50 new things we can do with Java 8

DEVOXX™

@JosePaumard

0x50 new things we can do with Java 8

@JosePaumard

50 new things we can do with Java 8

DEVOXX™

@JosePaumard

José PAUMARD

MCF Um. Paris 13

PhD Appl
C.S.

Open source dév.

Indépendant

José PAUMARD

Open source dév.

Indépendant

José PAUMARD

Paris JUG

Devoxx FRANCE

José PAUMARD

Paris JUG

pluralsight
hardcore dev and IT training

Parleys

Microsoft Virtual Academy

Questions?

#50new8

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, TemporalUnit.SECONDS) ;
```

Date: Duration

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

Date: Duration

The precision of the time line is 10^{-9} s (nanosecond)
You think it's high precision?

Date: Duration

The precision of the time line is 10^{-9} s (nanosecond)
You think it's high precision?

Well the λ of the Higgs boson is 10^{-23} s

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(TemporalAdjuster.next(DayOfWeek.SUNDAY)) ;
```

Date: TemporalAdjuster

Arithmetic with local dates

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

LocalDate nextSunday =
   now.with(TemporalAdjuster.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(TemporalAdjuster.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

Factory class: DateTimeFormatter

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

Date: bridges with java.util.Date

Factory class: DateTimeFormatter

```
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

Factory class: DateTimeFormatter

```
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

Factory class: DateTimeFormatter

```
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
```

String

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()
 .map(Character::toUpperCase)
 .forEach(System.out::println);
```

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()
 .map(Character::toUpperCase)
 .forEach(System.out::println);
```

```
> 72
69
76
76
79
```

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
 .map(Character::toUpperCase)
 .forEach(System.out::println);          // Prints an int!
```

```
> 7269767679
```

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
```

String: regular expressions

Building a Stream from a regexp

```
// book is a looooooooong String
Stream<String> words =
   Pattern
      .compile("[^\\p{javaLetter}]")
      .splitAsStream(book) ;
```

String: concatenation

The newbie writes:

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

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

String: concatenation

The ignorant tells him to write:

```
StringBuffer sb1 = new StringBuffer("hello") ;
sb1.append(" world") ;

String s3 = sb1.toString() ;
```

(and is wrong)

String: concatenation

The seasonned dev tells him to write:

```
StringBuilder sb1 = new StringBuilder("hello") ;
sb1.append(" world") ;

String s3 = sb1.toString() ;
```

(and is wrong too…)

String: concatenation

Because the newbie is right (even if he doesn't know)

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

```
LINENUMBER 10 L2
   NEW java/lang/StringBuilder
   DUP
   ALOAD 1
   INVOKESTATIC java/lang/String.valueOf(Ljava/lang/Object;)Ljava/lang/String;
   INVOKESPECIAL java/lang/StringBuilder.<init>(Ljava/lang/String;)V
   LDC " "
   INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
   ALOAD 2
   INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
   INVOKEVIRTUAL java/lang/StringBuilder.toString()Ljava/lang/String;
   ASTORE 3
```

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
```

Numbers

Numbers: new methods

Method max, min, sum

```
// max method available on number wrapper classes
long max = Long.max(1L, 2L);
long sum = Long.sum(1L, 2L);
```

Numbers: new methods

Method max, min, sum

```
// max method available on number wrapper classes
long max = Long.max(1L, 2L);
long sum = Long.sum(1L, 2L);
```

Is it really *that* useful?

Numbers: new methods

Method max, min, sum

```
// max method available on number wrapper classes
long max = Long.max(1L, 2L);
long sum = Long.sum(1L, 2L);
```

Is it really *that* useful?

```
persons.stream()
        .map(Person::getAge)
        .reduce(0, (a1, a2) -> Integer.max(a1, a2));
```

Numbers: new methods

Method max, min, sum

```
// max method available on number wrapper classes
long max = Long.max(1L, 2L);
long sum = Long.sum(1L, 2L);
```

Is it really *that* useful? Nice to write method references

```
persons.stream()
       .map(Person::getAge)
       .reduce(0, Integer::max);
```

Numbers: new methods

Method max, min, sum

```
// max method available on number wrapper classes
long max = Long.max(1L, 2L);
long sum = Long.sum(1L, 2L);
```

```
persons.stream()
        .mapToInt(Person::getAge)
        .max();
```

Numbers: new methods

Method max, min, sum

```
// max method available on number wrapper classes
long max = Long.max(1L, 2L);
long sum = Long.sum(1L, 2L);
```

```
persons.stream()
        .max(Comparator.comparingBy(Person::getAge));
```

Numbers: new methods

Hashcode computation

```
// JDK 7
long l = 3141592653589793238L;
int hash =  new Long(l).hashCode(); // -1985256439
```

Numbers: new methods

Hashcode computation

```
// JDK 7
long l = 3141592653589793238L;
int hash =  new Long(l).hashCode(); // -1985256439
```

```
// JDK 8
long l = 3141592653589793238L;
int hash =  Long.hashCode(l); // -1985256439
```

I/O

I/O: reading text files

Stream extends AutoCloseable

```
// 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
}
```

I/O: reading text files

Files.list returns the content of the directory

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("c:", "windows");
try (Stream<Path> stream = Files.list(path)) {

   stream.filter(path -> path.toFile().isDirectory())
         .forEach(System.out::println);

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

I/O: reading a tree of subdirs

Files.walk returns the content of the subtree

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("c:", "windows");
try (Stream<Path> stream = Files.walk(path)) {

   stream.filter(path -> path.toFile().isDirectory())
         .forEach(System.out::println);

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

I/O: reading a tree of subdirs

Files.walk can limit the depth of exploration

```
// Java 7 : try with resources and use of Paths
Path path = Paths.get("c:", "windows");
try (Stream<Path> stream = Files.walk(path, 2)) {

   stream.filter(path -> path.toFile().isDirectory())
         .forEach(System.out::println);

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

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
```

Comparator

Comparator!

What else?

```
Comparator.naturalOrder() ;
```

Comparator!

What else?

```
Comparator.naturalOrder()

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

Comparator!

```
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

   INSTANCE; // What is this???





}
```

Comparator!

```
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

    INSTANCE; // OMG: a s

}
```

```
public class MySingleton {

    INSTANCE;

    private MySingleton() {}

    public static MySingleton getInstance() {
        // some buggy double-checked locking code
        return INSTANCE;
    }
}
```

Comparator!

```
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

    INSTANCE;

    public int compare(Comparable<Object> c1, Comparable<Object> c2) {
        return c1.compareTo(c2);
    }




}
```

Comparator!

```
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

    INSTANCE;

    public int compare(Comparable<Object> c1, Comparable<Object> c2) {
        return c1.compareTo(c2);
    }

    public Comparator<Comparable<Object>> reversed() {
        return Comparator.reverseOrder();
    }
}
```

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());
```

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()) ;
```

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) -> HashMap::new
).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)
) ;
```

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 = ... ;
```

Parallel Arrays

Parallel Arrays

Arrays.parallelSetAll

```
long [] array = new long [...] ;

Arrays.parallelSetAll(array, index -> index % 3) ;

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

Parallel Arrays

Arrays.parallelPrefix: fold right

```
long [] array = new long [...] ;

Arrays.parallelPrefix(array, (l1, l2) -> l1 + l2) ;

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

```
long [] array = {1L, 1L, 1L, 1L} ;

> [1, 2, 3, 4]
```

Parallel Arrays

Arrays.parallelSort: in place sorting

```
long [] array = new long [...] ;

Arrays.parallelSort(array) ;

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

Completable Future

CompletableFuture

Extension of Future

```
CompletableFuture<String> page =
   CompletableFuture.supplyAsync(



   ) ;
```

CompletableFuture

Extension of Future

```
CompletableFuture<String> page =
   CompletableFuture.supplyAsync(
      () ->
      readWebPage(url) // returns String
   ) ;
```

CompletableFuture

Now we can create pipelines

```
CompletableFuture.supplyAsync(
   () ->
   readWebPage(url)
)
.thenApply(
   content -> getImages(content) // returns a List<Image>
) ; // returns a CompletableFuture<List<Image>>
```

CompletableFuture

Now we can create pipelines

```
CompletableFuture.supplyAsync(
    () ->
    readWebPage(url)
)
.thenApply(
    content -> getImages(content) // returns a List<Image>
)
.thenAccept(
    images -> images.forEach(System.out::println)
);
```

CompletableFuture

thenCompose(): does not wrap the result in a CF

```
CompletableFuture.supplyAsync(
    () ->
    readWebPage(url)
)
.thenCompose(
    content -> getImages(content) // returns List<Image>
)
```

CompletableFuture

allOf: returns when all the tasks are done (see also anyOf)

```
CompletableFuture.allOf(
    CompletableFuture.supplyAsync(
        () ->
        readWebPage(url)
    )
    .thenCompose(content -> getImages(content))
    .thenApply(image -> writeToDisk(image))
)
.join() ;
```

CompletableFuture

thenCombine: can combine more than one CF

```
CompletableFuture cf1 = ... ;
CompletableFuture cf2 = ... ;

cf1.thenCombine(cf2, (b1, b2) -> b1 & b2) ; // can combine
                                            // the results of CFs
```

The λ is applied once the two CF have completed

CompletableFuture

thenCombine: can combine more than one CF

```
CompletableFuture cf1 = ... ;
CompletableFuture cf2 = ... ;

cf1.thenCombine(cf2, (b1, b2) -> b1 & b2) ; // can combine
                                            // the results of CFs
```

The λ is applied once the two CF have completed
Also: thenAcceptBoth, runAfterBoth

CompletableFuture

applyToEither: takes the first available result

```
CompletableFuture cf1 = ... ;
CompletableFuture cf2 = ... ;

cf1.applyToEither(cf2, (b) -> ...) ; // applies to the first
                                     // CF that returns
```

CompletableFuture

applyToEither: takes the first available result

```
CompletableFuture cf1 = ... ;
CompletableFuture cf2 = ... ;

cf1.applyToEither(cf2, (b) -> ...) ; // applies to the first
                                     // CF that returns
```

acceptEither, runAfterEither

Concurrence

Atomic variables

From Java 5:

```
AtomicLong atomic = new AtomicLong() ;
long l1 = atomic.incrementAndGet() ;
```

Atomic variables

Java 8 brings:

```
AtomicLong atomic = new AtomicLong() ;
long l1 = atomic.incrementAndGet() ;

long l2 = atomic.updateAndGet(l -> l*2 + 1) ;
```

Atomic variables

Java 8 brings:

```
AtomicLong atomic = new AtomicLong() ;
long l1 = atomic.incrementAndGet() ;

long l2 = atomic.updateAndGet(l -> l*2 + 1) ;

long l3 = atomic.accumulateAndGet(12L, (l1, l2) -> l1 % l2) ;
```

LongAdder

From Java 8:

```
LongAdder adder = new LongAdder() ;

adder.increment() ; // in a thread
adder.increment() ; // in a 2nd thread
adder.increment() ; // in a 3rd thread

long sum = adder.sum() ;
```

LongAccumulator

Same thing, with λ:

```
LongAccumulator accu =
   new LongAccumulator((l1, l2) -> Long.max(l1, l2), 0L) ;

accu.accumulate(value1) ; // in a thread
accu.accumulate(value2) ; // in a 2nd thread
accu.accumulate(value2) ; // in a 3rd thread

long sum = accu.longValue() ;
```

StampedLock

A regular lock, with optimistic read

```
StampedLock sl= new StampedLock() ;
```

```
long stamp = sl.writeLock() ;
try {
   ...
} finally {
   sl.unlockWrite(stamp) ;
}
```

```
long stamp = sl.readLock() ;
try {
   ...
} finally {
   sl.unlockRead(stamp) ;
}
```

StampedLock

A regular lock, with optimistic read

```
StampedLock sl= new StampedLock() ;
```

```
long stamp = sl.writeLock() ;
try {
   ...
} finally {
   sl.unlockWrite(stamp) ;
}
```

```
long stamp = sl.readLock() ;
try {
   ...
} finally {
   sl.unlockRead(stamp) ;
}
```

Exclusive read / write, but…

StampedLock

A regular lock, with optimistic read

```
StampedLock sl= new StampedLock() ;
```

```
long stamp = sl.tryOptimisticRead() ;
// here we read a variable that can be changed in another thread
if (lock.validate(stamp)) {
   // the read was not concurrent
} else {
   // another thread acquired a write lock
}
```

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

Thank you!

DEVOXX™
Q/A
#50new8
@JosePaumard