Functional JDK 17

Jan Ypma

October 26, 2021

Contents

1 Introduction

1.1 About today's presentation

- *Functional JDK 17*
  - Selection of new JDK features that fit well for functional programming
  - Background and introduction to reactive streams
- A little about me, Jan Ypma
  - `jan@ypmania.net`
  - Independent software developer
  - Scala / Java, C++ (embedded), a little Rust, Lisp
- This presentation
  - https://github.com/jypma/java-17-demo

2 Recent new JDK features

2.1 Records (Java 16)

Declare an immutable class with constructor, getters, `equals`, and `hashCode()`:

```
record Point(int x, int y) { }
```

is equivalent to

```
class Point(int x, int y) {
    private final int x;
    private final int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int x() { return x; }
    int y() { return y; }

    @Override boolean equals(Object other) { ... }
    @Override int hashCode() { ... }
}
```

2.1.1 Demo

This means that we can do

```
record Point(int x, int y) {}
```

```
var p1 = new Point(1,2)
```

Use the getters:

```
"P1 is at " + p1.x() + ", " + p1.y()
```

Check for equality:

```
"P1 is equal to itself: " + p1.equals(new Point(1,2))
```

2.1.2 Summary

Use cases

- Data Transfer Objects (API client or server, database input or results)
- Nicer replacement of *tuples* (method-local records are allowed)

Limitations

- Still no immutable collections to use as field types
  - Use the VAVR library instead, more about that later
- No *copy* constructor (apparently being worked on)
  - Curiously, both Kotlin and Java allow this, using default arguments and a generated `copy` function:

```
val newPoint = p1.copy(x = 42)
```

2.2 Sealed classes

Interface or abstract class is marked as having a fixed set of implementations.

2.2.1 Implemented in different source files

```
package com.example.geometry;

public abstract sealed class Shape
    permits com.example.polar.Circle,
            com.example.quad.Rectangle,
            com.example.quad.simple.Square {
  // ... Shape abstract implementation...
}
```

2.2.2 Implemented in the same source file

```
abstract sealed interface Shape {
    record Circle(int diameter) implements Shape { /* ... */ }
    record Box(int width, int height) implements Shape { /* ... */ }
}
```

Each class implementing a `sealed` base can be either

- `final` (sealing the hierarchy here. All `record` classes are `final`.)
- `sealed` (implying further sub-types extending this one)
- `non-sealed` (breaking the promise of this being a sealed hierarchy)

2.2.3 Can't extend outside of what's sealed in

```
class OtherShape implements Shape {

}
```

2.3 "Pattern matching" for switch

Adds a type check to `switch` branches.

Let's recap and see how `switch` has evolved through recent Java versions.

2.3.1 Switch in Java 7

Simple replacement for `goto`, with mostly same semantics and syntax as `C`.

```
int value = 5;

switch(value) {
    case 1:
        System.out.println("One");
        break;
    case 5:
        System.out.println("five");
        break;
    default:
        System.out.println("Unknown");
}
```

2.3.2 Java 12: Strings, expressions

```
String day = "Tuesday";
```

We can now switch on `String`, have multiple values in one branch, and return as an expression.

Note: The following no longer compiles with Java 13+ (which requires `yield` instead of `break`).

```
switch(day) {
    case "Monday", "Tuesday":
        break "Week day";

    default:
        break "Unknown";
}
```

2.3.3 Java 12: Arrows for expressions

Instead of `break` or `yield`, arrows can be used to write a `switch` expression.

```
switch(day) {
    case "Monday", "Tuesday" -> "Week day";
    default -> "Unknown";
}
```

2.3.4 Java 13: Yield instead of break

Here's the Java 13+ equivalent:

```
switch(day) {
    case "Monday", "Tuesday":
        yield "Week day";
```

```
    default:
        yield "Unknown";
}
```

2.3.5 Java 17: Pattern matching objects, and guards

We can now check the type of an object, including additional constraints, right inside a `switch` case.

```
Shape shape = new Shape.Box(10, 5)

switch(shape) {
  case Shape.Circle c -> "It's a circle with diameter " + c.diameter();
  case Shape.Box b && b.width() == b.height() -> "It's a square of size " + b.width();
  case Shape.Box b -> "It's a box of size " + b.width() + " by " + b.height();
}
```

2.3.6 Exhaustiveness check

Since we defined `Shape` as a `sealed` class, the compiler will now inform us if we forget to add a case.

```
switch(shape) {
  case Shape.Box b && b.width() == b.height() -> "It's a square of size " + b.width();
  case Shape.Box b -> "It's a box of size " + b.width() + " by " + b.height();
}
```

2.3.7 Case branch for null

A case branch for `null` is now allowed (but, please, don't). And `default` still doesn't handle `null` (this is unchanged).

(set this to non-null to compile the `switch` below)

```
Object nothing = null;

switch (nothing) {
    case null     -> "null!";
    case String s -> "String";
    default       -> "Something else";
}
```

2.3.8 Limitations

- No decomposition
  - Can't match nested object graphs

```
record Drawing (Shape shape, int color)

switch (myDrawing) {
    // Does not compile:
    case Drawing(Shape.Box box, color) ->
}
```

2.4 Shenandoah GC and ZGC

2.4.1 Traditional garbage collectors

- Parallel GC
  - Stop-the-world GC for Young and Old generation
- Concurrent Mark-Sweep GC
  - Stop-the-world GC for Young, concurrent for Old generation
  - No compaction of Old generation
- G1 garbage collector
  - Stop-the-world GC for Young, concurrent mark for Old generation, stop-the-world compaction in segments
  - Configurable GC pauses: either shorter pauses, or less CPU wasted on GC
  - Default since Java 9
  - Problematic on large heaps or high allocation counts

2.4.2 ZGC and Shenandoah GC

- Scalable, low-latency GC
- No generations
- Concurrent mark *and* compaction

2.4.3 ZGC

- Since Java 11, but only on 64-bit linux (no compressed pointers)
- Store objects in ZPages (small, medium, large), compact when almost all objects in a page are dead
- Clever x86 JVM pointer tricks (*colored* pointers)
- More info on OpenJDK wiki

2.4.4 Shenandoah GC

- Developed by Red Hat
- Since Java 12 (but not in Oracle builds), but backported to 11 and 8
- Architecture independent (windows, linux and macOS)
- Derived from G1 (same marking), but divides heap into (many) *regions*
- Metadata in JVM object header
- More info on OpenJDK wiki
- So which one should I use?
  - Both ZGC and Shenandoah will probably improve your latencies
  - Try both!

2.5 macOS / AArch64 port

- Recent apple computers have 64-bit ARM processors, but don't run Linux
- There already was an `aarch64` port for Linux
- Java 17 brings native support for `aarch64` under MacOS

3 Practical reactive streams

3.1 Reactive manifesto

- Published in 2014, intends to push software systems to be better-behaved.
  - **Responsive**: *The system responds in a timely manner if at all possible.*
  - **Resilient**: *The system stays responsive in the face of failure.*
  - **Elastic**: *The system stays responsive under varying workload.*
  - **Message driven**: *Establish a boundary between components that ensures loose coupling, isolation and location transparency.*

3.2 Concurrency

3.2.1 Directness and laziness

- Direct value: `Person p`
  - Value is already calculated
  - This is good, we know there's no more I/O
- Direct asynchronous value: `CompletionStage<Person> p`
  - Computation already in progress: problematic
- Lazy value: `Supplier<Person> p`
  - Computation doesn't start until invoking `p.get()`
  - Nice, but not asynchronous
- Lazy asynchronous value: (no plain Java type) `"Supplier<CompletionStage<Person» p"`
  - All *Akka Streams* types are lazy and asynchronous (but multi-valued)
  - Hence, Akka can optimize and change a stream before starting it
    - For example, adding retry behavior to stream components

3.2.2 Reactive manifesto: Threads

- Synchronous method calls
  - Hard to make *responsive* (can't really abort a thread, unless all code constantly checks time)
  - Hard to make *resilient* in Java (failure is realistically limited to exceptions, of which many are unchecked and invisible)
  - Not *message-driven* (methods return values synchronously, and/or have side effects)
- Doesn't affect *elastic*

3.2.3 Reactive manifesto: Futures

- Java calls them CompletionStage (`CompletionStage<T>, CompletableFuture<T>,`)
  - Handle to an on-going background computation
  - Hard to make *responsive* (computation already started, not cancellable in practice)
  - Even harder than Threads to make *resilient* in Java (exceptions are now hidden behind `CompletionExceptio` plus cancellation)
  - Can model *message-driven* by having future callbacks
- Doesn't affect *elastic*

3.2.4 Reactive manifesto: Functional effect systems

- Think of "`CompletionStageRecipe<T,E>`"
  - Description of (not yet started) background computation
  - All of *responsive*, *elastic* (since description can be altered before launch) and *message-driven*
  - Very active in the Scala world (`cats-effect`, `ZIO`)
  - Not so much in plain Java or Kotlin, potentially due to missing language constructs
- Doesn't affect *elastic*

3.2.5 Reactive manifesto: Reactive streams

- Reactive streams
  - Covers a variety of independent frameworks
    * *rxJava* (2014), porting Microsoft's "reactive extensions" to Java
    * *Akka Streams* (2015), building on Akka with a component-based streaming framework
    * *Project Reactor* (2015), built by Spring directly decorating `java.util.concurrent.Flow`
    * Many others
  - Interoperability through `java.util.concurrent.Flow`
    * Low-level
  - We'll look at Akka Streams today
    * Trivially *responsive* (real time is a core element of streams)
    * *Resilient* due to well-defined error propagation and handling
    * Gives some *elastic* guarantees due to bounded processing (more on that later)
    * Integrates well in *message-driven* architectures (native actors support)

3.3 Immutability

- Asynchronous processing on data needs guarantees
  - Locks? Not if each and every data object is processed concurrently.
- *"I promise I won't change this object anymore"* just isn't cutting it
- Need actual immutability
  - Have compiler help guaranteeing objects won't be changed

- No setters
- `record` anyone?
- Can't use `java.util.List` or `java.util.Map`

3.3.1 VAVR

- Functional library for Java, focusing on immutable values
- JavaDoc shows collection, control and concurrency primitives

Create an immutable sequence:

```
Seq<Integer> seq = Vector.of(1, 2, 3)
```

```
seq.forEach(i -> System.out.println(i))
```

- All VAVR collections are *persistent data structures*, for example
  - `List` (single-linked list)
  - `Vector` (bit-mapped trie)
  - `HashMap` (hash array mapped trie)

3.4 Null-free style

- Nobody likes `NullPointerException`
- Reactive streams, and most functional libraries, don't allow (or like) `null` as values
- So, why are we still using `null` to indicate optionality?
  - Use `java.util.Optional` or the more powerful `io.vavr.control.Option` (or `io.vavr.control.Either`) instead.

```
Option<User> getUserIfExists(userId: long) {
  // ...
}
```

- In case of optional method arguments, consider method overloading instead of passing `null` (but `Option` is also fine here).

```
void saveUser(String userName, String petName) {
 // Save a user who signed up together with their pet.
}

void saveUser(String userName) {
 // Save a user who signed up by themselves.
}
```

- In short
  - The word `null` should never occur in your pull requests for new code
  - Only exception is interacting with external `null`-loving libraries

3.5 Akka streams introduction

- **Akka Streams**: Composable reactive streams framework
- Implemented on top of Akka *actors* (but invisibly so). You need an `ActorSystem` to launch streams:

```
ActorSystem system = ActorSystem.create("Demo")
```

- Streams form a graph, built using components called *graph stages*
  - Type-safe *input(s)* and/or *output(s)*
  - Number of inputs and outputs defines its *shape*
- Stream objects are descriptions only, and need to be *materialized* to actually do something

3.5.1 Source

source.gif

- Has a single output of type `T`, no inputs
- Emits elements

For example, a source that emits the same element every second:

```
Source<String, Cancellable> everySecond = Source.tick(Duration.ofSeconds(1),
↪  Duration.ofSeconds(1), "tick!")
```

Or a source that emits all integers up to one million, as fast as the stream can use them:

```
Source<Integer, NotUsed> integers = Source.range(1, 1000000)
```

3.5.2 Flow

flow.gif

- Has a single input of type `T`, and one output of type `U`
- Typically emits elements on its output as it receives them in the input

For example, a flow that converts integers to strings:

```
Flow<Integer,String,NotUsed> intToString = Flow.<Integer>create().
  map(i -> i.toString())
```

3.5.3 Flow (operators)

But we have more complex, useful operators. For example, process a sliding window of 10 elements: *(we'll map to VAVR's `Vector` to ensure immutability)*

```
Flow<Integer, Seq<Integer>, NotUsed> intSliding = Flow.<Integer>create().
  sliding(1, 10).
  map(Vector::ofAll)
```

Or, group elements up to a certain count, *OR* until some time has elapsed:

```
Flow<Integer, Seq<Integer>, NotUsed> intGrouped = Flow.<Integer>create().
  groupedWithin(256, Duration.ofSeconds(1)).
  map(Vector::ofAll)
```

3.5.4 Flow (connecting)

- Connecting a `Flow` to a `Source` (of compatible type) can be viewed as a `Source` (of the Flow's output type)

For example, let's hook up our `integers` source to the `intToString` flow:

```
Source<String,NotUsed> strings = integers.via(intToString)
```

In order to test, let's print the first 10 elements which that flow produces.

```
strings.
  take(10).
  runForeach(System.out::println, system).
  toCompletableFuture().get(1, TimeUnit.SECONDS)
```

3.5.5 Sink

sink.gif

- Has a single input of type `T`
- Typically "consumes" the elements

```
Sink<String, CompletionStage<Done>> printStrings = Sink.<String>foreach(s ->
↪  System.out.println(s))
```

- Connecting a `Source` to a `Sink` leaves no inputs or outputs
  - Akka calls this a `RunnableGraph`

```
RunnableGraph<NotUsed> graph = strings.to(printStrings)
```

- We won't run the above graph, since there's no `CompletionStage` indicating when it's done (only `NotUsed`)

3.5.6 Materialization

- Instances of graphs (`Source`, `Sink`, ...) are *descriptions*, and don't run yet
- Need to invoke `RunnableGraph.run()` (or one of the shorthands on `Source`) to actually start a stream
- Running a stream gives a *materialized value*
  - `Source<T, M>.` emits elements of type `T`, results in a value `M` when started
  - `Sink<T, M>.` consumes elements of type `T`, results in a value `M` when started
  - `RunnableGraph<M>.` results in a value `M` when started (`.run()` returns `M`)
- Now, we can construct `graph` again, but this time use the materialized value of the `sink`
  - By default, `.to()` uses the materialized value of the `source`

```
RunnableGraph<CompletionStage<Done>> graph = strings.take(10).toMat(printStrings,
↪  (sourceMat, sinkMat) -> sinkMat)
```

```
graph.run(system).toCompletableFuture().get(1, TimeUnit.SECONDS)
```

3.5.7 Bounded processing

- When writing data processing software, always make sure to be explicit in how much *of each* you want in memory
- Akka makes this explicit wherever possible
  - `groupedWithin` takes a maximum amount of elements AND a duration. There is no variant that only takes a duration.

```
source.groupedWithin(100, Duration.ofSeconds(1))
```

- `groupBy(Integer maxStreams, Function<T,K> key)` (grouping substreams by key) needs to specify the maximum number of open streams
- `mapAsync` (allowing to map each element to a `CompletionStage` 's result) needs to specify the number of in-flight elements
- Akka helps you towards bounded processing

3.5.8 Custom graph stages

- Writing your own `Source`, `Flow` or `Sink` is easy and well-documented
- These are ideal building blocks for data-processing systems
  - Encapsulate resource handling inside your building block
  - Well-defined error handling and propagation

3.5.9 Use cases for reactive streams

Good reasons to reach for reactive streams:

- Variance in iteration size
  - Being able to handle, simultaneously, both *many small* requests but also *few large* requests with the same code
- Heterogeneous systems
- Predictable memory usage

3.6 Case: Kafka processing with Akka Streams

3.6.1 Preparation

- Kafka is running locally, started from docker-compose.yml
- Let's make sure we have an empty topic to play with:

```
kafkactl delete topic demo 2>/dev/null
kafkactl create topic demo
```

- Akka can make use of Kafka through the Alpakka Kafka library

3.6.2 Writing to a topic

- Let's use akka's `Producer.plainSink` in a simple example

```
void writeToTopic() throws Exception {
    final ProducerSettings<String, String> producerSettings =
        ProducerSettings.create(system, new StringSerializer(), new StringSerializer())
        .withBootstrapServers("localhost:9092");

    Source.range(1, 10)
        .map(number -> number.toString())
        .map(value -> new ProducerRecord<String, String>("demo", value))
        .runWith(Producer.plainSink(producerSettings), system)
        .toCompletableFuture().get(10, TimeUnit.SECONDS);
}
```

```
writeToTopic()
```

- Let's see if they arrived:

```
kafkactl consume demo --from-beginning --exit
```

3.6.3 Producer variants

The Alpakka `Producer` class has several ways of defining a Kafka producer.

- `Producer.plainSink`: Sends `ProducerMessage` objects to Kafka
  - Suitable when sending to Kafka is the last step in a stream
- `Producer.flexiFlow`: Sends `Envelope` to Kafka, and passes it on down-stream
  - An `Envelope` can potentially contain more than one Kafka message, and an arbitrary *context* object
  - Useful when you need to do more after sending to Kafka
- `Producer.committableSink`: Automatically *commits* messages read from another Kafka topic
  - Useful in *consume* - *process* - *produce* type flows

3.6.4 Consuming from a topic

- Let's use the Alpakka `Consumer.plainSource` in a simple example

```
Seq<String> readFromTopic() throws Exception {
    final ConsumerSettings<String, String> consumerSettings =
        ConsumerSettings.create(system, new StringDeserializer(), new
        ↪  StringDeserializer())
        .withBootstrapServers("localhost:9092")
        .withGroupId("group1")
        .withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

    return Consumer.plainSource(consumerSettings, Subscriptions.topics("demo"))
        .take(1)
```

```
        .map(record -> record.value())
        .runWith(Sink.seq(), system)
        .thenApply(Vector::ofAll)
        .toCompletableFuture()
        .get(20, TimeUnit.SECONDS);
}
```

(demo is unfortunately not working due to JShell limitations)

```
readFromTopic()
```

3.6.5 Consumer offset management

Kafka can store the offset for consumer groups, or consumers can provide (and store) it themselves.

- Store offset in Kafka
  - `Consumer.committableSource`
- No offset management
  - `Consumer.plainSource(settings, Subscriptions.topics("topic"))`
- Do your own offset management
  - `Consumer.plainSource(settings, Subscriptions.assignmentWithOffset(new TopicPartition("topi partition0), fromOffset)))`
  - After each element, store its partition and offset in your own storage

3.6.6 Transactions and "exactly-once" processing

- Recent Kafka versions implement an extension that allows clients to atomically
  - Consume from one topic
  - Produce results to another topic
- Kafka refers to this both as *transactions* and *exactly-one processing*
- This feature can be used from Akka using the Alpakka `Transactional` class, e.g.

```
Transactional.source(consumerSettings, Subscriptions.topics(sourceTopic))
    .via(business())
    .map(
        msg ->
        ProducerMessage.single(
            new ProducerRecord<>(targetTopic, msg.record().key(), msg.record().value()),
            msg.partitionOffset()))
    .toMat(
        Transactional.sink(producerSettings, transactionalId),
        Consumer::createDrainingControl)
    .run(system);
```

- `PartitionOffset` holds the partition number and offset of the originally consumed message
- This is passed as *context* argument to the `ProducerRecord`

3.7 Case: RabbitMQ processing with Akka Streams

3.7.1 Preparation

- RabbitMQ is running locally, started from docker-compose.yml
- Communication is over AMQP, using akka's Alpakka AMQP library

3.7.2 Writing to a topic

```
Seq<WriteResult> writeToTopic() throws Exception {
    var settings = AmqpWriteSettings.create(AmqpLocalConnectionProvider.getInstance())
        .withRoutingKey("demo-queue")
        .withDeclaration(QueueDeclaration.create("demo-queue"))
        .withBufferSize(10)
        .withConfirmationTimeout(Duration.ofMillis(200));

    return Source.range(1, 10)
        .map(number -> number.toString())
        .map(value -> WriteMessage.create(ByteString.fromString(value)))
        .via(AmqpFlow.createWithConfirm(settings))
        .runWith(Sink.seq(), system)
        .thenApply(Vector::ofAll)
        .toCompletableFuture().get(10, TimeUnit.SECONDS);

}
```

```
writeToTopic()
```

3.7.3 Producer variants

RabbitMQ (and its underlying AMQP protocol) allows varying degrees of consistency when producing messages.

- *Fire-and-forget* : Fastest performance, but messages may be lost in case of broker or network issues
  - Use `AmqpFlow.apply`
- *Publisher confirms*: Asynchronous message from RabbitMQ to client (after fsync)
  - Use `AmqpFlow.withConfirm` (setting `bufferSize` to the allowed number of parallel in-flight messages)
  - Use `AmqpFlow.withConfirmUnordered` for maximum throughput, sacrificing ordering guarantees
- *Transactions*
  - Traditionally considered "slow" by RabbitMQ
  - Not directly supported by the Alpakka library (just use publisher confirms)

3.7.4 Reading from a topic

```
Seq<String> readFromTopic() throws Exception {
    var bufferSize = 10;
    Source<ReadResult, NotUsed> amqpSource =
        AmqpSource.atMostOnceSource(
            NamedQueueSourceSettings.create(AmqpLocalConnectionProvider.getInstance(),
            ↪  "demo-queue")
            .withDeclaration(QueueDeclaration.create("demo-queue"))
            .withAckRequired(false),
            bufferSize);

    return amqpSource.take(10)
        .map(readResult -> readResult.bytes().utf8String())
        .runWith(Sink.seq(), system)
        .thenApply(Vector::ofAll)
        .toCompletableFuture()
        .get(1, TimeUnit.SECONDS);
}
```

```
readFromTopic()
```

3.7.5 Consumer variants

RabbitMQ (and its underlying AMQP protocol) allows varying degrees of consistency when consuming messages.

- *Consumer acknowledgement*
  - Consumers send an `ack` message to RabbitMQ to indicate that they've successfully processed a message
  - Consumers can `ack` all messages up to the current one with one confirmation
  - Use `AmqpSource.committableSource`, process each element, and then invoke `.ack()` on it
    - `.mapAsync(committableReadResult -> committableReadResult.ack()`
- *Automatic acknowledgement*
  - Akka can automatically acknowledge messages as soon as they're read
  - Use `AmqpSource.atMostOnceSource`
- *Transactions*
  - Traditionally considered "slow" by RabbitMQ
  - Not directly supported by the Alpakka library (just use consumer acknowledgement)

4 Wrapping up

- Thanks for your participation!
- Any final thoughts / questions?
- Curious how this presentation was made?
  - Attend my talk at EmacsConf 2021