(5) Java Advanced

Nico Ludwig (@ersatzteilchen)

TOC

- (5) Java Advanced
  - Flatting of *Stream*s
  - Grouping of *Stream*s with *Map*s and Groups
  - Grouping of *Stream*s with Partitioning
  - Downstream *Collector*s
  - Subgroups
  - *Optional* and safe Object Navigation
  - *Optional*'s fp-style Operations
  - *Optional*'s fluent Interface
  - *Optional*'s primitive Specializations (*OptionalInt*)
  - Is *Optional* the "better null"?
  - *Stream*'s min and max Elements
  - Grouping to min and max Elements
  - Teeing of *Collector*s
  - Statistical *Collector*s
  - Excursus: creating own *Collector*s
  - *Stream* Collecting vs *Stream* Reduction

- Cited literature:
  - Java 8 in Action, Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Flatting of Streams – Part I

- Let's assume a *List<Person>*, *primaryPersons*, and each *Person* has a *List<Person>* as friends:

```
Person patricia = new Person("Patricia", 38);
patricia.setFriends(List.of(new Person("Henry", 41)));
Person bonnie =   new Person("Bonnie", 37);
bonnie.setFriends(List.of(new Person("Roger", 33), new Person("Bill", 40)));
Person conny = new Person("Conny", 32);
conny.setFriends(List.of(new Person("Gary", 67), new Person("Lex", 54)));

List<Person> primaryPersons = List.of(patricia, bonnie, conny);
```

Person
- friends : List<Person>
+ getFriends() : List<Person> + setFriends(friends : List<Person>)

  - Further assume following task: get a *List* of all *Person*s and their friends!

- What we need is a way to get <u>all elements contained in Lists</u> <u>in yet another List</u>, i.e. we need a flat *List* of *Person*s.
  - Such an operation, a <u>flattening</u> operation, is very common in fp. The *Stream* operator <u>*Stream.flatMap()*</u> does this job for us:

«interface» **Stream** (T)
+ <R> flatMap(mapper : Function<? super T, ? extends Stream<? extends R>>) : Stream<R>

  - *Stream.flatMap()*'s signature looks tricky, but it is really simple to use:

```
List<Person> allFriends = primaryPersons.stream().flatMap(person -> person.getFriends().stream()).collect(Collectors.toList());
List<Person> allPersons = Stream.concat(primaryPersons.stream(), allFriends.stream()).collect(Collectors.toList());
```

- Let's dissect what we have done here.

Flatting of Streams – Part II

patricia: *Person.name ("Patricia")*, *Person.age (38)*, *friends*: new Person("Henry", 41)
bonnie: *Person.name ("Bonnie")*, *Person.age (37)*, *friends*: new Person("Roger", 33), new Person("Bill", 40)
conny: *Person.name ("Conny")*, *Person.age (32)*, *friends*: new Person("Gary", 67), new Person("Lex", 54)
primaryPersons

- *Stream.flatMap()*'s lambda accepts a single *Stream* element and returns another *Stream*, so each element produces many elements.

```
List<Person> allFriends = primaryPersons.stream().flatMap(person -> person.getFriends().stream()).collect(Collectors.toList());
```

  - In our case, *Stream.flatMap()* returns the *Stream* of friends (*Stream<Person>*) of each *Person*. It yields this result:

new Person("Henry", 41)	new Person("Roger", 33)	new Person("Bill", 40)	new Person("Gary", 67)	new Person("Lex", 54)

allFriends

  - This is the flat *List<Person>* of all friends.

- Because we want to get all Persons, we have to concatenate *primaryPersons* with *allFriends*:

```
List<Person> allPersons = Stream.concat(primaryPersons.stream(), allFriends.stream()).collect(Collectors.toList());
```

new Person("Patricia", 38)	new Person("Bonnie", 37)	new Person("Conny", 32)	new Person("Henry", 41)	new Person("Roger", 33)	new Person("Bill", 40)	new Person("Gary", 67)	new Person("Lex", 54)

allPersons

- There also exists *Stream.flatMapToInt()*, *Stream.flatMapToLong()* and *Stream.flatMapToDouble()*.

Grouping of Streams with Maps – Part I

- We have to re-discuss *Collector*s. *Collector*s unleash their power, when we use them to group data.
  - Grouping means, that data is put into distinct groups after a certain criterion.

- Before we focus on grouping, we discuss a related operation: collecting *Stream* elements to *Map*s.
  - Therefor we use the *Collector* created by *Collectors.toMap()*.

Collectors
+ toMap(keyMapper : Function<T, K>, valueMapper : Function<T, U>) : Collector<T, ?, Map<K,U>>

- Let's create a *Map*, that associates each name (*String*) to its length:

```
Map<String, Integer> nameToLength = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
        .collect(Collectors.toMap(name -> name, name -> name.length()));
// nameToLength = {"Samuel" : 6, "Pat" : 3,  "Ashley" : 6, "Marion" : 6, "Lisa" : 4}
```

- Yes, it is really simple, but we can have it even simpler:

```
Map<String, Integer> nameToLength = Stream.of("Ashley", "Lisa", "Samuel", "Pat", "Marion")
        .collect(Collectors.toMap(Function.identity(), String::length));
// nameToLength = {"Samuel" : 6, "Pat" : 3,  "Ashley" : 6, "Marion" : 6, "Lisa" : 4}
```

  - We can use *Function.identity()*, because the *keyMapper* just maps each name (a *String*) to itself.
  - That we can use the method reference *String::length* instead of the lambda *name -> name.length()* should be clear.

Grouping of Streams with Maps – Part II

- Next, let's collect more complex objects into a *Map*: a *Stream<Person>* to a *Map* of a *Person*'s name and the *Person*:

```
Map<String, Person> persons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
            .collect(Collectors.toMap(Person::getName, Function.identity()));
```

  - But, this one doesn't work:

```
// Invalid! IllegalStateException Duplicate key Ashley (attempted merging values name = Ashley, age = 67 and name = Ashley, age = 23)
```

- After reading the *IllegalStateException*'s message and having another look at the collect call the reason is clear:
  - We have multiple *Person*s with the same name ("Ashley"), which cannot be added as keys to a *Map*!
  - How can we collect multiple *Person*s of the same name into a *Map*?

- We have to address two things:
  - (1) We must use another value-type for the *Map*, namely a *List<Person>*, not only *Person*!
  - (2) We must handle and collect *Person*s, whose name was already collected as key, i.e. we must handle the "key clash".

- Let's have a look at this *Map-Collector* using a *mergeFunction*, with which we can handle these things:

Collectors
+ <T, K, U> toMap(keyMapper : Function<T, K>, valueMapper : Function<T, U>, mergeFunction : BinaryOperator<U>) : Collector<T, ?, Map<K,U>>

Grouping of Streams with Maps – Part III

Collectors
+ <T, K, U> toMap(keyMapper : Function<T, K>, valueMapper : Function<T, U>, mergeFunction : BinaryOperator<U>) : Collector<T, ?, Map<K,U>>

- … and how do we have to address the *List*-value type and key clashes? Let's inspect the code:

```
Map<String, List<Person>> persons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
            .collect(Collectors.toMap(Person::getName,                       // (1) keyMapper
                                      person -> {
                                        List<Person> ps = new LinkedList<>();  // (2) valueMapper
                                        ps.add(person);
                                        return ps;
                                      },
                                      (collectedPersons, newPersons) -> {    // (3) mergeFunction
                                        collectedPersons.addAll(newPersons);
                                        return collectedPersons;
                                      }
                    )
);
```

  - (1) The *keyMapper* stays the same, it is still the name of the *Person*.
  - (2) The *valueMapper* accepts a *Person* and puts this *Person* into a new *List*.
  - (3) The *mergeFunction* is called, if for the *Person* in question a key was already collected (there is already a *List* for the *Person*).
    - I.e. here, key clashes are handled.
    - *collectedPersons* contains the already collected *Persons*, *newPersons* contains the *List* with the specific *Person* introducing the clash.
    - The "merge activity" we do is just adding the new *Person* into the *List* of already collected *Persons* of the same name.
    - Here we have the power of the *mergeFunction*: we can do any more or less complex merge activity beyond just adding all together.

Grouping of Streams with Groups – Part I

```
Map<String, List<Person>> persons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
            .collect(Collectors.toMap(Person::getName,
                                      person -> {
                                        List<Person> ps = new LinkedList<>();
                                        ps.add(person);
                                        return ps;
                                      },
                                      (collectedPersons, newPersons) -> {
                                        collectedPersons.addAll(newPersons);
                                        return collectedPersons;
                                      }
                 )
);
```

- This way to put multiple elements of the same key into a *List* is pretty complex (not complicated, but complex).
  - *Map*s keeping multiple values under the same key are sometimes called multimaps.
  - Actually, putting multiple elements matching the same criterion is so a common operation, that it got extra support with *Stream*s.

- The *Map*-collection shown above is a so called grouping operation.
  - We can use the *Collector* created by *Collectors.groupingBy()* to get the same grouping as above with much less code:

```
Map<String, List<Person>> persons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                    .collect(Collectors.groupingBy(Person::getName));
```

Grouping of Streams with Groups – Part II

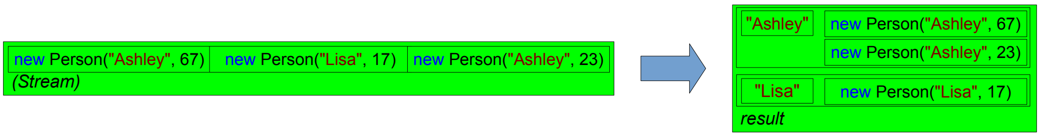
- Although grouping is a pretty mighty *Collector*, its setup is <u>super easy</u> to use:

Collectors
+ <T, K> groupingBy(classifier : Function<T, K>) : Collector<T, ?, Map<K, List<T>>>

- The terminology regarding grouping is also <u>more precise</u> in comparison to *Collector.toMap()*:
  - First and foremost, we also get a *Map*, more exactly a <u>multimap</u>, which maps <u>single keys to *List*s</u>.
  - And those individual entries of keys and *List*s make up the actual <u>groups</u>.
  - A key, which designates an individual group is called <u>class</u>. In our case, the class is the <u>name of each *Person*</u>.
  - The function, that provides the class for each group is therefore called <u>*classifier*</u>.

- Internally, grouping collects elements of the *Stream* into *List*s corresponding to the classifier.

```
Map<String, List<Person>> persons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                                          .collect(Collectors.groupingBy(Person::getName));
```



Grouping of Streams with Groups – Part III

- Effectively, following three snippets we have discussed up to here perform the <u>same grouping operation</u>:

```
// Traditional imperative implementation:
public static Map<String, List<Person>> collectPersons(List<Person> source) {
	Map<String, List<Person>> collected = new HashMap<>();
	for (Person person : source) {
		if (collected.containsKey(person.getName())) {
			collected.get(person.getName()).add(person);
		} else {
			collected.put(person.getName(), new ArrayList<>(List.of(person)));
		}
	}
	return collected;
}
```

```
// Functional implementation with the vavr library:
public static Map<String, List<Person>> collectPersons(List<Person> source, Map<String, List<Person>> collected) {
	return !source.isEmpty()
		? collectPersons(source.tail(),
			collected.containsKey(source.head().getName())
				? collected.replaceValue(source.head().getName(), collected.get(source.head().getName()).get().append(source.head()))
				: collected.put(source.head().getName(), List.of(source.head())))
		: collected;
}
```

```
// With Streams:
Map<String, List<Person>> result = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
				.collect(Collectors.groupingBy(Person::getName));
```

Grouping of Streams with Partitioning

- A simple form of grouping is partitioning.
  - Partitioning means to put the elements of the *Stream* into one of just two groups.
  - In other words: partitioning is a grouping with a boolean classifier and it results in a *Map* with a *Boolean* key type.

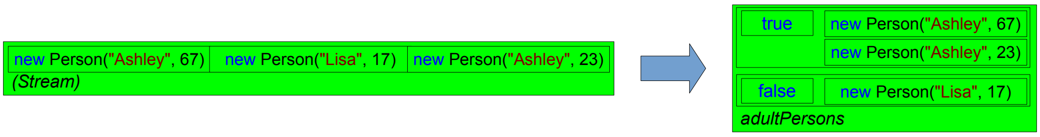
- To create *Person* groups of "being adult" and "being child" after the age, we could use *Collectors.groupingBy()*:

```
Map<Boolean, List<Person>> adultPersons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
        .collect(Collectors.groupingBy(person -> person.getAge() > 18));
```

  - Instead we can use the *Collector* created via *Collectors.partitioningBy()* and partition by the age:

```
Map<Boolean, List<Person>> adultPersons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
        .collect(Collectors.partitioningBy(person -> person.getAge() > 18));
```

  - Both *Collector*s provide the same result:



- Remarks:
  - Partitioning is slightly more efficient than grouping because it uses an optimized *Map* for *Boolean* keys.
  - The resulting *Map* of the partitioning operation will always contain entries for true and false.
    - If there are no entries for either key, the associated *List* will just be empty.

Downstream Collectors – Part I

- In many cases, we want to do <u>follow-up operations</u> with the data we got <u>after *Stream.collect()*</u>.
  - Consider following code, which determines <u>the count of elements per group</u>:

```
Map<String, List<Person>> persons = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                    .collect(Collectors.groupingBy(Person::getName));
for (Map.Entry<String, List<Person>> personsOfEqualName : persons.entrySet()) {
        System.out.printf("%d persons of name %s%n", personsOfEqualName.getValue().size(), personsOfEqualName.getKey());
}
```

```
Terminal
2 persons of name Ashley
1 persons of name Lisa
```

- *Collector*s can be built with so called <u>downstream *Collector*s</u>. E.g. an overload of *Collectors.groupingBy()* accepts such one:

Collectors
+ <T, K, A, D> groupingBy(classifier : Function<?, ?>, **downstream : Collector<?, A, D>**) : Collector<T, ?, Map<K, D>>

  - The idea of downstream *Collector*s is not directly visible: the idea is to <u>apply yet another *Collector* on a collected result</u>.
  - So, a downstream *Collector* is just <u>a *Collector*, which is applied on the result of another *Collector*</u>.

- We can use the *Collector* created by <u>*Collectors.counting()*</u> to apply counting of the groups' elements in the downstream:

```
Map<String, Long> namesToCount = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                    .collect(Collectors.groupingBy(Person::getName, Collectors.counting()));
for (Map.Entry<String, Long> nameToCount : namesToCount.entrySet()) {
        System.out.printf("%d persons of name %s%n", nameToCount.getValue(), nameToCount.getKey());
}
```

```
Terminal
2 persons of name Ashley
1 persons of name Lisa
```

Downstream Collectors – Part II

- Downstream *Collector*s allow follow-up collects on a collected result without leaving the fluent expression and pipeline.
  - Mind, that *Stream.collect()* is a terminal operation, with downstream *Collector*s we can do more operations but "stay intermediate".

- *Collectors* offers many simple factories for *Collector*s being used as downstream *Collector*s or accepting downstream *C*.:

```
Collectors
+ <T> counting() : Collector<T, ?, Long>
+ <T> averagingInt(mapper : ToIntFunction<?>) : Collector<T, ?, Double>
+ <T, A, R> filtering(predicate : Predicate<?>, downstream : Collector<?, A, R> ) : Collector<T, ?, R>
+ <T, U, A, R> mapping(mapper : Function<?, ?>, downstream : Collector<?, A, R>) : Collector<T, ?, R>
+ <T, U, A, R> flatMapping(mapper : Function<? super T, ? extends Stream<?>>, downstream : Collector<? super U, A, R>) : Collector<T, ?, R>
+ <T> summingDouble(mapper : ToDoubleFunction<?>) : Collector<T, ?, Double>
+ <T> minBy(comparator : Comparator<?>) : Collector<T, ?, Optional<T>>
...
```

  - The usage of those is often not easy to grab and requires reading tutorials, documentation and "inspiration".
  - Many of the *Collector*s, offered by the companion class *Collectors* are only meant to be used as downstream *Collector*s.
    - Esp. those *Collector*s used for aggregate functions like counting, summing and averaging, we'll discuss aggregate functions for extremes in a minute.

- To open the perspective for downstream *Collector*s, we should know which *Collector*s primary accept them:

```
Collectors
+ <T, K, A, D> groupingBy(classifier : Function<?, ?>, downstream : Collector<?, A, D>) : Collector<T, ?, Map<K, D>>
+ <T, D, A> partitioningBy(predicate : Predicate<?>, downstream : Collector<?, A, D>) : Collector<T, ?, Map<Boolean, D>>
```

  - It makes sense, because grouping and partitioning are closely related.

- Aggregate function is a term from SQL, where functions can be applied on groups.

Downstream Collectors – Part III

- For the time being we will show applications of *Collectors.averagingInt()* and *Collectors.filtering()*.
- Following grouping will downstream each *Person* name to the average age of its group:

```
Map<String, Double> personsNamesToAverageAge
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .collect(Collectors.groupingBy(Person::getName, Collectors.averagingInt(Person::getAge)));
```

"Ashley"	45.0
"Lisa"	17.0

personsNamesToAverageAge

- Following grouping will downstream only adult *Persons* into its name-group:

```
Map<String, List<Person>> adultPersons
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .collect(Collectors.groupingBy(Person::getName, Collectors.filtering(person -> 18 <= person.getAge(), Collectors.toList())));
```

  - We cascade 3 *Collectors*: *Collectors.groupingBy()* downstreams to *Collectors.filtering()*, which downstreams to *Collectors.toList()*:

"Ashley"	new Person("Ashley", 67)
	new Person("Ashley", 23)
"Lisa"	<empty *List*>

adultPersons

  - It works, because *Collections.filtering()* accepts a downstream *Collector*:

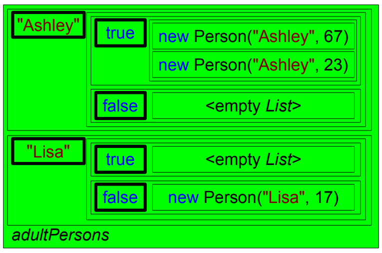
Collectors
+ <T, A, R> filtering(predicate, **downstream**) : Collector<T, ?, R>

Downstream Collectors – Subgroups – Part IV

- Another important aspect of downstream *Collector*s is, that we can create subgroups.
  - This expressions groups *Person*s after their names and then in the name groups, the *Person*s are grouped after their adulthood:

```
Map<String, Map<Boolean, List<Person>>> adultPersons
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .collect(Collectors.groupingBy(Person::getName, Collectors.partitioningBy(person -> 18 <= person.getAge())));
```

- The code expresses something like "group after name, then group (or partition) after adulthood".
  - Just think about how many cascaded loops can be saved by just applying this *Stream* expression! It yields a pretty complex result:



  - (If we had used *Collectors.groupingBy()* instead of *Collectors.partitioningBy()*, no empty Lists per partition would be created.)

Downstream Collectors – Finishers – Part V

- The Collector created by *Collectors.collectingAndThen()* can be used to apply a finisher function to a collected result.

Collectors
+ <T, A, R, RR> collectingAndThen(downstream : Collector<T, A, R>, finisher : Function<R, RR>) : Collector<T, A, RR>

  - E.g. we can put the resulting *Map* of a grouping into an unmodifiable *Map*:

```
Map<String, List<Person>> persons
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .collect(Collectors.collectingAndThen(Collectors.groupingBy(Person::getName), Collections::unmodifiableMap));
```

Further Tips using Collectors

- Having knowledge about the spectrum of predefined *Collector*s makes sense:
  - *Collector*s also work with parallel *Stream*s, esp. also for the thread safe concatenation of *String*s.
  - Downstream *Collector*s allow solving complex problems with pretty readable code.
    - Esp. aggregation *Collector*s and sub-grouping safes us a lot of cascaded loops!

- Tips using *Collector*s:
  - As soon as "doing something with the groups" is required, have a look into the predefined *Collector*s to safe loops!

- The JDK provides many predefined *Collector*s, nevertheless there are situations, in which we have to program our own.

Reviewing Object Navigation in Java

- Let's assume the classes *Person* and *Date* and create a *Person* object referring a *Date* object and other *Person* objects:

Person
+ getBirthday() : Date + setBirthday(birthday : Date) + getPromotions() : Date[] + setPromotions(promotions : Date[]) + getSuperior() : Person + setSuperior(superior : Person)

Date
+ getDay() : int + setDay(day : int) + getMonth() : int + setMonth(month : int) + getYear() : int + setYear(year : int)

```java
Person maggie = new Person();
maggie.setName("Maggie");

Date promotion = new Date();
promotion.setDay(1);
promotion.setMonth(3);
promotion.setYear(2001);

Person michael = new Person();
michael.setAge(54);
michael.setName("Michael");
michael.setPromotions(new Date[] {promotion});
michael.setSuperior(maggie);
```

- We can use dot-operators to access/modify objects following the references. This is called object navigation:

```java
// Read the name of Michael's superior:
System.out.println(michael.getSuperior().getName());
// Modifying the month of Michael's first promotion:
michael.getPromotions()[0].setMonth(12);
```

Safe object navigation

```java
// Print the name of Michael's superior to the console:
if (michael != null && michael.getSuperior() != null) {
        System.out.println(michael.getSuperior().getName());
}
// Modifying the month of Michael's first promotion:
if (michael != null && michael.getPromotions() != null && michael.getPromotions().length >= 1
        && michael.getPromotions()[0] != null) {
        michael.getPromotions()[0].setMonth(12);
}
```

  - But this is not safe:
    - If a reference is null in the navigation-chain and we're accessing it, we will get an *NullPointerException*.
    - If an array index does not exist in the navigation-chain and we're accessing it, we will get an *ArrayIndexOutOfBoundsException*.
  - To make safe object navigation, we have to add null-checks and array-bounds-checks.

Making Object Navigation safe with Optional

- Per se, safe object navigation is of course no problem, it is rather required to program null-aware code.
  - But there are syntactical flaws, that prove problematic with the *Stream* API.
- Safe object navigation must be done via multiple statements, because null-check and access are syntactically separated.
  - And in fp we want to write everything into a single expression ideally.
- Java's syntax wasn't extended, instead the class *Optional* was added, it leverages safe object navigation to a fluent API.
- Without further ado, we can formulate the just discusses safe object navigation with *Optional*:

```
// Print the name of Michael's superior to the console:
if (michael != null && michael.getSuperior() != null) {
	System.out.println(michael.getSuperior().getName());
}
// Modifying the month of Michael's first promotion:
if (michael != null && michael.getPromotions() != null && michael.getPromotions().length >= 1
	&& michael.getPromotions()[0] != null) {
	michael.getPromotions()[0].setMonth(12);
}
```

```
// Print the name of Michael's superior to the console:
Optional.ofNullable(michael)
		.map(Person::getSuperior)
		.ifPresent(superior -> System.out.println(superior.getName()));

// Modifying the month of Michael's first promotion:
Optional.ofNullable(michael)
		.map(Person::getPromotions)
		.filter(promotions -> promotions.length >= 1)
		.map(promotions -> promotions[0])
		.ifPresent(firstPromotion -> firstPromotion.setMonth(12));
```

  - The syntactic style makes clear, that the last call *ifPresent()* executes its lambda only if the full object navigation was successful.

Creating Optionals

- *Optional* is a generic class, whose instances wrap an object, which can be potentially null, i.e. being "optional".
  - Its most important methods support null-aware navigation without accessing the optional data until the last moment.
  - And this is possible, because these methods support a fluent interface: many methods return *Optional*s:

```
Optional                                                   T
+ <T> ofNullable(value : T) : Optional<T>
+ <T> empty() : Optional<T>
+ <U> map(mapper : Function<? super T, ? extends U>) : Optional<U>
+ filter(predicate : Predicate<? super T>) : Optional<T>
+ ifPresent(action : Consumer<? super T>)
+ isPresent() : boolean
+ isEmpty() : boolean
+ get() : T
```

- We can create *Optional*s from any reference.
  - If the reference in question could be a null-reference, we should use *Optional.ofNullable()*:

```
Optional<Person> optionalMichael = Optional.ofNullable(michael);
```

  - If we pass a null-reference to *Optional.ofNullable()*, the result will be *Optional.empty()*:

```
Optional<Person> optionalOfNullReference= Optional.ofNullable(null);
// optionalOfNullReference = Optional.empty()
```

  - If the reference in question is guaranteed to be no null-reference, we could use *Optional.of()*.
    - If we pass a null-reference to *Optional.of()*, an NPE will be raised, therefor *Optionals.ofNullable()* is the recommended way to go.

Checking an Optional's State and Accessing its Value

- The "unpresence" of a value can be expressed via *Optional.empty()*. Actually, *Optional.empty()* is kind of the "new null":

```
Optional<Person> optionalOfNullReference = Optional.empty();
```

- We can use the predicate *Optional.isPresent()* or its negation *Optional.isEmpty()* to check the presence of a value:

```
boolean isPresent = optionalOfNullReference.isPresent();
// isPresent = false
boolean isEmpty = optionalOfNullReference.isEmpty();
// isEmpty = true
```

  – In case *michael* was actually a null-reference, *optionalMichael.isPresent()* would return false:

```
boolean michaelIsNotNull = optionalMichael.isPresent();
```

- We can get the reference wrapped into the *Optional* with the method *optionalMichael.get()*:

```
Person michael2 = optionalMichael.get();
```

  – *Optional.get()* will never return a null-reference!
  – If *michael* was actually a null-reference, *Optional.get()* will throw an NPE.

Fp-Style Branching with Optional

- So *optionalMichael.get()* could throw a NPE, therefor it should only be called, if *optionalMichael.isPresent()* returns true.
  - We could use the combination of *Optional.isPresent()* and *Optional.get()* to access the wrapped reference null-safely:

```
if (optionalMichael.isPresent()) {
        Person michael = optionalMichael.get();
        System.out.println(michael.getName());
}
```

Optional T

+ ifPresent(action : Consumer<? super T>)

  - *Optional.ifPresent()* combines the check of presence and null-safe access into only one expression:

```
optionalMichael.ifPresent(michael -> System.out.println(michael.getName()));
```

  - Effectively, both solutions do the same; esp. if the wrapped reference is a null-reference just nothing will happen.

- With *Optional.ifPresentOrElse()* we can handle the if and else case of a reference's presence in an fp-style:

```
optionalMichael.ifPresentOrElse(
        /*if*/      michael -> System.out.printf("The director will be %s.%n", michael.getName()),
        /*else*/    () -> System.out.println("The director will be Peter."));
```

Optional T

+ ifPresentOrElse(action : Consumer<? super T>, emptyAction : Runnable)

  - For *Optional*, "else-case" means the function, that is called, if the *Optional* is empty.

Handling unpresence of Optional Values

- *Optional* also allows to handle the unpresence of a reference (empty *Optional*) nicely with a default value:

```
Person director
        = optionalMichael.orElseGet(() -> new Person("Peter", 54));
```

```
Optional<T>
+ orElse(other : T) : T
+ orElseGet(supplier : Supplier<? extends T>) : T
```

  - *Optional.orElse()* provides a predefined default value directly.
  - *Optional.orElseGet()* provides a default value lazily.

- *Optional* can also be used to throw an *Exception* instead of providing a default value in case of unpresence.

```
Person director
        = optionalMichael.orElseThrow(() -> new RuntimeException("Michael not found"));
```

```
Optional<T>
+ orElseThrow() : T
+ <X extends Throwable> orElseThrow(exceptionSupplier : Supplier<? extends X>) : T
```

  - *Optional.orElseThrow()* throws a *NoSuchElementException* in case of value-unpresence.
  - The other overload of *Optional.orElseThrow()* throws a user-created *Throwable*.

Using Optional's fluent Interface

- The operations of *Optional* we saw up to here exist more or less to evaluate the wrapped reference to a value.
- However, *Optional* unleashes its real power, when we keep the access to referenced data also in the expression.
  - *Optional.map()* is meant to access data of an *Optional* object. The trick: the method will put the accessed data also into an *Optional*:

Optional (T)
+ <U> map(mapper : Function<? super T, ? extends U>) : Optional<U>

  - *Optional.map()* allows to keep access to cascaded data in a single expression but fully null-aware:

```
if (michael != null && michael.getSuperior() != null) {
        System.out.println(michael.getSuperior().getName());
}
```

```
Optional.ofNullable(michael)
        .map(Person::getSuperior)
        .ifPresent(superior -> System.out.println(superior.getName()));
```

- *Optional*'s fluent API allows transparent accessing of deeply cascaded data: adding cascading-levels is super easy:

```
if (michael != null && michael.getSuperior() != null && michael.getSuperior().getName() != null) {
        System.out.println(michael.getSuperior().getName());
}
```

```
Optional.ofNullable(michael)
        .map(Person::getSuperior)
        .map(Person::getName)
        .ifPresent(System.out::println);
```

  - Instead of adding more control flow via extra conditions in the if statement, we just extend the *Optional* expression.
  - In opposite to *Stream.map()*, *Optional.map()*'s behavioral argument can be stateful!

Good to know
Java 9 introduces *Optional.stream()* to convert an *Optional* into a single-element or empty *Stream*.

- *Optional* is like a *Stream* with only one optional element.
  - *Optional*/*Stream.map()* and *Optional*/*Stream.filter()* correspond more or less exactly and *Optional.ifPresent()* corresponds to *Stream.forEach()*.

Optional's primitive Specializations (OptionalInt)

- Many of the operations possible with *Optional* are also interesting for primitive types, esp. when *Stream*s come into play.
- But *Optional* is a generic type and we already know, that generic types cannot have primitive types as type argument.
- The JDK solves this problems for us like it was solved for *Stream*s: there exist primitive *Optional* specializations.
  - Another aspect is of course, that we want to avoid unnecessary boxing operations.
- Namely the JDK offers the types *OptionalInt*, *OptionalDouble* and *OptionalLong*.

OptionalInt
+ of(value : int) : OptionalInt
+ ifPresent(action : IntConsumer)
+ isPresent() : boolean
+ orElse(other : int) : int
+ orElseThrow() : int
+ ifPresentOrElse(action : IntConsumer, emptyAction : Runnable)
+ getAsInt() : int
...

  - Those specialized versions of *Optional* lack *filter()* and *map()* methods, because they make only sense for cascading data.
  - The specializations use specialized functional interfaces for behavioral parameters like *IntConsumer*.
  - Instead of a common *get()* method, we have *getAsInt()*, *getAsDouble()* and *getAtLong()*, which return primitive values.

Is Optional the "better null"?

- Well, the question is: "What is the purpose of *Optional*?"
- Initially, many developers hoped, that *Optional* should more or less replace null.
  - In many other languages, also the JRE-based language Scala have such a type (*Option* in Scala) to "hide" the null idiom.
  - The believe is, that "null is bad" and people should strive to avoid null.
  - Advantage: using *Optional* in an API makes visible, where a value could be unpresent.
  - Advantage: we can tell error-cases from "unpresence"-cases: *Map.get()* returns null, if the key is not set or if its value is null.
- But *Optional* was not so intrinsically added into Java, i.e. it is no idiomatic replacement for null with language support etc.
  - *Optional* is only used as return type in a few interfaces and by no means pervasively!
  - Instead the JDK returns and accepts null-references since Java 1, such cannot be modified to feature a "cleaner" *Optional* concept.
  - => Pervasive replacement of dealing with null-references to *Optional*s would hurt Java's backward compatibility.
- And … what is the purpose of *Optional*?
  - *Optional* is not there to hide null-references, but to enable safe navigation with its fluent API.
  - The idea is to appreciate fluent chains of *Optional.map()*, *Optional.filter()*, *Optional.ifPresent()*, *Optional.orElse()* etc.

Tips using Optional

- Usage:
  - In new UDTs expose *Optional*s only as return type!
  - In a method's code you can of course use *Optional* freely to your liking.
  - When creating *Optional*s rather use *Optional.ofNullable()*.
  - *Optional.equals()* and *Optional.hashCode()* forward calls to the wrapped reference.

- Some warnings:
  - *Optional* can lead to expressive, but not necessarily more readable code!
    - In many cases in Java, esp. with *Collection*s and objects representing the problem domain, the null-object design pattern proves better.
    - I.e. it makes sense to underscore Java being an oo-language, when the oo-design of the problem domain comes into play.
  - Like with *Stream*s' fluent call chains, debugging can be more difficult due to deferred execution.
  - Identity comparisons of *Optional*s (esp. reference comparisons via ==) can have unpredictable results and should be avoided.
    - This is a hint, that the type *Optional* might get a more idiomatic role in future Java versions. Catchword: user defined value-types.

Picking Elements from potentially empty Streams

- After we have <u>laid the foundation with the type *Optional*</u>, we can discuss more *Stream* operations.

- Remember *Stream.anyMatch(),* it checks if any *Stream* element <u>matches the passed predicate</u>:

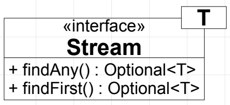
```
boolean anyPersonIsAdult
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .anyMatch(person -> person.getAge() >= 18);
```

  - *Stream.anyMatch()* just checks a predicate! If we want to <u>get</u> the element matching the predicate, we've to create another *Stream*.

- Instead of *Stream.anyMatch()*, we had to use *Stream.filter()*:

```
Stream<Person> adultPersons
            = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                    .filter(person -> person.getAge() >= 18);
```

  - But we only want to get <u>a single</u> element matching the criterion. Because it could be more that one <u>we can pick any or the first one</u>:




```
Optional<Person> optionalAdultPerson = adultPersons.findFirst();
optionalAdultPerson.map(Person::getName).ifPresent(name -> System.out.printf("%s is an adult person%n", name));
// >Ashley is an adult person
```

  - *Stream.findFirst()* returns an *Optional<T>* referencing the first *T* in the *Stream*, <u>the *Optional* is empty, if the *Stream* is empty</u>.
  - *Stream.findAny()* is free to select <u>any element</u>, <u>not strictly the first one</u>. <u>It can be more performant than *.findFirst()* for parallel *Streams*.</u>

- The nature of *Stream.findAny()* and *Stream.findFirst()* makes them <u>short cut operations</u>.

Picking min and max Elements from Streams

- The operators *Stream.min()* and *Stream.max()* also return *Optional*s:

```
«interface»
Stream                                                    T
+ max(comparator : Comparator<? super T>) : Optional<T>
+ min(comparator : Comparator<? super T>) : Optional<T>
```

```java
Optional<Person> oldestPerson
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .max((lhs, rhs) -> Integer.compare(lhs.getAge(), rhs.getAge()));
```

  - It makes sense, the problem is similar to the one with *Stream.findFirst()*: What should those operators return if the *Stream* is empty?
  - Answer: an empty *Optional*, on which operations like *Optional.ifPresent()* just do nothing.

- *Stream.min()* and *Stream.max()* accept *Comparator*s and the boilerplate code to create one from getters can be avoided.
  - *Comparator* provides a set of static methods to create a *Comparator* from only a key-extractor (getter), e.g. as method reference.

```
«interface»
Comparator                                                T
+ <T, U> comparing(keyExtractor : Function<? super T, ? extends U>, keyComparator : Comparator<? super U>) : Comparator<T>
+ <T, U extends Comparable<? super U>> comparing(keyExtractor : Function<? super T, ? extends U>) : Comparator<T>
+ <T> comparingInt(keyExtractor : ToIntFunction<? super T>) : Comparator<T>
+ <T> comparingLong(keyExtractor : ToLongFunction<? super T>) : Comparator<T>
+ <T> comparingDouble(keyExtractor : ToDoubleFunction<? super T>) : Comparator<T>
```

  - *Comparator*-generators are fairly simple to use:

```java
Optional<Person> oldestPerson
        = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .max(Comparator.comparingInt(Person::getAge));
```

Grouping to min and max Elements and Teeing of Collectors

- We can also create *Collector*s to collect extreme values via *Collectors.maxBy()* and *Collectors.minBy()* to *Optional*s:

```
Map<String, Optional<Person>> personsNamesToYoungestPersons
            = Stream .of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                      .collect(Collectors.groupingBy(Person::getName, Collectors.minBy(Comparator.comparingInt(Person::getAge))));
```

"Ashley"	new Person("Ashley", 23)
"Lisa"	new Person("Lisa", 17)

personsNamesToYoungestPersons

Collectors
+ <T> maxBy(comparator : Comparator<? super T>) : Collector<T, ?, Optional<T>> + <T> minBy(comparator : Comparator<? super T>) : Collector<T, ?, Optional<T>>

  - … those are the missing aggregate functions for extremes (max/min), when we use those *Collector*s as downstream *Collector*s for grouping.

- The *Collector* created with *Collectors.teeing()* (Java 12+) creates a *Collector* collecting the results of two independent *C*.

Collectors
+ <T, R1, R2, R> teeing(downstream1 : Collector<?, ?, R1>, downstream2 : Collector<?, ?, R2>, merger : BiFunction<?, ?, R>) : Collector<T, ?, R>

  - It looks more complicated as it is: the downstream *Collector*s provide their results and the *merger* puts the results into an object:

0	1
new Person("Lisa", 17)	new Person("Ashley", 67)

extremePersons

```
Person[] extremePersons
            = Stream .of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                      .collect(Collectors.teeing(
                        Collectors.minBy(Comparator.comparingInt(Person::getAge)),
                        Collectors.maxBy(Comparator.comparingInt(Person::getAge)),
                        (result1, result2) -> new Person[] {result1.get(), result2.get()}));
```

  - The resulting *Person[2]* just contains the youngest and oldest *Person*.
  - => In the *merger* we can basically create any kind of object combining the input *Collector*s fitting our needs.

Cascaded Collector Teeing

- We can use teeing *Collector*s as downstream *Collector*s of yet other teeing *Collector*s to collect yet more data at once:

```
Object[] dataAboutPersons
        = Stream .of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .collect(Collectors.teeing( // Teeing 1
                        Collectors.counting(), // Count
                        Collectors.teeing( // Teeing 2
                                Collectors.minBy(Comparator.comparingInt(Person::getAge)), // Min
                                Collectors.maxBy(Comparator.comparingInt(Person::getAge)), // Max
                                (youngest, oldest) -> new Person[] {youngest.get(), oldest.get()}), // Merge min and max to {min, max}
                        (count, extremes) -> new Object[] {count, extremes[0], extremes[1]})); // Merge count and {min, max}
```

new Integer(3) $_0$	new Person("Lisa", 17) $_1$	new Person("Ashley", 67) $_2$

dataAboutPersons

  - Here, we use two teeing *Collector*s and merge their results in two steps into an *Object[]* because we have heterogenous results.
  - We need a heterogeneous array because we have to collect the integral count and two "extremely aged" *Person*s.

- As can be seen, we can use the teeing *Collector* to combine as many other *Collector*s we want.
  - The downside lies more or less only in the type of the target object, into which the result of each *Collector* must be put in.
  - It would be more beneficial to have a record or tuple type in Java, which could be used as type of the target object.
  - Currently (Java 12) we have to resort to explicitly defined classes or *Object[]*.

Statistical Collectors

- For <u>numeric *Streams*</u>, we can also use special <u>statistical *Collectors*</u> to collect <u>multiple data at once</u>.
  - Those can be created by *Collectors.summarizingInt()*, *Collectors.summarizingDouble()* and *Collectors.summarizingLong()*:

Collectors
<u>+ <T> summarizingInt(mapper : ToIntFunction<? super T>) : Collector<T, ?, IntSummaryStatistics></u> <u>+ <T> summarizingDouble(mapper : ToDoubleFunction<? super T>) : Collector<T, ?, DoubleSummaryStatistics></u> <u>+ <T> summarizingLong(mapper : ToLongFunction<? super T>) : Collector<T, ?, LongSummaryStatistics></u>

  - E.g. *Collectors.summarizingInt()* can be used to collect information about our *Person*s' ages:

```java
IntSummaryStatistics ageStatistics
        = Stream .of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                .collect(Collectors.summarizingInt(Person::getAge));

System.out.println(ageStatistics);
// >IntSummaryStatistics{count=3, sum=107, min=17, average=35,666667, max=67}
```

- The type <u>*IntSummaryStatistics*</u> presents the collected information:

IntSummaryStatistics
+ getCount() : int + getMax() : int + getMin() : int + getSum() : long + getAverage() : double

Excursus: creating own Collectors – Part I

- To understand how *Collector*s function, we will implement our own.
  - But before doing this, lets inspect *Stream.collect()*'s overloads once more:

«interface» **Stream** T
<R, A> collect(collector : Collector<T, A, R>) : R <R> collect(supplier : Supplier<R>, accumulator : BiConsumer<R, T>, combiner : BiConsumer<R, R>) : R

- First lets concentrate on the overload accepting a ***supplier***, ***accumulator*** and ***combiner*** and leaving *Collector* on its own.
  - The *supplier*'s job is to supply (maybe create) an object to hold the result of collect. This object is called target container.
    - The *supplier* supplies a target container.
  - The *accumulator*'s job is to add an element to the target container applying its specific "collect-rule".
  - The *combiner*'s job is to combine two target containers.

Excursus: creating own Collectors – Part II

«interface»
Stream

```
<R> collect(supplier : Supplier<R>, accumulator : BiConsumer<R, T>, combiner : BiConsumer<R, R>) : R
```

- *Stream.collect()* applies *supplier*, *accumulator* and *combiner* depending on the sequential/parallel execution mode.

- If the source *Stream* is sequential:
  - *supplier* is used to create a target container.
  - Then each element is collected into the target container by *accumulator*.
  - The *combiner* is not required.

- If the source *Stream* is parallel:
  - In the fork-phase, the *Stream*'s workload is split into segments that are associated to tasks.
  - The tasks are executed in parallel in the execution-phase.
  - In the execution-phase each task creates its on target container via the *supplier*.
    - => Therefor, the *supplier* is not just a value, but a function!
  - Each task uses the accumulator to collect the elements into its target container.
  - In the join-phase the target containers of all tasks are combined with the *combiner* to get the final result.

Excursus: creating own Collectors – Part III

- Now we can implement our <u>own variant</u> of a collection of <u>statistical data</u>.
  - First, we create a <u>new target container type</u>, just containing some public double fields to collect data:

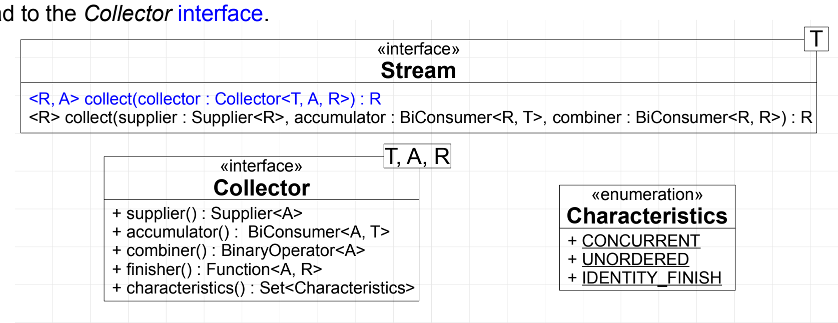
```
public class StatisticsContainer {
	public double sum;
	public double count;
	public double min;
	public double max;
}
```

  - The call of *Stream.collect()* must then be done with a suitable *supplier*, *accumulator* and *combiner*:

```
StatisticsContainer ageStatistics
	= Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
		.collect(
			StatisticsContainer::new, //////////////////////////////////////////// supplier
			(StatisticsContainer sc, Person person) -> { ////////////// accumulator
				++sc.count;
				sc.sum += person.getAge();
				sc.min = sc.min == 0
					? person.getAge()
					: Math.min(sc.min, person.getAge());
				sc.max = Math.max(sc.max, person.getAge());
			},
			(StatisticsContainer sc, StatisticsContainer sc2) -> { // combiner
				sc.count += sc2.count;
				sc.sum += sc2.sum;
				sc.min = Math.min(sc.min, sc2.min);
				sc.max = Math.max(sc.max, sc2.max);
			}
		);
```

Excursus: creating own Collectors – Part IV

- Now we'll head to the *Collector* interface.



- We can relate the three methods *supplier()*, *accumulator()* and *combiner()* to the other *Stream.collect()* overload.

- *finisher()*'s function is executed, <u>when the collection was done</u>, it is applied <u>against the resulting target container</u>.

- *characteristics()* provides a *Set<Characteristics>*, which <u>controls execution</u> of the *Collector* in detail.
  - *Characteristics* is a nested enum in *Collector*.
  - Example: when a *Collector* specifies *Characteristics.IDENTITY_FINISH*, it means the *finisher()* can be skipped.

Excursus: creating own Collectors – Part V

- In some sense, objects implementing the interface *Collector* are more or less "containers for some functions".

- There are three ways we could implement new *Collector*s:
  - We could use the simple factories in the companion class *Collector*s.
  - We could just "traditionally" implement the interface *Collector* in a new class.
  - We could use an overload of *Collector.of()* to directly create a *Collector* from "some functions".

T, A, R

«interface»
Collector

+ <T, A, R> of(supplier : Supplier<R>, accu : BiConsumer<R, T>, combi : BinaryOperator<R>, fin : Function<A, R>, chis : Characteristics...) : Collector<T, R, A>
+ <T, R> of(supplier : Supplier<R>, accu : BiConsumer<R, T>, combi : BinaryOperator<R>, chis : Characteristics...) : Collector<T, R, R>

- Let's transform our former *collect()*-call into a *Collector*-object and use *Collector.of()* to create the *Collector*...

Excursus: creating own Collectors – Part VI

```
StatisticsContainer ageStatistics
= Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
        .collect(
                StatisticsContainer::new, ///////////////////////////////////////// supplier
                (StatisticsContainer sc, Person person) -> { ///////////// accumulator
                        ++sc.count;
                        sc.sum += person.getAge();
                        sc.min = sc.min == 0
                                ? person.getAge()
                                : Math.min(sc.min, person.getAge());
                        sc.max = Math.max(sc.max, person.getAge());
                },
                (StatisticsContainer sc, StatisticsContainer sc2) -> { // combiner
                        sc.count += sc2.count;
                        sc.sum += sc2.sum;
                        sc.min = Math.min(sc.min, sc2.min);
                        sc.max = Math.max(sc.max, sc2.max);
                }
);
```

```
// Create Collector:
Collector<Person, StatisticsContainer, StatisticsContainer> statsCollector
= Collector.of(
        StatisticsContainer::new, ///////////////////////////////////////// supplier
        (StatisticsContainer sc, Person person) -> { ///////////// accumulator
                ++sc.count;
                sc.sum += person.getAge();
                sc.min = sc.min == 0
                        ? person.getAge()
                        : Math.min(sc.min, person.getAge());
                sc.max = Math.max(sc.max, person.getAge());
        },
        (StatisticsContainer sc, StatisticsContainer sc2) -> { // combiner
                sc.count += sc2.count;
                sc.sum += sc2.sum;
                sc.min = Math.min(sc.min, sc2.min);
                sc.max = Math.max(sc.max, sc2.max);
                return sc;
        }
);
// Apply Collector:
StatisticsContainer ageStatistics
= Stream.of(new Person("Ashley", 67), new Person("Lisa", 17)
                , new Person("Ashley", 23))
        .collect(statsCollector);
```

- There is only one notable difference: the *Collector*'s *combiner* is a *BinaryOperator*, i.e. it must return something.
  - In this case, we just return the lhs *StatisticsContainer*.

… but Java's Stream.collect() Operator is no Fp-Style!

- *Stream.collect()* is probably the mightiest *Stream* operation.
  - We already have a lot of predefined *Collector*s in the JDK.
  - The valuable type *Optional* and primitive specialization help us writing mighty, yet expressive code.
  - The ability to write own *Collector*s allows to collect basically anything from a *Stream*.

- Nevertheless, collect is not a real fp-style operation! – What?

- Well, the problem is visible in the *accumulator* and *combiner*:

```
BiConsumer<StatisticsContainer, Person> accumulator
= (StatisticsContainer sc, Person person) -> {
        ++sc.count; // modifies sc
        sc.sum += person.getAge(); // modifies sc
        sc.min
              = sc.min == 0 ? person.getAge() : Math.min(sc.min, person.getAge()); // modifies sc
        sc.max = Math.max(sc.max, person.getAge()); // modifies sc
};
```

```
BinaryOperator<StatisticsContainer> combiner
= (StatisticsContainer sc, StatisticsContainer sc2) -> {
        sc.count += sc2.count; // modifies sc
        sc.sum += sc2.sum; // modifies sc
        sc.min = Math.min(sc.min, sc2.min); // modifies sc
        sc.max = Math.max(sc.max, sc2.max); // modifies sc
        return sc;
};
```

  - A remarkable fact is that the target container is modified, we have side effects on the parameters!
  - This form of side effect is appreciated by the *Stream* API, *Stream.collect()* is explicitly designated as mutable reduction operation.

- The *Stream* API also provides a "really fp-style" immutable reduction operation. We'll now discuss *Stream.reduce()*.

Stream Reduction

- *Stream.reduce()* is represented with three overloads:

T

«interface»
Stream

+ reduce(accumulator : BinaryOperator<T>) : Optional<T>
+ reduce(identity : T, accumulator : BinaryOperator<T>) : T
+ <U> reduce(identity : U, accumulator : BiFunction<U, ? super T, U>, combiner : BinaryOperator<U>) : U

- We can easily program our former *ageStatistics* using *Stream.reduce()* instead of *Stream.collect()*:

```
StatisticsContainer ageStatistics = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
            .reduce(new StatisticsContainer(),
                  (StatisticsContainer sc, Person person) -> {
                     StatisticsContainer newSc = new StatisticsContainer();
                     newSc.count = sc.count + 1;
                     newSc.sum = sc.sum + person.getAge();
                     newSc.min = sc.min == 0 ? person.getAge() : Math.min(sc.min, person.getAge());
                     newSc.max = Math.max(sc.max, person.getAge());
                     return newSc;
                  },
                  (StatisticsContainer sc, StatisticsContainer sc2) -> {
                     StatisticsContainer newSc = new StatisticsContainer();
                     newSc.count = sc.count + 1;
                     newSc.sum = sc.sum + sc2.sum;
                     newSc.min = Math.min(sc.min, sc2.min);
                     newSc.max = Math.max(sc.max, sc2.max);
                     return newSc;
                  }
);
```

Stream.collect() vs Stream.reduce()

- *Stream.reduce()'s* overload we are using has an *accumulator* and *combiner* like *Stream.collect()*.
  - But the *accumulator* and *combiner* must be implemented differently respectively.

- <u>*Stream.collect()*</u>: *combiner* and *accumulator* operate on a <u>mutable target container</u> and <u>modify parameters</u>:

```
BiConsumer<StatisticsContainer, Person> accumulator
= (StatisticsContainer sc, Person person) -> {
        ++sc.count;
        sc.sum += person.getAge();
        sc.min = sc.min == 0 ? person.getAge() : Math.min(sc.min, person.getAge());
        sc.max = Math.max(sc.max, person.getAge());
};
```

```
BinaryConsumer<StatisticsContainer, StatisticsContainer> combiner
= (StatisticsContainer sc, StatisticsContainer sc2) -> {
        sc.count += sc2.count;
        sc.sum += sc2.sum;
        sc.min = Math.min(sc.min, sc2.min);
        sc.max = Math.max(sc.max, sc2.max);
};
```

```
StatisticsContainer ageStatistics = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                                          .collect(StatisticsContainer::new, accumulator, combiner);
```

- <u>*Stream.reduce()*</u>: *combiner* and *accumulator* operate on a <u>immutable target container</u>, but <u>return new target containers</u>:

```
BiFunction<StatisticsContainer, Person, StatisticsContainer> accumulator
= (StatisticsContainer sc, Person person) -> {
        StatisticsContainer newSc = new StatisticsContainer();
        newSc.count = sc.count + 1;
        newSc.sum = sc.sum + person.getAge();
        newSc.min = sc.min == 0 ? person.getAge() : Math.min(sc.min, person.getAge());
        newSc.max = Math.max(sc.max, person.getAge());
        return newSc;
};
```

```
BinaryOperator<StatisticsContainer> combiner
= (StatisticsContainer sc, StatisticsContainer sc2) -> {
        StatisticsContainer newSc = new StatisticsContainer();
        newSc.count = sc.count + 1;
        newSc.sum = sc.sum + sc2.sum;
        newSc.min = Math.min(sc.min, sc2.min);
        newSc.max = Math.max(sc.max, sc2.max);
        return newSc;
};
```

```
StatisticsContainer ageStatistics = Stream.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23))
                                          .reduce(new StatisticsContainer(), accumulator, combiner);
```

Why do we have Stream.collect() and Stream.reduce()?

- Both operations are there for reduction, *Stream.collect()* for mutable and *Stream.reduce()* for immutable reduction.

- In an ideal "functional programming world" there would be no mutable state! – Then *Stream.reduce()* would be enough!
  - *Stream.reduce()* should be used with immutable target containers and maybe also *String*s (which are immutable in Java).

- But Java is no fp language! It is an oo language, which intrinsically applies state modifications in objects.
  - Therefor, we also have *Stream.collect()*, which is based on mutable state, it is esp. allowed to mutate target containers.

- If the *Stream* API only provided immutable reduction (*Stream.reduce()*), reduction would be more costly than required.
  - *Stream.reduce()* forces accumulators and combiners to create a considerable amount of temporary target containers.
    - It not only means that many objects are created, it also means, that there is a lot of garbage. (Fp languages are optimized for the many temporaries.)
  - One could argue, that immutable algorithms may cost more memory, but they are thread safe.
    - We have to provide thread safe accumulators and combiners for *Stream.collect()*, then the mutable reduction is thread safe.
  - => The same reduction expressed with *Stream.reduce()* is often significantly slower as if expressed with *Stream.collect()*.

- => Take away: In Java we should rather use mutable reduction via *Stream.collect()*.

Not discussed Stream-related Topics

- More details on parallel *Streams*.
- *Stream* builders

Thank you!