Thread Local

Thread Local

Thread Local can be considered as a thread scope. You can set any object in Thread Local and this object will be global and local to the specific thread which is accessing this object.

- Values stored in Thread Local are global to the thread, meaning that they can be accessed from anywhere inside that thread. If a thread calls methods from several classes, then all the methods can see the Thread Local variable set by other methods (because they are executing in same thread). The value need not be passed explicitly. It's like how you use global variables.
- Values stored in Thread Local are local to the thread, meaning that each thread will have it's own Thread Local variable. One thread can not access/modify other thread's Thread Local variables.

Creating a ThreadLocal

private ThreadLocal myThreadLocal = new ThreadLocal();
This only needs to be done once per thread. Even if different threads execute the same code which accesses a ThreadLococal, each thread will see only its own ThreadLocal instance.

Accessing a ThreadLocal

The values can be set as below
myThreadLocal.set("A thread local value");
You read the value stored in a ThreadLocal like this:
String threadLocalValue = (String) myThreadLocal.get();
The get() method returns an Object and the set() method takes an Object as parameter.

Generic ThreadLocal

You can create a generic ThreadLocal so that you do not have to typecast the value returned by get(). Here is a generic ThreadLocal example:
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
Now you can only store strings in the ThreadLocal instance. Additionally, you do not need to typecast the value obtained from the ThreadLocal:
myThreadLocal.set("Hello ThreadLocal");
String threadLocalValue = myThreadLocal.get();

Initial ThreadLocal Value:

Since values set on a ThreadLocal object only are visible to the thread who set the value, no thread can set an initial value on a ThreadLocal using set() which is visible to all threads.
Instead you can specify an initial value for a ThreadLocal object by subclassing ThreadLocal and overriding the initialValue() method. Here is how that looks:

```
private ThreadLocal myThreadLocal = new ThreadLocal<String>() {
 @Override protected String initialValue() {
 return "This is the initial value";
 }
};
```

Now all threads will see the same initial value when calling get() before having called set() .

When to Use Thread Local?

Use-Case :]
Consider you have a Servlet which calls some business methods. You have a requirement to generate a unique transaction id for each and every request this servlet process and you need to pass this transaction id to the business methods, for logging purpose. One solution would be passing this transaction id as a parameter to all the business methods. But this is not a good solution as the code is redundant and unnecessary.
To solve that, you can use Thread Local. You can generate a transaction id (either in servlet or better in a filter) and set it in the Thread Local. After this, what ever the business method, that this servlet calls, can access the transaction id from the thread local. This servlet might be servicing more that one request at a time. Since each request is processed in separate thread, the transaction id will be unique to each thread (local) and will be accessible from all over the thread's execution (global).