JAVA 19

Jawel hoor, we zijn weer een half jaar verder en het is tijd voor een nieuwe versie van Java. In Java SE 19 staan zeven features (JEP's) gepland.

Om met wat Java SE 19 features te kunnen spelen (zonder de early access echt te hoeven installeren) is alle code in dit artikel uitgevoerd binnen een dockercontainer met OpenJDK 19 [2], zie Listing 1.

L1

```
$ docker run -it --rm \
      -v $(pwd)/src:/src \
      openjdk:19-slim /bin/bash
$ cd /src/java19
```

Dit artikel is in twee hoofdsecties verdeeld. Het eerste deel gaat over nieuwe standaardfeatures. Het tweede deel gaat in op preview en incubator features. Normaal gesproken is er nog een derde sectie, waarin we spreken over de features die uitgefaseerd (gaan) worden, maar er zijn er geen aangekondigd voor deze versie. Bij elke feature zal het JEP-nummer vermeld worden.

{ NIEUWE STANDAARD FEATURES }

In Java 19 is maar één nieuwe feature aangekondigd.

422: Linux/RISC-V Port

RISC-V (uitgesproken in het Engels als 'Risk-five') is een oorspronkelijk aan de Berkley Universiteit van California ontwikkelde RISC-instructiesetarchitectuur (ISA). De toenemende beschikbaarheid van RISC-V-hardware maakt een poort van de JDK waardevol. In Java 19 zal deze poort voltooid zijn en onderdeel worden van de JDK.

{ PREVIEW EN INCUBATOR FEATURES }

424: Foreign Function & Memory API (Preview)

Deze JEP vervangt twee eerdere incubatie-API's: de Foreign Memory Access API (JEP's 370, 383 en 393) en Foreign Linker API (JEP 389). De eerdere incubaties faalden. Het doel van deze JEP is om een gebruiksvriendelijkere en algemenere API te bieden om met code en gegevens buiten JVM te werken.

Ivo Woltring is Principal Expert en Codesmith bij Ordina JTech en houdt zich graag bezig met nieuwe ontwikkelingen in de softwarewereld.

426: Vector API (Fourth Incubator)

Deze JEP is de vierde incubatiefase van een API om vectorberekeningen te compileren tot optimale vectorinstructies op de ondersteunde CPU-architecturen. Deze fase richt zich vooral op verbeteringen uit feedback en op verbeterde implementatie en

L2

```
package java19;
public class JEP405 {
    record Point(int x, int y) {}
    static void printSumOld(Object o) {
        if (o instanceof Point p) {
            int x = p.x();
            int y = p.y();
            System.out.println(x + y);
        }
    }
    static void printSumNew(Object o) {
        if (o instanceof Point(int x,int y)) {
            System.out.println(x + y);
        }
    }
    public static void main(String[] args) {
        printSumOld(new Point(22, 20));
        printSumNew(new Point(20, 22));
    }
}
$ java --enable-preview --source 19 JEP405.java
42
42
```

performance. Deze JEP bouwt voort op JEP 417 uit Java SE 18, JEP 414 uit Java 17 en JEP 338 die in Java SE 16 is geïntroduceerd. Zie referentie [2] voor voorbeeldcode.

405: Record Patterns (Preview)
Java 16 is middels JEP 394 uitgebreid met een 'typepatronen test' (Engels: 'Type Pattern'). In Java 17 en 18 is ook het switch-case-statement hiermee uitgebreid, via respectievelijk JEP 406 en 420, zie referentie [3] voor codevoorbeelden.

Het gebruik van Type Pattern zal in de meeste gevallen de noodzaak van typecasten verwijderen. Dit is echter pas de eerste stap naar een meer declaratieve, datagerichte programmeerstijl. Aangezien Java nu met records een meer expressieve manier ondersteunt om gegevens te modelleren, kan pattern matching het gebruik van gegevens makkelijker maken door ze in staat te stellen de semantische bedoeling in hun modellen uit te drukken, zie Listing 2.

427: Pattern Matching for switch (Third Preview)
Dit is de derde preview van pattern matching voor switch-statements die voor het eerst is uitgebracht in Java 17 in JEP 406 en zijn tweede preview kreeg in Java 18 in JEP 420. In deze derde preview zijn vooral kleine verbeteringen doorgevoerd aan de hand van gebruikersfeedback en gebruikservaring. Bekijk de voorbeeldcode [2] en het Java 17 artikel [3].

425: Virtual Threads (Preview)
Virtual Threads zijn onderdeel van project Loom [4]. Project Loom is gericht op het verbeteren van de concurrency-prestaties in Java door de ontwikkelaar concurrency-toepassingen met bekende API's te laten schrijven, te onderhouden en hardwarebronnen efficiënter te laten gebruiken.

Virtuele Threads zijn nieuwe, lichtgewicht implementaties van Java's thread-klasse die worden ingepland door de JDK, in plaats van door het besturingssysteem (OS), zoals tot nu toe het geval is geweest in Java. Voorbeeldcode is hier achterwege gelaten, omdat je er in het volgende Java Magazine een heel artikel over kan lezen.

428: Structured Concurrency (Incubator)
Het idee achter Structured Concurrency is om de levensduur van een of meerdere threads hetzelfde te laten werken als codeblokken in gestructureerd programmeren. Structured Concurrency behandelt meerdere taken in verschillende threads als een enkele unit of work (eenheid van werk), waardoor foutafhandeling en afbreken worden gestroomlijnd, wat de betrouwbaarheid en ook de observeerbaarheid (debugging) verbetert.

Listing 3 geeft drie voorbeelden van een `foo`-method. Een method waar de code sequentieel wordt aangeroepen. Een waar het aangeroepen wordt op een multithreaded manier en eentje waar Structured Concurrency wordt toegepast. In de `fooSequential`-methode is het voor de gemiddelde ontwikkelaar overduidelijk wat er gebeurt als er in een van de statements een exception plaatsvindt. `fooSequential` zal falen op dat statement.

Hoe `fooThreaded` faalt als bijvoorbeeld in `baz()` een exception wordt gegooid, is een stuk moeilijker te begrijpen. De threads moeten namelijk eerst helemaal resolven, voordat de `foo`-method in de fout zal propageren. Deze zal namelijk `bar()` eerst evalueren en die zal pas na twee seconden terugkomen. Dit komt, omdat de `baz` en `bar`-calls in isolatie draaien. Het gaat nog verder. Stel dat deze `foo`-method zelf in de fout gaat voordat de joining calls worden gedaan. Dan zal `foo` al falen, maar de threads gewoon doorlopen.

In de `fooStructured`-method worden de aangemaakte threads als één unit of work gezien en zal de `foo`-method meteen terugkomen als een van de andere calls falen.

In Listing 4, waar de code gedraaid is zonder dat een exception gegooid wordt, kan je zien dat de Sequential-call meer dan 2500 ms duurt, omdat `bar` en `baz` na elkaar aangeroepen worden.

L3

```
import jdk.incubator.concurrent.*;
import java.io.IOException;
import java.util.concurrent.*;
public class JEP428 {
    String bar() throws InterruptedException {
        Thread.sleep(2000);
        return "bar";
    }
   String baz() throws IOException,
                       InterruptedException {
        Thread.sleep(500);
        return "baz";
//        throw new IOException("baz");
    }
    String fooSequential() throws
    InterruptedException, IOException {
        String bar = bar(); // kan een Exception
        opleveren
        String baz = baz(); //ditto
        return baz + bar;
    }
    String fooThreaded() throws
    ExecutionException, InterruptedException {
        var executorService =
        new ForkJoinPool(2);
        Future<String> bar =
        executorService.submit(this::bar);
```

Vervolg op de volgende pagina

L3

```
        Future<String> baz = executorService.
        submit(this::baz);
        return bar.get() + baz.get();
    }
    String fooStructured() throws
    ExecutionException, InterruptedException {
        try (var scope = new
        StructuredTaskScope.ShutdownOnFailure()) {
            Future<String> bar =
            scope.fork(this::bar);
            Future<String> baz =
            scope.fork(this::baz);
            scope.join();
            scope.throwIfFailed();
            return bar.resultNow() +
            baz.resultNow();
        }
    }
    public static void main(String[] args) {
        var self = new JEP428();
        long start = System.currentTimeMillis();
        try {
            System.out.println(
self.fooSequential());
        } catch (InterruptedException |
IOException e) {
            System.out.println("Error:
            sequential");
        }
        long end = System.currentTimeMillis();
        System.out.println("Sequential took " +
        (end - start) + " ms");
        start = System.currentTimeMillis();
        try {
            System.out.println(self.
            fooThreaded());
        } catch (ExecutionException |
                InterruptedException e) {
            System.out.println("Error:
            threaded");
        }
        end = System.currentTimeMillis();
        System.out.println("Threaded took " +
        (end - start) + " ms");
        start = System.currentTimeMillis();
        try {
            System.out.println(self.
            fooStructured());
        } catch (ExecutionException |
        InterruptedException e) {
```

L3

```
            System.out.println("Error:
            structured");
        }
        end = System.currentTimeMillis();
        System.out.println("Structured took " +
        (end - start) + " ms");
    }
}
```

Tussen Threaded en Structured zit niet veel verschil. Echter het grote verschil wordt duidelijk, zodra er wel wat fout gaat (Listing 5). Dan kan je zien dat bij het gebruik van Structured Concurrency de `foo`-methode faalt, zodra de eerste exception gegooid wordt (in de `baz`-method).

{ CONCLUSIE }

Veel incubator en preview features in deze versie van Java, maar ook wel echt potentiële features met een hoop belofte. Als de Virtuele Threads en de Structured Concurrency het halen om tot de core van Java te behoren, dan belooft dat veel goeds. Multi threaded werken alsof je gewoon gestructureerd aan het programmeren bent. Ik ben vóór! <

L4

```
$ java –source 19 –enable-preview \
–add-modules jdk.incubator.concurrent JEP428.java
bazbar
Sequential took 2504 ms
barbaz
Threaded took 2008 ms
barbaz
Structured took 2062 ms
```

L5

```
$ java –source 19 –enable-preview \
–add-modules jdk.incubator.concurrent JEP428.java
Interrupted in sequential
Sequential took 2512 ms
Interrupted in threaded
Threaded took 2019 ms
Interrupted in structured
Structured took 531 ms
```

{ REFERENTIES }

1 https://openjdk.org/projects/jdk/19/
2 http://ivo2u.nl/Vy
3 http://ivo2u.nl/oz
4 https://openjdk.org/projects/loom/