JEP 429: Scoped Values (Incubator)

Authors	Andrew Haley, Andrew Dinn
Owner	Andrew Haley
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	20
Component	core-libs
Discussion	loom dash dev at openjdk dot java dot net
Relates to	JEP 446: Scoped Values (Preview)
Reviewed by	Alan Bateman, Alex Buckley
Endorsed by	John Rose
Created	2021/03/04 11:03
Updated	2023/11/29 14:40
Issue	8263012

Summary

Introduce *scoped values*, which enable the sharing of immutable data within and across threads. They are preferred to thread-local variables, especially when using large numbers of virtual threads. This is an incubating 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* — Treat shared data as 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

Large Java programs typically consist of distinct and complementary components that need to share data between themselves. For example, a web framework might include a server component, implemented in the thread-per-request style, and a data access component, which handles persistence. Throughout the framework, user authentication and authorization relies on a `Principal` object shared between components. The server component creates a `Principal` for each thread that handles a request, and the data access component refers to a thread's `Principal` to control access to the database.

The diagram below shows the framework handling two requests, each in its own thread. Request handling flows upward, from the server component (`Server.serve(...)`) to user code (`Application.handle(...)`) to the data access component (`DBAccess.open()`). The data access component determines whether the thread is permitted to access the database, as follows:

- In Thread 1, the `ADMIN` principal created by the server component allows database access. The dashed line indicates the principal is to be shared with the data access component, which inspects it and proceeds to call `DBAccess.newConnection()`.
- In Thread 2, the `GUEST` principal created by the server component does not allow database access. The data access component inspects the principal, determines that the user code must not proceed, and throws an `InvalidPrincipalException`.

```
Thread 1                                 Thread 2
--------                                 --------
8. DBAccess.newConnection()              8. throw new InvalidPrincipalException()
7. DBAccess.open() <----------+          7. DBAccess.open() <----------+
   ...                        |             ...                        |
   ...                  Principal(ADMIN)    ...                  Principal(GUEST)
2. Application.handle(..)     |          2. Application.handle(..)     |
1. Server.serve(..) ----------+          1. Server.serve(..) ----------+
```

Normally, data is shared between caller and callee by passing it as method arguments, but this is not viable for a `Principal` shared between the server component and the data access component because the server component calls untrusted user code first. We need a better way to share data from the server component to the data access component than wiring it into a cascade of untrusted method invocations.

Thread-local variables for sharing

Developers have traditionally used *thread-local variables*, introduced in Java 1.2, to help components share data 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 multiple incarnations, one per thread; the particular incarnation 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 incarnation, while code in another thread automatically reads and writes its own distinct incarnation. Typically, a thread-local variable is declared as a `final static` field so it can easily be reached from many components.

Here is an example of how the server component and the data access component, both running in the same request-handling thread, can use a thread-local variable to share a `Principal`. The server component first declares a thread-local variable, `PRINCIPAL` (1). When `Server.serve(...)` is executed in a request-handling thread, it writes a suitable `Principal` to the thread-local variable (2), then calls user code. If and when user code calls `DBAccess.open()`, the data access component reads the thread-local variable (3) to obtain the `Principal` of the request-handling thread. Only if the `Principal` indicates suitable permissions is database access permitted (4).

```
class Server {
    final static ThreadLocal<Principal> PRINCIPAL = new ThreadLocal<>();  // (1)

    void serve(Request request, Response response) {
        var level     = (request.isAuthorized() ? ADMIN : GUEST);
        var principal = new Principal(level);
        PRINCIPAL.set(principal);                                         // (2)
        Application.handle(request, response);
    }
}

class DBAccess {
    DBConnection open() {
        var principal = Server.PRINCIPAL.get();                           // (3)
        if (!principal.canOpen()) throw new InvalidPrincipalException();
        return newConnection(...);                                        // (4)
    }
}
```

Using a thread-local variable avoids the need to pass a `Principal` as a method argument when the server component calls user code, and when user code calls the data access component. The thread-local variable serves as a kind of hidden method argument: A thread which calls `PRINCIPAL.set(...)` in `Server.serve(...)` and then `PRINCIPAL.get()` in `DBAccess.open()` will automatically see its own incarnation of the `PRINCIPAL` variable. In effect, the `ThreadLocal` field serves as a key that is used to look up a `Principal` value for the current thread.

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. The `ThreadLocal` API allows this in order to support a fully general model of communication, where data can flow in any direction between components. However, this can lead to spaghetti-like data flow, and to programs in which it is hard to discern which component 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 component to others.
- *Unbounded lifetime* — Once a thread's incarnation of a thread-local variable is written via the `set(...)` method, the incarnation 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 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 thread-local variables are mutable, and the `ThreadLocal` API requires that mutation in one thread 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 `Server.serve(...)`, `Application.handle(...)`, and `DBAccess.open()` would all execute in a new virtual thread for each incoming request.

It would obviously 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 mutable 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 immutable and inheritable per-thread data for thousands or millions of virtual threads. Because these per-thread variables would be 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 components 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 components.

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. Unlike a thread-local variable, a scoped value is written once and is then immutable, 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 an incarnation 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 syntactic structure of the code delineates the period of time when a thread can read its incarnation of a scoped value. This bounded lifetime, combined with immutability, 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. Immutability 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. This is the concept to which *scoped value* appeals, because binding a scoped value V in a `run(...)` method produces an incarnation of V that is usable 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 incarnation 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 server component declares a scoped value instead of a thread-local variable. At (2), the server component calls `ScopedValue.where(...)` and `run(...)` instead of a thread-local variable's `set(...)` method.

```
class Server {
    final static ScopedValue<Principal> PRINCIPAL =  ScopedValue.newInstance(); // (1)

    void serve(Request request, Response response) {
        var level     = (request.isAdmin() ? ADMIN : GUEST);
        var principal = new Principal(level);
        ScopedValue.where(PRINCIPAL, principal)                            // (2)
                   .run(() -> Application.handle(request, response));
    }
}

class DBAccess {
    DBConnection open() {
        var principal = Server.PRINCIPAL.get();                            // (3)
        if (!principal.canOpen()) throw new  InvalidPrincipalException();
        return newConnection(...);
    }
}
```

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

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

Rebinding scoped values

The immutability of scoped values 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 in the thread. The `ScopedValue` API allows a new binding to be established for nested calls.

As an example, consider a third component of the web framework: a logging component with a method `void log(Supplier<String> formatter)`. User code passes a lambda expression to the `log(...)` method; if logging is enabled, the method calls `formatter.get()` to evaluate the lambda expression and then prints the result. Although the user code may have permission to access the database, the lambda expression should not, since it only needs to format text. Accordingly, the scoped value that was initially bound in `Server.serve(...)` should be rebound to a guest `Principal` for the lifetime of `formatter.get()`:

```
8. InvalidPrincipalException()
7. DBAccess.open() <--------------------------+  X---------+
   ...                                        |            |
   ...                                  Principal(GUEST)   |
4. Supplier.get()                             |            |
3. Logger.log(() -> { DBAccess.open(); }) ----+      Principal(ADMIN)
2. Application.handle(..)                                  |
1. Server.serve(..) ---------------------------------------+
```

Here is the code for `log(...)` with rebinding. It obtains a guest `Principal` (1) and passes it as the new binding for the scoped value `PRINCIPAL` (2). For the lifetime of the invocation of `call` (3), `PRINCIPAL.get()` will read this new value. Thus, if the user code passes a malicious lambda expression to `log(...)` that performs `DBAccess.open()`, the check in `DBAccess.open()` will read the guest `Principal` from `PRINCIPAL` and throw an `InvalidPrincipalException`.

```
class Logger {
    void log(Supplier<String> formatter) {
        if (loggingEnabled) {
            var guest = Principal.createGuest();                         // (1)
            var message = ScopedValue.where(Server.PRINCIPAL, guest)  // (2)
                                     .call(() -> formatter.get());    // (3)
            write(logFile, "%s %s".format(timeStamp(), message));
        }
    }
}
```

(We here use `call(...)` instead of `run(...)` to invoke the formatter because the result of the lambda expression is needed.) The syntactic structure of `where(...)` and `call(...)` means that the rebinding is only visible in the nested dynamic scope introduced by `call(...)`. The body of `log(...)` cannot change the binding seen by that method itself but can change the binding seen by its callees, such as the `formatter.get(...)` method. 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 can execute framework code from the server component, then user code from the application developer, then more framework code from the data access component. 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.

Data shared by a component running in the request-handling thread needs to be available to components running in child threads. Otherwise, when user code running in a child thread calls the data access component, that component — now also running in the child thread — will be unable to check the `Principal` shared by the server component 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, in a variant of the `Application.handle(...)` method called from `Server.serve(...)`. The user code calls `StructuredTaskScope.fork(...)` (1, 2) to run the `findUser()` and `fetchOrder()` methods concurrently, in their own virtual threads. Each method calls the data access component (3), which as before consults the scoped value `PRINCIPAL` (4). Further details of the user code are not discussed here; see JEP 428 for further information.

```
class Application {
    Response handle() throws ExecutionException, InterruptedException {
        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
            Future<String>  user  = scope.fork(() -> findUser());          // (1)
            Future<Integer> order = scope.fork(() -> fetchOrder());        // (2)
            scope.join().throwIfFailed();  // Wait for both forks
            return new Response(user.resultNow(), order.resultNow());
        }
    }

    String findUser() {
        ... DBAccess.open() ...                                            // (3)
    }
}

class DBAccess {
    DBConnection open() {
        var principal = Server.PRINCIPAL.get();                            // (4)
        if (!principal.canOpen()) throw new  InvalidPrincipalException();
        return newConnection(...);
    }
}
```

`StructuredTaskScope.fork(...)` ensures that the binding of the scoped value `PRINCIPAL` made in the request-handling thread — when Server.serve(...) called ScopedValue.where(...) — is automatically visible to `PRINCIPAL.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
--------                           --------
                                   8. DBAccess.newConnection()
                                   7. DBAccess.open() <----------+
                                   ...                           |
                                   ...                     Principal(ADMIN)
                                   4. Application.findUser()     |
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.

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.