Nieuwigheden Java 8

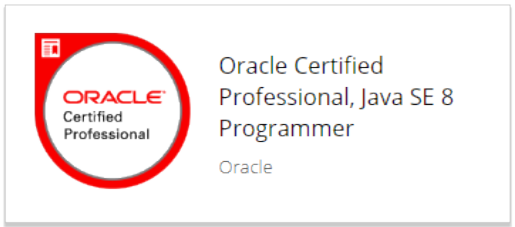
Maarten Dhondt

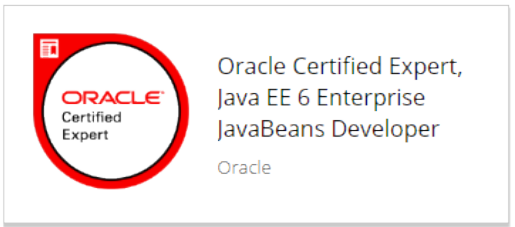
Realdolmen

October 25, 2018

Wie ben ik?

- Master in de ingenieurswetenschappen: computerwetenschappen (KUL)
  - Computationele informatica
- Master in Management (KUL)
- Software engineer @ Realdolmen sinds 2015
- Projecten:
  - Infrastructuur planning @ Infrabel
  - API platform @ Proximus
- Contact:
  - maarten.dhondt@realdolmen.com
  - maartendhondt
  - MDhondt





Outline

Outline

Outline

Interfaces

- Een interface gelijkt op een klasse, maar bevat enkel methoden en attributen. Interfaces hebben geen geïmplementeerde methoden, maar enkel de signatuur.
  - Implementaties hebben dezelfde signatuur maar return type kan een subklasse zijn.
  - Implementaties gooien geen andere checked exceptions dan diegene uit de interface.
  - Abstracte klassen kunnen methoden implementeren, maar niet vereist.

Interfaces

Implementatie zelfde signatuur maar return type kan subklasse zijn.

```
public abstract class Transaction {}
```

```
public class BankTransaction extends Transaction {}
```

```
public interface Transactionable {
    Transaction doTransaction();
}
```

```
public class BankTranserService implements Transactionable {
    @Override
    public BankTransaction doTransaction() {
        ...
    }
}
```

Interfaces

Implementatie gooit geen andere checked exceptions.

```
public interface ExceptionThrowingInterface {
    void doStuff() throws IOException;
}
```

```
public class ExceptionThrower implements ExceptionThrowingInterface {
    @Override
    public void doStuff() throws IOException, ReflectionException {
        // 'doStuff()' in 'ExceptionThrower' clashes with 'doStuff()' in
             'ExceptionThrowingInterface'; overriden method does not
            throw 'javax.management.ReflectionException'
    }
}
```

Interfaces

Abstracte klasse kan methode implementeren, maar niet vereist

```
public interface Moveable {
    void move();
}
```

```
public abstract class Furniture implements Moveable {}
```

```
public class Chair extends Furniture {
    @Override
    public void move() {
        System.out.println("Moved chair");
    }
}
```

Interfaces

- Java 8 introduceert `default` methoden.
  - Wat? Een implementatie in de interface.
  - Waarom? Optionele methoden
  - Waarom? Gedrag overerving van meerder klassen.
- Vorige regels blijven (uiteraard) geldig.

Interfaces

- Voorbeeld van een `default` methode.

```
public interface Animal {

    void eat();

    void move();

    void sleep();

    default void blinkEyes() {
        System.out.println("Blink");
    }
}
```

Interfaces

- `default` methoden: optionele methoden.

```
public interface Collection<E> extends Iterable<E> {

    default boolean removeIf(Predicate<? super E> filter) {
        Objects.requireNonNull(filter);
        boolean removed = false;
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
                removed = true;
            }
        }
        return removed;
    }

}
```

Interfaces

Java 8 introduceert ook SAM (Single Abstract Method) interfaces die we functionele interfaces noemen.

- Interface moet exact 1 abstracte methode hebben.
- Met of zonder `@FunctionalInterface` annotatie.

Kunnen gebruikt worden in lambda expressies en method references

Interfaces

```
package java.lang;

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

```
public class Main {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.submit(() -> {
            System.out.println(Thread.currentThread().getName());
        });
    }
}
```

Outline

Lambda expressies

Lambda expressies zijn een nieuwe en belangrijke functie uit Java 8 die:

- op een duidelijke en bondige manier een interface methode beschrijven in een expressie,
- een grote verbetering mogelijk maken van de Collection libraries.

Lambda expressies

- Lambda expressies bieden een oplossing aan de verbose anonieme inner klassen door 5 lijnen code te reduceren naar 1 lijn.
- Deze horizontale oplossing, lost het verticale probleem van inner klassen op.
- Een lambda expressie bestaat uit 3 delen:
  - Argumenten lijst
  - Pijltje: ->
  - Body

```
(int x, int y) -> x + y
```

Lambda expressies: syntax

Argumenten lijst:

- Kan 0, 1 of meer argumenten zijn,
- Types kunnen expliciet gedeclareerd of afgeleid worden,
- Argumenten worden tussen haakjes gescheiden door komma's,
- Lege haakjes worden gebruikt voor een lege argumenten lijst,
- Bij 1 argument met een afgeleid type, zijn haakjes niet nodig.

Lambda expressies: syntax

Body:

- Kan 0, 1 of meer statements bevatten,
- Bij 1 statement kunnen accolades en `return` keywoord weggelaten worden,
- Bij meerdere statements moeten die tussen accolades geplaatst worden en is een `return` statement verplicht wanneer er iets moet gereturned worden.

Lambda expressies: syntax

Voorbeelden:

```
n -> n % 2 != 0;

(char c) -> c == 'y';

(x, y) -> x + y;

(int a, int b) -> a * a + b * b;

() -> 42

() -> { return 3.14 };

(String s) -> { System.out.println(s); };

() -> { System.out.println("Hello World!"); };
```

Lambda expressies

```
@FunctionalInterface
public interface LambdaInterface {
  String doStuff(Integer x, String y);
}
```

```
public class Main {
  public static void main(String[] args) {
    LambdaInterface anonymousImpl = new LambdaInterface() {
      @Override
      public String doStuff(Integer x, String y) {
        return "x=" + x + ",y=" + y;
    }};

    LambdaInterface lambdaImpl = (x, y) -> "x=" + x + ",y=" + y;

    System.out.println(anonymousImpl.doStuff(5, "Abc"));
    System.out.println(lambdaImpl.doStuff(5, "Abc"));
}}
```

Method references

Method references zijn gelinkt aan lambda expressies:

- Wordt gebruikt om bestaande methode definities te hergebruiken
- Wordt doorgegeven op dezelfde manier als lambda expressies

```
Object::methodName
```

- Het object of de klasse die een methode bevat wordt voor de dubbele dubbelpunt geplaatst.
- De naam van de methode komt erna. Zonder argumenten.

Method references

- Referentie naar een static method

  λ `(args) -> ClassName.staticMethodName(args)`

  MR `ClassName::staticMethodName`

- Referentie naar instantie methode van een bestaand object

  λ `(args) -> object.instanceMethodName(args)`

  MR `object::instanceMethodName`

- Referentie naar instantie methode van een arbitrair object

  λ `(arg0,rest) -> arg0.instanceMethodName(rest)`

  MR `ClassName::instanceMethodName`

- Referentie naar een constructor

  λ `(args) -> new ClassName(args)`

  MR `ClassName::new`

Lambda expressies

- Lambda expressies en vooral method references hebben wat oefening nodig alvorens er vlug meer overweg te kunnen.
- Java 8 introduceert een nieuwe `package` `java.util.function` met 43 functionele interfaces waarvan een aantal een belangrijk deel uitmaken van uw dagdagelijkse code.

java.util.function

4 categorieën:

- Supplier
- Consumer
  - BiConsumer
- Predicate
  - BiPredicate
- Function
  - BiFunction
  - UnaryOperator
  - BinaryOperator

Suppliers

- Een `Supplier` neemt geen argumenten
- Een `Supplier` levert een waarde (via zijn `get()` methode)

```
static void display(Supplier<Integer> suppl) {
    System.out.println(suppl.get());
}
```

```
display(() -> 10);
display(() -> 100);
```

Consumers

- Een `Consumer` neemt argumenten
- Een `Consumer` returned `void` (via zijn `accept()` methode)

```
static void display(int value) {
    switch (value) {
        case 1:  System.out.println("Value is 1");
                 return;
        default: System.out.println("Value != 1");
                 return;
    }
}
```

```
Consumer<Integer> consumer = x -> display(x - 1);
consumer.accept(2);
```

Predicates

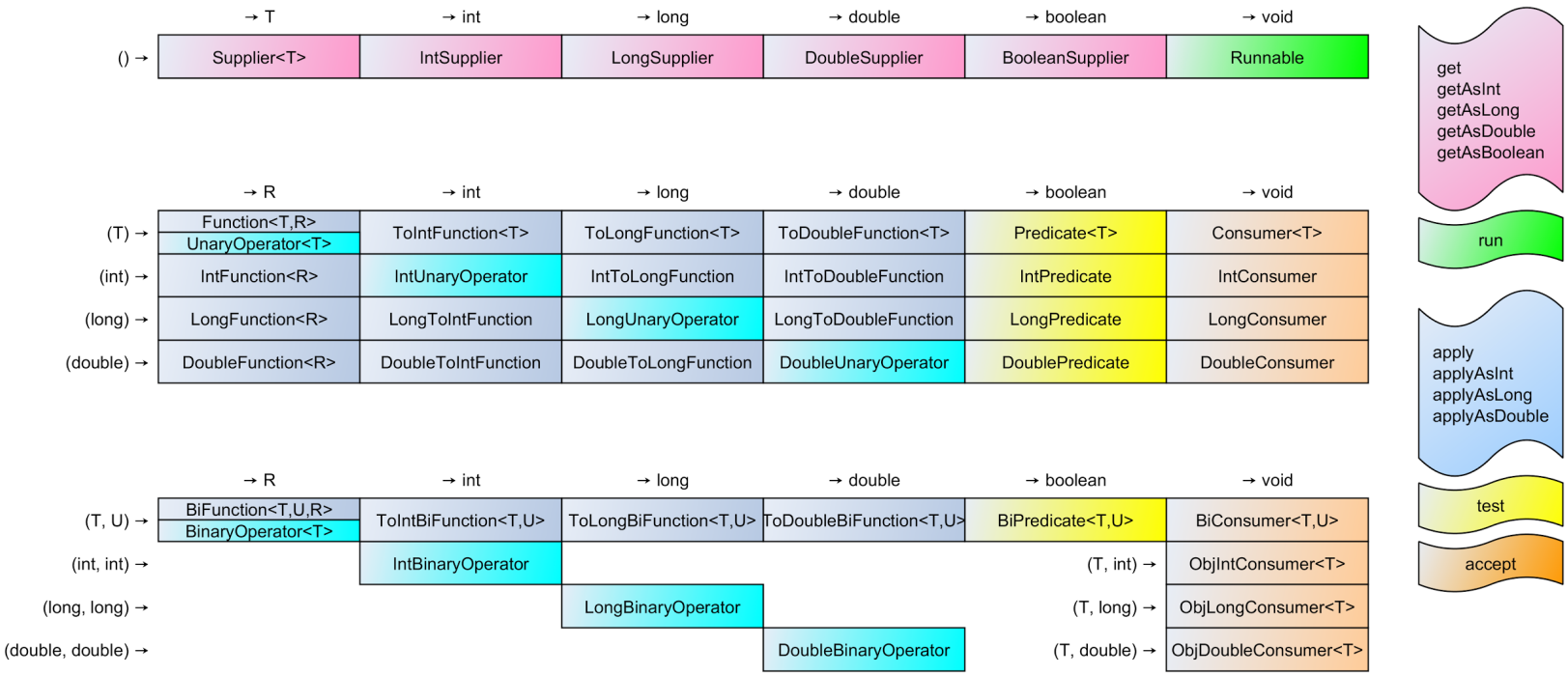
- Een `Predicate` neemt argumenten en returned een boolean

```
public interface Collection<E> extends Iterable<E> {
    default boolean removeIf(Predicate<? super E> filter) {
        Objects.requireNonNull(filter);
        boolean removed = false;
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
                removed = true;
        }}
        return removed;
}}
```

```
List<String> animals = Arrays.asList("cat", "dog", "cheetah", "deer");

animals.removeIf(element -> element.startsWith("c"));
```

Java Functional Interface Naming Guide by Esko Luontola



Java 8 Functional Interface Naming Guide
by Esko Luontola, www.orfjackal.net

Outline

Streams

Wat is een `Stream`? — een object

- waarop operaties mogelijk zijn,
- dat geen data bijhoudt (in tegenstelling to `Collection`)
- dat de data waarop het werkt niet veranderd
- dat enorme hoeveelheden data kan verwerken in 1 statement
- dat algoritmisch geöptimaliseerd is om data in parallel te verwerken.

Streams

Waarin verschillen Streams van Collections?

- Collections houden zich bezig met het efficiënte management en toegang tot data.
- Streams bieden geen rechtstreekse functionaliteit om data aan te passen, maar zijn gemaakt om op een declaratieve manier operaties op de data te omschrijven.
- Streams zijn:
  - geen opslag of datastructuur
  - functioneel by design (een resultaat produceren zonder de bron data aan te passen)
  - Lazy
  - Optioneel gebonden

Streams

Hoe bekom ik zo'n Stream?

- `java.util.Collection` heeft 2 nieuwe default methoden
  - `default Stream<E> stream()`
  - `default Stream<E> parallelStream()`
- `java.util.stream.Stream` heeft enkele static methoden
  - `public static<T> Stream<T> of(T... values)`
  - `public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)`
  - `public static<T> Stream<T> generate(Supplier<T> s)`

Stream non-terminal operaties

- `Stream<T> filter(Predicate<T> predicate);`
- `Stream<R> map(Function<T, R> mapper);`
- `Stream<R> flatMap(Function<T, Stream<R» mapper);`
- `Stream<T> distinct();`
- `Stream<T> peek(Consumer<T> action);`
- `Stream<T> limit(long maxSize);`
- `Stream<T> skip(long n);`

Stream terminal operaties

- `void forEach(Consumer<T> action);`
- `T reduce(T iden, BinaryOperator<T> accumulator);`
- `R collect(Collector<T, A, R> collector);`
- `Optional<T> min(Comparator<T> comparator);`
- `Optional<T> max(Comparator<T> comparator);`
- `long count();`
- `boolean anyMatch(Predicate<T> predicate);`
- `boolean allMatch(Predicate<T> predicate);`
- `boolean noneMatch(Predicate<T> predicate);`
- `Optional<T> findFirst();`
- `Optional<T> findAny();`

Stream voorbeelden

Als in de volgende voorbeelden de klasse `Person` gebruikt wordt, dan hebben we het over:

```
public class Person {

  private String name;
  private Integer age;
  private String country;
  private Map<Person, String> relationshipsByPerson;

  public String getName() { return name; }
  public Integer getAge() { return age; }
  public String getCountry() { return country; }
  public Map<Person, String> getConnections() {
    return relationshipsByPerson;
  }
}
```

Stream voorbeelden

Hebben we gegevens van volwassen mensen uit België?

```
public static boolean hasBelgianAdults(Collection<Person> people) {
  return people.stream()
               .filter(person -> person.getAge() >= 18)
               .filter(person -> person.getCountry().equals("Belgium"))
               .collect(Collectors.toList())
               .size() > 0;
}
```

```
return people.stream()
             .filter(person -> person.getAge() >= 18)
             .filter(person -> person.getCountry().equals("Belgium"))
             .count() > 0;
```

Stream voorbeelden

Hebben we gegevens van volwassen mensen uit België?

```
return people.stream()
             .filter(person -> person.getAge() >= 18)
             .anyMatch(person -> person.getCountry().equals("Belgium"));
```

```
return people.stream()
             .filter(Objects::nonNull)
             .filter(person -> person.getAge() >= 18)
             .map(Person::getCountry)
             .anyMatch("Belgium"::equalsIgnoreCase);
```

Stream voorbeelden

Vind alle personen die een dochter zijn:

```
public Collection<Person> getDaughters(Collection<Person> people) {
    return people.stream()
                 .filter(Objects::nonNull)
                 .map(Person::getConnections)
                 .map(Map::entrySet)
                 .flatMap(Set::stream)
                 .filter(entry -> entry.getValue().equals("daughter"))
                 .map(Map.Entry::getKey)
                 .distinct()
                 .sorted(Comparator.comparing(Person::getName))
                 .collect(Collectors.toList());
}
```

Stream voorbeelden

Een lijst van Strings die gesplitst moeten worden als ze een komma bevatten en het resultaat moet als Map gereturned worden:

```
public class Main {
    public static void main(String[] args) {
        System.out.println(splitToMap(Arrays.asList("11,21", "12,21",
            "13,23", "13,24", null, "14", "abc", "1,2,3", "12, 13")));
    }
}
```

```
{11=21, 12=21, 13=23}
```

Stream voorbeelden

```
public static Map<Long, Long> splitToMap(List<String> stringsList) {
  return stringsList.stream()




}
```

Stream voorbeelden

```
public static Map<Long, Long> splitToMap(List<String> stringsList) {
  return stringsList.stream()
                    .filter(Objects::nonNull)



}
```

Stream voorbeelden

```
public static Map<Long, Long> splitToMap(List<String> stringsList) {
  return stringsList.stream()
                    .filter(Objects::nonNull)
                    .map(line -> line.split(","))



}
```

Stream voorbeelden

```
public static Map<Long, Long> splitToMap(List<String> stringsList) {
  return stringsList.stream()
                    .filter(Objects::nonNull)
                    .map(line -> line.split(","))
                    .filter(array -> array.length == 2)


}
```

Stream voorbeelden

```
public static Map<Long, Long> splitToMap(List<String> stringsList) {
  return stringsList.stream()
                    .filter(Objects::nonNull)
                    .map(line -> line.split(","))
                    .filter(array -> array.length == 2)
                    .filter(array -> {
                      try {
                        Long.parseLong(array[0]);
                        Long.parseLong(array[1]);
                        return true;
                      } catch (NumberFormatException nfe) {
                        return false;
                      }
                    })

}
```

Stream voorbeelden

```
public static Map<Long, Long> splitToMap(List<String> stringsList) {
  return stringsList.stream()
                    .filter(Objects::nonNull)
                    .map(line -> line.split(","))
                    .filter(array -> array.length == 2)
                    .filter(array -> {
                      try {
                        Long.parseLong(array[0]);
                        Long.parseLong(array[1]);
                        return true;
                      } catch (NumberFormatException nfe) {
                        return false;
                      }
                    })
                    .collect(Collectors.toMap(
                            array -> Long.valueOf(array[0]),
                            array -> Long.valueOf(array[1]),
                            (first, second) -> first));
}
```

Outline

Java Date / Time API

De datum en tijd klassen voor Java 8 waren behoorlijk verwarrend:

- Er is `java.util.Date` die zowel datum als tijd bijhoudt
- Er is `java.sql.Date` die overerft van `java.util.Date` maar enkel de datum representeert

De Date klassen zijn mutable:

- Laat toe dat deze via reference worden aangepast

Java Date / Time API

```
public class SensitiveData {
    private Date creationDate;
    public Date getCreationDate() {
        return creationDate;
    }
}
```

```
sensitiveData.getCreationDate().setTime(new Date().getTime());
```

```
public class SensitiveData {
    private Date creationDate;
    public Date getCreationDate() {
        return new Date(creationDate);
    }
}
```

Java Date / Time API

Java 8 introduceert nieuwe klassen die wel immutable zijn:

- Instant
  - Instant 0 is 1 januari 1970 om middernacht GMT
  - precisie is in nanoseconden
- Duration
  - Tijdspanne tussen twee Instants
  - Handig voor uren, minuten en seconden

```
Instant now = Instant.now();
Instant newYear = Instant.parse("2019-01-01T00:00:00Z");
Duration between = Duration.between(now, newYear);
System.out.println(between.getSeconds() + " seconds to go!");
```

Java Date / Time API

Java 8 introduceert nieuwe klassen die wel immutable zijn:

- Period
  - Ook een tijdspanne, maar tussen twee datums in plaats van Instants
  - Handig voor dagen, weken, maanden en jaren

```
LocalDate mozartBirthDate = LocalDate.of(1756, Month.JANUARY, 27);
LocalDate today = LocalDate.now();
Period yearsAgo = Period.between(mozartBirthDate, today);
System.out.println("Mozart was born " + yearsAgo.get(ChronoUnit.YEARS) +
      " years ago");
```

- ChronoUnit
- LocalDate
- LocalTime
- LocalDateTime

Outline

Optional

- Nieuwe wrapper klasse die ofwel een waarde bevat of leeg is.
- Beschermt tegen `NullPointerExceptions`:

```
getEventWithId(10).getLocation().getCity();
```

```
public String getCityForEvent(int id) {
  Event event = getEventWithId(id);
  if(event != null) {
    Location location = event.getLocation();
    if(location != null) {
      return location.getCity();
    }
  }
  return "NotFound";
}
```

Optional

```
public String getCityForEvent(int id) {
    Optional.ofNullable(getEventWithId(id))
            .flatMap(Event::getLocation)
            .map(Location::getCity)
            .orElse("NotFound");
}
```

StringJoiner

```
StringJoiner sj = new StringJoiner(", ");
sj.add("one", "two", "three");
String string = sj.toString();
System.out.println(string);
```

```
asList("one", "two", "three").stream().collect(joining(", "));
```

```
String.join(", ", Arrays.asList("one", "two", "three"));
```

Comparators

`java.util.Comparator` heeft enkele nieuwe methoden gekregen:

- `static Comparator<T> comparing(Function<T,U>)`
- `default Comparator<T> thenComparing(Comparator<>)`
- `static Comparator<T> nullsFirst(Comparator<T>)`

```
Comparator<Person> compareBasedOnFullName =
                   Comparator.comparing(Person::getLastName)
                             .thenComparing(Person::getFirstName);
```

JavaFX 8

- JavaFX heeft een enorme update gekregen
- De vorige versie was 2.2, versionering volgt nu Java SE
- Is nu onderdeel van Java SE in plaats van apparte dependency

Outline