Java 9 bis 21 – Neue Features

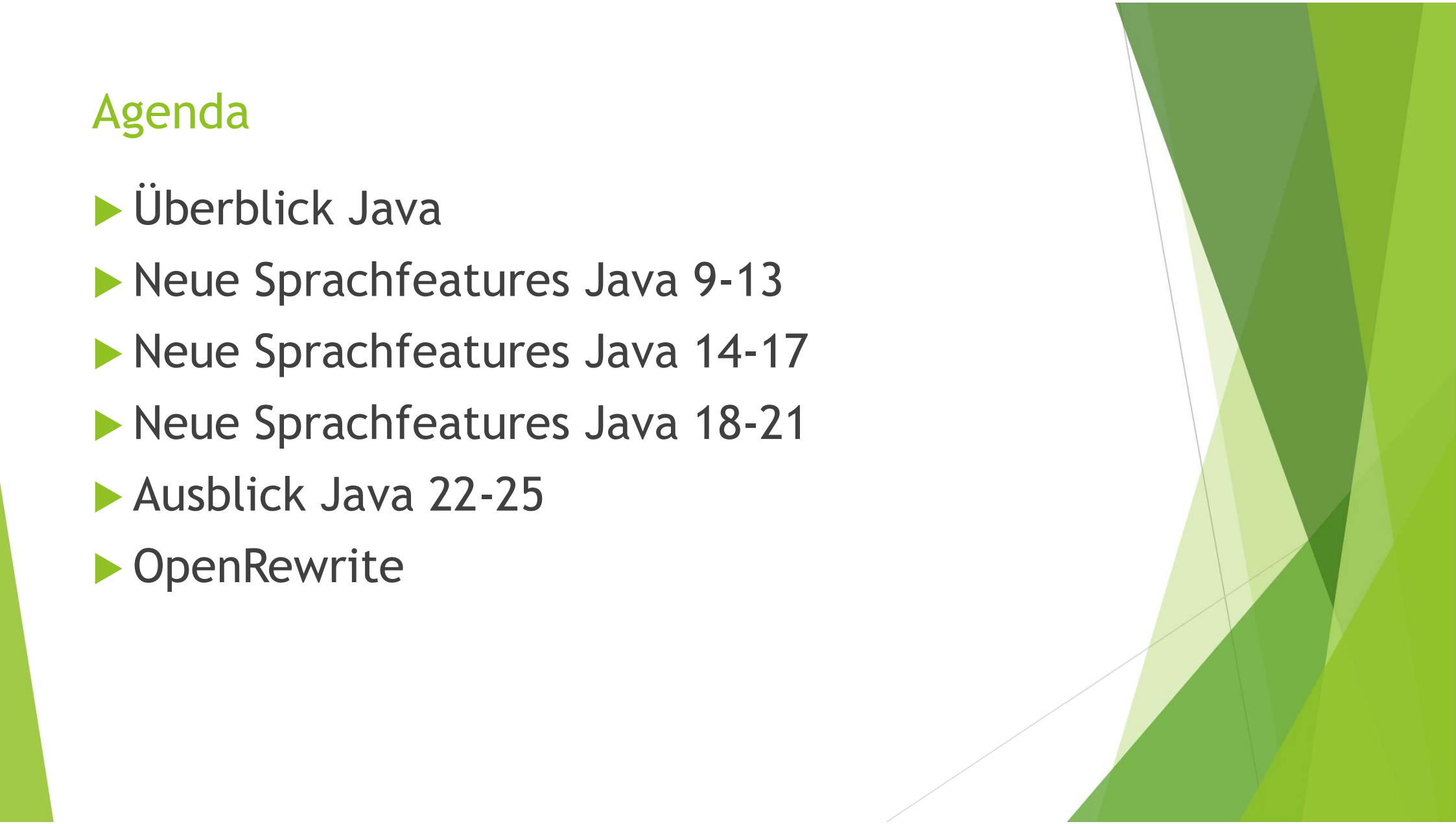
Vorstellung

- Trainer: Michael Zöller
- Aus Hamburg
- Software Architect, developer and trainer
- Xing: https://www.xing.com/profile/Michael_Zoeller3
- LinkedIn: https://www.linkedin.com/in/michael-z%C3%B6ller-579041256
- freelancermap : https://www.freelancermap.de/profil/michael-zoeller
- Mail: michael2.zoeller@gmail.com

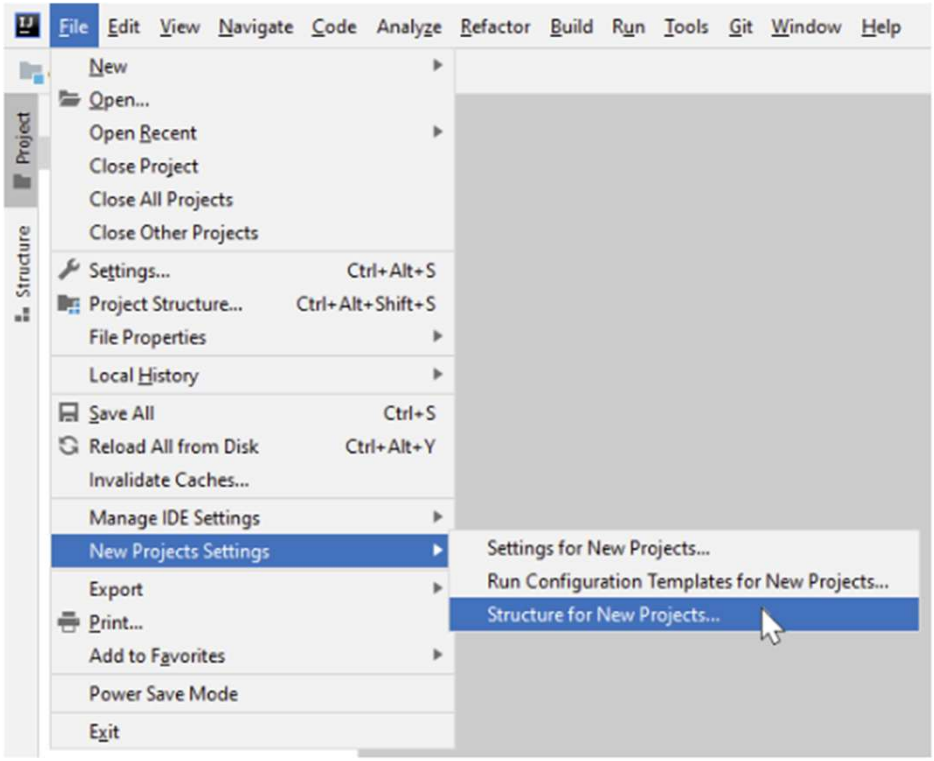
Du oder Sie?

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite



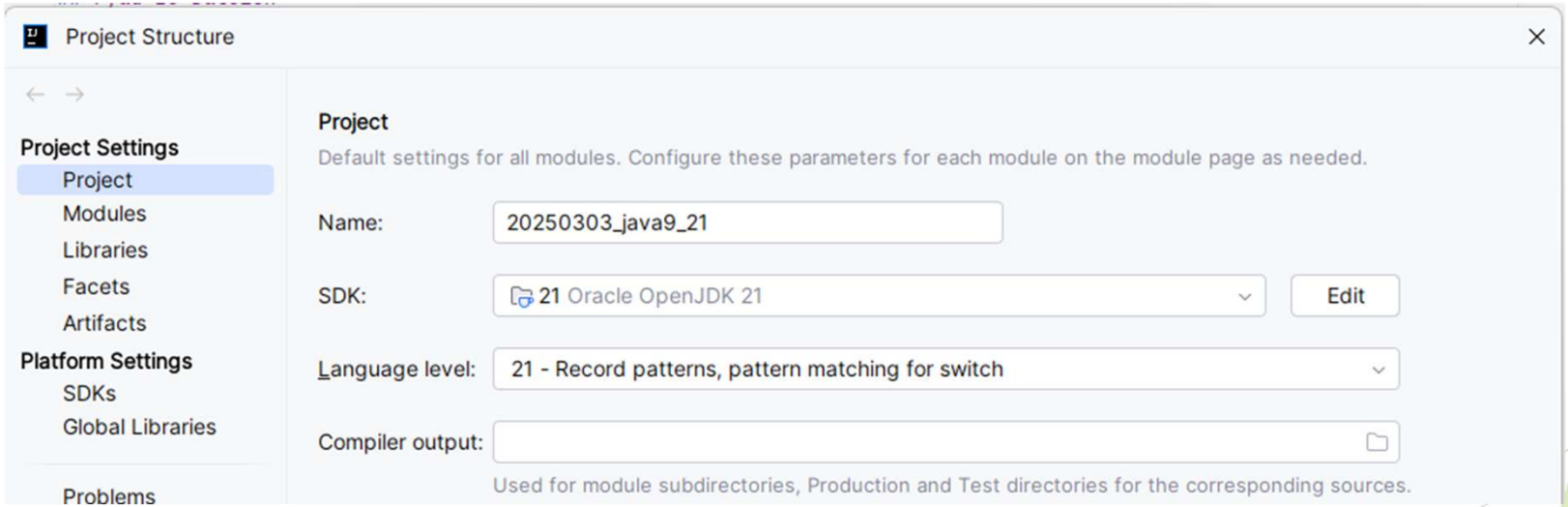
IntelliJ – default sdk für Projekte auswählen



- Dort jdk 21 einstellen

Repository

- https://github.com/MichaelZett/20250303_java_9_21



Agenda

- **Überblick Java**
  - **Java historisch**
  - Der Java Releaseprozess
  - Aktuelle Projekte
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite

Java Historie – Write Once, run everywhere

- Erste stabile Java Version (1.0.2) erschien im Januar 1996
- Java setzte sich von Anfang an als **die** neue Sprache durch:
  - **Plattformunabhängigkeit**: Java wurde mit dem Prinzip "Write Once, Run Anywhere" (WORA) entworfen, was bedeutet, dass Java-Programme, die auf einer Plattform kompiliert wurden, auf jeder anderen Plattform ausgeführt werden können, die über eine Java Virtual Machine (JVM) verfügt. Diese Eigenschaft machte Java besonders attraktiv für das Internet und Unternehmensanwendungen.
  - **Objektorientierte Programmierung (OOP)**: Java ist eine größtenteils objektorientierte Sprache, was in den 1990er Jahren der jüngste Hype war.
  - **Robustheit und Sicherheit**: Z.B. Verbot von direktem Zugriff auf Speicheradressen, um gängige Programmierfehler und Sicherheitsrisiken zu vermeiden (keine Pointer-Arithmetik).
  - **Automatische Speicherverwaltung**: Java führt eine automatische Speicherverwaltung durch (Garbage Collection), die hilft, Speicherlecks und andere Speicherprobleme zu vermeiden.

Java Historie – Weitere Vorzüge Javas

- **Reiche Standardbibliotheken**: Java bot sehr früh eine umfangreiche Sammlung von Standardbibliotheken an: Netzwerkprogrammierung, Dateizugriff, Benutzeroberflächengestaltung und Datenbankverbindung.
- **Einfachere Einstiegshürde für Entwickler**: Eine zu C und C++ vergleichbare Syntax, den damals vorherrschenden Sprachen für Unternehmensanwendungen. Es fiel diesen Entwicklern leichter, Java zu lernen und effektiv in der neuen Sprache zu programmieren. Java das bessere, weil sichere und einfachere C++.
- **Anpassungsfähigkeit und Skalierbarkeit**: Java arbeitete kontinuierlich an seiner Hauptschwäche Performance und erwies sich als äußerst anpassungsfähig an neue Technologietrends (Internet, Mobile, IoT, Funktional, Cloud).

Java Code im Laufe der Zeit

- „Opa, wie hast Du eigentlich damals Java entwickelt?“
  - Text Editor, wenn man Glück hatte mit Syntax-Highlighting
  - Kompiliert mit javac im Terminal
  - IDE ? – wir hatten doch nichts!
  - Jetzt zeig‘ ich Dir mal Code von mir….
    - Projekt 01_history

Agenda

- Überblick Java
  - Java historisch
  - **Der Java Releaseprozess**
  - Aktuelle Projekte
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite

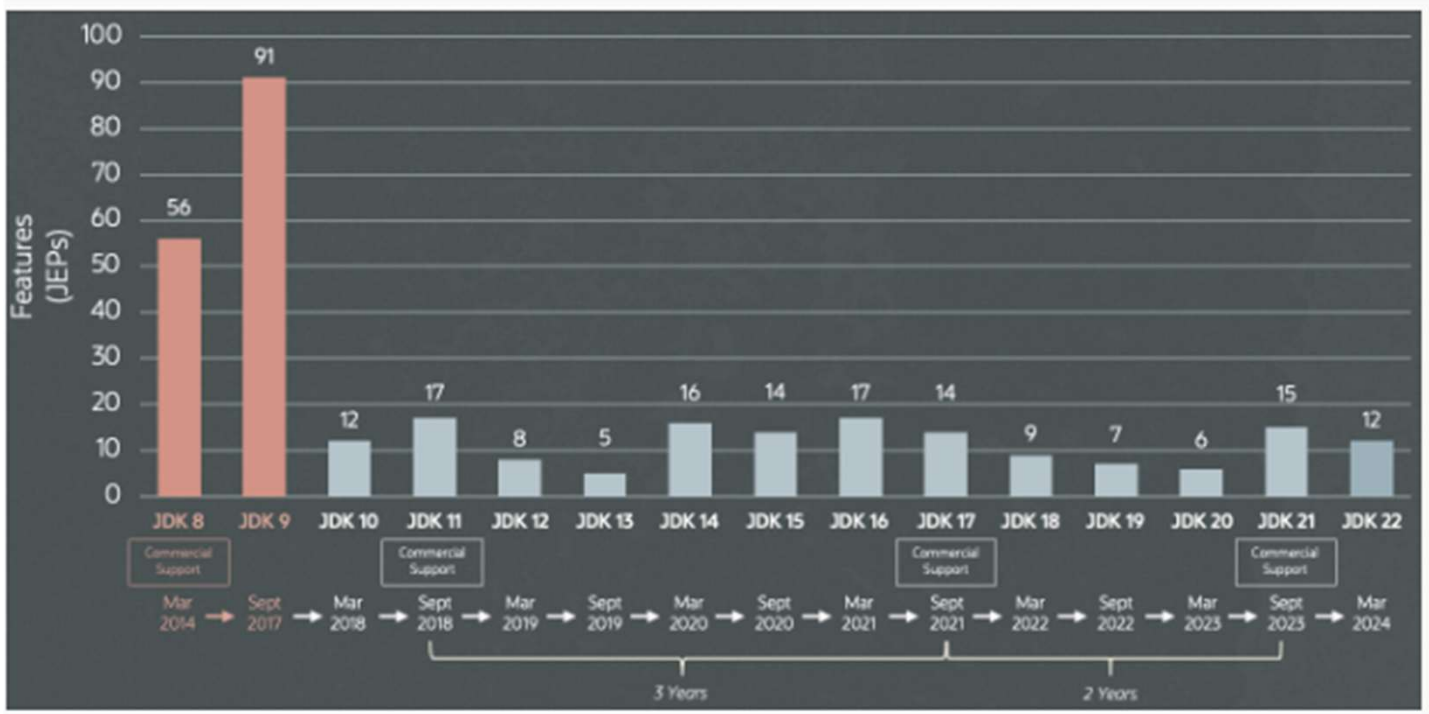
Java Entwicklungsprozess

- JCP: Java Community Process - Der JCP formalisiert die Art und Weise, wie neue Funktionen und Änderungen an Java (d.h. technische Spezifikationen) vorgeschlagen, geprüft und genehmigt werden, einschließlich der Definition verschiedener Rollen, die Personen einnehmen können.
- JSR: Java Specification Request – Vorschlagsprozess für “reifere” feature requests (Scheitern nicht erwartet)
- JEP: JDK Enhancement Proposal – leichtgewichtigerer Prozess für feature requests (Scheitern möglich)
- https://openjdk.org/jeps/0 - Liste aller JEPs

Neues „Feature Releases“ Modell seit Java 10

- Java Releases sollen sich nicht mehr verzögern (wie Java 7-9)
- Sie erscheinen seit Java 10 immer halbjährlich im März und September
- Major-Versionen erhalten nun lediglich ein halbes Jahr Patches bis zum nächsten Major (außer LTS Versionen)
- Alle 3 Jahre (**ab Java 21 alle 2 Jahre**) wird eine Version zur **L**ong **T**erm **S**upported Version erkoren, die für 8 Jahre LTS Security Patches erhält
  - Java 8 wurde auch zur LTS befördert mit Patches bis 2030
- Features, die noch in der Entwicklung sind und Feedback brauchen, sind dabei - aber müssen speziell eingeschaltet werden (Incubator, Preview)
- Es hat sich schnell gezeigt, dass dieses Vorgehen vorteilhaft ist, da neue Features so direkt produktions-gehärtet werden

Vorhersehbarkeit



Unterschiedliche Reifegrade

1. Preview: für neue Java platform features, die vollständig spezifiziert und implementiert, aber noch offen für Änderungen sind
2. Experimental: vor allem für neue features in der JVM
3. Incubating: für potentielle neue APIs und JDK tools

Entwicklung über mehrere Versionen

	Java 10	Java 11	Java 12	Java 13	Java 14	Java 15 (*)
Local-Variable Type Inference - `var`	Standard	→	→	→	→	→
Local-Variable Syntax for Lambda Parameters		Standard	→	→	→	→
Switch Expressions			Preview	2nd Preview	Standard	→
Text Blocks				Preview	2nd Preview	Standard
Records					Preview	2nd Preview
Pattern Matching for `instanceof`					Preview	2nd Preview
Sealed Classes						Preview

Lizenzmodelländerung Java 11

- Ab Java 11 wurde die zuvor übliche „Oracle Binary Code License“ (BCL) für neue Oracle-JDK-Releases durch eine Lizenz ersetzt, die kostenlose Nutzungen für geschäftliche Zwecke stark einschränkte.
- Während Oracle das Oracle JDK weiterhin zum Download bereitstellte, durfte man im kommerziellen Umfeld nur kostenlose Updates bis zum sogenannten „Ende des öffentlichen Updates“-Zeitpunkt nutzen. Danach benötigte man ein kostenpflichtiges Abonnement (Java SE Subscription), um weiterhin Sicherheitsupdates und Support von Oracle zu erhalten.
- Gleichzeitig gab (und gibt) es aber das *OpenJDK* (offiziell von Oracle und anderen), das weiterhin unter der *GPLv2 mit Classpath Exception*-Lizenz steht. Außerdem existieren diverse andere Distributoren wie z. B. Eclipse Temurin (Adoptium), Amazon Corretto, Red Hat, Azul, BellSoft, usw., die kostenlose und langfristig aktualisierte Builds anbieten.

Lizenzmodelländerung Java 17

- „Oracle No-Fee Terms and Conditions“ (NFTC) - Mit Java 17 (einem „Long-Term-Support“-Release) hat Oracle eine neue Lizenz eingeführt: Für die Laufzeit dieser Version (sprich bis zum Ende der Updates für Java 17) können die meisten Nutzer das Oracle JDK wieder frei einsetzen – auch kommerziell/produktiv.
- Die NFTC-Lizenz räumt eine kostenfreie Nutzung ein, allerdings nur für eine gewisse Zeit. Für Java 17 ist diese im September 2024 geendet und kommerzielle Nutzer benötigen ein kostenpflichtiges Abo von Oracle.
- Die aktuell kostenlose LTS ist Java 21
- Ausführliche Details stehen in den „No-Fee Terms and Conditions“.

Lizenzmodell - Auf Nummer sicher gehen

OpenJDK-Distributionen anderer Anbieter sind weiterhin eine gute Option, weil sie kostenfrei und oft mit Langzeit-Patches angeboten werden. Beispiele:

- Eclipse Temurin (Adoptium)
- Amazon Corretto
- Azul Zulu
- BellSoft Liberica

- Diese Distributionen sind vollkommen kostenfrei, auch für kommerzielle Nutzung, und bieten ebenfalls Sicherheitsupdates über den gesamten LTS-Zeitraum.

Agenda

- Überblick Java
  - Java historisch
  - Der Java Releaseprozess
  - **Aktuelle Projekte**
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite

Projekte

- Projekte und Roadmap: https://openjdk.org/jeps/1
- Amber – Kleine Verbesserungen für die Produktivität
- Babylon – Code Reflection, GPU Programming
- Liliput – Footprint reduzieren durch Kompression der object header
- Loom – Virtual Threads, Structured Concurrency
- Panama – Foreign Memory Access, Vektorrechnung
- Valhalla – Value Objects
- ZGC - a scalable low-latency garbage collector

Project Amber

- Das Ziel von Project Amber ist es, kleinere, produktivitätsorientierte Java-Sprachfeatures zu erforschen und zu entwickeln.
- Im Rahmen von Project Amber wurde z.B. auch local type inference (var) entwickelt, das größte Themenfeld ist/war aber „Pattern Matching“
- Pattern Matching in Java ist bislang mit Regex und Strings assoziiert, in anderen (funktionalen) Programmiersprachen definiert man es aber als:
- „Einen Mechanismus zur Überprüfung eines Wertes anhand eines Musters. Eine erfolgreiche Übereinstimmung kann einen Wert auch in seine Bestandteile zerlegen.“
- Die Switch Expressions (Java 14) sind der 1. Baustein, Pattern Matching für instanceof (16) der nächste.
- Weiter geht es mit: Records (16), Sealed Classes (17), Pattern Matching für Switch (21), Record Pattern (21) und mehr

Hintergründe und neue Themen

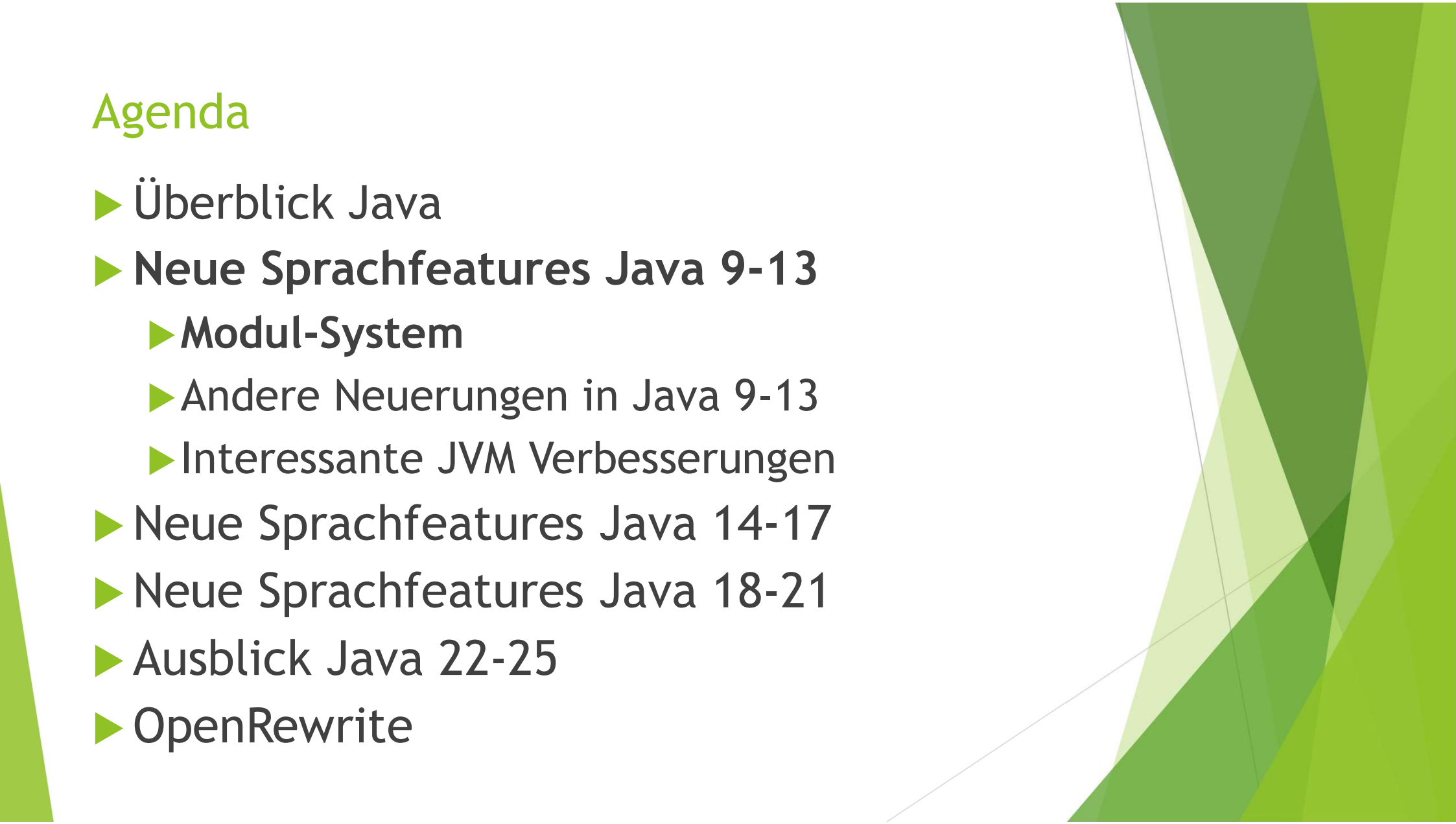
- https://www.infoq.com/articles/java-sealed-classes/
  - Data-centric Programming: Algebraische Typen nutzen und Pattern Matching einsetzen
  - Unterstütze Szenarien, besonders in gut verstandenen Domänen, wo Datenkapselung nicht genug Vorteile bringt sondern sogar Einfachheit und Transparenz erschwert
- Aktuelle Themen:
  - Primitive types in Patterns, instanceof, and switch
  - Statements before super(…)
  - String Templates
  - Implicitly Declared Classes and Instance main Methods

Projekt Valhalla

- Das Projekt Valhalla erweitert das Java-Objektmodell um Value Objects sowie Primitive Classes.
- Hier werden die Abstraktionen der objektorientierten Programmierung mit den Leistungsmerkmalen einfacher Primitiver Typen kombiniert:
  - *“Codes like a class, works like an int.”*
- Vorarbeiten: Nest-Based Access Control, Dynamic Class-File Constants (delivered in 11), JVM Constants API (12), Hidden Classes (15) und Warnings for Value-Based Classes (16)
- Nächstes Ziel: https://openjdk.org/jeps/401: Value Classes and Objects (Preview) – noch kein Zieltermin
- Dann noch: primitive classes, migrating the existing primitives, and universal generics

Agenda

- Überblick Java
- **Neue Sprachfeatures Java 9-13**
  - **Modul-System**
  - Andere Neuerungen in Java 9-13
  - Interessante JVM Verbesserungen
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite



Java 9 - 21.09.2017

1. New Language Features

- JEP 261: Module System
- JEP 277: Enhanced Deprecation
- Collections können bequem erzeugt werden
- Private Methoden in Interfaces
- Optional.stream()
- Try-with Resources improvement

2. JVM Improvements

- JEP 248: Make G1 the Default GC
- JEP 254: Compact Strings
- JEP 260: Encapsulate Most Internal APIs
- JEP 200: The Modular JDK
- JEP 220: Modular Run-Time Images
- Die neue ARM-Architektur AArch64 wird nun unterstützt.

3. New Tools and Libraries

- JEP 222: jshell
- Multi-Release-Jar files

4. Miscellaneous

- JEP 223: New Version-String Scheme
- Diverse Security Updates und Housekeeping
- Unicode 8 support

5. Incubator and Preview Features

- Neuer http Client (mit HTTP/2-Support)
- Experimental: Graal Compiler (AOT)

Module in der Software Entwicklung

- Modul ist kein neuer Begriff in der Softwareentwicklung
- Definitionen gehen in die Richtung „Teil einer Software“, „Sammlung von Algorithmen“ oder sogar „ein Arbeitspaket, dass man einem Programmierer oder Gruppe von Programmieren gibt“
- Heutzutage wird der Begriff meistens im Kontext von Information Hiding und Starker Kapselung also im Allgemeinen von Architektur benutzt
- Diese Sichtweise beruht ursprünglich auf einem Paper von David Parnas „ On the Criteria To Be Used in Decomposing Systems into Modules“ aus dem Jahr 1972(https://web.archive.org/web/20120223013018/http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf)
- Ende der 60er/Anfang der 70er wurde versucht aus der „Softwarekrise“(Software plötzlich teurer als Hardware) herauszukommen
- Das Design größerer Software musste verbessert werden

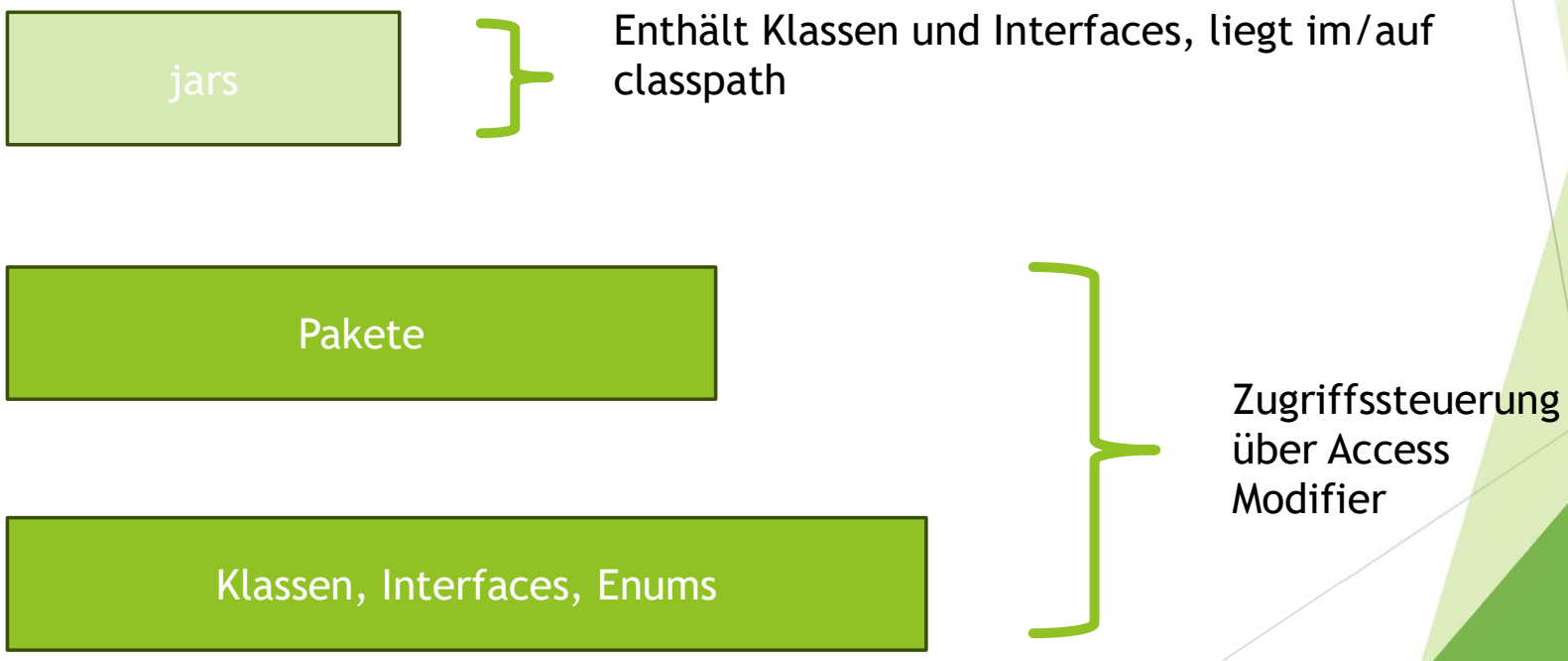
Vorteile von Modularisierung 1

- **Erhöhte Verständlichkeit**: Indem der Code in kleinere, überschaubare Module aufgeteilt wird, wird das Verständnis des Gesamtsystems erleichtert. Jedes Modul hat eine spezifische Aufgabe, was dazu beiträgt, die Komplexität des Gesamtcodes zu reduzieren.
- **Wiederverwendbarkeit**: Module können so gestaltet werden, dass sie in verschiedenen Teilen eines Projekts oder sogar in verschiedenen Projekten wiederverwendet werden können. Dies spart Entwicklungszeit und -kosten.
- **Einfachere Wartung und Fehlerbehebung**: Änderungen, Korrekturen oder Updates können auf ein bestimmtes Modul beschränkt werden, ohne das gesamte System zu beeinflussen. Dies erleichtert die Wartung und reduziert das Risiko, dass Änderungen an einem Teil des Systems unerwartete Probleme in anderen Teilen verursachen.

Vorteile von Modularisierung 2

- **Parallele Entwicklung**: Verschiedene Teams oder Entwickler können gleichzeitig an unterschiedlichen Modulen arbeiten, was die Entwicklungsgeschwindigkeit erhöht und den Koordinierungsaufwand reduziert.
- **Einfachere Tests**: Module können unabhängig voneinander getestet werden, was die Qualitätssicherung vereinfacht und eine effizientere Fehlersuche ermöglicht.
- **Flexibilität und Skalierbarkeit**: Änderungen oder Erweiterungen können leichter durchgeführt werden, da die Architektur es erlaubt, Module hinzuzufügen, zu entfernen oder zu ändern, ohne das gesamte System neu konzipieren zu müssen.
- **Abstraktion und Kapselung**: Durch die Modularisierung können interne Implementierungsdetails verborgen und eine klare Schnittstelle für die Interaktion mit dem Modul bereitgestellt werden. Dies fördert die Abstraktion und Kapselung von Code.

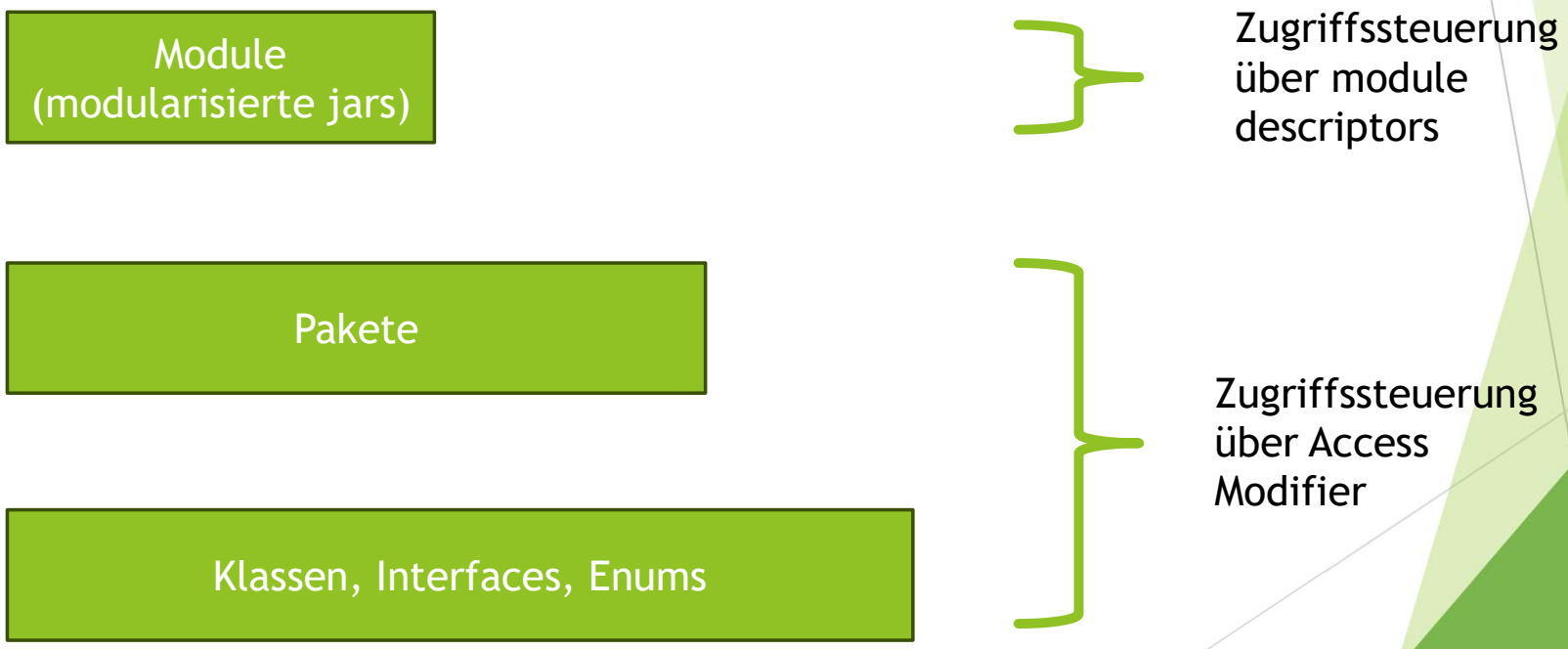
„Komponenten“ in Java bis Version 8



Was fehlt?

- Java Programme bis Version 8 bedienen sich des classpaths – das ist letztendlich ein simples Dateisystem: was drin liegt, darf benutzt werden
- Jars helfen nur insofern, als dass man diese über ein dependency management (maven, gradle) steuern kann
- Über Pakete und access modifiers kann man Schichten abbilden, aber es ist doch sehr kleinteilig
- Außerdem weiß man nicht, ob Nutzer des Codes nicht über Reflection auf Teile zugreifen, die als intern gedacht sind
- Module stoßen in diese Lücke

„Komponenten“ in Java ab Version 9



Das modularisierte jar

Modular JAR files

```
$ jar tf com.foo.bar-1.0.jar
META-INF/
META-INF/MANIFEST.MF
module-info.class
com/foo/bar/alpha/AlphaFactory.class
```

Neu

Java 9 - Module

- Module sind klassische jars mit einer zusätzlichen Beschreibungsdatei (module-info.class), die vor allem die Zugriffsrechte auf Paketebene regelt.
- Intern sind die jars weiterhin wie üblich in Paketen und Klassen organisiert
- Java Anwendungen können weiterhin herkömmlich (mit classpath) gestartet werden, dann verhalten sich Module wie jars
- Oder aber man startet die Anwendung als Modul, dann werden die Modul-Constraints mit einbezogen
- Man kann dabei noch-nicht modularisierte jars trotzdem problemlos mit einbeziehen
- Dabei werden direkt genutzte klassische jars zu sogenannten automatic modules

Nutzen von Java Modulen

- Stärkung der Kapselung, indem es verhindert, dass interne Implementierungsdetails ungewollt nach außen sichtbar werden
- Frameworks/Libraries können so sicherstellen, dass Consumer nur die öffentliche APIs nutzen → Überraschende Probleme bei Updates werden so vermieden (Einschränkung: wenn als Modul benutzt)
- Anwendungen können selbst modular aufgebaut werden, um die Wartbarkeit zu vereinfachen, so werden die Abhängigkeiten deutlich
- Große Codebasen können einfacher verwaltet werden
- Man kann eine auf das notwendige reduzierte JRE erzeugen und so Speicher für die Ausführung optimieren (besonders interessant für embedded)

Java 9 – Module declaration „modul-info.java“

- **module** - der Name des Moduls
- **requires** - Modul, von denen das Modul abhängt
- **exports** - Paket, auf die von außerhalb des Moduls zugegriffen werden kann
- **opens** - erlaubt anderen Klassen ausdrücklich den Zugriff auf die privaten Mitglieder eines Pakets über Reflection (sonst verboten)
- **provides** - wir können Dienstimplementierungen bereitstellen, die von anderen Modulen genutzt werden können
- **uses** – Services, die genutzt werden

Module declarations werden durch Kompilierung Module descriptors

Module declarations

```
$ cat module-info.java
module com.foo.bar {
    requires com.foo.baz;
    exports com.foo.bar.alpha;
    exports com.foo.bar.beta;
}
$ javac module-info.java
$ ls -l *.class
-rw-r--r-- 194 module-info.class
$
```

Module descriptors

Teil 1 - Zugriffssteuerung

Wird über **exports** (der packages) und **requires** (der Module) gesteuert

Man kann auch einschränken, für wen (welche Module) man exportiert:

- exports a.b.c to 1,2,3;

Reflection muss extra erlaubt werden mit **opens**

Auch das wieder einschränkbar

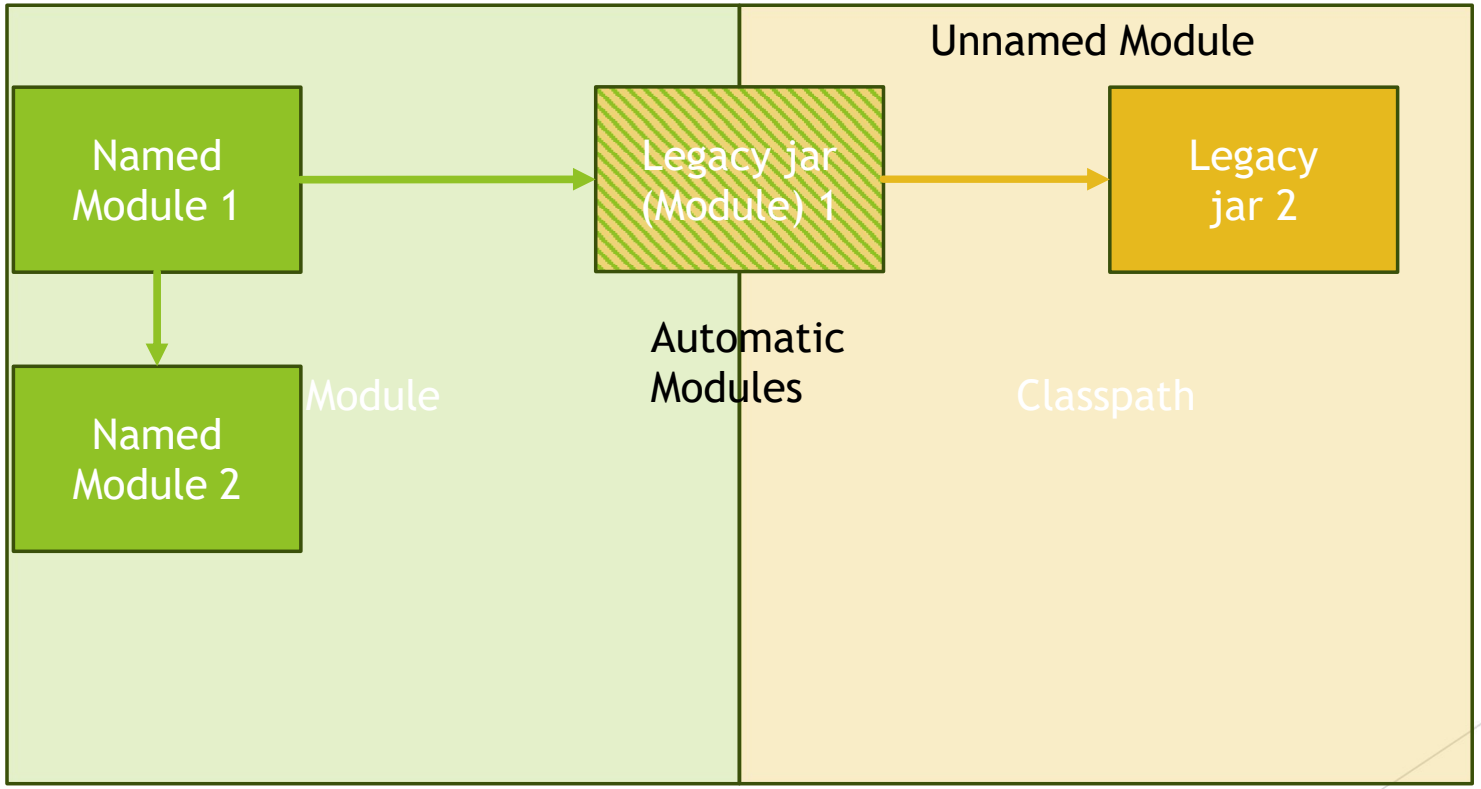
Zyklische Abhängigkeiten sind natürlich nicht erlaubt

Ein Paket kann nur genau in einem Modul exportiert werden (kein split package erlaubt)

Java 9 - Modultypen

- **Systemmodule** - Dies sind die Module des JDK und JSE.
- **Anwendungsmodule** - Diese Module sind das, was wir normalerweise bauen wollen, wenn wir uns für die Verwendung von Modulen entscheiden. Sie sind in der kompilierten Datei module-info.class benannt und definiert, die im assemblierten JAR enthalten ist.
- **Automatische Module** - Wir können inoffizielle Module einbinden, indem wir vorhandene JAR-Dateien zum Modulpfad hinzufügen. Der Name des Moduls wird aus dem Namen der JAR-Datei abgeleitet. Automatische Module haben vollen Lesezugriff auf alle anderen Module, die über den Pfad geladen werden.
- **Unbenanntes Modul** - Wenn eine Klasse oder ein JAR in den Klassenpfad, aber nicht in den Modulpfad geladen wird, wird sie/es automatisch dem unbenannten Modul hinzugefügt. Es ist ein Auffangmodul, um die Abwärtskompatibilität mit früher geschriebenem Java-Code zu wahren.

Java 9 - Rückwärtskompatibilität



Java 9 – Modul Beispiel

02b_usedLegacyLibs alleine öffnen und bauen

- 1 automatic module (02b_usedLegacyLibs)
  - legacy
- 1 jar, das in das unnamed module geladen wird (02b_usedLegacyLibs)
  - de.zettsystems.morelegacy

02_modules kann im großen Projekt dann benutzt werden

- 2 „echte“ Module (02_modules)
  - de.zettsystems.hello
  - de.zettsystems.main

Der Service Loader Mechanismus

Der Service Loader Mechanismus in Java ermöglicht das Laden von Diensten oder Service-Implementierungen zur Laufzeit auf eine modulare und entkoppelte Weise.

1. **Lose Kopplung**: Der Service Loader ermöglicht es, Dienste zu verwenden, ohne sich auf spezifische Implementierungen festzulegen, was die Kopplung zwischen verschiedenen Modulen oder Komponenten reduziert.
2. **Modularität**: Er unterstützt die Modularisierung von Anwendungen, indem er es ermöglicht, Dienste in unterschiedlichen Modulen zu definieren und zu verwenden.
3. **Erweiterbarkeit**: Neue Dienstleistungen können hinzugefügt werden, ohne bestehenden Code zu ändern, was die Erweiterung von Anwendungen erleichtert.
4. **Einfache Handhabung**: Der Service Loader ist relativ einfach zu verwenden und erfordert keine externen Bibliotheken oder Container.

Abgrenzung zu anderen ähnlichen Techniken 1

- **Dependency Injection (DI)**:
  - **DI-Frameworks** wie Spring oder Guice ermöglichen eine feinere Kontrolle über die Lebenszyklusverwaltung und die Konfiguration von Abhängigkeiten. Sie sind mächtiger und flexibler als der Service Loader, aber auch komplexer.
  - **Service Loader** ist einfacher und wird hauptsächlich für das Laden von Diensten verwendet, ohne erweiterte Funktionen wie Interceptors, Scopes, oder AOP (Aspect-Oriented Programming) zu bieten.
- **Java Naming and Directory Interface (JNDI)**:
  - **JNDI** wird verwendet, um auf eine Vielzahl von benannten Objekten und Diensten, typischerweise in einem Unternehmensumfeld, zuzugreifen. Es ist eher für den Zugriff auf externe Ressourcen in einem verteilten Umfeld gedacht.
  - **Service Loader** ist mehr auf das Laden von Diensten innerhalb der Anwendung selbst fokussiert und weniger komplex als JNDI.

Abgrenzung zu anderen ähnlichen Techniken 2

- **OSGi:**
  - **OSGi** bietet ein viel umfassenderes Modulsystem, das dynamisches Laden, Entladen und Aktualisieren von Modulen zur Laufzeit ermöglicht, einschließlich komplexer Abhängigkeitsverwaltung und Versionierung.
  - **Service Loader** ist einfacher und bietet nicht die dynamischen Funktionen von OSGi. Es ist mehr für statische Modulbeladung in Anwendungen gedacht, die nicht die volle Dynamik und Komplexität von OSGi benötigen.

Teil 2 – Service Loader Mechanismus in module-info

- Wird über **provides** (einer oder mehrerer Serviceimplementierung(en)) und **uses** (als Client der Services) gelöst
- Es ist möglich Service und Implementierung im gleichen Modul zu haben, das ist aber nicht die Idee
- Vielmehr sollen Service (Api) – Module und Implementierungsmodule getrennt sein, um lose Kopplung zu erzielen
- Im Kontext von Spring oder CDI nicht wirklich relevant, aber eine Art DI für plain Java Projekte

Java 9 – Modul Beispiel Service Lookup

02c_modules_loader

- de.zettsystems.greeting – API Modul
- de.zettsystems.greeting.impl - Implementierung (provides Service)
- de.zettsystems.app – Client, kennt Api aber nicht Impl (uses service)

Einsatzzwecke des Service Loader Mechanismus

- **Plattformübergreifende Dienste**: In Softwareplattformen, die verschiedene Implementierungen desselben Dienstes für unterschiedliche Betriebsumgebungen bereitstellen müssen
- **Plugin-Systeme**: In Anwendungen, die Erweiterungen oder Plugins unterstützen, kann der Service Loader verwendet werden, um diese Plugins dynamisch zu laden und zu integrieren, ohne den Hauptanwendungscode zu ändern
- **Treiber und Provider**: In Systemen, die verschiedene Treiber oder Service-Provider unterstützen (wie Datenbanktreiber, Sicherheitsdienste, etc.), wird der Service Loader verwendet, um diese Komponenten dynamisch zu entdecken und zu laden
- **Bibliotheken:** Beispielsweise verwendet die Java Cryptography Architecture (JCA) den Service Loader, um Kryptografie-Provider zu laden oder Logging-Frameworks können ihn verwenden, um verschiedene Logging-Implementierungen zu unterstützen

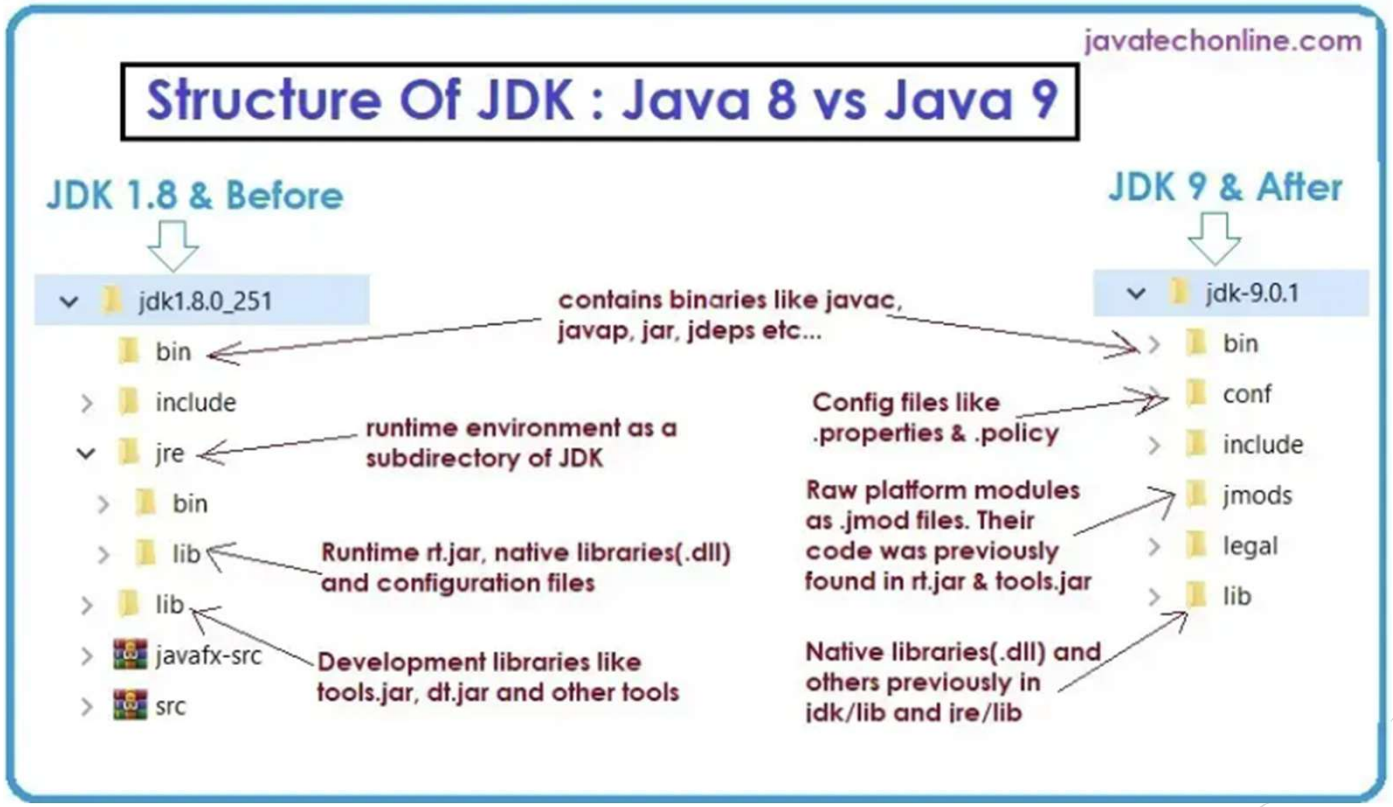
Design Patterns mit Hilfe des SLM

- **Service Provider Interface (SPI) Pattern**: Dieses Muster wird häufig in Verbindung mit dem Service Loader verwendet. Es trennt die Definition eines Dienstes (ein Interface oder abstrakte Klasse) von seiner Implementierung
- **Factory Pattern**: Der Service Loader kann verwendet werden, um das Factory Pattern zu implementieren, bei dem ein Interface für die Erstellung eines Objekts definiert ist, aber die tatsächliche Erstellung der Instanz zur Laufzeit basierend auf verfügbaren Dienstleistungen oder Konfigurationen erfolgt.
- **Strategy Pattern:** Bei diesem Muster können verschiedene Algorithmen oder Strategien durch unterschiedliche Dienstleistungen repräsentiert werden. Der Service Loader kann genutzt werden, um dynamisch die geeignete Strategie zu laden und zu verwenden, ohne den Client-Code zu ändern.
- **Adapter Pattern**: Dieses Muster kann genutzt werden, um die Schnittstelle eines Dienstes an die eines anderen anzupassen. Der Service Loader kann verwendet werden, um verschiedene Adapter zur Laufzeit zu laden, wodurch die Flexibilität erhöht wird.

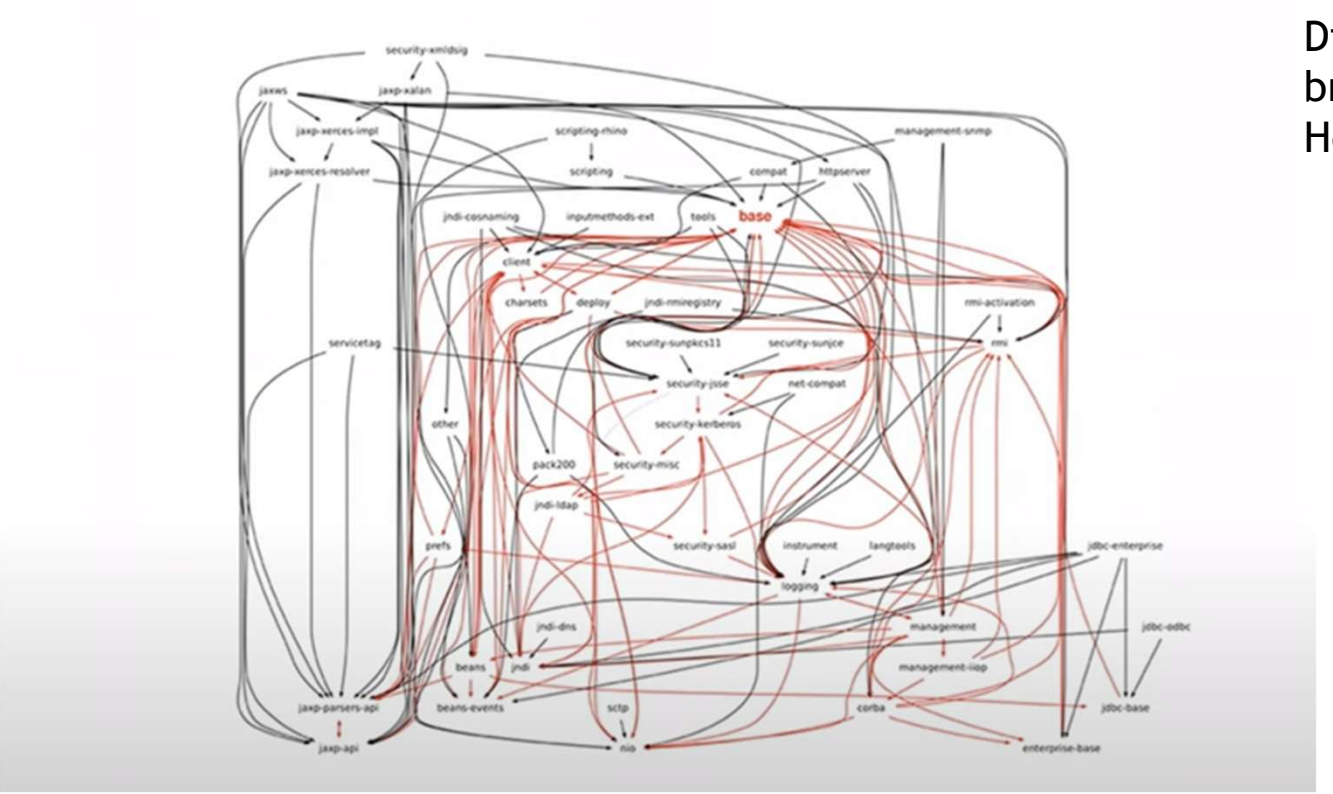
Übung

- Unter 03_modules_exercise sind zwei legacy Projekte
  - User benutzt Calculator
- Aufgabe 1: Die Projekte sollen modularisiert werden:
  - Beide Projekte zu Modulen machen
- Aufgabe 2: Die Projekte sollen Service Lookup nutzen:
  - Fleißbienchen: die Implementierung soll in ein 3. Projekt ausgelagert werden

Neue Struktur des JDK

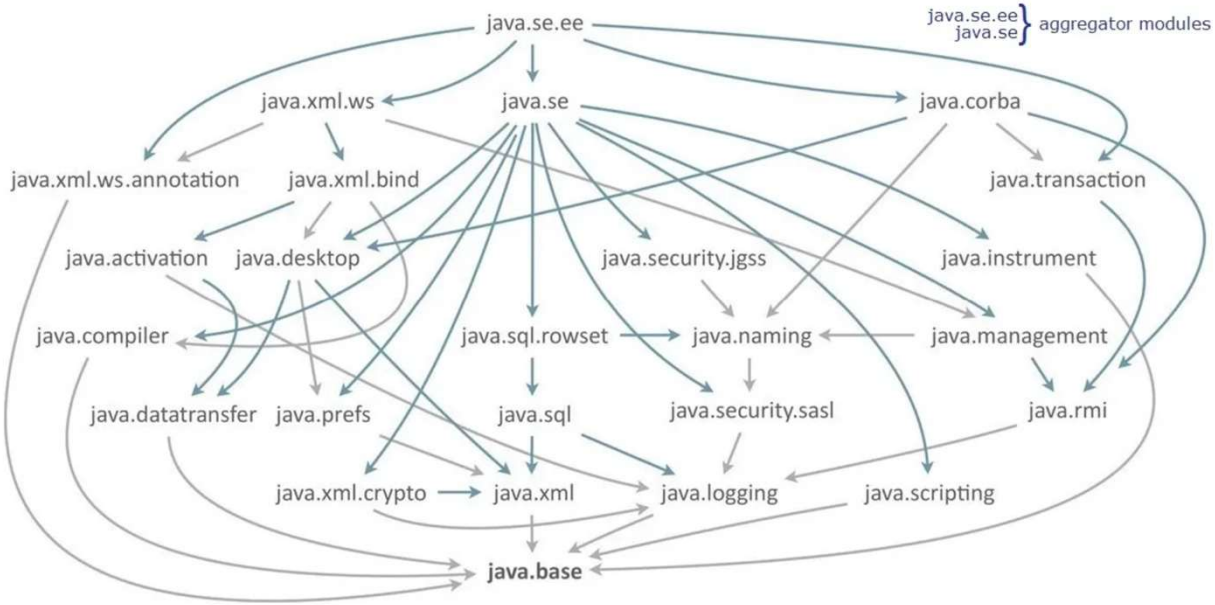


Abhängigkeiten im alten JDK



Die roten Kanten braucht man für HelloWorld…

Abhängigkeiten im modularisierten JDK



Man braucht nur
noch java.base für
HelloWorld…

Warum (nicht) Module

- Das jdk ist das beste Beispiel: Es hat 5 Jahre gedauert die Codebasis zu dem nun modularisierten Zustand zu refaktorisieren
  - Offenbar war dort grundsätzlich Etwas im Argen
  - Offenbar war es aber auch sehr teuer
- Module erzwingen, dass man sich auf höherem Niveau als der Klasse Gedanken über das Design seiner Anwendung macht
- Je größer die Codebasis und das Team, desto mehr bringt der Einsatz von Modulen
- Je weiter verbreitet die public API einer Software, desto mehr bringt der Einsatz von Modulen → Frameworks und libraries profitieren (wenn die Nutzer denn modular arbeiten…)

Alternativen zu Java Modulen

- Microservice-Architektur
  - Wenn die Software hinreichend klein ist und mit klaren Schnittstellen arbeitet, wie bei diesem Architektur-Ansatz, ist die Wartbarkeit vmtl. gut genug
- Maven (Sub-)Module
- Tools wie ArchUnit
- Im Spring Umfeld: Spring Modulith
- OSGI

Maven Module als Alternative

Schichtenarchitektur

- UI soll nur auf Service zugreifen, Service nur auf Data Access
- Mit Maven Modulen können wir den Zugriff verhindern

Vorher:

- 1 Maven Projekt

Nachher:

- 3 Maven Module
- UI kennt nur Service
- Service kennt nur Data Access
- Durch passende Scopes (provided) transitive Vererbung verhindern

ArchUnit als Alternative

- https://www.archunit.org/
- Schreibe Unit Tests, um Deine Architekturregeln zu überprüfen
- (auch andere Regeln wie bei statischen Code-Checkern (Sonarqube) üblich sind möglich)
- In dem Beispiel von gerade würde der Unit Test fehlschlagen, der überprüft, welche packages auf welche packages zugreifen
- So kann man leichtgewichtiger, schrittweise eine gewünschte Struktur aufbauen, die dann irgendwann in Modulen münden könnte

- https://github.com/TNG/ArchUnit-Examples

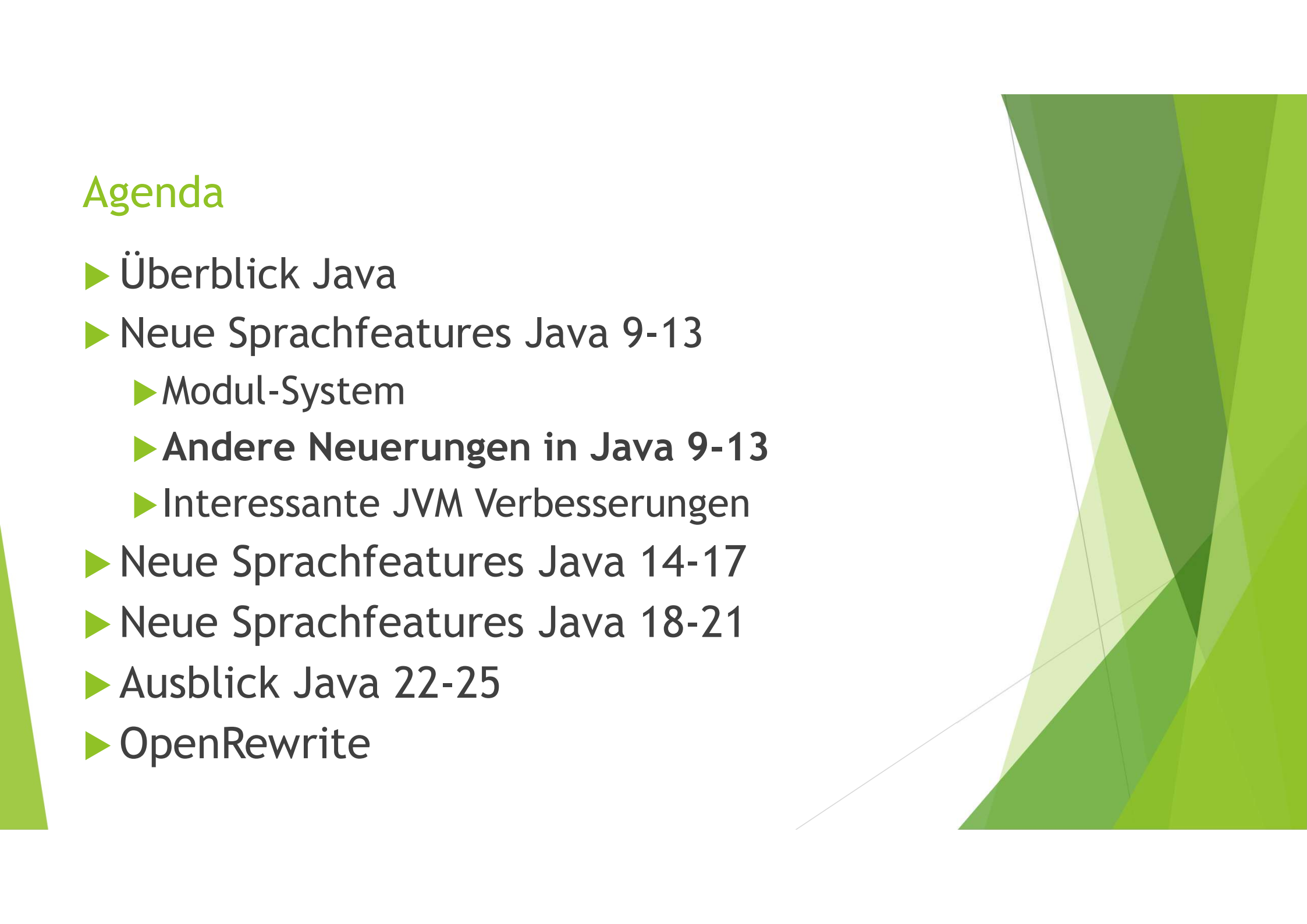
Probleme mit dem abgeschotteten jdk

- Auch wenn man nicht mit Modulen arbeitet, kann man seit Java 17 Probleme kriegen, wenn man bestimmte Klassen des jdk direkt oder per Reflection nutzt
- Die Internas sind nun abgesichert (seit java9) und der Kompatibilitäts-Schalter ist weg (seit Java 16 default: deny, seit Java 17 weg)
- Code: 03c_problem

```
Caused by: java.lang.reflect.InaccessibleObjectException: Unable to make field private final java.util.Comparator java.util.TreeMap.comparator accessible:
    at java.lang.reflect.AccessibleObject.throwInaccessibleObjectException (AccessibleObject.java:391)
    at java.lang.reflect.AccessibleObject.checkCanSetAccessible (AccessibleObject.java:367)

module java.base does not "opens java.util" to unnamed module @7d83e882
```

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
  - Modul-System
  - **Andere Neuerungen in Java 9-13**
  - Interessante JVM Verbesserungen
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite



Java 9 - 21.09.2017

1. New Language Features

- JEP 261: Module System
- JEP 277: Enhanced Deprecation
- Collections können bequem erzeugt werden
- Private Methoden in Interfaces
- Optional.stream()
- Try-with Resources improvement

2. JVM Improvements

- JEP 248: Make G1 the Default GC
- JEP 254: Compact Strings
- JEP 260: Encapsulate Most Internal APIs
- JEP 200: The Modular JDK
- JEP 220: Modular Run-Time Images
- Die neue ARM-Architektur AArch64 wird nun unterstützt.

3. New Tools and Libraries

- JEP 222: jshell
- Multi-Release-Jar files

4. Miscellaneous

- JEP 223: New Version-String Scheme
- Diverse Security Updates und Housekeeping
- Unicode 8 support

5. Incubator and Preview Features

- Neuer http Client (mit HTTP/2-Support)
- Experimental: Graal Compiler (AOT)

Java 9 - CompletableFuture

- In Java 9 wurden die bereits mit Java 8 eingeführten CompletableFutures um sinnvolle Methoden erweitert, etwa:
  - Timeout-Handling: orTimeout(...), completeOnTimeout(...)
  - Asynchrone Komplettierung: completeAsync(...)
  - Damit lassen sich asynchrone Berechnungen noch präziser steuern, zum Beispiel mit individuellen Thread-Pools, ohne den Code übermäßig zu verkomplizieren.
  - Vorteil: Bessere Kontrolle, insbesondere in komplexen Systemen, wo verschiedene Tasks parallel (und ggf. mit Timeout) laufen sollen.
- Code: 04_java9_13/de.zettsystems.java9_10.complete

Java 9 - Unmodifiable Collections erzeugen

- Fabrikmethoden: In Java 9 wurden statische Methoden wie List.of(...), Set.of(...), Map.of(...) und Map.ofEntries(...) eingeführt.
- Damit lassen sich unveränderliche (immutable) Collections in nur einer Zeile erstellen: List<String> list = List.of("A", "B", "C");
- Die zurückgegebenen Collections darf man nicht verändern (Schreiboperationen führen zu UnsupportedOperationException).
- Wichtig: Sie erlauben keine null-Elemente; der Versuch einer List.of(null) löst eine NullPointerException aus.

- Code: 04_java9_13/de.zettsystems.java9_10.collections
  - Auch dort: Deprecated mit forRemoval und since

Java 9 - Process API

- Die neu gestaltete Process-API in Java 9 ermöglicht es, alle laufenden Prozesse zu inspizieren oder den aktuellen Prozess detailliert abzufragen (z. B. PID, Kommandozeile, CPU-Zeit, Startzeit).
- Bietet zusätzlich Callback-Mechanismen (onExit()) oder Möglichkeiten, einzelne Prozesse zu beenden.
- Vorteil: Deutlich einfacher, plattformunabhängig auf Prozessinformationen zuzugreifen – kein (unzuverlässiges) Parsen externer Kommandoausgaben mehr erforderlich.
- Code: 04_java9_13/ de.zettsystems.java9_10.misc

Java 9 - Stack Walking

- Mit StackWalker gibt es eine effizientere Alternative zu Thread.currentThread().getStackTrace().
- Der Stack lässt sich gezielt durchsuchen und filtern, ohne gleich ein komplettes Throwable zu erzeugen oder alle Frames laden zu müssen.
- Vorteil: Bessere Performance und Flexibilität beim Debuggen, Loggen oder Sammeln von Telemetriedaten.
- Code: 04_java9_13/ de.zettsystems.java9_10.misc

Java 9 - Reactive Streams (Flow)

- java.util.concurrent.Flow definiert die standardisierten Interfaces (Publisher, Subscriber, Processor, Subscription) für Reactive Streams mit Backpressure.
- Ziel: Nicht-blockierende, asynchrone Datenverarbeitung großer Datenmengen, bei der der Datenkonsument (Subscriber) steuert, wie viele Elemente er verarbeiten kann (Backpressure).
- Vorteil: Ein einheitliches Fundament für reaktive Bibliotheken (z. B. RxJava, Project Reactor, Akka Streams), das bereits im JDK enthalten ist.
- Code: 04_java9_13/ de.zettsystems.java9_10.misc

Weitere kleine Themen in Java 9

04_java9_13 de.zettsystems.java9_10.misc

- Private Methods in Interfaces
- Try-with-resources verbessert

Java 9/10 – Neue Methoden Stream u .Optional

- In Java 9 (und teils in Java 10) wurden einige zusätzliche Methoden hinzugefügt, um noch komfortabler mit Optional umzugehen:
- ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction), or(Supplier<? extends Optional<? extends T>> supplier), stream(), orElseThrow() (ohne Argument) (ab Java 10)
- Auch Stream hat einige convenience-Methoden bekommen:
- takeWhile(Predicate<? super T>), dropWhile(Predicate<? super T>), ofNullable(T value), iterate(seed, hasNext, next)
- Code: 04_java9_13/ de.zettsystems.java9_10.streams

Java 10 - 20.03.2018

1. New Language Features

- JEP 286: Local-Variable Type Inference
- Optional.orElseThrow() Method
- Weitere APIs um Unmodifiable Collections zu erstellen

2. JVM Improvements

- JEP 307: Parallel Full GC for G1
- Bytecode Generation for Enhanced for Loop improved

3. New Tools and Libraries

- JEP 317: Experimental Graal Compiler (JIT)

4. Miscellaneous

- Neues Releasemodell
- Version nun zusätzlich mit Release Date
- Diverse Security Updates und Housekeeping

5. Incubator and Preview Features

- JEP 310: Application Class-Data Sharing

Java 10 - Unmodifiable Kopien erzeugen

- copyOf-Methoden in Java 10:
- List.copyOf(...), Set.copyOf(...) und Map.copyOf(...) erzeugen ebenfalls unmodifiable Kopien aus einer bestehenden Collection bzw. Map.
- Beispiel: List<String> unmodList = List.copyOf(anyOtherList);
- Wenn die Eingabe-Collection bereits unmodifiable ist, wird sie direkt zurückgegeben (keine neue Kopie).
- Vorteile: Kürzerer Code und weniger Boilerplate im Vergleich zu Collections.unmodifiableList(...) oder Arrays.asList(...) plus Kopie.
- Immutability erhöht die Sicherheit (keine versehentlichen Modifikationen) und kann Performance-/Konsistenzvorteile haben. Keine Null-Werte und daher weniger potenzielle NullPointerException-Stellen beim Umgang mit diesen Collections.
- Code: 04_java9_13/de.zettsystems.java9_10.collections

Java 10 (11) – Local-Variable Type Inference

- Java 10: Der Typ lokaler Variablen kann nun durch den Compiler inferiert werden, der Entwickler schreibt lediglich noch “var”
  - Sollte im Team besprochen werden, wie man das nutzen möchte
  - Sinnvoll bei sonst sehr langen z.B. parametrierten oder ge-wrapped-en Typen
  - Styleguide: https://openjdk.org/projects/amber/guides/lvti-style-guide

- Java 11: Local-Variable Syntax jetzt auch für Lambda
  - Meistens spart man sich den Typen beim Lambda (short form)
  - Wenn man ihn hinschreibt (long form) hat man aber die Möglichkeit Annotations hinzuzufügen
  - Hier kann man nun auch local type inference (var) einsetzen

- Code: 04_java9_13/de.zettsystems.java9_10.localtypeinf

Java 11 - 25.09.2018 (LTS bis 2026)

1. New Language Features

- New String Methods
- New File Methods
- Collection to an Array
- The Not Predicate Method
- Local-Variable Syntax for Lambda
- new HTTP API

2. JVM Improvements

- Dynamic Class-File Constants
- Enhanced Aarch64 Intrinsics

3. New Tools and Libraries

- directly run the source file
- Java Flight Recorder

4. Miscellaneous

- Änderung des Lizenzmodells
- Diverse Security Updates und Housekeeping
- Unicode 10 support
- Java FX und Java Mission Control nun eigene Downloads

5. Incubator and Preview Features

- JEP 333: ZGC A Scalable Low-Latency Garbage Collector (Experimental)
- JEP 318: Epsilon, A No-Op Garbage Collector (Experimental)

Java 11 – neuer http Client

- Löst die low-level HttpURLConnection-API ab
- Soll auf dem Niveau von Apache HttpClient usw. sein
- implementiert HTTP/2 und Web Socket
- Die API besteht aus drei Kernklassen:
  - **HttpRequest** stellt die Anfrage dar, die über den HttpClient gesendet wird.
  - **HttpClient** fungiert als Container für Konfigurationsinformationen, die für mehrere Anfragen gelten.
  - **HttpResponse** stellt das Ergebnis eines HttpRequest-Aufrufs dar.

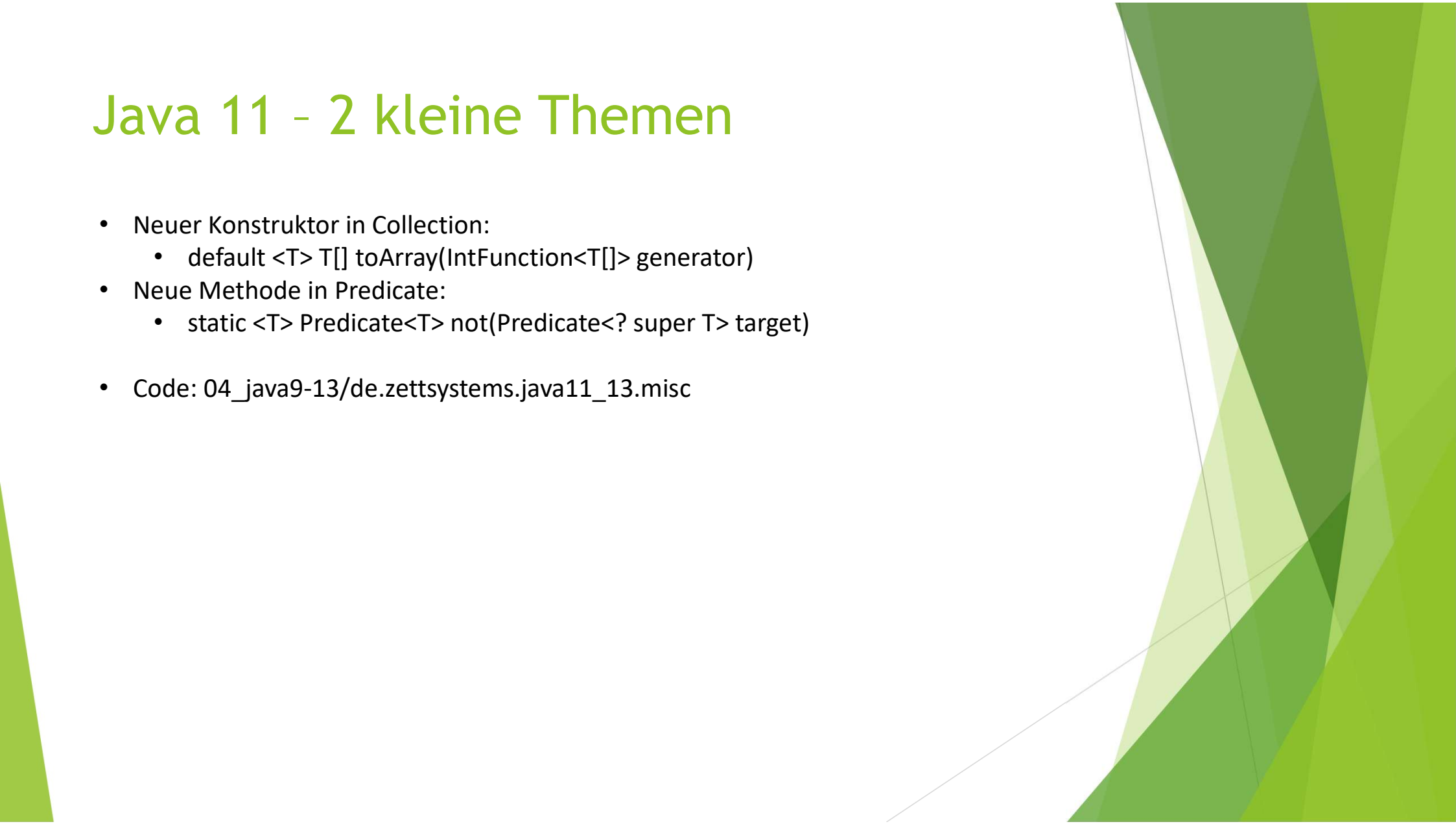
- Ein Vergleich unterschiedlicher Clients:
  - https://www.innoq.com/de/articles/2022/01/java-http-clients/

- Code: 04_java9-13/de.zettsystems.java11_13.http

Java 11 – 2 kleine Themen

- Neuer Konstruktor in Collection:
  - default <T> T[] toArray(IntFunction<T[]> generator)
- Neue Methode in Predicate:
  - static <T> Predicate<T> not(Predicate<? super T> target)

- Code: 04_java9-13/de.zettsystems.java11_13.misc



Java 12 – 19.03.2019

1. New Language Features

- Compact Number Formatting
- Teeing Collector
- File::mismatch Method
- New String methods indent(),transform()

2. JVM Improvements

- G1GC Improvements
- Default CDS Archives

3. New Tools and Libraries

- Microbenchmark Suite

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- Unicode 11 support

5. Incubator and Preview Features

- JEP 325: Switch Expressions (Preview)
- ZGC Concurrent Class Unloading (experimental)
- Shenandoah: A Low-Pause-Time Garbage Collector (experimental)

Java 13 – 17.09.2019

1. New Language Features

- FileSystems.newFileSystem() Method

2. JVM Improvements

- JEP 350: Dynamic CDS Archives
- JEP 353: Reimplement the Legacy Socket API

3. New Tools and Libraries

- New keytool -showinfo -tls Command for Displaying TLS Configuration Information

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- Unicode 12.1 support

5. Incubator and Preview Features

- JEP 354: Switch Expressions yield (2nd Preview)
- JEP 354: Text Blocks (Preview)
- JEP 351: ZGC: Uncommit Unused Memory (experimental)

Java 12 - 13 Code

- 04_java9-13 de.zettsystems.java11_13
  - .strings: neue Methoden in String
  - .files: neue Methoden in Files
  - .misc: Collectors.teeing()
  - .misc: Kompaktes NumberFormat

Aufgaben Java 9-13

- Schau Dir den Code unter 04_java9_13/de.zettsystems.uebung913 an – was kann man mit den neuen Features vereinfachen?

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
  - Modul-System
  - Andere Neuerungen in Java 9-13
  - **Interessante JVM Verbesserungen**
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite

Neue „interne“ Features „nutzen“

- Das Beste ist – vielleicht etwas überraschend: schon durch das Nutzen des neuen JDKs nach nur den notwendigen Anpassungen, um kompilieren zu können, nutzt man automatisch:
  - Alle Bugfixes
  - Security Verbesserungen
  - Performancesteigerungen

Java 9 - String Compaction

„Measurements have shown that roughly 25% of the Java heap live data set in these types of applications is consumed by String objects.“ (JEP-192)

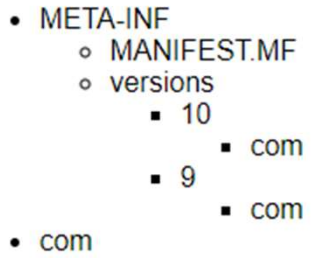
```
Inspecting heap file idea.hprof
Total number of String instances:
    LATIN1 2,656,321
    UTF16  47,231
    Total  2,703,552
Java 8 memory used by String instances:
    Total  275,473,456 bytes
Java 11+ memory used by String instances:
    LATIN1 192,371,520 bytes
    UTF16  8,066,152 bytes
    Total  200,437,672 bytes
Saving of 27.24%
```

```
Inspecting heap file javaspecialists.hprof
Total number of String instances:
    LATIN1 165,865
    UTF16  174
    Total  166,039
Java 8 memory used by String instances:
    Total  15,145,800 bytes
Java 11+ memory used by String instances:
    LATIN1 11,210,944 bytes
    UTF16  111,600 bytes
    Total  11,322,544 bytes
Saving of 25.24%
```

https://www.javaspecialists.eu/archive/Issue306-Measuring-compact-strings-memory-savings.html

Java 9 - Multi-Release-Jars

- Wir können nun jars ausliefern, die für unterschiedliche Java Versionen kompilierten Code mitbringen
- Das jar sieht dann z.B. so aus:



- Der Code für Java 8 und älter ist der direkt unter Root
- Die Versionen 9+ haben eigene Folder unter META-INF/versions für abweichende Klassen
- In der MANIFEST.MF steht Multi-Release: true
- Ein jdk in Version x kann Code auch für Versionen kleiner als x kompilieren

Java 9 -Showcase multi-jar

- https://github.com/trishagee/multi-version-jar
- https://blog.jetbrains.com/idea/2017/10/creating-multi-release-jar-files-in-intellij-idea/

Java 9/10 – Runtime Versionierung

- Versionierung wurde auf Semantic Versioning umgestellt
- Java 9: Vorher: 1.8.0_291 … Nachher: 9.0.2
- Java 10: Vorher: 9.0.2 … Nachher: 10.0.0 2018-03-20
- Api-Support dafür: de.zettsystems.java9_10.misc.RunVersion

Semantic Versioning

Auf Grundlage einer Versionsnummer von MAJOR.MINOR.PATCH werden die einzelnen Elemente folgendermaßen erhöht:

- MAJOR wird erhöht, wenn API-inkompatible Änderungen veröffentlicht werden,
- MINOR wird erhöht, wenn neue Funktionalitäten, die kompatibel zur bisherigen API sind, veröffentlicht werden, und
- PATCH wird erhöht, wenn die Änderungen ausschließlich API-kompatible Bugfixes umfassen.

Außerdem sind Bezeichner für Vorveröffentlichungen und Build-Metadaten als Erweiterungen zum MAJOR.MINOR.PATCH-Format verfügbar.

Bewertung Semantic Versioning

- Die meisten OpenSource Bibliotheken arbeiten mittlerweile mit Semantic Versioning
- Eigentlich sollte man Patch/Minor Updates machen können, ohne negative Auswirkungen zu erfahren
  - Es gibt aber sicherlich Ausnahmen (Pact JVM, Spring Boot Minor Updates manchmal,…)
  - Diese sollte man dann besonders berücksichtigen
- Major-Updates sind dann die, die oft nur mit (etwas) Aufwand durchführbar sind

Java 11 – Epsilon No-Op GC

- Ist und bleibt „experimental“
- Nur für Testumgebungen gedacht
- Es wird kein GC durchgeführt (das ist für bestimmte Testszenarien interessant)
- VM Options, um diesen zu verwenden:
  - `-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC`

Java 9 – Custom JRE

- Notwendige System Dependencies mit jdeps herausfinden
- Jlink nutzen, um abgespecktes JRE zu erzeugen
- Es ist auch möglich seine eigenen Module mit in das JRE zu packen

Showcases Kommandozeile

- 04_modules_cmd
  - jdeps
  - Jlink
- Java 9 - Jshell
- Java 11 - On-the-fly kompilieren

Java 12 - Microbenchmark

- Als neues Feature in Java 12 geführt, muss man aber tatsächlich extra dependencies benutzen wenn man nicht openjdk benutzt
- https://github.com/openjdk/jmh
- Tutorial: https://jenkov.com/tutorials/java-performance/jmh.html
- Kann man nutzen, um „verdächtige“ Codestrukturen auf Performance-Probleme zu untersuchen

Java 12 - Microbenchmark Code

- 04_java9-13de.zettsystems.java11_13.microbench

Java 12 - Benchmark Result - Array

Bench	0	1	10	100
arrayNew	(2,756, 4,460, 14,412), stdev = 1,668	(10,636, 14,387, 24,870), stdev = 2,660	(16,864, 29,366, 682,340), stdev = 66,262	(108,100, 131,660, 247,779), stdev = 24,271
simple	(3,492, 4,192, 12,730), stdev = 1,138	(8,405, 9,216, 22,331), stdev = 1,435	(11,569, 15,732, 193,210), stdev = 18,474	(66,482, 108,830, 798,137), stdev = 79,292
sized	(3,050, 3,552, 5,986), stdev = 0,554	(9,432, 10,823, 19,263), stdev = 1,243	(18,432, 21,171, 50,855), stdev = 4,131	(110,935, 122,572, 168,551), stdev = 11,204
Zero	(2,663, 4,159, 45,382), stdev = 4,441	(10,856, 12,610, 55,991), stdev = 4,586	(17,075, 19,529, 47,148), stdev = 3,451	(110,070, 165,873, 1136,497), stdev = 150,171

Java 12 - Benchmark Result - NumberVerification

Bench	False	True
Regex	0,886 s/op	0,899 s/op
Try/Catch	12,290 s/op	0,227 s/op
StringUtils	0,212 s/op	0,179 s/op

Java 12/13 – JVM Verbesserungen

- Default CDS Archives
  - Die seit Java 10 zur Verfügung stehende Option ist seit 12 default an
  - Ein solches Klassenarchiv steht allen JVM Instanzen gemeinsam zur Verfügung und reduziert daher die redundaten Haltung dieser Daten in jeder Instanz
- Dynamic CDS Archives
  - Mit Java 13 gibt es die Möglichkeit auch Archive von Anwendungsklassen dynamisch zu erzeugen und zu benutzen
  - https://docs.oracle.com/en/java/javase/17/vm/class-data-sharing.html#GUID-7EAA3411-8CF0-4D19-BD05-DF5E1780AA91
  - https://nipafx.dev/java-application-class-data-sharing/
- G1GC Verbesserungen
  - Der G1 prüft nun den Java-Heap-Speicher bei Inaktivität der Anwendung und gibt ihn ggf. an das Betriebssystem zurück.
  - Besseres Zeitmanagement durch abbrechbare Collections
- Legacy Socket API neu implementiert
  - Baut nun auf java.nio auf
  - Vorbereitung für virtual threads (Java 21)

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- **Neue Sprachfeatures Java 14-17**
  - **Java 14 - Switch Expressions**
  - Java 15 - Text Blocks
  - Java 16 - Pattern Matching for instanceof,Records
  - Java 17 - Sealed Classes
  - Interessante JVM Verbesserungen
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite

Java 14 – 17.03.2020

1. New Language Features

- JEP 361: Switch Expressions

2. JVM Improvements

- JEP 345: NUMA-Aware Memory Allocation for G1
- Parallel GC Improvements

3. New Tools and Libraries

- JFR Event Streaming for Flight Recorder

4. Miscellaneous

- Diverse Security Updates und Housekeeping

5. Incubator and Preview Features

- JEP 368: new escapes for Text Blocks (2nd Preview)
- JEP 305: Pattern Matching for instanceof (Preview)
- JEP 359: Records (Preview)
- JEP 358: Helpful NullPointerExceptions (Preview – default aus)
- ZGC on Windows (JEP 365) and macOS (JEP 364) – Experimental
- JEP 370: Foreign Memory Access API (Incubator)
- JEP 343: Packaging Tool (Incubator)

Switch Expressions (Java 14)

- Switch wurde 1:1 von C übernommen – wahrscheinlich wie viele Entscheidungen mit der Absicht der großen Menge von C-Entwicklern Java schmackhaft zu machen
- Wegen der seltsamen Semantik (fall-through, break) wurde aber in OO-Kreisen meist if/else bevorzugt
- Denoch gab es in Java 5 (Enum und Wrapper switch-bar) und 7 (String switch-bar) Weiterentwicklungen
- Im Kontext von Aufzählungen war es schon immer die übersichtlichere Alternative
- Mit Java 14 und den Switch Expression gibt es jetzt eine funktionale Variante bei der auch OO-Enthusiasten nicht mehr die Nase rümpfen müssen
- Mit Java 21 gibt es dann sogar die nächste Weiterentwicklung

Switch Expression Code

- 05_java14-17 de.zettsystems.java14_15.switchexpression

Quiz

```
enum Direction {
  UP, DOWN, HOLD;
}
…
public static void main(String[] args) {
  var dir = Direction.UP;
  switch(dir) {
    case UP -> System.out.print("up ");
    case DOWN ->  System.out.print("down"); // line n1
  }
}
```

Which statement is true? Choose one.

A. The code compiles and prints up.

The answer is A.

B. The code compiles and prints up down.

The answer is B.

C. The code fails to compile. To fix it you must add the following after line n1:
`case HOLD -> System.out.println("hold");`

The answer is C.

D. The code fails to compile. To fix it you must add the following after line n1:
`default -> System.out.println("default");`

The answer is D.

Quiz

```
enum Direction {
  UP, DOWN, HOLD;
}
…
public static void main(String[] args) {
  var dir = Direction.UP;
  switch(dir) {
    case UP -> System.out.print("up ");
    case DOWN ->  System.out.print("down"); // line n1
  }
}
```

Which statement is true? Choose one.

A. The code compiles and prints up.

The answer is A.

B. The code compiles and prints up down.

The answer is B.

C. The code fails to compile. To fix it you must add the following after line n1:
`case HOLD -> System.out.println("hold");`

The answer is C.

D. The code fails to compile. To fix it you must add the following after line n1:
`default -> System.out.println("default");`

The answer is D.

A ist richtig,
es ist Arrow Syntax aber im Switch Statement
und nicht Expression: kein fall-through,
aber man muss auch nicht alle Fälle abdecken

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
  - Java 14 - Switch Expressions
  - **Java 15 - Text Blocks**
  - Java 16 - Pattern Matching for instanceof, Records
  - Java 17 - Sealed Classes
  - Interessante JVM Verbesserungen
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite

Java 15 – 16.09.2020

1. New Language Features

- Added isEmpty Default Method to CharSequence
- JEP 378: Text Blocks released
- JEP 371: Hidden Classes

2. JVM Improvements

- JEP 377: ZGC released
- JEP 379: Shenandoah released
- Helpful NullPointerExceptions (default an)
- DatagramSocket API reimplemented

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping

5. Incubator and Preview Features

- JEP 384: Records (2nd Preview)
- JEP 375: Pattern Matching Type Checks (2nd Preview)
- JEP 360: Sealed Classes (Preview)
- JEP 383: Foreign Memory API (Incubator)

Text Blocks (Java 15)

- Text Blocks sind die neue Art und Weise Texte, die über mehrere Zeilen gehen, abzubilden
- Das Ergebnis eines Textblocks ist ein einfacher String
- Textblöcke beginnen mit einem """ (drei Anführungszeichen), gefolgt von optionalen Leerzeichen und einem Zeilenumbruch.
- Das Ende des Blocks ist wieder """ - was aber in der gleichen Zeile wie der letzte Text stehen kann
- Innerhalb der Textblöcke können wir Zeilenumbrüche und Anführungszeichen frei verwenden, ohne dass ein Zeilenumbruch erforderlich ist.
- So können wir wörtliche Fragmente von HTML, JSON, SQL oder was auch immer wir brauchen, auf elegantere und lesbarere Weise einfügen.
- Im resultierenden String werden der (Basis-)Einzug und der erste Zeilenumbruch nicht berücksichtigt.
- Dazu neue Methode formatted() in String als Alternative für String.format()…

Text Blocks und NPE Code

- 05_java14-17 de.zettsystems.java14_15.textblocks und de.zettsystems.java14_15.misc

Quiz

The closing delimiter of a Java text block is:

Select 1 option(s):

- """
- """ immediately followed by a new line
- """ followed by any number of white spaces and then a new line
- """ followed by one or more white spaces
- A new line followed by """.

Quiz

The closing delimiter of a Java text block is:

Select 1 option(s):

- """
- """ immediately followed by a new line
- """ followed by any number of white spaces and then a new line
- """ followed by one or more white spaces
- A new line followed by """.

A ist richtig

Quiz 2

Given:

```
String s1 = "Hello World";
String s2 = """
            Hello World""";
String s3 = """
            Hello World
            """;
System.out.println((s1 == s2)+" "+s2.equals(s3)+" "+s2.intern().equals(s3.intern()));
```

What is the result?

Select 1 option(s):

- ○ `true false true`
- ○ `true true true`
- ○ `false true false`
- ○ `true false false`
- ○ `false false false`

Quiz 2

Given:

```
String s1 = "Hello World";
String s2 = """
            Hello World""";
String s3 = """
            Hello World
            """;
System.out.println((s1 == s2)+" "+s2.equals(s3)+" "+s2.intern().equals(s3.intern()));
```

What is the result?

Select 1 option(s):

- ○ `true false true`
- ○ `true true true`
- ○ `false true false`
- ○ `true false false`
- ○ `false false false`

D (true false false) ist richtig

Quiz 3

What will the following code print?

```
String s1 = """
            a \
            b \t
            c \s
            """;
System.out.println(s1.length()+" "+s1.split("\\n").length);
```

Select 1 option(s):

- 11 3
- 12 3
- 11 4
- 10 2
- 10 3
- 9 2

Quiz 3

What will the following code print?

```
String s1 = """
            a \
            b \t
            c \s
            """;
System.out.println(s1.length()+" "+s1.split("\\n").length);
```

Select 1 option(s):

- 11 3
- 12 3
- 11 4
- 10 2
- 10 3
- 9 2

E isr richtig (10 2)
s1 contains: "a b \t\nc \s\n" i.e. a total of 10 characters.

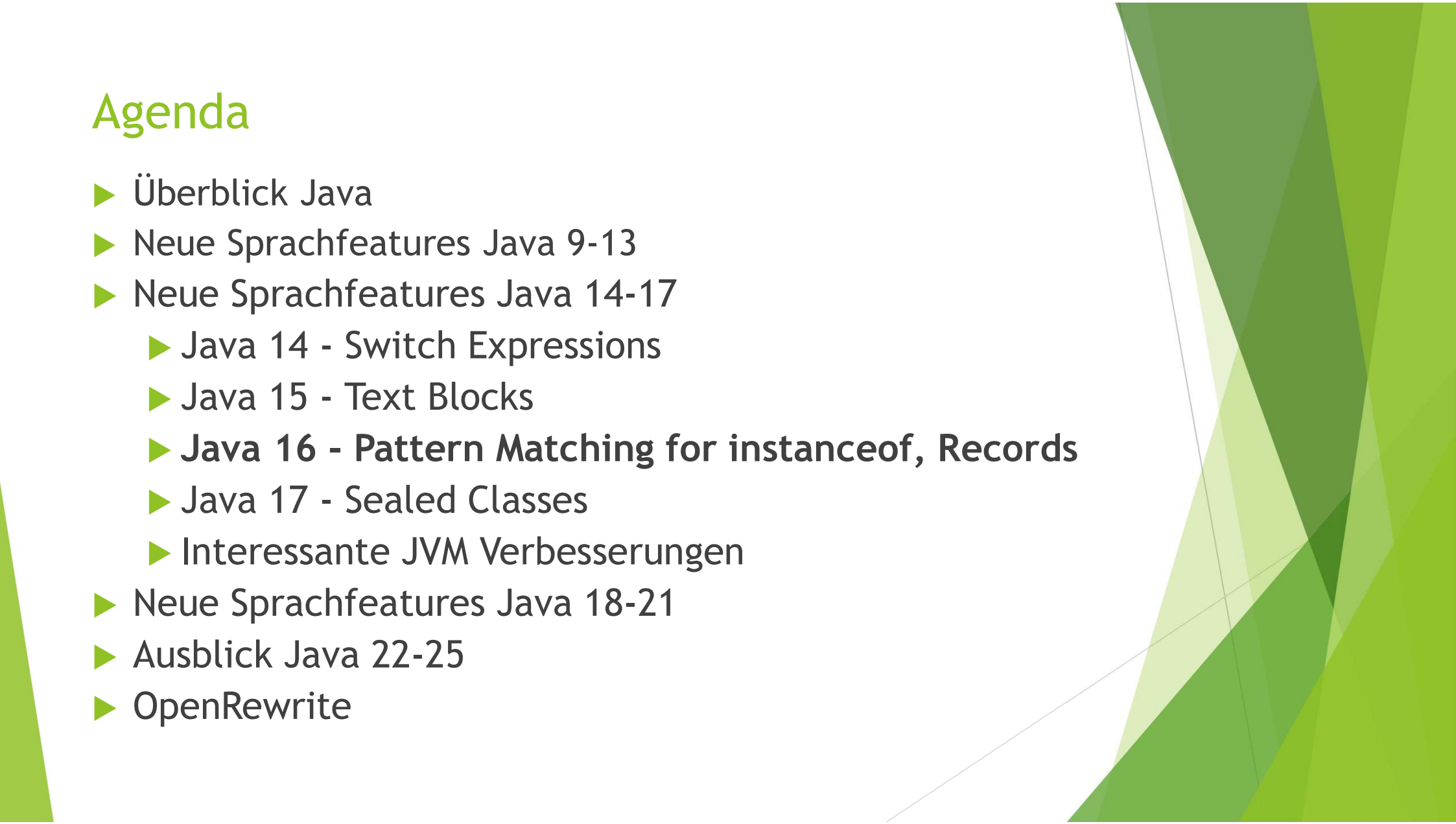
Finally, this string is being split using the new line character. String's split method returns an array of Strings. But the split method does not include trailing empty strings in the resulting array. That is why, although the string pointed to by s1 ends with a new line and thus contains 3 lines, the split method returns only 2 strings - "a b \t" and "c \s". Thus, s1.split("\\n").length returns 2.

Aufgaben

- Bitte alle TODOs in 05_java14-17 package de.zettsystems.java14_15.exercises14_15 lösen

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
  - Java 14 - Switch Expressions
  - Java 15 - Text Blocks
  - **Java 16 - Pattern Matching for instanceof, Records**
  - Java 17 - Sealed Classes
  - Interessante JVM Verbesserungen
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite



Java 16 – 16.03.2021

1. New Language Features

- JEP 394 Pattern Matching for instanceof
- JEP 395 Records
- Add Stream.toList() Method

2. JVM Improvements

- JEP 376 ZGC Concurrent Thread Processing
- JEP 387 Elastic Metaspace
- Concurrently Uncommit Memory in G1

3. New Tools and Libraries

- JEP 380 Unix-Domain Socket Channels
- JEP 392 Packaging Tool jpackage

4. Miscellaneous

- JEP 390 Warning for Value-Based Classes
- JEP 396 Strongly Encapsulate JDK Internals by default (flag)

5. Incubator and Preview Features

- JEP 338 Vector API (Incubator)
- JEP 389 Foreign Linker API (Incubator)
- JEP 393 Foreign Memory Access API (3rd Incubator)
- JEP 397 Sealed Classes (2nd Preview)

Pattern Matching inctanceof (Java 16)

- Bei der Typprüfung mit instanceof können wir nun direct eine Variable des entsprechenden Typs initialisieren
- Das spart den vorher üblichen Cast, der üblicherweise immer erfolgte
- Diese Variable können wir darüber hinaus direkt weiter in der if-Anweisung verwenden
- Aber ACHTUNG! – hier wird “Flow Scope” verwendet:
  - eine Pattern Matching-Variable ist nur dann bekannt, wenn der instanceof-Test bestanden wurde – nach der if-Anweisung, z.B. in einem else-Zweig ist sie nicht bekannt

Pattern Matching inctanceof Code

- 05_java14-17 de.zettsystems.java16_17.pminstanceof

Quiz

Given:

```
public class TestClass
{
  public static void main(String args[])
  {
    B b = new C();
    A a = b;
    if (a instanceof B b1) b1.b();
    if (a instanceof C c1) c1.c();
    if (a instanceof D d1) d1.d();
  }
}
class A {
    void a(){ System.out.println("a"); }
}
class B extends A {
    void b(){ System.out.println("b"); }
}
class C extends B {
    void c(){ System.out.println("c"); }
}
class D extends C {
    void d(){ System.out.println("d"); }
}
```

What is the result?

Select 1 option(s):

- Compilation failure.
- ClassCastException thrown at run time.
- a
  b
- b
  c
- b
  c
  d

Quiz

Given:

```
public class TestClass
{
  public static void main(String args[])
  {
    B b = new C();
    A a = b;
    if (a instanceof B b1) b1.b();
    if (a instanceof C c1) c1.c();
    if (a instanceof D d1) d1.d();
  }
}
class A {
    void a(){ System.out.println("a"); }
}
class B extends A {
    void b(){ System.out.println("b"); }
}
class C extends B {
    void c(){ System.out.println("c"); }
}
class D extends C {
    void d(){ System.out.println("d"); }
}
```

What is the result?

Select 1 option(s):

- Compilation failure.
- ClassCastException thrown at run time.
- a
  b
- b
  c
- b
  c
  d

D ist richtig

Records (Java 16)

- Ein neuer Grundtyp in Java (Neben Klasse, Interface, Annotation und Enum), um immutable Objekte zu erzeugen
- Immutable – Gültigkeit der Daten ohne synchronized sichergestellt:
  - private final field für jedes Datum
  - Getter für jedes Feld, aber kein Setter
  - öffentlicher Konstruktor mit einem entsprechenden Argument für jedes Feld
- Bisher mit recht viel Boilerplate oder etwa durch Lombok @Value herstellbar
- Die „getter“ von Records heißen wie die Attribute, also ohne get/is
- Man kann weitere Konstruktoren mit unterschiedlichen Parameterlisten ergänzen
- Man kann den Standard-Konstruktor (compact constructor) anpassen – das ist gedacht, um Validierungen einzufügen
- Statische Variablen und Methoden sind genauso möglich wie in classes
- Keine Vererbung, record implizit final
- … aber Implementierung von Interfaces möglich

Records und Stream.toList() Code

- 05_java14-17 de.zettsystems.java16_17.record und de.zettsystems.java16_17.misc

Quiz

Identify correct statement(s) about Java records.

Select 2 option(s):

- [ ] Records are implicitly final.
- [ ] Records are implicitly static.
- [ ] Records are meant to restrict the number of instances of that type.
- [ ] Records are implicitly sealed.
- [ ] Record instances are immutable.

Quiz

Identify correct statement(s) about Java records.

Select 2 option(s):

- [ ] Records are implicitly final.
- [ ] Records are implicitly static.
- [ ] Records are meant to restrict the number of instances of that type.
- [ ] Records are implicitly sealed.
- [ ] Record instances are immutable.

A,E sind richtig

Quiz 2

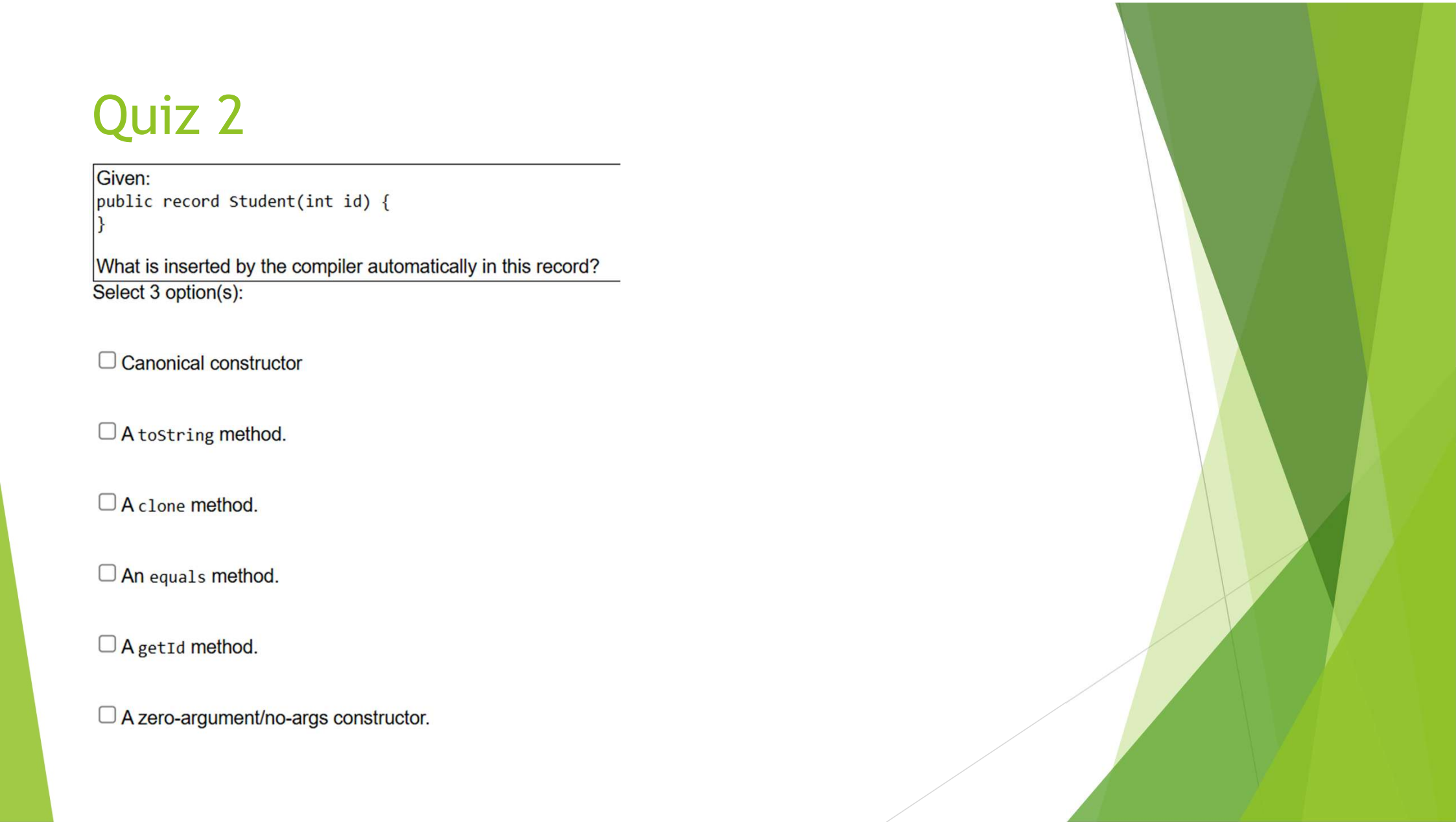
Given:

```
public record Student(int id) {
}
```

What is inserted by the compiler automatically in this record?

Select 3 option(s):

- [ ] Canonical constructor
- [ ] A `toString` method.
- [ ] A `clone` method.
- [ ] An `equals` method.
- [ ] A `getId` method.
- [ ] A zero-argument/no-args constructor.



Quiz 2

Given:

```
public record Student(int id) {
}
```

What is inserted by the compiler automatically in this record?

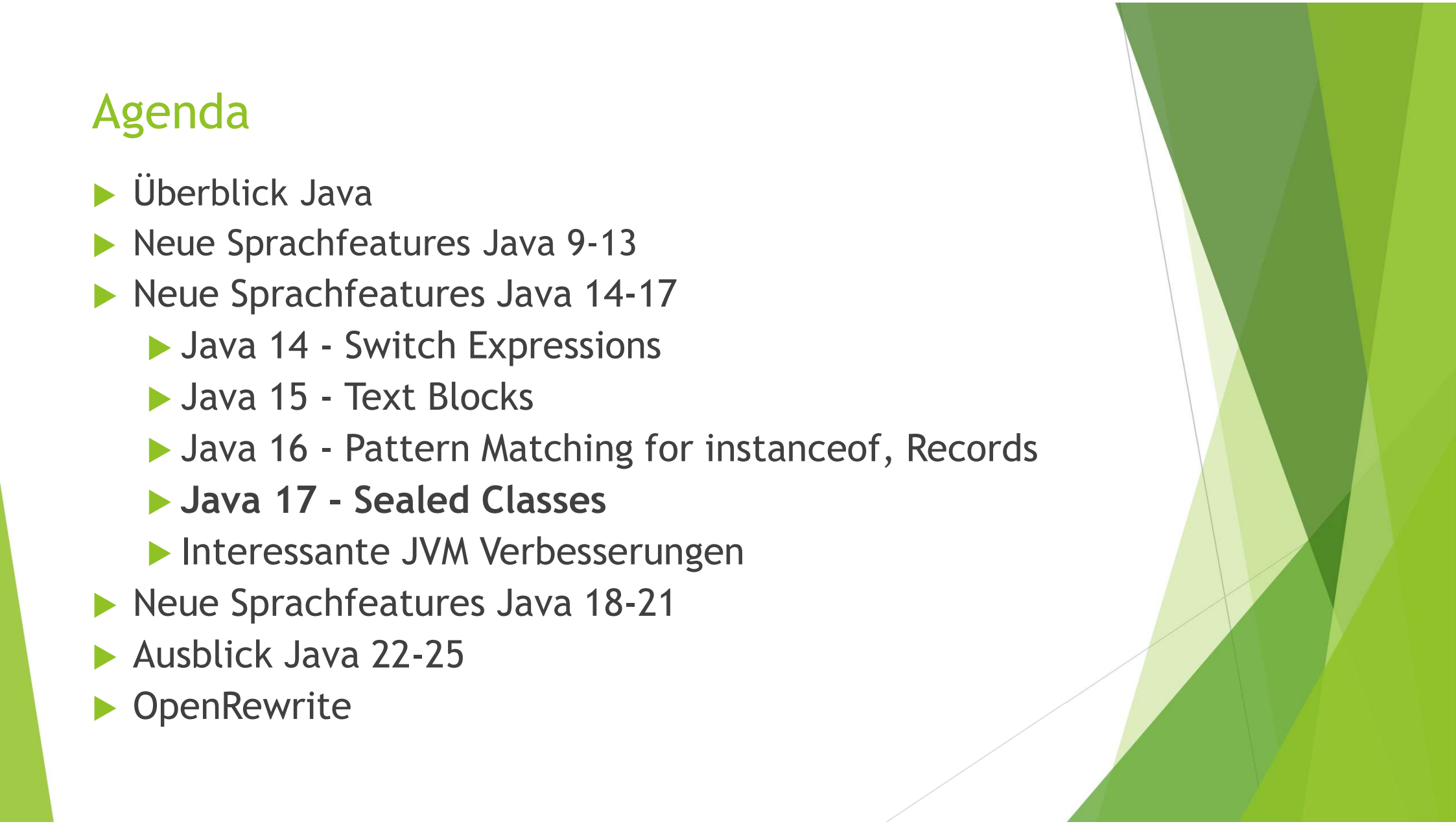
Select 3 option(s):

- [ ] Canonical constructor
- [ ] A `toString` method.
- [ ] A `clone` method.
- [ ] An `equals` method.
- [ ] A `getId` method.
- [ ] A zero-argument/no-args constructor.

A, B, D sind richtig

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
  - Java 14 - Switch Expressions
  - Java 15 - Text Blocks
  - Java 16 - Pattern Matching for instanceof, Records
  - **Java 17 - Sealed Classes**
  - Interessante JVM Verbesserungen
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite



Java 17 – 14.09.2021 (LTS bis 2029)

1. New Language Features

- JEP 409: Sealed Classes released

2. JVM Improvements

- ---

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- JEP 306: Restore Always-Strict Floating-Point Semantics
- JEP 356: Enhanced Pseudo-Random Number Generators
- JEP 403: Strongly Encapsulate JDK Internals (flag weg)

5. Incubator and Preview Features

- JEP 406: Pattern Matching for switch (Preview)
- JEP 412: Foreign Function & Memory API (Incubator)
- JEP 414: Vector API (Second Incubator)

Sealed Classes (Java 17)

- Sealed classes sind wie z. B. Enums zum Erfassen von Alternativen in Domänenmodellen geeignet
- Sie ermöglichen es Programmierern und Compilern, Schlussfolgerungen über die Abzählbarkeit/Vollständigkeit zu ziehen
- Arbeiten mit Records und Pattern Matching zusammen, um eine stärker datenzentrierte Programmierung zu unterstützen
- Sealed classes erlauben die Vererbung explizit
- Erben selber müssen im gleichen package liegen (oder gleichem Modul) und auch wieder sealed, final oder explizit non-sealed sein
- Interfaces können auch sealed sein
- Records können auch sealed interfaces implementieren

Sealed Classes 2

- Sealed classes sind weiterhin eine Möglichkeit, die Erweiterbarkeit von Superklassen einzuschränken, ohne dafür auch die Benutzbarkeit einzuschränken
- Bislang konnten wir:
  - Klasse public final machen -> keine Erweiterbarkeit möglich
  - Klasse package-private machen -> keine Benutzbarkeit außerhalb des Pakets, Erweiterbarkeit im gleichen Paket
- Mit sealed Klassen können wir nun eine Liste von erlaubten Subklassen festlegen
  - Erweiterbarkeit ist möglich wie designed, Benutzbarkeit der super class ist nicht eingeschränkt
- Dieser Aspekt ist für framework/library Entwickler interessant

Sealed Classes Code

- 05_java14-17 de.zettsystems.java16_17.sealed

Quiz

Given the following code:

```
//in file A.java
package p1;
public sealed class A permits p2.B{
}

//in file B.java
package p2;
public final class B extends p1.A{
}
```

Identify correct statements.

Select 1 option(s):

- ○ The code will compile fine if both A and B are part of the same named module.
- ○ Both the classes must belong to the same package for the code to compile irrespective of whether they belong to the same module or not.
- ○ Both the classes will compile as long as they are packaged in the same jar.
- ○ Class B will compile if, instead of final, it is made non-sealed.

Quiz

Given the following code:

```
//in file A.java
package p1;
public sealed class A permits p2.B{
}

//in file B.java
package p2;
public final class B extends p1.A{
}
```

Identify correct statements.

Select 1 option(s):

A ist richtig

- ○ The code will compile fine if both A and B are part of the same named module.
- ○ Both the classes must belong to the same package for the code to compile irrespective of whether they belong to the same module or not.
- ○ Both the classes will compile as long as they are packaged in the same jar.
- ○ Class B will compile if, instead of final, it is made non-sealed.

Quiz 2

Given:

```
interface Identifier{
    int id();
}

sealed class Person
    permits Student  //LINE A
{
}

record Student(int id,
        String subject)

        extends Person //LINE B

        implements Identifier //LINE C
{
   public static final long serialVersionUID  = 1L; //LINE D

   private String name = "unknown"; //LINE E

   String name(){ return "unknown"; } //LINE F

   public int id(){ return id; } //LINE G
}
```

Which lines will cause compilation issues?

Select 3 option(s):

- ☐ LINE A
- ☐ LINE B
- ☐ LINE C
- ☐ LINE D
- ☐ LINE E
- ☐ LINE F
- ☐ LINE G

Quiz 2

Given:

```
interface Identifier{
    int id();
}

sealed class Person
    permits Student  //LINE A
{
}

record Student(int id,
        String subject)

        extends Person //LINE B

        implements Identifier //LINE C
{
   public static final long serialVersionUID  = 1L; //LINE D

   private String name = "unknown"; //LINE E

   String name(){ return "unknown"; } //LINE F

   public int id(){ return id; } //LINE G
}
```

Which lines will cause compilation issues?

Select 3 option(s):

- ☐ LINE A
- ☐ LINE B
- ☐ LINE C
- ☐ LINE D
- ☐ LINE E
- ☐ LINE F
- ☐ LINE G

A, B, E sind richtig

Aufgaben

- Bitte alle TODOs in 05_java14-17 package de.zettsystems.java16_17.exercises16_17 lösen

History revisited

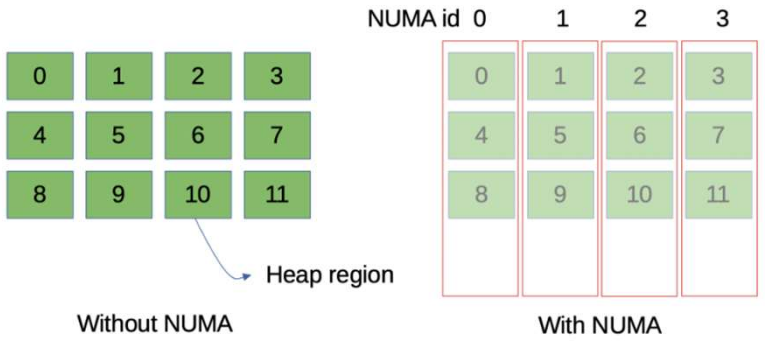
- 05_java14-17 de.zettsystems.history : Können wir das Eingangsbeispiel verbessern?

Agenda

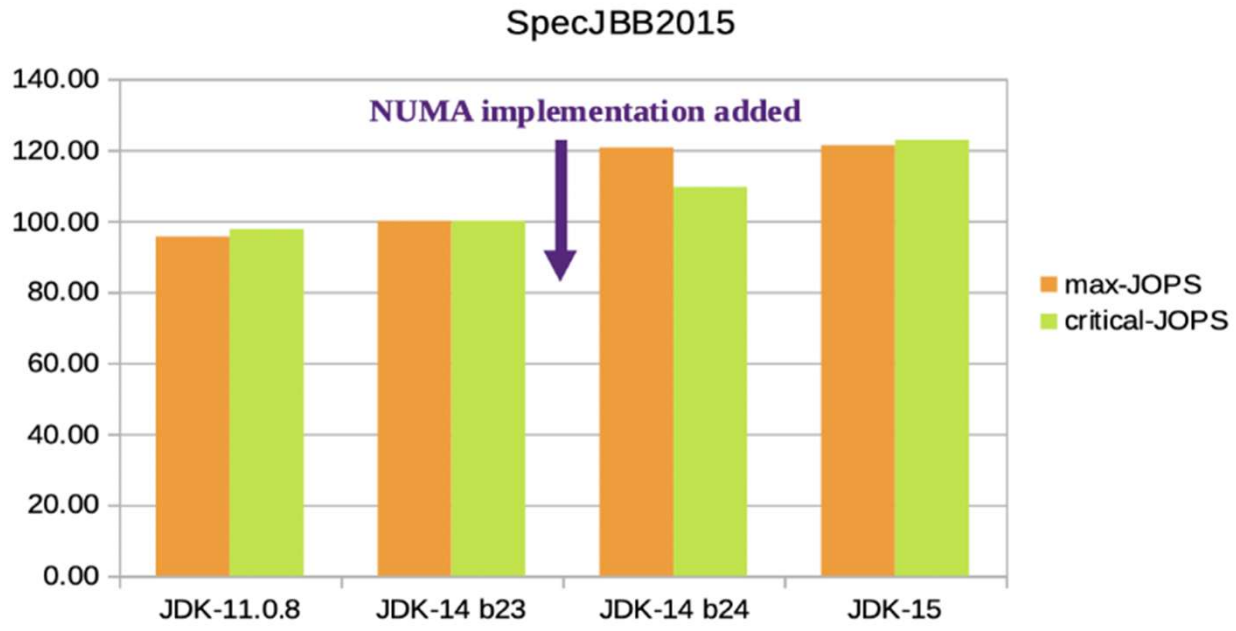
- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
  - Java 14 - Switch Expressions
  - Java 15 - Text Blocks
  - Java 16 - Pattern Matching for instanceof, Records
  - Java 17 - Sealed Classes
  - **Interessante JVM Verbesserungen**
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite

Java 14 – JVM Verbesserungen

- Parallel GC Improvements
  - Parallel GC hat denselben Task-Management-Mechanismus für die Planung paralleler Aufgaben übernommen wie andere GCs. Dies kann zu erheblichen Leistungsverbesserungen führen.
- NUMA-Aware Memory Allocation for G1
  - Moderne Multi-Socket-Maschinen haben zunehmend ungleichmäßigen Speicherzugriff (NUMA), d. h. der Speicher ist nicht von jedem Sockel oder Kern gleich weit entfernt. Der Parallel GC, der durch -XX:+UseParallelGC aktiviert wird, ist schon seit vielen Jahren NUMA-fähig. G1 hat dies nun auch mit -XX:+UseNUMA verfügbar



Java 14 – Numa Verbesserung



https://sangheon.github.io/2020/11/03/g1-numa.html

Java 15 – JVM Verbesserungen

- 2 neue GCs released
  - Shenandoah (nicht bei oracle dabei) und ZGC
  - Beide zielen auf Server mit großen Speicher (Terrabyte) und niedrige Latenzzeiten ab (kaum “stop-the-world”)
  - `java -XX:+UseZGC`
- Helpful NullPointerExceptions released
  - Es gibt mehr Infos, was denn genau null war
- DatagramSocket API re-implementiert
  - Nach socket-api nun diese 2. API, die auf virtual threads (Java 21) vorbereitet wurde

Java 15 - Hidden classes

- Hidden classes
  - Können nicht direkt vom bytecode anderer Klassen genutzt werden
  - Können nicht zum Deklarieren von Feldern, als Parameter, Return Wert oder Superclass benutzt werden
  - Können nicht durch classloader via Class::forName, ClassLoader::loadClass gefunden warden
  - Der Lebenszyklus kann feiner gesteuert werden als bei Anonymen Klassen
- Ein feature vor allem für Sprach/Framework/Tool-Entwickler
- Beispielsweise erzeugte der Compiler in Java für einen Lambda Ausdruck bytecode, um dynamische eine anonyme Klasse zu erzeugen – jetzt wird dafür eine hidden class genutzt
- Löst teilweise „sun.misc.Unsafe::defineAnonymousClass“ ab (und soll möglichst erweitert werden, um das ganz abzulösen)

Java 16 – JVM Verbesserungen

- JEP 376 ZGC Concurrent Thread Processing
  - Um stop-the-world weiter zu reduzieren (< 0.1ms) mussten weitere Prozessschritte in nebenläufige Phasen verschoben werden
- JEP 387 Elastic Metaspace
  - Rückgabe von ungenutztem Metaspace-Speicher an das OS, Verringerung des Metaspace-Footprints und Vereinfachung des Metaspace-Codes, um die Wartungskosten zu senken.
- Concurrently Uncommit Memory in G1
  - Diese neue Funktion ist immer aktiviert und ändert den Zeitpunkt, zu dem G1 Java-Heap-Speicher an das Betriebssystem zurückgibt. G1 trifft während der GC-Pause weiterhin Größenentscheidungen, verlagert aber die teure Arbeit auf einen Thread, der gleichzeitig mit der Java-Anwendung läuft

Java 16 – (Warning for) Value-Based Classes

- Das Projekt valhalla hat langfristig das Ziel „value objects“ einzuführen, das sind immutable Objekte ohne Identität
- Kandidaten dafür sind: die Wrapper-Klassen, Optional, Date/Time, collection implementation usw.
- Man möchte also die Performance von den primitiven Typen mit „richtigen“ Objekten kombinieren
- Nach einigen Vorarbeiten tief in der JVM wurden in diesem JEP nun die Kandidaten als @ValueObjects annotiert sowie deren public-Konstruktoren deprecated
- Neben der deprecation-Warning gibt es eine weitere Warning, wenn man diese Klassen unsachgemäß in synchronize verwendet

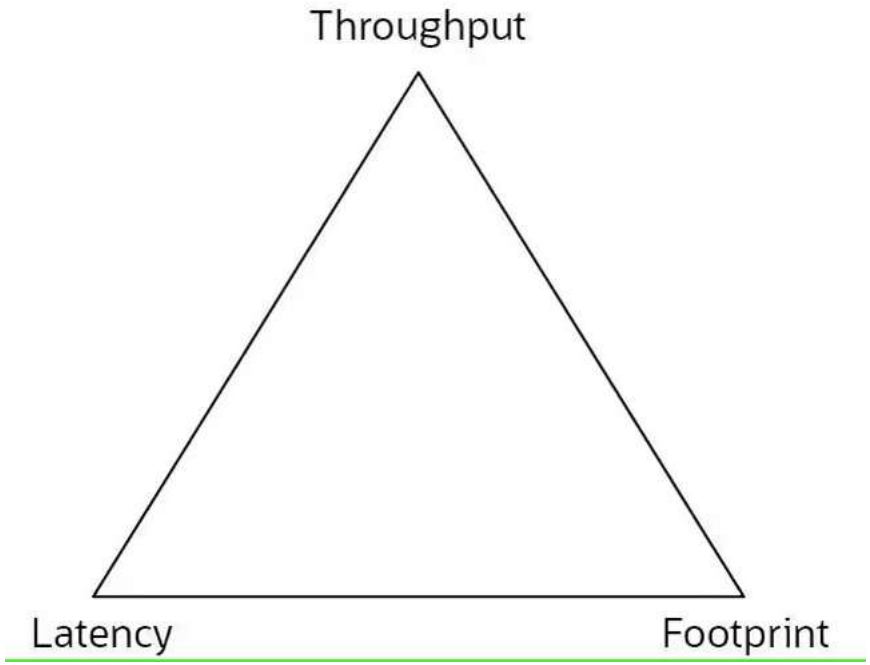
Java 16 – Showcase jpackage – Packaging Tool

- Endlich eine einfache Möglichkeit, Java Programme als ausführbare Dateien auszuliefern
- Windows, Linux, Mac werden unterstützt
- Die runtime wird mit ausgeliefert
  - Möchte man eine spezielle, optimierte Runtime (z.B. mit jlink erstellte), muss man die mit –runtime-image angeben
- Für Windows hat man eine ganz normale .exe Datei
  - Entweder als “richtige” .exe bzw. .msi Installer, dann muss man type exe nehmen, aber auch das WiX-tool im path haben (Version 3, jpackage kann mit 4 noch nicht umgehen)
  - Oder als starter-exe mit jar etc. dabei mit type app-image
- Optionen unterscheiden sich zwischen oldschool und modular
- Beispiel mit 05b_jpackage

Java 17 - Verschiedenes

- Restore Always-Strict Floating-Point Semantics:
  - In Java 2 (1998) wurde von strict-floating-point zu einem weniger strikten Verfahren gewechselt, weil die damaligen x87-Koprozessoren mit der strikten Variante nicht klar kamen
  - Seit Anfang der 2000er kam aber mit dem Pentium eine neue Architektur auf den Markt, die strikt konnte
  - Jetzt hat man das Verfahren wieder auf das strikte umgestellt
- Enhanced Pseudo-Random Number Generators:
  - Refactoring der vorhandenen Random-Klassen, um duplizierten Code zu entfernen
  - Streaming von Random Numbers ermöglichen
  - Umsetzung neuer, teils auch besserer Random-Algorithmen

GC Verbesserungen Ziele

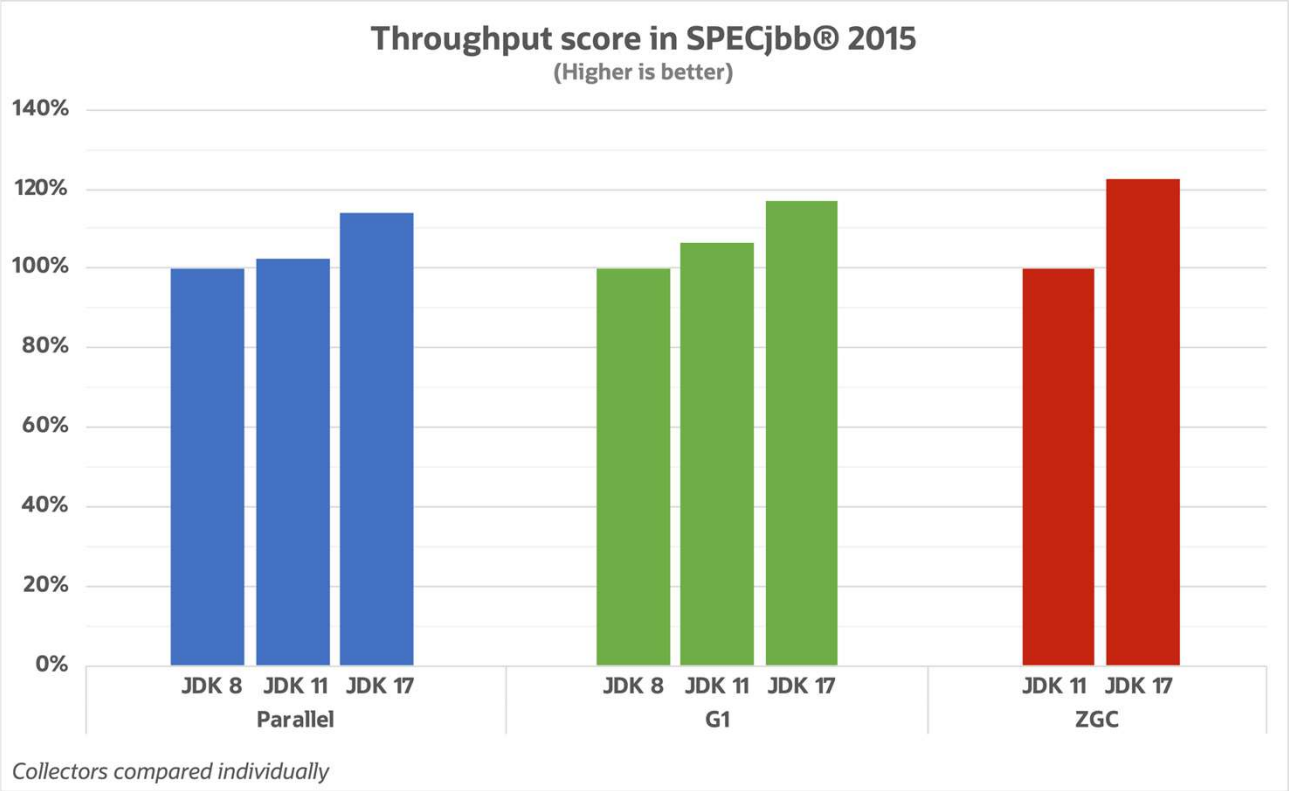


- Wir wollen hohen Throughput
- Bei niedriger Latenz und Footprint
- Unterschiedliche GC optimieren auf unterschiedliche Ziele

Relevante GCs im JDK

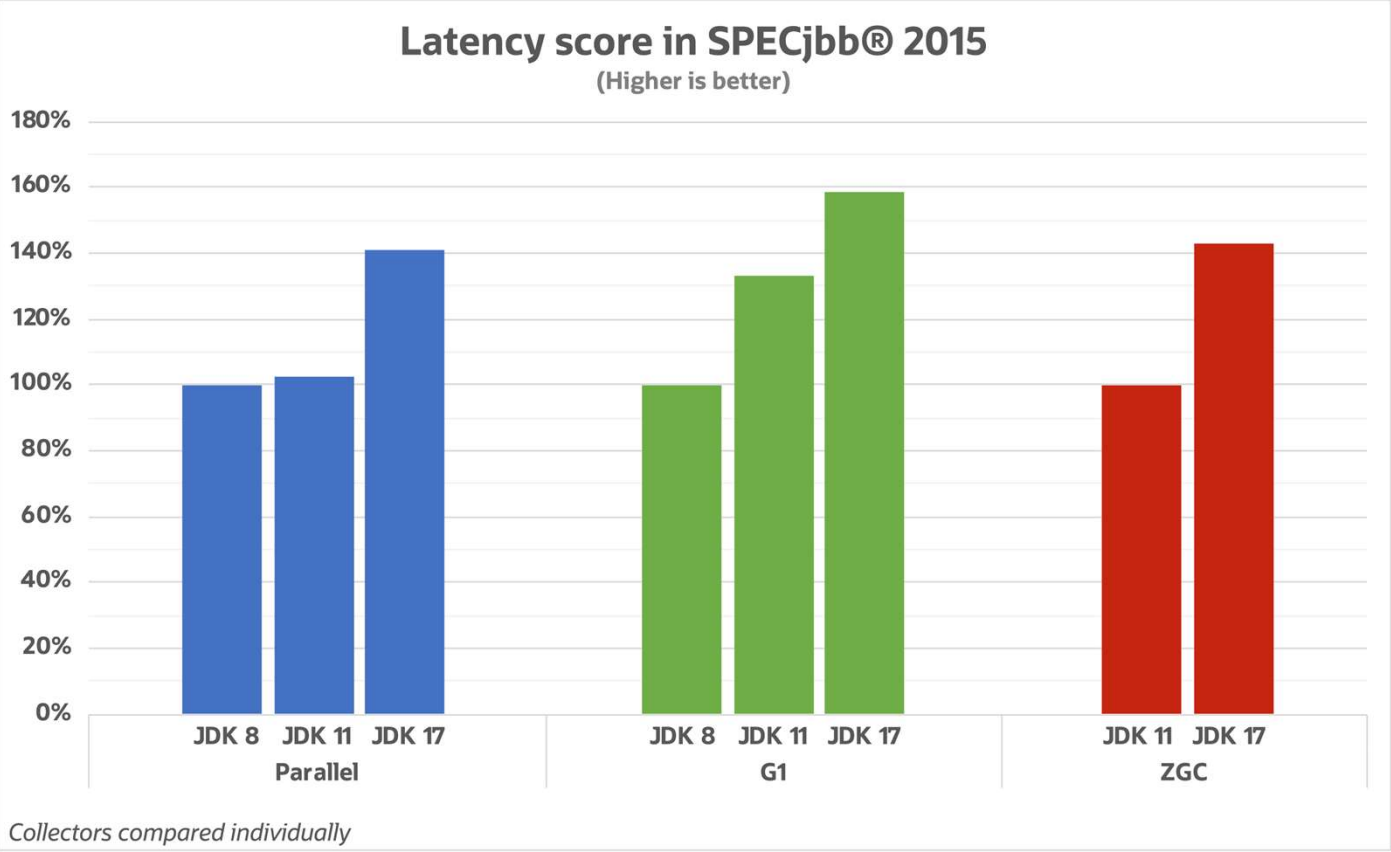
Name	Fokus	Konzept	Sonstiges
Parallel	Throughput	Multithreaded stop-the-world (STW) compaction and generational collection	Default bis JDK 8
G1	Balanced	Multithreaded STW compaction, concurrent liveness, and generational collection	Default ab JDK 9
ZGC	Latency	Everything concurrent to the application	Seit JDK 15 production-ready
Serial	Footprint and startup time	Single-threaded STW compaction and generational collection	Für Client Anwendungen

Throughput



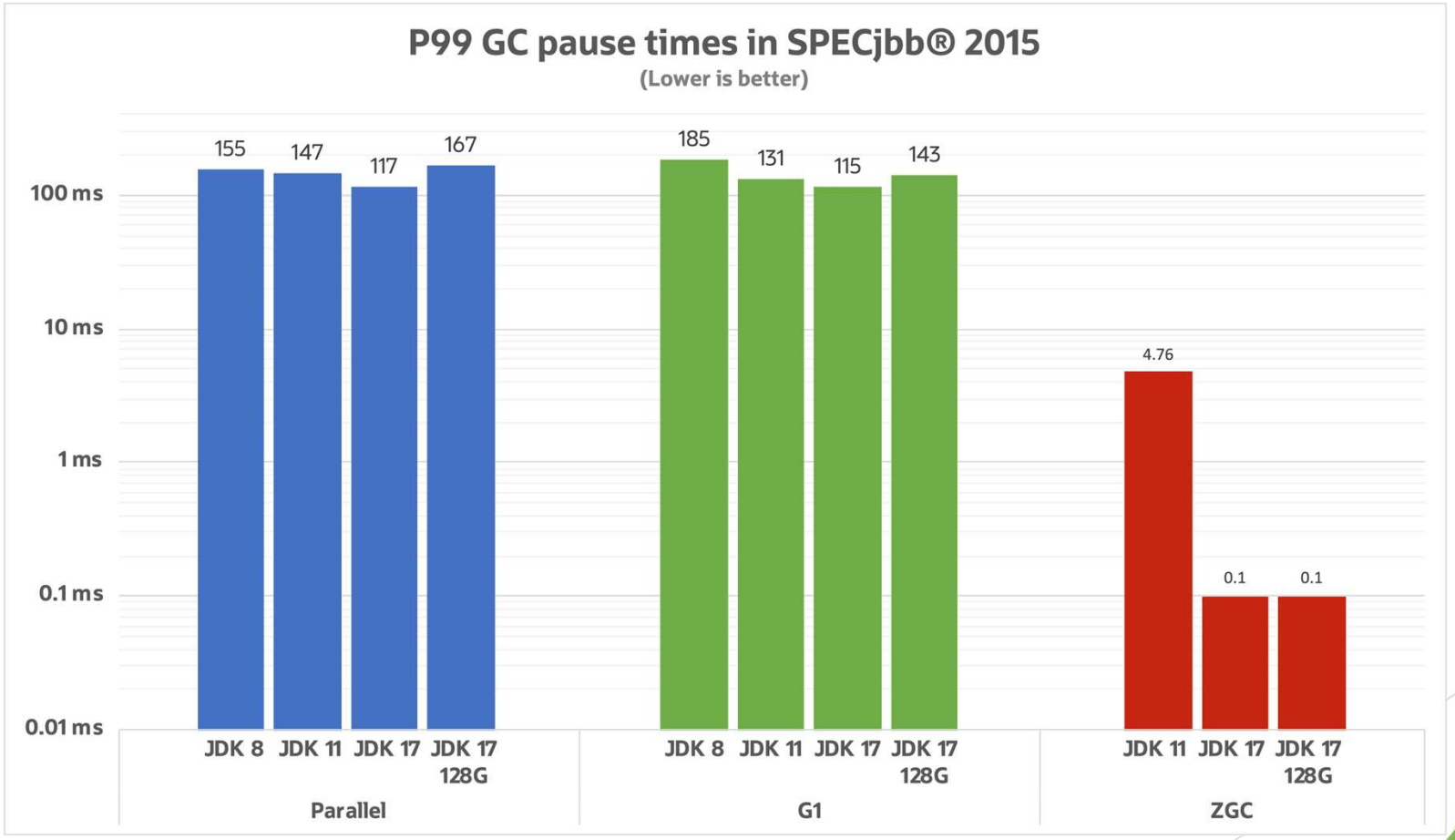
Collectors compared individually

Latency

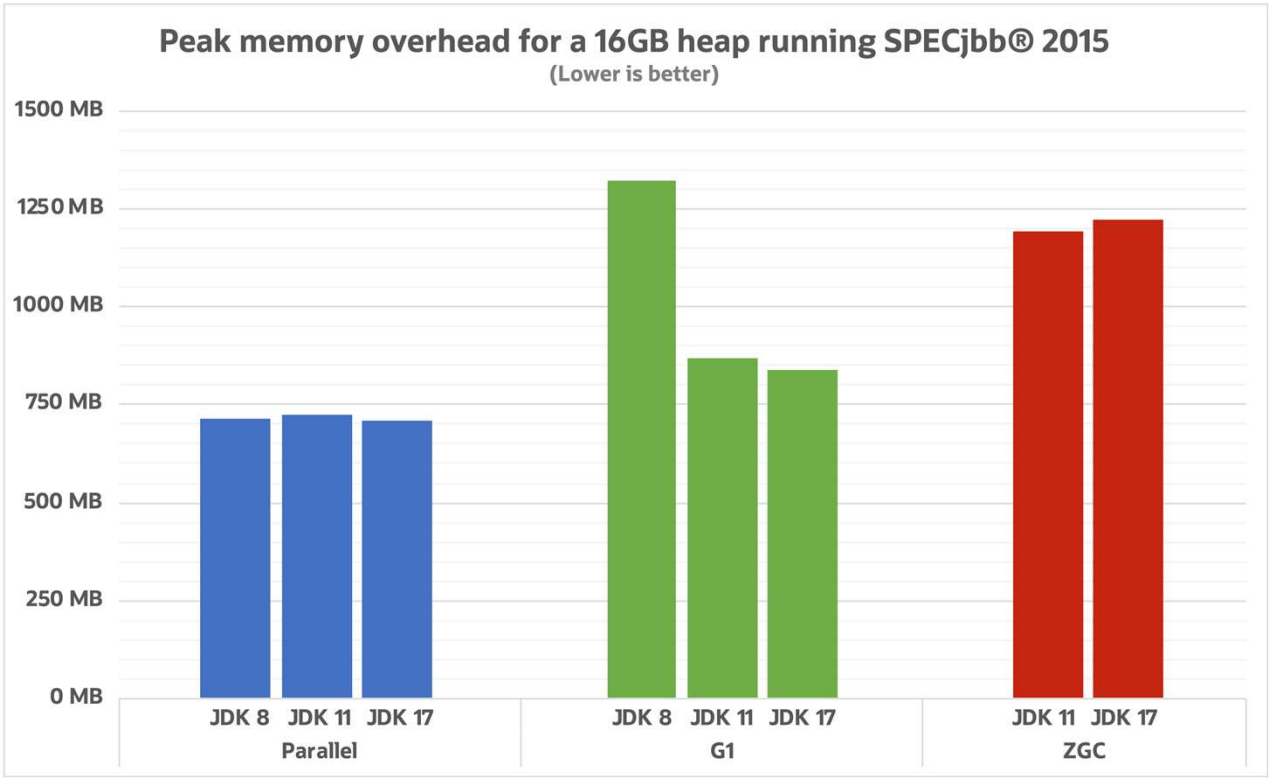


Collectors compared individually

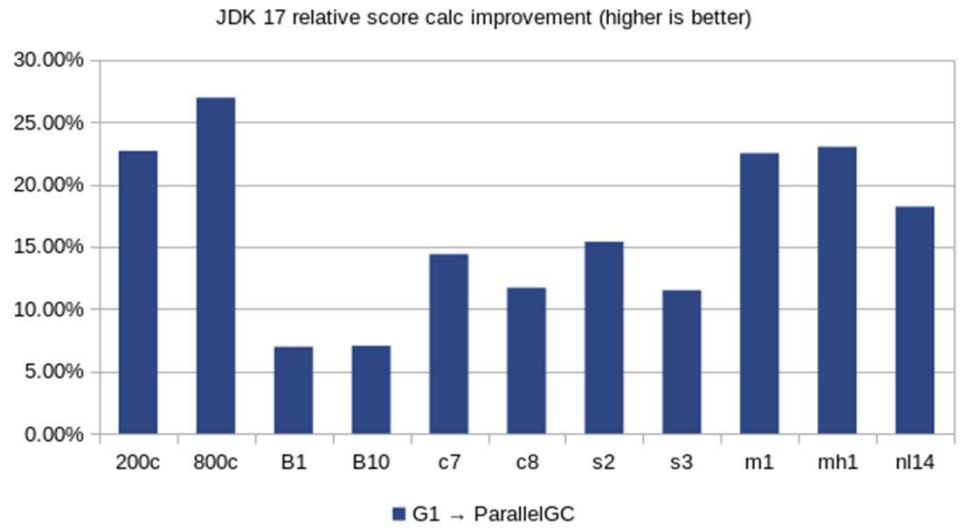
99p Latency


P99 GC pause times in SPECjbb® 2015
(Lower is better)
100 ms
10 ms
1 ms
0.1 ms
0.01 ms
155
147
117
167
185
131
115
143
4.76
0.1
0.1
JDK 8
JDK 11
JDK 17
JDK 17 128G
Parallel
JDK 8
JDK 11
JDK 17
JDK 17 128G
G1
JDK 11
JDK 17
JDK 17 128G
ZGC

Footprint


Peak memory overhead for a 16GB heap running SPECjbb® 2015
(Lower is better)
1500 MB
1250 MB
1000 MB
750 MB
500 MB
250 MB
0 MB
JDK 8
JDK 11
JDK 17
Parallel
JDK 8
JDK 11
JDK 17
G1
JDK 11
JDK 17
ZGC

Vergleich Parallel vs. G1 (optaplanner)



Executive summary

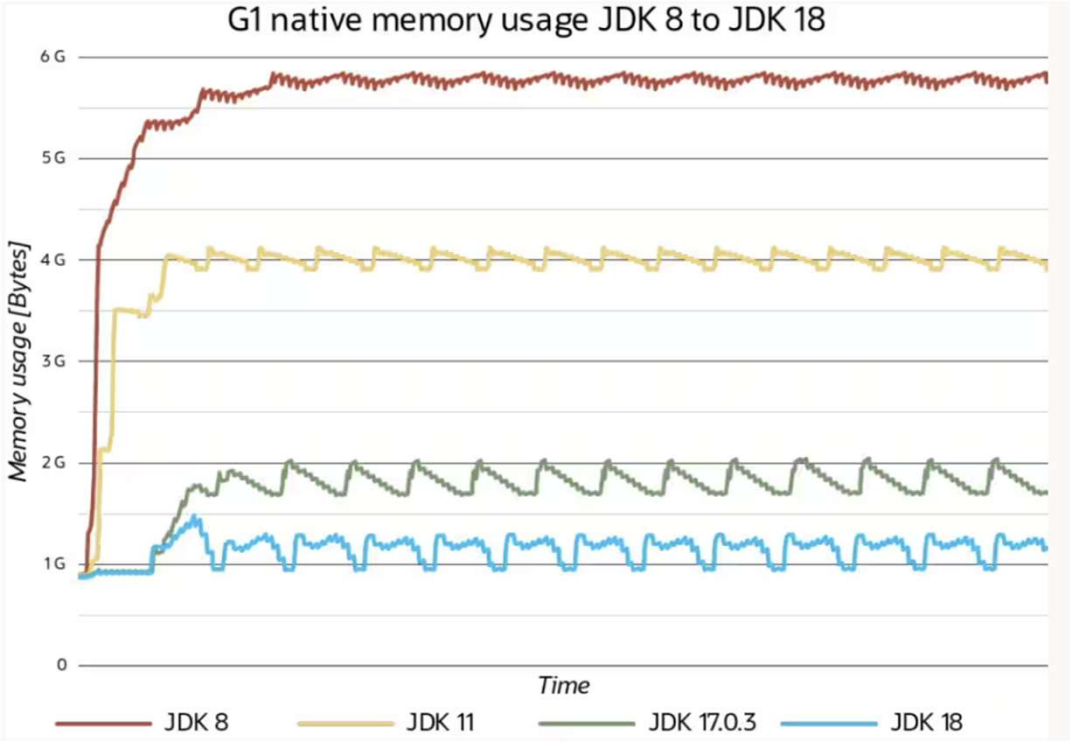
On average, for OptaPlanner use cases, these benchmarks indicate that:

- **Java 17 is 8.66% faster than Java 11** and 2.41% faster than Java 16 for G1GC (default).
- Java 17 is 6.54% faster than Java 11 and 0.37% faster than Java 16 for ParallelGC.
- The Parallel Garbage Collector is 16.39% faster than the G1 Garbage Collector.

	Average	Cloud balancing		Machine reassignment		Course scheduling		Exam scheduling		Nurse rostering		Traveling Tournament
Dataset		200c	800c	B1	B10	c7	c8	s2	s3	m1	mh1	nl14
G1GC		106,147	98,069	245,645	42,096	14,406	16,924	15,619	9,726	3,802	3,601	5,618
ParallelGC		130,215	124,498	262,753	45,058	16,479	18,904	18,023	10,845	4,658	4,430	6,641
G1 → ParallelGC	**16.39%**	22.67%	26.95%	6.96%	7.04%	14.39%	11.69%	15.39%	11.50%	22.50%	23.01%	18.20%

Table 3. Score calculation count per second on JDK 17 with different GCs

Java 18 – weitere Verbesserungen für G1

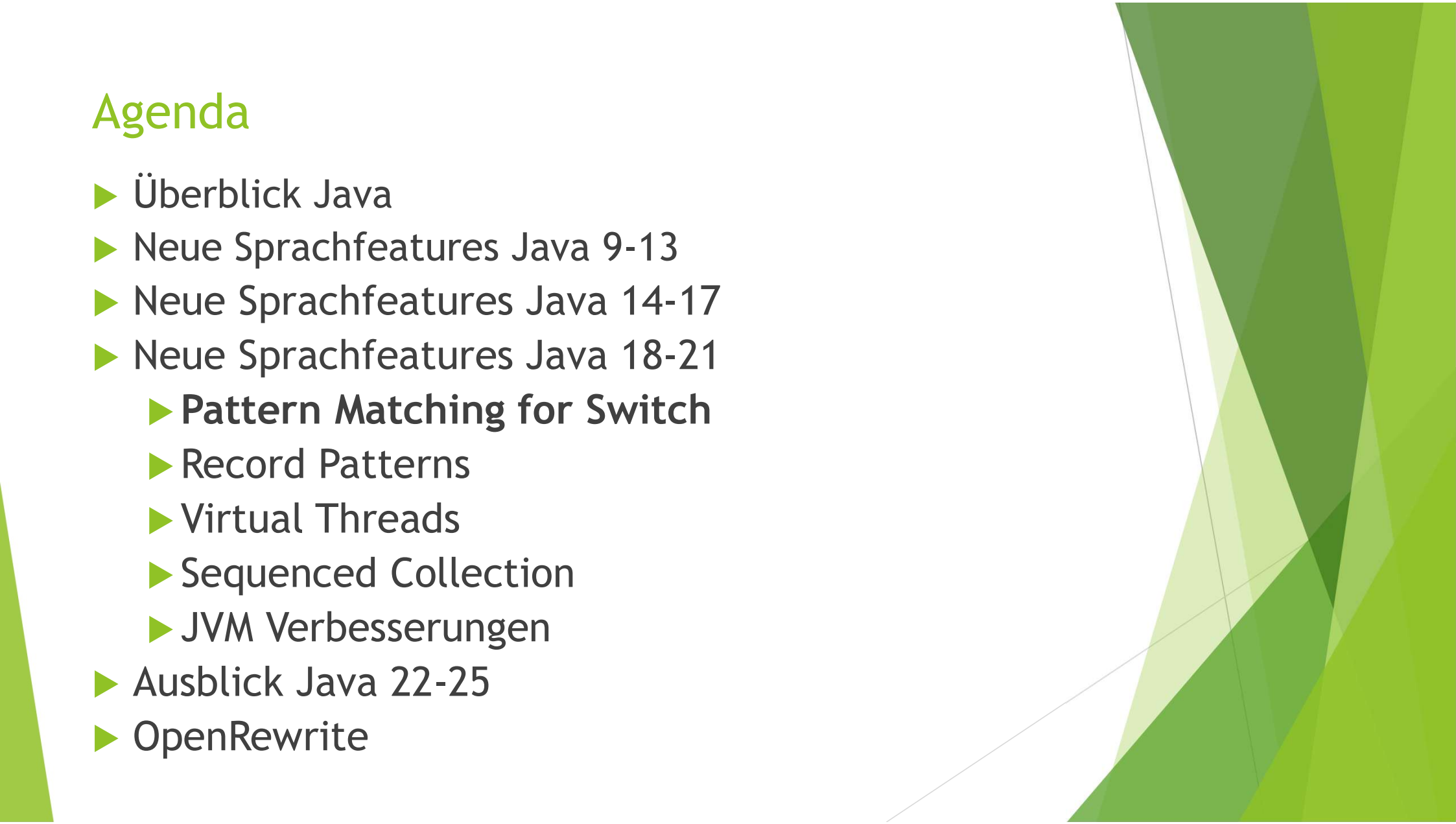


GC Verbesserungen links

- https://blogs.oracle.com/javamagazine/post/java-garbage-collectors-evolution
- https://kstefanj.github.io/2021/11/24/gc-progress-8-17.html
- https://www.optaplanner.org/blog/2021/09/15/HowMuchFasterIsJava17.html

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
  - **Pattern Matching for Switch**
  - Record Patterns
  - Virtual Threads
  - Sequenced Collection
  - JVM Verbesserungen
- Ausblick Java 22-25
- OpenRewrite



Java 18 – 22.03.2022

1. New Language Features

- ---

2. JVM Improvements

- JEP 416: Reimplement Core Reflection with Method Handles
- String Deduplication for Parallel, ZGC, SerialGC

3. New Tools and Libraries

- JEP 408 : Simple Web Server

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- JEP 400: UTF-8 by Default
- JEP 421: Deprecate Finalization for Removal
- JEP 413: Code Snippets in Java API Documentation

5. Incubator and Preview Features

- JEP 420: Pattern Matching for switch (2nd Preview)
- JEP 417: Vector API (3rd Incubator)

Java 19 – 20.09.2022

1. New Language Features

- ---

2. JVM Improvements

- ---

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- Linux/RISC-V port

5. Incubator and Preview Features

- JEP 427: Pattern Matching for switch (3rd Preview)
- JEP 424: Foreign Function & Memory API (Preview)
- JEP 426: Vector API (4th Incubator)
- JEP 405: Record Patterns (Preview)
- JEP 425: Virtual Threads(Preview)
- JEP 428: Structured Concurrency (Incubator)

Java 20 – 21.03.2023

1. New Language Features

2. JVM Improvements

- ---

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping

5. Incubator and Preview Features

- JEP 433: Pattern Matching for switch (4th Preview)
- JEP 434: Foreign Function & Memory API (2nd Preview)
- JEP 438: Vector API (5th Incubator)
- JEP 432: Record Patterns (2nd Preview)
- JEP 436: Virtual Threads(2nd Preview)
- JEP 437: Structured Concurrency (2nd Incubator)
- JEP 429: Scoped Values (Incubator)

Java 21 – 19.09.2023 (LTS bis 2031)

1. New Language Features

- JEP 441: Pattern Matching for switch
- JEP 440: Record Patterns
- JEP 431: Sequenced Collections
- JEP 444: Virtual Threads

2. JVM Improvements

- JEP 439 Generational ZGC
- JEP 444: Virtual Threads

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- JEP 451: Prepare to Disallow the Dynamic Loading of Agents

5. Incubator and Preview Features

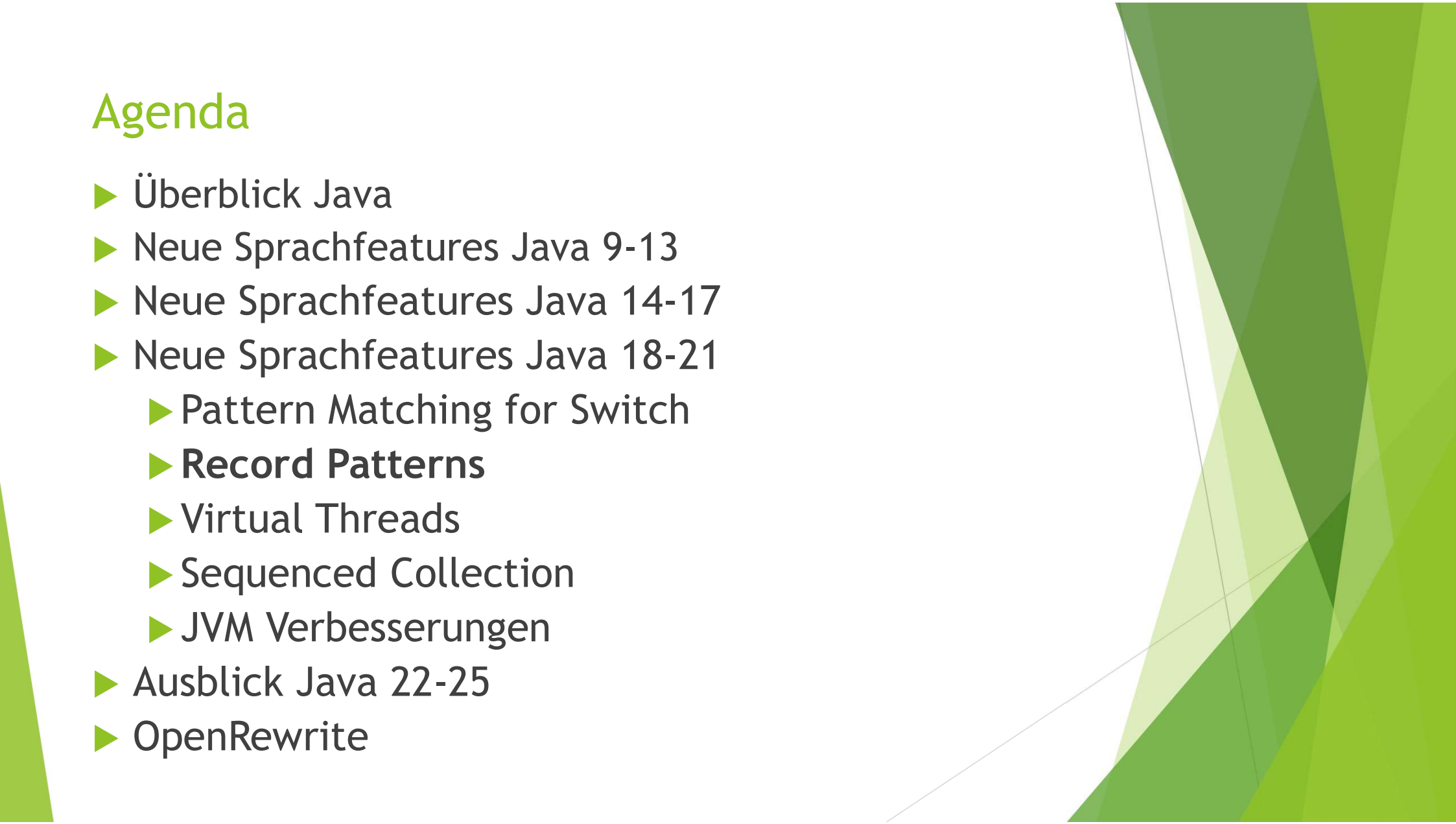
- JEP 442: Foreign Function & Memory API (3. Preview)
- JEP 443: Unnamed Patterns and Variables (Preview)
- JEP 445: Unnamed Classes and Instance Main Methods (Preview)
- JEP 446: Scoped Values (Preview)
- JEP 430: String Templates (Preview)
- JEP 448: Vector API (6. Incubator)
- JEP 453: Structured Concurrency (Preview)

Pattern Matching for switch (Java 21)

- Mit dem neuen Feature kann man nun ein Switch mit jedem Typ machen
- Die Typprüfung ist Teil des case
- Da alle möglichen Werte berücksichtigt werden müssen: default muss mit dabei sein (außer bei Sealed Classes Hierarchien)
- Genau wie bei instanceof kann man im Erfolgsfall weitere Prüfungen auf dem Typ vornehmen (Guarded Pattern)

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
  - Pattern Matching for Switch
  - **Record Patterns**
  - Virtual Threads
  - Sequenced Collection
  - JVM Verbesserungen
- Ausblick Java 22-25
- OpenRewrite



Record Patterns (Java 21)

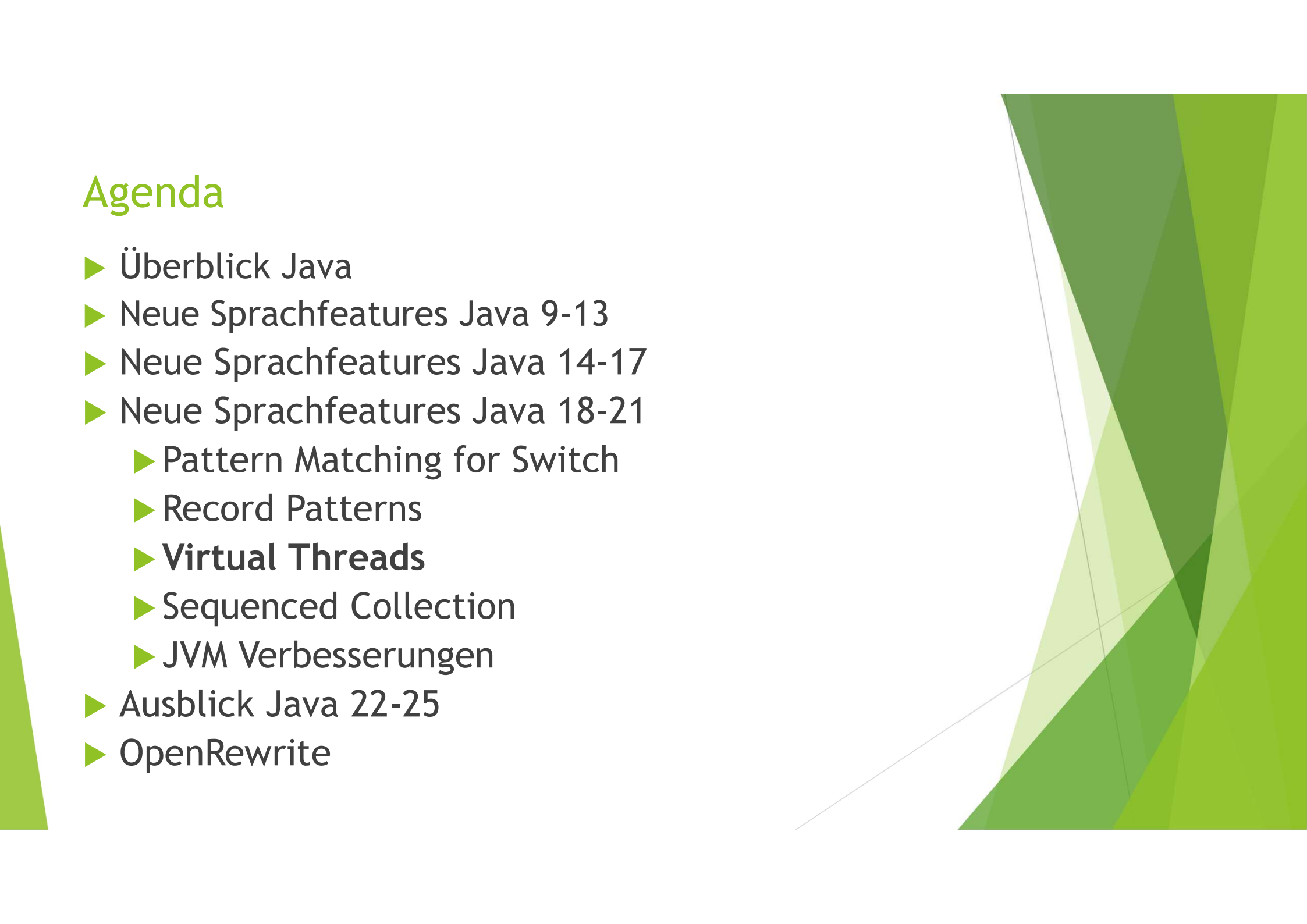
- „Zerlegung“ von Records beim Pattern Matching
- Wenn man nach dem „cast“ auf die Attribute des Records zugreifen möchte, gibt es jetzt eine Schreibweise, diese direkt als flow Variablen zu deklarieren

Pattern Matching for switch/ Record Pattern Code

- 06_java18_21\ de.zettsystems.java18_21.patternmatching

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
  - Pattern Matching for Switch
  - Record Patterns
  - **Virtual Threads**
  - Sequenced Collection
  - JVM Verbesserungen
- Ausblick Java 22-25
- OpenRewrite



Virtual Threads (Java 21)

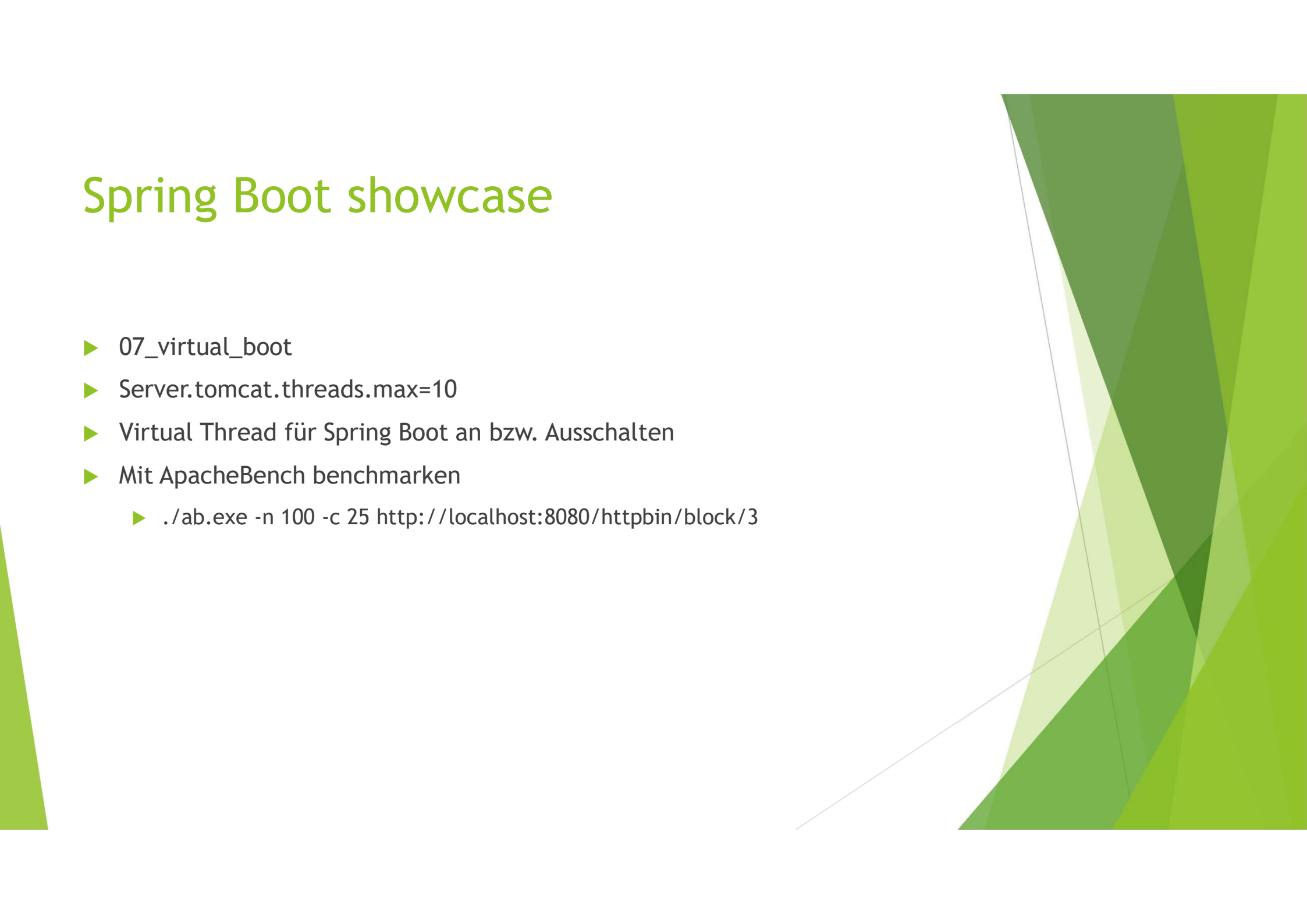
- Unsere bekannten Threads sind dünne Wrapper um einen Betriebssystem Prozess. Diesen nehmen sie sich und behalten ihn für ihre gesamte Lebenszeit, auch wenn sie blockieren und warten (z.B. auf Antwort von einem remote System). Diese threads werden (jetzt) „platform threads“ genannt.
- Die neuen virtual threads nehmen sich den nächstbesten freien OS-Prozess, wenn sie Arbeit erledigen. Wenn sie eine blockierende Aktion aufrufen, werden sie pausiert und der OS-Prozess, den sie gerade nutzten, wird wieder frei.
- → Die Anzahl an platform threads wird durch die Anzahl an OS-Prozessen limitiert. Sie sind eine wertvolle Ressource (Pooling). Virtual threads sind frei davon und daher gut für Applikationen geeignet, die viele Aktionen parallel auszuführen haben.

Virtual Threads Test

- 06_java18-21 test/java/de.zettsystems.java18_21.virtual

Spring Boot showcase

- 07_virtual_boot
- Server.tomcat.threads.max=10
- Virtual Thread für Spring Boot an bzw. Ausschalten
- Mit ApacheBench benchmarken
  - ./ab.exe -n 100 -c 25 http://localhost:8080/httpbin/block/3



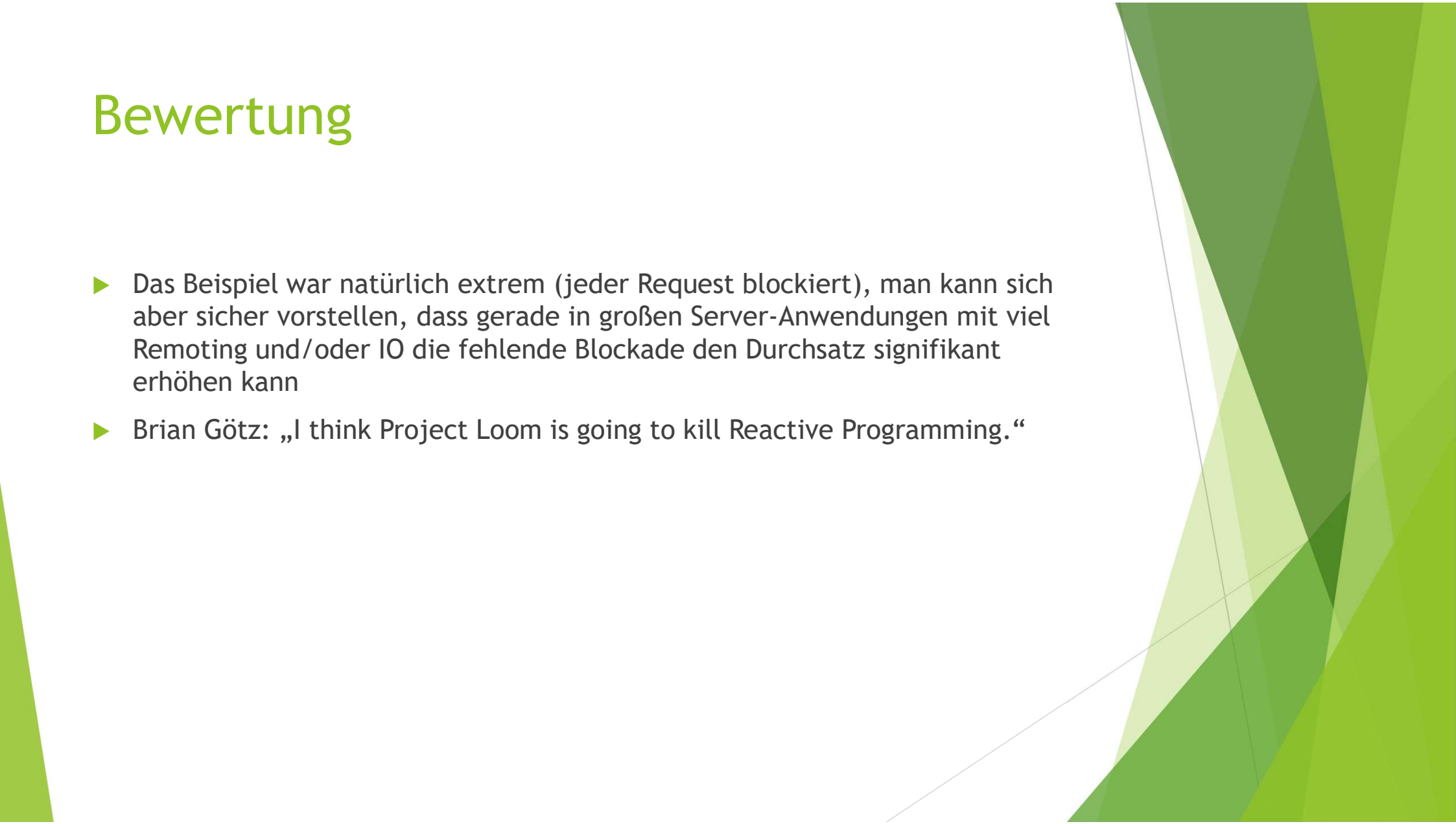
Platform vs. Virtual threads

```
Server Hostname:        localhost
Server Port:            8080

Document Path:          /httpbin/block/3
Document Length:        39 bytes

Concurrency Level:      25
Time taken for tests:   33.124 seconds
Complete requests:      100
Failed requests:        10
   (Connect: 0, Receive: 0, Length: 10, Exceptions: 0)
Total transferred:      17210 bytes
HTML transferred:       3910 bytes
Requests per second:    3.02 [#/sec] (mean)
Time per request:       8281.040 [ms] (mean)
Time per request:       331.242 [ms] (mean, across all concurrent requests)
Transfer rate:          0.51 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.5      0       1
Processing:  3009 6894 1972.3   6027    9041
Waiting:     3006 6892 1972.7   6024    9039
Total:       3010 6894 1972.2   6028    9041

Percentage of the requests served within a certain time (ms)
  50%   6028
  66%   9023
  75%   9026
  80%   9030
  90%   9035
  95%   9037
  98%   9039
  99%   9041
 100%   9041 (longest request)
```

```
Server Hostname:        localhost
Server Port:            8080

Document Path:          /httpbin/block/3
Document Length:        68 bytes

Concurrency Level:      25
Time taken for tests:   15.193 seconds
Complete requests:      100
Failed requests:        91
   (Connect: 0, Receive: 0, Length: 91, Exceptions: 0)
Total transferred:      20281 bytes
HTML transferred:       6981 bytes
Requests per second:    6.58 [#/sec] (mean)
Time per request:       3798.347 [ms] (mean)
Time per request:       151.934 [ms] (mean, across all concurrent requests)
Transfer rate:          1.30 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.5      0       1
Processing:  3006 3017  12.3   3018    3120
Waiting:     3003 3015  11.5   3016    3112
Total:       3006 3018  12.3   3019    3121

Percentage of the requests served within a certain time (ms)
  50%   3019
  66%   3020
  75%   3021
  80%   3022
  90%   3026
  95%   3027
  98%   3029
  99%   3121
 100%   3121 (longest request)
```

Bewertung

- Das Beispiel war natürlich extrem (jeder Request blockiert), man kann sich aber sicher vorstellen, dass gerade in großen Server-Anwendungen mit viel Remoting und/oder IO die fehlende Blockade den Durchsatz signifikant erhöhen kann
- Brian Götz: „I think Project Loom is going to kill Reactive Programming.“



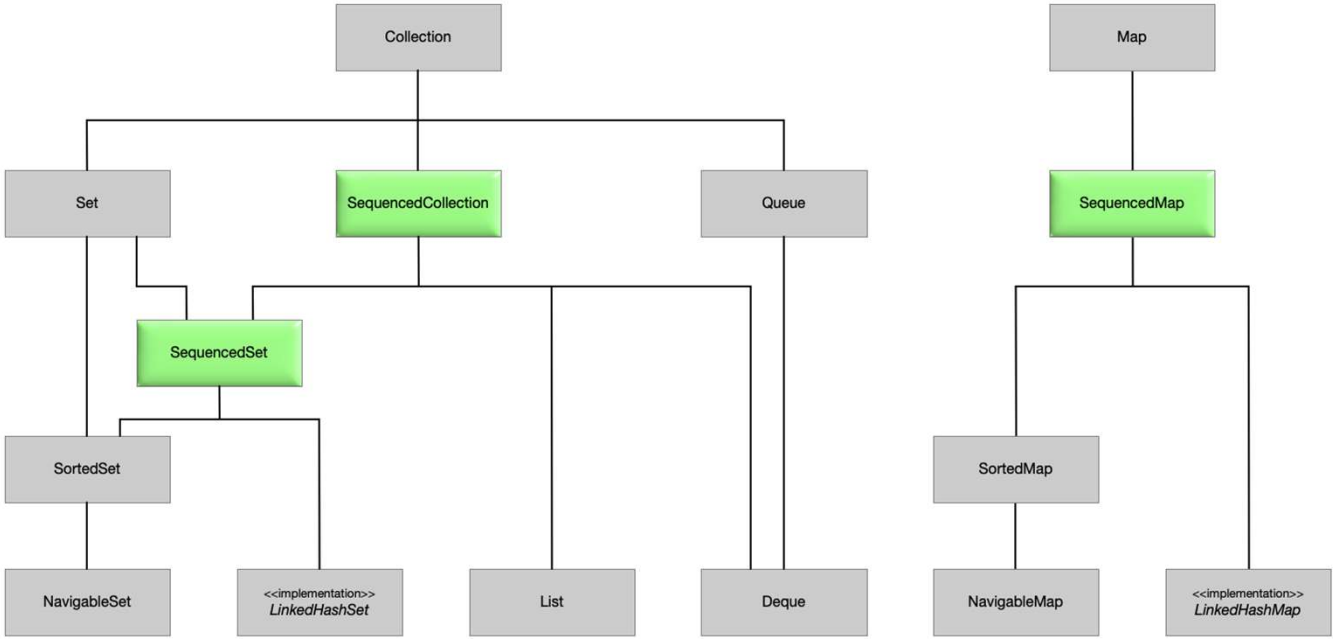
Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
  - Pattern Matching for Switch
  - Record Patterns
  - Virtual Threads
  - **Sequenced Collection**
  - JVM Verbesserungen
- Ausblick Java 22-25
- OpenRewrite

Sequenced Collections (Java 21)

- Neue interfaces in Collections, die geordnete Elemente darstellen
- Einheitlicher Zugriff auf 1. und letztes Element
- Einheitliches Durchgehen der Elemente vorwärts und rückwärts

Sortiert sich in alte Strukturen ein



Sequenced Collections JEP – Stuart Marks

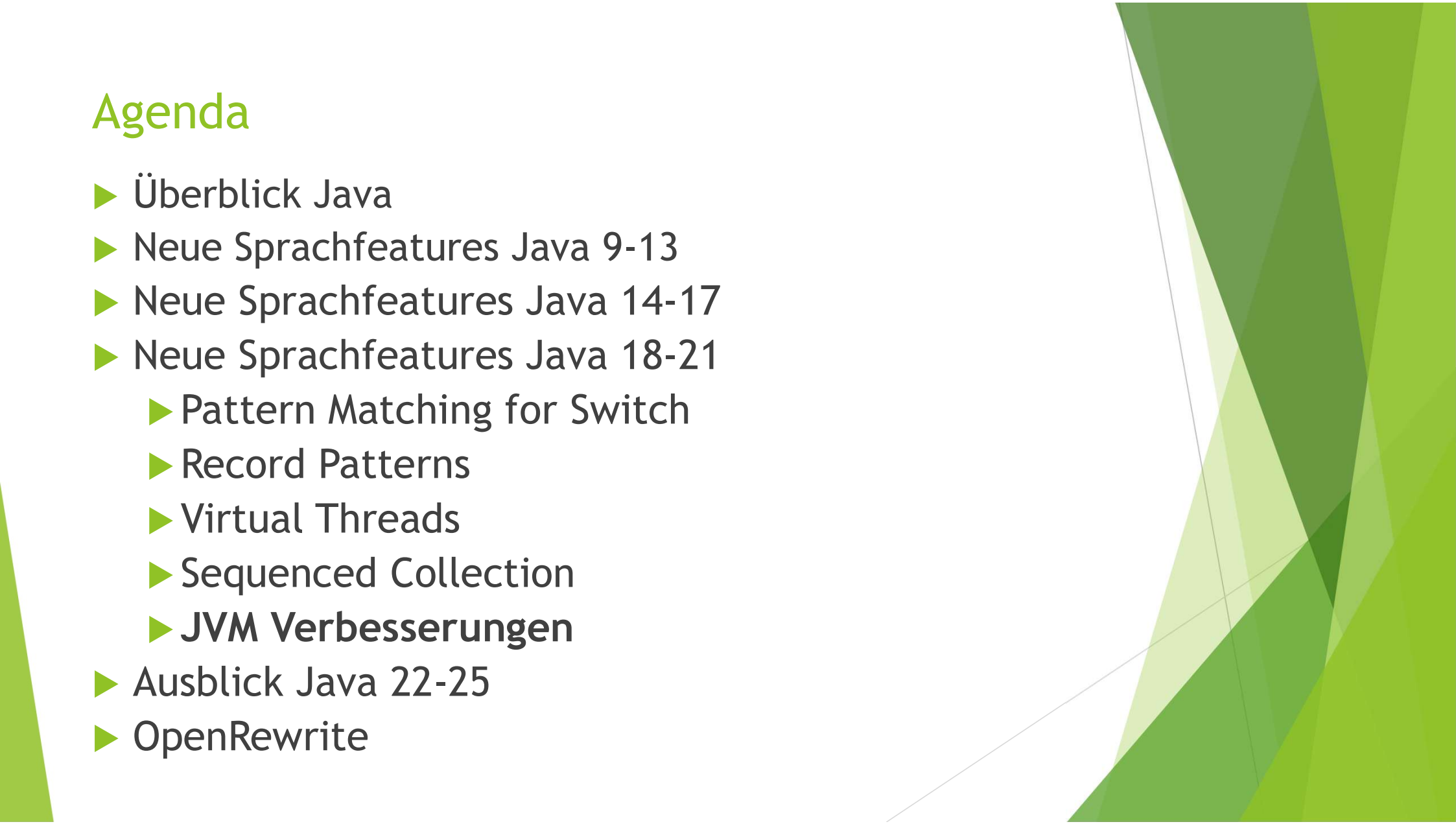
2022-02-16

Sequenced Collection Code

- 06_java18-21 de.zettsystems.java18_21.seqcol

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
  - Pattern Matching for Switch
  - Record Patterns
  - Virtual Threads
  - Sequenced Collection
  - **JVM Verbesserungen**
- Ausblick Java 22-25
- OpenRewrite



Java 18-21 – weitere Verbesserungen

- Parallel und Serial können nun auch -XX:+UseStringDeduplication benutzen
  - Es gibt eine große Menge doppelter String Objekte im Heap
  - Mit obiger Einstellung werden diese beim GC abgeräumt
    - Das gibt bis zu 13% Platz frei!
    - Das kostet aber auch Latenz, also es dauert
- ZGC ist nun auch ein Generational Garbage Collector
  - Generational GC bislang nur bei den anderen GC, reduziert die Zeit, da ältere Objekte nicht jedes Mal angeschaut werden
  - Muss man für Z extra einschalten in Java 21: -XX:+ZGenerational

Verschiedenes

- 06_java18-21 de.zettsystems.java18_21.history : Können wir das Eingangsbeispiel verbessern?

Aufgaben

- Bitte alle TODOs in 06_java_18_21 package de.zettsystems.java18_21.exercises lösen

OO vs. data-centric

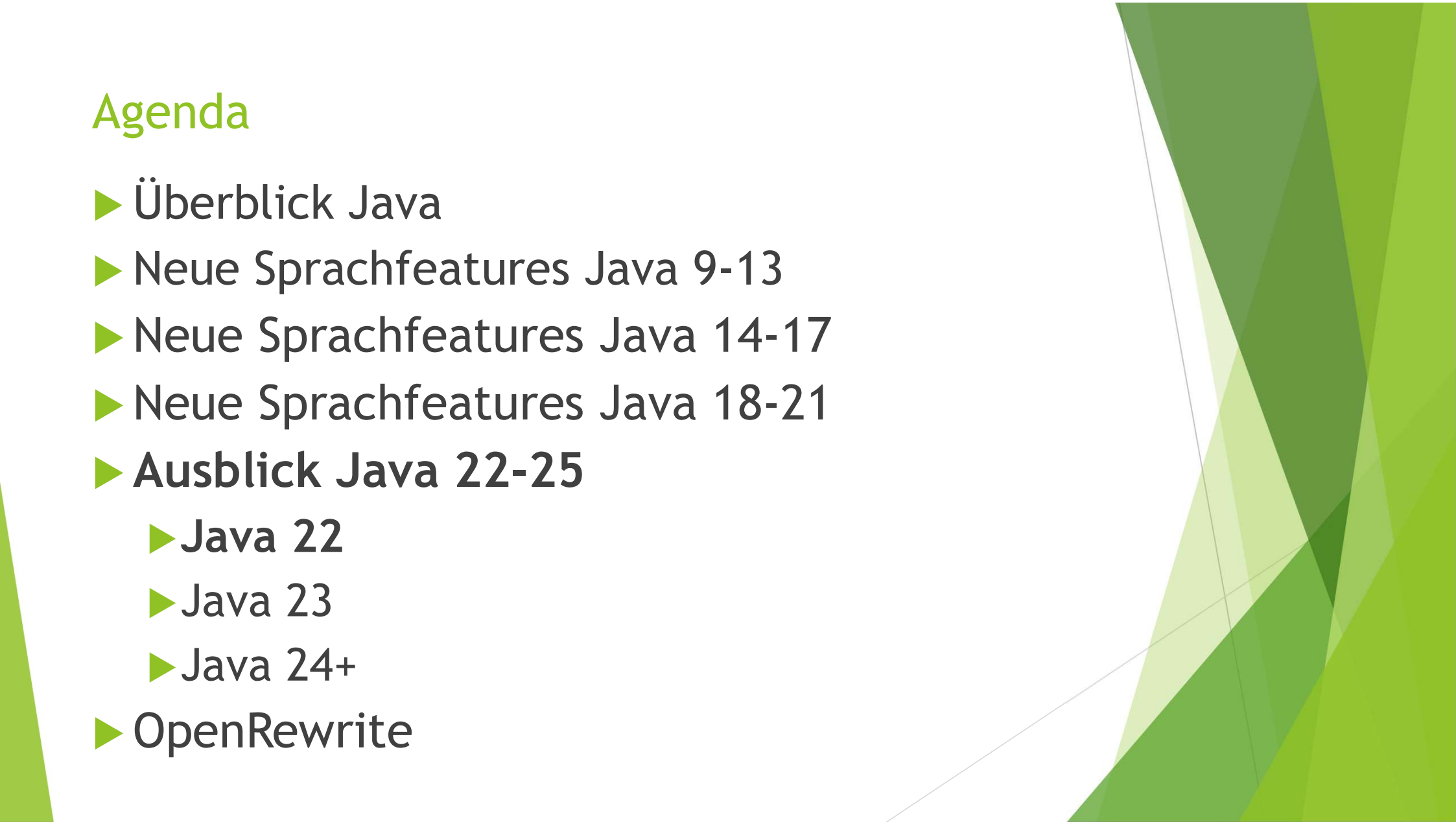
- Showcase OO

Warum nicht weiter OO sondern data-centric?

- In vielen Fällen könnte man die Logik genauso gut mit Vererbung und Polymorphie implementieren
- Trotzdem bietet Pattern Matching (insbesondere in Kombination mit sealed und record) für bestimmte Szenarien Vorteile:
  - etwa bei datenzentrierten Strukturen, bei denen man möglichst wenig „Business-Logik“ in den Klassen selbst haben möchte,
  - oder wenn man Klassen nicht ändern kann/darf.
- Ganz allgemein gilt: Ob man Polymorphie oder Pattern Matching verwendet, ist eine Gestaltungsfrage und hängt von den Anforderungen und dem Designstil (objektorientiert vs. algebraisch/funktional) ab.
- Oft kann man beides kombinieren.

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- **Ausblick Java 22-25**
  - **Java 22**
  - Java 23
  - Java 24+
- OpenRewrite



Java 22 – 19.03.2024

1. New Language Features

- JEP 456: Unnamed Variables & Patterns
- JEP 454: Foreign Function & Memory API

2. JVM Improvements

- JEP 423: Region Pinning for G1

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- JEP 458: Launch Multi-File Source-Code Programs

5. Incubator and Preview Features

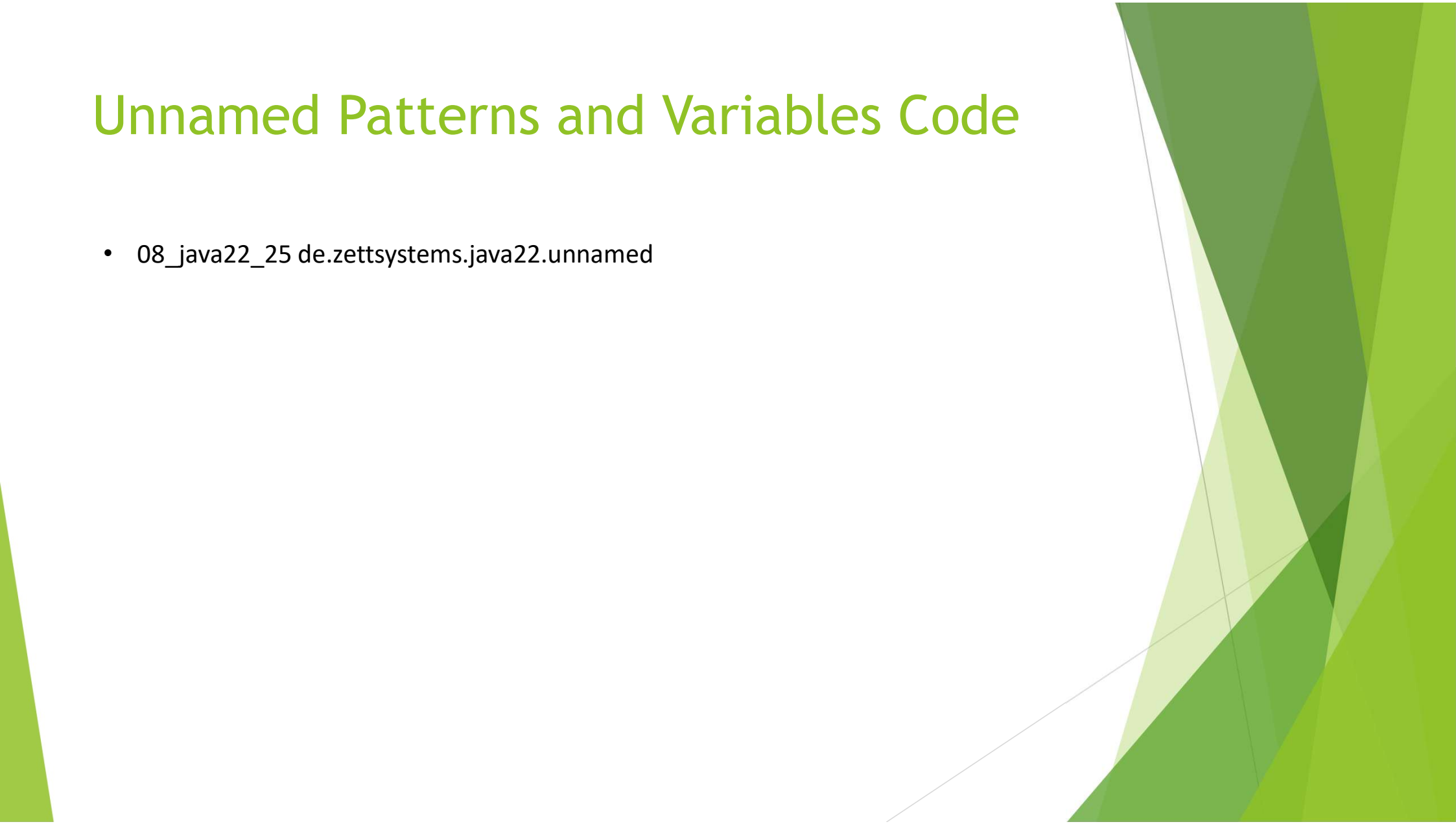
- JEP 447: Statements before super(…) (Preview)
- JEP 457: Class-File API (Preview)
- JEP 459: String Templates (Second Preview)
- JEP 464: Scoped Values (2nd Preview)
- JEP 459: String Templates (2nd Preview)
- JEP 460: Vector API (7. Incubator)
- JEP 461: Stream Gatherers (Preview)
- JEP 462: Structured Concurrency (Second Preview)
- JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)

Java 22 - Unnamed Patterns and Variables

- Das letzte (?) feature im Großthema Pattern Matching
- Auch hier allgemeiner Nutzen:
  - Man kann jetzt ungenutzte Variablen, die man konstruktionsbedingt aber braucht, durch einen Unterstrich ersetzen
  - Erstens muss man weniger schreiben und zweitens kann man so signalisieren, dass z.B. das ‚i' in der Schleife gar nicht gebraucht wird
- Und im Kontext von Record Patterns kann man es eben auch nutzen und die Attribute, die man schreiben muss, aber nicht braucht so ersetzen
- Ersetzt wird sowohl Name als auch Typ
- Außerdem kann man Variablen, auf die man nicht zugreifen möchte ebenfalls mit _ auszeichnen

Unnamed Patterns and Variables Code

- 08_java22_25 de.zettsystems.java22.unnamed



Java 22 - Foreign Function & Memory API

- Endlich fertig nachdem die ersten Sachen mit Java 14 gestartet sind:
  - The so-called “Foreign Memory Access API” was presented in the incubator stage back in March 2020 in Java 14 (JEP 370).
  - One year later, the “Foreign Linker API” was introduced in the incubator stage in Java 16 (JEP 389).
  - In Java 17, the two APIs were merged into the “Foreign Function & Memory API,” and this unified API was presented once again as an incubator version (JEP 412).
  - In Java 19, the FFM API was promoted to the preview stage (JEP 424).
  - In Java 22, the API was declared ready for production and finalized in March 2024 after a long development and maturation period (JEP 454).
- Kurz gesagt: die bessere Alternative, um Code/Bibliotheken außerhalb der JVM anzusprechen

Java 22 - Launch Multi-File Source-Code Programs

- Schon seit Java 11 braucht man 1-Datei Java Programme nicht mehr zuerst kompilieren, sondern kann sie direkt mit java Name.java starten
  - Diese werden dann on-the-fly kompiliert
- Mit Java 22 wird dies nun auf mehrere Dateien ausgedehnt.
- Wenn man die Datei mit der Main-Klasse startet und diese referenziert weitere, werden diese mit kompiliert
- Insgesamt hilft das dabei, Programmieranfängern den Zugang zu Java zu erleichtern, da man das Thema kompilieren schieben kann und nicht gleich am Anfang behandeln muss

Launch Multi-File Source-Code Programs Code

- 08_java22_25 de.zettsystems.java22.multfiles
- Java 23 in den Path!
- Java Prog.java

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- **Ausblick Java 22-25**
  - Java 22
  - **Java 23**
  - Java 24+
- OpenRewrite

Java 23 – 17.09.2024

1. New Language Features

- ---

2. JVM Improvements

- JEP 474: ZGC: Generational Mode by Default

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- JEP 467: Markdown Documentation Comments
- JEP 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal

5. Incubator and Preview Features

- JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview)
- JEP 466: Class-File API (Second Preview)
- JEP 459: String Templates (Second Preview)
- JEP 481: Scoped Values (3rd Preview)
- JEP 469: Vector API (8th Incubator)
- JEP 473: Stream Gatherers (2nd Preview)
- JEP 480: Structured Concurrency (3rd Preview)
- JEP 477: Implicitly Declared Classes and Instance Main Methods (3rd Preview)
- JEP 476: Module Import Declarations (Preview)
- JEP 482: Flexible Constructor Bodies (Second Preview)

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- **Ausblick Java 22-25**
  - Java 22
  - Java 23
  - **Java 24+**
- OpenRewrite

Java 24 – 18.03.2025

1. New Language Features

- JEP 484: Class-File API
- JEP 485: Stream Gatherers

2. JVM Improvements

- JEP 474: ZGC: Generational Mode by Default
- JEP 491: Synchronize Virtual Threads Without Pinning

3. New Tools and Libraries

- ---

4. Miscellaneous

- Diverse Security Updates und Housekeeping
- …

5. Incubator and Preview Features

- Fortsetzung der 23er Themen
- JEP 478: Key Derivation Function API (Preview)

Java 24 - Stream Gatherers

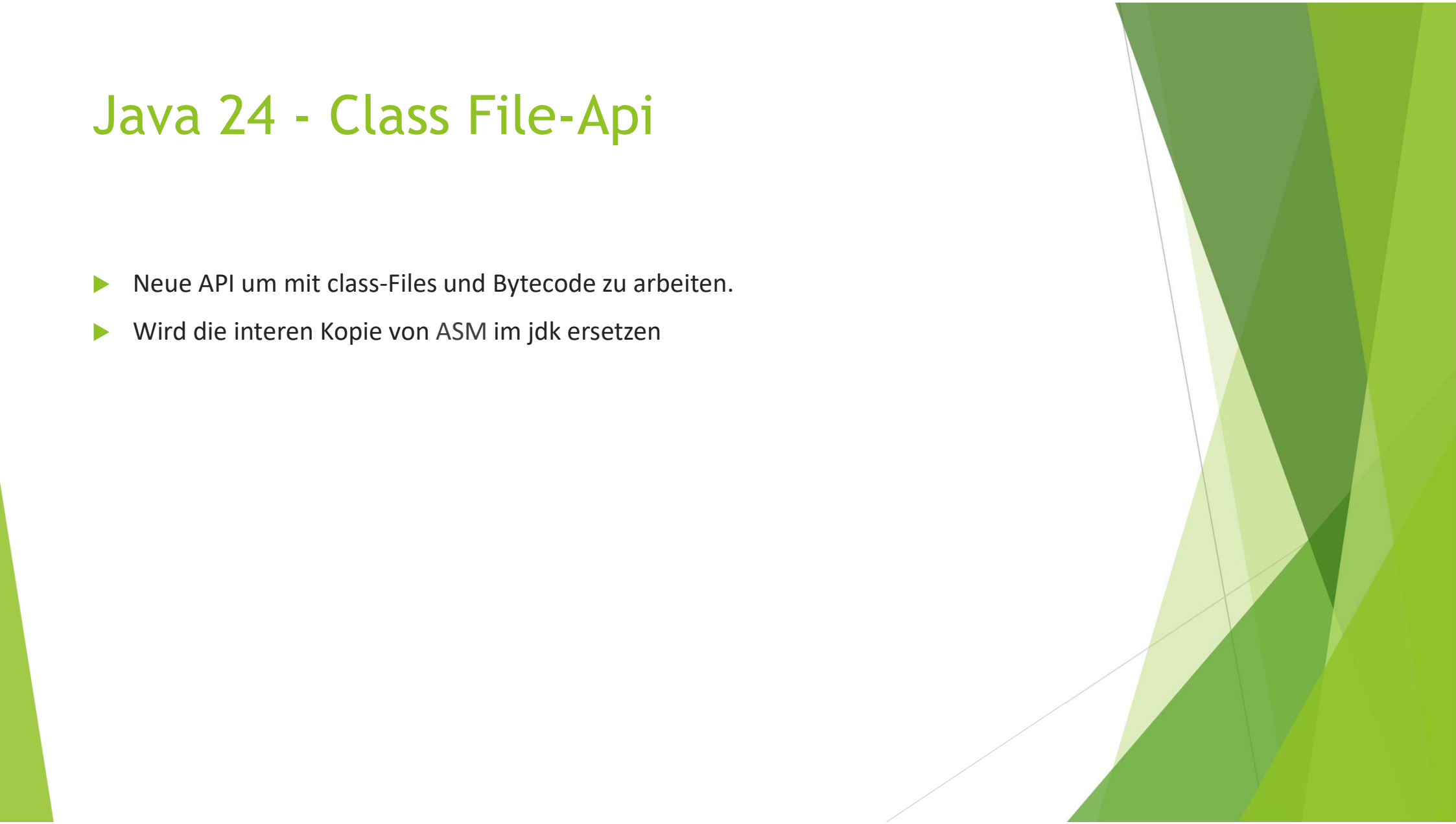
- Eine Api um eigene intermediate Operations für Streams zu schreiben
- Werden mit der Methode gather(Gatherer gatherer) in den Stream eingebunden
- Gibt einige vorgefertigte im jdk und man kann gatherer natürlich kombinieren
- Sehr mächtig aber auch nicht ganz einfach

Stream Gatherers Code

- 08_java22_25 de.zettsystems.java24.gather

Java 24 - Class File-Api

- Neue API um mit class-Files und Bytecode zu arbeiten.
- Wird die interen Kopie von ASM im jdk ersetzen



Class File Api Code

- 08_java22_25 de.zettsystems.java24.classfile

Previews nutzen

- Um Preview-Features in Java zu verwenden, müssen Sie sie sowohl beim Kompilieren als auch beim Ausführen explizit aktivieren.
  - Das Verfahren sieht folgendermaßen aus:
  - Kompilieren: javac --enable-preview --release 23 MeineKlasse.java
    - --enable-preview aktiviert die Preview-Features.
    - --release 23 stellt sicher, dass die Version des Quellcodes (und des Bytecodes) korrekt auf Java 23 eingestellt wird.
  - Ausführen: java --enable-preview MeineKlasse
  - Hiermit wird der Bytecode mit aktiven Preview-Features gestartet.
  - Wichtig: Preview-Features sind in einer Java-Version noch nicht final. Sie können sich in folgenden Releases ändern oder entfernt werden.

Unnamed Classes and Instance Main Methods

- Ein weiteres feature, um Java leichter zugänglich zu machen
- Man kann die Main-Methode reduzieren auf: void main() {}, also alle „Schlüsselwörter für große Programme“ und die Parameter weg lassen
- Darüber hinaus kann man sogar die umschließende Klasse weglassen, wenn man ein 1-Klassen Programm schreiben möchte

Instance Code

- 08_java22_25 de.zettsystems.java23_25.instance

Scoped Values (3rd preview)

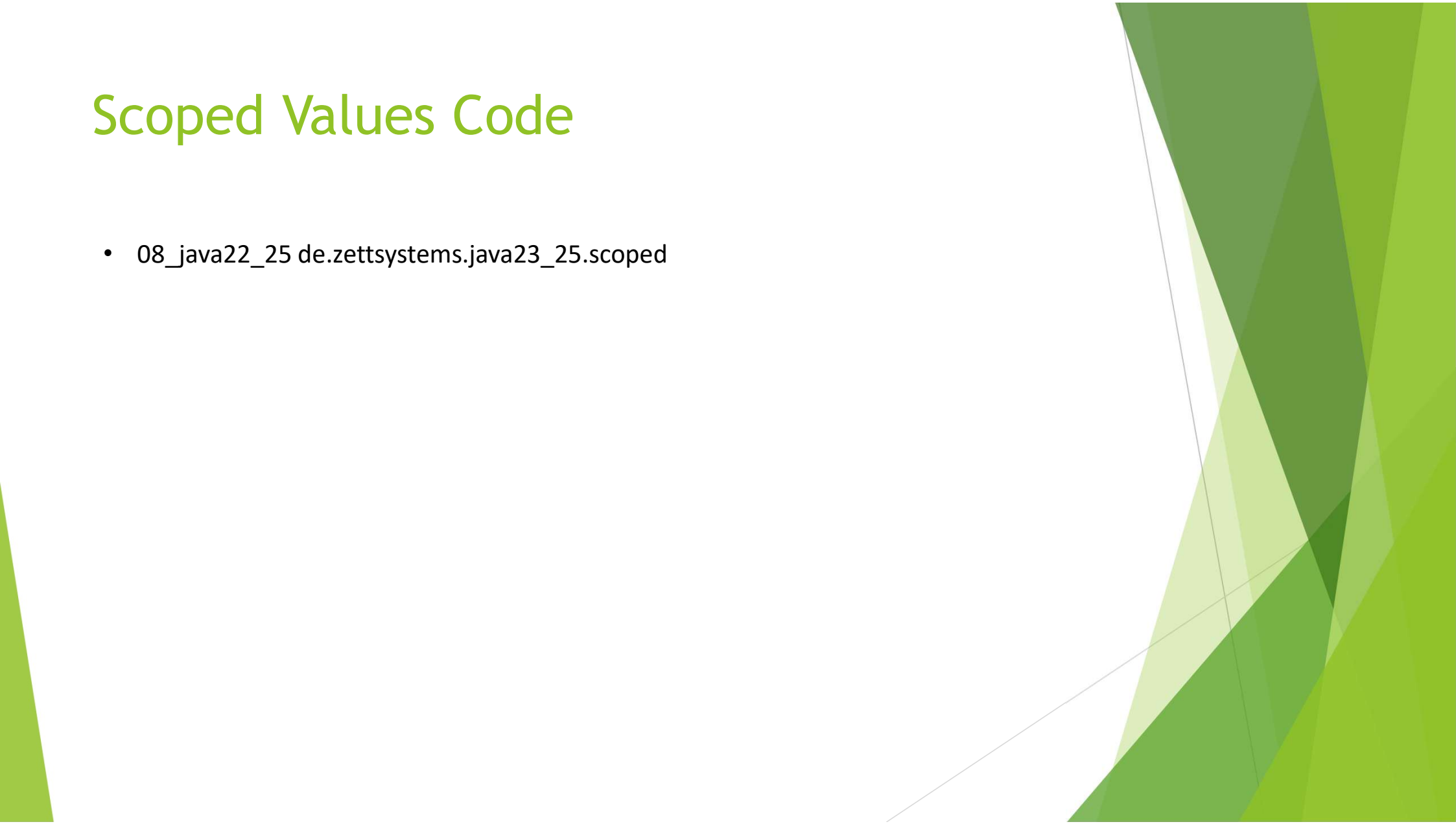
- Ermöglicht die effiziente gemeinsame Nutzung von immutable data innerhalb und zwischen Threads.
- Wert:
  - Benutzerfreundlichkeit - Bietet ein Programmiermodell für die gemeinsame Nutzung von Daten sowohl innerhalb eines Threads als auch mit untergeordneten Threads, um die Überlegungen zum Datenfluss zu vereinfachen.
  - Verständlichkeit - Macht die Lebensdauer gemeinsam genutzter Daten anhand der syntaktischen Struktur des Codes sichtbar.
  - Robustheit - Gewährleistet, dass Daten, die von einem Aufrufer gemeinsam genutzt werden, nur von legitimen Aufrufen abgerufen werden können.
  - Leistung - Behandelt gemeinsam genutzte Daten als unveränderlich, um die gemeinsame Nutzung durch eine große Anzahl von Threads und Laufzeitoptimierungen zu ermöglichen.

Vorteile von ScopedValues ggü. ThreadLocal

- ScopedValues sind nur während der Lebensdauer des an die run(...)-Methode übergebenen Runnable gültig und werden unmittelbar danach zur Garbage Collection freigegeben (sofern keine weiteren Referenzen auf sie existieren). – Ein thread-lokaler Wert hingegen bleibt im Speicher, bis entweder der Thread beendet wird (was bei der Verwendung eines Thread-Pools nie der Fall sein kann) oder er explizit mit ThreadLocal.remove() gelöscht wird. Da viele Entwickler vergessen, dies zu tun (oder es nicht tun, weil das Programm so komplex ist, dass es nicht offensichtlich ist, wann ein thread-lokaler Wert nicht mehr benötigt wird), sind Speicherlecks oft die Folge.
- Ein ScopedValue ist unveränderlich - er kann nur durch erneutes Binden für einen neuen Scope zurückgesetzt werden. Dies verbessert die Verständlichkeit und Wartbarkeit des Codes erheblich im Vergleich zu Thread-Locals, die jederzeit mit set() geändert werden können.
- Die von StructuredTaskScope erzeugten Child-Threads haben Zugriff auf den scoped value des Parent-Threads. Verwendet man dagegen InheritableThreadLocal, so wird dessen Wert in jeden Child-Thread kopiert, so dass ein Child-Thread den Thread-Local-Wert des Parent-Threads nicht ändern kann. Dies kann den Speicherbedarf erheblich vergrößern.

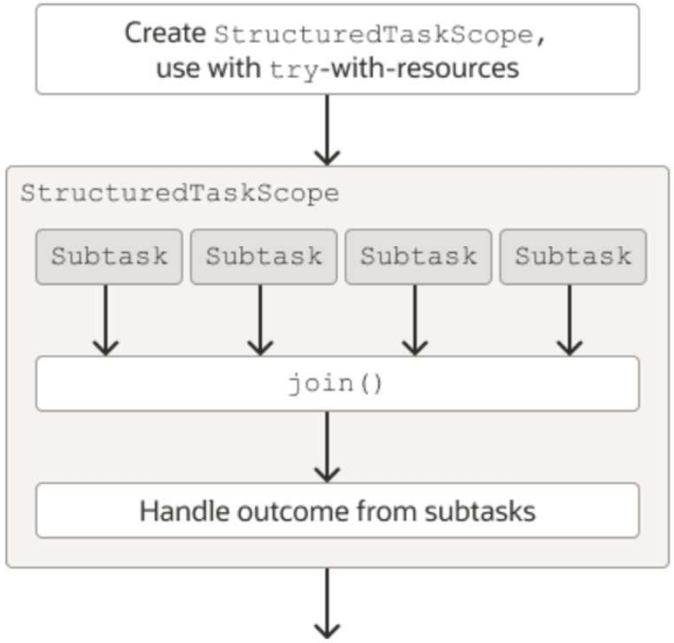
Scoped Values Code

- 08_java22_25 de.zettsystems.java23_25.scoped



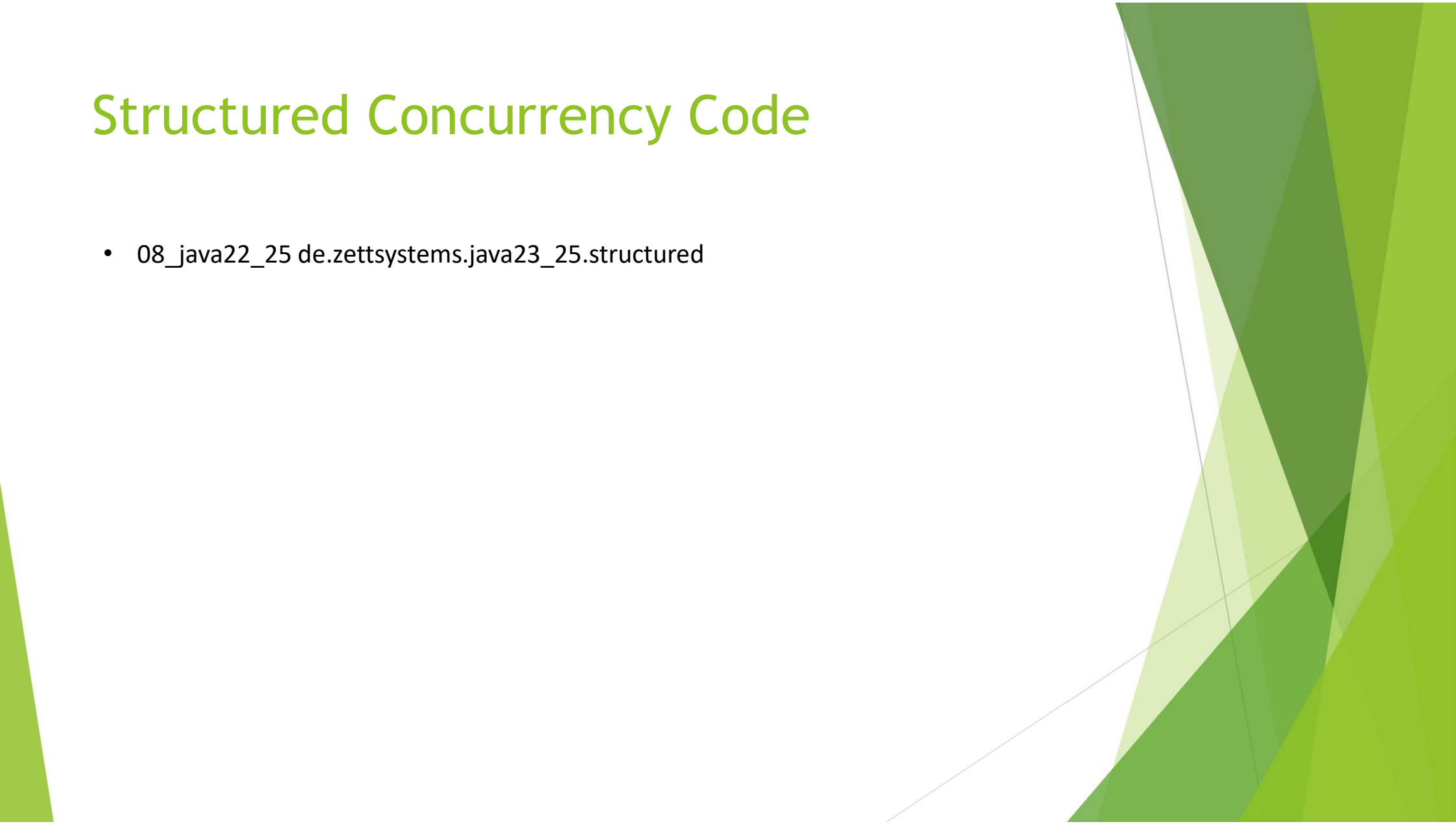
Structured Concurrency (3rd preview)

- Vereinfacht die nebenläufige Programmierung. Hierbei werden Gruppen zusammengehöriger Aufgaben, die in verschiedenen Threads ausgeführt werden, als eine einzige Arbeitseinheit behandelt.



Structured Concurrency Code

- 08_java22_25 de.zettsystems.java23_25.structured



Vector Api (8th Incubator!)

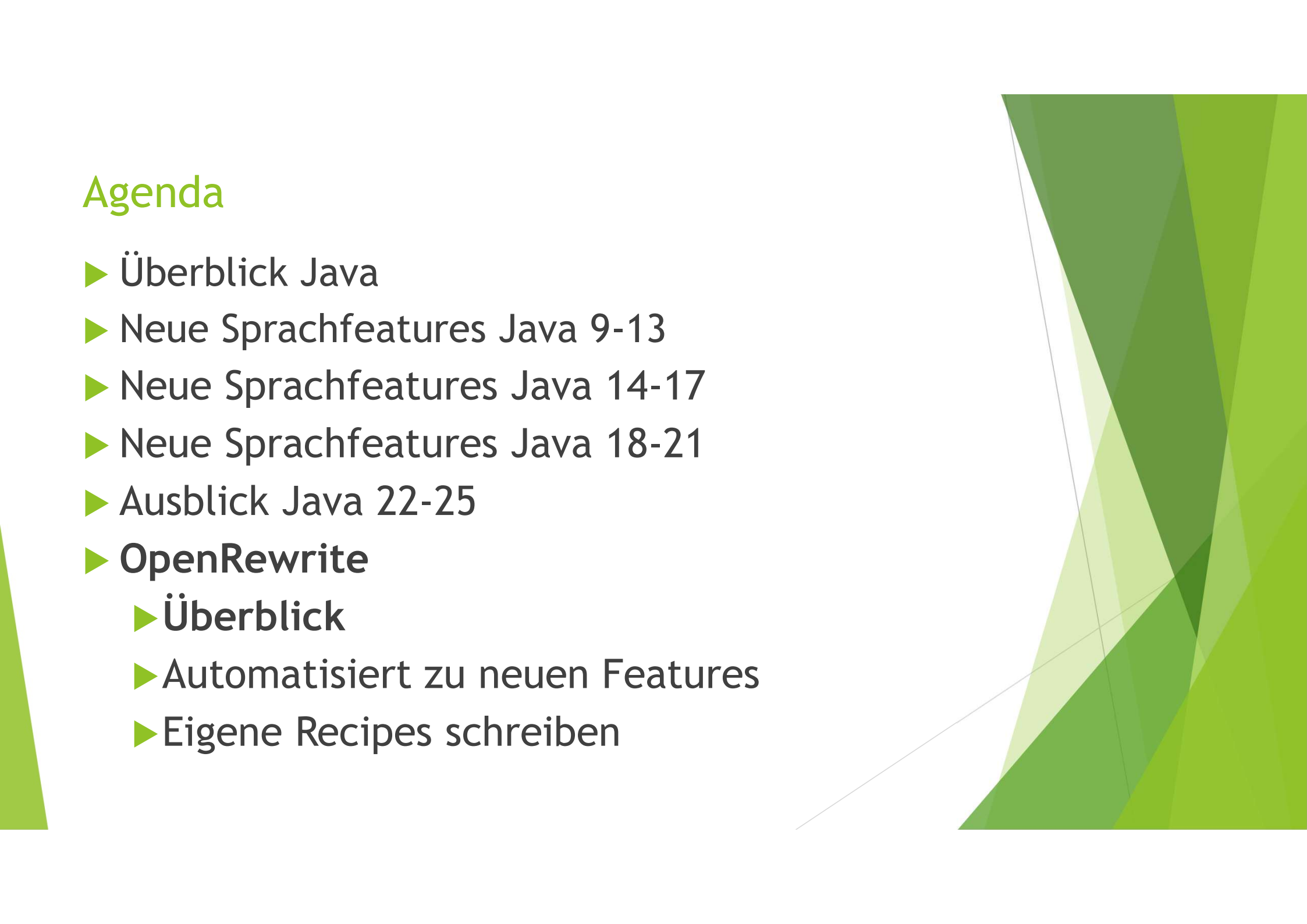
- Eine API zum Ausdrücken von Vektorberechnungen, die zur Laufzeit zuverlässig zu optimalen Vektoranweisungen auf unterstützten CPU-Architekturen kompiliert werden, wodurch eine höhere Leistung als bei äquivalenten Skalarberechnungen erreicht wird.
- In JDK 23 immer noch Incubating.
- Nutzt die neusten Features vor allem von Foreign Function & Memory API.
- Arbeitet eng mit Projekt Valhalla (Value Objects) zusammen
- Kein Endtermin in Sicht

Java 25 – September 2025 (LTS bis 2033)

- Abschluss hoffentlich vieler Preview-Themen
- String Templates werden noch mal re-designed…

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- **OpenRewrite**
  - **Überblick**
  - Automatisiert zu neuen Features
  - Eigene Recipes schreiben



Automatisiert Refactorings anwenden

Refactoring: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. (Fowler, https://martinfowler.com/bliki/DefinitionOfRefactoring.html)

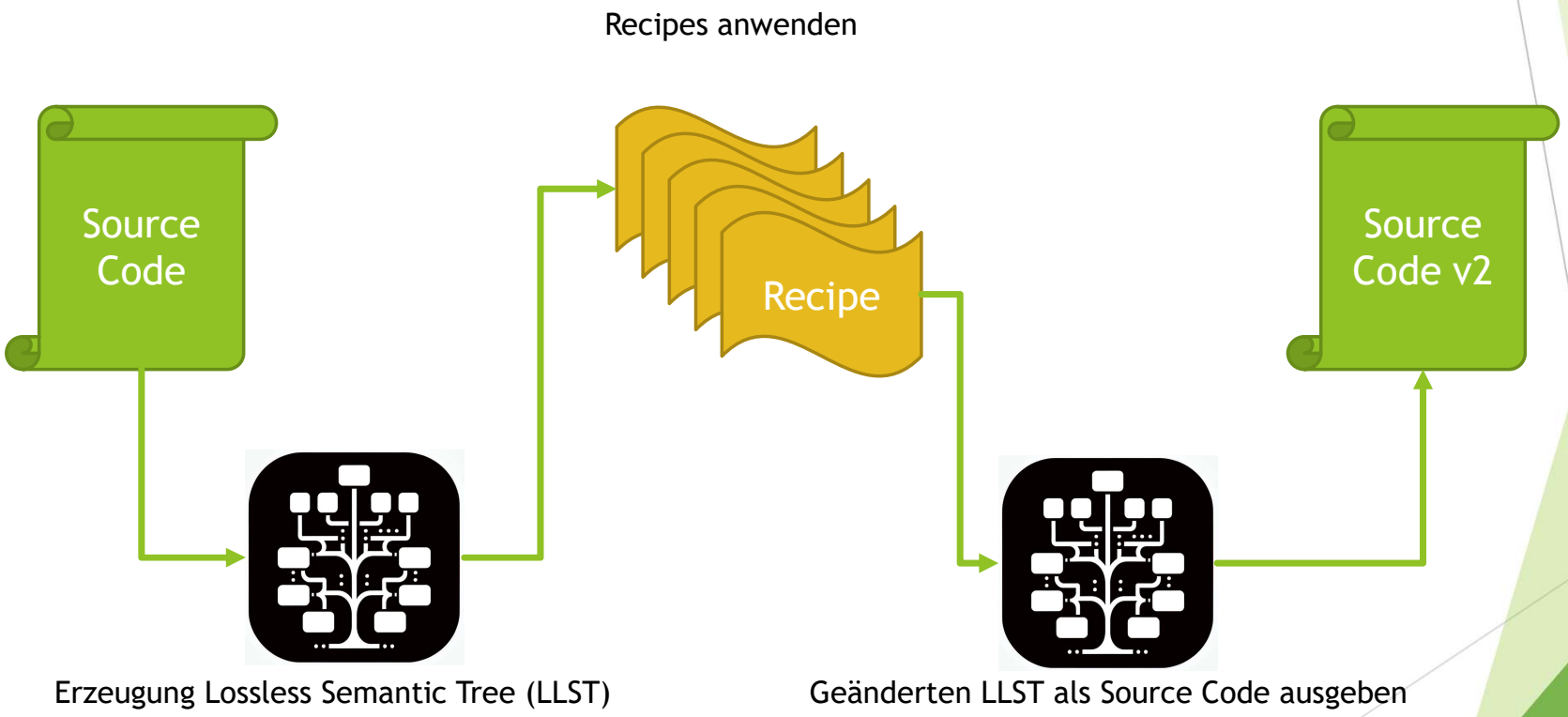
Automatisiert: OpenRewrite liefert Plugins für Maven und gradle und bringt bereits diverse sogenannte Recipes mit, so dass auf die Entwicklerin lediglich die Konfiguration dieser sowie das Review der Ergebnisse zukommt.

Beispiel (oder anders rum ;-)):

```
// Before OpenRewrite
import org.junit.Assert;
...

Assert.assertTrue(condition);
```

```
// After OpenRewrite
import static org.junit.Assert.assertTrue;
...

assertTrue(condition);
```

Funktionsweise



Was gibt es für Recipes („fachlich“)?

Code Style:

- im Prinzip vergleichbar zu PMD, Checkstyle, Sonarqube – nur dass etwaige Verstöße nicht angemakert, sondern gleich behoben werden. Z.B.:
  - org.openrewrite.staticanalysis.RemoveUnusedPrivateFields
  - org.openrewrite.staticanalysis.UnnecessaryParentheses

Updates:

- Durchführung von Java und Library Updates!
  - org.openrewrite.java.migrate.UpgradeToJava17
  - org.openrewrite.java.testing.junit5.JUnit5BestPractices
  - org.openrewrite.java.testing.assertj.Assertj

Typen von Recipes

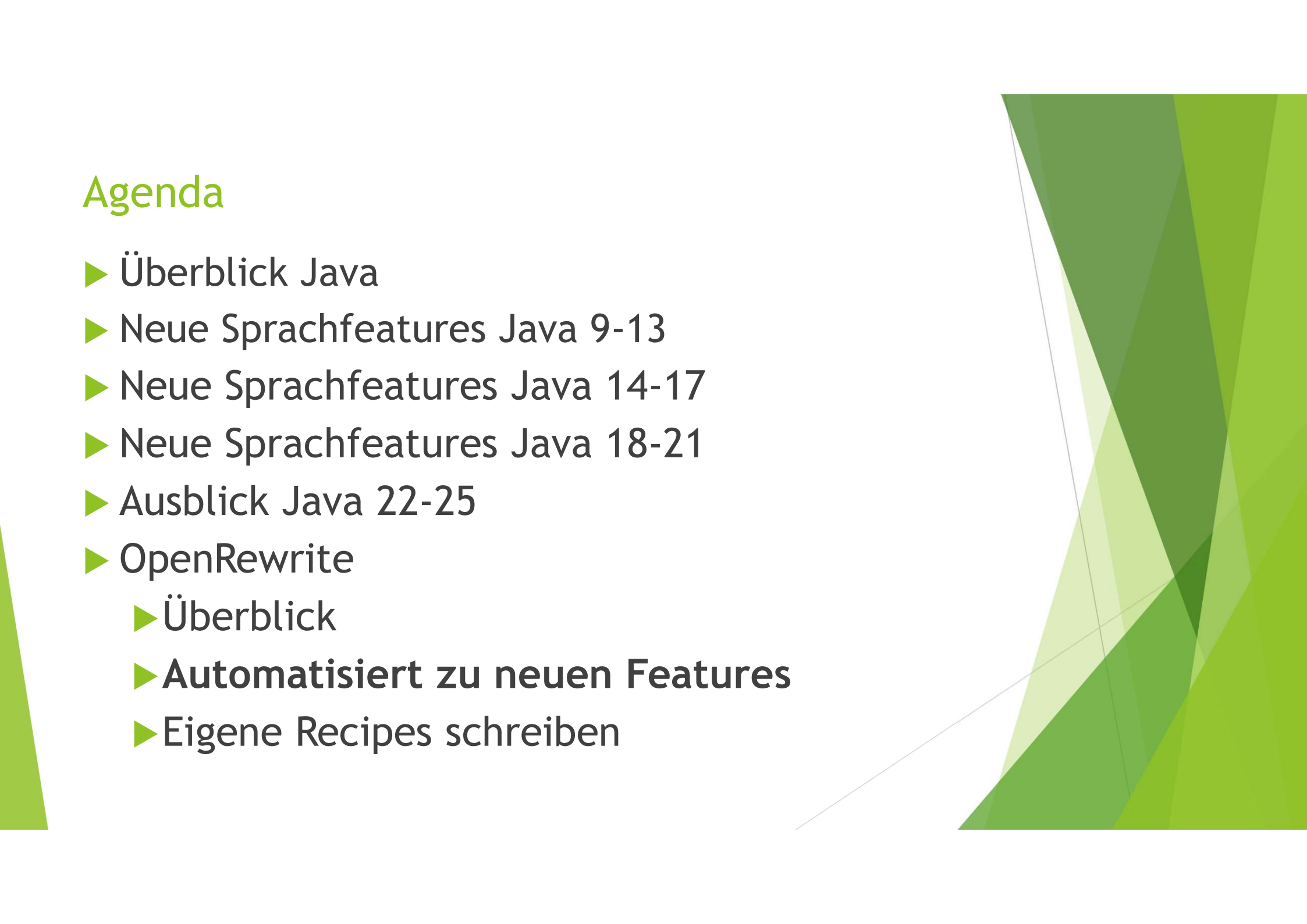
- **Deklarative Recipes**: Zusammenstellung vorhandener Recipes und ggf. deren Parametrierung in der rewrite.yml
- **Refaster Templates:** Einfache Eine-Zeile-Ersetzungen
- **Imperative Recipes:** Ein Programmiertes Recipe

```yaml
---
type: specs.openrewrite.org/v1beta/recipe
name: org.openrewrite.staticanalysis.CommonStaticAnalysis
displayName: Common static analysis issues
description: Resolve common static analysis issues discovered through 3rd party tools.
recipeList:
  - org.openrewrite.staticanalysis.AtomicPrimitiveEqualsUsesGet
  - org.openrewrite.staticanalysis.BigDecimalRoundingConstantsToEnums
  - org.openrewrite.staticanalysis.BooleanChecksNotInverted
  - org.openrewrite.staticanalysis.CaseInsensitiveComparisonsDoNotChangeCase
  - org.openrewrite.staticanalysis.CatchClauseOnlyRethrows
  - org.openrewrite.staticanalysis.ChainStringBuilderAppendCalls
```

```java
public static class RedundantCall {
    @BeforeTemplate
    public String start(String string) {
        return string.substring(0, string.length());
    }

    @BeforeTemplate
    public String startAndEnd(String string) {
        return string.substring(0);
    }

    @BeforeTemplate
    public String toString(String string) {
        return string.toString();
    }

    @AfterTemplate
    public String self(String string) {
        return string;
    }
}
```

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite
  - Überblick
  - **Automatisiert zu neuen Features**
  - Eigene Recipes schreiben

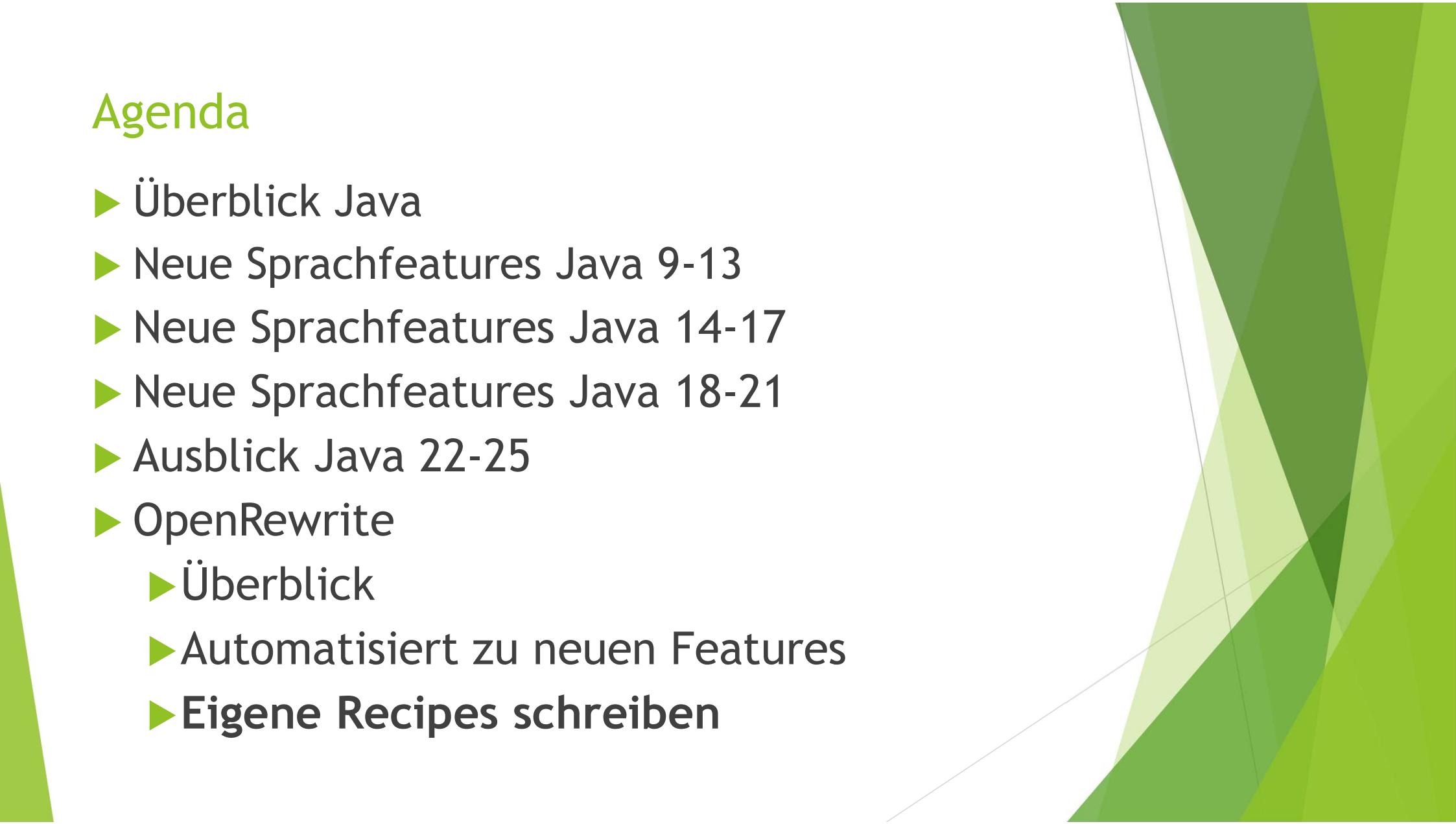


Altanwendung automatisiert modernisieren

- 08_openrewrite
- https://docs.openrewrite.org/recipes/java/migrate/javaversion21
  - TextBlocks
  - PatternMatching InstanceOf
  - Stream.toList() Recipe muss man zusätzlich nennen
  - TODO

Agenda

- Überblick Java
- Neue Sprachfeatures Java 9-13
- Neue Sprachfeatures Java 14-17
- Neue Sprachfeatures Java 18-21
- Ausblick Java 22-25
- OpenRewrite
  - Überblick
  - Automatisiert zu neuen Features
  - **Eigene Recipes schreiben**



Zettsystems-recipes

- https://github.com/MichaelZett/zettsystems-recipes

Abschluss

- Fragen?