JEP 446: Scoped Values (Preview)

Author	Andrew Haley & Andrew Dinn
Owner	Andrew Haley
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	core-libs
Discussion	loom dash dev at openjdk dot org
Reviewed by	Alan Bateman, Mark Reinhold
Endorsed by	Brian Goetz
Created	2023/03/16 16:01
Updated	2023/09/06 16:45
Issue	8304357

Summary

Introduce *scoped values*, values that may be safely and efficiently shared to methods without using method parameters. They are preferred to thread-local variables, especially when using large numbers of virtual threads. This is a preview API.

In effect, a scoped value is an *implicit method parameter*. It is "as if" every method in a sequence of calls has an additional, invisible, parameter. None of the methods declare this parameter and only the methods that have access to the scoped value object can access its value (the data). Scoped values make it possible to pass data securely from a caller to a faraway callee through a sequence of intermediate methods that do not declare a parameter for the data and have no access to the data.

History

Scoped Values incubated in JDK 20 via JEP 429. In JDK 21 this feature is no longer incubating; instead, it is a preview API.

Goals

- *Ease of use* — Provide a programming model to share data both within a thread and with child threads, so as to simplify reasoning about data flow.
- *Comprehensibility* — Make the lifetime of shared data visible from the syntactic structure of code.
- *Robustness* — Ensure that data shared by a caller can be retrieved only by legitimate callees.
- *Performance* — Allow shared data to be immutable so as to allow sharing by a large number of threads, and to enable runtime optimizations.

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.

Motivation

Java applications and libraries are structured as collections of classes which contain methods. These methods communicate through method calls.

Most methods allow a caller to pass data to a method by passing them as parameters. When method `A` wants method `B` to do some work for it, it invokes `B` with the appropriate parameters, and `B` may pass some of those parameters to `C`, etc. `B` may have to include in its parameter list not only the things `B` directly needs but also the things `B` has to pass to `C`. For example, if `B` is going to set up and execute a database call, it might want a Connection passed in, even if `B` is not going to use the Connection directly.

Most of the time this "pass what your indirect callees need" approach is the most effective and convenient way to share data. However, sometimes it is impractical to pass all the data that every indirect callee might need in the initial call.

An example

It is a common pattern in large Java programs to transfer control from one component (a "framework") to another ("application code") and then back. For example, a web framework could accept incoming HTTP requests and then call an application handler to handle it. The application handler may then call the framework to read data to the database or to call some other HTTP service.

```
@Override
public void handle(Request request, Response response) { // user code; called by framwork
    ...
    var userInfo = readUserInfo();
    ...
}

private UserInfo readUserInfo() {
```

```
        return (UserInfo)framework.readKey("userInfo", context);// call framework
    }
```

The framework may wish to maintain a `FrameworkContext` object, containing the authenticated user ID, the transaction ID, etc., and associate it with the current transaction. All framework operations use the context object, but it's unused by (and irrelevant to) user code.

In effect, the framework wishes to communicate its internal context from its `serve` method (which calls the user's `handle` method) to its `readKey` method:

```
4. Framework.readKey <--------+ use context
3. Application.readUserInfo   |
2. Application.handle         |
1. Framework.serve  ----------+ create context
```

The simplest way to do this is by passing the object as an argument to all methods in the call chain:

```
@Override
void handle(Request request, Response response, FrameworkContext context) {
    ...
    var userInfo = readUserInfo(context);
    ...
}

private UserInfo readUserInfo(FrameworkContext context) {
    return (UserInfo)framework.readKey("userInfo", context);
}
```

There is no way for the user code to *assist* in the proper handling of the context object. At worst, it could interfere by mixing up contexts; at best it is burdened with the need to add another parameter to all methods that may end up calling back into the framework. If the need to pass a context emerges during re-design of the framework, adding it requires not only the immediate clients -- those user methods that directly call framework methods or those that are directly called by it -- to change their signature, but all intermediate methods as well, even though the context is an internal implementation detail of the framework and user code should not interact with it.

Thread-local variables for sharing

Developers have traditionally used *thread-local variables*, introduced in Java 1.2, to help share data between methods on the call stack without resorting to method arguments. A thread-local variable is a variable of type `ThreadLocal`. Despite looking like an ordinary variable, a thread-local variable has one current value per thread; the particular value that is used depends on which thread calls its `get` or `set` methods to read or write its value. Code in one thread automatically reads and writes its value, while code in another thread automatically reads and writes its own distinct instantiation. Typically, a thread-local variable is declared as a `final static` field so it can easily be reached from different methods, and `private` so that it cannot be directly accessed by client (user) code.

Here is an example of how the two framework methods, both running in the same request-handling thread, can use a thread-local variable to share a `FrameworkContext`.

The framework declares a thread-local variable, `CONTEXT` (1). When`Framework.serve` is executed in a request-handling thread, it writes a suitable `FrameworkContext` to the thread-local variable (2), then calls user code. If and when user code calls `Framework.readKey`, that method reads the thread-local variable (3) to obtain the `FrameworkContext` of the request-handling thread.

```
public class Framework {
    private final Application application;
    public Framework(Application app) { this.application = app; }

    private final static ThreadLocal<FrameworkContext> CONTEXT
                       = new ThreadLocal<>();  // (1)

    void serve(Request request, Response response) {
        var context = createContext(request);
        CONTEXT.set(context);                  // (2)
        Application.handle(request, response);
    }

    public PersistedObject readKey(String key) {
        var context = CONTEXT.get();            // (3)
        var db = getDBConnection(context);
        db.readKey(key);
    }
}
```

Using a thread-local variable avoids the need to pass a `FrameworkContext` as a method argument when the framework calls user code, and when user code calls a framework method back. The thread-local variable serves as a hidden method argument: A thread that calls `CONTEXT.set` in `Framework.serve` and then `CONTEXT.get()` in `Framework.readKey` will automatically see its own local copy of the `CONTEXT` variable. In effect, the `ThreadLocal` field serves as a key that is used to look up a `FrameworkContext` value for the current thread.

While `ThreadLocals` have a distinct instantiation in each thread, the value that is currently set in one thread can be automatically inherited by another thread that the current thread creates if the `InheritableThreadLocal` class is used rather than the `ThreadLocal` class.

Problems with thread-local variables

Unfortunately, thread-local variables have numerous design flaws that are impossible to avoid:

- *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. This is still true even if an object in a thread-local variable is immutable due to every one of its fields being declared final. The `ThreadLocal` API allows this in order to support a fully general model of communication, where data can flow in any direction between methods. This can lead to spaghetti-like data flow, and to programs in which it is hard to discern which method updates shared state and in what order. The more common need, shown in the example above, is a simple one-way transmission of data from one method to others.
- *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. Unfortunately, developers often forget to call `remove()`, so per-thread data is often retained for longer than necessary. In particular, if a thread pool is used, the value of a thread-local variable set in one task could, if not properly cleared, accidentally leak into an unrelated task, potentially leading to dangerous security vulnerabilities. In addition, for programs that rely on the unconstrained mutability of thread-local variables, there may be no clear point at which it is safe for a thread to call `remove()`; this can cause a long-term memory leak, since per-thread data will not be garbage-collected until the thread exits. It would be better if the writing and reading of per-thread data occurred in a bounded period during execution of the thread, avoiding the possibility of leaks.
- *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.) When a developer chooses to create a child thread that inherits thread-local variables, the child thread has to allocate storage for every thread-local variable previously written in the parent thread. This can add significant memory footprint. Child threads cannot share the storage used by the parent thread because the `ThreadLocal` API requires that changing a thread's copy of the thread-local variable is not seen in other threads. This is unfortunate, because in practice child threads rarely call the `set` method on their inherited thread-local variables.

Toward lightweight sharing

The problems of thread-local variables have become more pressing with the availability of virtual threads (JEP 425). Virtual threads 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. This means that a web framework can dedicate a new virtual thread to the task of handling a request and still be able to process thousands or millions of requests at once. In the ongoing example, the methods `Framework.serve`, `Application.handle`, and `Framework.readKey` would all execute in a new virtual thread for each incoming request.

It would be useful for these methods to be able to share data whether they execute in virtual threads or traditional platform threads. Because virtual threads are instances of `Thread`, a virtual thread can have thread-local variables; in fact, the short-lived non-pooled nature of virtual threads makes the problem of long-term memory leaks, mentioned above, less acute. (Calling a thread-local variable's `remove()` method is unnecessary when a thread terminates quickly, since termination automatically removes its thread-local variables.) However, if each of a million virtual threads has its own copy of thread-local variables, the memory footprint may be significant.

In summary, thread-local variables have more complexity than is usually needed for sharing data, and significant costs that cannot be avoided. The Java Platform should provide a way to maintain inheritable per-thread data for thousands or millions of virtual threads. If these per-thread variables were immutable, their data could be shared by child threads efficiently. Further, the lifetime of these per-thread variables should be bounded: Any data shared via a per-thread variable should become unusable once the method that initially shared the data is finished.

Description

A *scoped value* allows data to be safely and efficiently shared between methods in a large program without resorting to method arguments. It is a variable of type `ScopedValue`. Typically, it is declared as a `final static` field so it can easily be reached from many methods.

Like a thread-local variable, a scoped value 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 scoped value is written once, and is available only for a bounded period during execution of the thread.

A scoped value is used as shown below. Some code calls `ScopedValue.where`, presenting a scoped value and the object to which it is to be bound. The call to `run` *binds* the scoped value, providing a copy that is specific to the current thread, and then executes the lambda expression passed as argument. During the lifetime of the `run` call, the lambda expression, or any method called directly or indirectly from that expression, can read the scoped value via the value's `get()` method. After the `run` method finishes, the binding is destroyed.

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

// In some method
ScopedValue.where(V, <value>)
           .run(() -> { ... V.get() ... call methods ... });

// In a method called directly or indirectly from the lambda expression
... V.get() ...
```

The structure of the code delineates the period of time when a thread can read its copy of a scoped value. This bounded lifetime greatly simplifies reasoning about thread behavior. The one-way transmission of data from caller to callees — both direct and indirect — is obvious at a glance. There is no `set` method that lets faraway code change the scoped value at any time. This also helps performance: Reading a scoped value with `get()` is often as fast as reading a local variable, regardless of the stack distance between caller and callee.

The meaning of "scoped"

The *scope* of a thing is the space in which it lives — the extent or range in which it can be used. For example, in the Java programming language, the scope of a variable declaration is the space within the program text where it is legal to refer to the variable with a simple name (JLS 6.3). This kind of scope is more accurately called *lexical scope* or *static scope*, since the space where the variable is in scope can be understood statically by looking for `{` and `}` characters in the program text.

Another kind of scope is called *dynamic scope*. The dynamic scope of a thing refers to the parts of a program that can use the thing as the program executes. If method `a` calls method `b` that, in turn, calls method `c`, the execution lifetime of `c` is contained within the execution of `b`, which is contained in that of `a`, even though the three methods are distinct code units:

```
  |
  |   +–– a
  |   |
  |   |  +–– b
  |   |  |
TIME  |  |  +–– c
  |   |  |  |
  |   |  |  |__
  |   |  |
  |   |  |__
  |   |
  |   |__
  |
  v
```

This is the concept to which *scoped value* appeals, because binding a scoped value `V` in a `run` method produces a value that is accessible by certain parts of the program as it executes, namely the methods invoked directly or indirectly by `run`.

The unfolding execution of those methods defines a dynamic scope; the binding is in scope during the execution of those methods, and nowhere else.

Web framework example with scoped values

The framework code shown earlier can easily be rewritten to use a scoped value instead of a thread-local variable. At (1), the framework declares a scoped value instead of a thread-local variable. At (2), the serve method calls `ScopedValue.where` and `run` instead of a thread-local variable's `set` method.

```
class Frameowrk {
    private final static ScopedValue<FrameworkContext> CONTEXT
                        = ScopedValue.newInstance();   // (1)

    void serve(Request request, Response response) {
        var context = createContext(request);
        ScopedValue.where(CONTEXT, context)            // (2)
                   .run(() -> Application.handle(request, response));
    }

    public PersistedObject readKey(String key) {
        var context = CONTEXT.get();            // (3)
        var db = getDBConnection(context);
        db.readKey(key);
    }

    ...
}
```

Together, `where` and `run` provide one-way sharing of data from the `serve` method to the `readKey` method. The scoped value passed to `where` is bound to the corresponding object for the lifetime of the `run` call, so `CONTEXT.get()` in any method called from `run` will read that value. Accordingly, when `Framework.serve` calls user code, and user code calls `Framework.readKey`, the value read from the scoped value (3) is the value written by `Framework.serve` earlier in the thread.

The binding established by `run` is usable only in code called from `run`. If `CONTEXT.get()` appeared in `Framework.serve` after the call to `run`, an exception would be thrown because `CONTEXT` is no longer bound in the thread.

As before, the framework relies on Java's access control to restrict access to its internal data: The `CONTEXT` field has private access, which allows the framework to share information internally between its two methods. That information is inaccessible to, and hidden from user code. We say that the `ScopedValue` object is a *capability* object that gives code with permissions to access it the ability to bind or read the value. Often the `ScopedValue` will have `private` access, but sometimes it may have `protected` or package access to allow multiple cooperating classes to read and bind the value.

Rebinding scoped values

That scoped values have no `set()` method means that a caller can use a scoped value to reliably communicate a constant value to its callees in the same thread. However, there are occasions when one of the callees might need to use the same scoped value to communicate a different value to its own callees. The `ScopedValue` API allows a new nested binding to be established for subsequent calls:

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

void foo() {
    ScopedValue.where(X, "hello").run(() -> bar());
}

void bar() {
    System.out.println(X.get()); // prints hello
    ScopedValue.where(X, "goodbye").run(() -> baz());
    System.out.println(X.get()); // prints hello
}

void baz() {
    System.out.println(X.get()); // prints goodbye
}
```

`bar` reads the value of `X` to be `"hello"`, as that is the binding in the scope established in `foo`. But then `bar` establishes a nested scope to run `baz` where `X` is bound to `goodbye`.

Notice how the `"goodbye"` binding is in effect only inside the nested scope. Once `baz` returns, `bar` sees the `"hello"` binding. The body of `bar` cannot change the binding seen by that method itself but can change the binding seen by its callees. This guarantees a bounded lifetime for sharing of the new value.

Inheriting scoped values

The web framework example dedicates a thread to handling each request, so the same thread executes some framework code, then user code from the application developer, then more framework code to access the database. However, user code can exploit the lightweight nature of virtual threads by creating its own virtual threads and running its own code in them. These virtual threads will be child threads of the request-handling thread.

Context data shared by a code running in the request-handling thread needs to be available to code running in child threads. Otherwise, when user code running in a child thread calls a framework method it will be unable to access the `FrameworkContext` created by the framework code running in the request-handling thread. To enable cross-thread sharing, scoped values can be inherited by child threads.

The preferred mechanism for user code to create virtual threads is the Structured Concurrency API (JEP 428), specifically the class `StructuredTaskScope`. Scoped values in the parent thread are automatically inherited by child threads created with `StructuredTaskScope`. Code in a child thread can use bindings established for a scoped value in the parent thread with minimal overhead. Unlike with thread-local variables, there is no copying of a parent thread's scoped value bindings to the child thread.

Here is an example of scoped value inheritance occurring behind the scenes in user code. The `Server.serve` method binds `CONTEXT` and calls `Application.handle` just as before. However, the user code in `Application.handle` calls run the `readUserInfo()` and `fetchOffers()` methods concurrently, each in its own virtual threads, using `StructuredTaskScope.fork` (1, 2). Each method may use `Framework.readKey` which, as before, consults the scoped value `CONTEXT` (4). Further details of the user code are not discussed here; see JEP 428 for further information.

```
@Override
public Response handle(Request request, Response response) {
      try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
          Supplier<UserInfo>    user   = scope.fork(() -> readUserInfo()); // (1)
```

```
            Supplier<List<Offer>> offers = scope.fork(() -> fetchOffers());   // (2)
            scope.join().throwIfFailed();  // Wait for both forks
            return new Response(user.get(), order.get());
        } catch (Exception ex) {
            reportError(response, ex);
        }
    }
```

`StructuredTaskScope.fork` ensures that the binding of the scoped value `CONTEXT` made in the request-handling thread — in Framework.serve(...) — is read by `CONTEXT.get()` in the child thread. The following diagram shows how the dynamic scope of the binding is extended to all methods executed in the child thread:

```
Thread 1                        Thread 2
--------                        --------
                                5. Framework.readKey <----------+
                                                                |
                                                             CONTEXT
                                4. Application.readUserInfo     |
3. StructuredTaskScope.fork                                     |
2. Application.handle                                           |
1. Server.serve     --------------------------------------------+
```

The fork/join model offered by `StructuredTaskScope` means that the dynamic scope of the binding is still bounded by the lifetime of the call to `ScopedValue.where(...).run(...)`. The `Principal` will remain in scope while the child thread is running, and `scope.join()` ensures that child threads terminate before run can return, destroying the binding. This avoids the problem of unbounded lifetimes seen when using thread-local variables. Legacy thread management classes such as `ForkJoinPool` do not support inheritance of ScopedValues because they cannot guarantee that a child thread forked from some parent thread scope will exit before the parent leaves that scope.

Migrating to scoped values

Scoped values are likely to be useful and preferable in many scenarios where thread-local variables are used today. Beyond serving as hidden method arguments, scoped values may assist with:

- *Re-entrant code* — Sometimes it is desirable to detect recursion, perhaps because a framework is not re-entrant or because recursion must be limited in some way. A scoped value provides a way to do this: Set it up as usual, with `ScopedValue.where` and `run`, and then deep in the call stack, call `ScopedValue.isBound()` to check if it has a binding for the current thread. More elaborately, the scoped value can model a recursion counter by being repeatedly rebound.
- *Nested transactions* — Detecting recursion can also be useful in the case of flattened transactions: Any transaction started while a transaction is in progress becomes part of the outermost transaction.
- *Graphics contexts* — Another example occurs in graphics, where there is often a drawing context to be shared between parts of the program. Scoped values, because of their automatic cleanup and re-entrancy, are better suited to this than thread-local variables.

In general, we advise migration to scoped values when the purpose of a thread-local variable aligns with the goal of a scoped value: one-way transmission of unchanging data. If a codebase uses thread-local variables in a two-way fashion — where a callee deep in the call stack transmits data to a faraway caller via `ThreadLocal.set` — or in a completely unstructured fashion, then migration is not an option.

There are a few scenarios that favor thread-local variables. An example is caching objects that are expensive to create and use, such as instances of `java.text.DateFormat`. Notoriously, a `DateFormat` object is mutable, so it cannot be shared between threads without synchronization. Giving each thread its own `DateFormat` object, via a thread-local variable that persists for the lifetime of the thread, is often a practical approach.

Alternatives

It is possible to emulate many of the features of scoped values with thread-local variables, albeit at some cost in memory footprint, security, and performance.

We experimented with a modified version of `ThreadLocal` that supports some of the characteristics of scoped values. However, carrying the additional baggage of thread-local variables results in an implementation that is unduly burdensome, or an API that returns `UnsupportedOperationException` for much of its core functionality, or both. It is better, therefore, not to modify `ThreadLocal` but to introduce scoped values as an entirely separate concept.

Scoped values were inspired by the way that many Lisp dialects provide support for dynamically scoped free variables; in particular, how such variables behave in a deep-bound, multi-threaded runtime such as Interlisp-D. scoped values improve on Lisp's free variables by adding type safety, immutability, encapsulation, and efficient access within and across threads.