From ‘Java Sucks’ to ‘Java…Eh, Not Bad’

How Vert.x & Java 21 Made Me Stop Complaining

Thomas Gebert

Who Am I?

- Software Engineer in New York City.

Who Am I?

- Software Engineer in New York City.
- There is nothing else interesting about me.

Java.

Java.

- If you are at this conference, you probably have an opinion of Java

Java.

- If you are at this conference, you probably have an opinion of Java
- Likely very negative.

Java.

- If you are at this conference, you probably have an opinion of Java
- Likely very negative.

Why the Java Hate?

Why the Java Hate?

- Java is bloated and verbose.

Why the Java Hate?

- Java is bloated and verbose.
- Encourages bad concurrency practices.

Why the Java Hate?

- Java is bloated and verbose.
- Encourages bad concurrency practices.
- Java Enterprise Edition.

Why the Java Hate?

- Java is bloated and verbose.
- Encourages bad concurrency practices.
- Java Enterprise Edition.
- Java programmers…

Examples of (Historically) Bad Things In Java.

Examples of (Historically) Bad Things In Java.

- IO is blocking by default

Examples of (Historically) Bad Things In Java.

- IO is blocking by default
- `synchronized` is evil.

Examples of (Historically) Bad Things In Java.

- IO is blocking by default
- `synchronized` is evil.
- Core design encourages mutation.

Examples of (Historically) Bad Things In Java.

- IO is blocking by default
- `synchronized` is evil.
- Core design encourages mutation.

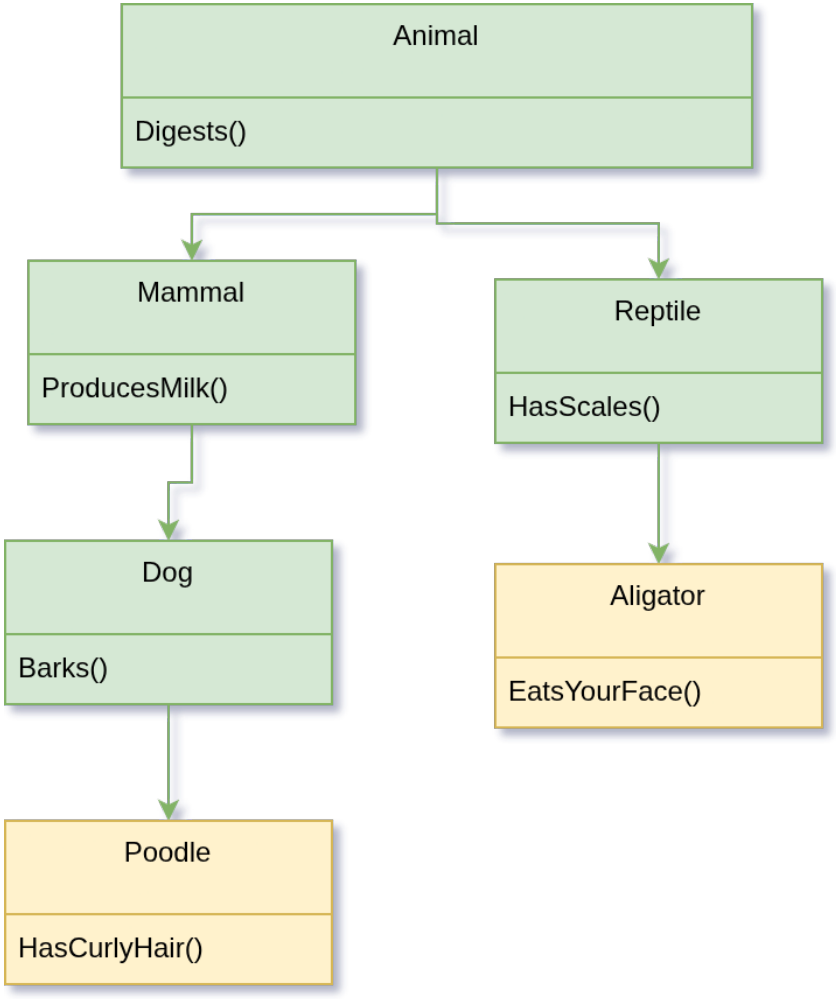
synchronized is Evil.

```
public class DeadlockExample {
  public void thread1() {
    synchronized (lockA) {
      sleep(100);
      synchronized (lockB) {
        System.out.println("Thread 1: Holding lockB");
      }
    }
  }

  public void thread2() {
    synchronized (lockB) {
      sleep(100);
      synchronized (lockA) {
        System.out.println("Thread 2: Holding lockA");
      }
    }
  }
```

Examples of (Historically) Bad Things In Java.

Examples of (Historically) Bad Things In Java.



Storytime.

Storytime.

ColdFusion
Programming
Language

Storytime.

ColdFusion
Programming
Language

So Why Would We Want To Even Use Java?

So Why Would We Want To Even Use Java?

- A metric ton of well-tested and supported libraries and guides online.

So Why Would We Want To Even Use Java?

- A metric ton of well-tested and supported libraries and guides online.
- Relatively portable, even still.

So Why Would We Want To Even Use Java?

- A metric ton of well-tested and supported libraries and guides online.
- Relatively portable, even still.
- Lots of great tooling around the language in the form of IDEs and benchmarking tools available.

So Why Would We Want To Even Use Java?

- A metric ton of well-tested and supported libraries and guides online.
- Relatively portable, even still.
- Lots of great tooling around the language in the form of IDEs and benchmarking tools available.
- (Can be) fast.

Why Not Kotlin? Or Clojure?

Why Not Kotlin? Or Clojure?

- You should use Clojure if you can!

Why Not Kotlin? Or Clojure?

- You should use Clojure if you can!
- Java is inescapable.

Why Not Kotlin? Or Clojure?

- You should use Clojure if you can!
- Java is inescapable.
- A lot of companies still have tens of thousands of lines of Java that already exist.

Why Not Kotlin? Or Clojure?

- You should use Clojure if you can!
- Java is inescapable.
- A lot of companies still have tens of thousands of lines of Java that already exist.
- Many companies will find it infeasible to migrate to a better language, and would rather spend infinitely more money hiring dozens of engineers to write a million incrementatal patches to a Java codebase.

Why Not Kotlin? Or Clojure?

- You should use Clojure if you can!
- Java is inescapable.
- A lot of companies still have tens of thousands of lines of Java that already exist.
- Many companies will find it infeasible to migrate to a better language, and would rather spend infinitely more money hiring dozens of engineers to write a million incrementatal patches to a Java codebase.
- Many of us are stuck in this hell.

Modern Java

Modern Java

- In 2024 I took a job doing Java full-time.

Modern Java

- In 2024 I took a job doing Java full-time.
- They were unreceptive to my pleas to use Clojure, no matter how much I complained.

Modern Java

- In 2024 I took a job doing Java full-time.
- They were unreceptive to my pleas to use Clojure, no matter how much I complained.
- Eventually, I realized that I wasn't going to win this fight and instead I should at least figure out what Java 21 had to offer.

Modern Java

- In 2024 I took a job doing Java full-time.
- They were unreceptive to my pleas to use Clojure, no matter how much I complained.
- Eventually, I realized that I wasn't going to win this fight and instead I should at least figure out what Java 21 had to offer.
- Much to my astonishment, I actually enjoyed it!

Modern Java

- In 2024 I took a job doing Java full-time.
- They were unreceptive to my pleas to use Clojure, no matter how much I complained.
- Eventually, I realized that I wasn't going to win this fight and instead I should at least figure out what Java 21 had to offer.
- Much to my astonishment, I actually enjoyed it!

What Changed?

What Changed?

- Since Java 8 and Java 11, there has been a much higher emphasis on functional programming concepts and updated syntax to facilitate it.

What Changed?

- Since Java 8 and Java 11, there has been a much higher emphasis on functional programming concepts and updated syntax to facilitate it.
- Java programmers have finally joined the 21st century and will occasionally use non-blocking IO.

What Changed?

- Since Java 8 and Java 11, there has been a much higher emphasis on functional programming concepts and updated syntax to facilitate it.
- Java programmers have finally joined the 21st century and will occasionally use non-blocking IO.
- Concurrency is an even bigger part of the language, and a lot of the features from concurrent-first languages have been brought over.

What Changed?

- Since Java 8 and Java 11, there has been a much higher emphasis on functional programming concepts and updated syntax to facilitate it.
- Java programmers have finally joined the 21st century and will occasionally use non-blocking IO.
- Concurrency is an even bigger part of the language, and a lot of the features from concurrent-first languages have been brought over.

Java 8 and 11 New features

Java 8 and 11 New features

```
int count = 0;
for (String word : words) {
  if (word.length() > 10) {
    count++;
  }
}
System.out.println("Long words: " + count);
```

Java 8 and 11 New features

```
int count = 0;
for (String word : words) {
  if (word.length() > 10) {
    count++;
  }
}
System.out.println("Long words: " + count);
```

```
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();

System.out.println("Long words: " + count);
```

Java 8 and 11 New features

Java 8 and 11 New features

```
public interface Greeter {
  void greet(String name);

  default void greetPolitely(String name) {
    System.out.println("Hello, " + name + ". It's nice to m
  }
}
```

Java 8 and 11 New features

```
public interface Greeter {
  void greet(String name);

  default void greetPolitely(String name) {
    System.out.println("Hello, " + name + ". It's nice to m
  }
}
```

Old Underutilized Java Feature

Old Underutilized Java Feature

BlockingQueue (java.util.concurrent)

Old Underutilized Java Feature

BlockingQueue (java.util.concurrent)

- A thread-safe queue that blocks on `put` and `take` operations

Old Underutilized Java Feature

BlockingQueue (java.util.concurrent)

- A thread-safe queue that blocks on `put` and `take` operations
- Useful for producer-consumer patterns

Old Underutilized Java Feature

BlockingQueue (java.util.concurrent)

- A thread-safe queue that blocks on `put` and `take` operations
- Useful for producer-consumer patterns
- Comes in several flavors: `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, etc.

Old Underutilized Java Feature

Old Underutilized Java Feature

```
BlockingQueue<String> queue = new LinkedBlockingQueue<>();

// Producer
new Thread(() -> {
    queue.put("data");
}).start();

// Consumer
new Thread(() -> {
    String item = queue.take();
}).start();
```

Java 21 New Features.

Java 21 New Features.

Virtual Threads.

Java 21 New Features.

Virtual Threads.

- Virtual Threads are what should have been in Java twenty years ago.

Java 21 New Features.

Virtual Threads.

- Virtual Threads are what should have been in Java twenty years ago.
- Roughly analogous to goroutines in Go.

Java 21 New Features.

Virtual Threads.

- Virtual Threads are what should have been in Java twenty years ago.
- Roughly analogous to goroutines in Go.
- Allow you to have blocking code inside the thread without it breaking the pool.
  - The JVM will park the thread upon seeing a blocking call.

Java 21 New Features.

Virtual Threads.

- Virtual Threads are what should have been in Java twenty years ago.
- Roughly analogous to goroutines in Go.
- Allow you to have blocking code inside the thread without it breaking the pool.
  - The JVM will park the thread upon seeing a blocking call.
- Extremely lightweight, hundreds of thousands can easily be spun up guilt-free.

Java 21 New Features.

Virtual Threads.

- Virtual Threads are what should have been in Java twenty years ago.
- Roughly analogous to goroutines in Go.
- Allow you to have blocking code inside the thread without it breaking the pool.
  - The JVM will park the thread upon seeing a blocking call.
- Extremely lightweight, hundreds of thousands can easily be spun up guilt-free.
- Implements the same interfaces as regular threads and thus are drop-in replacement.

Java 21 New Features.

Virtual Threads.

- Virtual Threads are what should have been in Java twenty years ago.
- Roughly analogous to goroutines in Go.
- Allow you to have blocking code inside the thread without it breaking the pool.
  - The JVM will park the thread upon seeing a blocking call.
- Extremely lightweight, hundreds of thousands can easily be spun up guilt-free.
- Implements the same interfaces as regular threads and thus are drop-in replacement.

Pre-virtual-threads

Pre-virtual-threads

```
ExecutorService executor = Executors.newFixedThreadPool(4);

for (int i = 0; i < 10; i++) {
  int taskId = i;
  executor.submit(() -> {
    System.out.println("Running task " + taskId +
                       " on thread " + Thread.currentThread
  });
}

executor.shutdown();
```

Pre-virtual-threads

Pre-virtual-threads

- Worked ok, but could break if you did any kind of blocking IO.

Pre-virtual-threads

- Worked ok, but could break if you did any kind of blocking IO.
- Did not properly park IO blocking.

New Executors

New Executors

- Still might want to support the old pattern if you have a lot of old code.

New Executors

- Still might want to support the old pattern if you have a lot of old code.

```
Executors.newVirtualThreadPerTaskExecutor()
```

Java 21 New Features*

Java 21 New Features*

* (Actually a Java 15 feature that I wasn't aware of until Java 21)

ZGC

Java 21 New Features*

* (Actually a Java 15 feature that I wasn't aware of until Java 21)

ZGC

- Low-latency garbage collector.

Java 21 New Features*

* (Actually a Java 15 feature that I wasn't aware of until Java 21)

ZGC

- Low-latency garbage collector.
- Pause times are generally sub-millisecond and almost never exceed ten milliseconds.

Java 21 New Features*

* (Actually a Java 15 feature that I wasn't aware of until Java 21)

ZGC

- Low-latency garbage collector.
- Pause times are generally sub-millisecond and almost never exceed ten milliseconds.
- Configurable, can be enabled or disabled per-project.

Java 21 New Features.

Java 21 New Features.

Records

Java 21 New Features.

Records

- Much simpler than a class.

Java 21 New Features.

Records

- Much simpler than a class.
- Doesn’t require its own dedicated file.

Java 21 New Features.

Records

- Much simpler than a class.
- Doesn’t require its own dedicated file.
- Can be pattern-matched.

Java 21 New Features.

Records

- Much simpler than a class.
- Doesn’t require its own dedicated file.
- Can be pattern-matched.

Before Records

Before Records

```
public class Point {
  private final int x;
  private final int y;

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

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

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Point)) return false;
    Point p = (Point) o;
    return x == p.x && y == p.y;
```

Sealed Interfaces

Sealed Interfaces

- Basically Algebraic Data Types

Sealed Interfaces

- Basically Algebraic Data Types
- Can be recursive.

Sealed Interfaces

- Basically Algebraic Data Types
- Can be recursive.
- Can be pattern matched.

Sealed Interfaces

- Basically Algebraic Data Types
- Can be recursive.
- Can be pattern matched.

Before Sealed Interfaces

Before Sealed Interfaces

```
public interface Shape {}

public class Circle implements Shape {
  public final double radius;
  public Circle(double radius) { this.radius = radius; }
}

public class Rectangle implements Shape {
  public final double width, height;
  public Rectangle(double w, double h) {
    this.width = w;
    this.height = h;
  }
}
```

Java 21 New Features.

Java 21 New Features.

Pattern Matching

Java 21 New Features.

Pattern Matching

- FINALLY! FINALLY!

Java 21 New Features.

Pattern Matching

- FINALLY! FINALLY!
- Can be done inside `if` statements and `switch` cases.

Java 21 New Features.

Pattern Matching

- FINALLY! FINALLY!
- Can be done inside `if` statements and `switch` cases.

Before pattern matching.

Before pattern matching.

```
public void handle(Object obj) {
  if (obj instanceof String) {
    String s = (String) obj;
    System.out.println("String length: " + s.length());
  } else if (obj instanceof Integer) {
    Integer i = (Integer) obj;
    System.out.println("Squared: " + (i * i));
  } else {
    System.out.println("Unknown type");
  }
}
```

Java NIO

Java NIO

- Java New IO.

Java NIO

- Java New IO.
- Gives fine-grained control over IO, both blocking and non-blocking.

Java NIO

- Java New IO.
- Gives fine-grained control over IO, both blocking and non-blocking.
- Not new at all, but underutilized.

Java NIO

- Java New IO.
- Gives fine-grained control over IO, both blocking and non-blocking.
- Not new at all, but underutilized.

Vert.x

Vert.x

- (In a hand-wavey way) a port of Node.js to Java.

Vert.x

- (In a hand-wavey way) a port of Node.js to Java.
- High performance.

Vert.x

- (In a hand-wavey way) a port of Node.js to Java.
- High performance.
- Provides constructs to handle local and distributed concurrency transparently.

Vert.x Core Primitives

Vert.x Core Primitives

Verticle

Vert.x Core Primitives

Verticle

- Units of deployment and concurrency

Vert.x Core Primitives

Verticle

- Units of deployment and concurrency
- Two types: StandardVerticle (blocking) and WorkerVerticle (non-blocking optional)

Vert.x Core Primitives

Verticle

- Units of deployment and concurrency
- Two types: StandardVerticle (blocking) and WorkerVerticle (non-blocking optional)
- Deployed with `vertx.deployVerticle(...)`

Vert.x Core Primitives

Event Loop

Vert.x Core Primitives

Event Loop

- Single or multi-threaded, async task execution

Vert.x Core Primitives

Event Loop

- Single or multi-threaded, async task execution
- Based on Netty

Vert.x Core Primitives

Event Loop

- Single or multi-threaded, async task execution
- Based on Netty
- Designed for minimal context switching and high throughput

Vert.x Core Primitives

Vert.x Core Primitives

Event Bus

Vert.x Core Primitives

Event Bus

- Lightweight messaging system

Vert.x Core Primitives

Event Bus

- Lightweight messaging system
- Supports publish/subscribe, point-to-point, and request-response

Vert.x Core Primitives

Event Bus

- Lightweight messaging system
- Supports publish/subscribe, point-to-point, and request-response
- Accepts JSON, POJOs (with codec), and buffers

Vert.x Core Primitives

Event Bus

- Lightweight messaging system
- Supports publish/subscribe, point-to-point, and request-response
- Accepts JSON, POJOs (with codec), and buffers

Vert.x Core Primitives

Vert.x Core Primitives

Future & Promise

Vert.x Core Primitives

Future & Promise

- Asynchronous result handling

Vert.x Core Primitives

Future & Promise

- Asynchronous result handling
- `Future<T>` is the result placeholder

Vert.x Core Primitives

Future & Promise

- Asynchronous result handling
- `Future<T>` is the result placeholder
- `Promise<T>` is the result provider

Vert.x Core Primitives

Future & Promise

- Asynchronous result handling
- `Future<T>` is the result placeholder
- `Promise<T>` is the result provider
- Supports chaining with `.compose(...)` and `.map(...)`

Vert.x Core Primitives

Future & Promise

- Asynchronous result handling
- `Future<T>` is the result placeholder
- `Promise<T>` is the result provider
- Supports chaining with `.compose(...)` and `.map(...)`

Vert.x Core Primitives

Vert.x Core Primitives

Context

Vert.x Core Primitives

Context

- Execution environment for a Verticle

Vert.x Core Primitives

Context

- Execution environment for a Verticle
- Ensures thread-affinity

Vert.x Core Primitives

Context

- Execution environment for a Verticle
- Ensures thread-affinity
- Helps avoid shared-state concurrency bugs

Vert.x Core Primitives

Context

- Execution environment for a Verticle
- Ensures thread-affinity
- Helps avoid shared-state concurrency bugs

Vert.x Core Primitives

Vert.x Core Primitives

Buffer

Vert.x Core Primitives

Buffer

- Efficient binary data container

Vert.x Core Primitives

Buffer

- Efficient binary data container
- Higher-level alternative to Netty's ByteBuf

Vert.x Core Primitives

Buffer

- Efficient binary data container
- Higher-level alternative to Netty’s ByteBuf
- Used in I/O and message passing

Vert.x Core Primitives

Buffer

- Efficient binary data container
- Higher-level alternative to Netty’s ByteBuf
- Used in I/O and message passing

Vert.x Core Primitives

Vert.x Core Primitives

WebClient / HttpClient

Vert.x Core Primitives

WebClient / HttpClient

- Non-blocking HTTP clients

Vert.x Core Primitives

WebClient / HttpClient

- Non-blocking HTTP clients
- Built-in connection pooling and retry logic

Vert.x Core Primitives

WebClient / HttpClient

- Non-blocking HTTP clients
- Built-in connection pooling and retry logic
- Supports JSON, form data, and streaming

Vert.x Core Primitives

WebClient / HttpClient

- Non-blocking HTTP clients
- Built-in connection pooling and retry logic
- Supports JSON, form data, and streaming

Vert.x Core Primitives

Vert.x Core Primitives

Timer / Periodic Tasks

Vert.x Core Primitives

Timer / Periodic Tasks

- Use `setTimer(...)` for delayed execution

Vert.x Core Primitives

Timer / Periodic Tasks

- Use `setTimer(...)` for delayed execution
- Use `setPeriodic(...)` for recurring tasks

Vert.x Core Primitives

Timer / Periodic Tasks

- Use `setTimer(...)` for delayed execution
- Use `setPeriodic(...)` for recurring tasks
- Executes on the event loop thread

Vert.x Core Primitives

Timer / Periodic Tasks

- Use `setTimer(...)` for delayed execution
- Use `setPeriodic(...)` for recurring tasks
- Executes on the event loop thread

```
void doSomethingAsync(Promise<String> promise) {
  vertx.setTimer(500, id -> {
    promise.complete("Hello, future!");
  });
}
```

Vert.x Core Primitives

Vert.x Core Primitives

SharedData

Vert.x Core Primitives

SharedData

- Minimal shared-state coordination mechanism

Vert.x Core Primitives

SharedData

- Minimal shared-state coordination mechanism
- Offers maps, locks, and counters

Vert.x Core Primitives

SharedData

- Minimal shared-state coordination mechanism
- Offers maps, locks, and counters
- Supports clustered and local modes

Vert.x Core Primitives

SharedData

- Minimal shared-state coordination mechanism
- Offers maps, locks, and counters
- Supports clustered and local modes

Backpressure in Vert.x

Backpressure in Vert.x

- Vert.x models backpressure using `ReadStream` and `WriteStream`

Backpressure in Vert.x

- Vert.x models backpressure using `ReadStream` and `WriteStream`
- Data is paused/resumed automatically when the receiver can't keep up

Backpressure in Vert.x

- Vert.x models backpressure using `ReadStream` and `WriteStream`
- Data is paused/resumed automatically when the receiver can’t keep up
- Useful when handling large streams (e.g., file uploads, HTTP bodies)

Backpressure in Vert.x

Backpressure in Vert.x

Example: Handling a slow WriteStream

```
source.pipeTo(slowSink, res -> {
  if (res.succeeded()) {
    System.out.println("All data written.");
  } else {
    res.cause().printStackTrace();
  }
});
```

Vert.x distributed concurrency example

Vert.x distributed concurrency example

Deploying Verticles: Local vs Clustered

Vert.x distributed concurrency example

Deploying Verticles: Local vs Clustered

- Verticles are the basic unit of deployment and concurrency

Vert.x distributed concurrency example

Deploying Verticles: Local vs Clustered

- Verticles are the basic unit of deployment and concurrency
- Deployment is nearly identical across local and clustered environments

Vert.x distributed concurrency example

Deploying Verticles: Local vs Clustered

- Verticles are the basic unit of deployment and concurrency
- Deployment is nearly identical across local and clustered environments

Vert.x Concurrency Example.

Vert.x Concurrency Example.

```
public class MyVerticle extends AbstractVerticle {

  @Override
  public void start(Promise<Void> startPromise) {
    System.out.println("Verticle started on thread: " + Thr

    vertx.setTimer(1000, id -> {
      System.out.println("Timer fired after 1 second");
    });

    startPromise.complete();
  }
}
```

Vert.x Concurrency Example.

```
public class MyVerticle extends AbstractVerticle {

  @Override
  public void start(Promise<Void> startPromise) {
    System.out.println("Verticle started on thread: " + Thr

    vertx.setTimer(1000, id -> {
      System.out.println("Timer fired after 1 second");
    });

    startPromise.complete();
  }
}
```

Vert.x Concurrency Example.

Vert.x Concurrency Example.

Local Deployment

```
Vertx vertx = Vertx.vertx();
vertx.deployVerticle(new MyVerticle());
```

Vert.x Concurrency Example.

Vert.x Concurrency Example.

Distributed Deployment

Vert.x Concurrency Example.

Distributed Deployment

```
Vertx.clusteredVertx(new VertxOptions(), res -> {
  if (res.succeeded()) {
    Vertx vertx = res.result();
    vertx.deployVerticle(new MyVerticle());
  } else {
    res.cause().printStackTrace();
  }
});
```

RxJava

RxJava

- Framework for reactive applications by Microsoft.

RxJava

- Framework for reactive applications by Microsoft.
- Gives us functional patterns for reactive applications.

RxJava: Basics of Reactive Programming

RxJava: Basics of Reactive Programming

- RxJava is based on the Observer pattern with functional style

RxJava: Basics of Reactive Programming

- RxJava is based on the Observer pattern with functional style
- Core idea: data flows over time, like a stream

RxJava: Basics of Reactive Programming

- RxJava is based on the Observer pattern with functional style
- Core idea: data flows over time, like a stream
- Common building block: `Observable<T>`

Creating an Observable

Creating an Observable

```
Observable<String> source = Observable.just("Alpha", "Beta"

Observable<Integer> lengths = source
  .map(str -> str.length())
  .filter(len -> len >= 5);
```

Creating an Observable

Creating an Observable

```
lengths.subscribe(
  item -> System.out.println("Received: " + item),
  error -> System.err.println("Error: " + error),
  () -> System.out.println("Stream complete")
);
```

Creating an Observable

```
lengths.subscribe(
  item -> System.out.println("Received: " + item),
  error -> System.err.println("Error: " + error),
  () -> System.out.println("Stream complete")
);
```

RxJava Niceties

RxJava Niceties

- Everything is async by default (with schedulers)

RxJava Niceties

- Everything is async by default (with schedulers)
- Supports composition, backpressure, error handling

RxJava Niceties

- Everything is async by default (with schedulers)
- Supports composition, backpressure, error handling
- Great for UI events, streams, or reactive pipelines

RxJava Niceties

- Everything is async by default (with schedulers)
- Supports composition, backpressure, error handling
- Great for UI events, streams, or reactive pipelines

Conclusion.

Conclusion.

- Java 21 isn’t that bad.

Conclusion.

- Java 21 isn’t that bad.
- Convince your employers to upgrade if you want to reclaim your sanity.
- Blah . . .
- Use libraries like Vert.x and Disruptor to make life simpler.

Conclusion.

- thomas@gebert.app
- blog.tombert.com