Scoped Values

A simple but powerful idea

Andrew Dinn on behalf of Andrew Haley
Distinguished Engineer(s)
Java Platform Team

Red Hat

Original motivation, in brief

- Loom's Virtual Threads change the rules: threads are no longer scarce, but are limited only by memory
- When programs needs some sort of "current context", the standard Java way is to use a `ThreadLocal` variable
- But `ThreadLocal` doesn't scale at all well when we have millions of virtual threads, because every thread has its own set of the `ThreadLocal` objects it needs
- What if, instead of copying, we could share?
- It turned out that scoped values aren't just useful for Virtual Threads

Wider (prior) motivation, in brief

- **TL;DR** ThreadLocal does not promote well-structured programming
- No clean model for communicating data via a ThreadLocal
  - `get()` and `set()` can be called at any point during a Thread's lifetime
  - – potential for spaghetti dataflow up, down and across method invocation hierarchies
- No bounded lifetime for data managed via a ThreadLocal
  - A `get()` always returns some value (even if it might be null)
  - A `set()` value persists until the next set() or the end of the Thread lifetime
- Most programs would benefit from a simpler, more constrained solution
  - Which, it turns out, can also avoid the performance issues

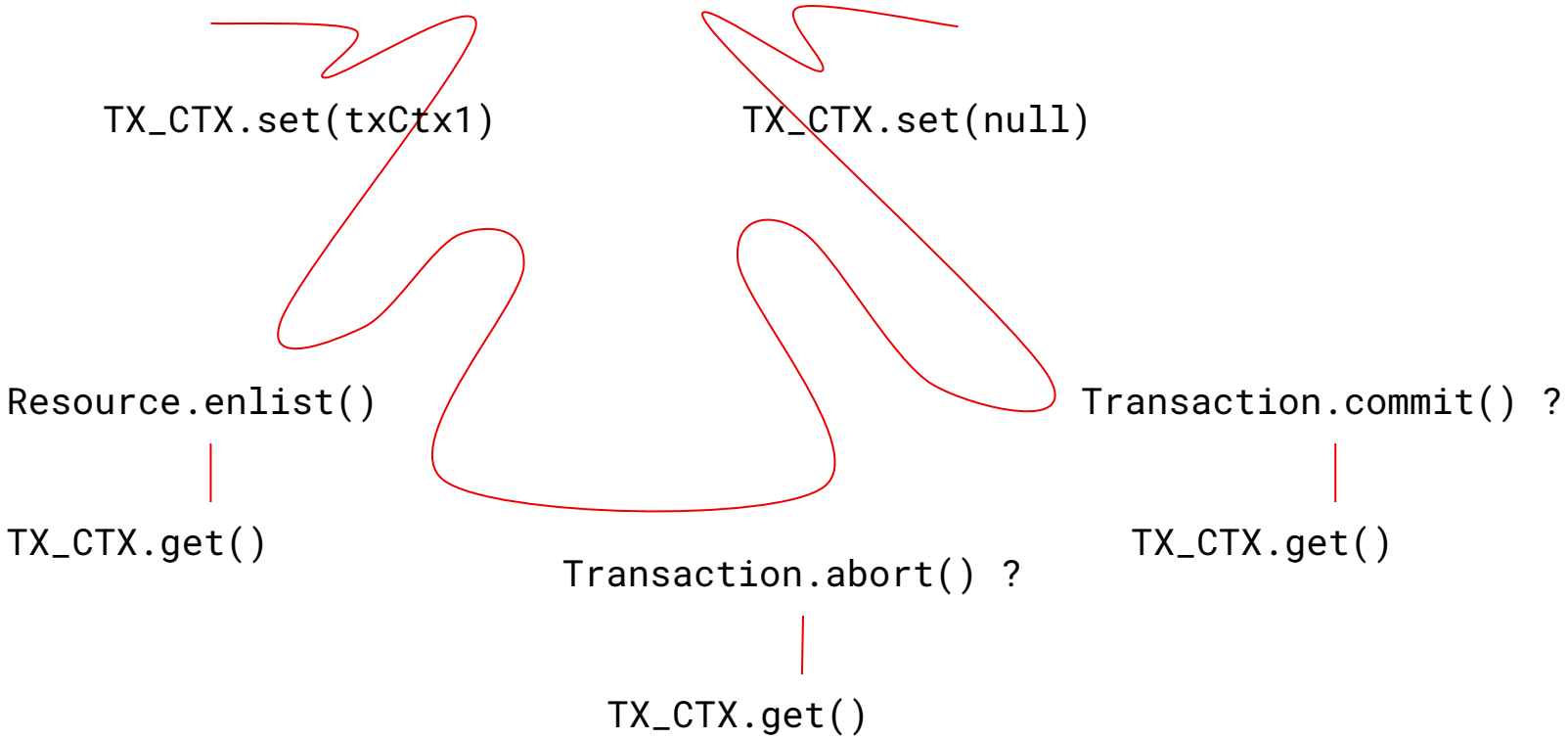
Wider (prior) motivation, in brief

- **TL;DR** ThreadLocal does not promote well-structured programming
- No clean model for communicating data via a ThreadLocal
  - `get()` and `set()` can be called at any point during a Thread's lifetime
  - – potential for spaghetti dataflow up, down and across method invocation hierarchies
- No bounded lifetime for data managed via a ThreadLocal
  - A `get()` always returns some value (even if it might be null)
  - A `set()` value persists until the next set() or the end of the Thread lifetime
- Most programs would benefit from a simpler, more constrained solution
  - Which, it turns out, can also avoid the performance issues

Let's see that in pictures . . .

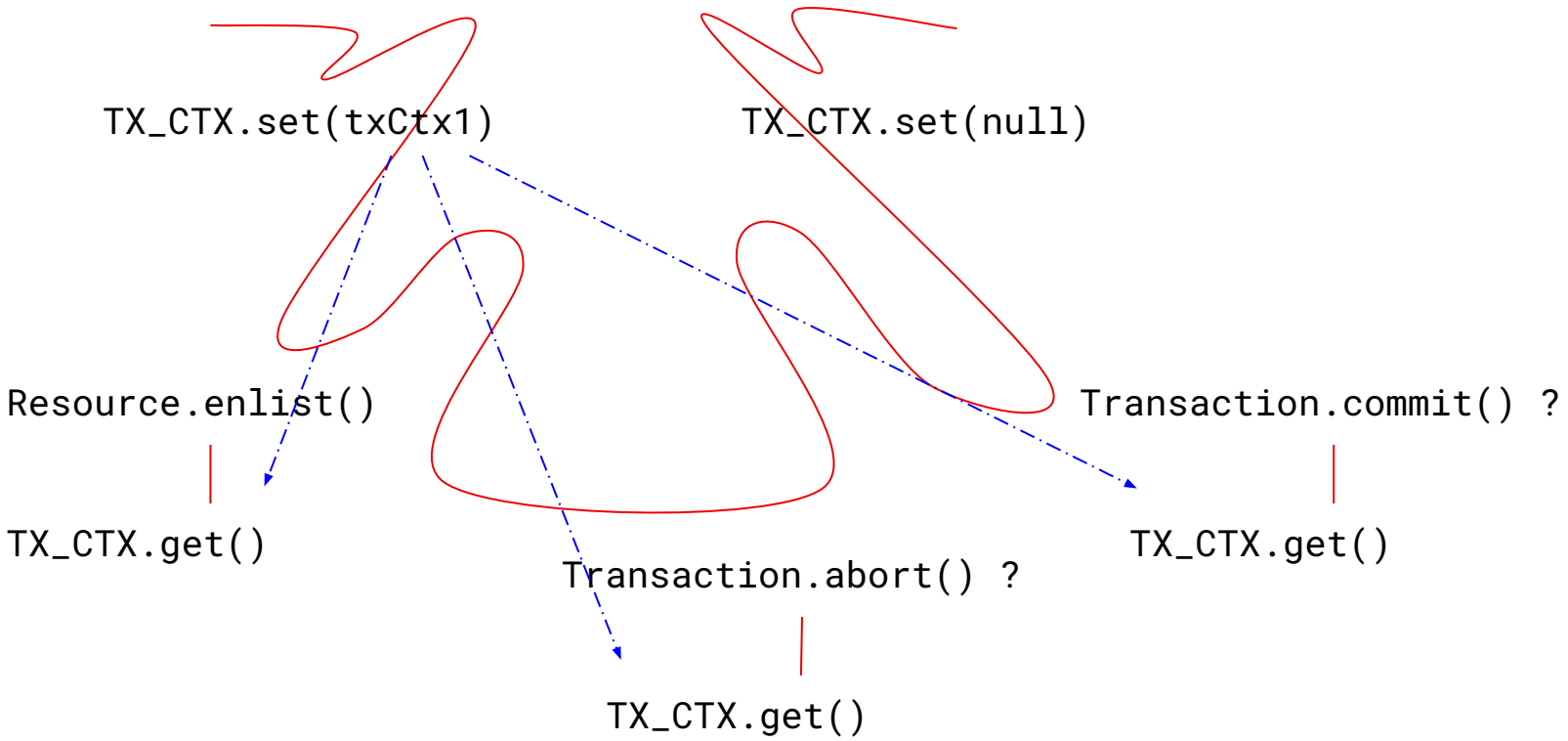
Clean `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



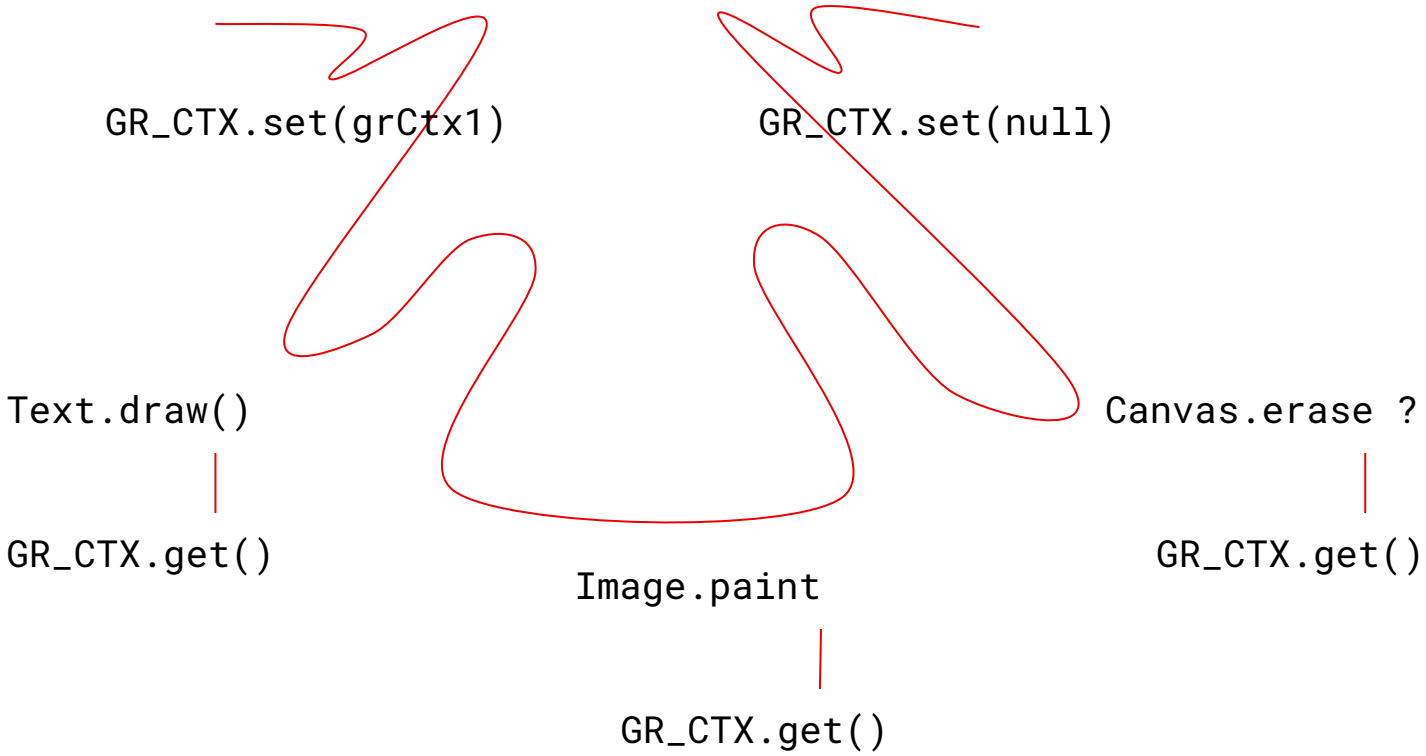
Clean `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



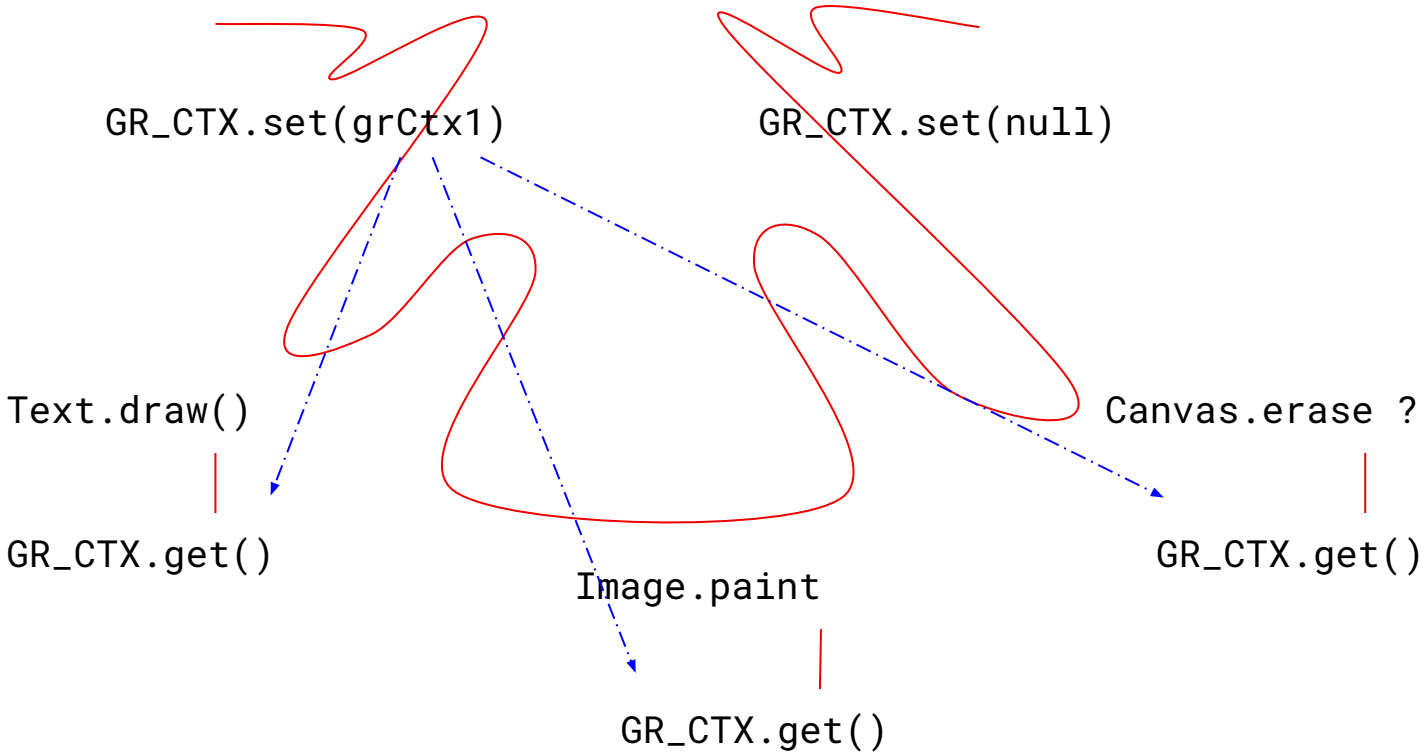
Clean `ThreadLocal` use (2)

```
final static ThreadLocal<GraphicContext> GR_CTX = . . .;
```



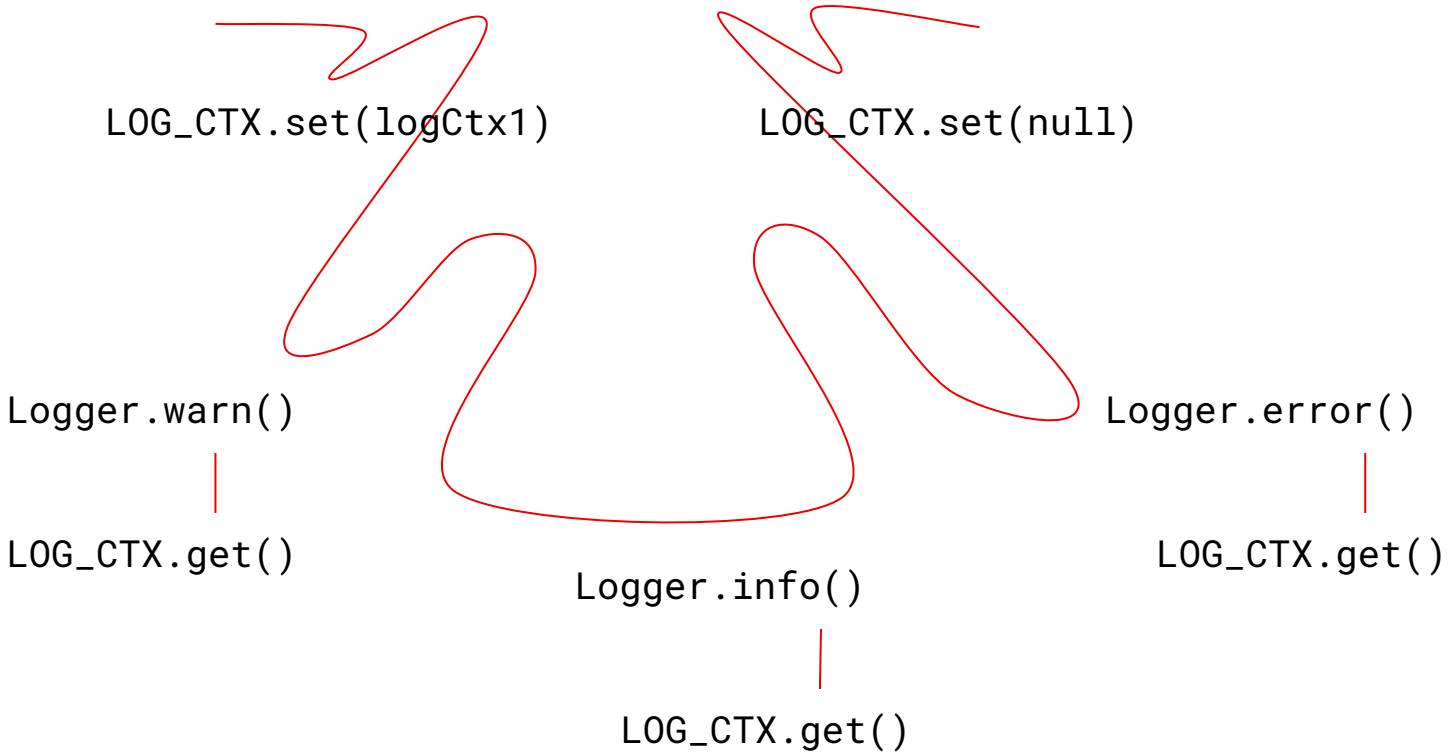
Clean `ThreadLocal` use (2)

```
final static ThreadLocal<GraphicContext> GR_CTX = . . .;
```



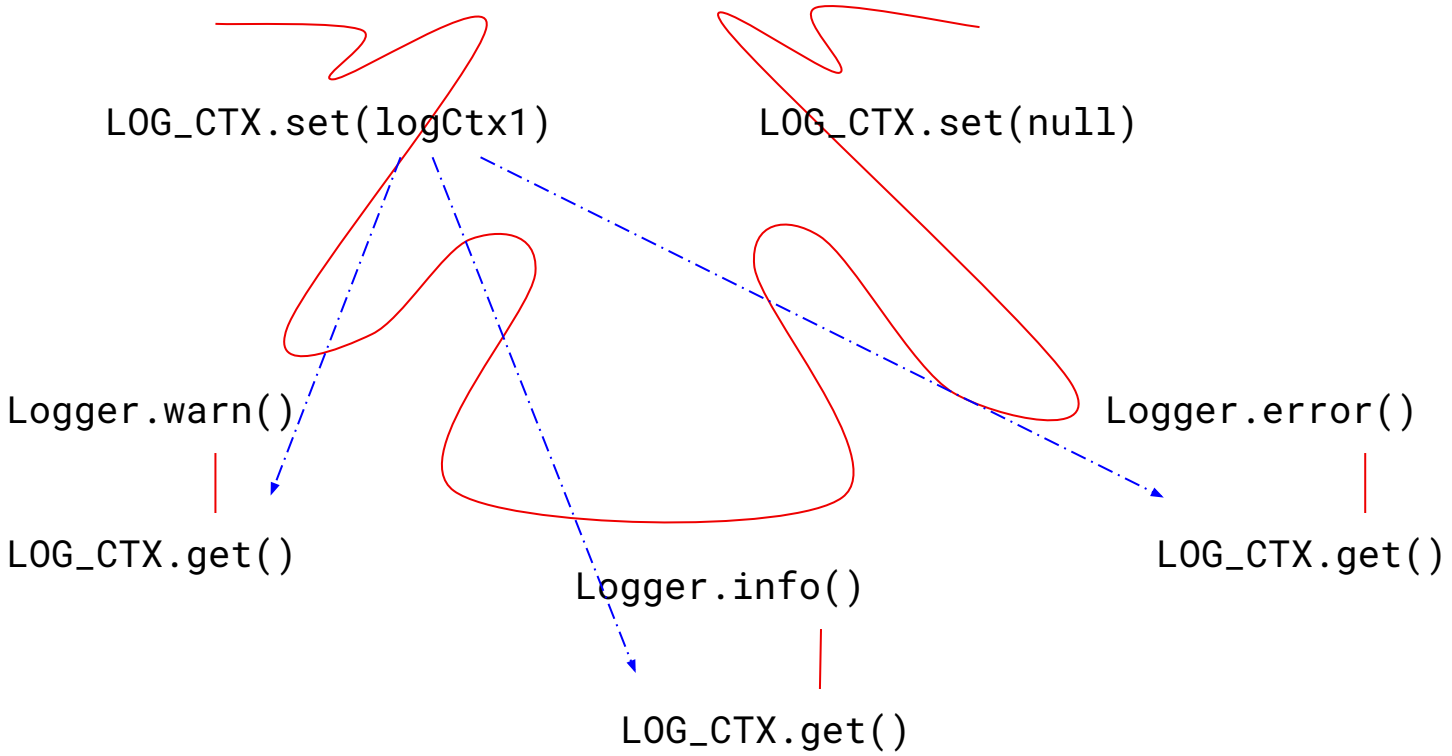
Clean `ThreadLocal` use (3)

```
final static ThreadLocal<LogContext> LOG_CTX = . . .;
```



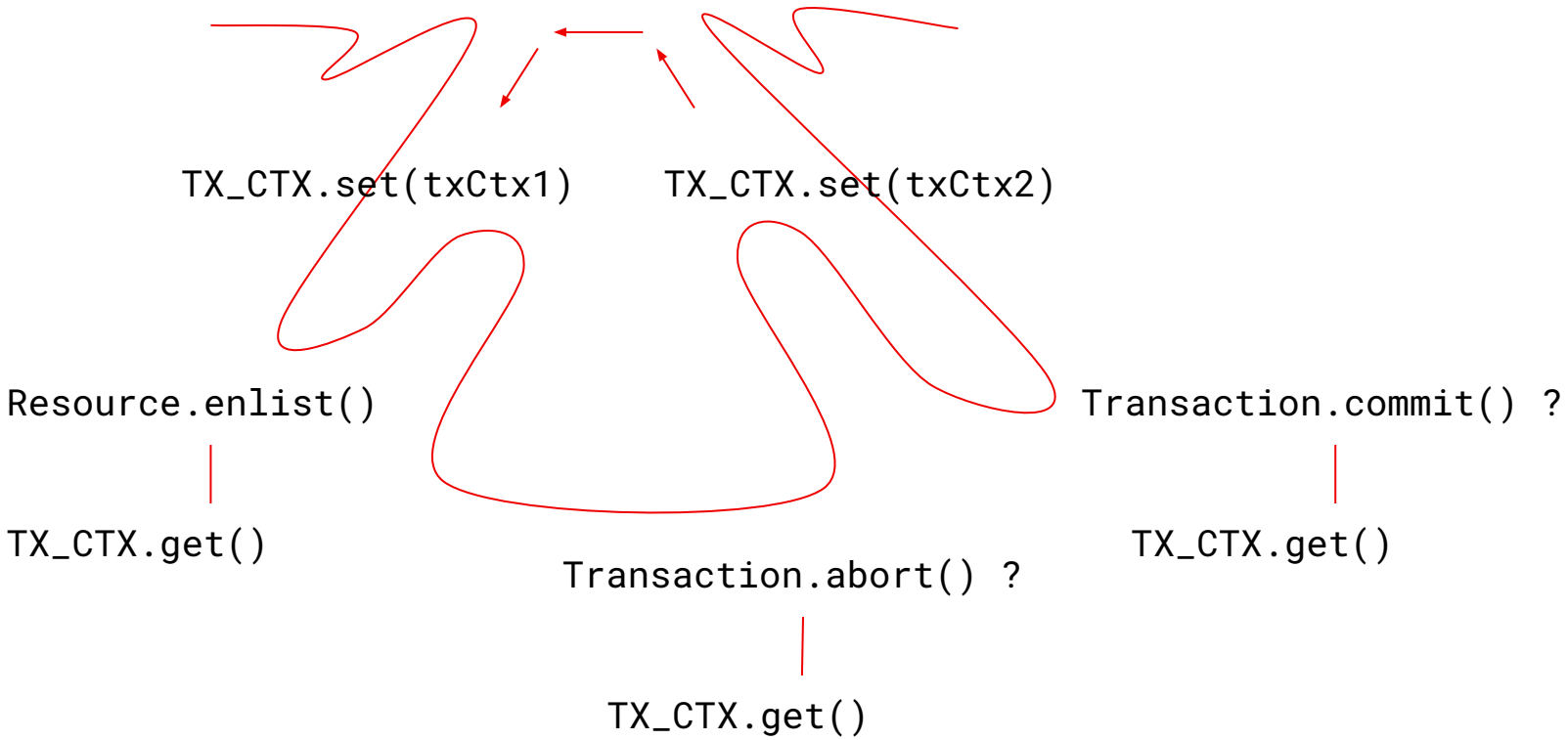
Clean `ThreadLocal` use (3)

```
final static ThreadLocal<LogContext> LOG_CTX = . . .;
```



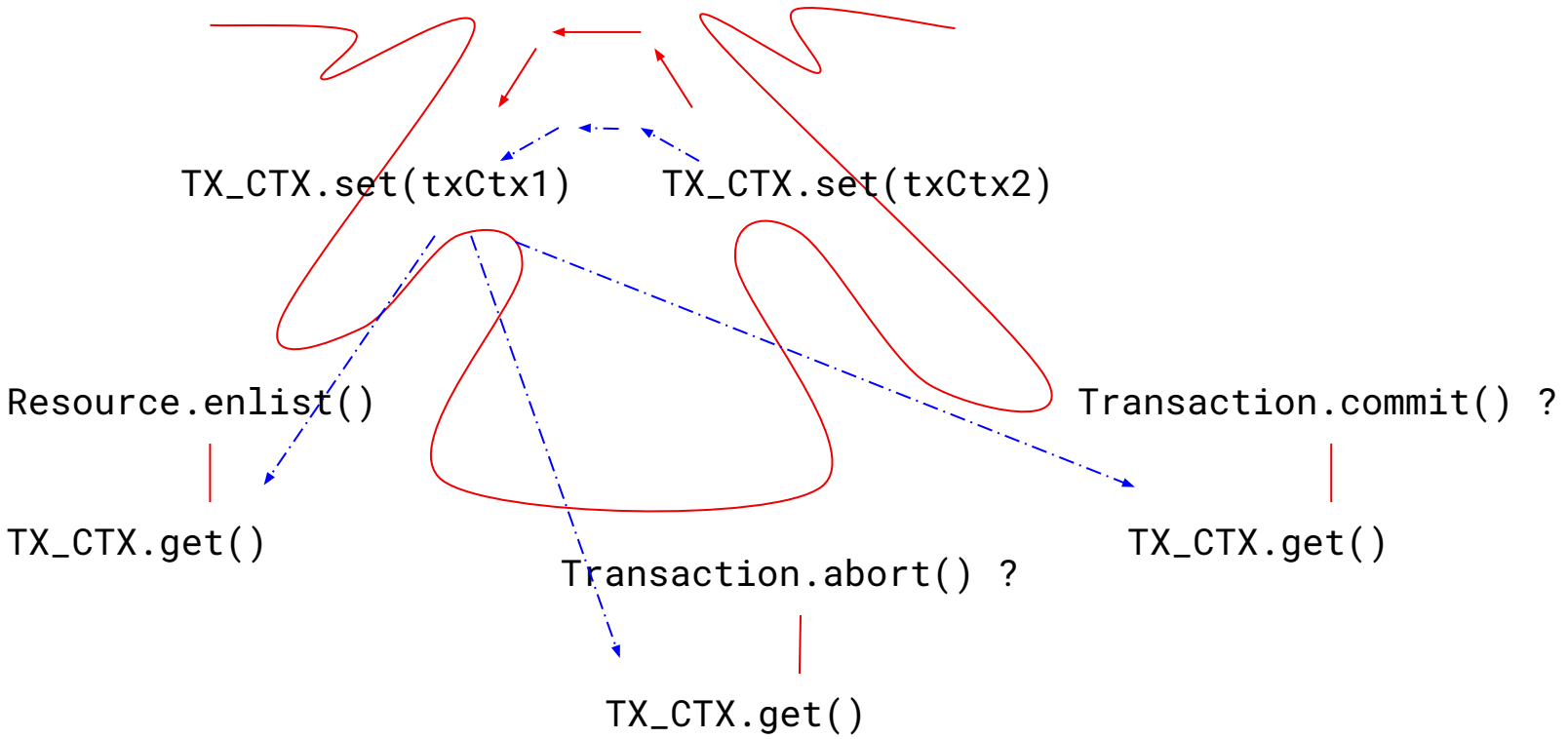
Common `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



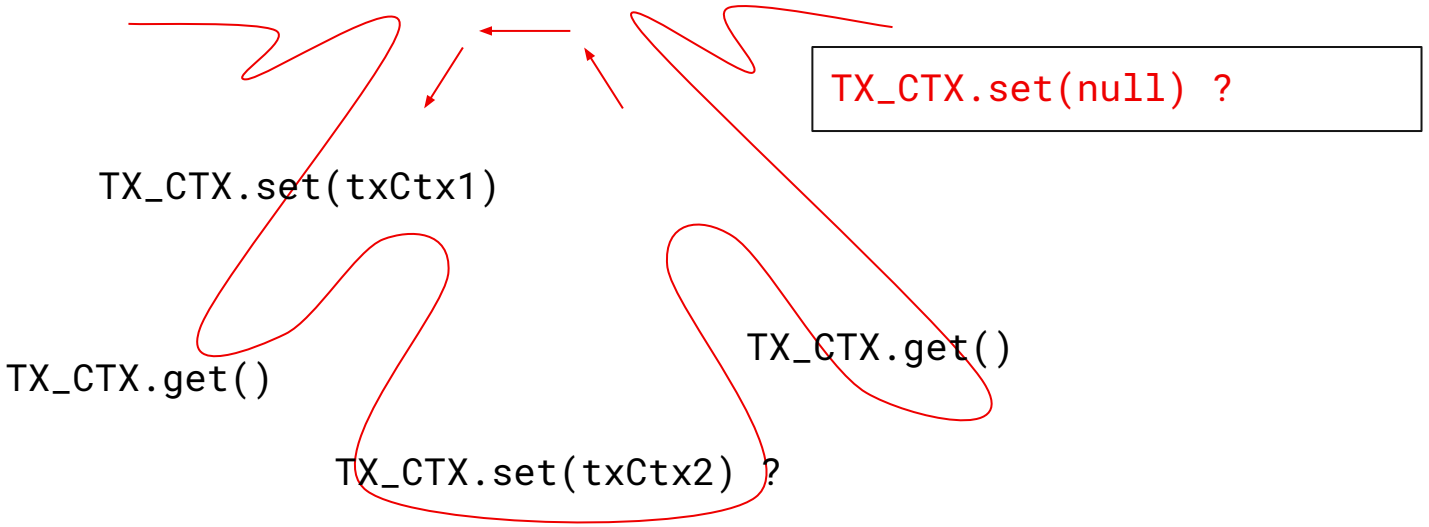
Common `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



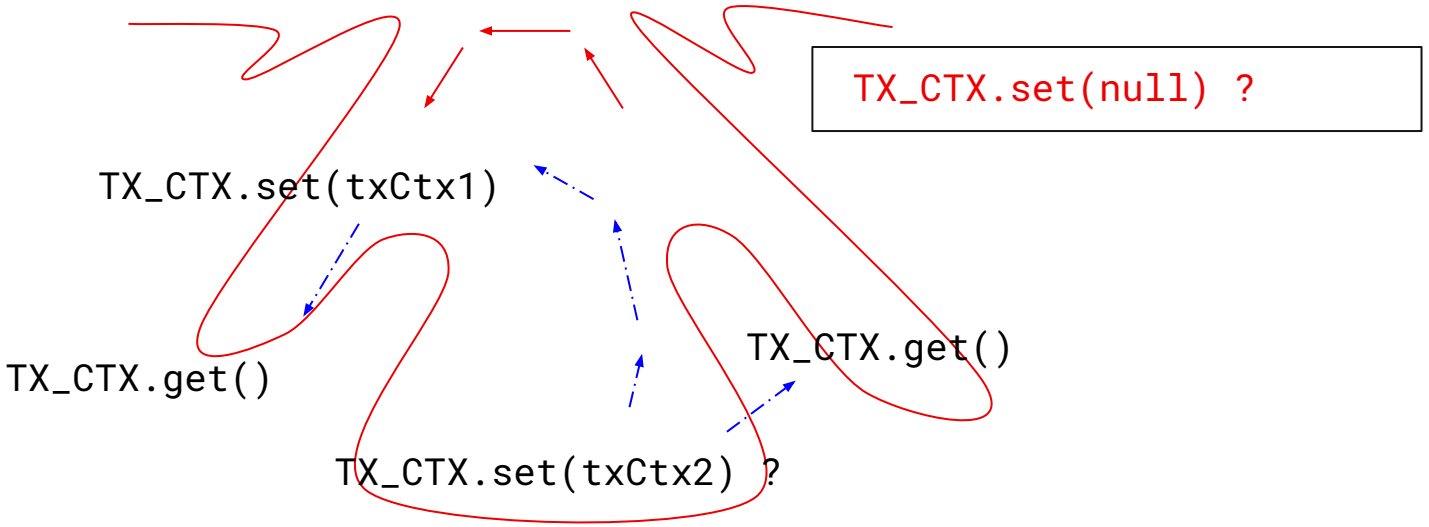
Worst(?) Case `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



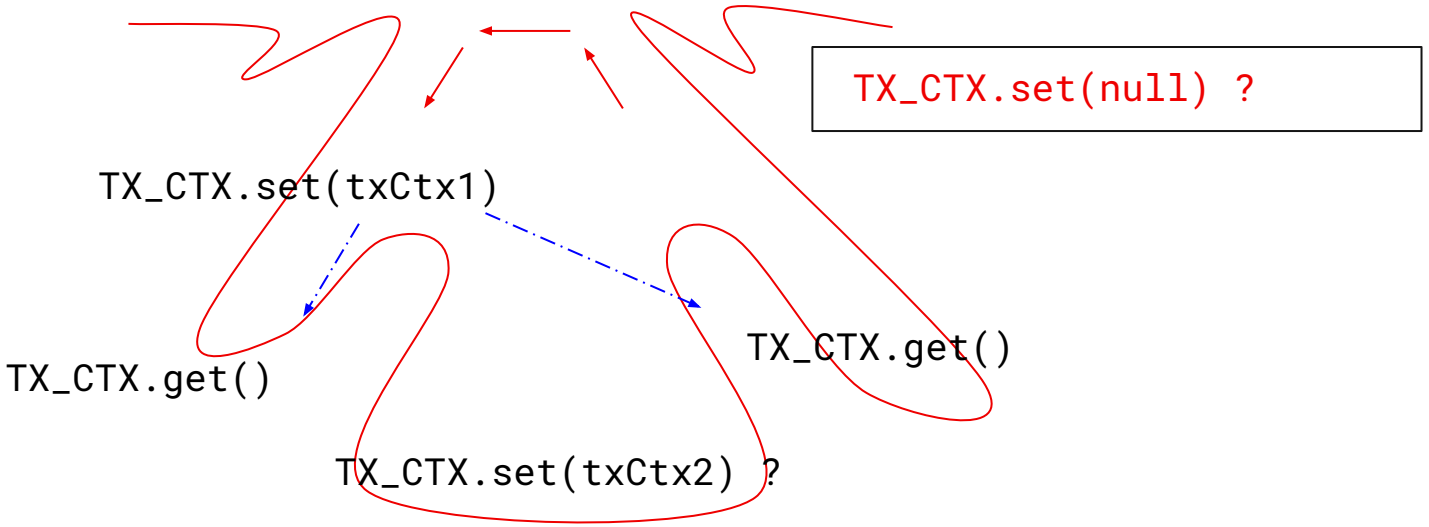
Worst(?) Case `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



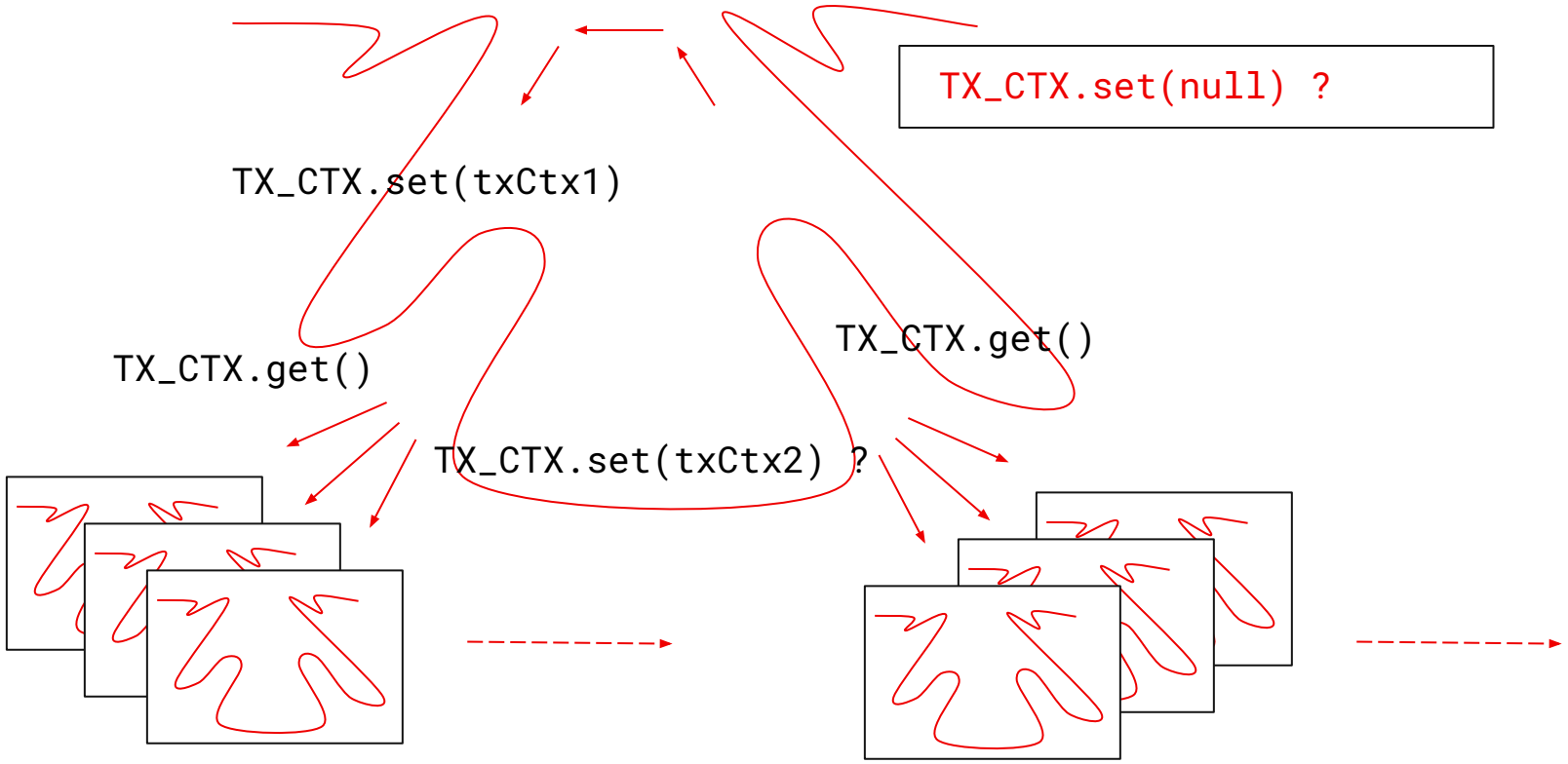
Worst(?) Case `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



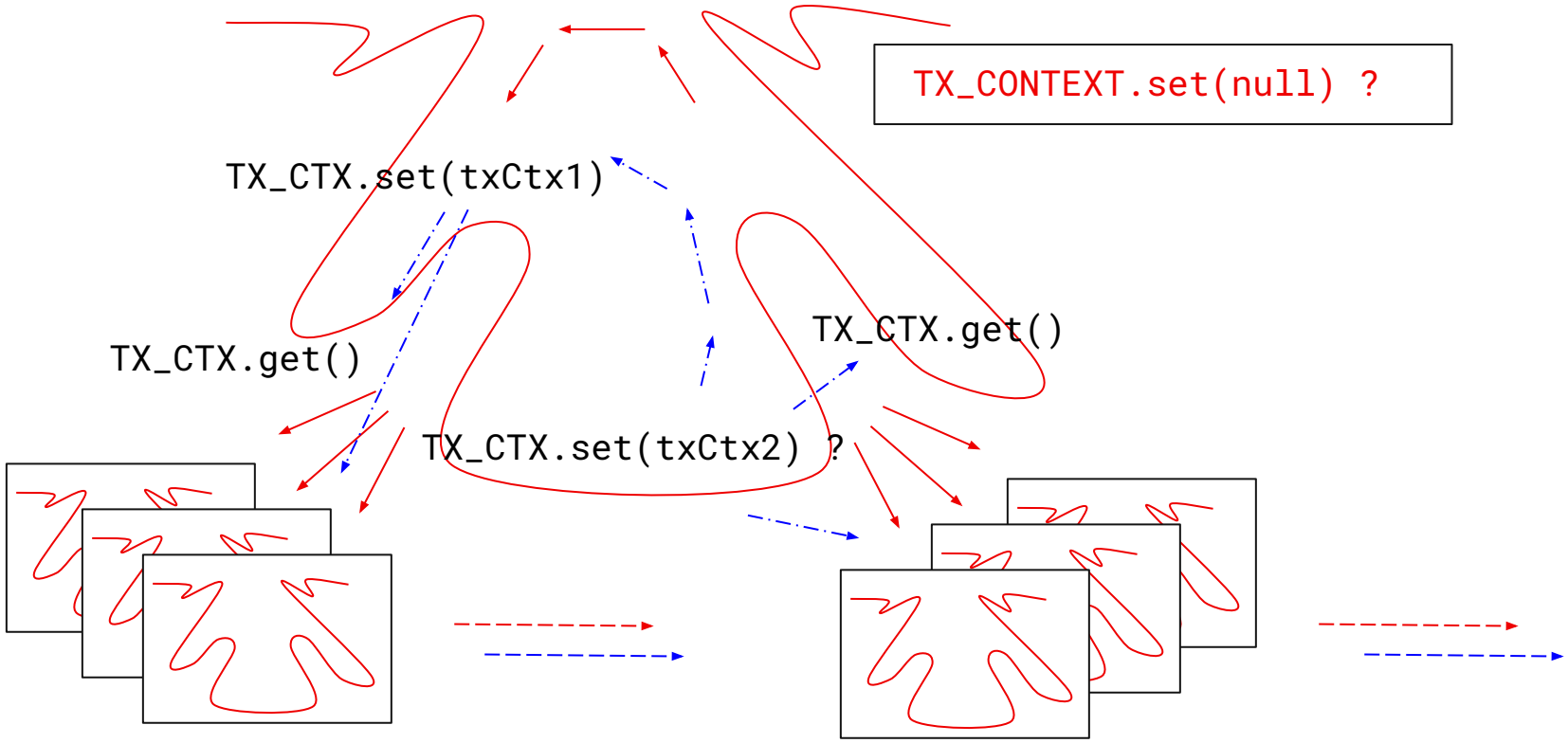
Even Worse Case `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



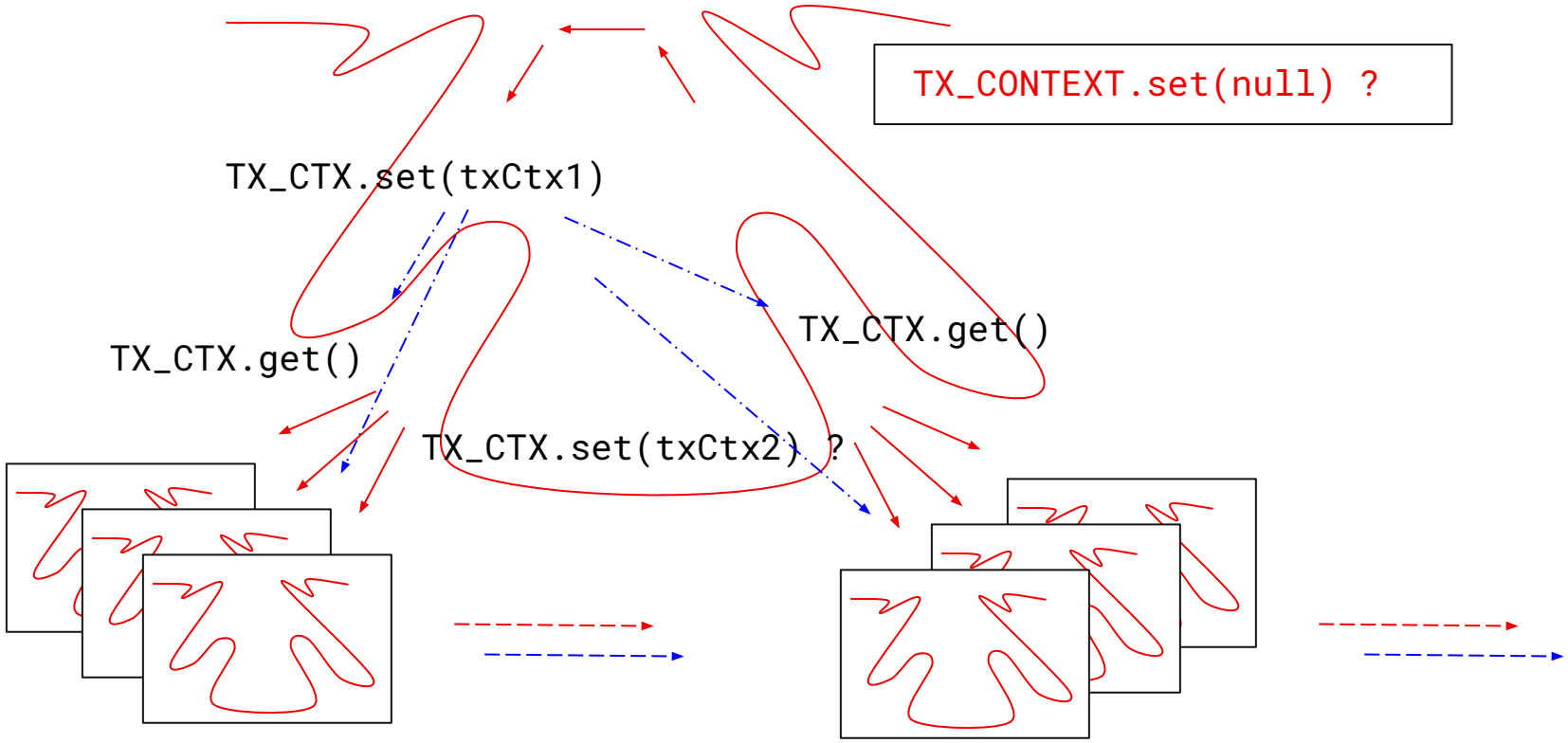
Even Worse Case `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



Even Worse Case `ThreadLocal` use

```
final static ThreadLocal<TransactionContext> TX_CTX = . . .;
```



Scoped values: basic use model

In class `Transaction`:

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

In some top-level server method:

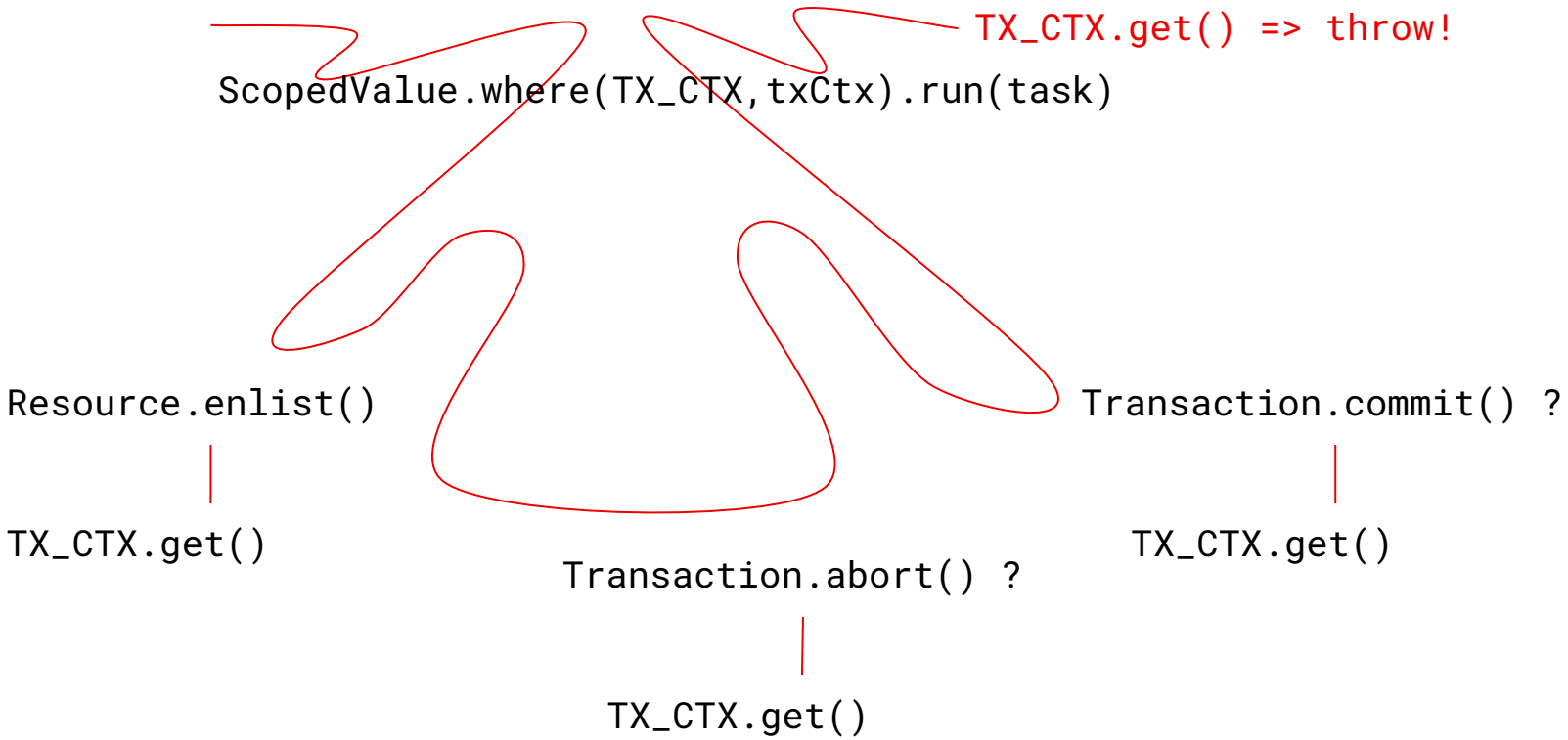
```
Runnable task = () -> { serveRequest(...); }
TransactionContext txCtx = new TransactionContext(...);
ScopedValue.where(TX_CTX, txCtx).run(task);
```

In a method called directly or indirectly from `serveRequest`:

```
... TX_CTX.get() ...
```

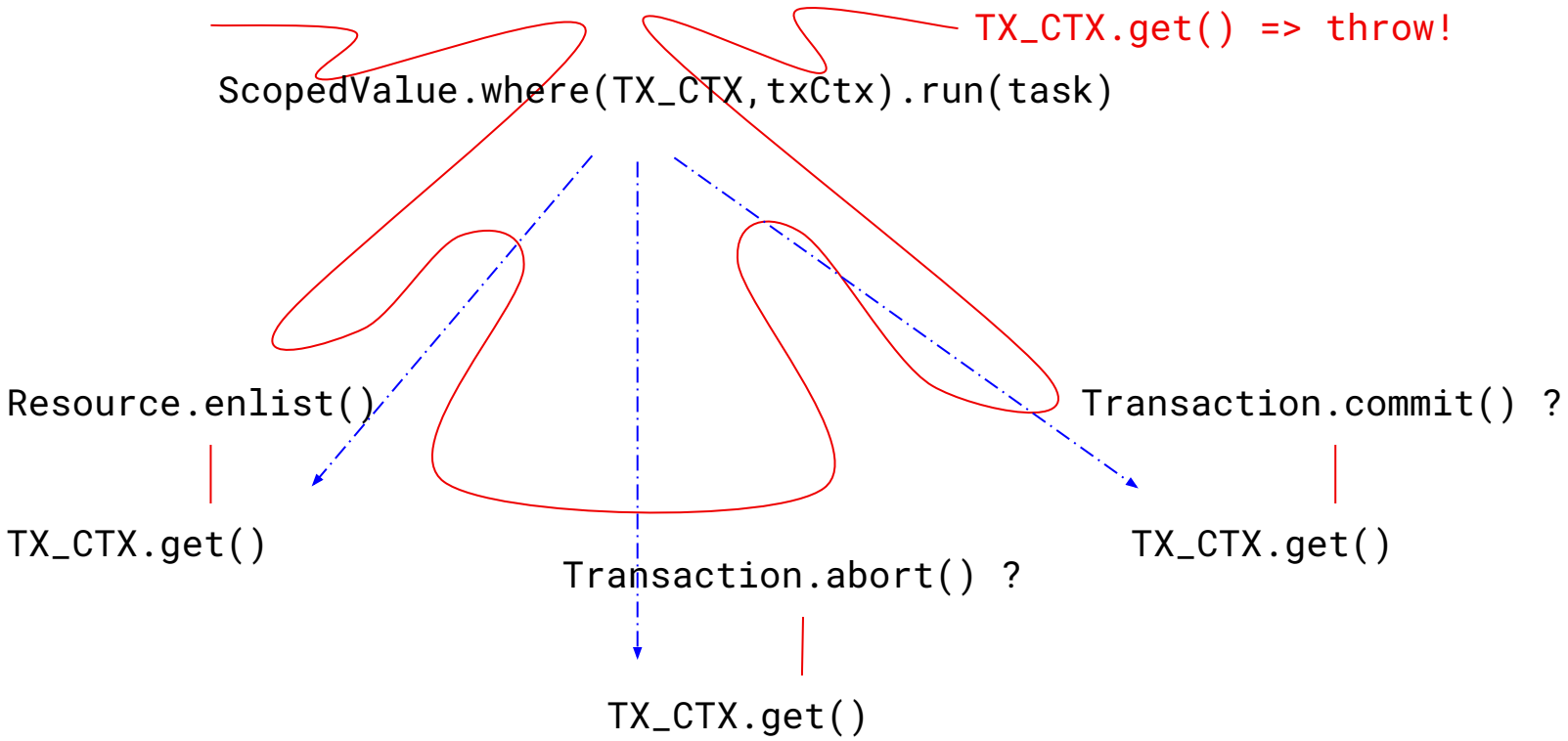

Clean `ScopedValue` use

```
final static ScopedValue<TransactionContext> TX_CTX = . . .;
```



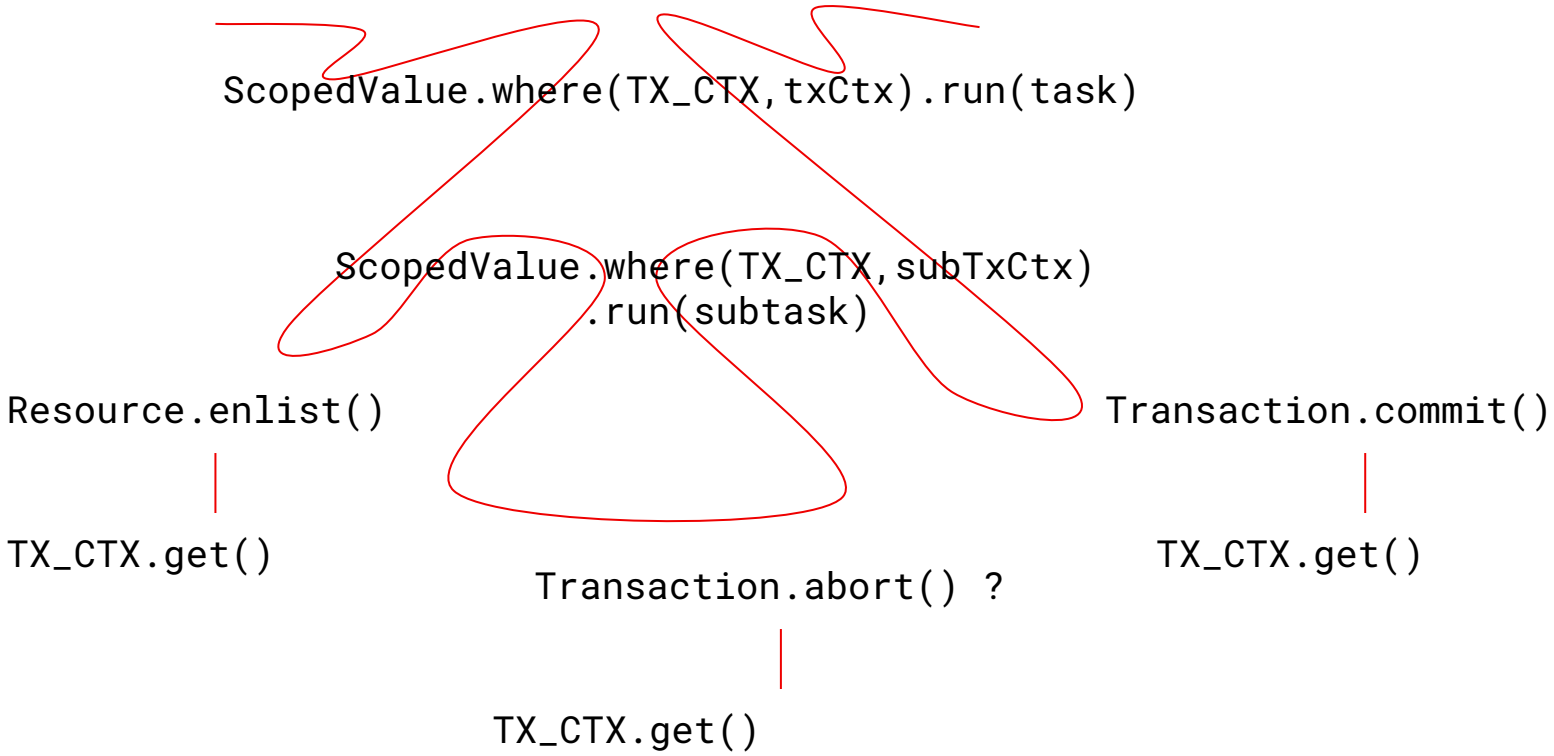
Clean `ScopedValue` use

```
final static ScopedValue<TransactionContext> TX_CTX = . . .;
```



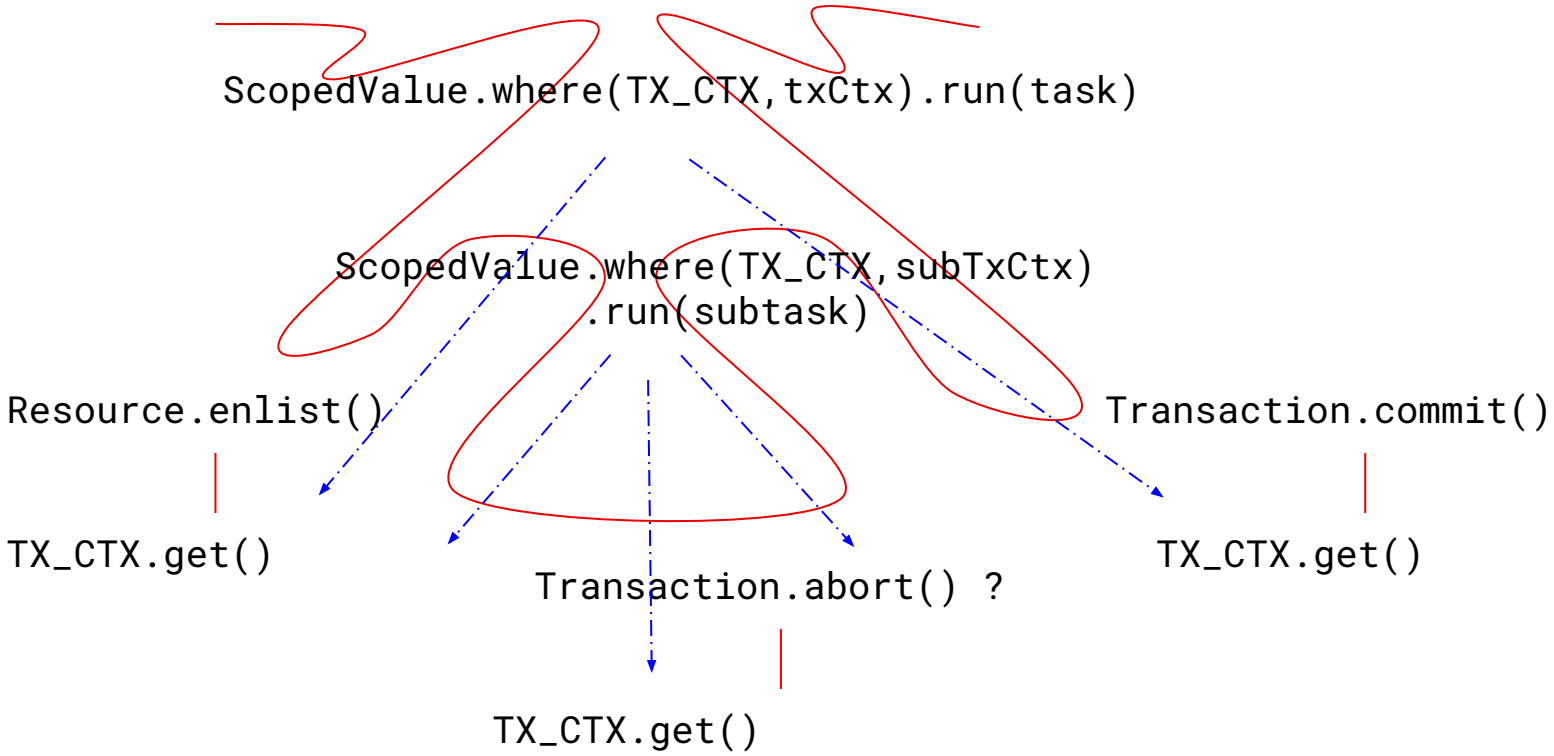
Rebinding a ScopedValue

```
final staticScopedValue<TransactionContext> TX_CTX = . . .;
```



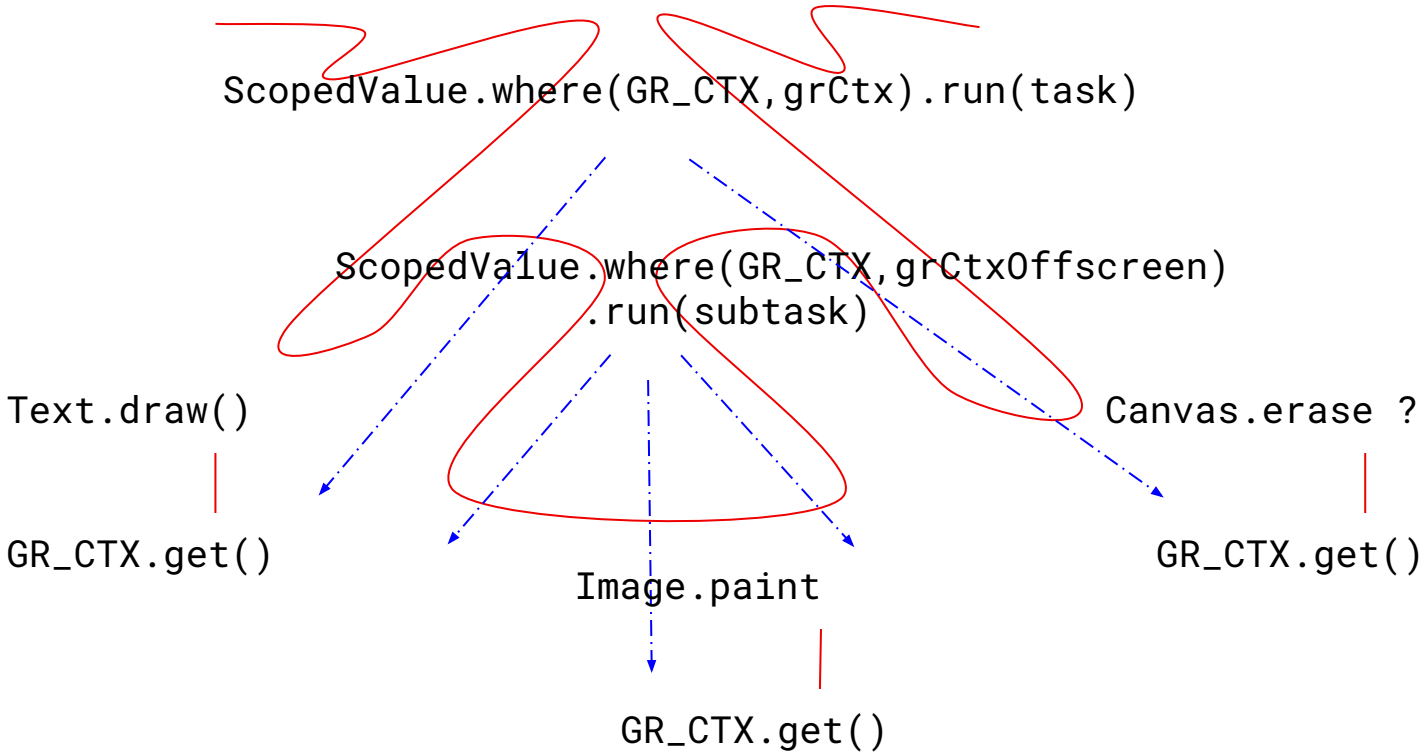
Rebinding a ScopedValue

```
final staticScopedValue<TransactionContext> TX_CTX = . . .;
```



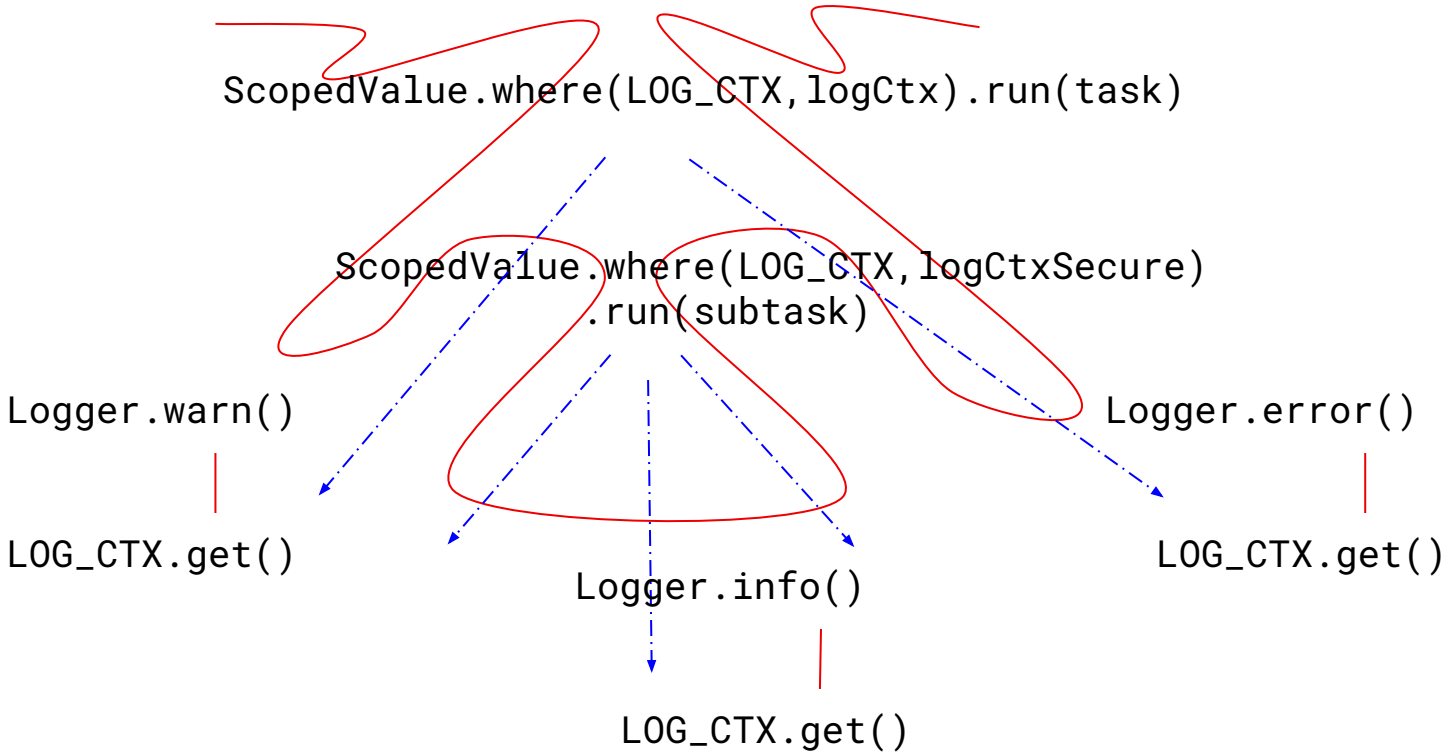
Rebinding a ScopedValue (2)

```
final staticScopedValue<GraphicsContext> GR_CTX = . . .;
```



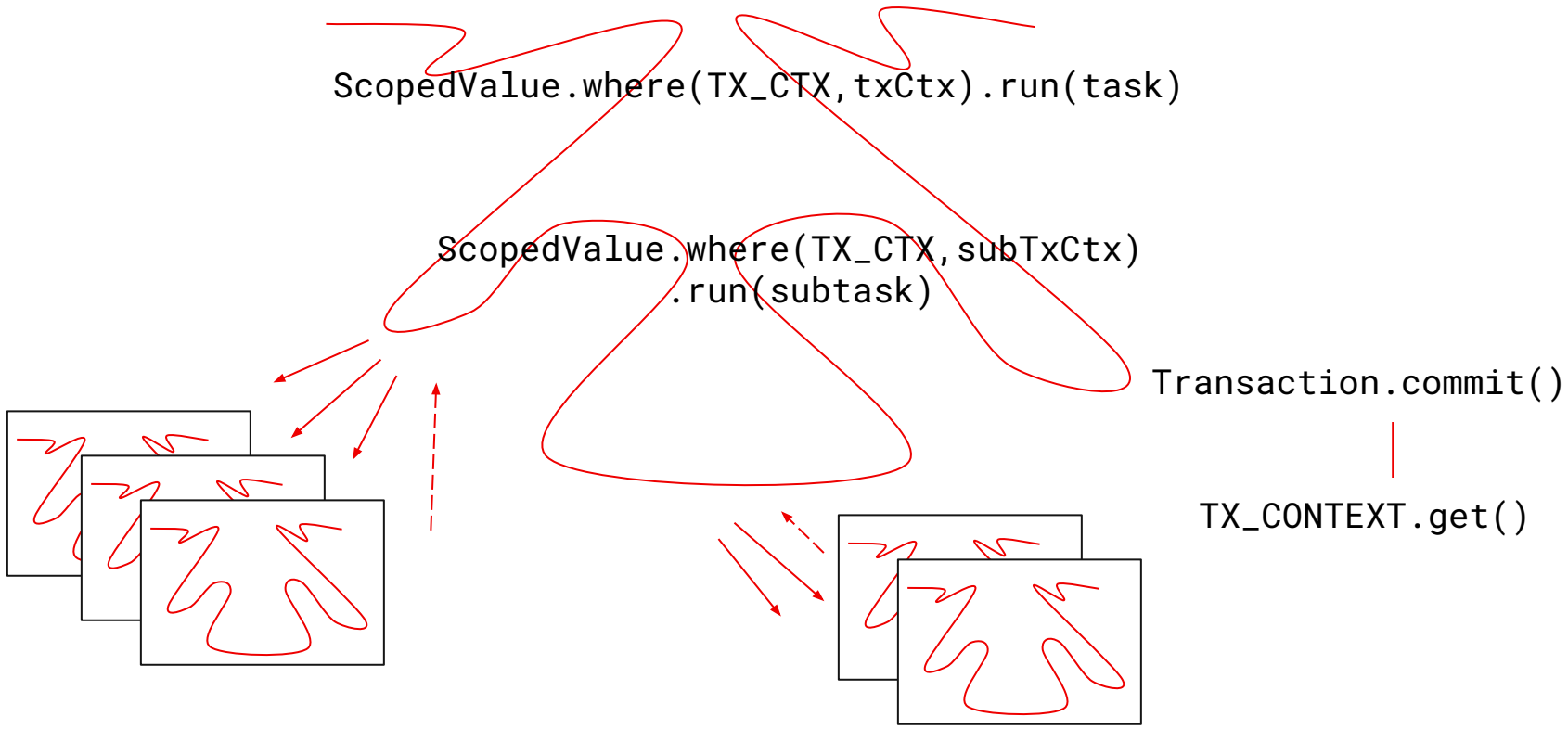
Rebinding a `ScopedValue` (3)

```
final staticScopedValue<LoggerContext> LOG_CTX = . . .;
```



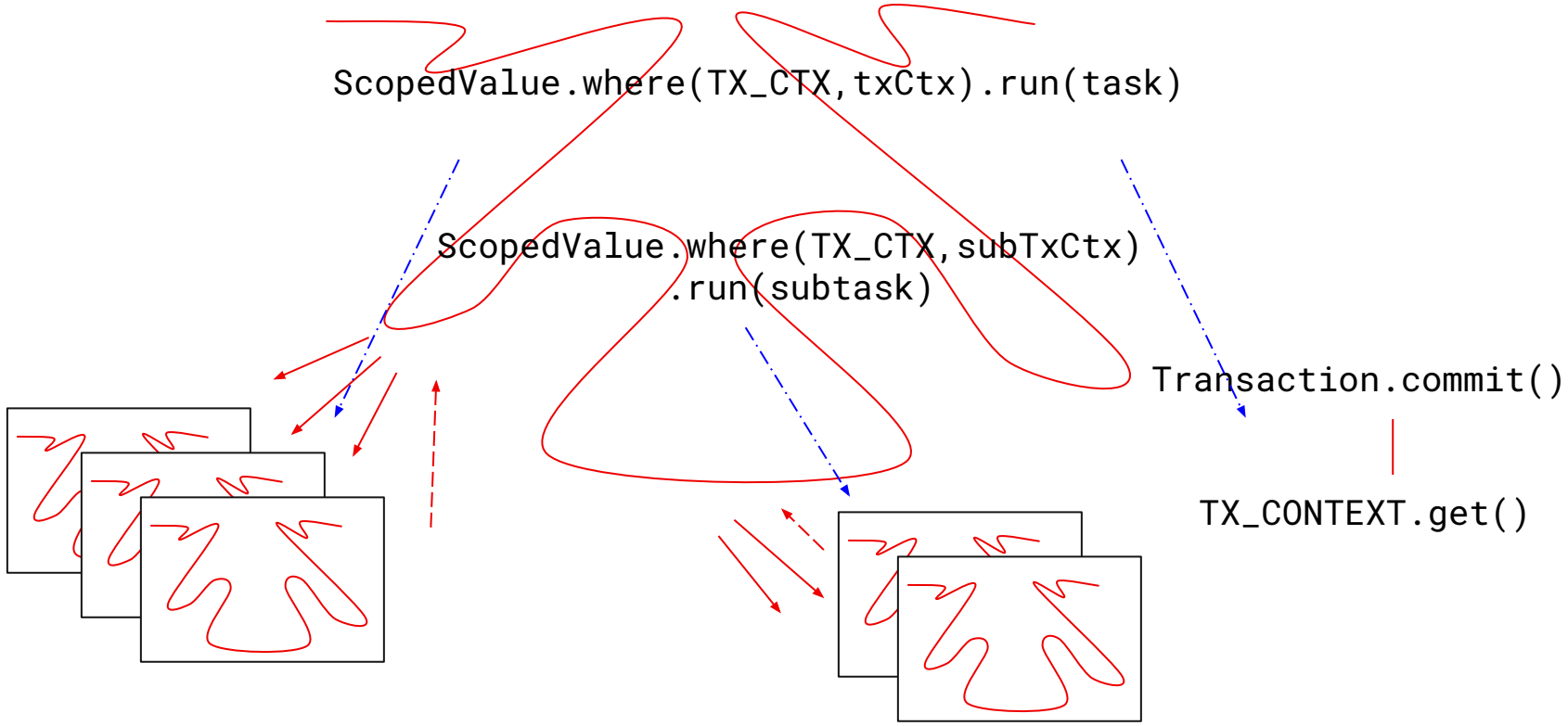
`ScopedValue` use with Structured Concurrency

```
final static ScopedValue<TransactionContext> TX_CTX = . . .;
```



`ScopedValue` use with Structured Concurrency

```
final staticScopedValue<TransactionContext> TX_CTX = . . .;
```



Key Differences between ScopedValues and ThreadLocals

- Scoped values have a `get()` method, but no `set()`.
- Data can only flow from a caller to its (transitive) callees
- Once the lambda which bound a scoped value exits, the binding disappears
- This is a *strong guarantee*: it is enforced by the JVM, even under extreme conditions such as stack overflow and out-of-memory

Scoped values: the wider picture

- It’s useful to think of scoped values as invisible, *effectively final*, parameters that are passed through every method invocation
- These parameters will be accessible within the *dynamic scope* of a scoped value's *binding operation*
- The dynamic scope is the set of methods invoked within the scope of the binding operation, and any methods invoked transitively by them
- The dynamic scope effectively extends across structured concurrency thread boundaries
- Access can be restricted using module/package/class access restrictions just as with Thread Locals.

Scoped values: the full API

- We tried many alternatives over a period of time, refining the API further and further, until what we have now is almost the simplest version imaginable

Scoped values: the API

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`[PREVIEW]`<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.Carrier`[PREVIEW]	`where(ScopedValue`[PREVIEW]`<T> key, T value)`	Creates a new `Carrier` with a single mapping of a `ScopedValue` *key* to a value.

Scoped values and Structured Concurrency

- Alan has talked about Structured Concurrency, but the key point is that scoped values in the parent thread are automatically inherited by child threads created with `StructuredTaskScope`.
- Because there is no `ScopedValue.set()` method, child threads can share data from the parent without the risk of one thread side-affecting another
- But if you really do want to share mutable state between child threads, you can still do that by using, for example, a `Reference` type

Scoped values and Structured Concurrency

- Structured Concurrency ensures that if a task splits up into concurrent subtasks then they all return to the same place, namely the task's code block. In effect, subtasks have a *bounded lifetime*.
- Scoped values ensure that once the lambda which bound a scoped value exits, the binding disappears. In effect, scoped values have a *bounded lifetime*.
- This is the essential synergy: we know that once a `StructuredTaskScope` exits, all of its child threads have terminated. It's safe to close any resources, such as files, that were shared via a Scoped Value.
- So, the combination of Scoped Values and Structured Concurrency provides a foundation for clear, safe, reliable per-thread resource management.

Addendum: Scoped values efficiency

- This talk is really about the API, not the implementation, but the API design makes efficient implementation possible
- Scoped value bindings are immutable, so are shared without any copying
- Fast access to scoped values is possible because of a small thread-local cache. This makes repeated accesses to a scoped value almost as fast as a local variable
- This cache is created for each thread as required, so threads which don't access scoped values pay no cost

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


x.com/RedHat

Red Hat