**Java 21 — Scoped Values**

**1. What are Scoped Values?**

* A **scoped value** is like a **temporary, read-only variable** that you make available to code inside a specific block of execution.
* You can **bind a value for a scope** → all methods and tasks inside that scope can read it.
* After the scope ends, the binding is **gone automatically**.

Think of it as:  
➡️ “Instead of passing a parameter everywhere, I can bind it once and all my methods/tasks can read it.”

**2. Why do we need Scoped Values?**

* Passing context (like request ID, user ID, security info) across many methods becomes messy.
* ThreadLocal was used earlier, but it has problems:
  + Mutable, easy to leak memory.
  + Heavy when used with **virtual threads**.
* **Scoped Values** solve this:
  + Immutable (cannot change once bound).
  + Automatically cleaned up.
  + Light-weight, works well with **virtual threads** and **structured concurrency**.

**3. When to use?**

✅ Use when you want to:

* Pass **read-only context** (request ID, user identity, trace ID, locale, etc.).
* Share context with **methods or child tasks** without changing method signatures.
* Work with **virtual threads / structured concurrency**.

❌ Don’t use when:

* You need mutable state.
* You need values that must live longer than the scope.

**4. API Basics**

* ScopedValue<T> name = ScopedValue.newInstance(); → Create a scoped value key.
* ScopedValue.where(key, value).run(() -> { ... }) → Bind a value for that block.
* ScopedValue.get() → Read the current bound value inside the block.
* Use **Carrier** if you want to run tasks/threads with the same binding.

**5. Example Code**

Use a **Carrier** if you want to propagate:

**package** Java21;

**import** java.lang.ScopedValue;

**public** **class** ScopedValueExample {

**public** **static** **final** ScopedValue<String> ***REQUEST\_ID*** = ScopedValue.*newInstance*();

**public** **static** **void** main(String[] args) **throws** InterruptedException {

// Create a carrier with REQUEST\_ID bound

ScopedValue.Carrier carrier = ScopedValue.*where*(***REQUEST\_ID***, "req-12345");

// Run inside the carrier in the main thread

carrier.run(() -> {

System.***out***.println("Main thread sees: " + ***REQUEST\_ID***.get());

// Propagate the same carrier to the child virtual thread

Thread vt = Thread.*ofVirtual*().start(() -> {

carrier.run(() -> {

System.***out***.println("Virtual child sees: " + ***REQUEST\_ID***.get());

});

});

**try** {

vt.join();

} **catch** (InterruptedException e) {

Thread.*currentThread*().interrupt();

}

});

// After scope ends

System.***out***.println("Outside scope, bound? " + ***REQUEST\_ID***.isBound());

}

}

Without Using carrier:

import java.lang.ScopedValue;

import java.util.concurrent.Executors;

public class ScopedValueExample {

// create a named scoped value (like a static key)

public static final ScopedValue<String> REQUEST\_ID = ScopedValue.newInstance();

public static void main(String[] args) throws Exception {

// Bind the scoped value for the duration of the runnable

ScopedValue.where(REQUEST\_ID, "req-12345").run(() -> {

// Inside this run: REQUEST\_ID is visible

System.out.println("Main thread sees: " + REQUEST\_ID.get());

// Start a child thread (example with a platform thread)

Thread child = new Thread(() -> {

// Child thread inherits the scoped value (if created inside the scope)

System.out.println("Child thread sees: " + REQUEST\_ID.get());

});

child.start();

try { child.join(); } catch (InterruptedException e) { Thread.currentThread().interrupt(); }

});

// Outside the run scope -> value not bound

boolean bound = REQUEST\_ID.isBound(); // false (API variant may differ)

System.out.println("Outside scope, bound? " + bound);

}

}

**Code Explanation:**

**Create a ScopedValue Key**

* public static final ScopedValue<String> REQUEST\_ID = ScopedValue.newInstance();

**Logic:**

* Here we create a **ScopedValue key** named REQUEST\_ID.
* This acts like a **placeholder** or **slot** for a value that we can bind later.
* static final means there is **one shared key** across the program, and it doesn’t change.

**Create a Carrier with a Binding**

* ScopedValue.Carrier carrier = ScopedValue.where(REQUEST\_ID, "req-12345");

**Logic:**

* A **Carrier** is like a **box that stores bindings**.
* Here we create a carrier and bind REQUEST\_ID → "req-12345".
* This carrier can now **propagate this binding** to other threads or tasks safely.

**Purpose:**

* Instead of the binding being limited to one thread/block, the carrier allows **child threads** to access it too.

**Run code inside the Carrier (Main Thread)**

* carrier.run(() -> {

System.out.println("Main thread sees: " + REQUEST\_ID.get());

...

});

**Logic:**

* carrier.run(...) temporarily **activates the binding** inside this block.
* Inside the block:
  + REQUEST\_ID.get() → returns "req-12345".
* After the block finishes, the binding is **automatically removed**.

**Propagate the Carrier to a Child Virtual Thread**

* Thread vt = Thread.ofVirtual().start(() -> {

carrier.run(() -> {

System.out.println("Virtual child sees: " + REQUEST\_ID.get());

});

});

**Logic:**

* We create a **virtual thread** (Thread.ofVirtual().start) to simulate a **subtask**.
* We use the same **carrier** inside the child thread:
  + This ensures the **child thread sees the same scoped value binding**.
  + Without the carrier, the child thread would throw NoSuchElementException because the binding is not automatically inherited for platform threads or virtual threads in some JDK versions.

**Wait for the Child Thread to Finish**

* try {

vt.join();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

**Logic:**

* vt.join() makes the main thread **wait** until the child virtual thread finishes execution.
* InterruptedException is handled by restoring the thread’s **interrupt flag**.

**Purpose:**

* Ensures the output from the child thread appears **before the main thread continues**.

**Outside the Scope**

* System.out.println("Outside scope, bound? " + REQUEST\_ID.isBound());

**Logic:**

* After the carrier’s run() block finishes, the binding is **removed automatically**.
* REQUEST\_ID.isBound() returns false because the value is no longer available outside the scope.

**Purpose:**

* Demonstrates that scoped values are **temporary** and **do not leak** outside their intended block.

OutPut:

Main thread sees: req-12345

Virtual child sees: req-12345

Outside scope, bound? False

* Without Carrier → child thread won’t see the value.
* With Carrier → child thread will see the value.

Flow Diagram:

ScopedValue.where(REQUEST\_ID, "req-12345")

|

|

[ Main Thread ]

| (REQUEST\_ID available)

|--- calls service/method

|--- starts child thread with Carrier

|

|

[ Child Thread ]--- sees REQUEST\_ID

|

|

Scope ends

|

|

REQUEST\_ID unbound (clean)

Difference from ThreadLocal:

| **Feature** | **ScopedValue** | **ThreadLocal** |
| --- | --- | --- |
| Mutability | Immutable (read-only) | Mutable |
| Lifetime | Limited to scope | Tied to thread lifetime |
| Cleanup | Auto after scope | Manual remove needed |
| Virtual threads | Efficient, light | Heavy, memory overhead |
| Safety | No leaks | Risk of leaks |

ScopedValue Class Methods:

### 1. static <T> ScopedValue<T> newInstance()

* **Purpose:** Creates a new ScopedValue key.

### 2. T get()

* **Purpose:** Returns the value currently bound in this scope.
* **Throws:** NoSuchElementException if no value is bound.

### 3. T getOrDefault(T defaultValue) (depends on API version)

* **Purpose:** Returns the bound value if exists; otherwise returns default.

### 4. boolean isBound()

* **Purpose:** Checks if a value is currently bound in the current scope.

### 5. static <T> ScopedValue<T>.WhereBinding where(ScopedValue<T> key, T value)

* **Purpose:** Creates a **temporary binding** of a scoped value.

### 6. run(Runnable r) / call(Callable<T> c) (inside WhereBinding or Carrier)

* **Purpose:** Runs code with the binding applied.

ScopedValue.Carrier Methods:

### 1. run(Runnable r)

* Runs the given code **inside the carrier’s binding**.

### 2. call(Callable<T> c)

* Runs the callable inside the carrier and **returns a result**.

### 3. static Carrier where(ScopedValue<T> key, T value)

* **Purpose:** Create a new Carrier with one binding.

### 4. carrier.run(...) inheritance

* **Important:** Carrier allows passing bindings to **threads, virtual threads, and other tasks**.

## Key Points to Remember

* Always declare scoped value keys as static final.
* Bind using where(...).run(...) or a Carrier.
* Use Carrier to propagate to child threads.
* After scope ends, value is cleaned up automatically.
* Works best with **virtual threads** and **structured concurrency**.