Module java.base
Package java.util.concurrent

Class StructuredTaskScope<T>

java.lang.Object
    java.util.concurrent.StructuredTaskScope<T>

Type Parameters:

T - the result type of tasks executed in the task scope

All Implemented Interfaces:

AutoCloseable

Direct Known Subclasses:

StructuredTaskScope.ShutdownOnFailure[PREVIEW], StructuredTaskScope.ShutdownOnSuccess[PREVIEW]

```
public class StructuredTaskScope<T>
extends Object
implements AutoCloseable
```

> **StructuredTaskScope is a preview API of the Java platform.**
> *Programs can only use StructuredTaskScope when preview features are enabled.*
> *Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.*

A basic API for *structured concurrency*. StructuredTaskScope supports cases where a task splits into several concurrent subtasks, and where the subtasks must complete before the main task continues. A StructuredTaskScope can be used to ensure that the lifetime of a concurrent operation is confined by a *syntax block*, just like that of a sequential operation in structured programming.

Basic operation

A StructuredTaskScope is created with one of its public constructors. It defines the fork method to start a thread to execute a subtask, the join method to wait for all subtasks to finish, and the close method to close the task scope. The API is intended to be used with the try-with-resources statement. The intention is that code in the try *block* uses the fork method to fork threads to execute the subtasks, wait

for the subtasks to finish with the `join` method, and then *process the results*. A call to the `fork` method returns a `Subtask`[PREVIEW] to representing the *forked subtask*. Once `join` is called, the `Subtask` can be used to get the result completed successfully, or the exception if the subtask failed.

```
Callable<String> task1 = ...
Callable<Integer> task2 = ...

try (var scope = new StructuredTaskScope<Object>()) {

    Subtask<String> subtask1 = scope.fork(task1);
    Subtask<Integer> subtask2 = scope.fork(task2);

    scope.join();

    ... process results/exceptions ...

} // close
```

The following example forks a collection of homogeneous subtasks, waits for all of them to complete with the `join` method, and uses the `Subtask.State`[PREVIEW] to partition the subtasks into a set of the subtasks that completed successfully and another for the subtasks that failed.

```
List<Callable<String>> callables = ...

try (var scope = new StructuredTaskScope<String>()) {

    List<Subtask<String>> subtasks = callables.stream().map(scope::fork).toList();

    scope.join();

    Map<Boolean, Set<Subtask<String>>> map = subtasks.stream()
            .collect(Collectors.partitioningBy(h -> h.state() == Subtask.State.SUCCESS,
                                               Collectors.toSet()));

} // close
```

To ensure correct usage, the `join` and `close` methods may only be invoked by the *owner* (the thread that opened/created the task scope), and the `close` method throws an exception after closing if the owner did not invoke the `join` method after forking.

`StructuredTaskScope` defines the `shutdown` method to shut down a task scope without closing it. The `shutdown()` method *cancels* all unfinished subtasks by interrupting the threads. It prevents new threads from starting in the task scope. If the owner is waiting in the `join` method then it will wakeup.

Shutdown is used for *short-circuiting* and allow subclasses to implement *policy* that does not require all subtasks to finish.

Subclasses with policies for common cases

Two subclasses of `StructuredTaskScope` are defined to implement policy for common cases:

1. `ShutdownOnSuccess`[PREVIEW] captures the result of the first subtask to complete successfully. Once captured, it shuts down the task scope to interrupt unfinished threads and wakeup the owner. This class is intended for cases where the result of any subtask will do ("invoke any") and where there is no need to wait for results of other unfinished subtasks. It defines methods to get the first result or throw an exception if all subtasks fail.
2. `ShutdownOnFailure`[PREVIEW] captures the exception of the first subtask to fail. Once captured, it shuts down the task scope to interrupt unfinished threads and wakeup the owner. This class is intended for cases where the results of all subtasks are required ("invoke all"); if any subtask fails then the results of other unfinished subtasks are no longer needed. If defines methods to throw an exception if any of the subtasks fail.

The following are two examples that use the two classes. In both cases, a pair of subtasks are forked to fetch resources from two URL locations "left" and "right". The first example creates a ShutdownOnSuccess object to capture the result of the first subtask to complete successfully, cancelling the other by way of shutting down the task scope. The main task waits in `join` until either subtask completes with a result or both subtasks fail. It invokes `result(Function)`[PREVIEW] method to get the captured result. If both subtasks fail then this method throws a `WebApplicationException` with the exception from one of the subtasks as the cause.

```
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<String>()) {

        scope.fork(() -> fetch(left));
        scope.fork(() -> fetch(right));

        scope.join();

        String result = scope.result(e -> new WebApplicationException(e));
```

```
        ...
    }
```

The second example creates a ShutdownOnFailure object to capture the exception of the first subtask to fail, cancelling the other by way of shutting down the task scope. The main task waits in `joinUntil(Instant)` until both subtasks complete with a result, either fails, or a deadline is reached. It invokes `throwIfFailed(Function)`[PREVIEW] to throw an exception if either subtask fails. This method is a no-op if both subtasks complete successfully. The example uses `Supplier.get()` to get the result of each subtask. Using `Supplier` instead of `Subtask` is preferred for common cases where the object returned by fork is only used to get the result of a subtask that completed successfully.

```
    Instant deadline = ...

    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {

        Supplier<String> supplier1 = scope.fork(() -> query(left));
        Supplier<String> supplier2 = scope.fork(() -> query(right));

        scope.joinUntil(deadline);

        scope.throwIfFailed(e -> new WebApplicationException(e));

        // both subtasks completed successfully
        String result = Stream.of(supplier1, supplier2)
                .map(Supplier::get)
                .collect(Collectors.joining(", ", "{ ", " }"));

        ...
    }
```

Extending StructuredTaskScope

`StructuredTaskScope` can be extended, and the `handleComplete` method overridden, to implement policies other than those implemented by `ShutdownOnSuccess` and `ShutdownOnFailure`. A subclass may, for example, collect the results of subtasks that complete successfully and ignore subtasks that fail. It may collect exceptions when subtasks fail. It may invoke the `shutdown` method to shut down and cause `join` to wakeup when some condition arises.

A subclass will typically define methods to make available results, state, or other outcome to code that executes after the `join` method. A subclass that collects results and ignores subtasks that fail may define a method that returns the 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.

The following is an example of a simple `StructuredTaskScope` implementation that collects homogenous subtasks that complete successfully. It defines the method "`completedSuccessfully()`" that the main task can invoke after it joins.

```
class CollectingScope<T> extends StructuredTaskScope<T> {
    private final Queue<Subtask<? extends T>> subtasks = new LinkedTransferQueue<>();

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

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

    public Stream<Subtask<? extends T>> completedSuccessfully() {
        super.ensureOwnerAndJoined();
        return subtasks.stream();
    }
}
```

The implementations of the `completedSuccessfully()` method in the example invokes `ensureOwnerAndJoined()` to ensure that the method can only be invoked by the owner thread and only after it has joined.

Tree structure

Task scopes form a tree where parent-child relations are established implicitly when opening a new task scope:

- A parent-child relation is established when a thread started in a task scope opens its own task scope. A thread started in task scope "A" that opens task scope "B" establishes a parent-child relation where task scope "A" is the parent of task scope "B".

- A parent-child relation is established with nesting. If a thread opens task scope "B", then opens task scope "C" (before it closes "B"), then the enclosing task scope "B" is the parent of the nested task scope "C".

The *descendants* of a task scope are the child task scopes that it is a parent of, plus the descendants of the child task scopes, recursively.

The tree structure supports:

- Inheritance of scoped values[PREVIEW] across threads.
- Confinement checks. The phrase "threads contained in the task scope" in method descriptions means threads started in the task scope or descendant scopes.

The following example demonstrates the inheritance of a scoped value. A scoped value `USERNAME` is bound to the value `"duke"`. A `StructuredTaskScope` is created and its `fork` method invoked to start a thread to execute `childTask`. The thread inherits the scoped value *bindings* captured when creating the task scope. The code in `childTask` uses the value of the scoped value and so reads the value `"duke"`.

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

    ScopedValue.runWhere(USERNAME, "duke", () -> {
        try (var scope = new StructuredTaskScope<String>()) {

            scope.fork(() -> childTask());
            ...
         }
    });

    ...

    String childTask() {
        String name = USERNAME.get();   // "duke"
        ...
    }
```

`StructuredTaskScope` does not define APIs that exposes the tree structure at this time.

Unless otherwise specified, passing a `null` argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

Memory consistency effects

Actions in the owner thread of, or a thread contained in, the task scope prior to forking of a subtask *happen-before* any actions taken by that subtask, which in turn *happen-before* the subtask result is retrieved[PREVIEW] or *happen-before* any actions taken in a thread after joining of the task scope.

See *Java Language Specification*:

17.4.5 Happens-before Order

Since:

21

Nested Class Summary

Nested Classes

Modifier and Type	Class	Description
`static final class`	`StructuredTaskScope.ShutdownOnFailure`[PREVIEW]	**Preview.** A `StructuredTaskScope` that captures the exception of the first subtask to fail[PREVIEW].
`static final class`	`StructuredTaskScope.ShutdownOnSuccess`[PREVIEW]`<T>`	**Preview.** A `StructuredTaskScope` that captures the result of the first subtask to complete successfully[PREVIEW].
`static interface`	`StructuredTaskScope.Subtask`[PREVIEW]`<T>`	**Preview.** Represents a subtask forked with `fork(Callable)`.

Constructor Summary

Constructors

Constructor	Description

`StructuredTaskScope()`	Creates an unnamed structured task scope that creates virtual threads.
`StructuredTaskScope(String name, ThreadFactory factory)`	Creates a structured task scope with the given name and thread factory.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
void	`close()`	Closes this task scope.
protected final void	`ensureOwnerAndJoined()`	Ensures that the current thread is the owner of this task scope and that it joined (with `join()` or `joinUntil(Instant)`) after forking subtasks.
<U extends T> StructuredTaskScope.Subtask[PREVIEW] <U>	`fork(Callable<? extends U> task)`	Starts a new thread in this task scope to execute a value-returning task, thus creating a *subtask* of this task scope.
protected void	`handleComplete (StructuredTaskScope.Subtask[PREVIEW]<? extends T> subtask)`	Invoked by a subtask when it completes successfully or fails in this task scope.
final boolean	`isShutdown()`	Returns true if this task scope is shutdown, otherwise false.
StructuredTaskScope[PREVIEW]<T>	`join()`	Wait for all subtasks started in this task scope to finish or the task scope to shut down.
StructuredTaskScope[PREVIEW]<T>	`joinUntil(Instant deadline)`	Wait for all subtasks started in this task scope to finish or the task scope to shut

		down, up to the given deadline.
void	**shutdown**()	Shut down this task scope without closing it.

Methods declared in class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Details

StructuredTaskScope

```
public StructuredTaskScope(String name,
                           ThreadFactory factory)
```

Creates a structured task scope with the given name and thread factory. The task scope is optionally named for the purposes of monitoring and management. The thread factory is used to `create` threads when subtasks are forked. The task scope is owned by the current thread.

Construction captures the current thread's scoped value[PREVIEW] bindings for inheritance by threads started in the task scope. The Tree Structure section in the class description details how parent-child relations are established implicitly for the purpose of inheritance of scoped value bindings.

Parameters:

`name` - the name of the task scope, can be null

`factory` - the thread factory

StructuredTaskScope

```
public StructuredTaskScope()
```

Creates an unnamed structured task scope that creates virtual threads. The task scope is owned by the current thread.

Implementation Requirements:

This constructor is equivalent to invoking the 2-arg constructor with a name of `null` and a thread factory that creates virtual threads.

Method Details

ensureOwnerAndJoined

```
protected final void ensureOwnerAndJoined()
```

Ensures that the current thread is the owner of this task scope and that it joined (with `join()` or `joinUntil(Instant)`) after forking subtasks.

API Note:

This method can be used by subclasses that define methods to make available results, state, or other outcome to code intended to execute after the join method.

Throws:

`WrongThreadException` - if the current thread is not the task scope owner

`IllegalStateException` - if the task scope is open and task scope owner did not join after forking

handleComplete

```
protected void handleComplete(StructuredTaskScope.Subtask[PREVIEW]<? extends T> subtask)
```

Invoked by a subtask when it completes successfully or fails in this task scope. This method is not invoked if a subtask completes after the task scope is shut down.

API Note:

The `handleComplete` method should be thread safe. It may be invoked by several threads concurrently.

Implementation Requirements:

The default implementation throws `NullPointerException` if the subtask is `null`. It throws `IllegalArgumentException` if the subtask has not completed.

Parameters:

`subtask` - the subtask

Throws:

`IllegalArgumentException` - if called with a subtask that has not completed

fork

```
public <U extends T>
StructuredTaskScope.Subtask[PREVIEW]<U> fork(Callable<? extends U> task)
```

Starts a new thread in this task scope to execute a value-returning task, thus creating a *subtask* of this task scope.

The value-returning task is provided to this method as a `Callable`, the thread executes the task's `call` method. The thread is created with the task scope's `ThreadFactory`. It inherits the current thread's scoped value[PREVIEW] bindings. The bindings must match the bindings captured when the task scope was created.

This method returns a `Subtask`[PREVIEW] to represent the *forked subtask*. The `Subtask` object can be used to obtain the result when the subtask completes successfully, or the exception when the subtask fails. To ensure correct usage, the `get()`[PREVIEW] and `exception()`[PREVIEW] methods may only be called by the task scope owner after it has waited for all threads to finish with the `join` or `joinUntil(Instant)` methods. When the subtask completes, the thread invokes the `handleComplete` method to consume the completed subtask. If the task scope is shut down before the subtask completes then the `handleComplete` method will not be invoked.

If this task scope is shutdown (or in the process of shutting down) then the subtask will not run and the `handleComplete` method will not be invoked.

This method may only be invoked by the task scope owner or threads contained in the task scope.

Implementation Requirements:

This method may be overridden for customization purposes, wrapping tasks for example. If overridden, the subclass must invoke `super.fork` to start a new thread in this task scope.

Type Parameters:

`U` - the result type

Parameters:

`task` - the value-returning task for the thread to execute

Returns:

the subtask

Throws:

`IllegalStateException` - if this task scope is closed

`WrongThreadException` - if the current thread is not the task scope owner or a thread contained in the task scope

`StructureViolationException`[PREVIEW] - if the current scoped value bindings are not the same as when the task scope was created

`RejectedExecutionException` - if the thread factory rejected creating a thread to run the subtask

join

```
public StructuredTaskScope[PREVIEW]<T> join()
                                  throws InterruptedException
```

Wait for all subtasks started in this task scope to finish or the task scope to shut down.

This method waits for all subtasks by waiting for all threads started in this task scope to finish execution. It stops waiting when all threads finish, the task scope is shut down, or the current thread is interrupted.

This method may only be invoked by the task scope owner.

Implementation Requirements:

This method may be overridden for customization purposes or to return a more specific return type. If overridden, the subclass must invoke `super.join` to ensure that the method waits for threads in this task scope to finish.

Returns:

this task scope

Throws:

`IllegalStateException` - if this task scope is closed

`WrongThreadException` - if the current thread is not the task scope owner

`InterruptedException` - if interrupted while waiting

joinUntil

```
public StructuredTaskScope<T> joinUntil(Instant deadline)
                                        throws InterruptedException,
                                               TimeoutException
```

Wait for all subtasks started in this task scope to finish or the task scope to shut down, up to the given deadline.

This method waits for all subtasks by waiting for all threads started in this task scope to finish execution. It stops waiting when all threads finish, the task scope is shut down, the deadline is reached, or the current thread is interrupted.

This method may only be invoked by the task scope owner.

Implementation Requirements:

This method may be overridden for customization purposes or to return a more specific return type. If overridden, the subclass must invoke `super.joinUntil` to ensure that the method waits for threads in this task scope to finish.

Parameters:

`deadline` - the deadline

Returns:

this task scope

Throws:

`IllegalStateException` - if this task scope is closed

`WrongThreadException` - if the current thread is not the task scope owner

`InterruptedException` - if interrupted while waiting

`TimeoutException` - if the deadline is reached while waiting

shutdown

```
public void shutdown()
```

Shut down this task scope without closing it. Shutting down a task scope prevents new threads from starting, interrupts all unfinished threads, and causes the `join` method to wakeup. Shutdown is useful for cases where the results of unfinished subtasks are no longer needed. It will typically be called by the `handleComplete(Subtask)` implementation of a subclass that implements a policy to discard unfinished tasks once some outcome is reached.

More specifically, this method:

- Interrupts all unfinished threads in the task scope (except the current thread).
- Wakes up the task scope owner if it is waiting in `join()` or `joinUntil(Instant)`. If the task scope owner is not waiting then its next call to `join` or `joinUntil` will return immediately.

The state[PREVIEW] of unfinished subtasks that complete at around the time that the task scope is shutdown is not defined. A subtask that completes successfully with a result, or fails with an exception, at around the time that the task scope is shutdown may or may not *transition* to a terminal state.

This method may only be invoked by the task scope owner or threads contained in the task scope.

API Note:

There may be threads that have not finished because they are executing code that did not respond (or respond promptly) to thread interrupt. This method does not wait for these threads. When the owner invokes the `close` method to close the task scope then it will wait for the remaining threads to finish.

Implementation Requirements:

This method may be overridden for customization purposes. If overridden, the subclass must invoke `super.shutdown` to ensure that the method shuts down the task scope.

Throws:

`IllegalStateException` - if this task scope is closed

`WrongThreadException` - if the current thread is not the task scope owner or a thread contained in the task scope

See Also:

`isShutdown()`

isShutdown

```
public final boolean isShutdown()
```

Returns true if this task scope is shutdown, otherwise false.

Returns:

true if this task scope is shutdown, otherwise false

See Also:

shutdown()

close

```
public void close()
```

Closes this task scope.

This method first shuts down the task scope (as if by invoking the shutdown method). It then waits for the threads executing any unfinished tasks to finish. If interrupted, this method will continue to wait for the threads to finish before completing with the interrupt status set.

This method may only be invoked by the task scope owner. If the task scope is already closed then the task scope owner invoking this method has no effect.

A StructuredTaskScope is intended to be used in a *structured manner*. If this method is called to close a task scope before nested task scopes are closed then it closes the underlying construct of each nested task scope (in the reverse order that they were created in), closes this task scope, and then throws StructureViolationException[PREVIEW]. Similarly, if this method is called to close a task scope while executing with scoped value[PREVIEW] bindings, and the task scope was created before the scoped values were bound, then StructureViolationException is thrown after closing the task scope. If a thread terminates without first closing task scopes that it owns then termination will cause the underlying construct of each of its open tasks scopes to be closed. Closing is performed in the reverse order that the task scopes were created in. Thread termination may therefore be delayed when the task scope owner has to wait for threads forked in these task scopes to finish.

Specified by:

close in interface AutoCloseable

Implementation Requirements:

This method may be overridden for customization purposes. If overridden, the subclass must invoke super.close to close the task scope.

Throws:

IllegalStateException - thrown after closing the task scope if the task scope owner did not attempt to join after forking

WrongThreadException - if the current thread is not the task scope owner

StructureViolationException[PREVIEW] - if a structure violation was detected