

Tejaswini Mandar Jog

Reactive Programming With Java 9

Develop concurrent and asynchronous applications
with Java 9



Packt

Reactive Programming With Java 9

Develop concurrent and asynchronous applications with
Java 9

Tejaswini Mandar Jog

Packt

BIRMINGHAM - MUMBAI

Reactive Programming With Java 9

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2017

Production reference: 1190917

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78712-423-3

www.packtpub.com

Credits

Author Tejaswini Mandar Jog	Copy Editor Safis Editing
Reviewer Jay Lee	Project Coordinator Prajakta Naik
Commissioning Editor Aaron Lazar	Proofreader Safis Editing
Acquisition Editor Karan Sadawana	Indexer Francy Puthiry
Content Development Editor Lawrence Veigas	Production Coordinator Nilesh Mohite
Technical Editor Tiksha Sarang	

About the Author

Tejaswini Mandar Jog is a passionate and enthusiastic Java trainer. She has more than nine years of experience in the IT training field, specializing in Java, J2EE, Spring, and relevant technologies. She has worked with many renowned corporate companies on training and skill enhancement programs. She is also involved in the development of projects using Java, Spring, and Hibernate. Tejaswini has written two books. In her first book, *Learning Modular Java Programming*, the reader explores the power of modular programming to build applications with Java and Spring. The second book, *Learning Spring 5.0*, explores building an application using the Spring 5.0 framework with the latest modules such as WebFlux for dealing with reactive programming.

I am very lucky to get a chance to write another book on the newest concept in the market added to Java 9. Nowadays, most the developers are talking about Java 9. I got a chance not only to explore Java 9 but also Reactive Programming. It's an exciting and challenging opportunity. The challenge was to introduce the concepts in the manner so that the reader will be able to understand them without difficulty.

Thank you, Mandar, for being the first reader of the book and for giving sincere feedback. Thank you so much for taking an active part in the writing process and helping me to improve.

This book would not have been completed without Lawrence Veigas, the editor, and Tiksha Sarang, the technical editor of the book. The valuable input and comments from them helped a lot to improve the content. The comments by the technical reviewer, Jay Lee, really made the difference. Thank you, Jay Lee. I want to take the opportunity to thank the Packt Pub team for all the effort at various stages of the book. Without you guys, it wouldn't have been possible.

How could I forget my lovely son, Ojas! Thank you, Ojas, for being so supportive and patient while I was busy with my work! Papa and Mumma, thank you for your support and encouragement, which motivates me to try giving my best. Love you!!

Finally, thank you all who helped by supporting me directly and indirectly to

complete this book. Thank you just for being with me as my biggest support.

About the Reviewer

Jay Lee is currently working at Pivotal as senior platform architect. His job is to help big enterprise Cloud-Native Journey with Spring, Spring Boot, Spring Cloud, and Cloud Foundry. Before joining Pivotal, he spent 10 years at Oracle and worked with big enterprises for their large-scale Java distributed system and middleware. Currently, Jay is authoring a microservices book (the name of the book is yet to be decided) using Spring Boot and Spring Cloud.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1787124231>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
 - Downloading the example code
 - Errata
 - Piracy
 - Questions

1. Introduction to Reactive Programming

- Asynchronous programming
 - Concurrency
 - Parallel programming
 - Streams
 - Features of Streams
 - The sequential elements
 - Source of a stream
 - Performing operations
 - Performing automatic operations
- Reactive Streams
 - What do Reactive Streams address?
 - Asynchronicity
 - Back pressure

Reactive Programming

- Reactive Manifesto
 - Responsiveness
 - Resilience
 - Elastic
 - Message-driven
 - Scalable
- Benefits of Reactive Programming
- Reactive Extensions
- RxJava

Advantages of RxJava
Project Reactor
The features of Project Reactor
Akka Streams
Actor
Ratpack
Aim of Ratpack
Quasar
MongoDB
Slick
Vert.x
The components of Vert.x
Reactive Streams in Java 9
The Technology Compatibility Kit (TCK)
API components
Reactive in and across JVM
ThreadPoolExecutor
ScheduledThreadPoolExecutor
The Fork/Join framework
Summary

2. Programming Paradigm Shift

Programming paradigm
Imperative programming
Procedural programming
Avoiding bugs
Functional programming
Need for functional programming
Concurrency
Functional programming in JVM
Functional interface
Lambda expressions
Advantages of using lambda expressions
Functional interfaces provided by JDK 8
Method reference
Streams API
Characteristics of functional programming
Declarative programming
Characteristics of declarative programming
Advantages of declarative programming

Operator fusion

Macro fusion

Micro fusion

Summary

3. Reactive Streams

Discussing the Collection framework

Streams in action

Delving into the Flow API

The building blocks

Publisher

The SubmissionPublisher class

Subscriber

The Subscription interface

The behavior of the Publisher-Subscriber system

Processor

The behavior of the Publisher-Processor-Subscriber system

Designing nonblocking interfaces

Understanding backpressure

Summary

4. Reactive Types in RxJava

ReactiveX

Observer design pattern

The Iterator design pattern

Advantages of ReactiveX

RxJava

Observable Streams

Components of RxJava

Observable

Observer

Subscriber

Hot or cold Observables

Creating Observable emitting sequential data

Using the just() operator

Using the defer() operator

Using the empty() operator

Using the never() operator

Using the error() operator

Using the create() operator

Transforming Rx non-compatible sequences into Observables

Conversion using the from() operator

Creating Observables for unrestricted infinite length

The interval() operator

The range() operator

The timer() operator

Types of observables

The Observable<T> operator

Scenarios for using an Observable

The Single<T> operator

The Flowable<T> operator

Scenarios to use Flowable

The Maybe<T> operator

Completable

Understanding the Disposable interface

Disposable Subscribers

The DefaultSubscriber class

DisposableSubscriber

The ResourceSubscriber interface

CompositeDisposable

Subject

Flavors of Subject

The AsyncSubject class

The BehaviorSubject class

ReplaySubject

PublishSubject

Summary

5. Operators

Demystifying RxMarbles using an Observable

Transforming Observables using operators

Using the buffer() operator

Using the flatMap() operator

Using the groupBy() operator

Using the map() operator

Using the scan() operator

Using the window() operator

Filtering Observable

Using the debounce() operator

Using the distinct() operator

Using the distinctUntilChanged() operator

Using the elementAt() operator

Using the filter() operator

Using the first(), never(), and empty() operators

Using the last() operator

Using the ignoreElements() operator

Using the sample() operator

Using the skip() operator

Using the skipLast() operator

Using the take() operator

Using the takeLast() operator

Combining Observables

Using the merge() operator for combining observables

Using the combineLatest() operator

Using the startWith() operator

Using the and(), then(), and when() operators

Using the switchIfEmpty() operator

Using the zip() operator

Using the join() operator

Conditional operators

Using the all() operator

Using the amb() operator

Using the contains() operator

Using the defaultIfEmpty() operator

Using the sequenceEqual() operator

Using the takeWhile() operator

The mathematical and aggregate operators

Using the average() operator

Using the concat() operator

Using the count() operator

Using the max() operator

Using the min() operator

Using the reduce() operator

Using the sum() operator

Summary

6. Building Responsiveness

Concurrency

Comparing asynchronicity with concurrency

Scheduling

Scheduler

- The subscribeOn() operator

- The observeOn() operator

Schedulers

- The computation() method

- The from() operator

- The io() operator

- The single() operator

- The newThread() operator

- The trampoline() operator

Operators and thread safety

- Operators are not thread safe

- The serialize() operator

Latency

Summary

7. Building Resiliency

Resilience

- Reliable applications

- Resiliency

Error handling in RxJava

- Exploring the doOnError() operator

- Reacting to an error

- Recovering from an error

- Absorb the error and switch over to the backup Observable which allows continuing the sequence using the onErrorResumeNext() operator

- Absorb the error and emit a specified default item using the onErrorReturn() operator

- Absorb the error and then try to restart the Observable that has failed using the onExceptionResumeNext() operator

- Absorb the error and then try to restart the Observable which has failed using the retry() operator without delay

- Absorb the error and then try to restart the Observable that has failed using the retryWhen() operator with delay

Exceptions specific to RxJava

- CompositeException

- MissingBackpressureException

- OnErrorHandlerException
- OnErrorHandlerFailedException
- OnErrorHandlerThrowable
- Error handler
 - Handling specific undeliverable exceptions
 - Exceptions introduced for tracking
 - Exception wrapping
- Summary

8. Testing

- Testing the need and role of a developer
- The traditional way of testing an Observable
- The modern approach for testing an Observable
 - The BaseTestConsumer class
 - The TestObserver class
 - The TestSubscriber class
 - Testing time-based Observable
 - The TestScheduler class
- Testing the Subscriber
- The test() operator
- Testing notifications
 - Demonstration 1 - updating code for the just() operator
 - Demonstration 2 - updating the code for the never() operator
 - Demonstration 3 - updating the code for Flowable
 - Demonstration 4 - updating the code for the error() operator
 - Demonstration 5 - updating the code for the interval() operator

Mockito testing

Summary

9. Spring Reactive Web

- An introduction to Spring Framework
- Plumbing code
 - Scalability
 - Boilerplate code
 - Unit testing of the application
- Features of the Spring Framework
 - POJO-based development
 - Loose coupling through Dependency Injection (DI)
 - Declarative programming
 - Boilerplate code reduction using aspects and templates

- The Spring architecture
 - Core modules
 - Data access and integration modules
 - Web MVC and remoting modules
 - AOP modules
 - Instrumentation modules
 - Test modules
 - Project Reactor
 - Operators
 - Spring web Reactive Programming
 - Server-side support
 - Annotation-based support
 - The functional programming model
 - HandlerFunction
 - ServerRequest
 - ServerResponse
 - RouterFunction
 - Client side
 - Creating a WebClient instance
 - Using the WebClient instance to prepare a request
 - Reading the response
 - Deploying the application
 - Deploying the application from Eclipse IDE
 - Creating and deploying WAR file on server
 - Testing the application
 - Working with WebTestClient
 - Server-Sent Events (SSE)
 - Summary
 - 10. Implementing Resiliency Patterns Using Hystrix
 - Hystrix- an introduction
 - How Hystrix works
 - Demonstrating HystrixCommand
 - Demonstrating HystrixObservableCommand
 - Handling fallback
 - Demonstrating HystrixCommand with a fallback
 - Demonstrating HystrixObservableCommand with fallback
 - Design patterns in resiliency
 - Bulkhead pattern
 - Partition the services

- Partition the resources
- Benefits
- Circuit breaker
 - Proxy states
 - Closed state
 - Open state
 - Half-Open state
 - Retry pattern
 - Retry
 - Retrying after delay
 - Cancel
- Queue-based load leveling pattern
- Patterns used by Hystrix
 - Fail-fast
 - Fail-silent
 - Static fallback
 - Stubbed fallback
 - Cache via network fallback
 - Dual-mode fallback
 - Isolation
 - Using threads and thread pools for isolation
 - Reasons for using thread pools in Hystrix
 - Benefits of using thread and thread pools
 - Drawbacks of the thread pool
 - Semaphores
- Request collapsing or request batching
 - The working
- Request caching
 - Difference between request collapsing and request caching
- Summary

11. Reactive Data Access

- Spring Data
 - Modules provided by Spring Data
 - Features of Spring Data
- Spring Data repositories
 - Using the repository
- Spring Data and Reactive Programming
 - ReactiveCrudRepository
 - RxJava2CrudRepository

- Spring Data Reactive and MongoDB
 - Features added to Spring Data MongoDB 2.0
 - Using MongoDB with Spring
 - ReactiveMongoTemplate
 - Spring Data Repositories and MongoDB
 - ReactiveMongoRepository
- Spring Data Reactive and Redis
 - Connecting to Redis
 - Jedis Connector
 - Lettuce connector
 - RedisTemplate
 - Spring Data Repositories and Redis
- Kafka
 - Reactive API for Kafka
 - Reactor Kafka components for handling reactive pipelines
 - Sender
 - Creating the Sender
 - Creating messages
 - Sending messages
 - Closing the Sender
 - Receiver
 - Creating a Receiver instance
 - Reading the messages
 - Running the Kafka application
 - Summary

Preface

At the moment, technology plays a major role in the success of businesses and in reaching out to more users as early as possible. This demand will increase day by day. Along with the increase in the demand, the user expectations also have increased. Now, the users are demanding a quick, more responsive, and reliable response. Reactive programming is a programming model that helps in tackling the essential complexity that comes with writing such applications. We, as Java developers, are very much familiar with the imperative style of programming; however, now to tackle the essential complexity, reactive programming uses declarative and functional paradigms to build the programs. This book aims at making the paradigm shift easily by discussing the concepts about functional programming in depth.

This book begins with explaining what reactive programming and the Reactive Manifesto is, and about the Reactive Streams specification. It uses Java 9 to introduce the declarative and functional paradigm, which is very necessary to write programs in reactive style. It explains Java 9's Flow API, an adoption of the Reactive Streams specification. From this point on, it focuses on RxJava 2.0, covering topics like such as creating, transforming, filtering, combining, and testing Observables. It will then talk about how to use Java's popular framework, Spring, to build event-driven, reactive apps and its deployment on the server. By the end of the book, readers you will be fully equipped with the tools and techniques to implement robust event-driven reactive applications.

What this book covers

[Chapter 1](#), *Introduction to Reactive Programming*, explores reactive programming by explaining different terms used in reactive systems such as responsiveness, resiliency, elasticity, scalable, and message driven and much more. We will discuss RxJava, Project Reactor, Akka, and Ratpack as reactive extensions available in the market.

[Chapter 2](#), *Programming Paradigm Shift*, is designed in such a way that you will smoothly shift from imperative programming to declarative programming since we, as Java developers, are using imperative style of coding to write the code. However, reactive programming is concentrated around functional programming, which is new to most of us. The chapter is designed in such a way that you will smoothly shift from imperative programming to declarative programming. This chapter is full of demos which help you to understand how to use functional programming that enables adoption of declarative programming and how to use functions as value types. We will discuss lambda expression, functional interfaces, lazy evaluation, and functional composition in depth. We will also discover the operator fusion in this chapter.

[Chapter 3](#), *Reactive Streams*, talks about Java 8 Streams and the newly introduced Flow API in Java 9. We will be discussing Publisher, Subscriber, Processor, and Subscription as backbones of Flow API. The ill application design may ruin the efforts of creating nonblocking applications. In this chapter, we will also discuss in depth about designing a nonblocking interface to achieve the goal.

[Chapter 4](#), *Reactive Types in RxJava*, helps you in understanding RxJava as another implementation of Reactive Streams. You will explore in detail about the important types in RxJava that represent the producer and consumer of asynchronous events and the subscription, which gets created when a consumer subscribes to a consumer.

[Chapter 5](#), *Operators*, focuses on the extensions of reactive extensions, known

as operators in RxJava that enables asynchronous events to be filtered and transformed. We will discuss various ways of combining streams of events from different sources. You will explore the operators and their working using RxMarbles diagrams.

[Chapter 6, Building Responsiveness](#), explores RxJava Scheduler and how RxJava uses Schedulers to execute asynchronous flow in different threads or at periodically or at scheduled times. We will discuss the differences between concurrency and parallelism and the criteria for selection of one versus the other.

[Chapter 7, Building Resiliency](#), dives deep into resiliency. The chapter discusses various kinds of failures that can happen while building asynchronous flow. We will also discuss how RxJava approaches error handling and how to use the various error handling operators and utilities provided by RxJava.

[Chapter 8, Testing](#), explores all about testing reactive applications. We will discuss the difference between testing traditional application and application designed for reactive applications. We will discuss various tips and tricks around unit testing the asynchronous flows. RxJava provides various testing utilities which makes unit testing the flows easier.

[Chapter 9, Spring Reactive Web](#), talks about the adoption of reactive programming by the Spring framework. The chapter gives an orientation of the Spring framework and of implementing reactive programming by developing an application.

[Chapter 10, Implementing Resiliency Patterns Using Hystrix](#), discusses Hystrix, which is a latency and fault tolerance library from Netflix that uses RxJava. It shows how various resiliency patterns have been implemented within Hystrix.

[Chapter 11, Reactive Data Access](#), talks about accessing the data from various data stores in a reactive fashion. You will explore reactive repositories using Spring Data Reactive, working with reactive Redis Access using Lettuce and reactive pipelines using Reactor Kafka.

What you need for this book

The most important thing you need to go with reactive programming is practical knowledge of Java. Any person who has basic practical knowledge of JDK can start with reactive programming. The practical knowledge of JDK 8 will be an added advantage to go ahead. You will need the basic knowledge of JDK 9 as we are using it for development throughout the book. The basic knowledge of multithreading, JUnit, RESTful web services, and Spring Framework 5.0 will be an added advantage. Practical knowledge of Eclipse IDE is required as we will be using it throughout the book.

Who this book is for

The book is designed in such a way that it will be helpful for both beginners as well as the developers who are currently working with Java 8 or RxJava. The book will also be helpful for all those who want to start working with reactive programming using Java 9. In every chapter, we will start with a discussion of the very basic knowledge and then dive deep into the core concepts. The organization of the chapters will help the student of computer science who wants to upgrade their skills with the cutting edge concepts in the market. So, be active and give an attempt to reactive programming without fear.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input and Twitter handles are shown as follows:

"The `buffer()` operator accepts arguments of type `integer` where developers can specify how many items to bundle together."

A block of code is set as follows:

```
public class Demo_flatMap {  
    public static void main(String[] args) {  
        Observable.range(1,5).flatMap(item->Observable.range(item,  
            3)).subscribe(value->System.out.print(value+"->"));  
    }  
}
```

Any command-line input or output is written as follows:

```
got:-12  
got:-30
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Many times, we click on the link to access very important information and suddenly we get a page saying No resource available or Service temporarily down."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Reactive-Programming-With-Java-9>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Introduction to Reactive Programming

The world is on the web. Nowadays, everyone uses the internet for banking applications, searching for information, online purchasing, marking products by companies, personal and professional communication via emails, and as a way to present themselves on public platforms, such as Twitter. We use the internet for personal as well as professional purposes. Most organizations have their websites, applications for mobiles, or some other way to exist on the internet. These organizations are wide-ranging. They have different functionalities and their nature of working is also very different. Also, they work in a variety of domains that need a lookalike of the developed application to serve all functionalities, application to serve more flexible, resilient, and robust functionalities. Application requirements, both from the user's perspective as well as the developer's perspective, have changed dramatically. The most important change now, which even a normal user may recognize, relates to the time taken for a response. You may be a technical person who understands terminologies such as a request, a response, or a server or someone who only knows how to browse and has no understanding of the technicalities of web programming. We all want answers on the web as quickly as possible. Today, no one wants to wait even for seconds to get the result or response they asked for; users demand a quick response. We are processing a huge amount of data as compared to applications developed in the past. Also, a lot of hardware-related changes have taken place. Today, instead of going with many servers for the deployment of an application, the preferred way is to deploy the application on the cloud. A cloud-based deployment makes the application more maintainable by saving hours of maintenance. Now, the market demands responsive, loosely coupled, scalable, resilient, and message-driven systems. The traditional approach to programming can't fulfil this requirement. The approach needs to be changed. We need to adapt to asynchronous, non-blocking reactive programming!

In this chapter, we will concentrate on the following points so as to get familiar with the terms, situations, and challenges in reactive programming:

- Reactive Manifesto
- Reactive Streams
- Reactive Extensions
- Reactive in a JVM
- Reactive across JVMs

In the 90s, when Java actually started acquiring a grip on the market, many companies started developing applications using it. The basic reason was, it allowed you to do this with the applications--*write once, run anywhere*. The internet was still not so famous and had very few users. But, in the early 2000s, the conditions started changing dramatically. Billions of users were now using the internet for different reasons, and experts were aware of the increase in these numbers. So applications, specifically web applications, had to handle enormous traffic. From the development perspective, this led to problems of scalability and how to handle the pressure of so many requests. The users of a system also became more demanding with regard to look and feel along the way. They also demanded a quick response.

Consider a pharmaceutical application dealing in pharmaceutical products. The application helps find distributor information in an area, the price of the product, the details of the product, the stock, and other relevant things. Let's consider one of the functionalities of the application to get the list of distributors. When a user wants to find a list of the pharmaceutical distributors in his/her city or area, what will they do? Yes! They will either enter or select the name of the city to find the distributor list. After entering the data, the user just needs to click on a button to initiate the request.

We have two options as developers:

- Keep the user waiting until the result has not been processed to find the list of the distributors that will be fetched from the database
- Second, allow the user to proceed further and use other stuff in the application, such as finding product details, availability of products, and so on

Instead of discussing what a developer will do, you will ask, "If you are the user, which of the preceding scenarios would you have preferred?"

Obviously, I will also choose not to wait. Instead, I'd enjoy the amazing

functionality of the application which will eventually process the result. If you haven't recognized it yet, let me tell you that you have chosen parallel programming. Correct! You have preferred an asynchronous approach rather than the traditional synchronous approach. What is an asynchronous approach? If you are already aware of this, you can skip the discussion and go on to the next discussion.

Asynchronous programming

The term **asynchronous programming** means parallel programming where some unit of work or functionality runs separately from the main application without blocking it. It's a model that allows you to take leverage of the multiple cores of the system. Parallel programming uses multiple CPU cores to execute the tasks, ultimately increasing the performance of the application. We know very well that each application has at least one thread, which we usually call the main thread. Other functionalities run in separate threads. We may call them child threads; they keep notifying the main thread about its progress as well as failure.

The major benefit provided by the asynchronous approach is that the performance and responsiveness of the application improve drastically. In today's world, the application market is too demanding. No one wants to be kept waiting while the application is processing things that one needs an answer for. No matter whether it's a desktop application or the web, users prefer to continue enjoying the application while the application is computing a complicated result or performing some expensive task on a resource. We need this in all the scenarios where developers don't want to block the user while using the application.

Now, you may be thinking of many questions: Do we need asynchronous programming? Should one choose asynchronous programming over concurrency? If yes, when do we need it? What are the ways? What are its benefits? Are there any flaws? What are the complications? Hold on! We will discuss them before we move ahead. So let's start with one interesting and well-known style of programming.

Concurrency

The term **concurrency** comes in whenever developers talk about performing more than one task at the same time. Though we keep on saying we do have tasks running simultaneously, actually they are not. Concurrency takes advantage of CPU time slicing, where the operating system chooses a task and runs a part of it. Then, it goes to the next task keeping the first task in the state of waiting.

The problems encountered in using a thread-based model are as follows:

- Concurrency may block the processing, keeping the user waiting for the result, and also waste computational resources.
- Some threads increase the performance of the applications and some degrade it.
- It's an overhead to start and terminate many threads at the same time.
- Sharing the available limited resource in multiple threads hampers performance.
- If data is being shared between multiple threads, then it's complicated to maintain its state. Sometimes, it is maintained by synchronization or using locks, which complicates the situation.

Now, you may have understood that concurrency is a good feature, but it's not a way to achieve parallelism. Parallelism is a feature that runs a part of the task or multiple tasks on multiple cores at the same time. It cannot be achieved on a single core CPU.

Parallel programming

Basically, the CPU executes instructions, usually one after another. Parallel programming is a style of programming in which the execution of the process is divided into small parts at the same time, with better use of multi-core processors. These small parts, which are generally called **tasks**, are independent of each other. Even the order of these tasks doesn't matter. A single CPU has performance limitations as well as the availability of memory, making parallel programming helpful in solving larger problems faster by utilizing the memory efficiently.

In any application, we have two things to complete as a developer. We need to design the functionalities that will help in continuing the flow of the application, helping users smoothly use their applications. Every application uses data to compute user requests. Thread-based models only help in the functional flow of the application. To process the data needed by the application, we need something more. But what do we need? We need a data flow computation that will help deal with data flows and their changes. We need to find the ease with which either static or dynamic data flows and the underlying model could use it. Also, changes in the data will be propagated through the data flow. It is reactive programming that will fulfil this requirement.

Streams

We are all very familiar with and frequently use the Collections framework to handle data. Though this framework enables a user to handle data quite efficiently, the main complexity lies in using loops and performing repeated checks. It also doesn't facilitate the use of multi-core systems efficiently. The introduction of Streams has helped to overcome these problems. Java 8 introduces Streams as a new abstract layer that enables the processing of data in a declarative manner.

A **Stream** is a series of different elements that have been emitted over a time period. Are Streams the same as an array? In a way, yes! They are like an array, but not an array. They do have distinct differences. The elements of an array are sequentially arranged in memory, while the elements in a Stream are not! Every Stream has a beginning as well as an end.

Let's make it simpler by discussing a mathematical problem. Consider calculating the average of the first six elements of an array and then Streams. Let's take the array first. What will we, as developers, do to calculate the average of an array? Yes, we will fetch each element from an array and then calculate their addition. Once we get the addition, we will apply the formula to calculate the average. Now, let's consider Streams. Should we apply the same logic as we just applied for an array? Actually, no! We should not directly start developing the logic. Before the logic, we must understand a very important thing that every single element in the Streams may not be visited. Also, each element from the Streams may be emitted at the same speed. So while the calculation is done, some elements may not have been visited at all. The calculated average is not the final value, but the answer will be of the values in motion. Confused?

Have you ever sat on the banks of a river or flowing water and put your legs in it? Most of us have at least once enjoyed this peaceful experience. The water moves around our legs and moves ahead. Have you ever observed the same flowing water passing through your legs twice? Obviously, not! It's the same when it comes to Streams.

Features of Streams

We just discussed Streams; now let's discuss the features offered by Streams.

The sequential elements

A Stream provides a sequence of the typical type of elements on demand.

Source of a stream

As a Stream contains elements, it needs elements. A Stream can take Collections, Arrays, files, or any other I/O resource as its input.

Performing operations

Streams strongly support performing various operations, such as filtering, mapping, matching, finding, and many more, on their elements.

Performing automatic operations

Streams don't need explicit iterations to perform on the elements from the source; they do the iteration implicitly.

Reactive Streams

Streams help in handling data. Today's application demands are oriented more toward live or real-time data operations. Today's world doesn't want only a collection of data; instead, most of the time, this is followed by modification and filtration. This processing requires more time, leading to performance bottlenecks. Streams help in turning this huge amount of data processed and provide a quick response. It means the ever-changing data in motion needs to be handled. The initiative of reactive streams started in late 2013 by engineers from Pivotal, Netflix, and Typesafe. Reactive Streams is a specification that has been developed for library developers. Library developers write code against Reactive Streams APIs. On the other hand, business developers work with the implementation of the Reactive Streams specification.

Business developers have to concentrate on the core business logic required for the application. Reactive Streams provide a level of abstraction to concentrate on the business logic instead of getting involved in low-level plumbing to handle streams.

What do Reactive Streams address?

We have just discussed Reactive Streams. The following are the features around which they are woven.

Asynchronicity

Today's application users don't like to wait for a response from the server. They don't care about the processing of the request--collecting the required information and then generating the response. They are just interested in getting a quick response without a block.

Asynchronous systems decouple the components of a system allowing the user to explore and use other parts of the application instead of wasting the time of the user in waiting for the response. Asynchrony enables parallel programming so as to compute the resources, collaborate the resources over the network, and collaborate with and use multiple cores of a CPU on a single machine to enhance the performance. In the traditional approach, developers compute one function after another. The time taken by the complete operation is the sum of the time taken by each of the functionalities. In the asynchronous approach, operations are carried out parallelly. The total time taken to complete the operation here is the time taken by the longest operation and not the sum of each operation of the application. This ultimately enhances the performance of the application to generate a quicker response.

Back pressure

The Reactive Streams specification defines a model for back pressure. The elements in the Streams are produced by the producer at one end, and the elements are consumed by the consumer at the other end. The most favorable condition is where the rate at which the elements are produced and consumed is the same. But, in certain situations, the elements are emitted at a higher rate than they are consumed by the consumer. This scenario leads to the growing backlog of unconsumed elements. The more the backlog grows, the faster the application fails. Is it possible to stop the failure? And if yes, how to stop this failure? We can certainly stop the application from failing. One of the ways to do this is to communicate with the source or the publisher to reduce the speed with which elements are emitted. If the speed is reduced, it ultimately reduces the load on the consumer and allows the system to overcome the situation.

Back pressure plays a very vital role as a mechanism that facilitates a gradual response to the increasing load instead of collapsing it down. It may be a possibility that the time taken to respond may increase, leading to degradation; however, it ensures resilience and allows the system to redistribute the increasing load without failure.

The elements get published by the publisher and collected by the subscriber or consumer at the downstream. Now, the consumer sends a signal for the demand in the upstream, assuring the safety of pushing the demanded number of elements to the consumer. The signal is sent asynchronously to the publisher, so the subscriber is free to send more requests for more elements with a pull strategy.

Reactive Programming

The term **reactive** or **Reactive Programming (RP)** has been in use at least since the paper, *The Reactive Engine* was published by Alan Kay in 1969. But the thought behind Reactive Programming is the result of the effort taken by Conal Elliot and Paul Hudak who published a paper *Function Reactive Animation* in 1997. Later on, the Function reactive animation was developed, which is also referred to as **functional reactive programming (FRP)**. FRP is all about the behavior and how the behavior is changing and interacting on the events. We will discuss FRP more in the next chapter. Now, let's only try to understand Reactive Programming.

RP designs code in such a way that the problem is divided into many small steps, where each step or task can be executed asynchronously without blocking each other. Once each task is done, it's composed together to produce a complete flow without getting bound in the input or output.

Reactive Manifesto

RP ensures it matches the requirements of the everyday changing market and provides a basis for developing scalable, loosely coupled, flexible, and interactive applications. A reactive application focuses on the systems that react to the events, the varying loads, and multiple users. It also reacts effectively to all the conditions, whether it's successful or has failed to process the request. The Reactive system supports parallel programming to avoid blocking of the resources in order to utilize the hardware to its fullest. This is totally opposite to the traditional way of programming. Reactive Programming won't keep the resources busy and allows them to be used by other components of the system. Reactive systems ensure they provide features such as responsiveness, resilience, elasticity, scalability, and a message-driven approach. Let's discuss these features one by one.

Responsiveness

Responsiveness is the most important feature of an application and ensures quick and effective responses with consistent behavior. This consistent behavior ensures the application will handle errors as well.

Applications that delay in giving a response are regarded by the users as not functioning well, and soon they start ignoring them. Responsiveness is a major feature required in today's applications. It optimizes resource utilization and prepares the system for the next request. When we open a web page that has too many images, it usually takes more time to open the images, but the content of the page gets displayed before the images. What just happened? Instead of keeping the user waiting for the web page, we allowed him/her to use the information and refer to the images once downloaded. Such applications are called **more responsive applications**.

Resilience

For resilience, the application will be responsive even in the event of a failure, making it resilient. Let's consider a situation. We have requested for data from the server. While the request is getting processed, the power supply fails. Obviously, all the resources and the data coming from the server as a response will suddenly become unavailable. We need to wait until the power supply is restarted and the server starts taking the load again. What if the server has an alternative power supply in case the main supply fails. Yes, no one has to wait as the alternative supply keeps the server running and enables it to work without fail. All of this happens only because the alternative supply replaces the main power supply. It's a kind of clustering. In the same way, the application may be divided into components that are isolated from each other so that even if one part of the system fails, it will recover without any kind of compromise, providing a complete application experience.

Elastic

Reactive programs react to the changes that happen in the input by allocating resources as per the requirements so as to achieve high performance. The reactive system achieves this elasticity by providing a scaling algorithm. The elastic system has the provision to allocate more resources to fulfil the increasing demand. Whenever the load increases, the application can be scaled up by allocating more resources, and if the demand decreases, it removes them so as to not waste the resources.

Let's consider a very simple example. One day, a few of your friends visit your house without prior intimation. You will be surprised and will welcome them happily. Being a good host, you will offer them coffee or cold drinks and some snacks. How will you serve drinks? Normally, we'd use some of the coffee mugs or glasses that we always keep to one side so that we can use them if required. After the use, will you keep them again with your daily crockery? No! We will again keep them aside. This is what we call elasticity. We have the resources, but we will not use them unless required.

Message-driven

Reactive systems use an asynchronous message, passing between the components for communication, to achieve isolation and loose coupling without blocking the resources. It facilitates easy to extend, maintainable applications that are flexible as well.

Scalable

The market is changing day by day, so are the client demands too. The workload on an application cannot be fully predictable. The developers need an application that will be able to handle increasing pressure, but in case it doesn't, then they need an application that is easily scalable. Scalability is not only in terms of code, but it must be able to adopt new hardware as well. This is the same as that of elasticity. No, it's not. Don't get confused between these two terminologies.

We'll consider the same example we discussed to understand elasticity, but with a change. Your friends inform you that they are coming this weekend. Now, you are well aware and want to be prepared for the party. Suddenly you realize you don't have enough glasses to serve the soft drinks. What will you do? Very simple, you will buy some use-and-throw glasses. What did you do? You scaled up your hardware by adding more resources.

Benefits of Reactive Programming

The benefits of RP are as follows:

- It increases the performance of the application
- It increases the utilization of computing resources on a multi-core
- It provides a more maintainable approach to deal with asynchronous programming
- It includes back pressure, which plays a vital role to avoid over-utilization of the resources

Reactive Extensions

Reactive Extensions are the set of the tools that allow operations on sequential data without considering whether the data is synchronous or asynchronous. They also provide a set of sequence operators that can be operated on each of the items in sequence. There are many libraries that implement the Reactive Streams specification. The libraries that support Reactive Programming include Akka, Reactor, RxJava, Streams, Ratpack, and Vert.x.

RxJava

ReactiveX has been implemented as a library for languages such as JavaScript, Ruby, C#, Scala, C++, Java, and many more. RxJava is the Reactive Extension for JVM, specifically for Java. It is open source. It was created by Netflix in 2014 and published under the Apache 2.0 license. It has been created to simplify concurrency on the server side. The main goal of RxJava is to enable the client to invoke a heavy request that will be executed in parallel on the server.

Now we are well aware that in Reactive Programming, the consumer reacts to the incoming data. Reactive programming basically propagates the changes in the events to the registered observers. The following are the building blocks of the RxJava :

- **Observable:** The observable represents the source of the data. The observable emits the elements which vary in numbers. There is no fixed number of elements to be emitted, it could vary. The observable could successfully emit the elements or with an error if something goes wrong. At a time the observable can have any number of the subscribers.
- **Observer or Subscriber:** The subscribers listen to the observables. It consumes the elements emitted by the observable.
- **Methods:** The set of methods enables the developers to handle, modify and compose the data.

Following are the methods used in RxJava programming:

- - `onNext()`: When a new item is emitted from the observable, the `onNext()` method is called on each subscriber
 - `onComplete()`: When the observable finishes the data flow successfully, the `onComplete()` method gets called
 - `onError()`: The `onError()` method will be called in situations where

the observable finishes data emission with an error

Advantages of RxJava

The following are a few of the advantages of RxJava:

- It allows a chain of asynchronous operations
- To keep track of the state, we usually need to maintain a counter variable or some variable to maintain the previously calculated value; however, in RxJava, we don't need to get involved in keeping track of the state
- It has a predefined way to handle errors that occur between the processes

Project Reactor

Spring 5 supports Reactive Programming to make more responsive and better-performing applications. Spring was added with Spring web reactive and reactive HTTP as new components along with support for Servlet 3.1 specification. Pivotal or Spring developed the Reactor as a framework for asynchronous programming. It enables writing of high-performance applications and works asynchronously using the event-driven programming paradigm. The Reactor uses the design pattern, where the services are received from the clients, and distributes them to different event handlers, where their processing will be done.

The Reactor is a Reactive Programming foundation, specifically for JVMs. It supports unblocking fully. The Project Reactor aims to address a very important drawback of the traditional approach by supporting asynchronicity. It provides an efficient way to handle back pressure. It also focuses on the following aspects, making it more useful for Reactive Programming:

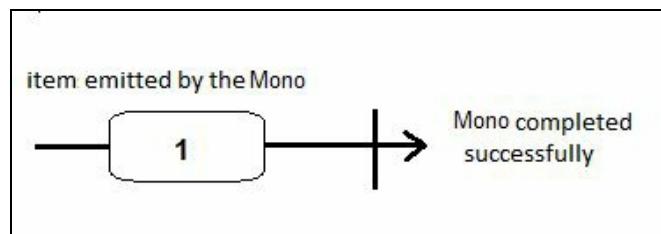
- It facilitates the use of the rich operators to manipulate a data flow
- Data keeps flowing until someone doesn't subscribe to it
- It provides a more readable way of coding so as to make it more maintainable
- It provides a strong mechanism that ensures signalling by the consumer to the producer about what speed it should produce the elements at

The features of Project Reactor

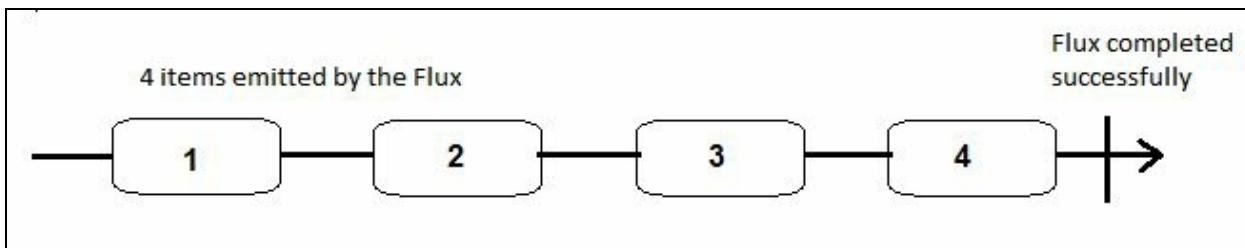
Project Reactor is facilitated by a library that focuses on the Reactive Streams specification, targeting Java 8, and also supports a few features offered by Java 9. This library can expose the operators confirmed by the RxJava API, which are `Publisher`, `Subscriber`, `Subscription`, and `Processor` as core interfaces.

It introduced a reactive type that implements `Publisher`, but it has the following building components:

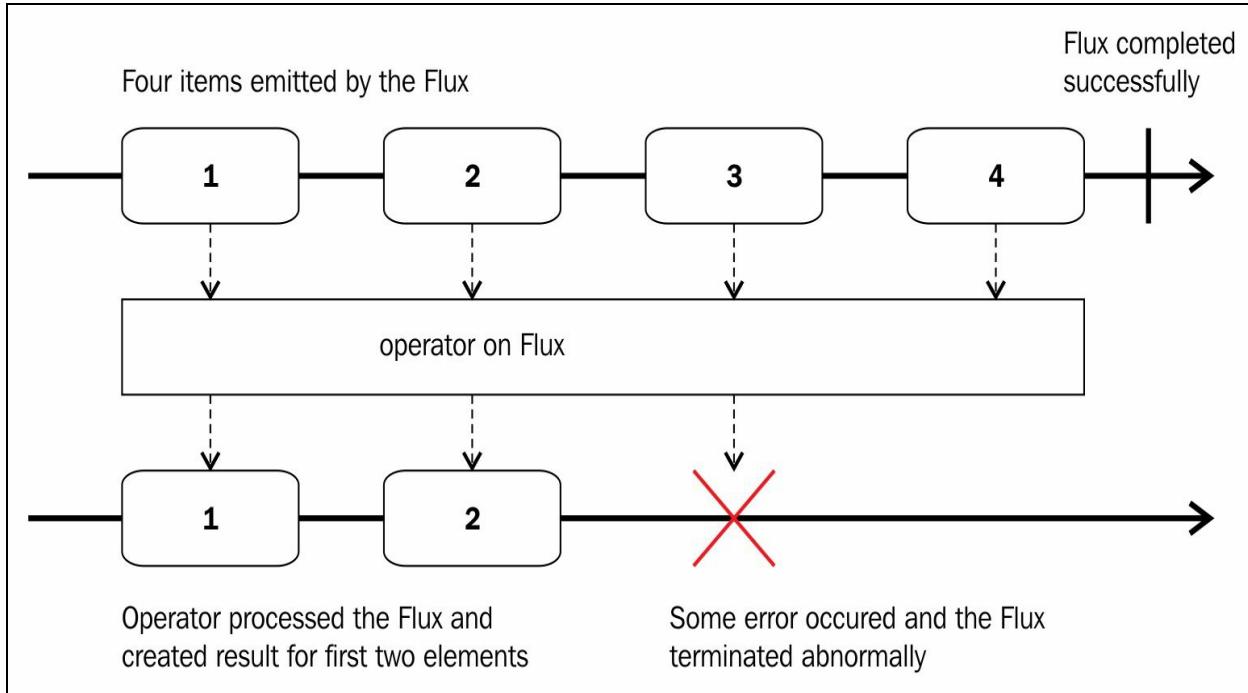
- **Mono**: Mono represents a single element or empty result. It's a very specialized kind of `Publisher<T>` that emits at the most one element. It can be used to represent asynchronous processes with no value denoted as `Mono<Void>`:



- **Flux**: Flux, as you may have guessed, represents a sequence of $0...n$ elements. A `Flux<T>` is a kind of `Publisher<T>` that represents the sequence of 0 to n elements emitted by it, and it either results in a success or an error. It also has three similar methods, namely `onNext()`, `onComplete()`, and `onError()`, for signalling. Let's consider a few of the elements emitted by `Flux` and applied by the operators for the transformation, as shown in the following figure:



- **Operators:** The elements emitted by the publisher may undergo transformations before getting consumed by the consumer. Operators help in handling, transforming, and filtering the elements:



Akka Streams

A huge amount of data is daily getting consumed on the internet through uploading as well as downloading. The increase in demand for the internet also demands the handling of a large amount of data over the network.

Today, data is not only getting handled, but also analyzed and computed for various reasons. This large amount of processing lowers the performance of the application. **Big Data** is the principle of processing such enormous data sequentially as a Stream on single or multiple CPUs.

Akka strictly follows the Reactive Manifesto. It is a JVM-based toolkit and an open source project introduced by Steven Haines along with an actor-based model. It is developed by keeping one aim in mind--providing a simplified construction for developing distributed concurrent applications. Akka is written in Scala and provides language binding between Java and Scala.

The following are the principles implemented by Akka Streams:

- Akka doesn't provide logic but provides strong APIs
- It provides an extensive model for distributed bound Stream processing
- It provides libraries that will provide the users with reusable pieces

It handles concurrency on an actor-based model. An actor-based model is an actor-based system in which everything is designed to handle the concurrency model, whereas everything is an object in object-oriented languages. The actors within the system pass messages for sharing information. I have used the word actor a number of times, but what does this actor mean?

Actor

In this case, an actor is an object in the system that receives messages and takes action to handle them accordingly. It is loosely coupled from the source of the message that generates them. The only responsibility of an actor is to recognize the type of the message it receives and take an action accordingly. When the message is received, the actor takes one or more of the following actions:

- Executing operations to perform calculations, manipulations, and persistency of the data
- Forwarding the message to some other actor
- Creating a new actor and then forwarding the message to it

In certain situations, the actor may choose not to take any action.

Ratpack

Ratpack is an easy-to-use, loosely coupled, and strongly typed set of Java libraries to build fast, efficient, and evolvable HTTP applications. It is built on an event-driven networking engine that gives better performance. It supports an asynchronous non-blocking API over the traditional blocking Java API. The best thing about Ratpack is that, to build any Ratpack-based application, any JVM build tool can be used.

Aim of Ratpack

Ratpack aims to do the following:

- To create applications that are fast, scalable, and efficient with non-blocking programming
- To allow the creation of complex applications with ease and without compromising its functionalities
- To allow the creation of applications that can be tested easily and thoroughly

Quasar

The Quasar framework provides the basis for developing most responsive web applications and hybrid mobile applications. It enables developers to create and manage the project by providing a CLI. It provides project templates that are thorough and well defined to make things easier for beginners. It also provides defaults to Stylus for the CSS, based upon the flexbox for its grid system. Quasar enables developers to write mobile applications that need web storage, cookie management, and platform detection.

MongoDB

The official MongoDB Reactive Streams Java drivers provides asynchronous Stream processing with non blocking back pressure for MongoDB.

MongoDB provides the implementations of the Reactive Streams API for reactive Stream operations.

Along with MongoDB, Redis is (open source) in memory data structure; it used as a database, cache management, and message brokers. We also have Cassandra, which is a highly scalable application with a high-performance distributed database. It is designed for handling huge amounts of data across multiple servers.

Slick

Slick is a library for Scala that facilitates querying and accessing data. It allows developers to work with the stored data similar to how you fetch data using a Scala collection. It enables you to query a database in Scala instead of SQL. It has a special compiler that has the capacity to generate the code as per the underlying databases.

Vert.x

Vert.x provides a library of the modular components to the developers for writing reactive applications in many different languages, such as Java, JavaScript, Groovy, Scala, Ruby, and many more. It can be used without the container as well. Being highly modular, Vert.x provides a free hand to the developers to use any components required for the application instead of using them all.

The components of Vert.x

The following table provides information about the core components of Vert.x:

Name of the component	Use of the component
Core	This provides low-level functionality to support HTTP, TCP, file systems, and many such features.
Web	This provides the components that enable developers to write advanced HTTP clients with ease.
Data access	This provides many different asynchronous clients, enabling you to store a range of data for an application.
Integration	This provides a simple library to write simple SMTP mail clients, which makes it easy to write applications with the functionalities to send mail.
Event Bus Bridge	This lets the developers interact with Vert.x from any application.
Authentication and authorization	This provides the APIs that are useful for the authentication and authorization process.
Reactive	This provides components such as Vert.x Rx, and Vert.x Sync that enable the developers to build more reactive applications.

Reactive Streams in Java 9

In JDK 9, the Flow APIs correspond to the Reactive Streams specification. The JEP 266 contains the interfaces that provide the publication and subscription. Reactive Streams is a standard specification to build Stream-oriented libraries for reactive and non-blocking JVM-based systems. The libraries provide the following:

- The processing of enormous numbers of elements
- The processing of the elements sequentially
- Passing of the elements between the elements asynchronously
- A mechanism to ensure proper communication between the source and the consumer of the elements so as to avoid an extra processing burden on the consumer

Standard Reactive Streams have the following two components:

- The Technology Compatibility Kit (TCK)
- API components

The Technology Compatibility Kit (TCK)

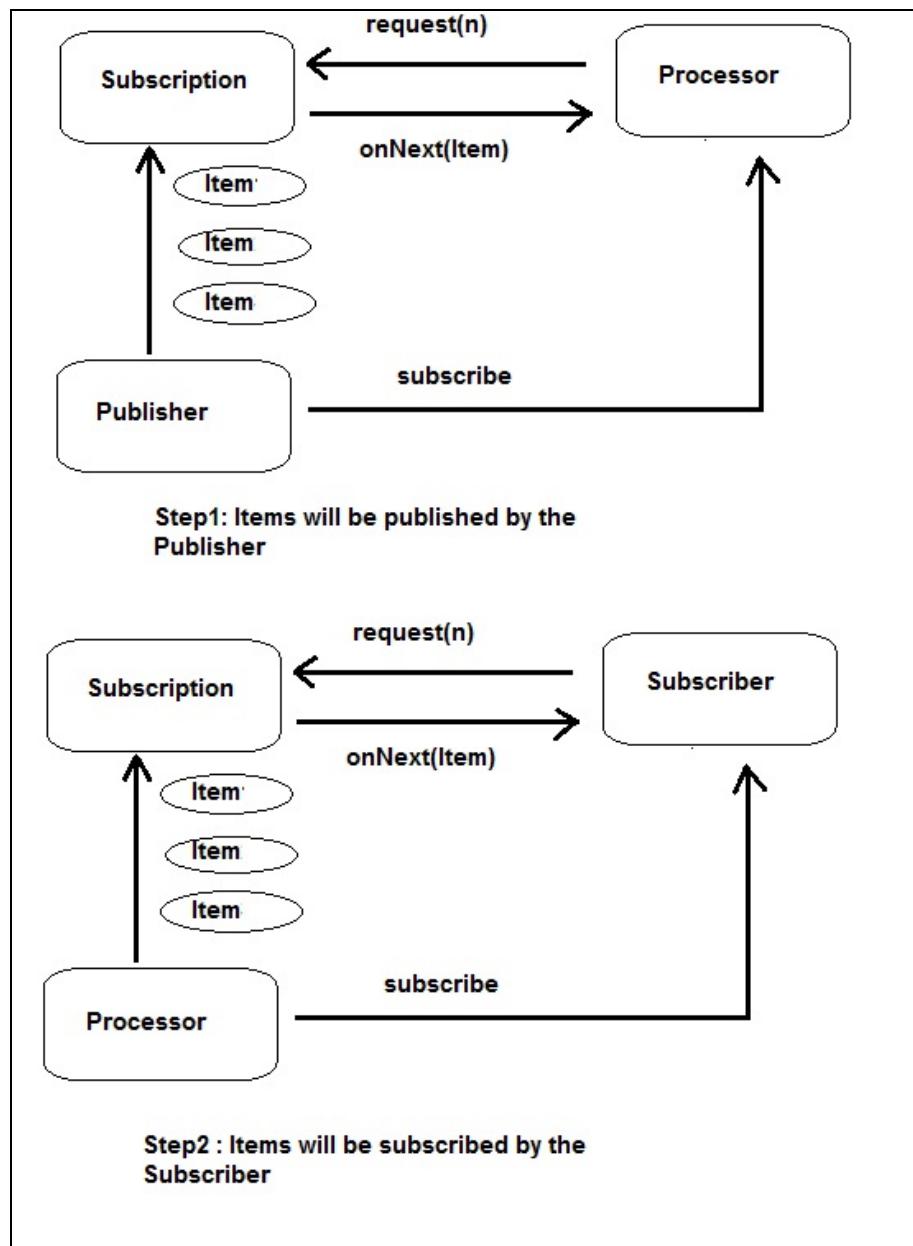
TCK is the standard tool suite that enables developers to test the implementations.

API components

The `java.util.concurrent` package provides the APIs to the developers to write reactive applications and provide interoperability among implementations. The following table describes the API components that provide Reactive Stream implementations:

Components	The API	Description of the components
Publisher	interface <code>Publisher<T></code>	This publishes a sequence of elements according to the demand coming in from the consumer.
Subscriber	interface <code>Subscriber<T></code>	Every <code>Subscriber</code> demands for elements to consume from the <code>Publisher</code> .
Subscription	interface <code>Subscription</code>	Subscription is the communication between the <code>Publisher</code> and the <code>Subscriber</code> . This communication happens in both the situations--when the elements are requested by the <code>Subscriber</code> and when the <code>Subscriber</code> no longer requires any elements.
Processor	interface <code>Processor <T, R></code>	The <code>Processor</code> is a processing stage that is obeyed by both the <code>Publisher</code> and the <code>Subscriber</code> .

The following diagram describes the flow and overall working of the reactive streams:



The preceding diagram describes the publishing and subscribing of the items as follows:

- The **Publisher** publishes the items. It pushes the items to **Processor**. The **Processor** pushes the items to **Subscriber** using the `Subscriber::onNext()` invocation.
- The **Processor** requests for publisher items using the `Subscription::request()` method invocation. The **Subscriber** requests the items from **Processor**.

Reactive in and across JVM

We all are well aware of the threads in Java to handle concurrency. We also know the `Thread` class and the `Runnable` interface both facilitate in achieving concurrency. I will not go into further details of how to create a `Thread` and the precautions to take while dealing with it. Though, the developers create and maintain threads, this has the following disadvantages:

- Each time we create a new thread, it causes some performance overhead
- When the application has too many threads, it hampers the performance because the CPU needs to switch between the created threads
- Controlling the number of threads may cause problems, such as running out of memory

Improved performance while handling threads, is provided by the `java.util.concurrent` package. Java 5 provided the Executor framework to manage a pool of worker threads. The work queue in a thread pool holds the tasks waiting for their turn for execution. The Executor framework has the following three main components:

- The `Executor` interface that supports the launching of new tasks
- The `ExecutorService` interface, which is a child of `Executor`, that facilitates the managing of the life cycle of the individual tasks and of the executor
- `ScheduledExecutorService`, which is a child of `ExecutorService`, supports the periodic execution of tasks

`ExecutorService` facilitates task delegation for an asynchronous execution. The current working thread delegates the task to `ExecutorService` and continues with its own execution without being bothered about the task it delegated.

We can create `ExecutorService` in many different ways, as shown here:

```
ExecutorService service1 = Executors.newSingleThreadExecutor();
ExecutorService service2 = Executors.newFixedThreadPool(5);
ExecutorService service3 = Executors.newScheduledThreadPool(5);
```

Once the `Executor` service is created now, it's time for task delegation.

`ExecutorService` provides different ways to accomplish task delegation, as discussed in the following list:

- `execute(Runnable)`: This method takes an instance of the `Runnable` object and executes it asynchronously.
- `submit(Runnable)`: This method takes an instance of the `Runnable` object; however, it returns an object of `Future`.
- `submit(Callable)`: This method takes an object of `Callable`. The `call()` method of `Callable` can return a result that can be obtained via an object of `Future`.
- `invokeAny(...)`: This method accepts a collection of `Callable` objects. As opposed to `submit(Callable)`, it doesn't return an object of `Future`; however, it returns the result of one of the `Callable` objects.
- `invokeAll(...)`: This method accepts a collection of `Callable` objects. It returns the result of the list of `Future` objects.

We just discussed the `ExecutorService` interface; now let's discuss its implementation.

ThreadPoolExecutor

`ThreadPoolExecutor` implements the `ExecutorService` interface. `ThreadPoolExecutor` executes a subtask using one of the threads obtained from its internal pool of threads. The number of threads in the thread pool can vary in number, which can be determined using the `corePoolSize` and `maximumPoolSize` variables.

ScheduledThreadPoolExecutor

`ScheduledThreadPoolExecutor` implements the `ExecutorService` interface, and as the name suggests, it can schedule tasks to run after a particular time interval. The subtasks will be executed by the worker thread and not by the thread that is handling the task for `ScheduledThreadPoolExecutor`.

We sublet the task to multiple threads, which obviously improves the performance; however, we still face the problem that these threads cannot put the completed task back to the queue so as to complete the blocked tasks. The Fork/Join framework addresses this problem.

The Fork/Join framework

Java 7 came up with the Fork/Join framework. It provides an implementation of the traditional thread pool or thread/runnable development to compute a task in parallel. It provides a two-step mechanism:

1. **Fork:** The first step is to split the main task into smaller subtasks, which will be executed concurrently. Each split subtask can be executed in parallel, either on the same or different CPUs. The split tasks wait for each other to complete.
2. **Join:** Once the subtasks finish the execution, the result obtained from each of the subtasks can be merged into one.

The following code creates a pool of Fork Join to split the task into subtasks, which will then be executed in parallel, and combines the result:

```
| ForkJoinPool forkJoinPool = new ForkJoinPool(4);
```

In the code, the number 4 represents the number of processors.

We can learn the number of available processors using the following code:

```
| Runtime.getRuntime().availableProcessors();
```

The Collection framework internally uses ForkJoinPool to support parallelism . We can obtain it as follows:

```
| ForkJoinPool pool = ForkJoinPool.commonPool();
```

Alternatively, we can obtain it this way:

```
| ForkJoinExecutor pool = new ForkJoinPool(10);
```

We discussed various ways to achieve parallelism while working with Reactive Programming. The combination of the classes we discussed, and the Flow APIs, help in achieving non-blocking reactive systems.

Summary

In this chapter, we discussed the traditional approach of synchronous application development, its drawbacks, and the newly added Reactive Programming approach that enhances the performance of applications with the help of parallel programming. We also discussed asynchronous Reactive Programming and its benefits in depth. The whole story actually starts with the problem of handling an enormous amount of data to and fro. We discussed an application that will not only handle the data, but also demand the processing and manipulation of data as well. Streams is a solution that provides a handy and abstract way to handle a huge amount of data. Streams is not the ultimate solution as we are not talking about data only, but about data in motion. This is where our Reactive Streams plays a vital role. We also discussed the Reactive Manifesto and its features. We continued the discussion with libraries such as RxJava, Slick, and Project Reactor as an extension for Reactive Programming. We also discussed briefly the interfaces involved in the reactive APIs in Java 9.

In the next chapter, we will discuss the application development approach of in depth, and how the approach has changed, and why adapting to the new approach is important. We will discuss these with examples to understand concepts such as declarative programming, functional programming, composition, and many more.

Programming Paradigm Shift

In the [Chapter 1](#), *Introduction to Reactive Programming*, we discussed in depth about the difference between traditional users and new generation users. We discussed about the user's demand specifically for the application performance. The discussion gave us the orientation about how the traditional approach of application development needs to be changed. The traditional approach doesn't facilitate faster execution with minimal resources due to the approach adapted for concurrent programming. Actually, we are not talking about concurrent programming. We are talking about parallel programming. I am well assured now, that you understand and are able to distinguish between these two terminologies and I can use them. Along with these we also frequently used terms such as imperative programming, functional programming, declarative programming, and programming paradigm. We haven't deliberately discussed about them in depth as in this chapter we will be concentrating around them with the help of following points:

- Functional Programming
- Value type
- Higher order functions
- Laziness
- Composition
- Declarative programming
- Operator fusion.

In mid 90s, software development was put forward as a thought to help humans in improving their efficiency. It all started with the hardware, software, operating system, and the programming language. Fortran emerged as the very first true programming language, but it didn't stop there. Then onward uncountable changes took place not only in software but also in hardware, computer languages and operating systems. I will not go into the details of hardware and operating systems, but it won't be possible to move on with this discussion without discussing languages. Don't worry we will not do the usual comparisons of languages; we will stick with our requirements to understand Reactive Programming.

In earlier days the users were happy if they just get the required data on a standalone machine. Then the users started using internet and now they started demanding the data which will be available from a remote machine. The list will continue, the bottom line is the changes in software lead to evolving new programming languages. The languages now need to cope up with hardware changes as well. The users instead of using standalone software are now willing to use it from more than one machine and have also started demanding for remote access. All these demands require the development of a new generation of languages. The story took an interesting turn when internet came in and started playing a vital role. Today uncountable languages are fulfilling different types of demands and successfully making a mark in today's competitive market. It's obviously very difficult as a programmer to study each language, but we can definitely understand their classification.

Programming paradigm

Programming paradigm is a way to classify programming languages depending upon their linguistic features. Today various programming paradigms are available considering implementation criteria, the execution model, the organization of the code, the style of the syntax and many more. I've mentioned a few of them here:

- Imperative programming
- Procedural programming
- Object-oriented programming
- Functional programming
- Declarative programming

Discussing each of them is very difficult and it's also unnecessary. Let's discuss few of them which we will be helpful for us in the discussion of Reactive Programming.

Imperative programming

Imperative programming is a very traditional approach in the software development. **Statements** are the main building blocks of imperative programming paradigm. Different statements combine together to complete a program, these statements facilitate change in the state of a program. It means, the statements in the program decide how the program will operate and how the output will be produced.

A complete program is written as a sequence of enormous amounts of statements increasing the length of the program. The length of the program can be neglected, however, a complete program in the imperative programming paradigm is just a bunch of statements. If something goes wrong, it's difficult to find the cause of the failure. This ultimately makes the program difficult to maintain.

Statements may be combined together in an appropriate sequence to perform a task, which is usually called a **procedure**. This is the basis of our next programming paradigm, procedural programming.

Procedural programming

As we already know, lengthy programs are difficult to maintain and are always prone to errors; then the idea of dividing into sub-programs arose. Procedural programming is the combination of many sub-programs, ultimately forming a complete solution. A complete program is written in imperative style, but now it is a combination of different simple and small imperative programs. Each sub program is now called a **procedure**. Many procedures combined together form a complete program.

In today's market, the required software is big in size, it is complex, and also needs to be free of bugs. Each procedure takes a task and produces an output. Each of the procedure is a small and easily maintainable unit. Whenever a shift in demand happens from the client side, developers need to change only the related procedure. It facilitates easy and maintainable application development. Procedural programming always follows an order of execution, where the execution may lead to side effects. These side effects are a major concern which we cannot ignore. They are also called bugs. Let's take a look at them in the following sections.

Avoiding bugs

Traditionally, a bug in an application is a coding error or a fault in the program. Such an error or fault causes the program to produce an unexpected output. Sometimes it can even cause serious issues with the system. Let's take a simple program which accepts two values, and after processing them, displays a color. If the first value is greater, it displays `RED`. If it's the other way round, it displays `GREEN`. We will consider three cases to observe the output:

- The first value is greater than the second value
- The second value is greater than the first
- Both values are the same

The following shows the code to display the colors:

```
public void showColor(int value1, int value2){  
    if(value1 > value2){  
        System.out.println("RED");  
    }  
    else if(value1 < value2){  
        System.out.println("GREEN");  
    }  
}
```

What will the output of the preceding code be if we pass `10` and `20` as two numbers? We will get the output as `GREEN` on the console.

Now, what will happen if we pass the value as `10` and `10`. What will the expected output be? The developer hasn't considered this condition of passing the same number. It's a flaw in the code, a *bug*. Yes, you are very correct. The bug in the code is we haven't considered what to do if both values are the same.

Bugs in an early development stage of an application are unavoidable, but at the same time, clients will not accept an application which has bugs. How to avoid these bugs? One very easy way to avoid bugs is to check the written code. Yes, test the code and change it if it goes wrong. Procedural programming makes testing easy, ultimately helping in writing applications,

which have minimal bugs.

Functional programming

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

- Martin Fowler

The preceding statement is the essence of functional programming.

A function is a basic building block of code which helps in evaluating mathematical expressions. The style puts emphasis on the expressions. In functional programming functions are the first class values, enabling them to pass as an argument to other functions. Also, they can be returned from the function as a result. In functional programming, the result of the expression after the evaluation is completely dependent upon the arguments which have been passed to the function and have not dealt with changing the state or the mutable data.

Each function performs a basic task which then may be iteratively called as and when required in building more complex behaviors. They operate on the methodology of choose, divide, and conquer to have a complete application experience. The imperative style of programming always explains how the code will be. It also says the code is mutable. The mutable code will consist of too many parts, and if any part is not treated properly, everything will be a mess. Consider writing the code for a loop. The developer needs to handle the iteration variable properly, otherwise it will lead to code with bugs. One more thing, the mutability makes it difficult to make the code concurrent.

It's easy to confuse procedural and functional programming. However, they are different. Procedural programming has statements which may lead to side effects. On the other hand, functional programming uses statements which work with mathematical expressions, often recursively. Functional programming compared to procedural programming always returns the same value for the given input. The function executes each time as if it is executing

for the very first time. It never considers what happened in the earlier execution. It helps in giving better performance, as it operates on the values provided to it as an argument and not on the outside values. The impure functions refer to the external state; they can change the external state. Consider the `println()` function and the `sqrt()` function, which every developer knows very well. The `println()` function can refer to the external state. The `sqrt()` function does not change anything in the external state. It just takes an argument and performs its manipulation. A purely functional program always yields the same value for its inputs. Also, the order is not well-defined, as it is best suited for parallel programming. It all leads to functional programming without side effects.

Functional programming is preferred when the performance of the system and the integrity are both very important. Functional programming enables doing the expected behavior in a shared environment over a network of computers.

One more very important thing about functional programming is its support for lazy evaluation. You may have heard this terminology while working with Hibernate. What has laziness got to do with a programming paradigm? Wait! Very soon we will discuss in depth about lazy evaluation.

Java-based programming languages such as Scala, Groovy, and Clojure are the best examples of languages supporting functional programming.

Need for functional programming

Today's computers are with advanced hardware such as multi-core processors. The increase in the number of cores facilitates running the code on more than one processor. It leads to enhancements in the software design to allow the developers to write code in such a way that they can make use of all processors without letting any of them stay idle for too long, which will result in quicker outputs.

Concurrency

Concurrency is the ability to run multiple programs or parts of a program in parallel. The time-consuming tasks in a program can be performed in parallel or asynchronously, allowing developers to improve performance. Modern computers have multi-core processors. Taking advantage of multi-core systems ultimately helps in improving performance. Concurrency executes multiple things communicating with each other.

Java strongly supports thread programming to obtain concurrency in an application. Concurrency deals with managing access to a shared state from the threads. But, using threads in the code needs experience to handle complex situations by synchronization locks. If not thread, then how to achieve concurrency?

Parallel processing is a type of concurrent processing which facilitates concurrent processing. It allows the execution of more than one set of instructions simultaneously. It may execute the instructions on multiple systems or on multiple cores in the same system.

Functional programming facilitates the parallel programming taking advantage of the multi-core systems. Now, how to write code supporting functional programming? Let's discuss it first before moving ahead.

Along with achieving better performance with parallel programming, functional programming also has the following benefits:

- One can write less lines of code to achieve maximum things compared with an imperative style of programming.
- It allows the writing of more expressive code. The code is self-explanatory and it's like narrating a story rather than writing a complex code.
- The functional style doesn't deal with global variables which leads to writing code having no bugs.

Functional programming in JVM

We just looked at an overview of functional programming. Indeed, we haven't mentioned anything about how Java supports it. The following are the fundamental building blocks to support functional programming in Java:

- Functional interface
- Lambda expression
- Predefined Java 8 functional interfaces
- Stream API

Let's discuss these building blocks one by one.

Functional interface

The functional interface as a concept has been added in JDK8. An interface with only one abstract function is called a **functional interface**. A functional interface can be created using the annotation `@FunctionalInterface`, which preliminarily supports behavior parameterization. The following interface can be considered a functional interface:

```
  @FunctionalInterface
  public interface MyFunctionalInterface {
      public void calculate();
      public String toString();
  }
```

Hey, wait! I just said the interface will have only one abstract method, so then why does the interface have two methods? You are thinking it's not a functional interface. But, I will still say it is a functional interface. Don't get confused. The `calculate()` is the only abstract method in the interface. The `toString()` is overridden by the class object and that's why it is concrete. Now, is that clear?

Prior to JDK 8, we had `Runnable` and `Callable` interfaces with a single abstract method, which can also be considered as functional interfaces. We will discuss about writing customized functional interfaces as well as a few functional interfaces from the Java API shortly.

Lambda expressions

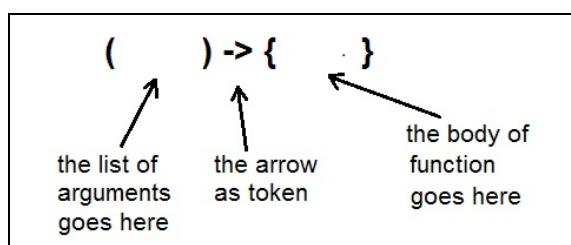
The power of functional programming has been adopted by Java in JDK 8. The JSR 335 describes the use, support for a multi-core environment, and other features related to lambda expressions. Lambda expressions have empowered developers so that they now can deal with functions with the same ease as they deal with variables. Lambda expressions are very useful to deal with collections facilitating to iterate, filter, or for extracting data.

A lambda expression is the representation of an anonymous function of the behavior. It supports the passing of the function as a parameter to achieve the behavior parameterization. The lambda expression has lists of the parameters, a return type, the body supporting the custom code and a list of the exceptions. You must be wondering, isn't it a function? Actually, it is a function, with a major thing to remember that this function is anonymous.

You must be wondering how to write the lambda expression for `MyFunctionalInterface`. Let's write the code for it.

```
public class MyFunctionalInterfaceDemo {  
    public static void main(String[] args) {  
        MyFunctionalInterface demo = (x, y) -> {  
            return (x + y);  
        };  
        System.out.println("result is:-"+demo.calculate(10, 20));  
    }  
}
```

The code will execute and will give the expected output too. But, you might be wondering how to write the lambda expression and what is the syntax? The syntax is shown in the following figure:



Here we have written a lambda expression in one way, however, there are many such ways. The following table shows a few of the examples of writing lambda expressions for different conditions:

Signature for function to write lambda expression	Expression for lambda expression
With no parameters	<code>() -> { System.out.println("hello from lambda expression");}</code>
With one parameter	<code>(name) -> { System.out.println("name is :-" + name);}</code>
With more than one parameter	<code>(val1, val2) -> (val1 * val2);</code>
For iterating through list	<code>list.forEach((num)-> System.out.println(num));</code>

There are many such conditions where lambda expressions can be used. We will be using them to write code for reactive programming.

Advantages of using lambda expressions

The following are the advantages of using lambda expressions in the code:

- It facilitates the implementation of the functional interface in a handy way
- Developers can achieve maximum things with minimum coding

Java 8 supplied a bunch of functional interfaces in the `java.util.function` package. Some of the frequently used functional interfaces from the package are `Consumer`, `Predicate`, `Supplier`. You can easily discover these on your own or continue reading as we will be discussing them one by one in upcoming pages. We will be using functional interfaces in the coming chapters.

Functional interfaces provided by JDK 8

JDK 8 is equipped with many important functional interfaces which are very useful when writing the lambda expression. These functional interfaces have been defined in the `java.util.function` package. The following table summarizes a few of the functional interfaces, however, there are many others which you can discover on your own:

Name of the functional interface	Description of the interface function
<code>Function <T, R></code>	The <code>Function</code> interface represents the function that accepts a single argument, where T is the argument accepted by the function and R is the return type.
<code>BiFunction <T, U, R></code>	The <code>BiFunction</code> interface represents a function which accepts two arguments, where T and U are the data types of the arguments and R is the return type.
<code>Consumer<T></code>	The <code>Consumer</code> interface represents a function which accepts a single argument and won't return any result.
<code>Predicate<T></code>	The <code>Predicate</code> interface presents a function which has a single argument of type <code>Boolean</code> .

Supplier<T>	The <code>Supplier</code> represents a supplier of the obtained result.
UnaryOperator<T>	The <code>UnaryOperator</code> represents an operation on a single operand which produces a result having the same data type as that of the operand.

The list could be continued further. But, here, we will take a look at the details of how to use the functional interface.

- Create a new Java project by providing the name as `Java8_Functional_interfaces`.
- Create a class `DemoBifunction` along with a main function.
- Now declare a `BiFunction` which will take two integer arguments and return the value of type `String`. You can perform any operation on these two arguments. Here we are adding them. The code is shown as follows:

```
public class DemoBifunction {
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        BiFunction<Integer, Integer, String> bifunction=(no1,no2)->{
            return "addition:-"+(no1+no2);
        };
        System.out.println(bifunction.apply(10,20));
    }
}
```

- Execute the preceding code to get the output.

Let's now find out how to use `Consumer`:

- Create a class `DemoConsumer` with a method accepting a list of numbers and `Consumer` as an argument, as shown in the following code:

```
public class DemoConsumer {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Consumer<Integer> consumer=num-> System.out.println("num:-"+num);
        List<Integer> data=Arrays.asList(144,256,4,55);
        printNumber(data,consumer);
    }
}
```

```

public static void printNumber(List<Integer> data,
Consumer<Integer> consumer) {
// TODO Auto-generated method stub
for(Integer num:data)
consumer.accept(num);
}

```

The `accept()` method takes an input and returns no value. In the preceding code, it is accepting each iterated number and then uses the definition provided by the lambda expression to give the output.

It has one more method `andThen()` which is a default method. The method takes another `Consumer` as an input and returns a new `Consumer` as the result. It is useful in executing action on the result given by one consumer.

Let's take a look on the use of `Predicate`:

- Create a class `DemoPredicate` with a main function. As usual we will consider a list and using `Predicate` we will print all the numbers which are divisible by 2. Observe the following code:

```

class DemoPredicate {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Predicate<Integer> predicate=num->num%2==0;
        List<Integer> data=Arrays.asList(144,256,4,55);
        for(Integer num:data)
        {
            if(predicate.test(num))
            {
                System.out.println(num);
            }
        }
    }
}

```

The `Predicate` has `test()` as abstract method. Along with it, the `Predicate` provides `filterList()` as a static method which facilitates looping through the list along with the condition for filtration. It will return the list of the items matching to the condition provided. It also has three default methods `and()`, `or()`, and `negate()` allowing developers to perform operation resulting in Boolean value.

Method reference

As we are from an object background, creating a reference of the class is not a very big deal for us. Now, we are treating functions as first class citizens. This means we need to understand the function reference also. The function reference is a simplified form of lambda expressions. The reference of the static method can be written as `name_of_the_class :: name_of_the_method`. If the method is non-static, the reference can be written as `name_of_the_instance::name_of_the_method`.

Let's take a look at how to create a reference for the methods. Here, we will write code to iterate over a list of fruits and display them. Consider the following steps:

- Create `DisplayData` as a functional reference, as shown in the following code:

```
|| @FunctionalInterface
|| interface DisplayData {
||     String write(String item, Object... arguments);
|| }
```

The interface has a `write()` method accepting two arguments:

- Create a class `DemoMethodRef`, which has a method `display()` that will iterate over a fruit list, as shown in the following code:

```
|| public class DemoMethodRef {
||     public static void display(List<String> fruits,
||         DisplayData displayData) {
||         for (String item : fruits)
||             System.out.println(displayData.write("data:-"+item,
||                 item));
||     }
|| }
```

There is nothing special about the function and we have already done with it. The surprise is yet to come. Just wait and watch!

- Now, add a main function in the class created in the earlier step. Here,

we will create a list of the fruits and will invoke the display function using a lambda expression, as shown:

```
public static void main(String[] args) {
    List<String> fruits = Arrays.asList("apple", "mango",
        "Lichi", "Strawberry");
    display(fruits, (fmt, arg) -> String.format(fmt, arg));
}
```

The second argument expects the instance of the `DisplayData` type. Instead of creating an implementation class or an inner class followed by creating their instance, we just did everything together. We create an implementation whose body passes the arguments to the `java.lang.String.format()` method.

- Now update the main function by adding the following statement:

```
display(fruits, String::format);
```

Have you observed something different? Observe the second argument of the display function. We passed just a written method reference of the `java.lang.String.format()` method. Hey, what's this? We are expected to pass the reference of `DisplayData`. No we are referring to a static method using the name of the class.

Let's do one more thing to understand the navigation from a lambda expression to a method reference. A few steps earlier, we did a demo for `Consumer`. We created consumer as:

```
Consumer<Integer> consumer=num->System.out.println("num:- "+num);
```

The `println()` is a static method so we can create a reference for it as well; the updated code will be shown as follows:

```
Consumer<Integer>consumer=System.out::println;
```

We are discussing many things just to understand functional programming. But, you may be wondering why functional programming? Why is it so important? We are slowly discovering many helpful things as developers. However, somewhere we just left the main aim behind, which is parallel programming. Hope you remember! Let's take the discussion a bit further and

find out about Streams API. JDK 8 provided supporting parallelism by including Streams API. It took advantage of the Java Fork Join Framework supporting many parallel operations performing processing operations.

Streams API

The Streams interface has been defined in the `java.util.stream` package. As we know, a Stream is a sequence of elements from any source. The best source of a Stream can be data coming from a collection. Stream operations can be chained to perform one operation after another, usually called **pipelining**. Pipelining allows developers to carry out multiple operations with shorter code. We will discuss this with the help of an example to understand it better.

Let's take an example, where we will obtain a Stream from a list of employees. We will filter the employees according to their names. We will try to incorporate a lambda expression, method reference, and Streams API. You can follow these steps:

- Create a **Plain Old Java Object (POJO)** class `Student` in the `com.packt.ch02.pojo` package with `name`, `rollNo`, and `total_marks` as data members.
- Add a default and parameterised constructor to it. Also, create getter and setter methods. Override the `toString()` method. The code will be shown as follows:

```
public class Student {  
    private String name;  
    private int rollNo;  
    private int total_marks;  
  
    public Student() {  
        // TODO Auto-generated constructor stub  
        this.name = "no name";  
        this.rollNo = 0;  
        this.total_marks = 0;  
    }  
  
    public Student(String name, int rollNo, int total_marks) {  
        // TODO Auto-generated constructor stub  
        this.name = name;  
        this.rollNo = rollNo;  
        this.total_marks = total_marks;  
    }  
  
    @Override
```

```

    public String toString() {
        // TODO Auto-generated method stub
        return name + "\t" + rollNo + "\t" + total_marks;
    }
    //getter and setter
}

```

- Create a `DemoStreamAPI` as a class.
- Add to it a main function and create a list of `students`, shown as follows:

```

public class DemoStreamAPI {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<Student> students=new ArrayList<Student>();
        students.add(new Student("A",1,89));
        students.add(new Student("B",12,80));
        students.add(new Student("A1",10,67));
        students.add(new Student("C",7,56));
        students.add(new Student("A",5,90));
    }
}

```

- Now, let's obtain a `Stream` from the list by:

```
Stream<Student> stream=students.stream();
```

- Let's print the data about the students from the list. We will not use external iteration, but we will do it using internal iteration. Streams API has `foreach()` method facilitating the iteration. For every iteration, we want to take some action and an action needs a `Consumer`. We will create a simple consumer, which will print the consumed values. Find the code as follows:

```

Stream<Student> stream=students.stream();
Consumer<Student> consumer=System.out::println;
stream.forEach(consumer);

```

The `foreach()` is the terminal operation. The API has provided a few more terminal operations, which we will discuss in upcoming chapters as well:

- When we execute the code, all the data from the list will get printed.
- Now, it's time to perform some operations on the data. Let's change all the names to lowercase and observe the output.

```
Function<Student, String> fun_to_lower=
```

```
    student->student.getName().toLowerCase();
    students.stream().map(fun_to_lower).forEach
    (System.out::println);
```

- We got the names in lowercase. Whenever developers perform multiple operations on the same collection of data, instead of performing operations separately, they can choose to chain the operation. It's just pipelining the operations.

Now, let's perform pipelining of the functionalities to find all the names starting with A and display them on the console. The code is shown as follows:

```
    students.stream().map(Student::getName).filter(name->
    name.startsWith("A")).forEach(System.out::println);
```

Characteristics of functional programming

The following are the characteristics of functional programming:

- Immutable state
- First class citizens
- Lazy evaluation
- Higher order function
- No side effects
- Function composition

There are many new terminologies and each one needs to be discussed in depth. So, let's start:

- **Immutable state:** It's just stateless! We all deal with objects every now and then in OOPs applications. We are also very much familiar with the state of an application and how it gets changed. A state changes usually in the applications while performing some tasks which update the values of the data members. Nothing wrong with it. But, now think about writing an application which handles synchronization along with an object. Oh! Now we are in a bit of trouble. We need to handle the change in state of an object properly using synchronization. If the state of an object can't change, the object is said to be immutable. In functional programming, the change in the state occurs by the series of the transformations, but this transformation is of events. These transformations change the state of an application while keeping the object immutable.
- **First class citizens:** The functions are first class citizens. Java is an OOP language which keeps objects in the center of attraction while writing the code. Our complete programming revolves around the objects. In OOPs programming, performing an action is the invocation of a method on an object. The series of such method calls facilitates the developers to get the expected output. In functional programming, it's

not about objects, it's actually about functions. The code is about communication between the functions instead of method invocation on the objects. It changes the focus from objects to the functions and makes functions as one of the value types which you can handle for arguments of a function or returning from a function. We already had a discussion about method references. In a demo, we also had to create a reference for the `format()` method of String class and invoke it as a `callback` method. It means we are treating functions as data members.

- **Lazy evaluation:** Lazy evaluation is also known as **call by need**. It's an evaluation strategy which allows the delay in the evaluation of an expression. The lazy evaluation suggests the function will not be executed until the value of the evaluation is needed. It's been beautifully done in Haskell where only necessary values will be computed. In Java, we have iterators, which facilitate iterating with a collection of items without knowing how many items to iterate. The `Iterator` discovers a number of iterations at runtime. You can go through the demo `DemoStreamAPI`, which deals with the Streams API. We perform `foreach()` as the terminal operation. The best thing about the terminal operation is they are lazy. The intermediate operations will not take place unless the terminal operation is invoked. It becomes a major advantage when we are processing an enormous amount of data. It increases the performance drastically.

The advantages of the lazy evaluation are as follows:

- It increases the performance of an application, as it avoids unnecessary calculations and conditions leading to the errors.
- It facilitates the developers writing code flexibly without bothering about the boundaries within which the manipulation will be done. The boundary will be decided at runtime.
- It allows defining the abstract structures.
- It has an ability to create a calculable list, but without writing infinite loops, like creating streams.
- The temporary computations and the conditions have been discarded, which reduces the time in executing the algorithm.

Consider the following code snippet to check if length of an input

string is greater than five or not:

```
public class DemoLazyFunction {  
    public static boolean evaluateMe(String data){  
        System.out.println("evaluation");  
        return data.length()>5;  
    }  
}
```

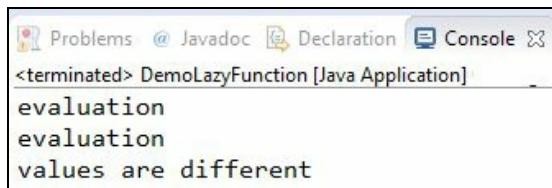
The following code is about checking if results are equal or not using short circuiting of `&&` operator.

```
public static String matchMe(boolean result1, boolean result2){  
    return result1&&result2?" both values are equal" :  
        "values are different";  
}
```

Let's write the main function shown as follows:

```
public static void main(String[] args) {  
    String output= matchMe(evaluateMe("one"), evaluateMe("three"));  
    System.out.println(output);  
}
```

The output of the preceding code will be as shown in the following screenshot:



The screenshot shows the Eclipse IDE's Console view. The title bar says 'Problems @ Javadoc Declaration Console <terminated> DemoLazyFunction [Java Application]'. The console output is:
evaluation
evaluation
values are different

Let's now add a `Supplier`, which is a functional interface, and modify the code as done in the following:

```
public static String matchMeLazy(Supplier<Boolean>length1,  
    Supplier<Boolean>length2){  
    return length1.get() && length2.get()?" both values are equal" :  
        "values are different";  
}
```

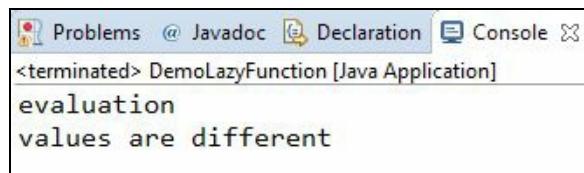
The preceding function `matchMeLazy()` can be invoked from the main function as follows:

```
public static void main(String[] args) {  
    System.out.println(matchMeLazy(()->evaluateMe("abcd"),
```

```
|     ()->evaluateMe("a")));  
| }
```

In the code, `Supplier` is a functional interface which facilitates its use as the assignment of the lambda expression.

The output of the preceding code will be:



The screenshot shows the Eclipse IDE's Console view. The tabs at the top are 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is selected. The output window displays the following text:
<terminated> DemoLazyFunction [Java Application]
evaluation
values are different

The output clearly shows the unnecessary function calls have been dropped, which we discussed earlier. Though Lambda expression provides us flexibility, at the same time we face a few challenges while dealing with it. Lazy evaluation increases in the space complexity while executing the algorithm, and finding the expression before the execution is a bit difficult.

- **Higher-order functions:** The **higher-order function (HOF)** is a special function which has one of the following characteristics:
 - It must take one or more functions as its argument
 - It must return a function as its result
 - A function can be created within another function

In OOPs programming, the data is treated as a first class citizen. The data may be of primary type or secondary. We use them in manipulation, expression, and quite frequently in the function as arguments or as return types. It's a new concept added in JDK 8 of creating a reference of a function so as to facilitate treating them as good as data. Now, we can create the reference of a function and deal with a function as an argument. Some of you will have easily understood what I am trying to say, but those who haven't dealt with JDK 8 features will find it weird. It is not weird; it's just different until you attempt it. Once you start, you will be amazed at how easy, yet powerful it is. Functional programming supports writing the HOF. So, let's start by taking an easy example and modify it for HOF.

Let's take an example to understand the concept. Consider the following code, where the `show()` method is displaying the length on the console:

```
public class ShowLength {  
    public void show(int length){  
        System.out.println("length is:-"+length +" cm");  
    }  
    public static void main(String[] args) {  
        ShowLength showLength=new ShowLength();  
        showLength.show(100);  
    }  
}
```

The output of the code is quite predictable. Yes, on the console, a line will be displayed as:

```
| length is:-100 cm
```

Have we done something different? No, we haven't. But, now we will modify the code to convert the length from centimeters to meters or from centimeters to kilometers. A better way will be to add a new class `LengthConverter` with methods where each method will help in conversion. You can get the details by looking at the following code:

```
public class LengthConverter {  
    public String toMeter(int length) {  
        return length / 100.0 + " meter";  
    }  
  
    public String toKilometer(int length) {  
        return length / 1000.0 + " kilometer";  
    }  
  
    public String convert(int length, int unit) {  
        if (unit == 1) {  
            return length / 100.0 + " meter";  
        }  
        else if (unit == 2) {  
            return length / 1000.0 + " kilometer";  
        }  
        return length + " cm";  
    }  
}
```

Now, as a traditional developer, we can choose an easy way to create an object of it and invoke it from the function. Instead of choosing it, we will go with functional programming and being specific, we will go with HOF.

The following code from the `ShowLength_new` class has a `show()` method with one very new member--a function. Yes, the argument is of Java type which the `show()` function is accepting as an argument. Obviously, we do have the right to call `show()` as a higher order function:

```
public class ShowLength_new {  
    // function will accept integer and return string  
    public void show(int length, Function<Integer, String> converter){  
        System.out.println(" the length is " +  
            converter.apply((length)));  
    }  
}
```

Now let's add the main code, which will do the following:

- Create objects for the `ShowLength` class and `LengthConverter` class
- Invoke the `show()` method by passing the reference of the `toMeter()` method of the `LengthConverter` class

The code will be as follows:

```
public static void main(String[] args) {  
    ShowLength_new length_new = new ShowLength_new();  
    LengthConverter converter = new LengthConverter();  
    length_new.show(100, converter::toMeter);  
}
```

The `converter::toMeter` means we are passing the reference of the `toMeter()` method to `show()` method. It means whenever `show()` will get invoked, `toMeter()` of `converter` will also get invoked. Isn't it simple?

- **No side effects:** While discussing functional programming, we had written a number of functions and lambda expressions. We processed the data using Stream APIs, however, we never modified the state of an object. Also, we haven't dealt with global variables. Actually, we are writing pure functions. A function is called as a **pure function** when the function returns the same values without any side effects. Here, side effects mean changing the arguments or performing any output operations. It means that, for the developers, the data can be freely shared which ultimately improves the memory requirements and parallelism. The pure function produces the same result for the same arguments, though the invocation is done multiple times. The code

doesn't lead to any change outside of its boundaries. Also, any change outside of its boundaries doesn't have any effect on it. This function accepts the arguments, performs the actions on them, and produces an output. It's easy to understand the behavior, so as to predict its output.

- **Function composition:** We all are aware of creating small maintainable modules to divide a large code into small manageable units. Each such unit, which is usually called as a function, performs its task. These small functions will be reused and combined to create a composed structure of a new function is called as **function composition**. A function composition combines one or more functions to get the desired end effect. You may call it as **function chaining** as well, where one function calls another function. Before moving ahead, let's take a look at the following code to get rid of the doubts about function composition. The `calculateAverage()` method calculates the average and gives the result. The probable code will be shown as follows:

```
public double calculateAverage(int x, int y) {  
    return ((x + y) / 2.0);  
}
```

Can we improve the code? Yes! Instead of using a single function to do both addition and average calculate, we can create two modules. The first `addMe()` will perform the addition and `calculateAverage()` will deal with the calculation of the average, as shown in the following code:

```
public int addMe(int x, int y) {  
    return x + y;  
}  
public double calculateAverage(int x, int y) {  
    return addMe(x, y) / 2.0;  
}
```

Is there any alternative for calculating the average? There is. Observe the following code:

```
public double calculateAverage(int res) {  
    return res / 2.0;  
}
```

The probable function call will be:

```
calculateAverage(addMe(x, y));
```

The `addMe()` function acts as an argument for `calculateAverage()`. And this is the idea behind function composition, which says the output of one function acts as an input for another function.

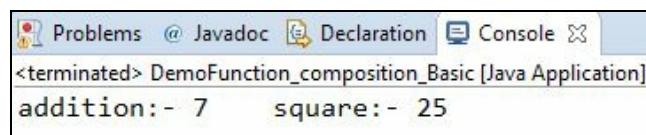
Most of the languages support function composition, specifically when functions are their first class citizens. Function composition handles finite amount of data, which will be sequentially processed to obtain the final desired result.

We already had done this in Stream programming. Stream programming uses more than one operator in a chain. Each of the operator helps in filtering the data to achieve the desired result. Such a chain or a **pipeline** is very much similar to the function composition. Java 8 provides a function interface which facilitates the writing of function composition easily. Observe the following code in which `square` is a reference to a function which deals with calculating the square of a number. `add()` is a reference to a function dealing with performing the addition:

```
public class DemoFunction_composition_Basic {
    public static void main(String[] args) {
        Function<Integer, Integer>add =number ->number+2;
        Function<Integer, Integer>square =number ->
            number*number;

        Integer value_add=add.apply(5);
        Integer value_square=square.apply(5);
        System.out.println("addition:- "+value_add +
            "\tsquare:- "+value_square);
    }
}
```

The output of the code is as follows:



```
Problems @ Javadoc Declaration Console
<terminated> DemoFunction_composition_Basic [Java Application]
addition:- 7      square:- 25
```

Now, what if the requirement is to calculate the square of a number after adding 2 to it? You may have figured out, we are willing to combine the two functions which we have written in earlier code. The code can be:

```
Integer value_add_square=square.apply(value_add);
System.out.println("value of square of addition:- "+
```

```
|     value_add_square);
```

Is there a better way? Yes, certainly. We get `compose()` and `andThen()` as two functions which facilitate elegant function composition. But, both these work very differently, so we need to use them wisely. The following code will demonstrate the difference:

```
| Integer result_composition=add.andThen(square).apply(5);  
| System.out.println("value of square after addition:- "+  
|     result_composition);
```

This will give the output as 49.

It means the operation of the invoking function is done first, followed by the operation in the invoked function. The output of the first result will be the input for the second operation.

Now, let's consider another piece of code:

```
| Integer result_composition1=add.compose(square).apply(5);  
| System.out.println("value of addition after squaring:- " +  
|     result_composition1);
```

The output of the preceding code will be 27.

It is now clear, that the operation of the invoked function is done first, followed by the operation in the invoking function.

Declarative programming

Declarative programming is one of the very much in trend paradigms to build the structure of a program, where the logic of computation is expressed without going into the details of the control flow. It is totally the opposite of the imperative and procedural styles of programming which emphasize implementing the algorithms. Functional programming supports the declarative style of programming and simplifies writing the programs which support parallel programming.

The declarative paradigm doesn't state how to solve a problem. However, it just states what the problem is. In declarative programming, a very important thing is to set some rules to match. Depending on the rules, the program continues its execution. When the rule matches, the operation executes, which is opposite to procedural programming, where an order of execution is followed. It doesn't follow any specific order of implementation. Many of the programming languages use it to minimize the side effects which occur while describing how the program must achieve the result.

Many of the markup languages, such as HTML, XML, and XSLT achieve the end result without getting involved into the complexity of how to achieve the end result step by step.

Let's consider HTML. We design the web page using HTML which describes what to display to the user like the font, color, layout of the text or size, and location of the image and many such things from a presentation perspective. HTML always gives static contents. The HTML pages may contain some forms where the user can fill in the information and submits it. After submission what to do is not under the control of the user. The HTML never specifies the control flow by which the page will be rendered or what to do if the user applies some action on it. For taking up the action, we need to use the scripting language.

Each application provides a solution to the problem. The developers develop an application keeping this problem in mind. Their main aim is to solve the

problem and provide a solution. The aim of an application development is to solve the problem is called as a **core concern**. The code needs to be managed which is not the main concern but, we cannot neglect it. Such a code is called as a **cross cutting concern**.

Transaction management, logging, and security are a few examples of cross cutting concerns. Usually, such codes are scattered throughout the application, making it more complex. The code is not only complex but also repetitive. Such repetitive code in declarative programming is taken out, enabling the developers to be concerned about their main concern of application development. The code handling cross cutting concern will be written at some centralized location along with the rules of when and how to apply. Ultimately, the application development becomes more easy and concentric. We all are very much familiar with Servlet programming. How to handle the URL is written in a `web.xml` file and not scattered in Java code. Now, we even apply annotations which just specify what to do and not how to do it. Those who have developed AOP using Spring can relate to the complexity solved using declarative programming for cross cutting concerns.

Characteristics of declarative programming

Let's look at the characteristics:

- Set of rules, equations, or definitions are set up, which specify what to compute and not how to compute.
- The order of execution doesn't matter.
- The flow control of the program cannot be predetermined, as it depends upon the input, set of rules. We can say the programmers are least responsible for the flow control.
- It defines the logic and not the control.
- The expressions can be used to set up the rules, which will be used as values in the program.

Advantages of declarative programming

Following are a few of the advantages of the declarative programming:

Increase in the readability: Less lines of code, along with less plumbing of code makes it more readable and easy to learn.

Less complex code: The code is being written at one place and then applied to multiple locations. The boilerplate code which makes coding more complex is being replaced by the abstract code so as to make it less complicated.

Reusability: The created code for transaction management and logging can be written at one place and applied to the complete application depending upon the set of rules. As the code is located independently, instead of writing the same code again and again it becomes easier to be applied on multiple matching points. Let's consider the following code to understand the difference between imperative and declarative programming in a better way. The code we are writing has the main function as building a list of fruits and pass it to the `matchCount()` function. The `matchCount()` function is iterating over a list of fruits. Each iteration tries to find whether the given fruit name is matching or not. If the match is found, we will increment the counter and once the iteration completes, we will return the last updated value and print it on the console. You can refer to the following code:

```
public class DemoImperative {  
    public static void main(String[] args) {  
        DemoImperative demo=new DemoImperative();  
        List<String>names=new ArrayList<>();  
        names.add("apple");  
        names.add("grapes");  
        names.add("apple");  
        names.add("mango");  
        names.add("chickoo");  
        names.add("strawberry");  
        intcount=demo.matchCount(names, "apple");  
        System.out.println("the counter is:-"+count);  
    }  
}
```

```

public int matchCount(List<String>names_input, String name)
{
    int count=0;
    for(String n:names_input)
    {
        if(n.equals(name))
        {
            count++;
        }
    }
    return count;
}
}

```

The output on console will display the counter is :- 2. I know it's not at all new, and we are doing similar code every now and then. It's imperative style of coding.

Now, let's improve the code to adopt declarative programming. We will keep the code the same except the `matchCount()` function shown as follows:

```

public int matchCount(List<String>names_input, String name)
{
    Int count= Collections.frequency(names_input, name);
    return count;
}

```

Oh! What did we just do? Nothing. Still, we will get the same output as that of the previous code. The code has less number of variables, so less clutter. It's easy to understand and maintain. The developers are more than happy, as they don't have to get involved in any iteration and matching. Still they achieve the same result. You can refer to the complete code at <https://github.com/PacktPublishing/Reactive-Programming-With-Java-9>.

We just modified the code to adapt declarative style. In JDK 8 the concept of stream has been added, which facilitates using declarative style as shown in the following code:

```

public int matchCount_new(List<String>names_input, String name) {
    long count=names_input.stream().
        filter(n->n.equals(name)).count();
    return (int)count;
}

```

The preceding code is clearly demanding to count the number of elements which are matching to the given input from the list without telling how to

find and count the elements. Don't worry, we are going to discuss a lot about the code in depth very soon.

Reactive programming revolves around the data flow which may need manipulation. We do have two options to manipulate the data. One is, we can write the logic of manipulation, or second is, to use in the declarative style with the help of operators.

Operator fusion

Reactive programming has collection of operators which mainly enable the developers to handle, modify, filter, and transform the data emitted by the source. The list of operators is unending. You can also consider the operators as predefined functions, taking lots of load from the developer's shoulder. It's always great to spend some time in understanding the operator and their functionalities. Studying them is good, but unfortunately we hardly find a perfect match of the operator for the data transformation which matches the requirements. Don't worry! Single operators don't match, so let's combine them.

The developers can use two or more operators and combine them in such a way that the memory and time overhead reduces while handling the data flow known as **operator fusion**. While performing the data fusion we must keep the following things in mind:

- One of the operators may need thread safety, but perhaps not both
- A few of the operators may share their internal component
- Operator fusion reduces the call overhead, however the operators may consume or drop some of the value elements

We have already discussed Streams as a sequence of elements. The source of the elements provides the sequence, which has already been discussed. The developers may modify these elements at various stages of the lifecycle of the Stream. We have already used the `toLowerCase()` method and `startsWith()` method while discussing the Streams API in the preceding sections of this chapter. The data can be modified at assembly time, subscription time, or runtime.

For assembly level, the operator fusion can be classified as follows:

- Macro fusion
- Micro fusion

Let's discuss them one by one in depth.

Macro fusion

Macro fusion takes place at assembly time which enables replacing of one or more operators with a single operator, leading to the reduction in time overhead. The following are the ways where macro fusion can happen:

- **Replacing one operator by another operator:** As the name suggests, the applied operator looks at the upcoming Stream. Then, instead of instantiating its own provided implementation, the operator will call some different operator. Now, you may be thinking, why such a replacement of the implementation is required? Let's take an example of an array with only one element. If the array acts as a source of Stream, what operations can be applied on it? Can we merge or concat it? No, we can't. How can we do that with just a single element? It means the instantiation of the implementation is just an overhead. Instead, return the single element as provided by RxJava 1.x.
- **Replace the operator by custom operator:** The single operator is failing to fulfill our requirement, so we combine more operators together. We are doing this so that the overhead can be avoided. But, if we take a close look into the internal mechanism, we will observe generation of the internal queues and also several atomic variable getting modified. To achieve one, we unnecessarily complicate things. Instead of just performing operator fusion, we can choose to pair a custom operator. This newly created custom operator will handle the scheduling and emission of the value.
- **Replacing the operator with modified parameters:** Sometimes, a few operators can be applied multiple times. One can think about combining them and replacing them with a single operator on a stream leading to measurable overheads. You can avoid this by replacing the operator with modified parameters. In this case, the macro fusion finds whether the operator has applied the same type. Then it takes the source and the operator will be applied on it so that the parameters get combined.

Micro fusion

Micro subscription usually happens at subscription time where two or more operators share either their resources or internal structure which leads to avoid overhead. The basic idea in micro fusion is to share the queue which internally gets created at the end of one operator and starts with a new operator. The operator which ends with a queue will be shared by the next starting operator as well. The following are the ways where macro fusion can happen:

- **Conditional subscription:** Conditional subscription is useful in request management. The source emits the elements and the consumer subscribes to them. We are also aware of message communication between the source and the consumer to communicate whether to emit the next element or not. It can be simply done by providing a method with a value. The returned value will indicate whether the emitted element has been consumed or not. The conditional subscription facilitates not to increment the emission counter and keep emitting the element until the request limit has not been reached after the successful consumption.
- **Synchronous fusion:** Synchronous fusion is applied in situations where operator and the source are synchronous in nature. Also, both pretend to be a queue. The operators such as `observeOn()` and `flatMap()` use queue as it is internal. The working is very simple to understand. The elements emitted by the source are of type `Queue` and the subscriber also implements the `Queue`. Generally, while subscribing the elements, it will provide a new `Queue` implementation. But, in this case, before subscribing, the subscriber performs a check on what is coming and uses it instead of creating one more implementation.
- **Asynchronous fusion:** Asynchronous is a subscription time fusion. It is a specialized case of operator fusion where the source emits the elements on upfront and the consumer consumes them in downstream, where the timing and count of elements is unknown to the source. The synchronous mode, which we just discussed in the previous bullet point, can be downgraded to asynchronous, but vice versa is not possible.

We are equipped with all the basics and should take a ride on reactive programming. So, let's get ready!

Summary

We are interested in understanding Stream programming. Yet, we haven't discussed anything regarding reactive programming. Though we haven't directly discussed it, this chapter gave us the orientation about different terms and concepts of reactive programming which we will be using throughout the book.

In this chapter, we discussed about changes in the development approach with programming paradigm to match the market requirements. We discussed about the imperative programming, its role in the application development, and its drawbacks. We then moved on to discuss the procedural programming and its approach in development. Our main motive was to discuss about functional programming and the concepts associated with it. We discussed about higher-order functions, function composition, function as value type, function laziness, operator fusion. We discussed these concepts with simple codes instead of getting into streams. Along with the functional approach, we also discussed declarative programming.

In the next chapter, we will discuss about reactive streams introduction. We will get a deep orientation about Java 9 Flow APIs and how to use them to build a reactive application. We will also discuss about backpressure in Streams programming. Enjoy reading!

Reactive Streams

In [Chapter 2, Programming Paradigm Shift](#), we explored many useful concepts about Reactive Programming. Our main focus was on understanding functional programming and how to code using a functional approach. In [Chapter 1, Introduction to Reactive Programming](#) and [Chapter 2, Programming Paradigm Shift](#), we discussed many things but in parts. It's like a jigsaw puzzle; the picture is scattered and now it's time to arrange the parts in their places. Don't be in a hurry! From here onward, each chapter will slowly give you an orientation into how to implement Reactive Programming in your application. It's all about doing the same thing but in different ways. Let's start this beautiful journey with Java 9.

We discussed reactive concepts and how to implement them. In this topic we will take a close look at implementing reactive applications using the Java 9 Flow API with the help of the following points:

- Asynchronous event streams
- Flow API
- Backpressure
- Designing non-blocking interfaces

Discussing the Collection framework

Now, as we have been discussing Reactive Programming since [Chapter 1](#), *Introduction to Reactive Programming*, we all are well aware that it is all about data flow. However, we need to go in depth to master it. We will start by briefly discussing the Java Collection framework. Each one of us handles the Collection quite confidently with regard to handling a bunch of data. The Collection framework is just an arrangement of data according to a specific data structure. It can be as simple as an `ArrayList` or something as complex as a `HashMap`. Each provides a specific strategy for the arrangement of elements. It also restricts developers. Let's take the example of a stack. We can pull and push only the topmost element. We cannot fetch the last element or elements from any particular position. What do we do now? We find a solution to fetch elements from an external mechanism provided by the iterator. This external iteration was used till Java 8. In Java 8, the Collection APIs got enhanced to use internal Iteration. It all starts with the Streams API.

Streams in action

The moment we say Stream, we developers start visualizing I/O Streams. You are on the right track, but the only thing is such Streams only handle bytes, while Java 8 Streams can either be of type `bytes` or `object`. Consider the following code snippet, which reads a number from the user:

```
Scanner scanner=new Scanner(System.in);
System.out.println("Enter your age");
int age=scanner.nextInt();
```

If you execute the preceding code your console will show `Enter your age` as an output and the application thread is then stuck. Why? The thread got suspended because it needs input from the user to resume execution of the remaining code. Unless the user provides an input the next statement will not execute. You must be wondering, what am I trying to prove? Actually, I don't want to prove anything. I just want you to observe the *blocking code* in the code.

Let's now perform the iteration over the list with internal iteration. The code is as follows:

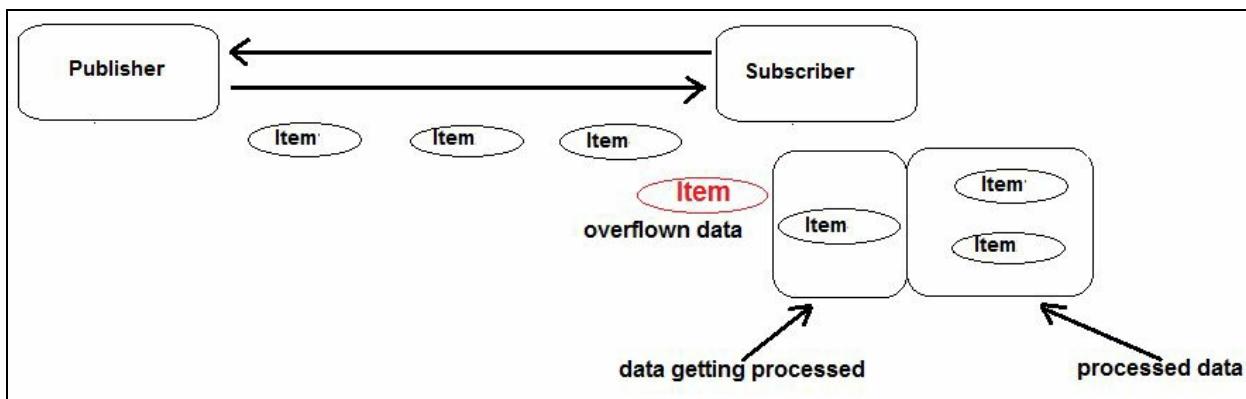
```
List<Integer> numbers = Arrays.asList(12,34,23,56,6,76,56,4,34,2);
numbers.stream().forEach(System.out::println);
```

On execution of the code, it will iterate over the list and give us the output. The output is similar to when we were writing an application using JDK 7. However, in this code, we are pulling the data from the collection. The major difference between JDK 8 and the Java 7 API is the way the code has been written. JDK 8 specifies what to do and not how to. The preceding written code doesn't have a loop iterating through the list, the code just says *Print the elements for me*. It's the power of functional programming. Are we done? No! We just moved one step ahead towards our goal. Java 8 Streams follow a functional style approach which obviously supports laziness. We have already discussed it in [Chapter 2, Programming Paradigm Shift](#). Functions such as `findFirst()` and `findAny()` facilitate short circuiting for faster response. The one thing we overlooked is asynchronous programming. Reactive

Programming is an asynchronous, non-blocking, functional approach. Java 8 Streams does not handle data asynchronously unless someone is not using `parallelStream()` instead of the method `stream()`. Don't be under the impression that we are performing parallel programming. Java 8 processes data concurrently and we know what is meant by concurrency.

We do have one more thing to consider here. Let's consider a producer and a consumer interacting with each other. In normal situations with a push-based mechanism, producers as well as consumers don't face any problem and complete the application normally. What if the consumer is performing heavy processing, which is time-consuming as well? The producer keeps on pushing items once produced and a pending workload increases on the consumer side. It leads to a thought of having pull-based systems rather than push-based. Pull-based systems keep on pulling the items from the producer when it is ready to process them.

Let's consider the following figure to get a clear idea about what is going on in Java 8 Streams:



The **Publisher** is publishing the produced items to the **Subscriber** where they will be consumed after processing. The time taken will be more if the consumer is performing heavy processing. In such cases, the time taken for production will be significantly less than the time taken in consumption. This will increase the number of items that are waiting in a queue. It ultimately builds the pressure on the consumer due to push-based systems. Here overflowing data is not the only problem. Along with that, it is a synchronous communication where we can't take advantage of asynchronous communication for parallel programming. Our multi core systems are now

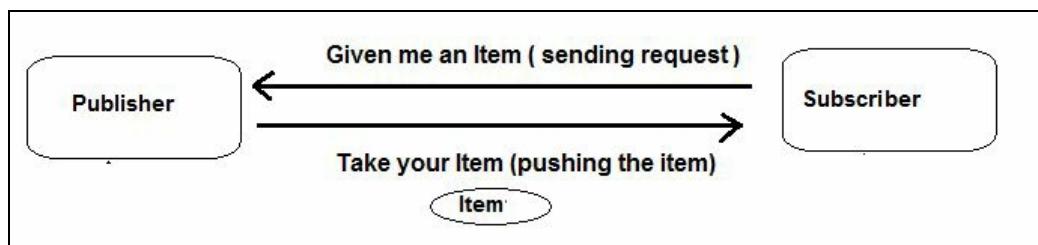
just a waste!

Now let's consider the `Future` interface. Java 5 introduced `java.util.concurrent.Future`, which represents the pending result of an asynchronous computation. It provides a `get()` method that returns the result of the computation once it is done. Sounds good but is that what we needed? No, we needed asynchronous programming but not with blocking APIs. The `get()` method blocks until the computation is completed. To our rescue came `CompletableFuture` introduced by JDK 8. It helps in building asynchronous non-blocking applications but still, those are not completely reactive. `CompletableFuture` implements `Future` and `CompletionStage` interfaces. Following are the enhancements added to `CompletableFuture` in Java 9:

- It now supports delays and timeouts to enable a future to complete in the specified duration or with an exception after a certain duration
- The support for subclassing has been improved to perform the process of creating child classes easily
- It added new factory methods that return an instance of `CompletableFuture`

`CompletableFuture` facilitates working with nonblocking, asynchronous push-based data processing. However, Flow APIs introduced in Java 9 implement the Reactive Streams specification; enabling Reactive Programming. Flow APIs also facilitates interoperability with third party libraries like RxJava, Ratpack, and Akka. Let's discuss Java 9 Flow APIs in depth.

Now, it's Java 9 which comes up with powerful Reactive Streams handled by Flow APIs supporting `Publisher- Subscriber` pattern. The APIs provide a bunch of interfaces that can be applied in concurrent as well as asynchronous distributed systems. The API uses the push-based mechanism to avoid the problems occurring in the pull-based systems. Let's take a closer look at the `Publisher- Subscriber` pattern with the help of the following figure:



The **Subscriber** will send a request for the number of items. The **Publisher** will now push the requested number of items to the **Subscriber**. **Publisher-Subscriber** is a bidirectional flow, where the **Publisher** emits the data and the **Subscriber** utilizes the data. The demand for an item has been done asynchronously. The **Publisher** is free to send as many requests as possible for its future work. It prevents wastage of resources. It is neither a pull-based nor a push-based system; rather it's a hybrid of these two. When the **Subscriber** is faster than the **Publisher** it works as push-based and if the **Subscriber** is slow it acts like a pull-based.

Delving into the Flow API

Java 9 has come up with `java.util.concurrent`. Flow APIs facilitate Reactive Streams based upon push and subscribe framework. Before we start the actual development we need to understand the basic building blocks of the system.

The building blocks

The Flow API has the following four major components acting as the backbone of the system:

- Publisher
- Subscriber
- Processor
- Subscription

Let's take a close look at each of them.

Publisher

As the name suggests, this a component which is involved in producing and emitting the data or items. The `Publisher` is a functional interface. Hope you all still remember our discussion on functional interfaces in [Chapter 2, Programming Paradigm Shift](#). The `Publisher` publishes the items asynchronously usually with the help of an `Executor`. You can observe the methods from the following code:

```
| @FunctionalInterface
| public static interface Flow.Publisher<T> {
|     public void subscribe(Flow.Subscriber<? super T> subscriber);
| }
```

The `Publisher` has a `subscribe()` method that takes an object of `Subscriber` as an input. Whenever developers want a `Publisher` they have to create their own implementation by overriding the `subscribe()` method.

Let's consider a scenario. We want to develop a `Publisher` that will emit a number of type `Long`. For a simple task, to publish the `Long` items, we may think why do we have to do so many things? Isn't there an alternative? Fortunately, we have an interesting alternative. The Java 9 APIs provide `SubmissionPublisher` as a concrete implementation of the `Publisher` interface.

The SubmissionPublisher class

The `SubmissionPublisher` class is an implementation of the `Publisher` interface whose role is to emit the data. The `SubmissionPublisher` class uses the supplied `Executor`, which delivers to the `Subscriber`. The developers are free to use any `Executor` depending upon the scenario in which they are working.

`Executors.newFixedThreadPool(int num_of_threads)` and `ForkJoinPool.commonPool` are some favorites of developers. `Executors.newFixedThreadPool` facilitates the creation of a thread pool with a fixed number of threads for the operation. Each thread in the pool will exist unless it has not been shut down explicitly. `ForkJoinPool.commonPool` is the default `Executor` used by `SubmissionPublisher`. `ForkJoinPool.commonPool` returns the instance of the common pool constructed statically. We will be using these in our application as well, so as to get the better understanding.

For each `Subscriber`, the `SubmissionPublisher` class uses an independent buffer whose size can be specified at the time of instance creation. These buffers get created on the very first use and eventually expanded to their maximum size as required. The default buffer size can be obtained from `defaultBufferSize()`. The `SubmissionPublisher` class implements the `AutoCloseable` interface. The call to the `close()` method invokes the `onComplete()` method of the current `Subscriber`.

The `SubmissionPublisher` class has the following two methods that facilitate the publishing items:

- **offer:** The method facilitates publishing items to the subscribers asynchronously by invoking its `onNext()` method without unblocking. The method may drop a few of the elements after a timeout. The developer can specify the timeout. It even facilitates specifying the drop handler, which is actually a `BiPredicate`. If the developer wants to publish the item just for a specific amount of time it's preferable to use `offer()`.
- **submit:** The `submit()` method is the simplest method that facilitates publishing items to the `Subscriber`. The `submit()` blocks the call until all

the resources are available for the current `Subscriber`. The `SubmissionPublisher` class also provides the following method which facilitates the consumption of the items.

- **consumeThis**: The `consumeThis()` method facilitates to take the action of the `Subscriber` which has requested the item. The `consume()` method will take the action provided by the argument and performs the operation. We will soon discover how to use it.

Let's take a look at how to use `SubmissionPublisher` with the help of a demo. `SubmissionPublisher` will emit the data of type `Long` and a consumer will consume it. We will do this by the following steps:

1. Create a project `ch03_DemoPublisher`.
2. Create a package `com.packt.ch03.submissionPublisher`.
3. Create a class `WelcomeSubmissionPublisher` in the package we have just created.
4. Declare a `publish()` method in the class.

We need a mechanism to generate the number of type `Long`. Either we can write our own logic for it or we can use predefined ones. We will be using the predefined `LongStream` class to generate the numbers in the specified range. The code for this is as follows:

```
|     LongStream.range(10, 20)
```

1. We want each of the generated items to be published to the `Subscriber`. We can write a separate `for` loop and go on publishing each item using the `submit` method. We have learned about lambda expression, so let's use it instead of external iteration. The following code publishes an item generated of type `Long`:

```
|     LongStream.range(10, 20).forEach(publisher::submit);
```

2. The item has been published, so where is the `Subscriber`? Let's now write the easiest `Subscriber`, which will print the elements on the console. The `consume` method helps us specify the consumer. The method returns an instance of the `CompletableFuture`. The `get()` method of the obtained instance will be useful to keep the application running till the processing of all the items is not completed. The code is as shown in the following

snippet :

```
public static void publish()throws InterruptedException,
    ExecutionException
{
    CompletableFuture<Long>future = null;
    try (SubmissionPublisher<Long>publisher = new
        SubmissionPublisher<Long>()) {
        System.out.println("Subscriber Buffer Size: " +
            publisher.getMaxBufferCapacity());
        future=publisher.consume(System.out::println);
        LongStream.range(10, 20).forEach(publisher::submit);
    }
    finally
    {
        future.get();
    }
}
```

3. Now, let's write the `main()` function to test how the items are published and subscribed is shown as follows:

```
public static void main(String[] args) throws
    InterruptedException, ExecutionException {
    publish();
}
```

4. Execute the code and you will get the following output:

```
We got the Subscriber of Buffer Size: 256
10
11
12
13
14
15
16
17
18
19
```

I know, we did a very simple job here and you are more interested in developing your own publisher. But, we still haven't discovered the other components of the system. So let's move on to the subscriber.

Subscriber

The `Subscriber` is a component that extensively deals with subscribing data or consuming data. The items will be pushed to the `Subscriber` only when the `Subscriber` requests them. A `Subscriber` can request more than an item. In a given subscription, the method invocation of `Subscriber` always follows a sequence. Take a glance at the API given as follows:

```
public static interface Flow.Subscriber<T> {  
    public void onSubscribe(Flow.Subscription subscription);  
    public void onNext(T item) ;  
    public void onError(Throwable throwable) ;  
    public void onComplete() ;  
}
```

Your observation is correct. It is not a functional interface. It has the methods for demanding, subscribing, and resubscribing the data. It also has the method to handle an error and the completion of items. The following table collects the callback methods along with the description:

Name of callback method	Method description
<code>onSubscribe</code>	This method will be invoked before invoking any other method on the <code>Subscriber</code> for a particular subscription
<code>onNext</code>	This method will be invoked with a subscription's next item.
<code>onError</code>	This is invoked when any unrecoverable error has occurred either in <code>Publisher</code> or <code>Subscriber</code> . Once the method is invoked, further invocation of the methods of the <code>Subscriber</code> will be stopped.
<code>onComplete</code>	This method is invoked when all the messages have been issued to the <code>Subscriber</code> and the <code>Subscriber</code> receives the <code>onComplete</code> event.

You may just want to quickly start with `Subscriber` development. But, we need to discuss one more very important component, `Subscription` as the `Subscriber` subscribes the data through `Subscription`.

The Subscription interface

The `Subscription` interface acts as a message controller between the `Publisher` and the `Subscriber`. We already stated that the system is push-based where the `Subscriber` has to request for the items. In some situations, the `Subscriber` may choose to even cancel the request. This interface provides the methods for these two tasks and these methods can be invoked only by the `Subscriber`. The following snippet shows the methods with their signature:

```
public static interface Flow.Subscription {  
    public void request(long n);  
    public void cancel();  
}
```

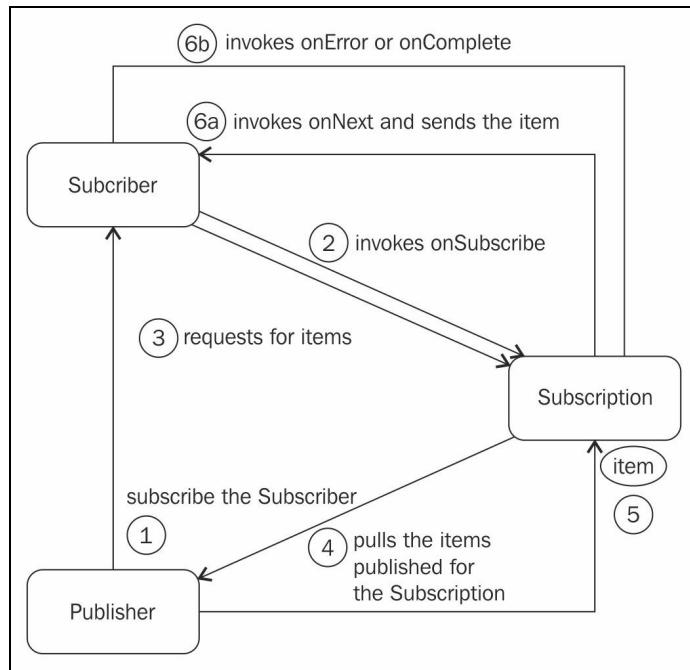
The following table describes the methods:

Name of callback method	Method description
request	This method adds the given number of items to the demand that is not yet fulfilled for the particular subscription. There are n number of requested elements. The value of n has to be a positive integer number that is greater than zero, otherwise, an <code>onError()</code> signal is sent to the <code>Subscriber</code> .
cancel	Receiving the messages will be stopped by the <code>cancel()</code> method. If the subscription is cancelled, <code>onComplete</code> or <code>onError</code> signals will not be received.

Till now, we have discussed `Publisher`, `Subscriber`, and `Subscription` as an individual component. Now is the right time to discuss the overall picture to understand the flow and role of the components in the `Publisher-Subscription` system.

The behavior of the Publisher-Subscriber system

The following figure helps us understand the flow of an item from the **Publisher** till it gets subscribed by the **Subscriber**:



The following discussion will help us understand the publishing-subscribing mechanism:

- The **Publisher** uses the `subscribe()` method denoting that the **Subscriber** can now subscribe the **Publisher**.
- The **Subscriber** subscribes the **Publisher** by invoking `onSubscribe()`. The **Subscriber** now has **Subscription**. Each **Publisher** defines its own **Subscription** and also has the logic to produce the data for the respective **Subscription**.
- The **Subscriber** uses the **Subscription** and requests for the items. The **Subscription** acts as the link between the **Publisher** and the **Subscriber**.

- The `request()` method invokes the `onNext()` method of the **Subscriber** for sending the data.
- The **Subscriber** may also choose to cancel the **Subscription** of items.
- The **Subscription** may choose to invoke `onError()` or `onComplete()` instead of `onNext()` of the **Subscriber** depending upon the scenario.

Now, as we have had lots of theoretical discussion about the **Subscription**., it's the time to do some implementation. Let's start it step by step as discussed in the following section. We will develop the `Subscriber` , which will subscribe the `Long` items published by `SubmissionPublisher`, as we did in the earlier demonstration:

1. In the `ch03_Java_Flow_API` project, add one more class that will be working as our customized `Subscriber`. Name the class `WelcomeSubscriber`.
2. In order to handle `Subscription`, implement the `Subscriber` interface by the `WelcomeSubscriber` as shown in the following code:

```
| public class WelcomeSubscriber implements Subscriber<Long> {  
| }
```

You can observe the `Subscriber` is of type `Long` as its generic type. We are using `Long` as the `Publisher` is going to publish the items of type `Long`. The generic data type is dependent upon which data type the developer wants to handle.

3. Declare the data members in the class shown as follows. We will slowly discover their use in the code:

```
| private Subscription subscription;  
| private final long maxCount;  
| private long counter;
```

4. Write a constructor to initialize the data member `maxCount` shown as follows:

```
| public WelcomeSubscriber(long maxCount) {  
|     this.maxCount =maxCount;  
| }
```

5. We now need to override the abstract methods of the `Subscriber`. Let's override them and develop them one by one starting from `onSubscribe()`.

It performs the initialization of `subscription` and requests the items. We will also display the maximum number of elements asked. The code is shown as follows:

```
public void onSubscribe(Subscription subscription) {  
    this.subscription = subscription;  
    System.out.printf(Thread.currentThread().getName()+"  
        "subscribed with max count %d\\n", maxCount);  
    subscription.request(maxCount);  
}
```

This is one of the most important methods for implementation. This method has a parameter of type `Subscription`. The `Publisher` defines its own `Subscription` and produces the items for that `Subscription`. The object of this `Subscription` is obtained in the `onSubscribe()` method by the parameter. When any `Subscriber` subscribes the `Publisher`, it's the `onSubscribe()` method that will get invoked. Now, the `Subscriber` is ready to link up with the `Publisher`. This linking is possible by invoking either the `request` method or the `cancel` method of the `Subscription`. Here we requested for `maxCount` items. You can request for any number of items as per your requirement.

6. Now, we will override the `onNext()` method for subscribing the published item. The code for it is shown as follows:

```
public void onNext(Long item) {  
    counter++;  
    System.out.println(Thread.currentThread().getName()+"  
        received :" + item);  
    if (counter >= maxCount) {  
        System.out.println(" Cancelling Processed item  
            count :" + counter);  
        // Cancel the subscription  
        subscription.cancel();  
    }  
}
```

The method is implemented for performing the custom action on the item. Most of the time it also uses an incremental counter to decide whether to continue requesting for the items or to cancel. Here we are just printing the item published by the `Publisher` along with the name of the thread.

7. Now override the `onError()` method to handle exceptions occurred while

performing the `Subscription`, shown as follows:

```
public void onError(Throwable t) {  
    t.printStackTrace();  
}
```

8. Now, the final method `onComplete()` which will get invoked once the `Subscription` process gets completed. Let's just print the completion message:

```
public void onComplete() {  
    System.out.printf(Thread.currentThread().getName()+"  
    got completed");  
}
```

9. Now, we will perform the following steps to develop the code for subscribing the published items of type `Long` by `Publisher`:

1. Our `Subscriber` is now ready. We need a `Publisher` that will publish the items for a subscription. We will use `SubmissionPublisher` again as the `Publisher` we did in an earlier demo.
2. Create an instance of the `WelcomeSubscriber` that will take an argument for the number of items to subscribe.
3. Invoke the `subscribe()` method of the `Publisher` by passing an instance of the `Subscriber`.
4. Now, use the `submit()` method of the `Publisher` to push each generated item to the `Subscriber`. We will write this code in the main function.

The complete code is as follows:

```
public static void main(String[] args) throws InterruptedException {  
    SubmissionPublisher<Long>publisher = new SubmissionPublisher();  
    int count = 5;  
    WelcomeSubscriber subscriber = new WelcomeSubscriber(count);  
    publisher.subscribe(subscriber);  
    LongStream.range(10, 20).forEach(publisher::submit);  
    Thread.sleep(1000);  
}
```

We are using the `sleep()` method in order to hold the running thread for a second.

Execute the code and you will get the following output:

```
ForkJoinPool.commonPool-worker-1subscribed with max count 5
ForkJoinPool.commonPool-worker-1 received :10
ForkJoinPool.commonPool-worker-1 received :11
ForkJoinPool.commonPool-worker-1 received :12
ForkJoinPool.commonPool-worker-1 received :13
ForkJoinPool.commonPool-worker-1 received :14
Cancelling Processed item count:5
```

Did you observe the name of the thread? It's `ForkJoinPool.commonPool-worker-1`. We haven't used any thread pool. You must be wondering about this. Have you missed something? We have already discussed that `SubmissionPublisher` uses `ForkJoinPool` as it's a default `Executor`.

We got the data from the `Publisher` and now we will be able to use it as well. However, what if we want to filter the data? Can we process the data before it gets pushed to the `Subscriber`? Let's take an example of publishing the student data. We know very well how to publish this data. We are not only interested in publishing the data, rather we only want to publish students whose name contains the letter `a`. We can consider another scenario as `Publisher` publishes numbers from 1 to 100 and only prime numbers need to be pushed to `Subscriber`. We do have scramble data for publishing. However, we don't want to publish everything we have, rather we want data based upon certain criteria. Now, the question is, how do we push data fulfilling certain conditions? The next discussion will give you the answer.

Processor

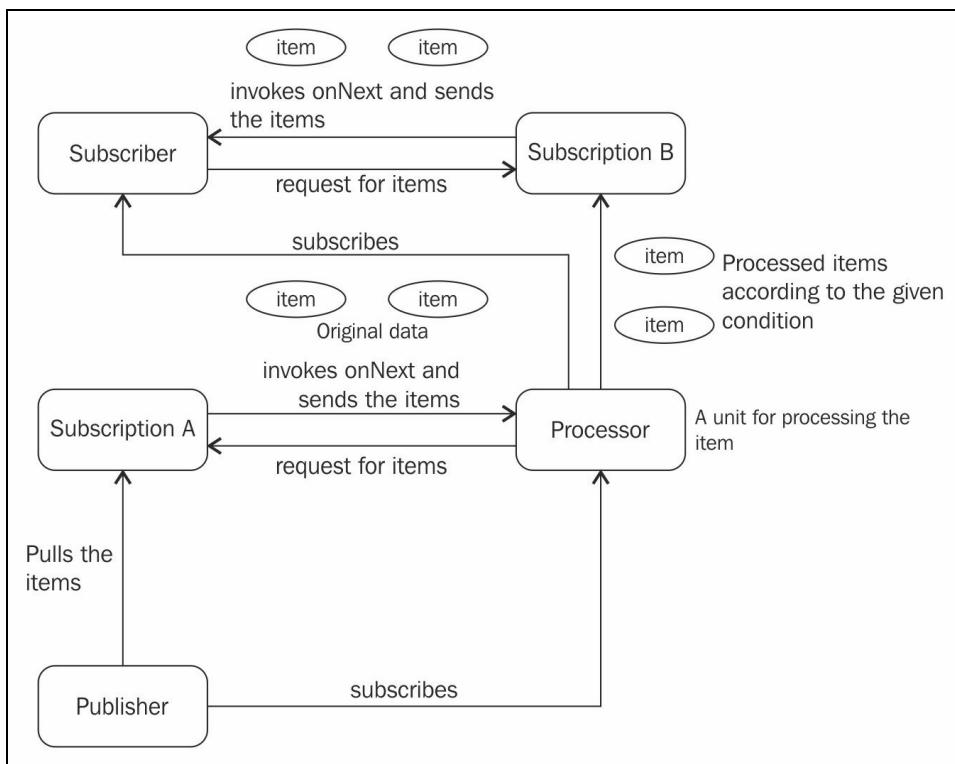
The Processor is a component in between the Subscriber and the Publisher in both the ways of transportation. It acts when the Publisher emits the items as well as when the Subscriber demands an item. In a system, developers can choose to use one or more Publisher, they can even chain them together. The Processor has the following code:

```
public static interface Flow.Processor<T, R> extends
    Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

The Processor has the methods inherited from the Publisher and Subscriber interface. The API doesn't provide any concrete implementation of the Processor. The developers have to write their implementation matching up with the requirements.

The behavior of the Publisher-Processor-Subscriber system

The following figure describes the mechanism of publishing the items by the Publisher to the Processor via Subscription, processing them, and publishing them to the Subscriber:



We already have discussed `SubmissionPublisher`, which publishes the items of type `Long` between the given range. What if we want to filter and get items divisible by 5? Yes, we will develop a processor to understand this better. Follow the steps to develop a `Processor` that we later will use for processing the elements. We will use the same project created in the earlier demos:

1. Create a class and name it `WelcomeProcessor`.
2. Extend the class from `SubmissionPublisher` and implement the `Processor` interface as shown in the following code:

```
    public class WelcomeProcessor<T> extends SubmissionPublisher<T>
        implements Processor<T, T> {
    }
```

3. We want to publish the items that are divisible by 5. We will handle this condition using `Predicate`. So declare a data member of type `Predicate` and initialize it through a parameterized constructor in the class as follows:

```
    private Predicate<? super T> filter;
    public WelcomeProcessor(Predicate<? super T> filter) {
        this.filter = filter;
    }
```

4. Let's now override the `onSubscribe()` method to request for the elements using `Subscription`. The code will be as follows:

```
    @Override
    public void onSubscribe(Subscription subscription) {
        subscription.request(Long.MAX_VALUE);
    }
```

`Long.MAX_VALUE` is considered as an unbounded value.

5. Each published item needs to undergo a conditional check. If the item passes the condition, it will be submitted to the `subscriber` as in the following:

```
    @Override
    public void onNext(T item) {
        System.out.println("Processor received: " + item);
        if (filter.test(item)) {
            this.submit(item);
        }
    }
```

Here we used the `submit()` method to asynchronously publish the item to the current `subscriber` by invoking `onNext()`

6. Now, let's override `onError()` and `onComplete()` shown as follows to handle the exceptions that occurred and for stream completion:

```
    @Override
    public void onError(Throwable t) {
        this.getExecutor().execute(() ->
            this.getSubscribers().forEach(s ->
```

```

        s.onError(t)));
    }
    @Override
    public void onComplete() {
        System.out.println("Processor is complete.");
        this.close();
    }
}

```

7. Now let's write the main function in the `Main_WelcomeProcessor` class, which will perform the following steps:

- Create an object of `SubmissionPublisher`
- Create an object of `WelcomeSubscriber`
- Write a `Predicate` to check if the published item is divisible by 5 or not
- Create an object of `WelcomeProcessor` for the developed `Predicate`
- Link the `processor` between the `publisher` and the `subscriber`

These steps are performed in the following code:

```

public static void main(String[] args) throws
    InterruptedException {
    SubmissionPublisher<Long>publisher = new SubmissionPublisher();
    int count = 5;
    WelcomeSubscriber subscriber = new WelcomeSubscriber(count);
    WelcomeProcessor<Long>processor =
        new WelcomeProcessor<>(n ->n % 5 == 0);
    publisher.subscribe(processor);
    processor.subscribe(subscriber);
    LongStream.range(10, 20).forEach(publisher::submit);
    Thread.sleep(1000);
}

```

8. Execute the code to get the following output on your console:

```

ForkJoinPool.commonPool-worker-2 subscribed with max count 5
Processor received: 10
Processor received: 11
Processor received: 12
ForkJoinPool.commonPool-worker-2 received :10
Processor received: 13
Processor received: 14
Processor received: 15
Processor received: 16
Processor received: 17
Processor received: 18
Processor received: 19
ForkJoinPool.commonPool-worker-2 received :15

```

The underlined output shows only two items that have been published to the `subscriber` and that have passed the condition provided by the `Predicate`.

Designing nonblocking interfaces

Now we've understood many concepts used in reactive applications, it's time to push ourselves a bit harder to understand which mistakes to avoid while writing the application. The `Publisher` created by us has an `executor`, which handles the threads for us. Though here we are not dealing with threads directly we need to be careful so that we will not come across situations of thread blocking. Let's find out how thread blocking can occur in the application.

Let's now develop an application where the `Publisher` will emit a number and the `Subscriber` will consume it. We not only want to concentrate on the `Publisher-Subscription` mechanism but also want to develop the application that will perform the consumption without blocking the application. First, we will develop the application that will block the thread and later on will update it to understand the unblocking process. Follow the steps to create an application:

1. Create a class `NumberPublisher` implementing `Publisher`. The `Publisher` is expected to publish elements where the user will give the start and end point. To facilitate this we need the data members and parameterized constructors shown as follows:

```
public class NumberPublisher implements Publisher<Long> {
    private final ExecutorService executor =
        Executors.newFixedThreadPool(2);
    long start_range, stop_range;

    public NumberPublisher(long start_range, long stop_range)
    {
        this.start_range = start_range;
        this.stop_range = stop_range;
    }
}
```

2. Now override the `subscribe()` method that will invoke the `onSubscribe()` method of `Subscriber`. We will pass the object of `Subscription` to it as an argument:

```

public void subscribe(Subscriber<? super Long> subscriber) {
    // TODO Auto-generated method stub
    subscriber.onSubscribe(new
        NumberSubscription(executor, subscriber,
            start_range, stop_range));
}

```

The `onSubscribe()` method is expecting an object of `Subscription` that is an interface. So now we need to implement it.

- Now create a class `NumberSubscription` as an inner class with `executor`, `subscriber`, the starting range, and the final range as data members. The code is as follows:

```

class NumberSubscription implements Subscription {
    private final ExecutorService executor;
    Subscriber<? super Long> subscriber;
    long start_range, stop_range;
    Future<?> future;
    public NumberSubscription(ExecutorService executor,
        Subscriber<? super Long> subscriber, long start_range,
        long stop_range) {
        // TODO Auto-generated constructor stub
        this.executor = executor;
        this.subscriber = subscriber;
        this.start_range = start_range;
        this.stop_range = stop_range;
    }
}

```

- Override the `cancel()` method to cancel the `Subscription`. The method will shut down the `executor` and cancel future work as shown in the following code:

```

public void cancel() {
    // TODO Auto-generated method stub
    executor.shutdown();
    System.out.println("shutting down");
    future.cancel(true);
}

```

- It's time to develop the `request()` method. The developed method will carry out the task of getting elements of type `Long` using the `LongStream` class. Using `Stream`, we will send each obtained element to the `onNext()` method of the `subscriber` for `Subscription`. We have taken advantage of `Executors.newFixedThreadPool(2)`, which will manage the thread pool. The code will be as shown in the following code snippet:

```

public void request(long item) {
    // TODO Auto-generated method stub
    System.out.println("requesting"+item);
    future= executor.submit(() -> {
        LongStream stream=LongStream.range(start_range,stop_range);
        stream.forEach(subscriber::onNext);
    });
    subscriber.onComplete();
}

```

6. It's now time to create the `Subscriber` as follows:

1. Create a `NumberSubscriber` as a class implementing the `Subscriber`.
2. Add a data member of type `int`.
3. Add a data member of type `Subscription` to it. The code will be as shown in the following code snippet:

```

public class NumberSubscriber implements
    Subscriber<Long> {
    private Subscription subscription;
    int count;

    public NumberSubscriber(int count) {
        // TODO Auto-generated constructor stub
        this.count = count;
    }
}

```

4. And now override the `onSubscribe()` method to pull the subscribed elements from the `Subscription` as follows:

```

public void onSubscribe(Subscription subscription) {
    (this.subscription = subscription).request(1);
}

```

5. The requested number of elements will be published and `onNext()` will be invoked. Let's now override `onNext()` for `subscription`. Here we will display them on the console with the name of the thread as shown in the following snippet:

```

public void onNext(Long item) {
    System.out.println("counter:-" + count);
    synchronized (this) {
        System.out.println(count + Thread.currentThread().getName()
            + " item:-" + item);
    }
}

```

```
|     }
```

6. Simultaneously override the `onError()`, `onComplete()` methods as well.

7. It's time to write the main code and test our application as follows:

- Create a `Main_NumberPublisher` as a class with the `main` method
- Create a `Publisher` with a start and end point to publish the elements
- Create a `Subscriber`
- Invoke the `subscribe()` method on `publisher` to register the `subscriber`

The code is as shown in the following snippet:

```
public class Main_NumberPublisher {  
    public static void main(String[] args) {  
  
        long start_range=10, stop_range=22;  
        Publisher<Long>publisher =  
            new NumberPublisher(start_range,stop_range);  
        // Register Subscriber  
        int count=5;  
        NumberSubscriber subscriber = new NumberSubscriber(count);  
        publisher.subscribe(subscriber);  
    }  
}
```

8. Execute the code to get the items published from `10` on your console. All the items as far as `21` will get displayed. Happy? Obviously! But, the application is still running. Have you noticed that? Do we need to bother about the running threads here? Threads are lightweight, so memory issues will not occur. Wait, we started a process and we need to elegantly complete it. It's our responsibility as a developer. Another thing is, if the threads are waiting means, they are blocked and the application will be stuck forever or we will throw an exception. Neither of these situations is good for our application. The simplest way is to cancel the `subscription` once the `subscriber` subscribes all elements.
9. Let's update the code so that the threads will not be blocked, which is shown as follows:

```
public void onNext(Long item) {  
    System.out.println("counter:-" + count);  
    if (count-- >= 0) {
```

```
        synchronized (this) {
            System.out.println(count + Thread.currentThread().getName()
                + " item:-" + item);
        }
    } else {
        System.out.println(Thread.currentThread().getName()
            + " cancelling");
        subscription.cancel();
    }
}
```

10. Execute the main code once again. And now, we have the output and there are no live threads. Each created thread has completed its life cycle.
11. We even can put a counter and subscribe only the required number of elements instead of getting them all. You can find the complete code at <https://github.com/PacktPublishing/Reactive-Programming-With-Java-9>.

We just discussed how the blocking of threads can happen; however, we also need to understand one more very important terminology, **stream backpressure**.

Understanding backpressure

In system handling for Reactive Programming, we have `Publisher`, which is the source of data, and `Subscriber`, which processes the emitted data. In normal situations, the data emitted by the `Publisher` is handled elegantly. However, the situation is not so simple every time. The rate at which the data is emitted by the `Publisher` is more than the rate at which the `Subscriber` is consuming. It leads to overflowing data, putting pressure on the `Subscriber`. Backpressure is built whenever the `Publisher` is faster than the `Subscriber`. We already have discussed backpressure in detail in [Chapter 1, Introduction to Reactive Programming](#).

Flow API doesn't provide any APIs that facilitate creating a mechanism to handle backpressure. As discussed earlier, sending a signal from the `Subscriber` to the `Publisher` will slow down the emission at the `Publisher` end. Though it allows handling of backpressure, the Flow APIs don't deal with it. It means handling backpressure needs a strategy to be developed at developers end. Backpressure can occur due to the following situations:

- **Slow publisher, fast subscriber:** As a developer, we don't have to worry at all if the `Publisher` is slow in emitting elements. Developers need not slow down the `Publisher`. However, we need to always remember this situation can change at any time. So, it's always good to handle such situations as well.
- **Fast publisher, slow subscriber:** This situation needs to be handled very carefully. It adds pressure on the `Subscriber` as new elements reach it and it is still busy dealing with the previous elements. Such a situation is generally handled by slowing down the `Publisher`.

As the JDK Flow API doesn't provide a mechanism, one of the following strategies can be adopted to control the flow in order to avoid backpressure:

- **Use of buffers:** The developer needs to create buffers. The unprocessed elements will be stored in a buffer in a bounded manner. Once processing is finished a signal can be sent to ask for the next elements.

- **Generation on demand:** In this strategy, the generation of the new elements will be done only on the arrival of the request from the Subscriber.
- **Dropping of elements:** Here, the Publisher will drop all the generated elements unless the signal to send the element is not received from the subscriber.

Let's update `NumbersSubscriber`. We will consider the situation where the Subscriber is slower than the Publisher. If the Publisher publishes items and pushes them it's an unnecessary burden on the Subscriber. Now, we will pull the elements from the Publisher once the Subscriber finishes processing its heavy workload. Here we are considering file reading as a time-consuming operation. Once we have finished reading a file, we will request for the next item. Observe the following code:

```

public void onNext(Long item) {
    System.out.println("counter:-" + count);
    if (count-- >= 0) {
        synchronized (this) {
            System.out.println(count + Thread.currentThread().getName()
                + " item:-" + item);
            try {
                final BufferedReader reader =
                    new BufferedReader(new FileReader("XXX"));
                //replace XXX with name of the file along with a
                //location. You can choose any lengthier file.
                reader.lines().forEach(System.out::print);
            }
            catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            subscription.request(1);
        }
    } else {
        System.out.println(Thread.currentThread().getName()
            + " cancelling");
        subscription.cancel();
    }
}

```

Here, we are requesting the `subscription` to send the next element using `subscription.request(1)`. It means it's now within the purview of the subscriber to handle the situation without getting a pressure of completing the work just because someone is waiting for their turn.

Summary

We started the chapter with a jigsaw puzzle with lots of information scattered here and there, some collected from earlier chapters. We were able to understand the various concepts. However, we were wondering how to put them together. How should we start writing the code?

In this chapter, we discovered the Flow API equipped with interfaces and classes to empower the user to develop reactive applications based upon the `Publisher-Subscription` model. We discussed in detail all the major components of the system along with the role each of them plays. We discussed them along with sample code to gain the implementation confidence. We developed each component step by step and discovered the beauty in the program. Our motive before starting the chapter was to understand how to develop an application that will give us better performance without blocking execution and also capable of handling the backpressure. Though the Flow API doesn't explicitly provide a mechanism to deal with backpressure, we discussed ways to tackle it and also implemented it in our application.

In the next chapter, we will discuss RxJava, a third-party API provided to create reactive applications and make the developer's life easy.

Reactive Types in RxJava

In the previous chapter, we discussed achieving Reactive Programming using Java 9 based APIs. We smoothly dived into Reactive Programming with the help of Flow APIs to understand developing applications with designing non-blocking interfaces. We discussed the term **backpressure** in depth with demonstrations; however, Flow APIs don't provide a mechanism as well as interfaces to handle backpressure. Use of the Flow API is not the only way to achieve Reactive Programming; we can achieve it using third-party APIs. RxJava is one of the famous libraries available on the market. In this chapter, we will discuss RxJava with the help of the following points:

- Reactive extensions
- `Observable`, `Single`, `Maybe`, `Flowable`, and `Completable`
- `Observer` and `Subscriber`
- `Disposable`
- `Subject`

ReactiveX

ReactiveX is a library provided as an extension for creating asynchronous event-based applications for JVM. ReactiveX doesn't use any typical design pattern; however, it's a combination of the `observer` design pattern, the Iterator design pattern, and functional programming. ReactiveX is for frontend managing the UI, and the backend, as well as for business logic components developed through Java, .Net, Scala, C Sharp, C++, Python, and many more. ReactiveX extensively works with `observable` Streams.

We have already discussed functional programming in the earlier chapter, so before moving ahead let's discuss the `observer` and Iterator design pattern.

Observer design pattern

It is one of the known behavioral design patterns in Java where the relation between the objects has been defined. It's a relationship between one to many objects, whenever one object changes its state, the other objects will be notified and updated automatically. Let's take an example of a loan application to know more about it. Whenever a person puts an application for a loan to the bank, what does the bank do? The respective person from the bank notifies the other departments. He informs the eligibility department to check the eligibility. In case he is eligible, the verification department does the verification of the documents and other relative things. If everything is well, ultimately the loan will be sanctioned and a kit will be sent. In each of the state changes, the other departments will be notified accordingly.

The Iterator design pattern

The Iterator design pattern is also one of the behavioral design patterns that facilitate walking through a group of objects. We, as Java developers, can relate to the design pattern easily. We all are well aware of the Collection framework and it has an `Iterator` interface that provides methods using which traversing through the elements of a collection becomes easy.

Advantages of ReactiveX

ReactiveX offers the following advantages, making it useful to be used in reactive applications:

- ReactiveX supports functional programming, facilitating writing clean code without getting involved in managing the state.
- It offers a complete set of operators offering an abstract mechanism to the developers to achieve handling, manipulation, and transformation without getting involved in how it has to be done.
- It offers to handle the data asynchronously, as well as provides a powerful mechanism for handling errors.
- The developers now don't need to get involved in low-level threading, synchronization, and managing concurrency. ReactiveX offers an `Observable` as well as a `Scheduler`, which takes away the issues occurring while handling threads for concurrency and provides an abstract solution.

RxJava

RxJava is an open source Java implementation of ReactiveX. It has been published under the license of Apache 2.0, originated at Netflix. The light weight library provides powerful APIs for creating asynchronous event-based programs using the sequence of observables. RxJava supports Java 6 or higher versions along with Groovy, Scala, Clojure, and JRuby.

RxJava also supports third-party libraries for asynchronous event-based programming. Some of them are mentioned as follows:

- **Camel Rx:** Camel Rx facilitates the support for Camel for reactive extensions. It allows the reuse of any component from Apache Camel components, protocols, data formats, and transports. The developers of Camel can use the RxJava API for processing the messages on endpoints with the help of typesafe composable APIs.
- **Hystric:** It is the fault tolerance and latency library that has been designed for isolating the points of access to the services, remote systems, as well as third-party libraries. It allows stopping the navigation of the failures still enabling to achieve resiliency in the distributed environment.
- **rxjava-jdbc:** The library enables the execution of the database calls using the JDBC APIs and Observables provided by RxJava. It composes the queries to run sequentially or in parallel. It also automatically facilitates the mapping of the obtained result from the query to the typed tuples or the classes automatically.

The following are the reasons to choose RxJava for reactive applications:

- The APIs provided by RxJava are easy to use
- Along with providing the interfaces to create the components of the reactive system, it also provides the set of operators allowing the developers to easily manipulate the data in a declarative manner
- It provides a mechanism to handle the errors that have occurred in the application

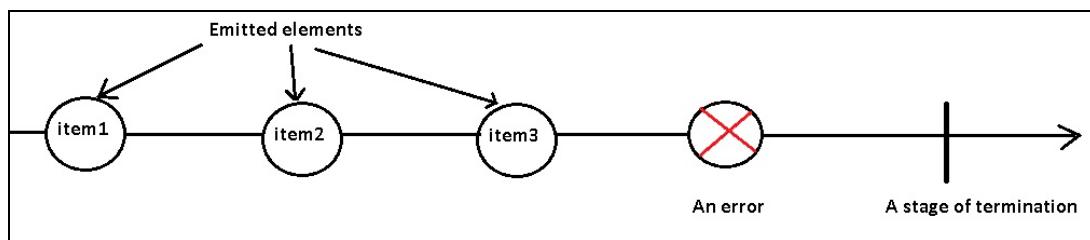
- The need for the variables that can manage the state of an object is reduced

Observable Streams

We all know Reactive Programming is all about reacting to the data emitted. In the discussion, we have seen that ReactiveX works with Observable Streams. What are Observable Streams? Some of us may be aware of it but many of us not. As it is the heart of ReactiveX and RxJava, it will not be justified to just move on without any discussion. When we say streams, the first thing that comes to mind is a sequence of data or events. The Observable Stream has a special power to emit any one of an element, an error, or a state of termination as discussed in the following scenarios:

- An element is actually a data or sequence of data that can be as simple as a number with a value of type `integer` to as complicated as a value of type `object`.
- In an application, the situation can change anytime due to some problem that might have occurred at the time of execution. An error will be emitted if something goes wrong with the emission of an element.
- The emission of all the elements have been finished and no more elements are left for emission, at this time the stage of termination will be emitted.

Here onwards as the Observable Streams are the center of discussion and every now and then we will have a discussion about emission, handling, or transformation of the Stream. So, you must be aware of the marble diagram that makes the discussion quite easier. Observe the following diagram:



The preceding diagram shows a timeline from left to right where the elements have been emitted. The Observable Stream has emitted three elements, an error, and a stage of completion. As each of these needs to be fetched one by

one we need to walk through it. Doesn't it remind us of the Iterator design pattern?

Also, some component needs to be sitting at the other end to consume them. It will first capture the events and then consume the data as discussed in the observer design pattern. Hold on, before we move into details, let's, first of all, find out the generation of RxJava.

In market, we have RxJava1 and RxJava2 as known versions. Before discussing them, one should be aware that both these versions are designed according to JDK 6, and higher versions of JDK, specifications. It won't support JDK 8 Streams. In case you need JDK 8 support, you need to consider Reactor-Core provided by Pivotal. The RxJava 2.x is directly targeting Reactive Streams API. The RxJava 2.x has been written from scratch and provides improved performance and a better model for memory consumption.

Components of RxJava

The following are the backbones of RxJava:

- `Observable` who is the source of data
- `Subscriber` who reacts to the changes in the `Observable`
- The **operator** who supplies a set of useful methods for utilizing, composing, and transforming the emitted data

Observable

In [Chapter 3](#), *Reactive Streams*, we discussed `Publisher`, which being the source of the data emits the data. RxJava is about the Observable Streams, hence it uses the name `observable` instead of `Publisher`. The `observable` is subscribed by the observer, which does consumption. The observer is usually called `subscriber`. The observer or the subscriber reacts to the items or notification emitted by the `observable` facilitating the operations without blocking. Keep in mind its Observable Streams, which means items can be data, an error, or stage of completion. `Observable` allows the execution of many instructions in parallel. Once all the instructions get processed their result is captured. You defined a mechanism for the transmission and retrieving the data rather than just invoking the method. The observer responds to the emission of the items done at the `observable` end. This mechanism facilitates the starting of many tasks at the same time, rather than waiting for one task to complete increasing the performance. How? Let's elaborate with an example. Consider we do have n tasks to perform. When one task runs after another, what will be the total time taken to complete the process? Yes, the total time required is the sum of the time taken by each of the subtasks. On the other hand, what if these tasks are running in parallel? Correct, the total time required for completion of the execution will be the time taken by the longest task for its completion and it will not be some time taken by each task. The faster the output, the quicker the application. The user is happy, and so are we!

The `observable` interface has the methods that will provide the facility to create `observable`, transforming `observable`, filtering `observable`, combining `observable`, error handling, and provide utility along with different flavors of the method `subscribe()`, to subscribe the source of the data and provide a mechanism to handle the callback for handling data, error, or the completion signal.

Let's find out the creation of an `observable` in the upcoming demo using the `observable.just()` method, which will emit a welcome message. You can follow these steps to create the demo:

1. First of all download `reactive-streams.1.0.0.jar` and `rxjava-2.0.8.jar`.
2. Create `RxJava_Demos` as a Java project.
3. Now, add `Demo_Create_Observable` as a class that contains a main function as well.
4. Every time we need to follow three main steps:
 1. Create a source of data.
 2. Create consumer of the data.
 3. Connect consumer to the source.
5. We will create `observable` using the `just()` operator, which is our source of data.
6. How do you crosscheck that the data has been emitted or not? We need an observer meaning the consumer for the data. The observer is a whole new story, here we will use printing operation as a `Subscriber` OR `Observer` instead of creating a custom one.
7. `observable` has a `subscribe()` method, which provides a facility for subscription or use of the data emitted. The `subscribe()` method is used to set up the connection between the consumer and the subscriber.

The steps can be written as follows:

```
public class Demo_Create_Observable {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Observable observable=Observable.just("Welcome to RxJava");
        observable.subscribe(s->System.out.println(s));
    }
}
```

Here we used the `just()` operator without any discussion. We will discuss in a few pages the `just()` operator as well as a few more operators to create `observable`.

The `Observer` has been attached to `Observable` by subscribing it using the `subscribe()` method. Whenever the call returns, the method of `Observer` will start operating on the values returned by the method, which actually is the data or sequence of the data emitted by the `observable`.

8. Execute the code to get `welcome to RxJava` displayed on the console.

This simple demo gives us an idea about how to create a simple observable using the `just` operator. The method `subscribe()` provides a connection between the observable and the observer. The `observer` reacts to the data, error, or the completion stage. In this case, our observer is simply dealing with the data.

Let's write one more demo where more than one item will be emitted. Here we will consider a list of numbers from where an observable will emit an element one by one and the observer will use the data as shown in the following code:

```
public class DemoObservable_numbers {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Integer[] nums = { 1, 23, 34, 21, 33, 4, 54 };
        List<Integer> numList = Arrays.asList(nums);

        Observable.just(numList).subscribe(item ->
            System.out.println(item));
    }
}
```

On execution, we will get the list printed on our console. We got the elements, however, in the application development, we need to provide the custom implementation on how to deal with data, an error, as well as the completion stage. In the upcoming pages, we will discuss more about it.

What is `subscriber` and how can we create our own `Subscriber`? OK, let's discuss `subscriber` before moving ahead with `observable`.

Observer

As we know, the `subscribe()` method provides a bridge that connects the observable to the observer, `observer` consumes the data. But guys, we are dealing with observable streams. It means along with data, we need to deal with an error as well as the completion stage. The code for the `observer` interface is as follows:

```
interface Observer<T> {  
    void onNext(T t)  
    void onError(Throwable t)  
    void onComplete()  
}
```

The observer implements the `observer` interface and so its methods need to be overridden. On the implementation of the `observer` interface, the observer has three types of events pushed to it, as discussed by the following methods for customizing the process of consumption:

- `onNext()`: Whenever an observable emits an item, it will call the `onNext()` method. The method accepts the data emitted by it as an argument, allowing the developers to utilize the data.
- `onError()`: The prime motive of an observable is to emit the data, however, sometimes while emitting the data something may go wrong. Due to the problem, the observable is unable to emit the data. `observer` is waiting for the data, and it's very much necessary to communicate the situation has gone out of control. Under such circumstances, the observable calls the `onError()` method and communicates the error. Once an error occurs the observable will not make further calls either to the `onNext()` method or `onComplete()` method.
- `onComplete()`: Observable at one stage will complete the emission of the items. Once the last item got emitted by the observable it calls the `onComplete()` method denoting the completion of the emission without any error.
- `onSubscribe(Disposable)`: The `onSubscribe()` method accepts `Disposable` as a parameter. This parameter can be used for disposing the connection between `observable` and `observer`. It is also used to find out whether we

are already disposed or not.



An observable can make 0 to n calls to the `onNext()` method for emission of the items. An observable can make a subsequent call for notification to either `onError()` method or `onComplete()` method, but not to both.

Let's use `observer` to demonstrate the discussed methods. We will write the following code in the main method. We will use the `rangeLong(long start, long count)` operator to create an `observable` that will emit a specified number from the starting point as shown by the following code:

```
|     Observable<Long>observable= Observable.rangeLong(11,31);
```

To start using the elements, `observable` needs to invoke the `subscribe()` method. The method accepts an instance of `observer`. So first off create an `observer` using an anonymous inner class as follows:

```
Observer< Long> observer=new Observer<Long>() {  
    @Override  
    public void onComplete() {  
        // TODO Auto-generated method stub  
        System.out.println("on complete");  
    }  
    @Override  
    public void onError(Throwable throwable) {  
        // TODO Auto-generated method stub  
        throwable.printStackTrace();  
    }  
  
    @Override  
    public void onNext(Long value) {  
        // TODO Auto-generated method stub  
        System.out.println(""+value);  
    }  
  
    @Override  
    public void onSubscribe(Disposable disposable) {  
        // TODO Auto-generated method stub  
        System.out.println(disposable.isDisposed());  
    }  
};
```

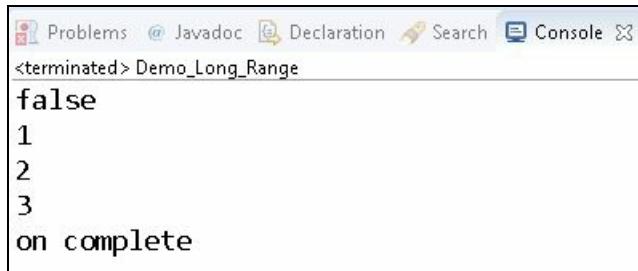
Here, we have created the `observer` of type `Long` as our `observable` is going to emit items of type `Long`.

Now, the last step remaining is to trigger the subscription mechanism using

the `subscribe()` method. When the subscription starts, methods from `onSubscribe()`, `onNext()`, `onComplete()`, or `onError()` will be invoked implicitly. The code for subscription is:

```
|     observable.subscribe(observer);
```

Let's execute the code to get the output as follows:



```
<terminated> Demo_Long_Range
false
1
2
3
on complete
```

The `isDisposed()` method is returning `false` and then the first method onward, the subscription of emitted items will be done. As there is no exception the subscription will be terminated normally. In order to dispose the resource, we need to invoke the `dispose()` method. Update the code for the `onSubscribe()` method by adding the following code:

```
|     disposable.dispose();
```

After updating, if we execute the code, code from the `onNext()` method will not be executed as the resource is disposed and a signal for `onNext()` will not be sent.

Subscriber

`Subscriber` is an advanced and lightweight `observer` as now it doesn't have to deal with lots of internal state handling. It combines the request management along with the cancellation process into the `Subscription` interface rather than creating `Producer` and `Subscription` separately. `Subscriber` provides the support for backpressure. The `Subscriber` interface along with its methods can be summarized as follows:

```
interface Subscriber<T> implements Observer<T>, Subscription {  
    void onNext(T t)  
    void onError(Throwable t)  
    void onComplete()  
    void onSubscribe()  
}
```

In RxJava 2.0, the package names have been changed to `rx` from `org.reactivestreams`. Also, `Subscriber` was not adding, cancelling, or requesting the resources from outside. In 2.0 to overcome this, the API provided abstract classes such as `DefaultSubscriber`, `DisposableSubscriber`, and `ResourceSubscriber` for `Observable` such as `Flowable`. These classes facilitate resource tracking and cancellation using the `dispose()` method.

We will use `Subscriber` in the upcoming demos in various situations. Let's now discuss these components in depth one by one. We will start with `Observable`.

Are the `observables` that we created, hot or cold? Hey, what's a hot and cold `Observable`? Don't worry, before moving ahead we will first discuss the hot `Observable` or cold `Observable` in depth.

Hot or cold Observables

An observable can be hot or cold depending upon when it starts emitting the items. If an observable starts emitting elements immediately after its creation, it becomes a hot observable. In such situations, the observer can start using the items anytime at the middle of the emission process, leading to a problem of knowing only a part of the emitted sequence.

To make it simple, let's take an example where we will generate a random number and will push it to the observer. Have a look at the following code:

```
public class Demo_Hot_Observable {  
    public static void main(String[] args) {  
        ConnectableObservable observable =  
            Observable.create(observer -> {  
                observer.onNext("I am Hot Observable "+Math.random()*100);  
                observer.onComplete();  
            }).publish();  
        observable.subscribe(consumer ->  
            System.out.println("message:-" + consumer));  
        observable.subscribe(consumer ->  
            System.out.println("message:-" + consumer));  
        observable.connect();  
    }  
}
```

On execution of the code, we will get the following output:

```
message:-I am Hot Observable 70.66375775619045  
message:-I am Hot Observable 70.66375775619045
```

We actually had two observers; instead of getting two random numbers we got a single. It did not emit data twice. Actually, emitted data has been observed twice.

Opposite to hot observable, a cold observable waits for the observer to subscribe to it only after which it will start emitting the items. A good thing about the cold observable is the emission starts only after it subscribes and so it can observe the sequence of items from the beginning.

Let's rewrite the same sample code to convert it to a cold observable:

```
public class Demo_Cold_Observable {  
    public static void main(String[] args) {  
        Observable observable = Observable.create(observer -> {  
            observer.onNext("I am Hot Observable " + Math.random()*100);  
            observer.onComplete();  
        });  
        observable.subscribe(consumer ->  
            System.out.println("message:-" + consumer));  
        observable.subscribe(consumer ->  
            System.out.println("message:-" + consumer));  
    }  
}
```

Execute the code and observe the numbers generated. Have you got a single value? No! We have two distinct values. Why? The cold observable will start emitting the data only after it gets the subscriber. We have two subscribers in action; the observable will emit data twice. I hope we made the point pretty clear.

We have used the `just()` operator to create an observable, without knowing about it. Here, we will discuss `just()` as well as many more operators that facilitate creation of observable under various situations.

Creating Observable emitting sequential data

It's time to dig into the creation of an observable using the operators provided by the RxJava library. We will start with creating an observable from a sequence using various operators as discussed in the following sections.

Using the just() operator

The `just()` operator allows the developers to create an `observable` emitting an item. The major difference between `just()` and `from()` is, `from()` pulls the data from an array, an iterable or a `Future` for emission, and `just()` emits the item and can be an array or an iterator as a single item.

We already have seen a couple of demos to use `just()`. The code also has one more sample code for your reference.

Using the defer() operator

The `defer()` operator works on a lazy basis to generate an observable. It only creates `observable` when the observer subscribes to it. At a time, an observable can have multiple observers. The `defer()` operator takes a factory function that is capable of returning an observable. It does the wrapping of the obtained observable and creates a new `observable` only when the subscriber subscribes to it. The thing to remember about `defer` is each observer gets its own sequence. It's useful when the requirement is to get an observable and not a value.

We had to create an observable using `just()` and understand how it works. Now, we will use `defer()` to create a new `observable` as we did earlier using `just()` operator `just()`. The code will be as follows:

```
public static void demoDefer() {
    String[] monthArray = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                           "July", "Aug", "Sept", "Oct", "Nov", "Dec" };
    List<String> months = Arrays.asList(monthArray);
    Observable.defer(() ->
        Observable.just(monthArray).create(subscriber -> {
            try {
                for (String data : months) {
                    subscriber.onNext(data);
                }
                subscriber.onComplete();
            }
            catch (Exception e) {
                // TODO: handle exception
                subscriber.onError(e);
            }
        }).subscribe(item -> System.out.println(item), error ->
            System.out.println(error),
            () -> System.out.println("Emission completed"));
    }
}
```

We wrap the originally created `observable` using `just()` by `defer()`. However, the emission of the data from the `observable` created using `just()` will not start unless we subscribe the `observable` that has been created using `defer()`.

Using the empty() operator

Under certain situations, we need an observable that doesn't emit any item but should be terminated normally. All we need is operator `empty()` to fulfill our requirement. This observable will be terminated immediately after the subscription.

The following code demonstrates the use of the `empty()` operator:

```
public class DemoObservable_empty {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Observable<String> observable = Observable.empty();  
        observable.subscribe(item -> System.out.println("we got" + item),  
                            error -> System.out.print(error),  
                            ()->System.out.print("I am Done!! Completed normally"));  
    }  
}
```

Usually, we get an `onNext()` method called after that, if an error occurs, `onError()` will be invoked. Under normal conditions, the `onComplete()` method will be invoked. However, here we are dealing with an empty observable, so no item will be emitted and hence we will only get the output from the code written as a body of the `onComplete()` method.

Using the never() operator

This is one of the special `observables` that we may require to create; under certain scenarios, the developers need `observable` with a condition that it will never emit any item. It never emits items or errors. Even such an `Observable` never gets completed. As a developer, we may need such an `observable` for testing purposes. Let's use the operator to create our `observable` as done in the following code:

```
public class DemoObservable_never {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Observable<String> observable=Observable.never();  
        observable.subscribe(item-> System.out.println("we got"+item));  
    }  
}
```

Execute the code and observe the output on the console. Oops! To our surprise, we haven't got any output. Why are you so surprised? We already discussed in the beginning of the operator, `never()` never ever emits an element. So it's obvious, the `onNext()` method will not be invoked and thus no line on our console.

Using the error() operator

As a `never()` operator is used when we need an `observable` with no items emission, an `error()` operator also doesn't emit any item, however, it terminates with an error. As there is no emission of items, such an `observable` never gets completed. Let's find out how to use it:

```
public class DemoObservable_error {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Observable<String> observable = Observable.error(new
            Exception("We got an Exception"));
        observable.subscribe(item -> System.out.println("we got"
            + item), error -> System.out.print(error));
    }
}
```

Execute the code and you will get `we got an exception` printed on the console, but `we got` is misspelling. And it's very obvious as this `observable` is not for the emission of items.

Using the `create()` operator

The `create()` operator is a factory method that creates an `observable` by calling the method on the observer programmatically. It is preferred whenever the developers want to implement the custom observable. The method facilitates the developer to accept a function that accepts the `observer` as its parameter. The function that specifies an observer is a kind of delegate that will be executed whenever a subscription is done. The subscription can be passed to the delegate in such a way that `onNext()`, `onError()`, and `onComplete()` methods can be invoked on it. One of the advantages of the `create` method is it supports lazy evaluation, which is an important part of the Rx implementation.

Let's add a method `demoCreate()` in a `DemoCreate_Observable` class to understand creating an `observable` using `create()` operator as follows:

```
public static void demoCreate() {
    String[] monthArray = { "Jan", "Feb", "Mar", "Apl", "May", "Jun",
                           "July", "Aug", "Sept", "Oct", "Nov", "Dec" };
    List<String> months = Arrays.asList(monthArray);
    Observable.create(subscriber -> {
        try {
            for (String data : months) {
                subscriber.onNext(data);
            }
            subscriber.onComplete();
        } catch (Exception e) {
            // TODO: handle exception
            subscriber.onError(e);
        }
    }).subscribe(item -> System.out.println(item), error ->
    System.out.println(error),
    () -> System.out.println("Emission completed"));
}
```

As you can see, the `create()` method accepts an object of type `Subscriber` and facilitates the operations to be performed as per the scenarios.

When the `subscribe()` method gets called the emission of the elements will be started. Each time the `onNext()` method will request a new element from the source. The method `subscribe()` is called the `Subscription` inside the `create()`

method and will be invoked. This may be a problem in case where the developers are dealing with multiple subscribers for the same `observable`. The `create()` method will be invoked separately for every created `subscriber` in the application. If the developers want to avoid calls to the `create()` operator, the developers need to use the `cache()` operator. We now will go into the details of `cache()`; however, in the next chapter, we will certainly discuss this in depth.

The `create()` operator is the most useful operator to create `observable` as it can be used for various situations, such as creating `s` for single item emission, completion of subscription without emitting any item, and `observable` emitting only errors.

Transforming Rx non-compatible sequences into Observables

We may have a sequence of data as a collection or some asynchronous events that are not compatible Rx types. The following discussed operators facilitates the developers to transform the data or events from non compatible to compatible types:

Conversion using the from() operator

The `from()` operator allows conversion of objects into an observable. When we work with `observable` it is always more convenient to work with only `observable` rather than dealing with heterogeneous data of `observable` and other data types.

The `from()` operator has the following flavors as follows facilitating developers to create `observable` from a specific type:

- `fromIterable()`: This allows the developers to take an `Iterable` and emit its values in their order
- `fromArray()`: Similar to `fromIterable`, the operator `fromArray` takes an argument of an array and emits its values
- `fromCallable()`: This operator facilitates and creates an `observable` for `Callable`
- `fromFuture()`: This operator facilitates the developers to create an `Observable` for a `Future`

Let's use the `from()` operator in the sample code to explore it more. We will use a collection of months to create `observable`:

```
public static void demoFromArray()
{
    String[] monthArray = { "Jan", "Feb", "Mar", "Apr", "May",
                           "Jun", "July", "Aug", "Sept", "Oct",
                           "Nov", "Dec" };
    Observable.fromArray(monthArray).subscribe(item ->
        System.out.println(item), error -> System.out.println(error),
        () -> System.out.println("Emission completed"));
}
```

The `subscribe()` method takes three parameters:

- Consumer of data
- Consumer of an error

- Action to take on completion

`Action` is an interface that allows throwing the checked exception. Its working is very much similar to the `Runnable` interface. Here we are writing a lambda expression to print the message.

You can now execute the code to get the names of months displayed on the console.

Now, it's time to discuss how to create an array from `Iterable`. We will use `java.util.List`, which can be iterated over to get the items. The code will be as follows:

```
public static void demoFromList()
{
    String[] monthArray = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                           "July", "Aug", "Sept", "Oct", "Nov", "Dec" };
    List<String> months = Arrays.asList(monthArray);
    Observable.fromIterable(months).subscribe(item ->
        System.out.println(item), error -> System.out.println(error),
        () -> System.out.println("Emission completed"));
}
```

Execute the code and get the names of the months displayed on the console.

Creating Observables for unrestricted infinite length

As of now we have created the `observable` having some sequence to follow or with a finite number of items using various operators. However, we can also create sequences of the unrestricted or infinite length of data using the factory methods as discussed in the following sections.

The interval() operator

The `interval()` operator facilitates the emission of items of type `integer` after a given interval. `observable` emits the items after an equal interval of an infinite sequence of ascending integers. The `interval()` operator can be useful in the applications that are long running.

Sounds interesting! Let's write code that will print the number starting from 0 with an interval of two seconds. We first off need to create `observable` using `interval()`, which can be done as follows:

```
|     Observable<Long> myObservable = Observable.interval(2, TimeUnit.SECONDS
```

The emitted items need to be observed by the observer. Let's create our `observer` and implement `onNext()`, `onError()`, and `onComplete()` methods as follows:

```
Observer< Long> myObserver=new Observer<Long>() {  
    @Override  
    public void onComplete() {  
        // TODO Auto-generated method stub  
        System.out.println("emission completed");  
    }  
  
    @Override  
    public void onError(Throwable throwable) {  
        // TODO Auto-generated method stub  
        System.out.println("sorry you got an error");  
    }  
  
    @Override  
    public void onNext(Long item) {  
        // TODO Auto-generated method stub  
        System.out.println("item"+item);  
    }  
  
    @Override  
    public void onSubscribe(Disposable arg0) {  
    }  
};
```

To set up the connection between the `observable` and the `observer` we will use the `subscribe()` method as follows:

```
|     myObservable.subscribe(myObserver);
```

Now, execute the method and observe the output. Oh! Nothing happened. Our application just terminated without taking any action. Why? What's wrong? Wait! Don't panic.

It happens as the `interval()` operates on `Scheduler.computation()`, which allocates a new thread other than main. The JVM exits immediately and the new thread will be stopped. In our case the main thread gets terminated without giving a chance to create a new `Observable`. To solve this, we just need to suspend our main thread by adding the following code:

```
try {
    Thread.sleep(10000);
}
catch (InterruptedException e) {
    e.printStackTrace();
}
```

Now, execute the code and get items displayed one by one with an interval of two seconds.

The range() operator

The `range()` operator is used to create an `observable` that will emit sequential `integer` elements within a given range. Let's create an observable that will emit the numbers from `10` to `15`, as shown in the following code:

```
public class DemoRange {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Observable<Integer> observable=Observable.range(10, 5);  
        observable.subscribe(number->System.out.println(number));  
    }  
}
```

The `range()` method accepts two arguments. The value of the first argument denotes the starting of the range and the second arguments is to denote how many items to emit. Here we have an observable that is emitting five numbers starting from `10`. On execution, these five numbers will get printed on the console.

The `timer()` operator

The `timer()` operator creates an observable that produces items that are in a time-based sequence. After a particular time delay, the next item will be produced.

Types of observables

The observables being a source emits the data. The emitted data can be a single item, an empty item, or multiple items that can be a sequence of data. Depending upon these emissions, the `Observable` data types are explained in the following sections.

The Observable<T> operator

`observable` emits either *0* or *n* items and will be terminated normally invoking the `onComplete()` method or the `onError()` method will be invoked if something goes wrong. `observable` is not able to handle backpressure.

Scenarios for using an Observable

The following are the scenarios that we can consider to use an observable:

- It's preferred to use an observable when we are dealing with about 1,000 elements, considering there will be very less chance to get an `outOfMemoryError` error
- When the developers are dealing with a GUI event having a very rare chance of backpressure
- We are handling synchronous flow and we don't have support for Java Streams

Let's use an observable to emit the name of month one by one and subscribe it by a `Subscriber` to display on the console:

```
public class DemoObservable {

    public static void main(String[] args) {
        Observable<String> month_observable = Observable.create(new
        ObservableOnSubscribe<String>() {
            @Override
            public void subscribe(ObservableEmitter<String> emitter)
                throws Exception {
                // TODO Auto-generated method stub
                try {
                    String[] monthArray = { "Jan", "Feb", "Mar",
                        "Apr", "May", "Jun", "July", "Aug",
                        "Sept", "Oct", "Nov", "Dec" };

                    List<String> months = Arrays.asList(monthArray);

                    for (String month : months) {
                        emitter.onNext(month);
                    }
                    emitter.onComplete();
                }
                catch (Exception e) {
                    // TODO: handle exception
                    emitter.onError(e);
                }
            }
        });
        month_observable.subscribe(s -> System.out.println(s));
    }
}
```

On execution of the code, you will observe that `observable` have emitted months one by one which has been consumed by the subscriber. We already have seen various operators get an `observable`.

The Single<T> operator

`single` emits a single item or an error, so when the subscriber subscribes to `single` it can get either a value returned or an error.

Here, instead of using three methods for subscribing, we will only use the following two methods:

- `onSuccess()`: A single item will be emitted
- `onError()`: If a problem occurs in the emission of the items, an error will be emitted and the method will be invoked

The following code snippet will clarify the use of `single`:

```
public class DemoSingle {  
  
    public static void main(String[] args) {  
  
        Single<List<String>> month_maybe = Single.create(emitter ->  
        {  
            try {  
                String[] monthArray = { "Jan", "Feb", "Mar", "Apr",  
                    "May", "Jun", "July", "Aug", "Sept", "Oct", "Nov", "Dec" };  
  
                List<String> months = Arrays.asList(monthArray);  
                if (months != null && !months.isEmpty()) {  
                    emitter.onSuccess(months);  
                }  
            }  
            catch (Exception e) {  
                emitter.onError(e);  
            }  
        });  
        month_maybe.subscribe(s->System.out.println(s));  
    }  
}
```

Execute the code and you will get the months displayed on the console.

The Flowable<T> operator

`Flowable` emits either 0 or n items and completes successfully. It may also happen that while emission is going on, an error occurs then `Flowable` completes with an error. `Flowable` supports handling of backpressure, which facilitates the control of the speed at which the items are emitted by `Observable`.

Scenarios to use Flowable

The following are the scenarios that we can consider to use `Flowable`:

- When we are dealing with more than 10,000 elements where the chain of elements can limit the amount of generated elements
- Dealing with a situation that is pull-based and blocking the operation, however, it must work well with backpressure
- Under Stream networking I/Os where the network or the protocol support requesting the elements from the source
- In the situation where we do have multiple blocking, pull-based data sources that can get non-blocking reactive in the future

Let's use `Flowable` to emit the names of the months and consume them by the subscriber as follows:

```
public class DemoFlowable {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Flowable<String> month_maybe = Flowable.create(emitter -> {
            try {
                String[] monthArray = { "Jan", "Feb", "Mar", "Apr",
                    "May", "Jun", "July", "Aug", "Sept", "Oct", "Nov", "Dec" };
                List<String> months = Arrays.asList(monthArray);
                for (String month : months) {
                    emitter.onNext(month);
                }
                emitter.onComplete();
            }
            catch (Exception e) {
                emitter.onError(e);
            }
        }, BackpressureStrategy.MISSING);
        month_maybe.subscribe(s -> System.out.println(s));
    }
}
```

The code is very much similar to the earlier observables, however, we can observe a very significant argument accepted by the `create()` method. It is a backpressure strategy. We have discussed backpressure earlier; `BackpressureStrategy` is an enum that provides different options for applying the backpressure to a source of a sequence of items. Let's discuss strategies

provided to handle backpressure:

Backpressure strategy	Description
BUFFER	This strategy buffers all the values of <code>onNext()</code> until it has not been consumed by the downstream
DROP	This strategy drops the most recent value given by <code>onNext()</code> , if the consumer is not able to keep that
ERROR	In this strategy, if the downstream is not able to keep up, <code>MissingBackPressureException</code> will be signaled
LATEST	In this strategy, the latest value (if it's not been used by the subscriber) will be overwritten by the recent value of <code>onNext()</code>
MISSING	The <code>onNext()</code> method events are written without buffering

The Maybe<T> operator

This operator has been introduced in RXJava 2.0.0.RC2 as a combination of `Completable` and `Single` operators. `Maybe` emits either 0 or one item and if the emission is successful it will terminate with the invocation of the `onComplete()` method. If something goes wrong an error will be thrown.

Let's write a code snippet that will demonstrate the `Maybe()` operator. We will use it for the list of months, and observe the absence of the `onNext()` method while dealing with the code:

```
public class DemoMaybe {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Maybe<List<String>> month_maybe = Maybe.create(emitter -> {
            try {
                String[] monthArray = { "Jan", "Feb", "Mar", "Apr",
                    "May", "Jun", "July", "Aug", "Sept", "Oct", "Nov", "Dec" };
                List<String> months = Arrays.asList(monthArray);
                if (months != null && !months.isEmpty()) {
                    emitter.onSuccess(months);
                }
                else {
                    emitter.onComplete();
                }
            }
            catch (Exception e) {
                emitter.onError(e);
            }
        });
        month_maybe.subscribe(s->System.out.println(s));
    }
}
```

We have just handled the `onSuccess()` and `onComplete()` events as we are dealing with `Maybe`.

Completable

`Completable` has been included in the APIs in the 1.0.x version. `Completable` never returns a value, it will either be terminated successfully by emitting a success event or an error event. As `Completable` doesn't need backpressure, it doesn't stream a single value.

Let's create a `Completable` observable `fromAction()` operator as shown in the following code:

```
public class DemoCompletable {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Completable completable=Completable.fromAction(()->  
            System.out.println("Welcome to Completable"));  
        completable.subscribe();  
    }  
}
```

If you observed the code snippet carefully, `fromAction()` has accepted `Action` implementation that doesn't return any value. As discussed earlier, instead of the `fromAction()` operator, we can use other `fromXXX()` operators to get `Completable`.

As we know, `Completable` doesn't return values; however, it is still of importance due to its representation of either successful completion or the failure of the event. Let's take an example of writing the data to the database. Whenever we write the data to the database, we are interested in whether the data got inserted in the database or not. Same is the case with file writing, as a developer, the file has been written or not, is important to us. Is this the same case when we want to read the data or a file? No, if it's a reading operation the values of the data read from the source are more important and not whether the values are read successfully or not. The point is if as a developer, you are interested in just whether the operation is taking place successfully or not, then use `Completable` as an observable.

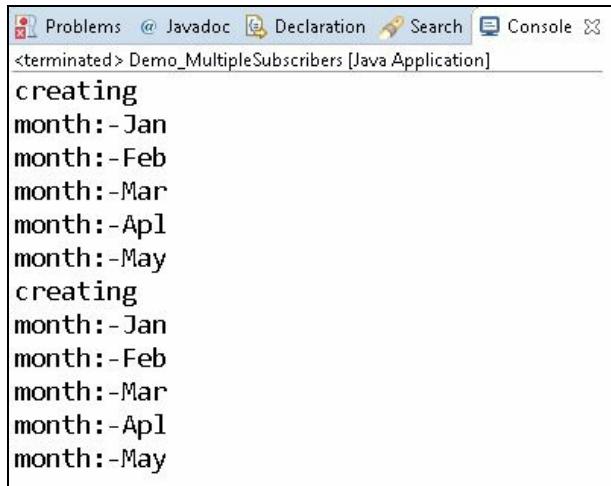
In the earlier discussion, we have created or used a single subscriber. However, under practical scenarios, we need to handle multiple subscribers at

a time. The source of the data which is `observable` doesn't emit the data until someone subscribes to it. Isn't it simple? Has to be, unfortunately, it's not! Whenever we invoke the `subscribe()` method, the subscription handler from the `create()` method will be invoked. Do we need a separate handler each time? Maybe under certain scenarios but not every time. In case, if we are handling heavy weight computation, then using single subscription handler is preferred. Let's write a sample code to understand the multiple subscribers as follows:

```
public class Demo_MultipleSubscribers {
    public static void main(String[] args) {
        String[] monthArray = { "Jan", "Feb", "Mar", "Apr", "May" };
        List<String> months = Arrays.asList(monthArray);

        Observable<Object> observable =
            Observable.create(subscriber -> {
                try {
                    System.out.println("creating ");
                    for (String data : months) {
                        subscriber.onNext(data);
                    }
                    subscriber.onComplete();
                } catch (Exception e) {
                    // TODO: handle exception
                    subscriber.onError(e);
                }
            });
        observable.subscribe(item -> System.out.println("month:-"
            + item));
        observable.subscribe(item ->
            System.out.println("month:-" + item));
    }
}
```

On invocation, we will get all the names of the months shown as follows:



```
Problems @ Javadoc Declaration Search Console X
<terminated> Demo_MultipleSubscribers [Java Application]
creating
month:-Jan
month:-Feb
month:-Mar
month:-Apr
month:-May
creating
month:-Jan
month:-Feb
month:-Mar
month:-Apr
month:-May
```

Along with the values of the data, have you observed how many times `creating` is printed? Yes, it got printed twice. It means we have given the call to the `create()` method twice. Now, the question is, can we call the method only once, irrespective of how many subscribers have been created? The `cache()` operator helps in reusing the subscription handler. Let's update the code shown as follows:

```
Observable<Object> observable = Observable.create("// previous
code as it is).cache();
```

If we execute the code now, we will get a single call to the `create()` method.

Understanding the Disposable interface

We discussed in depth about the emission of the items from `observable`, which will be consumed by `subscriber`. Not necessary all the emitted items every time needs to be consumed. It all depends on the scenario the developers are dealing with. We need to have a way to discard the resources which are in use. The `Subscriber` doesn't have such method. However, the `Disposable` interface provides the useful `dispose()` method. The `Disposable` interface is shown as follows:

```
public interface Disposable {  
    void dispose();  boolean isDisposed();  
}
```

The nonfunctional interface, `Disposable`, is now responsible for life cycle management of streams and resources. Now, subscribing `observable` happens when someone uses `DisposableObserver` and to unsubscribe the `dispose()` method, it needs to be invoked which is analogous to the `Subscription.cancel()` method.



RxJava1 has `Subscriber` which now has been renamed to `Disposable`.

Let's get back to `Subscriber` to discuss a few subscribers which are `Disposable` too.

Disposable Subscribers

Is Subscriber Disposable? Yes! It is. The abstract class `DisposableSubscriber`, `ResourceSubscriber`, and `DefaultSubscriber` implement the `Disposable` interface which exposes the `dispose()` method. Let's discuss them one by one.

The DefaultSubscriber class

The `DefaultSubscriber` is an abstract class and the predefined methods are final as well as thread safe. It allows the cancellation of upstream Subscription's using the `cancel()` method. It provides an `onstart()` method which by default will request `Long.MAX_VALUE`. We can override it to request a customized number of items. The developers need to use the `cancel()` method from the body of the `onNext()` method to cancel the sequence of the items. Let's develop a `DefaultSubscriber` class which will subscribe the values generated by `observable`. Our subscriber will request for 10 items from the `observable`. We will use `Flowable` to emit the number of type `Long` from 2 to 17 using the `rangeLong()` method shown as follows:

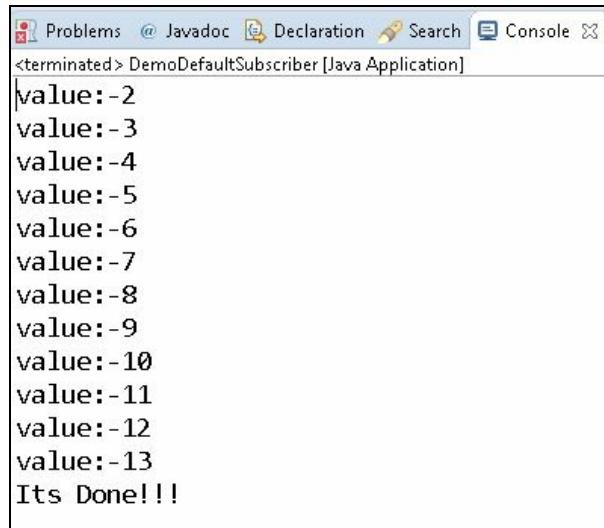
```
public class DemoDefaultSubscriber {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Flowable.rangeLong(2, 12).subscribe(new
            DefaultSubscriber<Long>() {

                @Override
                public void onComplete() {
                    // TODO Auto-generated method stub
                    System.out.println("Its Done!!!");
                }
                @Override
                public void onError(Throwable throwable) {
                    // TODO Auto-generated method stub
                    throwable.printStackTrace();
                }

                @Override
                public void onNext(Long value) {
                    // TODO Auto-generated method stub
                    System.out.println("value:-"+value);
                }
            });
    }
}
```

Execute the code to get following output:



```
value:-2
value:-3
value:-4
value:-5
value:-6
value:-7
value:-8
value:-9
value:-10
value:-11
value:-12
value:-13
Its Done!!!
```

Have we done something different? Actually no. It's very much similar to the earlier code. Let's update the code to override the `onStart()` method to the `request()` method and customize the number of items instead of the default request as shown in the following code:

```
    @Override
    protected void onStart() {
        request(5);
    }
```

Here, we are requesting for five items from the `observable`. Execute the code and get the following output on the console:



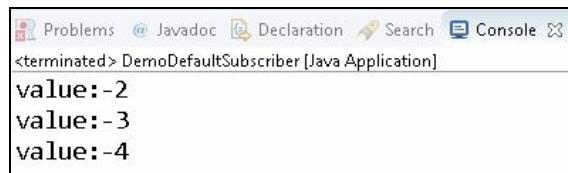
```
value:-2
value:-3
value:-4
value:-5
value:-6
```

Now, we understand how to request items as per requirement. We haven't got `Its Done` printed on the console. The reason is very simple. We haven't completed the process normally. Can we cancel the requested sequence? Yes, we can cancel it via the `cancel()` method from the `onNext()` method. Let's update the `onNext()` method shown as follows:

```
    @Override
    public void onNext(Long value) {
        if(value==4)
```

```
    cancel();
    System.out.println("value:-" + value);
}
```

Execute the code to get the following output:



A screenshot of an IDE's console window. The window has a header with tabs: Problems, Javadoc, Declaration, Search, and Console. Below the header, it says '<terminated> DemoDefaultSubscriber [Java Application]'. The main content area of the console shows three lines of text: 'value:-2', 'value:-3', and 'value:-4'. The text is in a monospaced font.

```
value:-2
value:-3
value:-4
```

The output clearly shows we stop the subscription in between once we reached the value of item to 4.

DisposableSubscriber

Now, we don't need to use the external `dispose()` method, we have `Subscriber` which is now `Disposable` and so do the internal `dispose()` method, which is quite similar to the `unsubscribe()` method provided by the `Subscription` interface. The predefined methods from the class are final as well as thread safe. The cancellation of a `Subscription` which is upstream using the `cancel()` method is allowed by it.

The following code snippet will demonstrate `DisposableSubscriber`:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    DisposableSubscriber<Long> disposableSubscriber = new
    DisposableSubscriber<Long>() {

        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("Its Done!!!");
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }

        @Override
        public void onNext(Long value_long) {
            // TODO Auto-generated method stub
            System.out.println("value :- " + value_long);
        }

        @Override
        protected void onStart() {
            // TODO Auto-generated method stub
            request(Long.MAX_VALUE);
        }
    };

    Flowable.rangeLong(5, 4).subscribe(disposableSubscriber);
    disposableSubscriber.dispose();
}
```

Execute the code to get the items emitted by the `observable` printed on the console. Now, observe the code, specifically the last line of the code,

`disposableSubscriber.dispose()` which is for unsubscribing the items.

In case of canceling the subscription in between, we can use the `cancel()` method from the body of the `onNext()` method shown as follows:

```
public void onNext(Long value_long) {  
    // TODO Auto-generated method stub  
    if(value_long==7)  
        dispose();  
    System.out.println("value :- " + value_long);  
}
```

The updated code will cancel the sequence of items after the value is reached to 7.

The ResourceSubscriber interface

Similar to `DisposableSubscriber`, `ResourceSubscriber` also has the `dispose()` method which facilitates the release of all associated resources. The developers usually need to call the `dispose()` method from the body of `onError()` method and `onComplete()` method. The `ResourceSubscriber` interface facilitates the cancellation of its subscription and associated resources asynchronously. The predefined methods such as `add()`, `dispose()`, `onSubscribe()` are final as well as thread safe. The cancellation of a `Subscription` which is upstream using the `dispose()` method is allowed by it.

It's very necessary to do all the initializations before the call to the `request()` method from the body of the `onNext()` may immediately trigger the emission of the items.

Observe the use of the `dispose()` method in the following code for unsubscription:

```
public static void main(String[] args) {

    ResourceSubscriber<Long> resourceSubscriber = new
        ResourceSubscriber<Long>() {

        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("Its Done!!!");
            dispose();
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
            dispose();
        }

        @Override
        public void onNext(Long value_long) {
            // TODO Auto-generated method stub
            System.out.println("value :- "+value_long);
        }
    };
}
```

```
|     Flowable.rangeLong(5, 4).subscribe(resourceSubscriber);
|     resourceSubscriber.dispose();
| }
```

On execution of the code, we will get the items displayed on the console. In case of canceling the sequence of items in between we can call the `dispose()` method as we did in the earlier case.

CompositeDisposable

The final class `compositeDisposable` implements `Disposable` interface, which works as a container holds multiple disposables. It means we can have many disposables which now can be treated as one. Whenever we call the `dispose()` method on `CompositeDisposable`, it will internally call `dispose` on each of the resources. It's preferred to use whenever the developers want to multiple `Subscription` executed one by one. All they need to do is to add the `Subscription` to `CompositeDisposable` as shown in the following code:

```
public class DemoCompositeDisposable {
    public static void main(String[] args) {
        CompositeDisposable disposable = new CompositeDisposable();
        disposable.add(Flowable.rangeLong(10, 5).
            subscribe(System.out::println));
        disposable.add(Flowable.rangeLong(1, 5).subscribe(item ->
            System.out.println("two" + item)));

        disposable.add(Observable.create(new
            ObservableOnSubscribe<String>() {

                @Override
                public void subscribe(ObservableEmitter<String> emitter)
                    throws Exception {
                    try {
                        String[] monthArray = { "Jan", "Feb", "Mar", "Apl",
                            "May", "Jun", "July", "Aug", "Sept", "Oct", "Nov", "Dec"
                        List<String> months = Arrays.asList(monthArray);

                        for (String month : months) {
                            emitter.onNext(month);
                        }
                        emitter.onComplete();
                    }
                    catch (Exception e) {
                        // TODO: handle exception
                        emitter.onError(e);
                    }
                }
            }).subscribe(s -> System.out.println(s)));
        disposable.dispose();
    }
}
```

We have created three different `Observables` emitting different types of items, we need them to emit the items one by one. What we did is just to add them

to `CompositeDisposable` instead of creating and subscribing them separately.

`CompositeDisposable` also provides the following useful methods:

- `addAll()`: This method facilitates the automatic addition of the given array of `Disposable` to the container. Whenever the container disposes, all of its `Disposable` also disposes.
- `remove()`: This method facilitates removal of `Disposable` from the container.
- `delete()`: This method facilitates deletion of `Disposable` from the container.
- `clear()`: This method facilitates to clear the container. Once the container is cleared it disposes all the previously contained `Disposable`.

Subject

We discussed `observable` and `subscriber` as two different components of the RxJava. A `Subject` is a very interesting component which can act as both `observer` as well as `observable`. As it's an `observer`, it can subscribe to one or many `observables`. Also, being an `observable`, it can emit item which can be subscribed.

Flavors of Subject

RxJava has provided four different flavors of Subject which fit in different scenarios. Let's discuss them one by one:

The AsyncSubject class

The final class `AsyncSubject` is used to emit the last value in a sequence. After the emission of the value, it will emit the completion event or an error to the observer. The implementation of the callback methods are thread safe, however, in case of the non-serialized call, it may lead to an undefined state of the observer which is currently subscribed.

Let's write the code in which the subject will emit more than one element and subscribe to the display shown as follows:

```
public class Demo_AsyncSubject {
    public static void main(String[] args) {
        AsyncSubject<Long> asyncSubject=AsyncSubject.create();
        asyncSubject.subscribe(new Observer<Long>() {

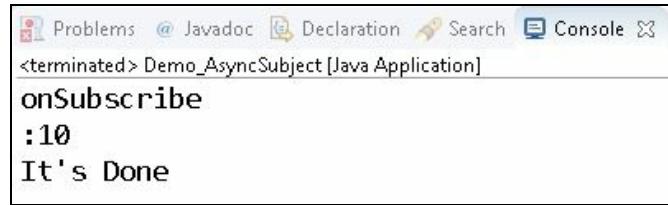
            @Override
            public void onComplete() {
                // TODO Auto-generated method stub
                System.out.println("It's Done");
            }

            @Override
            public void onError(Throwable throwable) {
                // TODO Auto-generated method stub
                throwable.printStackTrace();
            }

            @Override
            public void onNext(Long value) {
                // TODO Auto-generated method stub
                System.out.println(": "+value);
            }

            @Override
            public void onSubscribe(Disposable disposable) {
                // TODO Auto-generated method stub
                System.out.println("onSubscribe");
            }
        });
        asyncSubject.onNext(1L);
        asyncSubject.onNext(2L);
        asyncSubject.onNext(10L);
        asyncSubject.onComplete();
    }
}
```

On execution, we will get the output shown as follows:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_AsyncSubject [Java Application]
onSubscribe
:10
It's Done
```

If you observe the output carefully, it's clear that only the last value has been emitted and after that, the `Observable` completes. In case `Observable` doesn't emit any value, the `AsyncSubject` class completes normally without emitting any value.

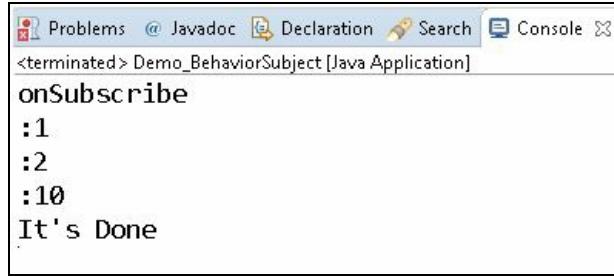
The BehaviorSubject class

`BehaviorSubject` emits the most recently emitted item and all the subsequent items once `Observer` subscribes to it.

Let's discuss it using the following code:

```
public class Demo_BehaviorSubject {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Observer<Long> observer=new Observer<Long>() {  
            @Override  
            public void onComplete() {  
                System.out.println("It's Done");  
            }  
  
            @Override  
            public void onError(Throwable throwable) {  
                throwable.printStackTrace();  
            }  
  
            @Override  
            public void onNext(Long value) {  
                System.out.println(": "+value);  
            }  
  
            @Override  
            public void onSubscribe(Disposable disposable) {  
                System.out.println("onSubscribe");  
            }  
        };  
        BehaviorSubject<Long>  
        behaviorSubject=BehaviorSubject.create();  
        behaviorSubject.subscribe(observer);  
        behaviorSubject.onNext(1L);  
        behaviorSubject.onNext(2L);  
        behaviorSubject.onNext(10L);  
        behaviorSubject.onComplete();  
    }  
}
```

Here, the `Observable` is emitting three items, `1L`, `2L`, and `10L` which will be subscribed by the `Observer`. Execute the code, and you will get the following code:



```
onSubscribe
:1
:2
:10
It's Done
```

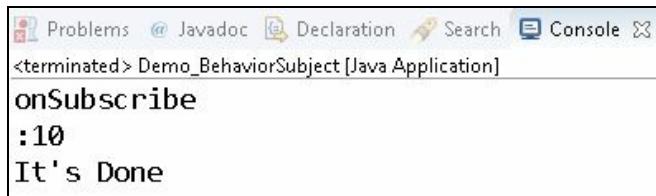
The output shows the most recent as well as the sequence which will be emitted by the `observable`, as opposed to `AsyncSubject` where only the most recent item will be emitted.

Keeping the `Observer` the same, let's modify the earlier code as follows:

```
BehaviorSubject<Long> behaviorSubject=BehaviorSubject.create();
behaviorSubject.onNext(1L);
behaviorSubject.onNext(2L);
behaviorSubject.onNext(10L);

behaviorSubject.subscribe(observer);
behaviorSubject.onComplete();
```

On execution, we will get the following output:



```
onSubscribe
:10
It's Done
```

Instead of getting the sequence, what we get is the latest emitted value, as all the previous values emitted have been overridden, only recently emitted value is available for the subscription. Once it's subscribed, the `onComplete` event will be raised.

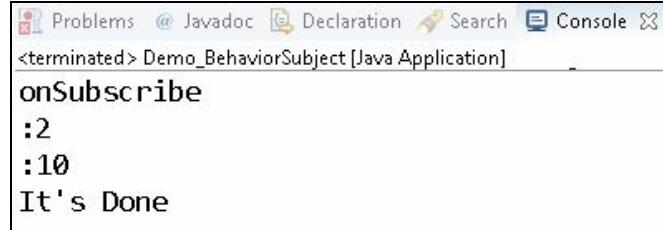
To make `BehaviorSubject` clear, let's update the earlier code as:

```
BehaviorSubject<Long> behaviorSubject=BehaviorSubject.create();
behaviorSubject.onNext(1L);
behaviorSubject.onNext(2L);

behaviorSubject.subscribe(observer);
behaviorSubject.onNext(10L);

behaviorSubject.onComplete();
```

After first two elements `1L` and `2L` we invoked the `subscribe()` method. It means the sequence for emission is `2L` followed by `10L` and then the `onComplete()` method will be invoked as shown in the following output:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_BehaviorSubject [Java Application]
onSubscribe
:2
:10
It's Done
```

Consider all the earlier situations, `Observable` was emitting items, what if we got an error? Let's find out the answer by updating the code as follows:

```
BehaviorSubject<Long> behaviorSubject=BehaviorSubject.create();
behaviorSubject.onNext(1L);
behaviorSubject.onNext(2L);
behaviorSubject.onNext(10L);

behaviorSubject.onError(new Exception("You got an Exception"));

behaviorSubject.subscribe(observer);
behaviorSubject.onComplete();
```

The output of the code shows no element will be emitted. Only the `onError` event will be received by the observer.

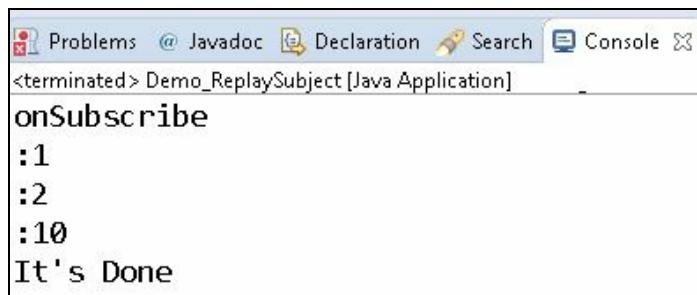
ReplaySubject

Independent of when the subscription happens by the `observer`, all the items emitted by the `observable` will be available. It means independent of whether the subscription is after partial emission or after complete emission of items, all the items will be consumed by the `observer`.

To demonstrate `ReplaySubject` let's write down the updated code. We will keep the `observer` the same as discussed in the earlier sample code. The updated code is shown as follows:

```
public class Demo_ReplaySubject {  
  
    public static void main(String[] args) {  
        //get an observer as did in earlier code  
        ReplaySubject<Long> replaySubject=ReplaySubject.create();  
        replaySubject.onNext(11);  
        replaySubject.onNext(21);  
        replaySubject.subscribe(observer);  
        replaySubject.onNext(101);  
        replaySubject.onComplete();  
    }  
}
```

On execution we will get all the items displayed on the console as shown in the following output:



```
onSubscribe  
:1  
:2  
:10  
It's Done
```

In the code, we have invoked `subscribe` in between, still, all the items emitted by the `observable` are subscribed by the `observer`, totally opposite to `BehaviourSubject`.

PublishSubject

observer will observe only those items or sequence of items which are available at the time of subscription. Whatever a number of items emitted after subscription will be observed and not those which are emitted before, as shown in the following code:

```
public class Demo_PublishSubject {
    public static void main(String[] args) {
        // get observer as did in earlier code
        PublishSubject< Long>
            publishSubject=PublishSubject.create();
        publishSubject.onNext(1L);
        publishSubject.onNext(2L);
        publishSubject.subscribe(observer);
        publishSubject.onNext(10L);
        publishSubject.onNext(20L);
        publishSubject.onComplete();
    }
}
```

On execution, we will get only get items with values 10 and 20 observed by the observer as these are emitted after the subscription.

Let's summarise all the flavors of the subjects with the help of an example of a serial coming every day on television. Each day before the episode starts, we will get a recap of the most recent event happened in the serial, this is BehaviourSubject.

In between the serial, we have commercial breaks, once the break is over, we get a recap of the part of the serial before the commercial break. It is a scenario of AsyncSubject.

Nowadays, we may have the facility to save the serials even if we are offline. Such saved serials can be rewound and seen from the beginning. This is what is given by ReplaySubject.

We are not interested what happened in the past, and just want to enjoy the ongoing part, we are dealing with PublishSubject which is only interested in what is going on and not what happened.

Summary

The source and the consumer are the backbones of RxJava. In this chapter, we discussed in depth about these backbones. `Observable` is the source of the data and `Observer` is a consumer. The `Observable` interface has many operators which enable the creation of the `Observable`. We discussed in depth about `just()`, `empty()`, `never()`, `from()`, `throw()` operators and also demonstrated their use. We not only created the `Observable`, but also discussed types of `Observable` such as `Flowable`, `Single`, and `Maybe` in depth. The enhanced version of `Observer` is `Subscriber` which supports handling of backpressure. `DefaultSubscriber`, `DisposableSubscriber`, `ResourceSubscriber`, and `CompositeSubscriber` are the types of `Subscribers` which we covered in our discussion. The `Subject` is a very interesting component which can act as both `Observer` as well as `Observable`. By taking examples, we discussed `AsyncSubject`, `BehaviorSubject`, `ReplaySubject`, and `PublishSubject`.

We now understand well how to create `Observable` and `Subscriber`, however, we haven't covered transformation, combination, filtration and error handling operators; we will look at these in the next chapter.

Operators

In [Chapter 4](#), *Reactive Types in RxJava*, we focused on understanding the components involved in the creation of a reactive application using RxJava. Our intention was to understand the ins and outs of the components and their various types that have been provided by the RxJava library. We discussed operators in depth such as `create()`, `just()`, and `from()` facilitating `observable` creation. However, a long list of operators remains to be discovered and will allow the creation, use, and manipulation of the items emitted by `observable`.

In this chapter, we will discuss the operators provided by RxJava, enabling developers to handle asynchronous events for filter and transformation with the help of the following topics:

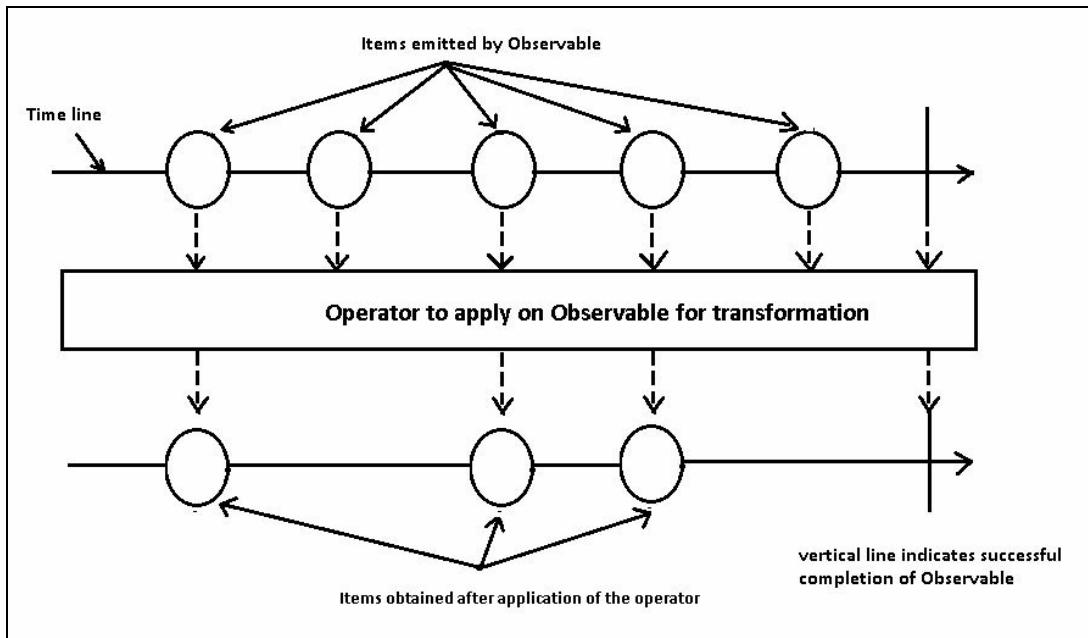
- Demystifying RxMarbles
- Transforming events
- Combining reactive flows
- Filtering out events
- Conditional operators
- Utility operators

We discussed various ways to get an `observable` in the earlier chapter using predefined factory methods. Each method provides us with an `observable` for fulfilling our requirements. `observable` emit items that can be consumed, transformed, one `observable` can be combined with the others to get one more `observable` using different operators. The built-in powerful operators in RxJava make the operation easy. Along with built-in operators, developers can create custom operators as well whenever required. The operators help developers to handle complex transformations easily. An operator is a function that accepts an `observable`, and processes and transforms it to return an X. The accepted and returned `observable` may or may not be of the same type.

Demystifying RxMarbles using an Observable

Let's take into consideration an operator used to filter an observable by emitting the names of months. We will apply the filter to get all the months having a name-length more than 3 such as April and July. Once the operator `filter()` is applied on the original observable we will get only a couple of month names. Yes, it happened due to the application of the filter. Does that mean we lost our original observable and now we can't recover it? Don't panic! Our original observable is untouched. Isn't that great?

Let's elaborate the scenario using a marble diagram. A marble diagram helps in visualizing the working of an operator; upstream we have observable, in the middle we have an operator, and downstream we have items that match the condition applied by the operator. Observe the following diagram:

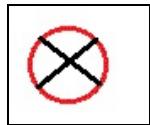


In the preceding figure, the timeline flows from left to right. The figure denotes the observable is emitting items. On those emitted items, the operator will perform operations. According to the applied operator, the items that

match the operator condition will be obtained on the other side.

Here we need to keep one thing in mind--under certain situations, the expected result cannot be obtained by a single operation; in such cases, multiple operations need to be performed in sequence so that the desired effect can be obtained. In such situations, the developers need to apply more than one operator in a sequence. Sometimes, we can apply the logical operators to get the desired output. We will discuss these situations in an upcoming section.

If the `Observable` terminates abnormally, we will denote it using the following figure:



RxJava provides a variety of operators that can be categorized as follows:

- `Observable` creation operators
- Operators for transforming `Observable`
- `Observable` filtering operators
- `Observable` creation operator by combining two `Observables`
- Error handling operators
- Utility operators
- Conditional operators
- Mathematical and aggregate operators
- Backpressure operators
- Connectable `Observable` operators
- Conversion operators

We know that a few of these facilitate `Observable` creation and we have already used them in an earlier chapter. Let's discover more operators one by one.

Transforming Observables using operators

Under this category, we will discuss the operators that facilitate the transformation of the items emitted by observable.

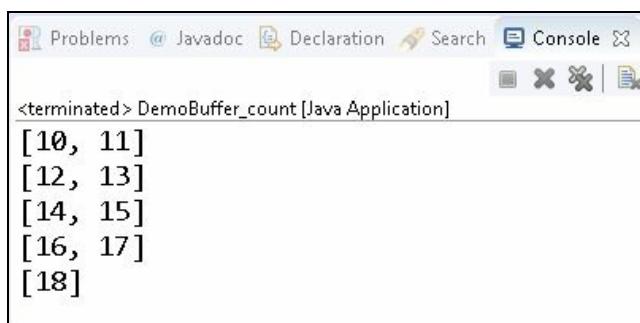
Using the buffer() operator

Usually the `observable` emits the items and the `observer` handles the emitted items one by one. Sometimes, we need to handle a bundle of items instead of individual items. The `buffer()` operator helps in creating a bundle or collection of the emitted items periodically. Under certain conditions `observable` emits an error instead of emitting an item. Under this scenario, even if the buffer contains a bundle of items, it immediately emits the error notification instead of processing the bundle.

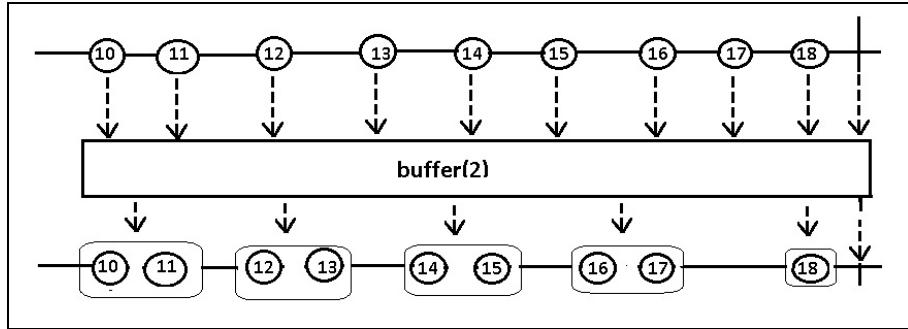
The `buffer()` operator accepts arguments of type `integer` where developers can specify how many items to bundle together. In the following code, the `range` observable will emit fifteen items starting from `10`. We will apply the `buffer()` operator to bundle two items together:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Observable.range(10, 9).buffer(2).subscribe(new
    Consumer<List<Integer>>() {
        @Override
        public void accept(List<Integer> items) throws Exception
        {
            // TODO Auto-generated method stub
            System.out.println(items);
        }
    });
}
```

After execution we will get the output as shown in the following screenshot:



The following marble diagram helps in understanding how the `buffer()` operator works:



The marble diagram shows the `buffer()` operator has bundled two items together.

The `buffer()` operator also has the ability to bundle the items together depending upon the time span. During the specified time, all the items emitted by the `observable` will be bundled together into windows with an equal length. At the end of each window, the complete bundle will be emitted.

Using the flatMap() operator

The `flatMap()` operator helps in transforming items emitted by `observable` into another `observable`. `flatMap()` accepts a function that will be applied to every item emitted by an `observable` to return `observable`. Later on, `flatMap()` merges a newly created `observable` and emits a sequence of merged results.

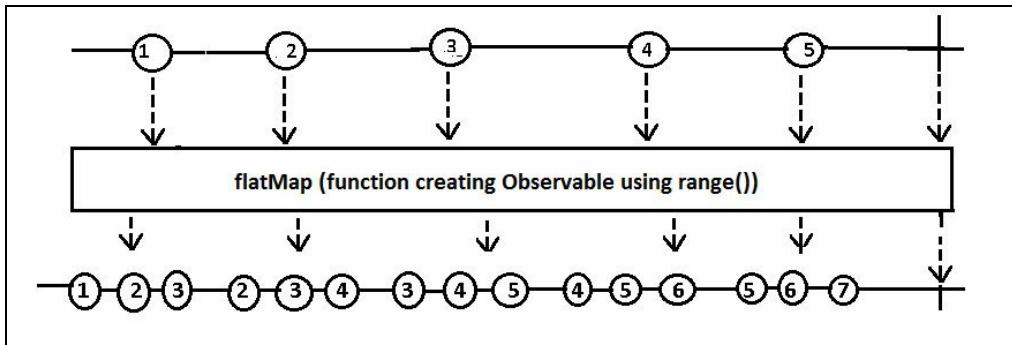
Let's take into consideration an `observable` created using the `range` operator, which is emitting items from `1` to `10`. Now every emitted item by the `observable` will be the starting range of the `observable` returned by the `flatMap()` as follows:

```
public class Demo_flatMap {  
    public static void main(String[] args) {  
        Observable.range(1,5).flatMap(item->Observable.range(item,  
            3)).subscribe(value->System.out.print(value+"->"));  
    }  
}
```

On execution we will get the following output:

```
1->2->3->2->3->4->3->4->5->4->5->6->5->6->7->
```

Observe the marble diagram denoting that the `observable` is emitting items from `1` to `5`. Each emitted item will be taken by the `flatMap()` function and in turn, it will emit three items. We are taking the item emitted by the source `observable` as the first item and after that two more items will be emitted. All the subsequent `observables` created by the `flatMap()` operator will be merged together:



We have a `map()` operator in RxJava. Why should we use `flatMap()`? A very obvious situation that arises during application development is handling exceptions. The `map()` operator allows handling runtime exception, but not checked exceptions. However, the `flatMap()` operator facilitates handling checked exceptions as follows:

```
Observable.range(1, 10)
    .flatMap(v -> {
        if (v < 5) {
            return Observable.just(2* v);
        }
        if(v==6)
        {
            return Observable.error(new Throwable("You Got An Exception"));
        }
        return Observable.just( v);
    }).subscribe(System.out::println, Throwable::printStackTrace);
```

On execution of the preceding code, we will get the first five items emitted and then since the condition is matching we will get an exception on the console. What if we want all the items emitted first and then throw an exception so that `Observer` will get all the items? Here we need to delay the error emission until all the sources get terminated

The operator also has an overloaded version that takes a `Boolean` value which specifies a delay if the error occurs as follows:

```
Observable.range(1, 10)
    .flatMap(v -> {
        if (v < 5) {
            return Observable.just(2* v);
        }
        if(v==6)
        {
            return
            Observable.error(new Throwable("You Got An Exception"));
        }
    }).subscribe(System.out::println, Throwable::printStackTrace);
```

```
    return Observable.just( v );
},true).subscribe(System.out::println,
Throwable::printStackTrace);
```

The output of the code will execute all the items from 1 to 10 and create `observable` as per the matching conditions; when an item with value 6 is emitted an exception will be thrown. Instead of directly throwing an exception we are asking `flatMap()` to delay the throwing of the exception until the `observable` has completed the emission.

Using the groupBy() operator

The `observable` will emit some items that then will be divided into different `sub Observable`. The items will be organized into a subset of an `observable` depending upon the discriminating functions that evaluate each item and assign a key to it. According to the assigned keys the new `observables` will be formed and will act as a source now.

Let's consider an array of `String` as our `observable`. On the emitted items we will apply the `groupBy()` operator to group together items as per their lengths as discussed in the following code:

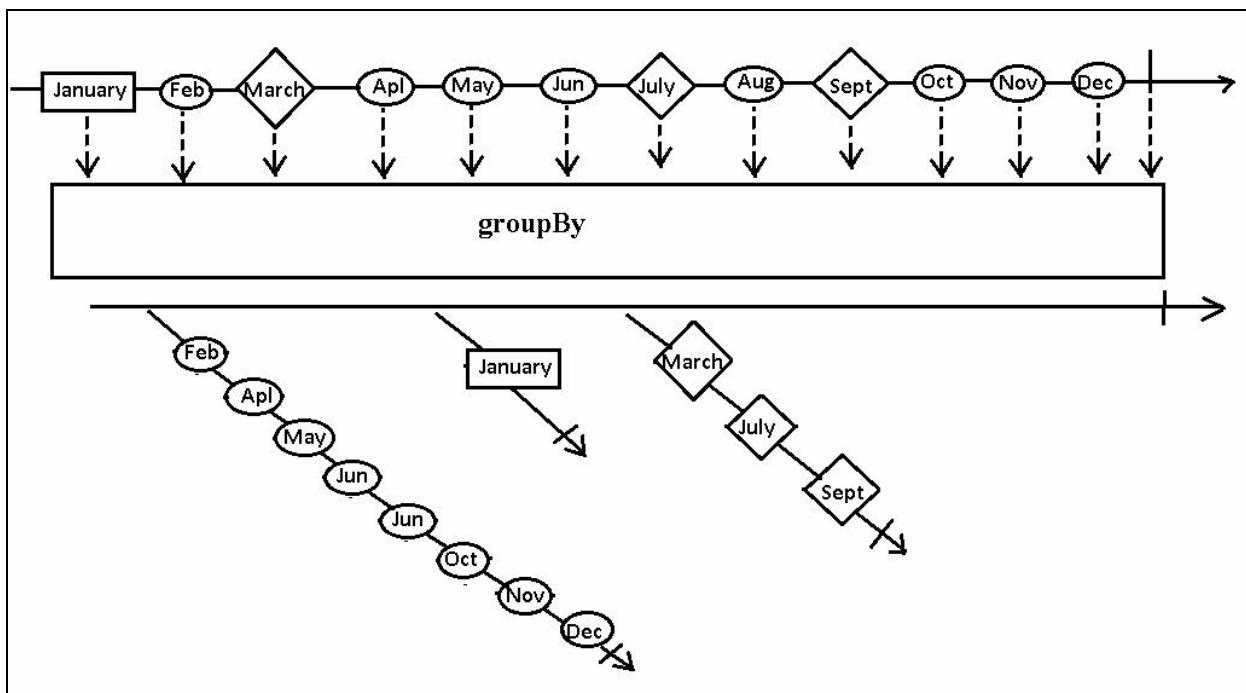
```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    String[] monthArray = { "January", "Feb", "March", "Apl",
        "May", "Jun", "July", "Aug", "Sept", "Oct", "Nov",
        "Dec" };

    Observable.fromArray(monthArray).groupBy(item -> {
        if (item.length() <= 3)
            return "THREE";
        else if (item.length() >= 4 && item.length() < 6)
            return ">4";
        return "DEFAULT";
    }).subscribe(observable -> {
        // System.out.println(observable.getKey());
        observable.subscribe(item -> System.out.println(item +
            ":" + observable.getKey()));
    });
}
```

Observe the following output, that shows the items having a length greater than 4 and less than 6 will be assigned the key `DEFAULT`. Items with a length less than three will get the key as `THREE` and all the rest will get a key `>4`:

```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_groupBy [Java Application]
January :DEFAULT
Feb :THREE
March :>4
Apl :THREE
May :THREE
Jun :THREE
July :>4
Aug :THREE
Sept :>4
Oct :THREE
Nov :THREE
Dec :THREE
```

The operation can be summarized as follows:



Using the map() operator

The `map` operator helps the developers to transform each emitted item from an `Observable` by applying a function to it. Once the item is emitted by the `Observable` a specified function will be applied on each item, resulting in returning `Observable`, which in turn emits the results achieved after transformation.

Let's apply the `map()` operator to find out the number of days of each month. Our `Observable` will emit the name of the month. The function inside the `map()` operator will check the name of the month and accordingly will emit the number of days as shown by the following code:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub

    String[] monthArray = { "January", "Feb", "March", "Apl",
        "May", "Jun", "July", "Aug", "Sept", "Oct", "Nov",
        "Dec" };

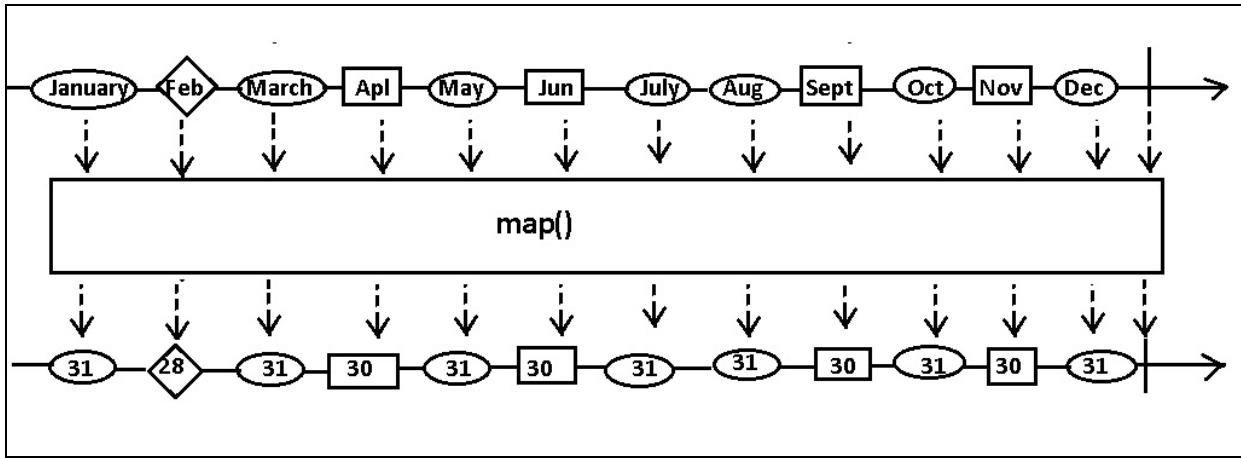
    Observable.fromArray(monthArray).map((item) -> {
        if (item.equals("January") || item.equals("March") ||
            item.equals("May") || item.equals("July") ||
            item.equals("Aug") || item.equals("Oct") ||
            item.equals("Dec"))
            return 31;
        else if (item.equals("Apl") || item.equals("Jun") ||
            item.equals("Sept") || item.equals("Nov"))
            return 30;
        return 28;
    }).subscribe(new Observer<Integer>() {
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("sequence completed");
        }

        @Override
        public void onError(Throwable arg0) {
            // TODO Auto-generated method stub
        }

        @Override
        public void onNext(Integer no_of_days) {
            // TODO Auto-generated method stub
            System.out.println("number of days"+no_of_days);
        }
    });
}
```

```
    @Override
    public void onSubscribe(Disposable disposable) {
        // TODO Auto-generated method stub
    }
});
```

On execution of the code, instead of getting the names of the months, we will get their number of days. The following marble diagram will clarify this:



Using the scan() operator

The `scan()` operator helps by applying a function to each item emitted by the `Observable` to emit successive values sequentially. The `scan()` function accepts a function that will be applied to the first emitted item. The resultant item is now the emission. The result will again be fed back to the function, now the function will be applied on the second item emitted by the original source or `Observable` to give the second emission. And the process continues until the `observable` completes its emission. Let's discuss a simple `observable` emitting prices of items purchased. We will add the price of each purchased product to get the final total using the `scan()` operator as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Integer[] prices = { 100, 20, 40, 10, 15 };
    Observable.fromArray(prices).scan((item1, item2) -> item1 +
    item2) .subscribe(new Observer<Integer>() {
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("completed the sequence
                successfully");
        }
        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }

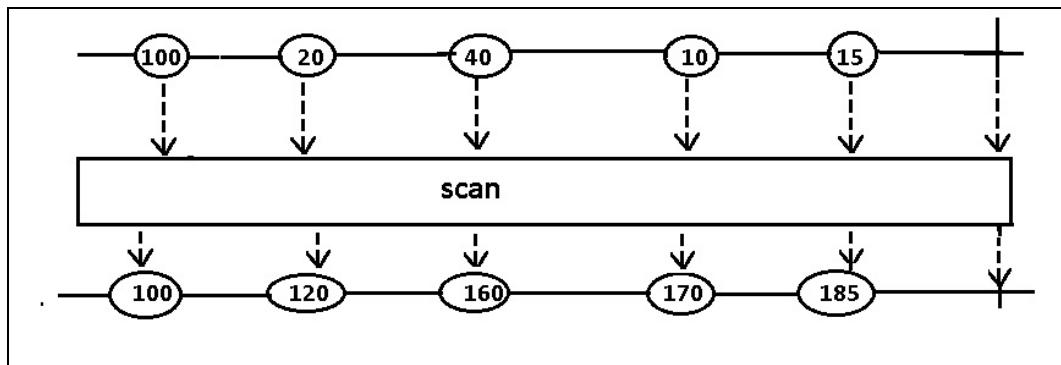
        @Override
        public void onNext(Integer item) {
            // TODO Auto-generated method stub
            System.out.println("amount:" + item);
        }

        @Override
        public void onSubscribe(Disposable arg0) {
            // TODO Auto-generated method stub
        }
    });
}
```

On execution we will get the following output:

```
Problems @ Javadoc Declaration Search Console 
<terminated> Demo_scan [Java Application]
amount:100
amount:120
amount:160
amount:170
amount:185
completed the sequence successfully
```

If you observe the output, clearly the addition of the first two numbers is fed back to the function in the map in order to consume the third item. Sequentially, the value of each item emitted by `Observable` will be added in the original amount to give the final result, as shown by the following marble diagram:



Using the `window()` operator

The `window()` operator allows developers to periodically bundle items emitted by an observable into an observable window and then emits each of these bundled windows rather than emitting one item at a time. We already have discussed the `buffer()` operator, which bundles items together into a packet and then emit these packets. The `window()` operator emits `observable` rather than emitting packets. We usually use the terminologies *window open* and *window close*, which suggests the source `observable` emits items and the new `observable` will start emitting items which have been emitted by original `observable`. Then the source `observable` has stopped the emission and has been terminated respectively.

Filtering Observable

The operators under this category help developers to write `observable` to emit selective items. Let's discuss them one by one.

Using the `debounce()` operator

The `debounce()` operator helps developers to write an observable that will emit items only after a particular time span has been passed without emitting any item.

Using the `distinct()` operator

An `Observable` can emit n number of items that may or may not have duplicate items. We can suppress the duplicate items emitted by the `Observable` using the `distinct()` operator. Now this duplication filtering can be a sequence of two immediate items of the same value.

Let's consider a list of fruits whose names we want to learn, rather than how many units in total there are, as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    String [] fruits= {"mango","pineapple","apple","mango","papaya",
                      "pineapple","apple","apple"};
    Single<Long> sing= Observable.fromArray(fruits).distinct().
        count();
    sing.subscribe(System.out::println);
    Observable.fromArray(fruits).distinct().
    subscribe(new Observer<String>() {

        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("collected all distinct items successfully");
        }

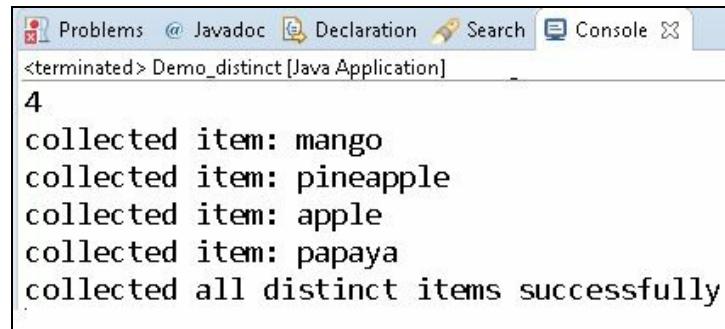
        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }

        @Override
        public void onNext(String value) {
            // TODO Auto-generated method stub
            System.out.println("collected item: "+value);
        }

        @Override
        public void onSubscribe(Disposable arg0) {
            // TODO Auto-generated method stub
        }
    });
}
```

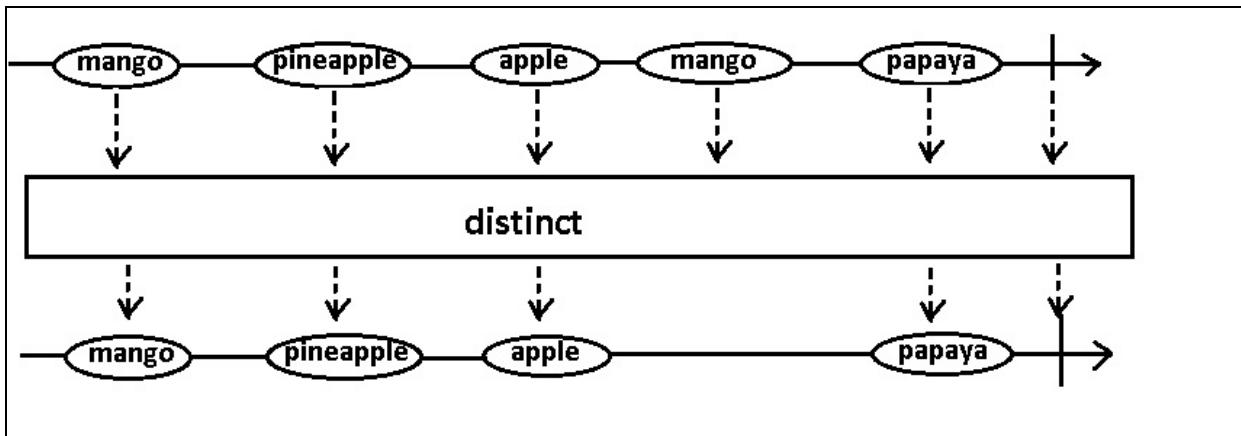
On execution of the preceding code, types of fruits in our list is four and not five, which specifies we have four distinct fruits, as shown in the following

screenshot:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_distinct [Java Application]
4
collected item: mango
collected item: pineapple
collected item: apple
collected item: papaya
collected all distinct items successfully
```

We can get better exposure to the `distinct()` operator with the help of the following marble diagram:



We can also have one more scenario as two fruits consecutively follow each other and we just want to consider the separate fruits as shown by in the following section.

Using the `distinctUntilChanged()` operator

We also can use the `distinctUntilChanged()` operator, which facilitates choosing distinct items that follow each other. If we get any two or more similar items consecutively in a sequence only one will be emitted and the rest of the duplicated items will not be shown by the following code:

```
public static void main(String[] args) {
    String[] fruits = { "mango", "pineapple", "apple", "mango",
        "papaya", "pineapple", "apple", "apple" };

    Single<Long> sing = Observable.fromArray(fruits).
        distinctUntilChanged().count();
    sing.subscribe(System.out::println);

    Observable.fromArray(fruits).distinctUntilChanged().
        subscribe(new Observer<String>() {

        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("collected all items successfully");
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }

        @Override
        public void onNext(String value) {
            // TODO Auto-generated method stub
            System.out.println("collected item: " + value);
        }

        @Override
        public void onSubscribe(Disposable arg0) {
            // TODO Auto-generated method stub
        }
    });
}
```

Using the elementAt() operator

The `Observable` emits a sequence of items. The `elementAt()` operator facilitates the developers to specify an index number; only the value from the specified index number will be available to the `Observer`. It's very similar to fetching the value at an index from an array or `java.util.ArrayList`. The only difference here is that we are asking the source `Observable` to emit the particular item and are not fetching the item.

Let's take the example of a list of fruits and apply the `elementAt()` operator to fetch the item from the fourth position as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    String[] fruits = { "mango", "pineapple", "apple", "mango",
        "papaya", "pineapple", "apple", "apple" };

    Observable.fromArray(fruits).elementAt(3).count()
        .subscribe(item -> System.out.println("we got: " +
            item + " items from the Observable"));

    Observable.fromArray(fruits).elementAt(3).subscribe(new
        MaybeObserver<String>() {

        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("successfully completed");
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            System.out.println(throwable.getMessage());
        }

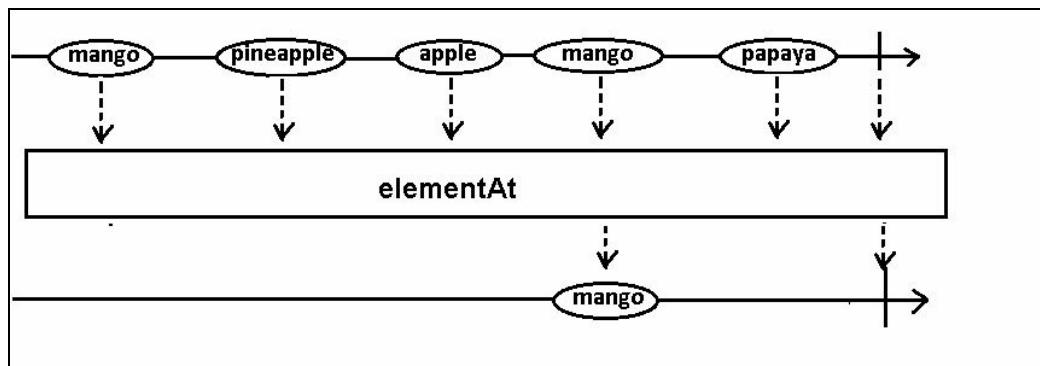
        @Override
        public void onSubscribe(Disposable disposable) {
            // TODO Auto-generated method stub
        }

        @Override
        public void onSuccess(String value) {
            // TODO Auto-generated method stub
            System.out.println("value at specified position is:-
                "+value);
        }
    });
}
```

```
|     });
}
```

Here, we have used `MaybeObserver` as it's a possibility that the index number specified by us may not be available in the sequence emitted. If an index specified by us is available, the item value from the position will be returned on invocation of the `onSuccess()` method, otherwise the `onComplete()` method will be invoked and successful termination will occur.

On execution we will get the count value as `1`, denoting only a single item was emitted, with the name `mango` as index number always starts with `0`, as shown by the following marble diagram:



If instead of `3` we search for index `10`, we will get, successfully completed on the console denoting the absence of the index specified.

Using the filter() operator

From the emitted items the `filter()` operator filters only those items that matched the specified criteria to be available to the observer. The criteria for filtering the items can be specified by a predicate.

Let's use the list of fruits to filter the fruits that contain `le` as a character sequence. The complete code to apply the `filter()` is as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    String[] fruits = { "mango", "pineapple", "apple", "mango",
        "papaya" };

    Observable.fromArray(fruits).filter(item ->
        item.contains("le")).count()
        .subscribe(item -> System.out.println("we got: " +
            item + " items from the Observable"));

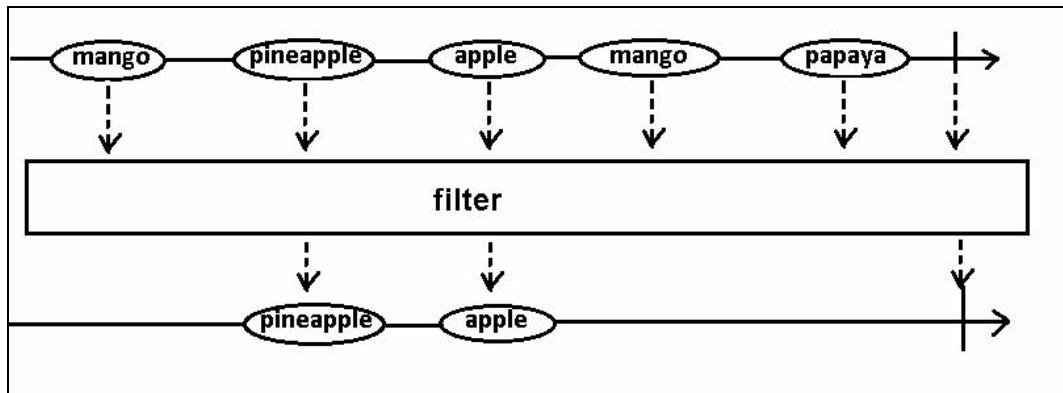
    Observable.fromArray(fruits).filter(item ->
        item.contains("le"))
        .subscribe(new Observer<String>() {

    @Override
    public void onComplete() {
        // TODO Auto-generated method stub
        System.out.println("filtering completed");
    }
    @Override
    public void onError(Throwable throwable) {
        // TODO Auto-generated method stub
        throwable.printStackTrace();
    }

    @Override
    public void onNext(String value) {
        // TODO Auto-generated method stub
        System.out.println(value);
    }
    @Override
    public void onSubscribe(Disposable disposable) {
        // TODO Auto-generated method stub
    }
});
}
```

When we execute the code we will get the number of items as `2` and their names as `pineapple` and `apple` on the console. The predicate accepted by the

`filter()` operator will filter on the criteria and only those items will be emitted, as explained by the following marble diagram:



Our observable is emitting five items, each one containing the names of the fruits. We have written a predicate to filter only those names that contain `le` as a character sequence. Only two items match the condition and hence the rest are all discarded and not emitted.

Using the `first()`, `never()`, and `empty()` operators

We already discussed how to emit items from an available sequence depending upon the index number. It is also possible to make only the first emitted from `Observable` available using the `first()` operator. Also, we can use some criteria to be applied on the emitted items and then choose the first item as an emission. We also have the `never()` operator, which never emits any item, and the `empty()` operator.

Let's take into consideration the list of fruits to create an observable and then apply the `first()`, `never()`, and `empty()` operators on it as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    String[] fruits = { "mango", "pineapple", "apple", "mango",
        "papaya" };

    Observable.fromArray(fruits).first("hello").subscribe(new
        SingleObserver<String>() {

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }

        @Override
        public void onSubscribe(Disposable disposable) {
            // TODO Auto-generated method stub
        }

        @Override
        public void onSuccess(String value) {
            // TODO Auto-generated method stub
            System.out.println("we got: "+value +" items from the
                Observable");
        }
    });

    Observable.empty().first("hello")
        .subscribe(item -> System.out.println("we got: " + item
        + " items from the Observable"));

    Observable.never().first("hello").subscribe(new BiConsumer() {
```

```

@Override
public void accept(Object value, Object throwable) throws
Exception
{
    // TODO Auto-generated method stub
    System.out.println("value:-"+value);
}
});
}

```

We will get the following output on the console:

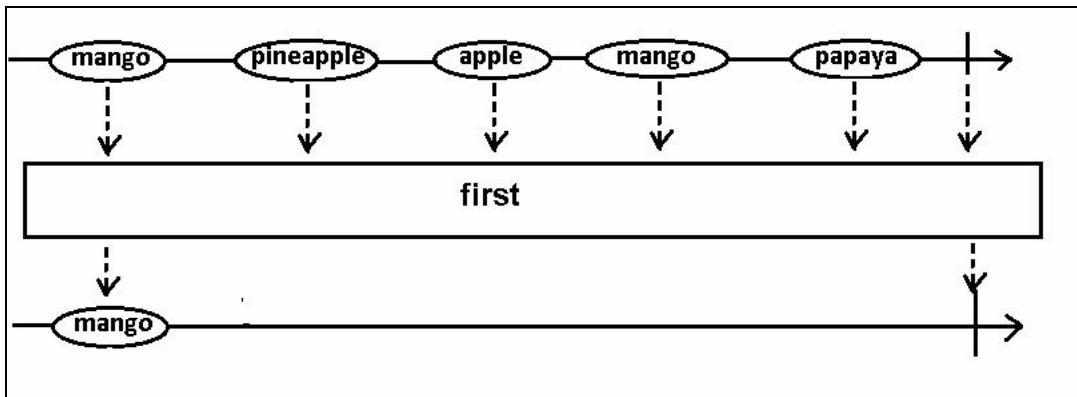
```

we got: mango items from the Observable
we got: hello items from the Observable

```

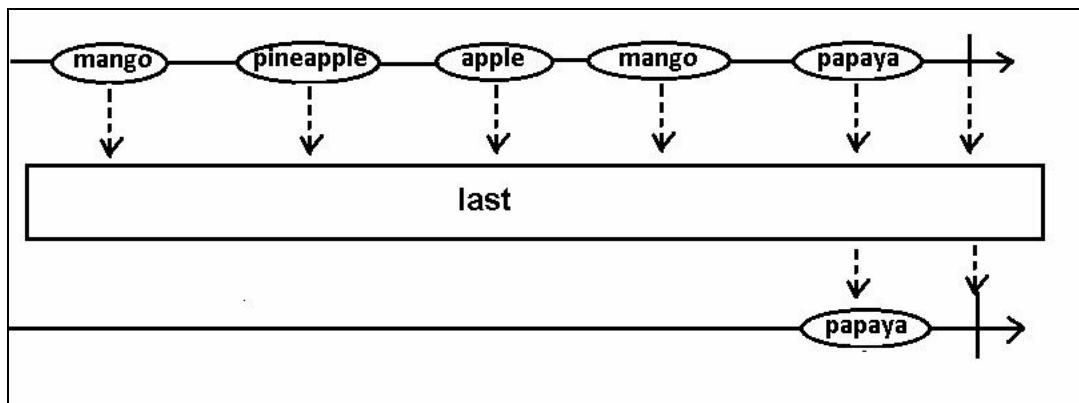
The output shows `mango` from the list is selected as it has `index=0`. The `empty()` operator will emit no elements but terminates normally. That means there are no elements and the default value specified by the `first()` operator has been emitted. In our case, the value is `hello`. The `never()` operator never emits the value and it never terminates. So no `onNext()` signal and no `onComplete()` signal will be emitted; hence it is not showing any output.

The marble diagram for the `first()` operator is as follows:



Using the last() operator

Just as we may be interested in observing only the first item, we may just be interested in the last emitted item by the observable. The `last()` operator emits only the last item depending on some criteria. The marble diagram is as follows:



Using the ignoreElements() operator

We are well aware that the observable emits a sequence of items one after another. Until now we have discussed how to filter the values as it is or depending upon some criteria. We also transform the values. However, sometimes we are not interested in the emitted values or their transformation. We are just interested in the terminating notification, which may be `onError` or `onComplete`. And `onNext` handlers will never be executed as discussed in the following code, which calculates the time taken for the process of subscription:

```
public static void main(String[] args) {
    String[] fruits = { "mango", "pineapple", "apple", "mango",
        "papaya" };

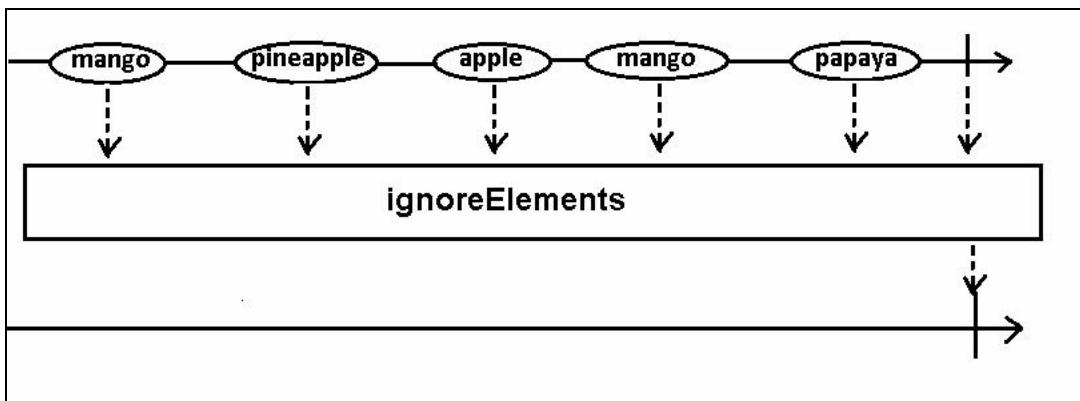
    Observable.fromArray(fruits).ignoreElements().subscribe(new
        CompletableObserver() {
        long time=0;

        @Override
        public void onSubscribe(Disposable disposable) {
            // TODO Auto-generated method stub
            time=System.currentTimeMillis();
            System.out.println(disposable.isDisposed()+
                "\t"+time);
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("completed");
            long time_to_complete=System.currentTimeMillis()-time;
            System.out.println("process completed in:
                "+time_to_complete+"ms");
        }
    });
}
```

On execution, we will get the time taken displayed on the console. Here we haven't dealt with items or their value; rather we are interested in when the termination of sequence takes place, as shown by the following marble

diagram:



Using the `sample()` operator

The `sample()` operator is useful when developers want to perform time-related operations along with the facility to deal with back pressure issues. The operator periodically looks at the observable and emits the most recent item emitted by it since the last data (sampling).

Using the skip() operator

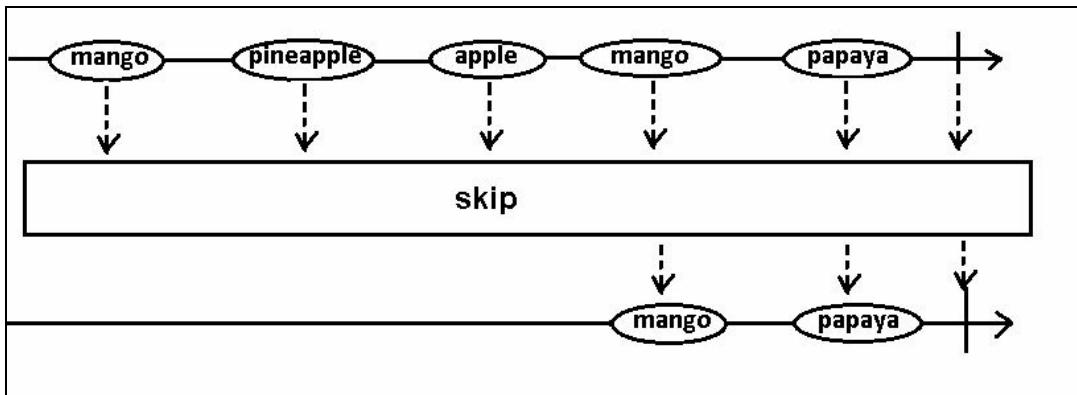
The `skip()` operator skips the first n elements emitted by the observable. All the items emitted after the specified number n will be emitted and available for emission.

Let's consider a list of fruits and skip the first three items emitted by the observable. All items after the third item will be emitted and subscribed by the `Subscriber` as in the following code:

```
public static void main(String[] args) {
    String[] fruits = { "mango", "pineapple", "apple", "mango",
        "papaya" };

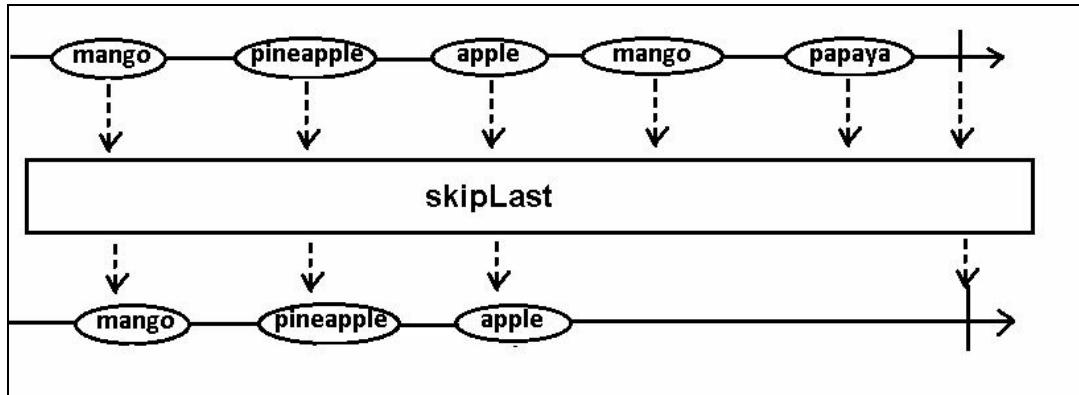
    Observable.fromArray(fruits).skip(3).subscribe(
        System.out::println);
}
```

The output will show the first three emitted items skipped and items after that will be emitted and subscribed by the `Subscriber`, as shown in the following marble diagram:



Using the `skipLast()` operator

Similar to the `skip()` operator, `skipLast()` also facilitates skipping items. The only difference is `skip()` allows skipping the first n items and `skipLast()` skips the last n items from `observable`, as shown by the following marble diagram:



Using the take() operator

The `take()` operator will emit the specified number of items from the observable and then completes without considering how many other items have been emitted. Let's use the `range()` operator to obtain an observable emitting items having values from 10 to 18 and the `take()` operator will filter and emit only the first three items , ignoring the rest. The code is as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Observable<Integer> observable = Observable.range(10, 9);
    observable.take(3).count().subscribe(item ->
        System.out.println("emitted " + item + " items"));

    observable.take(3).subscribe(new Observer<Integer>() {

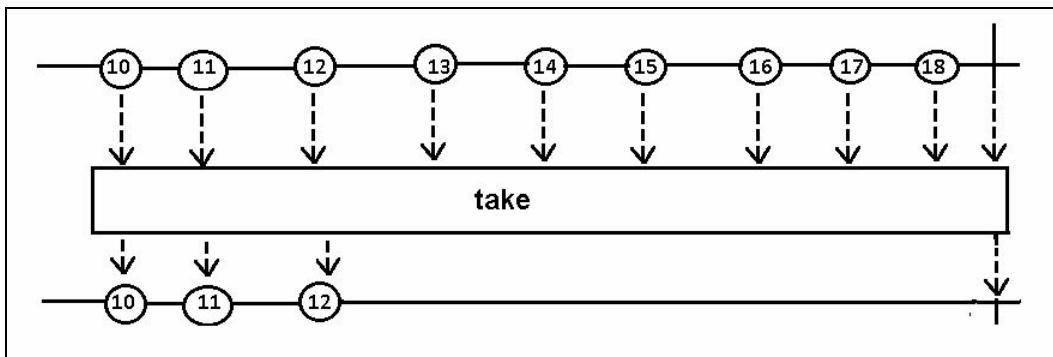
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("collected items successfully");
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }

        @Override
        public void onNext(Integer value) {
            // TODO Auto-generated method stub
            System.out.println("collected item: " + value);
        }

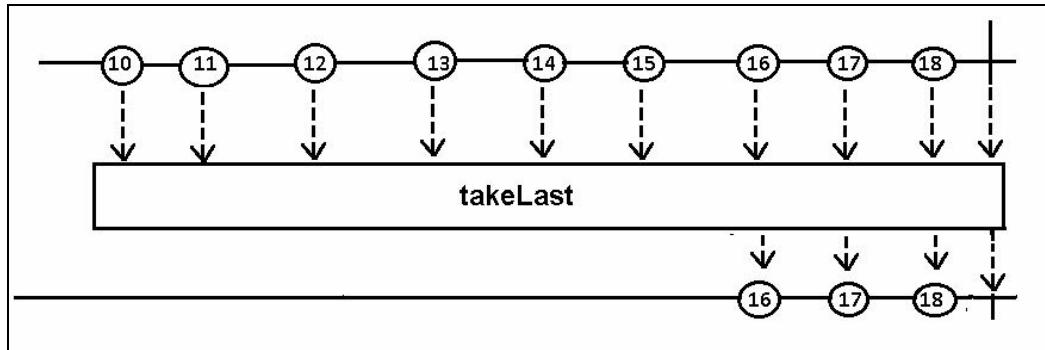
        @Override
        public void onSubscribe(Disposable disposable) {
            // TODO Auto-generated method stub
        }
    });
}
```

Here we will get only the three items emitted having values as 10, 11, 12, as shown by the following marble diagram:



Using the `takeLast()` operator

The `takeLast()` operator will emit the last n specified number of items without considering how many items `observable` have emitted before, as shown in the following diagram:



Combining Observables

Under this category, we will discuss operators that facilitate working with multiple `Observable` as sources to create a new `Observable` from the items emitted from sources.

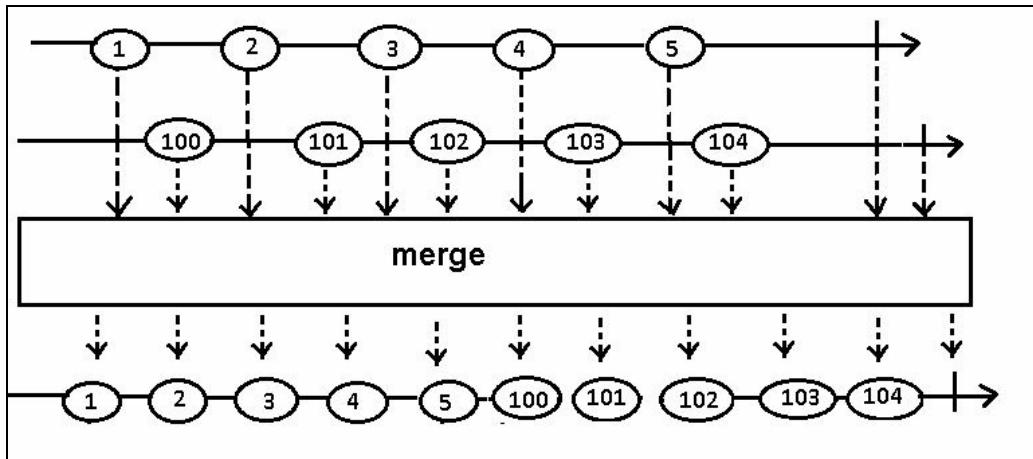
Using the merge() operator for combining observables

The `merge()` operator combines multiple `Observable` sources to merge the items emitted by them into a single resultant `Observable`. While the emission from multiple `Observable` is going on, any one of them may emit a `onError` signal; under such circumstances, the resultant `Observable` will also terminate.

Let's consider an `Observable` obtained by merging two `Observable` obtained from the `range()` operator, as shown by the following code:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Observable.merge(Observable.range(1, 5),
        Observable.range(100, 5)).subscribe(new Observer<Integer>()
    {
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("items merged successfully");
        }
        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }
        @Override
        public void onNext(Integer value) {
            // TODO Auto-generated method stub
            System.out.println("collected item: " + value);
        }
        @Override
        public void onSubscribe(Disposable disposable) {
            // TODO Auto-generated method stub
        }
    });
}
```

The newly created `Observable` will contain 10 items that have been obtained from two source `Observable`s, as explained by the following marble diagram:



We can even use the `take()` operator to specify how many items will be observed out of the emitted items by the `observer`, as shown by the following code:

```
Observable.merge(Observable.range(1, 5),
    Observable.range(100,5)).take(6).
    subscribe(System.out::println);
```

The obtained `observable` will contain six items having the first five emitted by the first `observable` and the sixth taken from the second source `observable`.

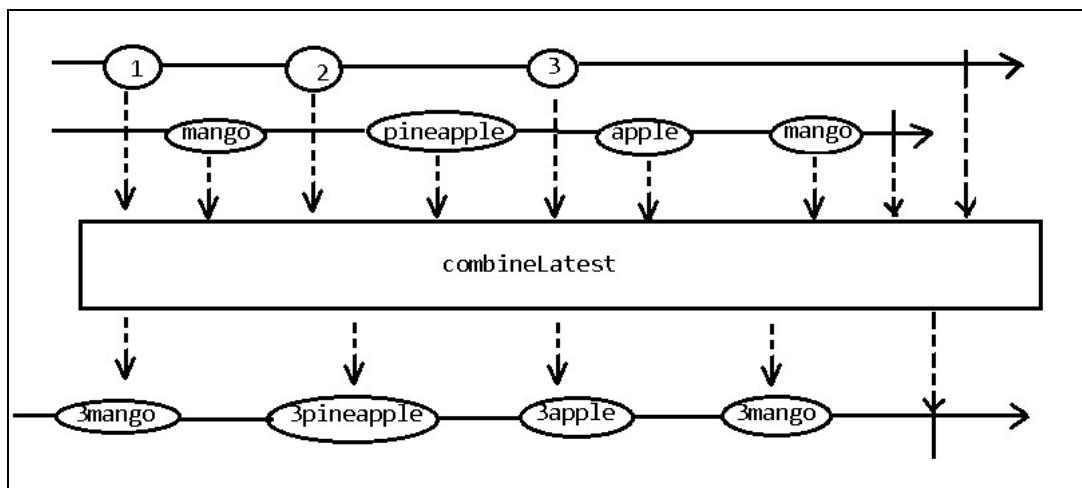
Using the combineLatest() operator

The `combineLatest()` operator combines the latest emitted items from the sources `observable` as per the specified function and the resultant item will be emitted.

Let's consider an `observable` obtained from the `range()` operator and the second `observable` will be obtained from an array of fruits as created earlier. Let's have a look at the code:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    String[] fruits = { "mango", "pineapple", "apple", "mango" };
    Observable.combineLatest(Observable.range(1, 3),
        Observable.fromArray(fruits), (item1, item2) -> item1 +
        item2)
    .subscribe(new Observer<String>() {
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("latest items combined
                successfully");
        }
        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }
        @Override
        public void onNext(String value) {
            // TODO Auto-generated method stub
            System.out.println(value);
        }
        @Override
        public void onSubscribe(Disposable arg0) {
            // TODO Auto-generated method stub
        }
    });
}
```

The output will be the latest element emitted by both observables, as shown in the following marble diagram:



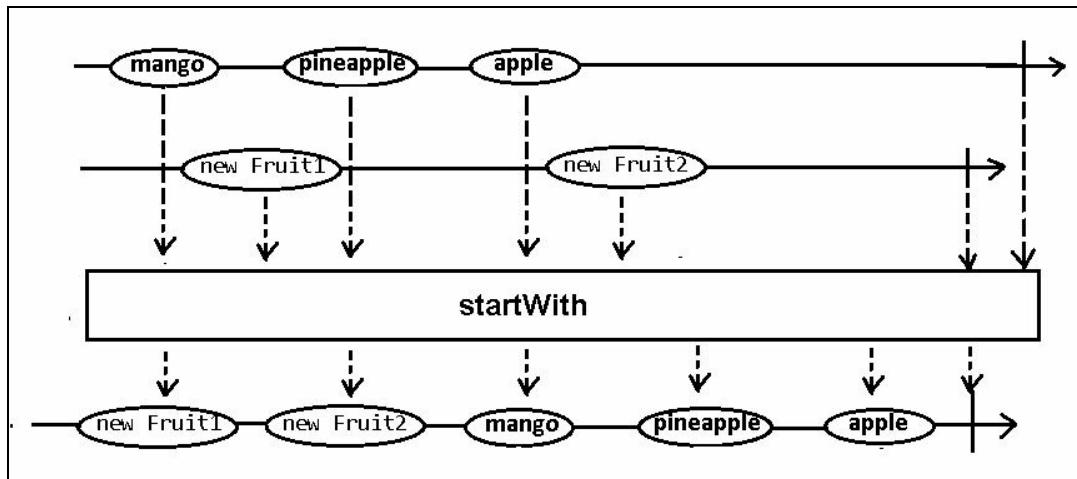
Using the `startWith()` operator

Whenever we want to concatenate a sequence of items with the source Observable we will use the `startWith()` operator.

Let's take two fruit arrays and combine them starting with the `newFruits` array, as shown in the following code:

```
public static void main(String[] args) {  
    String[] fruits = { "mango", "pineapple", "apple", "mango" };  
    String [] newFruits= {"newFruits","new Fruits2"};  
    Observable.fromArray(fruits).startWith(Arrays.asList(newFruits))  
        .subscribe(System.out::println);  
}
```

We will get the following output, represented by the following marble diagram:



Using the and(), then(), and when() operators

These operators will be used in combination to get an `observable` depending upon the intermediate data structure. The process of getting a new `observable` is a bit different from all other operators that we have discussed so far. Here `Pattern` and `Plan` play a very important role. First of all, items emitted by more than one `observable` will be combined together and they will be emitted one set at a time as an object of `Pattern`. Now the `then()` operator will transform these `Pattern` objects into `Plan` objects. Finally, the `when()` operator again transforms the obtained `Plan` objects into an emission from the `observable`.

Using the switchIfEmpty() operator

Sometimes the `observable` cannot emit any items and is an empty `observable`. However, we may want at least a single item to be emitted. The `switchIfEmpty()` operator checks whether the `observable` is empty or not and if it is empty, then it will return these specified items from the `observable`.

Let's create the `observable` using the `empty()` operator. As the `observable` is empty it will a emit value 10:

```
Observable source2 =  
    Observable.empty();source2.switchIfEmpty(Observable.just(10)).  
subscribe(new Consumer<Integer>() {  
    @Override  
    public void accept(Integer value) throws Exception {  
        // TODO Auto-generated method stub  
        System.out.println("value emitted:"+value);  
    }  
});
```

Instead of using an empty `observable` to any other `observable` that emits items, the item specified will be emitted.

Using the zip() operator

The `zip()` operator allows developers to combine sequential items emitted by multiple observables. Then, with the help of a function specified by the `zip()` operator, these will be combined together and then emitted as a single observable. It means from all the `Observable`, first items will be taken and on that, the function will be applied to get a single item as a result and then the resultant item will be emitted. The total number of items will be equal to the least number of items emitted by any `Observable` that we used as a source.

Let's consider a range `Observable` and two `Observable` obtained from an array, which will be zipped together as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Integer[] numbers = { 1,2,13,34,12,10};
    String[] fruits = { "mango", "pineapple", "apple", "mango",
    "papaya" };

    Observable<Integer> source1=Observable.fromArray(numbers);
    Observable<String> source2=Observable.fromArray(fruits);
    Observable<Integer> source3=Observable.range(30, 3);

    Observable.zip(source1,source2,source3,
        (value1,value2,value3)->{
            return value1+":"+value2+"*"+value3;
    }).subscribe(new Observer<String>() {

        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("zipping completed successfully");
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }

        @Override
        public void onNext(String value) {
            // TODO Auto-generated method stub
            System.out.println(value);
        }

        @Override
```

```

        public void onSubscribe(Disposable arg0) {
            // TODO Auto-generated method stub
        }
    });
}

```

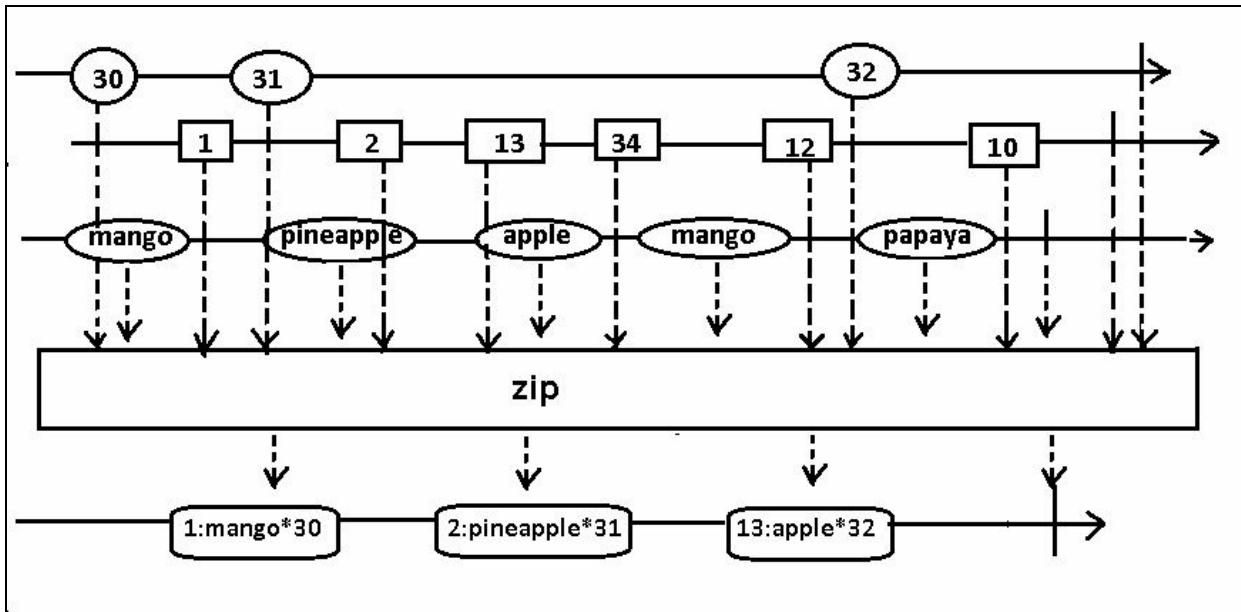
The `zip()` operator is accepting a function that has three arguments, out of which the first two are the items emitted by the observables and the third is the resultant. We have written the statement for zipping them as `value1+" "+value2+"*"+value3`. This value is returned from the function and thus subscribed by the `Subscriber`. On execution of the code we will get the following output:

```

1:mango*30
2:pineapple*31
13:apple*32
zipping completed successfully

```

The three observables we created are emitting 6, 5, and 3 items respectively. The final observable will emit only three items as it's the lowest number from the source and the value of these items will be the result of the expression used in the function, which we specified as follows:



Using the join() operator

The `join()` operator helps in combining the items emitted by a pair of `Observable` together. The two items to be combined will be based upon the duration or window that we have defined on a per-item basis. We will implement those windows whose life span starts with any one `Observable` emits the item. The window will be closed when either the `Observable` completes the emission or the window for the item with which it is associated closes. All the items emitted from the opening of the window until it gets closed down will be combined together.

Conditional operators

Conditional operators enable developers to evaluate the observable or items emitted by the observable as discussed for the following operators.

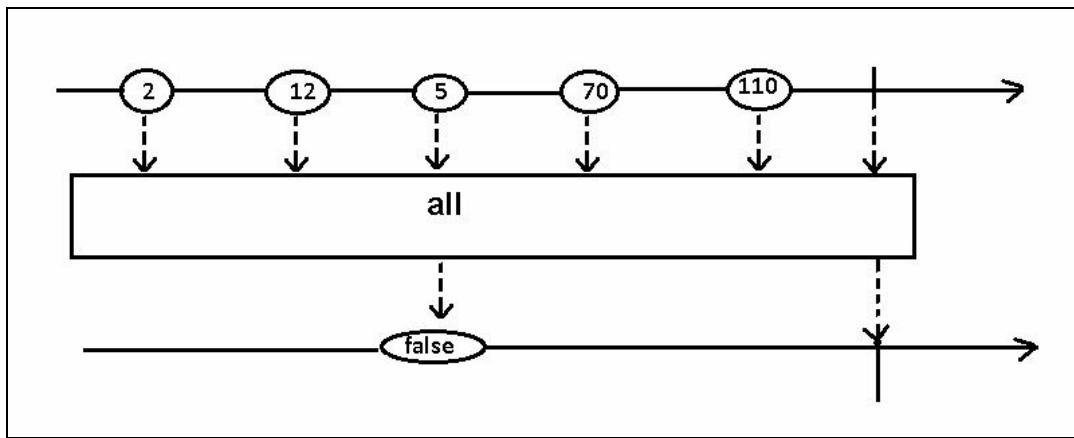
Using the all() operator

The `all()` operator helps in determining whether all the items emitted by the `Observable` meet the specified conditions or not. The predicate accepts a function as an argument for the `all()` operator, which accepts emitted items and returns a boolean value based on the evaluation. If all the emitted items pass the criteria, it will return `true`; otherwise, it will return `false`.

Let's consider an `Observable` that emits items to find whether all the emitted items are divisible by two or not as in the following code:

```
Single<Boolean> single = Observable.fromArray(
    Arrays.asList(new Integer[] { 2,
        12, 5, 70, 110 })).all( item -> {
    boolean flag=false;
    for(int i:item)
    {
        if(i%2!=0)
        {
            return false;
        }
    }
    return true;
});
single.subscribe(new Consumer<Boolean>() {
    @Override
    public void accept(Boolean passed) throws Exception {
        // TODO Auto-generated method stub
        System.out.println(passed);
    }
});
```

On execution it will return `false` as `5` is not divisible by `2`, as shown by the following marble diagram:



Using the `amb()` operator

The `amb()` operator accepts more than one `observable`, it won't combine them together. Out of these `observable`, the one that sends either items or notifications to the operator only those emissions will be passed through and all other items and notifications from any other `observable` will be ignored.

Using the contains() operator

The `contains()` operator enables developers to find out if the specified item is available within the emitted items or not. If the object specified as an argument is present in the emission it will return `true`. The `contains()` operator enables developers to find out if the specified item is available within the emitted items or not. If the object specified as an argument is present in the emission it will return `true`, otherwise, it will return `false` as shown by the following code:

```
observable.fromArray((Arrays.asList(
    new Integer[] { 2, 12, 5, 70, 110 })).
contains(10).subscribe(new Consumer<Boolean>()
{
    @Override
    public void accept(Boolean passed) throws Exception {
        System.out.println(passed);
    }
});
```

Here, in a sequence, the item having a value equal to `10` is not present so the item with a value equal to `false` will be emitted.

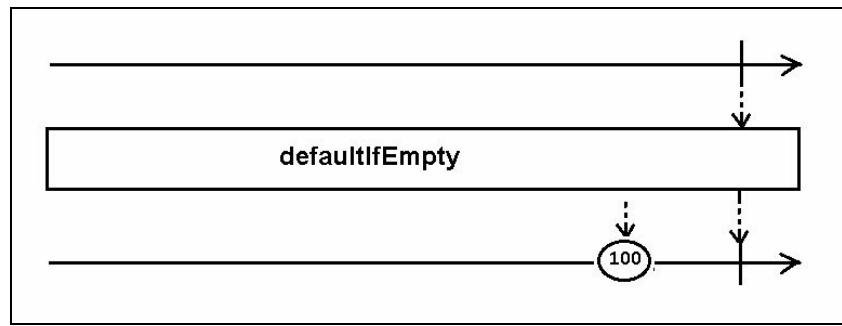
Using the `defaultIfEmpty()` operator

The `defaultIfEmpty()` operator is very special due to its nature. The operator emits all items without any transformation of the observable terminates normally. If it terminates normally without emitting any item then, instead of just terminating, the operator will emit the specified default value.

Let's directly create an empty `Observable` to which we will apply the `defaultIfEmpty()` operator as follows:

```
Observable.empty().defaultIfEmpty(100).  
    subscribe(new Observer() {  
  
        @Override  
        public void onComplete() {  
            // TODO Auto-generated method stub  
            System.out.println("the sequence got completed");  
        }  
  
        @Override  
        public void onError(Throwable throwable) {  
            // TODO Auto-generated method stub  
            throwable.printStackTrace();  
        }  
        @Override  
        public void onNext(Object value) {  
            // TODO Auto-generated method stub  
            System.out.println("emitted:-"+value);  
        }  
  
        @Override  
        public void onSubscribe(Disposable disposable) {  
            // TODO Auto-generated method stub  
            System.out.println(disposable.isDisposed());  
        }  
    });
```

As the `Observable` is empty it will terminate normally without emitting any item. However, as we are using the `defaultIfEmpty()` operator, it will emit item `100` as a default item as we specified it as an argument to the operator. The following marble diagram explains the emission:



Using the sequenceEqual() operator

When multiple `observable` emit a sequence of items, the `sequenceEqual()` operator facilitates developers to check whether two sequences are equal or not. The operator accepts two `Observable` and then compares items emitted by each of them and returns true if items from both `Observable` match.

Let's consider a couple of `Observable` and apply `sequenceEqual()` on them to compare them. We will use `SingleObserver` to subscribe the single item emitted after checking the sequence as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Observable names1 = Observable.fromArray(new String[] {
        "name1", "abc", "xyz", "lmn" });
    Observable names2 = Observable.fromArray(new String[] {
        "name1", "abc", "xyz", "lmn" });
    Observable.sequenceEqual(names1, names2).subscribe(new
        SingleObserver<Boolean>() {
    @Override
    public void onError(Throwable throwable) {
        // TODO Auto-generated method stub
        System.out.println(throwable.getMessage());
    }

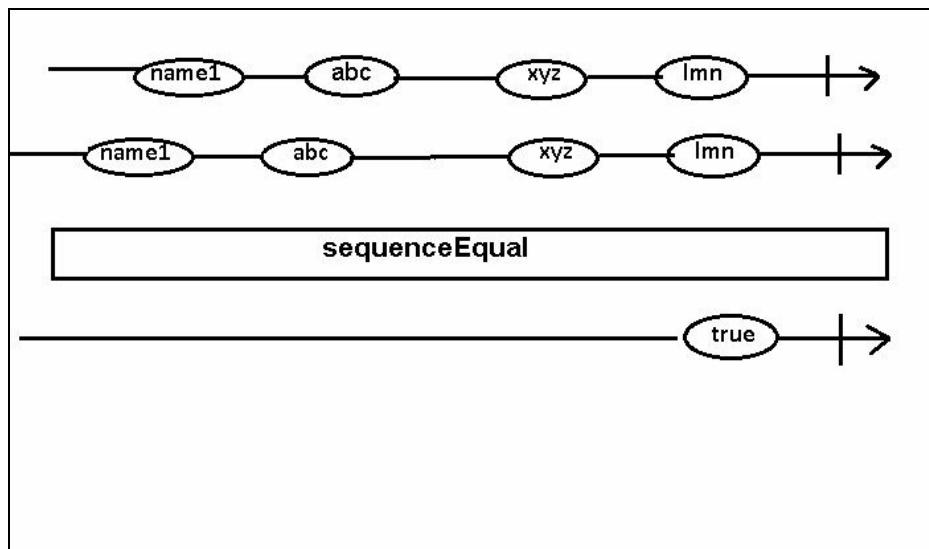
    @Override
    public void onSubscribe(Disposable disposable) {
        // TODO Auto-generated method stub
        System.out.println("is disposed:-" +
            disposable.isDisposed());
    }

    @Override
    public void onSuccess(Boolean value) {
        // TODO Auto-generated method stub
        if (value) {
            System.out.println("successfully finished
                comparison of two sequence and both sequences are equal");
        }
        else
            System.out.println("successfully finished comparison
                of two sequence and both sequences are NOT equal");
    }
});
}
```

Here, the `onSuccess()` method will get invoked once a successful comparison

is finished and will display whether the result is `true` or `false`. On execution of the code, we will get the output as `true`. In case we change the sequence of the items or the value of any one of them we will get the result as `false` and not `true`.

Observe the following marble diagram to understand this better:



Using the `takeWhile()` operator

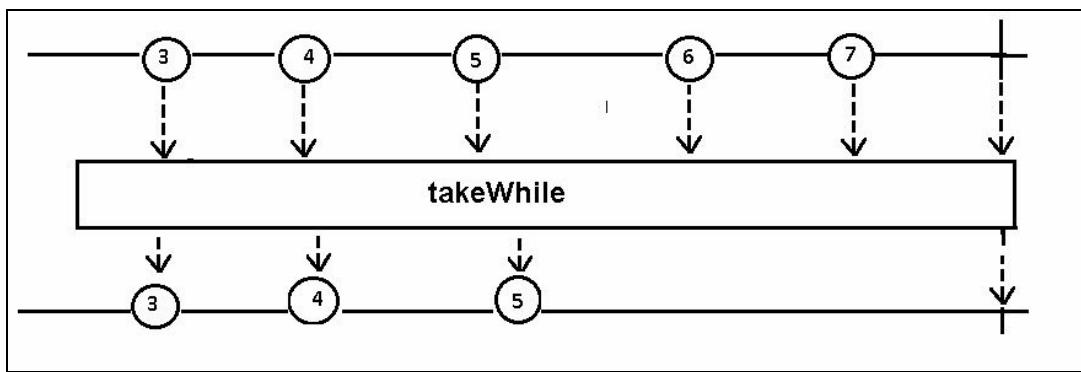
All the items emitted by the `observable` will be discarded if it's not going to match up the condition specified by the `takeWhile()` operator. Once the condition has failed, the operator stops mirroring the items and terminates its own `observable` as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Observable source1 = Observable.range(3, 5);
    source1.takeWhile(new Predicate<Integer>() {
        @Override
        public boolean test(Integer value) throws Exception {
            // TODO Auto-generated method stub
            if (value > 5)
                return false;
            return true;
        }
    }).subscribe(new Observer<Integer>() {
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("successfully completed the filtration");
        }

        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            System.out.println(throwable.getMessage());
        }
        @Override
        public void onNext(Integer value) {
            // TODO Auto-generated method stub
            System.out.println("after checking emitted:-" + value);
        }

        @Override
        public void onSubscribe(Disposable disposable) {
            // TODO Auto-generated method stub
            System.out.println(disposable.isDisposed());
        }
    });
}
```

On execution of the code, we will get 3, 4, and 5 as the output items. Once the emitted item has a value greater than 5, the condition becomes `false` and the mirroring items stop. The following marble diagram explains it better:



The mathematical and aggregate operators

The mathematical and aggregate operators discussed next help to perform operations on the entire sequence of the items that have been emitted by the Observable.

Using the average() operator

The `average()` operator calculates the average of the items emitted by an `Observable` and then emits the value of the calculated average as its emission as follows:

```
public static void main(String[] args)
{
    MathFlowable.averageDouble(Flowable.range(1, 10)).subscribe(new
        FlowableSubscriber() {

            @Override
            public void onComplete() {
                // TODO Auto-generated method stub
                System.out.println("completed successfully");
            }

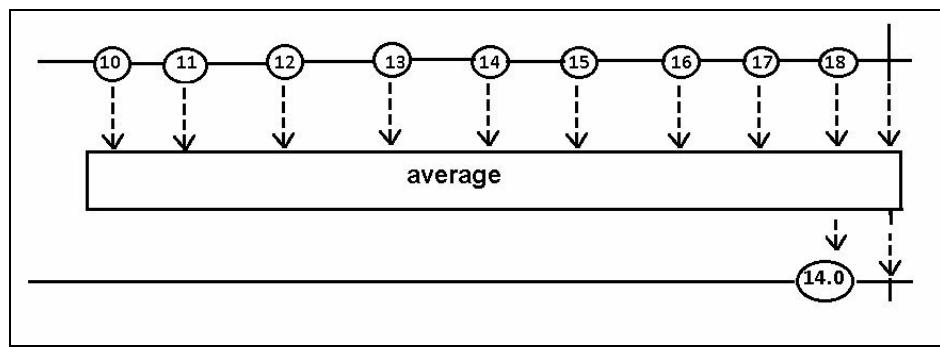
            @Override
            public void onError(Throwable arg0) {
                // TODO Auto-generated method stub
            }

            @Override
            public void onNext(Object value) {
                // TODO Auto-generated method stub
                System.out.println("average:-" + value);
            }

            @Override
            public void onSubscribe(Subscription subscription) {
                // TODO Auto-generated method stub
                subscription.request(1);
            }
        });
}
```

Here, we have used `MathFlowable`, which has the functionalities to work with `Flowable` sources to get the `sum`, `max`, and `average` of the items emitted

The marble diagram for `average()` is as follows:



Using the concat() operator

The `concat()` operator facilitates concatenating of one or more observables together into a single `observable` without interleaving them. The emission from the first source `observable` is followed by the emission of the second source `observable`.

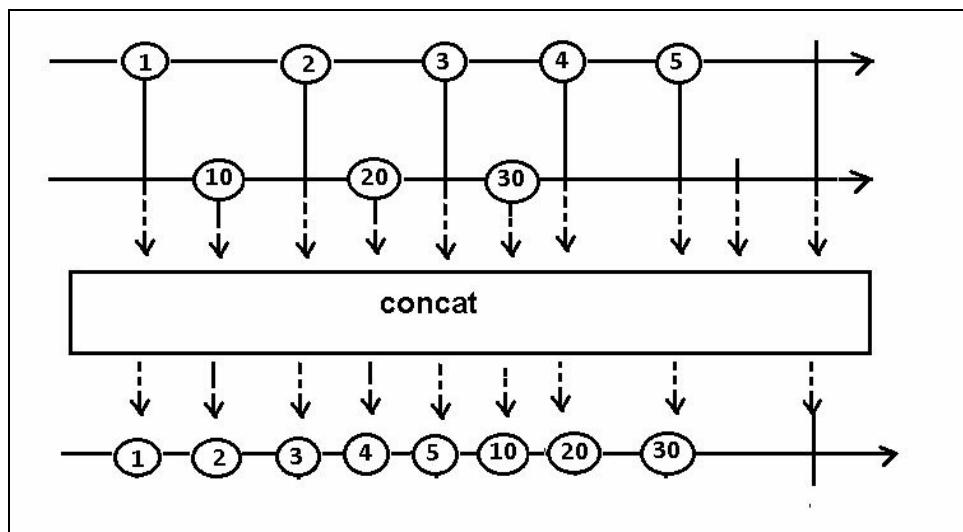
Let's concatenate two observables emitting integer items as follows:

```
public static void main(String[] args) {
    Observable source1= Observable.range(1,5);
    Observable source2=Observable.just(10,20,30);
    Observable.concat(source1,source2).subscribe
        (new Consumer<Integer>() {
        @Override
        public void accept(Integer value) throws Exception {
            // TODO Auto-generated method stub
            System.out.println(value);
        }
    });
}
```

The output of the code will be items emitted by the first `observable` followed by the items emitted by the second `observable` as follows:

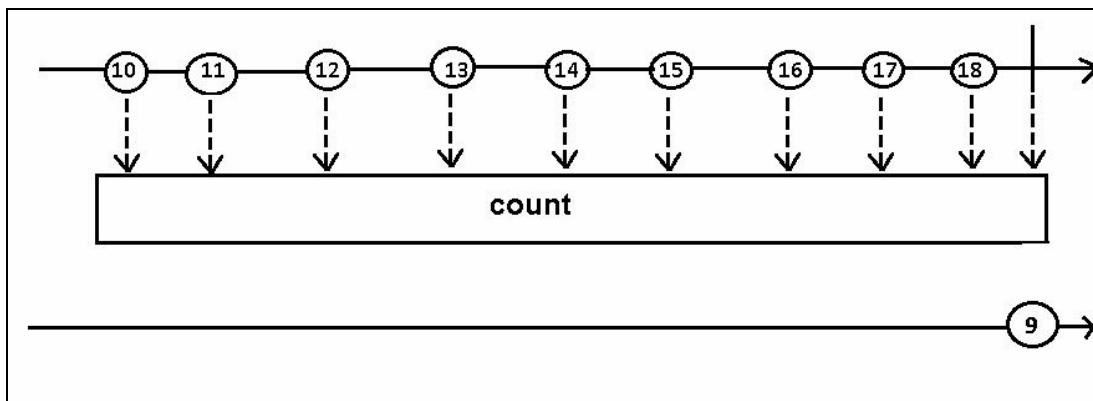
```
1
2
3
4
5
10
20
30
```

The output is shown in the following marble diagram:



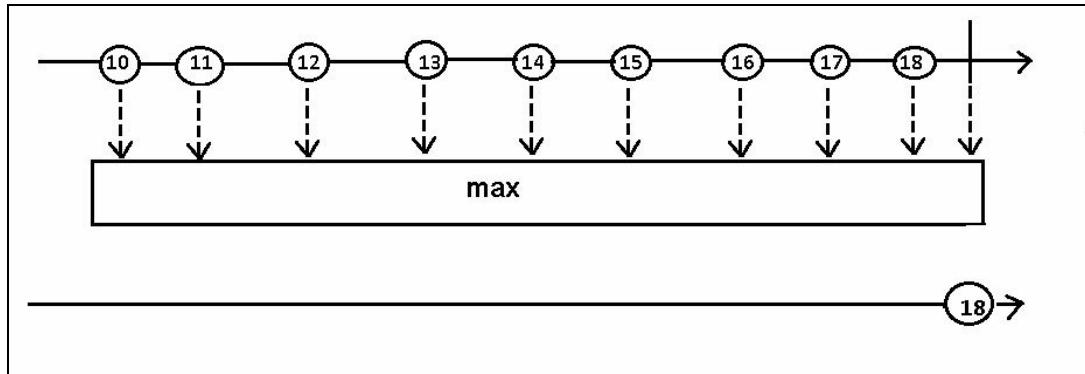
Using the count() operator

The `count()` operator emits a value that is equal to the number of items an observable has emitted. In case the `observable` emits a `onError` signal, the `count()` operator will pass this notification without emitting item. The following marble diagram displays this:



Using the `max()` operator

From the emitted sequence from an observable, the `max()` operator determines the maximum-valued item and then emits it. Consider the following marble diagram:



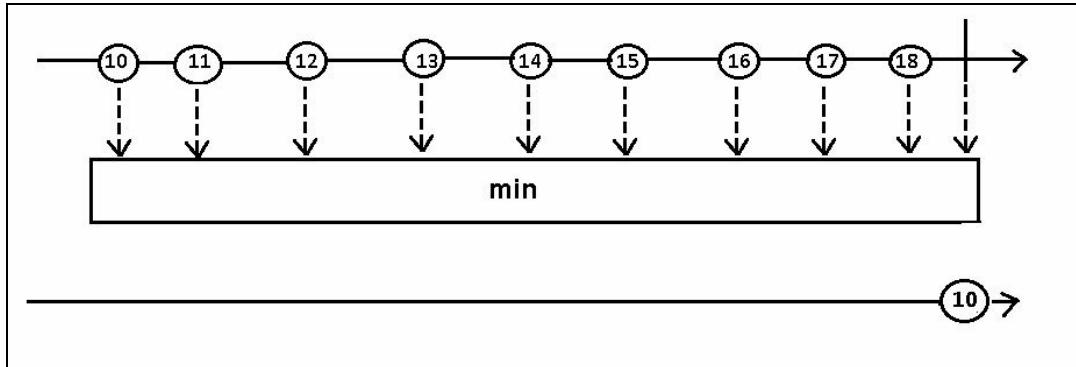
Using the `min()` operator

From the emitted sequence from an observable, the `min()` operator determines the minimum valued item and then emits it.

Let's write code to find out the smallest and greatest item from the emission:

```
public static void main(String[] args)
{
    MathFlowable.min(Flowable.range(1, 10))
        .subscribe(item -> System.out.println("minimum number
from the sequence is:- " + item));
    MathFlowable.max(Flowable.range(1, 10))
        .subscribe(item -> System.out.println("maximum number
from the sequence is:- " + item));
}
```

The marble diagram for the `min()` operator can be drawn as follows:



Using the reduce() operator

The `reduce()` operator applies a function to each emitted item from the `Observable` and to obtain a value, which will then emit its final value.

Let's create an `observable` that will emit an integer item and then by using the `reduce()` function calculates the sum of all items as follows:

```
public static void main(String[] args) {
    Integer[] numbers = { 1, 2, 13, 34, 12, 10 };
    Observable<Integer> source1 = Observable.fromArray(numbers);
    source1.reduce(new BiFunction<Integer, Integer, Integer>() {
        @Override
        public Integer apply(Integer value1, Integer value2)
            throws Exception {
            // TODO Auto-generated method stub
            // 1, 2, 13, 34, 12, 10
            int sum = 0;
            return value1 + value2;
        }
    }).subscribe(new MaybeObserver<Integer>() {
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("completed2");
        }
        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            System.out.println(throwable.getMessage());
        }
        @Override
        public void onSubscribe(Disposable arg0) {
            // TODO Auto-generated method stub
        }
        @Override
        public void onSuccess(Integer value) {
            // TODO Auto-generated method stub
            System.out.println(value);
        }
    });
}
```

The `apply()` function of the `BiFunction` interface will be invoked and the sum of each item emitted will be calculated. Once the emission is complete, the `onSuccess()` method of `MaybeObserver` will be invoked giving us the final emitted item. Finding the minimum number, maximum number, any `String` starting or

ending with values can be emitted.

Using the sum() operator

The `sum()` operator emits the value that is the calculated sum of all the items emitted by any observable. Let's calculate the sum of the items emitted by the `range()` operator as follows:

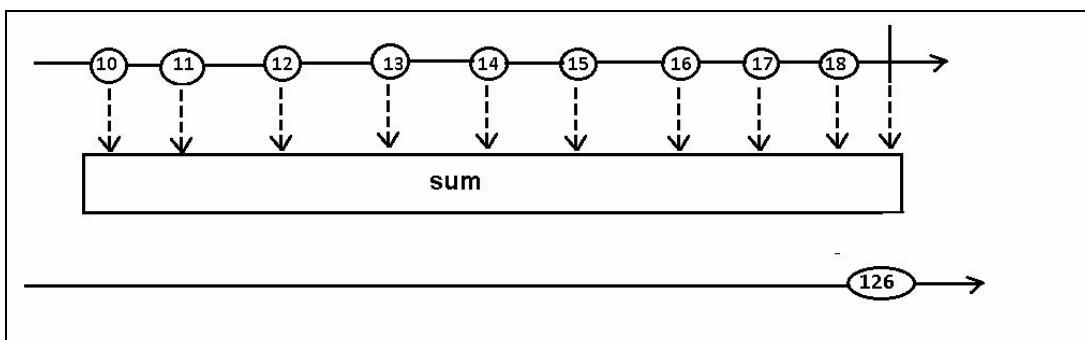
```
public static void main(String[] args)
{
    MathFlowable.sumInt(Flowable.range(1, 10)).subscribe(new
        Subscriber<Integer>() {

            @Override
            public void onComplete() {
                // TODO Auto-generated method stub
                System.out.println("completed successfully");
            }
            @Override
            public void onError(Throwable throwable) {
                // TODO Auto-generated method stub
                throwable.printStackTrace();
            }

            @Override
            public void onNext(Integer value) {
                // TODO Auto-generated method stub
                System.out.println(value);
            }

            @Override
            public void onSubscribe(Subscription subscription) {
                // TODO Auto-generated method stub
                subscription.request(1);
            }
        });
}
```

The following marble diagram describes the `sum()` operator:



Summary

In earlier chapters, we discussed `observable` in depth, how to obtain them using factory methods and how they work. However, we were not able to understand how to use, manipulate, and transform the emitted items. In this chapter, we discussed all these operations using various operators. We used `buffer()`, `flatMap()`, `map()`, `groupBy()`, `scan()`, and `window()` as transforming operators that help us to transform the emitted items; combining operators such as `combineLatest()`, `merge()`, `startsWith()` for combining various reactive flows; and the operators `debounce()`, `distinct()`, `elementAt()`, `filter()`, `first()`, `last()`, `skip()`, and `skipLast()` operators as filtering operators to filter the emitted items by the observable depending upon various conditions. Along with these we also discussed various conditional operators such as `all()`, `amb()`, `contains()`, and `sequenceEqual()`; operators such as `average()`, `concat()`, and `count()` for mathematical and aggregation operations on emitted items. We not only discussed these and the relevant code but also used marble diagrams to understand the workings of each of them in depth. Marble diagrams are really helpful to quickly understand the nature and workings of any operator.

Building Responsiveness

We have discussed many things, such as what reactive programming is, why we use reactive programming, how to implement Reactive Programming in Java applications, third-party libraries providing an implementation for reactive programming, such as RxJava, and the operators to perform tasks on the data. We cannot forget the core of Reactive Programming, which is all about asynchronous data in continuous motion from producer to subscriber. Though we are saying the data will flow asynchronously, to effectively handle all the items generated by the `observable` as a sequence, some level of concurrency control is a must be implemented. Developers need control, as the data that is going to be generated at the `observable` end and the data consumed at the `subscriber` end need to be concurrent.

In this chapter, we will consider various concurrency levels which will play an important role to handle the sequence of data generated. We understand handling concurrency is always an overhead; however, sometimes avoiding it is difficult. We will discuss when it's possible to avoid concurrency and when we need to use it with the help of the following points:

- Scheduler
- Scheduler implementations
- Multithreading operators
- Concurrency versus parallelism
- Reducing latency using the `flatMap()` operator

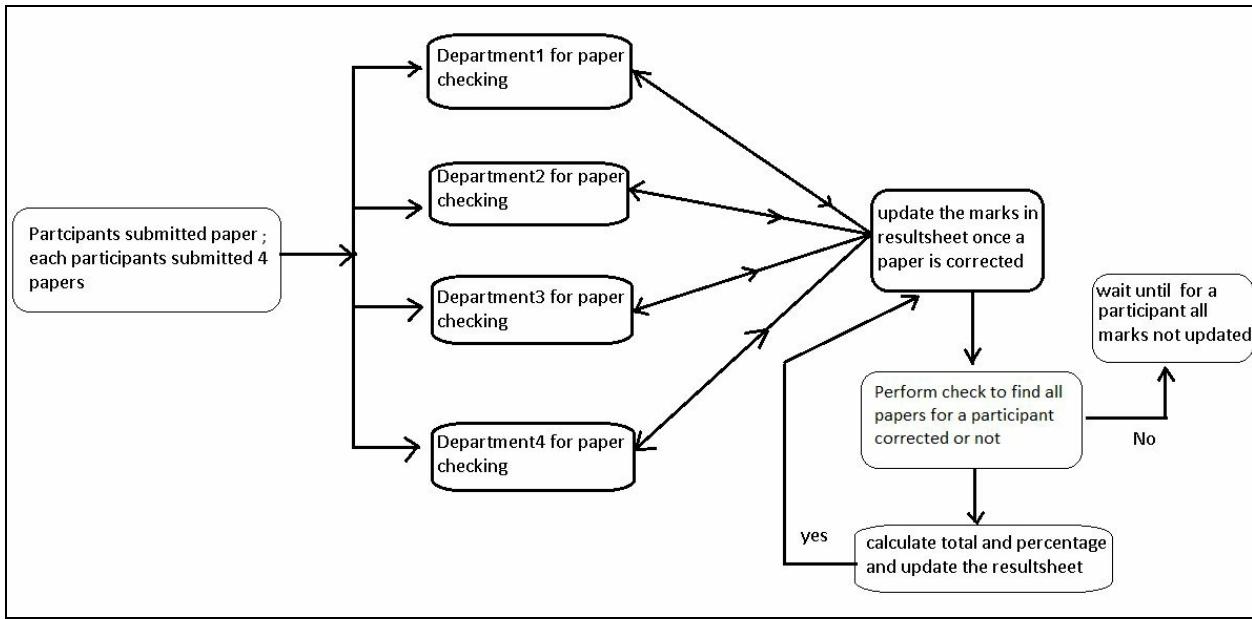
In the earlier chapters, we discussed how RxJava provides operators to handle the emission and its processing. We used `observable` to emit the data continuously and then used a few operators for the processing as well. We were happy as we got the emission of data and its subscription as well. But have we overlooked the point about keeping the user waiting for the response due to delay? Our main motive in earlier chapters was to understand how RxJava handles emission, transformation, and subscription. As we are stable in the framework, now we can discuss how to improve the performance. One very basic concept which will help to improve the performance is to divide

the work into subunits and complete them at the same time to avoid wasting time using threads, which is known as concurrency.

Concurrency

The term concurrency is used when multiple threads are used in the application. Concurrency is the ability to execute multiple programs or parts of programs in parallel. Take into consideration an operation which is time-consuming and involves many subtasks executed one by one. However, the subtasks involved in the process are not necessarily executed one by one. Sometimes, the order of execution of these functions is unimportant and they can be executed in any order.

Let's consider a task of generating a result sheet for any exam conducted for 100 participants and each participant will appear for 4 papers. The very first task has to correct the answer sheets, followed by entering the marks in the result sheet. Get marks for all the papers, and then apply the formula to calculate the total followed by a percentage, and then again, add these entries in the final sheet. Now, we get the result. Can we alter the process? Will it be possible to perform the tasks in parallel? Actually, no! Unless someone is not having the marks after correction, the process can't move ahead. Instead of a single person correcting the papers, what if we have a team of people to correct the papers? After each person has finished their correction, instead of waiting for the others, we can enter the marks in the sheet. In the same way, as the correction is about to finish, the calculation can also be done. We can even do the total and percentage calculation at the same time for all the participants. By dividing the work into groups and then performing it at the same time, we will be able to finish the task sooner, when compared to the situation when we will do the tasks one after another. The following diagram illustrates using multiple teams, showing multiple departments carrying out corrections and then communicating to and fro, updating the marks and later doing the calculation:



We have already discussed concurrency in Chapter 1, Introduction to Reactive Programming. If you need to, you can refer to the Asynchronous programming section for more information.

From the discussion now, it's clear that to achieve better performance and to release the allocated resources, performing multiple tasks at the same time is always a better choice. It means concurrency increases the performance of the application. Now, let's consider our RxJava application, which consists of an `Observable` emitting the data and an `Observer` which consumes the data under normal conditions. What sort of data will the `Observable` emit? The data contains values on which operations can be performed at the `Observer` end; apart from the data, the `Observable` will emit a signal for an error if something goes wrong or a signal when it completes the emission normally. Does RxJava do this concurrently?

The contract of the `Observable` says the `Observable` must issue the notification about data, error, or completion to the `Observer` serially. It means we can have multiple threads which can emit the data, but then won't it be a violation of the contract? Consider the following code using the `take()` operator:

```

public class Demo_take_no_threading {
    public static void main(String[] args) {
        Observable<Integer> observable = Observable.range(1, 50);
        observable.take(5).subscribe(new Observer<Integer>() {
    
```

```
    @Override
    public void onComplete() {
        // TODO Auto-generated method stub
        System.out.println(Thread.currentThread().getName() + " "
            finished reading of items");
    }

    @Override
    public void onError(Throwable throwable) {
        // TODO Auto-generated method stub
        System.out.println(Thread.currentThread().getName() + " "
            finished with exception");
    }

    @Override
    public void onNext(Integer value) {
        // TODO Auto-generated method stub
        System.out.println(Thread.currentThread().getName() + " "
            read item:-" +value));
    }

    @Override
    public void onSubscribe(Disposable arg0) {
    }
}

});
```

On execution, we will get only a single thread in action, which is the main thread! Surprised? Don't be. By default, it's a single thread application. It means we are not having concurrency and so the asynchronicity is also missing. It's the same scenario for operators such as `map()`, `distinctUntilChanged()`, and so many other operators which do not use scheduling. The operators such as `window()` and `debounce()` use schedulers. We already discussed these operators but not in terms of asynchronicity. We may confuse the concurrency and asynchronous applications. Let's quickly compare them before moving ahead.

Comparing asynchronicity with concurrency

We know concurrency is all about running more than one task simultaneously. It means concurrency starts, runs, and completes more than one task in the same time period which is overlapping. Does concurrency execute the task at the same time? No! It deals with many things at once, but by switching between the tasks. As the time taken to complete the operation is so small, for the user these tasks complete seemingly at the same time. But, the fact is they are not completing at the same time. This happens as these tasks take advantage of the CPU slicing feature provided by the OS. CPU slicing allows running a part of the task and then going to waiting state, in turn giving another task a chance to execute. The OS chooses the tasks depending upon the priorities.

On the contrary, parallelism is performing the tasks at the same time. Parallelism can be achieved using multicore systems. Parallelism carries out multiple tasks parallel to each other at the same time and, as it is going on at different cores, there is no need to switch between the tasks. Parallelism is not about how many tasks exist, it is all about dividing the task into subtasks and then running these subtasks on different cores or processors at the same time, later on combining the result of each of them and presenting the final result. Now, in this scenario, RxJava is not using multithreading. What about other operators?

Before starting the discussion about the differences between parallelism and asynchronicity, we left the point about asynchronicity in RxJava in between. We used the term scheduler in with some operators. What are these schedulers? And what do they facilitate? Let's discuss them in depth, taking multiple operators into consideration.

Scheduling

We used RxJava for the creation of the `observable`, the transformation and filtering with the help of various operators, and `subscriber` and `observer` to consume the items emitted. We have written a number of programs; however, we haven't explicitly written any code where we have created a thread to write concurrent applications. The operators such as `buffer()` and `delay()` perform the concurrency to some extent internally, without us as developers getting involved in it. It's the best example of abstraction, where developers keep themselves out of the processing without having knowledge of how it is going to be, and are just happy to achieve the desired effect.

What we have done up to now with RxJava is the following:

1. We have created an `observable` from different operators.
2. We applied various operators to filter, transform, or manipulate the data obtained.
3. The emitted items after the operation were consumed using the `Subscriber` OR `Observer`.

To conclude, we can say we know how to create the source of an item (`observable`), and depending upon the requirements, how to use one or more operators for filtering the item obtained and then, finally, how to consume the item using `observer` or `consumer`. We can have different scenarios discussed as follows:

- **Creation and subscription of items:** The thread which executes the creation of `observable` using any predefined operators. The same thread executes the `subscribe()` method.
- **Computation of items after creation by Observable:** Irrespective of how many operators have been invoked to perform the operations on the emitted items, the computation of upcoming operators happens on the same thread where the previous operators have executed. If the developers are using single operators and using a chain of operators, the operator will be executed on the thread where the computation of the

`observable` has been carried out.

- **Computation followed by subscription of emitted items:** While carrying out single or chain manipulation, the subscription happens on the same thread on which the operator has been executed. If we are not performing any computation on the data using the operators, the subscription happens on the same thread in which the `observable` creation has carried out.

Reactive Programming is asynchronous and we just talk all about single threads. Does it mean we are developing applications which are blocking applications? Before concluding, let's write a code to check whether the stated scenarios are the same in practice or whether theory and practice differ.

We are not using threads and asynchronicity uses threads. Does that mean the application we have written is using multiple threads internally? And we are not aware of how many threads we have allocated? Now, it's time to find out in depth about threading in RxJava. Let's first find out whether the applications we have written were multithreaded or not. To prove this, let's create an observable and subscribe it; we have already done this in [Chapter 4, Reactive Types in RxJava](#). This time, we will study the example considering threads in action. Before starting, we already used the operator `take()` to find threads in action. But what if we have a chain of operators? The following code is the usual code for creating an `observable` and subscribing the items emitted, along with getting the name of the thread handling the execution:

```
public class Demo_Callable {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Observable.fromCallable(new Callable() {

            @Override
            public Object call() throws Exception {
                // TODO Auto-generated method stub
                System.out.println("thread is:-" +
                    Thread.currentThread().getName());
                Thread.sleep(1000);
                System.out.println("thread is:-" +
                    Thread.currentThread().getName());
                return 5;
            }
        }).filter(new Predicate<Integer>() {

            @Override
```

```

public boolean test(Integer value) throws Exception {
    // TODO Auto-generated method stub
    System.out.println("thread is:-" +
        Thread.currentThread().getName());
    return value > 10;
}
).defaultIfEmpty(1).subscribe(new Observer<Integer>() {
    @Override
    public void onComplete() {
        // TODO Auto-generated method stub
        System.out.println("sequence completed successfully");
    }

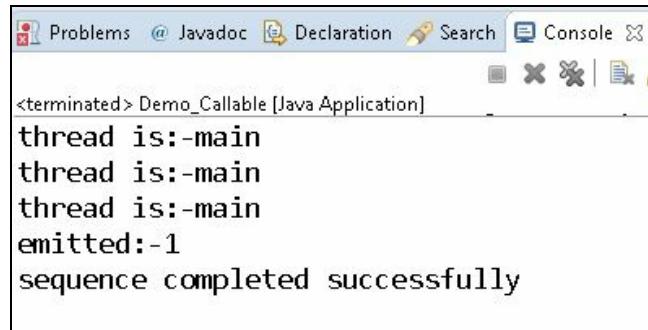
    @Override
    public void onError(Throwable throwable) {
        // TODO Auto-generated method stub
        System.out.println(throwable.getMessage());
    }

    @Override
    public void onNext(Integer item) {
        // TODO Auto-generated method stub
        System.out.println("emitted:-" + item);
    }

    @Override
    public void onSubscribe(Disposable disposable) {
        // TODO Auto-generated method stub
    }
});
}
}

```

The `fromCallable()` operator is used in this code to get an observable; then we apply the `filter()` operator to find whether the emitted item is greater than the value `10` or not. We extended the manipulation using the `defaultIfEmpty()` operator, which will return an item as `1` in case the test of filter fail. Finally, we have used the `Observer` to subscribe the emitted item. We purposely used `Thread.sleep()` to find out how many threads there are in action and, if there are any, how they are chosen by the JVM. Now, let's observe the following output:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_Callable [Java Application]
thread is:-main
thread is:-main
thread is:-main
emitted:-1
sequence completed successfully
```

The output clearly states that we don't have multiple threads; all the processing has been carried out in the single thread main.



By default, the execution of the RxJava program is blocking and not non-blocking.

Scheduler

A `Scheduler` object helps in scheduling units of work with or without delaying or periodically. A common instance of the `Scheduler` can be obtained from the `Schedulers` class. Some of the known `Schedulers` are `Schedulers.newThread()`, `Schedulers.computation()`, `Schedulers.from(Executor)`, and `Schedulers.io()`. We will discuss these `Schedulers` in depth in the upcoming pages. Now, let's concentrate on the `Scheduler`. To understand the `Scheduler`, we first need to understand `subscribeOn()` and `observeOn()` as two multithreaded operators.

The subscribeOn() operator

The `Observable` has a `subscribeOn()` operator which facilitates the observer to asynchronously resubscribe the items from an observable on the specified scheduler. If required, it also can route the requests from other threads to the same scheduler thread. The `Scheduler` needs to be provided by the developer; one easy way for `Scheduler` is to get it from `scheduler.newThread()` which instructs RxJava to perform the computation on a new thread every time. The name `subscribeOn` of the method suggests the computation on the `observable` item will be executed only when the `subscribe()` method is called on the `observable`, otherwise not.

Let's update the earlier code to invoke the `subscribeOn()` method as follows:

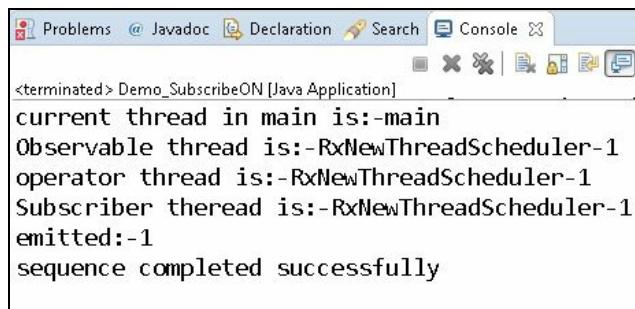
```
public class Demo_SubscribeON {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Observable.fromCallable(new Callable() {
            @Override
            public Object call() throws Exception {
                // TODO Auto-generated method stub
                System.out.println("Observable thread is:-" +
                    Thread.currentThread().getName());
                return 5;
            }
        }).filter(new Predicate<Integer>() {
            @Override
            public boolean test(Integer value) throws Exception {
                // TODO Auto-generated method stub
                System.out.println("operator thread is:-" +
                    Thread.currentThread().getName());
                return value > 10;
            }
        }).subscribeOn(Schedulers.newThread()).defaultIfEmpty(1).
        subscribe(new Observer<Integer>() {
            @Override
            public void onComplete() {
                // TODO Auto-generated method stub
                System.out.println("Subscriber thread is:-" +
                    "+Thread.currentThread().getName());
                System.out.println("sequence completed successfully");
            }
        }
    }
}
```

```
public void onError(Throwable throwable) {
    // TODO Auto-generated method stub
    System.out.println(throwable.getMessage());
}

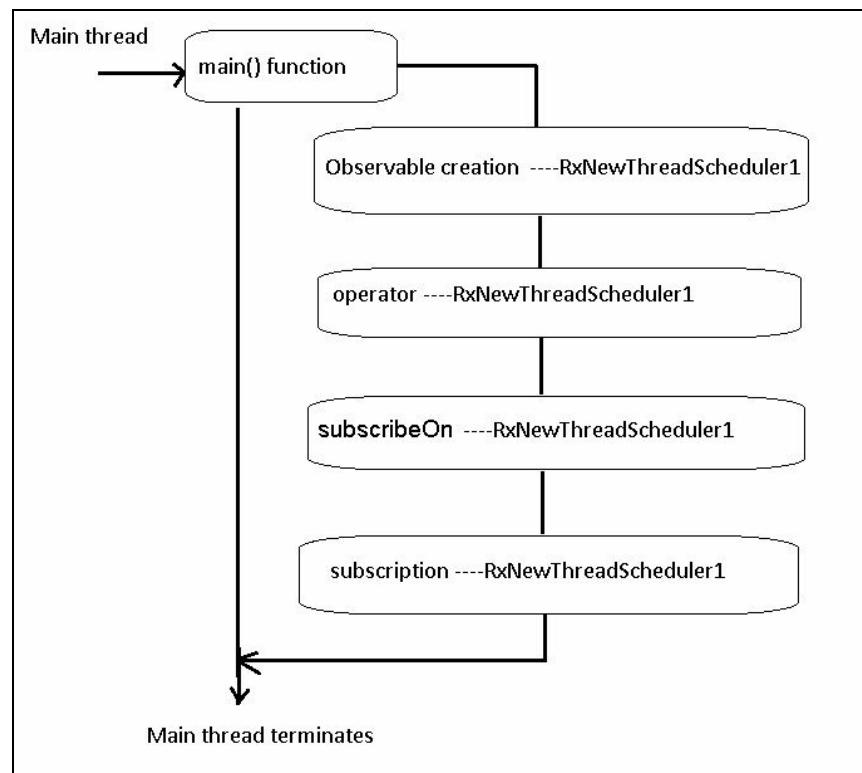
@Override
public void onNext(Integer item) {
    // TODO Auto-generated method stub
    System.out.println("emitted:-" + item);
}

@Override
public void onSubscribe(Disposable disposable) {
    // TODO Auto-generated method stub
}
});
try {
    Thread.sleep(100);
}
catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
```

Execute the code and observe the name of the threads being displayed on the console:



By default, the computation of the items emitted by the observable, the operation of manipulation and subscription, happens on the single thread. But our output is different from the default behavior explained by the following diagram:



The observeOn() operator

The method `observeOn()` instructs the RxJava library to perform the given computation by the operator or by the `Subscriber` declared after the given definition, on the thread which has been provided by the `scheduler`. The name `observeOn` suggests observing the items or events emitted by the `Observable`. If multiple operators are in chain then, after observing the data, one operator transmits it to the next operator. In the scenario that all the operators got completed and we have the `Subscriber`, then the items or events will eventually be passed on to the `Subscriber`.

Let's update the earlier code to use the `observeOn()` operator as shown in the following code:

```
public class Demo_ObserveOn {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Observable.fromCallable(new Callable() {

            @Override
            public Object call() throws Exception {
                // TODO Auto-generated method stub
                System.out.println("Observable thread is:-" +
                    Thread.currentThread().getName());
                return 5;
            }
        }).filter(new Predicate<Integer>() {

            @Override
            public boolean test(Integer value) throws Exception {
                // TODO Auto-generated method stub
                System.out.println("operator thread is:-" +
                    Thread.currentThread().getName());
                return value > 10;
            }
        }).defaultIfEmpty(1).observeOn(Schedulers.newThread()).
        subscribe(new Observer<Integer>() {

            @Override
            public void onComplete() {
                // TODO Auto-generated method stub
                System.out.println("sequence completed successfully");
            }

            @Override
```

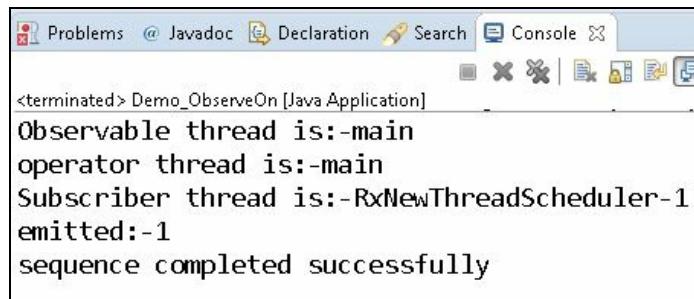
```
public void onError(Throwable throwable) {
    // TODO Auto-generated method stub
    System.out.println(throwable.getMessage());
}

@Override
public void onNext(Integer item) {
    // TODO Auto-generated method stub
    System.out.println("Subscriber thread is:-" +
        Thread.currentThread().getName());
    System.out.println("emitted:-" + item);
}

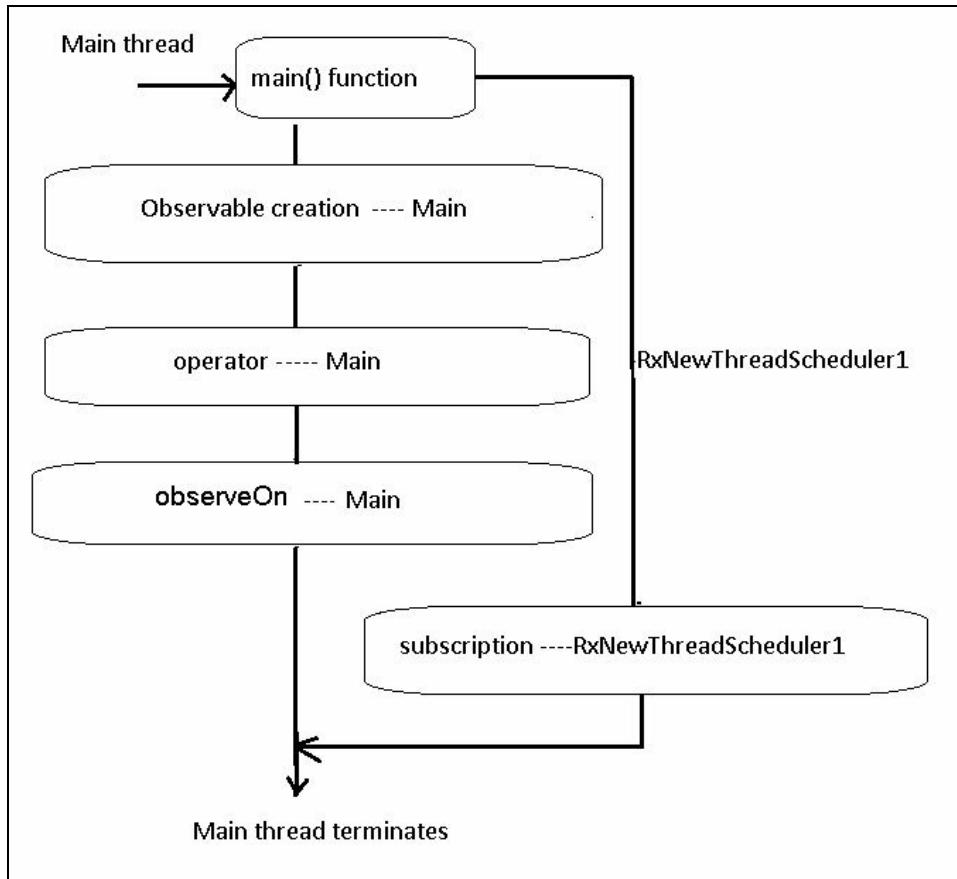
@Override
public void onSubscribe(Disposable disposable) {
    // TODO Auto-generated method stub
}
});

try {
    Thread.sleep(10);
}
catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
```

On execution, we will get the output as shown in the following screenshot:



Here, the subscription happens on a new thread while the computation happens on the same thread as that of the observable, as explained by the following diagram:



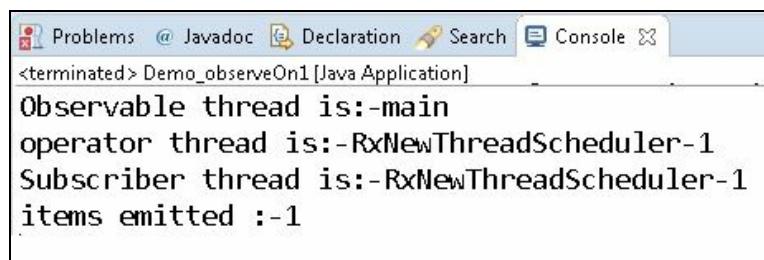
What will happen if we invoke the `observeOn()` method before the computation using operators? We will modify the code after the `observable` creation. The code can be written as follows:

```

public class Demo_observeOn1 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // new thread for operator
        Observable.fromCallable(new Callable() {
            @Override
            public Object call() throws Exception {
                // TODO Auto-generated method stub
                System.out.println("Observable thread is:-" +
                    Thread.currentThread().getName());
                return 5;
            }
        }).observeOn(Schedulers.newThread()).filter(new Predicate<Integer>()
        {
            @Override
            public boolean test(Integer value) throws Exception {
                // TODO Auto-generated method stub
                System.out.println("operator thread is:-" +
                    Thread.currentThread().getName());
                return value > 10;
            }
        })
    }
}
  
```

```
}).defaultIfEmpty(1).subscribe(item -> {
    System.out.println("Subscriber thread is:-"
        + Thread.currentThread().getName());
    System.out.println("items emitted :-" + item);
});
try {
    Thread.sleep(10);
}
catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
```

The output of the code is shown in the following screenshot:



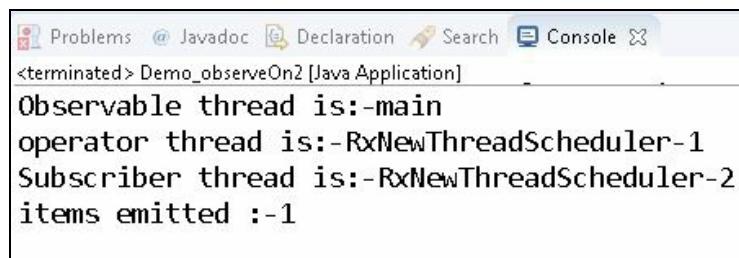
Here, the observable creation, the computation with the operator and the subscription happens on two different threads.

In the same way, we can ask for different threads for computation using operator and subscription, as shown by the following code:

```
public class Demo.observeOn2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // new thread for operator
        Observable.fromCallable(new Callable() {
            @Override
            public Object call() throws Exception {
                // TODO Auto-generated method stub
                System.out.println("Observable thread is:-" +
                    Thread.currentThread().getName());
                return 5;
            }
        }).observeOn(Schedulers.newThread()).filter(
            new Predicate<Integer>()
        {
            @Override
            public boolean test(Integer value) throws Exception {
                // TODO Auto-generated method stub
                System.out.println("operator thread is:-" +
                    Thread.currentThread().getName());
            }
        });
    }
}
```

```
        return value > 10;
    }
}).defaultIfEmpty(1).observeOn(Schedulers.newThread())
    .subscribe(item ->
{
    System.out.println("Subscriber thread is:-" +
        Thread.currentThread().getName());
    System.out.println("items emitted :-"+item);
});
try {
    Thread.sleep(10);
}
catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
```

The output will show us that we got two different threads, one for the operator for computation and second for the subscriber, as shown as follows:



Schedulers

We now are well aware that the `Scheduler` schedules work for us. The class which facilitates obtaining the `Scheduler` is `Schedulers`. It provides static factory methods, which return `Scheduler` instances. The available factory methods have been summarized in the following table:

Name of the method	Description of the method
<code>computation()</code>	The method returns a shared <code>Scheduler</code> which is deliberately used for computational work.
<code>from()</code>	The method wraps an <code>Executor</code> and returns a <code>Scheduler</code> instance and then it delegates <code>schedule()</code> calls to it.
<code>io()</code>	The method returns a default shared <code>Scheduler</code> instance deliberately used for I/O-bound work.
<code>newThread()</code>	The method returns a shared <code>Scheduler</code> which creates a new thread for each new unit of work.
<code>single()</code>	The method returns a <code>Scheduler</code> which is backed with a single thread which can be used for strongly sequential execution on a single background thread.
<code>trampoline()</code>	The method returns a shared <code>Scheduler</code> instance. The instance provides an instance of <code>Scheduler.Worker</code> which executes in FIFO manner.

Let's discuss these methods one by one in depth.

The computation() method

The `computation()` method is meant for computational work which is not meant for performing any blocking works such as I/O programming. It has the backing of a thread pool of single threaded `ScheduledExecutorService` instances. The pool consists of an equal number of processors which can be obtained using `Runtime.availableProcessors()` so it's possible to limit the number of threads in action equal to the number of cores available on the machine. In the situation of getting unhandled errors, they are delivered to the scheduler thread's `Thread.UncaughtExceptionHandler`.

Let's write a demo to understand how the method helps in achieving multithreading in the application. Now, we will consider a scenario as we will get the `Observable` using the `range()` operator, then, using the `map()` operator, we will map each emitted item to some value and then, before filtering, we will apply the `observeOn()` operator to get a new thread for the filtration. We will display the thread on each possible method to understand how many threads we have and which operation has been carried on which thread, as shown in the following code:

```
public class Demo_Scheduler_computation {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("current thread in main is:-" +
            Thread.currentThread().getName());
        Observable.range(10, 5).map(item -> item - 5).doOnNext(new
            Consumer<Integer>() {
                @Override
                public void accept(Integer value) throws Exception {
                    // TODO Auto-generated method stub
                    System.out.println(Thread.currentThread().getName() + " "
                        emitted " + value);
                }
            }).observeOn(Schedulers.computation()).filter(item -> {
                if (item%2 != 0)
                    return false;
                return true;
            }).subscribe(new Observer<Integer>() {
                @Override
                public void onComplete() {
                    // TODO Auto-generated method stub
                    System.out.println("sequence completed successfully");
                }
            });
    }
}
```

```
    }
    @Override
    public void onError(Throwable throwable) {
        // TODO Auto-generated method stub
        System.out.println(throwable.getMessage());
    }
    @Override
    public void onNext(Integer item) {
        // TODO Auto-generated method stub
        System.out.println("Subscriber thread is:-" +
            Thread.currentThread().getName());
        System.out.println("emitted:-" + item);
    }
    @Override
    public void onSubscribe(Disposable disposable) {
        // TODO Auto-generated method stub
    }
});
```

The observable emitted 5 items from 10 to 14; the emission of these items, by default, happened on the main thread. Then, before filtering the items which are divisible by two, we specify the `observeOn()` to switch the emission over to the computation thread. On execution of the application, the output displayed shows that we have a new thread in action, shown as follows:

```
Problems @ Javadoc Declaration Search Console <terminated> Demo_Scheduler_computation [Java Application]
current thread in main is:-main
main emitted 5 default main thread emitting items
main emitted 6
main emitted 7
main emitted 8
main emitted 9
Subscriber thread is:-RxComputationThreadPool-1 emission after filtration of items happened
emitted:-6 on another thread
Subscriber thread is:-RxComputationThreadPool-1
emitted:-8
sequence completed successfully
```

The output also denotes that once the emission is passed from the main thread to the newly obtained thread, the main thread is no longer concerned about it. In addition the newly obtained thread is solely responsible for the emission to the `Subscriber`.

Now, the question is, what if the main thread or previous thread creates the items faster than the thread which is involved in processing them? Yes, obviously using `observeOn()` has the chance of causing backpressure issues,

and we always need to keep them in mind. Under such scenarios, we can consider using the `subscribeOn()` operator for switching the emission to another thread. However, sometimes it also has been seen that the execution happens without any output. One of the reasons may be that the main thread may reach the end of the `main()` method, leading to the completion of its life cycle before the newly created thread gets any chance to come into action. Such scenarios will not occur in programs which have ongoing live sessions. To avoid such situations, we can use the `Thread.sleep()` method so that the newly created thread will get the chance to carry out the processing.

The from() operator

The `from()` operator accepts an instance of the `Executor` to get a new `Scheduler` instance, then this `Scheduler` instance will be delegated to the `schedule()` method in order to perform the scheduling of the tasks. The `Scheduler.Worker` helps in executing sequential actions on a single thread or on an event loop.

Let's use the `Executors` class to obtain a pool of threads, and then use it to get the `Scheduler` using the `from()` operator, as shown in the following code:

```
public class Demo_Schedulers_from {
    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(2);
        Observable.interval(100, TimeUnit.MILLISECONDS,
            Schedulers.from(executor))
            .take(10).forEach(item->
                System.out.println(Thread.currentThread().
                    getName()+" EMITTED "+item));

        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

In the code, we have used `Executors.newFixedThreadPool()` containing two thread instances. On it, we used the `take()` operator and consumed the emitted items using the `forEach()` operator. On execution, we will get the following output:



```
pool-1-thread-1 EMITTED 0
pool-1-thread-2 EMITTED 1
pool-1-thread-1 EMITTED 2
pool-1-thread-2 EMITTED 3
pool-1-thread-1 EMITTED 4
pool-1-thread-2 EMITTED 5
pool-1-thread-1 EMITTED 6
pool-1-thread-2 EMITTED 7
pool-1-thread-1 EMITTED 8
pool-1-thread-2 EMITTED 9
```

As we created a pool of two threads, the output is clearly showing we have two threads which are managing the emission task.

Update the code to use the `blockingSubscribe()` operator as follows:

```
Observable.interval(100, TimeUnit.MILLISECONDS,
    Schedulers.from(executor))
    .take(10).blockingSubscribe(item->
        System.out.println(Thread.currentThread().getName()+""
    EMITTED "+item));
```

On execution of the code, you will observe that it has subscribed to the `observable` and invoked the subscription methods on the same thread as it doesn't operate by default on a particular `Scheduler`, shown as follows:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_Schedulers_from [Java Application]
main EMITTED 0
main EMITTED 1
main EMITTED 2
main EMITTED 3
main EMITTED 4
main EMITTED 5
main EMITTED 6
main EMITTED 7
main EMITTED 8
main EMITTED 9
```

Th io() operator

The `io()` operator returns a shared `Scheduler`, which is recommended while performing I/O bound work and can also be used for asynchronously performing blocking I/O. The pool of single-threaded `ScheduledExecutorService` instances will be used to manage the threading. The `ScheduledExecutorService` instance will try to use already started and available threads. The `Scheduler.createWorker()` will return an instance of the worker thread; however, in the case of non-availability of the worker, it can obtain a new worker using `ScheduledExecutorService`. The real issue here is that the `Scheduler` may create unlimited worker threads, which in turn can result in `OutOfMemoryError`. To avoid the situation, the developers must dispose of `worker` instances using the `dispose()` method of `Disposable`.

The `io()` operators are useful in all the applications which are running for a long time while dealing with I/O, and which should not be executed on the main thread as in Android applications. Android applications deal with users and if the interaction is taking more time, such a task should not be executed on the main UI thread. If both tasks are running on the same thread, they may freeze up the application, ultimately making it unresponsive. Under such a situation, we can have the following code:

```
myObservable.subscribeOn(Schedulers.io()).  
    observeOn(AndroidSchedulers.mainThread()).subscribe(myObserver);
```

We can have code to use `io()` as shown:

```
public class Demo_Schedulers_IO {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Observable.create(new ObservableOnSubscribe<Integer>() {  
            @Override  
            public void subscribe(ObservableEmitter<Integer> emitter)  
                throws Exception  
            {  
                // TODO Auto-generated method stub  
                System.out.println("Thread:-"+Thread.currentThread().  
                    getName());  
                emitter.onNext(getNum());  
                emitter.onComplete();  
            }  
        }  
    }  
}
```

```
}).subscribeOn(Schedulers.io()).subscribe(new
    Consumer<Integer>()
{
    @Override
    public void accept(Integer value) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("Thread for subscription:-"
            "+Thread.currentThread().getName());
        System.out.println(value);
    }
});
try {
    Thread.sleep(2000);
}
catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
System.out.println("end of main:-"
    +Thread.currentThread().getName());
}
public static int getNum()
{
    System.out.println("Thread for reading data:-"
        "+Thread.currentThread().getName());
    Scanner scanner=new Scanner(System.in);
    System.out.println("enter a num");
    int num=scanner.nextInt();
    return num;
}
}
```

If we remove the statement `Thread.sleep()`, then, as the main thread is completed, the application will not terminate and the user will not get the chance to enter any data.

The `single()` operator

The `newThread()` operator returns a default, shared, single threaded `Scheduler` instance for work which needs to be sequentially executed on the same thread. It can be used for the execution of the loop containing main events, supporting `Schedulers.from(Executor)` and `from(ExecutorService)` while delaying the scheduling. All the unhandled errors will be delivered to the scheduler thread's `Thread.UncaughtExceptionHandler`.

The newThread() operator

The `newThread()` operator returns a default, shared `scheduler` instance which creates a new thread for every unit of task. The implementation of this `scheduler` creates a new, single threaded `ScheduledExecutorService` for each invocation of the methods, which binds a number of worker threads, which may result in an `OutOfMemoryError`. We have already seen the use of the `newThread()` operator when discussing multithreading in reactive applications.

The trampoline() operator

The `trampoline()` operator returns a default, shared an instance of the `Scheduler` which has a `Scheduler.Worker` instance queues the work and executes it in FIFO style. However, we cannot use it to reliably return the execution of the tasks to the current thread. The default implementation of the `Scheduler` executes the task on the current thread without any queuing, which adds time overload. The **trampoline** is a pattern in the functional programming. It facilitates the implementation of recursion without giving infinite calls for growing the stack. The idea behind it is to remove the current execution from the stack to eliminate the on-going stack. It manages to queue up the tasks one by one. The `Scheduler` starts with the first task, completes it, and then starts the queued task.

Let's discuss this using a `Scheduler` to schedule the work and an `Observable` to emit the items. We will use the `Scheduler` to schedule the work and to understand the way the `Scheduler` handles it, we will create one more schedule as shown by the following code:

```
public class Demo_trampoline {

    public static void main(String[] args) {
        Scheduler scheduler=Schedulers.trampoline();
        Scheduler.Worker worker= scheduler.createWorker();
        System.out.println("in begininng of the main method thread
            is:"+Thread.currentThread().getName());
        worker.schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println(" in outer run Thread:-"
                    +Thread.currentThread().getName());
                worker.schedule(new Runnable() {
                    @Override
                    public void run() {
                        Observable.just(10).subscribe(item->
                            System.out.println(" in inner run Thread:-
                                +Thread.currentThread().getName()+item));
                    }
                });
                Observable.just(100,200).subscribe(new
                    Observer<Integer>()
                {
                    @Override
                    public void onComplete() {
                
```

```
        System.out.println(" in complete Thread:-"
                           +Thread.currentThread().getName());
    }

    @Override
    public void onError(Throwable arg0) {
    }

    @Override
    public void onNext(Integer value) {
        // TODO Auto-generated method stub
        System.out.println("thread:-"
                           +Thread.currentThread().getName() + " got "+value);
    }

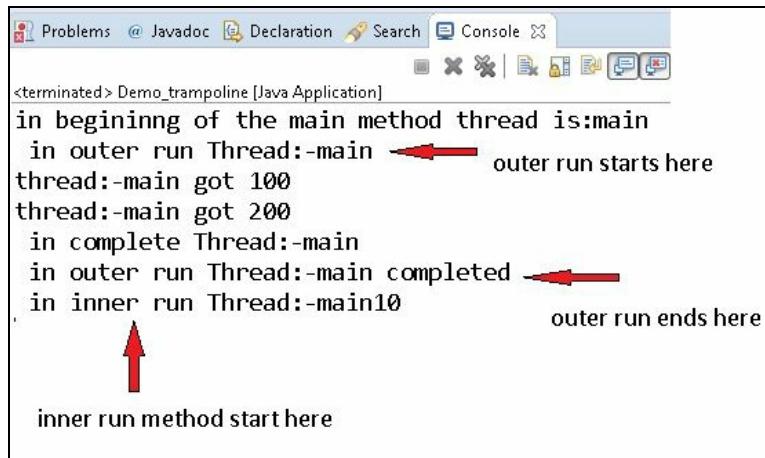
    @Override
    public void onSubscribe(Disposable arg0) {
        // TODO Auto-generated method stub
    }
};

System.out.println(" in outer run Thread:-"
                   +Thread.currentThread().getName()+" completed");
}

});

worker.dispose();
}
```

Two schedules are in action. The outer `schedule()` starts a new `schedule()` on the worker. Each `schedule()` is creating an observable. On execution, we will get the following output:



Observe the output. It is showing two tasks in the queue, one started by the first `schedule()` and another by the second `schedule()`. The outer scheduler emits two items with values `100` and `200` and the inner emits one item, `10`. From the queue, when the first schedule is completed, then the second

schedule will be started.

As we all know, Reactive Programming is all about asynchronously handling the continuous flow of data. When we talk about asynchronous programming, the first thing that pops up in mind is handling threads to perform the task. We have discussed taking threads in action to perform tasks using `Scheduler` and Schedulers in RxJava a lot. After having an intense discussion, can we say we have achieved parallelism using threads, to enhance the performance of the application without coming across the situation of blocking the calls? Does it mean we achieved parallelism and answered the all-time favorite question is multithreading and parallelism the same or different?

`Scheduler` helps in managing the schedules using multithreading. We are using many threads and an obvious question which may arise is if we are using multithreading then what about thread safety? Which operators are thread safe and which are not?

Operators and thread safety

Most of the operators provided by RxJava, such as `map()`, `filter()`, `take()`, and `distinctUntilChanged()`, are not thread safe. The list can continue further, but there is no point in adding more operators here. We are keen to understand if RxJava is all about synchronicity, then what about thread safety and why has it not been provided?

Operators are not thread safe

Most of the operators are not thread safe due to the following reasons:

- Adding thread safety will increase the complexity during implementation of the operators
- The operator becomes hard to maintain with thread safety
- When one or more operators combine, it will be hard to maintain the flow and how they can be combined

It doesn't mean we don't have any thread safe operators. We do! These operators are basically the operators which operate on multiple `Observables` to combine them, such as `merge()`, `zip()`, and `combineLatest()`. When the `Observable` emits the data concurrently, it is violating the `Observable` contract, which is not good. Under such scenarios, we can use the `serialize()` operator, helping to serialize the emission from an `Observable`.



The contract of an RxJava `Observable` states that the events such as `onNext()`, `onCompleted()`, and `onError()` can never be emitted concurrently and hence the single `Observable` must always be serialized and thread-safe.

The `serialize()` operator

Sometimes the methods of the `Observer` get invoked on the `Observable` asynchronously on different threads. Such an `Observable` is violating the `Observable` contract. And such an `Observable` may send `onComplete` OR `onError` before the `onNext` notification, or even an `onNext` notification from two different threads. Under such situations, the `Observable` needs to be synchronous and `serialize()` operator forces to emit synchronous emission.

Latency

Latency is the delay before the data transfer begins once the instruction for the transfer arrives. When the data travels over the network, getting the request by the subscriber and then sending the expected data back takes time, as the server needs to first find the data, then build the data, and later send it. This process is obviously going to take more time, and the time taken will be increased if the requested data is more. Thus, the application will take more time to respond.

Let's take a very simple example from our day-to-day life to understand latency easily. We all browse the internet. A few pages which we visit get loaded very quickly and a few take more time. Have you observed the general pattern where the time taken by the browser is more? These are the pages with lots of images and graphics.

Now, the question is, how will latency affect reactive applications? We have actually started using Reactive Programming to improve the performance and work around on the components, which takes more time to provide the data. Let's consider an observable which will be emitting the data which is subscribed by the subscriber, as shown in the following code:

```
public class DemoLatency {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        long time_start=System.currentTimeMillis();  
        Observable<Integer>observable=Observable.create(source->{  
            System.out.println("thread:"+  
                Thread.currentThread().getName());  
            source.onNext(10);  
            source.onNext(50);  
            source.onNext(100);  
            source.onComplete();  
        });  
        observable.map(item->item/5.0).subscribe(new  
            Observer<Double>() {  
  
                @Override  
                public void onComplete() {  
                    // TODO Auto-generated method stub  
                    System.out.println("time taken "+  
                }  
            }  
        );  
    }  
}
```

```
        (System.currentTimeMillis()-time_start);  
    }  
  
    @Override  
    public void onError(Throwable throwable) {  
        // TODO Auto-generated method stub  
        throwable.printStackTrace();  
    }  
  
    @Override  
    public void onNext(Double value) {  
        // TODO Auto-generated method stub  
        System.out.println("item arrived:-"+value+ "on  
        thread:-" +Thread.currentThread().getName());  
    }  
  
    @Override  
    public void onSubscribe(Disposable arg0) {  
    }  
});  
}  
}
```

On execution of the code, we will get the following output:

```
thread:-main
item arrived:-2.0on thread:-main
item arrived:-10.0on thread:-main
item arrived:-20.0on thread:-main
time taken122
```

Has something different happened this time? No, actually not. It's the same thing. Now we want to change the context under which we are going to observe this. If the operator `map()` is asynchronous, each item, `10`, `50`, and `100`, would need to be scheduled on a thread, and the transformation of each of the items would also happen on the same thread. The latency will be greater under such a scenario as, scheduling a thread, switching from the current thread to a new thread, then performing the task and then again switching between the intermediate threads will be more time-consuming.

Now, let's use the `flatMap()` operator, which facilitates transforming items emitted by an `Observable` into another `Observable` (you can refer to [Chapter 5, Operators](#) to learn more about `flatMap()`). Here, we will use it to understand latency shown as follows:

```
public class DemoLatency {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub
```

```
long time_start=System.currentTimeMillis();
Observable<Integer> observable = Observable.create(source -> {
    System.out.println("thread:-" +
        Thread.currentThread().getName());
    source.onNext(10);
    source.onNext(50);
    source.onNext(100);
    source.onComplete();
});
observable.map(item -> item / 5.0).subscribe(new
    Observer<Double>() {
        @Override
        public void onComplete() {
            // TODO Auto-generated method stub
            System.out.println("time taken"+
                (System.currentTimeMillis()-time_start));
        }
        @Override
        public void onError(Throwable throwable) {
            // TODO Auto-generated method stub
            throwable.printStackTrace();
        }
        @Override
        public void onNext(Double value) {
            // TODO Auto-generated method stub
            System.out.println("item arrived:-" + value + "onthread:-" +
                Thread.currentThread().getName());
        }
        @Override
        public void onSubscribe(Disposable arg0) {
            // TODO Auto-generated method stub
        }
    });
long time_start1=System.currentTimeMillis();
observable.flatMap(item -> Observable.just(item/5.0)).
subscribe(new Observer<Double>() {
    @Override
    public void onComplete() {
        // TODO Auto-generated method stub
        System.out.println("time taken"+
            (System.currentTimeMillis()-time_start1));
    }
    @Override
    public void onError(Throwable arg0) {
        // TODO Auto-generated method stub
    }
    @Override
    public void onNext(Double value) {
        // TODO Auto-generated method stub
        System.out.println("item arrived:-" + value + "on
            thread:-" + Thread.currentThread().getName());
    }
    @Override
    =public void onSubscribe(Disposable arg0) {
        // TODO Auto-generated method stub
    }
});
});
```

```
|  }}
```

Execute the code, which will give the following output:

```
thread:-main
item arrived:-2.0on thread:-main
item arrived:-10.0on thread:-main
item arrived:-20.0on thread:-main
time taken111
```

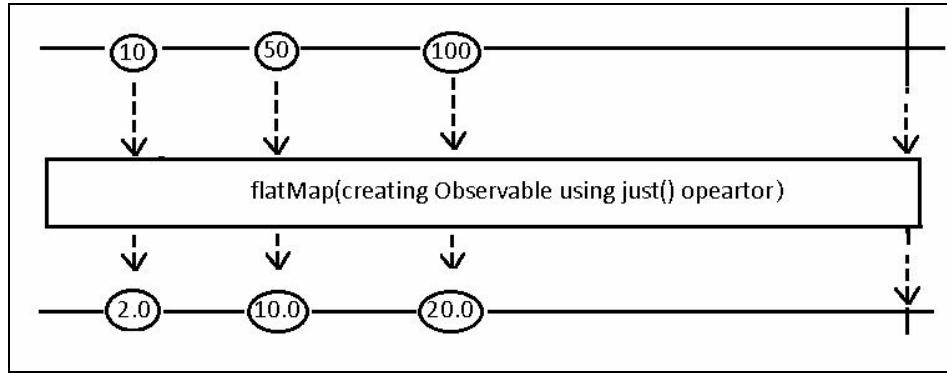
It's our usual output without anything weird. Now compare the time taken by the `map()` and `flatMap()` operators. Here, we have three elements emitted in less time compared to the `map()` operator.

A single `Observable` that is always synchronous tends to take more time, either for starting the dataflow or more time to complete the emission. However, we can have more than one `Observable` operating independently of each other, maybe concurrently or maybe in parallel. This is where we can use the `flatMap()` operator, which ends up composing the streams asynchronously.

It's one of the most important, easy-to-use, yet powerful operators in the RxJava library. Externally, or just because of the similarity in the name, we consider it as good as its `map()` operator; actually, these two are very different from each other and we already discussed them in the previous chapter, [Chapter 5, Operators](#). Let's once again briefly discuss them with regard to multithreading and latency.

The `map()` operator transforms the items emitted by the `Observable` using the specified function and will emit this transformed item having the same or different data type, while, in `flatMap()`, the specified function will take each item emitted by the source `Observable` and then transform it into another `Observable` instead of emitting items. It means we are performing asynchronous computation for each event received from the source `Observable`. Once the emitting `Observable` is complete, all the results from individual computation will be combined to produce the final output. Internally, after emission of the new `Observable`, the process doesn't stop; it continues ahead and subscribes the items emitted by all the newly created `Observables`. The `Subscription` of the emitted items from each `Observable` will be combined to produce a single value stream. It is a divide, perform, and then combine strategy. The process is happening in parallel, so obviously

taking less time. Also, out of all created observables, at least one started emitting items. The following diagram makes the concept much clearer:



As is clear from the diagram, the `Observable` keeps continuously emitting events, the operator transforms them, and concurrently subscribes as well. After the subscription, RxJava libraries merge them in one stream as well.

We have one more reason to use the `flatMap()` operator. As we are all well aware, the `flatMap()` operator can react to items. It can also react to events and notifications. It means the function which we will be writing to emit the `Observable` can be produced in the following scenarios:

- On receiving an item, the function specified in the argument of the `flatMap()` operator will produce an observable having the type specified in the return type of the function
- On receiving an error, the function specified in the argument of the `flatMap()` operator will produce an observable having the type as specified in the return type of the function.
- `flatMap()` operator will produce an observable having the type as specified in the return type of the function.
- On receiving the completion notification from the source `Observable`, the function specified in the argument of the `flatMap()` operator will produce an observable having the type specified in the return type of the function.

Summary

We used lots of operators to create, transform, and merge an `observable` in many ways, keeping in mind RxJava supports asynchronous Reactive Programming. We expected the operators provided by the RxJava library to be using multithreading internally. However, in the first half of this chapter, we proved the operators are not using multiple threads. They use a single thread to execute the complete process of the creation of an `observable` emitting the items, the processing of the emitted items, and then their subscription. In the second half, we discussed concurrency, how to achieve it, and the difference between concurrency and parallelism. We know creating threads and handling processes using the created threads adds overhead to the developers. We now need to schedule the process of emission, transformation, and subscription on different threads. RxJava provides a `Scheduler` object to help in scheduling a unit of work with, or without delaying or periodically. RxJava library provides `Schedulers` to schedule our tasks. Some of the known schedulers are `Schedulers.newThread()`, `Schedulers.computation()`, `Schedulers.from(Executor)` and `Schedulers.io()`, which provides `Scheduler` instances. We discussed them in detail along with implementation.

In the last part of the chapter, we discussed operators and thread safety, as well as latency, which is the delay before the data transfer begins once the instruction for the transfer arrives. Using the `flatMap()` operator, we found out how more emission can be done in minimum time. We also discussed `flatMap()` to improve the performance in order to decrease the initial time taken in the emission. We also discussed how `flatMap()` performs asynchronous computation for each event received from the source `Observable`.

In the next chapter, we will take a deeper dive into resiliency, where we will discuss various kinds of failures that can happen while building asynchronous flow.

Building Resiliency

Today's market demands for dynamic applications with high performance. Dynamic applications are in demand as they give the best user experience, in turn satisfying the market demand. We have many search engines available on the internet, dozens of websites for online shopping, and many companies maintain their own websites for public relations and business purposes. Thousands of Android applications are available in the market, providing services in unlimited areas. Users can use these available services to fulfill their requirements. The list continues, but what about the problems which occur while using these and many other such applications? Many times, we click on the link to access very important information and suddenly we get a page saying, No resource available or Service temporarily down. We get frustrated. However, the good thing is we get information about what went wrong and we are not in a state of confusion as to what is happening and whether we can access the page or not. In [Chapter 6, Building Responsiveness](#), we discussed how to improve the performance of the application to provide a quick response and to reduce the network latency. It was all about improving the performance. Responsiveness is a very important factor for improving the performance; in the same way, if something goes wrong while using the application, it must recover from it quickly.

This chapter takes a deeper dive into resiliency, where we will discuss various kinds of failures that can happen while building asynchronous flow. Our aim is to understand how RxJava approaches error handling and also show how to use the various error handling operators and utilities provided by RxJava, while covering the following points:

- Resilience
- Error handling in RxJava
- Error handling operators
- Error handling utilities

Resilience

The basic objectives of enterprise application development differ from application to application, while they basically aim to develop a solution to fulfill the enterprise's requirements. Whether it's an enterprise or common user, everyone wants the application to work smoothly. Users want the ability to do the things they need at a convenient time and enterprises want to serve market demands. It means we want the application to be reliable as well as resilient.

Reliable applications

Reliable applications are developed by keeping the problem statement in mind. The main aim is to provide solutions to enterprises. The software designers and developers design and develop an application keeping the problem in mind to get a planned outcome. There is always market demand for reliable applications.

Resiliency

Even if the designers have designed the best architecture and the developers have developed an application according to it, the application may fail at certain situations in runtime. **Resiliency** is the ability to recover from known failures and continue serving customers to get the output. It means, though any single functionality has failed, it doesn't cause other functions to fail. When functionality is restored, it will resume and start collaborating with other functionalities and components.

Our `observable` emits items and suddenly something goes wrong and an error occurs. Now, instead of sending the next item, an error signal will be sent, stopping the emission of items from the `observable` end. It ultimately affects the application's performance, as due to an error, the application stops responding. Applying resiliency enables developers to overcome the abrupt stopping of an application and continue with other functionalities, even when errors occur.



Resiliency is the ability of an application to recover from difficulties and continue serving.

Error handling in RxJava

In every application development, utmost care will be taken to design and develop it to sustain the application in all conditions and keep serving users. After application development, rigorous testing will be done to find all the possibilities where the application can fail and then the code is modified to overcome the problem. Still, we can't be 100% sure we have an application without errors, as they may occur. It means we need to take care to develop applications, and in the same way, we need to take care of handling errors.

Let's, first of all, find out what happens when an error occurs. We will create a simple observable using the `just()` operator and then we will filter the items to find out whether the item is less than five or not. If the condition matches, we will throw a `RuntimeException` using the `check(int item)` function, as shown in the following code:

```
public class Demo_Observable_RuntimeException {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Observable<Integer> observable =
            Observable.just(12, 30, 5, 50, 89);
        observable.filter(item -> { return check(item);
        }).subscribe(new Observer<Integer>() {

            @Override
            public void onComplete() {
                // TODO Auto-generated method stub
                System.out.println("sequence completed");
            }

            @Override
            public void onError(Throwable throwable) {
                // TODO Auto-generated method stub
                throwable.printStackTrace();
            }

            @Override
            public void onNext(Integer value) {
                // TODO Auto-generated method stub
                System.out.println("got:-" + value);
            }

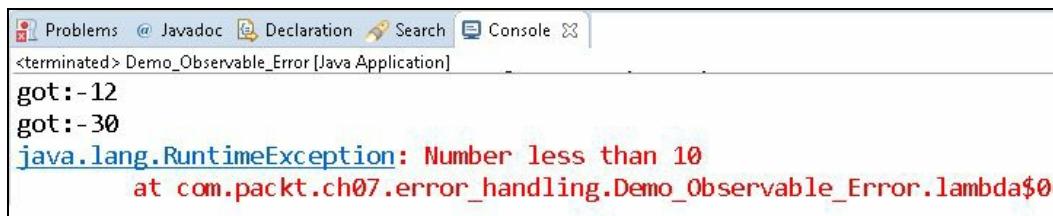
            @Override
            public void onSubscribe(Disposable disposable) {
```

```

        // TODO Auto-generated method stub
    });
}
public static boolean check(int item) throws RuntimeException
{
    boolean result = false;
    if (item > 10)
        result = true;
    else
        throw new RuntimeException("Number less than 10");
    return result;
}
}

```

On execution, we will get the following output:



```

Problems @ Javadoc Declaration Search Console
<terminated> Demo_Observable_Error [Java Application]
got:-12
got:-30
java.lang.RuntimeException: Number less than 10
    at com.packt.ch07.error_handling.Demo_Observable_Error.lambda$0

```

In the earlier example we used `RuntimeException`, but what will happen if we use checked exception instead? Let's try it out. The only code we will be changing is throwing checked exception instead of unchecked, keeping the rest as it is. The following code changes the `check()` function:

```

public static boolean check(int item) throws IOException {
    boolean result = false;
    if (item > 10)
        result = true;
    else
        throw new IOException("**** GOT AN EXCEPTION****");
    return result;
}

```

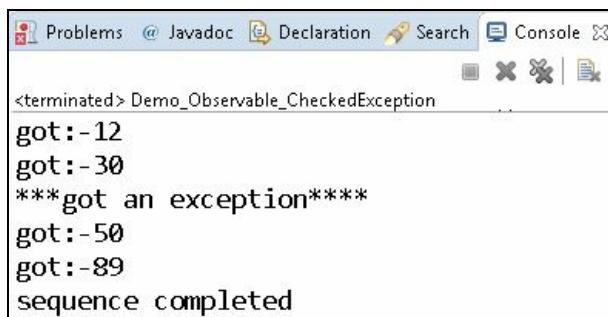
The function call remains the same as that of the previous code. Though the `check()` function this time is throwing a checked exception, we haven't got an error to handle it using the `try...catch` block. RxJava has its own system to handle checked errors. Keeping the code as it is, let's execute and find out what happens. The output is the same as that of your earlier output, after the third item when the condition matches, the exception is thrown and then immediately the `onError()` method will be executed.

We haven't got any errors to add `try...catch` block too, to handle the thrown

checked exception. Let's add the block and find out what happens, as in the following code:

```
observable.filter(item -> {
    try {
        return check(item);
    }
    catch (IOException e) {
        System.out.println(e.getMessage());
    }
    return false;
}).subscribe("//code for subscription);
```

On execution, we will get the output as shown in the screen shot:



The screenshot shows a Java IDE's console window. The title bar says 'Problems @ Javadoc Declaration Search Console'. The console tab is selected. The output window displays the following text:

```
<terminated> Demo_Observable_CheckedException
got:-12
got:-30
***got an exception****
got:-50
got:-89
sequence completed
```

Here, though an exception occurred, we got the sequence completed, proving that handling the exception is always good even if it's not insisted.

Sometimes, we try everything to recover from an error; it's a better option than failing.

Exploring the doOnError() operator

RxJava also provides `doOnError()` which gets a callback whenever an error occurs. It registers an action which will be taken when a source observable completes with an error. We can modify the earlier code as follows to tell it what to do if an error occurs:

```
observable.filter(item -> {
    try {
        return check(item);
    }
    catch (IOException e) {
        // TODO: handle exception
        throw new RuntimeException(e);
    }
}).doOnError(e ->
    System.out.println(e.getMessage())).subscribe(new
    Observer<Integer>() { //code for subscriber});
```

On execution, we will get the following output on the console:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_doOnError [Java Application]
got:-12
got:-30
java.io.IOException: ***got an exception***
java.lang.RuntimeException: java.io.IOException: ***got an exception***
```

The output shows that the exception occurred and `doOnError()` got invoked. Now, the question arises of why use it? We have already discussed the `flatMap()` operator, that creates observables and then emits them. It will be a good idea to use `doOnError()` operator within `flatMap()` to provide information on what went wrong with the current observable, keeping the observable stream still working. In this way, you can pass the message to the logger or to the UI.

Reacting to an error

We just caught an exception, but our operator or observer should respond to the `onError` notification. The `Observable` doesn't tend to emit an error, instead, it will emit a notification to the observer saying that an unrecoverable error has occurred and that the sequence will now be terminating. RxJava library provides a variety of operators which can be used by the developers to react to an `onError` signal or recover from it. We can use the operators to observe errors and take action to recover shown as follows:

- Absorb the error and switch over to the backup `Observable` that allows continuing sequence
- Absorb the error and emit a specified default item
- Absorb the error and then try to restart the `Observable` that has failed
- Absorb the error and then try to restart the `Observable` that has failed after some interval where the delay can be linear, exponential, or random

Recovering from an error

Resiliency is a feature supported by RxJava which aims to help applications recover from failure and continue with available functionalities. The resiliency has been supported by RxJava library using the operators as described in the following table:

Name of the method	When to use the method
onErrorResumeNext	Even when an error occurs, we want the <code>Observable</code> to emit a sequence
onErrorReturn	When an error occurs, we want an <code>Observable</code> to emit the specified item
onExceptionResumeNext	If an exception occurs, we want the <code>Observable</code> to continue emitting a sequence
retryWhen	Whenever the <code>Observable</code> emits an error, it will be passed to another <code>Observable</code> to make a choice to resubscribe to the source or not
retry	The <code>retry</code> operator facilitates resubscribing to the <code>Observable</code> when an error has been emitted

Whenever an error occurs, `onError` will be sent and this skips the code that is subscribing the data. However, now we are interested in invoking the `onNext()` method even though we have come across an error situation. The operators, as described in the table, help in facilitating recovery and also give a call to the `onNext()` method. Let's discuss them one by one.

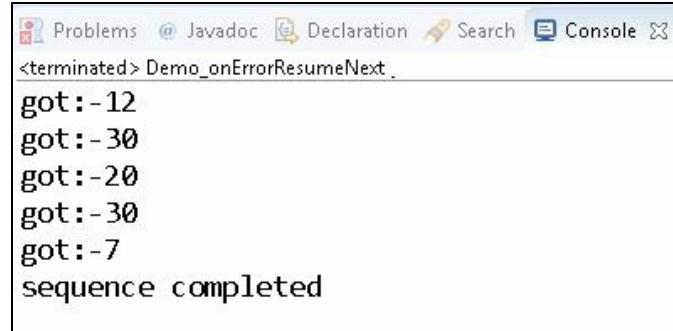
Absorb the error and switch over to the backup Observable which allows continuing the sequence using the `onErrorResumeNext()` operator

Whenever an error occurs, instead of terminating, we want the stream to resume with the new Observable sequence. The `onErrorResumeNext()` operator enables the developers to replace the current stream with a new one, if an error occurs.

Let's keep our `observable` and the operator as it is, however this time if an error occurs, we will create a new `Observable` to resume the emission, as shown in the following code:

```
public static void main(String[] args) {
    Observable<Integer> observable =
        Observable.just(12, 30, 5, 50, 89);
    observable.filter(item -> {
        try {
            return check(item);
        }
        catch (IOException e) {
            // TODO: handle exception
            throw new RuntimeException(e);
        }
    }).onErrorResumeNext(
        Observable.just(20, 30, 7)).subscribe(
            // code for subscriber);
}
```

The `onErrorResumeNext()` operator is creating an `Observable` which is emitting a sequence of items as 20, 30, and 7 as, shown in the following screenshot:



```
Problems @ Javadoc Declaration Search Console <terminated> Demo_onErrorResumeNext .
got:-12
got:-30
got:-20
got:-30
got:-7
sequence completed
```

From the output, it's clear that the `onErrorResumeNext()` operator facilitates emitting a new sequence, but what if we don't want the new sequence? Instead of a new sequence, we want to acquire the emitted value after which the error occurred.

Absorb the error and emit a specified default item using the `onErrorReturn()` operator

We just discussed how the `onErrorResumeNext()` operator facilitates the emission of an `observable`. What if we don't want an observable and we just want to emit a value? The `onErrorReturn()` operator facilitates replacing the `onError` signal with the emitted value, in turn giving the `onNext` signal. After subscribing the emission, the `onCompleted()` method will be invoked.

While unzipping the file at a location with a very long destination path, the software gives an error message saying that unzipping is not possible. Then we break the operation; the longer file path destination will not be zipped while others are. In the process of downloading, if something goes wrong, instead of terminating the complete download, we can write the code for downloading possible stuff in the body of the `onErrorResumeNext()` operator.

The following updated code shows how the operator `onErrorReturn()` emits a default value:

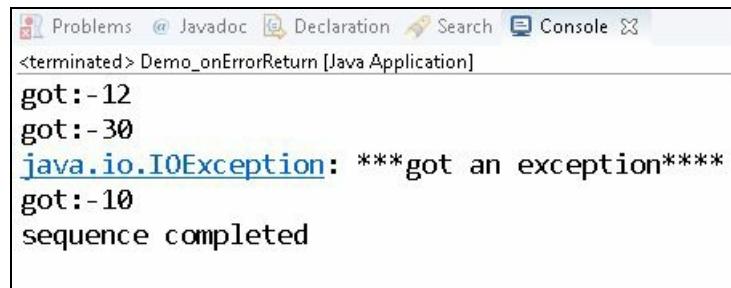
```
observable.filter(item -> {
    try {
        return check(item);
    }
    catch (IOException e) {
        // TODO: handle exception
        throw new RuntimeException(e);
    }
}).onErrorReturn(ex -> {
    System.out.println(ex.getMessage());
    return 10;
})
```

The `onErrorReturn()` operator accepts an argument of type `function`. The following code snippet shows the code for the `function`, which has been accepted by the `onErrorReturn()` operator:

```
|    new Function<Throwable, Integer>() {
```

```
    @Override
    public Integer apply(Throwable throwable) throws Exception {
        System.out.println(throwable.getMessage());
        return 10;
    }
}
```

Here the `Function` accepts an argument of type `Throwable` and handles it; after that, it will return a value which will be emitted instead of emitting an error signal. As the default item has been emitted, the signal sent is the `onNext` signal and not the `onError` signal. Execute the code and observe the output on the console:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_onErrorReturn [Java Application]
got:-12
got:-30
java.io.IOException: ***got an exception****
got:-10
sequence completed
```

The output shows an error occurred based on condition and then `onErrorReturn()` gets invoked. The operator has handled the exception and then returns the value `10`. The `Observable` has emitted an error but we replace it with an item. As the item is emitted its `onNext` signal will be sent which is ultimately subscribed by the subscriber. Once the subscription is done, there are no more items to be emitted, and so the `onCompleted` signal callbacks the `onComplete()` method.

Absorb the error and then try to restart the Observable that has failed using the `onExceptionResumeNext()` operator

The `onExceptionResumeNext()` operator is very much similar to `onErrorResumeNext()`, providing a new observable to emit the items. The main difference is that `onExceptionResumeNext()` will be used whenever we have a chance of getting an exception in the code.

The code to use `onExceptionResumeNext()` is shown as follows:

```
observable.filter(item -> {
    try {
        return check(item);
    }
    catch (IOException e) {
        // TODO: handle exception
        throw new RuntimeException(e);
    }
}).onExceptionResumeNext(Observable.just(10,20,78)).
    subscribe("//code for subscriber);
```

When an exception is thrown by the `check()` method, we have handled it using `try...catch` block. However, the `catch` block is rethrowing an exception. Now, the `onExceptionResumeNext()` method will be invoked, which returns the observable emitting specified items, available to the `Subscriber` for the subscription as shown in the following snapshot:



```
got:-12
got:-30
got:-10
got:-20
got:-78
sequence completed
```

Absorb the error and then try to restart the Observable which has failed using the retry() operator without delay

If an error occurs, the `retry()` operator enables the developers to retry the `Observable` for emission. The operator responds to the `onError` signal from the `Observable`, but this time instead of sending `onError` to the observer it will now resubscribe to the source `Observable`. It gives a new opportunity to the source `Observable` so that it will be able to complete the emission of the items. The operator `retry()` sends the `onNext` signal to the observer, which may lead to a duplicate item emission.

The `retry()` operator has the following four flavors which trigger resubscription to the source `Observable` immediately once it has failed:

- `retry()`: The `retry()` operator instantly starts retrying as long as the source `Observable` emits an error. However, this causes unending retrying and it's recommended you don't use the operator.
- `retry(long)`: This operator instantly starts trying for the resubscription until the maximum count specified by us doesn't reach. Once the count is reached to the specified value of the counter, no more resubscription happens. Once the maximum count has been reached, now instead of retrying, an error will be emitted.
- `retry(Predicate predicate)`: Instantly starts retrying as long as the predicate returns true.
- `retry(BiPredicate biPredicate)`: Instantly starts retrying as long as the `BiPredicate` returns true after checking the condition for retrying for a specific type of `Throwable`; developers can specify how many times retrying will be done.

Now we are aware of the flavors of the `retry()` method, let's use them one by

one.

Retrying for unlimited times using the `retry()` operator

The operator `retry()` is used to attempt retrying the `Observable` for emission, just because last time the emission happened, an error occurred. Now we want to fix the things in upstream. The `retry()` operator resubscribes the `Observable` whenever it receives a notification of `onError`. We will update the earlier `Observable` to give a second chance to the `Observable` to continue for the emission, as discussed in the following code:

```
public static void main(String[] args) {
    Observable<Integer> observable =
        Observable.just(12, 30, 5, 50, 89);
    observable.filter(item -> {
        try {
            return check(item);
        }
        catch (IOException e) {
            // TODO: handle exception
            throw new RuntimeException(e);
        }
    }).retry().subscribe("//code for subscription")
}
```

Now, when we execute the code we will get unlimited time `12` and `30` printed on the console. Once the `check()` method got the third item with a value equal to `5`, the condition passes, leading it to throw an exception. An `onError` signal will be sent which in turn gives a second chance to the source `Observable`. But instead of starting from the point where the exception occurred earlier, it actually restarts the `Observable`. It means we need to use the `retry()` operator very carefully as each time an error occurs, the `Observable` restarts the emission from the first item.

Specifying how many times retry will be attempted using the `retry(long)` operator

Let's rewrite the code written in the earlier program to use the overloaded `retry(long)` operator, trying to attempt resubscription from the `Observable` three times, as shown by the following code:

```
public static void main(String[] args) {
    Observable<Integer> observable =
```

```

        Observable.just(12, 30, 5, 50, 89);
observable.filter(item -> {
    try {
        return check(item);
    }
    catch (IOException e) {
        // TODO: handle exception
        throw new RuntimeException(e);
    }
}).retry(3).subscribe(new Observer<Integer>() {
    @Override
    public void onComplete() {
        // code for onComplete
    }

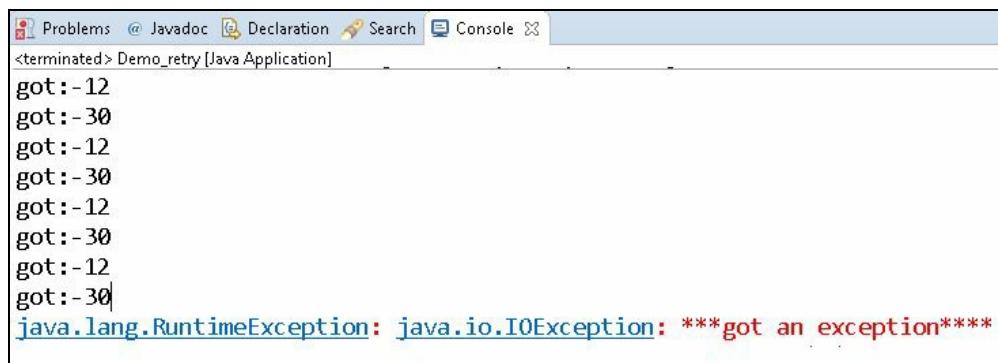
    @Override
    public void onError(Throwable throwable) {
        // TODO Auto-generated method stub
        throwable.printStackTrace();
    }

    @Override
    public void onNext(Integer value) {
        // code for displaying value on console
    }

    @Override
    public void onSubscribe(Disposable disposable) {
    }
});
}

```

Observe the code. `observable`, after filtering the items retrying for the resubscription three times as the `catch` block, is throwing an exception. After trying for the fourth time, if an exception occurs, instead of retrying once again, now the notification `onError` will be sent, causing the `onError()` method to be invoked, as shown in the following screenshot:



```

Problems @ Javadoc Declaration Search Console
<terminated> Demo_retry [Java Application]
got:-12
got:-30
got:-12
got:-30
got:-12
got:-30
got:-12
got:-30
java.lang.RuntimeException: java.io.IOException: ***got an exception***

```

Retrying for a particular type of Throwable using the retry(Predicate)

operator

We have overloaded the version of the `retry(Predicate)` method accepting a `Predicate`, which tests whether the thrown exception is of a specific type or not; if the test passes, the `Observable` will get a chance for resubscription. If the test fails, resubscription will not be done. Let's update the code to use the `retry()` operator, shown as follows:

```
observable.filter(item -> {
    try {
        return check(item);
    }
    catch (IOException e) {
        throw new RuntimeException(e);
    }
}).retry(new Predicate<Throwable>() {
    @Override
    public boolean test(Throwable throwable) throws
        Exception {
        // TODO Auto-generated method stub
        return throwable instanceof RuntimeException;
    }
}).subscribe("// code for subscription);
```

On execution of the code, unlimited resubscription happens, as every time the test will return the value as `true`.

Now, let's update the code to test whether the thrown exception is of some other type. Observe what happens if the test fails, as shown by the following code:

```
observable.filter(item -> {
    try {
        return check(item);
    }
    catch (IOException e) {
        // TODO: handle exception
        throw new RuntimeException(e);
    }
}).retry(new Predicate<Throwable>() {
    @Override
    public boolean test(Throwable throwable) throws
        Exception {
        // TODO Auto-generated method stub
        return throwable instanceof IOException;
    }
}).subscribe("//code for subscription);
```

Execute the code and observe the output, as shown by the following

screenshot:



```
Problems @ Javadoc Declaration Search Console 
<terminated> Demo_retry_Predicate [Java Application]
got:-12
got:-30
java.lang.RuntimeException: java.io.IOException: ***got an exception***
```

As the test written in for the `Predicate` doesn't match, instead of retrying, the `onError` notification will be sent, causing the `onError()` method to be invoked.

Retrying for specified times based upon condition using the retry(BiPredicate) operator

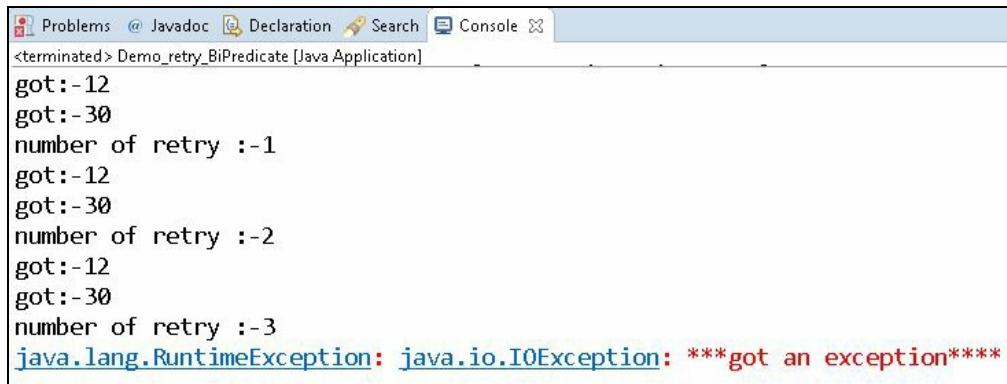
In this demo, we have just given resubscription, depending upon the testing and whether a typical exception is thrown or not. If it matches, retrying happens unlimited times, however, it's not feasible. Sometimes even though we retried 100 times, `observable` keeps on throwing exceptions. We need to give the `observable` a chance, but for limited times. `BiPredicate` helps us in choosing which `Throwable` retry will be attempted and how many times this attempt will be made. We can have the `Predicate` for runtime exception trying three times as shown by the following code:

```
observable.filter(item ->
{
    try {
        return check(item);
    }
    catch (IOException e) {
        // TODO: handle exception
        throw new RuntimeException(e);
    }
}).retry(new BiPredicate<Integer, Throwable>() {
    @Override
    public boolean test(Integer value, Throwable throwable)
        throws Exception {
        // TODO Auto-generated method stub
        System.out.println("number of retry :-" + value);
        if (value < 3 && throwable instanceof RuntimeException)
            return true;
        return false;
    }
}).subscribe("//code for subscription);
```

The `test()` method of `BiPredicate` accepts two arguments: the first argument is of type `integer`, which is a counter denoting how many times the `test()`

method has been invoked and the second argument is an instance of `Throwable`. In the code, we are checking whether three tries have been attempted and also whether the `Throwable` is an instance of the type `RuntimeException` or not.

On execution of the code, the output will try to attempt resubscription three times, as shown in the following screenshot:



The screenshot shows a Java application named "Demo_retry_BiPredicate" running in an IDE. The console output is as follows:

```
got:-12
got:-30
number of retry :-1
got:-12
got:-30
number of retry :-2
got:-12
got:-30
number of retry :-3
java.lang.RuntimeException: java.io.IOException: ***got an exception****
```

The output shows three attempts (number of retry :-1, :-2, :-3) with values -12 and -30. The final line is highlighted in red, indicating an exception was caught.

If we change the `Throwable` condition, no resubscription happens.

Absorb the error and then try to restart the Observable that has failed using the retryWhen() operator with delay

We have just discussed retrying using the `retry()` operator. The RxJava library provides the `retrywhen()` operator which gives flexibility to the developers to either take an action based upon the condition trigger, or ask for a resubscription.

The `retrywhen()` operator gets called whenever an error occurs on the source `observable`. Once it gets invoked, the function provided by the operator will be invoked, accepting an `observable` which contains the instance of the error which has just been emitted by the source `observable`. This `observable` can now emit an error, an item, or the `onComplete` signal. If the `observable` emits an error, it will be propagated to the downstream and no retrying happens. If it emits an item, a call will be given to `onNext`, giving a chance to the `observable` to retry. In case an `onComplete` signal is emitted, it will propagate to the downstream and no retrying happens. In the upcoming discussion we will cover these scenarios in depth.

Retrying based upon the condition by emitting an item using Observable using the `retryWhen()` operator

We will check the condition in the `retrywhen()` operator and then return an `observable` based upon the condition. We will update the earlier code to use `retrywhen()` shown as follows, to avoid the resubscription:

```
try {
    return check(item);
}
catch (IOException e) {
    throw new RuntimeException(e);
}
```

```

}).retryWhen(errors -> errors.flatMap(error ->
{
    if (error instanceof Exception)
    {
        System.out.println("this is error");
        return Observable.just(null);
    }
    return Observable.error(error);
})).subscribe("//code for subscription);

```

The function in the `retryWhen()` operator performs a check to find out if the acquired error is an instance of `Exception` or not. If the condition passes, the `Observable` emitting items will be returned. If the condition fails, the `Observable` emitting error will be returned. Whenever the `Observable` emitting items is returned, it triggers the resubscription from the source `Observable`. For the `Observable` emitting an error, the `onError` signal will be sent to the `Observer`, which results in the invocation of the `onError()` method.

Execute the code which we will show in the following output:



```

Problems @ Javadoc Declaration Search Console
<terminated> Demo_retryWhen [Java Application]
got:-12
got:-30
this is error
java.lang.NullPointerException: The item is null

```

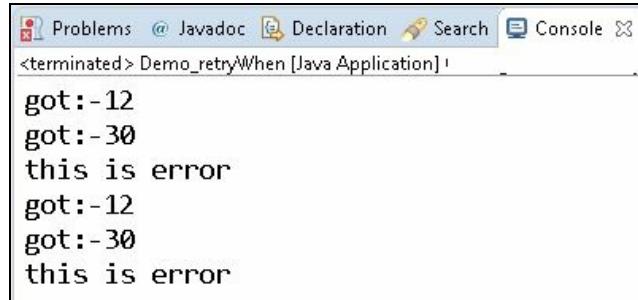
We got an exception just because instead of emitting an item, we emitted a null value. However, as per the discussion when the item got emitted, the `onNext` signal will be sent, which triggers the call to the `onNext()` method. Let's first of all check how the `onNext()` method gets invoked. Update the earlier code to emit an item, instead of emitting null, as shown in the following code:

```

retryWhen(errors -> errors.flatMap(error ->
{
    if (error instanceof Exception)
    {
        System.out.println("this is error");
        return Observable.just(1);
    }
    return Observable.error(error);
})).subscribe("// code for subscription);

```

After updating the preceding code, if we execute the code we will observe continuous resubscription from the observable on the console. A part of the output is as shown in the following screenshot:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_retryWhen [Java Application]
got:-12
got:-30
this is error
got:-12
got:-30
this is error
```

Retrying based upon a specific type of Throwable otherwise emitting an error using Observable using the retryWhen() operator

We discussed how to check for the condition in the `retryWhen()` operator and how to control the decision of whether to attempt retrial or not. We also discussed what happens when an item gets emitted; now let's find out what happens if an error is emitted by updating the code, as shown in the following code:

```
public static void main(String[] args) {
    Observable<Integer> observable =
        Observable.just(12, 30, 5, 50, 89);
    observable.filter(item -> {
        try {
            return check(item);
        }
        catch (IOException e) {
            // TODO: handle exception
            throw new RuntimeException(e);
        }
    }).retryWhen(errors -> errors.flatMap(error -> {
        if (error instanceof IOException) {
            System.out.println("this is error");
            return Observable.just(1);
        }
        return Observable.error(error);
    })).subscribe(new Observer<Integer>() {
        @Override
        public void onComplete() {
            // code for oncomplete
        }

        @Override
        public void onError(Throwable throwable) {
```

```

        // TODO Auto-generated method stub
        System.out.println("Got Exception:-
                           "+throwable.getMessage());
    }

    @Override
    public void onNext(Integer value) {
        // code for displaying the value on console
    }

    @Override
    public void onSubscribe(Disposable disposable) {
    }
);
}

```

We have thrown the `RuntimeException` from the `catch()` block. In the `retryWhen()` operator we have performed a check to find out whether the error is an instance of `IOException` or not. If the condition fails, we are returning an observable which will emit an error. Once the `Observable` emits an error, the `onError` signal will be sent to the `Observer` and the `onError()` method will be invoked, as shown in the following screenshot:



```

Problems @ Javadoc Declaration Search Console X
<terminated> Demo_retryWhen [Java Application]
got:-12
got:-30
Got Exception:-java.io.IOException: ***got an exception***

```

You can observe underlined statements from the code that correlate with the output shown in the preceding screenshot. The output clearly shows that no resubscription happened when the `onError` notification was sent.

Exceptions specific to RxJava

We already discussed that an `observable` doesn't tend to emit an error, instead, it emits the `onError` notification for the `observer`. However, there are a few exceptions such as when `onError()` fails, leading to an exception. We have the following RxJava specific exceptions which can help to handle the situation with a better approach, discussed as follows:

CompositeException

Whenever more than one exception occurs in the code, `CompositeException` occurs. As there are multiple exceptions raised we need a method which will provide us with information about the raised exceptions. Such information will be provided by the method `getExceptions()`.

MissingBackpressureException

We are well aware that the `Subscriber` or relevant operator facilitates the consumption of the emitted data. Sometimes the rate at which the data is emitted is very much higher than the rate at which it got consumed at the `Subscriber` end. Now, the operator that attempts consumption may apply reactive pull backpressure to the `Observable`, which doesn't implement the backpressure strategy. In such scenarios, the `MissingBackpressureException` will occur.

OnErrorNotImplementedException

An `observable` can emit an item, an error, or a notification. When an error is emitted by the `observable`, the `onError()` method gets invoked on the subscriber's end. The `OnErrorNotImplementedException` occurs when the `observable` tries to call the `onError()` method on the subscriber end but such a method doesn't exist.

The following are the ways to fix such an exception:

- To fix the `OnErrorNotImplementedException`, we need to work around the `observable` so that it won't reach the condition which will allow it to emit an error
- We can even can implement the `onError` handler on the `observer` to fix the exception
- Sometimes, we can intercept the `onError` notification before it reaches the `observer`

OnErrorFailedException

Once the `observable` emits the error, the Subscriber's `onError()` method gets invoked. Usually, we implement some of the exception handling mechanism here. Yes, we all know this. However, what if this implementation leads to us raising another exception? `onErrorFailedException` indicates that `observable` called the `onError()` method, but the method itself raised an exception.

OnErrorThrowable

`OnErrorThrowable` gets passed to the `onError()` method of the `observer`, which is of type `Throwable`. These `Throwable` instances contain information about the errors which have been raised. Along with this information, they will also contain the state of the system which is specific to the `Observable` when the error occurs.

We have just discussed various ways to swallow an error, keeping in mind that errors should not be emitted because the consumer at downstream has already completed its life cycle by reaching the terminal state or it has canceled the sequence which was about to emit an error. If the downstream is terminated, what will happen to such errors? There is a high possibility that the application may behave uncertainly or may stop working. We had a long discussion just to make the application keep working even though an error has occurred. Now, what do we do for such errors?

Error handler

The errors which have been emitted but cannot be swallowed can be routed to `RxJavaPlugins.onError` handler. `RxJavaPlugins` is a registry for plugin implementation which facilitates global override and handles the retrieval of correct implementation. The `RxJavaPlugins.onError` handles the emitted errors which have not been swallowed yet. By default, the `onError` prints the stack trace of the `Throwable` on the console and then calls the exception handler on the current thread for the uncaught exception.

Oops!! Too many things are happening. Let's first of all check what happens when we use the `RxJavaPlugins.onError`, as shown by the code written in `onError()` method as follows:

```
public class Demo_RxJava2 {  
  
    public static void main(String[] args) {  
        Observable<Integer> observable =  
            Observable.just(12, 30, 5, 50, 89);  
  
        observable.filter(item -> {  
            try {  
                return check(item);  
            }  
            catch (IOException e) {  
                // TODO: handle exception  
                throw new RuntimeException(e);  
            }  
        }).subscribe(new Observer<Integer>() {  
  
            @Override  
            public void onComplete() {// code goes here}  
  
            @Override  
            public void onError(Throwable throwable) {  
                //throwable.printStackTrace();  
                RxJavaPlugins.onError(throwable);  
            }  
  
            @Override  
            public void onNext(Integer value) {  
                // code for displaying values goes here  
            }  
  
            @Override  
            public void onSubscribe(Disposable disposable) {  
        }  
    }  
}
```

```

        // code goes here}
    }
}

public static boolean check(int item) throws IOException {
    // code goes here as written earlier
}
}
}

```

Execute the code. After execution, comment the underlined statement and uncomment the `throwable.printStackTrace()`. Once again, execute the code and now observe the output of both executions. The partial output has been shown in the following screenshot:

<pre> got:-12 got:-30 <u>io.reactivex.exceptions.UndeliverableException:</u> <u>java.lang.RuntimeException:</u> <u>java.io.IOException: ***got an exception****</u> <u>Caused by: java.lang.RuntimeException:</u> <u>java.io.IOException: ***got an exception****</u> <u>Caused by: java.io.IOException: ***got an exception****</u> <u>Exception in thread "main" io.reactivex.exceptions.UndeliverableException:</u> <u>java.lang.RuntimeException: java.io.IOException</u> </pre>	Stack trace using RxJavaPlugins.onError
<pre> got:-12 got:-30 <u>java.lang.RuntimeException:</u> <u>java.io.IOException: ***got an exception****</u> <u>Caused by: java.io.IOException: ***got an exception****</u> </pre>	stack trace using printStackTrace() method

The output shows by default the `throwable.printStackTrace()` method gets called and wherever the exceptions got propagated, the current thread's uncaught exception handler will be invoked.

`onError` prints the stack trace of the `Throwable` on the console and then calls the exception handler on the current thread for the uncaught exception.

We can even customise the handler by overriding `RxJavaPlugins.setErrorHandler()` to deal with the error as per our requirement.

Let's update the earlier code discussed as follows to use the custom handler:

- Add the custom handler using `setErrorHandler()` as follows:

```
RxJavaPlugins.setErrorHandler(new Consumer<Throwable>() {  
    @Override  
    public void accept(Throwable throwable) throws Exception  
    {  
        // TODO Auto-generated method stub  
        System.out.println("from the handler"+throwable.getMessage());  
    }  
});
```

Here, we have the `Consumer` of type `Throwable`. The `Consumer` has the `accept()` method which gets invoked as a callback.

- Keep the rest of the code as it is and execute it.
- We will get the following output on our console:

```
got:-12  
got:-30  
from the handlerjava.lang.RuntimeException: java.io.IOException: ***got an exception***
```

We got the output on our console from the `accept()` method, which gets a callback when we use `RxJavaPlugins.onError(throwable)`.

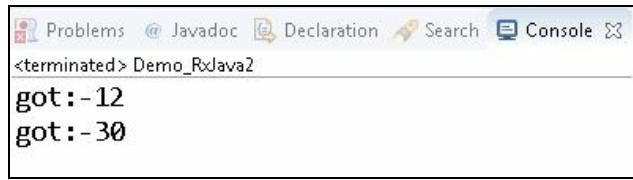
Sometimes, we want to avoid such calls to the uncaught exception handler in the application, in which case we should set a no-op handler. The code to set the handler is shown as follows:

```
| RxJavaPlugins.setErrorHandler(e->{});
```

Or we can even write the code as:

```
| RxJavaPlugins.setErrorHandler(Functions.emptyConsumer());
```

We can comment the handler written earlier and put the previous statement to get the following output:



The screenshot shows the Eclipse IDE's Console view. The title bar includes 'Problems', '@ Javadoc', 'Declaration', 'Search', and 'Console'. Below the title bar, it says '<terminated> Demo_RxJava2'. The main content area of the console shows two lines of text: 'got:-12' and 'got:-30'.

```
got:-12
got:-30
```

The output shows that no stack trace got printed on the console, as we haven't taken any action from the handler.

Handling specific undeliverable exceptions

We just used the `setErrorHandler()` method to provide a handler for the undeliverable exceptions. The thing to consider is that we caught all the undeliverable exceptions under the same hood. Only the cause will provide us with the information as to whether they represent a bug in the code or whether it is due to some network state. Depending upon the cause, the developers can take the decision of whether to treat them seriously or whether to ignore them. Now the question is, how to manage a typical undeliverable exception?

A very simple mechanism is to check the expected exceptions in the error handler and then take corrective measures. Let's take an example--in a code written by us we are expecting `IOException`, `NullPointerException`, `RuntimeException`, and a few more exceptions to be thrown, depending on the scenario. We will perform a simple check in the handler to find which exception has been thrown and then provide an action to take. We have thrown an `IOException` and caught it in the main function in the earlier codes. This time we will rethrow it as shown in the following code by the following statements:

```
public class Demo_RxJava2_Exception {
    public static void main(String[] args) {
        RxJavaPlugins.setErrorHandler(e -> {
            if (e instanceof UndeliverableException) {
                //undeliverable exception
                e = e.getCause();
            }
            if (e instanceof IOException) {
                // problems related to IO
                System.out.println(e.getMessage());
            }
        });
        Observable<Integer> observable =
            Observable.just(12, 30, 5, 50, 89);
        observable.filter(item -> {
            try {
                return check(item);
            }
            catch (UndeliverableException e) {
                // handle the exception
            }
        });
    }
}
```

```

        }
        catch (IOException e) {
            // TODO: handle exception
            throw new IOException("re throwing
                exception"+e.getMessage());
        }
    }).subscribe(new Observer<Integer>() {

        @Override
        public void onComplete() {// code goes here }

        @Override
        public void onError(Throwable throwable) {
            RxJavaPlugins.onError(throwable);
        }

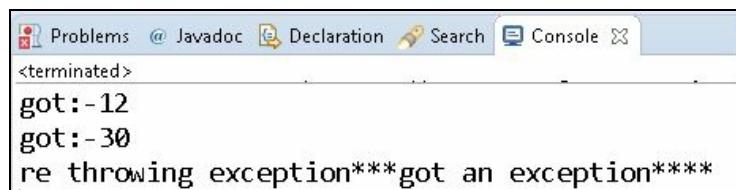
        @Override
        public void onNext(Integer value) {
            // code goes here for displaying the value
        }

        @Override
        public void onSubscribe(Disposable disposable) {
            // code goes here
        }
    });
}

public static boolean check(int item) throws IOException {
    boolean result = false;
    if (item > 10)
        result = true;
    else
        throw new IOException("****got an exception****");
    return result;
}
}

```

Execute the code to get an output from matching the `if` condition of `IOException` as shown in the following screenshot:



The screenshot shows a Java IDE's console window. The tabs at the top are 'Problems', 'Javadoc', 'Declaration', 'Search', and 'Console'. The 'Console' tab is active. The console output is as follows:

```

<terminated>
got:-12
got:-30
re throwing exception****got an exception****

```

Let's update the code to understand the mechanism of the handler by changing the rethrowing exception from the `catch` block as shown in the following code:

```

|     catch (IOException e)
|     {
|         // code goes here
|     }
| }

```

```
| {  
|     throw new RuntimeException("re throwing  
|         exception"+e.getMessage());  
| }
```

On execution, we will get the output as,

```
|     got:-12  
|     got:-30
```

Where is the exception trace? It's not displayed on the console, as the `setHandler()` method doesn't provide any condition to check for the instance `RuntimeException` and then to take any action. Let's add the condition to the existing `setHandler()` method to check for `RuntimeException` as shown in the following code:

```
| if (e instanceof RuntimeException) {  
|     Thread.currentThread().  
|         getUncaughtExceptionHandler().  
|             uncaughtException(Thread.currentThread(), e);  
|     return;  
| }
```

Here, we are using the `Thread.getUncaughtExceptionHandler()` method which returns an instance of `Thread.UncaughtExceptionHandler`. This instance will provide a handler. This handler will be invoked by the JVM whenever a thread terminates because of an uncaught exception to query the thread for its `UncaughtExceptionHandler`. The `uncaughtException()` method will be at this point by passing the thread and the exception as its argument. Whenever an explicit `UncaughtExceptionHandler` is absent, the instance `ThreadGroup` acts as `UncaughtExceptionHandler`.

Execute the code to get the following output:

```
got:-12  
got:-30  
Exception in thread "main"  
java.lang.RuntimeException: re throwing exception***got an exception****
```

We can extend the code of the handler to allow developers to handle various kinds of uncaught exception, shown as follows:

```
| RxJavaPlugins.setErrorHandler(e -> {  
|     if (e instanceof UndeliverableException) {
```

```

        //undeliverable exception
        e = e.getCause();
    }
    if (e instanceof IOException) {
        // problems related to IO
        System.out.println(e.getMessage());
    }
    if (e instanceof SocketException) {
        // network problem or API that throws on cancellation
        System.out.println("its socket
            exception"+e.getMessage());
        return;
    }
    if (e instanceof InterruptedException) {
        // blocking code was interruption
        return;
    }
    if (e instanceof NullPointerException) {
        // a bug in the application
        Thread.currentThread().getUncaughtExceptionHandler().
            uncaughtException(Thread.currentThread(), e);
        return;
    }
    if (e instanceof IllegalArgumentException)
    {
        return;
    }
    if (e instanceof RuntimeException) {
        Thread.currentThread().getUncaughtExceptionHandler().
            uncaughtException(Thread.currentThread(), e);
        return;
    }
    if (e instanceof IllegalStateException) {
        // that's a bug in operator
        Thread.currentThread().getUncaughtExceptionHandler().
            uncaughtException(Thread.currentThread(), e);
        return;
    }
});

```

We can extend the conditions to cover all those exceptions which may occur in the code.

Exceptions introduced for tracking

The error occurred and the exception may terminate, but why the application terminated, the source of the error and the reason behind it must be known. RxJava 2.0.6 introduced the following exceptions which help to track what went wrong when the error occurred:

Exception	Description
OnErrorNotImplementedException	The exception is used to detect when the user forgot to add the error while handling the <code>subscribe()</code> method
ProtocolViolationException	The exception denotes a bug in an operator
UndeliverableException	It's a default exception which is applied by the <code>RxJavaPlugins.onError</code> which facilitates tracking of the operator which rerouted an error

Exception wrapping

The stack trace of the code we executed using the handler shows the `UndeliverableException` wrapped the exceptions handled by the handler. However, whenever exceptions such as `IllegalStateException`, `IllegalArgumentException`, `NullPointerException`, `CompositeException`, `MissingBackpressureException`, or `OnErrorNotImplementedException`, and `UndeliverableException` occur, the wrapping won't happen.

Update the code as shown to throw `NullPointerException` to find out how wrapping doesn't happen:

```
public class Demo_RxJava2 {

    public static void main(String[] args) {
        Observable<Integer> observable =
            Observable.just(12, 30, 5, 50, 89);
        observable.filter(item -> {
            try {
                return check(item);
            }
            catch (IOException e) {
                throw new
                    NullPointerException(e.getMessage());
            }
        }).subscribe(new Observer<Integer>() {

            @Override
            public void onComplete() {
                // code goes here
            }

            @Override
            public void onError(Throwable throwable) {
                // TODO Auto-generated method stub
                RxJavaPlugins.onError(throwable);
            }

            @Override
            public void onNext(Integer value) {
                // code goes here for displaying the value
            }

            @Override
            public void onSubscribe(Disposable disposable) {
                // code goes here
            }
        });
    }
}
```

```

        });
    }
    public static boolean check(int item) throws IOException {
        boolean result = false;
        if (item > 10)
            result = true;
        else
            throw new NullPointerException("****got an
                exception****");
        return result;
    }
}

```

On execution of the code, we will get the output which shows that no wrapping happens by `UndeliverableException`. Observe the output for the comparison of the output obtained by executing the code with the earlier code using custom handler:

```

got:-12
got:-30
java.lang.NullPointerException: ***got an exception****
Exception in thread "main"
    java.lang.NullPointerException: ***got an exception****

got:-12
got:-30
io.reactivex.exceptions.UndeliverableException:
    java.lang.RuntimeException:
        java.io.IOException: ***got an exception****
Caused by: java.lang.RuntimeException:
    java.io.IOException: ***got an exception****
Caused by: java.io.IOException: ***got an exception****

Exception in thread "main" io.reactivex.exceptions.UndeliverableException:
    java.lang.RuntimeException: java.io.IOException

```

Stack trace
using
RxJavaPlugins.
onError
with
NullPointerException

Stack trace
using RxJavaPlugin.
onError
with
RuntimeException

Summary

The occurrence of exceptions at runtime is an avoidable part of any application. As a developer, we will take all the corrective measures so that an exception will not occur. However, we cannot give assurance that there is no exception. And whenever exception occurs, our application flow stops. We want our application to run smoothly, and if something goes wrong, the application should handle the situation by continuing ahead. We want our application to be resilient. Resiliency is the ability of an application to recover from difficulties and continue serving.

We discussed in this chapter, in depth, how to handle the errors in RxJava to recover with the help of the operators such as `onErrorResumeNext`, `onErrorReturn`, `retryWhen`, and `retry`. We also discussed `doOnError()` which gets a callback whenever an error occurs, to take an action which will be taken when a source observable completes with an error. We also discussed RxJava specific exceptions such as `CompositeException`, `MissingBackpressureException`, `OnErrorNotImplementedException`, and `OnErrorFailedException`.

Later, we talked about how the errors which have been emitted but cannot be swallowed will be routed to the `RxJavaPlugins.onError` handler. We also discussed the default as well as the customized handler using the `RxJavaPlugins.setErrorHandler()` to deal with the error, as per our requirements. Along with this, we also discussed how to handle the specific exceptions to take certain actions in the `setErrorHandler()` method.

In the next chapter, we will discuss various tips and tricks around unit testing the asynchronous flows. RxJava provides various testing utilities which makes unit testing the flows easier.

Testing

An orientation about the situations when an application may fail, and what are the corrective measures to be taken in order to recover from the failure, using various error handling operators and utilities was given in [Chapter 7, Building Resiliency](#). The application needs to be resilient to keep the application running even though the application has failed. Before resiliency, we discussed the creation of `observable` and `observer` using various operators, and how to process the data emitted by an observable. And now you are well aware of the way to develop an application supporting asynchronous reactive programming. Similar to non-reactive applications, while designing reactive applications, the developers want to provide a precise solution for the problem around which they design the application. The developers always take utmost care to design, as well as develop the application such that it fulfills the purpose.

Application development takes lots of time and money. After developing an application, the first thought that comes to mind is whether it will be accurate or not. Under which scenarios will it melt, and if it is going to melt down, how to handle such a situation? The answer is to test the application. Testing gives developers an assurance that the application developed will sustain under the given situations. In this chapter, we will discuss testing to cover the following points:

- Testing reactive sources
- Testing reactive consumers
- Time-based testing
- Testing utilities

Testing the need and role of a developer

Testing is one of the very important and time-consuming phases in application development, and it needs to be carried out seriously. As a developer, we have to keep in mind that along with developing the application, checking whether it is giving the desired results or not is our responsibility. You may think if we have testers, why do we need to test the application? Testing an application doesn't mean just checking how the final product works. It is actually a continuous process at each stage of development. And we, as developers, are a very important part of it. Testing can be carried out as unit testing, integration testing, system testing, and user-acceptance state, at different phases in application development.

We, as developers, start with testing the very basic structure of an application--a function. This is also called **unit testing**, where we consider a single function to carry out the test. Let's consider that we are developing a function which accepts two numbers and calculates their division as follows:

```
public static float div(int num1, int num2){  
    return (num1/num2);  
}
```

I know nothing can be simpler than this code. Now let's invoke the code with this set of commands:

```
public static void main(String []args)  
{  
    System.out.println(div(10,2));  
    System.out.println(div(10,3));  
}
```

After executing the code, we will get the answers displayed on the console as 5.0 and 3.0. Did we expect the same answer? Most of us did not. The division for the second computation has to be 3.33333. So, why did this happen? Yes, it's a very common mistake of dividing two integers, which results in an integer and not a float value. However, while developing the function, we just

overlooked it. Now, instead of this simple manipulation, what if we were to calculate loan interest? Can we afford to lose the value, the precision? Surely, not! The developer knows what he is developing, and what is the required result, and thus, he is the right person to cross-check the developed code. In case a problem occurs, he is capable of giving the solution by modifying the code. Such kind of code testing is known as unit testing.

As in all applications, unit testing is important to understand and overcome the problems at a very early stage of application development, so it is also important in a reactive application. The only difference lies in the testing approach. It's different as we don't have only the result to test. A reactive application has an `Observable`, `Subscriber`, and `Notification` as the components without which a reactive application is difficult to imagine. Reactive application testing consists of testing these components. We will not directly deep dive into a discussion on how to test them, let's do it step by step. We will carry out the testing using JUnit and the Mockito framework as utilities to carry out testing. Most of us already know unit testing, let's use the same knowledge of JUnit to start with.

The traditional way of testing an Observable

Unit testing concentrates on a function to find out if the expected result matches with the actual result or not, under various probable situations. To test reactive components, we will start with a simple `Observable` which emits items, and tests whether the expected result matches with the actual or not. We will create `observable` using the `just()` operator, emitting three items. We are also going to create a consumer which will add the items to an `ArrayList` instance. Later on, using the `assertXXX()` methods, we will cross-check the actual and expected values for the following test cases:

- The list is not null
- The list size is equal to three as our `observable` is emitting three items
- The second emitted item has the value equal to the actual value emitted by the `Observable`

The code for the preceding test cases is as follows:

```
public class Traditional_Testing {  
    @Test  
    public void test_just() {  
        Observable<Integer> observable =  
            Observable.just(12, 34, 6);  
        ArrayList<Integer> list = new ArrayList();  
        observable.subscribe(item -> list.add(item));  
  
        assertNotNull(list);  
        assertEquals(3, list.size());  
        assertEquals(6, list.get(2).intValue());  
    }  
}
```

The preceding test will execute successfully, proving that the items emitted by the `observable` have been correctly consumed by the consumer.

It works fine, so why do we say that it's the traditional approach? I guess you overlooked the most important thing--parallel programming. I am calling this the traditional approach, as the result and the code for updating the result

must be on the same thread, or we need to keep them so. As here, our `observable` and the `Observer` work in the same thread, the `assert` blocks will wait for the `subscribe()` method to finish its work and then execute. It does not work asynchronously, which is our main aim while writing reactive applications.

The modern approach for testing an Observable

The RxJava library has the `BaseTestConsumer`, `TestSubscriber`, and `TestObserver` classes which enable the developers to work with the asynchronous processing of the events.

The BaseTestConsumer class

`BaseTestConsumer` is an abstract class which shares infrastructure for testing its children classes, `TestSubscriber` and `TestObserver`. It provides various `assertXXX()` methods for testing the data, error, or notifications emitted by an observable. We will discuss the provided methods later in this chapter instead of discussing them together.

The TestObserver class

The RxJava library also provides `TestObserver`, which facilitates testing of a non-backpressured `Observable` such as `Single`, `Maybe`, and `Completable`. An `Observer` records the events and allows making assertions about them. The developers can override methods such as `onSubscribe()`, `onNext()`, `onComplete()`, `onSuccess()`, `onError()`, and `cancel()`.

Let's use `TestObserver` to test the `Observable` that we created in the earlier application. We will create an instance of `TestObserver`, and then pass it to the `subscribe()` method of the `Observable` to collect the emitted items. With the help of the `assertXXX()` method provided by `TestObserver`, we will carry out testing as follows:

```
@Test
public void test_just_new() {
    Observable<Integer> observable =
        Observable.just(12, 34, 6);

    TestObserver<Integer> testObserver = new TestObserver<>();
    observable.subscribe(testObserver);

    List<Integer> list = new ArrayList();
    testObserver.assertResult(12, 34, 6);
    testObserver.assertValueCount(3);
    testObserver.assertValueAt(2, (value)-> {
        // TODO Auto-generated method stub
        return value==6;
    });
}
```

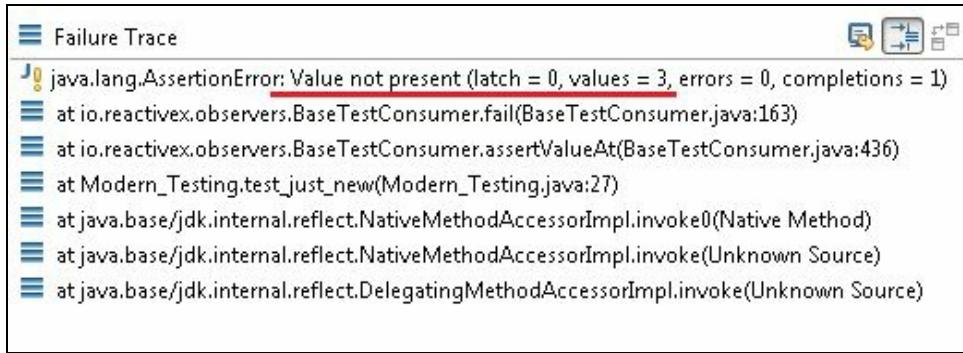
On execution, our test case will pass successfully proving that the items emitted are subscribed by the `testObserver`.

Now, change the condition for the predicate as shown next:

```
    testObserver.assertValueAt(2, (value) -> {
        // TODO Auto-generated method stub
        return value == 34;
    });
}
```

Yes, we are checking the wrong value, and obviously, the test case will fail

leading to the following failure trace output:



The screenshot shows a 'Failure Trace' window with the following stack trace:

```
java.lang.AssertionError: Value not present (latch = 0, values = 3, errors = 0, completions = 1)
at io.reactivex.observers.BaseTestConsumer.fail(BaseTestConsumer.java:163)
at io.reactivex.observers.BaseTestConsumer.assertValueAt(BaseTestConsumer.java:436)
at Modern_Testing.test_just_new(Modern_Testing.java:27)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
```

The line 'Value not present (latch = 0, values = 3, errors = 0, completions = 1)' is underlined, indicating the error point in the code.

The underlined trace directly shows that at position 2, we don't have an item whose value is 34. Along with that, it also shows how many numbers of `onNext`, `onError`, and `onCompletion` events are received by `TestObserver`.

The methods which we used for the assertion are shown in the following table:

Name of the method	Description	Class providing the method
<code>assertNotSubscribed()</code>	The method asserts that the method <code>onSubscribe()</code> has not been invoked	<code>TestObserver</code>
<code>assertValueCount(int count)</code>	<code>assertValueCount(int count)</code> asserts how many numbers of <code>onNext</code> events have been received	<code>BaseTestConsumer</code>
<code>assertResult(T..values)</code>	<code>assertResult()</code> asserts that the source <code>Observable</code> has signaled the items specified in the argument list, and has completed the emission normally	<code>BaseTestConsumer</code>
	The <code>assertValueAt()</code> method for asserting <code>TestObserver</code> or	

assertValueAt(int index, Predicate test)	TestSubscriber has received the onNext value at the index specified, and will have the value which is tested by the Predicate	BaseTestConsumer
--	---	------------------

Let's now test the `skip()` operator to get a grip on testing. We will use the code to create `observableDemo_skip` from the project `ch05_Demo_operators` given earlier in [Chapter 5, Operators](#) and test it as shown next:

```

    @Test
    public void test_skip() {
        String[] fruits =
            { "mango", "pineapple", "apple", "mango", "papaya" };
        Observable<String> observable=
            Observable.fromArray(fruits).skip(3);
        TestObserver<String> testObserver=new TestObserver<>();
        observable.subscribe(testObserver);
        testObserver.assertValueCount(2);
        testObserver.assertValues("mango", "papaya");
    }

```

Our test case will pass for all the conditions.

The newly introduced method in code is as discussed in this table:

Name of the method	Description	Class providing the method
assertValues(T...values)	The method asserts that the Observer or the Subscriber received the items with the values specified in the argument and in the same order	BaseTestConsumer

We also have obtained the `observable` using the `never()` operator which never emits items and never terminates; let's write the test case for it as follows:

```
    @Test
    public void test_never()
    {
        Observable<String> observable=observable.never();
        TestObserver<String> testObserver=new TestObserver<>();
        observable.subscribe(testObserver);
        testObserver.assertNoValues();
    }
```

This test case will pass successfully as there are no values emitted by the Observable.

Now, let's update the code once again by adding the following code to understand whether our observable terminated normally or not:

```
    |     testObserver.assertTerminated();
```

After the update, run the test case, and we will get the following failure trace:



The method `assertTerminated()` asserts whether `TestObserver` has terminated or not. On termination, the value of latch will be equal to zero.

The underlined trace shows that the observable has not been terminated--it's running--and the terminal latch will have a non-zero value.

The TestSubscriber class

`TestSubscriber` is a child of `BaseTestConsumer`, which is a subscriber that facilitates the recording of the events and allows asserting about them. Developers can also override the methods such as `onSubscribe()`, `onNext()`, `onError()`, and `onComplete()`. When developers call the default request method, they are actually requesting on behalf of the wrapper. We have `Flowable` as an `Observable` which can be tested by `TestSubscriber`.

Let's create a `Flowable` as we did in [Chapter 5, Operators](#), and then we will test the emitted items using `TestSubscriber`, as shown in the following code:

```
    @Test
    public void test_just_Flowable() {
        Flowable<String> observable =
            Flowable.just("mango", "papaya", "guava");
        TestSubscriber<String> testSubscriber =
            new TestSubscriber<>();

        observable.subscribe(testSubscriber);
        List<String> items = testSubscriber.values();

        testSubscriber.assertSubscribed();
        testSubscriber.assertValueCount(3);
        testSubscriber.assertValues("mango", "papaya", "guava");
    }
}
```

This preceding test case will pass successfully. The new methods used in the code are as shown in this table:

Name of the method	Description	Class providing the method
<code>assertSubscribed()</code>	The method asserts that the <code>onSubscribe()</code> method has invoked exactly once	The method is abstract in <code>BaseTestConsumer</code> which is implemented by <code>TestSubscriber</code>

`Observable` emits an item, a notification, or an error. We just discussed how to test the `observable` emitting an item or emitting nothing. Now, let's find out how to test an `observable` which emits an error using the `assertXXX()` methods as discussed in the table that follows:

Name of the method	Description	Class providing the method
<code>assertErrorMessage()</code>	The method asserts that a single error with the specified error message value has been received.	<code>BaseTestConsumer</code>
<code>assertFailure()</code>	The method asserts that the source <code>Observable</code> has failed and signaled. The method also helps to specify which subclass of <code>Throwable</code> causes the failure.	<code>BaseTestConsumer</code>
<code>assertFailureAndMessage()</code>	The method asserts that the source <code>Observable</code> has failed and signaled. The method also specifies which subclass of the <code>Throwable</code> causes the failure with the given error message.	<code>BaseTestConsumer</code>

Let's now use these methods to test the `Observable` emitting error.

We will use the `error()` operator to create an `observable` which will emit an instance of the `Exception`, and then we will test it using `TestObserver` as shown in the following code:

```

@Test
public void test_error() {
    Observable<String> observable = Observable.error(
        new Exception("We got an Exception"));
    TestObserver<String> testObserver = new TestObserver<>();
    observable.subscribe(testObserver);
}

```

```
    testObserver.assertError(Exception.class);
    testObserver.assertErrorMessage("We got an Exception");
    testObserver.assertFailure(exception -> {
        return exception instanceof Exception;
    });
}
```

The test case will pass successfully.

Testing time-based Observable

We just discussed testing an `Observable` which emits items that are not dependent of time. However, when we discussed creating an `Observable` in [Chapter 4, Reactive Types in RxJava](#), in the *Creating Observables for unrestricted infinite length* section we said that they are time-based. We discussed operators such as `interval()` and `timer()`. The `interval()` operator facilitates the emission of items of type `integer` after a given interval. An `Observable` emits the items after an equal interval of an infinite sequence of ascending integers. As it's a time-based operation, we need a `Scheduler` which will allocate a new thread other than the main. Similar to `TestObserver` and `TestSubscriber`, the RxJava library provides `TestScheduler` which facilitates testing using `Scheduler`.

The TestScheduler class

`TestScheduler` is a non-thread safe extension of the `Scheduler` class. This scheduler facilitates the testing of the operators that require a scheduler without real concurrency. The scheduler also facilitates the manual advancing of virtual time. The `TestScheduler` class has methods to facilitate scheduling of virtual time as discussed in the following table:

Name of the method	Description of the method
<code>advanceTimeBy(long delay, TimeUnit timeUnit)</code>	This method facilitates moving of the clock of the scheduler forward by the duration specified by the parameter.
<code>advanceTimeTo(long delay, TimeUnit timeUnit)</code>	This method facilitates moving of the clock of the scheduler to a specified moment in time.
<code>now(TimeUnit unit)</code>	This method returns the current time of the Scheduler in the particular time unit.
<code>triggerActions()</code>	This method facilitates triggering of any action which has not yet triggered. However, the action is scheduled to be triggered at or before the present time of the Scheduler.

Let's use `TestScheduler` to test observable emitting items after a specified interval as follows:

```
@Test
public void test_interval()
{
    TestScheduler testScheduler=new TestScheduler();
    Observable<Long>observable=Observable.interval(1,
        TimeUnit.SECONDS,testScheduler).take(5);
    TestObserver<Long> testObserver=new TestObserver<>();
    observable.subscribeOn(testScheduler).
```

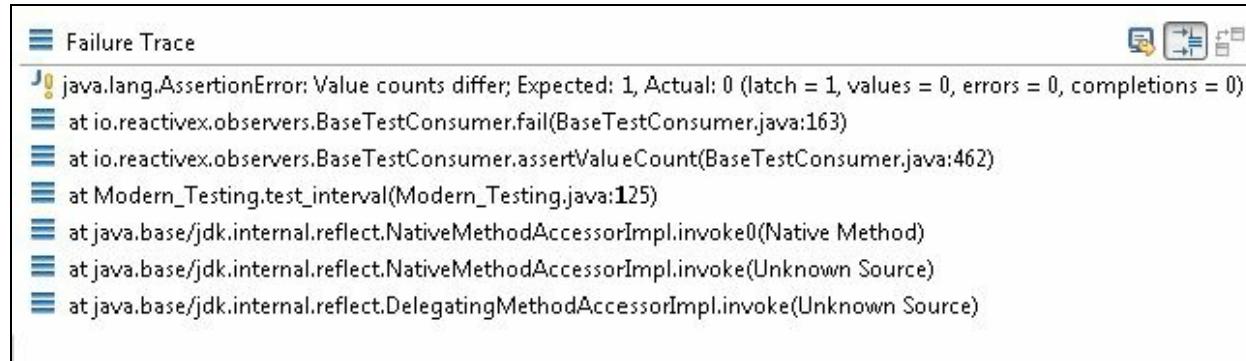
```

        subscribe(testObserver);

        testObserver.assertNoValues();
        testObserver.assertValueCount(1);
        testObserver.assertValues(01);
    }
}

```

Run the test case, and observe the failure trace as shown next:



The screenshot shows a 'Failure Trace' window with the following stack trace:

```

Failure Trace
java.lang.AssertionError: Value counts differ; Expected: 1, Actual: 0 (latch = 1, values = 0, errors = 0, completions = 0)
    at io.reactivex.observers.BaseTestConsumer.fail(BaseTestConsumer.java:163)
    at io.reactivex.observers.BaseTestConsumer.assertValueCount(BaseTestConsumer.java:462)
    at Modern_Testing.test_interval(Modern_Testing.java:125)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)

```

The output shows that we completed the execution even before the starting of the emission. Here, the `Scheduler` plays an important role. Let's update the code by adding the method of the `Scheduler`, `advanceTimeBy()`, to forward the clock of the `Scheduler` by a second to test emission. In the same way, we will use the `advanceTimeTo()` method to move the clock of the `Scheduler` to a specified moment in time. Let's update the code as follows:

```

@Test
public void test_interval()
{
    TestScheduler testScheduler=new TestScheduler();
    Observable<Long>observable=Observable.interval(1,
        TimeUnit.SECONDS,testScheduler).take(5);
    TestObserver<Long> testObserver=new TestObserver<>();
    observable.subscribeOn(testScheduler).subscribe(testObserver);

    testObserver.assertNoValues();
    testScheduler.advanceTimeBy(1, TimeUnit.SECONDS);
    testObserver.assertValueCount(1);
    testObserver.assertValues(01);

    testScheduler.advanceTimeTo(6, TimeUnit.SECONDS);
    testObserver.assertValueCount(5);
    testObserver.assertValues(01,11,21,31,41);
}

```

Now, on execution, the test case will pass giving us the desired result.

Until now, we discussed various ways to test the observable. Let's now move ahead and start testing the subscriber.

Testing the Subscriber

The `Subscriber` has `onNext()`, `onError()`, `onComplete()`, and `onSubscribe()` methods which get a callback on the arrival of their respective events. We already have discussed the `Subscriber` creations in [Chapter 4, Reactive Types in RxJava](#), and created many subscribers such as `DefaultSubscriber`, `DisposableSubscriber`, and `ResourceSubscriber`. If you want, you can refer to the chapter.

Now let's create a `Subscriber` as shown next:

```
class DemoSubscriber implements Subscriber<Long> {
    List<Long> items=new ArrayList();
    public List<Long> getItems() {
        return items;
    }

    @Override
    public void onComplete() {
        // TODO Auto-generated method stub
        System.out.println("Its Done!!!");
    }
    @Override
    public void onError(Throwable throwable) {
        // TODO Auto-generated method stub
        throwable.printStackTrace();
    }
    @Override
    public void onNext(Long value_long) {
        // TODO Auto-generated method stub
        System.out.println(value_long);
        items.add(value_long);
    }
    @Override
    public void onSubscribe(Subscription subscription) {
        // TODO Auto-generated method stub
        System.out.println("onSubscribe invoked");
        subscription.request(Long.MAX_VALUE);
    }
}
```

Our `Subscriber` subscribes to items of type `Long`. Now let's create an observable using the `rangeXXX()` operator, and then test the `DemoSubscriber` as follows:

```
public class Test_Subscriber {
    @Test
    public void test_subscriber() {
```

```

    DemoSubscriber demoSubscriber=new DemoSubscriber();
    Flowable.rangeLong(5, 4).subscribe(demoSubscriber);

    List<Long>items= demoSubscriber.getItems();
    assertEquals(4,items.size());
    assertTrue(6==items.get(1));
}
}

```

The test case will pass on execution.

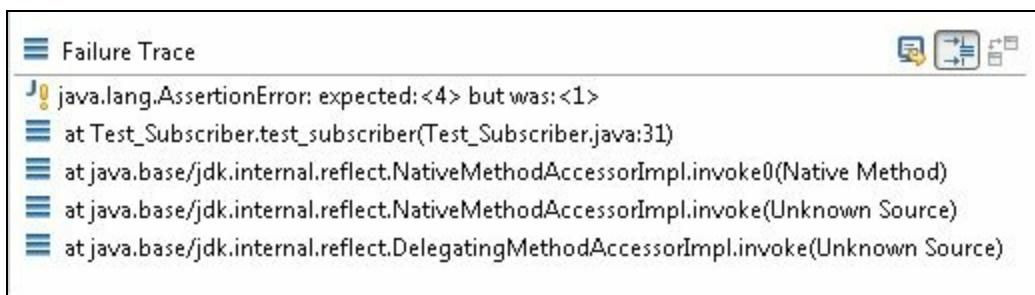
Now, let's change the following line of code:

```
| subscription.request(Long.MAX_VALUE);
```

The changed code would be as follows:

```
| subscription.request(1);
```

Keep the test case as it is, and we will get the failure trace as follows:



From the output, we can observe that only a single item has been requested by Subscriber. This is because we have changed

`subscription.request(Long.MAX_VALUE)` to `subscription.request(1)`.

We have multiple subscribers, one of which is `ResourceSubscriber`. Let's write the implementation of `ResourceSubscriber` as follows:

```

public class MyResourceSubscriber extends ResourceSubscriber<Long> {
    List<Long> values = new ArrayList<Long>();

    List<Throwable> errors = new ArrayList<Throwable>();
    int complete;

    @Override
    protected void onStart() {
        super.onStart();
    }
}

```

```

@Override
public void onNext(Long value) {
    values.add(value);
    request(Long.MAX_VALUE);
}

@Override
public void onError(Throwable e) {
    errors.add(e);
}

@Override
public void onComplete() {
    dispose();
}
}

```

For more information about the types of `Subscriber`, you can read [Chapter 4, *Reactive Types in RxJava*](#).

Now let's create a test case like this:

```

@Test
public void test_resourceSubscriber() {
    MyResourceSubscriber resourceSubscriber =
        new MyResourceSubscriber();

    assertFalse(resourceSubscriber.isDisposed());
    assertTrue(resourceSubscriber.values.isEmpty());
    assertTrue(resourceSubscriber.errors.isEmpty());

    Flowable.just(101).subscribe(resourceSubscriber);

    assertTrue(resourceSubscriber.isDisposed());
    assertEquals(101,
        resourceSubscriber.values.get(0).intValue());
    assertTrue(resourceSubscriber.errors.isEmpty());
}

```

The preceding test case will pass successfully.

In all the codes discussed till now, we used `TestObserver` and `TestSubscriber` by creating their instance. The `Observable` types also provide the `test()` operator as a utility for testing.

The test() operator

The RxJava library now has a `test()` operator which enables internal testing. The operator `test()` returns an instance of `TestSubscriber` OR `TestObserver`. We can use the `test()` operator as shown next:

```
public void testOperator() {  
    TestSubscriber<Long> test_Subscriber =  
        Flowable.rangeLong(10, 5).test();  
    TestObserver<Integer> testObserver =  
        Observable.just(12, 89, 67).test();  
    TestObserver<String> testObserver2 = Single.just("hello").test();  
    TestObserver<String> testObserver3 = Maybe.just("Mango").test();  
}
```

The `test()` operator provides a convenient way to obtain an instance of the `TestObserver` and `TestSubscriber`. Now we can assert the obtained result using the instance of the consumers. Let's use it to test `observer` as shown by the following code:

```
@Test  
public void testOperator_range() {  
    TestSubscriber<Long> test_Subscriber =  
        Flowable.rangeLong(10, 5).test();  
    test_Subscriber.assertResult(10L, 11L, 12L, 13L, 14L);  
    test_Subscriber.assertValueAt(2, (item) -> {  
        return item == 12L;  
    });  
}
```

On execution of the last code, the test case will pass successfully. We can even do it without using `TestSubscriber` OR `TestObserver` as shown next:

```
@Test  
public void test_just()  
{  
    Observable.just(12, 6, 33, 90).test().assertResult(12, 6, 33, 90);  
}
```

The new methods introduced for assertion are as discussed in the following table:

Name of the methods	Description
assertResult()	This method asserts that the <code>Subscriber</code> has received a number of items in the given order. After subscription, the <code>onComplete</code> event and a <code>no onError</code> signal will be received
assertFailure()	This method asserts that <code>TestObserver</code> or <code>TestSubscriber</code> has received the given items in the same order after which the <code>Throwable</code> error has been received.
awaitDone()	This method waits for the terminal event, and if some time elapses, then the sequence will be canceled.
assertOf()	This method is composed for asserting a chain.

The `test()` operator has a few overloaded versions as discussed in the following:

- `test(long request)`: This method specifies how many requests will be made
- `test(long initialRequest, boolean cancel)`: This specifies how many requests will be made, and whether to terminate the `Subscriber` or not.

Let's use it in the code to understand its working:

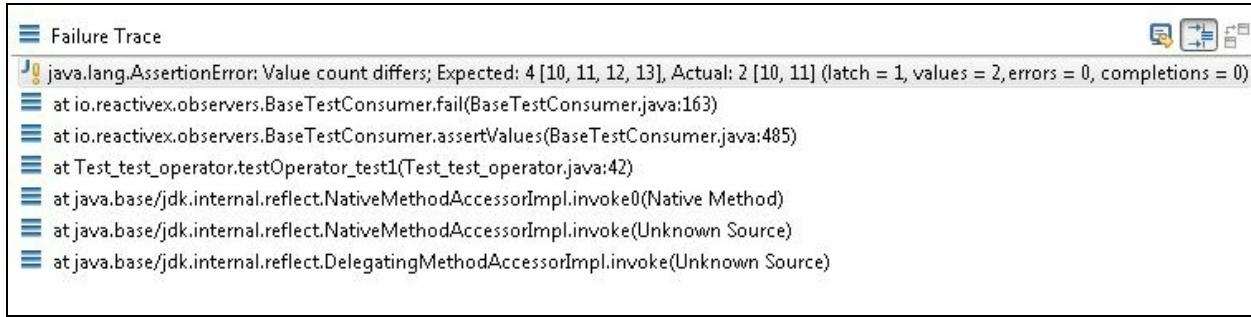
```

    @Test
    public void testOperator_test1() {
        TestSubscriber<Long> test_Subscriber =
            Flowable.rangeLong(10, 5).test(2);
        test_Subscriber.assertValues(101, 111);
        test_Subscriber.requestMore(2);
        test_Subscriber.assertValues(101, 111, 121, 131);
        test_Subscriber.requestMore(1);
        test_Subscriber.assertValues(101, 111, 121, 131, 141);
    }

```

The `requestMore()` method requests the number of items specified by the argument.

If we make the statement, `test_subscriber.requestMore(2)` of the code a comment, we will get the following failure trace suggesting that only two items have been requested, and unless we make any further request, no more items will be available:



```
java.lang.AssertionError: Value count differs; Expected: 4 [10, 11, 12, 13], Actual: 2 [10, 11] (latch = 1, values = 2, errors = 0, completions = 0)
at io.reactivex.observers.BaseTestConsumer.fail(BaseTestConsumer.java:163)
at io.reactivex.observers.BaseTestConsumer.assertValues(BaseTestConsumer.java:485)
at Test_test_operator.testOperator_test1(Test_test_operator.java:42)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
```

The method `awaitDone()` is used for blocking the terminal event as shown here:

```
@Test
public void test_asyn() {
    Flowable.just(1).subscribeOn(Schedulers.trampoline()).test()
        .awaitDone(5, TimeUnit.SECONDS).assertResult(1);
}
```

Testing notifications

The `Observable` emits an item, an error, or a notification to `Observer`. Testing of items can be done by checking the final result of the subscription using `TestObserver` or `TestSubscriber` with methods such as `assertValue()` or `assertResult()`, which we have just discussed. The RxJava library also provides methods for testing the notifications sent by the `Observable` as given in the following table:

Name of the method	Description	Class providing the method
<code>assertNotSubscribed()</code>	This method asserts that the method <code>onSubscribe()</code> has not been invoked	<code>TestObserver</code>
<code>assertSubscribed()</code>	This method asserts that the method <code>onSubscribe()</code> will be invoked only once	<code>TestObserver</code>
<code>assertComplete()</code>	This method asserts that <code>TestObserver</code> or <code>TestSubscriber</code> has received exactly one <code>onComplete</code> event	<code>BaseTestConsumer</code>
<code>assertNoErrors()</code>	The <code>assertNoErrors()</code> method asserts that <code>TestObserver</code> or <code>TestSubscriber</code> has not received any <code>onError</code> event	<code>BaseTestConsumer</code>
<code>assertError()</code>	This method asserts that <code>TestObserver</code> or <code>TestSubscriber</code> has received exactly one <code>onError</code> event which will be an instance of the specified class	<code>BaseTestConsumer</code>
	This method asserts that	

assertNotComplete()

TestObserver or TestSubscriber has
not received an onComplete event

BaseTestConsumer

Let's update the code written earlier in the chapter for `observable` to test the whether the notifications are received or not.

Demonstration 1 - updating code for the just() operator

Update the code for the test case `test_just_new()` for the `onComplete` and error notifications as follows:

```
@Test
public void test_just_new() {
    Observable<Integer> observable = Observable.just(12, 34, 6);

    TestObserver<Integer> testObserver = new TestObserver<>();
    observable.subscribe(testObserver);

    List<Integer> list = new ArrayList();
    testObserver.assertComplete();
    testObserver.assertResult(12, 34, 6);
    testObserver.assertValueCount(3);
    testObserver.assertNoErrors();
    testObserver.assertValueAt(2,(value)-> {
        // TODO Auto-generated method stub
        return value==6;
    });
}
```

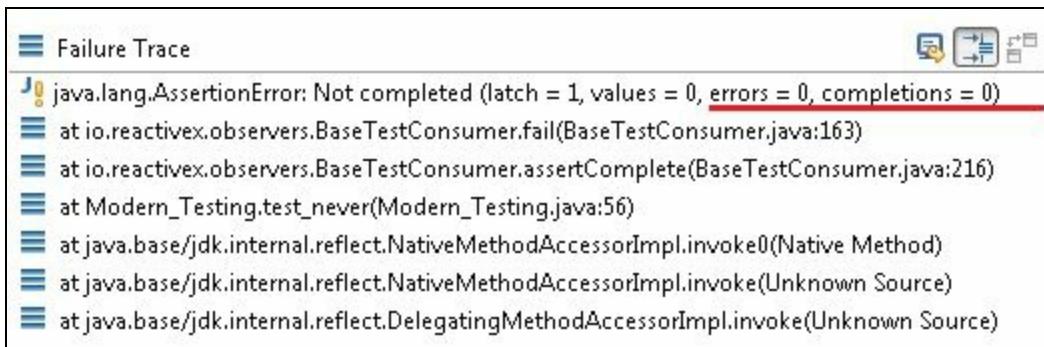
After executing the test case, we will pass successfully as `observable` completed its life cycle normally, and did not get any error notifications as no such notification is emitted.

Demonstration 2 - updating the code for the never() operator

Let's update the code for the operator `never()` by adding the following statement to find whether the observable emits an `onComplete` event or not:

```
@Test
public void test_never() {
    Observable<String> observable = Observable.never();
    TestObserver<String> testObserver = new TestObserver<>();
    observable.subscribe(testObserver);
    testObserver.assertNoValues();
    testObserver.assertComplete();
}
```

On execution, our test case will fail giving the following failure trace:



```
Failure Trace
java.lang.AssertionError: Not completed (latch = 1, values = 0, errors = 0, completions = 0)
    at io.reactivex.observers.BaseTestConsumer.fail(BaseTestConsumer.java:163)
    at io.reactivex.observers.BaseTestConsumer.assertComplete(BaseTestConsumer.java:216)
    at Modern_Testing.test_never(Modern_Testing.java:56)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
```

The output shows there is no `onComplete` event received, and that's why the test failed.

Demonstration 3 - updating the code for Flowable

We have created a `Flowable` using the `just()` operator. Let's update the code to test for its normal completion without any errors:

```
@Test
public void test_just_Flowable() {

    Flowable<String> observable =
        Flowable.just("mango", "papaya", "guava");
    TestSubscriber<String> testSubscriber = new TestSubscriber<>();

    observable.subscribe(testSubscriber);

    List<String> items = testSubscriber.values();
    testSubscriber.assertComplete();
    testSubscriber.assertSubscribed();
    testSubscriber.assertNoErrors();
    testSubscriber.assertValueCount(3);
    testSubscriber.assertValues("mango", "papaya", "guava");
}
```

The test case will run successfully.

Demonstration 4 - updating the code for the error() operator

Let's add the assertion for testing whether the `onComplete` and `onError` signals have been sent to `observer` or not by updating the test case for the `error()` operator as shown next:

```
@Test
public void test_error() {
    Observable<String> observable = Observable.error(
        new Exception("We got an Exception"));
    TestObserver<String> testObserver = new TestObserver<>();

    observable.subscribe(testObserver);

    testObserver.assertError(Exception.class);
    testObserver.assertNotComplete();
    testObserver.assertErrorMessage("We got an Exception");
    testObserver.assertNotComplete();
    testObserver.assertFailure(exception -> {
        return exception instanceof Exception;
    });
}
```

The test case will pass successfully.

Demonstration 5 - updating the code for the interval() operator

We have done the testing for observable emitting items after a specified interval. Let's update the code for a test case for `interval()` to test if the `onComplete` and `onError` signals have been sent or not:

```
@Test
public void test_interval()
{
    TestScheduler testScheduler=new TestScheduler();
    Observable<Long>observable=observable.interval(1,
        TimeUnit.SECONDS,testScheduler).take(5);
    TestObserver<Long> testObserver=new TestObserver<>();

    observable.subscribeOn(testScheduler).
    subscribe(testObserver);
    testObserver.assertNoValues();
    testObserver.assertNotComplete();
    testObserver.assertNoErrors();
    testScheduler.advanceTimeBy(1, TimeUnit.SECONDS);
    testObserver.assertValueCount(1);
    testObserver.assertValues(0L);
    testObserver.assertNotComplete();
    testScheduler.advanceTimeTo(6, TimeUnit.SECONDS);
    testObserver.assertValueCount(5);
    testObserver.assertValues(0L, 1L, 2L, 3L, 4L);
}
```

The test case will pass, as it got terminated normally, and also no `onError` signal has been received. You can update the rest of the code used in the chapter to test for the notifications.

Mockito testing

Mockito is an open source testing framework developed for testing Java-based applications and was released under the MIT License. Usually, in unit testing, we consider the unit or function to be dependent on the environment. And what if, maybe for some reason, the environment won't be available?

Consider that we are developing a web application, and want to test the `doGet()` or `doPost()` methods. We cannot test them using JUnit, as the request and response objects are uninitialized. And these objects cannot be initialized by us. We need to deploy the application on the server, and only then will the testing be possible. What if we were able to create dummy objects for request and response?

The Mockito framework enables the developers to create mock objects for test-driven development where the objects under testing are isolated from the framework. The objects are mock objects. The framework uses the Java Reflection API to create mock objects and has simple APIs to write test cases. Along with those, the framework also enables the developers to have a check on the order in which the methods are invoked.

Mockito has a static `mock()` method which can be used to create mock objects. It also facilitates the creation of mock objects using the `@Mock` annotation. The `MockitoJUnitRunner` is the custom runner that is used by JUnit.

Mockito works on the principle of returning predefined values. We will invoke the function under testing in the `when()` method with the arguments required by the function. Mock objects are fake objects, which means that the methods are also fake. So, depending on the return type of the method, we will define the set of values as an argument of Mockito using the method `thenXXX()`. The following are the methods that are used to specify what values are to be returned:

- `thenReturn()`: This is the method used to return a specified value
- `thenThrow()`: This is the method that throws a specified exception
- `then()`: This method returns an answer by the user-defined code

- `thenAnswer()`: This method is the same as the `then()` method
- `thenCallRealMethod()`: This method gives a call to the real method

We can use the following steps to carry out Mock testing:

1. The first task is to initialize all the dependencies using the mock object for the class under test.
2. Now, execute the function to test.

It's time to write the test conditions to check whether the operation gives the expected results or not.

Before moving ahead, we need to download a jar, `mokito-all-1.10.19.jar`, to support Mockito testing. Also, for this particular demo, we will be using JDK 8. Let's use Mockito to create a mock object for `Observable`, and write the test for it as shown next.

1. First, let's create a class, `MyOpeartor`, which has unimplemented methods shown as follows:

```
public class MyOpeartor {
    public Observable<int[]> createObservable() {
        return null;
    }

    public Single<String> getSingleValue() {
        return null;
    }
}
```

2. Now it's time to create the test class `Test_Mokito`.
3. Declare data members in the test class whose mock object will be created by the framework using the annotation `@Mock` as follows:

```
@Mock
MyOpeartor myOperator;
```

4. `MokitoRule` is an interface which facilitates keeping the tests clean. It also facilitates initialization of the mocks, and validation, and detection of the created stubs. Let's update the class by adding `MokitoRule` like this:

```
| @Rule  
| public MockitoRule rule = MockitoJUnit.rule();
```

5. It's time to write the test case for the function `getSingleValue()`. We will write the code to invoke the `getSingleValue()` method from `Mockito.when()`, and using `thenReturn()` will return our predefined values:

```
| @Test  
| public void test_single() {  
|     Mockito.when(myOperator.getSingleValue()).  
|         thenReturn(Single.just("MyValue"));  
|     TestObserver<String> testObserver =  
|         myOperator.getSingleValue().test();  
|     testObserver.awaitTerminalEvent();  
|     testObserver.assertValue("MyValue");  
| }
```

Though the actual implementation of `getSingleValue()` is not yet completed, our test will pass successfully, as we are returning dummy values from the `thenReturn()` method.

6. Let's add the test for `createObservable()` as follows:

```
| @Test  
| public void test_fromArray() {  
|     int[] arr = { 12, 34, 54, 6, 7 };  
|     Mockito.when(myOperator.createObservable()).  
|         thenReturn(Observable.fromArray(arr));  
|  
|     TestObserver<int[]> testObserver =  
|         myOperator.createObservable().test();  
|  
|     testObserver.awaitTerminalEvent();  
|     testObserver.assertValue(arr);  
| }
```

This test will also pass, suggesting that all the assertions are against the correct values.

Summary

The `Observable`, `Subscriber`, and operators created earlier just displayed some message or contents on the console. However, we cannot have a console each time, and in this book we are not talking about console-based applications, it's about reactive programming which can be used for web applications as well. Once we finish the code, we need to have the assurance that it works correctly. This is what testing is all about. Testing helps developers to be sure of the code they develop. We develop `Observable`, `Observer`, and operators as part of asynchronous applications. So, the traditional unit testing is not sufficient. The RxJava library provides `TestSubscriber` and `TestObserver` for testing the data emitted by the `Observable` using different operators such as `just()`, and `fromArray()`. We also discussed `TestScheduler` which helps in testing the operators, which allows creating time-based observables. The `Observable` not only emits data, but along with that, our `Observable` also emits errors and notifications. We used assertion methods such as `assertErrorMessage()` and `assertFailureAndMessage()` for testing the `Observable`. The methods such as `assertComplete()`, `assertError()`, and `assertNoError()` are used to test the notifications sent by the `Observable`.

Throughout the chapter, we used JUnit testing which enables testing of the code which is either completely implemented, or which does not have any dependency such as application deployment to the server or database connectivity. A code with such a dependency can be tested using the Mockito framework. We discussed the working of the Mockito framework, and how to inject the dependency. We also tested unimplemented methods such as `getSingleValue()` and `createObservable()`.

In the next chapter, we will discuss building a web application in the reactive style. We will use the support of Spring Framework 5 for Reactive Programming to build and discuss `Reactor`, a reactive library from Spring. So keep reading, it's time to experience the change in the application development using Reactive Programming.

Spring Reactive Web

We started our journey with a discussion of traditional application development, and the need for reactive programming in today's world. We covered the basics of reactive application architecture along with components of Flow API which was recently added to Java 9. Instead of sticking to it, we also discovered the reactive world using the RxJava library provided by ReactiveX. The point is, we are now at a stage where the acquired knowledge has to be used in real-time application development. In enterprise application development, we use frameworks for faster development, as the frameworks provide the basis for complex enterprise development. Spring 5.0 has been recently released in the market, and it provides an extensive support for Reactive Programming. In this chapter, we will discuss the support and use of reactive programming by the Spring Framework with the help of the following points:

- Spring WebFlux on server
- Spring WebFlux on client
- Deploying a reactive application
- Server-Sent Events

An introduction to Spring Framework

Rod Johnson, an Australian computer specialist and co-founder of SpringSource, laid down the basic infrastructure of the Spring framework; the Java enterprise application intends to provide solutions to the enterprise problem. The development is a very complex and time-consuming process. The Java enterprise application consists of various modules for handling services, business logic, and databases in separate layers. The application code is distributed among various layers, increases the level of complexity and maintainability. Today's market is very demanding, the developers need to follow a strict timeline. Developing the application from scratch takes more time in development as each and everything needs to be managed one by one. Spring provides an easy solution for developing enterprise applications by providing the basic architecture, services, and facilities.

The framework addresses problems such as scalability, plumbing code, boilerplate code, unavoidable non-functional code along with unit testing, as discussed further.

Plumbing code

Every Java application handles exceptions using the `try...catch` blocks; we used methods such as `commit()` and `rollback()` while handling database transactions and `connect()` for opening the communication on the network. Along with these codes, there are many more situations where we keep on writing unavoidable code in our programs, such as the code used to facilitate communication between the application and its underlying layers called **plumbing code**. It's an unavoidable part of the application without which the application would not be able to work as per requirements. However, such plumbing code increases the length of the code as well as makes debugging complex.

Scalability

Today's world is growing at a lightning speed; this growth is also seen in software and hardware. Research enables the fulfillment of the demanding market with changes in hardware. The rule is also applicable for software-- developers have to keep on updating applications to meet the increasing market demands. Whatever the application we develop today, it should be capable of withstanding the upcoming demands and growth without affecting the working application tomorrow. An application needs to be scalable to support the increased load of work to adapt to the growing environment instead of being replaced or failing when the demand increases.

Boilerplate code

The lines of statements that developers write in a number of functions throughout the application with little or no modification to carry out a task is called **boilerplate code**. Similar to plumbing code, the boilerplate code also makes the development code unnecessarily lengthy and complex.

Unit testing of the application

In [Chapter 8, *Testing*](#), we discussed unit testing of code and the problems faced by developers. Usually, most of the enterprise Java applications consist of many classes which are interdependent. The dependency exists in the objects throughout the application making it difficult to carry out testing.

Features of the Spring Framework

Spring addresses all these problems in application development with the following features:

POJO-based development

The class is a very basic structure of any application where the developer does the coding. These classes have different methods, and may extend a class or implement an interface of the framework. Reusing them becomes difficult as they are tightly coupled with the API. **Plain Old Java Object (POJO)** is a very famous terminology in Java application development. The Spring Framework has POJO-based development, where the code that we develop is not necessarily dependent on Spring APIs. These POJOs support loosely coupled modules which are reusable and easy to test.

Loose coupling through Dependency Injection (DI)

The term, coupling, refers to the degree of knowledge one class has about another class. Application developers try to write classes that are loosely coupled, specifying that a class is less dependent on the design of any other class. Interface-based programming allows developers to achieve loose coupling. The Spring Framework enables developers to keep the dependencies of the class separated from the code in a separate configuration file. We can also use interfaces and dependency injection techniques provided by Spring, that allow the developers to write loosely coupled code. As the classes are loosely coupled, the developers don't need to change the code frequently, making the application more flexible and maintainable.

Declarative programming

The term, declarative programming, suggests that the code will state what it is going to perform, but not how to perform. The Spring Framework supports declarative programming using XML-based configurations as well as annotation-based programming. Using Spring Framework, developers can keep all the configurations in XML from where it can be used by the framework to maintain the lifecycle of a bean. However, Spring 2.0 onward versions facilitate the use of a wide range of annotations.

Boilerplate code reduction using aspects and templates

We have just discussed boilerplate code, which is essential and without which the developers can't write code for transactions, security, logging, and so on. The framework gives a solution for writing code. Framework uses Aspect which helps in the reduction of boilerplate code. The framework also provides various templates such as `JdbcTemplate`, and `HibernateTemplate` for different requirements ensuring reduction of boilerplate code.

The Spring architecture

Spring provides more than 20 different modules that can broadly be summarized under seven main modules. These are listed as follows:

Core modules

Spring provides the core modules for creating Spring beans and injecting dependencies, maintaining the lifecycle of the bean in the context of the following modules:

- Core
- Context
- Beans
- SpEL

Data access and integration modules

The following are the modules provided by Spring enabling the integration of database, **Object Relational Mapping (ORM)**, in the application:

- JDBC
- ORM
- OXM
- JMS
- Transaction

Web MVC and remoting modules

The following are the modules offered by Spring to handle Web MVC and remoting in the application:

- Web
- Servlet
- WebSocket

AOP modules

The following are the modules in the Spring framework that facilitate handling of Aspect-oriented programming:

- AOP
- Aspect

Instrumentation modules

The following modules facilitate the use of Instrumentation in the application:

- Instrumentation
- Instrument Tomcat
- Messaging

Test modules

The framework facilitates the support for unit as well as integration testing with JUnit and TestNG. It also provides support for creating mock objects to simplify testing in an isolated environment.

Spring 5.0 introduces support for reactive streams in the application to help write applications with non-blocking calls using Spring Reactor, which has adopted Project Reactor and ReactiveX as well. We have already discussed the RxJava library in ReactiveX. Now let's discuss Project Reactor before moving ahead.

Project Reactor

Reactor is a framework developed for asynchronous programming by Pivotal as an implementation of Reactive Streams. It supports the writing of high-performance applications and works asynchronously using the event-driven programming paradigm. The ability to provide backpressure for asynchronous stream processing is its key feature. The Reactor framework can also be considered as the base library to develop an asynchronous application, and it's not a runtime environment. Reactor is based on the design pattern where the services received from the clients are distributed to different event handlers where their processing will be done.

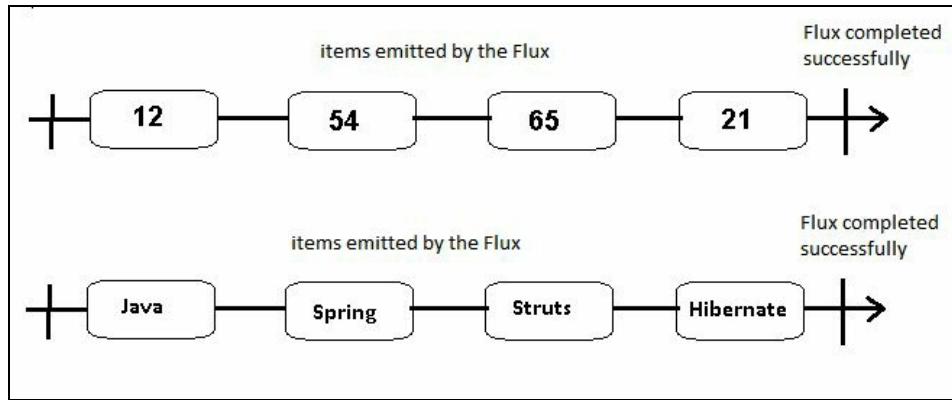
In RxJava, we used `Observable`, which is the source of the data, and `Observer`, which is the consumer of data. We have the same concepts as publisher and subscriber in the Reactor library as well. The `Publisher` is called `Flux`. `Flux` is as an event publisher that contains n number of events. If we look at the internals of `Flux`, it implements the `Publisher` interface. `Flux` publishes a sequence of events of POJO type. Consider the following code:

```
|     Flux<Integer> flux = Flux.just(12, 54, 65, 21);
```

For `String` type it can be like this:

```
|     Flux<String> flux= Flux.just("Java", "Spring", "Struts", "Hibernate");
```

In the code, we have declared `flux` of type `Flux` which is the publisher of the items on which the developers can now perform various operations. The generated `Flux` can be considered as shown in the following figure:



The preceding figure shows that `Flux` has emitted four events of type `Integer` and `String` using the `just()` method. The `just()` operator is not the only factory method we have. Along with `just()`, the following are the factory methods for creation of `Flux`:

- Empty `Flux` creation is as follows:

```

Flux<Integer> generateEmptyFlux() {
    return Flux.empty();
}

```

- `Flux` with a few elements in it is written as follows:

```

Flux<Long> generateFlux_Values() {
    return Flux.just(23L, 54L, 98L);
}

```

- Creating `Flux` from `java.util.List` is done as follows:

```

Flux<String> generateFlux_List() {
    List list=new ArrayList();
    // insert a code here to add objects to the ArrayList
    return Flux.fromIterable(list);
}

```

- Sometimes, while processing a `Flux`, an error may occur; the following code shows `Flux` with an error:

```

Flux<String> error_Flux(){
    return Flux.error(new RuntimeException());
}

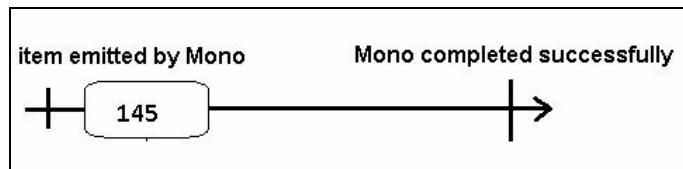
```

`Flux` has multiple elements in it. However, what if we don't want multiple elements, and we just need either 0 or 1 element? Here, `Flux` does not fulfill

our requirements; we may need something much more than it. `Mono` has been introduced to facilitate such a scenario. The `Mono` is a publisher that emits either 0 or a single item. The `Mono<Void>` denotes an empty sequence. Let's now consider the following code that will create a `Mono` having a single event of type `Long`:

```
Mono<Long> generate_Mono_WithValue() {  
    return Mono.just(145L);  
}
```

We can represent `Mono` as follows:



We used the `just()` operator to create `Mono`; let's discuss other factory methods facilitating the creation of the `Mono` one by one:

- Creating an empty `Mono` is done as follows:

```
Mono<String> generate_Empty_Mono(){  
    return Mono.empty();  
}
```

- Creating a `Mono` with an event is done like this:

```
Mono<Integer> generate_Mono_with_values (){  
    return Mono.just(100);  
}
```

- Sometimes, we need to create `Mono` that will not send any signal as shown by this code:

```
Mono<String> generate_Mono_Without_Signal(){  
    return Mono.never();  
}
```

- As a `Mono` can generate an event, it may also emit an error as shown by this code:

```
Mono<String> error_Mono(){  
    return Mono.error(new RuntimeException());
```

```
|     }
```

Along with the discussed scenarios, we also combine the generated `Mono` and `Flux` with each other to modify their behavior, and later on, we can add a subscriber to subscribe it.



Flux may contain 0 or n events, or stream sequences, while Mono may contain 0 or 1 events.

Now, it's time to discuss the processing of the items emitted by `Mono` OR `Flux` to perform the task as discussed next:

- Converting `Mono` to lowercase is done as follows:

```
Mono<String> convertToLowerCase() {  
    Mono<String> mono_caps= Mono.just("JAVA");  
    return mono_caps.map(String::toLowerCase);  
}
```

- To achieve the behavior as non-blocking asynchronous streams, we can wrap `Mono` to `Flux` as follows:

```
Flux<String> toUpperCase() {  
    Flux<String> flux = Flux.just("JAVA", "SPRING", "STRUTS");  
    return flux.flatMap(fruits -> Mono.just(fruits.toLowerCase()));  
}
```

- We can also combine `Flux` and `Mono` together with either of the methods shown here:

`mergeWith()`: Consider the following code for the use of `mergeWith()`:

```
Flux<String> mergerStreams() {  
    Flux<String> flux1= Flux.just("JAVA", "SPRING", "STRUTS");  
    Flux<String> flux2= Flux.just("HIBERNATE");  
    return flux1.mergeWith(flux2);  
}
```

`concatWith()`: Consider the following code for the use of `concatWith()`:

```
Flux<Long> contactStreams() {  
    Flux<Long> flux1= Flux.just(100L, 200L, 300L);  
    Flux<Long> flux2= Flux.just(400);  
    return flux1.concatWith(flux2);  
}
```

```
|     }
```

To process the publisher, many methods are provided by the APIs to enable the operations to perform.

Operators

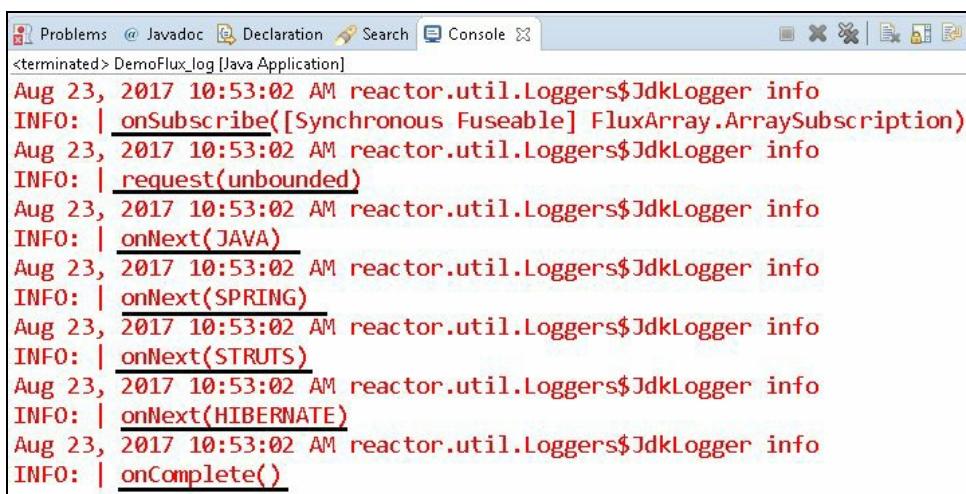
Operators are the functions provided by the library which facilitate the creation, transformation, or processing of `Flux` and `Mono`. Let's now discuss the operators facilitating the developers to manipulate `Flux` as follows:

- `log()`: The `log()` operator is used to log to the standard output as follows:

```
public class DemoFlux_log {
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Flux<String> flux = Flux.just("JAVA", "SPRING", "STRUTS",
            "HIBERNATE").log();
        flux.subscribe(new Consumer<String>() {
            @Override
            public void accept(String value) {
            }
        });
    }
}
```

Here we have added `Consumer` to subscribe the items, as we are well aware that unless there is a request from the `Subscriber`, our data source will not emit the items. Execute the code and observe the output:



```
Aug 23, 2017 10:53:02 AM reactor.util.Loggers$JdkLogger info
INFO: | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
Aug 23, 2017 10:53:02 AM reactor.util.Loggers$JdkLogger info
INFO: | request(unbounded)
Aug 23, 2017 10:53:02 AM reactor.util.Loggers$JdkLogger info
INFO: | onNext(JAVA)
Aug 23, 2017 10:53:02 AM reactor.util.Loggers$JdkLogger info
INFO: | onNext(SPRING)
Aug 23, 2017 10:53:02 AM reactor.util.Loggers$JdkLogger info
INFO: | onNext(STRUTS)
Aug 23, 2017 10:53:02 AM reactor.util.Loggers$JdkLogger info
INFO: | onNext(HIBERNATE)
Aug 23, 2017 10:53:02 AM reactor.util.Loggers$JdkLogger info
INFO: | onComplete()
```

The output clearly shows that we are able to get the event logged on the console. Now, let's update the `accept()` method by adding the

following statement:

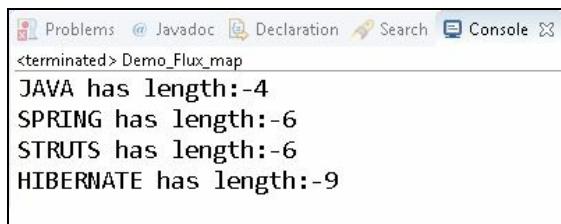
```
|     System.out.println(value);
```

Execute the code to get the values such as JAVA, SPRING, STRUTS, and HIBERNATE on the console.

- `map()`: The `map()` is another operator which is used to transform the streams, as shown here:

```
public class Demo_Flux_map {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Flux<String> flux = Flux.just("JAVA", "SPRING",
                                       "STRUTS", "HIBERNATE").
        map((item)->{
            if(item.length()>=0)
                return item+ " has length:-"+item.length();
            else
                return item+ "length:- cannot be calculated";
        });
        flux.subscribe(new Consumer<String>() {
            @Override
            public void accept(String value) {
                // TODO Auto-generated method stub
                System.out.println(value);
            }
        });
    }
}
```

Execute the code to the output on the console as seen in the following screenshot:



```
Problems @ Javadoc Declaration Search Console
<terminated> Demo_Flux_map
JAVA has length:-4
SPRING has length:-6
STRUTS has length:-6
HIBERNATE has length:-9
```

In the earlier chapters, we used `Observable`, `Single` or `Maybe` as reactive data types. Here, in both of the preceding demos, we used `Flux` instead of `Observable`. If you keenly observe, although we changed `Observable` to `Flux`, our basics of handling reactive programming remain the same.

The following operators are provided by the Reactor library to work with `Flux`:

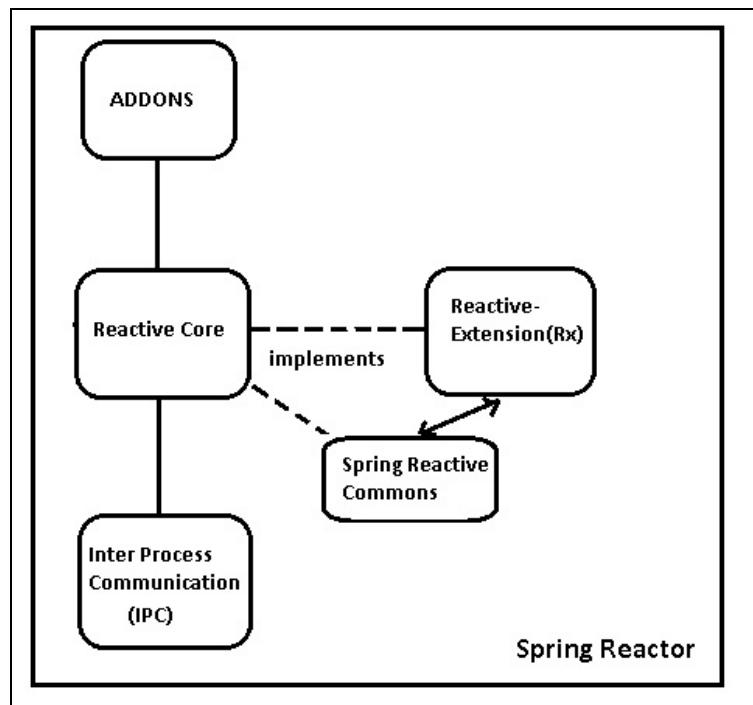
- `Flux<T> empty()`: This operator creates `Flux`, which gets completed without emitting any elements
- `Flux<T> concat()`: This operator facilitates the concatenation of all the sources that are pulled from `Iterator`, `Publisher`, or `Emitter`
- `Flux<T> create()`: This operator facilitates the creation of the `Flux` that has the capability of multiple emissions
- `Flux<T> delay()`: This operator signals to the `onNext()` method of the subscriber to delay `Flux` until the given period has not elapsed
- `Flux<Long> interval()`: This operator creates a `Flux` that emits data of type `Long` for the time period
- `Flux<T> just()`: This operator creates a new `Flux` that emits the specified number of items
- `Flux<T> merge()`: This operator facilitates the merging of the emitted publisher sequences

Similar to `Flux`, the following operators are provided for operating with `Mono`:

- `Mono<T> create()`: This creates a deferred emitter, that can be used with callback-based APIs
- `Mono<Tuple2<T, T2>> and()`: This combines the results from two `Mono`s in a `Tuple2` out of which one `Mono` is this
- `Mono<T> empty()`: This creates a `Mono` that will get completed without emitting any item
- `Mono<T> otherwise()`: This gets unsubscribed to a returned publisher when any error that matches the given type occurs

Spring web Reactive Programming

The Spring Reactive framework is based on the Reactor project along with which it also supports the RxJava library to implement functional Reactive Programming. Spring 5 is based upon the dependencies of Project Reactor 3.x. The Spring Framework 5 incorporates Reactive Streams, which has the contract for communicating among the asynchronous components between the components and the libraries along with a good support for backpressure. The collaboration made between the APIs of Project Reactor, Spring, and ReactiveX is shown in the following figure:



Let's look at each of these components one by one:

- **ADDONS:** The component ADDONS adds the support for RxJava1 and RxJava2. It enables the developers to use the reactive types such as `Observable`, `Flowable`, `Single`, and `scheduler` which we have already used in earlier chapters while discussing RxJava2.
- **Reactive Core:** Reactive Core is the main library that laid the foundation for non-blocking Reactive Streams. Reactive Core is the

implementation of **Reactive Extension (ReactiveX)**. It also provides support for the passing of the signals to and fro.

- **Inter Process Communication (IPC):** IPC supports encoding, sending, and decoding messages. Along with that, it also adds the support for Kafka and Netty.
- **Reactive Stream Commons:** Reactive Stream Commons is a research-based project started as a collaboration between Spring Reactor and Reactive Extension.

Server-side support

Spring 5 supports dealing with Reactive Programming with the addition of a new module named `spring-webflux`. It uses the Servlet 3.1 non-blocking I/O, which can be run on the Servlet 3.1 container. `ServletRequest` and `ServletResponse` expose `Flux<DataBuffer>` as the request and response body to allow reading and writing of streams. The Spring MVC API is defined in such a way that it is able to support asynchronous and non-blocking I/Os. Spring has the following two modules which add server-side WebFlux support:

- **Annotation based support:** The annotations facilitated by Spring Web MVC such as `@Controller`, `@GetMapping`, and `@PostMapping` are now supported to handle reactive types
- **Lambda expression style:** Functional Java 8 lambda expression style for routing and handling requests is supported for handling the reactive types

Annotation-based support

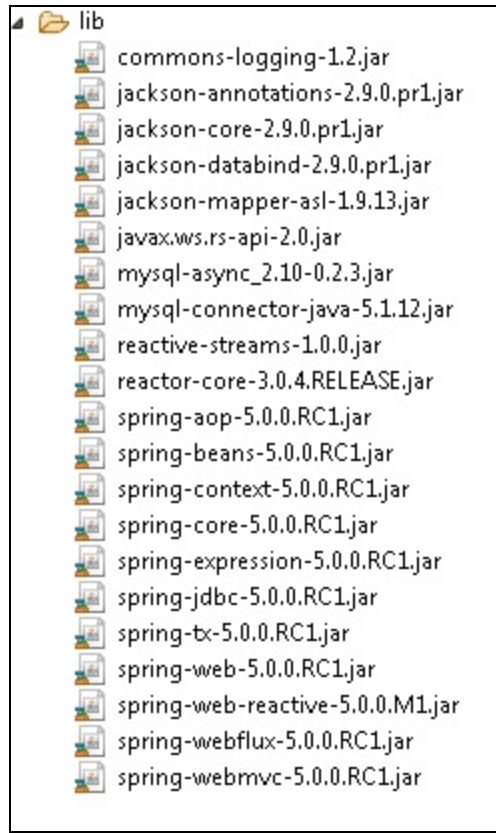
WebFlux supports annotation from the Spring Web MVC module with one major difference in the contract. Now, `HandlerMapping` and `HandlerAdapter` are non-blocking and operate on reactive `ServletRequest` and `ServletResponse` instead of `HttpServletRequest` and `HttpServletResponse`.

We usually return `java.util.List`, `Object`, `String`, or primitives from the controllers in Spring. Now, as Spring supports Reactive Programming, we need to know how to add reactive data types in the request or response. We can use the reactive types as discussed in the following scenarios:

- `Mono<Employee> employee`: This controller can use `Mono` after the employee is deserialized
- `Single<Employee> employee`: This is similar to `Mono` in the RxJava style
- `Flux<Employee> employee`: This is the input stream that accepts the `Flux`
- `Observable<Employee> employees`: This style suggests that the input is as per the RxJava style
- `Employee employee`: This `Employee` is deserialized without blocking before the controller is invoked

Let's develop an annotation-based Spring Web MVC application step by step as follows:

1. Create `ch09_spring_Reactive_Web` as a dynamic web application.
2. Add the jars in the `lib` folder as shown in the following screenshot:



3. Create Employee as a class in the package com.packt.ch09.pojo as shown by this code:

```
public class Employee implements Serializable {
    private String employeeFName;
    private long emp_id;
    private String address;
    private String mobileNumber;
    private String employeeLName;

    public Employee() {
        // TODO Auto-generated constructor stub
        employeeFName = "no name";
        employeeLName = "no l name";
        emp_id = 0;
        address = "Maharashtra";
        mobileNumber = "1234";
    }

    public Employee(String employeeFName, long emp_id,
                    String address, String mobileNumber, String employeeLName)
    {
        this.employeeFName = employeeFName;
        this.emp_id = emp_id;
        this.address = address;
        this.mobileNumber = mobileNumber;
        this.employeeLName = employeeLName;
    }
}
```

```

    }

    //add getters and setters to all the data members
    @Override
    public String toString() {
        return "Employee [employeeFName=" + employeeFName + ", "
            + emp_id=" + emp_id + ", address=" + address
            + ", mobileNumber=" + mobileNumber + "]";
    }
}

```

4. Now let's add the Front Controller mapping in the `web.xml` file.

```

<servlet>
    <servlet-name>employees</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <async-supported>true</async-supported>
</servlet>
<servlet-mapping>
    <servlet-name>employees</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>

```

Along with the servlet mapping, you can observe that we have added one more tag as `<async-supported>`, which enables handling of asynchronous requests to the application.

5. Now it's time to create a controller. Create `EmployeeController` as a Java class in the package `com.packt.ch09.controllers`. And we will add the following things in the class:

- The class to be annotated using `@RestController`, `@EnableWebMvc` to support the handling of RESTful web services
- Methods to be annotated by `@PostMapping`, `@GetMapping`, and `@DeleteMapping` on the methods to enable mapping of each incoming request to the method

The code for `EmployeeController` is as shown next:

```

@RestController
@EnableWebMvc
public class EmployeeController {
    @PostMapping("/employees")
    public ResponseEntity<Flux<Employee>>
        addEmployee(@RequestBody Mono<Employee> employee)
    {
        if (employee == null) {

```

```

        return
        new
        ResponseEntity<Flux<Employee>>(HttpStatus.NOT_FOUND);
    }
    return
    new
    ResponseEntity<Flux<Employee>>(HttpStatus.NOT_FOUND);
}

@PutMapping("/employees/{emp_id}")
public ResponseEntity<Flux<Employee>>
    updateEmployee(@PathVariable long emp_id,
    @RequestBody Mono<Employee> employee) {
    return new ResponseEntity(employee.flux(), HttpStatus.OK);
}

@GetMapping("/employees/{emp_id}")
public Mono<Employee> getBook(@PathVariable long emp_id)
{
    Employee employee = new Employee();
    employee.setEmp_id(emp_id);
    employee.setAddress("Pune, Maharashtra");
    employee.setEmployeeFName("Packt");
    employee.setEmployeeLName("Publication");
    return Mono.justOrEmpty(employee);
}
@GetMapping("/employees")
public Flux<List<Employee>> getAllBooks() {
    ArrayList<Employee> employees = new
        ArrayList<Employee>();
    employees.add(new Employee("Employee", 123, "Pune",
        "9890898777", "One"));
    employees.add(new Employee("Employee", 300, "Mumbai",
        "9890898700", "Two"));
    return Flux.just(employees);
}

@RequestMapping("/employeess/employee")
public Flux<Employee> findBook() {
    Employee employee = new Employee("Employee", 123, "Pune",
        "9890898777", "One");
    return Flux.interval(Duration.ofMillis(100))
        .map(l -> new Employee("Employee", 300, "Mumbai",
        "9890898700", "Two"));
}
@DeleteMapping("/employees/{emp_id}")
public ResponseEntity<Employee> deleteBook(
    @PathVariable long emp_id)
{
    return new ResponseEntity(new Employee(), HttpStatus.OK);
}
}

```

You may have observed that the method arguments have been annotated by `@PathVariable`. The annotation `@PathVariable` enables the

mapping of a request parameter to the argument without reading it manually from the request object.

6. Now it's time to add the mapping to find the controller we have written. To add the mapping, create a file, `employee-servlet.xml`, under the `WEB-INF` folder as follows:

```
| <context:component-scan  
|   base-package= "com.packt.ch09.*">  
| </context:component-scan>
```

Along with this, we need to enable annotation-based mapping, which can be done by adding the following configuration in the same file:

```
| <context:annotation-config />
```

7. The data will be navigated through the network in the form of a JSON object. Let's add a bean for converting the data as follows:

```
| <bean id="jsonMessageConverter" class=  
|   "org.springframework.http.converter.json.  
|   MappingJackson2HttpMessageConverter">  
| </bean>
```

The functional programming model

The Spring 5.0 framework introduces `HandlerFunction` and `RouterFunction` in the `org.springframework.web.reactive.function.server` package.

HandlerFunction

The incoming requests to the controller are handled by `HandlerFunction`. The `HandlerFunction` is a function that accepts an argument of type `ServerRequest` and returns `Mono<ServerResponse>`. The equivalent function as that of `HandlerFunction` is the one which has been annotated by `@RequestMapping`.

Immutable interfaces such as `ServerRequest` and `ServerResponse` offer JDK 8-friendly access to the underlying HTTP messages. Both these interfaces are fully reactive and are built on top of Reactor, which exposes the request with the body as `Flux` or `Mono`, and the response accepts `Publisher` as the body.

ServerRequest

`ServerRequest` facilitates access to the HTTP request elements such as method URI, query parameters, and headers. We can access the body methods as shown here:

```
|     Mono<String> mono_string = request.bodyToMono(String.class);
```

The preceding statement extracts the request body into an instance of `Mono<String>`.

We may need to extract body into `Flux` instead of extracting into `Mono` as shown by this statement:

```
|     Flux<Employee> flux_employee = request.bodyToFlux(Employee.class);
```

Spring provides the `BodyExtractor` interface as a functional strategy that enables us to write our own extraction logic. The `BodyExtractor` instances are found in the utility class `BodyExtractors`. Now, we can replace the previous statements by this code:

```
Mono<String> mono_string =
    request.body(BodyExtractors.toMono(String.class));
Flux<Employee> flux_employee =
    request.body(BodyExtractors.toFlux(Employee.class));
```

ServerResponse

We can access the HTTP response using `ServerResponse`, which can be created using a builder. The builder enables us to set the response properties such as response status, headers, and body. We can create `ServerResponse` for a JSON object, and another with an empty body, as shown next:

```
Mono<Employee> Employee = ServerResponse.ok().contentType  
(MediaType.APPLICATION_JSON).body(employee);
```

We can write `EmployeeHandler` which has three `HandlerFunctions` for returning all the `Employee` instances, adding a new `Employee` instance, and finding an `Employee` instance using `emp_id` as shown by the code that follows:

```
public class EmployeeHandler {  
    @Autowired  
    private final EmployeeDAO employeeDAO;  
    public Mono<ServerResponse> findAllEmployees(ServerRequest  
        request)  
    {  
        Flux<Employee> employee= employeeDAO().getAllEmployees();  
        return ServerResponse.ok().contentType(APPLICATION_JSON).  
            body(employee, Employee.class);  
    }  
  
    public Mono<ServerResponse> addEmployee(ServerRequest request)  
    {  
        Mono<Employee> employee= request.bodyToMono(Employee.class);  
        return ServerResponse.ok().build(employeeDAO.  
            addNewEmployee(employee));  
    }  
  
    public Mono<ServerResponse> findEmployeeByID(ServerRequest request)  
    {  
        int emp_id = Integer.valueOf(request.pathVariable("emp_id"));  
        Mono<ServerResponse> notavailable =  
            ServerResponse.notFound().build();  
        Mono<Employee> employee = employeeDAO.getEmployeeByID(emp_id);  
        return employee.then(employee ->  
            ServerResponse.ok().contentType(APPLICATION_JSON).  
            body(fromObject(employee))).otherwiseIfEmpty(notFound);  
    }  
}
```

RouterFunction

`RouterFunction` enables routing of the incoming request to the handler function which has a similar purpose as the `@RequestMapping` annotation in the `@Controller` classes. `RouterFunction` takes a `ServerRequest`, and returns a `Mono<HandlerFunction>`function. It returns a handler function whenever an incoming request matches a particular route, otherwise, it returns an empty `Mono`.

We use `RouterFunctions.route(RequestPredicate, HandlerFunction)` to create a `RouterFunction` instead of creating it from scratch. Whenever the `Predicate` matches, the request is routed to the handler function. In case the predicate doesn't match, outing doesn't take place, generating a `404` response. The framework also offers the `RequestPredicates` utility class which offers commonly used predicates for matching based on the path, HTTP method, content-type, and much more. Consider the following code:

```
RouterFunction<ServerResponse> routerFunction =  
    RouterFunctions.route(RequestPredicates.path("/spring_reactive"),  
    request -> Response.ok().body(fromObject("Welcome to Spring  
    Router")));
```

The last statement will route us to the `Welcome to Spring Router` handler function.

In order to route to the handler functions written in the preceding functions, we can have the router as follows:

```
EmployeeDAO employeeDAO =new EmployeeDAO();  
EmployeeHandler handler = new EmployeeHandler(employeeDAO);  
RouterFunction<ServerResponse> employeeRouter =  
    route(GET("/employee/{emp_id}").and(accept(APPLICATION_JSON)),  
        handler::findEmployeeByID)  
    .andRoute(GET("/employees").and(accept(APPLICATION_JSON)),  
        handler::findAllEmployees)  
    .andRoute(POST("/employee").and(contentType(APPLICATION_JSON)),  
        handler::addEmployee);
```

Client side

WebClient is a functional, reactive alternative to `RestTemplate` offered by `WebFlux`, that enables us to use fully non-blocking reactive programming. `WebClient`, which works over `HTTP/1.1` protocol, exposes `ClientHttpRequest` and `ClientHttpResponse` where the body of both, the request and response, is `Flux<DataBuffer>`. It also supports `JSON`, `XML`, and `SSE` serialization.

We can create `WebClient` and use the instance. Let's discuss the creation, preparing of a request, and getting the response using the instance step by step.

Creating a WebClient instance

We can create a WebClient instance in the following two ways:

- Using default settings, like this:

```
|   WebClient webClient = WebClient.create();
```

- Using a base URI, as follows:

```
|   WebClient webClient = WebClient.create("http://localhost:8080");
```

Using the WebClient instance to prepare a request

To prepare a request, you need to specify which HTTP method needs be invoked using the function method (`HttpMethod httpmethod`), as shown by the following code:

```
UriSpec<RequestBodySpec> uriSpec=
    webClient.method(HttpMethod.GET);
UriSpec<RequestBodySpec>
    uriSpec=webClient.post();
```

Now, we can specify the URI, set the request body, content type, and headers as well.

Let's set the request body using `Publisher` as shown here:

```
webClient.method(HttpMethod.POST).uri("/myURI").body
    (BodyInserters.fromPublisher(Mono.just("data")), String.class);
```

We can even use the `BodyInserter` interface which can be used for populating the `ReactiveMessageOutputMessage` body, which has an output message and a context, as follows:

```
webClient.post().uri(URI.create("/myURI")).body(
    BodyInserters.fromObject("display"));
```

In the same way, as we set the body, we can also set other properties as follows:

- Header (`HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE`)
- Accept (`MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML`)
- `acceptCharset(Charset.forName("UTF-8"))`

Once the request is set, it's time to get the response.

Reading the response

Once the request is sent and the resource is available, a response will be generated. So the final step is to read the response, which can be done using the `exchange()` or `retrieve()` method.

Let's use the gathered information to write the client code for the REST application we developed earlier. The code for the client will be as follows:

```
public class Test_WebClient {
    public static void main(String[] args) {
        WebClient webClient = WebClient.create(
            "http://localhost:8080");
        Employee employee_mono=webClient.get()
            .uri("/Ch09_Spring_Reactive_Web/employees/{emp_id}",14)
            .accept(MediaType.APPLICATION_JSON)
            .exchange()
            .then(response -> response.bodyToMono(Employee.class))
            .block();
        System.out.println(employee_mono);
    }
}
```

Before running the preceding code, we first need to deploy the application.

We have seen how the request body supports reactive types. Similarly, the response body can be in any one of the following ways:

- `Mono<Employee>`: This serialization will be completed without blocking the given employee when `Mono` is completed
- `Single<Employee>`: This serialization will be completed without blocking the given employee when the `Single` is completed; it uses RxJava
- `Observable<Employee>`: This is also a streaming scenario, but it uses the RxJava `Observable` type
- `Flowable<Employee>`: This is also a streaming scenario, but it uses the RxJava 2 `Flowable` type
- `Flux<ServerSentEvent>`: This is SSE streaming
- `Mono<Void>`: This is when `Mono` completes the request handling
- `Flux<Employee>`: This is the streaming scenario
- `Employee`: This serializes without blocking the given employee

- `Void`: This request handling will be completed when the method, which is a synchronous, non-blocking controller method, returns

Let's create a `WebClient` that will facilitate us querying the REST application as follows:

```
public class Test_WebClient {  
    public static void main(String[] args) {  
        WebClient webClient = WebClient.create("http://localhost:8080");  
        Employee employee_mono=webClient.get()  
            .uri("/Ch09_Spring_Reactive_Web/employees/{emp_id}",14)  
            .accept(MediaType.APPLICATION_JSON)  
            .exchange()  
            .then(response ->response.bodyToMono(Employee.class)).block();  
        System.out.println(employee_mono);  
    }  
}
```

Before executing the last code, we first need to deploy the application.

Deploying the application

To deploy the application, we can use any of the following ways:

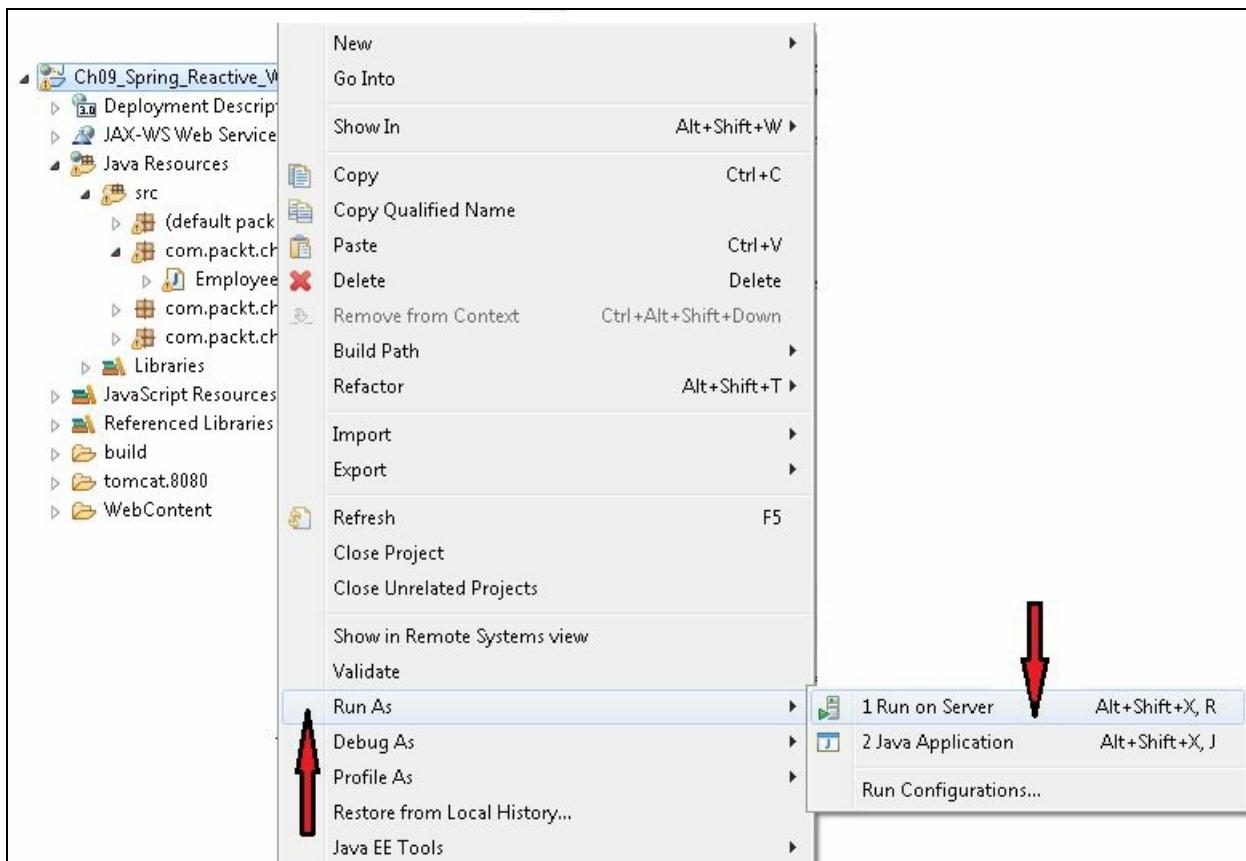
1. Deploying the application from Eclipse IDE.
2. Generating the `WAR` file and then deploying it on the server.

Let's use these ways one by one.

Deploying the application from Eclipse IDE

This is the simplest and easiest way of deploying and running the application. Perform the following steps:

1. Select the project from Project Explorer, and right-click on it. Now select the option Run As | Run On Server as shown in the following screenshot:

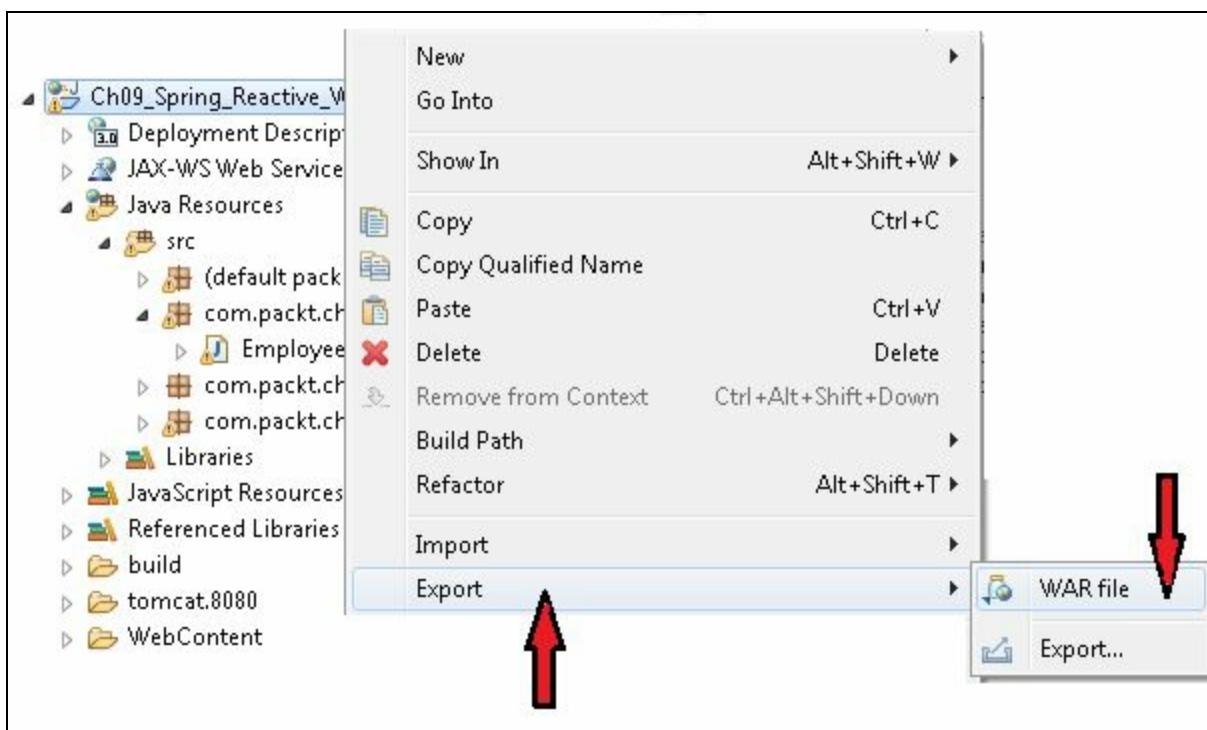


This step will create a `WAR` file and deploy it on the server. Now, we can use the browser for requesting our application.

Creating and deploying WAR file on server

Create the `WAR` file, and manually deploy it on the server as follows:

1. Select the project from Project Explorer, and select the Export option as shown in the next screenshot:



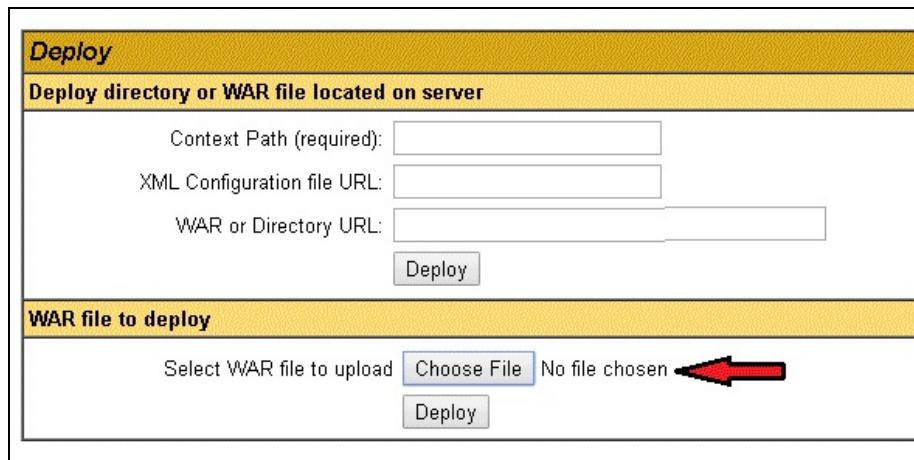
A dialogue box will appear which will ask you to select the destination. You can select any location. If you want, you can change the name of the `WAR` file as well. I am going to keep it the same as `Ch09_Spring_Reactive_Web.war`. Finally, click on the Finish button to complete the process.

2. After creating the `WAR` file, now we need to deploy it. Before deploying, make sure you have the following code available in the `conf/tomcat-user.xml` file in the Tomcat server:

```
<tomcat-users>
  <role rolename="manager-gui"/>
  <user username="mgr" password="mgr" roles="manager-gui"/>
</tomcat-users>
```

You can choose your own usernames and password keeping the role as `manager-gui`. If you edited the `.xml` file, don't forget to save it.

3. Launch the server by double-clicking on the `startup.bat` file from the `bin` directory of Tomcat.
4. Once the server starts, now open the browser and type the URL as `http://localhost:8080/manager`.
5. A dialogue box will open where we need to give the aforementioned configured credentials.
6. Once you have logged in successfully, it's time to deploy the application. Click on the Choose File button, and select the `.war` file that we created in an earlier step, as shown in the following screenshot:

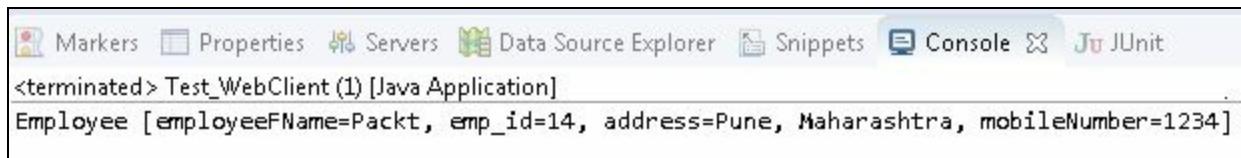


7. Click on the Deploy button to complete the deployment process.
8. Now an entry can be seen in the application table denoting the successful deployment of the application.

Testing the application

Now that everything is ready, let's run the application. You can either use external deployment or internal from the Eclipse IDE. Once the application is in the running mode from either of the ways, select the `Test_WebClient.java` file that we created, and run that as a Java application.

We will get the following output on the console:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:

```
<terminated> Test_WebClient (1) [Java Application]
Employee [employeeFName=Packt, emp_id=14, address=Pune, Maharashtra, mobileNumber=1234]
```

Working with WebTestClient

As we have `RestTemplate` for testing the REST applications, `WebTestClient` enables the testing WebFlux server endpoints, which is very much similar to `WebClient`. Client-side testing can be done with specific controllers or functions. It provides the `bindToServer()` method, which enables end-to-end integration tests with the actual requests to a running server. We can also use a specific router function using the `bindToRouterFunction()` method. The code will be as follows:

```
RouterFunction router_function = RouterFunctions.route(  
    RequestPredicates.GET("/employees"), request ->  
    ServerResponse.ok().build());  
WebTestClient.bindToRouterFunction(router_function)  
    .build().get().uri("/employee")  
    .exchange().expectStatus().isOk()  
    .expectBody().isEmpty();
```

Server-Sent Events (SSE)

The servers enable pushing of the data to the Web pages over HTTP, or they may use server-push protocols. The subclass of the `ResponseBodyEmitter` class, `SseEmitter`, supports working with SSE. It's actually another variation of the HTTP Streaming technique in which the pushed events from the server are formatted according to the W3C Server-Sent Events specification.

These events can be pushed from the server to the clients, which now can very easily be done in Spring MVC, and requires returning a value of type `SseEmitter`. This sending of events plays a vital role in online gaming applications, applications for collaboration, and financial applications. Spring's WebSocket support includes SockJS style WebSocket, a higher-level messaging pattern for interacting with clients.

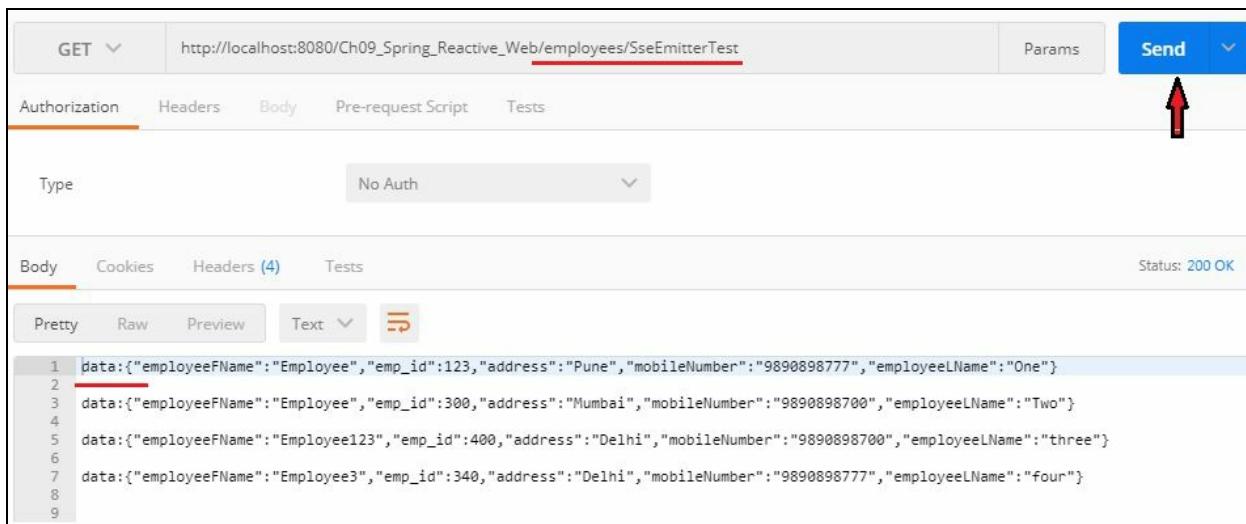
Let's update `EmployeeController` to add one more method that will now send the events to the browser. We will create a list of employees which we will send to the client along with the events using the `send()` method for `MediaType` as `APPLICATION_JSON`.

```
    @RequestMapping("/SseEmitterTest")
    public ResponseBodyEmitter handleSseEmitter() {
        ArrayList<Employee> employees = new ArrayList<Employee>();
        employees.add(new Employee("Employee", 123, "Pune",
            "9890898777", "One"));
        employees.add(new Employee("Employee", 300, "Mumbai",
            "9890898700", "Two"));
        employees.add(new Employee("Employee123", 400, "Delhi",
            "9890898700", "three"));
        employees.add(new Employee("Employee3", 340, "Delhi",
            "9890898777", "four"));
        final SseEmitter emitter = new SseEmitter();
        ExecutorService service = Executors.newSingleThreadExecutor();
        service.execute(() -> {
            for (Employee employee : employees) {
                try {
                    emitter.send(employee, MediaType.APPLICATION_JSON);
                    Thread.sleep(1000);
                }
                catch (Exception e) {
                    e.printStackTrace();
                    emitter.completeWithError(e);
                    return;
                }
            }
        });
    }
```

```
        }
    }
    emitter.complete();
});
return emitter;
}
```

Now run the application, and launch Postman from Google Chrome.

Enter the URL, select the method supported as shown in the following screenshot, and then click on the Send button. We will get the output in the JSON format as specified. Each entry in the JSON array has a key as `data`, which denotes that the server has sent data to the client. Consider the following screenshot:



GET http://localhost:8080/Ch09_Spring_Reactive_Web/employees/SseEmitterTest Send

Authorization Headers Body Pre-request Script Tests

Type: No Auth

Body Cookies Headers (4) Tests Status: 200 OK

Pretty Raw Preview Text

```
1 data:{"employeeFName":"Employee","emp_id":123,"address":"Pune","mobileNumber":"9890898777","employeeLName":"One"}  
2 data:{"employeeFName":"Employee","emp_id":300,"address":"Mumbai","mobileNumber":"9890898700","employeeLName":"Two"}  
3 data:{"employeeFName":"Employee123","emp_id":400,"address":"Delhi","mobileNumber":"9890898700","employeeLName":"three"}  
4 data:{"employeeFName":"Employee3","emp_id":340,"address":"Delhi","mobileNumber":"9890898777","employeeLName":"four"}
```



You can refer to https://www.getpostman.com/docs/postman/launching_postman/installation_and_updates to get further information and installation details about the Postman app.

Summary

Plumbing code, scalability, boilerplate code, and unit testing facilitate faster development of any Java enterprise applications. In this chapter, we took an overview of how the Spring Framework helps us to overcome these problems. Along with this, we also discussed the modules offered by the framework. We moved ahead and discussed the project Reactor developed by the Pivotal team to support reactive programming. We introduced `Flux` and `Mono`, which work as the sources of emission, emitting 0 to n or 0 to 1 items along with their operators. We also discussed how the Spring Project Reactor has adopted ReactiveX, Addons, Reactive Core, IPC, and Reactive Stream Commons. To understand the handling of server-side non-blocking API, we developed a REST application as well. We also discussed in depth about `HandlerFunction` and `RouterFunction`. For performing client-side operations, we discussed and demonstrated, in depth, about WebClient. Finally, we created a `WAR` file, and deployed and tested it using the Postman tool.

In the next chapter, we will demonstrate the usage of Hystrix, a latency and fault tolerance library from Netflix that uses RxJava.

Implementing Resiliency Patterns Using Hystrix

We usually talk about enterprise applications that provide sophisticated solutions to the problems faced by an enterprise. Actually, in today's market, we have distributed enterprise applications that are based on modular architecture. The modularity suggests that the entire application gets broken down into small maintainable modules that deal with very specific tasks. For more clarity, consider an online shopping application. The application deals with registering and authenticating the user, storing the products in the persistence layer, and processing the payment and shipment details. For better performance, the layers of the application can be deployed on multiple servers, and when the user requests it, it will be invoked over the network.

The distributed applications may have dependencies. Under normal conditions, these dependencies can be fetched successfully. However, sometimes, these dependencies may fail, causing the application to fail. Sometimes, after the request, one of the dependencies delays the data transfer. The delay by one of the dependencies causes the blocking of all other requested threads, and the resources may get saturated. Now, all the requests that are reaching this particular dependency will fail. The situation may get worse if we are performing the operations using third-party libraries.

Now, consider, we are trying to set up the connection over the network, the network fails or it is taking too long to get the connection back. Sometimes, we upgrade our systems to new libraries and then, to our surprise, our application stops working. This situation is discussed here; it is caused just because some of the dependency failed. Practically, a single dependency should not cause the failure of an entire application. The situation can be overcome by simply isolating the dependencies and the system. Yes, you heard me right. In such a scenario, we need Hystrix. In this chapter, we will discuss Hystrix in detail by covering the following topics:

- Hystrix and RxJava

- `HystrixCommand` and `HystrixObservableCommand`
- Resiliency pattern
- Circuit breaker
- Isolation
- Request collapsing
- Request cache

Hystrix- an introduction

Hystrix is a framework that enables developers to control the interactions between the distributed services, along with providing additional support for latency tolerance and fault tolerance. This is achieved by isolating the failing services, which, in turn, stop cascading the effect of failure, and provides the fallback options.

In 2011, the Netflix API team started working on the resiliency engineering, and, up until 2012, it has been adopted by many teams within Netflix. Today, billions of isolated calls are executed using Hystrix. Hystrix aims to provide the following facilities:

- It provides protection from, as well as control over, the latency and the failure which is usually caused when we try to access third-party dependencies
- It doesn't cascade the failure to the components in the distributed system
- It facilitates fast recovery from the failure
- It provides the facility for real-time monitoring, alerting, and operational control

How Hystrix works

The working of Hystrix is as follows:

1. **Creating an instance of command:** We need to create an instance of either `HystrixCommand` or `HystrixObservableCommand` to present the request that we are making to the dependency. These instances will be working as a wrapper around the dependency that we are requesting.
 - The instance of `HystrixCommand` is used to wrap a request that will execute the functionality or service over the network with fault and latency tolerance with a blocking call. However, the `observe()` method returns a non-blocking `Observable`. We prefer the instance whenever a single response is expected from the dependency. The following lines of code demonstrate the creation of an instance of `HystrixCommand`:

```
| HystrixCommand hystcmd = new  
|   HystrixCommand(argument1, argument2);
```

The arguments passed to the constructor will be required when the request is made.

- The `HystrixObservableCommand` instance is used for the asynchronous execution of the command when the dependency is expected to return an `Observable` type that will emit a sequence of response. We can create the instance as shown here:

```
| HystrixObservableCommand hystCmdObservable =  
|   new HystrixObservableCommand(argument);
```

2. **Executing the command:** Once the instance of command is available, the library provides four different ways that can be used to execute the command, as discussed next:

- Executing the `HystrixCommand` command:
The `HystrixCommand` provides the following two methods to execute the command:
 - The `execute()` method makes a blocking, synchronous

execution of the command, which can return either a single response or exception if something goes wrong.

- The `queue()` method facilitates the asynchronous execution of the command that returns `Future`. The single response can be retrieved from the returned `Future`.
- **Executing `observableHystrixCommand`:** The `observableHystrixCommand` provides the following two methods to execute the command:
 - `observe()`: This method is used for the asynchronous execution of the command. It returns an `observable` type that represents the response obtained from the dependency. It is used to obtain hot observables as it eagerly starts the execution of the command.
 - `toObservable()`: This method is used for the asynchronous execution of the command. It returns an `observable` type that represents the response obtained from the dependency. It is preferred for the lazy execution of the command and so, it can be used to get the cold `observable` type.

3. **Enabling caching:** Here, whether request caching is enabled or not, it is cross-checked. If the response to the request made is available in the cache, the available response will be returned to type `observable`.
4. **Open circuit:** In the earlier stage, if the response for the request was not available in the cache, Hystrix checked for the circuit breaker. If the circuit breaker is open, then Hystrix will not be able to execute the command, and the flow will be routed to get the fallback. If the circuit may be closed, then the flow will proceed further to check whether the capacity to run the command is available or not.
5. **Capacity to run the command:** Here, the capacity of the thread pool, queue, or semaphore is checked. If the thread pool or the queue associated with a command is full, then Hystrix will not be able to execute the command and the flow will route to get fallback.
6. **Invoking the request:** Now, Hystrix will invoke the request to the dependency by using one of the following methods:

- `HystrixCommand.run()`: This method either returns a single response or an exception.
- `HystrixObservableCommand.constructs()`: This method returns an `observable` type, which may emit either a response or an error. After

emitting the response, the `observable` type will emit an `onComplete` notification, as discussed in the earlier chapters.

In case the `run()` or `construct()` method takes more time than the `timeout` value, the `TimeoutException` will be thrown by the thread and the response will be routed to fallback.

7. **Obtaining the circuit health:** On each execution, Hystrix reports success, failure, rejection, or timeouts to the circuit breaker. The circuit breaker maintains a rolling set of the counters from which, the statistics can be calculated. The calculated statistics can now be used to determine when to keep the circuit open, close it, or to perform short-circuiting. Don't worry about the details now, in a short while, we will discuss circuit in depth.
8. **Fallback:** Consider the following situations:
 - An exception is thrown by either `construct()` or `run()`
 - The circuit is open and the command is short-circuited
 - Either the thread pool or the semaphore has reached capacity
 - The command has exceeded the timeout

Under all of the preceding scenarios, the command execution fails and Hystrix tries to revert the flow to the fallback.

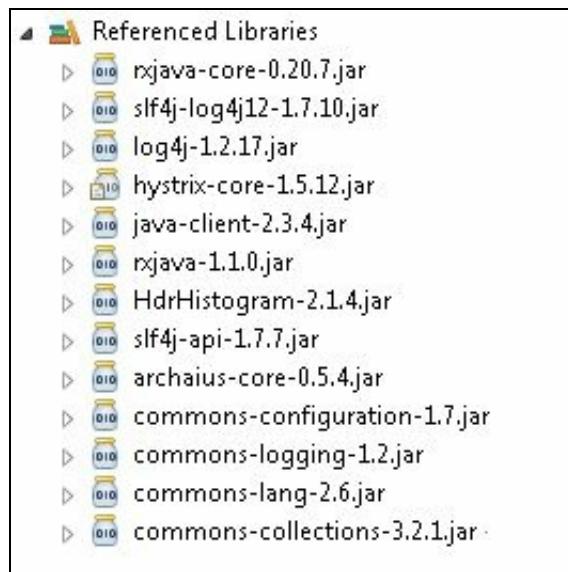
The logic to execute whenever the command is returned to fallback will be written either in the `HystrixCommand.getFallback()` or `HystrixObservableCommand.resumeWithFallback()` method. It means that the fallback will either return a response or an `observable` type that will emit the response. As we know, an exception can be thrown during the execution of the logic from the fallback. If fallback throws an exception, then Hystrix will return an empty `observable` type that emits nothing.

9. **Returning the response:** If everything goes well, Hystrix will return an `observable` type to the caller. Although the `observable` type has been returned, what the actual response type will be is dependent on the actual method (`execute()`, `queue()`, `observer()`, OR `toObservable()`), that we have used to invoke the command.

Demonstrating HystrixCommand

Let's demonstrate `HystrixCommand` to understand the working step by step:

1. Create a `ch10_Hystrix_Introduction` file as a Java project.
2. Add the JARs as shown in the following screenshot:



You can even add them using the Maven by adding the following dependencies:

```
<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
    <version>1.5.4</version>
</dependency>
<dependency>
    <groupId>com.netflix.rxjava</groupId>
    <artifactId>rxjava-core</artifactId>
    <version>0.20.7</version>
</dependency>
```

3. Now, create `DemoHystrixCommand` as a Java class that will extend from `HystrixCommand`.
4. Add `user` as a data member of type `String` in the class.
5. Also, add a parameterized constructor in the class. The class is shown

as follows:

```
public class DemoHystrixCommand extends HystrixCommand<String>
{
    private String user;
    public DemoHystrixCommand(String user) {
        // TODO Auto-generated constructor stub
        super(HystrixCommandGroupKey.Factory.asKey("packtGroup"));
        this.user=user;
    }
}
```

To group commands together for similar tasks, Hystrix uses the command group key. We can have the commands for reporting, alerting and dashboards grouped together. By default, Hystrix uses this to define the command thread pool.

`HystrixCommandGroupKey` is an interface. The interface has the helper `Factory` class that we are using here. We can also implement the interface for customization.

Here, we have set the key to `packtGroup`.

6. Now, override the `run()` method where we will write the logic for the network call. However, to understand, we will write only a `return` statement; later, you can replace it by adding the network call to the service or the repositories. The code is as follows:

```
@Override
protected String run() throws Exception {
    // TODO Auto-generated method stub
    return "Welcome to Hystrix, "+ user;
}
```

7. Our command class is ready for testing. Let's test it using the JUnit test case, as shown in the following piece of code:

```
public class TestDemoHystrixCommand {
    @Test()
    public void test_command() {
        HystrixCommand<String>command=new
            DemoHystrixCommand("Packt Pub");
        assertEquals("Welcome to Hystrix, Packt Pub",
            command.execute());
    }
    @Test
```

```
public void test_queue() {
    HystrixCommand<String> command = new
        DemoHystrixCommand("Packt Pub");
    try {
        assertEquals("Welcome to Hystrix, Packt Pub",
            command.queue().get());
    }
    catch (InterruptedException e) {
        // TODO Auto-generated catch block
        fail(e.getMessage());
    }
    catch (ExecutionException e) {
        // TODO Auto-generated catch block
        fail(e.getMessage());
    }
}
```

`test_command()` is checking the response obtained from the `execute()` method, which is a plain `String`. However, `command.queue()` returns `Future`. In the `test_queue()` method, we are using the `get()` method to obtain the response from the `Future`.

8. Run the test case and our test will pass successfully.

Demonstrating HystrixObservableCommand

Similar to the `HystrixCommand` implementation, we can also create an extension of `HystrixObservableCommand` to support reactive programming, as shown here:

```
public class DemoHystrixObservableCommand extends
    HystrixObservableCommand<String> {
    private String user;
    public DemoHystrixObservableCommand(String user) {
        super(HystrixCommandGroupKey.Factory.asKey("packtGroup"));
        this.user=user;
    }
    @Override
    protected Observable<String> construct() {
        // TODO Auto-generated method stub
        return Observable.just("Welcome to Hystrix, "+user);
    }
}
```

Here, the `construct()` method has `Observable` as the return type that facilitates reactive programming.

We can now test `HystrixObservableCommand` by creating the test case, as shown in the following code block:

```
public class TestDemoObservableHystrixCommand {
    @Test()
    public void test_observe() {
        HystrixObservableCommand<String>command=new
            DemoHystrixObservableCommand("Packt Pub");
        TestSubscriber<String > testSubscriber=new
            TestSubscriber<>();
        command.observe().subscribe(testSubscriber);

        List<String>list=new ArrayList<>();
        list.add("Welcome to Hystrix, Packt Pub");
        testSubscriber.assertReceivedOnNext(list);
    }
    @Test()
    public void test_toObservable() {
        HystrixObservableCommand<String>command=new
            DemoHystrixObservableCommand("Packt Pub");
        TestSubscriber<String > testSubscriber=new
            TestSubscriber<>();
        command.toObservable().subscribe(testSubscriber);
```

```
    List<String>list=new ArrayList<>();
    list.add("Welcome to Hystrix, Packt Pub");
    testSubscriber.assertReceivedOnNext(list);
}

@Test()
public void test_toObserve_blocking() {
    HystrixObservableCommand<String>command=new
        DemoHystrixObservableCommand("Packt Pub");
    Observable<String>observable=command.observe();
    assertEquals("Welcome to Hystrix, Packt Pub",
        observable.toBlockingObservable().single());
}
}
```

We already discussed testing `observable` using `TestSubscriber` in [Chapter 8, Testing](#), in detail; for more details, you can refer to it.

The test case will be executed successfully.

Handling fallback

We have just used the `execute()`, `queue()`, `observe()`, and `toobservable()` methods to execute the command object to request the dependencies. If we are using `HystrixCommand`, the `run()` method gets invoked, and, in case of `HystrixObservableCommand`, the `construct()` method will be invoked. Under normal conditions, the consumer will get the response without any problem. However, due to network problems, increase in a load of service, or timeout, or thread pool rejection, the method may throw an exception. When the exception is thrown to support graceful degradation, we can add the method that facilitates handling of fallback.

Demonstrating HystrixCommand with a fallback

Let's implement a fallback method for `HystrixCommand`. To understand the working mechanism, we are purposely adding the following piece of code for throwing an exception from the `run()` method:

```
public class DemoHystrixCommand_Fallback extends
    HystrixCommand<String>{
    private String user;
    public DemoHystrixCommand_Fallback(String user) {
        // TODO Auto-generated constructor stub
        super(HystrixCommandGroupKey.Factory.asKey("packtGroup"));
        this.user=user;
    }
    @Override
    protected String run() throws Exception {
        // TODO Auto-generated method stub
        throw new RuntimeException("Exception occurred");
    }
}
```

The `getFallback()` method handles the exception gracefully as and when the exception will occur. Let's override the method, as shown in the following code snippet:

```
@Override
protected String getFallback() {
    // TODO Auto-generated method stub
    return "Sorry!!! We failed";
}
```

Now it's time to test the code. Let's write the test case as follows:

```
@Test
public void test_command() {
    HystrixCommand<String> command=new
        DemoHystrixCommand_Fallback("Packt Pub");
    assertEquals("Sorry!!! We failed",command.execute());
}
@Test
public void test_queue() {
    HystrixCommand<String> command=new
        DemoHystrixCommand_Fallback("Packt Pub");
    try {

```

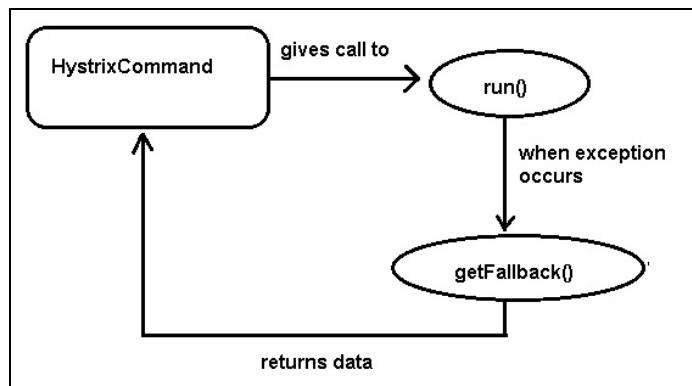
```

        assertEquals("Sorry!!! We failed", command.queue().get());
    }
    catch (InterruptedException e) {
        // TODO Auto-generated catch block
        fail(e.getMessage());
    }
    catch (ExecutionException e) {
        // TODO Auto-generated catch block
        fail(e.getMessage());
    }
}

```

You can easily observe that the `assertEquals()` method is comparing the values returned from the `getFallback()` method rather than the value returned from the `run()` method.

Executing the method, our test will successfully pass, as once the `run()` method throws an exception, the `getFallback()` handler method will be invoked to give the response as shown here:



Demonstrating HystrixObservableCommand with fallback

To handle fallback using `HystrixObservableCommand`, we need to override the `resumeWithFallback()` method instead of the `getFallback()` method, as shown in the following piece of code:

```
public class DemoHystrixObservableCommand_Fallback extends
    HystrixObservableCommand<String> {
    private String user;
    public DemoHystrixObservableCommand_Fallback(String user) {
        // TODO Auto-generated constructor stub
        super(HystrixCommandGroupKey.Factory.asKey("packtGroup"));
        this.user = user;
    }

    @Override
    protected Observable<String> construct() {
        // TODO Auto-generated method stub
        throw new RuntimeException("Got an exception");
    }

    @Override
    protected Observable<String> resumeWithFallback() {
        // TODO Auto-generated method stub
        return Observable.just("resumeFallback");
    }
}
```

When an exception is thrown by the `construct()` method, the second `Observable` type from the `resumeWithFallback()` method takes over.

During execution, when any non-recoverable error occurs, it is returned via the `onError` notification from the `Observable` type. After that, the `resumeWithFallback()` method gets invoked, returning the second `Observable` type.

The following table summarizes the execution exception:

Type of failure	Name of Exception class	Handled by fallback or not?
FAILURE	HystrixRuntimeException	Yes
TIMEOUT	HystrixRuntimeException	Yes
SHORT_CIRCUITED	HystrixRuntimeException	Yes
BAD_REQUEST	HystrixBadRequestException	No
THREAD_POOL_REJECTED	HystrixRuntimeException	Yes
SEMAPHORE_REJECTED	HystrixRuntimeException	Yes

We just discussed creating command instances for requesting services and how to handle fallback if it occurs in depth. During this discussion, we came across the terminologies such as fallback, circuit breaker, and thread pool. Let's delve deeper and discover the design patterns implemented by Hystrix to handle requests and request failure gracefully.

Design patterns in resiliency

We already discussed resiliency which is all about the ability to handle a system gracefully and recover from the failures as well. To make the application resilient, the detection of failure and recovering from it as quickly as possible and efficiently is very important. The following are the design patterns that can be implemented to achieve resiliency in the application.

Bulkhead pattern

The isolate design pattern isolates the elements of an application into small pools. This division facilitates continuing the functionality of the application, even though one of the functionality fails.

Let's consider the various services that can be requested by one or more consumers. As multiple consumers will request the resource, the excessive load on the application may lead to the failure, which in turn will impact the services to all of the requests made by different consumers.

Nowadays, users can make simultaneous requests one after another to various services in the application. While making the request, the user may use different resources as well. The request that has served correctly will not lead to any problem. However, what if the request made to a service is not responding due to some reason? Yes, we will not get the response. What will happen to the resource used by the failed request? The resource will be exhausted as it has not freed up in time, and ultimately, the service will become unresponsive.

As one consumer is making too many requests, the available resource in the service may get exhausted. In such a scenario, other consumers who want to use the service may no longer be able to use, which may cause cascading failures.

Many available applications have prime users, standard users, or critical users, and we want these consumers to be treated in isolation from each other so that the service fetched by one type of consumer may not affect another type of consumer.

Partition the services

In order to avoid the failures due to increasing load, the service can be partitioned into groups. These groups can be based on consumer load and availability requirement. As the service is divided into groups, and isolated from each other even though one of the part fails with other consumers.

Partition the resources

Along with the service, even the resources can be portioned. Consider Consumer A and Consumer B as two consumers requesting a service. Each one of them is using the connection. If service with Consumer A fails, then Consumer B will not be able to continue. Now, if we assigned a different connection to Consumer A and Consumer B, then, even if service with one consumer fails, the other consumer will continue without any problem.

Benefits

The bulkhead pattern offers the following benefits:

- It helps in isolating the consumers and the services from cascading failures and allows the application to continue working
- When one or more services fail, it won't affect the other services and features
- It allows developers to deploy the services based upon priorities

Circuit breaker

The bulkhead pattern is designed to provide the solution to handle fault applications that take more time to recover from the failure while trying to connect with the remote service or resource.

In a distributed environment, connecting to a remote service or resource will take significantly more time. This delay may be due to the problem in establishing communication, slow network connections, or timeouts or the resource may be overcommitted. These problems are not actually the failure of the service; rather, it's just a delay that may get corrected after a certain amount of time. Such kinds of faults correct themselves after a short period of time. By using the retry pattern, most of the robust applications are designed to handle such faults.

Although, for the scenarios discussed earlier, after retrying the request will get the resource and the task gets completed. However, sometimes, the time delay is due to some unpredictable reason that may take a long time to fix. These faults may either result in a partial connection problem or, sometimes, the complete service will also fail. Here, retrying for the connection is pointless. Now, instead of retrying, handling the failure is more important.

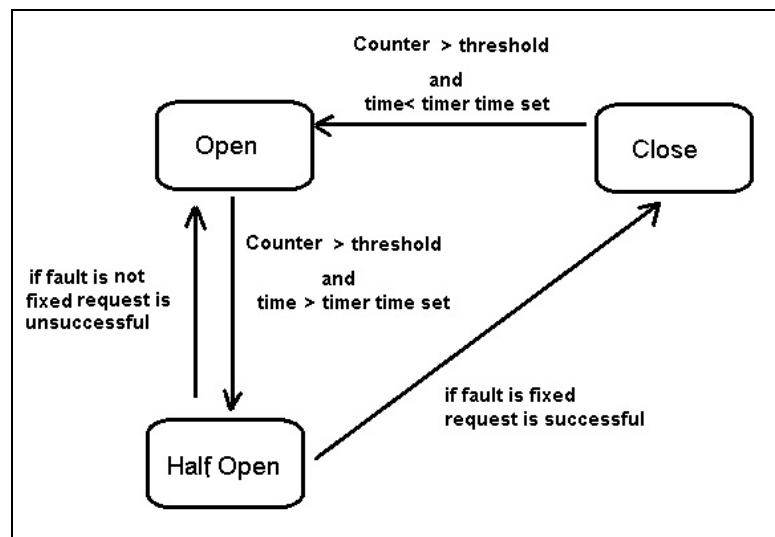
Service failure or delay in providing the service is one kind of fault. Sometimes, the service is busy in serving the request. While the service is busy, part of the application fails. The failure in the part may get cascaded to the service, that is busy performing the task. Let's consider a service where the timeout strategy has been implemented. The consumer is requesting the service and the service is not responding, even the consumer is getting a failure message due to timeout implementation. The consumer keeps on requesting as he is clueless about what is going on. All the requests will get blocked due to a timeout. What can be done? One probable solution is to keep the timeout shorter. However, keep in mind the timeout should not be so small that all requests will be failing.

The preceding problem discussed is due to repeatedly trying to execute a

service. The circuit breaker design pattern provides a solution. The pattern enables continuation rather than waiting for the fault to be fixed. It also enables finding whether the fault has been fixed or not; in case the issue has been resolved, the application can try to invoke the service.

Are the circuit breaker and retry patterns same? Actually not! The retry pattern enables the application to keep on trying the service until it's not successful. On the other hand, the circuit pattern doesn't allow the application to perform an operation or service that is likely to fail.

A circuit breaker works as a proxy for the operation or service that is likely to fail. This proxy keeps on monitoring the recently occurred failures. The collected information is then used to make the decision--should the operation proceed or should an exception be returned? The following diagram shows the circuit breaker states:



Proxy states

The proxy states are discussed as follows:

Closed state

The proxy maintains counters that contain the number of recent failures. Whenever the call to the operation fails, the counter will be incremented. If the counter value exceeds the threshold value within a particular time, then the proxy will be placed in an Open state. At the same time, the proxy starts a timeout timer. Once the timer time expires, the proxy will be placed in the Half-Open state.

Open state

If the proxy is placed in the Open state and a request is received by the operation, the operation fails immediately. An exception will be returned to the application.

Half-Open state

If the proxy is at a Half-Open state, then the limited number of requests will be allowed to pass through for invoking the operation. If the fault causing the failure is already fixed, then the requests made will be successful and the circuit breaker will be switched to a Closed state. If the fault causing the failure is not yet fixed, then the circuit breaker is switched to the Open state, along with a restarting timer.

Retry pattern

The stability of the application can be improved by enabling the application to handle the failures in the service when it tries to reconnect to the service, again and again. While requesting to the service, due to some network issues, momentary loss of connectivity causes temporary unavailability of the service. Sometimes, the service is busy in serving some other request that may cause timeouts, leading to the failure of the service.

If due to some connecting failures, the service is not currently available, we keep on trying for it, and most of the time, such errors are self-correcting. To make it easy, considering you are trying to call up your friend to convey wishes on Diwali Eve. Due to network conjunction, we get a message to try later, and we keep on retrying till, at one instance, our call gets connected.

If the application failure is detected, the following strategies can be used to handle it.

Retry

Sometimes, the occurred fault is not usual or very rarely occurs. Under such scenarios, the application can immediately retry for the failing request.

Retry after delay

Some faults occur as the service or resource is busy in serving the request. Retrying may not succeed immediately; instead, sometimes, if we retry after some time delay, we have a high chance of getting a successful connection.

Cancel

Some of the faults when detected, recovery from them is very rare, even though the service is tried multiple times. Now, continuing to retry is just a waste of time and resources and the application should cancel them.

Queue-based load leveling pattern

The queue will work as a buffer between the task and the service it invokes. As a queue acts as a buffer, it facilitates handling the load that may occur at peak demands, leading to the failure of the service.

The application keeps on serving enormous requests continuously coming from consumers. Whenever the application has few requests, it serves each request successfully. While, at peak hours, when more requests hit the service due to increase in the load, the service may face reliability issues.

Some of the services in the application use common resources concurrently. The volume of the request is not predictable, and may ultimately cause failure due to an increase in the load.

The increase in the load can be managed by introducing the queue in between the task and the queue. The task and the services run asynchronously, where the task posts the message required by the service to a queue. The message contains the data that is required by the service. The service keeps on retrying for the message from the queue to process it. The request made from various tasks for the service is passed through the same message queue. As the queue facilitates the decoupling of the service from the task, the service is free to serve the messages at its own pace, regardless of the heavy load of the requests.

The benefits of the patterns are listed as follows:

- As the queue acts as a buffer, the increase in the load doesn't affect the service or time delay in serving the request, making the application more scalable
- The task makes the request for the service through the queue that helps in managing the time delay, and other tasks can continue requesting, even though currently, the service is not serving
- As the load can be managed through the queue, it helps in controlling the cost of the servers or hardware.

Patterns used by Hystrix

The following are the common patterns used for `HystrixCommand` and `HystrixObservableCommand`:

Fail-fast

As per application designing, when the failure or condition to fail happens, it is immediately reported to the system. Fail-fast systems are designed so that the normal operation will stop instead of continuing. The system is designed in such a way that often checks, at several points will be done, to detect any failures. A fail-fast module doesn't handle the error; rather, it passes the responsibility to the next level of the system.

The very simple thing that as a developer we can do is to implement fail-fast, in our Hystrix system and not to provide any fallback mechanism. Yes! Don't override the `getFallback()` or `resumeWithFallback()` methods. Let's implement a fail-fast implementation by checking the length of the user, and if it is greater than 10, then we will throw `RuntimeException`, as shown in the following piece of code:

```
public class DemoHystrixCommand_Failfast extends
    HystrixCommand<String> {

    private String user;
    public DemoHystrixCommand_Failfast(String user) {
        // TODO Auto-generated constructor stub
        super(HystrixCommandGroupKey.Factory.asKey("packtGroup"));
        this.user = user;
    }

    @Override
    protected String run() throws Exception {
        // TODO Auto-generated method stub
        if (user.length() > 10)
            throw new RuntimeException("Length greater than 10");
        return "Welcome to Hystrix, " + user;
    }
}
```

The test case to check the fail-fast implementation is given as follows:

```
@Test
public void test_command() {
    HystrixCommand<String> command=new
        DemoHystrixCommand_Failfast("Packt Pub");
    assertEquals("Welcome to Hystrix, Packt Pub",
        command.execute());
}
```

```
    @Test
    public void test_queue_exception() {
        HystrixCommand<String> command=new
        DemoHystrixCommand_Failfast("Packt Publication");
        try {
            command.queue();
        }
        catch (Exception e) {
            // TODO Auto-generated catch block
            assertEquals("Length greater than 10", e.getMessage());
        }
    }
```

Fail-silent

The fail-fast pattern suggests not to implement the fallback mechanism. However, the fail-silent pattern provides an empty fallback. The implementation of the fallback methods returns an empty `List`, `Map`, or similar responses, as shown in the following code block:

```
public class DemoHystrixCommand_FailSilent extends
    HystrixCommand<String> {
    private String user;
    public DemoHystrixCommand_FailSilent(String user) {
        // TODO Auto-generated constructor stub
        super(HystrixCommandGroupKey.Factory.asKey("packtGroup"));
        this.user = user;
    }
    @Override
    protected String run() throws Exception {
        // TODO Auto-generated method stub
        if (user.length() > 10)
            throw new RuntimeException("Length greater than 10");
        return "Welcome to Hystrix, " + user;
    }
    @Override
    protected String getFallback() {
        // TODO Auto-generated method stub
        return null;
    }
}
```

We can even write the code as `Collections.emptyList()` in the `getFallback()` method. Let's now write the test case to find the result:

```
@Test
public void test_queue_exception() {
    HystrixCommand<String> command = new
        DemoHystrixCommand_FailSilent("Packt Publication");
    try {
        assertEquals(null, command.queue().get());
    }
    catch (InterruptedException e) {
        // TODO Auto-generated catch block
        fail(e.getMessage());
    }
    catch (ExecutionException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Here, we dealt with the `HystrixCommand` instance with the fail silent pattern. To deal with `HystrixObservableCommand`, we can override the `resumeWithFallback()` method returning an empty `Observable` type, as shown here:

```
public Observable<String> resumeWithFallback() {  
    return Observable.empty();  
}
```

Static fallback

As discussed earlier with the fail-silent pattern, that returns empty fallback, such as null value or empty `Observable`, the static fallback suggests returning a static value instead of null. The static value that the method will return is just default behavior, instead of keeping the user waiting for a long time, it is done earlier while discussing how to handle fallback. Just for a reference, observe the following lines of code:

```
protected String getFallback() {  
    // TODO Auto-generated method stub  
    return "Sorry!!! We failed";  
}
```

The method is returning a default `String` value as a response.

Stubbed fallback

In all of the earlier demos, we dealt with a simple value of type `String`. However, in practice, we deal with more complex domain objects that are obtained from a remote call over the network. In a normal situation, we will get the objects from the service method; however, if due to some fault, the call doesn't get completed, we will return the object. The cookies, request parameters, header values, or the values from the previous request provide the values that can be used for settings. If some of the values are unavailable, we can set the default values as shown in the in the following piece of code.

For this demo, we need to create a POJO class whose values can be fetched from the persistence layer. However, due to some reason, an exception occurred, leading to fallback. Here, the `getFallback()` method will return a stubbed dummy instance. Let's follow these steps to demonstrate it:

1. Create `Employee` as a POJO class and add the data members, default and parameterized constructor, and getters and setters, as shown in the following code snippet:

```
public class Employee implements Serializable {
    private String employeeFName;
    private long emp_id;
    private String address;
    private String mobileNumber;
    private String employeeLName;

    public Employee() {
        // TODO Auto-generated constructor stub
        employeeFName = "no name";
        employeeLName = "no l name";
        emp_id = 0;
        address = "Maharashtra";
        mobileNumber = "1234";
    }
    public Employee(String employeeFName, long emp_id, String address,
        String mobileNumber, String employeeLName) {
        super();
        this.employeeFName = employeeFName;
        this.emp_id = emp_id;
        this.address = address;
        this.mobileNumber = mobileNumber;
        this.employeeLName = employeeLName;
    }
}
```

```
    }
    // add getters and setters for all the data members
}
```

2. Now, create `DemoHystrixCommand_Stubbed` as a Java class and extend it from `HystrixCommand`, as shown here:

```
public class DemoHystrixCommand_Stubbed extends
    HystrixCommand<Employee> {
    private long employeeId;
    public DemoHystrixCommand_Stubbed(long employeeId) {
        super(HystrixCommandGroupKey.Factory.asKey("packtGroup"));
        this.employeeId = employeeId;
    }
}
```

3. It's time to override the abstract `run()` method. In practice, this method will give a call to the service methods that may lead to some kind of exception. However, here, for the purpose of demonstration, we are throwing `RuntimeException`, as shown here:

```
@Override
protected Employee run() throws Exception {
    // TODO Auto-generated method stub
    throw new RuntimeException("You got an exception");
}
```

4. After getting an exception, the fallback mechanism will be triggered, giving a call to the `getFallback()` method. Now, here, we will create a stubbed instance and will return it as shown in the following code snippet:

```
@Override
protected Employee getFallback() {
    // TODO Auto-generated method stub
    return new Employee("dummy_name", employeeId, "dummy
        address", "1234567", "dummy last name");
}
```

The `emp_id` of the instance has the same value as obtained from the request. However, other values are not available, so we are assuming some default values here.

5. Now, it's time to test the code using the test case as follows:

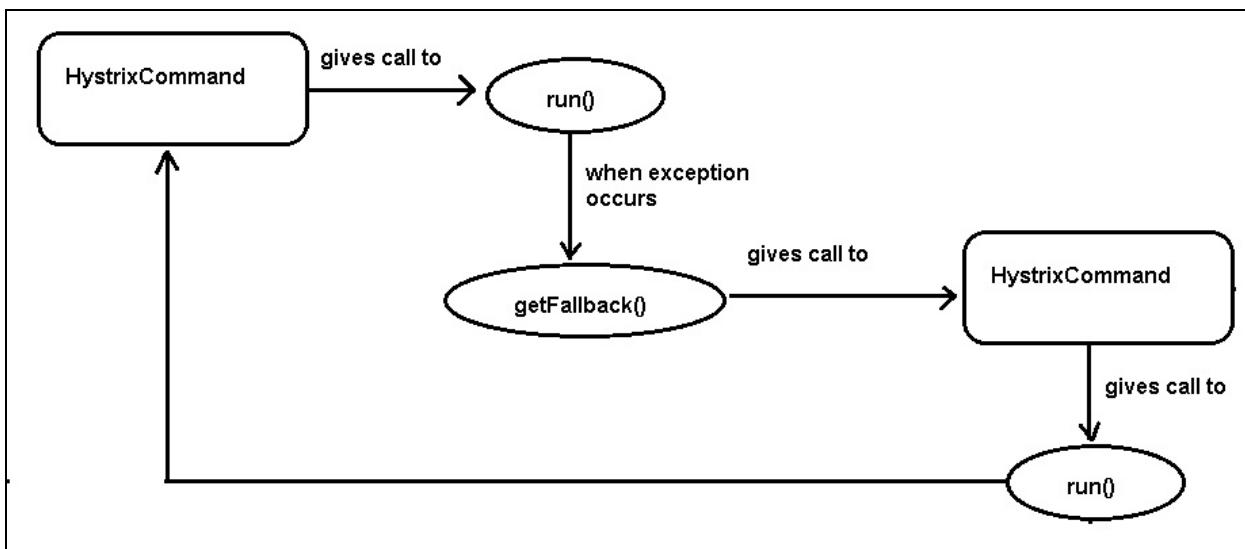
```
public class TestDemoHystrixCommand_Stubbed {
    @Test
```

```
public void test()
{
    DemoHystrixCommand_Stubbed demo=new
        DemoHystrixCommand_Stubbed(1111);
    Employee employee=demo.execute();
    Assert.assertEquals("dummy_name",
        employee.getEmployeeFName());
    Assert.assertEquals("dummy last name",
        employee.getEmployeeLName());
    Assert.assertEquals(1111, employee.getEmp_id());
    Assert.assertEquals("dummy address",
        employee.getAddress());
    Assert.assertEquals("1234567",
        employee.getMobileNumber());
}
```

On execution, the test case will pass successfully.

Cache via network fallback

We have made a network call to the service to fetch the data from a service, and, due to network failure, a fault occurred. Instead of returning nothing, we can retrieve the data from the cached service, such as **memcached**. The point to remember here is that the fallback will go over the network, which may again cause another fault. Hence, it's required to be wrapped by the instance of a command and to execute it on a separate thread pool. The following image gives a more detailed idea about cache via network fallback:



Memcached is a high-performance, open source, simple, powerful, and distributed memory object caching system.

Let's create a demo for network fallback where the fallback will go over the network where the network fault may occur. Here, we are using `RuntimeException`; however, in practice, the actual exception will cause the fault. Consider the following code:

```
public class DemoHystrixCommand_FallbackViaNetwork extends
    HystrixCommand<String> {
    private final int id;
    public DemoHystrixCommand_FallbackViaNetwork(int id) {
        super(Setter.withGroupKey(
            "FallbackViaNetwork")
            .withCommandKey(
                "FallbackViaNetwork"))
    }
}
```

```

        HystrixCommandGroupKey.Factory.asKey("packt_group"))
            .andCommandKey(HystrixCommandKey.Factory.asKey(
                "packt_group_Command")));
        this.id = id;
    }
    @Override
    protected String run() {
        throw new RuntimeException("Got an exception");
    }
    @Override
    protected String getFallback() {
        return new HystrixCommand_Network(id).execute();
    }

    private static class HystrixCommand_Network extends
        HystrixCommand<String> {
        private final int id;
        public HystrixCommand_Network(int id) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.
                Factory.asKey("packt_group"))
                .andCommandKey(HystrixCommandKey.Factory.asKey(
                    "packt_group_FallbackCommand"))
                .andThreadPoolKey
                (HystrixThreadPoolKey.Factory.asKey(
                    "packt_group_remote_Fallback")));
            this.id = id;
        }
        @Override
        protected String run() {
            // MemCacheClient.getValue(id);
            throw new RuntimeException("the fallback also failed");
        }
        @Override
        protected String getFallback() {
            return null;
        }
    }
}

```

We are using a different thread pool for the fallback command so that even though the pool for `packt_group` gets saturated, it will not prevent the fallback from executing. The `getFallback()` method from the network command object is handling the fallback using the fail-silent strategy.

The test case can be written in the same way as we did for all earlier demos. You can find the complete code file at <https://github.com/PacktPublishing/Reactive-Programming-With-Java-9>

Dual-mode fallback

Sometimes, some applications implement primary and secondary as dual-mode behavior. In this scenario, we have a main `HystrixCommand` working as a facade that will facilitate the flipping between the primary or secondary systems. The facade command will hide the implementation of the primary and secondary command, and take the responsibility of implementing the logic that will decide whether to invoke a primary command or secondary command. In case of the failure of both primary and secondary commands, the control will switch back to the facade command.

Both the primary as well as the secondary implementation of the `HystrixCommand` are thread safe, as they are involved in performing the business logic and network traffic. They may also implement different performance characteristics to take advantage of tuning them individually.

Let's create a command facade step by step that will decide to divert the flow to either primary command instance or secondary command instance:

1. Create a `DemoCommandFacade` class that will be working as our command facade.
2. Declare `user` as the data member of type `String` and `property` of type `DynamicBooleanProperty`. The property will decide whether to execute the primary command instance or the secondary command instance to call the underlying service.
3. Declare the parameterized constructor to set `HystrixCommandGroupKey`, `HystrixCommandKey`, and execution isolation strategy, as shown in the following code block:

```
public class DemoCommandFacade extends HystrixCommand<String> {  
    private String user;  
    private final static DynamicBooleanProperty property =  
        DynamicPropertyFactory.getInstance()  
        .getBooleanProperty("property.usePrimaryCommand", true);  
    public DemoCommandFacade(String user) {  
        // TODO Auto-generated constructor stub  
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.
```

```

        asKey("Hystrix_Facade"))
        .andCommandKey(HystrixCommandKey.Factory.
        asKey("packtGroup_Facade")).andCommandPropertiesDefaults(
            // use semaphore-isolation
            HystrixCommandProperties.Setter()
            .withExecutionIsolationStrategy
            (ExecutionIsolationStrategy.SEMAPHORE));
        this.user = user;
    }
}

```

Here, we use semaphore as an isolation strategy as the primary and secondary command instances we will implement will be thread-isolated.

4. Let's override the `run()` method. Using the data member property, we will decide whether to use the primary command instance or secondary command instance for giving a call to the service, as shown in the following code snippet:

```

@Override
protected String run() throws Exception {
    // TODO Auto-generated method stub
    if (property.get())
        return new DemoPrimaryHystrix(user).queue().get();
    else
        return new DemoSecondaryHystrix(user).queue().get();
}

```

5. Now, let's start with the primary command instance creation. Declare `DemoPrimaryHystrix` as an inner class. This is the same old command implementation that we created earlier. The class is shown as follows:

```

private class DemoPrimaryHystrix extends HystrixCommand<String> {
    private String user;
    public DemoPrimaryHystrix(String user) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.
        Factory.asKey("Hystrix_Facade"))
            .andCommandKey(HystrixCommandKey.Factory.
            asKey("Command1"))
            .andThreadPoolKey(HystrixThreadPoolKey.Factory.
            asKey("Command1"))
            .andCommandPropertiesDefaults(
                HystrixCommandProperties.Setter().
                withExecutionTimeoutInMilliseconds(600)));
        this.user = user;
    }
    @Override
    protected String run() {
        // give a service call here and return the obtained value
    }
}

```

```
        return "I am from DemoPrimaryHystrix:-" + user;
    }
}
```

6. In the same way, it's time to implement the secondary command implementation as shown here:

```
private class DemoSecondaryHystrix extends
    HystrixCommand<String> {
    private String user;
    public DemoSecondaryHystrix(String user) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.
            Factory.asKey("Hystrix_Facade"))
            .andCommandKey(HystrixCommandKey.Factory.
                asKey("Command2"))
            .andThreadPoolKey(HystrixThreadPoolKey.Factory.
                asKey("Command2"))
            .andCommandPropertiesDefaults(
                HystrixCommandProperties.Setter().
                    withExecutionTimeoutInMilliseconds(600)));
        this.user = user;
    }
    @Override
    protected String run() {
        // give a service call here and return the obtained value
        return "I am from DemoSecondaryHystrix:-" + user;
    }
}
```

The primary and secondary command implementations are thread-isolated.

7. In case both the primary and secondary command failed, then obviously the control will switch to fallback of the facade command. Let's handle it using the `getFallback()` method with the static fallback strategy as shown here:

```
@Override
protected String getFallback() {
    // TODO Auto-generated method stub
    return "sorry failed:-";
}
```

8. Let's write a test case to test the implementation that we just created. Before starting the testing and the working, we need to set the `property.usePrimaryCommand` property to `true`, and then, using the `execute` method, perform testing, as shown in the following piece of code:

```
@Test
public void testPrimaryCommand() {
    HystrixRequestContext context =
        HystrixRequestContext.initializeContext();
    try {
        ConfigurationManager.getConfigInstance().setProperty("property.usePrimaryCommand", true);
        assertEquals("I am from DemoPrimaryHystrix:-packt", new DemoCommandFacade("packt").execute());
    }
    finally {
        context.shutdown();
        ConfigurationManager.getConfigInstance().clear();
    }
}
```

The test case will run successfully.

In the same way, we can write the code to test a secondary instance. You can refer to the preceding code for more information.

Isolation

We all are developers and we all know, even though we have taken the utmost care, we can't make a 100% failure-free application. Our application has enormous dependencies; when one dependency fails, in turn, our application also fails. What if, instead of complete failure, only a part of the application fails? Yes, it will make our application perform better. Right now, we have all the functionalities, services, or resources as an intact part of the application, because of which, failure at one part renders the complete system unable to work.

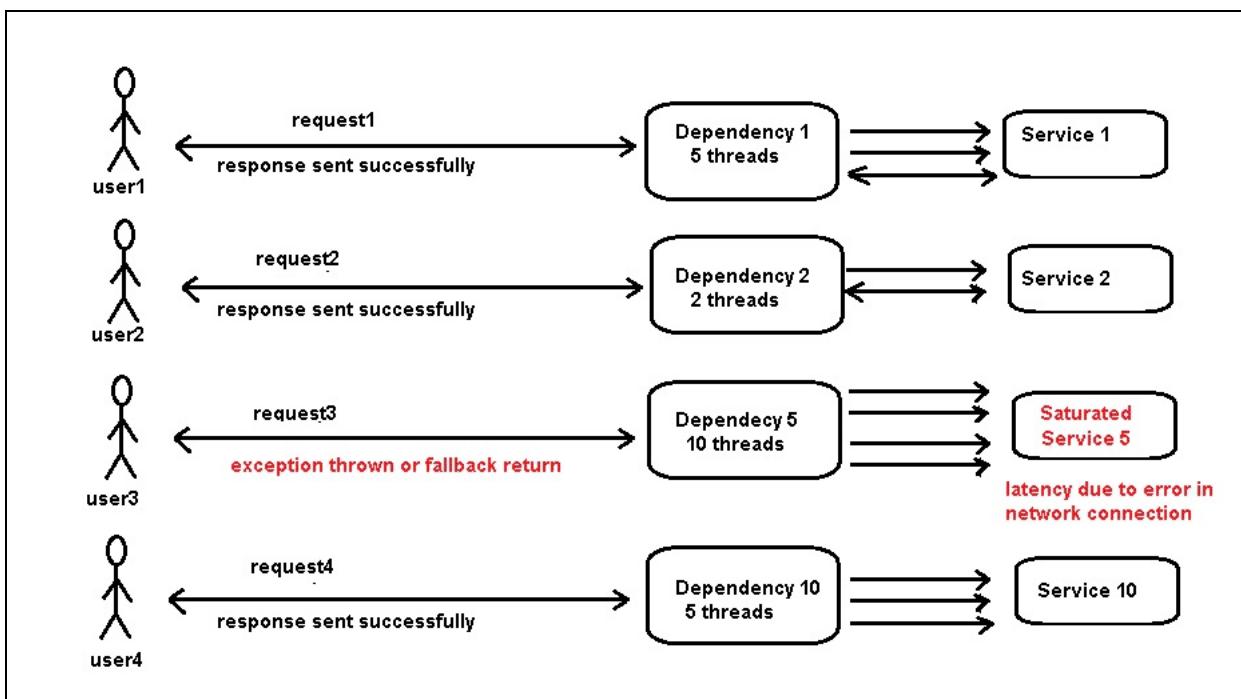
The problem can be solved using a very simple solution--isolation! We can design the application in parts. Each service of the application will be designed and implemented separately. Multiple consumers can make various requests to the same or different services. Now, even though one service fails as we made a clear separation between various services, the propagation of the error will not happen from one layer to the other.

We already discussed the bulkhead pattern, which isolates the elements of an application into small pools. Hystrix uses the bulkhead pattern to employ isolation and limit the concurrent access in the application.

Using threads and thread pools for isolation

As we know, usually, the consumer calls are executed on separate threads. As each request is getting executed on a separate thread, they are also isolated from the calling thread, enabling the consumer to move away if the response is taking more time.

Hystrix uses a separate thread pool for each dependency. One of the dependencies in the application may have more workload than the other. It means that it has more threads in the thread pool. If the load increases, the thread pool associated with the dependency may get saturated, which may lead to failure or keep the consumer waiting. However, the other pools still keep on serving the request. The following diagram illustrates the concept of thread pools:



Reasons for using thread pools in Hystrix

The following are the reasons to include threads and thread pools in the design of Hystrix:

- Each service in the application may use its own client library
- The libraries used by the service may be frequently changing
- The logic implemented for the network call may have changed
- Enormous amounts of applications keep on executing various backend services developed by different teams
- Network calls used in the application are synchronous
- Along with network call failure, sometimes the client-side implementation may fail

Benefits of using thread and thread pools

The benefits of using thread pools are listed as follows:

- Separation of a thread pool maintained for different dependencies facilitates filling up of a single thread pool without affecting the other thread pools.
- It enables the developers to add a new client library, as and when required, as it is totally isolated from the rest of the application. Even under certain scenarios, if the newly added library fails, it won't have any adverse effect on the running application.
- After retrying, the service may resume. Once the service becomes healthy and resumes, it will clear up the thread pool, allowing the application to start performing.
- By mistake, if any client library is misconfigured, it may affect the performance of the application; the developers can handle it without affecting the other parts of the application.
- As the dependency is managed by a dedicated thread pool, it provides built-in concurrency, leveraging the developers to take advantage of a build in.

Drawbacks of the thread pool

As we all know, the basic drawback of the thread pool is that it adds computational overhead as each command involves queuing, scheduling, and context switching.

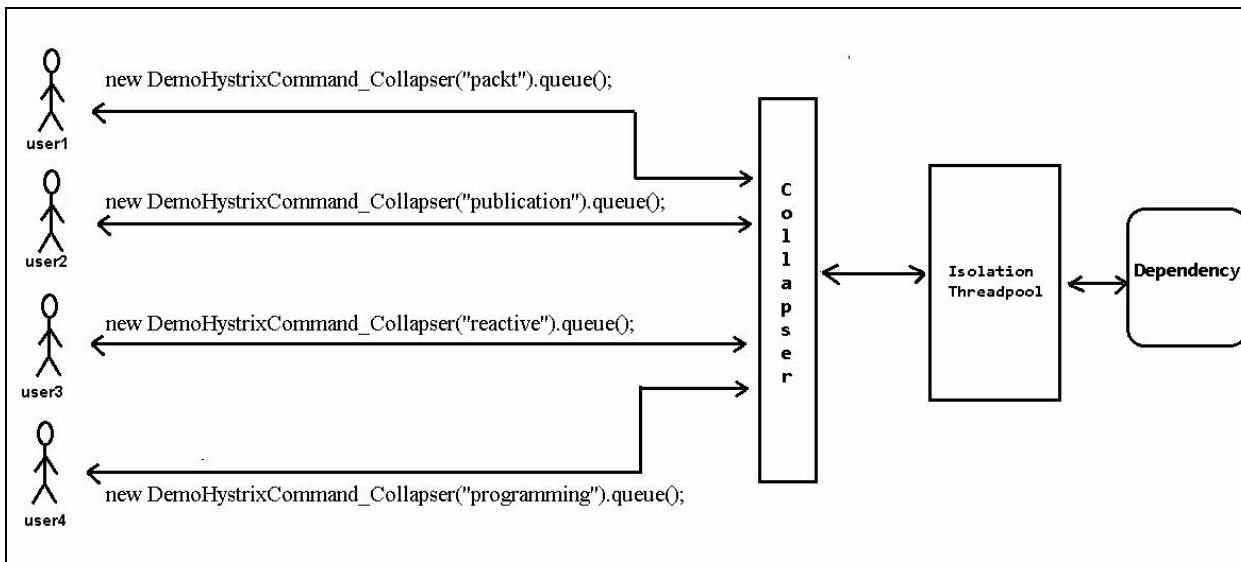
Semaphores

Instead of using the thread pool or the size of the queue, the semaphore is used to limit the number of concurrent calls to any dependency. The `HystrixCommand` OR `HystrixObservableCommand` instances support the following two places to use semaphore:

- **On fallback:** When Hystrix retrieves the fallback, it supports semaphore by calling the Tomcat thread
- **Execution:** This is the value of the `execution.isolation.strategy` property, when set to `SEMAPHORE`; Hystrix uses a semaphore instead of threads to control the concurrent threads that will invoke the command

Request collapsing or request batching

In JDBC, we can combine more than one process as a part of the batch and then execute the batch at one time instead of executing individual queries one by one. Similar to JDBC, we can combine multiple requests together into a single `HystrixCommand` instance execution, as shown in the following figure:



Request-scoped and globally-scoped are the styles of request collapsing supported by Hystrix. The request-scoped collapser collects the batch per `HystrixRequestContext` and the globally-scoped collapse collects the batch for multiple `HystrixRequestContext`.

We can create the collapse implementation as an extension of `HystrixCollapser`. Each `HystrixCollapser` accepts the following three generic types:

- `BatchReturnType`: This is the type of batched command response. The collapser turns multiple commands into a batch of command. Here, we will specify the type of that batch of command's response.
- `ResponseType`: This is the return type of each individual command that is

getting collapsed.

- `RequestArgumentType`: This is the type of the input of individual command we will be processing. When we batch multiple commands together, we are actually replacing all of them with a single batched command.

Let's write the implementation of the collapse, having `List<String>` as `BatchReturnType`, as written here:

```
public class DemoHystrixCommand_Collapser extends
    HystrixCollapser<List<String>, String, String> {
    private String userName;
    public DemoHystrixCommand_Collapser(String userName) {
        // TODO Auto-generated constructor stub
        super(Setter.withCollapserKey(
            HystrixCollapserKey.Factory.
            asKey("demoHystrixCommand_Collapser"))
            .andCollapserPropertiesDefaults(
                HystrixCollapserProperties.Setter().
                withTimerDelayInMilliseconds(2000)));
        this.userName = userName;
    }
    @Override
    protected HystrixCommand<List<String>> createCommand(
        Collection<CollapsedRequest<String, String>>
        collapsedRequests)
    {
        return new BatchHystrixCommand1(collapsedRequests);
    }
    @Override
    public String getRequestArgument() {
        // TODO Auto-generated method stub
        return userName;
    }
    @Override
    protected void mapResponseToRequests(List<String>
        batch_response,
        Collection<CollapsedRequest<String, String>>
        collapsedRequests)
    {
        // TODO Auto-generated method stub
        System.out.println("mapping response with size:-" +
            collapsedRequests.size());
        int i = 0;
        for(CollapsedRequest<String,
            String>request:collapsedRequests)
            request.setResponse(batch_response.get(i++));
    }
}
```

Here, the `BatchHystrixCommand1` class will handle the collected list and return the following:

```

class BatchHystrixCommand1 extends
    HystrixCommand<List<String>> {

    Collection<CollapsedRequest<String, String>>
        collapsedRequests;

    public BatchHystrixCommand1(Collection<CollapsedRequest<String,
        String>> collapsedRequests ) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.
            asKey("ExampleGroup")) .andCommandKey(HystrixCommandKey.
            Factory.asKey("GetValueForKey")));
        this.collapsedRequests = collapsedRequests;
    }
    @Override
    protected List<String> run() throws Exception {
        // TODO Auto-generated method stub
        int i=0;
        List<String> users = new ArrayList<>();
        for (CollapsedRequest<String, String> request :
            collapsedRequests) {
            users.add("user "+i++request.getArgument());
        }
        return users;
    }
}

```

Now, it's time to test the code using a test case. Here, we are dealing with collapsing, which is one of the request-scoped features, so it's a must to manage the `HystrixRequestContext` life cycle for initializing and cleaning it up, shown in the following statements:

```

HystrixRequestContext hystrixContext =
    HystrixRequestContext.initializeContext();

```

To terminate the request, we can use this:

```

hystrixContext.shutdown();

```

Most of the time, we handle web requests as well. To initialize `HystrixContextRequest` for a Java Web application using `Filter` execute the following code:

```

public class HystrixServletFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws IOException,
        ServletException
    {
        HystrixRequestContext hysContext =
            HystrixRequestContext.initializeContext();
        try {
            chain.doFilter(request, response);
        }
    }
}

```

```
        }
        finally {
            hysContext.shutdown();
        }
    }
}
```

And then, map the filter in deployment descriptor, as shown here:

```
<filter>
    <filter-name>hystrixServletFilter</filter-name>
    <filter-class>
        com.packt.ch10.HystrixRequestContextServletFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>hystrixServletFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

The working

The collapsing happens when two requests occur at the same time. However, nothing happens at the same time. Actually, when the first collapsible request comes in, Hystrix starts a timer that we have set it to 100 milliseconds. For 100 milliseconds, our command is suspended and we are waiting for other commands to join. After this configurable period, Hystrix will call `thecreateCommand()` method and run it. The `createCommand()` method gathers up all request keys, which is done by calling the `getRequestArgument()` method. When the batched command completes, it will dispatch results to all the awaiting commands.

Request caching

As a Java developer, we all are well aware of the concept of caching that facilitates decreasing the load on the network. Whenever multiple command instances fall to the same request scope, having the matching key, then only the first will be executed and all others are fetched from the cache instead of giving a call to the service on network.

By default, the cache is disabled to enable the cache we need to override the `getCacheKey()` method in our command implementation. The method returns a value of type `String`, which represents the state of a command instance.

Let's implement caching for `HystrixCommand`, as shown in the following piece of code:

```
public class DemoHystricCommand_cache extends
    HystrixCommand<String> {

    private final int value;
    public DemoHystricCommand_cache(int value) {
        super(HystrixCommandGroupKey.Factory.asKey("packt_pub"));
        this.value = value;
    }
    @Override
    protected String run() {
        return "welcome:-" + value;
    }
    @Override
    protected String getCacheKey() {
        return String.valueOf(value);
    }
}
```

Here, the `getCacheKey()` method is converting a data member value into `String`. Apart from the method, it's not different from our earlier implementation.

Now let's write a test case to find out the working of cache. The `HystrixCommand` class has the `isResponseFromCache()` method that facilitates us to find out whether the response is from service or cache, as shown in the following piece of code:

```
| @Test
```

```

public void testWithCache() {
    HystrixRequestContext context =
        HystrixRequestContext.initializeContext();
    try {
        DemoHystricCommand_cache command1 = new
            DemoHystricCommand_cache(2);
        DemoHystricCommand_cache command2 = new
            DemoHystricCommand_cache(2);

        assertTrue(command1.execute().equals("welcome:-2"));
        assertFalse(command1.isResponseFromCache());

        assertTrue(command2.execute().equals("welcome:-2"));
        assertTrue(command2.isResponseFromCache());
    }
    finally {
        context.shutdown();
    }
}

```

Both `command1` and `command2` are within the same request context. When we invoke `command2.isResponseCache()`, we are getting the result as `true` and the test case gives a successful execution.

Now, let's find out what happens when we initiate a new request context by updating the `testWithCache()` method with the following piece of code:

```

// start a new request context
context = HystrixRequestContext.initializeContext();
try {
    DemoHystricCommand_cache command3 = new Demo
        HystricCommand_cache(2);
    Assert.assertEquals("welcome:-2", command3.execute());
    assertFalse(command3.isResponseFromCache());
} finally {
    context.shutdown();
}

```

The `isResponseFromCache()` method returns the value as `false` as `command3` is from a different request context.

After having an in-depth discussion about request collapsing and request caching, the question that may arise in our minds is that, is cache and collapsing one and the same? Can we use them as a replacement for each other?

Difference between request collapsing and request caching

No, we cannot use collapsing or caching as a replacement for each other. The cache can be used when the resource or the service is accessed frequently. It's safe to use cache when we are quite confident that the value will be valid for some period of time. However, when we use collapsing, each time the request hits the service, it never returns any stale data. The collapsing may lead to unnecessary calls to the service, keeping it busy in serving, whereas, the caching facilitates decreasing the load on the service. We can use collapsing and caching together so that consulting to the cache can be made possible before executing the batch of commands.

Summary

The enterprise application developed by the developers may be distributed over the network. These distributed applications may have dependencies. Under normal conditions, these dependencies can be fetched successfully. However, sometimes, these dependencies may fail, causing the failure of the application due to various reasons. In this chapter, we discussed the Hystrix framework that enables developers to control the interactions between the distributed services along with additional support for latency and fault tolerance. We discussed the working of Hystrix using `HystrixCommand` or `HystrixObservableCommand` to present the request we are making to the dependency using the `execute()`, `queue()`, `observe()`, and `toObservables()` methods. However, due to network problems, increase in the load on service, timeout, and thread pool rejection, these methods may throw an exception that we gracefully handled by fallback.

We also discussed patterns such as bulkhead, circuit breaker, retry, queue-based load leveling, fail-fast, fail-silent, static fallback, and stubbed fallback adopted by Hystrix to handle the failures. The developers can combine together multiple requests together into a single `HystrixCommand` instance execution using the Hystrix collapsing. We demonstrate request collapsing using `HystrixCollapser`. More users using the application, in turn, increases the load on the application, and it may also hamper the application performance. The implementation of caching enables us to decrease the load on the networks and increases the performance.

In the next and the final chapter of our book, we will discuss data access logic using reactive drivers present for some data stores. We will use Spring data reactive to access data to make reactive repositories, even if the underlying data access driver is not reactive.

Reactive Data Access

In [Chapter 10, Implementing Resiliency Patterns Using Hystrix](#), we discussed in depth about the Hystrix library which facilitates fine-tuning of the interaction between services along with fault and latency tolerance. Before that, we discussed handling web requests using the Spring WebFlux module. All this gave us a sound idea of how the implementation of Reactive Programming has been done with these various concepts. We say that Reactive Programming is all about handling a continuous data flow. However, the major source of fetched data is the persistence layer. And as all are aware, the APIs that are available to handle data are all blocking APIs. Now, the question arises, if, ultimately, we are going to reach the persistence layer that has the blocking calls, then what's the use? Don't worry! We don't have to hit the layer which is non-reactive. In this chapter, we will discuss handling persisted data with non-blocking APIs using the following points:

- Reactive repositories using Spring Data Reactive
- Reactive Redis Access using Lettuce
- Reactive pipelines using Reactor Kafka

Those who've already worked with Spring are well aware of handling data persistence using the Spring DAO module. We can handle JDBC using the plain JDBC API using the Spring-JDBC module. To handle the database ORM style, we can easily integrate ORM frameworks such as Hibernate and JPA using the ORM module. However, it's still slightly cumbersome process, as we need to provide the implementation classes which will provide the facility for handling CRUD operations for mapped entities using JDBC, `JDBCTemplate`, or `HibernateTemplate`. The template classes used in the code take away the boilerplate and duplicate code. What if we are able to achieve the same persistence without creating such implementation classes with templates? Isn't it a great idea?

Spring Data

Spring Data is an umbrella project which facilitates the creation of Spring-based programming models to access data, keeping the characteristics of the underlying data store intact. It supports data access technologies which handle relational as well as non-relational databases, frameworks handling MapReduce, and cloud-based data services. The features of Spring Data can be listed as follows:

- It facilitates creation and maintenance of repositories
- It provides abstract custom-object mapping
- It dynamically creates derived queries from the method names of the repository
- It supports providing easy customization of the repository implementation
- It facilitates advanced integration with Spring MVC controllers

Modules provided by Spring Data

The following table describes the modules provided by Spring Data for easy access to the underlying persistent layer:

Name of the module	Description
Spring Data Commons	This provides the core concepts which will be used by all Spring Data projects
Spring Data JPA	This provides implementation of JPA-based repositories
Spring Data KeyValue	This module provides the facilities to access Map-based repositories and SPIs
Spring Data MongoDB	This module provides support for handling object-document-based repositories for MongoDB
Spring Data Redis	This module facilitates easy integration of Redis in Spring applications
Spring Data REST	This module facilitates exporting Spring Data repositories as hypermedia-driven RESTful resources
Spring Data Gemfire	This module facilitates easy integration of Gemfire in Spring applications
Spring Data LDAP	This module provides Spring Data repository support for Spring LDAP
Spring Data for Apache Cassandra	This module is designed for working with Apache Cassandra
Spring Data for Apache Solr	This module is designed for working with Apache Solr

Features of Spring Data

Spring Data provides the following features:

- It facilitates easy integration of multiple data stores
- It facilitates the use of the default CRUD functionality
- It is based upon the name of the methods it parses and creates queries to fire
- It provides support for auditing

Spring Data repositories

The `Repository` is the central interface in the Spring Data repositories which manages the domain class and the `ID` type of that particular domain class. The main task of the interface is to find types to work with for handling data and help the developers to discover all those interfaces which extend the `Repository` interface.

The `CrudRepository` interface extends the `Repository` interface, and facilitates handling CRUD functionalities to deal with the database. The following table explains the methods facilitating the CRUD operations provided by the `crudRepository` interface:

Name of the method	Operations the method handles
<code>count()</code>	The <code>count()</code> method returns the number of entities available
<code>delete(ID id_delete)</code>	The <code>delete()</code> method facilitates deletion of the entity having the given ID
<code>delete(Iterable iterable_delete)</code>	This overloaded version of the <code>delete()</code> method facilitates deleting all the given entities
<code>delete(T entity_to_delete)</code>	This overloaded version of the <code>delete()</code> method facilitates deleting the given entity
<code>deleteAll()</code>	The method helps in deleting all the entities managed by the repository
<code>exists(ID id_exists)</code>	The <code>exists()</code> method helps to find whether the entity specified by the <code>ID</code> exists or not
<code>findAll()</code>	The <code>findAll()</code> method returns all the available instances
<code>findAll(Iterable ids)</code>	The method returns all the instances with the specified IDs by the <code>Iterable</code>

<code>findOne(Id id_find)</code>	The method returns the instance of an entity specified by <code>id_find</code>
<code>save(Iterable iteratable_save)</code>	The method facilitates saving all the entities in an <code>Iterable</code>
<code>save(S entity)</code>	The method facilitates saving the given entity

We have various flavors of `Repository`; however, using these various repositories is the same process. Let's discuss the general process for using the Spring Data repository.

Using the repository

The standard CRUD functionality repositories have the queries to fire on the underlying database. While working with Spring Data, the declaration of these queries is a five-step process as discussed next:

1. Declaring the repository interface.
2. Declaring the methods in the repository interface.
3. Discovering the repository interface.
4. Enabling the proxy creation of the interface.
5. Using the repositories.

Let's discuss the following steps one by one:

1. **Declaring the repository interface:** The first step for using the repository is to create an interface which will extend the `Repository` interface or any of its sub interfaces as shown here:

```
interface EmployeeRepository extends Repository<Employee, Long>
{
    // method declaration will go here
}
```

The preceding interface declares an extension of the `Repository` interface which handles `Employee` with an ID of type `Long`.

If you want `Repository` to expose the CRUD methods for a particular domain type, then you can extend the interface from `CrudRepository` instead of using the `Repository` interface.

Usually, we use a single Spring Data module in our application, which is very simple as well as easy. In such a scenario, all the repository interfaces which we have defined are bound to the Spring Data module. However, sometimes, the application demands we use more than one Spring Data module. Here, it's important for a repository definition to distinguish between persistence technologies.

2. **Discovering the repositories:** To enable the discovery of the repositories we need to add the following configuration:

```
@EnableJpaRepositories(basePackages = "com.packt.dao.jpa")
@EnableMongoRepositories(basePackages = "com.packt.dao.mongo")
interface MyConfiguration {
    // query methods goes here
}
```

The preceding configuration suggests the framework to use `MongoRepository` on the `com.packt.dao.mongo` package and `JpaRepository` to be used in the `com.packt.dao.jpa` package.

3. **Declaring the methods in the repository:** Now, declare the query methods in the interface created previously. The updated interface will be as shown here:

```
interface EmployeeRepository extends Repository<Employee, Long>
{
    List<Employee> findAllEmployees();
    List<Employee> findAllEmployeesByName();
}
```

The repository proxy uses a store-specific query from the name of the method and writes the defined query manually.

Let's go into a bit of depth about writing and discovering the query.

- **Looking up the queries:** The query can be resolved for the available repository via the configuration based-strategy by configuring the attribute `query-lookup-strategy` in the XML file. If we want to use the annotation-based configuration, we use the `queryLookupStrategy` attribute along with `@EnableXXXRepository`, where `xxx` can be replaced with the name of the repository store. The following are some of the strategies available to us:
 - **CREATE:** The `CREATE` strategy constructs the query for a specific store from the name of the query method.
 - **USE_DECLARED_QUERY:** This strategy tries to find out a declared query, and in case the query is available then an exception will be thrown.
 - **CREATE_IF_NOT_FOUND:** This is the default strategy that allows quick query definition, which combines the strategy `CREATE` and

`USE_DECLARED_QUERY`. This strategy first looks up the query; if no declared query is found, then the strategy creates a method name-based query.

- **Creating queries:** The Spring Data repository has a built-in query builder mechanism which facilitates the building of queries with constraints over the entities in the repository. Let's discuss some of the scenarios for declaring the name of the methods, as the method name plays a vital role in building queries:
 - A method which wants to perform some query operation using `find..By`, `read...By`, `count...By` and `get...By` clauses. We can have the method declaration as shown by the following cases:

```
| List<Employee> findByFirstNameAndLastname(String  
|   firstName, String lastname);
```

- The query-builder mechanism takes off the prefixes such as `find..By` from the methods, and then parses the rest of the method name.
- A method with the `distinct` clause can be written as,

```
| List<Employee> findDistinctEmployeeByFirstname(  
|   String firstname);
```

Or, it can also be written as this:

```
| List<Employee> findEmployeeDistinctByFirstname(  
|   String firstname);
```

- Writing the method for the `clause` while ignoring cases of the values of query parameters used in the query is done as follows:

```
| List<Employee> findByFirstNameIgnoreCase(String  
|   firstName);
```

- Writing the method for the `ORDER by` clause is done like this:

```
| List<Employee> findByFirstNameOrderByIDAsc(String  
|   firstName);
```

- The result can be limited by using keywords such as `top` and `first`, as in the following:

```
| List<Employee> findFirst5ByFirstName(String FirstName);
```

Or it can be:

```
| List<Employee> findTop5ByLastname(String lastName);
```

- The result obtained from the query methods can now be processed using a Java 8 `Stream<T>` as the return type. Here, the query results in a Stream-data-store-specific methods will be used to perform streaming:

```
| @Query("select e from Employee e")
  Stream<Employee> findAllByQueryAndStream();
```

- The annotation `@Query` is used to specify the query to fire on the repository. It can also be written like this:

```
| Stream<Employee> readAllEmployeesByLastNameNotNull();
```

- The `@Async` annotation enables querying the repositories asynchronously as follows:

```
| @Async
  Future<Employee> findByFirstname(String firstName);
```

Or it can also be written like this::

```
| @Async
  ListenableFuture<Employee> findOneByFirstName
  (String firstName);
```

4. **Enabling the proxy creation of the repository interface:** We just declare the interface with methods. Unless someone is not proving the implementation how will the execution take place? No, we don't need to write the implementation as the framework provides the proxies having an implementation. It's time to configure the proxy instance of the created interface. It can be done in either of two ways as discussed here:

1. In the XML configuration, we can add the following configuration:

```
| <beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa_data=
```

```
"http://www.springframework.org/schema/data/jpa"
xsi:schemaLocation=
"http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/data/jpa
http://www.springframework.org/schema/data/jpa/
    spring-jpa.xsd">
<jpa_data:repositories base-package=
    "com.packt.ch11.dao"/>
<!-bean configuration will goes here -->
</beans>
```

2. By using annotations in XML configuration, we can also use the `class-level @EnableJpaRepositories` annotation to enable the proxy instances as shown next:

```
@Configuration
@EnableJpaRepositories
class MyConfiguration {
    // method will come here
}
```

5. **Using the repositories:** Now it's time to inject the implementation of the interface in the service class using the `@Autowired` annotation as follows:

```
class MyService {
    @Autowired
    EmployeeRespository employeeRepository;
    // The methods will go here
}
```

Now we need to create the client which may be a Controller in Spring MVC or `public static void main()` function which facilitates obtaining the `bean` from the context to perform the operations on the repository data.

Spring Data and Reactive Programming

In all our previous discussion and configurations, we just collected the basics required of Spring Data repositories. However, the Spring Data module has been updated to enable Reactive Programming. The Spring Data MongoDB module leverages the reactive programming model in Spring Framework 5. Spring Data Key M1 is the first release which supports reactive data access. Its initial version supports stores such as MongoDB, Apache Cassandra, and Redis. For all these stores, the reactive drivers are ready to use. Let's take a look, in detail, at Spring Data and Reactive Programming.

ReactiveCrudRepository

The `ReactiveCrudRepository` interface extends from `Repository`, and facilitates Reactive Programming. Observe the interface method specified next, which returns data not of type `Object`. However, these methods deal with reactive types such as `Mono` and `Flux` in contrast to the traditional repository interfaces. The following code snippet describes the `ReactiveCrudRepository` interface:

```
public interface ReactiveCrudRepository<T, ID> extends
    Repository<T, ID> {
    <S extends T> Mono<S> save(S entity);
    <S extends T> Flux<S> saveAll(Iterable<S> entities);
    Mono<T> findById(ID id);
    Mono<T> findById(Publisher<ID> id);
    Mono<Boolean> existsById(ID id);
    Mono<Boolean> existsById(Publisher<ID> id);
    Flux<T> findAll();
    Flux<T> findAllById(Iterable<ID> ids);
    Flux<T> findAllById(Publisher<ID> idStream);
    Mono<Long> count();
    Mono<Void> deleteById(ID id);
    Mono<Void> deleteAll(Iterable<? extends T> entities);
    Mono<Void> deleteAll(Publisher<? extends T> entityStream);
    Mono<Void> deleteAll();
}
```

RxJava2CrudRepository

`RxJava2CrudRepository` from the library also provides functionalities for generic CRUD operations on the repository, which follows reactive paradigms and allows developers to use RxJava 2 types. Observe this `RxJava2CrudRepository` interface, which facilitates handling the data types supported by RxJava2:

```
interface RxJava2CrudRepository<T> {
    Single<Long> count();
    Completable delete(T entity);
    Completable deleteAll();
    Completable deleteAll(Flowable<? extends T> entityStream);
    Completable deleteAll(Iterable<? extends T> entities);
    Completable deleteById(ID id);
    Single<Boolean> existsById(ID id);
    Single<Boolean> existsById(Single<ID> id);
    Flowable<T> findAll();
    Flowable<T> findAllById(io.reactivex.Flowable<ID> idStream);
    Flowable<T> findAllById(Iterable<ID> ids);
    Maybe<T> findById(ID id);
    Maybe<T> findById(Single<ID> id);
    Single<S> save(S entity);
    Flowable<S> saveAll(Flowable<S> entityStream);
    Flowable<S> saveAll(Iterable<S> entities);
}
```

Spring Data Reactive and MongoDB

The Spring Data MongoDB project applies the core concepts of Spring to develop solutions using the MongoDB document style. It provides the templates supporting high-level abstraction for storing and querying documents from the data store.

Features added to Spring Data MongoDB 2.0

The features added recently to Spring data MongoDB 2.0 are listed as follows:

- It provides complete support for Java 8
- It provides support for aggregation result streaming using Java Stream.
- It supports performing Reactive Programming using MongoDB
- It facilitates integration of collection and index creation and query operations
- It supports automatic implementation of reactive `Repository` interfaces to support custom finder methods

Using MongoDB with Spring

For using Spring MongoDB, we need to use MongoDB 2.6 or higher along with Java SE 8 or higher. The following configuration in `pom.xml` helps in managing the required dependencies:

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-mongodb</artifactId>
  <version>{version}</version>
</dependency>

<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongodb-driver-reactivestreams</artifactId>
  <version>{mongo.reactivestreams.version}</version>
</dependency>

<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
  <version>{reactor.version}</version>
</dependency>
```

One can use `com.mongodb.reactivestreams.client.MongoClient` for registering the `MongoClient` instance using the following piece of code:

```
MongoClients.create("mongodb://localhost");
```

In the upcoming pages, we will also discuss how to connect to the embedded server. If you don't want to use `MongoClient` for registering the instance, then `ReactiveMongoClientFactoryBean` can be used for performing the task as shown here:

```
ReactiveMongoClientFactoryBean factory = new
    ReactiveMongoClientFactoryBean();
factory.setHost("localhost");
```

ReactiveMongoTemplate

`ReactiveMongoTemplate` plays a vital role in supporting Spring's Reactive MongoDB with features for interacting with the database. The template class implements the interface, `ReactiveMongoOperations`. Developers prefer to use the `ReactiveMongoTemplate` instance via `ReactiveMongoOperations`. The template facilitates the methods for performing the operations such as create, update, delete, and query for MongoDB documents. It also provides the mapping between domain objects and MongoDB documents. The interface `MongoConverter` facilitates mapping between `MongoDocument` and the domain classes. `MongoMappingConverter` is the default implementation of the `MongoConverter` interface, which is used by the framework. `ReactiveMongoTemplate` can be instantiated as shown by this code:

```
ReactiveMongoTemplate template= new reactiveMongoTemplate(  
    MongoClients.create("mongodb://localhost"), "mydb");
```

Now we can use methods such as `insert()`, `insertAll()`, `insertDocumentList()`, `remove()`, `save()`, `saveDocument()`, `updateFirst()`, `find()`, `findOne()`, `findById()`, `findAll()`, `exists()`, `count()`, and many more to perform operations on the database.

Now that you know the basics of working with MongoDB, it's time to discover how to use MongoDB in Spring Data Repositories.

Spring Data Repositories and MongoDB

The Spring framework supports the integration of MongoDB using the namespace as shown in the following code:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:my_mongo="http://www.springframework.org/schema/data/mongo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/data/mongo
                           http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd"
```

Once the namespace configuration is done, we now need to tell the framework where to find the repositories, which can be done as follows:

```
|   <my_mongo:repositories base-package="com.packt.ch11.repositories"/>
```

The base packages will be scanned for interfaces which get extended from `MongoRepository`, and now the framework will create proxy beans for each one of them.

By default, the repositories will get an instance of `MongoTemplate`, so it's important to configure one in our configuration as follows:

```
<bean id="mongoTemplate"
      class="org.springframework.data.mongodb.core.MongoTemplate">
      <constructor-arg ref="mongoClient" />
      <constructor-arg value="databaseName" />
</bean>
<my_mongo:mongo-client id="mongoClient" />
```

We also have the `@EnableMongoRepositories` annotation to be used as an alternative for the preceding configuration. In upcoming pages, we will use it to handle the repository. We got `ReactiveMongoRespository` as an interface to deal with Spring Data repositories, which is specific to MongoDB.

ReactiveMongoRepository

The interface, `ReactiveMongoRepository`, extends `ReactiveSortingRepository` and `ReactiveQueryByExampleExecutor`. Similar to `RxJava2CrudRepository`, it supports handling reactive types. This interface is a Mongo-specific Repository interface with reactive support such as `Mono` and `Flux`. The following are the interface-specific methods which facilitate performing of CRUD operations on the repository:

```
public interface ReactiveMongoRepository<T, ID> extends
    ReactiveSortingRepository<T, ID>, ReactiveQueryByExampleExecutor<T>
{
    Flux<S>    findAll(Example<S> example)
    Flux<S>    findAll(Example<S> example, Sort sort)
    Flux<S>    insert(Iterable<S> entities)
    Flux<S>    insert(Publisher<S> entities)
    Mono<S>    insert(S entity)
}
```

Spring Data Reactive and Redis

NoSQL storages provide an alternative to the traditional RDBMS approach, providing horizontal scalability as well as speed. The NoSQL implementation represents the `KeyValue` stores. The **Spring Data Redis (SDR)** framework helps developers to write Spring applications which use the Redis key value store. The framework facilitates the elimination of redundant and boilerplate code for quick application development. The following are the features added to Spring Data Redis 2.0:

- The API is updated to support Java 8 features
- The support for the SRP and JRedis drivers is removed
- The API is upgraded to use Lettuce 5.0
- A reactive connection support provides usage of the `lettuce-io` and `lettuce-core` modules
- The `RedisConnection` interface has been introduced for providing a Redis-specific connection
- The implementation of `RedisCache` is revised
- Added the support for SPOP with count command for Redis 3.2



The Spring Data Redis binaries 1.0.0.x requires JDK level 6.0 to be used along with Spring Framework 5.0.0.RC3 and above.

Connecting to Redis

To start with Redis, the very first step is to get a connection. `RedisConnection` facilitates communication between the Spring application and Redis. We can also create active connections through the `RedisConnectionFactory` interface. These factories also act like `PersistenceExceptionTranslator`. It means that once the factory is declared, it helps us to perform transparent exception translation. `RedisConnectionFactory` can be configured using an appropriate connector through IoC.

Jedis Connector

Jedis is one of the connectors which are supported by the Spring Data Redis module for obtaining `RedisConnectionFactory`. We can configure Jedis as follows:

```
<bean id="connectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">
    <property name="host_name" value="localhost"/>
    <property name="port" value="6379"/>
</bean>
```

Lettuce connector

We can use Lettuce connector as an open source connector for connecting to Redis using the following configuration:

```
<bean id="connectionFactory" class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory">
    <property name="host_name" value="localhost"/>
    <property name="port" value="6379"/>
</bean>
```

Now, it's time to use `RedisConnection` to fire the query on the data store. However, the Spring Framework provides the template classes to facilitate getting rid of the unnecessary duplicate code. Let's explore `RedisTemplate` to perform the operations.

RedisTemplate

`RedisTemplate` supports rich features and offers a high-level abstraction for interacting with Redis. The template facilitates serialization and connection management so that developers don't need to worry about them.

We can configure `RedisTemplate` by using this configuration:

```
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
<property name="connection-factory" ref="connectionFactory"/>
</bean>
```

`RedisTemplate` facilitates the operations view, which provides rich, generic interfaces so that working against the type or a certain key becomes easy. The following table lists all such interfaces:

Name of interface	Description
ValueOperations	The interface facilitates Redis string (or value) operations
ListOperations	The interface facilitates Redis list operations
ZSetOperations	The interface facilitates Redis ZSET (or sorted set) operations
SetOperations	The interface facilitates Redis set operations
HashOperations	The interface facilitates Redis hash operations
BoundValueOperations	The interface facilitates Redis string key-bound operations
BoundListOperations	The interface facilitates Redis list key-bound operations
BoundSetOperations	The interface facilitates Redis set key-bound operations
BoundHashOperations	The interface facilitates Redis hash key-bound

operations

Spring Data Repositories and Redis

Redis repositories allow developers to convert and store domain objects in Redis Hashes. The repository allows applying custom mapping strategies by making use of secondary indexes. We can simply use the annotation `@EnableRedisRepositories` for enabling Redis-based repositories.

In order to enable Redis to work with Spring Data, we need to follow these steps:

1. Create a domain entity to be stored in the repository. Annotate the entity with the `@RedisHash` annotation.
2. Now add the `@Id` annotation to make the data member work as the `Primary Key`.
3. Create a basic repository to extend from the predefined repository.
4. The final step is to enable the repository to be discovered using `@EnableRedisRepository`.
5. Now declare a class which will use the created repository to perform operations on repositories such as adding data or finding data.

After lots of theoretical discussions, I am sure you are eagerly waiting to use the concepts in an application. Let's develop an application step by step to use Spring Data `ReactiveCrudRepository`, `MongoDB`, and `Maven` integration:

1. Create a new Maven project with the group ID as `com.packt.ch11.dao.repositories` and `ArtifactId` as `Spring_reactive_maven`.
2. Add the following dependencies in `pom.xml`:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.BUILD-SNAPSHOT</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-
      reactive</artifactId>
  </dependency>
```

```

    </dependencies>
<repositories>
    <repository>
        <id>spring-libs-snapshot</id>
        <name>Spring Snapshot Repository</name>
        <url>http://repo.spring.io/libs-snapshot</url>
    </repository>
</repositories>

```

- Now, it's time to add a POJO whose data will be saved in the data store. Create `Employee` as a class in `com.packt.ch09.pojos`. You can even use the class created in [Chapter 10, Implementing Resiliency Patterns Using Hystrix](#). Annotate the class with the `@Document` annotation. The class will be as shown next:

```

@Document
public class Employee implements Serializable {
    private String name;
    @Id
    private Long id;
    private String address;
    private String mobileNumber;
    private String lastName;
    //add getters and setters for all the data members
    // add toString() method
}

```

Here, we have annotated `id` by `@Id` to define it as the Primary Key.

- After defining the entity, let's define a data repository as discussed earlier. Here, we will create an interface as `ReactiveEmployeeRepository` extended from `ReactiveCrudRepository<Employee, String>`, as follows:

```

public interface ReactiveEmployeeRepository extends
    ReactiveCrudRepository<Employee, String> {
    Flux<Employee> findByName(String name);
    Flux<Employee> findByAddress(String lastName);
    Flux<Employee> findByAddressAndName(String
        address, String name);

    @Query("{'name':?0, 'address': ?1}")
    Mono<Employee> findByNameAndAddress(String name,
        String address);
}

```

We have already discussed the naming convention for declaring methods under the *Creating queries* section of this chapter.

- Let's now define a class which will be extended from `AbstractReactiveMongoConfiguration`, which acts as the base class for the reactive Spring Data MongoDB configuration.
- The class will define a client creation for MongoDB as an embedded server. The code will be as follows:

```

    @SpringBootApplication(exclude = { MongoAutoConfiguration.class,
        MongoDataAutoConfiguration.class
    })
    @EnableReactiveMongoRepositories
    @AutoConfigureAfter(EmbeddedMongoAutoConfiguration.class)
    public class ApplicationConfiguration extends
        AbstractReactiveMongoConfiguration
    {
        private final Environment environment;
        public ApplicationConfiguration(Environment environment)
        {
            this.environment = environment;
        }
        @Override
        @Bean
        @DependsOn("embeddedMongoServer")
        public MongoClient mongoClient() {
            int port = environment.getProperty("local.mongo.port",
                Integer.class);
            return MongoClients.create(
                String.format("mongodb://localhost:%d", port));
        }
        @Override
        protected String getDatabaseName() {
            return "mydb";
        }
    }
}

```

The name of database needs to be configured as per the configuration of your database.

- Finally, it's time to test the application. Let's create `ReactiveEmployeeRepository` as a JUnit test case, and annotate it with `@RunWith` and `@SpringBootTest`, as shown in the following code:

```

    @RunWith(SpringRunner.class)
    @SpringBootTest
    public class ReactiveEmployeeRepositoryTest {
    }

```

- Now, add `ReactiveEmployeeRepository` and `ReactiveMongoOperations` as data members, and annotate both of them with the `@Autowired` annotation, like this:

```

    @Autowired
    ReactiveEmployeeRepository repository;

    @Autowired
    ReactiveMongoOperations operations;

```

ReactiveMongoOperations is provided by Spring Data MongoDB, which is implemented by `ReactiveMongoTemplate`, and facilitates the use of Project Reactor's reactive types, such as `Mono` and `Flux`, for wrapping the responses.

9. Set up the data using the `setUp()` method to save the data to store as follows:

```

@Before
public void setUp() {
    operations.collectionExists(Employee.class)
        .flatMap(exists -> exists ?
        operations.dropCollection(Employee.class) :
        Mono.just(exists))
        .flatMap( o ->
        operations.createCollection(
        Employee.class, new CollectionOptions(1024 * 1024, 100,
            true))).then().block();
    repository.save( Flux.just(
        new Employee("name", "Pune", "1234", "lastname", 101),
        new Employee("name123", "MUMBAI", "1234",
            "lastname123", 1121),
        new Employee("name", "Pune", "1234", "lastname", 1001)))
        .then().block();
}

```

10. Now add `findByNameTest` as a test method which is annotated with the `@Test` annotation for testing the method `findByName()`, as shown in the following code:

```

@Test
public void findByNameTest() {
    List<Employee> employee = repository.findByName(
        "name").collectList().block();
    assertThat(employee).hasSize(2);
}

```

11. Run the test case as a `Unit Test`, and observe the output. The test case will execute successfully, creating the data store with three entries.

Now, without hesitation, add various methods in the repository for various

operations on the repositories, and test them in the same way as we did here for the `findByName()` method.

In the same way, we can create Spring Data Repositories for Redis by performing a few changes in the demo we just created, as follows:

1. Add an annotation to the domain object as `@RedisHash("employees")` instead of using `@Document`.
2. Change the `ApplicationConfiguration` class to acquire the connection with Redis as follows:

```
@SpringBootApplication
@EnableRedisRepositories
public class ApplicationConfiguration {
    @Bean
    LettuceConnectionFactory connectionFactory() {
        return new LettuceConnectionFactory();
    }
    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        RedisTemplate<String, Object> template = new
            RedisTemplate<String, Object>();
        template.setConnectionFactory(connectionFactory());
        return template;
    }
}
```

3. Add the JARs for Redis, and we are set to go with Spring Data Redis.
4. Create a JUnit test to test the application.

In this section, we discussed using reactive repositories with Spring data. We discuss handling data using, MongoDB and Redis using `ReactiveCrudRepository`. It's an amazing thing to perform CRUD operations in a reactive way on a data store without writing any implementation, isn't it? Here, we worked intensively with data; now it's time to move on and handle the messages as well. So, without wasting time let's explore Kafka for dealing with messages.

Kafka

Kafka is a distributed, scalable, fast system which is used for publishing and subscribing messages, and building real-time data pipelines; and applications with streaming. **Topic** is the most important component of Kafka. A producer writes the data to the topic, and the consumer reads the data from the topic. The good thing about Kafka being distributed is that the topics are partitioned, and can be replicated over multiple nodes. The data sent to the topic is usually called **messages**, which can be as simple a type as **string**, and as complex as **JSON**.

Data Pipeline



Data Pipeline is an embedded data processing engine developed to be used with the Java Virtual Machine. The data pipeline facilitates the conversion of the incoming data, analyzing the data, migration of the data to the database, and much more. It also allows the developers to associate any arbitrary information (metadata) about the record.

The following are the features of Reactive Kafka:

- **Functional interface support:** Reactor Kafka is the functional Java API for Kafka.
- **Nonblocking back pressure:** The Reactor Kafka APIs are benefited by non-blocking pressure so that backpressure can be applied to either the whole or a part of the pipeline taking control over the memory usage.
- **End-to-end reactive pipeline:** Reactor Kafka helps in efficiently using application resources with multiple external interactions. An end-to-end reactive pipeline (a topic with messages) benefits from non-blocking back-pressure and an efficient use of threads so that a number of concurrent requests can be processed.

The following are the uses of Kafka:

- Kafka can be used for tracking website activities using events.

- Kafka can be used for collecting logs from various services and making them available to consumers.
- Some frameworks such as Spark Streaming read the data to the topic. It can also process it and then add to the new topic, which can be consumed by the consumer.

Reactive API for Kafka

The reactive API Reactor Kafka by Apache Kafka is based on Project Reactor. It enables publishing messages to Kafka topics, and they can be consumed using functional APIs which facilitate non-blocking back-pressure, and also support low overheads. Such applications help us to use Reactor Kafka as a message bus for the streaming platform. We can integrate it with other systems to provide an end-to-end reactive pipeline. The reactive pipelines support non-blocking back-pressure and efficiency in using threads to provide various concurrent requests to be processed efficiently. It allows the development of reactive applications with few overheads and the capacity to deliver low-latency and high-throughput pipelines.

Reactor Kafka components for handling reactive pipelines

The Reactor Kafka API consists of these two main classes:

- **Sender:** The class which publishes messages to the Kafka topics
- **Receiver:** The class for consuming messages from the Kafka topics

Sender

The Sender is a thread-safe class which allows sending the outbound messages to Kafka. The `KafkaProducer` is associated to `Sender`, which is used as a transporter for transporting the messages to Kafka.

Creating the Sender

`SenderOptions` is used to get an instance of `Sender`. Once `Sender` is created using `SenderOptions`, keep one thing in mind, whatever change we made after creating the `SenderOptions` will not be reflected by the `Sender`; it will still use the earlier properties. The `Sender` can be created as follows:

```
Map<String, Object> props = new HashMap<>();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "my-sender");
props.put(ProducerConfig.ACKS_CONFIG, "all");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    IntegerSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class);
SenderOptions<Integer, String> senderOptions =
    SenderOptions.create(props);
sender = KafkaSender.create(senderOptions);
```

Now, `Sender` can send the messages to Kafka, however, the connection has not been set yet. The lazy `KafkaProducer` is created only when the first message is ready for sending.

Creating messages

The instance of `SendRecord` represents an outbound message. Each `SendRecord` class consists of two components: one is an instance of `ProducerKafka`, and the second is correlation metadata, which matches the send results to the record. The `ProducerKafka` has the key-value pair and the name of the topic which will be sent to the Kafka. The following statements create a sequence of messages for sending to Kafka:

```
Flux<SendRecord<Integer, String>> messages = Flux.range(1, 3).  
    map(index -> SenderRecord.create(  
        new ProducerRecord("mytopic", "Packt Message_" + index), index))
```

Sending messages

The outbound messages created now can be sent to Kafka using `sender`, which we created in an earlier step, as shown here:

```
sender.<Integer>send(messages)
.doOnError(e -> e.printStackTrace()).subscribe(record -> {
    RecordMetadata data = record.recordMetadata();
    System.out.printf("Message %d sent successfully, topic-
        partition=%s-%d offset=%d timestamp=%s\n",
        record.correlationMetadata(), data.topic(),
        data.partition(), data.offset());
}).subscribe();
```

Closing the Sender

Once the operations on `sender` are done, and `sender` is no longer required, it can be closed by using the `close()` method. In turn, `KafkaProducer` will be closed, which closes all the connections with the clients, freeing all memory.

Receiver

Now that `Sender` has sent messages to Kafka, these stored messages can be consumed using the `Receiver` instance. Each `Receiver` is associated with an instance of `KafkaConsumer`. We need to be careful while handling `Receiver`, as it's not thread-safe, and so can't be accessed concurrently.

Creating a Receiver instance

The instance of `ReceiverOptions` facilitates the creation of a `Receiver` instance. Similar to `SenderOptions`, the changes made to `ReceiverOptions` will not be used by the already created `Receiver`. The properties set on `ReceiverOptions` will be passed down to the underlying `KafkaConsumer`. The following code shows creating an instance of `ReceiverOptions`:

```
Map<String, Object> props = new HashMap<>();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    "localhost:9092");
props.put(ConsumerConfig.CLIENT_ID_CONFIG, "my-consumer");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "group-1");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    IntegerDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    StringDeserializer.class);
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "smallest");
props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG,
    Integer.toString(Integer.MAX_VALUE));
ReceiverOptions<Integer, String> receiverOptions =
    ReceiverOptions.create(props);
```

Now that we have set the properties, and obtained an instance of `ReceiverOptions`, it's time to create a lazy `KafkaReceiver`, as shown by the following code:

```
ReceiverOptions<Integer, String> options =
    receiverOptions.subscription(Collections.singleton("mytopic"))
KafkaReceiver receiver=KafkaReceiver.create(options);
```

The created `KafkaReceiver` is bound to read the messages sent by `mytopic`.

Reading the messages

The underlying `KafkaConsumer` instance is created when the inbound `Flux` is subscribed to and ready to consume messages from Topic. The `receive()` method of `KafkaReceiver` returns a `Flux` of type `ReceiverRecord`. The record received consists of `ConsumerRecord` and an instance of `ReceiverOffset`:

```
Flux<ReceiverRecord<Integer, String>> kafkaReceiver =  
    KafkaReceiver.create(options).receive();
```

We manually need to acknowledge the offset after the message is processed as the unacknowledged offsets will not be committed. The code to print the read messages from the topic, and then to acknowledge that record, is as follows:

```
kafkaReceiver.subscribe(record -> {  
    ReceiverOffset offset = record.receiverOffset();  
    System.out.printf("Received message from the topic: ");  
    offset.acknowledge();  
});
```

Running the Kafka application

We just discussed `Sender` and `Receiver` as Kafka components and their creation. Follow the steps listed next to create an application which adds and receives messages using a `Sender` and a `Receiver`:

1. Creating a Sender

1. Create a class with a main method which will work as a sender. Name the class `MySender`.
2. Write the code using `KafkaSender` to create a `Sender`, as discussed earlier, to add messages to the topic named `mytopic`.

2. Creating a Receiver:

1. Create another class with a main method which will act as our receiver. Name the class `MyReceiver`, which will read the messages from the topic named `mytopic`.
2. Write the code using `KafkaReceiver` as discussed earlier.
3. **Setting up the Kafka server:** Follow the steps in the following section to install and set up the Kafka server:

1. Installing Zookeeper server:

Apache Kafka uses a Zookeeper instance for reliable distributed coordination, so let's first install it as follows:

1. Install Zookeeper from the link <http://zookeeper.apache.org/releases.html>.
2. Once the download is complete, unzip the file. I unzipped it at `d:\zookeeper`.
3. Rename the file `zoo_sample.cfg` to `zoo.cfg` in the `Zookeeper\conf` folder.
4. Open the file renamed `zoo.cfg` file, and edit `dataDir=/tmp/zookeeper` to `d:\zookeeper\zookeeper-3.4.9\data`.
5. Now, add `ZOOKEEPER_HOME` and the path in `System` variables.
6. Open the command prompt, and type `zkserver` to start Zookeeper server.

2. Downloading and starting Kafka Server:

Now it's time to install

the Kafka server. The following steps provide the instructions for downloading, installing and starting the server:

1. Download the **ZIP** file for Kafka from the link <http://kafka.apache.org/downloads.html>.
2. Unzip the file at any location, I unzipped it at `d:\kafka`.
3. Now, edit the `server.properties` file under `D:\kafka\kafka_2.10-0.10.2.1\config` to change the value of `log.dir` to `D:\kafka\kafka_2.10-0.10.2.1\kafka-logs`.
4. Open the `zookeeper.config` file in `D:\kafka\kafka_2.10-0.10.2.1\config` to change the value of `dataDir` to `D:\kafka\kafka_2.10-0.10.2.1\zookeeper-data`.
5. Open the command prompt, and run the following command from the home directory of Kafka:

```
.\bin\windows\zookeeper-server-start.bat  
.\config\zookeeper.properties
```

The preceding command will start the Zookeeper server.

6. Open another command prompt, and run the following command from the Kafka home directory to start the Kafka server:

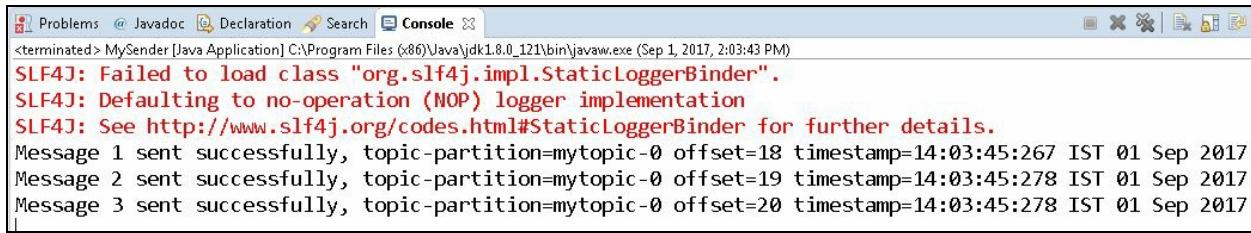
```
.\bin\windows\kafka-server-start.bat  
.\config\server.properties
```

3. **Creating topic:** The servers are started, so let's create a topic as follows:

1. Open another command prompt and to create `mytopic` as a topic, run the following command from Kafka Home:

```
.\bin\windows\kafka-topics.bat --create --zookeeper  
localhost:2181 --replication-factor 1  
--partitions 1 --topic mytopic
```

2. After creating the topic now, it's time to add messages to `mytopic`. Run the `MySender` class, which will send the message to Kafka. On execution, we will get the following output on the console:



```
Problems @ Javadoc Declaration Search Console
<terminated> MySender [Java Application] C:\Program Files (x86)\Java\jdk1.8.0_121\bin\javaw.exe (Sep 1, 2017, 2:03:43 PM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Message 1 sent successfully, topic-partition=mytopic-0 offset=18 timestamp=14:03:45:267 IST 01 Sep 2017
Message 2 sent successfully, topic-partition=mytopic-0 offset=19 timestamp=14:03:45:278 IST 01 Sep 2017
Message 3 sent successfully, topic-partition=mytopic-0 offset=20 timestamp=14:03:45:278 IST 01 Sep 2017
```

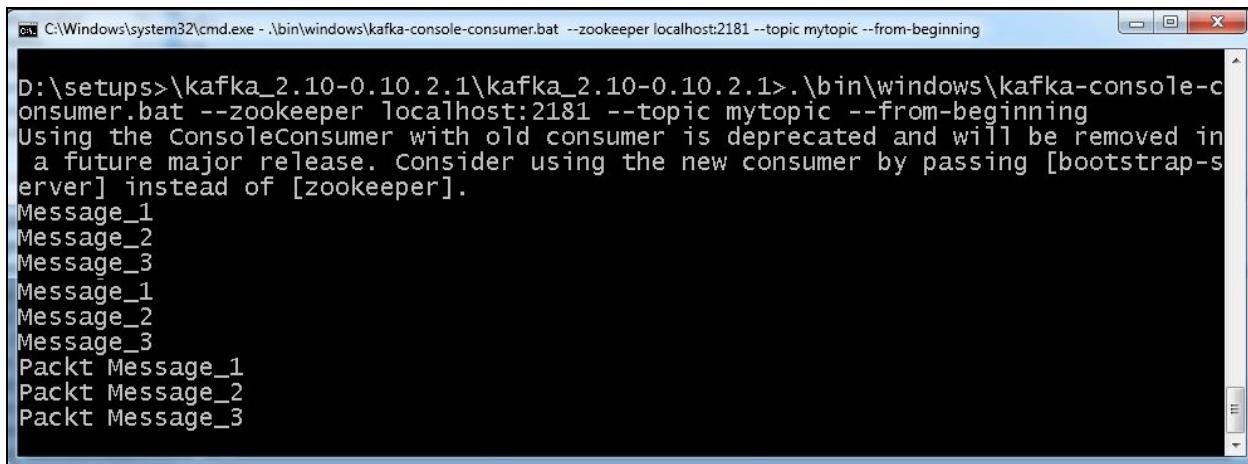
The output shows that three messages have been sent to Kafka.

4. Consuming the messages: We added the messages to the topic, it's time to consume the messages using one of the following ways:

1. Now, either run `MyConsumer.java` to read the messages.
2. Open another command prompt and run the following command to read the messages:

```
.\bin\windows\kafka-console-consumer.bat --zookeeper
localhost:2181 --topic mytopic --from-beginning
```

On execution, we will get the following output:



```
C:\Windows\system32\cmd.exe - .\bin\windows\kafka-console-consumer.bat --zookeeper localhost:2181 --topic mytopic --from-beginning

D:\>.\setups>\kafka_2.10-0.10.2.1\kafka_2.10-0.10.2.1>.\bin\windows\kafka-console-consumer.bat --zookeeper localhost:2181 --topic mytopic --from-beginning
Using the ConsoleConsumer with old consumer is deprecated and will be removed in
a future major release. Consider using the new consumer by passing [bootstrap-s
erver] instead of [zookeeper].
Message_1
Message_2
Message_3
Message_1
Message_2
Message_3
Packt Message_1
Packt Message_2
Packt Message_3
```

The output may vary if you run `MySender` more than once, or if the messages you sent are of different values. The value of the messages will be similar to one of those we have already set in `MySender.java`.

Summary

In this chapter, we started the discussion with Spring Data project, the modules supported by it, and its features such as easy to integrate multiple data stores, using the default CRUD functionality, and the support for auditing. We discussed in depth about the interfaces such as `Repository`, `CrudRepository`, and `ReactiveRepository` to create custom Spring Data repositories. While creating these repositories, we may need to add the methods for querying the repositories. As the query builder builds the queries from the method names, we also discussed naming the methods to obtain the result from the repository. We created an interface, and now, to implement the framework, we asked the framework to create proxies of the repository interfaces. The Spring Framework supports both XML-based as well as annotation-based proxy creation annotations such as `@EnableJpaRepositories`. The MongoDB and Lettuce connectors for Redis support the creation of the repositories which handle reactive APIs. `ReactiveCrudRepository` and `RxJava2CrudRepository` facilitate creating a custom data repository. We discussed creating the custom `ReactiveCrudRepository` practically. Towards the end of the chapter, we discussed Kafka, which is a distributed, scalable, fast system used for publishing and subscribing messages using `topic` for data pipelining. We created an application using `Sender` and `Receiver` to understand the concepts of adding messages to the topic and consuming them as well.

We have reached the end of this book. Throughout the book, we discussed various reactive concepts, data types, and frameworks for creating reactive applications. I have tried my best to give an orientation about reactive concepts. For me, this is the best time to end the discussion, as we have opened many threads for discussing the implementation of the concepts we discussed. As I always say, it's not the end; it's the beginning of a new Reactive world. Be active to learn Reactive. All the best!