GEDOPLAN

Die besten Features in Java 22 und 23

Ein Blick auf die neuesten Sprach- und Syntaxverbesserungen

Michael Kulla
Enterprise Java Team, GEDOPLAN GmbH
JUG Saxony Day 2024

GEDOPLAN

Michael Kulla

- Sun Certified Java Trainer
- Oracle University Delivery Instructor Java
- Autor diverser Videotrainings u.a. zu Java, Java EE und Entwurfsmustern

- lange Zeit freiberuflicher Softwareentwickler und Berater
- seit 2017 festangestellt bei GEDOPLAN GmbH in Bielefeld als Trainer, Berater und Softwareentwickler Java und Jakarta EE
- Proofreader diverser Java-Bücher

GEDOPLAN

GEDOPLAN GmbH

- GEDOPLAN GmbH in Bielefeld
  - Softwareentwicklung, Beratung, IT-Training
  - Mittelpunkt: Java, JEE, Spring, Tools
- Beratung und Softwareentwicklung
  - professionelle Abwicklung von IT-Projekten
  - Beratung, Reviews, Konzeption, Softwareentwicklung
  - Unterstützung vor Ort und remote
- IT-Training in Berlin, Bielefeld, on-site und remote
  - offene und firmenspezifische Kurse
  - praxiserfahrene Trainer

GEDOPLAN

Agenda

- Java 22 und Java 23: Neue Versionen – was ist drin?
- Im Detail: 3 Features, 4 Previews
- Fazit: Java 22 und Java 23 – Was können wir mitnehmen?
- Ausblick

GEDOPLAN

Java-Versionen seit Java 17

- Halbjährlicher Release-Zyklus
  - 6 Monate Support bis zum Erscheinen des nächsten Release
  - alle 3 Jahre LTS-Release: Long-Term-Support, 3 Jahre
  - seit Java 17: LTS-Release-Zyklus verkürzt auf 2 Jahre

Java 17 LTS	14. September 2021
Java 18	22. März 2022
Java 19	20. September 2022
Java 20	21. März 2023
Java 21 LTS	19. September 2023
Java 22	19. März 2024
Java 23	17. September 2024

GEDOPLAN

Preview Features, Incubating Features

- Preview Feature
  - Fertig, aber noch nicht abschließend spezifiziert
  - Zum Sammeln von Feedback
  - Nicht in Produktiv-Code verwenden
  - Aktivieren mit `--enable-preview --release 23 …`
- Incubating Feature
  - Experimentell
  - Frühzeitig erstes Feedback sammeln
  - Gar nicht in Produktiv-Code verwenden
  - Aktivieren mit `--add-modules <module-name>`

GEDOPLAN

Features in J22 und 23, die wo vorgestellt werden

- Final – Teil des Standards
  - 456: Unnamed Variables & Patterns
  - 458: Launch Multi-File Source-Code Programs
  - 467: Markdown Documentation Comments

GEDOPLAN

Features in J22 und 23 die wo vorgestellt werden

- Preview – schon dabei, aber vielleicht auch wieder weg
  - 459: String Templates (Second Preview) – schon wieder weg
  - 455: Primitive Types in Patterns, instanceof, and switch (Preview)
  - 473: Stream Gatherers (Second Preview)
  - 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
  - 480: Structured Concurrency (Third Preview)

GEDOPLAN

Features in J22 und J23 – was sonst noch passierte

423: Region Pinning for G1

454: Foreign Function & Memory API

466: Class-File API (Second Preview)

469: Vector API (Eighth Incubator)

471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal

474: ZGC: Generational Mode by Default

476: Module Import Declarations (Preview)

481: Scoped Values (Third Preview)

482: Flexible Constructor Bodies (Second Preview)

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - 23 Primitive Types in Patterns, instanceof, and switch (Preview)
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - 23 Primitive Types in Patterns, instanceof, and switch (Preview)
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

Launch Multi-File Source-Code Programs (Java 22)

Bei der Direct Compilation (Ausführung eines Java-Programms ohne Compilerlauf) kann die Anwendung nun auch in mehrere Dateien aufgeteilt werden.

GEDOPLAN

Launch Multi-File Source-Code Programs

- Bisher: Direct Compilation (seit Java 11):
  - Gesamtes Java-Programm in einer Quelltextdatei
  - Direkt ausführbar, ohne Compilerlauf

```
java DirectCompilationDemo.java
```

- Alle Klassen in einer Datei
- schnell unübersichtlich
- Keine Wiederverwendbarkeit

GEDOPLAN

Launch Multi-File Source-Code Programs

- Jetzt:
  - Bei mehreren Klassen kann jede Klasse in eigene Datei
  - Weiterhin direkt ausführbar

```
MultiFileSourceCodeProgram.java
Helper.java

java MultiFileSourceCodeProgram.java
```

GEDOPLAN

Launch Multi-File Source-Code Programs

Was bringt's?

- Bessere Strukturierung
- Wiederverwendbarkeit einzelner Klassen
- Java als Skriptsprache
  - Ändern – Ausführen
  - Einfach Java – statt bash, zsh, cmd, Powershell, ...
- Niedrige Einstiegsschwelle
  - kein Buildwerkzeug (Maven, Gradle, ...)
  - simple Projektstruktur

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - 23 Primitive Types in Patterns, instanceof, and switch (Preview)
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

JEP 477: Implicitly Declared Classes and Instance Main Methods

Java 21 bis 23, Third Preview

Ein vollständiges und ausführbares Java-Programm muss nur noch eine einfache `main()`-Methode enthalten.

Klassendeklaration, Methodenparameter und „`System.out.`“ können entfallen.

GEDOPLAN

Implicitly Declared Classes and Instance Main Methods

- Bisher:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("hello world!");
  }
}
```

- Für Einsteiger viele neue Konzepte auf einmal
  - Klassen, Methoden, statische Methoden, Methodenargumente, Sichtbarkeit, `System.out`

GEDOPLAN

Implicitly Declared Classes and Instance Main Methods

- Jetzt:

```
void main() {
  print("simply hello world!");
}
```

- Reduzierung auf das Wesentliche
- Klasse wird vom Compiler implizit erzeugt
- `main()` ohne Sichtbarkeitsmodifizierer und ohne Argumente
- Neue Klasse `java.io.IO` mit statischen Methoden `print()`, `println()`, `readln()`, automatisch importiert

GEDOPLAN

Implicitly Declared Classes and Instance Main Methods

Was bringt's?

- Deutliche Erleichterung für Einsteiger
- Automatischer Import des Moduls `java.base`
- Dadurch viele Klassen ohne Imports nutzbar
  - `java.util`, `java.io`, `java.math`...
- Vereinfacht zusammen mit Multi-File Source-Code Programs das Erstellen kleiner Java-Programme für alltägliche Aufgaben
- Programme können schrittweise erweitert werden, Umstieg auf „richtiges" Programm jederzeit möglich
- Preview-Feature, Verwendung nur mit `--enable-preview` möglich

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - 23 Primitive Types in Patterns, instanceof, and switch (Preview)
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

JEP 456: Unnamed Variables & Patterns

Java 22

Ungenutzte Variablen und Record-Teile können durch einen Unterstrich _ ersetzt werden.

GEDOPLAN

Unnamed Variables and Patterns

- **Bisher:** Variable wird nicht benötigt, muss aber trotzdem deklariert werden

```
try {
  Thread.sleep(1000);
  System.out.println("done sleeping");
} catch (InterruptedException ignored) {
  // do nothing
}
```

- **Jetzt:** Unnamed Variable _ für nicht verwendete Variablen

```
try {
  Thread.sleep(1000);
  // ...
} catch (InterruptedException _) { }
```

GEDOPLAN

Unnamed Variables and Patterns

- Unnamed Variable _ für nicht verwendete Variablen

```
try {

  Person p = personRepo.findById(id);
  return Optional.ofNullable(p);

} catch (IOException _) {

  return Optional.empty();
}
```

GEDOPLAN

Unnamed Variables and Patterns

- Einführung von _ wurde schon länger vorbereitet:
  - _ als Bezeichner bereits in Java 8 Compilerwarnung
  - und ab Java 9 Compilerfehler

- ansonsten aber weiterhin erlaubt:
  - als Teil eines Bezeichners mit zwei oder mehr Zeichen

```
int _number = 123;
```

  - in Zahlenliteralen zur Verbesserung der Lesbarkeit

```
int distance = 12_000_000;
```

GEDOPLAN

Unnamed Variables and Patterns

Bisher:

```
double calculateAreaNew(Shape shape) {
 return switch (shape) {
  case Point(int x, int y) -> 0;
  case Square(Point p, int s) -> s * s;
  case ColoredSquare(Square(Point p, int s), Color c) -> s * s;
 };
}
```

Jetzt: Unnamed Record Pattern:

```
double calculateAreaNew(Shape shape) {
  return switch (shape) {
    case Point _ -> 0;
    case Square(_, int s) -> s * s;
    case ColoredSquare(Square(_, int s), _) -> s * s;
  };
}
```

GEDOPLAN

Unnamed Variables and Patterns

- Unbenannte Variablen können verwendet werden:
  - in einer lokalen Variablendeklaration in einem Block
  - in der Ressourcenangabe in try-with-resources
  - im Schleifenkopf einer einfachen oder erweiterten for-Schleife

```
for (var _ : names) {
  System.out.println("Processing ...");
}
```

  - als formaler Parameter eines Lambdaausdrucks

```
names.forEach(_ -> System.out.println("Processing ..."));
```

  - als Exceptionparameter in einem catch-Block

GEDOPLAN

Unnamed Variables and Patterns

Was bringt's?

- unnötige Informationen werden entfernt
- Code ist leichter lesbar und verständlicher
- verdeutlicht, dass ein Wert nicht für die Verwendung vorgesehen ist
- Compiler verhindert versehentliche Schreibzugriffe

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - 23 Primitive Types in Patterns, instanceof, and switch (Preview)
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

JEP 467: Markdown Documentation Comments

Java 23

Javadoc-Kommentare können in Markdown verfasst werden, anstatt eine Mischung aus HTML und @-Tags verwenden zu müssen.

GEDOPLAN

Markdown Documentation Comments

Javadoc der Methode `min()` aus `java.lang.Math` – bisher:

```
/**
 * Returns the smaller of two {@code int} values. That is,
 * the result the argument closer to the value of
 * {@link Integer#hashCode}. If the arguments have the same
 * value, the result is that same value.
 *
 * @param   a   an argument.
 * @param   b   another argument.
 * @return  the smaller of {@code a} and {@code b}.
 */
```

Jetzt mit Markdown:

```
/// Returns the smaller of two `int` values. That is,
/// the result is the argument closer to the value of
/// [Integer#MIN_VALUE]. If the arguments have the same
/// value, the result is that same value.
///
/// @param a an argument.
/// @param b another argument.
/// @return the smaller of `a` and `b`.
```

GEDOPLAN

Markdown Documentation Comments

Was?	Javadoc	Markdown
Sourcecode	`{@code int x = 5;}`	`` `int x = 5;` ``

Codeblock	`<pre>`
	`  var x;`
	`  var y;`
	`</pre>`
		```` ````
		`  var x;`
		`  var y;`
		```` ````

Absatz	`<p>`	Leerzeile
Link	`{@link Object#hashCode()}`	`[Object#hashCode()]`
Fett	`<b>fett</b>`	`**fett**`
Kursiv	`<i>kursiv</i>`	`*kursiv*`

Aufzählung	`<ul>`
	`<li>dies</li>`
	`<li>das</li>`
	`</ul>`
	`<ol>`
	`<li>hier</li>`
	`<li>da</li>`
	`</ol>`
		`- dies`
		`- das`
		`1. hier`
		`2. da`

GEDOPLAN

Markdown Documentation Comments

Was?	Javadoc	Markdown
Tabelle	<table> <tr> <th>english</th> <th>deutsch</th> </tr> <tr> <td>one</td> <td>eins</td> </tr> <tr> <td>two</td> <td>zwei</td> </tr> </table>	/// | english | deutsch | /// |---|---| /// | one | eins | /// | two | zwei |

GEDOPLAN

Markdown Documentation Comments

- Javadoc-Tags (`@param`, `@return`, `@see` etc.) bleiben erhalten
- und funktionieren wie bisher
- Innerhalb von Code oder Codeblöcken (im Markdown) werden sie nicht ausgewertet

GEDOPLAN

Markdown Documentation Comments

Was bringt's?

- Verbesserte Lesbarkeit
- Leichter zu verstehen und besser wartbar
- Einfachere Formatierung
- Leicht zu erlernende Syntax
- Funktionalität von Javadoc geht nicht verloren
- Kommentare und Doku schreiben macht wieder Spaß

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - **23 Primitive Types in Patterns, instanceof, and switch (Preview)**
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

JEP 455: Primitive Types in Patterns, instanceof, and switch

Java 23, Preview

In Patterns, in instanceof und switch können jetzt auch primitive Datentypen verwendet werden.

GEDOPLAN

Primitive Types in Patterns, instanceof, and switch

Bisher:

- `instanceof` nur mit Referenztypen möglich, nicht für primitive Datentypen
- z.B. Bereichsüberprüfung vor Cast von `int` auf `byte`

```
if (value >= -128 && value <= 127) {
  byte byteValue = (byte) value;
  System.out.println("byteValue = " + byteValue);
}
```

Jetzt:

```
if (value instanceof byte byteValue) {
  System.out.println("byteValue = " + byteValue);
}
```

GEDOPLAN

Primitive Types in Patterns, instanceof, and switch

Bisher:

- `switch patterns` nur mit Referenztypen möglich
- z.B. unnötiger doppelter Methodenaufruf:

```
String msgOld = switch (logLevel.severity()) {
  case 0 -> "info";
  case 1 -> "warning";
  case 2 -> "error";
  default -> "unknown severity: " + logLevel.severity();
};
```

Jetzt:

```
String msgNew = switch (logLevel.severity()) {
  case 0 -> "info";
  case 1 -> "warning";
  case 2 -> "error";
  case int severity -> "unknown severity: " + severity;
};
```

GEDOPLAN

Primitive Types in Patterns, instanceof, and switch

Was bringt's?

- Einheitliche Verwendung von primitiven und Referenztypen
- Vereinfachtes Pattern Matching
- Kompakterer, besser lesbarer Code

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - 23 Primitive Types in Patterns, instanceof, and switch (Preview)
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

JEP 480: Structured Concurrency

Java 19 bis 23, Third Preview

Aufgaben werden in parallel ausführbare Teilaufgaben mit klarer Lebensdauer zerlegt, sodass Fehler geordnet behandelt und Teilaufgaben sauber abgebrochen werden können.

GEDOPLAN

Structured Concurrency

- Zwei parallele Prozesse, Ergebnis wird kombiniert und zurückgegeben:

```
public UserOrder findWithExecutorService() {
  try (var service = Executors.newCachedThreadPool()) {
    Future<User> userFuture = service.submit(() -> findUser());
    Future<Order> orderFuture = service.submit(() -> fetchOrder());

    String user = userFuture.get();
    Integer order = orderFuture.get();

    return new UserOrder(user, order);
  }
}
```

- Was, wenn in einer der Teilaufgaben ein Fehler auftritt?
- Wie die anderen Teilaufgaben abbrechen?
- Wie Teilaufgaben abbrechen, wenn Ergebnis der ersten ausreicht?

GEDOPLAN

Structured Concurrency

```
try (var scope = new
                 StructuredTaskScope.ShutdownOnFailure()) {
  var findUserSubtask = scope.fork(() -> findUser());
  var fetchOrderSubtask = scope.fork(() -> fetchOrder());

  // wartet auf Abschluss aller Teilaufgaben
  scope.join();

  // Abbruch bei Fehler in einer Teilaufgabe
  scope.throwIfFailed();

  // Ergebnisse beider Teilaufgaben zusammenführen
  return new UserOrder(findUserSubtask.get(),
                       fetchOrderSubtask.get());
}
```

GEDOPLAN

Structured Concurrency

- `StructuredTaskScope.ShutdownOnFailure()` definiert Scope der Gesamtaufgabe
- Bei Fehler in einer Teilaufgabe werden alle anderen Teilaufgaben abgebrochen.
- Teilaufgaben werden mit `scope.fork()` in separatem Virtual Thread gestartet.
- `scope.join()` wartet auf Abschluss aller Teilaufgaben
- `scope.joinUntil(Instant)` ermöglicht Abbruch bei Timeout
- Ergebnisse werden mit `subtask.get()` abgeholt

GEDOPLAN

Structured Concurrency

- Wie Teilaufgaben abbrechen, wenn ein Ergebnis ausreicht?
- `StructuredTaskScope.ShutdownOnSuccess()`
- Bei Erfolg einer Teilaufgabe werden alle anderen Teilaufgaben abgebrochen.

```
try (var scope = new
                 StructuredTaskScope
                .ShutdownOnSuccess<NetworkConnection>()) {
  scope.fork(() -> tryToGetWifi());
  scope.fork(() -> tryToGet5g());
  scope.fork(() -> tryToGet4g());
  scope.join();
  return scope.result());
}
```

GEDOPLAN

Structured Concurrency

Was bringt's?

- Struktur der Aufgabe entspricht der Struktur im Code
- Aufgabe und Teilaufgaben bilden eine Einheit und sind leicht als solche erkennbar
- bessere Nachvollziehbarkeit und Wartbarkeit des Codes
- Aufgabe steuert Lebensdauer der Teilaufgaben
- bessere Steuermöglichkeiten bei Auftreten von Exceptions
- verwendet Virtual Threads -> schneller, weniger Ressourcenverbrauch

GEDOPLAN

Die besten Features

- Leichterer Einstieg und weniger Overhead
  - 22 Launch Multi-File Source-Code Programs
  - 23 (seit 21) Implicitly Declared Classes and Instance Main Methods (Third Preview)
- Kompakter und lesbarer
  - 22 (seit 21) Unnamed Variables & Patterns
  - 23 Markdown in Javadoc
  - 23 Primitive Types in Patterns, instanceof, and switch (Preview)
- Concurrency
  - 23 (seit 19) Structured Concurrency (Third Preview)
- Streams
  - 23 (seit 22) Stream Gatherers

GEDOPLAN

JEP 473: Stream Gatherers

Java 22 bis 23, Second Preview

Mit benutzerdefinierten intermediate operations in Streams können Daten effizient gruppiert, aggregiert oder transformiert werden.

GEDOPLAN

Stream Gatherers

- Bisher: Java 8 Streams
  - viele Intermediate Operationen
  - aber viele fehlen auch: `fold()`, `distinctBy()`, `window()`...

```
// not possible, just hypothetical
var result = Stream.iterate(0, i -> i + 1)
    // .windowFixed(4) // hypothetical
    .limit(3)
    .toList();
// result ==> [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

GEDOPLAN

Stream Gatherers

- Jetzt: Stream Gatherers
  - Neue intermediate operation `gather()`

```
var result = Stream.iterate(0, i -> i + 1)
    .gather(Gatherers.windowFixed(4))
    .limit(3)
    .toList();
// result ==> [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

GEDOPLAN

Stream Gatherers

Vorgefertigte Gatherers:

- `windowFixed()` – sammelt in Gruppen fester Größe
- `windowSliding()` – sammelt in überlappende Gruppen fester Größe
- `fold()` – kombiniert Elemente des Streams, ähnlich `reduce()`
- `scan()` – ähnlich `fold()`, aber für jede Kombination ein neues Ergebnis
- `mapConcurrent()` – wie `map()`, aber parallel in angegebener Anzahl von Virtual Threads

- Und Gatherers können auch selbst implementiert werden

GEDOPLAN

Stream Gatherers

- Vortrag von Michael Wiedeking auf dem JSD 2024:

  „Zusammengerafft: Die Stream-Gatherers“

  Raum Dresden

  15:20 Uhr

GEDOPLAN

Agenda

- Java 22 und Java 23: Neue Versionen – was ist drin?
- Im Detail: 3 Features, 4 Previews
- Fazit: Java 22 und Java 23 – Was können wir mitnehmen?
- Ausblick

GEDOPLAN

Java 22 und Java 23: Positives

- Java wird einfacher und attraktiver
- Viele schöne Verbesserungen in Syntax und APIs
  - Unnamed Variables & Patterns
  - Launch Multi-File Source-Code Programs
- Markdown – endlich lesbare und einfacher erstellbare Javadoc!
- Interessante Previews
  - Primitive Types in Patterns, instanceof, and switch
  - Stream Gatherers
  - Implicitly Declared Classes and Instance Main Methods
  - Structured Concurrency

GEDOPLAN

Java 22 und Java 23: Nicht ganz so Positives

- etwas dünn bezüglich wichtiger Neuerungen
  - 3 für den Anwendungsentwickler interessante neue Features
  - 8 Preview Features
  - 1 Incubator (7th incubator!)
- Interessante Features frühestens in einem Jahr als LTS verfügbar
- Dokumentation neuer Features teilweise sehr dürftig
- Wegfall der String Templates in Java 23 war unerwartet, zeigt aber deutlich, dass man Preview-Features nicht in Produktivcode einsetzen sollte.

GEDOPLAN

Agenda

- Java 22 und Java 23: Neue Versionen – was ist drin?
- Im Detail: 3 Features, 4 Previews
- Fazit: Java 22 und Java 23 – Was können wir mitnehmen?
- Ausblick

GEDOPLAN

Was können wir erwarten?

- Project Valhalla:
  - Anpassung an moderne Hardware (Multi-Core, Caching)
  - Erweiterung und Vereinheitlichung des Typsystems für JVM-basierte Sprachen
  - Lücke zwischen primitiven Typen und Objekttypen schließen
- Project Babylon:
  - Nutzung von GPU-Ressourcen zur massiven Parallelisierung in Java-Anwendungen, maschinelles Lernen, Grafikanwendungen

GEDOPLAN

Was können wir erwarten?

- Project Leyden: Verbesserung von Startup- und Warmup-Zeiten
  - Beschleunigung von Neustarts durch „Coordinated Restore at Checkpoint“ (CRaC)
  - Java-Anwendungen können ihren Zustand speichern und schnell wiederherzustellen

- Project Panama:
  - optimierte Interoperabilität zwischen dem JDK und fremden APIs ab
  - Vereinheitlichung der Nutzung von JNI und Foreign Function & Memory API

GEDOPLAN

Demos

- Demo-Projekt auf `github.com/GEDOPLAN/java23`

GEDOPLAN

Fragen?

GEDOPLAN

Danke!

GEDOPLAN

JEP 482: Flexible Constructor Bodies

Java 22 bis 23, Second Preview

In Konstruktoren dürfen Anweisungen nun auch vor dem Aufruf des Konstruktors der Oberklasse mittels `super()` oder eines anderen Konstruktors derselben Klasse mittels `this()` stehen.

GEDOPLAN

JDK Enhancement Proposal – JEP

- JEP: JDK Enhancement Proposal
- formelles Dokument, das eine neue Funktion oder Änderung für das JDK vorschlägt
- wird von der OpenJDK-Community eingereicht
- API-Änderungen, neue Sprachfunktionen, Änderungen der JVM
- ermöglicht organisierte Diskussion der Verbesserungen oder Änderungen