Multi-Thread Synchronization - CountDownLatch, CyclicBarrier, & Phaser

Bhaskar S *04/12/2014*

Overview

In this article, we will explore the use of the following multi-thread synchronization aids in Java:

- **CountDownLatch**
- **CyclicBarrier**
- **Phaser**

All these synchronization classes are defined in the Java package **java.util.concurrent**.

For our code samples, we will consider a hypothetical Debit Processing System with 3 services - an Authentication Service to verify a card number and pin, an Account Balance Service to verify if there is enough money in the account for the debit transaction, and a Fraud Service to check for fraud activity.

So without further ado, lets get started !!!

CountDownLatch

When we start-up our Debit Processing System, we can initialize and setup each of the 3 services in a serial fashion, one after the other. This can delay the start-up time. The other approach is to initialize and setup the services in parallel threads. For this we need to ensure we do not begin processing any debit transactions until all the 3 services are initialized and setup. This is a one-time synchronization that needs to happen at start-up. This is where the **CountDownLatch** synchronization mechanism comes in handy.

A **CountDownLatch** allows one **Thread** to wait for the other **Thread**s to complete some action(s).

A **CountDownLatch** is initialized with an integer count which represents the number of **Thread**s to wait for.

The **Thread** that needs to wait will invoke the **await()** method.

Each of the **Thread**s performing some action(s) will invoke the **countDown()** method to signal the completion of some action(s).

Each instance of **CountDownLatch** object is good for one-time use.

The following is the example that demonstrates the use of the synchronization mechanism **CountDownLatch**:

Listing.1

```
/*
 *
 * Name:   MyCountDownLatch
 *
 * Author: Bhaskar S
 *
 * Date:   04/05/2014
 *
 */

package com.polarsparc.java.synchronization;

import java.util.concurrent.CountDownLatch;

public class MyCountDownLatch {
    public static void main(String[] args) {
        final int COUNT = 3; // 3 Threads

        CountDownLatch latch = new CountDownLatch(COUNT);

        Thread as = new Thread(new AuthenticationService(latch));
        as.setName("AuthenticationServiceThread");

        Thread bs = new Thread(new AccountBalanceService(latch));
        bs.setName("AccountBalanceServiceThread");

        Thread fs = new Thread(new FraudService(latch));
        fs.setName("FraudServiceThread");

        System.out.printf("Initialization started ...\n");

        as.start();
        bs.start();
        fs.start();

        try {
            latch.await();
        }
        catch (Exception ex) {
            ex.printStackTrace(System.err);
        }

        System.out.printf("Initialization completed !!!\n");
    }

    static class AuthenticationService implements Runnable {
```

```
        private final CountDownLatch latch;

        AuthenticationService(CountDownLatch latch) {
            this.latch = latch;
        }

        @Override
        public void run() {
            try {
                System.out.printf("Initializing authentication service ...\n");
                Thread.sleep(2000); // 2 seconds
                System.out.printf("Authentication service ready !!!\n");
                latch.countDown();
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }

    static class AccountBalanceService implements Runnable {
        private final CountDownLatch latch;

        AccountBalanceService(CountDownLatch latch) {
            this.latch = latch;
        }

        @Override
        public void run() {
            try {
                System.out.printf("Initializing account balance service ...\n");
                Thread.sleep(2000); // 2 seconds
                System.out.printf("Account balance service ready !!!\n");
                latch.countDown();
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }

    static class FraudService implements Runnable {
        private final CountDownLatch latch;

        FraudService(CountDownLatch latch) {
            this.latch = latch;
        }

        @Override
        public void run() {
            try {
                System.out.printf("Initializing fraud service ...\n");
                Thread.sleep(2000); // 2 seconds
                System.out.printf("Fraud service ready !!!\n");
                latch.countDown();
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }
}
```

Executing the program will generate the following output:

Output.1

```
Initialization started ...
Initializing authentication service ...
Initializing account balance service ...
Initializing fraud service ...
Authentication service ready !!!
Account balance service ready !!!
Fraud service ready !!!
Initialization completed !!!
```

CyclicBarrier

Now that our Debit Processing System is up and running, we will start processing debit transactions. Each debit transaction involves 3 steps - authentication, balance verification, and fraud verification. We can process these steps in a serial fashion, one after the other. This will reduce the processing throughput. The other approach is to process the 3 steps in parallel threads. For this we need to ensure that the main processing thread waits for the 3 steps to complete. Debit transaction processing is not a one-time synchronization operation. This is where the **CyclicBarrier** synchronization mechanism comes in handy.

A **CyclicBarrier** allows one or more **Thread**s to wait for the other **Thread**s in the set to arrive at a common synchronization point called the barrier point.

A **CyclicBarrier** is initialized with an integer count which represents the fixed number of **Thread**s in the set.

Each of the **Thread**s will invoke the **await()** method which will block and wait for the other **Thread**s in the set to arrive at the barrier point.

Each instance of **CyclicBarrier** object is cyclic as they can be re-used again once all the **Thread**s in the set have crossed the barrier point.

The following is the example that demonstrates the use of the synchronization mechanism **CyclicBarrier**:

Listing.2

```
/*
 *
 * Name:   MyCyclicBarrier
 *
 * Author: Bhaskar S
```

```
 *
 * Date:   04/05/2014
 *
 */

package com.polarsparc.java.synchronization;

import java.util.concurrent.CyclicBarrier;

public class MyCyclicBarrier {
    public static void main(String[] args) {
        final int COUNT = 4; // 1 main thread and 3 verify threads

        final String[] CARD_NOS = { "1234-5678-9000-1111", "1234-5678-9000-2222" };
        final String[] PINS = { "1234", "5678" };
        final double[] AMOUNTS = { 1500.99, 1249.99 };

        CyclicBarrier barrier = new CyclicBarrier(COUNT);

        for (int i = 0; i < CARD_NOS.length; i++) {
            Thread ac = new Thread(new AuthenticationCheck(barrier, CARD_NOS[i], PINS[i]));
            ac.setName("AuthenticationCheckThread");

            Thread bc = new Thread(new BalanceCheck(barrier, CARD_NOS[i], AMOUNTS[i]));
            bc.setName("BalanceCheckThread");

            Thread fc = new Thread(new FraudCheck(barrier, CARD_NOS[i], AMOUNTS[i]));
            fc.setName("FraudCheckThread");

            System.out.printf("Transaction processing started for %s\n", CARD_NOS[i]);

            barrier.reset();

            ac.start();
            bc.start();
            fc.start();

            try {
                barrier.await();
            }
            catch (Exception ex) {
            }

            System.out.printf("Transaction processing completed for %s\n", CARD_NOS[i]);
        }
    }

    static class AuthenticationCheck implements Runnable {
        private final String cardNo;
        private final String pin;
        private final CyclicBarrier barrier;

        AuthenticationCheck(CyclicBarrier barrier, String cardNo, String pin) {
            this.barrier = barrier;
            this.cardNo = cardNo;
            this.pin = pin;
        }

        @Override
        public void run() {
            try {
                System.out.printf("Ready to perform authentication check for %s\n", cardNo);
                Thread.sleep(3000); // 3 seconds
                System.out.printf("Completed authentication check for %s\n", cardNo);
                barrier.await();
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }

    static class BalanceCheck implements Runnable {
        private final String cardNo;
        private final double amount;
        private final CyclicBarrier barrier;

        BalanceCheck(CyclicBarrier barrier, String cardNo, double amount) {
            this.barrier = barrier;
            this.cardNo = cardNo;
            this.amount = amount;
        }

        @Override
        public void run() {
            try {
                System.out.printf("Ready to perform balance check on %s for amount %.02f\n", cardNo, amount);
                Thread.sleep(2000); // 2 seconds
                System.out.printf("Completed balance check for %s for amount %.02f\n", cardNo, amount);
                barrier.await();
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }

    static class FraudCheck implements Runnable {
        private final String cardNo;
        private final double amount;
        private final CyclicBarrier barrier;

        FraudCheck(CyclicBarrier barrier, String cardNo, double amount) {
            this.barrier = barrier;
            this.cardNo = cardNo;
            this.amount = amount;
        }
```

```
        @Override
        public void run() {
            try {
                System.out.printf("Ready to perform fraud check on %s for amount %.02f\n", cardNo, amount);
                Thread.sleep(1000); // 1 second
                System.out.printf("Completed fraud check on %s for amount %.02f\n", cardNo, amount);
                barrier.await();
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }
}
```

Executing the program will generate the following output:

Output.2

```
Transaction processing started for 1234-5678-9000-1111
Ready to perform authentication check for 1234-5678-9000-1111
Ready to perform fraud check on 1234-5678-9000-1111 for amount 1500.99
Ready to perform balance check on 1234-5678-9000-1111 for amount 1500.99
Completed fraud check on 1234-5678-9000-1111 for amount 1500.99
Completed balance check for 1234-5678-9000-1111 for amount 1500.99
Completed authentication check for 1234-5678-9000-1111
Transaction processing completed for 1234-5678-9000-1111
Transaction processing started for 1234-5678-9000-2222
Ready to perform authentication check for 1234-5678-9000-2222
Ready to perform balance check on 1234-5678-9000-2222 for amount 1249.99
Ready to perform fraud check on 1234-5678-9000-2222 for amount 1249.99
Completed fraud check on 1234-5678-9000-2222 for amount 1249.99
Completed balance check for 1234-5678-9000-2222 for amount 1249.99
Completed authentication check for 1234-5678-9000-2222
Transaction processing completed for 1234-5678-9000-2222
```

Phaser

A **Phaser** is similar to a **CyclicBarrier** except that it is more flexible and dynamic in terms of usage. With either the **CountDownLatch** or the **CyclicBarrier** one needs to specify the count of threads involved in the synchronization. With the **Phaser**, the number of threads can be dynamic and vary with time.

A **Phaser** allows one or more **Thread**s to wait for the other **Thread**s in the set to arrive at a common synchronization point called a phase.

To participate in the synchronization, a **Thread** (also referred to as a party) has to invoke the **register()** method on an instance of **Phaser**.

A **Thread** will invoke the **arriveAndAwaitAdvance()** method to block and wait for the other **Thread**s (or parties) in the set to arrive at the barrier point.

A **Thread** will invoke the **arriveAndDeregister()** method to signal that it has arrived at the barrier point (without waiting for others) and also deregister from an instance of the **Phaser**.

Like the **CyclicBarrier**, an instance of the **Phaser** object can be re-used again once all the **Thread**s in the set have crossed the barrier point.

Invoking the method **getPhase()**, returns the current phase.

Invoking the method **getRegisteredParties()**, returns the current count of **Thread**s (or parties) registered with the **Phaser**.

The following is the example that demonstrates the use of the synchronization mechanism **Phaser**:

Listing.3

```
/*
 *
 * Name:   MyPhaser
 *
 * Author: Bhaskar S
 *
 * Date:   04/05/2014
 *
 */

package com.polarsparc.java.synchronization;

import java.util.concurrent.Phaser;

public class MyPhaser {
    public static void main(String[] args) {
        final String[] CARD_NOS = { "1234-5678-9000-1111", "1234-5678-9000-2222" };
        final String[] PINS = { "1234", "5678" };
        final double[] AMOUNTS = { 1500.99, 1249.99 };

        Phaser phaser = new Phaser();

        for (int i = 0; i < CARD_NOS.length; i++) {
            System.out.printf("Current phase %d, Registered threads %d (begin)\n", phaser.getPhase(), phaser.getRegisteredParties());

            phaser.register(); // For main thread

            System.out.printf("Current phase %d, Registered threads %d (main)\n", phaser.getPhase(), phaser.getRegisteredParties());

            Thread ac = new Thread(new AuthenticationCheck(phaser, CARD_NOS[i], PINS[i]));
            ac.setName("AuthenticationCheckThread");

            Thread bc = new Thread(new BalanceCheck(phaser, CARD_NOS[i], AMOUNTS[i]));
            bc.setName("BalanceCheckThread");

            Thread fc = new Thread(new FraudCheck(phaser, CARD_NOS[i], AMOUNTS[i]));
            fc.setName("FraudCheckThread");

            System.out.printf("Transaction processing started for %s\n", CARD_NOS[i]);
```

```
            ac.start();
            bc.start();
            fc.start();

            phaser.arriveAndAwaitAdvance();

            System.out.printf("Transaction processing completed for %s\n", CARD_NOS[i]);

            System.out.printf("Current phase %d, Registered threads %d (end)\n", phaser.getPhase(), phaser.getRegisteredParties());
        }
    }

    static class AuthenticationCheck implements Runnable {
        private final String cardNo;
        private final String pin;
        private final Phaser phaser;

        AuthenticationCheck(Phaser phaser, String cardNo, String pin) {
            this.phaser = phaser;
            this.cardNo = cardNo;
            this.pin = pin;
        }

        @Override
        public void run() {
            try {
                phaser.register();
                System.out.printf("Current phase %d, Registered threads %d (ac-begin)\n", phaser.getPhase(), phaser.getRegisteredParties());
                System.out.printf("[Phase: %d] Ready to perform authentication check for %s\n", phaser.getPhase(), cardNo);
                Thread.sleep(3000); // 3 seconds
                System.out.printf("[Phase: %d] Completed authentication check for %s\n", phaser.getPhase(), cardNo);
                phaser.arriveAndDeregister();
                System.out.printf("Current phase %d, Registered threads %d (ac-end)\n", phaser.getPhase(), phaser.getRegisteredParties());
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }

    static class BalanceCheck implements Runnable {
        private final String cardNo;
        private final double amount;
        private final Phaser phaser;

        BalanceCheck(Phaser phaser, String cardNo, double amount) {
            this.phaser = phaser;
            this.cardNo = cardNo;
            this.amount = amount;
        }

        @Override
        public void run() {
            try {
                phaser.register();
                System.out.printf("Current phase %d, Registered threads %d (bc-begin)\n", phaser.getPhase(), phaser.getRegisteredParties());
                System.out.printf("[Phase: %d] Ready to perform balance check on %s for amount %.02f\n", phaser.getPhase(), cardNo, amount);
                Thread.sleep(2000); // 2 seconds
                System.out.printf("[Phase: %d] Completed balance check for %s for amount %.02f\n", phaser.getPhase(), cardNo, amount);
                phaser.arriveAndDeregister();
                System.out.printf("Current phase %d, Registered threads %d (bc-end)\n", phaser.getPhase(), phaser.getRegisteredParties());
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }

    static class FraudCheck implements Runnable {
        private final String cardNo;
        private final double amount;
        private final Phaser phaser;

        FraudCheck(Phaser phaser, String cardNo, double amount) {
            this.phaser = phaser;
            this.cardNo = cardNo;
            this.amount = amount;
        }

        @Override
        public void run() {
            try {
                phaser.register();
                System.out.printf("Current phase %d, Registered threads %d (fc-begin)\n", phaser.getPhase(), phaser.getRegisteredParties());
                System.out.printf("[Phase: %d] Ready to perform fraud check on %s for amount %.02f\n", phaser.getPhase(), cardNo, amount);
                Thread.sleep(1000); // 1 second
                System.out.printf("[Phase: %d] Completed fraud check on %s for amount %.02f\n", phaser.getPhase(), cardNo, amount);
                phaser.arriveAndDeregister();
                System.out.printf("Current phase %d, Registered threads %d (fc-end)\n", phaser.getPhase(), phaser.getRegisteredParties());
            }
            catch (Exception ex) {
                ex.printStackTrace(System.err);
            }
        }
    }
}
```

Executing the program will generate the following output:

Output.3

```
Current phase 0, Registered threads 0 (begin)
Current phase 0, Registered threads 1 (main)
Transaction processing started for 1234-5678-9000-1111
Current phase 0, Registered threads 2 (ac-begin)
[Phase: 0] Ready to perform authentication check for 1234-5678-9000-1111
```

```
Current phase 0, Registered threads 3 (bc-begin)
[Phase: 0] Ready to perform balance check on 1234-5678-9000-1111 for amount 1500.99
Current phase 0, Registered threads 4 (fc-begin)
[Phase: 0] Ready to perform fraud check on 1234-5678-9000-1111 for amount 1500.99
[Phase: 0] Completed fraud check on 1234-5678-9000-1111 for amount 1500.99
Current phase 0, Registered threads 3 (fc-end)
[Phase: 0] Completed balance check for 1234-5678-9000-1111 for amount 1500.99
Current phase 0, Registered threads 2 (bc-end)
[Phase: 0] Completed authentication check for 1234-5678-9000-1111
Current phase 1, Registered threads 1 (ac-end)
Transaction processing completed for 1234-5678-9000-1111
Current phase 1, Registered threads 1 (end)
Current phase 1, Registered threads 1 (begin)
Current phase 1, Registered threads 2 (main)
Transaction processing started for 1234-5678-9000-2222
Current phase 1, Registered threads 3 (bc-begin)
[Phase: 1] Ready to perform balance check on 1234-5678-9000-2222 for amount 1249.99
Current phase 1, Registered threads 5 (fc-begin)
[Phase: 1] Ready to perform fraud check on 1234-5678-9000-2222 for amount 1249.99
Current phase 1, Registered threads 4 (ac-begin)
[Phase: 1] Ready to perform authentication check for 1234-5678-9000-2222
[Phase: 1] Completed fraud check on 1234-5678-9000-2222 for amount 1249.99
Current phase 1, Registered threads 4 (fc-end)
[Phase: 1] Completed balance check for 1234-5678-9000-2222 for amount 1249.99
Current phase 1, Registered threads 3 (bc-end)
[Phase: 1] Completed authentication check for 1234-5678-9000-2222
Current phase 1, Registered threads 2 (ac-end)
```