



- Installing
- Contributing
- Sponsoring
- Developers' Guide
- Vulnerabilities
- JDK GA/EA Builds
- Mailing lists
- Wiki · IRC
- Bylaws · Census
- Legal
- Workshop
- JEP Process
- Source code
- Mercurial
- GitHub
- Tools
- Git
- jtreg harness
- Groups
- (overview)
- Adoption
- Build
- Client Libraries
- Compatibility & Specification
- Review
- Compiler
- Conformance
- Core Libraries
- Governing Board
- HotSpot
- IDE Tooling & Support
- Internationalization
- JMX
- Members
- Networking
- Porters
- Quality
- Security
- Serviceability
- Vulnerability
- Web
- Projects
- (overview, archive)
- Amber
- Audio Engine
- CRaC
- Caciocavallo
- Closures
- Code Tools
- Coin
- Common VM
- Interface
- Compiler Grammar
- Detroit
- Developers' Guide
- Device I/O
- Duke
- Font Scaler
- Galahad
- Graal
- Graphics Rasterizer
- IcedTea
- JDK 7
- JDK 8
- JDK 8 Updates
- JDK 9
- JDK (... , 21, 22)
- JDK Updates
- JavaDoc.Next
- Jigsaw
- Kona
- Kulla
- Lambda
- Lanai
- Leyden
- Lilliput
- Locale Enhancement
- Loom
- Memory Model
- Update
- Metropolis
- Mission Control
- Modules
- Multi-Language VM
- Nashorn
- New I/O
- OpenJFX
- Panama
- Penrose
- Port: AArch32
- Port: AArch64
- Port: BSD
- Port: Haiku
- Port: Mac OS X
- Port: MIPS
- Port: Mobile
- Port: PowerPC/AIX
- Port: RISC-V
- Port: s390x
- Portola
- SCTP
- Shenandoah
- Skara
- Sumatra
- Tiered Attribution
- Tsan
- Type Annotations
- Valhalla
- Verona
- VisualVM
- Wakefield
- Zero
- ZGC



JEP 453: Structured Concurrency (Preview)

<i>Author</i>	Ron Pressler & Alan Bateman
<i>Owner</i>	Alan Bateman
<i>Type</i>	Feature
<i>Scope</i>	SE
<i>Status</i>	Closed / Delivered
<i>Release</i>	21
<i>Component</i>	core-libs
<i>Discussion</i>	loom dash dev at openjdk dot org
<i>Relates to</i>	JEP 437: Structured Concurrency (Second Incubator)
<i>Reviewed by</i>	Brian Goetz
<i>Endorsed by</i>	Brian Goetz
<i>Created</i>	2023/04/21 06:18
<i>Updated</i>	2023/09/26 08:29
<i>Issue</i>	8306641

Summary

Simplify concurrent programming by introducing an API for *structured concurrency*. Structured concurrency treats groups of related tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability. This is a [preview API](#).

History

Structured Concurrency was proposed by [JEP 428](#) and delivered in JDK 19 as an incubating API. It was re-incubated by [JEP 437](#) in JDK 20 with a minor update to inherit scoped values ([JEP 429](#)).

We here propose to make Structured Concurrency a Preview API in the `java.util.concurrent` package. The only significant change is that the `StructuredTaskScope::fork(...)` method returns a [Subtask] rather than a `Future`, as discussed [below](#).

Goals

- Promote a style of concurrent programming that can eliminate common risks arising from cancellation and shutdown, such as thread leaks and cancellation delays.
- Improve the observability of concurrent code.

Non-Goals

- It is not a goal to replace any of the concurrency constructs in the `java.util.concurrent` package, such as `ExecutorService` and `Future`.
- It is not a goal to define the definitive structured concurrency API for the Java Platform. Other structured concurrency constructs can be defined by third-party libraries or in future JDK releases.
- It is not a goal to define a means of sharing streams of data among threads (i.e., [channels](#)). We might propose to do so in the future.
- It is not a goal to replace the existing thread interruption mechanism with a new thread cancellation mechanism. We might propose to do so in the future.

Motivation

Developers manage complexity by breaking tasks down into multiple subtasks. In ordinary single-threaded code, the subtasks execute sequentially. However, if the subtasks are sufficiently independent of each other, and if there are sufficient hardware resources, then the overall task can be made to run faster (i.e., with lower latency) by executing the subtasks concurrently. For example, a task that composes the results of multiple I/O operations will run faster if each I/O operation executes concurrently in its own thread. Virtual threads ([JEP 444](#)) make it cost-effective to dedicate a thread to every such I/O operation, but managing the huge number of threads that can result remains a challenge.

Unstructured concurrency with ExecutorService

The `java.util.concurrent.ExecutorService` API, introduced in Java 5, helps developers execute subtasks concurrently.

For example here is a method, `handle()`, that represents a task in a server application. It handles an incoming request by submitting two subtasks to an `ExecutorService`. One subtask executes the method `findUser()` and the other subtask executes the method `fetchOrder()`. The `ExecutorService` immediately returns a [Future](#) for each subtask, and executes the subtasks concurrently according to the scheduling policy of the `Executor`. The `handle()` method awaits the subtasks' results via blocking calls to their futures' `get()` methods, so the task is said to *join* its subtasks.

```
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);
    }
}
```

Because the subtasks execute concurrently, each subtask can succeed or fail independently. (Failure, in this context, means to throw an exception.) Often, a task such as `handle()` should fail if any of its subtasks fail. Understanding the lifetimes of the threads can be surprisingly complicated when failure occurs:

- If `findUser()` throws an exception then `handle()` will throw an exception when calling `user.get()` but `fetchOrder()` will continue to run in its own thread. This is a *thread leak* which, at best, wastes resources; at worst, the `fetchOrder()` thread will interfere with other tasks.
- If the thread executing `handle()` is interrupted, the interruption will not propagate to the subtasks. Both the `findUser()` and `fetchOrder()` threads will leak, continuing to run even after `handle()` has failed.
- If `findUser()` takes a long time to execute, but `fetchOrder()` fails in the meantime, then `handle()` will wait unnecessarily for `findUser()` by blocking on `user.get()` rather than cancelling it. Only after `findUser()` completes and `user.get()` returns will `order.get()` throw an exception, causing `handle()` to fail.

In each case, the problem is that our program is logically structured with task-subtask relationships, but these relationships exist only in the developer's mind.

This not only creates more room for error, but it makes diagnosing and troubleshooting such errors more difficult. Observability tools such as thread dumps, for example, will show `handle()`, `findUser()`, and `fetchOrder()` on the call stacks of unrelated threads, with no hint of the task-subtask relationship.

We might attempt to do better by explicitly cancelling other subtasks when an error occurs, for example by wrapping tasks with `try-finally` and calling the `cancel(boolean)` methods of the futures of the other tasks in the catch block for the failing task. We would also need to use the `ExecutorService` inside a `try-with-resources statement`, as shown in the examples in [JEP 425](#), because `Future` does not offer a way to wait for a task that has been cancelled. But all this can be very tricky to get right, and it often makes the logical intent of the code harder to discern. Keeping track of the inter-task relationships, and manually adding back the required inter-task cancellation edges, is asking a lot of developers.

This need to manually coordinate lifetimes is due to the fact that `ExecutorService` and `Future` allow unrestricted patterns of concurrency. There are no constraints upon, or ordering of, any of the threads involved. One thread can create an `ExecutorService`, a second thread can submit work to it, and the threads which execute the work have no relationship to either the first or second thread. Moreover, after a thread has submitted work, a completely different thread can await the results of execution. Any code with a reference to a `Future` can join it (i.e., await its result by calling `get()`), even code in a thread other than the one which obtained the `Future`. In effect, a subtask started by one task does not have to return to the task that submitted it. It could return to any of a number of tasks — or even none.

Because `ExecutorService` and `Future` allow for such unstructured use they do not enforce or even track relationships among tasks and subtasks, even though such relationships are common and useful. Accordingly, even when subtasks are submitted and joined in the same task, the failure of one subtask cannot automatically cause the cancellation of another: In the above `handle()` method, the failure of `fetchOrder()` cannot automatically cause the cancellation of `findUser()`. The future for `fetchOrder()` is unrelated to the future for `findUser()`, and neither is related to the thread that will ultimately join it via its `get()` method. Rather than ask developers to manage such cancellation manually, we want to reliably automate it.

Task structure should reflect code structure

In contrast to the freewheeling assortment of threads under `ExecutorService`, the execution of single-threaded code always enforces a hierarchy of tasks and subtasks. The body block `{ . . . }` of a method corresponds to a task, and the methods invoked within the block correspond to subtasks. An invoked method must either return to, or throw an exception to, the method that invoked it. It cannot outlive the method that invoked it, nor can it return or throw an exception to a different method. Thus all subtasks finish before the task, each subtask is a child of its parent, and the lifetime of each subtask relative to the others and to the task is governed by the syntactic block structure of the code.

For example, in this single-threaded version of `handle()` the task-subtask relationship is apparent from the syntactic structure:

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

We do not start the `fetchOrder()` subtask until the `findUser()` subtask has completed, whether successfully or unsuccessfully. If `findUser()` fails then we do not start `fetchOrder()` at all, and the `handle()` task fails implicitly. The fact that a subtask can return only to its parent is significant: It implies that the parent task can implicitly treat the failure of one subtask as a trigger to cancel other unfinished subtasks and then fail itself.

In single-threaded code, the task-subtask hierarchy is reified in the call stack at run time. We thus get the corresponding parent-child relationships, which govern error propagation, for free. When observing a single thread, the hierarchical relationship is obvious: `findUser()` (and later `fetchOrder()`) appear subordinate to `handle()`. This makes it easy to answer the question, "what is `handle()` working on now?"

Concurrent programming would be easier, more reliable, and more observable if the parent-child relationships between tasks and their subtasks were evident from the syntactic structure of the code and also reified at run time — just as for single-threaded code. The syntactic structure would delineate the lifetimes of subtasks and enable a runtime representation of the inter-thread hierarchy, analogous to the intra-thread call stack. That representation would enable error propagation and cancellation as well as meaningful observation of the concurrent program.

(The Java Platform already has an API for imposing structure on concurrent tasks, namely `java.util.concurrent.ForkJoinPool`, which is the execution engine behind parallel streams. However, that API is designed for compute-intensive tasks rather than tasks which involve I/O.)

Structured concurrency

Structured concurrency is an approach to concurrent programming that preserves the natural relationship between tasks and subtasks, which leads to more readable, maintainable, and reliable concurrent code. The term "structured concurrency" was coined by [Martin Sústrik](#) and popularized by [Nathaniel J. Smith](#). Ideas from other languages, such as Erlang's hierarchical supervisors, inform the design of error handling in 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.

In structured concurrency, subtasks work on behalf of a task. The task awaits the subtasks' results and monitors them for failures. As with structured programming techniques for code in a single thread, the power of structured concurrency for multiple threads comes from two ideas: (1) well-defined entry and exit points for the flow of execution through a block of code, and (2) a strict nesting of the lifetimes of operations in a way that mirrors their syntactic nesting in the code.

Because the entry and exit points of a block of code are well defined, the lifetime of a concurrent subtask is confined to the syntactic block of its parent task. Because the lifetimes of sibling subtasks are nested within that of their parent task, they can be reasoned about and managed as a unit. Because the lifetime of the parent task is, in turn, nested within that of its parent, the runtime can reify the hierarchy of tasks into a tree that is the concurrent counterpart of the call stack of a single thread. This allows code to apply policies, such as deadlines, to an entire sub-tree of tasks, and allows observability tools to present subtasks as subordinate to their parent tasks.

Structured concurrency is a great match for virtual threads, which are lightweight threads implemented by the JDK. Many virtual threads share the same operating-system thread, allowing for very large numbers of virtual threads. In addition to being plentiful, virtual threads are cheap enough to represent any concurrent unit of behavior, even behavior that involves I/O. This means that a server application can use structured concurrency to process thousands or millions of incoming requests at once: It can dedicate a new virtual thread to the task of handling each request, and when a task fans out by submitting subtasks for concurrent execution then it can dedicate a new virtual thread to each subtask. Behind the scenes, the task-subtask relationship is reified into a tree by arranging for each virtual thread to carry a reference to its unique parent, similar to how a frame in the call stack refers to its unique caller.

In summary, virtual threads deliver an abundance of threads. Structured concurrency can correctly and robustly coordinate them, and enables observability tools to display threads as they are understood by the developer. Having an API for structured concurrency in the JDK will make it easier to build maintainable, reliable, and observable server applications.

Description

The principal class of the structured concurrency API is `StructuredTaskScope` in the `java.util.concurrent` package. This class allows developers to structure a task as a family of concurrent subtasks, and to coordinate them as a unit. Subtasks are executed in their own threads by *forking* them individually and then *joining* them as a unit and, possibly, cancelling them as a unit. The subtasks' successful results or exceptions are aggregated and handled by the parent task. `StructuredTaskScope` confines the lifetimes of the subtasks to a clear [lexical scope](#) in which all of a task's interactions with its subtasks — forking, joining, cancelling, handling errors, and composing results — takes place.

Here is the `handle()` example from earlier, written to use `StructuredTaskScope` (`ShutdownOnFailure` is explained [below](#)):

```
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());
}
}
```

In contrast to the original example, understanding the lifetimes of the threads involved here is easy: Under all conditions their lifetimes are confined to a lexical scope, namely the body of the try-with-resources statement. Furthermore, the use of StructuredTaskScope ensures a number of valuable properties:

- *Error handling with short-circuiting* — If either the findUser() or fetchOrder() subtasks fail, the other is cancelled if it has not yet completed. (This is managed by the shutdown policy implemented by ShutdownOnFailure; other policies are possible).
- *Cancellation propagation* — If the thread running handle() is interrupted before or during the call to join(), both subtasks are cancelled automatically when the thread exits the scope.
- *Clarity* — The above code has a clear structure: Set up the subtasks, wait for them to either complete or be cancelled, and then decide whether to succeed (and process the results of the child tasks, which are already finished) or fail (and the subtasks are already finished, so there is nothing more to clean up).
- *Observability* — A thread dump, as described [below](#), clearly displays the task hierarchy, with the threads running findUser() and fetchOrder() shown as children of the scope.

StructuredTaskScope is a [preview API](#), disabled by default

To use the StructuredTaskScope API you must enable preview APIs, as follows:

- Compile the program with javac --release 21 --enable-preview Main.java and run it with java --enable-preview Main; or,
- When using the [source code launcher](#), run the program with java --source 21 --enable-preview Main.java; or,
- When using [jshell](#), start it with jshell --enable-preview.

Using StructuredTaskScope

The [StructuredTaskScope](#) API is:

```
public class StructuredTaskScope<T> implements AutoCloseable {

    public <U extends T> Subtask<U> fork(Callable<? extends U> task);
    public void shutdown();

    public StructuredTaskScope<T> join() throws InterruptedException;
    public StructuredTaskScope<T> joinUntil(Instant deadline)
        throws InterruptedException, TimeoutException;
    public void close();

    protected void handleComplete(Subtask<? extends T> handle);
    protected final void ensureOwnerAndJoined();

}
```

The general workflow of code using StructuredTaskScope is:

1. Create a scope. The thread that creates the scope is its *owner*.
2. Use the [fork\(Callable\)](#) method to fork subtasks in the scope.
3. At any time, any of the subtasks, or the scope's owner, may call the scope's [shutdown\(\)](#) method to cancel unfinished subtasks and prevent the forking of new subtasks.
4. The scope's owner joins the scope, i.e., all of its subtasks, as a unit. The owner can call the scope's [join\(\)](#) method, to wait until all subtasks have either completed (successfully or not) or been cancelled via shutdown(). Alternatively, it can call the scope's [joinUntil\(java.time.Instant\)](#) method, to wait up to a deadline.
5. After joining, handle any errors in the subtasks and process their results.
6. Close the scope, usually implicitly via try-with-resources. This shuts down the scope, if it is not already shut down, and waits for any subtasks that have been cancelled but not yet completed to complete.

Each call to fork(...) starts a new thread to execute a subtask, which by default is a virtual thread. A subtask can create its own nested StructuredTaskScope to fork its own subtasks, thus creating a hierarchy. That hierarchy is reflected in the code's block structure, which confines the lifetimes of the subtasks: All of the subtasks' threads are guaranteed to have terminated once the scope is closed, and no thread is left behind when the block exits.

Any subtask in a scope, any sub-subtasks in a nested scope, and the scope's owner can call the scope's shutdown() method at any time to signify that the task is complete — even while other subtasks are still executing. The shutdown() method [interrupts](#) the threads that are still executing subtasks, and causes the join() or joinUntil(Instant) method to return. All subtasks should, therefore, be written

in a way that is responsive to interruption. A new subtask that is forked after a call to `shutdown()` will be in the `UNAVAILABLE` state and will not be run. In effect, `shutdown()` is the concurrent analog of the `break` statement in sequential code.

Calling either `join()` or `joinUntil(Instant)` within a scope is mandatory. If a scope's block exits before joining then the scope will wait for all subtasks to terminate and then throw an exception.

It is possible for a scope's owning thread to be interrupted either before or while joining. For example, it could be a subtask of an enclosing scope that has been shut down. If this occurs then `join()` and `joinUntil(Instant)` will throw an exception because there is no point in continuing. The `try-with-resources` statement will then shut down the scope, which will cancel all the subtasks and wait for them to terminate. This has the effect of automatically propagating the cancellation of the task to its subtasks. If the `joinUntil(Instant)` method's deadline expires before either the subtasks terminate or `shutdown()` is called then it will throw an exception and, again, the `try-with-resources` statement will shut down the scope.

When `join()` completes successfully then each of the subtasks has either completed successfully, failed, or been cancelled because the scope was shut down.

Once joined, the scope's owner handles failed subtasks and processes the results of subtasks that completed successfully; this is typically done by the shutdown policy (see [below](#)). The result of a task that completed successfully can be obtained with the `Subtask.get()` method. The `get()` method never blocks; it throws an `IllegalStateException` if it is mistakenly called before joining or when the subtask has not completed successfully.

Subtasks forked in a scope inherit `ScopedValue` bindings ([JEP 446](#)). If a scope's owner reads a value from a bound `ScopedValue` then each subtask will read the same value.

If a scope's owner is itself a subtask of an existing scope, i.e., it was created as a forked subtask, then that scope becomes the parent of the new scope. Scopes and subtasks thus form a tree.

The structured use of `StructuredTaskScope` is enforced at run time. For example, attempts to call `fork(Callable)` from a thread that is not in the tree hierarchy of the scope — i.e., the owner, the subtasks, and subtasks in nested scopes (sub-subtasks) — will fail with an exception. Using a scope outside of a `try-with-resources` block and returning without calling `close()`, or without maintaining the proper nesting of `close()` calls, may cause the scope's methods to throw a `StructureViolationException`.

`StructuredTaskScope` enforces structure and order upon concurrent operations. Thus it does not implement the `ExecutorService` or `Executor` interfaces since instances of those interfaces are commonly used in a non-structured way (see [below](#)). However, it is straightforward to migrate code that uses `ExecutorService`, but would benefit from structure, to use `StructuredTaskScope`.

In practice, most uses of `StructuredTaskScope` will not utilize the `StructuredTaskScope` class directly, but rather use one of the two subclasses described in the next section that implement shutdown policies. In other scenarios, users will likely write their own subclasses to implement custom shutdown policies.

Shutdown policies

When dealing with concurrent subtasks it is common to use *short-circuiting patterns* to avoid doing unnecessary work. Sometimes it makes sense, for example, to cancel all subtasks if one of them fails (i.e., *invoke all*) or, alternatively, if one of them succeeds (i.e., *invoke any*). Two subclasses of `StructuredTaskScope`, [ShutdownOnFailure](#) and [ShutdownOnSuccess](#), support these patterns with policies that shut down the scope when the first subtask fails or succeeds, respectively.

Shutdown policies additionally provide centralized methods for handling exceptions and, possibly, successful results. This is in line with the spirit of structured concurrency, according to which an entire scope is treated as a unit.

Here is a `StructuredTaskScope` with a shutdown-on-failure policy (used also in the `handle()` example above) that runs a collection of tasks concurrently and fails if any one of them fails:

```
<T> List<T> runAll(List<Callable<T>> tasks)
    throws InterruptedException, ExecutionException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        List<? extends Supplier<T>> suppliers = tasks.stream().map(scope::fork).toList();
        scope.join()
            .throwIfFailed(); // Propagate exception if any subtask fails
        // Here, all tasks have succeeded, so compose their results
        return suppliers.stream().map(Supplier::get).toList();
    }
}
```

Here is a `StructuredTaskScope` with a shutdown-on-success policy that returns the result of the first successful subtask:

```
<T> T race(List<Callable<T>> tasks, Instant deadline)
    throws InterruptedException, ExecutionException, TimeoutException {
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<T>()) {
        for (var task : tasks) {
```

```
        scope.fork(task);
    }
    return scope.joinUntil(deadline)
        .result(); // Throws if none of the subtasks completed successfully
}
}
```

As soon as one subtask succeeds this scope automatically shuts down, cancelling unfinished subtasks. The task fails if all of the subtasks fail or if the given deadline elapses. This pattern can be useful in, for example, server applications that require a result from any one of a collection of redundant services.

While these two shutdown policies are provided out of the box, developers can create custom policies that abstract other patterns (see [below](#)).

Processing results

After joining and centrally processing exceptions via the shutdown policy (e.g., with `ShutdownOnFailure::throwIfFailed`), the scope's owner can process the results of the subtasks using the [Subtask] objects returned from the calls to `fork(...)` if they are not processed by the policy (e.g., by `ShutdownOnSuccess::result()`).

Typically, the only Subtask method that a scope owner will invoke is the `get()` method. All other Subtask methods will ordinarily be used only in the implementation of the `handleComplete(...)` method of custom shutdown policies (see [below](#)). In fact, we recommend that variables referencing a Subtask returned by `fork(...)` be typed as, e.g., `Supplier<String>` rather than `Subtask<String>` (unless, of course, you opt to use `var`). If the shutdown policy itself processes subtask results — as in the case of `ShutdownOnSuccess` — then the Subtask objects returned by `fork(...)` should be avoided altogether, and the `fork(...)` method treated as if it returned `void`. Subtasks should return as their result any information that the scope owner should process after centralized exception handling by the policy.

If the scope owner processes subtask exceptions to produce a composite result, rather than use a shutdown policy, then exceptions can be returned as values from the subtasks. For example, here is a method that runs a list of tasks in parallel and returns a list of completed Futures containing each task's respective successful or exceptional result:

```
<T> List<Future<T>> executeAll(List<Callable<T>> tasks)
    throws InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        List<? extends Supplier<Future<T>>> futures = tasks.stream()
            .map(task -> asFuture(task))
            .map(scope::fork)
            .toList();
        scope.join();
        return futures.stream().map(Supplier::get).toList();
    }
}

static <T> Callable<Future<T>> asFuture(Callable<T> task) {
    return () -> {
        try {
            return CompletableFuture.completedFuture(task.call());
        } catch (Exception ex) {
            return CompletableFuture.failedFuture(ex);
        }
    };
}
```

Custom shutdown policies

`StructuredTaskScope` can be extended, and its protected `handleComplete(...)` method overridden, to implement policies other than those of `ShutdownOnSuccess` and `ShutdownOnFailure`. A subclass can, for example,

- Collect the results of subtasks that complete successfully and ignore subtasks that fail,
- Collect exceptions when subtasks fail, or
- Invoke the `shutdown()` method to shut down and cause `join()` to wake up when some condition arises.

When a subtask completes, even after `shutdown()` has been called, it is reported to the `handleComplete(...)` method as a Subtask:

```
public sealed interface Subtask<T> extends Supplier<T> {
    enum State { SUCCESS, FAILED, UNAVAILABLE }

    State state();
    Callable<? extends T> task();
    T get();
    Throwable exception();
}
```

The `handleComplete(...)` method is called for subtasks that have completed either successfully (the `SUCCESS` state) or unsuccessfully (the `FAILED` state) before `shutdown()` has been called. The `get()` method can only be called if the subtask is in the `SUCCESS` state and the `exception()` method can only be called if the subtask is in the `FAILED` state; calling `get()` or `exception()` in other situations will

cause them to throw an `IllegalStateException`. The `UNAVAILABLE` state indicates one of the following: (1) the subtask was forked but has not yet completed; (2) the subtask completed after shutdown, or (3) the subtask was forked after shutdown and has therefore not started. The `handleComplete(...)` method is never called for a subtask in the `UNAVAILABLE` state.

A subclass will typically define methods to make available results, state, or other outcome to code that executes after the `join()` method returns. A subclass that collects results and ignores subtasks that fail can define a method that returns a collection of results. A subclass that implements a policy to shut down when a subtask fails may define a method to get the exception of the first subtask to fail.

Here is an example of a `StructuredTaskScope` subclass that collects the results of subtasks that complete successfully. It defines the method `results()` to be used by the main task to retrieve the results.

```
class MyScope<T> extends StructuredTaskScope<T> {

    private final Queue<T> results = new ConcurrentLinkedQueue<>();

    MyScope() { super(null, Thread.ofVirtual().factory()); }

    @Override
    protected void handleComplete(Subtask<? extends T> subtask) {
        if (subtask.state() == Subtask.State.SUCCESS)
            results.add(subtask.get());
    }

    @Override
    public MyScope<T> join() throws InterruptedException {
        super.join();
        return this;
    }

    // Returns a stream of results from the subtasks that completed successfully
    public Stream<T> results() {
        super.ensureOwnerAndJoined();
        return results.stream();
    }

}
```

This custom policy can be used like so:

```
<T> List<T> allSuccessful(List<Callable<T>> tasks) throws InterruptedException {
    try (var scope = new MyScope<T>()) {
        for (var task : tasks) scope.fork(task);
        return scope.join()
            .results().toList();
    }
}
```

Fan-in scenarios

The examples above focused on *fan-out* scenarios, which manage multiple concurrent outgoing I/O operations. `StructuredTaskScope` is also useful in *fan-in* scenarios, which manage multiple concurrent incoming I/O operations. In such scenarios we typically create an unknown number of subtasks in response to incoming requests.

Here is an example of a server that forks subtasks to handle incoming connections inside a `StructuredTaskScope`:

```
void serve(ServerSocket serverSocket) throws IOException, InterruptedException {
    try (var scope = new StructuredTaskScope<Void>()) {
        try {
            while (true) {
                var socket = serverSocket.accept();
                scope.fork(() -> handle(socket));
            }
        } finally {
            // If there's been an error or we're interrupted, we stop accepting
            scope.shutdown(); // Close all active connections
            scope.join();
        }
    }
}
```

From the perspective of concurrency, this scenario is different not so much in the direction of requests, but in the duration and number of tasks. Here, unlike the previous examples, the scope's owner is unbounded in its duration — it will stop only when it is interrupted. The number of subtasks is also unknown, since they are forked dynamically in response to external events.

All of the connection-handling subtasks are created within the scope, so it is easy to see their purpose in a thread dump, which will display them as children of the scope's owner. It is also easy to shut down the entire service as a unit.

Observability

We extend the new JSON thread-dump format added by [JEP 444](#) to show StructuredTaskScope's grouping of threads into a hierarchy:

```
$ jcmd <pid> Thread.dump_to_file -format=json <file>
```

The JSON object for each scope contains an array of the threads forked in the scope, together with their stack traces. The owning thread of a scope will typically be blocked in a join method waiting for subtasks to complete; the thread dump makes it easy to see what the subtasks' threads are doing by showing the tree hierarchy imposed by structured concurrency. The JSON object for a scope also has a reference to its parent so that the structure of the program can be reconstituted from the dump.

The [com.sun.management.HotSpotDiagnosticsMXBean](#) API can also be used to generate such thread dumps, either directly or indirectly via the platform [MBeanServer](#) and a local or remote JMX tool.

Why doesn't fork(...) return a Future?

When the StructuredTaskScope API was incubating, the fork(...) method returned a Future. This provided a sense of familiarity, by making fork(...) resemble the existing [ExecutorService::submit](#) method. However, given that StructuredTaskScope is intended to be used differently from ExecutorService — in a structured way, as described above — the use of Future brought more confusion than clarity.

- The familiar use of Future involves calling its get() method, which blocks until a result is available. But in the context of StructuredTaskScope, using Future in this way is not only discouraged but is counterproductive. Structured Future objects should be queried only after join() returns, at which point they are known to be completed or cancelled, and the method that should be used is not the familiar get() but rather the newly introduced resultNow(), which never blocks.
- Some developers wondered why fork(...) did not return more capable CompletableFuture objects instead. Since the Future returned by fork(...) should only be used after it is known to have completed, CompletableFuture would have offered no benefit since its advanced features are helpful only for uncompleted futures. In addition, CompletableFuture is designed for the asynchronous programming paradigm, whereas StructuredTaskScope encourages the blocking paradigm.

Future and CompletableFuture are, in short, designed to offer degrees of freedom that are counterproductive in structured concurrency.

- Structured concurrency is about treating multiple tasks running in different threads as a single unit of work, whereas Future is mostly useful when treating multiple tasks as individual tasks. A scope should block only once to await its subtasks' results, and it should then process exceptions centrally. In the vast majority of cases, therefore, the only method that should be called on a Future returned from fork(...) was resultNow(). This was a marked change from the ordinary use of Future, and the Future interface served as a distraction from its proper use in this context.

In the current API, Subtask::get() behaves exactly as Future::resultNow() did when the API was incubating.

Alternatives

- Enhance the ExecutorService interface. We prototyped an implementation of this interface that always enforces structure and restricts which threads can submit tasks. However, we found it to be problematic because most uses of ExecutorService (and its parent interface Executor) in the JDK and in the ecosystem are not structured. Reusing the same API for a far more restricted concept is bound to cause confusion. For example, passing a structured ExecutorService instance to existing methods that accept this type would be all but certain to throw exceptions in most situations.