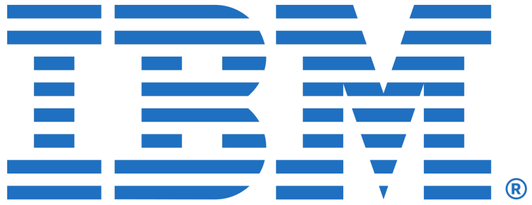
The De-Evolution of Java

May 2025

Ben Evans (He / Him)
Senior Principal Software Engineer
beevans@ibm.com

About Me - Career

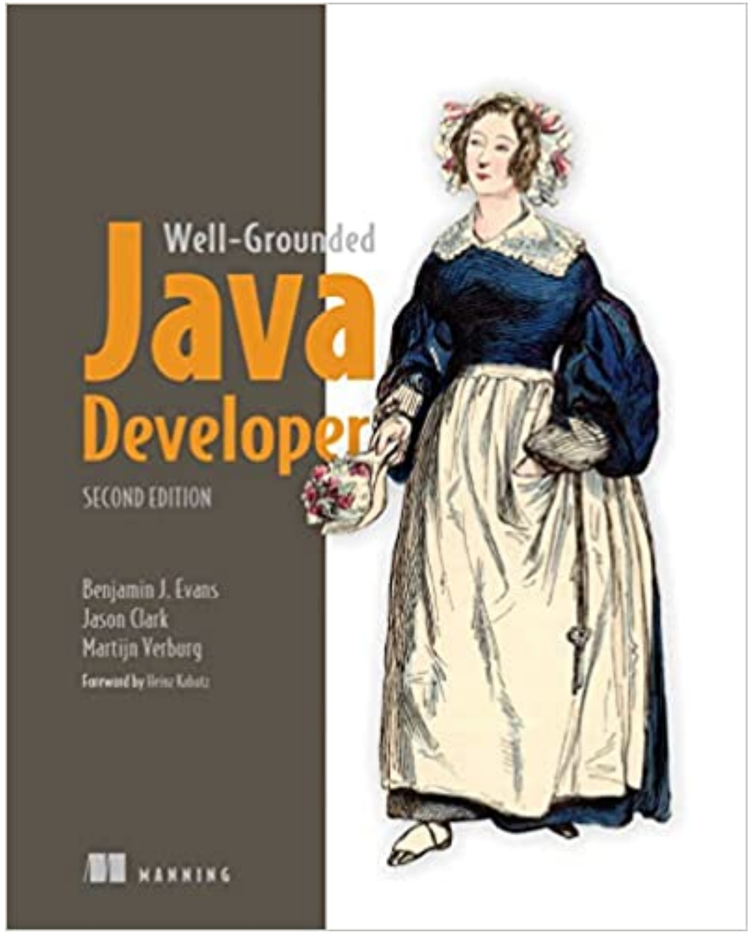
- IBM
- Red Hat, SPSE
- New Relic, Lead Architect
- jClarity, Co-founder (acq MSFT)
- Deutsche Bank
  - Chief Architect (Listed Derivatives)
- Morgan Stanley
  - Google IPO
  - MATRIX

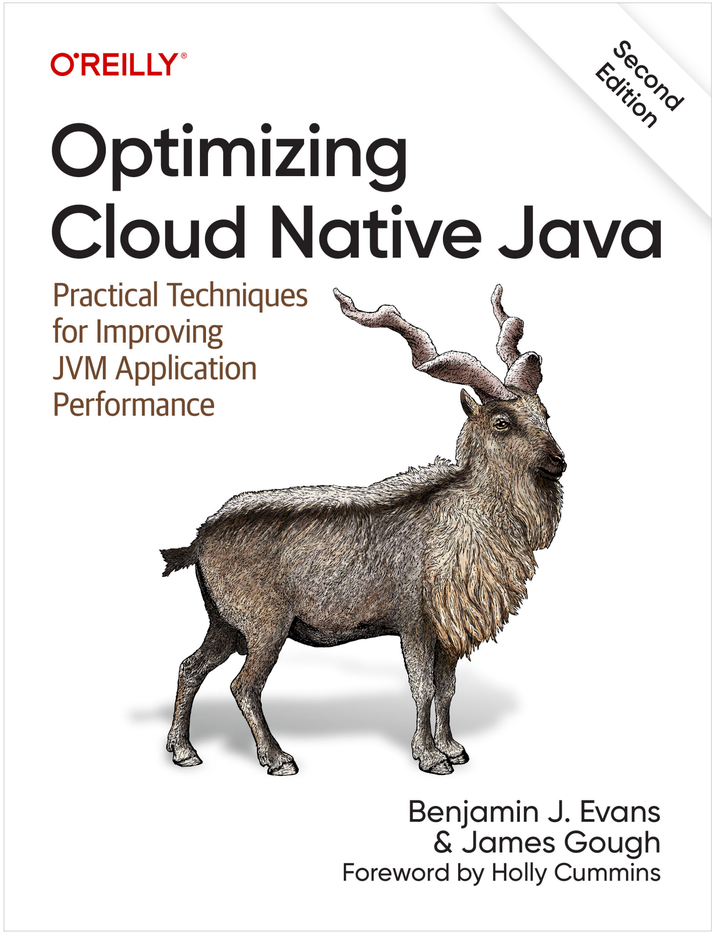


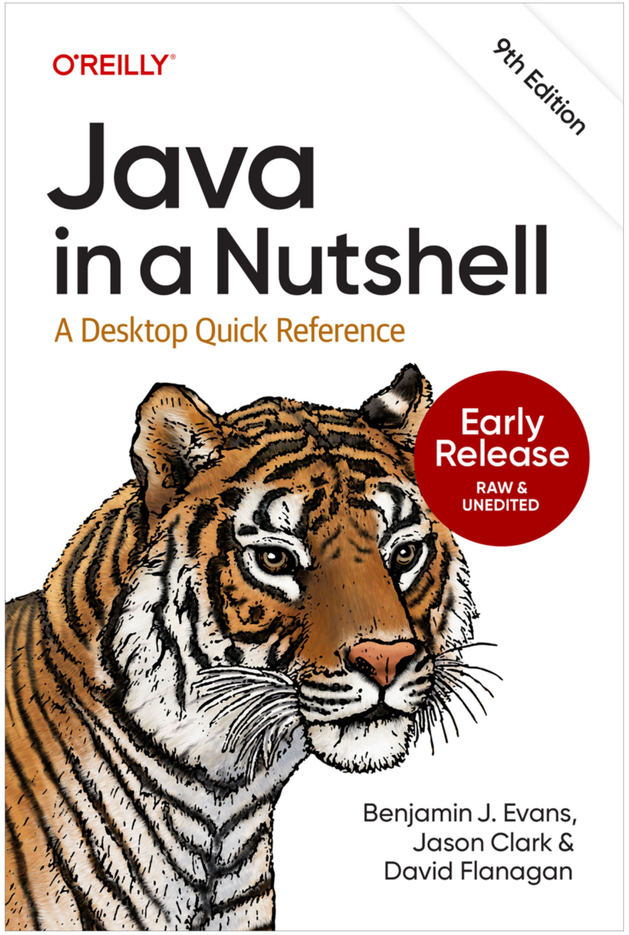
About Me - Community

- Java Champion
- JavaOne Rock Star Speaker
- Java Community Process Executive Committee
- London Java Community
  - Organising Team
  - Co-founder, AdoptOpenJDK

Recent Books







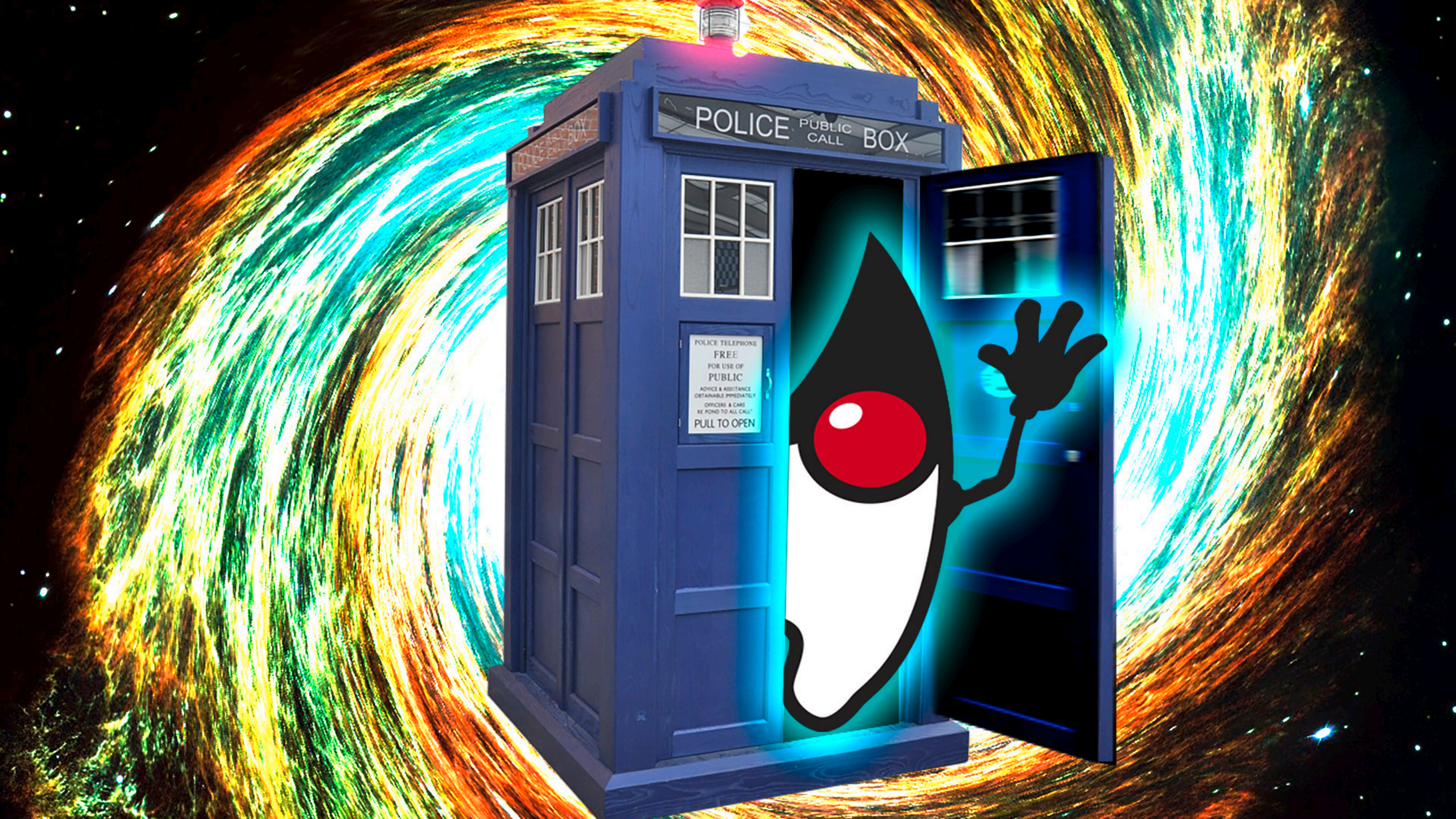
https://kittylyst.com

Agenda

- Java 25 -> Java 17
- Java 17 -> Java 11
- Java 11 -> Java 8
- Java 8 -> Java 7
- Java 7 -> Java 6
- The Before Times…?

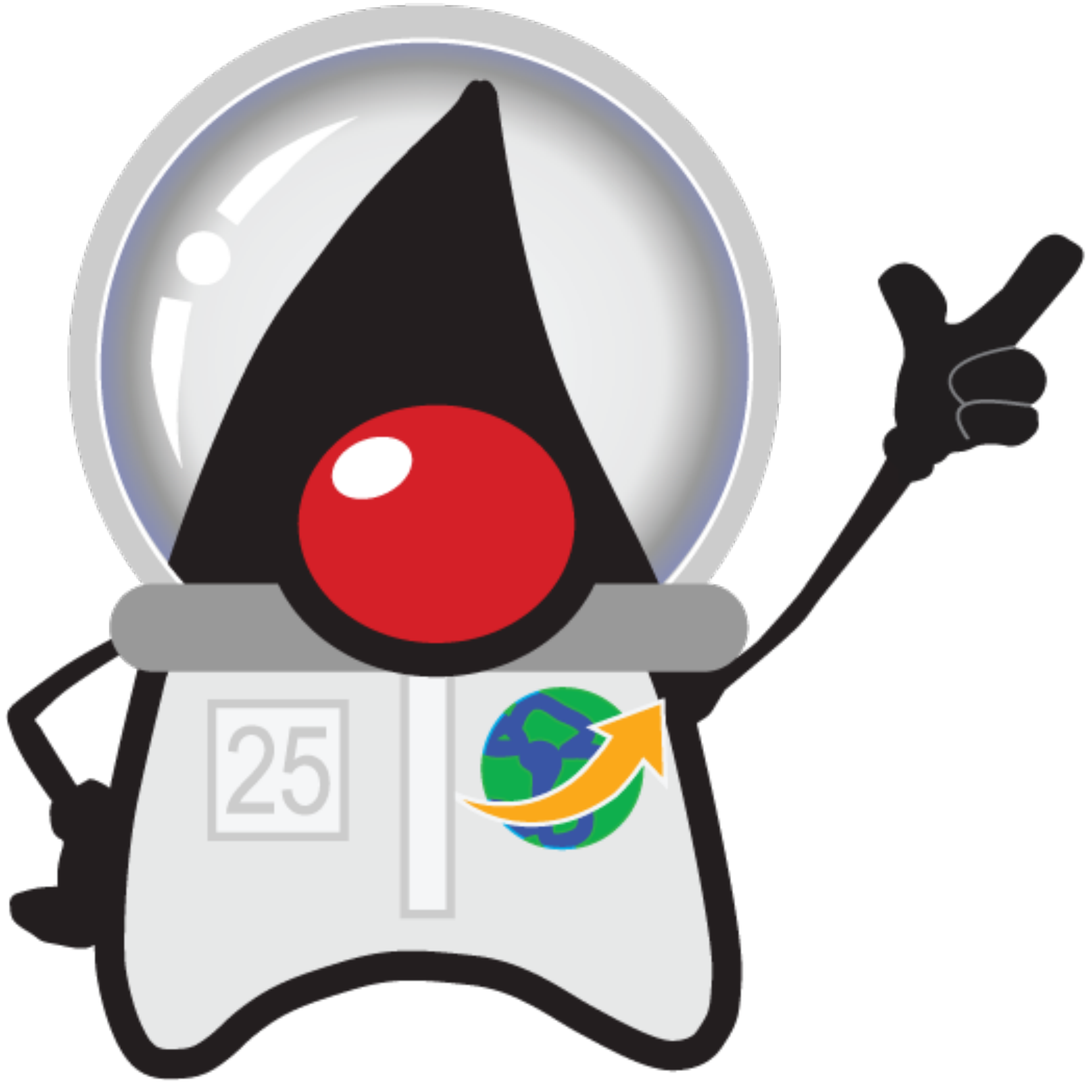
Meet the Application

- A mock FX trading application
- 2 Order Managers
  - Clientside
  - Marketside
- Order and OrderResponse domain model
- Simple Logging Facade


POLICE PUBLIC CALL BOX
POLICE TELEPHONE
FREE
FOR USE OF
PUBLIC
ADVICE & ASSISTANCE
OBTAINABLE IMMEDIATELY
OFFICERS & CARS
RESPOND TO ALL CALLS
PULL TO OPEN

Java 25 -> Java 17

- Pattern Matching
- Scoped Values
- Virtual Threads



Java 25 - Pattern Matching

- Patterns can be used in switch cases
- Including with pattern guards using `when`

```
boolean isDog(Pet p) {
    var day = LocalDate.now().getDayOfWeek();
    return switch (p) {
        case Cat c when day.equals(MONDAY) -> true;
        case Cat c -> false;
        case Dog d -> true;
    };
}
```

Java 25 - Record Patterns

- Records have explicit, guaranteed deconstruction semantics
  - A record is just a transparent carrier of fields
- Component fields can always be extracted safely
  - Other languages might call this destructuring
  - Called a "record pattern" in Java

```
if (p instanceof Dog(String s)) {
    System.out.println("Saw a Dog called: "+ s);
}
```

Java 25 - Record Patterns

```
sealed interface Pet permits Cat, Dog {}
record Cat(String name) implements Pet {}
record Dog(String name) implements Pet {}

class A {}
class B extends A {}
record Pair<T>(T x, T y) {}
```

Java 25 - Record Patterns

```
Pair<A> p1;

var isB = switch (p1) {
    case Pair(B b1, B b2) -> b1.equals(b2);
    case Pair(A a1, B b1) -> false;
    case Pair(B b1, A a1) -> true;
    case Pair(A a1, A a2) -> false;
//      case null -> false;
//      default -> false;
};
```

Java 25 - Unnamed Patterns & Variables

```
Pair<Pet> p2;

var secondName = switch (p2) {
     case Pair<Pet>(Pet _, Dog(String name)) -> name;
     case Pair<Pet>(Pet _, Cat(String _)) -> "Cat";
};

var firstName = switch (p2) {
     case Pair<Pet>(Cat(String n), Pet _) -> "Sir "+ n;
     case Pair<Pet>(Dog(String n), Cat(String cN)) -> n +" & "+ cN;
     case Pair<Pet>(Dog(String name), Dog _) -> name;
};
```

Java 25 - Primitive Patterns?

- As of Java 25, Primitive Patterns are in Third Preview
- Why is this important?
  - Remember the Java type system is not single-rooted
  - Another building block...
- Only intend to cover Final features in main part of this talk

Java 25 - Scoped Values

- `ScopedValue<T>`
  - Provides a binding of a value to a thread or scope
  - Written once & then immutable per-thread
  - Combines with fire-and-forget vthread patterns
  - Modern alternative to thread-local variables
- No `set()` method to let faraway code change scoped value
- Provides aspects of “implicit” behaviour
  - Much more controlled than Scala’s implicits
  - (See also discussion on forthcoming possible typeclasses)

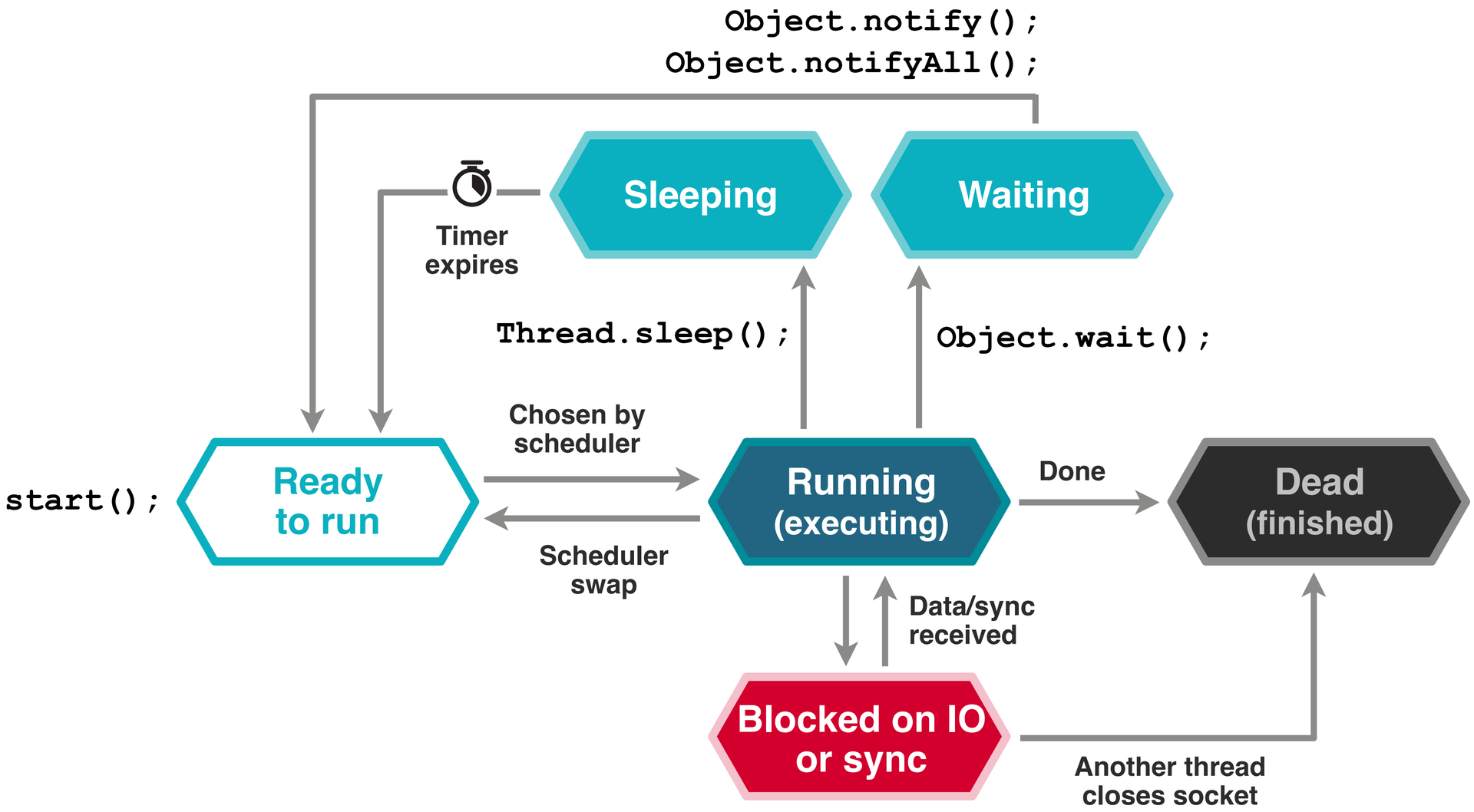
Java 25 - Scoped Values

- Share data within a thread and with child threads
- Lifetime is controlled / bounded
  - Visible from the structure of code
- Shared data is immutable
  - Allows sharing by lots of threads
  - Enables possible future runtime optimizations
- Not necessary to move away from `ThreadLocal`
  - Immutability & explicit lifetime is often a better fit

Java 25 - Scoped Values In Action

- Pattern: Use a public static final field as a holder
- Can be accessed from anywhere
- Provides a dynamic scope
  - Similar to that found in some Lisps and shells (& Perl)
  - Contrast with lexical (aka static) scope
- Examples
  - Transaction context
  - Security Principals
  - “Ambient context”

Java 21 - Solving the thread bottleneck



Java 21 - Practicalities of Thread lifecycle

- Threads *can* run to completion purely in user mode
  - Use up their entire timeslice
- In practice, this rarely happens
  - Threads often hit a blocking call (e.g. I/O)
- Java does not have “bare” syscalls
  - Java is a “fully managed environment”
  - All “system calls” are actually library calls
  - JVM makes syscalls on behalf of the user thread

Java 21 - A New Type of Thread?

- What if there was a new kind of thread?
  - Created & managed by the JVM, NOT the OS
  - Doesn’t have a dedicated OS thread to run on - must share “carrier threads”
  - Doesn’t have to reserve a stack segment
  - Designed for tasks which do (at least some) I/O
  - BUT they are also just a thread, same concept we’re already familiar with
- JEP 353 (Java 13) reimplemented Java Socket API
  - Previously, based on blocking I/O
  - Now, backed by non-blocking I/O (NIO)
  - Did not change the Java API (only internals)
  - Important stepping stone

Java 21 - Virtual Threads

- Project Loom: “easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform”
- The “virtual thread” (vthread)
  - Similar in intent to a goroutine (or an Erlang process)
  - Stack frame is stored in the Java heap (removes bottleneck)
  - vthread is mounted on a platform thread (the carrier) to execute
    - Stack frames are copied onto the carrier threads stack when necessary
- Have to specifically create a vthread
  - No automatic virtualisation of threads
  - No existing code changes meaning

Java 21 - Sharing Carrier Threads

- “Every time vthread makes a blocking I/O call, it yields”
  - Nearly all blocking points have been modified to unmount a vthread
- Actual I/O impl is non-blocking, proceeds while vthread is paused
- Another vthread can use the carrier thread instead
- Carriers are a standard Java thread pool (ExecutorService)
- vthread cannot observe the current carrier
  - `Thread.currentThread()` returns the vthread
  - Carrier frames do not show up in stack trace of vthread

Java 21 - vThreads vs Other Languages?

- Most similar to goroutines
- async / await approaches - considered & rejected
  - Similarly, bringing “reactive approaches” into the JDK
  - No “colored functions”
- Thread.yield() does work for vthreads
  - But discouraged
- vThreads do have continuations
  - But they are not accessible (for now?) - jdk.internal.vm

Java 17 -> Java 11

- Text Blocks
- Algebraic Data Types
  - Sealed Types
  - Records
- Pattern remnants
  - instanceof Patterns
  - Switch Expressions

Java 17 - Text Blocks

- Allow string literals that extend over multiple lines
  - Avoid the need for clumsy escape sequences
  - A new way to specify String literals
- Use """ to start and terminate blocks
  - Possibly ignorable whitespace
- Do not currently support interpolation
  - String Templates was contemplated, but withdrawn
  - JEP 465: https://openjdk.org/jeps/465
  - Being rethought and reworked

Java 17- Algebraic Data Types

- Sealed Types
  - “x is of type A-OR-B”
- Records
  - “Just Carries Fields” Pattern
  - Java’s approach to tuples
  - Graceful migration path from records to “full” classes

```
sealed interface Pet permits Cat, Dog {}
record Cat(String name) implements Pet {}
record Dog(String name) implements Pet {}
```

Java 17 - Pattern remnants

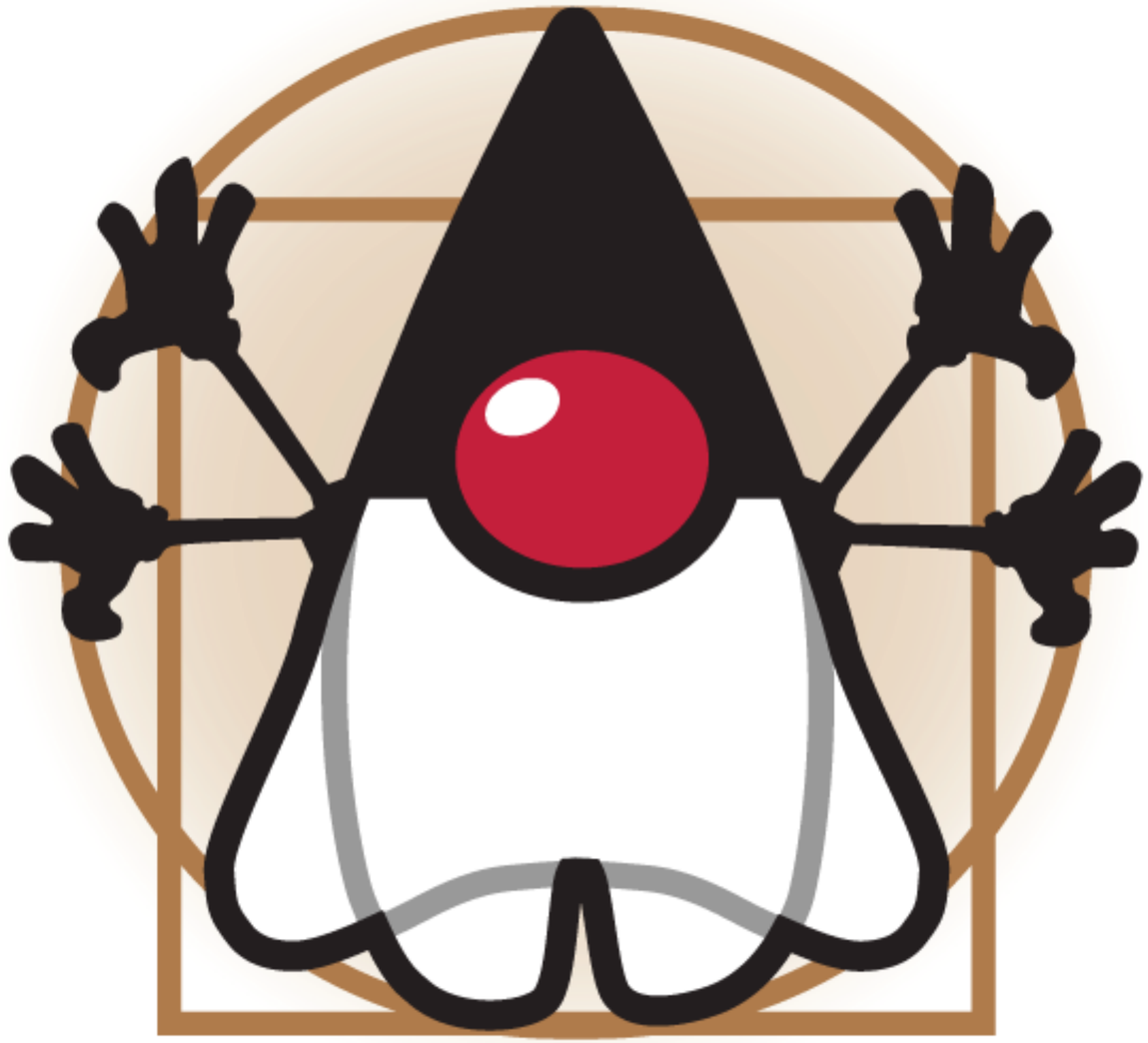
```
boolean isDog(Pet p) {
    if (p instanceof Dog d) {
        return true;
    }
    if (p instanceof Cat c) {
        return false;
    }
    throw new IllegalStateException("Case cannot occur");
}
```

Java 17 - Pattern remnants

- Switch expressions
- Java 1.0 `switch` is statement-only / side-effect-only (C heritage)
- Retrofitted to allow an expression form as well
- Introduce the `yield` keyword
- Arrow form for expressions is much more common
- No default-fallthrough
- Expression must be *total* & include a `case default` if needed

Java 11 -> Java 8

- LVTI (`var`)
- Encapsulated Internals (Modules)



Java 11 - LVTI (var)

- Local Variable Type Inference
- `var` is a reserved, "magic" type name not a new keyword
- Implemented solely in `javac` - no runtime / performance effect
- Uses “constraint solving” for type resolution (like lambdas)
- https://openjdk.org/projects/amber/guides/lvti-style-guide

Encapsulate The Internals

????

Public Methods (Java 8)

- You can call public methods on any public class you like
  - Directly
  - Reflectively

Public Methods (Java 9+)

- You can call public methods on any public class you like
  - Directly (*)
  - Reflectively
- * Subject to additional restrictions from modules

Encapsulation 8 -> 11

- Warnings have become errors
  - Some working code will no longer compile
- Things have already changed
- But most people didn’t notice
- Only applications / libs that broke the “rules” were caught by this
- Compiler change

Encapsulation 8 -> 11

- Public method on a public class not automatically accessible
- Platform can finally enforce the long-standing convention
  - Package that starts `java` or `javax` is a public API
  - Everything else is internal-only
- But what about Reflection?
  - Basically same as Java 8

Public Methods (Java 16+)

- You can call public methods on any public class you like
  - Directly (*)
  - Reflectively (*)
- * Subject to additional restrictions from modules

Public Methods in non-exported packages

Version	Direct	Reflective
8	Y	Y
11	N	Y
16	N	N*
17	N	N

(*) Can be re-enabled via a switch

What About Unsafe?

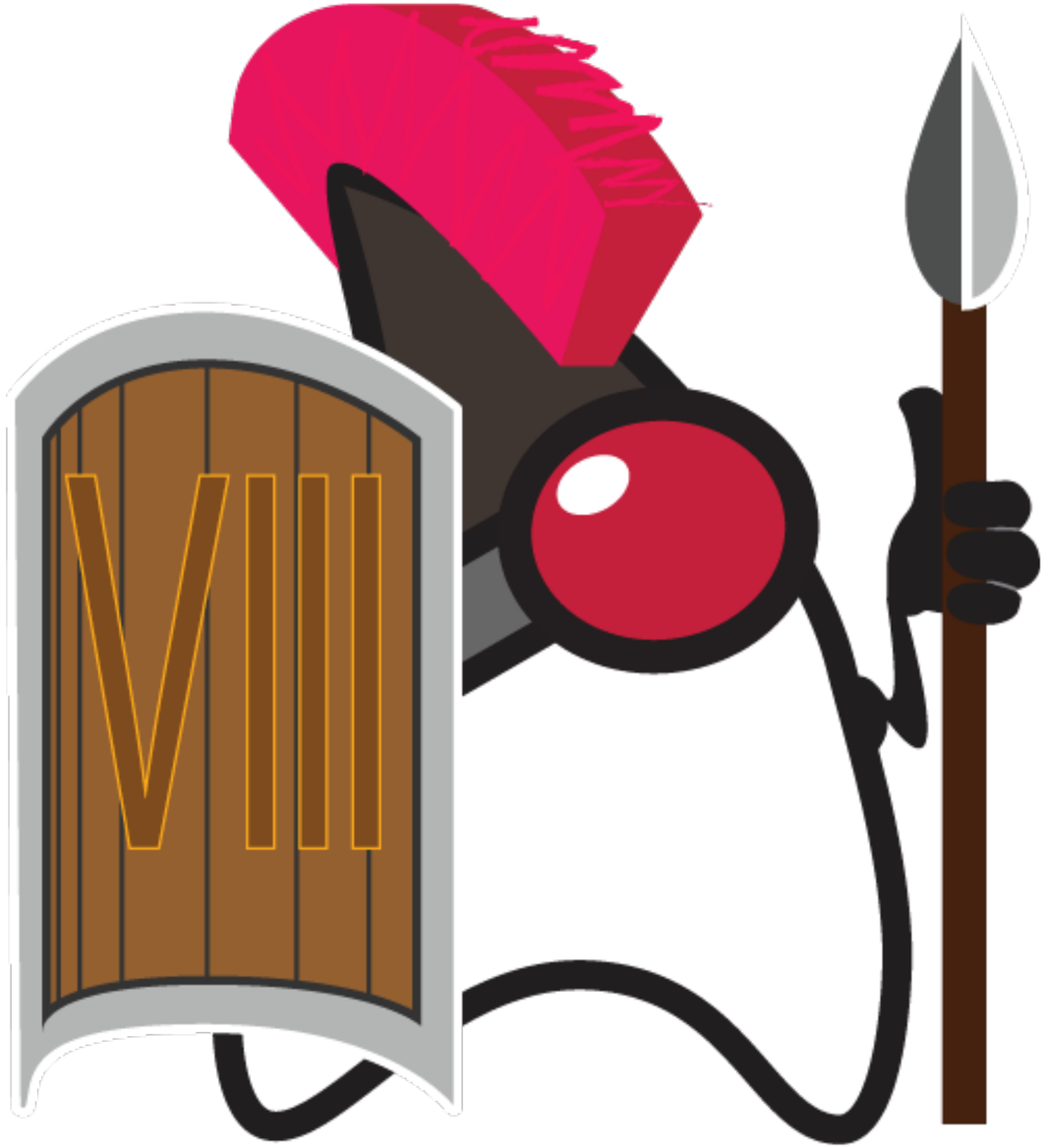
- Unsafe can only be accessed reflectively
- In Java 9+ it has been moved to the module `jdk.unsupported`
- `jdk.unsupported` is visible to classpath by default
  - Open module
- In practice, just as available in Java 9+ as in 8

Upgrade (from 8) to 11 Before Going to 17

- 8 -> 11 is fundamentally about library upgrades
- Virtually all libraries are now 11 compatible (with a version bump)
- 17 is a different story
  - Mostly due to encapsulation changes
- OpenJDK devs have made big steps to smooth migration
  - But it’s not perfect…

Java 8 -> Java 7

- Lambdas
- Streams
- (java.time)
- (Optional)
- (CompleteableFuture)
- Default methods
- Static methods on interfaces



Java 8 - Lambdas

- Lambda expressions create a new object of unknown type
  - Implement a specific “Single Abstract Method” interface (a SAM type)
- Compact syntax for declaration
- No relaxation of Java’s method call syntax
  - f.apply(x) rather than f(x) for a lambda f
- Not implemented as inner classes
  - Check the compiler output - no secondary class files created
- Uses invokedynamic as the implementing engine

Java 8 - Streams

- Java collections are mutable & fully-materialised
- Not an easy on-ramp for more-functional approaches
- Stream is an abstraction for dealing with more general data structures
- Stream doesn’t manage storage for or provide access to elements
- Lazy and functional by design
- Co-introduced with lambdas & default methods
  - They need each other

Java 8 - Streams

- Collections are first-class
  - Embedded in signatures throughout the JDK
  - Streams are not
- Java has no type-level `Traversable` concept
- Collections are not “functional containers”
  - Streams are - to a limited degree
- Streams are ephemeral, consumable objects
  - They are not stable objects to be long-lived & passed around

Java 8 - Default Methods

- Java may not add methods to an existing interface within the JDK
  - Maintain backward compatibility
- Allow upgrade of interfaces by adding default methods
  - Any implementation may implement the default method
  - Implementations that don’t implement use the default code
  - Must implement in the case of two colliding default methods
    - This will only happen if implementing multiple interfaces w defaults
- Implemented by patching implementations during class loading

Java 8 - Static Methods on Interfaces

- As of Java 8 interfaces can contain static methods
- There was never any good technical reason why they didn’t
- Original Java 1.0 philosophy “interfaces don’t contain code”
- Lead to “companion” or “plurals” classes to hold static methods
  - Helper methods that should live on the interface

Java 7 -> Java 6

- Try with resources
- NIO.2
- (Invokedynamic)
- (Fork and Join)

Java 7 - Try-with-resources

- Automatically handle resource cleanup when exiting a block
- Original Java 6 code was wrong 60+% of the time
  - Even in the JDK
- Classic case where the computer should be doing the book-keeping
- Obvious case of a feature that’s “part of the furniture”

Java 7 - NIO.2

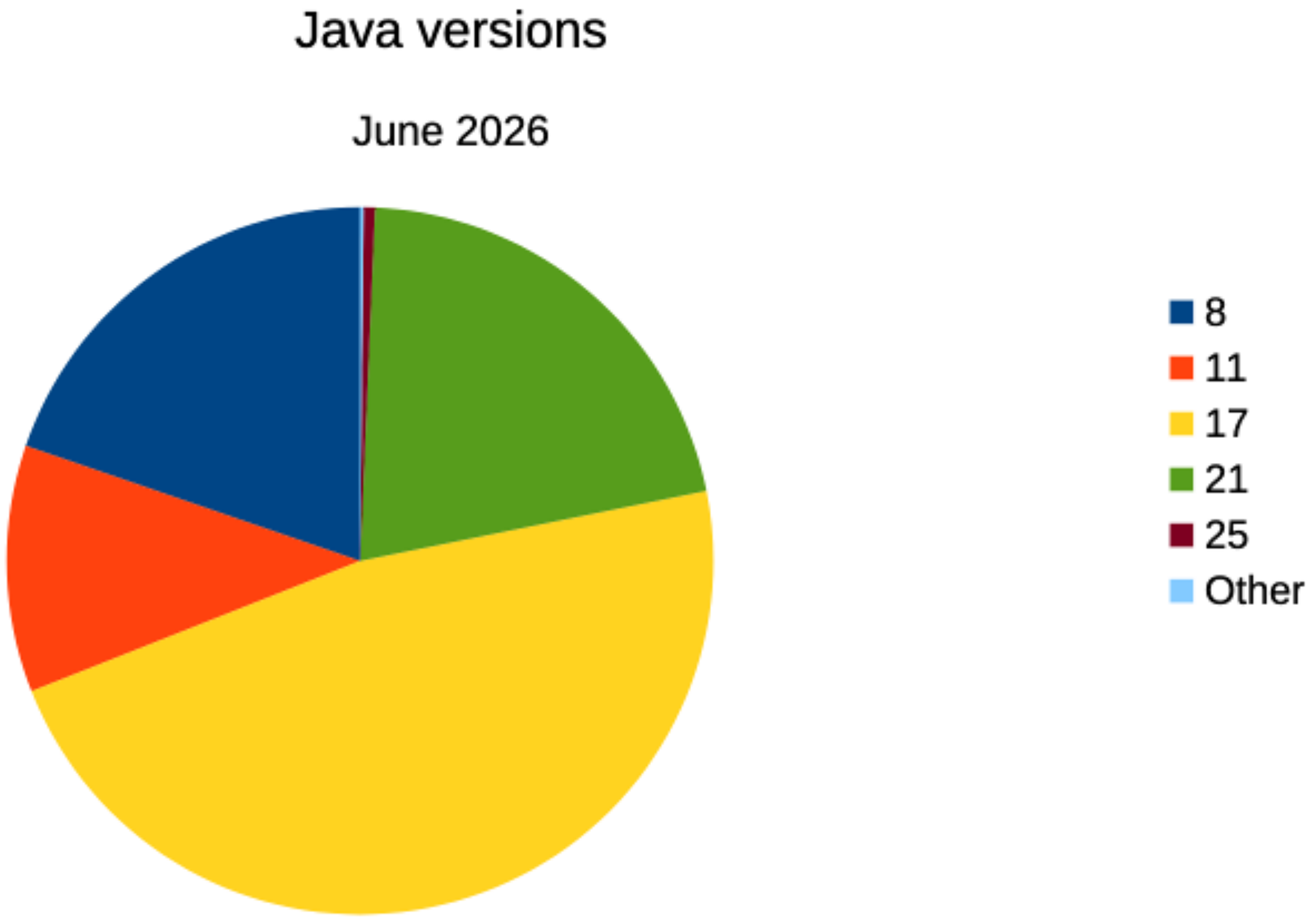
- Path & Files were a huge step forward
- Original File is missing basic functionality
  - Does not provide a way to read the contents of a file directly
    - Has to be paired with the I/O stream abstraction
- Abstraction over files and directories is cumbersome
  - Prioritises platform-independence over usability
  - Original Java 1.0 design bet - has not aged well

With Apologies To Evolutionary Biologists…

The Primordial Swamp (6 -> 5 and before…)

- Generics
  - “Mystery Meat” collections
- java.util.concurrent
- Varargs
- It’s very difficult to get projects to build on such old versions at all
- OpenJDK 6 exists, everything earlier is technically not OSS
- It is possible, if a bit hairy…
  - https://www.chainguard.dev/unchained/fully-bootstrapping-java-from-source-in-wolfi

Java Versions in Production



Java’s Type System

- Static
- Strictly Nominal
- Object / Imperative
- Type erased
- Modestly type-inferred
- “Slightly Functional”

Static

- Type system enforced by Java language (& javac)
- The JVM has a philosophy of:
  - Trust but Verify
  - Go Classfile or go home
  - Single entry point (checkpoint) for code
  - Helps enforce runtime type system
- Every generation rediscovers static typing
  - And why the field invented it in the first place

“Strong Typing”

“Strong typing: A type system that I like and feel comfortable with.

Weak typing: A type system that worries me, or makes me feel uncomfortable…”

– Curtis Poe

Strictly Nominal

- Java's types are Nominal (name-based)
- Types only compatible if explicit inheritance exists
  - Either class inheritance or interface inheritance
- No way of talking about a type except by name
  - At least, for lvalues
- No structural typing
  - E.g. `Pair<Key k1, Key k2>` & `Tuple2<Key k1, Key k2>`
    - Totally separate types
    - Need explicitly written conversion methods
    - No type constructors

Object / Imperative

- Java has a simple OO model
- Not pure OO
  - Simplifies working with primitives
  - Makes it easier to teach (in theory)
- Historically
  - Aided transition for C/C++ devs
  - Improved performance on extremely limited 90s hardware
    - JVM heaps were measured in single-digit megabytes

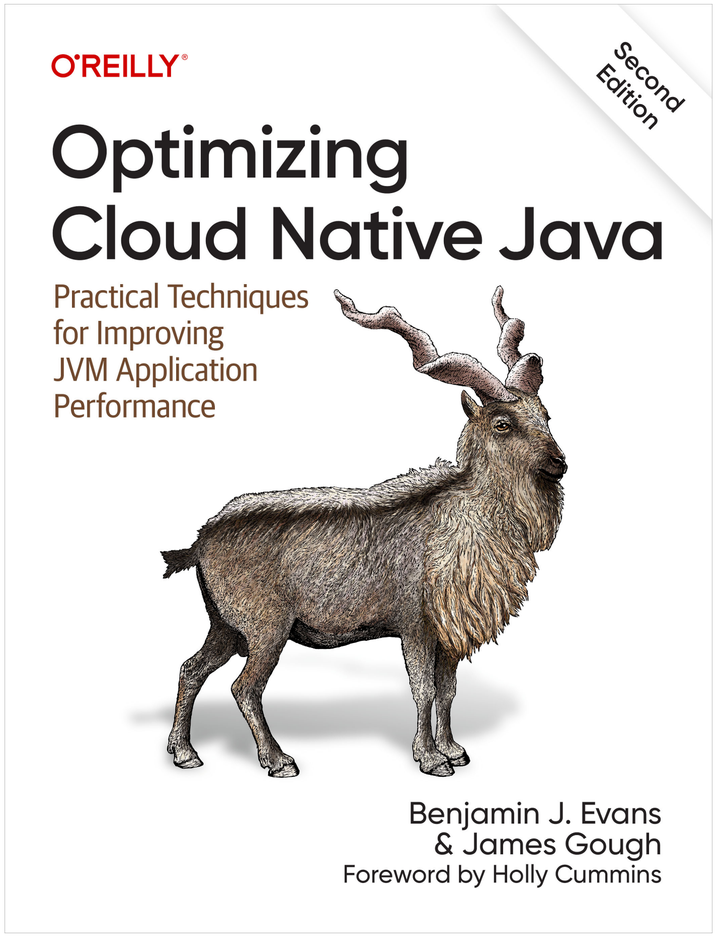
Type Erased

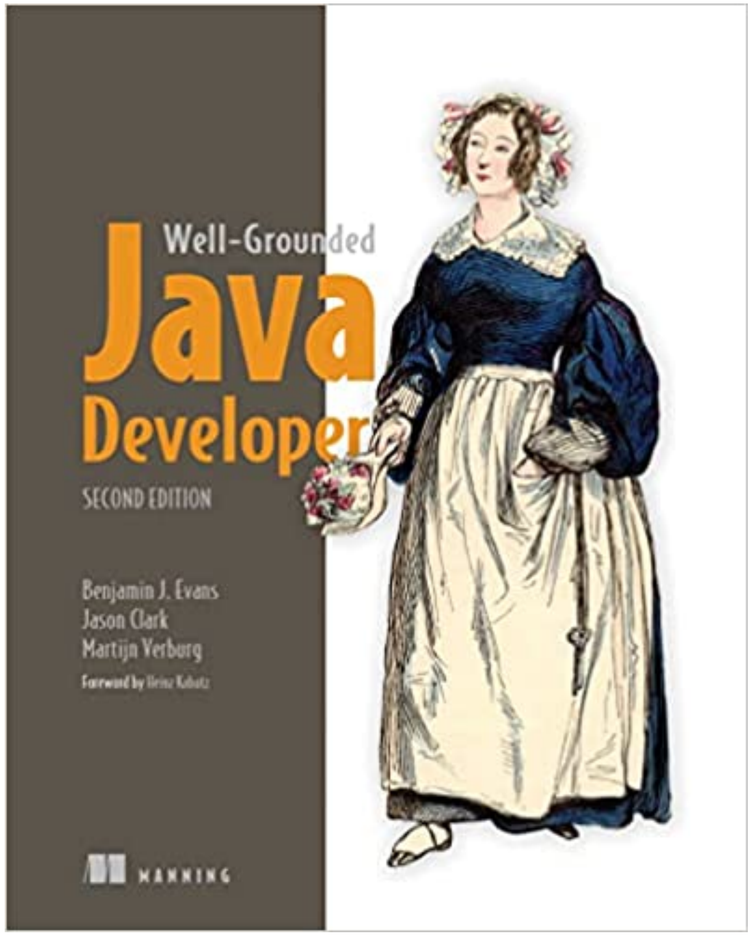
- Java has parametrised types / generics
  - Unbounded type parameters exist only at compile time
  - Generation of bytecode removes information
- Erasure exists for backwards compatibility
  - “Raw types” provided as a back bridge to ancient Java versions
- Often complained about
  - Also often misunderstood

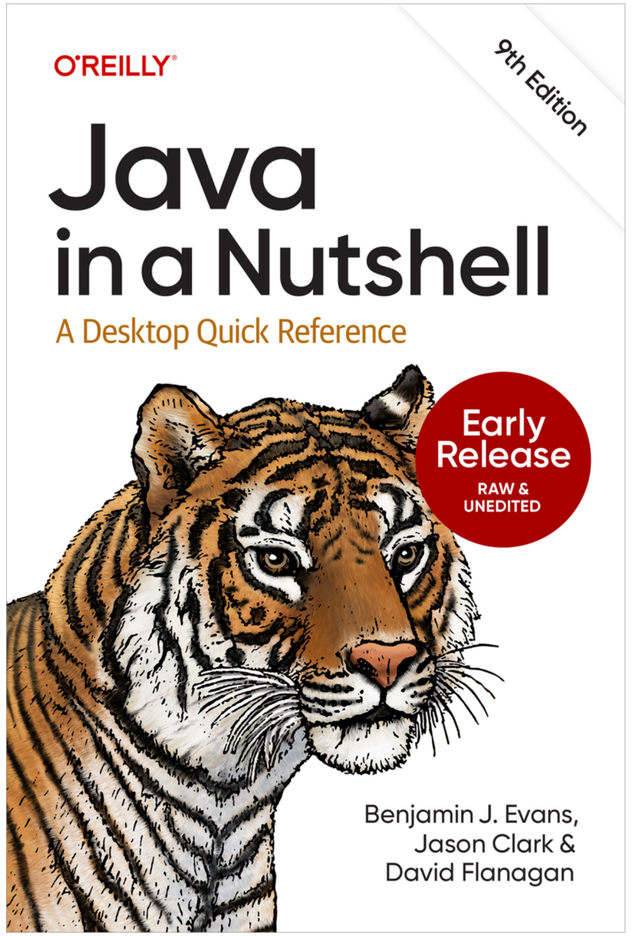
Slightly Functional

- Java isn't (that) functional
  - Implements key idioms (map, filter etc) - but only on `Stream`
  - Java Collections are very concrete
- Type erasure affects higher-order function type safety
- Existence of `void` complicates matters
- No laziness
- No true "monadic" approach
- No HKTs (TypeClass pattern)
- But do these limitations really matter (to most devs)?

Thank You!







https://kittylyst.com