CHAPTER 6

■ ■ ■

Threads

QUESTIONS AND EXERCISES

1. What is a thread? Can threads share memory? What is thread local storage?

   Answer:

   A thread is a unit of execution in a program. All threads within a process share memory. Thread local storage is private memory of a thread which cannot be shared with other threads, even if they are in the same process.

2. What is a multi-threaded program?

   Answer:

   A program that uses multiple threads is called a *multi-threaded program*.

3. What is the name of the class whose objects represent threads in Java programs?

   Answer:

   An object of the `java.lang.Thread` class represents a thread in a Java program.

4. Suppose you create an object of the `Thread` class:

   ```
   Thread t = new Thread();
   ```

   What do you need to do next so that this `Thread` object will get CPU time?

   Answer:

You need to call the `start()` method of the `Thread` object to start the thread. This schedules the thread to receive the CPU time. Using the following statement will start the thread:

```
t.start();
```

5. What is a race condition when using multiple threads? How do you avoid a race condition in your program?

Answer:

A race condition is the situation where multiple threads manipulate and access a shared data concurrently and the outcome depends on the order in which the execution of threads take place. It can be avoided by making sure that only one of the threads works with the shared data at a time.

6. What is a critical section in a program?

Answer:

In a multi-threaded program, a section of code that may have undesirable effects on the outcome of the program if executed by multiple threads concurrently is called a critical section.

7. What is the effect of using the `synchronized` keyword in a method's declaration?

Answer:

In the case of a `synchronized` instance method, the entire method is a critical section and it is associated with the monitor of the object for which this method is executed. That is, a thread must acquire the object's monitor lock before executing the code inside a `synchronized` instance method of that object. In case of a `synchronized` static method, the entire method is a critical section and it is associated with the class object that represents that class. That is, a thread must acquire the class object's monitor lock before executing the code inside a `synchronized` static method of that class.

8. What is thread synchronization? How is thread synchronization achieved in a Java program?

Answer:

Controlling and coordinating the access to a critical section by multiple threads is known as thread synchronization. It can be achieved by mutual exclusion where only one thread is allowed to have access to a code section at a point in time or by conditional synchronization where multiple threads work together to achieve a result.

9. What are an entry set and a wait set of an object?

Answer:

Any thread that wants to acquire the object's monitor lock must enter the object's entry set first. All threads that have released the object's monitor lock and are waiting for some conditions to occur are put in a set called a wait set.

10. Describe the use of the `wait()`, `notify()`, and `notifyAll()` methods in thread synchronization.

Answer:

Suppose a thread has acquired an object's monitor lock and is in middle of a task. The thread needs to wait for some condition to be met to complete its remaining task, so it must release the lock and wait. To do so, it must call the `wait()` method to place itself in the wait set. A thread that has ownership of the monitor must notify the threads waiting in the wait set about the fulfillment of the conditions on which they are waiting. The call to the `notify()` method wakes up one thread from the wait set, whereas the call to the `notifyAll()` method wakes up all threads in the wait set.

11. What method of the `Thread` class do you use to check if a thread is terminated or alive?

Answer:

The `isAlive()` method.

12. Describe the following six states of a thread: New, Runnable, Blocked, Waiting, Timed-waiting, and Terminated. What method in the `Thread` class returns the state of a thread?

Answer:

- New: When a thread is created and its `start()` method is not yet called, it's in new state.
- Runnable: When the `start()` method of a thread is invoked, the thread is ready to run and it's in runnable state.

- Blocked: A thread is in blocked state when it's trying to enter (or re-enter) a synchronized method or block but the monitor is being used by another thread.
- Waiting: When a thread invokes `wait()`, `join()`, or `park()` method, it goes to waiting state.
- Timed-waiting: When a thread invokes `sleep()`, `wait()`, or `join()` method with a timeout in milliseconds, `parkNanos()` or `parkUntil()` with timeout in nanoseconds, it goes to timed-waiting state.
- Terminated: When a thread completes its execution, it goes to the terminated state.

13. Can you restart a thread by calling its `start()` method after the thread is terminated?

Answer:

A terminated thread cannot transition to any other state. It cannot be restarted using the `start()` method.

14. What is thread starvation?

Answer:

Thread starvation occurs when a lower priority thread waits indefinitely or for a long time to get CPU time.

15. What is a daemon thread? What happens when the JVM detects that there are only daemon threads running in the application? Are the main thread and garbage collector thread daemon thread?

Answer:

A daemon thread is a kind of a service provider thread. A user thread uses the services of daemon threads. JVM exits if it detects that all threads are daemon threads. Main thread is non-daemon thread and garbage collector is an example of a daemon thread.

16. How do you interrupt a thread? What is the difference in calling the instance `isInterrupted()` method and static `interrupted()` method of the `Thread` class? What happens when a blocked thread is interrupted?

Answer:

A thread can be interrupted by calling the `interrupt()` method. Both, the `interrupted()` and `isInterrupted()` can be used to check whether thread has been interrupted. The call to the `interrupted()` method clears the interrupted status of a thread whereas calling non-static `isInterrupted()` method does not clear the interrupted status. When a thread blocked thread is interrupted, an `InterruptedException` is thrown and the interrupted status of the thread is cleared.

17. What is a thread group? What is the default thread group of a thread? How do you get an estimate of active threads in a thread group?

Answer:

A thread is always a member of a thread group. By default, the thread group of a thread is the group of its creator thread. The `activeCount()` method of the `ThreadGroup` class returns an estimate of the number of active threads in the group.

18. Describe the use of `volatile` variables in Java programs.

Answer:

Declaring a shared variable as volatile ensures that thread does not cache its value in the thread's working memory. The thread always reads the value of a `volatile` variable from the main memory and writes the value of a `volatile` variable to the main memory. This is faster and cheaper than using a `synchronized` block.

19. What is the difference between using an `AtomicLong` variable and a `long` variable with a synchronized getter and setter?

Answer:

Compare and swap instruction enables lock-free synchronization for one variable. So there is no need for explicit synchronization when using an `AtomicLong`. In case of a `long` variable with `synchronized` getter and setter, there is locking, unlocking of monitor lock, and suspension, resuming of threads.

20. What are semaphores, barriers, phasers, latches, and exchangers? Name the classes in Java that represent instances of these synchronizers.

Answer:

- Semaphore: Controls the number of threads that can access a resource.
- Barrier: Used to make a group of threads meet at a barrier point. A thread from a group arriving at the barrier waits until all threads in that group arrive. Once the last thread from the group arrives at the barrier, all threads in the group are released.
- Phaser: Is a synchronization barrier, provides functionality similar to the `CyclicBarrier` and `CountDownLatch` synchronizers.
- Latch: Works similar to a barrier, makes a group of threads wait until it reaches its terminal state. Once a latch reaches its terminal state, it lets all threads pass through. It is a one-time object. Once it has reached its terminal state, it cannot be reset and reused.
- Exchanger: An exchanger lets two threads wait for each other at a synchronization point. When both threads arrive, they exchange an object and continue their activities.

21. What is the Executor framework? What is the difference between an instance of the `Executor` interface and an instance of the `ExecutorService` interface? What class do you use to get a preconfigured `Executor` instance?

Answer:

The Executor framework provides a way to separate task submission from task execution. A task is created and submitted it to an executor. The executor takes care of the execution details of the task. It also provides configurable policies to control many aspects of the task execution. `ExecutorService` provides some advanced features like managing the shutdown of the executor and checking the status of the submitted tasks. The factory methods of the `java.util.concurrent.Executors` class provide preconfigured `Executor` instances.

22. If you want to submit a result-bearing task to an `Executor`, the task needs to be an instance of which interface: `Runnable` or `Callable<T>`?

Answer:

The task that can return a result upon its execution has to be represented as an instance of the `Callable<T>` interface.

23. What does an instance of the `Future<T>` interface represent?

Answer:

An instance of the `Future<T>` interface indicates that task returns a value of type `T` as its result.

24. What is the difference in using the `shutdown()` and `shutdownNow()` methods to shut down an executor?

Answer:

Invoking the `shutdown()` method shuts down the executor after submitting all tasks. The `shutdownNow()` method attempts to stop the executing tasks by interrupting them and discards the pending tasks. It returns the list of all pending tasks that were discarded.

25. What is the Fork/Join framework?

Answer:

The fork/join framework is an implementation of the executor service, focused on solving problems related to parallelism efficiently, by taking advantage of multiple processors or multiple cores on a machine.

26. Describe the use of the `ThreadLocal<T>` class.

Answer:

A thread-local variable provides a way to maintain a separate value for a variable for each thread. The `java.lang.ThreadLocal<T>` class provides the implementation of a thread-local variable.

27. What JVM option do you use to set the Java thread's stack size?

Answer:

Use `-Xss<size>`, where `<size>` is the size of the thread stack. For example, the option `-Xss512K` sets the stack size to 512 KB.

28. Create a class inheriting it from the `Thread` class. When an instance of the class is run as a thread, it should print text like `1<name> 2<name>, …N<name>` where `<name>` is the name of the thread you specify and `N` is the upper limit on the number of integers starting from 1 to be printed. For example, if you create an instance of your class with 100 and "A", it should print 1A 2A 3A…100A. Create three threads of your class and run them simultaneously.

Solution:

```
public class SampleThread extends Thread {
    private final int num;
    private final String name;

    public SampleThread(String name, int n) {
        this.name = name;
        this.num = n;
    }

    @Override
    public void run() {
        for (int i = 1 ; i <= num ; i++)
            System.out.println(i + name);
    }

    public static void main(String[] args) {
        SampleThread ta = new SampleThread("A", 10);
        SampleThread tb = new SampleThread("B", 20);
        SampleThread tc = new SampleThread("C", 30);
        ta.start();
        tb.start();
        tc.start();
    }

}
```

29. Create a class named `BankAccount`. An instance of this class represents a bank account. It should contain three methods — `deposit()`, `withdraw()`, and `balance()`. They deposit, withdraw, and return the balance in the account. Its `balance` instance variable should store the balance in the account and it is initialized to 100. The balance in the account must not go below 100. Do not use any thread synchronization constructs or keywords in this class. Create an instance of the `BankAccount` class. Pass this instance to four threads – two threads should deposit money and two should withdraw money. The deposit and withdrawal amount should be selected randomly between 1 and 10. Start another thread, a monitor thread, that keeps calling the `balance()` method to

check if the balance goes below 100. When the balance goes below 100, it should print a message and exit the application.

Solution:

```
// BankAccount.java
package com.jdojo.generics.exercises;

import java.util.Random;

public class BankAccount {
    private int balance = 100;

    public void deposit(int amt) {
        balance += amt;
    }

    public void withdraw(int amt) {
        balance -= amt;
    }

    public int balance() {
        return balance;
    }

    public static void main(String[] args) {
        final BankAccount ba = new BankAccount();
        System.out.println("Initial balance: " + ba.balance());
        final Random random = new Random();
        startDepositThread(ba, random);
        startWithdrawThread(ba, random);
        startDepositThread(ba, random);
        startWithdrawThread(ba, random);
        startMonitorThread(ba);
    }

    private static void startDepositThread(BankAccount ba, Random random) {
        Thread t = new Thread(() -> {
            while (true) {
                ba.deposit(random.nextInt(10) + 1);
            }
        });
        t.start();
    }

    private static void startWithdrawThread(BankAccount ba, Random random) {
        Thread t = new Thread(() -> {
            while (true) {
                ba.withdraw(random.nextInt(10) + 1);
            }
        });
        t.start();
    }

    private static void startMonitorThread(BankAccount ba) {
```

```
        Thread t = new Thread(() -> {
            while (true) {
                if (ba.balance() < 100) {
                    System.out.println("Balance dipped to " + ba.balance());
                    System.exit(0);
                }
            }
        });
        t.start();
    }
}
```

30. Create another copy of the `BankAccount` class and name it `Account`. Use thread synchronization to guard the access to the `balance` instance variable in the `Account` class, so its value never goes below 100. Run the same number of threads as in the previous exercise for five minutes. This time, the monitor thread should not print any message. After five minutes, all your threads should be interrupted and your threads should respond to the interruption by finishing its task. This way, your application should exit normally after five minutes.

Solution:

```
// Account.java
package com.jdojo.generics.exercises;

import java.util.Random;

public class Account {
    private long balance = 100;

    public synchronized void deposit(long amt) throws InterruptedException {
        balance += amt;
    }

    public synchronized void withdraw(long amt)
            throws InterruptedException, IllegalStateException {
        while (balance - amt < 100) {
            throw new IllegalStateException("Withdrawal of " + amt
                    + " rejected. Balanace was " + this.balance);
        }

        balance -= amt;
    }

    public synchronized long balance() {
        return balance;
    }

    public static void main(String[] args) throws InterruptedException {
        final Account account = new Account();
```

```
        System.out.println("Initial balance: " + account.balance());
        final Random random = new Random();
        Thread t1 = startDepositThread(account, random, "T1");
        Thread t2 = startWithdrawThread(account, random, "T2");
        Thread t3 = startDepositThread(account, random, "T3");
        Thread t4 = startWithdrawThread(account, random, "T4");
        Thread monitor = startMonitorThread(account);

        try {
            Thread.sleep(5 * 60 * 1000); // Sleep for 5 minutes
        } catch (InterruptedException ex) {
            System.out.println("Main thread interrupted: " + ex.getMessage());
        }

        System.out.println("Main thread woke up after 5 minutes.");
        t1.interrupt();
        t2.interrupt();
        t3.interrupt();
        t4.interrupt();
        monitor.interrupt();

        t1.join();
        t2.join();
        t3.join();
        t4.join();
        monitor.join();

        System.out.println("Normal exit.");
    }

    private static Thread startDepositThread(Account account, Random random, String name) {
        Thread t = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                try {
                    account.deposit(random.nextInt(10) + 1);
                } catch (InterruptedException ex) {
                    // was interrupted, break from loop
                    break;
                }
            }
            System.out.println("Exiting thread " + name);
        });

        t.start();

        return t;
    }

    private static Thread startWithdrawThread(Account account, Random random, String name) {
        Thread t = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                try {
                    account.withdraw(random.nextInt(10) + 1);
                } catch (IllegalStateException e) {
                    System.out.println(e.getMessage());
                } catch (InterruptedException e) {
```

```
                    break;
                }
            }
            System.out.println("Exiting thread " + name);
        });

        t.start();

        return t;
    }

    private static Thread startMonitorThread(Account account) {
        Thread t = new Thread(() -> {
            while (!Thread.currentThread().isInterrupted()) {
                if (account.balance() < 100) {
                    System.out.println("Balance dipped to " + account.balance());
                    System.exit(0);
                }
            }
            System.out.println("Exiting monitor thread");
        });
        t.start();
        return t;
    }
}
```