**Quiz 1**

**Question # 1**

***What are some of the differences between a process and a thread?***

* A process can have many threads, whereas a thread can belong to only one process.
* A thread is lightweight than a process and uses less resources than a process.
* A thread has some state private to itself but threads of a process can share the resources allocated to the process including memory address space.

**Question # 2**

***Given the below code, can you identify what the coder missed?***

    void defectiveCode(final int n) throws ExecutionException, InterruptedException {  
   
        ExecutorService threadPool = Executors.newFixedThreadPool(5);  
   
        Callable<Void> sumTask = new Callable<Void>() {  
   
            public Void call() throws Exception {  
                System.out.println("Running");  
                return null;  
            }  
        };  
   
        threadPool.submit(sumTask);  
        f.get();  
    }

The above code forgets to ***shutdown*** the executor thread pool. The thread pool when instantiated would also create 5 worker threads. If we don't shutdown the executor when exiting the main method, then JVM would also not exit. It will keep waiting for the pool's worker threads to finish, since they aren't marked as daemon. As an example execute the below code snippet.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

class Demonstration {

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

      ExecutorService threadPool = Executors.newFixedThreadPool(5);

      Callable<Void> someTask = new Callable<Void>() {

            public Void call() throws Exception {

                System.out.println("Running");

                return null;

            }

        };

      threadPool.submit(someTask).get();

      System.out.println( "Program Exiting" );

    }

}





RUN

SAVERESET

The above program execution will show execution timed out, even though both the string messages are printed. You can fix the above code by adding **threadPool.shutdown()** as the last line of the method.

**Question # 3**

***Which compute() method do you think would get invoked when getWorking is called?***

class ThreadsWithLambda {  
   
    public void getWorking() throws Exception {  
        compute(() -> "done");  
    }  
   
    void compute(Runnable r) {  
        System.out.println("Runnable invoked");  
        r.run();  
    }  
   
    <T> T compute(Callable<T> c) throws Exception {  
        System.out.println("Callable invoked");  
        return c.call();  
    }  
}

The lamda expression is returning the string done, therefore the compiler will match the call to the second compute method and the expression will be considered a type of interface **Callable**. You can run the below snippet and verify the output to convince yourself.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

import java.util.concurrent.Callable;

class Demonstration {

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

        (new LambdaTargetType()).getWorking();

    }

}

class LambdaTargetType {

    public void getWorking() throws Exception {

        compute(() -> "done");

    }

    void compute(Runnable r) {

        System.out.println("Runnable invoked");

        r.run();

    }

    <T> T compute(Callable<T> c) throws Exception {

        System.out.println("Callable invoked");

        return c.call();

    }

}





RUN

SAVERESET

**Question # 4**

***What are the ways of representing tasks that can be executed by threads in Java?***

**Q**

**Correct Answer**

**A)**

Runnable interface and subclassing Thread class

Explanation

Runnable interface is the basic task abstraction in Java. A task can be represented through a class implementing the runnable or callable interface. Another way is to extend the thread class itself and override the run method.

**B)**

Passing anonymous class instance to thread constructor

**C)**

IComparable interface and subclassing Thread class

----

**You skipped the question. Would you like to try again?**

Retake Quiz

**Question # 5**

***Given the code snippet below, how many times will the innerThread print its messages?***

    public void spawnThread() {  
   
        Thread innerThread = new Thread(new Runnable() {  
   
            public void run() {  
   
                for (int i = 0; i < 100; i++) {  
                    System.out.println("I am a new thread !");  
                }  
            }  
        });  
   
        innerThread.start();  
        System.out.println("Main thread exiting");  
    }

**Q**

**A)**

innerThread prints a few messages and dies when the main thread exits

**Correct Answer**

**B)**

innerThread prints exactly 100 messages even if the main thread exits before innerThread is done

Explanation

Even if the main thread exits earlier than the innerThread, the JVM will wait for the innerThread to finish. If the snippet is run on the console, exactly 100 messages will be printed by the innerThread.

**C)**

innerThread dies as soon as the main thread exits without printing any messages

----

**You skipped the question. Would you like to try again?**

Retake Quiz

**Question # 6**

***Given the below code snippet how many messages will the innerThread print?***

    public void spawnDaemonThread() {  
   
        Thread innerThread = new Thread(new Runnable() {  
   
            public void run() {  
   
                for (int i = 0; i < 100; i++) {  
                    System.out.println("I am a daemon thread !");  
                }  
            }  
        });  
   
        innerThread.setDaemon(true);  
        innerThread.start();  
        System.out.println("Main thread exiting");  
    }

**Q**

**A)**

exactly 100

**B)**

none

**Correct Answer**

**C)**

a few

Explanation

Since we marked the innerThread as a daemon thead, when the main thread exits, the JVM also kills any threads marked daemon, therefore if run on the console, only a few messages will be printed by the innerThread. The innerThread will be killed by JVM before it can run to completion.