Plugins allow you to modify the default behavior of RxJava in several respects:

- by changing the set of default computation, i/o, and new thread Schedulers
- by registering a handler for extraordinary errors that RxJava may encounter
- by registering functions that can take note of the occurrence of several regular RxJava activities

As of 1.1.7 the regular `RxJavaPlugins` and the other hook classes have been deprecated in favor of `RxJavaHooks`.

# RxJavaHooks

The new `RxJavaHooks` allows you to hook into the lifecycle of the `Observable`, `Single` and `Completable` types, the `Scheduler`s returned by `Schedulers` and offers a catch-all for undeliverable errors.

You can now change these hooks at runtime and there is no need to prepare hooks via system parameters anymore. Since users may still rely on the old hooking system, RxJavaHooks delegates to those old hooks by default.

The `RxJavaHooks` has setters and getters of the various hook types:

| Hook | Description |
|---|---|
| onError : `Action1<Throwable>` | Sets the catch-all callback |
| onObservableCreate : `Func1<Observable.OnSubscribe, Observable.OnSubscribe>` | Called when operators and sources are instantiated on `Observable` |
| onObservableStart : `Func2<Observable, Observable.OnSubscribe, Observable.OnSubscribe>` | Called before subscribing to an `Observable` actually happens |
| onObservableSubscribeError : `Func1<Throwable, Throwable>` | Called when subscribing to an `Observable` fails |
| onObservableReturn : `Func1<Subscription, Subscription>` | Called when the subscribing to an `Observable` succeeds and before returning the `Subscription` handler for it |
| onObservableLift : `Func1<Observable.Operator, Observable.Operator>` | Called when the operator `lift` is used with `Observable` |
| onSingleCreate : `Func1<Single.OnSubscribe, Single.OnSubscribe>` | Called when operators and sources are instantiated on `Single` |
| onSingleStart : `Func2<Single, Observable.OnSubscribe, Observable.OnSubscribe>` | Called before subscribing to a `Single` actually happens |
| onSingleSubscribeError : `Func1<Throwable, Throwable>` | Called when subscribing to a `Single` fails |
| onSingleReturn : `Func1<Subscription, Subscription>` | Called when the subscribing to a `Single` succeeds and before returning the `Subscription` handler for it |
| onSingleLift : `Func1<Observable.Operator,` | Called when the operator `lift` is used (note: |

| | |
|---|---|
| `Observable.Operator>` | `Observable.Operator` is deliberate here) |
| onCompletableCreate : `Func1<Completable.OnSubscribe, Completable.OnSubscribe>` | Called when operators and sources are instantiated on `Completable` |
| onCompletableStart : `Func2<Completable, Completable.OnSubscribe, Completable.OnSubscribe>` | Called before subscribing to a `Completable` actually happens |
| onCompletableSubscribeError : `Func1<Throwable, Throwable>` | Called when subscribing to a `Completable` fails |
| onCompletableLift : `Func1<Completable.Operator, Completable.Operator>` | Called when the operator `lift` is used with `Completable` |
| onComputationScheduler : `Func1<Scheduler, Scheduler>` | Called when using `Schedulers.computation()` |
| onIOScheduler : `Func1<Scheduler, Scheduler>` | Called when using `Schedulers.io()` |
| onNewThreadScheduler : `Func1<Scheduler, Scheduler>` | Called when using `Schedulers.newThread()` |
| onScheduleAction : `Func1<Action0, Action0>` | Called when a task gets scheduled in any of the `Scheduler`**S** |
| onGenericScheduledExecutorService : `Func0<ScheduledExecutorService>` | that should return single-threaded executors to support background timed tasks of RxJava itself |

Reading and changing these hooks is thread-safe.

You can also clear all hooks via `clear()` or reset to the default behavior (of delegating to the old RxJavaPlugins system) via `reset()`.

Example:

```
RxJavaHooks.setOnObservableCreate(o -> {
    System.out.println("Creating " + o.getClass());
    return o;
});
try {
    Observable.range(1, 10)
    .map(v -> v * 2)
    .filter(v -> v % 4 == 0)
    .subscribe(System.out::println);
} finally {
    RxJavaHooks.reset();
}
```

In addition, the `RxJavaHooks` offers the so-called assembly tracking feature. This shims a custom `Observable`, `Single` and `Completable` into their chains which captures the current stacktrace when those operators were instantiated (assembly-time). Whenever an error is signalled via onError, these middle components attach this

assembly-time stacktraces as last causes of that exception. This may help locating the problematic sequence in a codebase where there are too many similar flows and the plain exception itself doesn't tell which one failed in your codebase.

Example:

```
RxJavaHooks.enableAssemblyTracking();
try {
    Observable.empty().single()
    .subscribe(System.out::println, Throwable::printStackTrace);
} finally {
   RxJavaHooks.resetAssemblyTracking();
}
```

This will print something like this:

```
java.lang.NoSuchElementException
at rx.internal.operators.OnSubscribeSingle(OnSubscribeSingle.java:57)
...
Assembly trace:
at com.example.TrackingExample(TrackingExample:10)
```

The stacktrace string is also available in a field to support debugging and discovering the status of various operators in a running chain.

The stacktrace is filtered by removing irrelevant entries such as Thread entry points, unit test runners and the entries of the tracking system itself to reduce noise.

# RxJavaSchedulersHook

**Deprecated**

This plugin allows you to override the default computation, i/o, and new thread Schedulers with Schedulers of your choosing. To do this, extend the class `RxJavaSchedulersHook` and override these methods:

- `Scheduler getComputationScheduler()`
- `Scheduler getIOScheduler()`
- `Scheduler getNewThreadScheduler()`
- `Action0 onSchedule(action)`

Then follow these steps:

1. Create an object of the new `RxJavaDefaultSchedulers` subclass you have implemented.
2. Obtain the global `RxJavaPlugins` instance via `RxJavaPlugins.getInstance()` .
3. Pass your default schedulers object to the `registerSchedulersHook()` method of that instance.

When you do this, RxJava will begin to use the Schedulers returned by your methods rather than its built-in defaults.

# RxJavaErrorHandler

**Deprecated**

This plugin allows you to register a function that will handle errors that are passed to `SafeSubscriber.onError(Throwable)`. ( `SafeSubscriber` is used for wrapping the incoming `Subscriber` when one calls `subscribe()` ). To do this, extend the class `RxJavaErrorHandler` and override this method:

- `void handleError(Throwable e)`

Then follow these steps:

1. Create an object of the new `RxJavaErrorHandler` subclass you have implemented.
2. Obtain the global `RxJavaPlugins` instance via `RxJavaPlugins.getInstance()`.
3. Pass your error handler object to the `registerErrorHandler()` method of that instance.

When you do this, RxJava will begin to use your error handler to field errors that are passed to `SafeSubscriber.onError(Throwable)`.

For example, this will call the hook:

```
RxJavaPlugins.getInstance().reset();

RxJavaPlugins.getInstance().registerErrorHandler(new RxJavaErrorHandler() {
    @Override
    public void handleError(Throwable e) {
        e.printStackTrace();
    }
});

Observable.error(new IOException())
.subscribe(System.out::println, e -> { });
```

however, this call and chained operators in general won't trigger it in each stage:

```
Observable.error(new IOException())
.map(v -> "" + v)
.unsafeSubscribe(System.out::println, e -> { });
```

# RxJavaObservableExecutionHook

**Deprecated**

This plugin allows you to register functions that RxJava will call upon certain regular RxJava activities, for instance for logging or metrics-collection purposes. To do this, extend the class `RxJavaObservableExecutionHook` and override any or all of these methods:

| method | when invoked |
| --- | --- |
| `onCreate()` | during `Observable.create()` |
| `onSubscribeStart()` | immediately before `Observable.subscribe()` |
| `onSubscribeReturn()` | immediately after `Observable.subscribe()` |
| | |

| | |
|---|---|
| `onSubscribeError()` | upon a failed execution of `Observable.subscribe()` |
| `onLift()` | during `Observable.lift()` |

Then follow these steps:

1. Create an object of the new `RxJavaObservableExecutionHook` subclass you have implemented.
2. Obtain the global `RxJavaPlugins` instance via `RxJavaPlugins.getInstance()`.
3. Pass your execution hooks object to the `registerObservableExecutionHook()` method of that instance.

When you do this, RxJava will begin to call your functions when it encounters the specific conditions they were designed to take note of.