Module jdk.incubator.concurrent
Package jdk.incubator.concurrent

Class ScopedValue<T>

java.lang.Object
    jdk.incubator.concurrent.ScopedValue<T>

Type Parameters:

T - the type of the object bound to this ScopedValue

```
public final class ScopedValue<T>
extends Object
```

A value that is set once and is then available for reading for a bounded period of execution by a thread. A ScopedValue allows for safely and efficiently sharing data for a bounded period of execution without passing the data as method arguments.

ScopedValue defines the where(ScopedValue, Object, Runnable) method to set the value of a ScopedValue for the bouned period of execution by a thread of the runnable's run method. The unfolding execution of the methods executed by run defines a ***dynamic scope***. The scoped value is bound while executing in the dynamic scope, it reverts to being *unbound* when the run method completes (normally or with an exception). Code executing in the dynamic scope uses the ScopedValue get method to read its value.

Like a thread-local variable, a scoped value has multiple incarnations, one per thread. The particular incarnation that is used depends on which thread calls its methods.

Consider the following example with a scoped value USERNAME that is *bound* to the value "duke" for the execution, by a thread, of a run method that invokes doSomething().

```
    private static final ScopedValue<String> USERNAME = ScopedValue.newInstanc

    ScopedValue.where(USERNAME, "duke", () -> doSomething());
```

Code executed directly or indirectly by doSomething() that invokes USERNAME.get() will read the value "duke". The scoped value is bound while executing doSomething() and becomes unbound when doSomething() completes (normally or with an exception). If one thread were to call doSomething() with USERNAME bound to "duke1", and another thread were to call the method with USERNAME bound to "duke2", then USERNAME.get() would read the value "duke1" or "duke2", depending on which thread is executing.

In addition to the where method that executes a run method, ScopedValue defines the where(ScopedValue, Object, Callable) method to execute a method that returns a result. It also defines the where(ScopedValue, Object) method for cases where it is useful to accumulate mappings of ScopedValue to value.

A ScopedValue will typically be declared in a final and static field. The accessibility of the field will determine which components can bind or read its value.

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

Rebinding

The `ScopedValue` API allows a new binding to be established for *nested dynamic scopes*. This is known as *rebinding*. A `ScopedValue` that is bound to some value may be bound to a new value for the bounded execution of some method. The unfolding execution of code executed by that method defines the nested dynamic scope. When the method completes (normally or with an exception), the value of the `ScopedValue` reverts to its previous value.

In the above example, suppose that code executed by `doSomething()` binds `USERNAME` to a new value with:

```
    ScopedValue.where(USERNAME, "duchess", () -> doMore());
```

Code executed directly or indirectly by `doMore()` that invokes `USERNAME.get()` will read the value "`duchess`". When `doMore()` completes (normally or with an exception), the value of `USERNAME` reverts to "`duke`".

Inheritance

`ScopedValue` supports sharing data across threads. This sharing is limited to structured cases where child threads are started and terminate within the bounded period of execution by a parent thread. More specifically, when using a `StructuredTaskScope`, scoped value bindings are *captured* when creating a `StructuredTaskScope` and inherited by all threads started in that scope with the `fork` method.

In the following example, the `ScopedValue` `USERNAME` is bound to the value "`duke`" for the execution of a runnable operation. The code in the `run` method creates a `StructuredTaskScope` and forks three child threads. Code executed directly or indirectly by these threads running `childTask1()`, `childTask2()`, and `childTask3()` will read the value "`duke`".

```
    private static final ScopedValue<String> USERNAME = ScopedValue.newInstanc

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

            scope.fork(() -> childTask1());
            scope.fork(() -> childTask2());
            scope.fork(() -> childTask3());

            ...
         }
    });
```

Implementation Note:

Scoped values are designed to be used in fairly small numbers. `get()` initially performs a search through enclosing scopes to find a scoped value's innermost binding. It then caches the result of the search in a small thread-local cache. Subsequent invocations of `get()` for that scoped value will almost always be very fast. However, if a program has many scoped values that it uses cyclically, the cache hit rate will be low and performance will be poor. This design allows scoped-value inheritance by `StructuredTaskScope` threads to be very fast: in essence, no more than copying a pointer, and leaving a scoped-value binding also requires little more than updating a pointer.

Because the scoped-value per-thread cache is small, clients should minimize the number of bound scoped values in use. For example, if it is necessary to pass a number of values in this way, it makes sense to create a record class to hold those values, and then bind a single `ScopedValue` to an instance of that record.

For this incubator release, the reference implementation provides some system properties to tune the performance of scoped values.

The system property `jdk.incubator.concurrent.ScopedValue.cacheSize` controls the size of the (per-thread) scoped-value cache. This cache is crucial for the performance of scoped values. If it is too small, the runtime library will repeatedly need to scan for each `get()`. If it is too large, memory will be unnecessarily consumed. The default scoped-value cache size is 16 entries. It may be varied from 2 to 16 entries in size. `ScopedValue.cacheSize` must be an integer power of 2.

For example, you could use `-Djdk.incubator.concurrent.ScopedValue.cacheSize=8`.

The other system property is `jdk.preserveScopedValueCache`. This property determines whether the per-thread scoped-value cache is preserved when a virtual thread is blocked. By default this property is set to `true`, meaning that every virtual thread preserves its scoped-value cache when blocked. Like `ScopedValue.cacheSize`, this is a space versus speed trade-off: in situations where many virtual threads are blocked most of the time, setting this property to `false` might result in a useful memory saving, but each virtual thread's scoped-value cache would have to be regenerated after a blocking operation.

Since:

20

Nested Class Summary

Nested Classes

Modifier and Type	Class	Description
`static final class`	**`ScopedValue.Carrier`**	A mapping of scoped values, as *keys*, to values.

Method Summary

All Methods **Static Methods** **Instance Methods** **Concrete Methods**

Modifier and Type	Method	Description
`T`	**`get`**`()`	Returns the value of the scoped value if bound in the current thread.
`boolean`	**`isBound`**`()`	Returns `true` if this scoped value is bound in the current

		thread.
`static <T> ScopedValue<T>`	`newInstance()`	Creates a scoped value that is initially unbound for all threads.
`T`	`orElse(T other)`	Returns the value of this scoped value if bound in the current thread, otherwise returns `other`.
`<X extends Throwable> T`	`orElseThrow(Supplier<? extends X> exceptionSupplier)`	Returns the value of this scoped value if bound in the current thread, otherwise throws an exception produced by the exception supplying function.
`static <T> ScopedValue.Carrie`	`where(ScopedValue<T> key, T value)`	Creates a new `Carrier` with a single mapping of a `ScopedValue` *key* to a value.
`static <T> void`	`where(ScopedValue<T> key, T value, Runnable op)`	Run an operation with a `ScopedValue` bound to a value in the current thread.
`static <T,R> R`	`where(ScopedValue<T> key, T value, Callable<? extends R> op)`	Calls a value-returning operation with a `ScopedValue` bound to a value in the current thread.

Methods declared in class java.lang.Object

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

Method Details

where

```
public static <T> ScopedValue.Carrier where(ScopedValue<T> key,
                                             T value)
```

Creates a new `Carrier` with a single mapping of a `ScopedValue` *key* to a value. The `Carrier` can be used to accumlate mappings so that an operation can be executed with all scoped values in the mapping bound to values. The following example runs an operation with k1 bound (or rebound) to v1, and k2 bound (or rebound) to v2.

```
    ScopedValue.where(k1, v1).where(k2, v2).run(() -> ... );
```

Type Parameters:

T - the type of the value

Parameters:

key - the ScopedValue key

value - the value, can be null

Returns:

a new Carrier with a single mapping

where

```
public static <T,R> R where(ScopedValue<T> key,
                            T value,
                            Callable<? extends R> op)
                     throws Exception
```

Calls a value-returning operation with a ScopedValue bound to a value in the current thread. When the operation completes (normally or with an exception), the ScopedValue will revert to being unbound, or revert to its previous value when previously bound, in the current thread.

Scoped values are intended to be used in a *structured manner*. If op creates a StructuredTaskScope but does not close it, then exiting op causes the underlying construct of each StructuredTaskScope created in the dynamic scope to be closed. This may require blocking until all child threads have completed their sub-tasks. The closing is done in the reverse order that they were created. Once closed, StructureViolationException is thrown.

Implementation Note:

This method is implemented to be equivalent to:

```
    ScopedValue.where(key, value).call(op);
```

Type Parameters:

T - the type of the value

R - the result type

Parameters:

key - the ScopedValue key

value - the value, can be null

op - the operation to call

Returns:

the result

Throws:

Exception - if the operation completes with an exception

where

```
public static <T> void where(ScopedValue<T> key,
                             T value,
                             Runnable op)
```

Run an operation with a `ScopedValue` bound to a value in the current thread. When the operation completes (normally or with an exception), the `ScopedValue` will revert to being unbound, or revert to its previous value when previously bound, in the current thread.

Scoped values are intended to be used in a *structured manner*. If `op` creates a `StructuredTaskScope` but does not close it, then exiting `op` causes the underlying construct of each `StructuredTaskScope` created in the dynamic scope to be closed. This may require blocking until all child threads have completed their sub-tasks. The closing is done in the reverse order that they were created. Once closed, `StructureViolationException` is thrown.

Implementation Note:

This method is implemented to be equivalent to:

```
    ScopedValue.where(key, value).run(op);
```

Type Parameters:

`T` - the type of the value

Parameters:

`key` - the `ScopedValue` key

`value` - the value, can be `null`

`op` - the operation to call

newInstance

```
public static <T> ScopedValue<T> newInstance()
```

Creates a scoped value that is initially unbound for all threads.

Type Parameters:

`T` - the type of the value

Returns:

a new `ScopedValue`

get

```
public T get()
```

Returns the value of the scoped value if bound in the current thread.

Returns:

the value of the scoped value if bound in the current thread

Throws:

`NoSuchElementException` - if the scoped value is not bound

isBound

```
public boolean isBound()
```

Returns `true` if this scoped value is bound in the current thread.

Returns:

`true` if this scoped value is bound in the current thread

orElse

```
public T orElse(T other)
```

Returns the value of this scoped value if bound in the current thread, otherwise returns `other`.

Parameters:

`other` - the value to return if not bound, can be `null`

Returns:

the value of the scoped value if bound, otherwise `other`

orElseThrow

```
public <X extends Throwable> T orElseThrow
(Supplier<? extends X> exceptionSupplier)
                                   throws X
```

Returns the value of this scoped value if bound in the current thread, otherwise throws an exception produced by the exception supplying function.

Type Parameters:

`X` - the type of the exception that may be thrown

Parameters:

`exceptionSupplier` - the supplying function that produces the exception to throw

Returns:

the value of the scoped value if bound in the current thread

Throws:

`X` - if the scoped value is not bound in the current thread