ADVANCING WITH JAVA KCDC 2024

Daniel Hinojosa

JEP

JAVA ENHANCEMENT PROPOSALS

- Enhancement Proposals of Java Features that will be added to future versions of Java
- https://openjdk.org/jeps/0
- All features are typically done in first preview, second preview, and third preview.
- Don’t be surprised if there are fourth and fifth previews for some advanced features that will be introduced.

JAVA ALMANAC

JAVA ALMANAC

- One-stop shop for viewing everything Java
- https://javaalmanac.io/
- Some features that are amazing:
  - Find all APIs, whatever the version
  - Compare and contrast versions, anything from the latest to version 1.0
- Find language specifications, VM Specifications and more

VALUE CLASSES

PRIMITIVE OBSESSION

- Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
- Primitives in this case does not mean, Java primitives only
- It can also mean `List`s, `Set`s, `Map`s

Refactoring Guru: Primitive Obsession

DOMAIN DRIVEN DESIGN AND PRIMITIVE OBSESSION

- Domain Driven Design strongly advocates for Value Classes
  - An email shouldn't be a `String`, there should be an `Email` class that encapsulates what an email is which includes validation
  - A social security number shouldn't be a `String`, there should be a `SocialSecurity` class that encapsulates what a Social Security number is *including* any invariants

DISCUSSION: WHAT ABOUT AGE?

- Given a human age, what would the value class look like?
- How would you model that?

PRIMITIVE OBSESSION LEADS TO FEATURE ENVY

- A method accesses the data of another object more than its own data.
- You will find that when you move data into value classes, methods can get left behind
- Picture have a class called `Bank` that maintains a `balance` but the `deposit` or `withdrawal` are elsewhere?

Refactoring Guru: Primitive Obsession

DEMO: PRIMITIVE OBSESSION AND FEATURE ENVY

Let's discuss, some quick refactoring techniques and then bring up the salient points

- You can domain model with any of the following new features:
  - `records`
  - `enum`
  - `classes`

RECORDS

RECORDS

- Classes that act as transparent carriers for immutable data.
- Records can be thought of as *nominal* tuples. Nominal meaning - *by name*

Source: JEP 395

RECORDS MOTIVATION

- It is a common complaint that "Java is too verbose" or has "too much ceremony". Some of the worst offenders are classes that are nothing more than immutable data carriers for a handful of values.
- Properly writing such a data-carrier class involves a lot of low-value, repetitive, error-prone code: constructors, accessors, `equals`, `hashCode`, `toString`, etc.

Source: JEP 395

RECORDS ANATOMY

- Record classes help to model plain data aggregates with less ceremony than normal classes.
- The declaration of a record class primarily consists of a declaration of its state; the record class then commits to an API that matches that state.
- This means that record classes give up a freedom that classes usually enjoy — the ability to decouple a class's API from its internal representation — but, in return, record class declarations become significantly more concise.

Source: JEP 395

RECORDS EXAMPLE

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

- A record class declaration consists of:
  - A name
  - Optional type parameters
  - A header
  - A body.
- The header lists the components of the record class, which are the variables that make up its state.
- This list of components is sometimes referred to as the state description.

Source: JEP 395

WHAT GETS IMPLICITLY CREATED?

A record class acquires many standard members automatically

- For each component in the header, two members: a public accessor method with the same name and return type as the component, and a private final field with the same type as the component;
- A canonical constructor whose signature is the same as the header, and which assigns each private field to the corresponding argument from a new expression which instantiates the record;
- `equals` and `hashCode` methods which ensure that two record values are equal if they are of the same type and contain equal component values
- A `toString` method that returns a string representation of all the record components, along with their names.

Source: JEP 395

CONSTRUCTOR RULES

- A normal class without any constructor declarations is automatically given a default constructor.
- In contrast, a record class without any constructor declarations is automatically given a canonical constructor that assigns all the private fields to the corresponding arguments of the new expression which instantiated the record.

Source: JEP 395

WHAT YOU CANNOT DO WITH RECORDS

- A record class declaration does not have an extends clause. The superclass of a record class is always `java.lang.Record`, similar to how the superclass of an enum class is always `java.lang.Enum`. Even though a normal class can explicitly extend its implicit superclass `Object`, a record cannot explicitly extend any class, even its implicit superclass `Record`.
- A record class is implicitly `final`, and cannot be abstract. These restrictions emphasize that the API of a record class is defined solely by its state description, and cannot be enhanced later by another class.

Source: JEP 395

WHAT YOU *CANNOT* DO WITH RECORDS

- The fields derived from the record components are final. This restriction embodies an immutable by default policy that is widely applicable for data-carrier classes.
- A record class cannot explicitly declare instance fields, and cannot contain instance initializers. These restrictions ensure that the record header alone defines the state of a record value.

Source: JEP 395

WHAT YOU *CANNOT* DO WITH RECORDS

- Any explicit declarations of a member that would otherwise be automatically derived must match the type of the automatically derived member exactly, disregarding any annotations on the explicit declaration.
- Any explicit implementation of accessors or the `equals` or `hashCode` methods should be careful to preserve the semantic invariants of the record class.
- A record class cannot declare native methods.
  - If a record class could declare a native method then the behavior of the record class would by definition depend on external state rather than the record class's explicit state.
  - No class with native methods is likely to be a good candidate for migration to a record.

Source: JEP 395

WHAT YOU *CAN* DO WITH RECORDS

- Instances of record classes are created using a new expression.
- A record class can be declared top level or nested, and can be generic.
- A record class can declare `static` methods, fields, and initializers.
- A record class can declare instance methods.

Source: JEP 395

WHAT YOU *CAN* DO WITH RECORDS

- A record class can implement `interfaces`.
- A record class cannot specify a superclass since that would mean inherited state, beyond the state described in the header.
- A record class can, however, freely specify superinterfaces and declare instance methods to implement them.
- A record class can declare nested types, including nested record classes.
- If a record class is itself nested, then it is implicitly `static`; this avoids an immediately enclosing instance which would silently add state to the record class.

Source: JEP 395

WHAT YOU *CAN* DO WITH RECORDS

- A `record` class, and the components in its header, may be decorated with annotations.
- Instances of record classes can be serialized and deserialized.
  - However, the process cannot be customized by providing `writeObject`, `readObject`, `readObjectNoData`, `writeExternal`, or `readExternal` methods.
- The components of a record class govern serialization(how it is written), while the canonical constructor of a record class governs deserialization (how it is read)

Source: JEP 395

WARNING: CLASSES NAMED `Record`

- The abstract class `java.lang.Record` is the common superclass of all record classes.
- Every Java source file implicitly imports the `java.lang.Record` class, as well as all other types in the `java.lang` package, regardless of whether you enable or disable preview features.
- However, if your application imports another class named `Record` from a different package, you might get a compiler error.

SOLUTION: CLASSES NAMED `Record`

- Both `Record` in the `com.myapp` package and `Record` in the `java.lang` package are imported with wildcards. Consequently, neither class takes precedence, and the compiler generates an error message when it encounters the use of the simple name `Record`.
- To enable this example to compile, the `import` statement can be changed so that it imports the fully qualified name of `Record`:

```
import com.myapp.Record;
```

REFLECTION API

We add two public methods to `java.lang.Class`

- `RecordComponent[] getRecordComponents()` — Returns an array of `java.lang.reflect.RecordComponent` objects. The elements of this array correspond to the record's components, in the same order as they appear in the record declaration. Additional information can be extracted from each element of the array, including its name, annotations, and accessor method.
- `boolean isRecord()` — Returns `true` if the given class was declared as a record. (Compare with `isEnum`.)

DEMO: RECORDS

In the advancing_with_java_kcdc_2024 project, go to the /src/main/java directory and navigate to the com.evolutionnext.records package.

PATTERN MATCHING

PATTERN MATCHING FOR `instanceof`

- In the following the snippet we don't require

```
if (obj instanceof String) {
    String s = (String) obj;
    // use s
}
```

- Now, we can pattern match without the formal requirement of a cast

```
if (obj instanceof String s) {
    // can use s here
} else {
    // can't use s here
}
```

DEMO: PATTERN MATCHING

In the *advancing_with_java_kcdc_2024/src/test/java* directory, navigate to the `com.xyzcorp.patternmatching` package and open all the classes.

SEALED classES

SEALED `classES`

- Used to model a closed set of classes, much like `enum`s do
- The difference is that this is modelling a "fixed set of kinds of values"
- My take, we are wanting to model

HOW TO CREATE A `sealed` CLASS

- A `sealed class` or `interface` can be extended or implemented only by those classes and interfaces permitted to do so.
- A class is `sealed` by applying the `sealed` modifier to its declaration.
- After any `extends` and `implements` clauses, the `permits` clause specifies the classes that are permitted to extend the sealed class.

EXAMPLE OF A `sealed` CLASS

```
package com.example.geometry;

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

LOCATION OF ALL THE CHILDREN

- The classes specified by `permits` must be located near the superclass, either:
  - In the same module (if the superclass is in a named module)
  - In the same package (if the superclass is in the unnamed module)

`sealed` CLASS RULES

1. The `sealed` class and its permitted subclasses must belong to the same module, and, if declared in an unnamed module, to the same package.

2. Every permitted subclass must directly extend the sealed class.

SEALED CLASS RULES (CONTINUED)

3. Every permitted subclass must use a modifier to describe how it propagates the sealing initiated by its superclass:

    1. A permitted subclass may be declared `final` to prevent its part of the class hierarchy from being extended further. (Record classes are implicitly declared final.)

    2. A permitted `subclass` may be declared `sealed` to allow its part of the hierarchy to be extended further than envisaged by its sealed superclass, but in a restricted fashion.

    3. A permitted subclass may be declared `non-sealed` so that its part of the hierarchy reverts to being open for extension by unknown subclasses. A sealed class cannot prevent its permitted subclasses from doing this. (The modifier `non-sealed` is the first hyphenated keyword proposed for Java.)

DEMO: `sealed` CLASSES AND INTERFACES

In the *advancing_with_java_kcdc_2024/src/main/java* directory, navigate to the `com.xyzcorp.sealed` package, and open the `shapes`, `optionals`, `functional` packages

DISCUSSION: LET'S DISCUSS SOME `sealed` MODELS

- Maybe your domain or maybe some other domains, think of some `sealed` classes
- To get things started how would you model bank transactions?
- How would you model a `List` if you were to use `sealed`?

ENHANCED `switch` EXPRESSIONS

SWITCH EXPRESSIONS

- Previously in Java, since 1.0, there have always been `switch` *statements*
- With the advancement of JEP 361 we now have `switch` _expressions

DEMO: SWITCH EXPRESSIONS

In the *advancing_with_java_kcdc_2024/src/test/java* directory, navigate to the `com.xyzcorp.enhancedswitch` package, and open the `EnhancedSwitchTest`

TRACK ALL THE ADVANCEMENTS WITH `switch`

- Once it is all brought in you will find that all the following are now related
  - Pattern Matching
  - Enhanced `switch` expressions
  - `records`
  - `sealed` classes

ENHANCED `switch` AND PATTERN MATCHING

- Called JEP 441 pattern matching for `switch`
- This fuses the two notions of *pattern matching* and *switch expressions*

DEMO: SWITCH EXPRESSIONS

In the *advancing_with_java_kcdc_2024/src/test/java* directory, navigate to the `com.xyzcorp.enhancedswitch` package, and open the `EnhancedSwitchTest` and view the "Pattern Match with Switch"

PATTERN MATCHING ON `RECORD`S

- JEP 405 introduced us to pattern matching on records
- This gives us the ability to break apart or destructure a `record`

DEMO: SWITCH EXPRESSIONS WITH `recordS`

In the *advancing_with_java_kcdc_2024/src/test/java* directory, navigate to the `com.xyzcorp.enhancedswitch` package and open all the classes. Viewing the records.

PRIMITIVE `switch` HANDLING

JEP 455 now add support for pattern matching primitives!

```
switch (x.getStatus()) {
    case 0 -> "okay";
    case 1 -> "warning";
    case 2 -> "error";
    default -> "unknown status: " + x.getStatus();
}
```

PRIMITIVE PATTERN MATCHING WITH `instanceof`

- Primitive type patterns in instanceof would subsume the lossy conversions built into the Java language
- Avoids the painstaking range checks that developers have been coding by hand for almost three decades.
- If the conversion would lose information then the pattern does not match and the program should handle the invalid input in a different branch.

```
if (getPopulation() instanceof float pop) {
    ... pop ...
}

if (i instanceof byte b) {
    ... b ...
}
```

UNNAMED VARIABLES

UNNAMED VARIABLES

- Developers sometimes declare variables that they do not intend to use, whether as a matter of code style or because the language requires variable declarations in certain contexts.
- The intent of non-use is known at the time the code is written, but if it is not captured explicitly then later maintainers might accidentally use the variable, thereby violating the intent.
- If we could make it impossible to accidentally use such variables then code would be more informative, more readable, and less prone to error.

NOT REQUIRING VARIABLES IN ITERATION

```
static int count(Iterable<Order> orders) {
    int total = 0;
    for (Order order : orders)    // order is unused
        total++;
    return total;
}
```

NOT REQUIRING VARIABLES IN ITERATION

```
static int count(Iterable<Order> orders) {
    int total = 0;
    for (Order _ : orders)    // Unnamed variable
        total++;
    return total;
}
```

NOT REQUIRING VARIABLES IN "TRY-WITH-RESOURCES"

```
try (var acquiredContext = ScopedContext.acquire()) {
    ... acquiredContext not used ...
}
```

NOT REQUIRING VARIABLE IN "TRY-WITH-RESOURCES"

```
try (var _ = ScopedContext.acquire()) {    // Unnamed variable
    ... no use of acquired resource ...
}
```

NOT REQUIRING VARIABLES IN `Exception`S

```
String s = ...;
try {
    int i = Integer.parseInt(s);
    ... i ...
} catch (NumberFormatException ex) {
    System.out.println("Bad number: " + s);
}
```

NOT REQUIRING VARIABLES IN `Exception`S

```
String s = ...
try {
    int i = Integer.parseInt(s);
    ... i ...
} catch (NumberFormatException _) {        // Unnamed variable
    System.out.println("Bad number: " + s);
}
```

NOT REQUIRING VARIABLES IN LAMBDAS

```
stream.collect(Collectors.toMap(String::toUpperCase,
                                v -> "NODATA"));
```

NOT REQUIRING VARIABLES IN LAMBDAS

```
stream.collect(Collectors.toMap(String::toUpperCase,
                                _ -> "NODATA"))
```

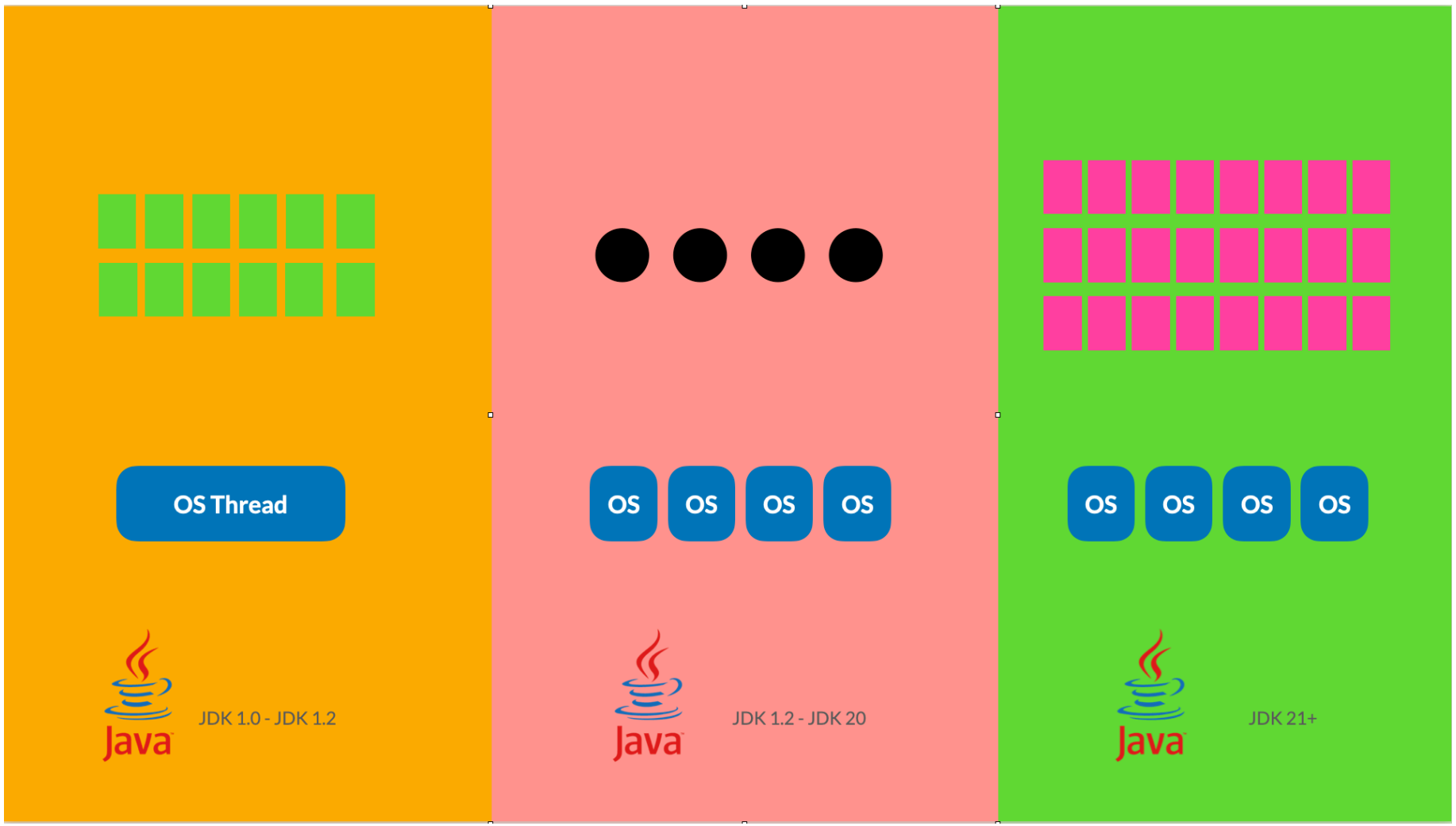
DEMO: SWITCH EXPRESSIONS

In the *advancing_with_java_kcdc_2024/src/test/java* directory, navigate to the `com.xyzcorp.enhancedswitch` package, and open the `EnhancedSwitchTest` and review `testStringMatchWithRecordMatchingWithUnnamedVariablesAndWhen`

VIRTUAL THREADS

VIRTUAL THREADS

- A virtual thread is a Thread — in code, at runtime, in the debugger and in the profiler.
- A virtual thread is not a wrapper around an OS thread, but a Java entity.
- Creating a virtual thread is cheap — have millions, and **don't pool them!**
- Blocking a virtual thread is cheap — **be synchronous**!
- No language changes are needed.
- Pluggable schedulers offer the flexibility of asynchronous programming.


OS Thread
OS
OS
OS
OS
OS
OS
OS
OS
Java
JDK 1.0 - JDK 1.2
Java
JDK 1.2 - JDK 20
Java
JDK 21+

CONTEXT SWITCHING

- If there are more runnable threads than CPUs, eventually the OS will preempt one thread so that another can use the CPU.
- This causes a context switch, which requires saving the execution context of the currently running thread and restoring the execution context of the newly scheduled thread.

SCHEDULERS

- In our dream world, it would be great if we had one to one thread to CPU correspondence
- Either the JVM or the underlying platform's operating system deciphers how to share the processor resource among threads, this is *Thread Scheduling*
- The JVM or OS that performs the scheduling of the threads is the *Thread Scheduler*

USER VS. KERNEL

- User Level is `java.lang.Thread`
  - Runs within the JVM
  - Specific Handling which is up to the language, in this case Java
- Kernel Threads are more general
  - Each user-level thread created by the application is known to the kernel

BLOCKING OPERATIONS

- Operations that will hold the `Thread` until complete
- You cannot avoid it entirely
- Examples:
  - `Thread.sleep`, `Thread.join`
  - `Socket.connect`, `Socket.read`, `Socket.write`
  - `Object.wait`

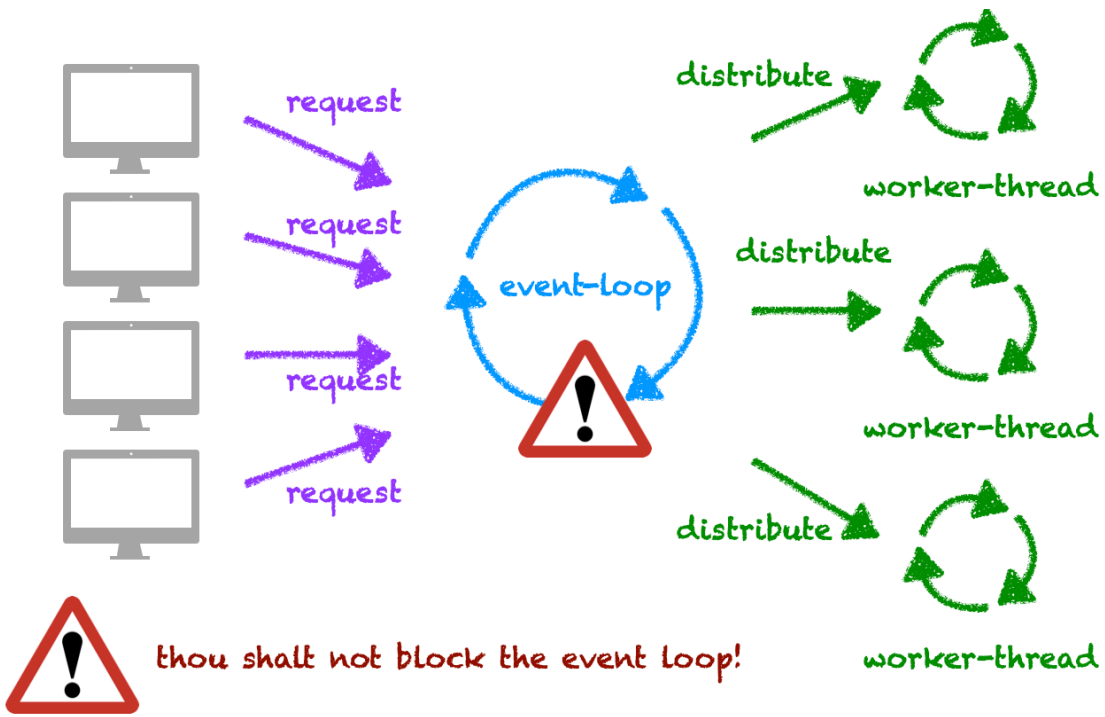
THREAD WEIGHT AND EXPENSE

- Threads by themselves are expensive to create
- We cannot have millions of `Thread`s
- `Thread`s are mostly idle, we are not maximizing its use
- `Thread`s are also non-scalable

HOW DO WE OR HOW DID WE COMPENSATE?

- `java.util.Future<T>`
- Callbacks
- Async/Await
- Promises or Incomplete Futures
- Reactive Programming (RXJava, Reactor)
- Coroutines, Suspendable Functions (Kotlin)

EXAMPLE FROM VERT.X



Vert.x is a concurrency framework build on Futures and Thread Pools. You do have to follow the rules.

RXJAVA/PROJECT REACTOR

```
Observable.just(1, 3, 4, 5, 10, 100)     // io
   .doOnNext(i -> debug("L1", i))        // io
   .observeOn(Schedulers.computation()) // Scheduler change
   .doOnNext(i -> debug("L2", i))        // computation
   .map(i -> i * 10)                     // computation
   .observeOn(Schedulers.io())           // Scheduler change
   .doOnNext(i -> debug("L3", i))        // io
   .subscribeOn(schedulers.io())         // Dictate at beginning to use
   .subscribe(System.out::println);      // Print
```

COMPLEX CANCELLATION

- `Thread`s support a cooperative interruption mechanism comprised of:
  - `interrupt()`
  - `interrupted()`
  - `isInterrupted()`
  - `InterruptedException`
- When a `Thread` is interrupted it has to clear and reset the status because it has to go back into a pool

LOSING STACK TRACES

- Stack Traces are also applicable to the live thread
- If the thread is recycled back into a pool, then the stack trace is lost
- This can prove to be valuable information

VIRTUAL THREADING

- *Virtual threads* are just threads, except lightweight 1kb and fast to create
- Creating and blocking virtual thread is cheap and encouraged. 23 million virtual threads in 16GB of memory
- They are managed by the Java runtime and, unlike the existing platform threads, are not one-to-one wrappers of OS threads, rather, they are implemented in userspace in the JDK.
- Whereas the OS can support up to a few thousand active threads, the Java runtime can support millions of virtual threads
- Every unit of concurrency in the application domain can be represented by its own thread, making programming concurrent applications easier

VIRTUAL THREADING USES CONTINUATIONS

- Object Representing a Computation that may be suspended, resumed, cloned or serialized
- When it reaches a call that blocks it will yield allowing the JVM to use another virtual thread to process

SCHEDULING

- The Kernel Scheduler is very general, and makes assumptions about what the user language requires
- Virtual Threads are tailored and specific for the task and are on the JVM
- Virtual Threads Scheduling with the Kernel Space is abstracted, and typically you don’t need to know much unless you feel pedantic or want to create your own Scheduler
- Your Virtual Threads that you create will be running on a Worker Thread called a "Carrier Thread"

EXECUTING BY CARRIER THREAD

- Schedulers are implemented with a `ForkJoinPool`
- Carrier Threads are daemon threads
- The number of initial threads is `Runtime.getRuntime().availableProcessors()`
- New Threads can be initialized with a Managed Blocker - If a thread is blocked, new threads are created

PARKING

- Parking (blocking) a virtual thread results in yielding its continuation
- Unparking it results in the continuation being resubmitted to the scheduler
- The scheduler worker thread executing a virtual thread (while its continuation is mounted) is called a carrier thread.

RELEARN THE OLD WAYS

- Create your full task represented by your own Thread
- Add Blocking Code
- Forget about:
  - Thread Pools
  - Reactive Frameworks
- If you want to do more, then make more `Threads`!
- It is OK to block, just program without consideration to blocking, virtual threads will handle it for you!

PINNING

- Virtual thread is pinned to its carrier if it is mounted but is in a state in which it cannot be unmounted
- If a virtual thread blocks while pinned, it blocks its carrier
- This behavior is still correct, but it holds on to a worker thread for the duration that the virtual thread is blocked, making it unavailable for other virtual threads
- Occasional pinning is not harmful
- Very frequent pinning, however, will harm throughput

PINNING SITUATIONS

- **Synchronized Block**
  - If you have a `Thread` blocked with `synchronized` opt for `ReentrantLock` or `StampedLock`, which is preferred anyway for better performance
- **JNI (Java Native Interface)**
  - When there is a native frame on the stack — when Java code calls into native code (JNI) that then calls back into Java
  - JNI Pinning will never go away
  - JNI will likely be replaced by the *Foreign Function and Memory API*

DEBUGGING

- Java Debugger Wire Protocol (JDWP) and the Java Debugger Interface (JDI) used by Java debuggers and supports ordinary debugging operations such as breakpoints, single stepping, variable inspection etc., works for virtual threads as it does for classical threads
- Not all debugger operations are supported for virtual threads.
- Some operations pose special challenges, since there can be a million virtual threads, it would take special consideration how to handle that

PROFILING

- It has been proven difficult to profile asynchronous code, particularly since a Thread can be phased out.
- Now that all code in a continuation and a continuations can be members of an overarching context, we can collate all subtasks and maintain information better
- Java Flight Recorder the foundation of profiling and structured logging in the JDK — to support virtual threads.
- Blocked virtual threads can be shown in the profiler and time spent on I/O measured and accounted for.

DEMO: VIRTUAL THREADS

In the *advancing_with_java_kcdc_2024/src/main/java* directory, navigate to the `com.xyzcorp.virtualthread` package and open all the classes.

SCOPED VALUES

SCOPED VALUES

- Enable a method to share immutable data both with its callees within a thread, and with child thread
- Scoped values are easier to reason about than thread-local variables.
- They also have lower space and time costs, especially when used together with virtual threads (JEP 444 and structured concurrency (JEP 480).

Source: JEP 481

GOALS

- *Ease of use* — It should be easy to reason about dataflow.
- *Comprehensibility* — The lifetime of shared data should be apparent from the syntactic structure of code.
- *Robustness* — Data shared by a caller should be retrievable only by legitimate callees.
- *Performance* — Data should be efficiently sharable across a large number of threads.

Source: JEP 481

NON-GOALS

- It is not a goal to change the Java programming language.
- It is not a goal to require migration away from thread-local variables, or to deprecate the existing ThreadLocal API.

Source: JEP 481

WHAT WAS THE PROBLEM WITH THREAD LOCAL?

- *Unconstrained mutability* — Every thread-local variable is mutable: Any code that can call the get method of a thread-local variable can call the set method of that variable at any time.
- *Unbounded lifetime* — Once a thread's copy of a thread-local variable is set via the set method, the value to which it was set is retained for the lifetime of the thread, or until code in the thread calls the remove method.
- *Expensive inheritance* — The overhead of thread-local variables may be worse when using large numbers of threads, because thread-local variables of a parent thread can be inherited by child threads. (A thread-local variable is not, in fact, local to one thread.)

Source: JEP 481

`ThreadLocal`

- If you have a data structure that isn't safe for concurrent access, you can sometimes use an instance per thread, hence `ThreadLocal`.
- `ThreadLocal`s have been used for Thread Locality
- `ThreadLocal` has some shortcomings. They're unstructured, they're mutable
- Once a `ThreadLocal` value is set, it is in effect throughout the thread's lifetime or until it is set to some other value
- `ThreadLocal` is shared among multiple tasks
- `ThreadLocal`s can leak into one another

EXAMPLE OF `ThreadLocal`

`ThreadLocal` may already be set

```
var oldValue = myTL.get();
myTL.set(newValue);
try {
  //...
} finally {
  myTL.set(oldValue);
}
```

COMPLICATED TRACING

- Using `ThreadLocal`s in the way we do threading now, it is complicated to appropriately to do spans
- Spans require inheritance, where a `ThreadLocal` is inherited **by copy** to a subthread.
- `ThreadLocal`s are mutable and cannot be shared hence the copy
- If `ThreadLocal`s were immutable, then it would be efficient to handle
- Be aware that setting the same `ThreadLocal` would cause an `IllegalStateException`

HOW IS THIS TIED TO VIRTUAL THREADS

- The problems of thread-local variables have become more pressing with the availability of virtual threads (JEP 444)
- Can be used for both Virtual Threads, and classic threads.
- The Java Platform should provide a way to maintain inheritable per-thread data for thousands or millions of virtual threads.

Source: JEP 481

WHAT IS THE `ScopedValue` RECIPE?

1. A scoped value is a container object that allows a data value to be safely and efficiently shared by a method with its direct and indirect callees within the same thread, and with child threads, without resorting to method parameters.
2. It is a variable of type `ScopedValue`.
3. It is typically declared as a `final static field`
4. Its accessibility is set to `private` so that it cannot be directly accessed by code in other classes.

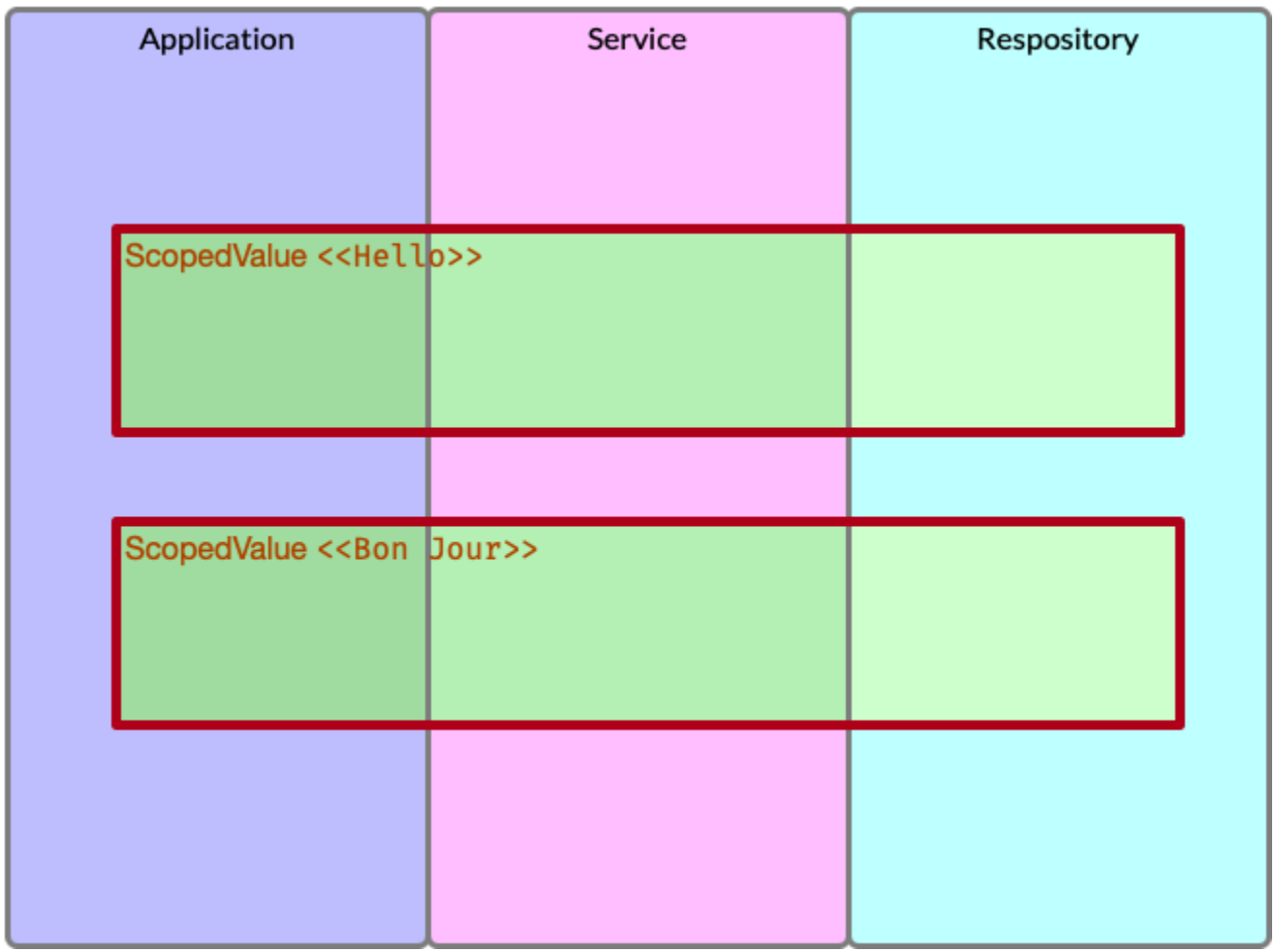
Source: JEP 481

DIFFERENCE BETWEEN `ScopedValue` AND `ThreadLocal`

- Like a thread-local variable, a `ScopedValue` has multiple values associated with it, one per thread.
- The particular value that is used depends on which thread calls its methods.
- Unlike a thread-local variable, a `ScopedValue` is written once, and is available only for a bounded period during execution of the thread.

Source: JEP 481

SCOPED VALUES



WITHOUT SCOPED VALUES

Without Scoped Values, we are left to carry variables, stacks down

```
var x = …
VirtualThread.start(() -> {
   application.method1(x);
}
```

```
application.method1(var
carried)
```

```
service.method3(var carried)
```

```
repository.method2(var carried)
```

WITH SCOPED VALUES

With Scoped Values we can establish a context, and recall it stacks down

```
private final static
ScopedValue<X> scoped =
  ScopeValue.newInstance();

VirtualThread.start(() -> {
  application.method1(scoped);
}
```

```
application.method1()
```

```
service.method3()
```

```
repository.method2() {
   scoped.get()
}
```

DEMO: SCOPED VALUES

In the *advancing_with_java_kcdc_2024/src/main/java* directory, navigate to the `com.xyzcorp.scopedvalues` package and open all the classes.

STRUCTURED CONCURRENCY

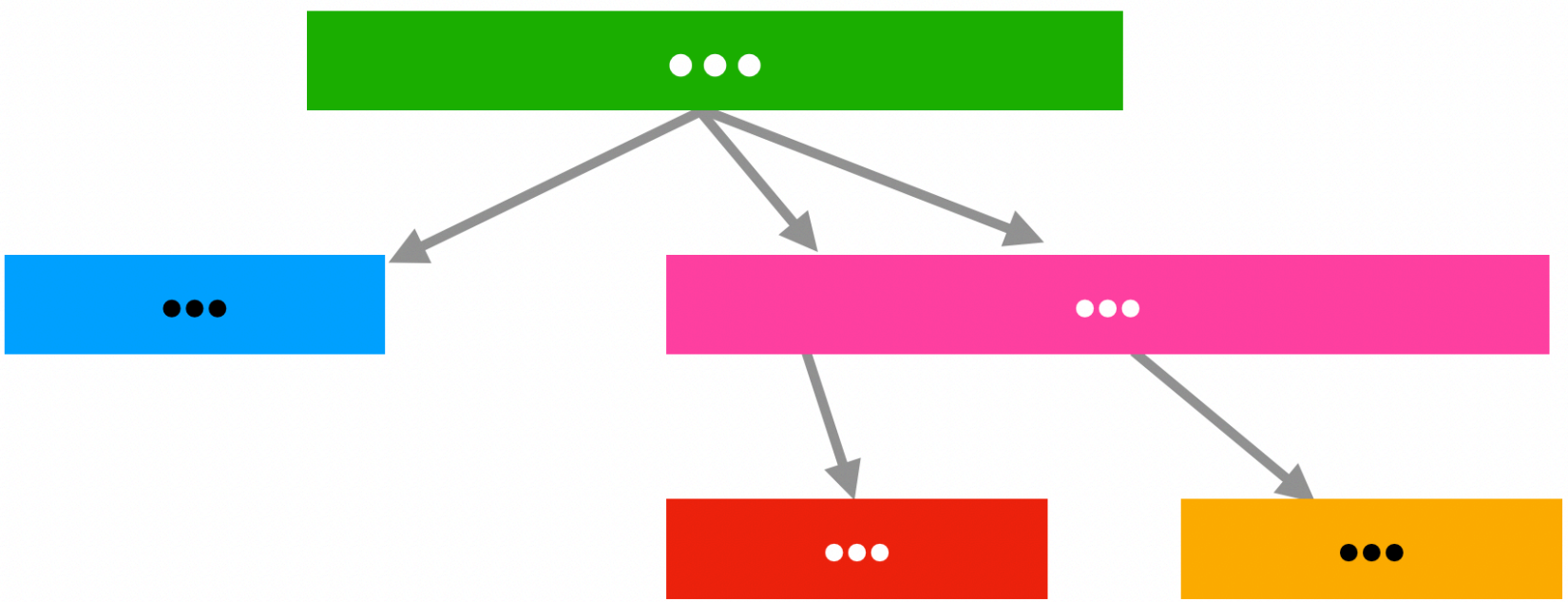
STRUCTURED CONCURRENCY

- **"Launching a task in a new thread is really no better than programming with GOTO"**
- Structured concurrency corrals thread lifetimes into code blocks.
- Similar to how structured programming confines control flow of sequential execution into a well-defined code block, structured concurrency does the same with concurrent control flow

STRUCTURED CONCURRENCY PRINCIPLE

- Threads that are created in some code unit must all terminate by the time we exit that code unit
- If execution splits to multiple thread inside some scope, it must join before exiting the scope.
- Eliminate common risks arising from cancellation and shutdown, such as thread leaks and cancellation delays
- Improve Observability

STRUCTURED CONCURRENCY DIAGRAM



ISSUE WITH `ExecutorService` WITH STRUCTURED CONCURRENCY

- `ExecutorService` can perform tasks concurrently, but can fail *independently*
- `ExecutorService` and `Future<T>` allow unrestricted patterns of concurrency
- `ExecutorService` does not enforce or even track relationships among tasks and subtasks, even though such relationships are common and useful

UNSTRUCTURED CONCURRENCY

- What happens if each of the `Future`'s will fail?
- What happens if any of the `Future`'s will take a long time to complete?

```
Response handle() throws ExecutionException, InterruptedException {
    Future<String>  user  = esvc.submit(() -> findUser());
    Future<Integer> order = esvc.submit(() -> fetchOrder());
    String theUser  = user.get();   // Join findUser
    int    theOrder = order.get();  // Join fetchOrder
    return new Response(theUser, theOrder);
}
```

STRUCTURED CONCURRENCY

Structured concurrency derives from the simple principle that:

If a task splits into concurrent subtasks then they all return to the same place, namely the task's code block.

NOTE THE DIFFERENCE

```
Response handle() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Supplier<String>  user  = scope.fork(() -> findUser());
        Supplier<Integer> order = scope.fork(() -> fetchOrder());

        scope.join()            // Join both subtasks
             .throwIfFailed();  // ... and propagate errors

        // Here, both subtasks have succeeded, so compose their results
        return new Response(user.get(), order.get());
    }
}
```

SHUTDOWN POLICIES

- A ShutdownPolicy describes what should be done when there is an error
- Two subclasses of `StructuredTaskScope`:
  - `ShutdownOnFailure` - (invoke all)
  - `ShutdownOnSuccess` - (invoke any)

As soon as one subtask succeeds this scope automatically shuts down, cancelling unfinished subtasks.

DEMO: STRUCTURED CONCURRENCY

In the *advancing_with_java_kcdc_2024/src/main/java* directory, navigate to the `com.xyzcorp.concurrency.structuredconcurrency` package and open all the classes.

STRING TEMPLATING

STRING TEMPLATES

- Many attempts convey string interpolation has required too much work or been too verbose
- String concatenation has been fraught with potential security issues namely injection

STRING INTERPOLATION IS DANGEROUS

- The convenience of interpolation has a downside: It is easy to construct strings that will be interpreted by other systems but which are dangerously incorrect in those systems.
- Since the Java programming language cannot possibly enforce all such rules, it is up to developers using interpolation to validate and sanitize

SQL INJECTION EXAMPLE

```
String query = "SELECT * FROM Person p WHERE p.last_name = '${name}'";
ResultSet rs = connection.createStatement().executeQuery(query);
```

If `name` had the troublesome `value`

```
Smith' OR p.last_name <> 'Smith
```

then the query string would be

```
SELECT * FROM Person p WHERE p.last_name = 'Smith' OR p.last_name <> 'Smith'
```

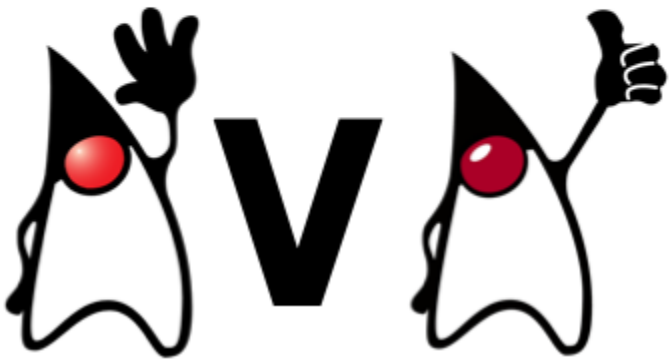
TEMPLATE EXPRESSIONS

- Template expressions can perform string interpolation but are also programmable in a way that helps developers compose strings safely and efficiently.
- In addition, template expressions are not limited to composing strings — they can turn structured text into any kind of object, according to domain-specific rules.

DEMO: TEMPLATE EXPRESSIONS

In the *advancing_with_java_kcdc_2024/src/main/java* directory, navigate to the `com.xyzcorp.stringtemplates` package and open all the classes.

WIN WITH JAVA!



THANK YOU

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext