Coole neue Java Features – Java 17 bis 23 – Add On

https://github.com/Michaeli71/Best-Of-Java-17-23

Michael Inden

Head of Development, freiberuflicher Buchautor und Trainer

JEP 481: Scoped Values (Third Preview)

https://openjdk.org/jeps/481

JEP 481: Scoped Values

- Scoped Values sind eine moderne Alternative zu `ThreadLocal`-Variablen, die dazu dienen, Informationen anstatt Thread-spezifisch Kontext-spezifisch zu speichern.
- Scoped Values arbeiten mit Plattform- und Virtual Threads

- In Kombination mit virtuellen Threads sind Scoped Values oftmals eine bessere Alternative, weil sie die folgenden Vorteile gegenüber `ThreadLocal`-Variablen besitzen:
  - Sie sind nur für einen bestimmten Bereich (Scope) und eine spezifische Ausführung und damit auch Zeitabschnitt gültig.
  - Während der Ausführung sind sie unveränderlich, können aber danach neu gebunden werden.
  - Sie geben ihre Belegung automatisch an alle Kind-Threads weiter, etwa diejenigen, die von einen `StructuredTaskScope` erzeugt werden. Bei `ThreadLocal` wird dessen Wert in Form eines `InheritableThreadLocal` in die Kind-Threads kopiert.
  - Der letzte Punkt ist wichtig, weil es ja potenziell Millionen von virtuellen Threads geben kann und dann eine Kopie von Daten speichertechnisch ungünstig sein kann.

JEP 481: Scoped Values

- Bei der Ausführung der `performLogin()`-Methode werden aus dem als Parameter übergebenen Request die User-Daten extrahiert und dann per `where()` im Scoped Value hinterlegt.
- Mithilfe der Methode `run()` und einer Aktion in Form eines `Runnable` wird die Verarbeitung in dem Scope gestartet. Hier wird dadurch die Methode `performAction()` eines Services aufgerufen, die aber keine Parameter erhält:

```
public class LoginUtil
{
    public static final ScopedValue<User> LOGGED_IN_USER = ScopedValue.newInstance();

    // …

    public void performLogin(final Request request)
    {
        User loggedInUser = authenticateUser(request);
        ScopedValue.where(LOGGED_IN_USER,
                          loggedInUser).run(() -> service.performAction());
    }
```

```
    // …
```

JEP 481: Scoped Values

- **Auslesen der Kontextwerte über `get()`-Methode auf der statischen Variable:**

```
public class XyzService
{
    public void performAction() {
        var loggedInUser = LoginUtil.LOGGED_IN_USER.get();

        System.out.println("performing action with: " + loggedInUser);
        System.out.println("collected data: "  + retrieveDataFor(loggedInUser));
    }

    private String retrieveDataFor(User loggedInUser) {
        return "SOME DATA";
    }
}
```

- **=>**

```
performing action with: User[name=FALLBACK, pwd=[C@15aeb7ab]
collected data: SOME DATA
```

JEP 481: Scoped Values – Spezialfälle

- **Prüfe auf Wert inklusive Bereitstellung von Fallback oder Auslösen einer Exception:**

```
if (LoginUtil.LOGGED_IN_USER.isBound())
{
    var loggedInUser = LoginUtil.LOGGED_IN_USER.get()
    System.out.println("performing action with: " + loggedInUser);
}
else
{
    // perform fallback actions
}

var loggedInUser = LoginUtil.LOGGED_IN_USER.orElse(new User("FALLBACK",
        "PWD".toCharArray()));

var loggedInUser = LOGGED_IN_USER.orElseThrow(() ->
        new IllegalStateException("invalid user"));
```

JEP 481: Scoped Values – Exception Handling Java 22

- Nehmen wir an, `performCalculation()` kann eine `IOException` und damit eine Checked Exception auslösen. Dies muss auf folgende ungewöhnliche Weise behandelt werden:
-
```
try
{
    var result = ScopedValue.callWhere(ScopedValuesExample.LOGGED_IN_USER,
                                       user, Java23Examples::performCalculation);
    System.out.println("Calculated result: " + result);
}
catch (Exception e)
{
    if (e instanceof IOException ioe)
        handleIoException(ioe);

    throw new RuntimeException(e);
}
```
- Wenn man versucht, `catch (IOException ioe)` zu schreiben, resultiert das in einem Kommpilierfehler: `«Unhandled exception: java.lang.Exception.»`

JEP 481: Scoped Values – Exception Handling Java 23

- **Als Verbesserung bringt Java 23 eine Vereinfachung des Exception Handling:**

```
try
{
    var result = ScopedValue.callWhere(ScopedValuesExample.LOGGED_IN_USER,
                                       user, Java23Examples::performCalculation);
    System.out.println("Calculated result: " + result);
}
catch (IOException ioe)
{
    handleIoException(ioe);
}
```

- **Und noch besser für Unchecked Exceptions:**

```
var result = ScopedValue.callWhere(ScopedValuesExample.LOGGED_IN_USER,
                                   user,
                                   Java23Examples::performCalculationUnchecked);
System.out.println("Calculated result: " + result);
```

DEMO & Hands on

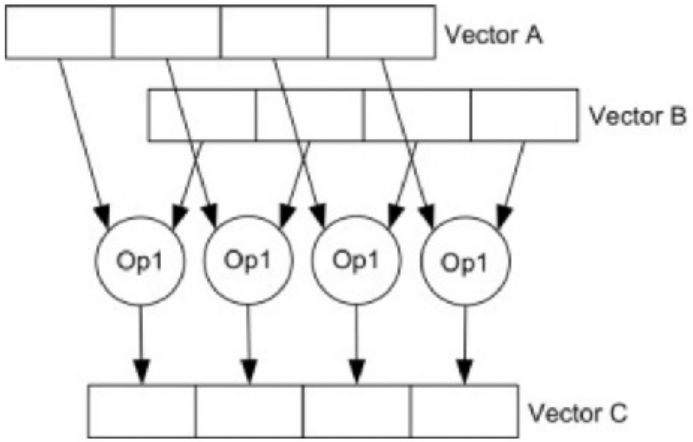
- `LoginUtil & XyzService`
- `ScopedValuesExample`
- `Java23Examples`

Incubator Features in Java 23

- JEP 469: Vector API (Eighth Incubator)

JEP 469: Vector API (Eighth Incubator in Java 23)

https://openjdk.org/jeps/469

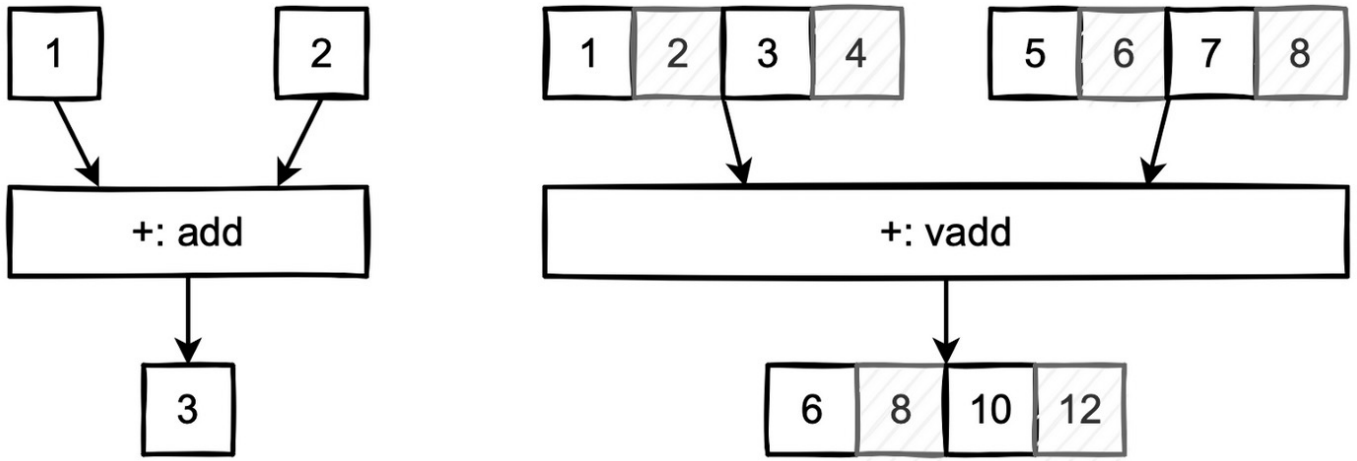


https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data

JEP 469: Vector API

- Dieser JEP hat nichts mit der Klasse java.util.Vector zu tun! Vielmehr geht es um eine plattformunabhängige Unterstützung von sogenannten Vektorberechnungen.

- Moderne Prozessoren können beispielsweise eine Addition oder Multiplikation nicht nur für zwei Werte, sondern für eine Vielzahl von Werten ausführen. Man spricht dabei auch von Single Instruction Multiple Data (SIMD). Die folgende Grafik visualisiert das Grundprinzip:



https://medium.com/@Styp/auto-vectorization-how-to-get-beaten-by-compiler-optimization-java-jit-vector-api-92c72b97fba3

JEP 469: Vector API

- Das Vector API zielt darauf ab, die Leistung von Vektorberechnungen zu verbessern.
- Eine Vektorberechnung besteht aus einer Folge von Operationen mit Vektoren. Dabei kann man sich einen Vektor wie ein Array von primitiven Werten vorstellen.
- Wollte man nun zwei Vektoren respektive Arrays mit einer mathematischen Operation verknüpfen, so würde man dazu herkömmlich eine Schleife über alle Werte nutzen.

```
int[] a = {1, 2, 3, 4, 5, 6, 7, 8};
int[] b = {1, 2, 3, 4, 5, 6, 7, 8};

var c = new int[a.length];
for (int i = 0; i < a.length; i++)
{
     c[i] = a[i] + b[i];
}
```

JEP 469: Vector API

- Mit dem Vector API lassen sich die Besonderheiten und Optimierungen in modernen Prozessoren ausnutzen. Dazu gibt man gewisse Hilfestellungen für die Berechnungen vor.

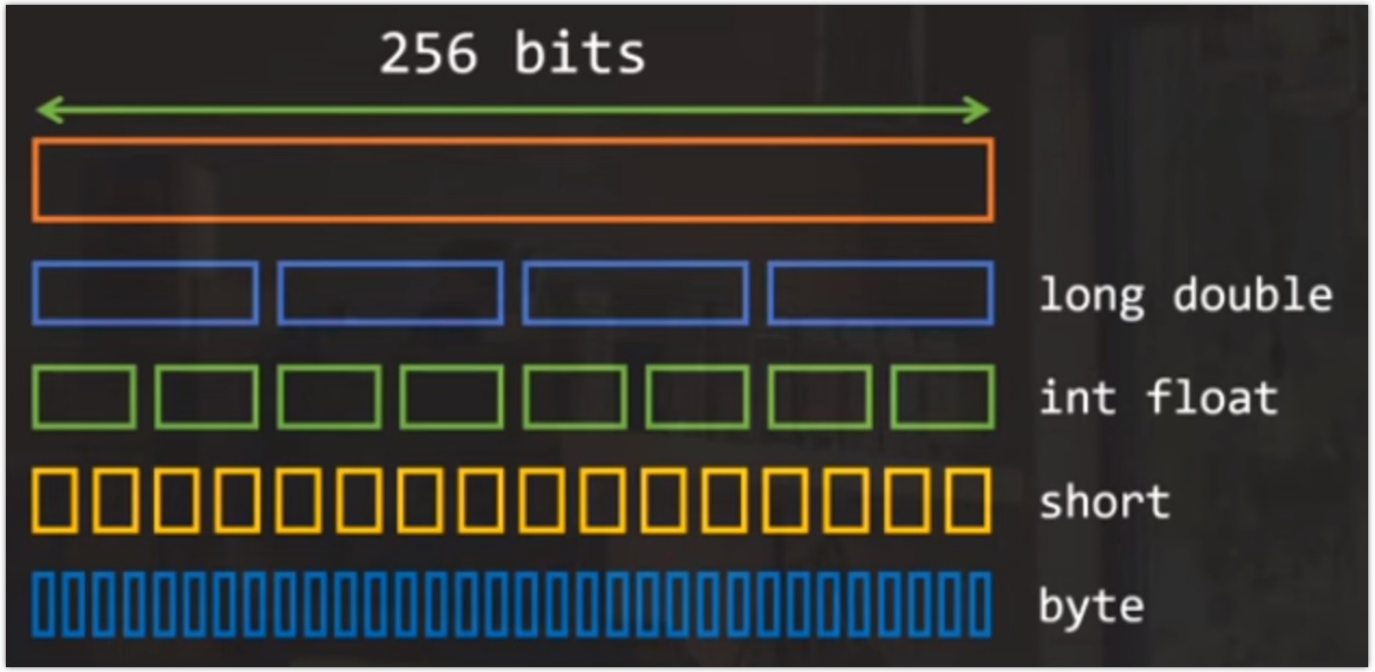
```
int[] a = {1, 2, 3, 4, 5, 6, 7, 8};
int[] b = {1, 2, 3, 4, 5, 6, 7, 8};

var c = new int[a.length];
var vectorA = IntVector.fromArray(IntVector.SPECIES_256, a, 0);
var vectorB = IntVector.fromArray(IntVector.SPECIES_256, b, 0);
var vectorC = vectorA.add(vectorB);
// var vectorC = vectorA.mul(vectorB);
vectorC.intoArray(c, 0);
```

- Das steuernde Element ist hier die Größe des Vektors, den wir durch das Argument `IntVector.SPECIES_256` auf 256 Bit festlegen.

- Danach kann dann die passende Aktion erfolgen, hier `add()` oder `mul()`.

JEP 469: Vector API

- **Einfluss der Vektor-Größe:**



- **Bevorzugen Sie: `IntVector.SPECIES_PREFERRED`**

https://openjdk.org/jeps/469

JEP 460: Vector API

- **Betrachten wir ein Beispiel aus JEP 448 und die Skalarberechnung als Ausgangsbasis:**

```
void scalarComputation(float[] a, float[] b, float[] c)
{
    for (int i = 0; i < a.length; i++)
    {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

- **Algorithmus ist absolut verständlich und leicht nachvollziehbar**

https://openjdk.org/jeps/469

JEP 469: Vector API

```
static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;

void vectorComputation(float[] a, float[] b, float[] c)
{
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length())
    {
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var vc = va.mul(va)
                   .add(vb.mul(vb))
                   .neg();
        vc.intoArray(c, i);
    }

    for (; i < a.length; i++)
    {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

```
WARNING: Using incubator modules: jdk.incubator.vector
result using scalar calculation: [-2.0, -8.0, -18.0, -32.0,
-50.0, -72.0, -98.0, -128.0, -162.0, -200.0]
result using Vector API: [-2.0, -8.0, -18.0, -32.0, -50.0,
-72.0, -98.0, -128.0, -162.0, -200.0]
```

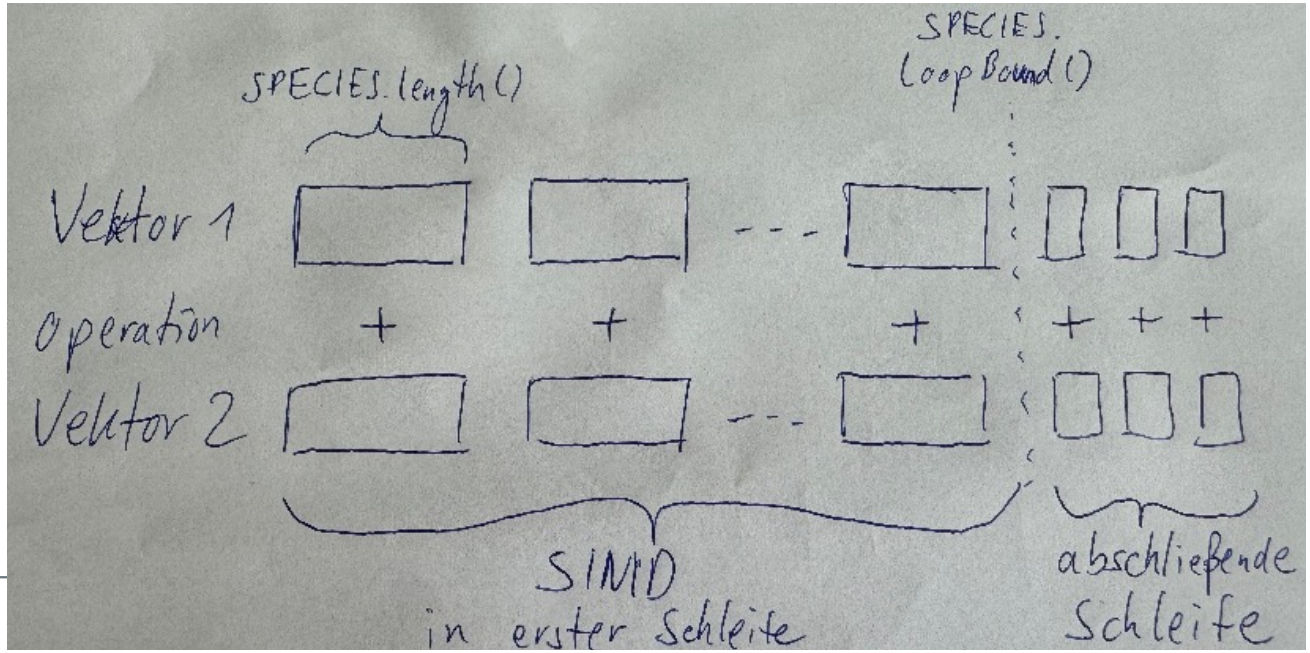
https://openjdk.org/jeps/469

https://medium.com/@Styp/java-18-vector-api-do-we-get-free-speed -up-c4510eda50d2

JEP 469: Vector API

```
void vectorComputation(float[] a, float[] b, float[] c)
{
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length())
    {
        var va = FloatVector.fromArray(SPECIES, a, i);
        var vb = FloatVector.fromArray(SPECIES, b, i);
        var vc = va.mul(va)
                   .add(vb.mul(vb))
                   .neg();
        vc.intoArray(c, i);
    }

    for (; i < a.length; i++)
    {
        c[i] = (a[i] * a[i] +
                b[i] * b[i]) * -1.0f;
    }
}
```

Wenn die Ausgangsdaten nicht perfekt auf die Schrittweite passen, müssen die Aktionen entsprechend aufgeteilt werden, damit sie passen.



https://openjdk.org/jeps/469

JEP 469: Vector API

- **Lassen Sie uns die letzte Schleife auskommentieren:**

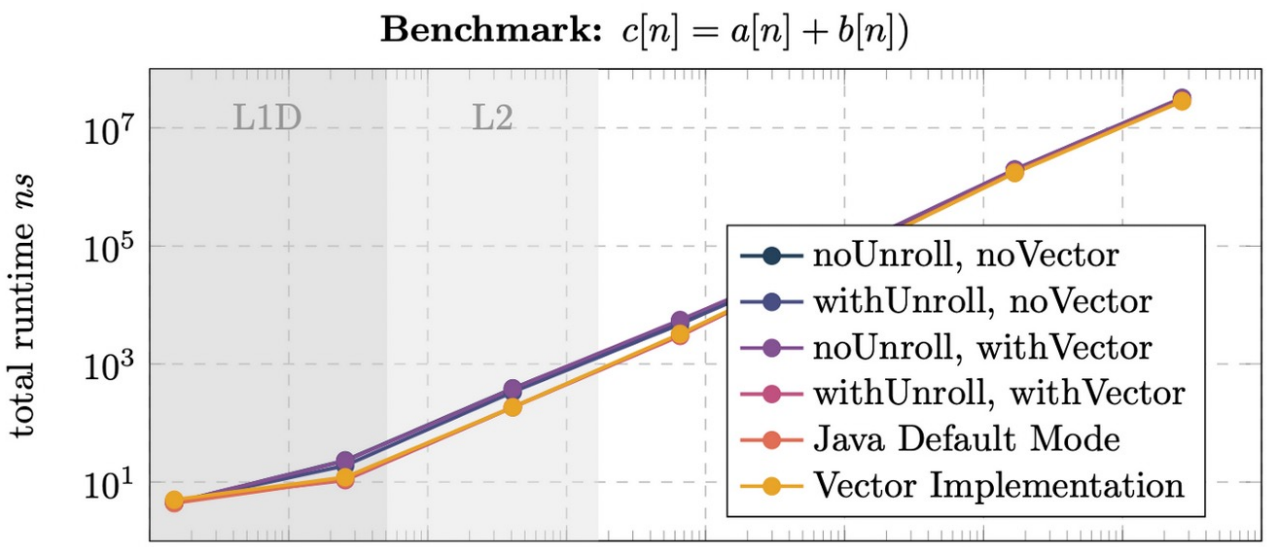
```
void vectorComputation(float[] a, float[] b, float[] c)
{
    …

    // for (; i < a.length; i++)
    // {
    //     c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    // }
}
```

- **=>**

```
WARNING: Using incubator modules: jdk.incubator.vector
result using scalar calculation: [-2.0, -8.0, -18.0, -32.0, -50.0, -72.0, -98.0, -128.0, -162.0,
-200.0]
result using Vector API: [-2.0, -8.0, -18.0, -32.0, -50.0, -72.0, -98.0, -128.0, 0.0, 0.0]
```

- **Einige Werte würden dann einfach nicht berechnet und hätten daher den Wert 0,0.**
- **Der Einfluss der letzten Schleife auf die Leistung ist vernachlässigbar, da bei sehr großen Vektoren nur ein minimaler Bruchteil der Werte damit berechnet wird. Dies ist jedoch unbedingt erforderlich, um eine korrekte Berechnung für jede Vektorlänge zu gewährleisten.**

JEP 469: Vector API Benchmarking



Method	Peak Speed-Up
No Unroll, No Vector	1.00x
With Unroll, No Vector	1.22x
No Unroll, With Vector	1.01x
With Unroll, With Vector	2.15x
Java Default	2.06x
Vector Implementation	2.08x

https://medium.com/@Styp/auto-vectorization-how-to-get-beaten-by-compiler-optimization-java-jit-vector-api-92c72b97fba3

JEP 469: Vector API (in Console and JShell)

```
$ java --enable-preview --source 23 --add-modules jdk.incubator.vector \
       src/main/java/api/VectorApiExample.java

WARNING: Using incubator modules: jdk.incubator.vector
[2, 4, 6, 8, 10, 12, 14, 16]



$ jshell --enable-preview --add-modules jdk.incubator.vector
|  Willkommen bei JShell – Version 23
|  Geben Sie für eine Einführung Folgendes ein: /help intro

jshell> import jdk.incubator.vector.FloatVector;

jshell> import jdk.incubator.vector.IntVector;

jshell> import jdk.incubator.vector.VectorSpecies;
```

DEMO & Hands on

Questions?

Thank You