**Quiz 8**

Questions on working with ThreadLocal variables

**Question # 1**

***Consider the class below:***

public class Counter {  
   
    ThreadLocal<Integer> counter = ThreadLocal.withInitial(() -> 0);  
   
    public Counter() {  
        counter.set(10);  
    }  
   
    void increment() {  
        counter.set(counter.get() + 1);  
    }  
}

***What would be the output of the method below when invoked?***

    public void usingThreads() throws Exception {  
   
        Counter counter = new Counter();  
        Thread[] tasks = new Thread[100];  
   
        for (int i = 0; i < 100; i++) {  
            Thread t = new Thread(() -> {  
                for (int j = 0; j < 100; j++)  
                    counter.increment();  
            });  
            tasks[i] = t;  
            t.start();  
        }  
   
        for (int i = 0; i < 100; i++) {  
            tasks[i].join();  
        }  
   
        // What is the output of the the below line?  
        System.out.println(counter.counter.get());  
    }

**Q**

**Your Answer**

**A)**

0

**Correct Answer**

**B)**

10

**C)**

100

**D)**

110

**E)**

10000

?????

**Oops, you got that wrong. Want to try again?**

Retake Quiz

**Hide Explanation**

Note that aside from the 100 threads that we create, there's the **main** thread that creates those 100 threads. The constructor for the **counter** variable is invoked only by the main thread and its copy of the threadlocal variable is initialized to 10. The main thread never increments the variable so the print statement prints out 10.

Note that we must initialize the ThreadLocal variable inline than in the constructor because the constructor would only be invoked for a single thread. Usually, static variables are declared threadlocal to keep separate values on a per thread basis.

**Question # 2**

***Given the same Counter class as in the previous question, what is the output of println statement below:***

    public void usingSingleThreadPool() throws Exception {  
   
        Counter counter = new Counter();  
        ExecutorService es = Executors.newFixedThreadPool(1);  
        Future<Integer>[] tasks = new Future[100];  
   
        for (int i = 0; i < 100; i++) {  
            tasks[i] = es.submit(() -> {  
                for (int j = 0; j < 100; j++)  
                    counter.increment();  
   
                return counter.counter.get();  
            });  
        }  
   
        // What is the output of the below line?  
        System.out.println(tasks[99].get());  
   
        es.shutdown();  
    }

?????

**Oops, you got that wrong. Want to try again?**

Retake QuizCheck Answers

**Hide Explanation**

This example may seem counterintuitive but drives home the warning of using threadlocal variables with threadpools! Note that even though we submit 100 tasks but our threadpool consists of a lone thread. This thread has a copy of threadlocal vaiable **counter** which gets reused when executing different tasks. It is not issued a new copy of the variable for every task as one might mistakenly assume. The same copy gets incremement for a 100 tasks a 100 times resulting in a value of 10,000 being printed.

**Question # 3**

***What would have been the output of the print statement from the previous question if we created a pool with 20 threads?***

?????

**Oops, you got that wrong. Want to try again?**

Retake QuizCheck Answers

**Hide Explanation**

Since our threadpool consists of 20 threads the print statement may print a value between the two extremes. If only a single thread gets to execute all the tasks then counter will print a maximum value of 10,000. If a thread only executed the 100th task and all the other 99 tasks were performed by the remaining 19 threads then the printed value would be 100.

The code for all the three scenarios discussed above appears below.

30

31

28

29

24

25

26

27

21

22

23

19

20

15

16

17

18

13

14

11

12

9

10

8

6

7

5

4

2

3

1

        System.out.println(counter.counter.get());

            tasks[i].join();

        }

            t.start();

        }

        for (int i = 0; i < 100; i++) {

                    counter.increment();

            });

            tasks[i] = t;

            Thread t = new Thread(() -> {

                for (int j = 0; j < 100; j++)

        Counter counter = new Counter();

        Thread[] tasks = new Thread[100];

        for (int i = 0; i < 100; i++) {

    static void usingThreads() throws Exception {

    }

      usingSingleThreadPool();

      usingMultiThreadsPool();

      usingThreads();

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

class Demonstration {

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import java.util.concurrent.ExecutorService;





RUN

SAVERESET

**Question # 4**

***Consider the below method:***

    int countTo100() {  
   
        ThreadLocal<Integer> count = ThreadLocal.withInitial(() -> 0);  
        for (int j = 0; j < 100; j++)  
            count.set(count.get() + 1);  
   
        return count.get();  
   
    }

***The above code is invoked like so:***

        ExecutorService es = Executors.newFixedThreadPool(1);  
        Future<Integer>[] tasks = new Future[100];  
   
        for (int i = 0; i < 100; i++) {  
            tasks[i] = es.submit(() -> countTo100());  
        }  
   
        for (int i = 0; i < 100; i++)  
            System.out.println(tasks[i].get());  
   
        es.shutdown();

***What would the output of the print statement for the 100 tasks?***

**Q**

**Correct Answer**

**A)**

100

**Your Answer**

**B)**

10000

**C)**

between 100 and 10,000 inclusive

?????

**Oops, you got that wrong. Want to try again?**

Retake Quiz

**Hide Explanation**

Note that the threadlocal variable is now within the instance method. Even though we have a single thread but each time it invokes the **countTo100()** method, it creates a fresh threadlocal object which has no relation to the threadlocal object from the previous invocation. The scope of the threadlocal variable is limited to within the instance method and as soon as the thread exits the method, it is eligible for garbage collection. On the next invocation a new threadlocal object is created.

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

1

2

import java.util.concurrent.Future;

class Demonstration {

    @SuppressWarnings("unchecked")

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

        ExecutorService es = Executors.newFixedThreadPool(1);

        Future<Integer>[] tasks = new Future[100];

        for (int i = 0; i < 100; i++) {

            tasks[i] = es.submit(() -> countTo100());

        }

        for (int i = 0; i < 100; i++)

            System.out.println(tasks[i].get());

        es.shutdown();

    }

    static int countTo100() {

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

        for (int j = 0; j < 100; j++)

            count.set(count.get() + 1);

        return count.get();

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;





RUN

SAVERESET

**Question # 5**

***Is there any benefit to declaring count as a threadlocal variable in the method countTo100()?***

    int countTo100() {  
   
        ThreadLocal<Integer> count = ThreadLocal.withInitial(() -> 0);  
        for (int j = 0; j < 100; j++)  
            count.set(count.get() + 1);  
   
        return count.get();  
   
    }

The variables defined inside an instance method are already created on a per-thread basis and live on the thread stack without any sharing with other threads. The per-thread level isolation for a variable that we can achieve using threadlocal is already being provided because of the scope of the variables declared within an instance method. Therefore, there's no benefit to declaring variables within instance methods as threadlocal.