Modern Java in 2026

Marek Landa

B0B36PJV

2026

Outline

1. Today's setup
2. Records
3. Lambda functions
4. Flexible Constructors
5. Pattern matching
6. Sealed classes
7. Virtual threads & Scoped Values
8. Extras setup
9. Extras: Vector API, Generational ZGC, AOT, Profiling...

Today's setup

- all examples will use JDK 25

Records

- Immutable data carriers
- Compact syntax
- Automatic equals(), hashCode(), toString()

Lambda functions & Stream API

```
@FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);

}
```

- JDK 8+
- Anonymous functions
- Syntactic sugar for functional interfaces
- Predicate, Function, Consumer, Supplier
- Map, filter, reduce

Demo time! #1: Records and Lambda functions

Flexible Constructors

- JDK 25 (JEP 513)
- statements before super() call
- fail fast (finally we can check parameters before initialization, yay!)

Flexible Constructors Demo

```java
public class Transaction {  1 usage  1 inherit
    private final double amount;  1 usage

    public Transaction(double amount) {
        this.amount = amount;
    }
}
```

```java
public class SecureTransaction extends Transaction {  1 usage  Marek Landa *
    private final String txId;  1 usage

    public SecureTransaction(double amount, String txId) {  1 usage  Marek Landa
        // Before Java 25, `super()` had to be the VERY FIRST line.
        // Now, we can perform fast-failing validation BEFORE invoking the superclass constructor.
        if (amount <= 0) {
            throw new IllegalArgumentException("Amount must be positive");
        }
        if (txId == null || txId.isBlank()) {
            throw new IllegalArgumentException("Transaction ID required");
        }

        super(amount); // Super call can happen later!
        this.txId = txId;
    }
}
```


I HAVE A VISION THAT OUR NEXT PRODUCT WILL BE A TABLET COMPUTER THE SIZE OF A DIME.
USERS WILL LICK IT AND ATTACH IT TO THEIR EYEBALLS.
CAN YOU FINISH THAT IN A MONTH?
I CAN FAIL AT ANY SPEED YOU LIKE.
Dilbert.com DilbertCartoonist@gmail.com
© 2013 Scott Adams, Inc./Dist. by Universal Uclick
1-23-13

Pattern matching

- most Pattern Matching features available since JDK 21, but we are still getting more
- alternative to tedious use of instanceof and casting

Pattern matching

- Record deconstruction (we ignore irrelevant components of record using underscore "_")

```
switch (item) {
    case Book(var title, _, _, var pages, _) -> "Book: " + title + "pages: " + pages;
    case Magazine(var title, _) -> "Magazine: " + title;
}
```

- Guarded patterns

```
switch (item) {
    case String s when s.length() < 6 -> "Short string: " + s.toUpperCase();
    case String s -> "Long String: " + s.toUpperCase();
}
```

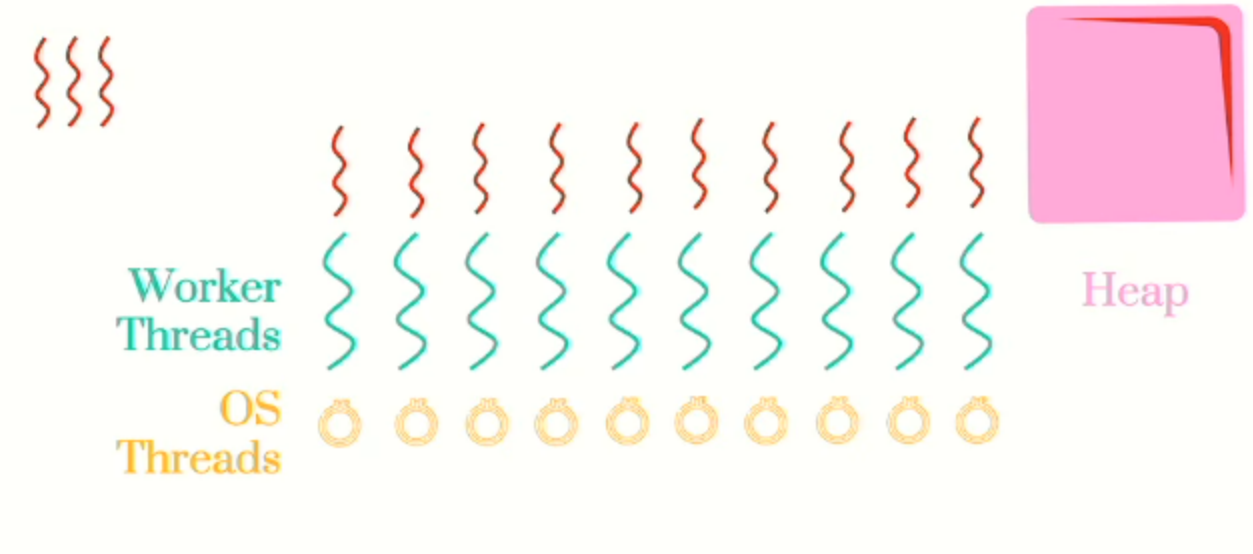
Demo time! #2: Pattern matching

Sealed Classes

- Enforce controlled inheritance hierarchies (we want to restrict which classes can inherit from our class)
- Compiler ensures exhaustiveness in switch/instanceof checks
- Child must be in the same module
- Child must explicitly state whether it can be further extended, or not

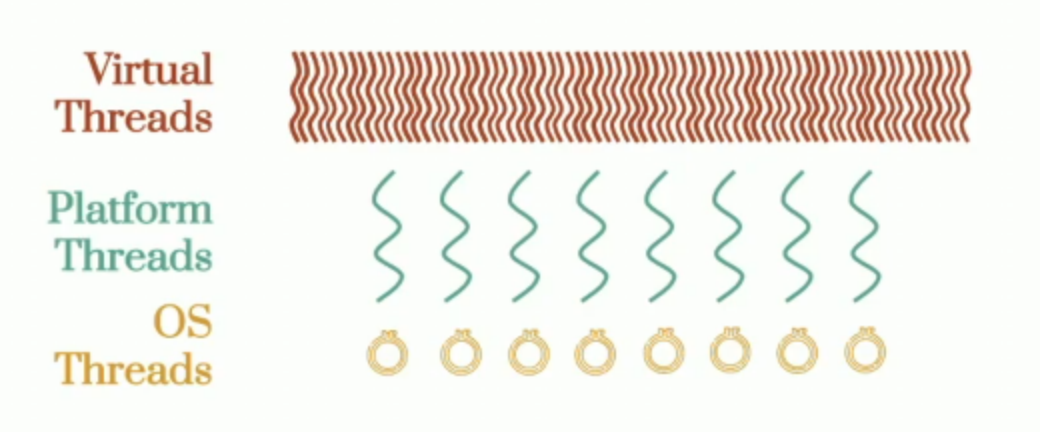
```
public sealed class Shape permits Rectangle, Circle {}

non-sealed class Rectangle extends Shape {} // can be further extended
final class Circle extends Shape {} // non extendable (like `java.lang.String`!)
class Square extends Rectangle {} // not sealead anymore
```

Platform Threads



- OS schedules threads to CPU cores
- if thread is blocked => CPU core is idle
- we need many OS threads for many concurrent requests
- threads are expensive

Virtual Threads



- JVM schedules threads
- if thread is blocked => JVM uses other threads to do other work, so CPU is not idle
- thread creation is much cheaper, does not take as much memory

Scoped Values

- shared among threads in a defined scope (parent thread and all its children)

```
// Immutable, safe, and automatically cleaned up when the scope exits.
    public final static ScopedValue<String> REQUEST_ID = ScopedValue.newInstance();
```

```
ScopedValue.where(REQUEST_ID, "req-998877").run(() -> {
            //spawn more threads
            try (var scope = StructuredTaskScope.open())
            {
                // All the subtasks can access REQUEST_ID without passing it as parameter!
                var userTask = scope.fork(() -> fetchUserProfile());
                var balanceTask = scope.fork(() -> fetchAccountBalance());
                var orderTask = scope.fork(() -> fetchRecentOrders());
            }
```

Demo time! #3: Virtual Threads and Scoped Values

Summary

- Records: Immutable data carriers without boilerplate code
- Lambda functions: Syntactic sugar for anonymous methods
- Stream API: Functional processing of collections
- Flexible Constructors: Execute statements before calling super()
- Pattern matching: Destructuring objects safely and concisely
- Sealed interfaces: Restrict which classes can implement an interface
- Switch expressions: Switch statements that return values
- Virtual threads: Lightweight threads managed by the JVM

Sources

- inside.java
- https://jug.cz/

Czech JUG

Thank you for your attention!

Extras setup

- we will use JDK 26 with preview features enabled (`--enable-preview` as VM argument)

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.13.0</version>
    <configuration>
        <compilerArgs>
            <!-- needed for preview features -->
            <arg>--enable-preview</arg>
            <!-- needed for Vector API -->
            <arg>--add-modules</arg>
            <arg>jdk.incubator.vector</arg>
        </compilerArgs>
    </configuration>
</plugin>
```

Bonus: Preview & Incubator Features in JDK 25

Bonus 1: The Vector API (JEP 508)

- **SIMD (Single Instruction, Multiple Data)** hardware acceleration. Process multiple array elements in a single CPU cycle.

(Switch to IDE: Show `BonusVectorApiDemo.java` *)*

Bonus 2: Production Profiling & AOT

- **AOT Command-Line Ergonomics (JEP 514)**: Create an Ahead-of-Time cache in one step: `java -XX:AOTCacheOutput=app.aot ...` for instant startup times.
- **JFR Method Timing & Tracing (JEP 520)**: Built-in, low-overhead method profiling directly via Java Flight Recorder.