title: 2. RxJS 中的响应(reacting with RxJS)

本章节覆盖

\* 将流视为主要工作单元

\* 了解函数式编程对 RxJS 的影响

\* 识别不同类型的数据源以及如何处理它们

\* 将数据源建模为 RxJS 中的 observable

\* observer 使用 observable

当以面向对象的方式编写代码时，我们被教会将问题分解为组件、交互和状态。这种拆解反复进行，并且拆分为许多层次，每个部分都进一步细分为更多的组件，直到最后我们达到实现交互明确定义的聚合类(cohesive class)。因此，在面向对象(object-oriented - OO)方式中，类是主要工作单元。每当创建一个组件时，就会同时具有与组件关联的状态，并且以结构化的方式操作该状态，来推进应用程序逻辑。例如，考虑一个典型的网上银行网站。银行系统包含的模块不仅仅封装提款、存款和转账相关的业务逻辑，还包含了存储和管理其他权限（例如账户和用户资料）的领域模型。操纵这个状态（产生的行为）会导致数据转化为所需的输出。换句话说，行为是由系统状态持续不断的变化所驱动的。如果一个系统是通过面向对象编程设计的，那么工作单元就是负责建模出账户、用户、金钱等的类。

RxJS 编程则略有不同。在一般的 RxJS 响应式编程中，基本的工作单元是流。

在本章中，我们要求你从流的角度思考（响应式）和设计代码，而不是将数据留存起来，这可以使数据流经(flow through)和应用(apply)转换规则，沿着转换规则到达到你想要的状态。你将学习如何处理不同类型的数据源，无论是静态还是动态，这是因为 RxJS 流使用基于 Observable 数据类型的一致计算模型。但是，与使用其他 JavaScript 库不同的是，在你的应用程序中使用 RxJS 意味着远远不止是要通过新的 API 实现整个应用程序；对于将你的问题，不是看作由类中的方法所操纵的状态集合的总和（译者注：上文提到的面向对象编程），而是通过一组实现你预期行为的操作符，将一系列数据从生产者持续不断地传递给消费者。（译者注：上文提到的流式编程）。

这种思维方式把时间的概念放在首位;这个概念作为暗流通过RxJS流的组件，并导致数据永远不会被存储，而只是短暂流动。将其与真实世界的物理水流关联起来，您可以将数据源看作流的顶部，而数据使用者看作流的底部。因此，数据总是沿着一个方向流向下游，就像河流中的水一样，在这个过程中，你可以建立控制大坝来改变这条河流的性质。以这种方式思考将帮助您理解数据应该如何在应用程序中移动。

这并不是说这种理解很容易实现——就像任何新技能一样，它必须随着时间的推移和对概念的迭代应用而建立起来。正如您在第1章中的伪流示例中看到的，对于大多数人来说，动态数据和保存在变量中的数据的概念是很难理解的。在这本书中，我们将为您提供必要的工具来简化学习曲线。为了开始构建您的工具包，本章为帮助您更好地理解流奠定了基础。RP背后的许多基本原则来自函数式编程，因此让我们从这里开始。

2.1函数式编程是反应式编程的支柱

支持RP的抽象构建在FP之上，因此FP是RP的基础。许多关于RP的炒作来自于开发社区和行业，他们意识到FP提供了一种引人注目的设计代码的方法。这就是为什么对你至少有一个FP原则的基本理解是重要的。如果您在函数式编程方面有坚实的背景知识，可以跳过这一节，但是我们建议您继续阅读，因为它将帮助您更好地理解RxJS背后的一些设计决策。

就像在第1章中一样，我们要求你快速浏览一下Reactive Extensions项目的主网站(http://reactivex.io)。在它里面，你会发现下面的定义:

ReactiveX是观察者模式、迭代器模式和函数式编程的最佳思想的组合。

您在第1章(生产者和消费者)中了解了观察者模式的主要组件;现在，您将了解关于Rx项目的其他部分，即函数编程和迭代器。下面的图(图2.1)更好地说明了这些范例之间的关系。

让我们从探索FP的基础开始。

图2.1 RP范例从FP构建并扩展。此外，它还利用了常见的设计模式，如迭代器和观察者。

### 2.1.1函数式编程

函数式编程是一种软件范例，强调使用函数来创建声明性的、不可变的和无副作用的程序。你是不是被“不可变”这个词绊倒了?我们同意你的看法;程序永远不会改变状态的概念有点令人费解。毕竟，这就是为什么我们把数据放在变量中，然后修改它们以满足我们的需要。到目前为止，您编写的所有面向对象或过程性应用程序代码都依赖于来回更改和传递变量来解决问题。那么，如果不这样做，你怎么能实现同样的目标呢?以钟表为例。当时钟从下午1点到2点的时候。毫无疑问，它正在改变，不是吗?但是从功能的角度来看，我们认为最好每秒钟返回一个新的时钟实例，而不是每秒钟一个单一的时钟实例变化。从理论上讲，两者会同时到达，最后都能得到一个状态。

RxJS借鉴了FP的许多原则，特别是函数链接、延迟求值和使用抽象数据类型编排数据流的概念。这些是通过可观察数据类型驱动RxJS流编程开发的一些设计决策。在开始之前，我们将解释刚才给出的FP定义的主要部分，然后展示一个涉及数组的快速示例。

再次重申，函数式程序具有以下特点:  
\*声明式——函数式代码有一种特殊的风格，它利用JavaScript的高阶函数来应用专门的业务逻辑。稍后您将看到，函数链(也称为管道)以惯用方式描述数据转换步骤。大多数人将SQL语法视为声明性代码的完美示例

\*不可变的——不可变的程序(这里指的是任何不可变的函数、模块或整个程序)是指在创建或声明变量后从不改变或修改数据的程序。这可能是一个难以理解的概念，特别是当您具有OO背景时。

函数式程序将数据视为不可变的、常数值。一个熟悉的模块的好例子是String类型，因为任何操作都不会改变它们所操作的字符串;相反，它们都返回新字符串。在本书中，您将看到我们使用的一个好实践是使用const限定所有变量，以创建良好的块作用域的不可变变量，这些变量不能被重新赋值。

这并不能解决不变性的所有问题，但是在全局共享数据和函数时，它提供了一些额外的支持。

\*副作用自由-有副作用的函数依赖于驻留在它自己局部作用域之外的数据。函数的作用域是由它的参数和其中声明的任何局部变量组成的。与此之外的任何交互(如读取文件、写入控制台、在HTML页面上呈现元素等)被认为是副作用，应该避免，或者至少应该隔离。在本书中，您将了解RxJS如何通过将有效的计算推入订阅服务器来处理这些问题。

一般来说，突变和副作用使功能不可靠和不可预测。也就是说，如果一个函数无意中改变了一个对象的内容，那么它将损害其他期望这个对象保持其原始状态的函数。对此的OO解决方案是封装状态并保护它不受系统其他组件的直接访问。相反，FP通过消除状态来处理状态，以便您的函数可以放心地依赖它运行。

例如，图2.2通过一个名为data的共享状态变量说明了doWork()和doMoreWork()两个函数之间的依赖关系。

这种耦合带来了一个问题，因为doMoreWork现在依赖于doWork先运行。可能会出现两个问题:

\* doMoreWork()的结果完全取决于doWork()的成功结果，而不取决于系统的其他部分更改该变量。

\*针对这个函数的单元测试不能像应该的那样独立完成，所以测试结果很容易受到测试用例运行顺序的影响(在第9章中，我们将更详细地探讨测试)。

由于对共享状态的依赖(副作用)，函数doWork()临时耦合到doMoreWork()。因此，必须在doMoreWork()之前调用doWork()，否则程序将停止工作。

共享变量，特别是在全局范围内，增加了代码推理的认知负荷，因为这些变量要求您在跟踪代码时跟踪它们。另一种考虑全局数据的方法是将其视为所有函数中的一个隐藏参数。因此，维护的状态越全局，维护代码就越困难。图2.2中的示例是一个明显的副作用，但它们并不总是如此清晰。考虑这个返回数值数组中最小值的普通函数

```

const lowest = arr => arr.sort().shift();

```

虽然这段代码对您可能无害，但它包含了一个可怕的副作用。你能认出来吗?此函数实际更改输入数组的内容，如下面的代码片段所示。如果你在其他地方使用了数组的第一个元素，它现在完全消失了:

```

let source = [3,1,9,8,3,7,4,6,5];

let result = lowest(source); //-> 1

console.log(source); //-> [3, 3, 4, 5, 6, 7, 8, 9] // The original array changed!

```

原来的数组改变了!

稍后，我们将讨论一个函数库，该库提供了一组丰富的函数，可以不变地使用数组，这样就不会无意中出现类似的情况。

如果您拥有共享数据结构并在不同组件中使用的并发异步进程，情况就会变得更糟。因为延迟是不可预测的，你需要嵌套你的函数调用或使用一些其他健壮的同步机制，以确保它们执行和改变这种状态在正确的顺序;否则，您将遇到随机且难以排除的错误。

幸运的是，JavaScript是单线程的，所以您不需要担心通过不同线程运行的共享状态。但是作为JavaScript开发人员，在与web工作人员或进行同步HTTP调用时，我们经常要处理并发代码。考虑图2.3中所示的琐碎但频繁的用例，它涉及到异步代码与同步代码的混合。这带来了巨大的挑战，因为后者假设在成功完成之前执行的函数，如果存在一些延迟，就不一定是这样。

幸运的是，JavaScript是单线程的，所以您不需要担心通过不同线程运行的共享状态。但是作为JavaScript开发人员，在与web工作人员或进行同步HTTP调用时，我们经常要处理并发代码。考虑图2.3中所示的琐碎但频繁的用例，它涉及到异步代码与同步代码的混合。这带来了巨大的挑战，因为后者假设在成功完成之前执行的函数，如果存在一些延迟，就不一定是这样。

在这个场景中，doAsyncWork()从服务器获取一些数据，这些数据永远不会在固定的时间内完成。因此doMoreWork()无法正常运行，因为它读取了尚未初始化的数据。回调和承诺可以帮助您解决这个问题，因此您不必为了预期延迟而硬编码自己的超时。直接处理时间是一种灾难，因为您的代码将非常脆弱，难以维护，并且会导致您在应用程序遇到比平时稍微多一点流量的情况下，不得不在周末加班。不变地处理数据，使用FP，以及像RxJS这样的异步库的帮助，可以使这些时间问题消失——不可变的变量受到时间的保护。在第4章和第6章中，我们将讨论具有可观察性的定时和同步，这为这个问题提供了更好的解决方案。

尽管JavaScript不是一种纯函数式语言，但通过一些规则和适当的库的帮助，您可以完全按功能使用它。当你学习使用RxJS，我们要求你也开始拥抱功能编码风格;这是我们坚信并在本书中所有代码示例中推广的东西。

除了使用const来保护变量的引用之外，JavaScript还支持通用的数组数据结构，其中包括map、reduce、filter等方法。这些函数称为高阶函数或一级函数，它们是该语言中最重要的函数特性之一，允许您以惯用方式表示JavaScript程序。高阶函数的定义是可以接受其他函数的参数，也可以返回其他函数;它们在RxJS中被广泛使用，就像在任何函数数据类型中一样。

下面的清单显示了一个简单的程序，它接受一个数字数组，提取偶数，计算它们的平方，并计算它们的总数。

> 清单2.1使用map、reduce和filter处理集合

```

const isEven = num => num % 2 === 0;

const square = num => num \* num;

const add = (a, b) => a + b;

const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

arr.filter(isEven).map(square).reduce(add); //-> 220

```

在本例中，由于这些操作是无副作用的，因此给定相同的输入数组，这个程序将始终生成相同的值(220)。

> 我在哪里可以找到这个代码?

> 本书的所有代码都可以在RxJS in Action GitHub知识库中找到，网址是https://github.com/RxJSInAction。在那里，您将找到两个子存储库。在rxjs-in-action中，您将发现一个简单的应用程序，其中包含第1章到第9章的所有单独章节清单的代码。所有示例都是可运行的RxJS代码片段，您可以与之交互。另外，在运行中的银行存储库下，您将发现我们的web应用程序，它展示了嵌入到React/Redux架构中的RxJS。我们在书中使用的一些api不允许跨源资源共享(CORS)。解决这个问题最简单的方法是通过安装扩展或附加组件在浏览器级别禁用它。

如果您想象一下必须使用非函数或命令式方法来编写这个程序，那么您可能需要编写一个循环、一个条件语句和一些变量来跟踪事情。另一方面，FP提高了抽象级别，并鼓励一种声明式编码风格，这种编码风格清楚地阐明了程序的目的，描述了它要做什么，而不是它如何做。在这个简短的程序中，没有出现循环、if/else或任何命令式控制流机制。

FP和RP中使用的一个主要主题是无循环编程。在清单2.1中，您利用了map、reduce和filter来隐藏手动循环结构—允许您通过函数的参数实现循环逻辑。此外，这些函数也是不可变的，这意味着每一步都创建新的数组，保持原始数组不变。

如果您想象一下必须使用非函数或命令式方法来编写这个程序，那么您可能需要编写一个循环、一个条件语句和一些变量来跟踪事情。另一方面，FP提高了抽象级别，并鼓励一种声明式编码风格，这种编码风格清楚地阐明了程序的目的，描述了它要做什么，而不是它如何做。在这个简短的程序中，没有出现循环、if/else或任何命令式控制流机制。

FP和RP中使用的一个主要主题是无循环编程。在清单2.1中，您利用了map、reduce和filter来隐藏手动循环结构—允许您通过函数的参数实现循环逻辑。此外，这些函数也是不可变的，这意味着每一步都创建新的数组，保持原始数组不变。

> 想了解更多关于函数式编程的知识吗?

> JavaScript的数组对象在函数编程中有特殊的位置，因为它作为一种称为仿函数的功能非常强大的数据类型。简单地说，仿函数是可以封装数据并公开映射方法的容器，该映射方法允许您对该数据进行不变的应用转换，如Array.map()方法所示。稍后您将看到，RxJS流遵循同样的类函数式设计。

> 函数式编程是一个庞大的主题。在这本书中，我们将只涵盖足够的FP来帮助你理解和精通RxJS和RP。如果你想了解更多关于FP和FP主题的信息，你可以在Luis Atencio的《JavaScript函数式编程》(Manning, 2016)中详细了解。

清单2.1所示的代码可以很好地使用数组，但也可以转换为流。沿着我们在第1章中讨论的伪流数据类型，看看数组和流在处理一些数字序列时是如何类似地工作的:

```

Stream([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

.filter(isEven)

.map(square)

.reduce(add)

.subscribe(console.log); //-> 220

```

你可以清楚地看到Rx是如何受到FP的启发的。我们所要做的就是将数组封装到流中，然后订阅它，以侦听从流的管道中声明的步骤序列中派生出来的计算值。这就好比说流是容器，您可以使用它将数据(事件)提升到它们的上下文中，以便您可以对这些数据应用一系列操作，直到达到您想要的结果。幸运的是，由于多年使用数组，您已经熟悉了这个概念。您可以将一个值提升到数组中，并将任何函数映射到它

```

['rxjs'].map(toUpper).map(slice(0, 2)).map(repeat(2)); //-> 'RXRX'

```

古希腊哲学家赫拉克利特曾经说过，你不能两次踏入同一条河流。他把这个表述表述为他关于变化和运动是宇宙中心组成部分的学说的一部分——万物都在不断地运动。这个史诗般的实现就是RxJS流的全部:当数据在流中连续流动和移动时，通过这种数据类型进行编排就是你正在学习的所谓流的数据类型。尽管流是动态的，但它是不可变的数据类型。一旦流被声明为封装数组、侦听鼠标点击或响应HTTP调用，您就不能对其进行修改或添加新值——必须在声明时进行更改。因此，您需要声明性地、不变地指定对象或值的动态行为。我们将在下一章再次讨论这个主题。

此外，该程序的业务逻辑是纯粹的，并利用映射到流上的无副作用的函数来将生成的数据转换为所需的结果。这样做的好处是，所有副作用都是隔离的，并推送给使用者(在本例中是登录到控制台)。这种关注点分离是理想的，可以保持业务逻辑的干净和纯粹。图2.4显示了生产者和消费者所扮演的角色。

图2.4生产者发出的事件通过无副作用函数的管道推送，这些函数实现程序的业务逻辑。这些数据流到负责使用和显示它的所有观察者。

从FP借鉴的另一个流设计原则是惰性求值。延迟求值意味着直到真正需要时才调用代码。换句话说，只有将函数的结果作为其他表达式的一部分使用时，函数才会求值。在下面的示例中，流一直处于空闲状态，直到订阅者(消费者)附加到它;只有这样，它才会发出值1-10:

```

Stream([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

.filter(isEven)

.map(square)

.reduce(add); // Nothing runs here because no subscriber is added.

```

当订阅者开始监听时，流将通过管道以单个、单向的流从生产者向消费者发出事件。如果您的函数有副作用(因为管道是单向运行的)，这是有益的，有助于确保函数调用的有序执行。这是不惜一切代价避免副作用的另一个原因，尤其是当您开始组合多个流时，因为事情可能会回到您首先试图摆脱的混乱状态。延迟计算是流的强制要求

> Reactive Manifesto被动的宣言

> 反应系统的关键原则之一是在不同的工作负载下保持漂浮的能力——即弹性。显然，这对架构和基础设施有许多影响，超出了本书的范围，但由此得出的一个推论是，无论您处理一个、100个或数千个事件，您使用的范式都不应该改变。RxJS提供了一个单一的计算模型来处理有限和无限的流。

> 反应性宣言(http://www.reactivemanifesto.org)是由一个致力于识别构建反应性系统的模式的工作组发布的。它与Rx库没有直接关系，但从哲学上讲，它们有许多共同点。

例如，如果没有惰性求值，像这样使用无限流的代码将导致应用程序耗尽内存并停止:

```

//1

Stream.range(1, Number.POSITIVE\_INFINITY) // Reads infinitely many numbers in memory

.take(100)

.subscribe(console.log);

//2

Stream.fromEvent('mousemove') // Listens to all mouse moves the user is performing

.map(e => [e.clientX, e.clientY])

.subscribe(console.log);

```

在示例1中，延迟求值使得流足够智能，能够理解在取前100之前，它不需要实际无限遍历所有正数。即使要存储的数字量很大，流也不会持久地保存数据;相反，任何发出的数据在它生成时立即广播给所有订阅者。在例子2中，假设你需要在内存中存储屏幕上所有鼠标移动的坐标;这可能会占用大量内存。RxJS不保留这些数据，而是让数据自由流动，并使用迭代器模式遍历任何类型的数据源，而不管它是如何创建的。

* ### 2.1.2 The iterator pattern迭代器模式

RxJS流背后的一个关键设计原则是为您提供一种熟悉的遍历机制，就像您使用数组一样。迭代器用于以结构无关的方式或独立于用于利用这些元素的底层数据结构(数组、树、映射甚至流)来遍历数据容器。此外，此模式可以有效地将应用于每个元素的业务逻辑与迭代本身解耦。目标是为访问每个元素并转移到下一个元素提供一个协议，如图2.5所示。

图2.5迭代器抽象了遍历机制，无论是for循环还是while循环，因此处理任何类型的数据都是完全相同的方式。

现在我们将简要解释此模式，稍后您将看到如何将其应用于流。JavaScript ES6(或ES2015)标准定义了迭代器协议，它允许您定义或定制任何可迭代对象的迭代行为。您最熟悉的可迭代对象是数组和字符串。ES6添加了Map和Set。使用RxJS，我们也将流视为可迭代的数据类型。

You can make any object iterable by manipulating its underlying iterator. We’ll be using some ES6-specific syntax to show this. Consider an iterator object that traverses an array of numbers and buffers a set amount of contiguous elements. Here, the business logic performed is the buffering itself, which can be useful to group elements together to form numerical sets of any dimension, like the ones illustrated in figure 2.6.

Now let’s see what the code would look like. The next listing shows the internal implementation of this custom iterator, which contains the buffer logic.

Figure 2.6 Using an iterator to display sets of numbers of size 2

> Listing 2.2 Custom BufferIterator function

```

function BufferIterator(arr, bufferSize = 2) { // Assigns a default buffer size of 2

this[Symbol.iterator] = function () { // Overrides the provided array’s iterator mechanism. Symbol.iterator represents the array’s iterator function.

let nextIndex = 0;

return {

next: () => { // The next() function is part of the Iterator interface and marks the next element in the iteration.

if (nextIndex >= arr.length) {

return {done: true}; // Returns an object with a done = true property, which causes the iteration mechanism to stop

} else {

let buffer = new Array(bufferSize);

for(let i = 0; i < bufferSize; i++) { // Creates a temporary buffer array to group contiguous elements

buffer[i] = (arr[nextIndex++]);

}

return {value: buffer, done: false}; // Returns the buffered items and a status of done = false, which indicates to the iteration mechanism to continue

}

}

}

};

}

```

Any clients of this API need only interact with the next() function, as outlined in the class diagram in figure 2.7. The business logic is hidden from the caller, the for...of block, which is the main goal of the iterator pattern.

The next() function in listing 2.2 is used to customize the behavior of the iteration through for...of or any other looping mechanism. As you’ll see later on, RxJS observers also implement a similar interface to signal to the stream to continue emitting elements.

\*\*DID ITERATORS THROW YOU FOR A LOOP?\*\* The ES6 iterator/iterable protocols are powerful features of the language. RxJS development predates this protocol, so it doesn’t use it at its core, but in many ways the pattern is still applied. We don’t use iterators in this book; nevertheless, we recommend you learn about them. You can read more about this protocol here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\_protocols#iterator.

Figure 2.7 A class diagram (UML) highlighting the components of the iterator pattern. The Iterator interface defines the next() function, which is implemented by any concrete iterator (BufferIterator). Users of this API need only interact with the interface, which is general and applies to any custom traversal mechanism.

Iterators allow you to easily take advantage of the JavaScript runtime to take care of the iteration on your behalf. Following, we show some examples of this using our simple numerical domain. Buffering is built into RxJS, and it’s really useful to gather up a sequence of events and make decisions about the nature of these events or apply additional logic. An example of this is when you need to invoke an expensive operation in response to a sequence of mouse events, like drag and drop. Instead of running expensive code at each mouse position, you buffer a specific number of them and emit a single response, taking all into account. Implementing this yourself would be tricky, because it would involve time management and keeping external state that tracks the frequency and speed with which the user moves the mouse; certainly, you’ll want to delegate this to libraries that understand how to manage all this for you. We’ll examine buffers in more detail in chapter 4. In RxJS, buffers aren’t implemented as in listing 2.2, but it serves to show you an example of how you can buffer data using iterators, which is how you think about these sorts of operations. Here’s our BufferIterator in action:

```

const arr = [1, 2, 3, 4, 5, 6];

for(let i of new BufferIterator(arr, 2)) { // Buffers two elements at once

console.log(i);

}

//-> [1, 2] [3, 4] [5, 6]

for(let i of new BufferIterator(arr, 3)) { // Buffers three elements at once. Notice how the iteration mechanism is completely separate from the buffering logic.

console.log(i);

}

//-> [1, 2, 3] [4, 5, 6]

```

When you subscribe to a stream, you’ll be traversing through many other data sources such as mouse clicks and key presses in the exact same way. Theoretically speaking, because our pseudo Stream type is an iterable object, you could traverse a set of key press events as well with a conventional loop:

```

const stream = Stream(R, x, J, S)[Symbol.iterator](); // Creating a stream that wraps key presses for those four letters

for(let keyEvent of stream) { // Traversing a stream is semantically equivalent to subscribing to it (more on this later).

console.log(event.keyCode);

}

//-> 82, 120, 74, 83

```

Streams in RxJS also respect the Iterator interface, and subscribers of this stream will listen for all the events contained inside it. As you saw previously, iterators are great at decoupling the iteration mechanism and data being iterated over from the business logic. When data defines the control flow of the program, this is known as data-driven code.

## 2.2 Stream’s data-driven approach

RxJS encourages a style of development known as data-driven programming. The data-driven approach is a way of writing code such that you can separate the behavior of an application from the data that’s passing through it. This is a core design decision of RxJS and the main reason why you can use the same paradigm to process arrays, mouse clicks, or data from AJAX calls.

In the OO approach, you place more emphasis on the supporting structures than the data itself. This explains why pure OO languages like Java have many different implementations to store a sequential collection of elements, each tackling different use cases: Array, ArrayList, LinkedList, DoublyLinkedList, ConcurrentLinkedList, and others. To put it another way, imagine that you run a local florist that performs deliveries. Your business in this case is importing flowers, cutting them, packaging them, handling orders, and sending those orders out for delivery. These tasks are all part of your business logic; that is, they are the important bits that your customers care about and the parts that bring in revenue. Now imagine that in addition to those tasks, you’re also tasked with designing the type of delivery van to use. Creating this structure is itself a full-time job and one that would likely distract from your primary business without meaningfully lending to it.

Data, as in the data that you care about and that which gives rise to search engines, websites, and video games, is the flower component of software design. Creating software should therefore be about how you manipulate data rather than how you create approximations of real-world objects (as you might in OO programming). Bringing data to the forefront and separating it from the behavior of the system is at the heart of data-driven/data-centric design. Similarly, loosely coupling functions from the objects that contain data is a design principle of FP and, by extension, RP.

To be driven by data is to be compelled to act by the presence of it and to let it fuel your logic. Without data to act on, behavior should do nothing. The idea of data giving life to behavior ties back to our earlier definition of what it means to be reactive— reacting to data instead of waiting for it. Streams are nothing more than a passive process that sits idle when nothing is pushed through them and no consumer is attached, as shown in figure 2.8.

This design pattern seems intuitive to most people because we think of data as requiring some sort of behavior in order to be meaningful. In a physics simulation, the mass of a ball is just a decimal number without context until the behavior of gravity is applied to it. Thus, if we are to imagine that both are intertwined by nature, it seems only natural that they should cohabitate logically within an object. In theory, this would seem to be a fairly obvious approach, and indeed the prevalence and popularity of OO programming stands testament to its power as a programming paradigm.

But it turns out that the greatest strength of OO design is also perhaps its greatest weakness. The intuition of representing components as objects with intrinsic behavior makes sense to a certain extent, but much like the real world, it can become difficult to reason about as the complexity of the application grows. For instance, if you hadn’t used the BufferIterator type before, you would’ve had to implement the buffering logic with the application logic that uses this data. To keep things simple, you just logged the numbers to the screen, but in real life you’ll use iterators for something more meaningful.

Figure 2.8 Initially, streams are lazy programs that wait for a subscriber to become available. Any events received at this point are discarded. Subscribing to the stream puts the wheels in motion, and event data flows through the pipeline and out for consumers to use.

The data-centric approach seeks to remedy this issue by separating the concerns of data and behavior, through its producer/consumer model. Data would be lifted out of the behavior logic and instead would pass through it. Behavior could be loosely linked such that the data moved from one part of the application to another, independent of the underlying implementation. Earlier you saw how iterators help with this:

```

Stream([1, 2, 3, 4, 5, 6])

.buffer(2)

.subscribe(console.log)); //-> [1, 2] [3, 4] [5, 6]

```

Each step in the pipeline resides within its own scope that’s externalized from the rest of the logic. In this case, you can see that just like iterators, the buffering step is done separately from the code acting on the data. By constructing it so, you’ve both declared the intent of each step and effectively decoupled the data from the underlying implementation, because each component reacts only to the step that preceded it.

Furthermore, producers come in all shapes and sizes. Event emitters are one of the most common ones; they’re used to respond to events like mouse clicks or web requests. Also, there are timer-based sources like `setTimeout` and `setInterval` that will execute a task at a specified point in the future. There are subtler ones such as arrays and strings, which you might recognize as collections of data but not necessarily producers of data.

Traditionally, when dealing with each of these data sources, you’ve been conditioned to think of them as requiring a different approach. For instance, event emitters require named event handlers, `Promises` require the continuation-passing “thenable” function, `setTimeout` needs a callback, and arrays need a loop in order to iterate through them. What if we told you that all of these data sources can be consolidated and processed in the exact same way?

## 2.3 Wrapping data sources with Rx.Observable

All along, we’ve been using a pseudo data type called Stream as a substitute for the real Rx.Observable type available in RxJS 5. We did this to help you understand the paradigm and what it means to think in streams, rather than focus on the specifics of the library. In this section, we’ll begin diving into the RxJS 5 APIs (for information about installing RxJS 5 on the client or on the server, please visit appendix A). Through the Rx.Observable type, you can subscribe to events produced from different types of data sources.

> ES7 SPECIFICATION One of the key design decisions behind the development of RxJS 5 was to create an Observable type that follows the proposed observable specification slated for the next version of JavaScript ES7. You can find all the details of this API here: https://github.com/zenparsing/es-observable.

You can lift a heterogeneous set of inputs into the context of an observable object. Doing so allows you to unlock the power of RxJS to transform or manipulate them to reach your desired outcome. First, let’s identify these different types of data.

### 2.3.1 Identifying different sources of data

We mentioned earlier that the advantage of separating data and behavior is that you can reason about a holistic model to account for any type of data. Hence, the first step to break the data free is to understand that all of these data sources are the same when viewed through a data-driven (or stream-driven?) lens. First, let’s re-categorize the types of data we’ll encounter. Rather than dealing with them as strict JavaScript types, let’s look at some broader categories of data.

\_\_EMITTED DATA\_\_

Emitted data is data that will be created as a result of some sort of interaction with the system; this can be either from a user interaction such as a mouse click or a system event like a file read. As we alluded to in chapter 1, some of these will have at most one event; that is, you request data and then, at some point in the future, you receive a response. For this, Promises can be a good solution. Others, like a user’s clicks and key presses, are part of a continuous process, and this requires you to treat them as event emitters that produce multiple discrete events at future times.

\_\_STATIC DATA\_\_

Static data is data that’s already in existence and present in the system (in memory); for example, an array or a string. Artificial unit test data also falls into this category. Interacting with it is usually a matter of iterating through it. If you were wrapping a stream around an array, for instance, the stream would never actually store the array; it would extend it with a mechanism that flushes the elements within the array (based on iterators). Arrays are a common and heavily used static data source, but you could also think of associative arrays or maps as unordered static data. Most of the examples so far have dealt with static data such as strings, numbers, and arrays, which we used to illustrate some of the basic concepts. In later parts of the book, we’ll focus on emitted data and generated data.

\_\_GENERATED DATA\_\_

Generated data is data that you create periodically or eventually, like a clock sounding a chime every quarter hour; it can also be something more procedural like generating the Fibonacci sequence using ES6 generators. In the latter case, because the sequence is infinite, it’s not feasible to store it all in memory. Instead, each value should be generated on the fly and yielded to the client as needed. In this category, you can also place the traditional setTimeout and setInterval functions, which use a timer to trigger events in the future.

Just like the saying, “When you’re a hammer, every problem looks like a nail,” the Rx.Observable data type can be used to normalize and process each of these data sources using a single programming model—it’s the hammer. With this approach, you gain the most code reuse and avoid creating specific ad hoc functions to deal with the idiosyncrasies of each event type.

### 2.3.2 Creating RxJS observables

In Rx, an observer subscribes to an observable. As you learned in chapter 1, this is analogous to the observer pattern with the subject acting as the observable; Rx.Observable represents the object that pushes notifications for observers to receive. The observers asynchronously react to any events emitted from the observable, which allows your application to remain responsive instead of blocking in the face of a deluge of events. This is ideal to implement asynchronous, responsive code both on the client and on the server.

Rx.Observable has different meanings to different people. To functional programming purists, it falls under a special category called a functor, an endofunctor to be exact. (We don’t cover functors in this book because they’re not essential to understanding Rx, but if you want learn more about them, you’ll find them in the functional programming book mentioned earlier.) To most others, it’s simply a data type that wraps a given data source, present in memory or eventually in the future, and allows you to chain operations onto it by invoking observable instance methods sequentially. Figure 2.9 shows a simple visualization of this concept.

Figure 2.9 The sequential application of methods or operators that transform an input into the desired outcome, which is what subscribers see

Here’s a quick look at how observables implement chaining extremely well:

```

Rx.Observable.from(<data-source>) // Wraps a data source with a stream

.operator1(...) // Invokes a sequence of operations chained by

.operator2(...) // the dot operator. In chapter 3, we’ll spend a lot

.operator3(...) // more time with observable instance methods.

.subscribe(<process-output>); // Processes the results

```

Whether you choose to accept one definition over the other, it’s important to understand that an observable doesn’t just represent a value now but also the idea of a value occurring in the future. In FP, this is the same definition given to pure functions, which are nothing more than to-be-computed values, and part of the reason why we refer to the “methods” invoked on an observable instance as operators.

Because observables in RxJS are immutable data types, this pattern works quite well and should not look that foreign to you. Consider a familiar data type, String. Look at this trivial example and notice its similarity to the previous pattern:

```

String('RxJS')

.toUpperCase()

.substring(0, 2)

.concat(' ')

.repeat(3)

.trim()

.concat('!') //-> "RX RX RX!"

```

Learning about a shiny new tool is always exciting, and there’s a tendency among developers to try to use that tool in every conceivable situation where it might potentially apply. But as is often the case, no tool is meant for every situation, and it’s just as important to understand where RxJS won’t be used.

You can divide your computing tasks into four groups within two different dimensions. The first dimension is the number of pieces of data to process. The second is the manner in which the data must be processed, that is, synchronously or asynchronously. In enumerating these possibilities, we want to highlight where RxJS would be most beneficial to your applications.

### 2.3.3 When and where to use RxJS

Learning to use a new tool is as important as learning when not to use it. The types of data sources we’ll be dealing with in this book can be classified into the four different categories listed in figure 2.10, which we’ll explain next.

Figure 2.10 Different types of data sources with examples in each quadrant

\_\_SINGLE-VALUE, SYNCHRONOUS\_\_

The simplest case is that you have only a single piece of data. In programming, you know there are operations that return a single value for each invocation. This is the category of any function that returns a single object. You can use the Rx.Observable .of() function to wrap a single, synchronous value. As soon as the subscriber is attached, the value is emitted (we haven’t yet explained the details behind subscribe, but we’ll cover that in a bit):

```

Rx.Observable.of(42).subscribe(console.log); //-> 42

```

Although there are cases where you’ll need to wrap single values, in most cases, if your goal is just to perform simple operations on them (concatenating another string, adding another number, and others), an observable wrapper may be overkill. The only time you’ll wrap simple values with observables is when they combine with other streams.

\_\_MULTI-VALUE, SYNCHRONOUS\_\_

You can also group single items together to form collections of data, mainly for arrays. In order to apply the same operation that you used on the single item on all of the items, you would traditionally iterate over the collection and repeatedly apply the same operation to each item in the collection. With RxJS, it works in exactly the same way:

```

Rx.Observable.from([1, 2, 3]).subscribe(console.log);

// -> 1

2

3

Rx.Observable.from('RxJS').subscribe(console.log);

// -> "R"

"x"

"J"

"S"

```

The RxJS from() operator is probably one of the most commonly used. And to make it a bit more idiomatic, RxJS has overloaded the forEach observable method as well, with the exact same semantics as subscribe:

```

const map = new Map();

map.set('key1', 'value1');

map.set('key2', 'value2');

Rx.Observable.from(map).forEach(console.log);

//-> ["key1", "value1"] ["key2", "value2"]

```

Both of these groups operate synchronously, which means each subsequent block of code must wait for the previous block to complete before executing. In the multi-value example, each item will be processed serially (one by one) until the collection is exhausted. This behavior is useful when dealing with items that have been preallocated, like arrays, sets, or maps, or if they can be generated, in place, on demand. Essentially, you can consider synchronous behavior to be actions on demand with results returning immediately (or at the very least before any further processing is done). When this is not the case, data is known as asynchronous.

\_\_SINGLE-VALUE, ASYNCHRONOUS\_\_

This brings us to the second dimension of computing tasks, where RxJS gives you the most benefits. This dimension addresses whether a task will execute synchronously or asynchronously. In the latter case, code is only guaranteed to run at some time in the future; thus, subsequent code blocks can’t rely on any execution of a previous block having already taken place. Like with the first dimension, you also have a single-value case, where the result of a task will result in a single return value. This kind of operation is usually used to load some remote resource via an AJAX call or wait on the result of some non-local calculation wrapped in a Promise, without blocking the application. In either case, after the operation is initiated, it will expect a single return value or an error.

As we mentioned previously, in JavaScript this case is often handled using Promises. A Promise is similar to the single-value data case in that it resolves or errors only once. RxJS has methods to seamlessly integrate with Promises. Consider this simple example of a Promise resolving into a single, asynchronous value:

```

const fortyTwo = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(42);

}, 5000);

});

Rx.Observable.fromPromise(fortyTwo)

.map(increment)

.subscribe(console.log); //-> 43

console.log('Program terminated');

```

> NOTE The promised value is being computed asynchronously, but Promises differ from Observables in that they’re executed eagerly, as soon as they’re declared.

Running this program as is produces the following output:

```

'Program terminated'

43 //-> after 5 seconds elapse

```

And because Promises are single-value and immutable, they’re never run again. So if you subscribe to one 10 seconds later, it will return the same value 10 times—this is a desirable trait of a Promise by design. In chapter 7, you’ll learn that you can retry a Promise Observable and force it to be executed many times by nesting it within another Observable, which has support for retries. Using the version of ajax(url) that returns a Promise, you can write the following:

```

Rx.Observable.fromPromise(ajax('/data'))

.subscribe(data => console.log(data.id));

```

Another frequently used alternative is to use jQuery’s deferred objects, which also implement the Promise interface. In particular, you can use functions like $.get(url) or $.getJSON(url):

```

Rx.Observable.fromPromise($.get('/data'))

.subscribe(data => console.log(data.id));

```

\_\_MULTI-VALUE, ASYNCHRONOUS\_\_

For those keeping score, this brings us to our fourth and final group of computing tasks. The tasks in the fourth group are those that will produce multiple values over time, yet do so asynchronously. You create this category especially for the DOM events, which are all asynchronous and can occur infinitely many times. This means that you’ll need a mix of semantics from both the iterator and the promise patterns. More specifically, you need a way to process infinitely many items in sequence and capture any errors that occur. These items could be data fetched from remote AJAX calls or data generated from dragging the mouse across the screen. For this you need to invert your control structures to operate asynchronously.

The typical solution to a problem of this nature would be to use an EventEmitter. It provides hooks or callbacks to which closures can be passed; in this way it’s very much like the Promise. But an event emitter doesn’t stop after a single event; instead, it can continue to invoke the registered callbacks for each event that arrives, creating a practically infinite stream of events. The emitter will fulfill both of your criteria for handling multi-value, asynchronous events. But it’s not without its share of problems. Though simple to use, event emitters don’t scale well for larger systems, because their simplicity leads to a lack of expressiveness. The semantics for unsubscribing and disposing of them can be cumbersome, and there’s no native support for error handling. These deficits can make it difficult to compose and synchronize complex tasks where multiple events from different parts of the system can be in flight simultaneously.

Rather, you can use RxJS to wrap event emitters, with all their benefits and versatility. The following code attaches a callback to a click event on a link HTML element:

```

const link = document.querySelector('#google'); // Queries the DOM for the link HTML element

const clickStream = Rx.Observable.fromEvent(link, 'click') // Creates an observable around click events on this link

.map(event => event.currentTarget.getAttribute('href')) // Extracts the link’s href attribute

.subscribe(console.log); //-> http://www.google.com

```

Note that in this example, the subscribe() method was used to process click events and perform the required business logic, in this case extracting the href attribute, as shown in figure 2.11. Later on, when we cover the Observable instance methods that form the pipeline, you’ll see concrete examples of how to decouple the business logic from the printing of the result.

Figure 2.11 Observable that wraps click events and passes them down to the observer for processing

You can also use Observables to wrap any custom event emitters. Going back to our calculator emitter in Node.js, instead of listening for the add event,

```

addEmitter.on('add', (a, b) => {

console.log(a + b); //-> Prints 5

});

```

you can subscribe to it:

```

Rx.Observable.fromEvent(addEmitter, 'add', (a, b) => ({a: a, b: b}))

.map(input -> input.a + input.b)

.subscribe(console.log); //-> 5

addEmitter.emit('add', 2, 3);

```

In this section, we covered only a few of the ways for creating Observables with RxJS.

Later on, we’ll tackle more-complex problems as well as new Observable methods.

### 2.3.4 To push or not to push

Event emitters have been around as long as the JavaScript language. In that time, they haven’t had any significant improvements to their interface in the latest releases of the language. This contrasts with Promises, iterators, and generators, which were part of the JavaScript ES6 specification and are already supported in many browsers at the time of writing. This is one of the reasons why RxJS is so important; it brings many improvements to JavaScript’s event system.

Figure 2.12 Notice the positions of the consumer and the direction of the data. In pull-based semantics, the consumer requests data (iterators work this way), whereas in pushed-based semantics, data is sent from the source to the consumer without it requesting it. Observables work this way.

Event emitters parse through a sequence of events asynchronously, so they come really close to being an iterator and, hence, a stream. The difference, however, lies in the way data is consumed by its clients—whether it is pulled or pushed. This is extremely important to understand, because most of the literature for RxJS defines observables as objects that represent push-based collections. Figure 2.12 highlights the main difference between the pull and push mechanisms, which we’ll explain immediately.

Iterators use a pull-based semantic. This means that the consumer of the iterator is responsible for requesting the next item from the iterator. This data-on-demand model has two major benefits. First, it creates an abstraction over the data structure that’s being used. Essentially, any data source that exposes some common method of iteration can be used interchangeably with another. The second benefit of data on demand is for sequences of data that result from some calculation. Such is the case with JavaScript generators.

For instance, for a Fibonacci number sequence, which is infinite, you need only calculate numbers as they’re requested rather than wasting computing time generating parts of a sequence that the caller doesn’t care about. This is immensely helpful if the data source is expensive or difficult to calculate. In the next listing, you use a generator to create a lazy Fibonacci calculator. Generators are nothing more than iterators behind the scenes, so each value will be produced only when the consumer calls (or pulls) the next() method.

> Listing 2.3 Fibonacci function using generators

```

function\* fibonacci() { // A generator function is denoted by the \* (star) notation.

let first = 1, second = 1; // Fibonacci sequence must be initialized with at least two values.

for(;;) { // yield will return the result of each intermediate step in the loop.

let sum = second + first;

yield sum;

first = second;

second = sum;

}

}

const iter = fibonacci(); // Creates the generator

console.log(iter.next()); //-> {value: 2, done: false}

console.log(iter.next()); //-> {value: 3, done: false}

console.log(iter.next()); //-> {value: 5, done: false}

```

> \_\_Want to learn more about generators?\_\_

> Generators are a language feature added into JavaScript as part of the ES6 specification. From a syntax point of view, generators introduce the function\* and yield keywords. A function with an asterisk declares that a function behaves as a generator, which means it can exit with a return value via yield and later reenter. Under the hood, generators don’t actually execute immediately but return an Iterator object, which is accessed via its next() method. Through this Iterator object, a generator can pause and resume exactly where it left off, and any context (closure) is kept across reentrances. A generator is a rare but powerful construct for producing infinite data using a given formula or template. If you want to learn more about them, we recommend you read the documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function\*.

A pull-based paradigm is useful in cases where you know that a value can be returned immediately from a computation. But in scenarios like listening for a mouse click, where the consumer has no way of knowing when the next piece of data will become available, this paradigm breaks down. For this reason, you require a corresponding type on the asynchronous side that is push-based—the opposite of the pull-based approach. In a push paradigm, the producer is responsible for creating the next item, whereas the consumer only listens for new events. As an example of this, consider your phone’s email client. A pull-based mechanism that checks for new email every second can drain the resources of your mobile device quickly, whereas with push email, or any push notifications for that matter, your email client needs to react to any incoming messages only once.

RxJS observables use push-based notifications, which means they don’t request data; rather, data is pushed onto them so that they can react to it. Push notifications bring the reactive paradigm to life. RxJS proposes observables as an improvement over event emitters because they’re more versatile and extensible. The observable also serves as a better contemporary to the Iterator type, given that it possesses similar semantics but with a push-based mechanism.

You can see from our discussion so far how iterators and Promises can be potential data sources that can be wrapped as observables, even though we earlier classified them as distinct groups. This ability to adapt not just the types they are replacing but also types from other groups is immensely powerful—observables work equally well across synchronous and asynchronous boundaries. It not only makes interfacing with legacy code incredibly easy, but also it allows consumer code to be written independently of how the producer is implemented.

> WATCH OUT! This power comes with responsibility as well, for although you’re able to convert anything your heart desires into Observables, it doesn’t always mean that you should. In particular, processes that are strictly synchronous and iterative or will only ever deal with a single value do not need to be “Rx-ified” just for the sake of being cool. Even though Observables are cheap to create, there’s a bit of overhead associated with applying simple operations on data. For instance, just transforming a string from lower-to-uppercase does not require it to be wrapped with an observable; you should directly use the string methods. Don’t be reactive just because you can.

In RxJS, you’ll always have a pipeline that takes data from the source to the corresponding consumer. Data will always be created or materialized from a data source. Again, the type of data source isn’t relevant to how your abstraction operates; when data reaches the end of its journey and must be consumed, it’s immaterial where the data came from. We’ll reiterate that the separation and abstraction of these two concepts, data production and data consumption, is important for three reasons:

\* It enables you to hide differences of implementation behind a common interface, which lets you focus more on the business logic of your task. This has the benefit of not only optimizing development time but also reducing code complexity by removing extra noise from code.

\* The separation of production and consumption builds a clear separation of concerns and makes the direction of data flow clear.

\* It makes streams testable by allowing you to attach mock versions of the producer and wire the corresponding matching expectations in the observer.

Now that you understand how streams can be constructed, you’re missing only the last place where observers come into play—stream consumption.

## 2.4 Consuming data with observers

Every piece of data that’s emitted and processed through an observable needs a destination. In other words, what was the purpose of capturing and processing a certain event? Observers are created within the context of a subscription, which means that the result of calling subscribe() on an observable source is a Subscription object. Because observables operate synchronously or asynchronously, the consumer of an observable must in some way support the inversion of control that also happens with callbacks. This is consistent with its push-based mechanism. That is, because you don’t know when a DOM element, for instance, will fire an event or when the result of an AJAX call will return, observables must be able to call into or signal the observer structure that more data is available by using the observer’s next() method, as illustrated in figure 2.13. This mechanism is directly inspired in the iterator and observer patterns. An iterator doesn’t know (or care) about the size of the data structure it’s looping over or if it will ever end; it only knows whether there’s more data to process.

Figure 2.13 Observables calling into an observer’s methods. Observers expose a simple iterator-like API with a next() method. Upon subscription, an object of type Subscription is returned to the calling code, which it can use for cancellation and disposal, as we’ll discuss in a bit.

Through a concise iterator-like API, observables are able to signal to their subscribers whether more events have occurred. This gives you the flexibility to control what data observers receive.

### 2.4.1 The Observer API

An observer is registered with an observable in much the same way that you registered callbacks on an event emitter. An observable becomes aware of an observer during the subscription process, which you’ve seen a lot of so far. The subscription process is a way for you to pass an observer reference into an observable, creating a managed, one-way relationship.

Figure 2.14 Observables call into the Observer API to send the next event in the stream, the completed flag when a stream has finished, or any errors that occur during the pipeline’s operation. We’ll discuss more about error handling in later chapters.

Figure 2.14 shows how observables call an observer’s methods to signal more data, completion, and even errors. As you can see, aside from next(), two other methods are called on observers: error() and complete().

Figure 2.14 shows that once the subscribe method is called, an observer is implicitly created with an API that exposes three (optional) methods: next, complete, and error (in RxJS 4 these were called onNext, onCompleted, and onError, respectively). In code, the resulting object has the following structure:

```

const observer = {

next: function () {

// process next value

},

error: function () {

// alert user

},

complete: function () {

}

}

```

Up until now, you’ve used a single function call only to process the results. This function maps to next(). Each method serves a specific purpose in the lifetime of the observer, as shown in table 2.1.

Alternatively, you can use this API directly by creating your own observable.

Table 2.1 Defining the Observer API

Name | Description

-- | --

next(val):void | Receives the next value from an upstream observable. This is the equivalent of update in the observer pattern. When a single function is passed into subscribe() instead of an observer object, it maps to the observer’s next().

complete():void | Receives a completion notification from the upstream observable. Subsequent calls to next(), if any, are ignored.

error(exception):void | Receives an error notification from the upstream observable. This indicates that it encountered an exception and won’t be emitting any more messages to the observer (subsequent calls to next() are ignored). Generally, error objects are passed in, but you could customize this to pass other types as well.

### 2.4.2 Creating bare observables

Most of the time, you’ll use the RxJS factory operators like from() and of(), as you learned at the beginning of this chapter, to instantiate observables. In practice, these should cover all your needs. But it’s important to understand how observables work under the nice RxJS abstraction and how they interact with the observer to emit events. We’ll show you a barebones model of an observable that emits events asynchronously and exposes the mechanism to unsubscribe. At the core, an observable is a function that processes a set of inputs and returns a subscription to the caller to manage the disposal of the stream:

```

const observable = events => {

const INTERVAL = 1 \* 1000;

let schedulerId;

return {

subscribe: observer => {

schedulerId = setInterval(() => {

if(events.length === 0) {

observer.complete();

clearInterval(schedulerId);

schedulerId = undefined;

} else {

observer.next(events.shift());

}

}, INTERVAL);

return {

unsubscribe: () => {

if(schedulerId) {

clearInterval(schedulerId);

}

}

};

}

}

};

```

You can call this function by passing the observer object:

```

let sub = observable([1, 2, 3]).subscribe({

next: console.log,

complete: () => console.log('Done!')

});

//-> 1

(...1 second)

2

(...1 second)

3

(...1 second)

Done!

```

This is a simplistic model of RxJS, and there’s much more that goes into it. But the main takeaway here is that an observable behaves like a function that begins chipping away at the data pushed into it as soon as a subscriber is available; the subscriber has the key to turn the stream off via sub.unsubscribe(). Now, let’s move on to using RxJS.

Using RxJS, you can register an observer object through Rx.Observable.create(). Like the previous code, this function expects an observer object that you can use to signal the next emitted event by invoking its next() method. Most of the time, you’ll provide the observer object literal directly into the subscription and use the static create() method when you want full control of how and when the data is emitted from the observable through the Observer API. For instance, you create observables artificially by calling into the observer’s methods directly:

```

const source$ = Rx.Observable.create(observer => {

observer.next('4111111111111111');

observer.next('5105105105105100');

observer.next('4342561111111118');

observer.next('6500000000000002');

observer.complete(); // If an observable is finite, you can signal its completion by calling the observer’s complete() method.

}); // At this point, the observable stands idle and none of the data is emitted or passed into the observer.

const subscription = source$.subscribe(console.log); // With subscribe(), the observer logic is executed; in this case, it’s printing to the console.

A marble diagram of this stream would look like figure 2.15.

```

Figure 2.15 A marble diagram showing a synchronous set of events ended by a call to complete()

This sample code is simple because it just emits a series of account numbers, but

you could do much more. You could create your own observables with custom behavior that can be reused anywhere in your application.

### 2.4.3 Observable modules

Directly calling the observer object allows you to define the data that’s pushed to the subscriber. How this data is generated and where it comes are encapsulated into the observable’s context—kind of like a module. For instance, suppose you wanted to create a simple progress indicator widget that can be used when a user is performing a long-running operation. This module will emit percentage values 0% to 100% at a certain speed, as shown in the following listing.

> Listing 2.4 Custom progress indicator module using RxJS

```

const progressBar$ = Rx.Observable.create(observer => {

const OFFSET = 3000;

const SPEED = 50;

let val = 0;

function progress() {

if(++val <= 100) {

observer.next(val); // Emits a new progress value every 50 milliseconds

setTimeout(progress, SPEED); // Calls the progress function recursively

} else {

observer.complete(); // Sends the complete signal after reaching 100%

}

};

setTimeout(progress, OFFSET); // Starts the progress indicator counter after three seconds

});

const label = document.querySelector('#progress-indicator');

progressBar$

.subscribe(

val => label.textContent = (Number.isInteger(val) ? val + "%" : val),

error => console.log(error.message),

() => label.textContent = 'Complete!'

);

```

The business logic of how the values are generated and emitted belongs in the observable, whereas all the details of rendering, whether you want a simple number indicator or use some third-party progress bar widget, are for the caller to implement within the observer.

> NOTE You could also achieve this by using RxJS’s time operators. More about

this in the next chapter.

Using these methods gives you more opportunities to react to the different states of the program. Stepping back into our discussion about iterators and generators in chapter 2, observers operate similarly to these artifacts. The key difference is that the iterator uses a pull-based mechanism as opposed to an observable’s push-based nature—an observable pushes values into an observer. For iterators and generators, the consuming code is controlling the pace of consumption. For instance, a for loop controls (or requests) what to pull from an iterator or a generator, not the other way around. This means that each time a new piece of data is needed (by a call to next() or yield), the consumer of the iterator will call the appropriate method to advance the state of the iterator. Figure 2.16 shows another example using the Fibonacci sequence.

```

for (let nums of new BufferIterator(arr, 2)) {

console.log(nums); // The loop pulls the next element from the iterator by calling .next().

}

for(let num of fibonacci()){

console.log(num); // The loop pulls data from the generator function, requesting it to yield the next element.

}

```

Figure 2.16 The pull mechanism of iterators

As a result, iterators must have a way to inform the consumer that there are no longer any items for consumption. Bank tellers are real-world iterators. Each time a customer comes up, that person must be handled before the next customer can be helped. When the teller becomes available, they yell “Next!” to “pull” the next customer in. If they were to call “Next!” and no one responded, they would know that the line was complete and it might be safe to take their lunch break.

Something to keep in mind, though, is that infinite event emitters, like the DOM, will never fire the complete() function (or error() for that matter) on any of its events. Therefore, it’s entirely up to you to unsubscribe from them or roll your own autodispose mechanism. But for finite event sequences, when an observer is called with either of these methods, it knows that contractually it won’t receive any more messages from its owning observable. This again is a tight parallel to an iterator, which by definition should stop returning values when the iteration generates an exception or completes.

Consider a simple Promise object that resolves to the value 42 after 5 seconds (shown in figure 2.17).

Figure 2.17 An observable (wrapped Promise) that emits a value after 5 seconds

We mentioned in chapter 1 that Promises can be used to model an immutable, single (future) value. You’ll use the setTimeout() function to simulate this; now, instead of creating your own observable, you’ll use the generic creational methods in RxJS, such as the following:

```

Rx.Observable.fromPromise():

const computeFutureValue = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(42);

}, 5000);

});

Rx.Observable.fromPromise(computeFutureValue)

.subscribe(

val => {

console.log(val);

},

err => {

console.log(`Error occurred: ${err}`);

},

() => {

console.log('All done!');

}

);

```

Because Promises emit a single value, this stream will eventually send the completed status after 5 seconds have passed, printing “All done!” at the end. Now, suppose that instead of a resolved Promise, something goes wrong in computing this value and the Promise is rejected:

```

const computeFutureValue = new Promise((resolve, reject) => {

setInterval(() => {

reject(new Error('Unexpected Exception!'));

}, 5000);

});

```

This will cause the observable to invoke the error() method on the observer and

print the following message after 5 seconds:

```

"Error occurred: Unexpected Exception!"

```

This is quite remarkable because RxJS not only takes care of error handling for you (without messy, imperative try/catch statements) but also provides logic that ties in with Promise semantics of resolve/reject. We’ll cover all there is to know about error handling in chapter 7.

An important takeaway from this discussion about observers is that the callbacks passed to it are, for all practical purposes, future code. That is, you don’t know when the callbacks will actually be called, so other code shouldn’t make assumptions about their execution. This relates to the larger point made earlier about the nature of the code within a stream. Because one of your goals is to move away from the messy business of keeping track of state changes, avoiding the introduction of side effects is one of the ways that you can keep your streams pure and prevent unwanted changes from adversely seeping into the application logic. This works well with RxJS because pure functions can run in any order and at any time (now or in the future) and will always yield the correct results.

With observers, we’ve finish introducing the three main parts of RxJS: producers (observables), the pipeline (business logic), and consumers (observers). This chapter is just the start of your journey of learning how to think reactively (and functionally). It will take much more time and many more examples to truly understand how you can think reactively, but you were able to get your feet wet on some advanced APIs. Much of what you’ve seen so far has been abstract in nature with very little coding, but this step is crucial for understanding how this approach differs from ones you’ve been taught in the past. In the next chapter, we’ll look more closely at the operations that you can perform on streams as well as how you can cancel them if needed. By doing so, we’re officially taking the training wheels off and introducing you to the core operations for building applications in RxJS.

## 2.5 Summary

\* RxJS and, more generally, the concept of thinking in streams derive many of their foundational principles from functional programming.

\* The declarative style of RxJS allows you to translate almost exactly from your problem statement into working code.

\* Data sources can often operate quite differently, even within the observable contract.

\* Mouse clicks, HTTP requests, or simple arrays are all the same under the eyes of observables.

\* Push-based and pull-based semantics are represented through observables and iterators, respectively. Wrapping data sources is the first step in creating a pipeline/observable.

\* Observables abstract the notion of production and consumption of events such that you can separate production, consumption, and processing into completely self-contained constructs.

\* Observers expose an API with three methods: next(), complete(), and error().