Deep dive into structured concurrency

Marcin Chrost

DEVOXX POLAND

Platinum Sponsors: GRAND PARADE Red Hat

#DevoxxPL

JEP 453: Structured Concurrency (Preview)

- Introduces an API for structured concurrency.
- Treats groups of related tasks as a single unit of work.
- Simplifies error handling, cancellation, reliability, and observability.

History

- First incubated in JDK 19 (JEP 428).
- Re-incubated in JDK 20 (JEP 437).
- Preview version in JDK 21.

Goals

- Eliminate risks like thread leaks and cancellation delays.
- Improve observability of concurrent code.

Concurrency challenges

Concurrent programming can improve performance but is hard to manage.

Unstructured Concurrency with ExecutorService

- Introduced in Java 5 for concurrent subtask execution
- Subtasks submitted to ExecutorService return Future objects
- Tasks joined via blocking calls to Future.get()

Example of Unstructured Concurrency

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

Issues with Unstructured Concurrency

- **Thread leaks**: Subtasks continue running after failure
- **No interruption propagation**: Parent task failure doesn't stop subtasks
- **Unnecessary waiting**: Tasks block on unrelated subtasks

Observability Challenges

- Task-subtask relationships exist only in the developer's mind
- Thread dumps lack hierarchy or context
- Debugging and troubleshooting become harder

Structure is a king

Task Structure in Single-Threaded Code

- Natural hierarchy of tasks and subtasks in single-threaded code
- Method blocks - tasks, invoked methods - subtasks
- Subtasks must return results or throw exceptions to parent
- No subtask survival beyond calling method's lifetime

Example of Task Structure

```
Response handle() throws IOException {
    String theUser  = findUser();
    int    theOrder = fetchOrder();
    return new Response(theUser, theOrder);
}
```

Benefits of Task Structure

- Parent-child relationships reflected in call stack
- Automatic error propagation
- Obvious hierarchy in thread analysis
- Easy identification of parent task's current activity

Structured Concurrency

- Establishes hierarchical relationships between tasks and subtasks.
- Improves code readability, reliability, and observability.
- Works well with lightweight virtual threads.

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

Structured Task Scope

StructuredTaskScope Overview

- Core class in java.util.concurrent for structured concurrency
- Groups concurrent subtasks into a single unit
- Subtasks executed in separate threads
- Supports forking, joining, and cancellation

Example Using StructuredTaskScope

```
Response handle() throws InterruptedException {
    try (var scope = StructuredTaskScope.open()) {
        Subtask<String> user = scope.fork(() -> findUser());
        Subtask<Integer> order = scope.fork(() -> fetchOrder());
        scope.join();   // Join subtasks, propagating exceptions
        // Both subtasks have succeeded, so compose their results
        return new Response(user.get(), order.get());
    }
}
```

Benefits of StructuredTaskScope

- **Error handling**: Cancels other subtasks if one fails
- **Cancellation propagation**: Interruptions cancel all subtasks
- **Clarity**: Clear and structured task lifecycle
- **Observability**: Thread hierarchy visible in thread dumps

StructuredTaskScope API

- **StructuredTaskScope<T, R>**
  - T: Result type of forked tasks
  - R: Result type of join() method
- **Key methods:**
  - open(): Opens a scope with default policy
  - fork(...): Forks subtasks
  - join(): Joins subtasks, may throw exceptions
  - close(): Cancels scope and waits for termination

General Workflow

1. Open a scope using open() or open(Joiner).
2. Fork subtasks using fork(...).
3. Join subtasks with join().
4. Process the results or handle exceptions.
5. Close the scope (usually via try-with-resources).

Default Completion Policy

- open() creates a scope with default policy:
  - Fails if any subtask fails.
- Custom policies can be implemented using Joiner.

Hierarchy of Scopes

- Subtasks can create their own StructuredTaskScope.
- This creates a **hierarchy of scopes**:
  - Reflects block structure of the code.
  - Guarantees all threads terminate when scope closes.

Key Rules and Behavior

- join() must be called by the **owner thread**.
- Exiting the block before join() cancels the scope.
- Results processed using Subtask::get after joining.
- Subtask::get throws if called before joining.

Cancellation in StructuredTaskScope

- Interruption Handling:
    - Owner thread may be interrupted before or during join().
    - join() throws an exception, and the scope is cancelled.
    - All subtasks are cancelled and waited for termination.
- Subtask Behavior:
    - Subtasks must handle interrupts to finish quickly.
    - Blocking methods that ignore interrupts may delay scope closure.
- close() Method:
    - Always waits for subtasks to finish, even if cancelled.
    - Execution resumes only after all threads terminate.

Scoped Values

- Inheritance of Scoped Values:
  - Subtasks inherit ScopedValue bindings from the scope's owner.
  - Values read by the owner are visible to all subtasks.

Enforced Structured Use

- Runtime Enforcement:
  - Only the scope's owner can call fork() methods.
  - Using a scope outside try-with-resources or improper nesting of close() calls throws StructureViolationException.
- Non-Compatibility with ExecutorService:
  - StructuredTaskScope is not an ExecutorService.
  - Designed for structured concurrency, unlike ExecutorService.
- Migration:
  - Code using ExecutorService can be migrated to StructuredTaskScope for better structure.

Exception Handling Overview

- join() Behavior:
  - Throws FailedException if the scope fails.
  - Cause of failure is the exception from a failed subtask.
- Handling Exceptions:
  - Use try-with-resources to manage scope.
  - Add catch block to handle exceptions after scope closure.

Example: Handling Exceptions

```
try (var scope = StructuredTaskScope.open()) {
    // Fork and join subtasks
} catch (StructuredTaskScope.FailedException e) {
    Throwable cause = e.getCause();
    switch (cause) {
        case IOException ioe -> handleIOException(ioe);
        default -> handleOtherException(cause);
    }
}
```

Configuration in StructuredTaskScope

- Third open() method:
  - Accepts a Joiner and a configuration function.
  - Allows setting:
    - Scope name (for monitoring/management).
    - Timeout.
    - Thread factory (ThreadFactory).

Example: Configuring a Scope

```
<T> List<T> runConcurrently(Collection<Callable<T>> tasks, ThreadFactory factory, Duration timeout)
    throws InterruptedException  {
    try (var scope = StructuredTaskScope.open(Joiner.<T>allSuccessfulOrThrow(),
                                            cf -> cf.withThreadFactory(factory).withTimeout(timeout))) {
        tasks.forEach(scope::fork);
        return scope.join().map(Subtask::get).toList();
    }
}
```

Benefits of Configuration

- Thread factory:
  - Sets thread names or other properties.
- Timeout:
  - Defined as java.time.Duration.
  - Cancels all incomplete subtasks if exceeded.
- Outcome:
  - Better control over threads and execution time.

Joiners

Completion Policies

- Default Completion Policy:
  - If any subtask fails, join() throws an exception, and the scope is cancelled.
  - If all subtasks succeed, join() completes normally and returns null.
- Custom Policies:
  - Use StructuredTaskScope with a Joiner to define custom completion policies.
  - Joiner determines the outcome of join() (e.g., result, stream, or object).

Built-in Joiners

- awaitAllSuccessfulOrThrow(): Waits for all subtasks to succeed.
- allSuccessfulOrThrow(): Like above but allows to return a stream of results
- anySuccessfulResultOrThrow(): Waits for first successful task
- awaitAll(): Waits for all subtasks to complete (regardless of their results).
- allUntil(Predicate<Subtask<? extends T>> isDone): Cancels scope when a condition is met.

Example: anySuccessfulResultOrThrow()

```
<T> T race(Collection<Callable<T>> tasks) throws InterruptedException {
    try (var scope = StructuredTaskScope.open(Joiner.<T>anySuccessfulResultOrThrow())) {
        tasks.forEach(scope::fork);
        return scope.join();
    }
}
```

Example: allSuccessfulOrThrow()

```
<T> List<T> runConcurrently(Collection<Callable<T>> tasks) throws InterruptedException {
    try (var scope = StructuredTaskScope.open(Joiner.<T>allSuccessfulOrThrow())) {
        tasks.forEach(scope::fork);
        return scope.join().map(Subtask::get).toList();
    }
}
```

Best practices

- Always create a new Joiner for each StructuredTaskScope.
- Never reuse Joiner objects across different scopes or after scope closure.

Custom Joiners Overview

- Purpose:
  - Implement custom completion policies for StructuredTaskScope.
  - Define how subtasks are handled and how results are produced.
  - Must ensure thread safety for concurrent operations.
- Key Methods:
  - onFork(Subtask<? extends T> subtask): Invoked when a subtask is forked.
  - onComplete(Subtask<? extends T> subtask): Invoked when a subtask completes.
  - result(): Produces the result for join() or throws an exception.

Example: Collecting successful results

```
class CollectingJoiner<T> implements Joiner<T, Stream<T>> {
    private final Queue<T> results = new ConcurrentLinkedQueue<>();
    public boolean onComplete(Subtask<? extends T> subtask) {
        if (subtask.state() == Subtask.State.SUCCESS) {
            results.add(subtask.get());
        }
        return false; // Do not cancel the scope
    }
    public Stream<T> result() {
        return results.stream();
    }
}
```

Using the Custom Joiner

```
<T> List<T> allSuccessful(List<Callable<T>> tasks) throws InterruptedException {
    try (var scope = StructuredTaskScope.open(new CollectingJoiner<T>())) {
        tasks.forEach(scope::fork);
        return scope.join().toList();
    }
}
```

Observability

- Thread hierarchy visible in JSON thread dumps.
- Easier debugging and monitoring of concurrent applications.

Why fork Doesn't Return Future ?

- Future and CompletableFuture don't align with structured concurrency.
- API focuses on blocking programming rather than asynchronous patterns.

Scoped Values

ThreadLocal variables

- Used to share data between methods on the call stack without method parameters.
- Has one value per thread.
- Value depends on the thread calling `get` or `set` methods.
- InheritableThreadLocal allows values to be inherited by child threads.

ThreadLocal design flaws

- Unconstrained Mutability
  - Any code that can call `get` can also call `set` at any time.
- Unbounded Lifetime
  - Values persist for the thread's lifetime unless `remove` is called.
  - Possible leaks and security vulnerabilities
- Expensive Inheritance
  - Child threads allocate storage for all inherited variables.
  - Rarely needed, as child threads seldom modify inherited variables.

Scoped Values - introduction

- Scoped values are container objects for safely sharing data:
  - Between methods within the same thread.
  - With child threads.
- Alternative to method parameters and thread-local variables.

Scoped Values - key characteristics

- One value per thread, like ThreadLocal variables.
- **Differences from thread-local variables:**
  - Written once.
  - Available only for a bounded period during execution.
- One-way data transmission from caller to callees.

Scoped Values - usage example (1)

```
static final ScopedValue<String> NAME = ScopedValue.newInstance();

// In some method:

where(NAME, "value").run(() -> {

    System.out.println(NAME.get());

    callMethods();

});

// In a method called from the lambda:

System.out.println(NAME.get());
```

Scoped Values - usage example (2)

```
try {
    var result = where(X, "hello").call(() -> bar());
    // call() allows to return value and / or throw checked exception
} catch (Exception e) {
    handleFailure(e);
}
//........
where(X, v).where(Y, w).run(() -> {
    // Operation with X bound to v and Y bound to w
});
```

Scoped Values - rebinding

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

void foo() {

   where(X, "hello").run(() -> bar());

}

void bar() {

    System.out.println(X.get()); // prints hello

    where(X, "goodbye").run(() -> baz());

    System.out.println(X.get()); // prints hello

}

void baz() {

    System.out.println(X.get()); // prints goodbye

}
```

Scoped Values - inheritance

- Automatically inherited by child threads created with StructuredTaskScope
- StructuredTaskScope ensures proper lifecycle management of bindings.

Scoped Values - use cases

- Re-entrant Code
  - Detect recursion using `ScopedValue.isBound`.
  - Model recursion counters with repeated rebinding.
- Nested Transactions
  - Flatten transactions by detecting recursion.
- Graphics Contexts
  - Share drawing contexts with automatic cleanup and re-entrancy.