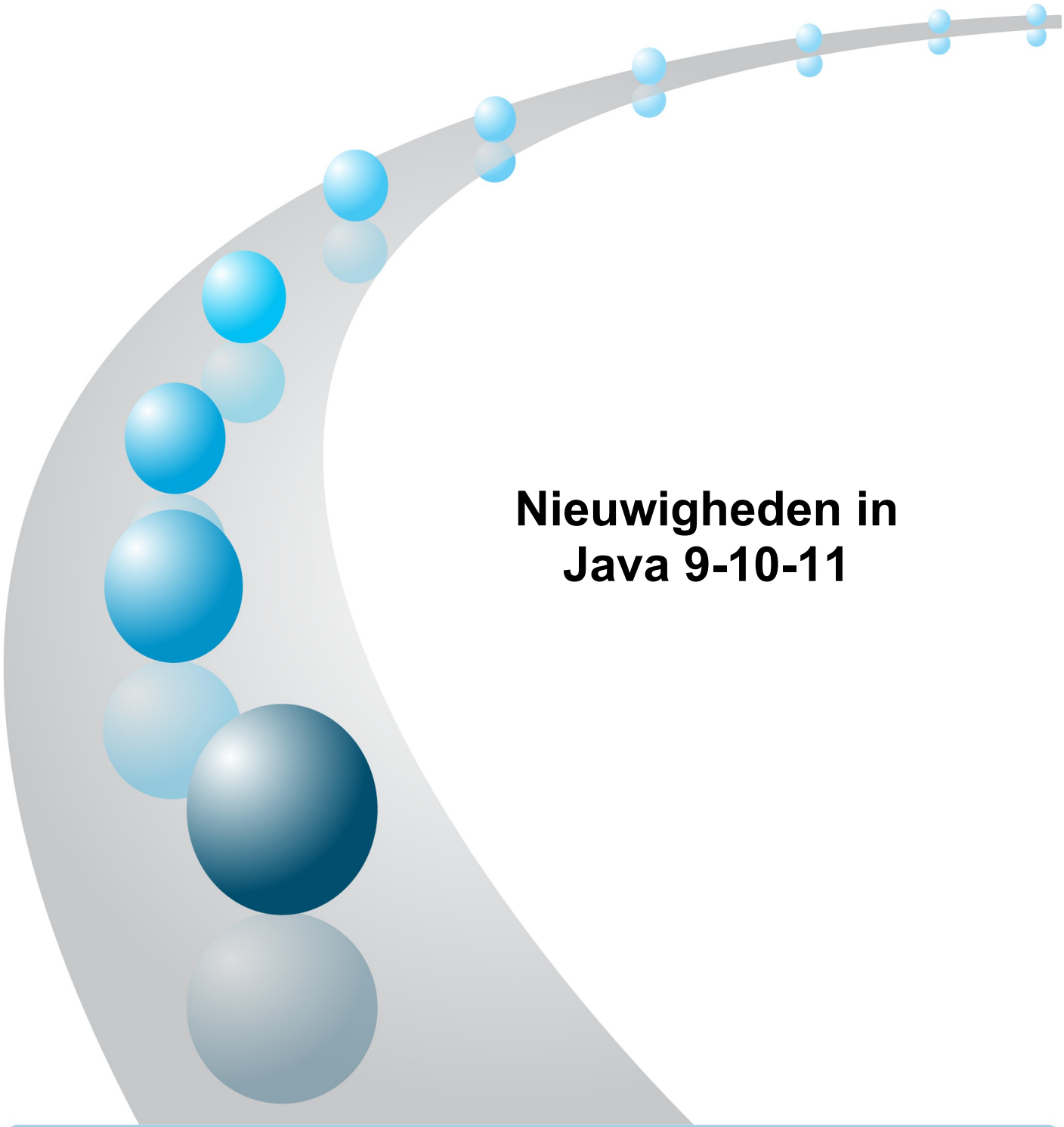
Noël Vaes
Java Trainer & Consultant



Nieuwigheden in Java 9-10-11

Roode Roosstraat 5
3500 Hasselt
België

+32 474 38 23 94
noel@noelvaes.eu
www.noelvaes.eu

Noël Vaes
Roode Roosstraat 5
3500 Hasselt
België

Tel: +32 474 38 23 94

noel@noelvaes.eu
www.noelvaes.eu

22/10/2020

Inhoudsopgave

Hoofdstuk 1: Java versies

De nieuwe *releases* van Java volgen elkaar inmiddels in sneltempo op. Dit heeft vooral te maken met een gewijzigd beleid van Oracle inzake *releases*. Er verschijnt nu tweemaal per jaar een nieuwe *release* van het platform en die wordt telkens gekenmerkt door een verhoging van het hoofdnummer: Java 9, Java 10 enzovoort. Doorgaans zijn we gewend dat een wijziging van het hoofdnummer duidt op een *major release* maar dat is nu niet meer het geval. Het zijn kleinere *releases* die nu toch een verhoging van het hoofdnummer krijgen. Voorheen werden dergelijke *releases* aangeduid door het tweede nummer: 8.1, 8.2 enzovoort.

Als ontwikkelaar krijg je daardoor al snel het gevoel helemaal achter te lopen op de laatste ontwikkelingen. Maar dat is niet noodzakelijk het geval. Belangrijk is te kijken naar de grote mijlpalen. Vroeger waren dat inderdaad de *major releases* maar momenteel moeten we eerder kijken naar de LTS-*releases*. Dit zijn de *releases* waarbij er ondersteuning aangeboden wordt voor een lange termijn: *Long Term Service Releases*.

Java 11 is zo'n LTS-*release* en de volgende zal Java 17 zijn die gepland staat voor september 2021.

Migreren naar een nieuwe versie van Java is niet altijd een sinecure en zo'n stap wordt in een productie-omgeving doorgaans weloverwogen en goed voorbereid gezet.

Op dit moment is Java 11 al een tijdje beschikbaar en het is momenteel de aangewezen versie om naar over te schakelen.

Aangezien de vorige grote *release* Java 8 was, zullen we in deze cursus de nieuwigheden van Java 9, 10 en 11 samen behandelen.

We kunnen Java 11 downloaden van de volgende locatie:
https://www.oracle.com/java/technologies/.

Wat opvalt is dat enkel de JDK beschikbaar is, en niet langer ook een afzonderlijke JRE. We zullen later in deze cursus leren hoe we onze eigen JRE kunnen bouwen.

Opdracht 1: JDK 11 installeren

- Haal de laatste versie van JDK 11 van de website:
  https://www.oracle.com/java/technologies/.
- Installeer JDK 11 en accepteert hierbij de standaardinstellingen.
- Voeg een omgevingsvariabele toe met het pad naar de JDK.

  Bijvoorbeeld: `JAVA_HOME=C:\Program Files\Java\jdk-11.0.8`

- Voeg vervolgens een pad toe aan de omgevingsvariabele `PATH` met verwijzing naar de map `bin` van de JDK-installatie. Voeg dit pad bij voorkeur aan het begin toe.

  Bijvoorbeeld: `PATH=%JAVA_HOME%\bin;...`

- Open een consolevenster en controleer de installatie:

  ```
  javac -version
  ```
- Configureer je IDE voor het gebruik van deze JDK. Dit is afhankelijk van de gebruikte IDE.

Hoofdstuk 2: Nieuwigheden in de Java-taal

2.1. De _ als naam van een variabele (Java 9)

Sinds Java 9 mag de naam van een variabele niet meer bestaan uit enkel een *underscore*.

```
int _ = 5; // Invalid
```

2.2. Try-with-resources met final variabelen (Java 9)

Bij de *try-with-resources* is het niet langer nodig de variabele te definiëren in de `try()`. We kunnen nu ook gewoon een bestaande variabele gebruiken indien die (impliciet) *final* is.

Een voorbeeld:

```
try (FileWriter file = new FileWriter("MyFile.txt")) {
   file.write("Hello World");
}
```

Dit kan nu ook als volgt geschreven worden:

```
final FileWriter file = new FileWriter("MyFile.txt");
try (file) {
   file.write("Hello World");
}
```

Indien de compiler kan vaststellen dat de variabele impliciet *final* is, kan het ook zonder de variabele expliciet als *final* te definiëren:

```
FileWriter file = new FileWriter("MyFile.txt");
try (file) {
   file.write("Hello World");
}
```

Indien we gebruikmaken van meerdere variabelen ziet het er als volgt uit:

```
FileWriter file1 = new FileWriter("File1.txt");
FileWriter file2 = new FileWriter("File2.txt");
try (file1; file2) {
   file1.write("Hello World");
   file2.write("Hello World");
}
```

2.3. Private methoden in interfaces (Java 9)

Sinds Java 9 mogen methoden van interfaces tevens *private* zijn. Dit laat toe de code te organiseren en te hergebruiken binnen de interface. *Default* methoden en andere statische methoden kunnen dan gebruikmaken van deze private methoden.

We illustreren dit met het volgende voorbeeld:

```
public interface MyInterface {
   public final int CONST = 1;

   public void method();

   public default void defaultMethod() {
       System.out.println("call of defaultMethod()");
       privateMethod();
       privateStaticMethod();
   }

   public static void staticMethod() {
      System.out.println("call of staticMethod()");
      privateStaticMethod();
   }

   private void privateMethod() {
       System.out.println("call of privateMethod()");
   }

   private static void privateStaticMethod() {
       System.out.println("call of privateStaticMethod()");
   }
}
```

2.4. *Diamond operator* in anonieme geneste klassen (Java 9)

De *diamand operator* <> kan gebruikt worden in omstandigheden waar de compiler de type-parameter kan afleiden uit de context.

Bijvoorbeeld:

```
List<String> words = new ArrayList<>();
```

Aangezien het hier duidelijk is dat de type-parameter `String` moet zijn, kunnen we deze gewoon weglaten en enkel <> gebruiken.

Dit werkt voortaan ook met anonieme geneste klassen.

In het onderstaande voorbeeld maken we een *comparator* met een anonieme geneste klasse:

```
   List<String> words = new ArrayList<>();

   // add words to list

   words.sort(new Comparator<>() {
      public int compare(String s1, String s2) {
         return s1.length() - s2.length();
      }
   });
```

We maken hier een `Comparator` voor het type `String`. Aangezien dit datatype uit de context afgeleid kan worden, kunnen we de type-parameter ook hier weglaten. De compiler vult die dan automatisch in.

2.5. Afleiding van het datatype voor lokale variabelen (Java 10)

Indien we een lokale variabele declareren en onmiddellijk een waarde toekennen is het niet altijd nodig om het datatype expliciet op te geven. Indien de compiler uit de context kan afleiden wat het datatype is, kunnen we ook gebruikmaken van het algemene type `var`.

Bijvoorbeeld:

```
var number = 5;
```

In dit geval kan de compiler uit de toekenning afleiden dat het datatype `int` is omdat de waarde 5 van het type `int` is. Daarom zal ook de variabele `number` van het type `int` zijn en volstaat het om in dit geval `var` te gebruiken.

De compiler zal `var` vervangen door `int`.

Dit afleiden van het datatype kan niet indien de variabele pas later een waarde krijgt. Dit is daarom niet toegelaten:

```
var number; // Not allowed !!
number = 5;
```

Enige voorzichtigheid is geboden indien we gebruikmaken van klassen die een interface implementeren. Bijvoorbeeld:

```
var words = new ArrayList<String>();
words.ensureCapacity(100);
```

Het datatype van `words` zal hier `ArrayList` zijn. Het is evenwel gebruikelijk om de interface als datatype te gebruiken en niet de implementerende klasse. Dit voorkomt dat we methoden gebruiken die specifiek zijn voor de implementerende klasse en niet beschikbaar zijn in de interface. In dat geval is het gebruik van `var` niet aangewezen.

Het afleiden van het datatype en het gebruik van `var` is enkel bedoeld om in complexe omstandigheden de hoeveelheid code te reduceren. Het introduceert geenszins een nieuw datatype en zeker geen dynamisch datatype zoals we dat bijvoorbeeld in *JavaScript* kennen.

Deze afleiding van het datatype kan overigens enkel gebruikt worden voor lokale variabelen.

2.6. Afleiding van het datatype voor lokale variabelen in *lambda expressions* (Java 11)

Het gebruik van `var` is sinds Java 11 ook mogelijk in een *lambda expression*.

We geven een voorbeeld van een *comparator* met een *lambda expression*:

```
var words = new ArrayList<String>();
// Add words
words.sort((String s1, String s2) -> s1.length()-s2.length());
```

Dit kunnen we nu als volgt schrijven:

```
var words = new ArrayList<String>();
// Add words
words.sort((var s1, var s2) -> s1.length()-s2.length());
```

Doorgaans laten we evenwel het datatype gewoon weg in een *lambda expression* en is het gebruik van `var` niet nodig:

```
var words = new ArrayList<String>();
// Add words
words.sort((s1, s2) -> s1.length()-s2.length());
```