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

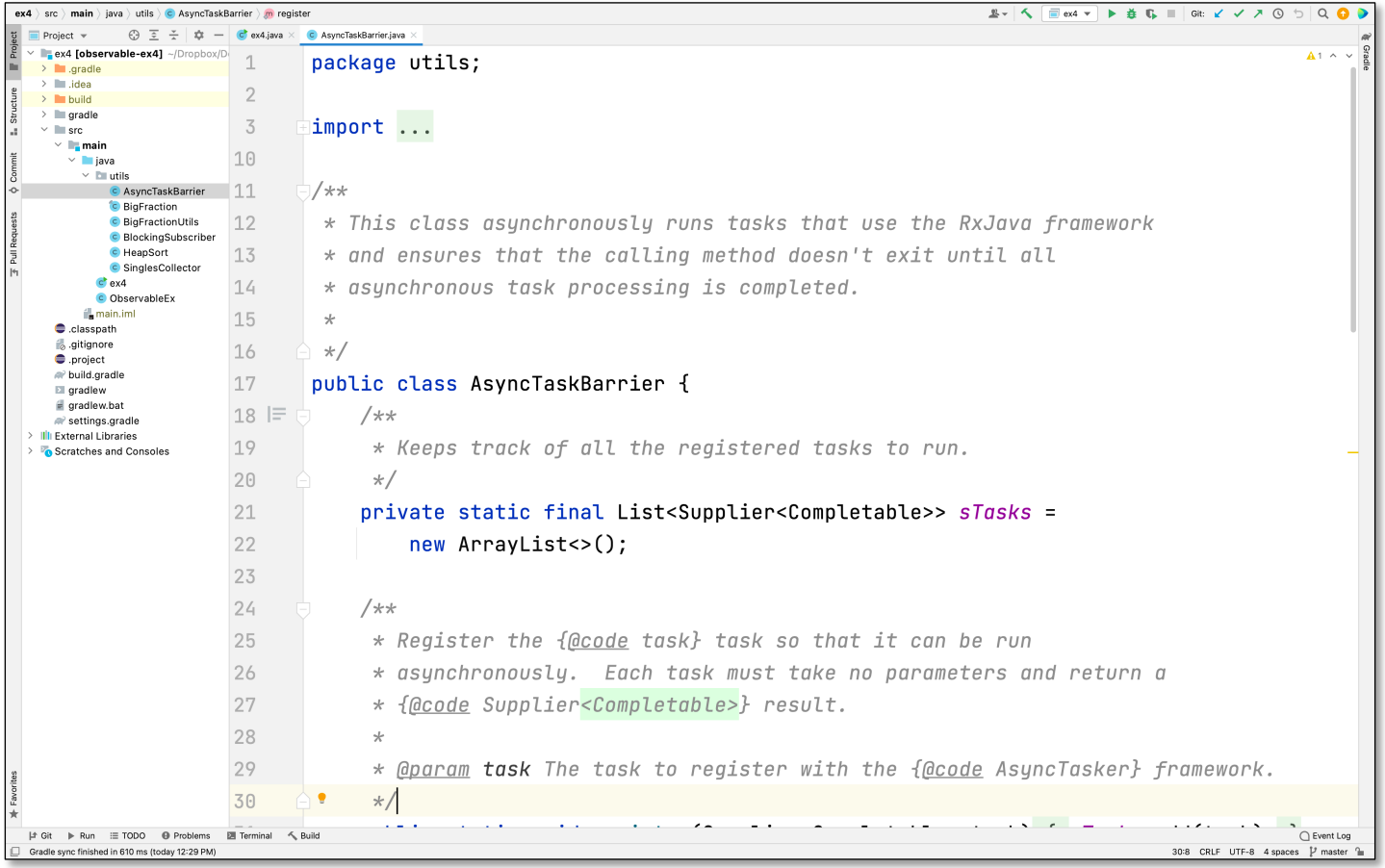
- Understand the API of the Async TaskBarrier class for RxJava
- Know how to use AsyncTaskBarrier in practice
- Recognize how RxJava operators are used to implement the Async TaskBarrier framework

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

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

    .flatMapCompletable(c -> c)

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

See Reactive/Observable/ex4/src/main/java/utils/AsyncTaskBarrier.java

Implementing the Async TaskBarrier Framework

Implementing the AsyncTaskBarrier Framework



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

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#fromIterable

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

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#map

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()));
}
```

Swallow any exception that is thrown after first recording it

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

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#map

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()));
}
```

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Observable.html#flatMapCompletable

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

See reactivex.io/RxJava/3.x/javadoc/io/reactivex/rxjava3/core/Completable.html#toSingleDefault

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