String Rendezvous Project

1. Complete the class **`StringHandoffImpl`** to fully implement the behavior described in the **`StringHandoff`** interface.
2. There are two projects in the zip file: "PROJ_ThreadRendezvous" *and* "Parallel-Common-PROJ-ThreadRendezvous". Import them both from the zip archive.
    a. `StringHandoff.java` and most of the tests are located in "Parallel-Common-PROJ-ThreadRendezvous". The Javadoc comments inside `StringHandoff` explain in detail the desired behavior—read this fully.
    b. `StringHandoffImpl.java`, `TestUltraBasic.java`, and `TestStringHandoff.java` are located in "PROJ_ThreadRendezvous".
3. *Only* modify the **`StringHandoffImpl.java`** file (and this is the only file considered when grading [you can add nested classes if you'd like to, but they must be inside `StringHandoffImpl.java`]).
4. Plan of attack:
    a. Before considering *anything* about timeouts, multiple passers, multiple receivers, shutdown, or any of the tests marked as advanced, design and code the most basic functionality. You can enhance your code later when you've got the core part working. This is definitely the best approach.
    b. Every time you get a new chunk of code working, make a zip of your source folder, name the zip file something unique, and save it somewhere safe. This way you can always go back to it no matter how much your next steps accidentally mangle things.
    c. You should not need special classes beyond what you've seen in class. You *cannot* use "wait helper" utility for the project, but instead must write all the wait-notify code manually.
    d. the starting code for `StringHandoffImpl.java` has methods marked as `synchronized`—keep it that way. We're *not* using `synchronized` blocks for the project.
    e. Have the methods that don't take a timeout call the ones that do passing in a timeout of **`0L`** (zero) and keep these methods this way through your final solution:

```
@Override
public synchronized void pass(String msg)
        throws InterruptedException, ShutdownException, IllegalStateException {
    pass(msg, 0L);
}

@Override
public synchronized String receive()
        throws InterruptedException, ShutdownException, IllegalStateException {
    return receive(0L);
}
```

f. ***For now***, the methods that do take a timeout should be written to check for zero and immediately throw a `RuntimeException` if a non-zero timeout is passed in. We'll fix this later, but it allows us to create compilable and runnable code that can get the basic functionality working without any complexity trying to deal with timeouts:

```
@Override
public synchronized void pass(String msg, long msTimeout)
    throws InterruptedException, TimedOutException, ShutdownException, IllegalStateException {

    if (msTimeout != 0L) throw new RuntimeException("non-zero timeout is not YET supported");
    //...
}

@Override
public synchronized String receive(long msTimeout)
    throws InterruptedException, TimedOutException, ShutdownException, IllegalStateException {

    if (msTimeout != 0L) throw new RuntimeException("non-zero timeout is not YET supported");
    //...
}
```

g. Run **`TestUltraBasic`** and get your code to work with it before moving on to the large test suite. Until you get this working, there's no point in adding any complexity.

h. Run **`TestStringHandoff`** and notice the suite of tests which run in parallel. They run against separate instances of your `StringHandoffImpl` class, so the parallel testing does not cause any interference.

   i. In general, attack the tests in the order they are written.

   ii. The "Points" column shows how many points your code is scoring for passing tests. Ignore the "Valid" column *(I need that for my debugging of your submissions)*

      1. The "Total Points" box near the top is out of 100, but this is just a starting score value and I look at your source code for other potential issues and subtract from there (but no matter what, I won't add deductions to bring your score below 90).

   iii. Some of the "interrupt check" tests won't pass until non-zero timeouts are actually supported.

   iv. Get "single item pass and receive" to work first. This is worth a lot of points and the rest won't matter unless this works.

   v. Next, get "multiple item pass and receive" to work. At this point, definitely zip up your src folder and put that zip file someplace safe.

   vi. Then add support for timeouts and remove the throwing of a `RuntimeException` we added earlier and confirm that all of the "interrupt check" tests now pass.

   vii. Getting everything working to this point puts your starting score at 84. This is all before you have even begun to consider multiple passers, multiple receivers, shutdown, or any of the tests marked as advanced!

This is another time to zip up your src folder and put that zip file someplace safe.

viii. Next attack multiple passers and multiple receivers.

ix. After that, attack shutdown functionality.

x. When all that is working, you'll be up to a starting score of 96. Definitely create a zip of your src folder and store it safely.

xi. Try the advanced tests one at a time and remember that they may be really difficult to implement and/or may require some significant design changes.

5. Here's the Javadoc generated from the interface description of `StringHandoff.java`. Read it all carefully, it fully explains how your code needs to work:

`public interface` **`StringHandoff`**

Rendezvous

`StringHandoff` is used to pass a `String` from one thread to another. The passer and the receiver meet inside an instance for the hand-off. For example, if `pass()` is called and there is no receiver waiting, the thread calling `pass()` will block until a receiver arrives. Similarly, if `receive()` is called and there is no passer waiting, the thread calling `receive()` will block until a passer arrives.

Timeouts Cause a `TimedOutException` to be Thrown

For this class, methods that can timeout do not return `true` for success and `false` for timeout. Methods that take a timeout parameter (`pass(String, long)` or `receive(long)`) will throw a **`TimedOutException`** if the specified number of milliseconds passes without the hand-off occurring.

If the hand-off has completed—regardless of how long it took—no `TimedOutException` should be thrown.

A timeout value of `0L` (zero) is used to indicate that the calling thread should wait indefinitely for the condition to be met (never timeout).

Negative timeouts should be tolerated and be treated as a very short timeout.

`TimedOutException` is a `RuntimeException`.

One Passer at a Time, One Receiver at a Time

There can only be one thread waiting to pass at any given time. If threadA is waiting inside `pass()` and threadB calls `pass()`, threadB must throw an **`IllegalStateException`**. threadB throwing an exception should not affect threadA at all (threadA should continue waiting).

Similarly, there can only be one thread waiting to receive at any given time. If threadA is waiting inside `receive()` and threadB calls `receive()`, threadB must throw an

`IllegalStateException`. threadB throwing an exception should not affect threadA at all (threadA should continue waiting).

`IllegalStateException` is a `RuntimeException`.

Shutdown

The methods that declare that they may throw a **`ShutdownException`** will do so after `shutdown()` has been called. The `shutdown()` method itself does not ever throw `ShutdownException` (in fact, it doesn't ever throw any kind of exception). The `shutdown()` method *may* be called multiple times and calling `shutdown()` multiple times must be harmless.

After threadA has already called `shutdown()`, if threadB calls `pass(String)`, `pass(String, long)`, `receive()`, or `receive(long)` threadB must immediately throw `ShutdownException`.

If threadA is waiting inside `pass(String)`, `pass(String, long)`, `receive()`, or `receive(long)` and threadB calls `shutdown()`, threadA should stop waiting and throw `ShutdownException`.

ShutdownException is a RuntimeException.

Exception Precedence

Exception precedence from highest to lowest is listed below:

1. `InterruptedException`
2. `ShutdownException`
3. `IllegalStateException`
4. `TimedOutException`

Example 1: while threadA is waiting, if both of these happen before threadA can re-acquire the lock to get back from `wait`: i) threadB calls `shutdown()` and then ii) threadC interrupts threadA, the result should be that threadA throws `InterruptedException`.

Example 2: while threadA is waiting with a timeout, if both of these happen before threadA can re-acquire the lock to get back from `wait`: i) threadB calls `shutdown()` and then ii) threadA times out, the result should be that threadA throws `ShutdownException`.

Example 3: while threadA is waiting inside `pass()`, if both of these happen before threadA can re-acquire the lock to get back from `wait`: i) threadB calls `shutdown()` and then ii) threadC calls `pass()`, the result should be that threadC should throw `ShutdownException<` (*not* `IllegalStateException`!) AND threadA should throw `ShutdownException`.