

Implementing the AsyncTaskBarrier Framework Using RxJava (Part 2)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



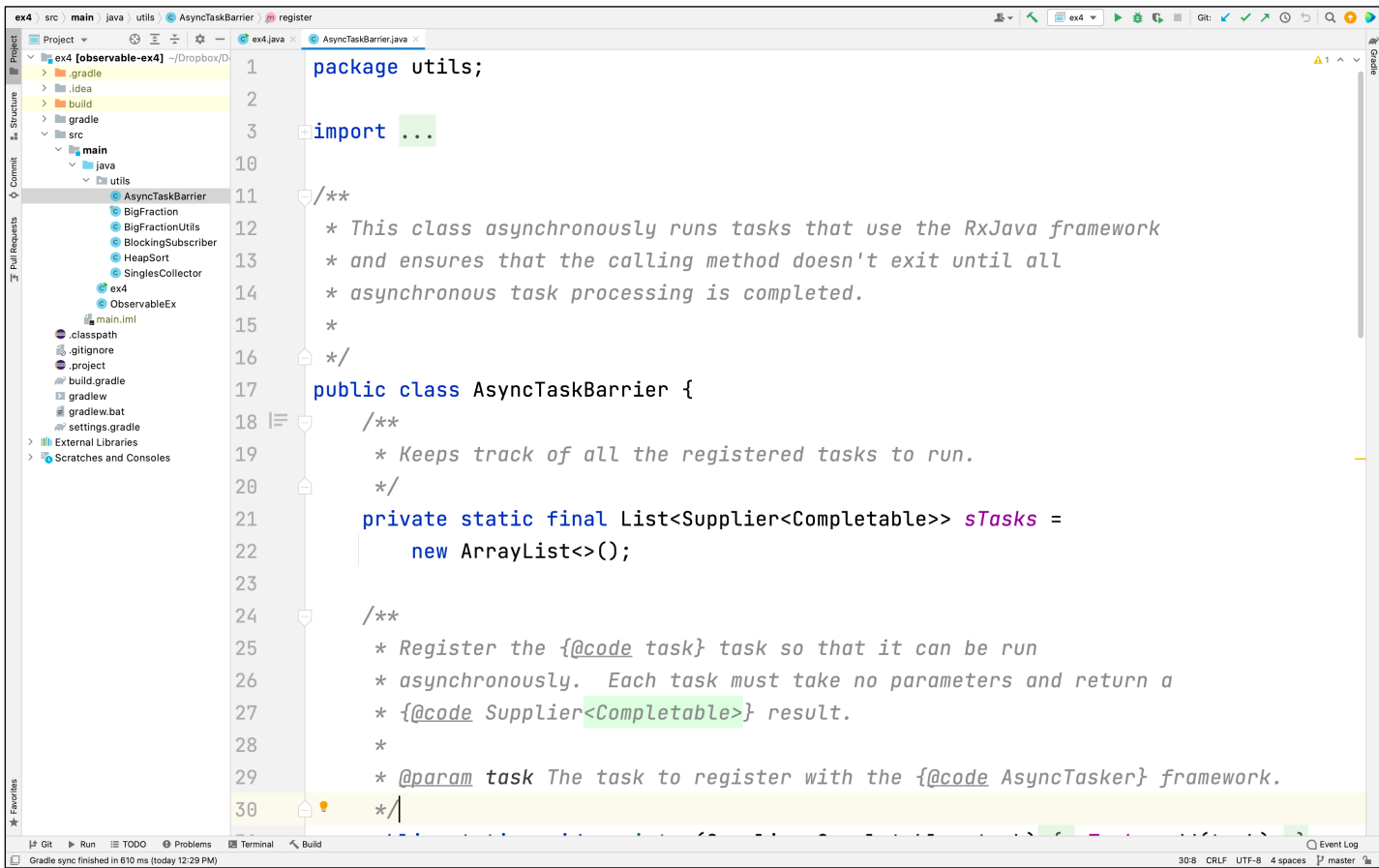
Learning Objectives in this Part of the Lesson

- Understand the API of the Async TaskBarrier class for RxJava
- Know how to use AsyncTaskBarrier in practice
- Recognize how RxJava operators are used to implement the Async TaskBarrier framework

```
static Single<Long> runTests() {  
    ...  
    return Observable  
        .fromIterable(sTests)  
  
        .map(s -> s.get()  
            .onErrorResumeNext  
                (errorHandler))  
  
        .flatMapCompletable(c -> c)  
  
        .toSingle(() ->  
            sTests.size()  
            - exceptionCount.get()));  
}
```

Implementing the Async TaskBarrier Framework

Implementing the AsyncTaskBarrier Framework



```
1 package utils;
2
3 import ...
4
5
6
7
8
9
10
11 /**
12  * This class asynchronously runs tasks that use the RxJava framework
13  * and ensures that the calling method doesn't exit until all
14  * asynchronous task processing is completed.
15  *
16  */
17 public class AsyncTaskBarrier {
18     /**
19      * Keeps track of all the registered tasks to run.
20      */
21     private static final List<Supplier<Completable>> sTasks =
22         new ArrayList<>();
23
24     /**
25      * Register the {@code task} task so that it can be run
26      * asynchronously. Each task must take no parameters and return a
27      * {@code Supplier<Completable>} result.
28      *
29      * @param task The task to register with the {@code AsyncTaskBarrier} framework.
30      */
```

The screenshot shows an IDE window with the following details:

- Project: ex4 [observable-ex4] - ~/Dropbox/D
- Structure: ex4 > gradle > build > gradle > src > main > java > utils > AsyncTaskBarrier
- Code: The Java code for AsyncTaskBarrier.java is displayed, including package, imports, a class-level comment, and the class definition with a static list of tasks.
- Status Bar: 30:8 CRLF UTF-8 4 spaces master
- Bottom Panel: Run, TODO, Problems, Terminal, Build. A message indicates "Gradle sync finished in 610 ms (today 12:29 PM)".

See [Reactive/Observable/ex4/src/main/java/utils/AsyncTaskBarrier.java](#)

Implementing the AsyncTaskBarrier Framework

- The sTasks field keeps track of all the registered tasks to run

```
public class AsyncTaskBarrier {  
    private static final  
        List<Supplier<Completable>>  
        sTasks = new ArrayList<>();  
    ...  
}
```

Implementing the AsyncTaskBarrier Framework

- The register() & unregister() methods simply add & remove registered tasks to an internal list, respectively

```
static void register
    (Supplier<Completable> task) {
    sTasks.add(task);
}
```

```
static boolean unregister
    (Supplier<Completable> task) {
    return sTasks.remove(task);
}
```

Implementing the AsyncTaskBarrier Framework

- The register() & unregister() methods simply add & remove registered tasks to an internal list, respectively
- Each task is a Supplier whose get() method performs a task that returns Completable

```
static void register  
    (Supplier<Completable> task) {  
    sTasks.add(task);  
}
```

```
static boolean unregister  
    (Supplier<Completable> task) {  
    return sTasks.remove(task);  
}
```

Implementing the AsyncTaskBarrier Framework

- The register() & unregister() methods simply add & remove registered tasks to an internal list, respectively
- Each task is a Supplier whose get() method performs a task that returns Completable
 - This return type is used to signal when a task completes

```
static void register
    (Supplier<Completable> task) {
    sTasks.add(task);
}
```

```
static boolean unregister
    (Supplier<Completable> task) {
    return sTasks.remove(task);
}
```


Implementing the AsyncTaskBarrier Framework

- The register() & unregister() methods simply add & remove registered tasks to an internal list, respectively
- Each task is a Supplier whose get() method performs a task that returns Completable
 - This return type is used to signal when a task completes
- The method implementations simply add & remove tasks from the List

```
static void register
    (Supplier<Completable> task) {
    sTasks.add(task);
}
```

```
static boolean unregister
    (Supplier<Completable> task) {
    return sTasks.remove(task);
}
```

Implementing the AsyncTaskBarrier Framework

- The runTasks() method runs all the registered tasks

```
static Single<Long> runTests () {  
    ...  
    return Observable  
        .fromIterable(sTests)  
  
        .map(s -> s.get()  
            .onErrorResumeNext  
                (errorHandler))  
  
        .flatMapCompletable(c -> c)  
  
        .toSingle(() ->  
            sTests.size()  
            - exceptionCount.get()));  
}
```

Implementing the AsyncTaskBarrier Framework

- The runTasks() method runs all the registered tasks
- It returns a Single<Long> that triggers when all tasks complete

Emits the # of tasks that completed successfully

```
static Single<Long> runTests() {  
    ...  
    return Observable  
        .fromIterable(sTests)  
  
        .map(s -> s.get()  
            .onErrorResumeNext  
                (errorHandler))  
  
        .flatMapCompletable(c -> c)  
  
        .toSingle(() ->  
            sTests.size()  
            - exceptionCount.get()));  
}
```

Implementing the AsyncTaskBarrier Framework

- The runTasks() method runs all the registered tasks
- It returns a Single<Long> that triggers when all tasks complete

```
static Single<Long> runTests() {  
    ...  
    return Observable  
        .fromIterable(sTests)  
  
        .map(s -> s.get())  
        .onErrorResumeNext  
            (errorHandler)  
  
        .flatMapCompletable(c -> c)  
  
        .toSingle(() ->  
            sTests.size()  
            - exceptionCount.get()));  
}
```

*Factory method that converts
the list of suppliers into an
Observable stream of suppliers*

Implementing the AsyncTaskBarrier Framework

- The runTasks() method runs all the registered tasks
- It returns a Single<Long> that triggers when all tasks complete

```
static Single<Long> runTests() {  
    ...  
    return Observable  
        .fromIterable(sTests)  
  
        .map(s -> s.get())  
        .onErrorResumeNext  
            (errorHandler)  
  
        .flatMapCompletable(c -> c)  
  
        .toSingle(() ->  
            sTests.size()  
            - exceptionCount.get()));  
}
```

Run all registered tasks, which can execute asynchronously & each return a Completable

Implementing the AsyncTaskBarrier Framework

- The runTasks() method runs all the registered tasks
- It returns a Single<Long> that triggers when all tasks complete

Swallow any exception that is thrown after first recording it

```
Function<Throwable, Completable>
errorHandler = t -> {
    exceptionCount.getAndIncrement();
    return Completable.complete();
};
```

```
static Single<Long> runTests() {
    ...
    return Observable
        .fromIterable(sTests)

        .map(s -> s.get())
        .onErrorResumeNext
            (errorHandler))

        .flatMapCompletable(c -> c)

        .toSingle(() ->
            sTests.size()
            - exceptionCount.get()));
}
```

Implementing the AsyncTaskBarrier Framework

- The runTasks() method runs all the registered tasks
- It returns a Single<Long> that triggers when all tasks complete

Map each element of the Observable into a CompletableSource, subscribe to them, wait until the upstream & all CompletableSources complete, & then return a single Completable

```
static Single<Long> runTests() {  
    ...  
    return Observable  
        .fromIterable(sTests)  
  
        .map(s -> s.get())  
        .onErrorResumeNext  
            (errorHandler)  
  
        .flatMapCompletable(c -> c)  
  
        .toSingle(() ->  
            sTests.size()  
            - exceptionCount.get()));  
}
```

Implementing the AsyncTaskBarrier Framework

- The runTasks() method runs all the registered tasks
- It returns a Single<Long> that triggers when all tasks complete

```
static Single<Long> runTests() {  
    ...  
    return Observable  
        .fromIterable(sTests)  
  
        .map(s -> s.get())  
        .onErrorResumeNext  
            (errorHandler)  
  
        .flatMapCompletable(c -> c)  
  
        .toSingle(() ->  
            sTests.size()  
            - exceptionCount.get()));  
}
```

Convert the returned Completable into a Single that returns the # of tasks that completed successfully

End of Implementing the AsyncBarrierTask Framework Using RxJava (Part 2)