JAVA MAGAZINE .nl.jug
Onafhankelijk tijdschrift voor de Java-professional
01 > 2024

WRITING THE CLEANEST CODE

› TERUGBLIK
J-FALL 20 JAAR

› INTRODUCTION TO
KEY ENCAPSULATION MECHANISM API

› FUN WITH
FUNCTIONAL PROGRAMMING

de coöperatieve **Rabobank**

{ COLOFON } JAVA MAGAZINE 01-2024

Hoofdredacteur:
Lieke Stomps

Communitymanager:
Simon de Groot

Eindredactie:
Suzanne Muller

Auteurs:
Felienne Hermans, Brian Vermeer, Bert Breeman, Julien Lengrand-Lambert, Maja Reissner, Ivo Woltring, Hanno Embregts

Redactiecommissie:
Stijn van de Blankevoort, Hanno Embregts, Julien Lengrand-Lambert, Helma Maassen, Elias Nogueira, Maja Reißner, Ivo Woltring, Thomas Zeeman

Vormgeving:
Wonderworks, Haarlem

Traffic & Media order:
Marco Verhoog

Drukkerij:
Senefelder Misset, Doetinchem

Advertenties:
Richelle Bussenius
E-mail: richelle.bussenius@nljug.org
Telefoon: 023 752 39 22
Fax: 023 535 96 27
Marc Post
E-mail: mpost@reshift.nl
Telefoon: 023 543 00 08

Abonnementenadministratie:
Tanja Ekel

De prijs van een lidmaatschap van de NLJUG verschilt per membership tier die u afneemt. Een reguliere "base" membership van de NLJUG kost € 59,50 per jaar, waarbij Java Magazine gratis verschijnt. Naast het Java Magazine krijgt u gratis toegang tot de vele NLJUG workshops en het J-Fall congres. Het NLJUG is lid van het wereldwijde netwerk van JAVA user groups. Voor meer informatie of wilt u lid worden, **zie www.nljug.org**. Een nieuw lidmaatschap wordt gestart met de eerst mogelijke editie voor een bepaalde duur. Het lidmaatschap zal na de eerste (betalings)periode stilzwijgend worden omgezet naar lidmaatschap van onbepaalde duur, tenzij u uiterlijk één maand voor afloop van het initiële lidmaatschap schriftelijk (per brief of mail) opzegt. Na de omzetting voor onbepaalde duur kan op ieder moment schriftelijk worden opgezegd per wettelijk voorgeschreven termijn van 3 maanden.
Een lidmaatschap is alleen mogelijk in Nederland en België.
Uw opzegging ontvangen wij bij voorkeur telefonisch. U kunt de Klantenservice bereiken via: 023-5364401. Verder kunt u mailen naar **members@nljug.org** of schrijven naar NLJUG BV, Ledenadministratie, Nijverheidsweg 18, 2031 CP Haarlem.
Verhuisberichten of bezorgklachten kunt u doorgeven via **members@nljug.org** (Klantenservice).

Wij nemen je gegevens, zoals naam, adres en telefoonnummer op in een gegevensbestand. De verwerking van uw gegevens voeren wij uit conform de bepalingen in de Algemene Verordening Gegevensbescherming. De gegevens worden gebruikt voor de uitvoering van afgesloten overeenkomsten, zoals de abonnementenadministratie en, indien je daar toestemming voor hebt gegeven, om je op de hoogte te houden van interessante informatie en/of aanbiedingen. Je kunt uw persoonsgegevens opvragen om inzicht te krijgen in welke gegevens wij van je hebben, deze te corrigeren of, na beëindiging van de abonnee-overeenkomst, te laten verwijderen. Stuur hiertoe een kaartje aan NLJUG BV, afd. klantenservice, Nijverheidsweg 18, 2031 CP Haarlem of een e-mail naar **members@nljug.org**.

VOORWOORD

Vooruitkijken

We trappen 2024 af met een nieuwe editie van ons magazine, wederom boordevol interessante artikelen en een uitgebreide terugblik op de geslaagde 20ste editie van J-Fall. Wat was het een feest, die jubileumeditie van J-Fall in november! Met maar liefst 1.800 developers, + 80 top sprekers en 58 partners overtrof het alle verwachtingen. De sfeer zat er goed in en de line-up met sprekers uit binnen en buitenland, de inspirerende keynotes, De Innovation Award, Masters of Java, de Preconference workshops en break-out sessies werd gemiddeld genomen weer gewaardeerd met een mooie 7,6. Een mooier eerbetoon aan 20 jaar J-Fall was nauwelijks denkbaar. Op pagina 25 vind je een goeie sfeerimpressie van deze feestelijke dag!

Na dit geslaagde jubileum is het nu tijd om vooruit te blikken. Want de NLJUG community staat nooit stil. In deze editie duiken we in enkele inspirerende onderwerpen. Zo deelt Michael Koers tips om je code overzichtelijker en beter leesbaar te maken in *Writing Cleaner Code with these Simple Tips*. Voor de programmers is er *Fun with Functional Programming*, waarin Laurens van der Kooi je meeneemt in de wereld van monaden, of leer alles over de nieuwe encryptiestandaard in *Introduction to Key Encapsulation Mechanism API* van Ana Maria Mihalceanu. Dit en veel meer interessante onderwerpen komen aan bod!

Zou jij het leuk vinden om je kennis te delen? Dan kan dat hier! Heb je ervaring met een leuk/interessant Java-gerelateerde tool en zou je deze kennis willen delen? Waag dan een poging en stuur een e-mail naar **info@nljug.org**. Ook als het je eerste artikel is, is dat geen probleem, we begeleiden je graag! Laat je verrassen door de diversiteit aan onderwerpen!

Ik wens je veel leesplezier en inspiratie voor het nieuwe jaar!

Lieke Stomps
Projectcoördinator NLJUG
info@nljug.org


THE WORLD NEEDS MORE INNOVATORS
#STARTWITHUS
A team of six people created Tikkie. After successfully launching an app like Tikkie, it would be easy to say "Okay, we made it". But that's not how innovation works. Innovation is about constantly challenging yourself and spotting opportunities. Daring to acknowledge that something can always be better. Because it's this mindset that makes the difference. And to make a difference we need more innovators – people like those who developed Tikkie and Groepie. We need you. Employees that will continue to motivate others, and us.
www.workingatabnamro.com
ABN·AMRO

INHOUD

Semantic Kernel

14 Large Language Models (LLMs) are changing the way we develop and interact with applications and software. From back-office employees to business users to developers, natural language is the future of user interaction. This brings challenges and innovation in the way we build and code AI applications.

Jakarta EE 11

28 The work on defining the next major update of Jakarta EE has been going on for a while. Ed Burns (Microsoft), with the help of Arjan Tijms (OmniFish), has taken on the role of release coordinator for Jakarta EE 11. By the time this article is published, a plan should be presented and available publicly.

Toegankelijkheid

32 Laten we direct antwoord geven op de vraag: “Zijn er echt mensen die niet op de gewone manier met apps en websites kunnen omgaan?” Het antwoord is een duidelijke JA! Er zijn veel mensen met een handicap die permanent, tijdelijk of situationeel is. Deze mensen kunnen moeilijkheden ondervinden met het gebruiken van apps en websites. Sterker nog, als deze apps en websites een digitale twin zijn van de fysieke wereld, kunnen deze mensen ook uitgesloten worden uit de fysieke wereld.

SCHRIJVEN VOOR JAVA MAGAZINE?

Ben je een enthousiast lid van NLJUG en zou je graag willen bijdragen aan Java Magazine? Of ben je werkzaam in de IT en zou je vanuit je functie graag je kennis willen delen met de NLJUG-community? Dat kan! Neem contact op met de redactie, leg uit op welk gebied je expertise ligt en over welk onderwerp je graag zou willen schrijven. Direct artikelen inleveren mag ook.
Mail naar info@nljug.org en wij nemen zo spoedig mogelijk contact met je op.

SIMPLE TIPS FOR CLEANER CODE!

A less smelly codebase!

Everyone wants to write clean code, right? Sadly, writing clean code is not always as straightforward as you might think. If you don't know how to recognize them, it's very easy to miss those pesky code smells during a pull request or even when writing code yourself! This article will showcase some not uncommon code smells, how to identify and solve them, making your codebase just a bit less smelly! Did you for example know the use of comments can, more often than not, be considered a code smell? Find out why in this article!

Michael Koers is a software consultant at Info Support, with most of his knowledge in Java and Azure. He enjoys sharing his knowledge, teaching and gets excited about home automation.

{ INTRODUCTION }

Writing clean code is much easier said than done. It's not always immediately evident what the cleanest solution is for a given problem. Code smells can be the result of coding alongside your thought process, without thinking about the structure of your classes or methods beforehand.

Sometimes it's hard to identify code smells. You look at a piece of code and you know there's something icky about it, but you just can't pinpoint where the smell is coming from, or maybe you don't know how to improve it.

This article will go over some code smells that can take you by surprise if you're not programming with clean code in mind. The types of smells discussed are picked because they can easily slip into your code, and resolving them won't take much time while also greatly cleaning up your code.

Showcasing the code smells via code snippets, this article will show you what the code smells look like, and how to remove the stink from it. There's plenty more code smells than what's discussed in here, but hopefully it will get you interested or even excited to start looking more into writing cleaner code!

{ NESTED LOGIC }

This kind of code smell is a good example of what happens when you code alongside your thought process without thinking about the structure beforehand. Let's say you need to write some logic in a method, however, this logic only needs to be executed if some boolean expression holds true. Without thinking, you might put the logic inside an if-statement. For an example, see Listing 1.

L1

```
// Smelly, all logic is in if-statement
public String transform(String someString) {
    if (!someString.isEmpty()) {
        someString = someString.strip();
        someString = someString.toUpperCase();

        /* More logic */
    }
    return someString;
}
```

In the smelly variant of the transform method, all logic is inside an if-statement. This way of writing code derives from your thought process going: "Okay, only execute this logic if the given String is not empty". However, this quickly leads you to putting all your

logic inside an if-statement, which is smelly. Why? Because indented code is harder to read. Over-indentation makes it harder to easily scan the code, because the statements are spread across the page instead of being in a compact format [1].

The solution to this problem is very simple: you invert the if-statement. Instead of nesting all the logic in an if-statement, the if-statement is inverted - stopping the execution of your method if that boolean expression doesn't hold true. See the refactored version in Listing 2. Now your logic doesn't need to be nested any more. This kind of solution is called a "Guard Clause".

L2

```
// Refactored by inverting if-statement, applying a guard-clause
public String transform(String someString) {
    if (someString.isEmpty()) {
        return someString;
    }

    someString = someString.strip();
    someString = someString.toUpperCase();

    /* More logic */

    return someString;
}
```

A more extreme version of this kind of code smell is when you have multiple nested if-statements resulting in what you see in Listing 3. All this nesting makes the code look and feel very convoluted.

L3

```
// Smelly, all logic is nested
public static String transform(String someString) {
    if (!someString.isEmpty()) {
        if (someString.length() < 5) {
            someString = someString.toUpperCase();
        } else {
            if (someString.length() > 10) {
                someString = someString.toUpperCase();
            } else {
                someString = someString.toLowerCase();
            }
        }
    }
    return someString;
}
```

This problem is actually rather simple to solve if you know how to approach it. Find the outermost condition and turn that into a guard clause. In case of this example, the outermost condition is the .isEmpty() check. By changing this into a guard clause we can

L4

```
// Refactored by inverting if-statements using guard clauses
public static String transform(String someString) {
    if (someString.isEmpty()) return someString;
    // consolidated conditional expression
    if (someString.length() < 5 || someString.length() > 10) return someString.toUpperCase();

    return someString.toLowerCase();
}
```

remove one layer of the nested logic. Now repeat this process for all eligible layers and the result is a method that has a much clearer flow. See Listing 4 for the refactored version.

If you notice multiple conditions returning the same result, you can consolidate the conditional checks into a single conditional check using the '`and`' or '`or`' operator, reducing the cyclomatic complexity a bit.

{ SPLIT STATE AND BEHAVIOUR }

This code smell is interesting as it goes against the idea of *Object Oriented design*. With this code smell, state and behaviour of an object are separated in two or more classes, whereas they could've been part of the same object. See Listing 5. In the example, because the UserManager needs to modify information on the User, User's fields have to be accessible, which prevents information hiding.

This code smell can be harder to detect, but one way to keep an eye out for this problem is to check if a method inside a class solely operates on the method parameters. This indicates that the class doesn't hold any data used by the method, meaning state and behaviour are out of sync here.

L5

```
// Smelly, state (User) and behaviour (UserManager) are separated
class User {
    int securityLevel;
    boolean admin;
}

class UserManager {
    private static final int MAX_LEVEL = 10;

    boolean hasSecurityClearance(User user, int req) {
        return user.admin || user.securityLevel >= req;
    }

    int determineClearanceLevel(User user) {
        return user.admin ? MAX_LEVEL : user.securityLevel;
    }
}
```

With this problem, you want to bring state and behaviour back together. In case of the example, this means moving the methods from User Manager to the User object itself. The User's fields don't have to be accessible any longer and can be hidden behind the class's methods. Now you can check a User's security level on the User object itself, which feels much cleaner and also requires one less parameter now. Another benefit is that the User object is now more in control of what happens to its state, paving the road to immutable objects. See Listing 6 for the refactored version.

L6

```
// Refactored to combine state and behaviour
class User {
    private int securityLevel;
    private boolean admin;

    boolean hasSecurityClearance(int req) {
        return admin || securityLevel >= req;
    }

    int determineClearanceLevel() {
        return admin ? MAX_LEVEL : securityLevel;
    }
```

In my experience, you're more likely to find this code smell in utility-like classes as they tend to "manage" other objects.

{ COMMENTS }

Part of SonarQube's metrics collection is the "*Comments (%)*" that measures how many comment lines you write in a file compared to code [2], on which you can set certain thresholds for your quality gates to act on. But if you manage to write clean code with clear naming and concise methods, writing any comments would be arbitrary. So why would you want to write comments in that case?

Depending on who you ask, you might get different answers on if and how many comments one should write in their application. If you search the web for when to write comments, you'll probably run into someone answering that question with "code should be self-explanatory", which is true, but is such a shorthand answer it's difficult to understand what that entails. There are certain things that simply cannot be captured by writing clean code, like decisions made for certain solutions.

So comments can be good, but can also often be considered bad. Examples of bad comments are:

› **Superfluous comments**: Comments that don't add any new information as opposed to what you can already read or understand from the code itself. These kinds of comments should just be removed.
› **Commented out code**: Code that used to be a part of the system, but was commented out at a certain point in time. Commented out code snippets add clutter and should simply be removed. By now all projects should (hopefully) be placed in some kind of version control, so if you ever need that code back, you'll have your version control to retrieve it. If you really want to, you could leave a comment about the previous existence of the commented out code, in case other developers have to check out the removed code again.
› **Comments explaining variables/methods**: Comments explaining what methods do or what variables mean are simply masking a code smell. It means the method or variable should be renamed to more clearly reflect what they do or mean. Once refactored, the comment can be removed.

But not all comments have to be bad. Some examples of good comments are:

› **Documenting implementation decisions**: Implementation decisions can't be derived from method or variable names, so these could be documented using comments in your code. Documenting them makes it clear for other developers why a certain decision was made. Valid reasons could be compatibility, a bug, or other limitations.
› **JavaDoc:** Documenting your packages, classes, interfaces, methods and/or constructors with JavaDoc is generally a good idea, especially if it concerns code, like an API, that is shared across teams. Writing good JavaDoc will allow other developers to understand and properly use your code.

See Listing 7 for an example filled with useless comments, and Listing 8 for the same code, but this time decorated with much more meaningful comments.

Remember that comments, if not done properly, just mean more clutter to the code and more lines for developers to read. Only write comments when they really add value and keep them concise to make your fellow developers happy.

{ CONCLUSION }

This article only points out a few code smells, but there are many more kinds of code smell that can sneak their way into your codebase. If by now you are interested in reading more clean code tips in a similar fashion as this article, check out the book *Java by Comparison, by Simon Harrer, Jörg Lenhard and Linus Dietz* [3]. It touches upon 70 different clean code tips, all accompanied with before and after code examples, making the problem and solution as clear as possible. Another good read on refactoring is Martin Fowler's book Refactoring, which touches upon the same subject [4].

L7

```
// Smelly, lots of comments that don't add any value or mask a code smell
class Order {

    // List of items in the order                   <- Superfluous
    private List<String> list = new ArrayList<>();
    private double total;

    // Add item to order                            <- Superfluous
    void add(String item, double price) {
        list.add(item);
        total += price * 1.08d; // Add sales tax     <- Explains constant
        Collections.sort(list);
        // list.sort((o1, o2) -> o1.compareTo(o2)); <- Commented out code
    }

    // See if order contains some other item        <- Explains method
    boolean isInOrder(String o) {
        // Fast search                              <- Poorly documented decision
        return Collections.binarySearch(list, o) != -1;
    }
}
```

L8

```
// Refactored, commented out code and superfluous comments have been removed
// Design implementation has been documented more clearly and the
// sales tax has been turned into a constant
class Order {
    private List<String> items = new ArrayList<>();
    private double total;

    /**
     * Adds given item to the order
     * @param item      Item name, to add to the order
     * @param price     Item price, must be excluding sales tax
     */
    void add(String item, double price) {
        items.add(item);
        total += price * SALES_TAX;
        // Important to keep collection sorted, see contains() method
        Collections.sort(items);
    }

    /** {...}
     */
    boolean contains(String other) {
        /*
         * We use Binary Search on the items collection because
         * we noticed performance issue when the size of the order
         * grew to be > 1000, this drastically improved search performance
         * but requires us to keep the collection sorted at all times.
         */
        return Collections.binarySearch(items, other) != -1;
    }
}
```

Hopefully you learned some new clean coding tips to apply in your daily work, and don't forget to educate your fellow colleagues/developers so we can all start writing cleaner code! ‹

{ REFERENCES }

1 https://www.cs.umd.edu/~ben/papers/Miara1983Program.pdf
2 https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/#size
3 https://java.by-comparison.com/
4 https://martinfowler.com/books/refactoring.html

IMPROVING YOUR TESTCONTAINERS EXPERIENCE

Testcontainers. An easy tool that gives you a lot of power. All it requires is a Docker-compatible environment to run your test cases and some will write them. It is simple to start using it.

So you start using it. You write several test cases that make you sleep better at night. Especially, if the system under test is a (let's not be afraid of this word) *legacy system*. You don't have to worry about maintaining all these mocks/fakes/stubs/spies/..., you just use real dependencies (as long as they run using Docker, that is), and you don't have to worry that much about migrating them to newer versions, you can easily use different data sets for testing your *Important Stuff*™.

Time flies. Several pair programming sessions with your teammates later, a long weekend or two, and a holiday, and your operations and accounting teams start having some questions. Your manager might also touch on the same topic during a call with you because the CI/CD pipeline is taking longer and longer (read: pricier and pricier) to run.

What's worse, your fellow developers start skipping local test runs, and they *git push* more and more often, so the CI can take the heavy lifting instead of their dev machines, and as a result the CI not only burns more money, but also red builds happen more and more often (because the code isn't tested locally before pushing).

The atmosphere gets dense, "you brought the technology to the team - you own it", someone uses the word "abandon", and the fever grows... What can you do when the fever grows? You break the thermometer, obviously, so that `no reading -> no symptoms -> no fever` ;-)

Perhaps you'll allow me to stop here.. Let's have a chat, discuss the reasons, and propose some options.

Piotr Przybył is a Senior Developer Advocate at Elastic, Java Champion, Testcontainers Champion, Oracle ACE

{ ANY SOPHISTICATED TECHNOLOGY FEELS LIKE MAGIC. ESPECIALLY WHEN IT'S EASY TO USE }

Testcontainers is not different here. From a certain angle, Testcontainers may seem like a "simple wrapper for Docker": instead of controlling the containers (read: dependencies) using terminal, shell scripts, and (heavens forbid) the pesky YAML files, you can do that using the language you know: Java. This approach has a lot of advantages, like integration with *@BeforeAll* for starters. The disadvantage is that it hides the complexity and the cost from your sight, and more importantly, all the people who shamelessly copy and paste your test cases, just to have their own ones.

I'd say it's important to not forget that your tests using Testcontainers stand on the shoulders of a giant, called *Docker*. If something doesn't make much sense in terminal/manual use, it doesn't make much sense when translated to Java4 either.

Here comes my subjective list of Things That Should Be Avoided when writing TC-powered tests, when it comes to performance/run duration. Before applying, make sure it fits your context, please.

Another thing I'd like to ask you is to have a scientific approach to this: don't make a batch of hectic changes, hoping for the best. As

with any optimization:

- Measure
- Identify the biggest obstacle
- Change one thing and one thing only
- Measure again
- Withdraw if no visible improvement
- Repeat

{ DON'T DO SOMETHING IF IT DOESN'T MAKE SENSE }

Chances are in your project there are many kinds of tests. Not everyone is happy with the exact shape of the testing pyramid, but tests with *mocks* are perfectly fine for many occasions. One of their strong points is their execution speed: getting rapid feedback, within a second or a few, is their big selling point. However, if you execute them together with tests using *real dependencies* (thanks

to Testcontainers) you lose this benefit. Even if they fail fast, how can *Maven* or *Gradle* know they should stop the build and mark it red? For this reason, fast tests and not-so-fast tests shall be executed in two subsequent steps: only if the fast detection grid is okay, we summon the real dependencies.

You can achieve this in many ways: some people prefer *high-level separation* (even in different modules/subprojects), some are okay just with a *separate source set*, some opt for plugins (like *Surefire* and *Failsafe*), for some having only a naming convention (for example *...IntTest.java*) is okay. Whatever you do, please make sure you call Testcontainers only after mocks are okay.

Don't do many times what you can do only once

Let's imagine that you're a manual tester and your tests are written on paper. And the paper says, for every test:
- start the database (e.g. using a shell command)
- click, type, click
- verify
- stop the database

At first glance, it seems to be okay. After all, you have to start the DB for the test to make things testable, right? Only after you run test after test, after test, you realize that you spend a lot of time just waiting for the DB to start. And as a clever person, instead of starting and stopping the database, you just bring it to the right state before the next test instead of starting it from scratch because a simple `DROP SCHEMA; CREATE SCHEMA` works just as well, only it doesn't take one minute (or even a half).

The same applies to databases (and other dependencies) operated by Testcontainers. Just look at how many times you start them. If you keep them as instance fields, annotated by *@Container*, the framework will restart the containers for you before each test! For this reason, you want your containers to sit under *static* fields, so they are started only once for test execution. And if you need to bring them to proper shape before each test (e.g. by setting a proper DB schema), this is what you do in *@BeforeEach*.

On a side note: I saw quite a few times that people extend TC container classes, i.e. create their own container subclasses. That is often a bad smell of "code reuse" because they say "Oh, we don't want to copy this long customized container config to every test class". And this is exactly what you should try to avoid doing most of the time. "Every test class" should refer to the same *static Container oneContainerForAllClasses*-field, instead of creating or inheriting its own instance fields.

{ DON'T WAIT FOR NON-BLOCKERS }

Let's imagine you're the manual tester again. If the test scenario says `start the DB` in step one, and in step two `open your browser`, you don't have to wait for the DB to be `ready for requests`, staring at the startup log, before you open your browser, right? Heck, in the meantime you'll even squeeze taking a look at your mailbox!

So if doing many things at the same time works in real life, why don't we encode this approach in automated tests? If your test suite relies on two or more dependencies started by Testcontainers, don't start them sequentially when not needed, but start them in parallel. Say, if one container starts in 35 seconds and the other in 20, instead of waiting for 55 seconds in total, you may wait for just 378!

This is easy to do. If you rely on the annotations (and you keep your stuff updated, so you use at least Testcontainers version 1.19.0), then simply go for `@Container(parallel=true)`, and Bob's your uncle! If you start your containers manually (which is perfectly fine!), just make that start parallel too. You can go for a parallel stream or simply do `Startables.deepStart(containerA, containerB,...).join()`

{ PARALLELISM: WE HAVE TO GO DEEPER }

I bet you have ordered something online. And changes are, there were times when you received like three or four parcels the same day, during a single delivery. Why is it so? Couldn't the delivery person just bring you the first box, then go back to the storehouse, bring another, and so on? Well, they could, but that would make no sense because this truck of theirs can surely take all your parcels at once, so they can utilize the hardware to its full capacity (instead of underutilization).

The same happens to your CI environment... When your Testcontainers-powered tests run, take a look at *htop*. Or even better, monitor the CPU/RAM/network consumption. If you can't, just start using *Java Flight Recorder*. However, you can just grab the data.

I suspect that you may see something similar:
initial spike when the containers are started
hardly any CPU consumption when the tests are running

This indicates that your delivery truck or CI machine is mostly filled with air, instead of cargo. Worry not, you can easily run many forks of your tests. For example, in Gradle, you can set *maxParallelForks*, in Maven *forkCount*, or do that in any other way. What matters is that you load your truck and CPU as much as you can (I guess you pay your cloud provider per minute, not CPU load).

I can't tell you exactly what numbers you should put here. But I can share some hints:
The number of forks will be different for "mock tests" and "Testcontainers tests", as they have different CPU consumption characteristics. However, since you run them as separate steps now, that should not be a problem.

Don't just set `forks=CPU_cores/2`. That's a good starting point, but it might turn out you can shorten your build by increasing this number. Or decreasing, because of congestion issues. It takes some experimentation and test runs (pun intended) to figure out the sweet spot, it depends on what your tests are doing.

Please keep in mind that the number of forks varies between the machines, so I guess I'd put that as an env variable, not just hard-code it in *POM* or *YAML*.

{ AGAIN: DON'T DO MANY TIMES WHAT YOU CAN DO JUST ONCE }

Oftentimes preparing the container to work as a dependency takes some time apart from just starting. A common example of this is various schema migrations or data preparations, with tools like *Flyway*, *Liquibase*, or the like. If you have two or three steps, it's usually not a big deal, but once you start having dozens of them (even with containers running locally) this might have a noticeable effect.
Let's think about it: if you have 42 DB migrations and 50 tests, does it make sense to run all these migrations 50 times? Does it add any value? What it does for sure is a round-trip delay, especially if your containers aren't running locally. There are ways to mitigate this (like batching), but the ultimate killer might be having just one small network call before each test, telling the DB to restore to its desired state by using a cache.

The trick isn't rocket science:
After starting the DB in the container capture its state (keep the file in the container, you may also copy it back to your machine)
Before running each test, instead of running all 42 migrations (especially from the machine where the tests are running), simply drop the schema and recreate it using the DB dump created in the step before.

There are variations of this technique: some are using pre-baked images in the company's registry, some prefer having an extra store procedure to clean all the tables, and so on. Your mileage might vary, yet the key part is this: capture the state of your dependency when it's ready to serve your tests and restore it in the thinnest single network call possible. Of course, this is some sort of caching, and as such it's important to evict the cache when necessary (which is one of the two major IT problems).

{ Developer stations can reuse }

Perhaps this hint should be called "don't do many times what you can do just once", but I've already used it twice ;-). Think about the old days before Docker: if you were testing something manually using a DB, you didn't start and stop the DB between every test, did you? You can tell Testcontainers to do the same thing: don't stop-and-start containers between the tests, but keep them running. This reuse feature is experimental, but has been around for a while already. Enabling it is a two-step process:
If you want your container to be reused, when constructing it, call `.withReuse(true)` in the chain
The environment where the tests are running needs to be allowed to keep the containers running, this can be done e.g. by adding `testcontainers.reuse.enable=true` to `~/.testcontainers.properties`.
This way you can tune where the containers are to be reused (developer station, when a single test is run over and over again seems to be a good place) by opt-in, and skipped the other places (like Jenkins running on-premises, where too many containers would eat up the whole RAM).

Please keep in mind that the containers will be running as long as no `docker stop` is called or e.g. the whole machine is restarted. Also, this mechanism relies on container hashes, so if you change the container definition in a way that has an impact on the hash, the library will see it as a new container which has to be started (and you may end up with many stale containers running in the background).

{ IS THAT THE LIMIT? }

Not at all, your journey doesn't have to stop here. There are still many techniques that can shorten your build because you start and use your dependencies more smartly. Just keep in mind that these are real things running in Docker containers. This mindset (combined with actual data on how long things take) can help you from here. E.g. on Linux runners, you can use *tmpfs*, so your DB container can store data in RAM, thus avoiding slow IO (and since these containers are ephemeral, you don't care that the data is lost after the tests). If your data tells you that you have reached RAM capacity (so you can't squeeze in more test forks and containers because you have no more RAM), perhaps offloading the containers to another machine and using remote Docker execution can help you a lot. You can do that on your own, your *OpsTeam* can help you, or you can go for Testcontainers *Cloud* (a SaaS to run your TC containers for you) if you prefer easy solutions. If you want to squeeze the lemon for more juice, you can even go for *image prefetching* and other advanced optimization techniques.
Sometimes it's even possible to increase the number of integration tests five times and still go from ten minutes to twenty seconds ;-). For some inspiration and code examples please go to **https://github.com/pioorg/Testcontainers-performance-up**. Good luck!

AI ORCHESTRATION WITH SEMANTIC KERNEL

Why AI orchestration

Large Language Models (LLMs) are changing the way we develop and interact with applications and software. From back-office employees to business users to developers, natural language is the future of user interaction. This brings challenges and innovation in the way we build and code AI applications.

We need modern ways to manage these AI applications. *AI orchestration* is the process of managing multiple AI applications and services in a coordinated way. It helps to optimize the performance, scalability, and reliability of AI solutions. AI orchestration is solving the problem of complexity and fragmentation in the AI landscape, where different tools, platforms, and frameworks are used for different tasks and goals.

{ WHAT IS A SEMANTIC KERNEL? }

Semantic Kernel (SK) is a powerful SDK that allows you to combine traditional programming languages, such as Java, C#, and Python, with the most advanced *Large Language Model* (LLM) AI "prompts" that support prompt templating, chaining, and planning features. This lets you create new functionalities in your apps that can boost the productivity of your users: for example, summarizing a long chat conversation, highlighting an important "next step" that's automatically added to your to-do list, or planning a whole vacation instead of just booking a flight, see image 1.

{ COMPONENTS OF SK }

There are several components within SK to build AI applications. In the following section let me walk through them and explain their usage know-how.

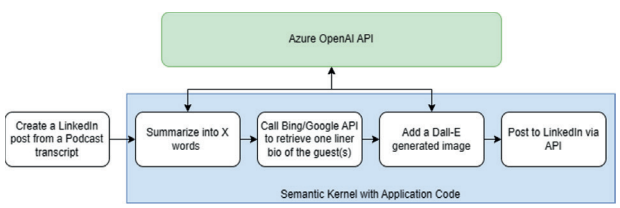


Image 1.

Soham Dasgupta is an Azure Cloud Solution Architect at Microsoft working with Dutch ISVs.

{ KERNEL }

The *kernel* orchestrates a user's task. To do so, the kernel runs a pipeline or chain of tasks that is defined. While the pipeline or chain is executing, a common context is provided by the kernel so data can be shared and passed between those underlying tasks.

First to create a Kernel we need the OpenAI endpoint and model details. In the example you find in Listing 1 we are using Azure OpenAI endpoint.
Then to initialize Kernel, we would need to read these properties (in this example from *conf.properties* file), see Listing 2.
You can select the LLM service while instantiating Kernel objects, for example, all models of *OpenAI*, *Azure OpenAI* or *Hugging Face*.
Also, to gain visibility of what the Kernel is doing you can add *telemetry* and *logs* to this object.

{ PLUGINS }

Plugins are the body of your AI app. They consist of prompts and native functions. You can connect your application to AI plugins, enabling interactions with the real world. By leveraging plugins, you can encapsulate various capabilities into a cohesive unit of functionality. This unified functionality can then be executed by the kernel. Plugins have the flexibility to incorporate both native code and requests to AI services through semantic functions.
A plugin consists of one or more Semantic functions. To create a Semantic function, you need to define a **skprompt.txt**, which holds the prompt you want to use with input definitions.
See Listing 3 for the example of a `Translate` function.

L1

```
client.azureopenai.key=XXYYZZ1234
client.azureopenai.endpoint=https://nljug.
   openai.azure.com/
client.azureopenai.deploymentname=gpt-35-turbo
```

L2

```
AzureOpenAISettings settings = new
AzureOpenAISettings(SettingsMap.
                    getWithAdditional(List.
of(new File("src/main/resources/conf.
properties"))));

OpenAIAsyncClient client = new
OpenAIClientBuilder().endpoint(settings.
getEndpoint())
        .credential(new
AzureKeyCredential(settings.getKey())).
buildAsyncClient();

TextCompletion textCompletion = SKBuilders.
chatCompletion()
        .withOpenAIClient(client)
        .withModelId(settings().
getDeploymentName())
        .build();

Kernel kernel = SKBuilders.kernel().
withDefaultAIService(textCompletion).build();
```

To provide a semantic description of this function (and configure the AI service), you'll need to create a **config.json** file in the same folder as the prompt. This file outlines the function's input parameters and description as you see in Listing 4.
To use this in the Kernel object and execute the Semantic function, you will need to write the lines of code as shown in Listing 5.
We can also create Plugins and functions as a *Java class*. Then we can instantiate and use it in a type-safe way. Semantic Kernel provides a few out-of-the-box Plugins to use, for example, `ttpSkill` (to invoke REST endpoints), `ConversationSummarySkill` (to summarize, get actions items etc.). You can use these directly in your application with adding a dependency : `semantickernel-plugin-core`, But it is also possible to implement your own, using annotations `@DefineSKFunction`, `@SKFunctionInputAttribute`, `@SKFunctionParameters`.

This pattern of defining plugins increases reusability in different applications and is specifically used to invoke external APIs, or some execute business logic and is useful when you chain different semantic functions.

L2

```
Translate the input below into {{$language}}

MAKE SURE YOU ONLY USE {{$language}}.

{{$input}}

Translation:
```

L4

```
{
  «schema»: 1,
  «type»: «completion»,
  «description»: «Translate the input into a
language of your choice»,
  «completion»: {
    «max_tokens": 2000,
    «temperature»: 0.7,
    «top_p": 0.0,
    «presence_penalty": 0.0,
    «frequency_penalty": 0.0,
    «stop_sequences": [
      «[done]»
    ]
  },
  «input»: {
    «parameters»: [
      {
        «name»: «input»,
        «description»: «Text to translate»,
        «defaultValue": ""
      },
      {
        «name»: «language»,
        «description»: «language of
translation»,
        «defaultValue": ""
      }
    ]
  }
}
```

{ CHAINING PLUGINS/SEMANTIC FUNCTIONS }

Chaining plugins or semantic functions involves linking multiple functions together to create a cohesive pipeline. Each function processes input data and passes the result to the next function. This allows developers to build complex workflows, combining various AI services and native code seamlessly. Think of it as assembling a series of interconnected building blocks, where each function contributes to the overall functionality of the application. Referring to the first image of this article, where a LinkedIn post is

produced from a podcast with several interconnected steps is a perfect example of chaining pipelines.
The code in Listing 6 demonstrates chaining of Plugins where Summarize and Translate happens one after another for a given input text.

{ PLANNERS }

Planner in the Semantic Kernel is a powerful feature that automatically orchestrates AI services. It takes a user's request and generates a plan on how to achieve it. By intelligently combining registered plugins, planners create workflows for tasks like reminders or complex data mining, enhancing the flexibility and efficiency of AI-powered applications.

For a given construct, as you find in image 2, if the incoming task is to Summarize and then Translate a given text, the Planner will pick up "*Summarizer*" and "*Translator*" functions to generate the outgoing result, see image 3.

There are 3 different planners available in Semantic Kernel, *Action*, *Sequential* and *Stepwise*.

The code in Listing 7 shows a Sequential planner which has 3 Plugins with multiple semantic functions in each of them, and depending on the task it will intelligently pick the semantic functions necessary to perform the task.

{ GIVING MEMORIES TO SK }

What's memory? Think of giving LLM information to support it when executing a (or a set of) plugin(s). For example, you want to get a summary of your dental insurance details from a 100-page insurance document, here you might choose to send the whole document along with your query to the LLM, but every model in LLM has a limit on tokens what it can process with per request. To manage that token limitation, it is helpful if you can search the document first, get the relevant pages or texts out of it and then query the LLM with those selected pages/texts. That way LLM can efficiently summarize the content and not run out of tokens per request. The *selected pages/texts* in this example are *Memories*. For this example, we will be using 2 Kernels, one with *AI Services*

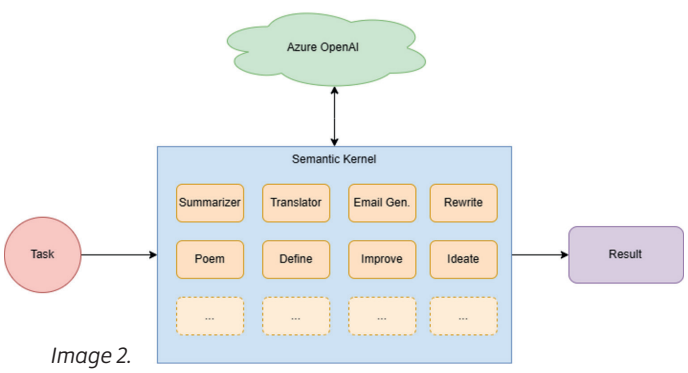


Image 2.

L5

```
ReadOnlyFunctionCollection skill = kernel.

importSkillFromDirectory("TranslateSkill", "src/
main/resources/Skills", "TranslateSkill");
CompletionSKFunction translateFunction = skill.
getFunction("Translate", CompletionSKFunction.
class);
SKContext translateContext = SKBuilders.
context().build();
translateContext.setVariable("input", "How are
you doing?");
translateContext.setVariable("language",
"Dutch");
Mono<SKContext> result = translateFunction.
invokeAsync(summarizeContext);
```

L6

```
kernel.
importSkillFromDirectory("SummarizeSkill", "src/
main/resources/Skills", "SummarizeSkill");
kernel.
importSkillFromDirectory("TranslateSkill", "src/
main/resources/Skills", "TranslateSkill");

SKContext summarizeContext = SKBuilders.
context().build();
summarizeContext.setVariable("input",
ChatTranscript);
summarizeContext.setVariable("language",
"dutch");
Mono<SKContext> result = kernel.runAsync(
                summarizeContext.getVariables(),
                kernel.
getSkill("SummarizeSkill").
getFunction("Summarize"),
                kernel.
getSkill("TranslateSkill").
getFunction("Translate"));
```

type Embedding and another one with *TextCompletion*.
Kernel with embedding needs a *MemoryStore*, this can be in-memory or any vector store. Although, right now, SK in Java supports only Azure Cognitive Search, support for other vector stores will be coming soon.
The kernel with Embedding with Azure Cognitive Search (ACS) as data or vector store is instantiated as you see in Listing 8.
If documents and/or data are already indexed in ACS, you can search on it, which returns the relevant sections of the memory, see Listing 9.

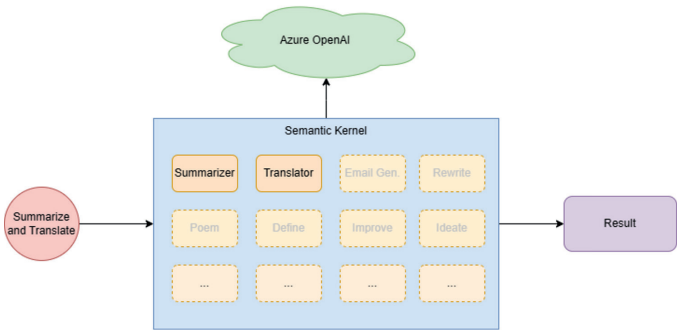


Image 3.

L7

```
kernel.importSkillFromDirectory("WriterSkill",
"src/main/resources/Skills", "WriterSkill");
kernel.
importSkillFromDirectory("SummarizeSkill", "src/
main/resources/Skills", "SummarizeSkill");
kernel.
importSkillFromDirectory("DesignThinkingSkill",
"src/main/resources/Skills",
"DesignThinkingSkill");

SequentialPlanner planner = new
SequentialPlanner(kernel, new
SequentialPlannerRequestSettings(
                    <relevancyThreshold>,
<maxRelevantFunctions>, Set.of(), Set.of(), Set.
of(),
                    <maxTokens>
                    ), <SystemPrompt>);

Mono<SKContext> result = planner.
        createPlanAsync("rewrite the
following text in Yoda from Starwars style" +
<TextToSummarize>)
        .invokeAsync();
```

L8

```
EmbeddingGeneration<String>
textEmbeddingGenerationService =
        SKBuilders.textEmbeddingGeneration()

.withOpenAIClient(openAIAsyncClient)
                .withModelId("embedding")
                .build();
Kernel kernel = SKBuilders.kernel()

.withDefaultAIService(textEmbeddingGenerati-
onService)
        .withMemoryStorage(new AzureCognitiveSe-
archMemory("<ACS_ENDPOINT>", "<ACS_KEY>"))
        .build();
```

L9

```
Mono<List<MemoryQueryResult>> relevantMemory =
        kernel.getMemory()
        .searchAsync(<INDEX_NAME>,<QUERY> , 2,
0.7f, true);
```

L10

```
List<MemoryQueryResult> relevantMems =
relevantMemory.block();

StringBuilder memory = new StringBuilder();
relevantMems.forEach(relevantMem -> memory.
append("text: ").append(relevantMem.
getMetadata().getText()));

Kernel kernel = kernel();
ReadOnlyFunctionCollection
conversationSummarySkill =
kernel.importSkill(new
ConversationSummarySkill(kernel),null);

Mono<SKContext> summary =
conversationSummarySkill
.getFunction("SummarizeConversation",
SKFunction.class)
.invokeAsync(relevantMemory);
```

Then we can use these search results as an extra context while invoking LLM, for example, a Summarizer function on the search results of Listing 9.
The example in Listing 10 uses Summarizer defined as a Java class using annotation `@DefineSKFunction`.

{ WRAPPING UP }

This article introduces SK, there are many more components and patterns you can implement using this SDK which I could not include here. Reference [1] en [2] will take you to the official documentation and the GitHub repo of SK.

If you are interested in playing around with examples, please check out my personal GitHub repo [3]. <

{ REFERENCES }

1 https://learn.microsoft.com/en-us/semantic-kernel/overview/
2 https://github.com/microsoft/semantic-kernel/tree/experimental-java
3 https://github.com/sohamda/SemanticKernel-Basics

NLJUG INNOVATION AWARD 2023

THE WINNER IS TRIOPSYS!

TRIOPSYS WINS THE NLJUG INNOVATION AWARD 2023

November 9th, during the 20th edition of Europe's largest Java conference, J-Fall 2023, the NLJUG Innovation Award was presented for the third time. TriOpSys emerged as the winner!

Among the three nominees selected by the jury, TriOpSys was chosen by the J-Fall visitors for their Decision Support Tool (DST), developed for the Air Traffic Control the Netherlands (LVNL). This recognition of their outstanding innovation in air traffic management is a testament to their exceptional work.

DST significantly improved efficiency and safety in air traffic management. The success lies in the expert integration of advanced technology and machine learning, optimizing flight operations and seamlessly aligning them with the capacities of Schiphol and air traffic controllers.

This victory is not just TriOpSys's; it symbolizes technical mastery and a forward-thinking approach.

The NLJUG Jury and the entire NLJUG community extend heartfelt congratulations to TriOpSys for this well-deserved win.

MASTERS OF JAVA 2023

Woensdag 8 november - Masters of Java 2023 vanuit Veenendaal: De jaarlijkse code challenge voor Java Developers van elk niveau.

65 enthousiaste developers, verdeeld over 42 teams stonden te popelen om te beginnen. Wat voor opdrachten zouden het zijn? Welke kennis heb ik nodig? Kan ik het wel?

{ OPDRACHTEN }

De deelnemers kregen 5 totaal verschillende opdrachten. Per opdracht was er een tijdslimiet van 30 minuten. Je had je hardcore java skills nodig en de opdrachten varieerden van het uitwerken van algoritmen tot het gebruik van bekende en minder bekende Java API's. Er kon niet gegoogled worden, wel hadden de duo's de beschikking over een extra toetsenbord. Per opdracht kregen de deelnemers een beknopt Java project waarvan je in regel slechts één bestand kan aanpassen.

{ SMÖRGÅSBORD OF EQUALS STUFF }

Een verzameling van drie kleine puzzels rondom de equals() functie in Java. Niet alle equals() functies zijn.. equal? De drie opdrachten werden in verschillende volgordes opgelost, afhankelijk van waar het team de meeste ervaring mee had.

{ THE BROWSER ACCEPT-LANGUAGE HEADER }

De Accept-Language header is een stukje protocol om de juiste taal te selecteren op een website. De onderhandeling hierover gaat via de Accept-Language header van een browser. De opdracht was om dit te bouwen. Het snelste team had dit heel snel handmatig in elkaar gezet, de nummers twee waren iets slimmer (maar toch nog langzamer) en gebruikten hiervoor simpelweg de bestaande Java API.

{ POKEMON CALCULATOR }

In deze opdracht moesten de deelnemers een bug maken. In de bekende Pokemon games zit een calculator met raar gedrag wanneer bepaalde talen geselecteerd zijn. Deze bug namaken bleek een flinke kluif, slechts drie teams hadden hem uiteindelijk goed.

{ MOJ GAME SERVER }

Gelukkig bleek dat de teams beter zijn in bug fixen dan in bugs maken. In deze self-diss ervaarden de teams hoe het is om aan de MoJ organisatie kant te zitten. Vorig jaar verliep het niet helemaal soepeltjes en in deze opdracht hebben de teams een half uur de tijd om een bug te vinden in de MoJ Game Server.

{ SUDOKU SOLVER }

Een klassiekertje: implementeer een solver voor Sudoku door zogenaamde singles, pairs, triples en quads te zoeken.

Ze konden alleen gebruikmaken van de standaard Java SDK, er was geen framework kennis nodig. Door gebruik te maken van een aantal meegeleverde tests konden de teams hun oplossing controleren en daar soms ook extra punten mee verdienen. Dachten ze de juiste oplossing te hebben, dan konden ze die indienen. Was de opdracht goed, dan kregen ze de resterende tijd aan punten plus een bonus.

{ UITSLAG }

Thomas Withaar en Maarten van der Zwaart | team OVSoftware | 8.352 punten

Jasper van Merle | team camel_case | 6.765 punten

Jaap Beetstra en Jeroen Rakhorst | team Exception Eradicators | 5.938 punten

Allen, proficiat met deze mooie prestatie en veel (spel)plezier met het First8 Conclusion bordspellenpakket. En voor de andere deelnemers: dank je wel voor jullie deelname, hopelijk zien we jullie bij Masters of Java 2024!

FIRST8 CONCLUSION

2024 MASTERS OF JAVA nljug

BENIEUWD NAAR DE OPDRACHTEN?

CRAFTING A CUSTOM MONAD-LIKE TYPE IN JAVA - PART 1

As a Java developer, I have a real passion for working with functional programming techniques. In my day-to-day work, I often find myself immersed in Java's monad-like container types like Optional and Stream. Recently, I've been exploring the concept of crafting my own container type, specifically designed for managing conditional operations in a functional manner. This endeavor led to the creation of the Conditional.

Laurens van der Kooi is a Java Developer at Ordina JTech. He enjoys sharing knowledge and is a big fan of the Spring Framework and Functional Programming.

Since I really enjoyed crafting this functional abstraction, this article is designed to bring you on this journey with me without diving into the practical implementation just yet!

{ MONAD }

First, let's define what a monad is. A monad is a design pattern used to handle computations and manage side effects in a purely functional way. It provides a structured approach to encapsulate values, operations, and control flow within a specific context.

When explaining monads to fellow programmers, I draw inspiration from Sander Hoogendoorn's [1] keynote at J-Fall 2021, titled: *The Zen of Programming* [2]. In his talk, he portrayed a monad as a box that can contain an object. Once the object is inside this box, a programmer can send functions to it. These functions are then applied to the object within the box, transforming it. Programmers have the flexibility to send as many functions as necessary to the box, ultimately retrieving the transformed object from within, see image 1. These sequences of functions are commonly referred to as functional pipelines.

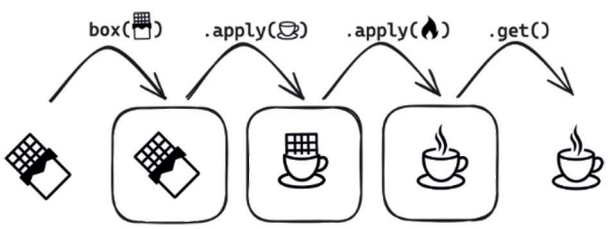


Image 1

The Java standard library offers three functional containers that closely resemble monads:

- **Optional**: A container that may or may not hold a non-null value;
- **Stream**: A container containing a sequence of elements supporting sequential and parallel aggregate operations;
- **CompletableFuture**: A container designed to handle the complexities of concurrent and asynchronous programming.

Monads have two fundamental operations. The Unit operation wraps a value into the monad (e.g. Optional.of()). Bind functions transform the object within the monad from one context to another; a common function in this category is flatMap.

Let's now explore the definition of a new type of monad-inspired container type.

{ THE IDEA: THE CONDITIONAL }

While working on a project in which I had to deal with quite some conditional business logic, I came up with the idea of creating a monad-like type that is able to organize conditional actions as a functional alternative to the traditional if-then-else-statements. The idea is basically to take a method that looks like Listing 1, and

L1

```
public static BigDecimal calculateNumber(int
number) {
    if (number % 2 == 0) {
        return BigDecimal.valueOf(number * 2.5);
     }

    if (number > 100) {
        return BigDecimal.valueOf(number * 0.5);
     }

    return BigDecimal.ZERO;
 }
```

turn it into a functional pipeline that elegantly expresses which action should be executed under specific conditions.

While playing around with this idea, I started writing code to figure out what this functional pipeline would look like without having an implementation yet, see Listing 2. Drawing inspiration from Java's *Optional*, I called this data type the *Conditional*. For the sake of readability and reusability, I opted to use separate helper methods (*times*, *isEven* and *isLargerThan*) instead of passing lambdas directly to the Conditional.

The *Conditional* in Listing 2 can be distilled as follows:

- Take an integer and wrap it in a Conditional (of);
- Apply the first action that contains a matching condition (`firstMatching`):
- If the number is even, multiply the number by 2.5 (`applyIf`);
- If the number is greater than 100, halve it (`applyIf`).
- Transform the resulting value into a BigDecimal (`map`);
- If none of the conditions are satisfied, default to BigDecimal.ZERO (`orElse`).

As you can read, this is the same logic as in Listing 1 but then composed in a functional manner. The `applyIf` within the *firstMatching* operation is added as syntactic sugar to improve the readability of the pipeline.

{INTERMEDIATE AND TERMINAL OPERATIONS}

In Java, operations in a functional pipeline can be broadly categorized into two types:

- **Intermediate** operations;
- **Terminal** operations.

Intermediate operations are operations that transform or filter the object/elements within a monadic type. These operations are applied to the object that is wrapped by the monad and return a

L2

```
public static BigDecimal calculateNumber(int
number) {
    return Conditional.of(number)
      .firstMatching(
         applyIf(isEven(), times(2.5)),
         applyIf(isLargerThan(100), times(0.5))
       )
       .map(BigDecimal::valueOf)
       .orElse(BigDecimal.ZERO);
 }

 private static Function<Integer, Double>
   times(double number) {
     return i -> i * number;
  }

 private static Predicate<Integer> isEven() {
     return i -> i % 2 == 0;
  }

 private static Predicate<Integer> isLargerThan
   (int number) {
       return i -> i > number;
   }
```

new monad containing the transformed object. This allows you to chain multiple intermediate operations together to create a pipeline. Translating this into the box analogy mentioned earlier, envision a chain of intermediate operations as a set of instructions that must be applied to the object within the box.

Terminal operations, on the other hand, are operations that produce a final result or side effect. Unlike intermediate operations, a terminal operation triggers the execution of the entire pipeline. They act as the endpoint of the sequence of transformations and typically return a non-monadic result or perform an action. Returning to the box analogy, this final operation becomes essential to initiate the execution of the list of instructions and get the transformed object from the box or trigger an alternative action.

Understanding the behavior and purpose of the different types of operations yields two key takeaways:

- Intermediate operations are lazy evaluated, meaning they are only executed when a terminal operation is invoked;
- A functional pipeline can contain multiple intermediate operations, but at most one terminal operation, always positioned at the end of the pipeline.

With these insights in mind, let's explore the functional pipeline using `Optional` in Listing 3. In this example, `Optional.ofNullable(number)` encapsulates an Integer within an Optional. Subsequently, as long as the Optional contains a value, it undergoes two types of intermediate operations:

- A filter operation (checking if the number is even and larger than 26).
- Three map operations (multiplying the number by 2, transforming from a Double to an Integer, and ultimately creating a message with the outcome).

The terminal operation, `orElse`, serves as the conclusion of this pipeline. It retrieves the String value from the Optional or defaults to *No calculation needed* if the Optional becomes empty during the process.

{ SPECIFYING OPERATIONS FOR THE CONDITIONAL }

Now that we understand the two types of operations in a functional pipeline, let's apply this knowledge to identify these types in the initial setup of the *Conditional* as seen in Listing 2. Since `orElse` produces a non-monadic result (the Double eventually returned by the `calculateNumber` method), it serves as the terminal operation in this pipeline. As a result, both the `firstMatching` and the `map` operation can be identified as intermediate operations.

L3

```
public static String
generateOutcomeMessage(Integer number) {
    return Optional.ofNullable(number)
          .filter(isEven().and(isLargerThan(26)))
          .map(times(2))
          .map(Double::intValue)
          .map("Outcome: %d"::formatted)
          .orElse("No calculation needed");
 }
```

L4

```
public class Conditional<S, T> {

    public static <S> Conditional<S, S> of(S
    value) {
        throw new
        UnsupportedOperationException();
     }

    // ... some methods we've specified

    public <U> Conditional<S, U> map(Function
    <T, U> mapFunction) {
        throw new
        UnsupportedOperationException();
     }

    public T orElse(T defaultValue) {
        throw new
        UnsupportedOperationException();
     }

    // ... some more methods we've specified

 }
```

So, for intermediate operations, we aim to have the following:

- `firstMatching`: capable of receiving condition/function pairs and applying the function of the first pair that has a matching condition;
- `map`: designed to apply a transformation from one type to the next within the functional pipeline.

As mentioned, a monad commonly includes `flatMap` as a bind-operation, so let's add:

- `flatMap`: designed to apply a function that transforms the value in the functional pipeline to another Conditional while unwrapping the outer Conditional (a concept we'll delve into in the next article).

For terminal operations, our preference is:

- `orElse`: either returns the value from the Conditional, or the value supplied to `orElse`.

Drawing inspiration from the Optional, let's also include:

L5

```
@Test
@DisplayName("a condition matches -> apply matching function")
void conditionalWithOneConditionThatEvaluatesToTrue() {
    var outcome =
            Conditional.of(2)
                    .firstMatching(applyIf(isEven(), times(2)))
                    .orElse(0.00);

    assertThat(outcome).isEqualTo(4.00);
 }

@Test
@DisplayName("no condition matches -> return default value")
void conditionalWithOneConditionThatEvaluatesToFalse() {
    var outcome =
            Conditional.of(3)
                    .firstMatching(applyIf(isEven(), times(2)))
                    .orElse(0.00);

    assertThat(outcome).isEqualTo(0.00);
 }
```

- `orElseGet`: either returns the value from the Conditional, or the result of the Function supplied to `orElseGet`;
- `orElseThrow`: either returns the value from the Conditional, or throws an exception to be created by the provided supplier.

{ A FIRST SETUP }

Now armed with the specification for the *Conditional*, we can begin crafting the class by outlining its methods, leaving the implementation for later stages. Listing 4 shows the start of this.

The Conditional class contains 2 generic types: *S* and *T*. The purpose of this is to represent a conditional operation where a value of type S is initially provided, and subsequent operations may transform it to a value of type T. The map method introduces the generic type *U* to facilitate the transformation of the object contained in the Conditional from one type to a new type. We'll delve deeper into these concepts in the next article.

As demonstrated, we can already outline the class and its method signatures. This allows for writing (non-working) functional pipelines, which we can use to write tests asserting the expected behavior. This aligns with the principles of Test-Driven Development (TDD). In Listing 5, we encounter the initial pair of tests that presently fail.

By implementing the methods that are used in these tests to make them pass, we can seamlessly alternate between expanding tests and incorporating logic into the Conditional. This iterative approach ensures steady progress towards completing the entire implementation.

{ CONCLUSION }

In conclusion, we've explored the conceptual landscape of crafting the Conditional. Armed with a detailed specification, we've initiated the class structure, outlining methods crucial for functional pipelines. In the next article, we'll dive into the practical implementation of the Conditional. Along the way, we'll explore key concepts of functional programming that are integral to understanding the implementation process.

If you're already eager to dive into coding, feel free to start implementing the Conditional yourself. Head over to my GitHub [3] to explore the setup of Listing 4, accompanied by over 35 unit tests to facilitate Test Driven Development. Rest assured, my GitHub won't reveal the implementation we'll be discussing in the next article yet ;). <

{ REFERENCES }

1 sanderhoogendoorn.com
2 https://www.youtube.com/watch?v=KSQTI8pFkrg
3 https://github.com/LvdKooi/javamag2024

.nl. jug

NLJUG thanks our sponsors for making J-Fall 2023 possible!

PLATINUM PARTNERS

ABN·AMRO

Rabobank

Red Hat

Rijksoverheid

GOLD PARTNERS

42

adyen

alliander

ASML

azul

bluerig

CAMUNDA

Capgemini

CGI

CHILIT

CONSPECT CONSULTING & ICT

Contrast SECURITY

DEV TALENTS

DEVOXX 4KIDS

EDSN energiedata voor iedereen

EVIDEN

FIRST8

foojay.io

group9 Elite. Java. Development

A proud member of the alfa1 group.

ilionx experts in eenvoud

InfoSupport Solid Innovator

ICT GROUP InTraffic Mastering Mobility

JCORE

kadaster

luminis Conversing worlds

Microsoft

NATIONALE POSTCODE LOTERIJ

nedap

Auth0 by Okta

ORDINA a Sopra Steria company

pan company proven IT experts

PIC NIC

Planon Building Connections

profit4cloud

QUAD

QUALOGY

Quintor

SynTouch

Sytac

THALES

topicus

TrailBlazers.

VX COMPANY achieve more.

Xebia

20 JAAR J-FALL

PATATKRAAM SPONSORED BY

ICT GROUP InTraffic
Mastering Mobility

HET LIJKT WEL EEN REÜNIE!

Wat is J-Fall toch een geweldig event. Eigenlijk een beproefd concept en toch ieder jaar weer anders, vernieuwend. Nieuwe onderwerpen, andere sprekers en nieuwe sponsoren. Wat Rabobank betreft was het afgelopen november weer een doorslaand succes. Hebben we dan alle Javanen aangenomen die daar aanwezig waren? Nee zeker niet maar dat is ook helemaal niet de insteek van ons sponsorship.

Maar waarom staan we er dan met zo'n grote stand als een van de sponsoren? Waarom willen we daar het gesprek aan gaan? Omdat we geloven in het delen van kennis. Rabobank is een grote Java speler. Onze IT organisatie herbergt honderden Javanen. Die doen met elkaar en individueel veel kennis op. Het zou zonde zijn als we de wereld buiten Rabobank niet zouden delen in wat zij leren. En andersom. We brengen wat naar J-Fall, maar we halen ook wat bij J-Fall vandaan. En zo hoort een community ook te werken wat ons betreft: zorgen voor een goede 'kruisbestuiving' om te kunnen groeien en bloeien. Precies zoals onze Rabobank (voorheen Raiffeisen Bank en Boerenleenbank)- coöperatie al 125 jaar succesvol doet.
We stonden dit jaar voor het eerst met onze nieuwe stand, het Rabobank Experience Platform (REP) op J-Fall. Er werd flink met de aanwezige interactieve contentcards gewerkt en dat leverde goede gespreksstof op tussen stand crew en bezoekers. Om het geheel nog iets dynamischer te maken zijn we bezig met een interactief concept wat aanvullend zal zijn op wat er nu al in de stand te zien is maar daarover later meer op J-Fall 2024. Want uiteraard zijn we daar ook weer vertegenwoordigd.

Een speciaal woord van dank aan NL-JUG en alle bezoekers en mede sponsoren want zonder hen zou J-Fall helemaal geen succes zijn: dank voor jullie aanwezigheid, sponsoring en organisatie. En ja, omdat we elkaar ieder jaar weer tegen komen zou je dit gerust een reünie kunnen noemen. Dat woord hebben we tijdens deze en voorgaande edities vaker voorbij horen komen.
Wat doe je dan op een reünie? Elkaar ontmoeten, spreken, ervaringen uitwisselen maar er ontstaan ook persoonlijke contacten en relaties. Niet alleen omdat 'Java' een gemene deler en het hoofd thema is, maar ook gewoon omdat het 'klikt' tussen 2 of meer mensen. We zien vriendengroepen ontstaan, die samen naar J-Fall komen. Voormalig collega's en business partners die elkaar weer ontmoeten. We zien onze collega-sponsoren weer en praten bij tijdens de vele gelegenheden buiten de workshops om. Kortom.... Een reünie van een community. En als klap op de vuurpijl ook nog eens op een geweldige locatie: Pathé in Ede. Geslaagd dus!
En in 2024 hopen we jullie allemaal weer te zien en namens Rabobank en mijn team wens ik alle bezoekers, sprekers, sponsoren en de organisator NL-JUG een fantastisch 2024 toe. Dat het weer een mooi Java jaar mag worden!

Richard Gerestein
Lead Recruitment & Contingent Work IT and Innovation
Rabobank

Rabobank

Hoe meer praten bijdraagt aan betere IT bij Justid

Rijksoverheid

Een 'nee' kan een voorzichtige 'ja' zijn. Met die uitspraak wil Maarten Koller, DevOps engineer bij de Justitiële Informatiedienst (Justid), duidelijk maken dat we meer met elkaar moeten praten. Op J-Fall, hét evenement voor Java-developers, gaf hij een presentatie over betere onderlinge communicatie. Met groot succes: zijn presentatie werd door bezoekers als beste beoordeeld!

{ SNEL EEN LASTIGE BUG FIXEN }

Via de ICT-voorzieningen van Justid worden justitiële gegevens van organisaties uit de jeugd-, strafrecht- en migratieketen beheerd. Justid zorgt dat cruciale informatie beschikbaar is op het juiste moment voor de juiste persoon. Als DevOps engineer is Maarten verantwoordelijk voor de complete ontwikkelcyclus van software. Samen met zijn collega Sharona Jasper loste hij bijvoorbeeld in een paar uur een lastige bug op voor de applicatie MijnSlachtofferzaak, waar zowel Maarten als zij aan werkten. Sharona: 'Voor deze bug zorgde Maarten via zijn contacten voor de juiste gegevens, en kon ik het snel fixen. Het is mooi om te zien dat het zo lukte.'

{ VIA IT-TRAINEESHIP NAAR DEVOPS ENGINEER }

De vrijheid en ruimte om over hokjes spreekt Maarten enorm aan binnen de Rijksoverheid. Na een studie psychologie schoolde hij zich via een IT-traineeship om tot developer. 'Er kan vaak meer dan je denkt. De functies binnen Justid zijn niet vastomlijnd. Mijn opdracht was bijvoorbeeld: maak een pipeline. Toen liep ik er tegenaan dat we altijd vóór dinsdag bij de Change Advisory Board een aanvraag moesten doen, om donderdag toestemming te krijgen om naar productie te mogen. Dat was voor ons niet snel genoeg, dus hebben we samen het proces aangepast. Zij zagen ook de toegevoegde waarde. Nu hebben we afgesproken om de administratie te automatiseren en dat wij daarom elk moment naar productie mogen gaan.'

{ INFORMATIESYSTEMEN MET ELKAAR VERBONDEN }

Maarten: 'Ik kreeg dus de vrijheid om veel verder te gaan dan de oorspronkelijke opdracht. Daarmee hebben we nu veel bereikt, en dat geeft mij weer energie'. Een ander voorbeeld: uit gesprekken met psychologen bij een tbs-kliniek bleek dat zij erg veel tijd kwijt waren aan omslachtige administratie. 'Wij hebben verschillende informatiesystemen met elkaar verbonden: de Forensische Tijdlijn. Dit is een applicatie waarmee documenten en rapportages centraal en chronologisch worden aangeboden. De psychologen zijn van 12 uur naar 4 uur administratie per patiënt gegaan. Ze kunnen zich weer op hun werk focussen, terwijl computers de rest doen. Dat is toch geweldig? En dat allemaal door gewoon te vragen waar ze tegenaan lopen. Kortom: meer praten.'

PERFORMANCE AND DEVELOPER PRODUCTIVITY WITH JAKARTA EE 11

DISCLAIMER: The plan for Jakarta EE 11 is not yet final, so everything in this article is subject to change as the work progresses.

The work on defining the next major update of Jakarta EE has been going on for a while. Ed Burns (*Microsoft*), with the help of Arjan Tijms (*OmniFish*), has taken on the role of release coordinator for Jakarta EE 11. By the time this article is published, a plan should be presented and available publicly.

Back in January 2023, the Jakarta EE Working Group Steering Committee defined the high-level guidelines for Jakarta EE 11. One of the items in these guidelines was to target Java 21. To accommodate this, the runtimes implementing Jakarta EE 11 will be tested on Java 21. The initial plan was to only test on Java 21, but recent discussions have revealed a desire to be able to test on Java 17 as well. A decision regarding this has not been made at the time of writing, so readers are encouraged to follow the communication from the Jakarta EE Working Group on the matter.

The Steering Committee also wishes to establish a regular cadence of 2 years where Jakarta EE is released approximately 6 months after a Java LTS release. For Jakarta EE 11, the Jakarta EE Platform project is a little more cautious and targets a date that is 9 to 12 months after the latest LTS of Java. This means that the target date for Jakarta EE 11 is H1 2024, more specifically June/July 2024.

Some of the specifications will make the necessary changes to be able to support the following language features introduced between Java 11 and Java 21:

Virtual Threads
Records

The list could probably be longer, but these two are the ones that stand out a little and also potentially require API changes. Virtual Threads will only be supported if the implementation is run on Java SE 21 or higher.

With this release, Jakarta EE will also continue the road to becoming more CDI-centric as well as continuing with carefully removing the optional parts to slim down the platform even further.

Ivar Grimstad works at Eclipse Foundation as the Jakarta EE Developer Advocate. Besides advocating the Jakarta EE technologies, Ivar is the PMC Lead for Eclipse Enterprise for Java (EE4J).

All of the above boils down to a theme of Performance and Developer Productivity for Jakarta EE 11.

{ SPECIFICATION UPDATES }

The following specifications will all have updated versions for Jakarta EE 11. Each of them is described in more detail below.

- Jakarta Annotations 3.0
- Jakarta Authentication 3.1
- Jakarta Authorization 3.0
- Jakarta Bean Validation 3.1
- Jakarta Concurrency 3.1
- Jakarta Contexts and Dependency Injection 4.1
- Jakarta Data 1.0
- Jakarta Expression Language 6.0
- Jakarta Faces 4.1
- Jakarta Interceptors 2.2
- Jakarta Pages 4.0
- Jakarta Persistence 3.2
- Jakarta RESTful Web Services 3.2 *
- Jakarta Security 4.0
- Jakarta Servlet 6.1
- Jakarta WebSocket 2.2

* The Jakarta RESTful Web Services project was initially planning a 4.0 release but is now discussing a potentially smaller minor update as a 3.2 release.

Image 1: How Jakarta EE 11 will look if predictions fall through

Jakarta Data is a new specification that will be included in both the Jakarta EE Platform and Jakarta EE Web Profile specifications.

A new concept called *Prospective Specifications* will be introduced in Jakarta EE 11. Prospective Specifications are not part of the Jakarta EE Platform specification but are listed as specifications to pay attention to as they are candidates for later release. The candidates for being listed as Prospective Specifications are:

- Jakarta MVC 3.0
- Jakarta NoSQL 1.0

Note that Jakarta NoSQL will have to release a final version of the specification to be listed as a Prospective Specification.

Image 1 shows how Jakarta EE 11 will look if these predictions fall through.

As you can see in the figure, about half of the specifications of the Jakarta EE Platform are updated in Jakarta EE 11. Let's take a look at the changes planned for each of the specifications.

{ JAKARTA ANNOTATIONS 3.0 }

A plan for Jakarta Annotations has not yet been provided, so it is a bit unclear what the scope of this release will be. Most likely updates related to the removal of the dependency on `ManagedBeans`.

{ JAKARTA AUTHENTICATION 3.1 }

Jakarta Authentication 3.1 will contain minor updates to support the overall goals of Jakarta Security. The changes include clarifications around state and concurrency as well as the removal of all references to the `SecurityManager` [1].

{ JAKARTA AUTHORIZATION 3.0 }

The primary goal of Jakarta Authorization 3.0 is to future-proof the specification as well as make it more suitable for cloud deployments. This will be achieved by adding an API for programmatically registering policy providers, creating a replacement for `java.security.Policy`, and remove all references to the `SecurityManager` [2].

{ JAKARTA VALIDATION 3.1 }

Jakarta Validation 3.1 will add support for Java Records. Records are classes that act as transparent carriers for immutable data and were introduced by JEP 395 in Java 16 [3].

{ JAKARTA CONCURRENCY 3.1 }

The most significant feature of Jakarta Concurrency 3.1 is adding integration to Virtual Threads. Virtual Threads are lightweight threads and were introduced by JEP 444 in Java 21. Other changes in this release of Jakarta Concurrency are taking further steps toward being more CDI-centric [4].

{ JAKARTA CONTEXTS AND DEPENDENCY INJECTION 4.1 }

The main objective of Jakarta Contexts and Dependency Injection 4.1 is to address several minor issues in the Specification, APIs, and TCK [5].

{ JAKARTA DATA 1.0 }

Jakarta Data will provide an API for easier data access by letting the developer split the persistence from the model with the Repository interface, thus increasing the productivity of performing common database operations.. This release will provide the `CrudRepository` feature as well as allow pagination on a repository with `PageableRepository` [6].

{ JAKARTA EXPRESSION LANGUAGE 6.0 }

With Jakarta Expression Language 6.0, the dependency on `java.desktop` module is made optional. It also removes the references to the `SecurityManager`, as well as a small number of enhancements [7].

{ JAKARTA FACES 4.1 }

Jakarta Faces 4.1 removes references to the SecurityManager, continues the alignment with CDI, and provides various small enhancements and clarifications [8].

{ JAKARTA INTERCEPTORS 2.2 }

Jakarta Interceptors 2.2 is a minor update that adds a standard accessor to interceptor bindings [9].

{ JAKARTA PAGES 4.0 }

Jakarta Servlet 5.0 removes functionality that was deprecated in previous releases, as well as updates to align with Jakarta Servlet 6.1 and Jakarta Expression Language 6.0 [10].

{ JAKARTA PERSISTENCE 3.2 }

The Jakarta Persistence 3.2 release comes with a lot of goodies for developers. Java Records will be possible to use as embeddable types, added support for subqueries, programmatic schema management, stateless `EntityManager`, and a wide range of additional functions and operators. It also deprecates support for `java.util.Calendar, java.util.Date, java.sql.Time`, and `java.sql.Timestamp` [11].

{ JAKARTA RESTFUL WEB SERVICES 3.21 }

The goal of Jakarta RESTful Web Services 4.0 was to provide better alignment with Jakarta Contexts and Dependency Injection (CDI), exploration of better integration with Jakarta Concurrency will also be explored, and drop the support for `@Context` injection and related artifacts. The contents of a potential 3.2 release will be significantly smaller, most likely small enhancements and updates as well as explicit deprecations of what to be removed in future releases [12].

{ JAKARTA SECURITY 4.0 }

Jakarta Security 4.0 focuses on evolving the API in various ways. It will also provide APIs for the authorization theme, including interceptors and an abstraction for the permission store. This release will also remove all references to the `SecurityManager` [13].

{ JAKARTA SERVLET 6.1 }

Jakarta Servlet 6.1 will provide small enhancements and adjustments to the specification as well as remove all references to the `SecurityManager` [14].

{ JAKARTA WEBSOCKET 2.2 }

The major topic of Jakarta WebSocket 2.2 is to remove all references to the `SecurityManager`. In addition to this, the release also contains minor updates and adjustments [15].

{ PROSPECTIVE SPECIFICATIONS }

The following three specifications are candidates for being listed as Prospective Specifications in Jakarta EE 11.

{ JAKARTA MVC 3.0 }

This is the third major release of Jakarta MVC. It will ensure alignment with Jakarta RESTful Web Services 4.0 and remove the requirement to support the `Facelets` view engine. Other features in this release are adding an accessor for `FORM` method overwrite field name and switching the CSRF default to `implicit` [16].

{ JAKARTA NOSQL 1.0 }

Jakarta NoSQL is a framework for streamlining the integration of Jakarta EE applications with NoSQL databases. This release defines mapping annotations for Entity, Id, and Column as well as the `DocumentTemplate`, `ColumnTemplate`, and `KeyValueTemplate` specializations [17]. <

{ REFERENCES }

1 https://jakarta.ee/specifications/authentication/3.1/
2 https://jakarta.ee/specifications/authorization/3.0/
3 https://jakarta.ee/specifications/bean-validation/3.1/
4 https://jakarta.ee/specifications/concurrency/3.1/
5 https://jakarta.ee/specifications/cdi/4.1/
6 https://jakarta.ee/specifications/data/1.0/
7 https://jakarta.ee/specifications/expression-language/6.0/
8 https://jakarta.ee/specifications/faces/4.1/
9 https://jakarta.ee/specifications/interceptors/2.2/
10 https://jakarta.ee/specifications/pages/4.0/
11 https://jakarta.ee/specifications/persistence/3.2/
12 https://jakarta.ee/specifications/restful-ws/4.0/
13 https://jakarta.ee/specifications/security/4.0/
14 https://jakarta.ee/specifications/servlet/6.1/
15 https://jakarta.ee/specifications/websocket/2.2/
16 https://jakarta.ee/specifications/mvc/3.0/
17 https://jakarta.ee/specifications/nosql/1.0/

Cavero

Bij Cavero geloven wij dat leuke, gemotiveerde, verschillige en wederkerige collega's het succes van het bedrijf bepalen. Daarom bieden wij onze medewerkers ruime mogelijkheden om zich te ontwikkelen, kennis te delen en plezier te maken.

Bij Cavero krijg je de kans om te werken voor klanten uit diverse sectoren, zoals financiën, gezondheidszorg, onderwijs en overheid. Je krijgt de vrijheid om je eigen opdrachten te kiezen, zodat je kunt werken aan wat je leuk vindt en waar je goed in bent. Als Java-ontwikkelaar bij Cavero krijg je de kans om aan boeiende projecten te werken voor diverse klanten. Deze klanten zijn echt toegewijd aan het gebruik van technologie om een organisatie te bouwen die klaar is voor de toekomst, met altijd de mens in gedachten.

Je kunt ook gebruik maken van de Cavero Academy, waar je de verdieping kunt zoeken in verschillende onderwerpen en je kunt laten certificeren door gecertificeerde trainers.

Cavero heeft een uniek kantoor in Rijswijk, genaamd De Loods. Dit is hét ecosysteem, waarin samenwerken en kennis delen centraal staan en de ideale plek waar werk en ontspanning samen komen.

Kortom, bij Cavero streven we ernaar om futureproof te blijven binnen het Software Development landschap, waarbij Java de basis zal blijven.

cavero

Kabisa

Hier draait alles om software waar je trots op kunt zijn. Als toonaangevende software specialist combineren we diepgaande technische kennis met praktische ervaring in Hightech, Fintech & E-commerce. We zijn experts, creatievelingen en gepassioneerde probleemoplossers!

Onze kracht ligt in het creëren van bedrijfskritische web- en mobiele applicaties. Met een team van ervaren Java professionals, bekwaam in diverse disciplines, streven we naar het hoogst haalbare. We omarmen de diepgewortelde Scrum-methodiek als tweede natuur, waardoor we niet alleen aan de kwaliteitseisen voldoen, maar deze overtreffen.
Als Java ontwikkelaar bij Kabisa ben je onderdeel van het Backend Gilde, waar we kennis met elkaar delen over onderwerpen als Quarkus, Spring Boot 3, Java Virtual Threads en nog veel meer.
"Reveal Your Genius" is onze missie. We excelleren in het omgaan met complexe vraagstukken en leveren stabiele, duurzame software met persoonlijke ontwikkeling en een gezonde werk-privé balans als fundament. Onze gedrevenheid zorgt keer op keer voor verbluffende resultaten waar zowel wij als onze klanten trots op zijn.
Het draait bij ons om oprechte interesse, een gezonde werkwijze en uiterste gedrevenheid. We onthullen niet alleen jouw potentieel, maar dagen je ook uit om deel uit te maken van ons team. Ontdek Kabisa, waar softwareontwikkeling een kunst wordt en genialiteit wordt onthuld.

Kabisa
Software Artisans

ZIJN ER ECHT MENSEN DIE NIET 'GEWOON' MET APPS EN WEBSITES KUNNEN OMGAAN?

Laten we direct antwoord geven op de vraag: "Zijn er echt mensen die niet op de gewone manier met apps en websites kunnen omgaan?" Het antwoord is een duidelijke JA! Er zijn veel mensen met een handicap die permanent, tijdelijk of situationeel is. Deze mensen kunnen moeilijkheden ondervinden met het gebruiken van apps en websites. Sterker nog, als deze apps en websites een digitale twin zijn van de fysieke wereld, kunnen deze mensen ook uitgesloten worden uit de fysieke wereld.

In dit artikel schrijf ik als toegankelijkheidsexpert binnen DDSoft vzw over de principes van inclusie van mensen met diverse handicaps en noden. Daarnaast ga ik het hebben over officiële standaarden voor toegankelijk software ontwerp die uitsluiting van mensen met een handicap tegengaan. Ik zal enkele toegankelijkheid principes uitleggen en je op weg helpen hoe je deze kan implementeren.

Maar, omdat toegankelijk ontwerpen en inclusie van mensen met handicaps een lang traject kan zijn binnen jouw onderneming, bedrijf of organisatie zal ik Quick Fixes tonen. Deze hebben met minimale inspanning een maximum bereik. Laten we starten met de vloek waar wij IT'ers allemaal mee zitten...

{ DE VLOEK VAN DE IT-ER }

Want laat ons eerlijk zijn. We zitten allemaal met de vloek van kennis en interesse in technologie en een netwerk van gelijkgestemden. Onze collega's zijn even gek als wijzelf van nieuwe Flashy Features. We zien doorbraken als Kunstmatige Intelligentie als een reden om productiever te zijn. En geef toe: alle problemen in de fysieke wereld kunnen opgelost worden met IT.

Dennie is Microsoft MVP en heeft ervaring in toegankelijkheid door Microsoft technologieën. Verder is hij voorzittervan DDSoft, een vereniging die de IT-wereld verbindt met mensen die minder IT-vaardig zijn.

We vinden apps uit in tijden van COVID die gevaccineerde mensen laten bewijzen dat ze gevaccineerd zijn. Hiermee kunnen ze binnenkomen in cafés, restaurants en bioscopen. De problematiek van het klimaat en ook de reductie van illegale tewerkstelling - hiermee bedoel ik mensen die werken zonder dit officieel aan te geven en belastingen te betalen - kan het best opgelost worden door volledig cashless te gaan. Betalen doen we wel via een app. Had ik niet gehoord dat Nederland er een woord voor had: pinnen of tikkie? Ik ben Belg, dus ik weet het exacte woord niet.

Geef toe, door onze IT overal toe te passen wordt de wereld eerlijker, gezonder en klimaatneutraler. Wat zijn wij productieve helden! Laat ons eerlijk zijn, wij zijn helden. We kunnen veel bewerkstelligen met IT.

Ik ben ook een IT'er. Maar ik ben de vloek ontkomen. Want wat ik hierboven beschreef is zo mooi voor technologie-gezinde mensen met een IQ van boven het gemiddelde. Dit is helaas ook een vloek op de realiteit. Ik zal het uitleggen met behulp van de metafoor van de ver van mijn bed show.

{ DE VER VAN MIJN BED SHOW }

Heel ver weg van ons kantoor, huis en zelfs ons bed leven er mensen met diverse handicaps. Mensen die niet kunnen lezen. Deze

mensen worden ook wel ongeletterd genoemd. Er leven mensen die niet kunnen rekenen, en die niet weten wat je de dag van vandaag nog kan kopen voor €5. Er zijn mensen die door gezondheidsproblemen niet de financiële mogelijkheden hebben om de nieuwste smartphone te kopen. Laat staan de nieuwste Windows op hun persoonlijke computer te hebben.

Mensen zijn blind, kleurenblind of slechtziend, en als je doof bent of niet goed hoort, mis je zeker die notificatie die toch zo belangrijk is. Daarnaast zijn er ook mensen die niet de luxe hebben om twee armen en handen te gebruiken. En dat laatste is toch wel zo handig bij bijvoorbeeld twee factor verificatie.

Deze mensen zijn op het eerste gezicht ver weg van ons kantoor (en ons bed). Maar ze bestaan weldegelijk, in grotere aantallen dan je denkt. Zo zijn er in Nederland 2 miljoen mensen geregistreerd met een beperking [4]. En wie weet wordt jezelf vroeg of laat ook een van hen. Een van de mensen die vaak uitgesloten wordt in de digitale wereld, en dus ook vaak in de fysieke wereld.

{ LATEN WE INCLUSIE NAAR ONS BED BRENGEN }

Dus laten we deze mensen dichter bij ons bed brengen. Of laat ons tenminste de kennis over deze doelgroepen tot bij ons brengen. Want willen we niet allemaal een maatschappij die productief is voor iedereen? Niet enkel voor mensen als wij (om duidelijk te zijn: een kleine groep in de samenleving).

Dit gaan we doen zoals we gewoon zijn om te doen. Door bij te leren. Ons bij te scholen in de principes van inclusie van mensen met een handicap en toegankelijk ontwerp.

Rond 2000 zijn enkele geleerde mensen binnen de zorgsector waaronder *Prof. Dr. Bob Schalock* en *Van Gennep* samengekomen om de visie te creëren dat iedereen recht heeft op een gelijkwaardig leven in de maatschappij. Dat mensen met een handicap wegplaatsen in gespecialiseerde instellingen ver van het centrum van de stad een slecht idee is in deze tijd.

Met de huidige IT-revolutie is het daarom ook belangrijk om aan deze mensen te denken in ons software ontwerp. Dit doen we door principes van toegankelijkheid toe te passen. Gelukkig kunnen we op de schouders staan van de mensen in diverse consortia en bedrijven die reeds mooie bouwstenen legden voor ons.

{ TOEGANKELIJKHEID }

Deze bouwstenen zijn samengevat onder de paraplu van toegankelijkheid. Een ontwerpprincipe overgewaaid uit de fysieke architectuur. En dit maakt de term toegankelijk, net zo tastbaar en goed te begrijpen.

Als je een gebouw hebt waarbij de entree van het gebouw niet op dezelfde hoogte is als de straat moet er een schuine rand zijn om er met een rolstoel te kunnen inrijden. Zebrapaden met verkeerslichten moeten tegels met een speciale structuur hebben om mensen die blind zijn te duiden op een overgang. Ook moeten de verkeerslichten een geluid en niet alleen op rood en groen springen.

Hierboven noem ik slechts twee voorbeelden van diverse adviezen die gebouwen, locaties en straten letterlijk toegankelijk maken voor mensen met een handicap. Je mag toegankelijkheid vrij letterlijk nemen: door deze richtlijnen en maatregelen hebben deze mensen toegang tot het gebouw, dienst of deel van de stad waar ze anders geen toegang tot hebben.

Voor IT bestaat er gelukkig hetzelfde. Het *World Wide Web Consortium* heeft de **Web Content Accessibility Guidelines** [1] opgesteld (ook wel afgekort door de **WCAG** van het W3C). Ook heeft Microsoft **Inclusive Design** [2] opgesteld. De WCAG heeft richtlijnen waaruit rechtstreekse test scenario's ontwikkeld kunnen worden. Het Inclusive Design geeft je een grondige aanzet tot nadenken met handicap inclusie in gedachten.
Omdat beide richtlijnen al bestaan en al uitgebreid zijn, zal ik ze in dit artikel niet herhalen. Ik zal ingaan op waarom je deze zou gebruiken om uw applicaties te testen door meer uitleg te geven over verschillende handicaps en toegankelijkheid behoeften waarmee de gebruikers van onze software in contact komen.

{ B2B }

Waar denken we aan als we over B2B software spreken? Software voor bedrijven. Waarschijnlijk denk je dan niet automatisch aan mensen met een handicap. Maar het tegendeel is waar. Ook in de bedrijfswereld werken er mensen met extra behoeften. Helaas tonen ze dit vaak niet aan hun verantwoordelijke of collega's. Dit kan leiden tot burn-out, omdat mensen met extra behoeften vaak meer moeite moeten doen om zaken te lezen, verwerken, begrijpen, etc. dan hun andere collega's. Dit is nog te veel een taboe.

Laten we dit concreet maken met een simpel voorbeeld: als je autisme hebt is het begrijpen en juist toepassen van communicatie vaak niet zo vanzelfsprekend als voor mensen zonder autisme. Wil je dan goed je best doen om geen fouten te maken, zal dit meer energie van je vergen dan van collega's die geen extra moeite hebben.

Op kantoren van boekhoudings- en administratie centra werken wel degelijk mensen die slechthorend of neuro divers zijn. Neuro divers betekent dat je brein een andere bedrading heeft. Enkele voorbeelden zijn: ADHD (moeilijkheden met concentratie), dyslexie (moeilijkheden met lezen en schrijven) en autisme (nood aan structuur in combinatie met een andere manier van denken en prikkelverwerking).

Je vindt ook mensen die kleurenblind zijn op de werkvloer. En

deze doelgroep is niet klein. Eén op de twaalf mannen is kleurenblind (inderdaad mannen, bij vrouwen ligt dit percentage veel lager). Ook zijn er op de werkvloer mensen die blind of slechtziend zijn [3].

Door bedrijfssoftware, apps en websites te maken volgens de principes van toegankelijkheid die in de WCAG en het Microsoft Inclusive Design staan zullen deze mensen minder moeilijkheden hebben. Vaak kunnen de mensen die we hierboven noemen, werken met software die niet aangepast is, maar vraagt het veel extra energie van die mensen. Deze mensen zijn uiteraard sneller opgebrand [4].

{ B2B KANTOORSOFTWARE IS HET BEGIN }

Naast B2B software is er natuurlijk ook veel software waar gebruikers en eindgebruikers mee in contact komen. Vaak onder de noemer van apps en websites. Hier maak ik graag een verschil tussen de termen *gebruiker* en *eindgebruiker*.

Een *gebruiker* is de persoon die zelf beslist de applicatie te gebruiken. De applicatie is ontwikkeld om door gebruikers gekozen te worden in hun dagelijkse leven. Enkele voorbeelden zijn YouTube, Spotify, sociale media, games etc..

Een *eindgebruiker* neemt niet zelf de beslissing om software te gebruiken, het gebruik van de software, website of app is opgelegd door een bedrijf of overheid. Een voorbeeld is software voor verplegend personeel om patiëntenadministratie te doen of software om je persoonlijke belastingaangifte in te dienen.

Als je software maakt voor gebruikers en eindgebruikers wordt de doelgroep van mensen met een handicap die jouw ontwikkelde software gebruiken nog een pak groter.

{ MET WELKE BEHOEFTEN REKENING HOUDEN? }

Als we rekening houden met deze gebruikers en eindgebruikers en willen dat alle software toegankelijk is zodat zij niet uitgesloten worden in de digitale en/of fysieke maatschappij volgens de inclusie visie van *Schalock* en *Van Gennep* (hierboven benoemd) moeten we nu ook rekening houden met mensen die een verlamming hebben, een lage geletterdheid of een verstandelijke handicap. Hiermee bedoel ik dus ook de groot ouder met (beginnende) dementie.

Zelf ben ik fan van software waarbij met schuifregelaars bepaalde toegankelijkheidsfuncties in of uitgeschakeld kunnen worden. Software met zo min mogelijk jargon (behalve als de software ontwikkeld is voor een niche, daar kan jargon juist verbinding creëren.) Taalgebruik in applicaties kan er ook voor zorgen dat mensen met psychische problematiek wel of geen angstaanvallen krijgen. Mensen voelen zich in het algemeen veiliger.

Software waarvoor je twee toestellen nodig hebt op hetzelfde moment kan ook mensen uitsluiten. Hier denk ik aan MFA (Multi-Factor-Authenticatie). Hiermee bedoel ik natuurlijk niet dat je geen MFA kan gebruiken in de software. Maar alles begint met bewustzijn over de problematiek die mensen kunnen ondervinden. Een keuze aanbieden om MFA te gebruiken zonder 2 toestellen nodig te hebben (bv: gezichtsherkenning of genoeg tijd aanbieden na een activatiemail) kan al een mooie stap zijn.

{ QUICK WINS }

Graag geef ik **4 Quick Wins** mee die je in al de software kan gebruiken. Begin met deze toe te passen in nieuwe features die je aan de software toevoegt. Op deze manier sluit je stap voor stap minder mensen uit. Deze winsten kunnen zowel in software voor B2B, gebruikers en eindgebruikers een aanzienlijke impact hebben.

1. Zorg voor een standaard lettertype. Verdwaal niet in de jungle van custom fonts. Standaard goed leesbare lettertypes worden overal weergegeven zonder extra inspanning of plug-ins. Zorg voor een lettergrootte die groot genoeg is.
2. Gebruik eenvoudige woordenschat en pas deze aan via het jargon op een maatgerichte manier. De regel is als volgt: hoe meer bedrijfsgericht hoe meer jargon, hoe meer naar gebruikers toe, hoe minder jargon.
3. Voorzie voldoende kleurcontrast over de volledige toepassing. Wit op zwart, zwart op wit zijn vaak de duidelijkste keuzes voor tekst. Menu's en grafieken die een groot contrast hebben tussen de achtergrond en de voorgrond maken het lezen een pak eenvoudiger.
4. Gebruik de standaarden. Responsive Design zorgt vaak dat er geen overflow is, en past zich vaak automatisch aan voor mensen met schermlezers of vergroot-software.

{ TOT SLOT }

Nu je het waarom van toegankelijkheid weet, wens ik je veel succes om de standaarden WCAG en Microsoft Inclusive Design te bestuderen. Ik wens je veel succes en impact op de mensen waarvoor je ontwikkelt. Weet dat er bedrijven als DDSoft vzw bestaan die helpen om standaarden te implementeren.

{ REFERENTIES }

1 WCAG: https://www.w3.org/WAI/standards-guidelines/wcag/
2 Microsoft Inclusive Design: https://inclusive.microsoft.design/
3 Bron: kleurenblindheid.nl https://www.kleurenblindheid.nl/
4 Bron: Rijksoverheid.nl https://www.rijksoverheid.nl/onderwerpen/rechten-van-mensen-met-een-handicap

BYTE SIZE

{ WASI PREVIEW 2 }

WASI Preview 2 has officially launched! This milestone marks a significant step in the evolution of **#WebAssembly**. Now, with stable standard interfaces, developers can build Wasm Components, unlocking a new era of portable and secure applications!
Wassim Chegham (@manekinekko) / X (twitter.com)

Results

These are the results from running all entries into the challenge on eight cores of a Hetzner AX161 dedicated server (32 core AMD EPYC™ 7502P (Zen2), 128 GB RAM).

#	Result (m:s.ms)	Implementation	JDK	Submitter	Notes
1	00:01.878	link	21.0.2-graal	Thomas Wuerthinger, Quan Anh Mai, Alfonso² Peterssen	GraalVM native binary, uses Unsafe
2	00:01.926	link	21.0.2-graal	Artsiom Korzun	GraalVM native binary, uses Unsafe
3	00:01.970	link	21.0.2-graal	Van Phu DO	GraalVM native binary, uses Unsafe

{ THE ONE BILLION ROW CHALLENGE }

Have you heard of the One Billion Row Challenge? It's a friendly competition started by Java Champion Gunnar Morling. The challenge runs until the end of January and aims to find Java code that processes one billion rows in the fastest time. At this time, we just passed under the 2 seconds and there's still one day to go!
https://www.morling.dev/blog/one-billion-row-challenge/

{ KOTLIN 2.0 IS AROUND THE CORNER! }

Jetbrains just released Kotlin 2.0.0-Beta3, making the release of Kotlin 2.0 as close as ever! The new Kotlin K2 compiler will bring major performance improvements, speed up new language and is supporting all the target platforms from the get go. Can't wait for what's coming!
https://kotlinlang.org/docs/whatsnew-eap.html

HOW TO WRITE A BYTE SIZE' BYTE SIZE

Do you have a tip to share, an opinion, or simply want to share something short? That's what the byte size format is for! Byte sizes are short bits of information, in a Twitter format: you have roughly 280 characters, and space enough for a link, or a picture! Send us your byte sizes by @JavaMagazineNL on Twitter or by mailing **info@nljug.org**!

› Java Magazine Editorial Committee

DEVITJOBS.NL

Back End Developer

Salarisindicatie:
65.000 - 95.000 EUR

Locatie:
Amstelveen

Technologieën:
PHP, Symfony, Java, API

EVIDEN

Medior Full-stack developer

Salarisindicatie:
80.000 - 110.000 EUR

Locatie:
Zeist

Technologieën:
Spring Boot, Kubernetes, SQL, Java, CI/CD, Angular, Backend, Cloud, Docker, DevOps, Frontend, Fullstack, Git, Gradle, GitLab

CHILIT

Java Developer

Salarisindicatie:
75.000 - 105.000 EUR

Locatie:
's-Hertogenbosch

Technologieën:
Java, Java EE, Spring, Spring Boot, Docker, Kubernetes, JBoss, AWS, Azure, Cloud, Backend, Frontend, Fullstack

pan company
proven IT experts

Software Engineer Java medior/senior

Salarisindicatie:
45.000 - 75.000 EUR

Locatie:
Amsterdam

Technologieën:
PHP, SQL, Oracle, MongoDB, .NET Framework, Hibernate, Doctrine, Java

furore

Find all these vacancies and more

at Devitjobs.nl

INTRODUCTION TO KEY ENCAPSULATION MECHANISM API

Across the vast expanse of the galactic internet, distant entities can share information securely without ever having to meet in advance. This extraordinary ability is fueled by key exchange algorithms, which are vital to widely used protocols like Transport Layer Security (TLS).

Ana-Maria Mihalceanu (@ammbra1508) is a Java Champion, Developer Advocate at Red Hat. She also enjoys curating content for conferences as a program committee member.

{ INTRODUCTION }

Imagine this Star Wars-inspired scenario: a Jedi master wishes to send its Padawan a secret message, using the BD-1 unit. The Jedi can generate a key, hand that to the BD-1 unit, and send the robot to the Padawan. Later, the Jedi can shuffle the message with the key, and send the result to the Padawan, as he/she can unscramble the message since they are already in possession of the same key.

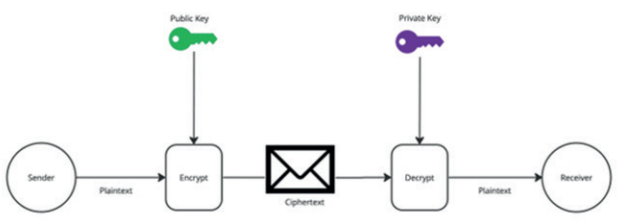
But the Empire's watchful eyes are everywhere, making this exchange very dangerous: should the BD-1 unit be captured, the Empire might intercept the key, thus jeopardizing future communication. In this predicament, the Jedi can engage in a key exchange algorithm with the Padawan, creating a secret value known only to them and unreadable by the prying eyes of the Empire.

The widely-used key exchange algorithms are based on complex mathematical concepts, like integer factorization or the discrete logarithm problem. However, quantum computers possess the capability to solve these problems, thus breaking the confidentiality of communication. In the light of the potential quantum attacks, Java developers and third-party security providers need a new encryption technique for securing symmetric keys using public key cryptography.

At present, ML-KEM [1] is believed to be secure even against attackers who possess a quantum computer, and *Key Encapsulation Mechanism* (KEM) is one of the two main areas of NIST's *Post-Quantum Cryptography Standardization Project* (the other one being digital signature). So JDK 21 brings the *new Key Encapsulation Mechanism API* (KEM API) [2], which is valuable for both classical and post-quantum cryptography!

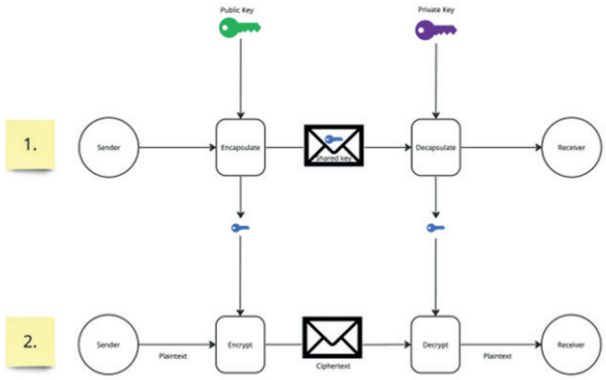
{ COMPONENTS }

When two individuals communicate in an encrypted form, the sender encrypts a message and only the receiver can decrypt it. With *a Public Key Encryption* (PKE) scheme, the sender encrypts the message using the receiver's public key and the receiver decrypts it with their private key.



Afbeelding 1: Public Key Encryption scheme

But Public Key Encryption schemes are usually less efficient than symmetric encryption schemes, and so there is interest to switch to symmetric encryption. A Key Encapsulation Mechanism (KEM) is also a scheme with public and private keys, where the sender uses the public key (of the receiver) to create a ciphertext, an encapsulation, containing a randomly chosen symmetric key – and the receiver decapsulates the encapsulation with its private key. Unlike traditional cryptography methods, a key encapsulation mechanism does not require *padding* [3] because it uses properties of the public key to derive a related symmetric key. So you can use a KEM to first transmit the shared/symmetric key, and later use it to efficiently encrypt data using a symmetric algorithm. Figure 2 exemplifies how Key Encapsulation Mechanism works.



Afbeelding 2:How Key Encapsulation Mechanism works

{ NOTE }

In practice, the sender uses a trusted certificate issued by a CA to verify that the public key indeed belongs to the receiver. This type of verification is not in scope of this article.

The Key Encapsulation Mechanism API makes this possible through its components:
A key pair generation function that returns a public key and a private key `(a keypair).KeyGen() -> public_key, private_key` This function is already covered by *the KeyPairGenerator API* [4].
A key encapsulation function, called by the sender, which takes a public key as input and outputs a shared secret value and an encapsulation (a ciphertext) of this secret value. `Encapsulate(public_key) -> ciphertext, shared_secret`
A key decapsulation function, that takes as input the encapsulation and the private key, and outputs the shared secret value. `Decapsulate(private_key, ciphertext) -> shared_secret`

Let's have a look at the simplified KEM example in Listing 1. When using a KEM, the receiver would run generateKeyPair and share the public key. The sender takes this public key, runs encapsulate(), keeps the generated secret to itself, and sends the encapsulation to the receiver. The receiver runs decapsulate() over this encapsulation and obtains the same shared secret that the sender holds. Even if an attacker would see the encapsulation, they wouldn't be able to infer the secret value.

Yet a single KEM algorithm can have multiple configurations, so let's check those out.

{ CONFIGURATIONS }

A KEM configuration should map to a specific algorithm that creates a fixed size shared secret and a fixed size key encapsulation message. Each configuration of a KEM algorithm can:

L1

```
// receiver side
KeyPairGenerator kpg = KeyPairGenerator.
getInstance("X25519");
KeyPair kp = kpg.generateKeyPair();
// sharePublicKey(kp.getPublic());

// sender side
// PublicKey publicKey = retreivePublicKey();
KEM kem1 = KEM.getInstance("DHKEM");
KEM.Encapsulator sender = kem1.
newEncapsulator(publicKey);
KEM.Encapsulated encapsulated = sender.
encapsulate();
SecretKey sharedSecret1 = encapsulated.key();
//sendBytes(encapsulated.encapsulation());

// receiver side
// byte[] encapsulation = receiveBytes();
KEM kem2 = KEM.getInstance("DHKEM");
KEM.Decapsulator receiver = kem2.
newDecapsulator(kp.getPrivate());
SecretKey sharedSecret2 = receiver.
decapsulate(encapsulation);

assert Arrays.equals(sharedSecret1.getEncoded(),
sharedSecret2.getEncoded());
```

- accept different types of public or private keys
- use different methods to derive the shared secrets
- emit different key encapsulation messages.

You can determine the configuration mapped to a specific algorithm by using:

The algorithm name passed to a `getInstance` method, like in the example from the previous section where the algorithm is `DHKEM`.
The type of the key passed to a `newEncapsulator` or `newDecapsulator` method, like in the example from the previous section where the key is `of X25519`.
The optional `AlgorithmParameterSpec` object passed to a `newEncapsulator` or `newDecapsulator` method. The DHKEM algorithm does not define any parameter. However, a KEM algorithm can define an `AlgorithmParameterSpec` subclass to provide additional information to the `newEncapsulator` method.

For example, in the fitness test that implements RSA-KEM as described in RFC 5990 [5], the implementation supports different configurations based on different RSA key sizes and different key derivation function (KDF) settings. These different key derivation function settings are conveyed via an `RSAKEMParameterSpec` array.

```
RSAKEMParameterSpec[] kspecs = new
RSAKEMParameterSpec[]{
    RSAKEMParameterSpec.kdf1(«SHA-256», «AES_128/
KW/NoPadding”),
    RSAKEMParameterSpec.kdf1(«SHA-512», «AES_256/
KW/NoPadding”),
    RSAKEMParameterSpec.kdf2(«SHA-256», «AES_128/
KW/NoPadding”),
    RSAKEMParameterSpec.kdf2(«SHA-512», «AES_256/
KW/NoPadding”),
    RSAKEMParameterSpec.kdf3(«SHA-256», new
byte[10], “AES_128/KW/NoPadding”),
    RSAKEMParameterSpec.kdf3(«SHA-256», new
byte[0], “AES_128/KW/NoPadding”),
    RSAKEMParameterSpec.kdf3(«SHA-512», new
byte[0], “AES_128/KW/NoPadding”),
};
```

Next, the sender configures the encapsulator with each of the `RSAKEMParameterSpec` object.

```
KEM kem1 = KEM.getInstance(“RSA-KEM”);
KEM.Encapsulator e = kem1.
newEncapsulator(publicKey, kspec, null);
KEM.Encapsulated enc = e.encapsulate();
// sendBytes(enc.encapsulation());
// sendBytes(enc.params());
```

The receiver can recover the same `RSAKEMParameterSpec` object from the encoding with an `AlgorithmParameters` implementation used to configure the decapsulator.

```
// byte[] encapsulation = receiveBytes();
// byte[] params = receiveBytes();
AlgorithmParameters ap = AlgorithmParameters.
getInstance(“RSA-KEM”);
ap.init(params);
KEM kem2 = KEM.getInstance(“RSA-KEM”);
KEM.Decapsulator d = kem2.
newDecapsulator(privateKey,
ap.getParameterSpec(AlgorithmParameterSpec.class));
SecretKey k = d.decapsulate(encapsulation);
```

As a KEM algorithm may support a family of configurations, let's check out how security service providers make that possible via the KEM service provider interface.

{ THE KEM SERVICE PROVIDER INTERFACE (SPI) }

A security provider implements the KEMSpi interface to provide an implementation of a Key Encapsulation Mechanism (KEM) algorithm.

```
package javax.crypto;

public interface KEMSpi {

    interface EncapsulatorSpi {
        int engineSecretSize();
        int engineEncapsulationSize();
        KEM.Encapsulated engineEncapsulate(int
from, int to, String algorithm);
    }

    interface DecapsulatorSpi {
        int engineSecretSize();
        int engineEncapsulationSize();
        SecretKey engineDecapsulate(byte[]
encapsulation, int from, int to,
                                    String
algorithm)
                throws DecapsulateException;
    }

   EncapsulatorSpi engineNewEncapsulator(PublicKey
pk, AlgorithmParameterSpec spec,

SecureRandom sr)
            throws
InvalidAlgorithmParameterException,
InvalidKeyException;
    DecapsulatorSpi
engineNewDecapsulator(PrivateKey sk,
AlgorithmParameterSpec spec)
            throws
InvalidAlgorithmParameterException,
InvalidKeyException;

}
```

For an implementation to be compliant with the KEM API, it must implement the `EncapsulatorSpi` and `DecapsulatorSpi` interfaces, and return objects of these types from the `engineNewEncapsulator` and `engineNewDecapsulator` methods.

Additionally, calls to `secretSize()`, `encapsulationSize()`, `encapsulate()` and `decapsulate()` methods of `Encapsulator` and `Decapsulator` are delegated to the `engineSecretSize`, `engineEncapsulationSize`, `engineEncapsulate`, and `engineDecapsulate` methods in the `EncapsulatorSpi` and `DecapsulatorSpi` implementations.

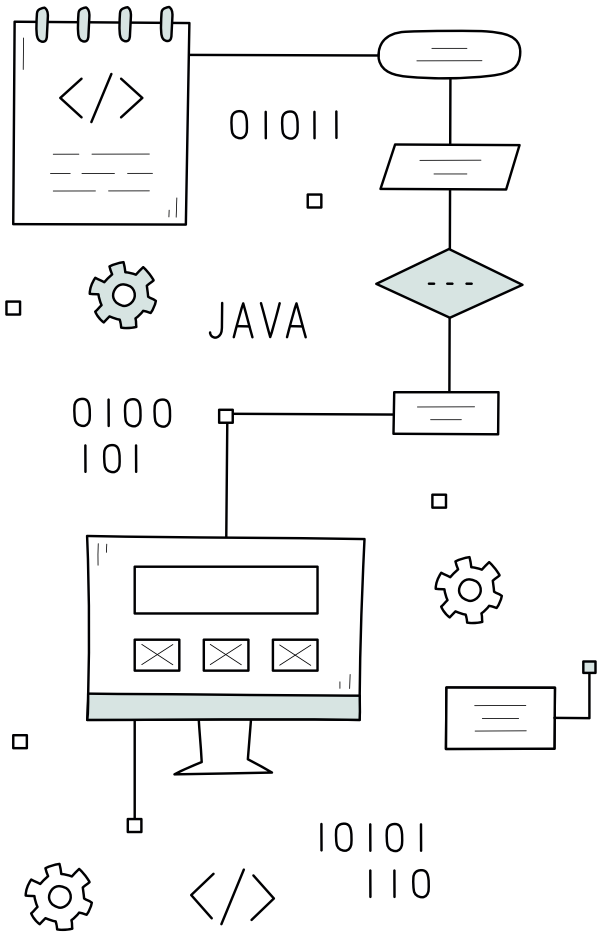
For example:

```
public static class KEMImpl implements KEMSpi {

@Override
public KEMSpi.EncapsulatorSpi engineNewEncapsulator(
PublicKey pk, AlgorithmParameterSpec spec,
SecureRandom secureRandom)
throws InvalidAlgorithmParameterException,
InvalidKeyException {
        // check key type...
return Handler.newEncapsulator(spec, rpk,
secureRandom);
}

@Override
public KEMSpi.DecapsulatorSpi engineNewDecapsulator(
PrivateKey sk, AlgorithmParameterSpec spec)
throws InvalidAlgorithmParameterException,
InvalidKeyException {
        // check key type...
return Handler.newDecapsulator(spec, rsk);
}

static class Handler implements KEMSpi.
EncapsulatorSpi, KEMSpi.DecapsulatorSpi {
        static Handler
newEncapsulator(AlgorithmParameterSpec spec,
RSAPublicKey rpk, SecureRandom sr)
                throws
InvalidAlgorithmParameterException {
            //...
        }
        static Handler
newDecapsulator(AlgorithmParameterSpec spec,
RSAPrivateCrtKey rsk)
                throws
InvalidAlgorithmParameterException {
            //...
        }
        public KEM.Encapsulated engineEncapsulate(int
from, int to, String algorithm) {
            //...
        }
        public SecretKey engineDecapsulate(byte[]
encapsulation,
                int from, int to, String algorithm)
throws DecapsulateException {
            //...
        }
        //...
}
}
```



Yet...when using a KEM algorithm, your choice depends not only upon the name of the algorithm, but also on the available security providers. So let's talk a bit about how that happens.

{ DELAYED PROVIDER SELECTION }

The `KEM.getInstance()` methods, such as `KEM.getInstance("DHKEM")`, return a provider-neutral algorithm handle. The JDK delays the selection of the provider until the relevant `newEncapsulator` or `newDecapsulator` method is called. `newEncapsulator` and `newDecapsulator` methods accept a `Key` object and can determine at that point which provider can accept the specified Key object. This process ensures that the selected provider can use the specified Key object.

```
KEM kem1 = KEM.getInstance("DHKEM");
KEM.Encapsulator sender = kem1.newEncapsulator(kp.
getPublic());
System.out.println(sender.providerName()); //prints
SunJCE
```

If you would like to discover which provider is selected, use the `providerName()` methods of the `Encapsulator` and `Decapsulator` instances. And speaking of security providers, not all of them share the same behavior when extracting the shared secret in a byte array.

{ SHARED SECRETS ARE NOT ALWAYS EXTRACTABLE }

A KEM algorithm returns shared secrets in a byte array. Yet, you should consider that a Java security provider might employ a native-code implementation, thus leading to cases when the shared secret is not extractable.

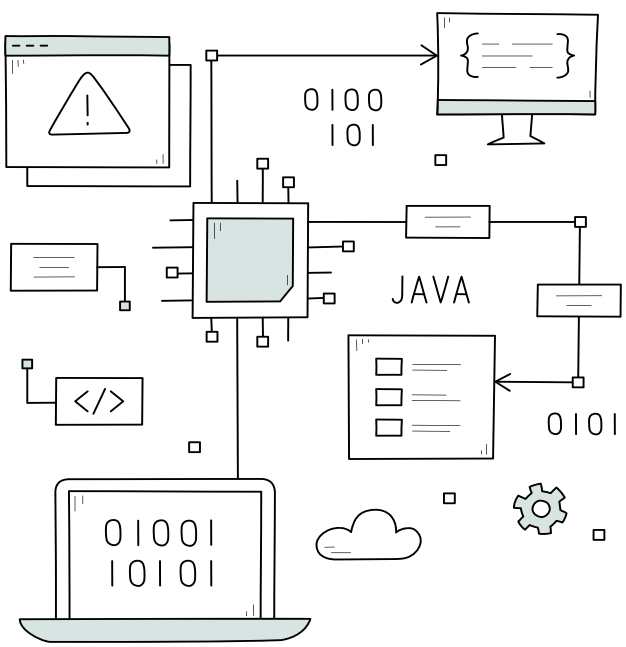
That's the reason why the `encapsulate` and `decapsulate` methods always return the shared secret in a `SecretKey` object. In cases where the key is extractable, the format of the key must be `RAW` and its `getEncoded()` method must return either:

the full shared secret, or
the slice of the shared secret specified by the from and to parameters of an extended encapsulate or decapsulate method

```
KEM kem = KEM.getInstance("DHKEM");
KEM.Encapsulator sender = kem.newEncapsulator(kp.
getPublic());

// Get the full shared secret as a key
KEM.Encapsulated encapsulated = sender.
encapsulate();
SecretKey key1 = encapsulated.key();

// Get a slice of the shared secret's first 16
bytes as an AES key
KEM.Encapsulated slice = sender.encapsulate(0, 16,
"AES");
SecretKey key2 = slice.key();
```



If the keys above are not extractable, the `getFormat()` and `getEncoded()` methods should return null. But don't worry about it, as the security provider should have provided Cipher, Mac or KDF implementations that make use of this secret key.

While on the subject of extracting the shared secret, let's see how you can extract the key encapsulation message from concatenated data.

{ RETRIEVE THE SIZE OF THE KEY ENCAPSULATION MESSAGE }

Sometimes higher-level protocols, like Hybrid TLS Key Exchange, can directly concatenate key encapsulation messages with other data, without including any length information. In these scenarios, it is assumed that the length of the key encapsulation message remains fixed and known once the KEM configuration is set.
When an application needs to extract the key encapsulation message from the concatenated data, you can use the `encapsulationSize()` methods. With these methods your application can retrieve the exact size of the key encapsulation message, enabling a precise extraction and utilization of the encapsulated information.

```
byte[] blob = receiveBytes();
KEM kem = KEM.getInstance("DHKEM");
KEM.Decapsulator receiver = kem.
newDecapsulator(privateKey);
int encapsulationSize = receiver.
encapsulationSize();
byte[] encapsulation = Arrays.copyOfRange(blob, 0,
encapsulationSize);
SecretKey key = receiver.decapsulate(encapsulation);
```

{ FINAL WORDS }

I hope this article clarified the importance of Key Encapsulation Mechanism API, as it becomes the key to helping ensure secure communication in this quantum-driven era. May the Encryption force be with you!
I am grateful to *Sean Mullan* and *Wei-Jun Wang* for their expertise that contributed to this article.

{ REFERENCES }

1 https://csrc.nist.gov/pubs/fips/203/ipd
2 https://openjdk.org/jeps/452
3 https://en.wikipedia.org/wiki/Padding_(cryptography)
4 https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/security/KeyPairGenerator.html
5 https://github.com/openjdk/jdk21/blob/master/test/jdk/javax/crypto/KEM/RSA_KEM.java

How to design Reliable Microservice Chains using the principles of Systems Thinking

Mehmet Ali Kipcak, Lead Software Architect at ING

Designing reliable microservice architectures necessitates adopting a systems thinking mindset. Microservices comprise interconnected components that function interdependently as a unified whole system. Engineers should consider both the individual parts in isolation as well as their complex interactions synthesized together. Because changes impacting one area can propagate effects across the entire system architecture in unpredictable and nonlinear ways due to emergent behaviors from these dynamic relationships.

To develop an accurate mental model of such a complex system, engineers should begin with an analytical decomposition. Break down each microservice to understand key attributes and behaviors in isolation. This involves examining factors like core functionality, interfaces, dependencies, data flows, error handling mechanisms, non-functional properties and more. Engineers should also conduct individual testing of components to validate base functionalities and expectations.

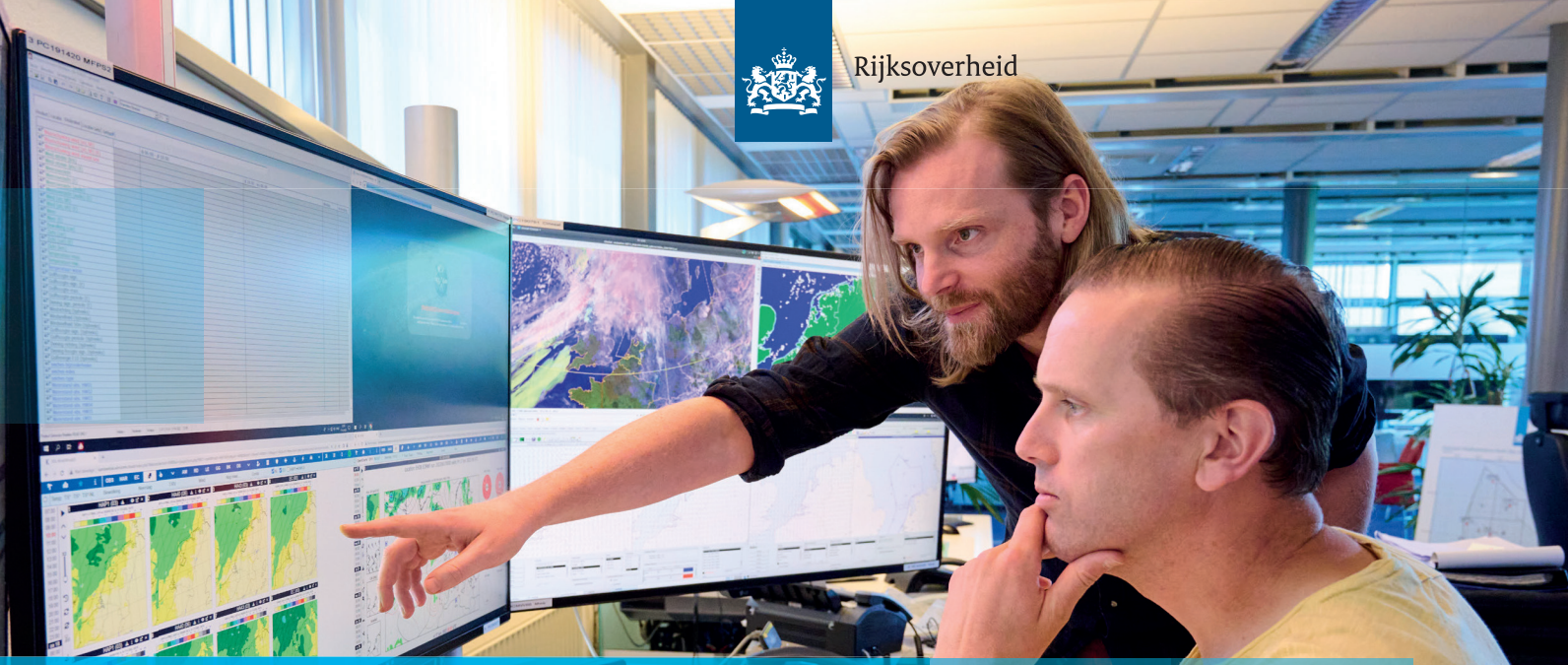
However, systems thinking demands to also engage in synthetic reasoning to integrate analytical work. They must mentally model and simulate dynamic interactions between microservices under a variety of conditions. This involves assessing relationships like interfaces, data and control flows, cascading impacts of failures or changes, environmental couplings, emergence of unintended behaviors and more. Develop methods to simulate potential system-wide behaviors and scenarios that may arise from complex, nonlinear interactions between components.

Measuring and tracking complexity over time is pivotal. However, it is commonly overlooked for designing reliability. Traditional source code complexity metrics can offer valuable insights when applied from a systems perspective. Unchecked, complexity tends to steadily accumulate until surpassing cognitive limits, hindering holistic comprehension. Average issue resolution durations expand as root causes become obscure without a unified architectural model. Customers may experience outages resulting from engineers struggling to synthesize analytical findings. Organizations must develop sophisticated models of dynamic interactions to prevent complexity-induced problems. Assessing relationships between services aids understanding diverse, emergent system behaviors and scenarios. Interactions form the basis for probabilistic risk simulations evaluating impacts of variations, changes or failures. Rather than subjective risk labeling, such simulations guide strategic complexity management.

Formalizing governance practices for service call chains brings structure to API management. Centralized visual mappings of all components and interconnections help establish a common taxonomy. Guidelines and oversight functions codify practices while continuous monitoring feeds learnings back into dynamic models. Structured call chain governance sustains comprehension ensuring adaptability.

Cultivating comprehensive team mental models yields highest returns through improved collaboration. Shared understandings streamline diagnosing anomalies, hypothesizing causes and resolving incidents. Mental models facilitate rapidly comprehending complex journeys, recognizing patterns and synthesizing analyses holistically.

In conclusion, applying systems thinking principles is crucial. Measuring and managing complexity and interactions, probabilistic risk modeling, governance frameworks, and precise team mental modeling offer perspectives needed to design reliable microservices architectures. These architectures must be able to evolve over the long term. Formalizing such practices treats architectures holistically as interconnected, dynamic business systems.



Rijksoverheid

Werk jij mee aan IT voor 17 miljoen Nederlanders?

Als IT-professional bij de Rijksoverheid werk je mee aan een zo goed mogelijk functionerende digitale samenleving voor iedereen in Nederland.
Je houdt je als developer bezig met de ontwikkeling van applicaties om het weer te voorspellen. Innoveert als data-expert met big data, slimme algoritmes en toolsvoor verkeersinformatie.
Of draagt als securityspecialist bij aan de beschermingvan systemen die Nederland draaiende houden.

Wat wij jou bieden? Volop mogelijkheden om je breed te ontwikkelen in een werkomgeving waar je dagelijks werkt aan complexe IT-vraagstukken.
Denk aan RPA, front-end en back-end technologieën, programmeren met bestaande of zelfontwikkelde tools in Java en het automatiseren van testanalyses, uitvoeren van risicoanalyses of inzetten van data-tools.
Benieuwd wat jij bij de Rijksoverheid kunt doen?

Werken bij de Rijksoverheid is werken voor Nederland.

Scan de QR-code en check onze vacatures.

www.werkenvoornederland.nl

COLUMN

Uniek als een sneeuwvlokje

Maja Reißner *is*
Docent cyber security.

Heb je er wel eens op gelet hoe verschillend sneeuwvlokjes er eigenlijk uitzien? Misschien stond je wel eens buiten naar een paar sneeuwvlokjes te kijken en viel je daadwerkelijk op dat ze niet allemaal hetzelfde zijn. Maar je hebt vast niet ook echt gevalideerd of ze dan wel allemaal verschillend zijn. Sneeuwvlokjes zijn uniek. Of in ieder geval zegt men dat. Daarbij voelt het ene vlokje toch verdacht gelijk aan als het buurvlokje en zien ze er soms verdacht hetzelfde uit. Waar we het dan eigenlijk over hebben is dus niet het zichtbare effect van de vlok maar de moleculaire opbouw van de sneeuwvlok en dan hebben we het over kristalstructuren, een paar mogelijke oriëntaties maar vooral over heeeeeel veel watermoleculen en uiteindelijk over exponentiatie en statistiek.

Een sneeuwvlokje bestaat uit zo'n 10^18 watermoleculen en hoe het kristal vormt hangt sterk van de temperatuur en de luchtvochtigheid af. In een lab konden onder gecontroleerde omstandigheden dus wel bijna identieke sneeuwvlokjes gemaakt worden maar de kans is erg klein dat je identieke vlokjes buiten in de natuur tegenkomt. Ja, ok, het is mogelijk en de kans bestaat. Toch blijven we een sneeuwvlokje uniek noemen en is dit geen fout maar een bewuste keuze om de kans te negeren.

In de IT gebruiken we ook het concept van unieke waardes. Specifiek gebruik je wellicht de UUID class uit de Java utils. Deze universeel unieke id bestaat uit 128 bits. Deze 128 bits kun je bijvoorbeeld met de `randomUUID()` methode genereren. Onder water wordt daar een pseudo random number generator gebruikt wat betekent dat je echt willekeurige waardes krijgt. De kans dat twee UUID's dan hetzelfde zijn, is dan 2^128 (vanwege de 128 bits) wat ongeveer 10^38 is.

Het is dus vele malen waarschijnlijker dat je dezelfde twee sneeuwvlokjes tegenkomt dan dat je dezelfde twee UUID's tegenkomt. Mits de cryptografische algoritme voor het genereren van de willekeurige waarde ook echt klopt, natuurlijk. En gelukkig maar! Twee dezelfde sneeuwvlokjes hebben waarschijnlijk veel minder fatale gevolgen dan een unieke id in je code die toch niet uniek blijkt te zijn.

Het maakt trouwens een groot verschil of we voor een specifiek sneeuwvlokje (UUID) kijken of er een tweede bestaat die hetzelfde is of dat we in de hele groep van alle sneeuwvlokjes (heel veel random gekozen UUID's) kijken of er 2 hetzelfde zijn. In andere woorden: Er is een groot verschil in kans of 1 specifieke waarde uniek is of dat alle waardes uniek zijn. Vind je dit een interessante gedachte? Zoek dan maar even het *birthday paradox* op :).

Bij deze wens ik je nog een fijne winter en alvast gefeliciteerd met je verjaardag dit jaar!

VAN HET BESTUUR

Alweer de eerste editie van 2024 van het Java Magazine. Terwijl we vol enthousiasme dit nieuwe jaar instappen, is het ook een moment om terug te blikken op een memorabel 2023, met name de 20ste editie van J-Fall - een mijlpaal die we niet snel zullen vergeten!

{ J-FALL: 20TH ANNIVERSARY EDITION }

De 20ste J-Fall was weer het bekende jaarlijkse feest, met een schat aan inhoud van goede talks, gave stands en leuke gesprekken. Ook waren we getuige van een historisch moment met de introductie van *lazy seats* in de bioscoopzalen, waardoor we het weer zeer geslaagde inhoudelijke programma op nog comfortabelere manier konden ervaren. Daarnaast waren er meer historische momenten met *Sharat Chander*, die vanuit *Oracle* de NLJUG als hele community eerde, het gezamenlijk bedanken van de volledige organisatie achter de NLJUG door de jaren heen (met een geweldige trip down memory lane via foto's over de afgelopen twintig jaar), en de benoeming van de kersverse Java Champion *Elias Nogueira*. Het was weer een geweldige sfeer, van begin tot eind, en als altijd een feest om iedereen binnen de community weer te zien, spreken en de dag gezamenlijk te ervaren.

{ INNOVATION AWARD 2023: AND THE WINNER IS... }

De strijd om de NLJUG Innovation Award was dit jaar intenser dan ooit. Met genomineerden *ASML*, *EDSN* en *TriOpSys* was de keuze niet eenvoudig voor het publiek, en de video pitches spraken goed tot de verbeelding. Elk van deze bedrijven toonde aan hoe Java-technologie kan worden ingezet om baanbrekende oplossingen te bieden in diverse sectoren. Maar... er kan er maar één de winnaar zijn: TriOpSys heeft met hun *Decision Support Tool* voor de Luchtverkeersleiding Nederland - een project dat niet alleen technische innovatie demonstreert, maar ook een significante impact heeft op de veiligheid en efficiëntie van onze luchtruimte - de harten van de developers in het publiek weten te winnen en de prijs in de wacht gesleept. Proficiat TriOpSys, de Innovation Award Winner 2023!

{ MASTERS OF JAVA: VAKMANSCHAP MET BLINDDOEK }

Het officieuze NK Java programmeren, dat uitgevoerd wordt zonder hulp van een IDE (vandaar de blinddoek), was weer een groots evenement, met maar liefst 65 strijdende developers! Ook hier kan er maar een (team) de winnaar zijn, en dit jaar zijn dat (weer eens) *Thomas Withaar* en *Maarten van der Zwaart* van *Team OVSoftware*! Gefeliciteerd met het winnen van de Masters of Java 2023, heren!

Het was in onze ogen een prachtig 2023 voor de Java community met een bijzonder jubileumjaar voor J-Fall. Laten we er gezamenlijk ook in 2024 weer een innovatief en knallend jaar van maken. Wij hopen jullie allemaal te zien bij onze aankomende evenementen!

Namens het hele bestuur wens ik iedereen graag een prachtig 2024 toe. Geniet van deze eerste Java Magazine editie van het jaar!

Met vriendelijke groet,
Bas W. Knopper

Save the dates!

APRIL 17

FUTURE TECH 2024

THE IT CONFERENCE FOR MICROSOFT TECHNOLOGIES

WWW.FUTURETECH.NL

MAY 22

TEQNATION

DEVELOPER PLATFORM

WWW.CONFERENCE.TEQNATION.COM

JUNE 13

J-SPRING 2024

WWW.JSPRING.NL

OCTOBER 10

FRONT MANIA

WWW.FRONTMANIA.COM

NOVEMBER 7 of 14

J-FALL 2024

WWW.JFALL.NL

ing.nl/careers/tech

do your thing