Hello World!

The following sample implementations of "Hello World" in Java, Groovy, Clojure, and Scala create an Observable from a list of Strings, and then subscribe to this Observable with a method that prints "Hello *String*!" for each string emitted by the Observable.

You can find additional code examples in the `/src/examples` folders of each language adaptor:

- RxGroovy examples
- RxClojure examples
- RxScala examples

Java

```
public static void hello(String... args) {
  Flowable.fromArray(args).subscribe(s -> System.out.println("Hello " + s + "!"));
}
```

If your platform doesn't support Java 8 lambdas (yet), you have to create an inner class of `Consumer` manually:

```
public static void hello(String... args) {
  Flowable.fromArray(args).subscribe(new Consumer<String>() {
      @Override
      public void accept(String s) {
          System.out.println("Hello " + s + "!");
      }
  });
}
```

```
hello("Ben", "George");
Hello Ben!
Hello George!
```

Groovy

```
def hello(String[] names) {
    Observable.from(names).subscribe { println "Hello ${it}!" }
}
```

```
hello("Ben", "George")
Hello Ben!
Hello George!
```

Clojure

```
(defn hello
  [&rest]
```

```
  (-> (Observable/from &rest)
    (.subscribe #(println (str "Hello " % "!")))))
```

```
(hello ["Ben" "George"])
Hello Ben!
Hello George!
```

Scala

```
import rx.lang.scala.Observable

def hello(names: String*) {
  Observable.from(names) subscribe { n =>
    println(s"Hello $n!")
  }
}
```

```
hello("Ben", "George")
Hello Ben!
Hello George!
```

How to Design Using RxJava

To use RxJava you create Observables (which emit data items), transform those Observables in various ways to get the precise data items that interest you (by using Observable operators), and then observe and react to these sequences of interesting items (by implementing Observers or Subscribers and then subscribing them to the resulting transformed Observables).

Creating Observables

To create an Observable, you can either implement the Observable's behavior manually by passing a function to `create()` that exhibits Observable behavior, or you can convert an existing data structure into an Observable by using some of the Observable operators that are designed for this purpose.

Creating an Observable from an Existing Data Structure

You use the Observable `just()` and `from()` methods to convert objects, lists, or arrays of objects into Observables that emit those objects:

```
Observable<String> o = Observable.from("a", "b", "c");
```

```
def list = [5, 6, 7, 8]
Observable<Integer> o2 = Observable.from(list);
```

```
Observable<String> o3 = Observable.just("one object");
```

These converted Observables will synchronously invoke the `onNext()` method of any subscriber that subscribes to them, for each item to be emitted by the Observable, and will then invoke the subscriber's `onCompleted()` method.

Creating an Observable via the `create()` method

You can implement asynchronous i/o, computational operations, or even "infinite" streams of data by designing your own Observable and implementing it with the `create()` method.

Synchronous Observable Example

```
/**
 * This example shows a custom Observable that blocks
 * when subscribed to (does not spawn an extra thread).
 */
def customObservableBlocking() {
    return Observable.create { aSubscriber ->
        50.times { i ->
            if (!aSubscriber.unsubscribed) {
                aSubscriber.onNext("value_${i}")
            }
        }
        // after sending all values we complete the sequence
        if (!aSubscriber.unsubscribed) {
            aSubscriber.onCompleted()
        }
    }
}

// To see output:
customObservableBlocking().subscribe { println(it) }
```

Asynchronous Observable Example The following example uses Groovy to create an Observable that emits 75 strings.

It is written verbosely, with static typing and implementation of the `Func1` anonymous inner class, to make the example more clear:

```
/**
 * This example shows a custom Observable that does not block
 * when subscribed to as it spawns a separate thread.
 */
def customObservableNonBlocking() {
    return Observable.create({ subscriber ->
        Thread.start {
            for (i in 0..<75) {
```

```
                if (subscriber.unsubscribed) {
                    return
                }
                subscriber.onNext("value_${i}")
            }
            // after sending all values we complete the sequence
            if (!subscriber.unsubscribed) {
                subscriber.onCompleted()
            }
        }
    } as Observable.OnSubscribe)
}

// To see output:
customObservableNonBlocking().subscribe { println(it) }
```

Here is the same code in Clojure that uses a Future (instead of raw thread) and is implemented more consisely:

```
(defn customObservableNonBlocking []
  "This example shows a custom Observable that does not block
   when subscribed to as it spawns a separate thread.

  returns Observable<String>"
  (Observable/create
    (fn [subscriber]
      (let [f (future
                (doseq [x (range 50)] (-> subscriber (.onNext (str "value_" x))))
                ; after sending all values we complete the sequence
                (-> subscriber .onCompleted))
        ))
      ))

; To see output
(.subscribe (customObservableNonBlocking) #(println %))
```

Here is an example that fetches articles from Wikipedia and invokes onNext with each one:

```
(defn fetchWikipediaArticleAsynchronously [wikipediaArticleNames]
  "Fetch a list of Wikipedia articles asynchronously.

   return Observable<String> of HTML"
  (Observable/create
    (fn [subscriber]
      (let [f (future
                (doseq [articleName wikipediaArticleNames]
                  (-> subscriber (.onNext (http/get (str "http://en.wikipedia.org/wiki/" art
```

```
                ; after sending response to onnext we complete the sequence
                (-> subscriber .onCompleted))
        ))))

(-> (fetchWikipediaArticleAsynchronously ["Tiger" "Elephant"])
  (.subscribe #(println "--- Article ---\n" (subs (:body %) 0 125) "...")))
```

Back to Groovy, the same Wikipedia functionality but using closures instead of anonymous inner classes:

```
/*
 * Fetch a list of Wikipedia articles asynchronously.
 */
def fetchWikipediaArticleAsynchronously(String... wikipediaArticleNames) {
    return Observable.create { subscriber ->
        Thread.start {
            for (articleName in wikipediaArticleNames) {
                if (subscriber.unsubscribed) {
                    return
                }
                subscriber.onNext(new URL("http://en.wikipedia.org/wiki/${articleName}").tex
            }
            if (!subscriber.unsubscribed) {
                subscriber.onCompleted()
            }
        }
        return subscriber
    }
}

fetchWikipediaArticleAsynchronously("Tiger", "Elephant")
    .subscribe { println "--- Article ---\n${it.substring(0, 125)}" }
```

Results:

```
--- Article ---
 <!DOCTYPE html>
<html lang="en" dir="ltr" class="client-nojs">
<head>
<title>Tiger - Wikipedia, the free encyclopedia</title> ...
--- Article ---
 <!DOCTYPE html>
<html lang="en" dir="ltr" class="client-nojs">
<head>
<title>Elephant - Wikipedia, the free encyclopedia</tit ...
```

Note that all of the above examples ignore error handling, for brevity. See below for examples that include error handling.

More information can be found on the [[Observable]] and [[Creating Observables|Creating-Observables]] pages.

Transforming Observables with Operators

RxJava allows you to chain *operators* together to transform and compose Observables.

The following example, in Groovy, uses a previously defined, asynchronous Observable that emits 75 items, skips over the first 10 of these (`skip(10)`), then takes the next 5 (`take(5)`), and transforms them (`map(...)`) before subscribing and printing the items:

```
/**
 * Asynchronously calls 'customObservableNonBlocking' and defines
 * a chain of operators to apply to the callback sequence.
 */
def simpleComposition() {
    customObservableNonBlocking().skip(10).take(5)
        .map({ stringValue -> return stringValue + "_xform"})
        .subscribe({ println "onNext => " + it})
}
```

This results in:

```
onNext => value_10_xform
onNext => value_11_xform
onNext => value_12_xform
onNext => value_13_xform
onNext => value_14_xform
```

Here is a marble diagram that illustrates this transformation:

This next example, in Clojure, consumes three asynchronous Observables, including a dependency from one to another, and emits a single response item by combining the items emitted by each of the three Observables with the `zip` operator and then transforming the result with `map`:

```
(defn getVideoForUser [userId videoId]
  "Get video metadata for a given userId
   - video metadata
   - video bookmark position
   - user data
  return Observable<Map>"
    (let [user-observable (-> (getUser userId)
                (.map (fn [user] {:user-name (:name user) :language (:preferred-language user)
          bookmark-observable (-> (getVideoBookmark userId videoId)
                (.map (fn [bookmark] {:viewed-position (:position bookmark)})))
          ; getVideoMetadata requires :language from user-observable so nest inside map func
```

```
        video-metadata-observable (-> user-observable
            (.mapMany
              ; fetch metadata after a response from user-observable is received
              (fn [user-map]
                (getVideoMetadata videoId (:language user-map)))))]
        ; now combine 3 observables using zip
        (-> (Observable/zip bookmark-observable video-metadata-observable user-observable
              (fn [bookmark-map metadata-map user-map]
                {:bookmark-map bookmark-map
                 :metadata-map metadata-map
                 :user-map user-map}))
          ; and transform into a single response object
          (.map (fn [data]
                  {:video-id videoId
                   :video-metadata (:metadata-map data)
                   :user-id userId
                   :language (:language (:user-map data))
                   :bookmark (:viewed-position (:bookmark-map data))
                  })))))
```

The response looks like this:

```
{:video-id 78965,
 :video-metadata {:video-id 78965, :title House of Cards: Episode 1,
                  :director David Fincher, :duration 3365},
 :user-id 12345, :language es-us, :bookmark 0}
```

And here is a marble diagram that illustrates how that code produces that response:

The following example, in Groovy, comes from Ben Christensen's QCon presentation on the evolution of the Netflix API. It combines two Observables with the `merge` operator, then uses the `reduce` operator to construct a single item out of the resulting sequence, then transforms that item with `map` before emitting it:

```
public Observable getVideoSummary(APIVideo video) {
   def seed = [id:video.id, title:video.getTitle()];
   def bookmarkObservable = getBookmark(video);
   def artworkObservable = getArtworkImageUrl(video);
   return( Observable.merge(bookmarkObservable, artworkObservable)
      .reduce(seed, { aggregate, current -> aggregate << current })
      .map({ [(video.id.toString() : it] }))
}
```

And here is a marble diagram that illustrates how that code uses the `reduce` operator to bring the results from multiple Observables together in one structure:

Error Handling

Here is a version of the Wikipedia example from above revised to include error handling:

```
/*
 * Fetch a list of Wikipedia articles asynchronously, with error handling.
 */
def fetchWikipediaArticleAsynchronouslyWithErrorHandling(String... wikipediaArticleNames) {
    return Observable.create({ subscriber ->
        Thread.start {
            try {
                for (articleName in wikipediaArticleNames) {
                    if (true == subscriber.isUnsubscribed()) {
                        return;
                    }
                    subscriber.onNext(new URL("http://en.wikipedia.org/wiki/"+articleName).g
                }
                if (false == subscriber.isUnsubscribed()) {
                    subscriber.onCompleted();
                }
            } catch(Throwable t) {
                if (false == subscriber.isUnsubscribed()) {
                    subscriber.onError(t);
                }
            }
            return (subscriber);
        }
    });
}
```

Notice how it now invokes `onError(Throwable t)` if an error occurs and note that the following code passes `subscribe()` a second method that handles `onError`:

```
fetchWikipediaArticleAsynchronouslyWithErrorHandling("Tiger", "NonExistentTitle", "Elephant'
    .subscribe(
        { println "--- Article ---\n" + it.substring(0, 125) },
        { println "--- Error ---\n" + it.getMessage() })
```

See the Error-Handling-Operators page for more information on specialized error handling techniques in RxJava, including methods like `onErrorResumeNext()` and `onErrorReturn()` that allow Observables to continue with fallbacks in the event that they encounter errors.

Here is an example of how you can use such a method to pass along custom information about any exceptions you encounter. Imagine you have an Observable or cascade of Observables — `myObservable` — and you want to intercept any

exceptions that would normally pass through to an Subscriber's `onError` method, replacing these with a customized Throwable of your own design. You could do this by modifying `myObservable` with the `onErrorResumeNext()` method, and passing into that method an Observable that calls `onError` with your customized Throwable (a utility method called `error()` will generate such an Observable for you):

```
myModifiedObservable = myObservable.onErrorResumeNext({ t ->
   Throwable myThrowable = myCustomizedThrowableCreator(t);
   return (Observable.error(myThrowable));
});
```