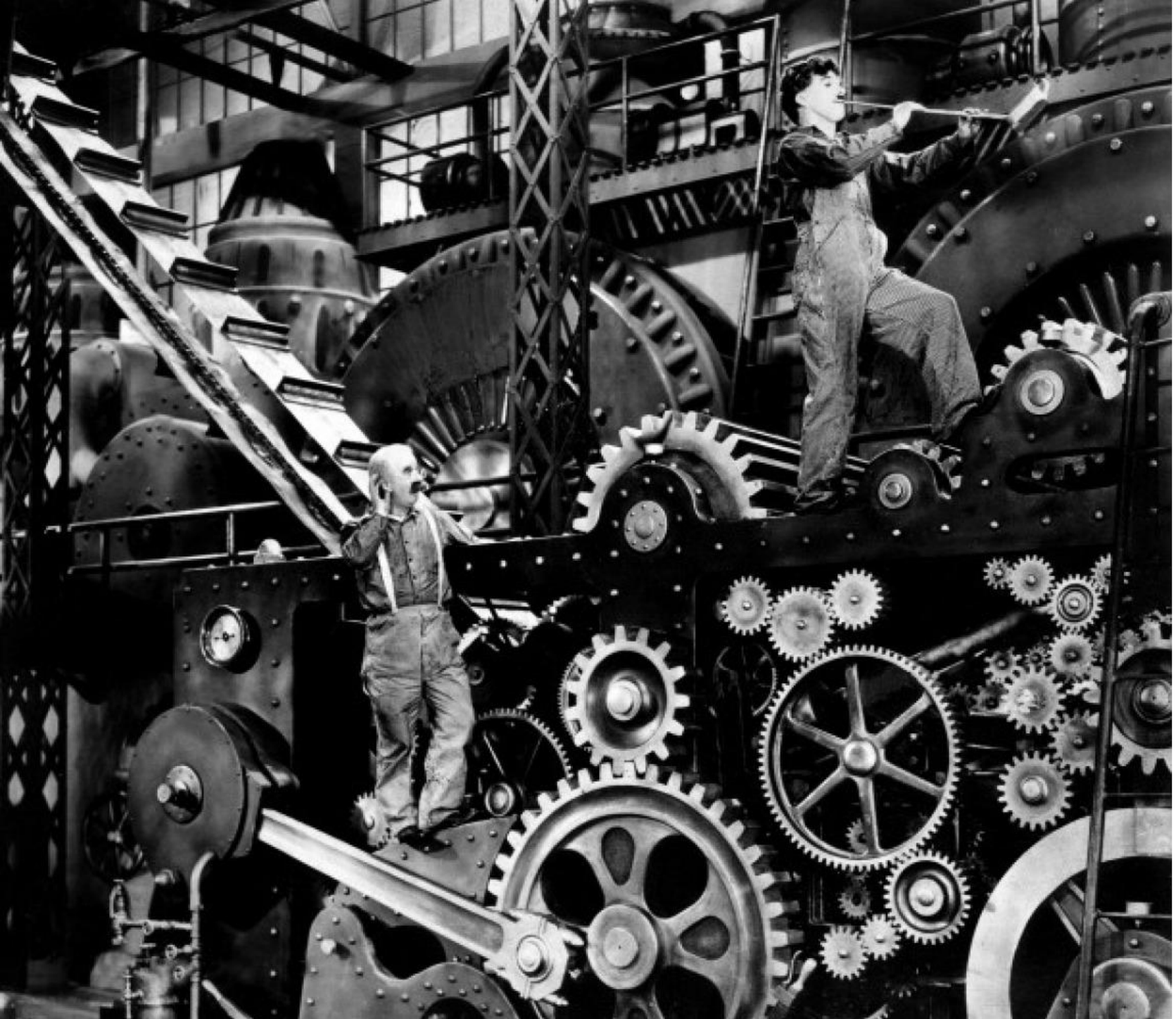
Annotations can be now put on types

Example 1: tell that a variable should not be null

```
private @NonNull List<Person> persons = ... ;
```

Example 2: the content should not be null neither

```
private @NonNull List<@NonNull Person> persons = ... ;
```



String

String: concatenation

The newbie writes:

```
String s1 = "hello" ;
String s2 = " world!" ;

String s3 = s1 + " " + s2 ;
```

String: concatenation

The Java 8 well-trained dev writes:

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ") ;
sj.add("one").add("two").add("three") ;
String s = sj.toString() ;
System.out.println(s) ;
```

String: concatenation

The Java 8 well-trained dev writes:

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ") ;
sj.add("one").add("two").add("three") ;
String s = sj.toString() ;
System.out.println(s) ;
```

And it prints:

```
> one, two, three
```

String: concatenation

The Java 8 well-trained dev writes:

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ") ;
sj.add("one") ;
String s = sj.toString() ;
System.out.println(s) ;
```

And it prints:

```
> one
```

String: concatenation

The Java 8 well-trained dev writes:

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ", "{", "}") ;
sj.add("one").add("two").add("three") ;
String s = sj.toString() ;
System.out.println(s) ;
```

And it prints:

```
> {one, two, three}
```

String: concatenation

The Java 8 well-trained dev writes:

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ", "{", "}") ;
sj.add("one") ;
String s = sj.toString() ;
System.out.println(s) ;
```

And it prints:

```
> {one}
```

String: concatenation

The Java 8 well-trained dev writes:

```
// The JDK 8 way
StringJoiner sj = new StringJoiner(", ", "{", "}") ;
// we dont put anything in it
String s = sj.toString() ;
System.out.println(s) ;
```

And it prints:

```
> {}
```

String: concatenation

And it’s nearly accessible from the String class itself:

```
// From the String class, with a vararg
String s = String.join(", ", "one", "two", "three");
System.out.println(s);
```

And it prints:

```
> one, two, three
```

String: concatenation

And it’s nearly accessible from the String class itself:

```
// From the String class, with an Iterable
String [] tab = {"one", "two", "three"} ;
String s = String.join(", ", tab) ;
System.out.println(s) ;
```

And it prints:

```
> one, two, three
```

String: concatenation

And it’s nearly accessible from the String class itself:

```
// From the String class, with an Iterable
String [] tab = {"one", "two", "three"} ;
String s = String.join(", ", tab) ;
System.out.println(s) ;
```

And it prints:

```
> one, two, three
```

Comparator

Comparator!

The good ol' way!

```
// comparison using the last name
Comparator<Person> compareLastName =
    new Comparator<Person>() {

        @Override
        public int compare(Person p1, Person p2) {
            return p1.getLastName().compareTo(p2.getLastName());
        }
    };
```

Comparator!

```
// comparison using the last name then the first name
Comparator<Person> compareLastNameThenFirstName =
    new Comparator<Person>() {

        @Override
        public int compare(Person p1, Person p2) {
            int lastNameComparison =
                p1.getLastName().compareTo(p2.getLastName());
            return lastNameComparison == 0 ?
                p2.getFirstName().compareTo(p2.getFirstName());
                lastNameComparison;
        }
    };
```

Comparator!

The JDK 8 way!

```
Comparator.comparingBy(Person::getLastName);     // static method
```

Comparator!

The JDK 8 way!

```
Comparator.comparingBy(Person::getLastName)      // static method
          .thenComparing(Person::getFirstName); // default method
```

Comparator!

The JDK 8 way!

```
Comparator.comparingBy(Person::getLastName)      // static method
          .thenComparing(Person::getFirstName)   // default method
          .thenComparing(Person::getAge);
```

Comparator!

Dont like the natural order?

```
Comparator<Person> comp = Comparator.naturalOrder();

Comparator<Person> reversedComp = Comparator.reversedOrder();
```

Comparator!

Dont like the natural order?

```
Comparator<Person> comp = Comparator.comparingBy(Person::getLastName);

Comparator<Person> reversedComp = comp.reversed();
```

Comparator!

And what about null values?

```
Comparator<Person> comp =
    Comparator.naturalOrder();
```

Comparator!

And what about null values?

```
Comparator<Person> comp =
    Comparator.nullsFirst(Comparator.naturalOrder());
```

Comparator!

And what about null values?

```
Comparator<Person> comp =
    Comparator.nullsFirst(Comparator.naturalOrder());

Comparator<Person> comp =
    Comparator.nullsLast(Comparator.naturalOrder());
```

List

Iterable: forEach

ForEach: consumes the elements

```
// method forEach defined on Iterable
List<String> strings =
   Arrays.asList("one", "two", "three") ;

strings.forEach(System.out::println) ;
```

Doesn’t work on arrays

```
> one, two, three
```

Collection: removeIf

Removes an object, takes a Predicate

```
Collection<String> strings =
   Arrays.asList("one", "two", "three", "four");

// works « in place », no Collections.unmodifiable...
Collection<String> list = new ArrayList<>(strings);

// returns true if the list has been modified
boolean b = list.removeIf(s -> s.length() > 4);
```

```
> one, two, four
```

List: replaceAll

Replaces an object with its transform

```
List<String> strings =
   Arrays.asList("one", "two", "three", "four");

// works « in place », no Collections.unmodifiable...
List<String> list = new ArrayList<>(strings);

// returns nothing
list.replaceAll(String::toUpperCase);
```

```
> ONE, TWO, THREE, FOUR
```

List: sort

Sorts a List in place, takes a Comparator

```
List<String> strings =
   Arrays.asList("one", "two", "three", "four");

// works « in place », no Collections.unmodifiable...
List<String> list = new ArrayList<>(strings);

// returns nothing
list.sort(Comparator.naturalOrder()) ;
```

```
> four, one, three, two
```

What we might have in 9 (JEP 269)

New patterns to build collections

```
Set<String> strings = Set.of("one", "two", "three", "four");

List<String> strings = List.of("one", "two", "three", "four");
```

Map

Map: forEach

Takes a BiConsumer

```
// the existing map
Map<String, Person> map = ... ;

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

Map: get

```
// the existing map
Map<String, Person> map = ... ;

Person p = map.get(key);
```

What happens if key is not in the map?

Map: get

```
// the existing map
Map<String, Person> map = ... ;

Person p = map.getOrDefault(key, Person.DEFAULT_PERSON);
```

Map: put

```
// the existing map
Map<String, Person> map = ... ;

map.put(key, person);
```

Map: putIfAbsent

```
// the existing map
Map<String, Person> map = ... ;

map.put(key, person);
map.putIfAbsent(key, person);
```

Map: replace

Replaces a value from its key

```
// the existing map
Map<String, Person> map = ... ;

// key, newValue
map.replace("six", john) ;

// key, oldValue, newValue
map.replace("six", peter, john) ;
```


Map: replaceAll

Replaces all the values from the map, using a λ

```
// the existing map
Map<String, Person> map = ... ;

// key, oldValue
map.replaceAll(
    (key, value) -> key + " -> " + value ;
) ;
```

Map: remove

Removes a key / value pair if it's there

```
// the existing map
Map<String, Person> map = ... ;

// key, oldValue
map.remove("six", john) ;
```

Map: compute

Computes a value from the existing key / value pair and a λ

```
// the existing map
Map<String, Person> map = ... ;

// key, oldValue
map.compute(
    key,
    (key, person) -> key + "::" + person // person can be null
) ;
```

Map: computeIfAbsent

Computes a value from a key that is not in the map

```
// the existing map
Map<String, Person> map = ... ;

// key, no existing value
map.computeIfAbsent(
    key,
    key -> person
) ;
```

Map: computeIfPresent

Computes a value from the existing key / value pair and a λ

```
// the existing map
Map<String, Person> map = ... ;

// key, the existing value
map.computeIfPresent(
    key, person,
    (key, person) -> newPerson // person cannot be null
) ;
```

Map: compute*

The trick is: these methods return the value, whether it was there of just created

Building maps of maps

Making maps of maps becomes sooo easy!

```
// the existing map
Map<String, Map<String, Person>> map = ... ;

// key, newValue
map.computeIfAbsent(
    "one",
    (key, value) -> new HashMap<>()
).put("two", john);
```

Map: merge

Computes the new value from the existing value, the newly added value and a λ

```
// the existing map
Map<Long, String> map = ... ;

// key, otherValue
map.merge(
    key,
    otherValue,
    (value, otherValue) -> String.join(", ", value, otherValue)
) ;
```

What we might have in 9 (JEP 269)

New patterns to build maps

```
Map<Integer, String> map =
    Map.of(1, "one", 2, "two");

Map<Integer, String> map =
    Map.fromEntries(
        entry(1, "one"),
        entry(2, "two"),
        entry(3, "three")
    );
```

Streams

What is a Stream?

A new API

A typed interface

A new concept

What is a Stream?

A new API

A typed interface

A new concept

Let’s see some code!

Readable and efficient patterns

Extract histograms from data

```
List<Person> people = Arrays.asList();

Map<City, Double> agesByCity =
    people.stream()
        .filter(p -> p.getAge() > 20)
        .collect(
           Collectors.groupingBy(
              Person::getCity,
              Collectors.averagingDouble(Person::getAge)
           )
        );
```

Readable and efficient patterns

Extract histograms from data

```
List<Person> people = Arrays.asList();

Map<City, Double> agesByCity =
    people.stream()
        .filter(p -> p.getAge() > 20)
        .collect(
            Collectors.groupingBy(
                Person::getCity,
                Collectors.averagingDouble(Person::getAge)
            )
        );
```

Readable and efficient patterns

Extract histograms from data

```
List<Person> people = Arrays.asList();

Map<City, Double> agesByCity =
    people.stream()
        .filter(p -> p.getAge() > 20)
        .collect(
            Collectors.groupingBy(
                Person::getCity,
                Collectors.averagingDouble(Person::getAge)
            )
        );
```

Readable and efficient patterns

Extract histograms from data

```
List<Person> people = Arrays.asList();

Map<City, Double> agesByCity =
    people.stream()
        .filter(p -> p.getAge() > 20)
        .collect(
            Collectors.groupingBy(
                Person::getCity,
                Collectors.averagingDouble(Person::getAge)
            )
        );
```

Readable and efficient patterns

Extract histograms from data

```
List<Person> people = Arrays.asList();

Map<City, Double> agesByCity =
    people.stream()
        .filter(p -> p.getAge() > 20)
        .collect(
            Collectors.groupingBy(
                Person::getCity,
                Collectors.averagingDouble(Person::getAge)
            )
        );
```

Readable and efficient patterns

Extract histograms from data

```
List<Person> people = Arrays.asList();

Map<City, Double> agesByCity =
    people.stream()
        .filter(p -> p.getAge() > 20)
        .collect(
            Collectors.groupingBy(
                Person::getCity,
                Collectors.averagingDouble(Person::getAge)
            )
        );
```

Building a Stream on a String

Building a Stream on the letters of a String:

```
String s = "hello";
IntStream stream = s.chars(); // stream on the letters of s
```

Building a Stream on a String

Building a Stream on the letters of a String:

```
String s = "hello";
IntStream stream = s.chars(); // stream on the letters of s

s.chars()                                  // IntStream
 .mapToObj(letter -> (char)letter) // Stream<Character>
 .map(Character::toUpperCase)
 .forEach(System.out::println);    // Prints a Character
```

```
> HELLO
```

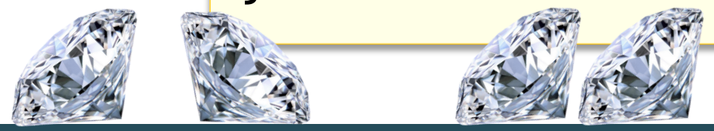
Build a Stream from a text file

Find the first error line from a log file

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("d:", "tmp", "debug.log");
try (Stream<String> stream = Files.lines(path)) {

   stream.filter(line -> line.contains("ERROR"))
         .findFirst()
         .ifPresent(System.out::println);

} catch (IOException ioe) {
   // handle the exception
}
```



Build a Stream from a regex

Building a Stream from a regexp

```
// book is a looooooooooong String
Stream<String> words =
    Pattern
        .compile(" ")
        .splitAsStream(book) ;
```

Flatmap: Files.lines + regex

Splitting a text files into words

```
Stream<String> streamOfLines = Files.lines(path);

Function<String, Stream<String>> splitter =
   line -> Pattern.compile(" ").splitAsStream(line);

long numberOfWords =
streamOfLines.flatMap(splitter)
             .map(String::toLowerCase)
             .distinct()
             .count();
```

Why is it important for Java EE?

Given this JSON file

```
[
  {
    "name":"Duke",
    "gender":"M",
    "phones":{
      "home":"650-123-4567",
      "mobile":"650-111-2222"
    }
  }
]
```

Why is it important for Java EE?

And some JSONB magic:

```
Map<String, Long> names =
    contacts.getValue(JsonObject.class).stream()
        .filter(x -> "F".equals(x.getString("gender")))
        .map(x -> (x.getString("name"))
        .collect(
            Collectors.groupingBy(
                Function.identity(),
                Collectors.counting()
            )
        );
```

Why is it important for Java EE?

We can event collect into a JSon Array

```
JsonArray names =
   contacts.getValue(JsonObject.class).stream()
      .filter(x -> "F".equals(x.getString("gender")))
      .map(x -> (x.getString("name"))
      .collect(
         JsonCollectors.toJsonArray()
      );
```

Optional

Optional

Optional has been created to tell that this method could return « no value »

Ex: max(), min()

Optional

Optional has been created to tell that this method could return « no value »

Ex: max(), min(), average()

Optional

An Optional is a wrapping type, that can be empty

How can I build one?

```
Optional<String> opt = Optional.<String>empty() ;
```

```
Optional<String> opt = Optional.of("one") ; // not null
```

```
Optional<String> opt = Optional.ofNullable(s) ; // may be null
```

Optional

An Optional is a wrapping type, that can be empty

How can I use it?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {
   ...
}
```

Optional

An Optional is a wrapping type, that can be empty

How can I use it?

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {
   ...
}
```

```
String s = opt.orElse("") ; // this is a default value that
                            // is valid for our application
```

Optional

An Optional is a wrapping type, that can be empty

How can I use it?

```
String s = opt.orElseThrow(MyException::new) ; // lazy initialization
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

```
Optional<T> filter(Predicate<T> mapper) ;
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

```
Optional<T> filter(Predicate<T> mapper) ;
```

```
Optional<U> map(Function<T, U> mapper) ;
```

Optional: more patterns

An Optional can be seen as a special Stream with zero or one element

```
void ifPresent(Consumer<T> consumer) ;
```

```
Optional<T> filter(Predicate<T> mapper) ;
```

```
Optional<U> map(Function<T, U> mapper) ;
```

```
Optional<U> flatMap(Function<T, Optional<U>> mapper) ;
```

Optional: new patterns sighted!

```
public class NewMath {

   public static Optional<Double> inv(Double d) {

      return d == 0.0d ? Optional.empty() :
                         Optional.of(1/d) ;
   }

   public static Optional<Double> sqrt(Double d) {

      return d < 0.0d ? Optional.empty() :
                        Optional.of(Math.sqrt(d)) ;
   }
}
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.forEach(
   d1 -> NewMath.inv(d1)                                  // Optional<Double>
                .flatMap(d2 -> NewMath.sqrt(d2)) // Optional<Double>
                .ifPresent(result::add)
) ;
```

```
doubles : [-1.0, 0.0, 1.0]
result  : [1.0]
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.forEach(
   d1 -> NewMath.inv(d1)                                  // Optional<Double>
                .flatMap(d2 -> NewMath.sqrt(d2)) // Optional<Double>
                .ifPresent(result::add)
) ;
```

```
doubles : [-1.0, 0.0, 1.0]
result  : [1.0]
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.forEach(
   d1 -> NewMath.inv(d1)                                  // Optional<Double>
                .flatMap(d2 -> NewMath.sqrt(d2)) // Optional<Double>
                .ifPresent(result::add)                   // Baaaaad pattern ☹
) ;
```

```
doubles : [-1.0, 0.0, 1.0]
result  : [1.0]
```

Optional: new patterns sighted!

```
Function<Double, Optional<Double>> f =
 d -> NewMath.inv(d)                              // Optional<Double>
             .flatMap(d -> NewMath.sqrt(d))// Optional<Double>
```

Optional: new patterns sighted!

```
d -> NewMath.inv(d)                          // Optional<Double>
              .flatMap(NewMath::sqrt)        // Optional<Double>
              .map(Stream::of)               // Optional<Stream<Double>>
```

Optional: new patterns sighted!

```
d -> NewMath.inv(d)                          // Optional<Double>
              .flatMap(NewMath::sqrt)        // Optional<Double>
              .map(Stream::of)               // Optional<Stream<Double>>
              .orElse(Stream.empty())        // Stream<Double>
```

Optional: new patterns sighted!

```
Function<Double, Stream<Double>> f =
d -> NewMath.inv(d)                          // Optional<Double>
                .flatMap(NewMath::sqrt)      // Optional<Double>
                .map(Stream::of)             // Optional<Stream<Double>>
                .orElse(Stream.empty()) ;    // Stream<Double>
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.stream()
        .flatMap(
           d -> NewMath.inv(d)                  // Optional<Double>
              .flatMap(NewMath::sqrt)           // Optional<Double>
              .map(Stream::of)                  // Optional<Stream<Double>>
              .orElse(Stream.empty())           // Stream<Double>
        )                                       // Stream<Double>
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;
List<Double> result = new ArrayList<>() ;

doubles.stream()
       .flatMap(
          d -> NewMath.inv(d)                  // Optional<Double>
              .flatMap(NewMath::sqrt)          // Optional<Double>
              .map(Stream::of)                 // Optional<Stream<Double>>
              .orElse(Stream.empty())          // Stream<Double>
       )                                       // Stream<Double>
       .collect(Collectors.toList()) ;
```

Optional: new patterns sighted!

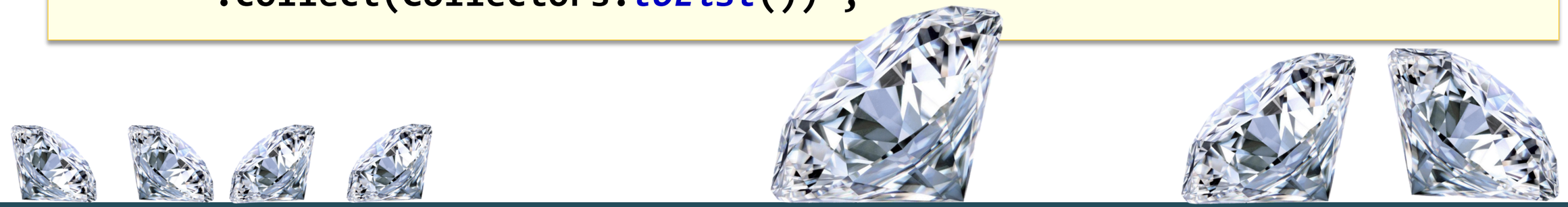
```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;

List<Double> result =
doubles.stream()
        .flatMap(
           d -> NewMath.inv(d)                  // Optional<Double>
              .flatMap(NewMath::sqrt)           // Optional<Double>
              .map(Stream::of)                  // Optional<Stream<Double>>
              .orElse(Stream.empty())           // Stream<Double>
        )                                       // Stream<Double>
        .collect(Collectors.toList()) ;
```

Optional: new patterns sighted!

```
List<Double> doubles = Arrays.asList(-1d, 0d, 1d) ;

List<Double> result =
doubles.stream().parallel()
        .flatMap(
           d -> NewMath.inv(d)                  // Optional<Double>
              .flatMap(NewMath::sqrt)           // Optional<Double>
              .map(Stream::of)                  // Optional<Stream<Double>>
              .orElse(Stream.empty())           // Stream<Double>
        )                                       // Stream<Double>
        .collect(Collectors.toList()) ;
```



Concurrent HashMap

ConcurrentHashMap

The old ConcurrentHashMap V7 has been removed

Thread safe

No locking ≠ ConcurrentHashMap V7

New methods (a lot…)

ConcurrentHashMap

6000 lines of code

ConcurrentHashMap

6000 lines of code

54 member classes

ConcurrentHashMap

6000 lines of code

54 member classes

FYI: 58 classes in java.util.concurrent

ConcurrentHashMap

6000 lines of code

54 member classes

FYI: 58 classes in java.util.concurrent

New patterns!

ConcurrentHashMap

Forget about size()

```
int   count = map.size() ;           // should not be used

      count = map.mappingCount() ; // new method
```

ConcurrentHashMap

Forget about size()

```
int   count = map.size() ;          // should not be used

long count = map.mappingCount() ; // new method
```

ConcurrentHashMap

Search() method

```
ConcurrentHashMap<Integer, String> map = ... ;

map.search(10L, (key, value) -> value.length() < key) ;
```

search(), searchKey(), searchValue(), searchEntry()

Returns the 1st element that matches the predicate

ConcurrentHashMap

Search() method

```
ConcurrentHashMap<Integer, String> map = ... ;

map.search(10L, (key, value) -> value.length() < key) ;
```

search(), searchKey(), searchValue(), searchEntry()

Returns the 1st element that matches the predicate

ConcurrentHashMap

Search() method

```
ConcurrentHashMap<Integer, String> map = ... ;

map.search(10L, (key, value) -> value.length() < key) ;
```

If there are more than 10 elements, then the search will be conducted in parallel!

ConcurrentHashMap

Search() method

```
ConcurrentHashMap<Integer, String> map = ... ;

map.search(10L, (key, value) -> value.length() < key) ;
```

If there are more than 10 elements, then the search will be conducted in parallel!

One can pass 0 or Long.*MAX_VALUE*

ConcurrentHashMap

ForEach

```
ConcurrentHashMap<Integer, String> map = ... ;

map.forEach(10L,
            (key, value) ->
            System.out.println(String.join("->", key, value))
) ;
```

forEach(), forEachKey(), forEachEntries()

ConcurrentHashMap

Reduction

```
ConcurrentHashMap<Integer, String> map = ... ;

map.reduce(10L,
           (key, value) -> value.getName(), // transformation
           (name1, name2) -> name1.length() > name2.length() ?
                             name1 : name2) // reduction
) ;
```

reduce(), reduceKey(), reduceEntries()

No ConcurrentHashSet

But…

```
Set<String> set = ConcurrentHashMap.newKeySet() ;
```

No ConcurrentHashSet

But…

```
Set<String> set = ConcurrentHashMap.newKeySet() ;
```

Creates a *concurrent hashmap* in which the values are Boolean.*TRUE*

Acts as a concurrent set

Parallelism

A warning on parallelism

All parallel operations (Streams, ConcurrentHashMap) take place in the *common ForkJoinPool*

The common ForkJoinPool takes all the available cores / CPUs

A warning on parallelism

All parallel operations (Streams, ConcurrentHashMap) take place in the *common ForkJoinPool*

The common ForkJoinPool takes all the available cores / CPUs

If you want to preserve your application server:

```
System.setProperty(
   "java.util.concurrent.ForkJoinPool.common.parallelism", "1") ;
```

Completable Future

CompletableFuture

New addition to j.u.concurrent

Allow to chain asynchronous tasks

Use case: test the asynchronous calls in the Servlet API

Creation of an asynchronous task

The Jersey way to create an asynchronous call

```
@Path("/resource")
public class AsyncResource {
    @GET
    public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

        new Thread(new Runnable() {
            public void run() {
                String result = longOperation();
                asyncResponse.resume(result);
            }
        }).start();
    }
}
```

Creation of an asynchronous task

(let us fix this code, this is Java 8)

```
@Path("/resource")
public class AsyncResource {
    @GET
    public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

        new Thread(() -> {
            String result = longOperation();
            asyncResponse.resume(result);
        }).start();
    }
}
```

How to test it?

We want to check if the `result` object
is passed to the `resume()` method of the `asyncResponse`

It is a very basic test, but tricky to write since we are in an asynchronous world

We have mocks for that!

How to test it?

We can inject a mock `AsyncResponse`, even mock the result

Then verify the correct interaction:

```
Mockito.verify(mockAsyncResponse).resume(result);
```

But we need to verify this once the `run()` method has been called…

How to test it?

This is where `CompletionStage` come to the rescue!

```
@Path("/resource")
public class AsyncResource {

   @Inject ExecutorService executor;

   @GET
   public void asyncGet(@Suspended final AsyncResponse asyncResponse) {

       executor.submit(() -> {
          String result = longOperation();
          asyncResponse.resume(result);
       });
   }
}
```

How to test it?

```
@Path("/resource")
public class AsyncResource {

   @Inject ExecutorService executor;

   @GET
   public void asyncGet(@Suspended final AsyncResponse asyncResponse) {
       executeAsync(asyncResponse);
   }

   public CompletableFuture<Void> executeAsync(final AsyncResponse asyncResponse) {
       return CompletableFuture.runAsync(() -> {
          asyncResponse.resume(longOperation());
       }, executor);
   }
}
```

How to test it?

```
AsyncResource asyncResource = new AsyncResource();

asyncResource.executeAsync(mockAsyncResponse); // returns the CompletableFuture
   .thenRun(() -> {                            // then execute this Runnable
      Mockito.verify(mockAsyncResponse).resume(result);
   }
);
```

How to test it?

Be careful of visibility issues

1) It’s better to run this in the same thread
2) Since the mocks are used and checked in this thread, create them in this thread too

How to test it?

So the complete pattern becomes this one

1) First we create our mocks

```
String result = Mockito.mock(String.class);
AsyncResponse response = Mockito.mock(AsyncResponse.class);

Runnable train =  () -> Mockito.doReturn(result).when(response).longOperation();
Runnable verify = () -> Mockito.verify(response).resume(result);
```

How to test it?

So the complete pattern becomes this one

2) Then we create the call & verify

```
Runnable callAndVerify = () -> {
   asyncResource.executeAsync(response).thenRun(verify);
}
```

How to test it?

So the complete pattern becomes this one

3) Then we create the task

```
ExecutorService executor = Executors.newSingleThreadExecutor();

AsynResource asyncResource = new AsyncResource();
asyncResource.setExecutorService(executor);

CompletableFuture
    .runAsync(train, executor)                    // this trains our mocks
    .thenRun(asyncVerify);                        // this verifies our mocks
```

How to test it?

Since a `CompletableFuture` is also a `Future`, we can fail with a timeout if the test does not complete fast enough

```
ExecutorService executor = Executors.newSingleThreadExecutor();

AsynResource asyncResource = new AsyncResource();
asyncResource.setExecutorService(executor);

CompletableFuture
    .runAsync(train, executor)                    // this trains our mocks
    .thenRun(asyncVerify)                         // this verifies our mocks
    .get(10, TimeUnit.SECONDS);
```


Thank you!


Q/A
#SE84EE
@Delabasee @JosePaumard