**ThreadLocalMap**

Occasionally, we may want to create a variable such that each thread has its own separate storage for it. This is a common way to simplify code, if passing a value as a parameter is especially awkward or frequent. We can do this in a straightforward way with Java’s *ThreadLocal* class. But what if we want a convenient way to store key / value pairs as thread-local?

A picture containing text

Description automatically generated

**Thread Local Variables**

As we just mentioned, a *thread-local variable*is a variable which stores a different value for each thread that accesses it. We can create a *ThreadLocal* field like this:

ThreadLocal<Integer> level = ThreadLocal**.withInitial(() -> 0)**;

Here, the thread-local variable *level* is initialized to 0 for each thread. After this, each thread can change its own value with *get()* or *set()*without affecting the others:

void **enter()**  
{  
 **level.set**(level.get() + 1);  
}void **exit()**  
{  
 **level.set**(level.get() - 1);  
}

Methods very similar to these are in [*ReentrancyTracker*](https://github.com/Telenav/kivakit/blob/develop/kivakit-kernel/src/main/java/com/telenav/kivakit/kernel/language/threading/status/ReentrancyTracker.java) in [KivaKit](https://www.kivakit.org/). We can track method re-entrancy (when a method directly or indirectly calls itself) like this:

ReentrancyTracker tracker = new **ReentrancyTracker**();void complicated()  
{  
 try  
 {  
 var reentrancy = **tracker.enter()**;  
 switch (reentrancy)  
 {  
 case **ENTERED**:  
 [...] case **REENTERED**:  
 [...]  
 }  
 final  
 {  
 **tracker.exit();**  
 }  
}

Shape

Description automatically generated

**A Thread-Local Map**

Now, for the main event. We can create a *ThreadLocalMap* class like this:

class ThreadLocalMap<Key, Value> extends AbstractMap<Key, Value>  
{  
 ThreadLocal<Map<Key, Value>> map =  
 ThreadLocal.*withInitial*(HashMap::new);  
  
 public Set<Entry<Key, Value>> entrySet()  
 {  
 return map.get().entrySet();  
 }  
  
 public Value put(Key key, Value value)  
 {  
 return map.get().put(key, value);  
 }  
}

Here, Java’s *AbstractMap* class allows us to create a functional *Map* implementation by overriding just two methods: *entrySet()* and *put()*. In our version of these methods, instead of storing values in the data structure managed by the superclass, we store values in the thread-local *HashMap* field named *map*.

So long as we are not concerned with performance, we’re all done. If we want to create large maps, however, we will want to override *get()*as well, because the default implementation of this in *AbstractMap* does a linear search of the entries returned by *entrySet()*. This is, of course, trivial:

public Value get(Key key)  
{  
 return map.get().get(key, value);  
}

And there we have it, a thread-local map. For the full source, see KivaKit’s [ThreadLocalMap](https://github.com/Telenav/kivakit/tree/develop/kivakit-kernel/src/main/java/com/telenav/kivakit/kernel/language/threading/local) class on GitHub.

Uses of ThreadLocal

1. Data isolation between threads

2. Session management of database connection

3. storage of transactional information of the thread