There are three ways to implement a custom combiner, as follows:

- **Merging**: Some data structures have an efficient merge operation that can be used to implement the `combine` method.
- **Two-phase evaluation**: Here, elements are first partially sorted into buckets that can be efficiently concatenated, and placed into the final data structure once it is allocated.
- **Concurrent data structure**: The += method is implemented by modifying a concurrent data structure shared between different combiners, and the `combine` method does not do anything.

Most data structures do not have an efficient merge operation, so we usually have to use two-phase evaluation in the combiner implementation. In the following example, we implement the combiners for parallel strings using two-phase evaluation. The `ParStringCombiner` class contains a resizable array, called `chunks`, containing `StringBuilder` objects. Invoking the += method adds a character to the rightmost `StringBuilder` object in this array:

```
class ParStringCombiner extends Combiner[Char, ParString] {
  private val chunks = new ArrayBuffer += new StringBuilder
  private var lastc = chunks.last
  var size = 0
  def +=(elem: Char) = {
    lastc += elem
    size += 1
    this
  }
```

The `combine` method takes the `StringBuilder` objects of the `that` combiner, and adds them to the `chunks` array of the `this` combiner. It then returns a reference to the `this` combiner:

```
  def combine[N <: Char, NewRepr >: ParString]
    (that: Combiner[U, NewTo]) = {
    if (this eq that) this else that match {
      case that: ParStringCombiner =>
        size += that.size
        chunks ++= that.chunks
        lastc = chunks.last
        this
  }
```

Finally, the `result` method allocates a new `StringBuilder` object and adds the characters from all the chunks into the resulting string:

```
    def result: ParString = {
      val rsb = new StringBuilder
      for (sb <- chunks) rsb.append(sb)
      new ParString(rsb.toString)
    }
  }
```

We test the performance of the parallel `filter` method with the following snippet:

```
val txt = "A custom txt" * 25000
val partxt = new ParString(txt)
val seqtime = warmedTimed(250) { txt.filter(_ != ' ') }
val partime = warmedTimed(250) { partxt.filter(_ != ' ') }
```

Running this snippet on our machine takes 11 milliseconds for the sequential version, and 6 milliseconds for the parallel one.

Summary

In this chapter, we learned how to use parallel collections to improve program performance. We have seen that sequential operations on large collections can be easily parallelized and learned the difference between parallelizable and non-parallelizable collections. We investigated how mutability and side effects impact correctness and determinism of parallel operations and saw the importance of using associative operators for parallel operations. Finally, we studied how to implement our custom parallel collection class.

We also found, however, that tuning program performance is tricky. Effects such as memory contention, garbage collection, and dynamic compilation may impact the performance of the program in ways that are hard to predict by looking at the source code. Throughout this section, we urged you to confirm suspicions and claims about program performance by experimentally validating them. Understanding the performance characteristics of your program is the first step toward optimizing it.

Even when you are sure that parallel collections improve program performance, you should think twice before using them. Donald Knuth once coined the phrase *Premature optimization is the root of all evil*. It is neither desirable nor necessary to use parallel collections wherever possible. In some cases, parallel collections give negligible or no increase in speed. In other situations, they could be speeding up a part of the program that is not the real bottleneck. Before using parallel collections, make sure to investigate which part of the program takes the most time, and whether it is worth parallelizing. The only practical way of doing so is by correctly measuring the running time of the parts of your application. In `Chapter 9`, *Concurrency in Practice*, we will introduce a framework called ScalaMeter, which offers a more robust way to measure program performance than what we saw in this chapter.

This chapter briefly introduced concepts such as Random Access Memory, cache lines, and the MESI protocol. If you would like to learn more about this, you should read the article, *What Every Programmer Should Know About Memory*, by Ulrich Drepper. To gain a more in-depth knowledge about the Scala collections hierarchy, we recommend you to search for the document entitled *The Architecture of Scala Collections*, by Martin Odersky and Lex Spoon, or the paper *Fighting Bit Rot with Types*, by Martin Odersky and Adriaan Moors. To understand how data-parallel frameworks work under the hood, consider reading the doctoral thesis entitled *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*, by Aleksandar Prokopec.

So far, we've assumed that all the collection elements are available when the data-parallel operation starts. A collection does not change its contents during the data-parallel operation. This makes parallel collections ideal in situations where we already have the dataset, and we want to process it in bulk. In other applications, data elements are not immediately available, but arrive asynchronously. In the next chapter, we will learn about an abstraction called an event stream, which is used when asynchronous computations produce multiple intermediate results.

Exercises

In the following exercises, you will use data-parallel collections in several concrete parallel collection use cases, and implement custom parallel collections. In all examples, a special emphasis is put on measuring the performance gains from parallelization. Even when it is not asked for explicitly, you should ensure that your program is not only correct but also faster than a corresponding sequential program:

1. Measure the average running time for allocating a simple object on the JVM.
2. Count the occurrences of the whitespace character in a randomly generated string, where the probability of a whitespace at each position is determined by a `p` parameter. Use the parallel `foreach` method. Plot a graph that correlates the running time of this operation with the `p` parameter.
3. Implement a program that renders the Mandelbrot set in parallel.
4. Implement a program that simulates a cellular automaton in parallel.
5. Implement a parallel *Barnes-Hut N-body* simulation algorithm.
6. Explain how you can improve the performance of the `result` method in the `ParStringCombiner` class, as shown in this chapter. Can you parallelize this method?
7. Implement a custom splitter for the binary heap data structure.
8. The binomial heap, described in the doctoral thesis of Chris Okasaki entitled *Purely Functional Data Structures*, is an immutable data structure that efficiently implements a priority queue with four basic operations: insert the element, find the smallest element, remove the smallest element, and merge two binomial heaps:

   ```
   class BinomialHeap[T] extends Iterable[T] {
     def insert(x: T): BinomialHeap[T]
     def remove: (T, BinomialHeap[T])
     def smallest: T
     def merge(that: BinomialHeap[T]): BinomialHeap[T]
   }
   ```

   Implement the `BinomialHeap` class. Then, implement splitters and combiners for the binomial heap, and override the `par` operation.

9. Implement the `Combiner` trait for the Red-Black tree from the Scala standard library. Use it to provide a parallel version of the `SortedSet` trait.

10. Implement a `parallelBalanceParentheses` method, which returns `true` if the parentheses in a string are properly balanced, or `false` otherwise. Parentheses are balanced if, going from left to right, the count of left parenthesis occurrences is always larger than, or equal to, the count of right parenthesis occurrences, and the total count of the left parentheses is equal to the total count of the right parentheses. For example, string `0(1)(2(3))4` is balanced, but strings `0)2(1(3)` and `0((1)2` are not. You should use the `aggregate` method.

6 Concurrent Programming with Reactive Extensions

> *"Your mouse is a database."*
>
> *- Erik Meijer*

The futures and promises from `Chapter 4`, *Asynchronous Programming with Futures and Promises*, push concurrent programming to a new level. First, they avoid blocking when transferring the result of the computation from the producer to the consumer. Second, they allow you to idiomatically compose simple future objects into more complex ones, resulting in programs that are more concise. Futures encapsulate patterns of asynchronous communication in a way that is clear and easily understandable.

One disadvantage of futures is that they can only deal with a single result. For HTTP requests or asynchronous computations that compute a single value, futures can be adequate, but sometimes we need to react to many different events coming from the same computation. For example, it is cumbersome to track the progress status of a file download with futures. Event streams are a much better tool for this use case; unlike futures, they can produce any number of values, which we call events. First-class event streams, which we will learn about in this chapter, can be used inside expressions as if they were regular values. Just as with futures, first-class event streams can be composed and transformed using functional combinators.

In computer science, **event-driven programming** is a programming style in which the flow of the program is determined by events such as external inputs, user actions, or messages coming from other computations. Here, a user action might be a mouse click, and an external input can be a network interface. Both futures and event streams can be classified as event-driven programming abstractions.

Reactive programming, which deals with the propagation of change and the flow of data in the program, is a closely related discipline. Traditionally, reactive programming is defined as a programming style that allows you to express various constraints between the data values in the program. For example, when we say `a = b + 1` in an imperative programming model, it means that `a` is assigned the current value of `b` increased by `1`. If the value `b` later changes, the value of `a` does not change. By contrast, in reactive programming, whenever the value `b` changes, the value `a` is updated using the constraint `a = b + 1`. With the rising demand for concurrency, the need for event-driven and reactive programming grows even larger. Traditional callback-based and imperative APIs have shown to be inadequate for this task: they obscure the program flow, mix concurrency concerns with program logic, and rely on mutable state. In larger applications, swarms of unstructured callback declarations lead to an effect known as the callback hell, in which the programmer can no longer make sense of the control flow of the program. In a way, callbacks are the `GOTO` statement of reactive programming. **Event stream composition** captures patterns of callback declarations, allowing the programmer to express them more easily. It is a much more structured approach for building event-based systems.

Reactive Extensions (**Rx**) is a programming framework for composing asynchronous and event-driven programs using event streams. In Rx, an event stream that produces events of type `T` is represented with the type `Observable[T]`. As we will learn in this chapter, the Rx framework incorporates principles present both in reactive and in event-driven programming. The fundamental concept around Rx is that events and data can be manipulated in a similar way.

In this chapter, we will study the semantics of `RxObservable` objects, and learn how to use them to build event-driven and reactive applications. Concretely, we will cover the following topics:

- Creating and subscribing to the `Observable` objects
- The observable contract and how to implement custom `Observable` objects
- Using the subscriptions to cancel event sources
- Composing observable objects using Rx combinators
- Controlling concurrency with Rx scheduler instances
- Using Rx subjects for designing larger applications

We will start with simple examples that show you how to create and manipulate the `Observable` objects, and illustrate how they propagate events.

Creating Observable objects

In this section, we will study various ways of creating `Observable` objects. We will learn how to subscribe to different kinds of event produced by `Observable` instances and learn how to correctly create custom `Observable` objects. Finally, we will discuss the difference between cold and hot observables.

An `Observable` object is an object that has a method called `subscribe`, which takes an object called an observer as a parameter. The observer is a user-specified object with custom event-handling logic. When we call the `subscribe` method with a specific observer, we can say that the observer becomes subscribed to the respective `Observable` object. Every time the `Observable` object produces an event, its subscribed observers get notified.

The Rx implementation for Scala is not a part of the Scala standard library. To use Rx in Scala, we need to add the following dependency to our `build.sbt` file:

```
libraryDependencies +=
  "com.netflix.rxjava" % "rxjava-scala" % "0.19.1"
```

Now, we can import the contents of the `rx.lang.scala` package to start using Rx. Let's say that we want to create a simple `Observable` object that first emits several `String` events and then completes the execution. We use the `items` factory method on the `Observable` companion object to create an `Observable` object `o`. We then call the `subscribe` method, which is similar to the `foreach` method on futures introduced in `Chapter 4`, *Asynchronous Programming with Futures and Promises*. The `subscribe` method takes a callback function and instructs the `Observable` object `o` to invoke the callback function for each event that is emitted. It does so by creating an `Observer` object behind the scenes. The difference is that, unlike futures, the `Observable` objects can emit multiple events. In our example, the callback functions print the events to the screen by calling the `log` statement, as follows:

```
import rx.lang.scala._
object ObservablesItems extends App {
  val o = Observable.items("Pascal", "Java", "Scala")
  o.subscribe(name => log(s"learned the $name language"))
  o.subscribe(name => log(s"forgot the $name language"))
}
```

Upon running this example, we notice two things. First, all the `log` statements are executed on the main program thread. Second, the callback associated with the first `subscribe` call is invoked for all three programming languages before the callback associated with the second `subscribe` call is called for these three languages:

```
run-main-0: learned the Pascal language
run-main-0: learned the Java language
run-main-0: learned the Scala language
run-main-0: forgot the Pascal language
run-main-0: forgot the Java language
run-main-0: forgot the Scala language
```

We can conclude that the `subscribe` call executes synchronously–it invokes callback for all the events emitted by the event stream `o` before returning. However, this is not always the case. The `subscribe` call can also return the control to the main thread immediately, and invoke the callback functions asynchronously. This behavior depends on the implementation of the `Observable` object. In this Rx implementation, the `Observable` objects created using the `items` method have their events available when the `Observable` object is created, so their `subscribe` method is synchronous.

In the previous example, the `Observable` object feels almost like an immutable Scala collection, and the `subscribe` method acts as if it is a `foreach` method on a collection. However, the `Observable` objects are more general. We will see an `Observable` object that emits events asynchronously next.

Let's assume that we want the `Observable` object that emits an event after a certain period of time has elapsed. We use the `timer` factory method to create such an `Observable` object and set the timeout to 1 second. We then call the `subscribe` method with two different callbacks, as shown in the following code snippet:

```
import scala.concurrent.duration._
object ObservablesTimer extends App {
  val o = Observable.timer(1.second)
  o.subscribe(_ => log("Timeout!"))
  o.subscribe(_ => log("Another timeout!"))
  Thread.sleep(2000)
}
```

This time, the `subscribe` method calls are asynchronous; it makes no sense to block the main thread for an entire second and wait until the timeout event appears. Running the example shows that the main thread continues before the callback functions are invoked:

```
RxComputationThreadPool-2: Another timeout!
RxComputationThreadPool-1: Timeout!
```

Furthermore, the `log` statements reveal that the callback functions are invoked on the thread pool internally used by Rx, in an unspecified order.

> TIP: The `Observable` objects can emit events either synchronously or asynchronously, depending on the implementation of the specific `Observable` object.

As we will see, in most use cases, events are not available when calling the `subscribe` method. This is the case with UI events, file modification events, or HTTP responses. To avoid blocking the thread that calls the `subscribe` method, the `Observable` objects emit such events asynchronously.

Observables and exceptions

In `Chapter 4`, *Asynchronous Programming with Futures and Promises*, we saw that asynchronous computations sometimes throw exceptions. When that happens, the `Future` object associated with the exception fails; instead of being completed with the result of the computation, the `Future` object is completed with the exception that failed the asynchronous computation. The clients of the `Future` objects can react to exceptions by registering callbacks with the `failed.foreach` or `onComplete` methods.

Computations that produce events in `Observable` objects can also throw exceptions. To respond to exceptions produced by the `Observable` objects, we can use an overload of the `subscribe` method that takes two callback arguments to create an observer–the callback function for the events and the callback function for the exception.

The following program creates an `Observable` object that emits numbers `1` and `2`, and then produces a `RuntimeException`. The `items` factory method creates the `Observable` object with the numbers, and the `error` factory method creates another `Observable` object with an exception. We then concatenate the two together with the `++` operator on `Observable` instances. The first callback logs the numbers to the standard output and ignores the exception. Conversely, the second callback logs the `Throwable` objects and ignores the numbers. This is shown in the following code snippet:

```
object ObservablesExceptions extends App {
  val exc = new RuntimeException
  val o = Observable.items(1, 2) ++ Observable.error(exc)
  o.subscribe(
    x => log(s"number $x"),
    t => log(s"an error occurred: $t")
  )
}
```

The program first prints numbers `1` and `2`, and then prints the exception object. Without the second callback function being passed to the `subscribe` method, the exception will be emitted by the `Observable` object `o`, but never passed to the observer. Importantly, after an exception is emitted, the `Observable` object is not allowed to emit any additional events. We can redefine the `Observable` object `o` as follows:

```
import Observable._
val o = items(1, 2) ++ error(exc) ++ items(3, 4)
```

We might expect the program to print events `3` and `4`, but they are not emitted by the `Observable` object `o`. When an `Observable` object produces an exception, we say that it is in the error state.

> TIP: When an `Observable` object produces an exception, it enters the error state and cannot emit more events.

Irrespective of whether the `Observable` object is created using a factory method, or is a custom `Observable` implementation described in the subsequent sections, an `Observable` object is not allowed to emit events after it produces an exception. In the next section, we will examine this contract in more detail.

The Observable contract

Now that we have seen how to create simple `Observable` objects and react to their events, it is time to take a closer look at the lifetime of an `Observable` object. Every `Observable` object can be in three states: uncompleted, error, or completed. As long as the `Observable[T]` object is uncompleted, it can emit events of type `T`. As we already learned, an `Observable` object can produce an exception to indicate that it failed to produce additional data. When this happens, the `Observable` object enters the error state and cannot emit any additional events. Similarly, when an `Observable` object decides that it will not produce any additional data, it might enter the completed state. After an `Observable` object is completed, it is not allowed to emit any additional events.

In Rx, an object that subscribes to events from an `Observable` object is called an `Observer` object. The `Observer[T]` trait comes with three methods: `onNext`, `onError`, and `onCompleted`, which get invoked when an `Observable` object emits an event, produces an error, or is completed, respectively. This trait is shown in the following code snippet:

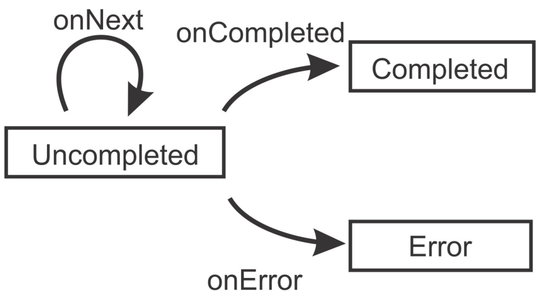
```
trait Observer[T] {
  def onNext(event: T): Unit
  def onError(error: Throwable): Unit
  def onCompleted(): Unit
}
```

In the previous examples, whenever we called the `subscribe` method, Rx created an `Observer` object and assigned it to the `Observable` instance. Alternatively, we can provide an `Observer` object directly to an overloaded version of the `subscribe` method. The following program uses the `from` factory method which converts a list of movie titles into an `Observable` object. It then creates an `Observer` object and passes it to the `subscribe` method:

```
object ObservablesLifetime extends App {
  val classics = List("Good, bad, ugly", "Titanic", "Die Hard")
  val movies = Observable.from(classics)
  movies.subscribe(new Observer[String] {
    override def onNext(m: String) = log(s"Movies Watchlist - $m")
    override def onError(e: Throwable) = log(s"Ooops - $e!")
    override def onCompleted() = log(s"No more movies.")
  })
}
```

This program first prints our favorite movies, and terminates after calling `onCompleted` and printing `"No more movies"`. The `Observable` object `movies` is created from a finite collection of strings; after these events are emitted, the `movies` event stream calls the `onCompleted` method. In general, the `Observable` objects can only call the `onCompleted` method after it is certain that there will be no more events.

Every `Observable` object can call the `onNext` method on its `Observer` objects zero or more times. An `Observable` object might then enter the completed or error state by calling the `onCompleted` or `onError` method on its `Observer` objects. This is known as the `Observable` contract, and is shown graphically in the following state diagram, where different nodes denote `Observable` states, and links denote calls to different `Observer` methods:



Note that an `Observable` object can call the `onCompleted` or `onError` method if it knows that it will not emit additional events, but it is free to call neither. Some Observable objects, such as `items`, know when they emit the last event. On the other hand, an `Observable` instance that emits mouse or keyboard events never calls the `onCompleted` method.

> An `Observable` object can call the `onNext` method on the subscribed `Observer` objects an unlimited number of times. After optionally calling the `onCompleted` or `onError` method, an `Observable` object is not allowed to call any `Observer` methods.

The `Observable` objects produced by the Rx API implement the `Observable` contract. In practice, we do not need to worry about the `Observable` contract, unless we are implementing our own custom `Observable` object. This is the topic of the next section.

Implementing custom Observable objects

To create a custom `Observable` object, we can use the `Observable.create` factory method as follows:

```
def create(f: Observer[T] => Subscription): Observable[T]
```

The preceding method takes a function `f` from an `Observer` to a `Subscription` object and returns a new `Observable` object. Whenever the `subscribe` method gets called, the function `f` is called on the corresponding `Observer` object. The function `f` returns a `Subscription` object, which can be used to unsubscribe the `Observer` object from the `Observable` instance. The `Subscription` trait defines a single method called `unsubscribe`:

```
trait Subscription {
  def unsubscribe(): Unit
}
```

We will talk about the `Subscription` objects in more detail in a subsequent section. For now, we only use the empty `Subscription` object, which does not unsubscribe the `Observer` object.

To illustrate how to use the `Observable.create` method, we implement an `Observable` object `vms`, which emits names of popular virtual machine implementations. In `Observable.create`, we take care to first call `onNext` with all the VM names, and then call `onCompleted` once. Finally, we return the empty `Subscription` object. This is shown in the following program:

```
object ObservablesCreate extends App {
  val vms = Observable.apply[String] { obs =>
    obs.onNext("JVM")
    obs.onNext("DartVM")
    obs.onNext("V8")
    obs.onCompleted()
    Subscription()
  }
  vms.subscribe(log _, e => log(s"oops - $e"), () => log("Done!"))
}
```

The `Observable` object `vms` has a synchronous `subscribe` method. All the events are emitted to an `obs` observer before returning the control to the thread that called the `subscribe` method. In general, we can use the `Observable.create` method in order to create an `Observable` instance that emits events asynchronously. We will study how to convert a `Future` object into an `Observable` object next.

Creating Observables from futures

Futures are objects that represent the result of an asynchronous computation. One can consider an `Observable` object as a generalization of a `Future` object. Instead of emitting a single success or failure event, an `Observable` object emits a sequence of events, before failing or completing successfully.

Scala APIs that deal with asynchronous computations generally return the `Future` objects, and not `Observable` instances. In some cases, it is useful to be able to convert a `Future` object into an `Observable` object. Here, after a `Future` object is completed successfully, the corresponding `Observable` object must emit an event with the future value, and then call the `onCompleted` method. If the `Future` object fails, the corresponding `Observable` object should call the `onError` method. Before we begin, we need to import the contents of the `scala.concurrent` package and the global `ExecutionContext` object, as shown in the following code snippet:

```
import scala.concurrent._
import ExecutionContext.Implicits.global
```

We then use the `Observable.create` method to create an `Observable` object `o`. Instead of calling the `onNext`, `onError`, and `onCompleted` methods directly on the `Observer` object, we will install callbacks on the `Future` object `f`, as shown in the following program:

```
object ObservablesCreateFuture extends App {
  val f = Future { "Back to the Future(s)" }
  val o = Observable.create[String] { obs =>
    f foreach { case s => obs.onNext(s); obs.onCompleted() }
    f.failed foreach { case t => obs.onError(t) }
    Subscription()
  }
  o.subscribe(log _)
}
```

This time, the `subscribe` method is asynchronous. It returns immediately after installing the callback on the `Future` object. In fact, this pattern is so common that Rx comes with the `Observable.from` factory method that converts a `Future` object into an `Observable` object directly, as shown by the following code snippet:

```
val o = Observable.from(Future { "Back to the Future(s)" })
```

Still, learning how to convert a `Future` object into an `Observable` object is handy. The `Observable.create` method is the preferred way to convert callback-based APIs to `Observable` objects, as we will see in subsequent sections.

Use the `Observable.create` factory method to create the `Observable` objects from callback-based APIs.

In the examples so far, we have always returned an empty `Subscription` object. Calling the `unsubscribe` method on such a `Subscription` object has no effect. Sometimes, the `Subscription` objects need to release resources associated with the corresponding `Observable` instance. We will study how to implement and work with such `Subscription` objects next.

Subscriptions

Recall the example monitoring the filesystem for changes in `Chapter 4`, *Asynchronous Programming with Futures and Promises*, where we used the file monitoring package from the Apache Commons IO library to complete a `Future` object when a new file is created. A `Future` object can be completed only once, so the future was completed with the name of the first file that was created. It is more natural to use `Observable` objects for this use case, as files in a filesystem can be created and deleted many times. In an application such as a file browser or an FTP server, we would like to receive all such events.

Later in the program, we might want to unsubscribe from the events in the `Observable` object. We will now see how to use the `Subscription` object to achieve this. We first import the contents of the **Apache Commons IO file monitoring** package, as follows:

```
import org.apache.commons.io.monitor._
```

We define the `modified` method, which returns an `Observable` object with filenames of the modified files in the specified directory. The `Observable.create` method bridges the gap between the Commons IO callback-based API and Rx. When the `subscribe` method is called, we create a `FileAlterationMonitor` object, which uses a separate thread to scan the filesystem and emit filesystem events every 1000 milliseconds, a `FileAlterationObserver` object, which specifies a directory to monitor; and a `FileAlterationListener` object, which reacts to file events by calling the `onNext` method on the Rx `Observer` object. We then call the `start` method on the `fileMonitor` object.

Finally, we return a custom `Subscription` object, which calls `stop` on the `fileMonitor` object. The `modified` method is shown in the following code snippet:

```
def modified(directory: String): Observable[String] = {
  Observable.create { observer =>
    val fileMonitor = new FileAlterationMonitor(1000)
    val fileObs = new FileAlterationObserver(directory)
    val fileLis = new FileAlterationListenerAdaptor {
      override def onFileChange(file: java.io.File) {
        observer.onNext(file.getName)
      }
    }
    fileObs.addListener(fileLis)
    fileMonitor.addObserver(fileObs)
    fileMonitor.start()
    Subscription { fileMonitor.stop() }
  }
}
```

We used the `apply` factory method on the `Subscription` companion object in the preceding code snippet. When the `unsubscribe` method is called on the resulting `Subscription` object, the specified block of code is run. Importantly, calling the `unsubscribe` method, the second time will not run the specified block of code again. We say that the `unsubscribe` method is **idempotent**; calling it multiple times has the same effect as calling it only once. In our example, the `unsubscribe` method calls the `stop` method of the `fileMonitor` object at most once. When sub-classing the `Subscription` trait, we need to ensure that the `unsubscribe` method is idempotent, and the `Subscription.apply` method is a convenience method that ensures idempotence automatically.

TIP

Implementations of the `unsubscribe` method in the `Subscription` trait need to be idempotent. Use the `Subscription.apply` method to create the `Subscription` objects that are idempotent by default.

We use the `modified` method to track file changes in our project. After we call the `subscribe` method on the `Observable` object returned by the `modified` method, the main thread suspends for 10 seconds. If we save files in our editor during this time, the program will log file modification events to the standard output. This is shown in the following program:

```
object ObservablesSubscriptions extends App {
  log(s"starting to monitor files")
  val sub = modified(".").subscribe(n => log(s"$n modified!"))
  log(s"please modify and save a file")
  Thread.sleep(10000)
  sub.unsubscribe()
  log(s"monitoring done")
}
```

Note that, in this example, the `FileAlterationMonitor` object is only created if the program invokes the `subscribe` method. The `Observable` instance returned by the `modified` method does not emit events unless there exists an `Observer` object subscribed to it. In Rx, the `Observable` objects that emit events only when subscriptions exist are called **cold observables**. On the other hand, some `Observable` objects emit events even when there are no associated subscriptions. This is usually the case with `Observable` instances that handle user input, such as keyboard or mouse events. `Observable` objects that emit events regardless of their subscriptions are called **hot observables**. We now reimplement an `Observable` object that tracks file modifications as a hot observable. We first instantiate and start the `FileAlterationMonitor` object, as follows:

```
val fileMonitor = new FileAlterationMonitor(1000)
fileMonitor.start()
```

The `Observable` object uses the `fileMonitor` object to specify the directory in order to monitor. The downside is that our `Observable` object now consumes computational resources even when there are no subscriptions. The advantage of using a hot observable is that multiple subscriptions do not need to instantiate multiple `FileAlterationMonitor` objects, which are relatively heavyweight. We implement the hot `Observable` object in the `hotModified` method, as shown in the following code:

```
def hotModified(directory: String): Observable[String] = {
  val fileObs = new FileAlterationObserver(directory)
  fileMonitor.addObserver(fileObs)
  Observable.create { observer =>
    val fileLis = new FileAlterationListenerAdaptor {
      override def onFileChange(file: java.io.File) {
        observer.onNext(file.getName)
      }
    }
    fileObs.addListener(fileLis)
    Subscription { fileObs.removeListener(fileLis) }
  }
}
```

The `hotModified` method creates an `Observable` object with file changes for a given directory by registering the specified directory with the `fileMonitor` object, and only then calls the `Observable.create` method. When the `subscribe` method is called on the resulting `Observable` object, we instantiate and add a new `FileAlterationListener` object. In the `Subscription` object, we remove the `FileAlterationListener` object in order to avoid receiving additional file modification events, but we do not call the `stop` method on the `fileMonitor` object until the program terminates.

Composing Observable objects

Having seen different ways of creating various types of the `Observable` objects, subscribing to their events, and using the `Subscription` objects, we turn our attention to composing the `Observable` objects into larger programs. From what we have seen so far, the advantages of using the `Observable` objects over a callback-based API are hardly worth the trouble.

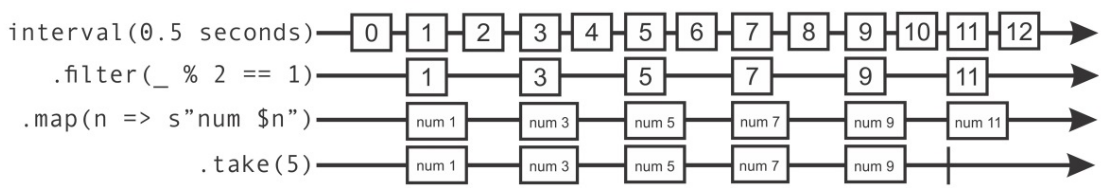
The true power of Rx becomes apparent when we start composing the `Observable` objects using various combinators. We can think of an `Observable` object in a similar way as we think of Scala sequence collections. In a Scala sequence, represented by the `Seq[T]` trait, elements of type `T` are ordered in the memory according to their indices. In an `Observable[T]` trait, events of type `T` are ordered in time.

Let's use the `Observable.interval` factory method in order to create an `Observable` object, which asynchronously emits a number every 0.5 seconds, and then output the first five odd numbers. To do this, we first call `filter` on the `Observable` object in order to obtain an intermediate `Observable` object that emits only odd numbers. Note that calling the `filter` on an `Observable` object is similar to calling `filter` method on a Scala collection. Similarly, we obtain another `Observable` object by calling the `map` method in order to transform each odd number into a string. We then call `take` to create an `Observable` object `odds`, which contains only the first five events. Finally, we subscribe to `odds` so that we can print the events it emits. This is shown in the following program:

```
object CompositionMapAndFilter extends App {
  val odds = Observable.interval(0.5.seconds)
    .filter(_ % 2 == 1).map(n => s"num $n").take(5)
  odds.subscribe(
    log _, e => log(s"unexpected $e"), () => log("no more odds"))
  Thread.sleep(4000)
}
```

To concisely explain the semantics of different Rx combinators, we often rely on marble diagrams. These diagrams graphically represent events in an `Observable` object and transformations between different `Observable` objects. The marble diagram represents every `Observable` object with a timeline containing its events. The first three intermediate `Observable` objects never call the `onCompleted` method on its observers.

The `Observable` object `odds` contains at most five events, so it calls `onCompleted` after emitting them. We denote a call to the `onCompleted` method with a vertical bar in the marble diagram, as shown in the following diagram:



Note that the preceding diagram is a high-level illustration of the relationships between different `Observable` objects, but some of these events can be omitted during execution. The particular Rx implementation can detect that the events `11` and `12` cannot be observed by the `subscribe` invocation, so these events are not emitted to save computational resources.

As an expert on sequential programming in Scala, you probably noticed that we can rewrite the previous program more concisely using the for-comprehensions. For example, we can output the first five even natural numbers with the following for-comprehension:

```
val evens = for (n <- Observable.from(0 until 9); if n % 2 == 0)
yield s"even number $n"
evens.subscribe(log _)
```

Before moving on to more complex for-comprehensions, we will study a special kind of `Observable` object whose events are other `Observable` objects.

Nested Observables

A nested observable, also called a higher-order event stream, is an `Observable` object that emits events that are themselves `Observable` objects. A higher-order function such as the `foreach` statement is called a higher-order function because it has a nested function inside its `(T => Unit) => Unit` type. Similarly, higher-order event streams earned this fancy name because they have a type `Observable[T]` as part of their type `Observable[Observable[T]]`. In this section, we will study when the `nestedObservable` objects are useful and how to manipulate them.

Let's assume that we are writing a book and we want to add a famous quote at the beginning of each chapter. Choosing the right quote for a chapter is a hard job and we want to automate it. We write a short program that uses `Observable` objects to fetch random quotes from the *I Heart Quotes* website every 0.5 seconds and prints them to the screen. Once we see a nice quote, we have to quickly copy it to our book chapter.

We will start by defining a `fetchQuote` method that returns a `Future` object with the text of the quote. Luckily, the HTTP API of the *I Heart Quotes* website returns plain text, so we do not need to parse any JSON or XML. We use the `scala.io.Source` object to fetch the contents of the proper URL, as follows:

```
import scala.io.Source
def fetchQuote(): Future[String] = Future {
  blocking {
    val url = "http://quotes.stormconsultancy.co.uk/random.json" +
      "show_permalink=false&show_source=false"
    Source.fromURL(url).getLines.mkString
  }
}
```

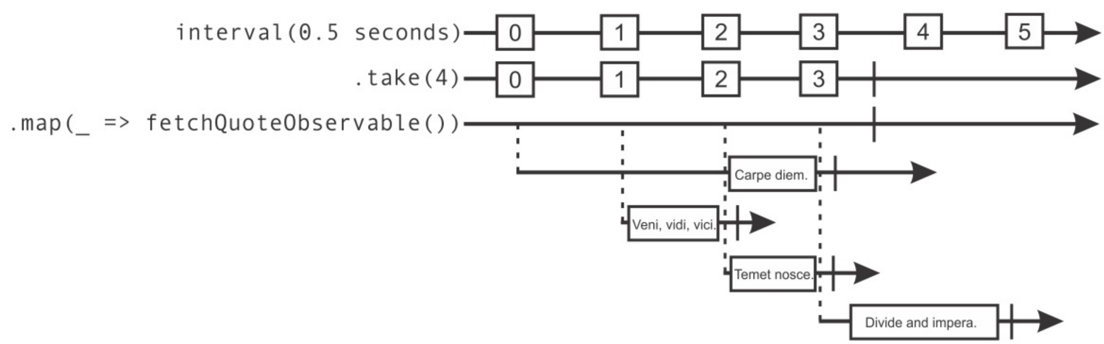
Recall that we can convert a `Future` object to an `Observable` object using the `from` factory method:

```
def fetchQuoteObservable(): Observable[String] = {
  Observable.from(fetchQuote())
}
```

We now use the `Observable.interval` factory method in order to create an `Observable` object that emits a number every 0.5 seconds. For the purposes of our example, we take only the first four numbers. Then, we map each of these numbers into an `Observable` object that emits a quote, prefixed with the ordinal number of the quote. To do this, we call the `fetchQuoteObservable` method and map the quotes using a nested `map` call, as shown in the following code snippet:

```
def quotes: Observable[Observable[String]] =
  Observable.interval(0.5 seconds).take(4).map {
    n => fetchQuoteObservable().map(txt => s"$n) $txt")
  }
```

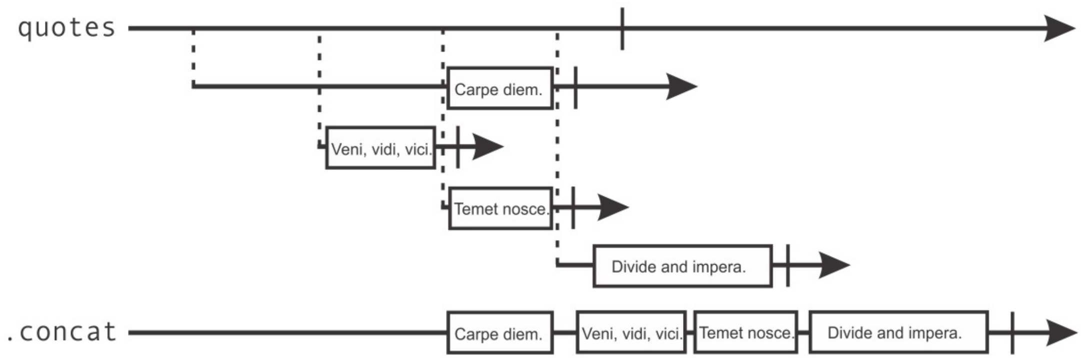
Note that the inner `map` call transforms an `Observable[String]` instance, which contains the quote text, to another `Observable[String]` instance, which contains the quote prefixed with a number. The outer `map` call transforms the `Observable[Long]` object, which contains the first four numbers, to an `Observable[Observable[String]]` instance, which contains `Observable` objects emitting separate quotes. The `Observable` objects created by the `quotes` method are shown in the following marble diagram. Events in the nested `Observable` objects presented last are themselves `Observable` objects that contain a single event: the text of the quote returned in the `Future` object. Note that we omit the nested `map` call from the diagram to make it more readable:



Drawing a marble diagram makes the contents of this `Observable` object more understandable, but how do we subscribe to events in an `Observable[Observable[String]]` object? Calling the `subscribe` method on quotes requires observers to handle the `Observable[String]` objects, and not the `String` events directly.

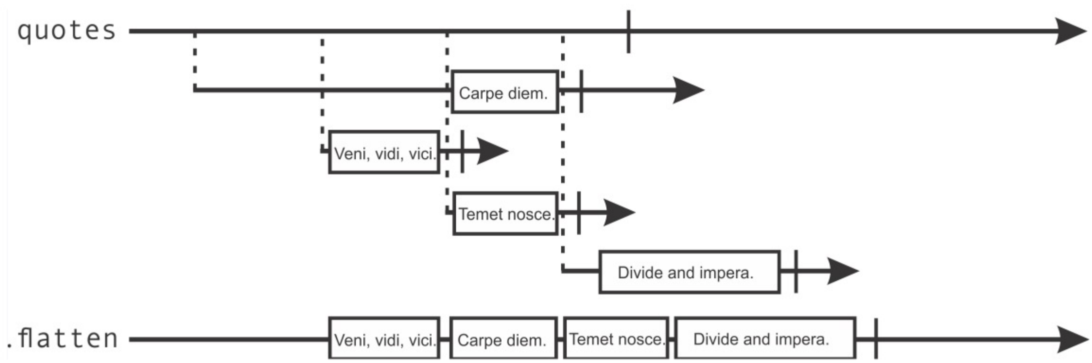
Once again, an analogy with Scala sequence collections is useful in order to understand how to solve this issue. Whenever we have a nested sequence, say `Seq[Seq[T]]`, we can flatten it to a `Seq[T]` collection by calling the `flatten` method. When we do this, elements of the nested sequences are simply concatenated together. The Rx API provides similar methods that flatten the `Observable` objects, but they must deal with the additional complexity associated with the timing of events. There are different ways of flattening the `Observable` objects depending on the time when their events arrive.

The first method, called `concat`, concatenates the `nestedObservable` objects by ordering all the events in one nested `Observable` object before the events in a subsequent `Observable` object. An `Observable` object that appears earlier must complete before the events from a subsequent `Observable` object can be emitted. The marble diagram for the `concat` operation is shown in the following figure. Although the quote **Veni, vidi, vici.**, arrives before the quote **Carpe diem**., the quote **Veni, vidi, vici.** is emitted only after the `Observable` object associated with the quote **Carpe diem.** completes. The resulting `Observable` object completes only after the `Observable` object `quotes` and all the nested `Observable` objects complete:



The second method is called `flatten`, analogously to the similar method in the Scala collections API. This method emits events from the nested `Observable` objects in the order in which they arrive in time, regardless of when the respective nested `Observable` object started. An `Observable` object that appears earlier is not required to complete before events from a subsequent `Observable` object are emitted.

This is illustrated in the following marble diagram. A quote is emitted to the resulting `Observable` object as soon as it appears on any of the nested `Observable` objects. Once `quotes` and all the nested `Observable` objects complete, the resulting `Observable` object completes as well.



To test the difference between the `concat` and `flatten` methods, we subscribe to events in `quotes` using each of these two methods. If our network is unreliable or has particularly nondeterministic latency, the order in which the second `subscribe` call prints the `quotes` object can be mangled. We can reduce the interval between queries from 0.5 to 0.01 seconds to witness this effect. The ordinal numbers preceding each quote become unordered when using the `flatten` method. This is illustrated in the following program:

```
object CompositionConcatAndFlatten extends App {
  log(s"Using concat")
  quotes.concat.subscribe(log _)
  Thread.sleep(6000)
  log(s"Now using flatten")
  quotes.flatten.subscribe(log _)
  Thread.sleep(6000)
}
```

How do we choose between the `concat` and `flatten` methods? The `concat` method has the advantage that it maintains the relative order between events coming from different `Observable` objects. If we had been fetching and printing quotes in a lexicographic order, then the `concat` method would be the correct way to flatten the nested `Observable` objects.

Use `concat` to flatten nested `Observable` objects whenever the order of events between different nested `Observable` objects needs to be maintained.

The `concat` method does not subscribe to subsequent `Observable` objects before the current `Observable` object completes. If one of the nested `Observable` objects takes a long time to complete or does not complete at all, the events from the remaining `Observable` objects are postponed or never emitted. The `flatten` method subscribes to a nested `Observable` object as soon as the nested `Observable` object is emitted, and emits events as soon as they arrive.

If at least one of the nested `Observable` objects has an unbounded number of events or never completes, use the `flatten` method instead of the `concat` method.

We can also traverse events from multiple `Observable` objects in a `for` comprehension. The `Observable` objects come with the `flatMap` method, and this allows you to use them in `for` comprehensions. Calling the `flatMap` method on an `Observable` object is equivalent to mapping each of its events into a nested `Observable` object, and then calling the `flatten` method. Thus, we can rewrite the `quotes.flatten` method as follows:

```
Observable.interval(0.5 seconds).take(5).flatMap({
  n => fetchQuoteObservable().map(txt => s"$n) $txt")
}).subscribe(log _)
```

Having already mastered `for` comprehensions on Scala collections and `for` comprehensions on futures, this pattern of `flatMap` and `map` calls immediately rings a bell, and we recognize the previous expression as the following `for` comprehension:

```
val qs = for {
  n   <- Observable.interval(0.5 seconds).take(5)
  txt <- fetchQuoteObservable()
} yield s"$n) $txt"
qs.subscribe(log _)
```

This is much more concise and understandable, and almost feels like we're back with collections land. Still, we need to be careful, because for-comprehensions on `Observable` objects do not maintain the relative order of the events in the way that the for-comprehensions on collections do. In the preceding example, as soon as we can pair a `n` number with some quote `txt`, the `s"$n) $txt"` event is emitted, irrespective of the events associated with the preceding `n` number.

> Calling the `flatMap` method or using `Observable` objects in `for` comprehensions emits events in the order in which they arrive, and it does not maintain ordering between events from different `Observable` objects. Invoking the `flatMap` method is semantically equivalent to calling `map` followed by the `flatten` call.

An attentive reader will notice that we did not consider the case where one of the nested `Observable` objects terminates by calling the `onError` method. When this happens, both `concat` and `flatten` call the `onError` method with the same exception. Similarly, `map` and `filter` fail the resulting `Observable` object if the input `Observable` object produces an exception, so it is unclear how to compose failed `Observable` objects. This is the focus of the next section.

Failure handling in Observables

If you ran the previous examples yourself, you might have noticed that some of the quotes are long and tedious to read. We don't want to put a long quote at the beginning of the chapter. If we did that, our readers might lose interest. The best quotes are short and straight to the point.

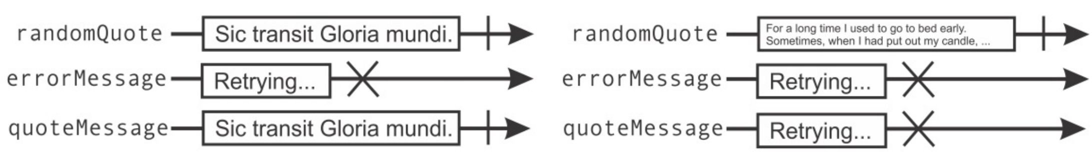
Our next goal will be to replace quotes longer than 100 characters with a string `Retrying...` and print the first quote shorter than 100 characters. This time, we define an `Observable` object called `randomQuote`, which emits a random quote every time we subscribe to it. We use the `Observable.create` method in order to obtain a random quote as before and emit the quote to the observer. We then return an empty `Subscription` object. This is shown in the following code snippet:

```
def randomQuote = Observable.create[String] { obs =>
  val url = "http://www.iheartquotes.com/api/v1/random?" +
    "show_permalink=false&show_source=false"
  obs.onNext(Source.fromURL(url).getLines.mkString)
  obs.onCompleted()
  Subscription()
}
```

There is a subtle difference between the `Observable` object returned by the `randomQuote` method and the one returned by the `fetchQuoteObservable` method, defined earlier. The `fetchQuoteObservable` method creates a `Future` object in order to obtain a quote and emits the quote in that `Future` object to every observer. By contrast, the `randomQuote` method fetches a new quote every time the `subscribe` method is called. In the previously introduced terminology, the `randomQuote` method creates cold `Observable` objects, which emit events only when we subscribe to them, whereas the `fetchQuoteObservable` method creates hot `Observable` objects, which emit the same quote to all their observers.

To re-subscribe to a failed `Observable` object, we can use the `retry` combinator. The `retry` combinator takes an input `Observable`, and returns another `Observable` object that emits events from the input `Observable` object until it either completes or fails. If the input `Observable` object fails, the `retry` combinator subscribes to the input `Observable` object again.

We now use the `retry` combinator with the `randomQuote` method to fetch quotes until we obtain a quote shorter than 100 characters. We first transform the long quotes from the `randomQuote` method into failed observables, which enables `retry` to subscribe again to obtain another quote. To do this, we define a new `Observable` object called `errorMessage`, which emits a string `"Retrying..."` and then fails. We then traverse the `text` quote from `randomQuote` in a `for` comprehension. If the `text` quote is shorter than 100 characters, we traverse an `Observable` object that emits text. Otherwise, we traverse the `errorMessage` object to output `"Retrying..."` instead of `text`. This `for` comprehension defines an `Observable` object `quoteMessage`, which either emits a short quote, or emits `"Retrying..."` and fails. The marble diagram of the resulting `Observable` object, called `quoteMessage`, is shown for these two cases, in which the exception in the `Observable` object is shown with a cross symbol:



Finally, we call the `retry` method on the `quoteMessage` object and subscribe to it. We specify that we want to retry up to five times, as omitting the argument would retry forever. We implement the `Observable` object `quoteMessage` in the following program:

```
object CompositionRetry extends App {
  import Observable._
  def errorMessage = items("Retrying...") ++ error(new Exception)
  def quoteMessage = for {
    text    <- randomQuote
    message <- if (text.size < 100) items(text) else errorMessage
  } yield message
  quoteMessage.retry(5).subscribe(log _)
  Thread.sleep(2500)
}
```

Run this program several times. You will notice that a short quote is either printed right away, or after a few retries, depending on some random distribution of the quotes. You may be wondering how many quotes are on average longer than 100 characters. It turns out that it is easy to do this statistic in Rx. We introduce two new combinators. The first one is called `repeat`, and it is very similar to `retry`. Instead of re-subscribing to an `Observable` object when it fails, it re-subscribes when an `Observable` object completes. The second combinator is called `scan` and it is similar to the `scanLeft` operator on collections. Given an input `Observable` object and a starting value for the accumulation, it emits the value of the accumulation by applying the specified binary operator to the accumulation and the event, updating the accumulation as the events arrive. The usage of the `repeat` and `scan` combinators is illustrated in the following program:

```
object CompositionScan extends App {
  CompositionRetry.quoteMessage.retry.repeat.take(100).scan(0) {
    (n, q) => if (q == "Retrying...") n + 1 else n
  } subscribe(n => log(s"$n / 100"))
}
```

In the preceding example, we use the `Observable` object `quoteMessage` defined earlier in order to obtain a short quote or a message `"Retrying..."` followed by an exception. We retry quotes that have failed because of being too long, and repeat whenever a quote is short enough. We take 100 quotes in total, and use the `scan` operator to count the short quotes. When we ran this program, it turned out that 57 out of 100 quotes are too long for our book.

The `retry` method is used in order to repeat events from failed `Observable` objects. Similarly, the `repeat` method is used in order to repeat events from completed `Observable` objects.

In the examples shown so far, we use the same `Observable` object to re-subscribe and emit additional events if that `Observable` object fails. In some cases, we want to emit specific events when we encounter an exception, or fall back to a different `Observable` object. Recall that this is what we did with `Future` objects previously. The Rx methods that replace an exception with an event, or multiple events from another `Observable` object, are called `onErrorReturn` and `onErrorResumeNext`, respectively. In the following program, we first replace the exception from `status` with a string `"exception occurred."`. We then replace the exception with strings from another `Observable` object:

```
object CompositionErrors extends App {
  val status = items("ok", "still ok") ++ error(new Exception)
  val fixedStatus =
    status.onErrorReturn(e => "exception occurred.")
  fixedStatus.subscribe(log _)
  val continuedStatus =
    status.onErrorResumeNext(e => items("better", "much better"))
  continuedStatus.subscribe(log _)
}
```

Having seen various ways to compose `Observable` objects, we turn to the concurrency features of Rx. So far, we did not pay close attention to the thread on which an `Observable` object emits events. In the next section, we will study how to transfer events between `Observable` objects on different threads, and learn when this can be useful.

Rx schedulers

At the beginning of this chapter, we observed that different `Observable` objects emit events on different threads. A synchronous `Observable` object emits on the caller thread when the `subscribe` method gets invoked. The `Observable.timer` object emits events asynchronously on threads internally used by Rx. Similarly, events in `Observable` objects created from `Future` objects are emitted on `ExecutionContext` threads. What if we want to use an existing `Observable` object to create another `Observable` object bound to a specific thread?

To encapsulate the choice of the thread on which an `Observable` object should emit events, Rx defines a special class called `Scheduler`. A `Scheduler` class is similar to the `Executor` and `ExecutionContext` interfaces we saw in `Chapter 3`, *Traditional Building Blocks of Concurrency*. The `Observable` objects come with a combinator called `observeOn`. This combinator returns a new `Observable` object that emits events using the specified `Scheduler` class. In the following program, we instantiate a `Scheduler` object called `ComputationScheduler`, which emits events using an internal thread pool. We then emit events with and without calling the `observeOn` combinator:

```
object SchedulersComputation extends App {
  val scheduler = schedulers.ComputationScheduler()
  val numbers = Observable.from(0 until 20)
  numbers.subscribe(n => log(s"num $n"))
  numbers.observeOn(scheduler).subscribe(n => log(s"num $n"))
  Thread.sleep(2000)
}
```

From the output, we can see that the second `subscribe` call uses a thread pool:

```
run-main-42: num 0
...
run-main-42: num 19
RxComputationThreadPool-1: num 0
...
RxComputationThreadPool-1: num 19
```

The `ComputationScheduler` object maintains a pool of threads intended for computational tasks. If processing the events blocks or waits for I/O operations, we must use the `IOScheduler` object, which automatically spawns new threads when necessary. Exceptionally, if processing each event is a very coarse-grained task, we can use the `NewThreadScheduler` object, which spawns a new thread for each event.

Using custom schedulers for UI applications

Built-in Rx schedulers are useful for most tasks, but in some cases we need more control. Most UI toolkits only allow you to read and modify UI elements from a special thread. This thread is called the **event-dispatching** thread. This approach simplifies the design and the implementation of a UI toolkit, and protects clients from subtle concurrency errors. Since the UI usually does not usually represent a computational bottleneck, this approach has been widely adopted; the Swing toolkit uses an `EventDispatchThread` object in order to propagate events.

The `Observable` objects are particularly useful when applied to UI applications; a user interface is all about events. In subsequent examples, we will use the Scala Swing library to illustrate the usefulness of Rx in UI code. We start by adding the following dependency to our project:

```
libraryDependencies +=
  "org.scala-lang.modules" %% "scala-swing" % "1.0.1"
```

We will start by creating a simple Swing application with a single button. Clicking on this button will print a message to the standard output. This application illustrates how to convert Swing events into an `Observable` object. We will start by importing the relevant Scala Swing packages as follows:

```
import scala.swing._
import scala.swing.event._
```

To create a Swing application, we need to extend the `SimpleSwingApplication` class. This class has a single abstract method, `top`, which needs to return a `Frame` object. The Swing's abstract `Frame` class represents the application window. We return a new `MainFrame` object, which is a subclass of the `Frame` object. In the `MainFrame` constructor, we set the window title bar text to `Swing Observables`, and instantiate a new `Button` object with the `Click` text. We then set the contents of the `MainFrame` constructor to that button.

So much for the UI elements and their layout; we now want to add some logic to this simple application. Traditionally, we would make a Swing application interactive by installing callbacks to various UI elements. Using Rx, we instead convert callbacks into event streams; we define an `Observable` object called `buttonClicks` that emits an event every time the button element is clicked on. We use the `Observable.create` method in order to register a `ButtonClicked` callback that calls the `onNext` method on the observer. To log clicks to the standard output, we subscribe to `buttonClicks`. The complete Swing application is shown in the following code snippet:

```
object SchedulersSwing extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "Swing Observables"
    val button = new Button {
      text = "Click"
    }
    contents = button
    val buttonClicks = Observable.create[Button] { obs =>
      button.reactions += {
        case ButtonClicked(_) => obs.onNext(button)
      }
      Subscription()
    }
    buttonClicks.subscribe(_ => log("button clicked"))
  }
}
```

Running this application opens the window, shown in the following screenshot. Clicking on the **Click** button prints a string to the standard output. We can see that the events are emitted on the thread called `AWT-EventQueue-0`, which is the event-dispatching thread in Swing:

One downside of single-threaded UI toolkits is that long-running computations on the event-dispatching thread block the UI and harm the user experience. If we issue a blocking HTTP request each time the user clicks on a button, we will witness a noticeable lag after each click. Luckily, this is easy to address by executing long-running computations asynchronously.

Usually, we are not content with just starting an asynchronous computation. Once the asynchronous computation produces a result, we would like to display it in the application. Recall that we are not allowed to do this directly from the computation thread; we need to return the control back to event-dispatching thread. Swing defines the `invokeLater` method, which schedules tasks on Swing's event-dispatching thread. On the other hand, Rx has a `Schedulers.from` built-in method that converts an `Executor` object into a `Scheduler` object. To bridge the gap between Swing's `invokeLater` method and Rx schedulers, we implement a custom `Executor` object that wraps a call to `invokeLater`, and we pass this `Executor` object to `Schedulers.from`. The custom `swingScheduler` object is implemented as follows:

```
import java.util.concurrent.Executor
import rx.schedulers.Schedulers.{from => fromExecutor}
import javax.swing.SwingUtilities.invokeLater
val swingScheduler = new Scheduler {
  val asJavaScheduler = fromExecutor(new Executor {
    def execute(r: Runnable) = invokeLater(r)
  })
}
```

We can use the newly-defined `swingScheduler` object in order to send events back to Swing. To illustrate this, let's implement a small web browser application. Our browser consists of a `urlfield` address bar and the **Feeling lucky** button. Typing into the address bar displays suggestions for the URL, and clicking on the button displays the raw HTML of the webpage. The browser is not a trivial application, so we separate the implementation of the UI layout from the UI logic. We start by defining the `BrowserFrame` class, which describes the layout of the UI elements:

```
abstract class BrowserFrame extends MainFrame {
  title = "MiniBrowser"
  val specUrl = "http://www.w3.org/Addressing/URL/url-spec.txt"
  val urlfield = new TextField(specUrl)
  val pagefield = new TextArea
  val button = new Button {
    text = "Feeling Lucky"
  }
  contents = new BorderPanel {
    import BorderPanel.Position._
    layout(new BorderPanel {
      layout(new Label("URL:")) = West
      layout(urlfield) = Center
      layout(button) = East
    }) = North
    layout(pagefield) = Center
  }
  size = new Dimension(1024, 768)
}
```

Scala Swing was implemented long before the introduction of Rx, so it does not come with event streams. We use Scala's extension method pattern in order to enrich the existing UI element classes with `Observable` objects, and add implicit classes, `ButtonOps` and `TextFieldOps`, with methods, `clicks` and `texts`, respectively. The `clicks` method returns an `Observable` object that emits an event each time the corresponding button is clicked on. Similarly, the `texts` method emits an event each time the content of a text field changes:

```
implicit class ButtonOps(val self: Button) {
  def clicks = Observable.create[Unit] { obs =>
    self.reactions += {
      case ButtonClicked(_) => obs.onNext(())
    }
    Subscription()
  }
}
implicit class TextFieldOps(val self: TextField) {
  def texts = Observable.create[String] { obs =>
    self.reactions += {
      case ValueChanged(_) => obs.onNext(self.text)
    }
    Subscription()
  }
}
```

We now have the necessary utilities to concisely define the logic of our web browser. We implement the browser logic in a trait called `BrowserLogic`, annotated with a self-type `BrowserFrame` object. The type `self` allows you to mix the `BrowserLogic` trait only into classes that extend the `BrowserFrame` object. This makes sense; the browser logic needs to know about UI events to react to them.

There are two main functionalities supported by the web browser. First, the browser needs to suggest possible URLs while the user types into the address bar. To facilitate this, we define a helper method, `suggestRequest`, which takes a term from the address bar and returns an `Observable` object with the possible completions. This `Observable` object uses Google's query suggestion service to get a list of possible URLs. To cope with network errors, the `Observable` object will time-out after 0.5 seconds if there is no reply from the server, and emit an error message.

Second, our browser needs to display the contents of the specified URL, when we click on the **Feeling lucky** button. To achieve this, we define another helper method named `pageRequest`, which returns an `Observable` object with the raw HTML of the web page. This `Observable` object times-out after four seconds if the page is not loaded by that time.

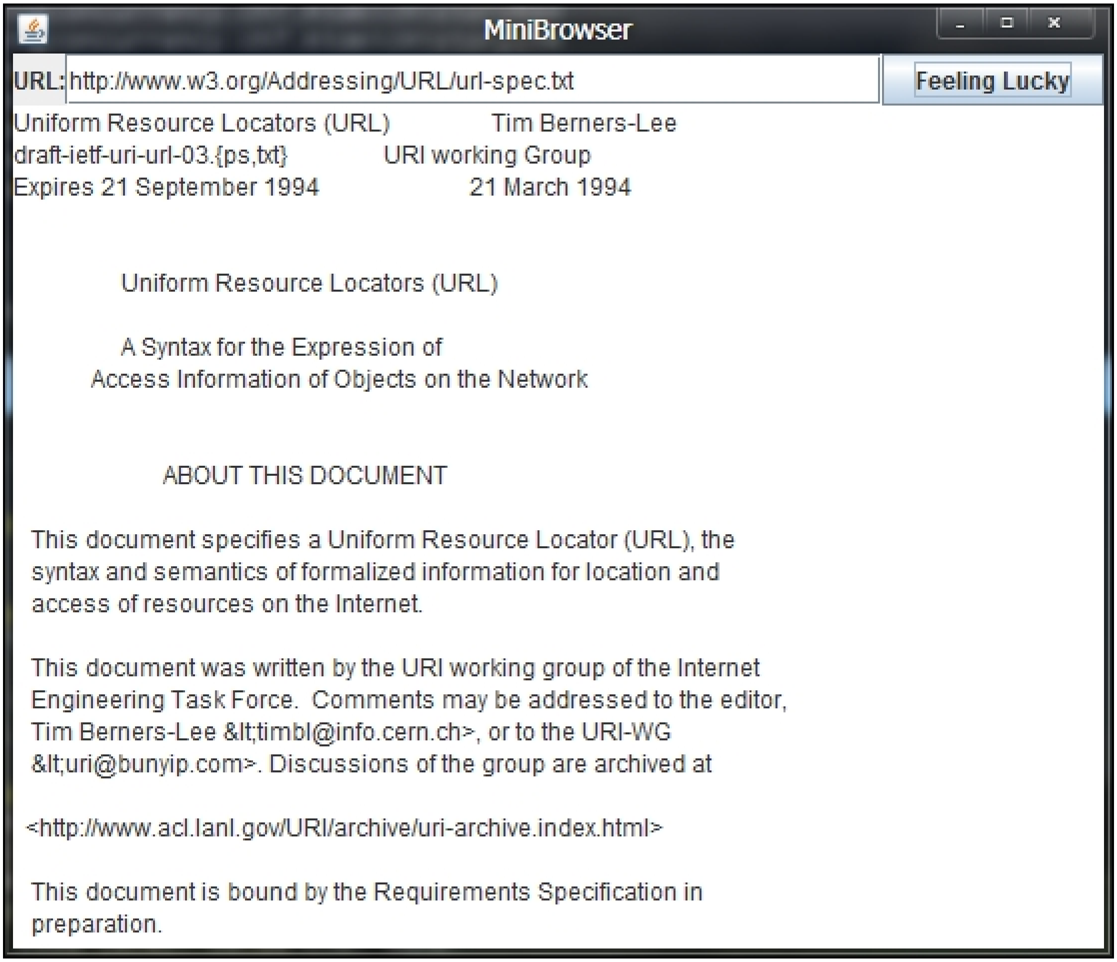
Using these helper methods and the UI element `Observable` objects, we can encode the browser logic more easily. Each `urlField` text modification event maps into a nested `Observable` object with the suggestion. The call to `concat` then flattens the nested `Observable` object. The suggestion events transfer back to the Swing event-dispatching thread using the `observeOn` combinator. We subscribe to the events on the Swing event-dispatching thread in order to modify the contents of the `pagefield` text area. We subscribe to `button.clicks` in a similar way:

```
trait BrowserLogic {
  self: BrowserFrame =>
  def suggestRequest(term: String): Observable[String] = {
    val url = "http://suggestqueries.google.com/" +
      s"complete/search?client=firefox&q=$term"
    val request = Future { Source.fromURL(url).mkString }
    Observable.from(request)
              .timeout(0.5.seconds)
              .onErrorReturn(e => "(no suggestion)")
  }
  def pageRequest(url: String): Observable[String] = {
    val request = Future { Source.fromURL(url).mkString }
    Observable.from(request)
              .timeout(4.seconds)
              .onErrorReturn(e => s"Could not load page: $e")
  }
  urlfield.texts.map(suggestRequest).concat
                .observeOn(swingScheduler)
                .subscribe(response => pagefield.text = response)
  button.clicks.map(_ => pageRequest(urlfield.text)).concat
               .observeOn(swingScheduler)
               .subscribe(response => pagefield.text = response)
}
```

After defining both the UI layout and the UI logic, we only need to instantiate the browser frame in a Swing application:

```
object SchedulersBrowser extends SimpleSwingApplication {
  def top = new BrowserFrame with BrowserLogic
}
```

Running the application opens the browser frame, and we can start surfing in our very own Rx-based web browser. The guys at Mozilla and Google will surely be impressed when they see the following screenshot:



Although our web browser is very simple, we managed to separate its functionality into the UI layout and browser logic layers. The UI layout layer defines `Observable` objects such as `urlfield.texts` and `button.clicks` as part of its interface. The browser logic layer relies on the functionality from the UI layout layer; for example, we could not describe the updates to the `pagefield` UI element without referencing the `Observable` object `button.clicks`.

We say that the browser logic depends on the UI layout, but not vice versa. For a UI application, this can be acceptable, but other applications require a more loosely coupled design, in which different layers do not refer to each other directly.

Subjects and top-down reactive programming

Composing `Observable` objects is similar to composing functions, collections, or futures. Complex `Observable` objects are formed from simpler parts using functional composition. This is a very Scala-idiomatic pattern, and it results in concise and understandable programs.

A not-so-obvious downside of functional composition is that it favors the **bottom-up programming style**. An `Observable` object cannot be created without a reference to another `Observable` object that it depends on. For instance, we cannot create an `Observable` object using the `map` combinator without having an input `Observable` object to call the `map` method on. In a bottom-up programming style, we build complex programs by implementing the simplest parts first, and then gradually working our way up. By contrast, in a **top-down programming style**, we first define the complex parts of the system, and then gradually divide them into successively smaller pieces. The top-down programming style allows first declaring an `Observable` object, and defining its dependencies later.

To allow building systems in a top-down programming style, Rx defines an abstraction called a subject, represented by the `Subject` trait. A `Subject` trait is simultaneously an `Observable` object and an `Observer` object. As an `Observable` object, a `Subject` trait can emit events to its subscribers. As an `Observer` object, a `Subject` trait can subscribe to different input `Observable` objects and forward their events to its own subscribers.

A `Subject` trait is an `Observable` object whose inputs can change after its creation.

To see how to use a Subject trait in practice, let's assume that we are building our own operating system. Having witnessed how practical the Rx event streams are, we decide to use them throughout our operating system, which we name **RxOS**. To make RxOS pluggable, its functionality is divided into separate components called kernel modules. Each kernel module might define a certain number of `Observable` objects. For example, a `TimeModule` module exposes an `Observable` object named `systemClock`, which outputs a string with the system uptime every second:

```
object TimeModule {
  import Observable._
  val systemClock = interval(1.seconds).map(t => s"systime: $t")
}
```

System output is an essential part of every operating system. We want RxOS to output important system events such as the system uptime. We already know how to do this by calling `subscribe` on the `systemClock` object from the `TimeModule` module, as shown in the following code:

```
object RxOS {
  val messageBus = TimeModule.systemClock.subscribe(log _)
}
```

Let's say that another team now independently develops another kernel module named `FileSystemModule`, which exposes an `Observable` object called `fileModifications`. This `Observable` object emits a filename each time a file is modified:

```
object FileSystemModule {
  val fileModifications = modified(".")
}
```

Our core development team now decides that the `fileModifications` objects are important system events and wants to log these events as part of the `messageBus` subscription. We now need to redefine the singleton object `RxOS`, as shown in the following code snippet:

```
object RxOS {
  val messageBus = Observable.items(
    TimeModule.systemClock,
    FileSystemModule.fileModifications
  ).flatten.subscribe(log _)
}
```

This patch solves the situation, but what if another kernel module introduces another group of important system events? With our current approach, we will have to recompile the RxOS kernel each time some third-party developer implements a kernel module. Even worse, the `RxOS` object definition references kernel modules, and thus, depends on them. Developers who want to build custom, reduced versions of RxOS now need to tweak the kernel source code.

This is the classic culprit of the bottom-up programming style; we are unable to declare the `messageBus` object without declaring its dependencies, and declaring them binds us to specific kernel modules.

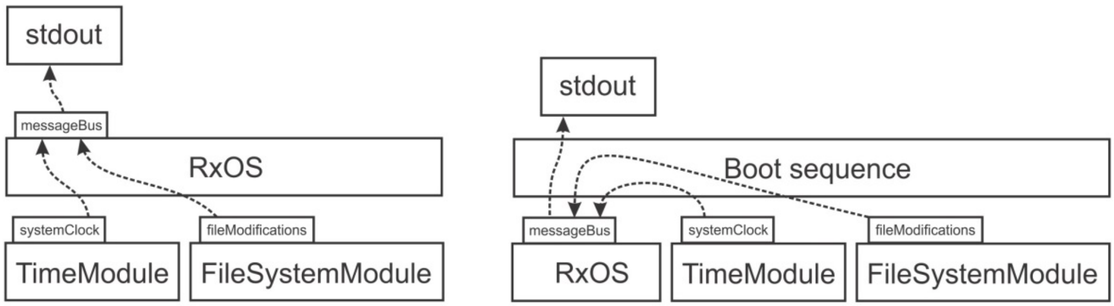
We now redefine the `messageBus` object as an Rx subject. We create a new `Subject` instance that emits strings, and we then subscribe to it, as shown in the following example:

```
object RxOS {
  val messageBus = Subject[String]()
  messageBus.subscribe(log _)
}
```

At this point, the `messageBus` object is not subscribed to any `Observable` objects and does not emit any events. We can now define the RxOS boot sequence separately from the modules and the kernel code. The boot sequence specifies which kernel modules to subscribe with the `messageBus` object, and stores their subscriptions into the `loadedModules` list:

```
object SubjectsOS extends App {
  log(s"RxOS boot sequence starting...")
  val loadedModules = List(
    TimeModule.systemClock,
    FileSystemModule.fileModifications
  ).map(_.subscribe(RxOS.messageBus))
  log(s"RxOS boot sequence finished!")
  Thread.sleep(10000)
  for (mod <- loadedModules) mod.unsubscribe()
  log(s"RxOS going for shutdown")
}
```

The boot sequence first subscribes the `messageBus` object to each of the required modules. We can do this because the `messageBus` object is an `Observer` object, in addition to being an `Observable` object. The `RxOS` then stays up for 10 seconds before calling `unsubscribe` on the modules and shutting down. During this time, the system clock emits an event to the `messageBus` object every second. Similarly, the `messageBus` object outputs the name of the modified file every time a file modification occurs, as shown in the following diagram:



The difference between the two approaches is shown in the preceding figure. In the bottom-up approach, we first need to define all the kernel modules and then make RxOS depend on them. In the top-down approach, RxOS does not depend on the kernel modules. Instead, it is glued together with them by the boot sequence module. The RxOS clients no longer need to tweak or recompile the kernel code if they want to add a new kernel module. In fact, the new design even allows hot-plugging kernel modules into a running RxOS instance, long after the boot sequence is completed.

TIP

Use `Subject` instances when you need to create an `Observable` object whose inputs are not available when the `Observable` object is created.

In our example, designing a web browser is a lot like ordering a MacBook. After specifying the preferred processor type and the hard disk size, the MacBook is assembled, and its components cannot be exchanged easily. Analogously, after implementing the browser's UI layout, the event streams that describe the interaction between UI components are declared only once, and cannot change if the UI components are replaced. On the other hand, building an OS is more like building a desktop computer from custom components. After putting the motherboard into the case, we can plug in components such as the graphics card or the RAID controller independently. Similarly, after declaring the `messageBus` subject, we can plug in any number of kernel modules at any time during the execution of the program.

Although the `Subject` interface is more flexible than the `Observable` interface, you should not always use the `Subject` instances and rely exclusively on the top-down programming style. While declaring the dependencies of an `Observable` object at its creation point makes the application less flexible, it also makes it more declarative and easier to understand. Modern large-scale applications usually combine both bottom-up and top-down approaches.

Rx defines several other types of subject. The type `ReplaySubject` is a `Subject` implementation that buffers the events it receives as an `Observer` object. When another `Observer` object subscribes to a `ReplaySubject` instance, all the events previously buffered by the `ReplaySubject` instance are replayed. In the following code snippet, we define a `ReplaySubject` instance called `messageLog` in `RxOS`:

```
object RxOS {
  val messageBus = Subject[String]()
  val messageLog = subjects.ReplaySubject[String]()
  messageBus.subscribe(log _)
  messageBus.subscribe(messageLog)
}
```

The `messageLog` object subscribes to the `messageBus` object in order to buffer all the system messages. If, for example, we now want to dump all the messages into a log file, we can subscribe to the `messageLog` object immediately before the application ends, as shown in the following example:

```
log(s"RxOS dumping the complete system event log")
RxOS.messageLog.subscribe(logToFile)
log(s"RxOS going for shutdown")
```

Rx also defines two other subjects called `BehaviorSubject` and `AsyncSubject`. The `BehaviorSubject` class buffers only the most recent event, and the `AsyncSubject` class only emits the event immediately preceding `onComplete`. We will not study their exact semantics and use case here, but we refer you to the online documentation to find out more about them.

Summary

First-class event streams are an extremely expressive tool for modelling dynamic, event-based systems with time-varying values. Rx `Observable` objects are an event stream implementation designed to build scalable, concurrent, event-based applications. In this chapter, we saw how to create Rx `Observable` objects and how to subscribe to their events. We studied the `Observable` contract and learned how to compose complex `Observable` objects from simple ones. We investigated various ways of recovering from failures and saw how to use Rx schedulers to transfer events between threads. Finally, we learned how to design loosely coupled systems with Rx subjects. These powerful tools together allow us to build a plethora of different applications, ranging from web browsers, FTP servers, the music and video players to real-time games and trading platforms, and even operating systems.

Due to the increasing popularity of reactive programming, a number of frameworks similar to Rx have appeared in the recent years: REScala, Akka Streams, and Reactive Collections, to name a few. We did not study the semantics of these frameworks in this chapter, but leave it to the readers to explore them on their own.

We have seen that `Observable` objects are very declarative in nature, making the Rx programming model easy to use and understand. Nevertheless, it is sometimes useful to model a system imperatively, using explicit state. In the next chapter, we will study software transactional memory, which allows accessing shared program state without the risk of deadlocks and race conditions, which we learned about in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*.

Exercises

In the following exercises, you will need to implement different `Observable` objects. The exercises show different use cases for `Observable` objects, and contrast the different ways of creating `Observable` objects. Also, some of the exercises introduce new reactive programming abstractions, such as reactive maps and reactive priority queues.

1. Implement a custom `Observable[Thread]` object that emits an event when it detects that a thread was started. The implementation is allowed to miss some of the events.
2. Implement an `Observable` object that emits an event every 5 seconds and every 12 seconds, but not if the elapsed time is a multiple of 30 seconds. Use functional combinators on `Observable` objects.

3. Use the `randomQuote` method from this section in order to create an `Observable` object with the moving average of the quote lengths. Each time a new quote arrives, a new average value should be emitted.
4. Implement the reactive signal abstraction, represented with the `Signal[T]` type. The type `Signal[T]` comes with the method `apply`, used to query the last event emitted by this signal, and several combinators with the same semantics as the corresponding `Observable` methods:

   ```
   class Signal[T] {
     def apply(): T = ???
     def map(f: T => S): Signal[S] = ???
     def zip[S](that: Signal[S]): Signal[(T, S)] = ???
     def scan[S](z: S)(f: (S, T) => S) = ???
   }
   ```

   Then, add the method `toSignal` to the type `Observable[T]`, which converts an `Observable` object to a reactive signal:

   ```
   def toSignal: Signal[T] = ???
   ```

   Consider using Rx subjects for this task.

5. Implement the reactive cell abstraction, represented with the type `RCell[T]`:

   ```
   class RCell[T] extends Signal[T] {
     def :=(x: T): Unit = ???
   }
   ```

   A reactive cell is simultaneously a reactive signal from the previous exercise. Calling the `:=` method sets a new value to the reactive cell, and emits an event.

6. Implement the reactive map collection, represented with the `RMap` class:

   ```
   class RMap[K, V] {
     def update(k: K, v: V): Unit
     def apply(k: K): Observable[V]
   }
   ```

   The `update` method behaves like the update on a regular `Map` collection. Calling `apply` on a reactive map returns an `Observable` object with all the subsequent updates of the specific key.

7. Implement the reactive priority queue, represented by the `RPriorityQueue` class:

   ```
   class RPriorityQueue[T] {
     def add(x: T): Unit = ???
     def pop(): T = ???
     def popped: Observable[T] = ???
   }
   ```

   The reactive priority queue exposes the `Observable` object `popped`, which emits events whenever the smallest element in the priority queue gets removed by calling `pop`.

8. Implement the `copyFile` method, which copies a file specified with the `src` parameter to the destination specified with the `dest` parameter. The method returns an `Observable[Double]` object, which emits an event with the file transfer progress every 100 milliseconds:

   ```
   def copyFile(src: String, dest: String): Observable[Double]
   ```

   The resulting `Observable` object must complete if the file transfer completes successfully, or otherwise fail with an exception.

9. Create a custom Swing component, called `RxCanvas`, which exposes mouse events using `Observable` objects:

   ```
   class RxCanvas extends Component {
     def mouseMoves: Observable[(Int, Int)]
     def mousePresses: Observable[(Int, Int)]
     def mouseReleases: Observable[(Int, Int)]
   }
   ```

   Use the `RxCanvas` component to build your own Paint program, in which you can drag lines on the canvas using a brush, and save the contents of the canvas to an image file. Consider using nested `Observable` objects to implement dragging.

10. Implement a method called `scatterGather` on the type `Observable`, which forwards every event to one of the worker threads, performs some work on those threads, and emits the computed results on a new `Observable` object. The signature of this method is as follows, where type `T` is the type of the events in the original `Observable`:

    ```
    def scatterGather[S](f: T => S): Observable[S]
    ```

11. Implement the `sorted` method on the type `Observable`, which emits incoming events in the sorted order. The events can be emitted only after the original `Observable` terminates.

7
Software Transactional Memory

> *"Everybody who learns concurrency and thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all."*
>
> *- Herb Sutter*

While investigating the fundamental primitives of concurrency in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, we recognized the need for protecting parts of the program from shared access. We saw that a basic way of achieving this isolation is the `synchronized` statement, which uses intrinsic object locks to ensure that at most a single thread executes a specific part of the program at the same time. The disadvantage of using locks is that they can easily cause deadlocks, a situation in which the program cannot progress.

In this chapter, we will introduce **Software Transactional Memory** (**STM**), a concurrency control mechanism for controlling access to shared memory, which greatly reduces the risk of deadlocks and races. An STM is used to designate critical sections of the code. Instead of using locks in order to protect critical sections, STM tracks the reads and writes to shared memory, and serializes critical sections with interleaving reads and writes. The `synchronized` statement is replaced with the atomic blocks that express segments of the program that need to be executed in isolation. STM is safer and easier to use, and at the same time, guarantees relatively good scalability.

The idea of *memory transactions* stems from database transactions, which ensure that a sequence of database queries occurs in isolation. A memory transaction is a sequence of reads and writes to shared memory that logically occur at a single point in time. When a memory transaction T occurs, concurrent memory transactions observe the state of the memory either before the transaction T started, or after the transaction T completed, but not the intermediate states during the execution of T. This property is called **isolation**.

As we will see, **composability** is another important advantage of using an STM. Consider a lock-based hash table implementation with thread-safe `insert` and `remove` operations. While the individual `insert` and `remove` operations can be safely invoked by different threads, it is impossible to implement a method that removes an element from one hash table and adds it to another hash table, without exposing the intermediate state in which the element is not present in either hash table. Traditionally, STM was proposed as a part of the programming language with the advantage that certain transaction limitations can be ensured at compile time. Since this approach requires intrusive changes to a language, many software transactional memories are implemented as libraries. ScalaSTM is one such example. We will use ScalaSTM as the concrete STM implementation. Concretely, we cover the following topics in this chapter:

- The disadvantages of atomic variables
- The semantics and internals of STM
- Transactional references
- The interaction between transactions and external side effects
- Semantics of single operation transactions and nested transactions
- Retrying transactions conditionally and timing out transactions
- Transaction-local variables, transactional arrays, and transactional maps

We already learned in `Chapter 3`, *Traditional Building Blocks of Concurrency*, that using atomic variables and concurrent collections allows expressing lock-free programs. Why not just use atomic variables to express concurrently shared data? To better emphasize the need for STM, we will start by presenting a situation in which atomic variables prove inadequate.

The trouble with atomic variables

Atomic variables from `Chapter 3`, *Traditional Building Blocks of Concurrency*, are one of the fundamental synchronization mechanisms. We already know that volatile variables, introduced in `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, allow race conditions, in which the program correctness is subject to the precise execution schedule of different threads. Atomic variables can ensure that no thread concurrently modifies the variable between a read and a write operation. At the same time, atomic variables reduce the risk of deadlocks. Regardless of their advantages, there are situations when using atomic variables is not satisfactory.

In `Chapter 6`, *Concurrent Programming with Reactive Extensions*, we implemented a minimalistic web browser using the **Rx** framework. Surfing around the Web is great, but we would like to have some additional features in our browser. For example, we would like to maintain the browser's history–the list of URLs that were previously visited. We decide to keep the list of URLs in the Scala `List[String]` collection. Additionally, we decide to track the total character length of all the URLs. If we want to copy the URL strings into an array, this information allows us to quickly allocate an array of an appropriate size.

Different parts of our browser execute asynchronously, so we need to synchronize access to this mutable state. We can keep the list of URLs and their total character length in private mutable fields and use the `synchronized` statement to access them. However, having seen the culprits of the `synchronized` statement in earlier chapters, we decide to avoid locks. Instead, we will use atomic variables. We will store the list of URLs and their total character length in two atomic variables, that are `urls` and `clen`:

```
import java.util.concurrent.atomic._
val urls = new AtomicReference[List[String]](Nil)
val clen = new AtomicInteger(0)
```

Whenever the browser opens URL, we need to update these atomic variables. To do this more easily, we define a helper method called `addUrl`:

```
import scala.annotation.tailrec
def addUrl(url: String): Unit = {
  @tailrec def append(): Unit = {
    val oldUrls = urls.get
    val newUrls = url :: oldUrls
    if (!urls.compareAndSet(oldUrls, newUrls)) append()
  }
  append()
  clen.addAndGet(url.length + 1)
}
```

As we learned in the introductory chapters, we need to use atomic operations on atomic variables to ensure that their values consistently change from one state to another. In the previous code snippet, we use the `compareAndSet` operation to atomically replace the old list of URLs called `oldUrls` with the updated version `newUrls`. As discussed at length in `Chapter 3`, *Traditional Building Blocks of Concurrency*, the `compareAndSet` operation can fail when two threads call it simultaneously on the same atomic variable. For this reason, we define a nested, tail-recursive method, `append`, which calls the `compareAndSet` method and restarts if the `compareAndSet` method fails. Updating the `clen` field is easier. We just call the atomic `addAndGet` method defined on atomic integers.

Other parts of the web browser can use the `urls` and `clen` variables to render the browsing history, dump it to a `log` file or to export browser data, in case our users decide they like Firefox better. For convenience, we define a `getUrlArray` auxiliary method that returns a character array in which the URLs are separated with a newline character. The `clen` field is a quick way to get the required size of the array. We call the `get` method to read the value of the `clen` field and allocate the array. We then call `get` to read the current list of URLs, append the newline character to each URL, flatten the list of strings into a single list, zip the characters with their indices, and store them into the array:

```
def getUrlArray(): Array[Char] = {
  val array = new Array[Char](clen.get)
  val urlList = urls.get
  for ((ch, i) <- urlList.map(_ + "\n").flatten.zipWithIndex) {
    array(i) = ch
  }
  array
}
```

To test these methods, we can simulate user interaction with two asynchronous computations. The first asynchronous computation calls the `getUrlArray` method to dump the browsing history to a file. The second asynchronous computation visits three separate URLs by calling the `addURL` method three times, and then prints the `"done browsing"` string to the standard output:

```
import scala.concurrent._
import ExecutionContext.Implicits.global
object AtomicHistoryBad extends App {
  Future {
    try { log(s"sending: ${getUrlArray().mkString}") }
    catch { case e: Exception => log(s"Houston... $e!") }
  }
  Future {
    addUrl("http://scala-lang.org")
    addUrl("https://github.com/scala/scala")
    addUrl("http://www.scala-lang.org/api")
    log("done browsing")
  }
  Thread.sleep(1000)
}
```

Running this program several times reveals a bug. The program sometimes mysteriously crashes with an `ArrayIndexOutOfBoundsException` exception. By analyzing the `getUrlArray` method, we find the cause to the bug. This bug occurs when the retrieved value of the `clen` field is not equal to the length of the list. The `getUrlArray` method first reads the `clen` atomic variable, and later reads the list of the URLs from the `urls` atomic variable. Between these two reads, the first thread modifies the `urls` variable by adding an additional URL string. By the time `getUrlArray` reads the `urls` variable, the total character length becomes longer than the allocated array, and we eventually get an exception.

This example illustrates an important disadvantage of atomic variables. Although specific atomic operations are themselves atomic and occur at a single point in time, invoking multiple atomic operations is typically not atomic. When multiple threads simultaneously execute multiple atomic operations, the operations might interleave in unforeseen ways and lead to the same kind of race conditions that result from using volatile variables. Note that swapping the updates to the `clen` and `urls` variables does not solve the problem. Although there are other ways to ensure atomicity in our example, they are not immediately obvious.

Reading multiple atomic variables is not an atomic operation and it can observe the program data in an inconsistent state.

When all threads in the program observe that an operation occurs at the same, single point in time, we can say that the operation is *linearizable*. The point in time at which the operation occurs is called a **linearization point**. The `compareAndSet` and `addAndGet` operations are inherently linearizable operations. They execute atomically, usually as a single processor instruction and at a single point in time, from the perspective of all the threads. The `append` nested method in the previous example is also linearizable. Its linearization point is a successful `compareAndSet` operation, because that is the only place where `append` modifies the program state. On the other hand, the `addUrl` and `getUrlArray` methods are not linearizable. They contain no single atomic operation that modifies or reads the state of the program. The `addUrl` method modifies the program state twice. First, it calls the `append` method and then it calls the `addAndGet` method. Similarly, the `getUrlArray` method reads the program state with two separate atomic `get` operations. This is a commonly misunderstood point when using atomic variables, and we say that atomic variables do not compose into larger programs.

We can fix our example by removing the `clen` atomic variable, and computing the required array length after reading the `urls` variable once. Similarly, we can use a single atomic reference to store a tuple with the URL list and the size of that list. Both approaches would make the `addUrl` and `getUrlArray` methods linearizable.

Concurrent programming experts have proven that it is possible to express any program state using atomic variables, and arbitrarily modify this state with linearizable operations. In practice, implementing such linearizable operations efficiently can be quite challenging. It is generally hard to implement arbitrary linearizable operations correctly, and it is even harder to implement them efficiently.

Unlike atomic variables, multiple `synchronized` statements can be used together more easily. We can modify multiple fields of an object when we use the `synchronized` statement, and we can even nest multiple `synchronized` statements. We are thus left with a dilemma. We can use atomic variables and risk race conditions when composing larger programs, or we can revert to using the `synchronized` statement, but risk deadlocks. Luckily, STM is a technology that offers the best of both worlds; it allows you to compose simple atomic operations into more complex atomic operations, without the risk of deadlocks.

Using Software Transactional Memory

In this section, we will study the basics of using STM. Historically, multiple STM implementations were introduced for Scala and the JVM platform. The particular STM implementation described in this chapter is called **ScalaSTM**. There are two reasons that ScalaSTM is our STM of choice. First, ScalaSTM was authored by a group of STM experts who agreed on a standardized set of APIs and features. Future STM implementations for Scala are strongly encouraged to implement these APIs. Second, the ScalaSTM API is designed for multiple STM implementations, and comes with an efficient default implementation. Different STM implementations can be chosen when the program starts. Users can write applications using a standardized API, and seamlessly switch to a different STM implementation later.

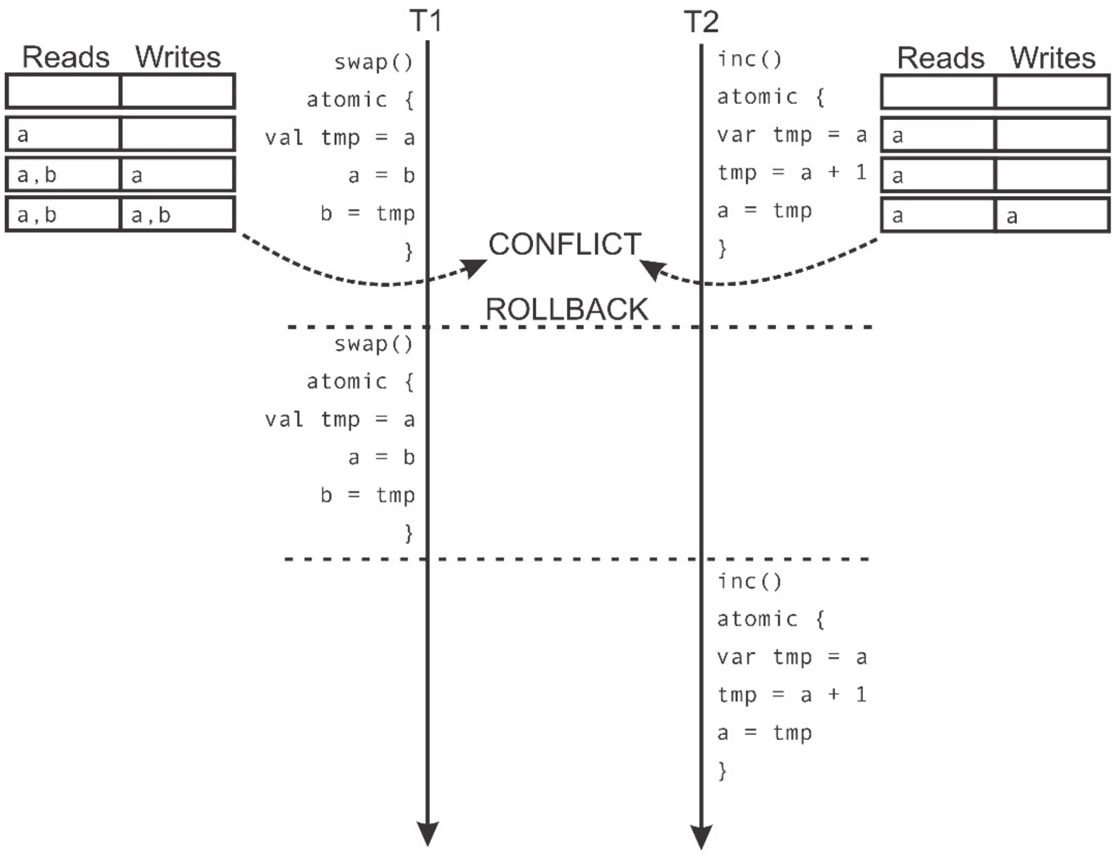
The `atomic` statement is a fundamental abstraction at the core of every STM. When the program executes a block of code marked with the `atomic` symbol, it starts a memory transaction, a sequence of reads and writes operations to memory that occur atomically for other threads in the program. The `atomic` statement is similar to the `synchronized` statement, and ensures that a block of code executes in isolation, without the interference of other threads, thus avoiding race conditions. Unlike the `synchronized` statement, the `atomic` statement does not cause deadlocks.

The following methods, `swap` and `inc`, show how to use the `atomic` statement on a high level. The `swap` method atomically exchanges the contents of two memory locations, `a` and `b`. Between the time that a thread reads the memory location `a` (or `b`) and the time that the `atomic` statement ends, no other thread can effectively modify the value at location `a` (or `b`). Similarly, the `inc` method atomically increments the integer value at the memory location `a`. When a thread, which calls the `inc` method, reads the value of `a` in the `atomic` statement, no other thread can change the value of the memory location `a` until the `atomic` statement ends:

```
def swap() = atomic { // not actual code
  val tmp = a
  a = b
  b = tmp
}
def inc() = atomic { a = a + 1 }
```

The ways in which an STM implements deadlock-freedom and ensures that no two threads simultaneously modify the same memory locations are quite complex. In most STM implementations, the `atomic` statement maintains a log of read and write operations. Every time a memory location is read during a memory transaction, the corresponding memory address is added to the log. Similarly, whenever a memory location is written during a memory transaction, the memory address and the proposed value are written to the log. Once the execution reaches the end of the `atomic` block, all the writes from the transaction log are written to the memory. When this happens, we say that the transaction is committed. On the other hand, during the transaction, the STM might detect that another concurrent transaction performed by some other thread is concurrently reading or writing the same memory location. This situation is called a **transactional conflict**. When a transactional conflict occurs, one or both of the transactions are cancelled, and re-executed serially, one after another. We say that the STM *rolls back* these transactions. Such STMs are called **optimistic**. Optimistic STMs try to execute a transaction under the assumption that it will succeed, and roll back when they detect a conflict. When we say that a transaction is completed, we mean that it was either committed or rolled back, and re-executed.

To illustrate how a memory transaction works, we consider the scenario in which two threads, **T1** and **T2**, simultaneously call the `swap` and `inc` methods. Since both the `atomic` statements in these methods modify the memory location `a`, the execution results in a runtime transactional conflict. During the execution of the program, the STM detects that the entries in the transactional logs overlap: the transaction associated with the `swap` method has both memory locations `a` and `b` in its read and write sets, while the `inc` method has `a` in its read and write sets. This indicates a potential conflict. Both the transactions can be rolled back, and then executed serially one after another, as shown in the following diagram:



We will not delve more deeply into the internals of the ScalaSTM implementation, as this is beyond the scope of this book. Instead, we will focus on how to use ScalaSTM to easily write concurrent applications. Where reasonable, we hint at some implementation details to better understand the reasons behind the ScalaSTM semantics.

In some STMs, the `atomic` statement tracks all the reads and writes to the memory. ScalaSTM only tracks specially marked memory locations within transactions. There are several reasons for this. First, an STM cannot ensure safety if some parts of the program access memory locations outside the `atomic` statements, while other parts access the same memory locations inside the `atomic` statements. ScalaSTM avoids accidental uses outside transactions by explicitly marking the memory locations that can only be used in transactions. Second, STM frameworks for the JVM need to use post-compilation or bytecode introspection in order to accurately capture all the reads and writes. ScalaSTM is a library-only STM implementation, so it cannot analyze and transform the program in the same way a compiler can.

In ScalaSTM, the effects of the `atomic` statement are limited to special objects called **transactional references**. Before showing how to use the `atomic` statement to perform memory transactions, we will study how to create transactional references.

Transactional references

In this section, we will study how to declare transactional references. A transactional reference is a memory location that provides transactional read and write access to a single memory location. In ScalaSTM, transactional references to the values of type `T` are encapsulated within the objects of the `Red[T]` type:

Before we begin using STM in Scala, we need to add an external dependency to our project, since ScalaSTM is not a part of the Scala standard library:

```
libraryDependencies += "org.scala-stm" %% "scala-stm" % "0.7"
```

To use the ScalaSTM `atomic` statement in a compilation unit, we import the contents of the `scala.concurrent.stm` package:

```
import scala.concurrent.stm._
```

To instantiate a `Ref` object, we use the `Ref.apply` factory method on the `Ref` companion object. Let's rewrite our browser history example using transactional memory. We start by replacing atomic variables with transactional references. We pass the initial value of each transactional reference to the `Ref.apply` method:

```
val urls = Ref[List[String]](Nil)
val clen = Ref(0)
```

Calling the `apply` method on a transactional reference returns its value, and calling the `update` method modifies it. However, we cannot call these methods from outside of a transaction. The `apply` and `update` methods take an implicit argument of type `InTxn` (which stands for *in transaction*), which designates that a transaction is under way. Without the `InTxn` object, we cannot call the `apply` and `update` methods. This constraint protects us from accidentally circumventing the ScalaSTM safety mechanisms.

To read and modify transactional references, we must first start a transaction that provides the implicit `InTxn` object. We will study how to do this next.

Using the atomic statement

After redefining the `urls` and `clen` variables as transactional references, we redefine the `addUrl` method. Instead of separately updating two atomic variables, we start a memory transaction with the `atomic` statement. In ScalaSTM, the `atomic` statement takes a block of type `InTxn => T`, where `InTxn` is the type of the previously mentioned transaction object, and `T` is the type of the return value of the transaction. Note that we can annotate the `InTxn` parameter with the `implicit` keyword:

```
def addUrl(url: String): Unit = atomic { implicit txn =>
  urls() = url :: urls()
  clen() = clen() + url.length + 1
}
```

The new definition of `addUrl` is surprisingly simple. It first reads the value of the `urls` list, prepends a new URL to the list, and assigns the updated list back to the `urls` variable. Then, it reads the current value of the total character length `clen`, increments it by the length of the new URL, and assigns the new value back to `clen`. Note that the new definition of the `addUrl` method looks almost identical to a single-threaded implementation.

An important limitation of the `atomic` statement in ScalaSTM is that it does not track reads and writes to ordinary local variables and object fields. As we will see later, these are considered as arbitrary side effects, and are not allowed inside the transaction.

We reimplement the `getUrlArray` method in a similar fashion. We start by creating a transaction with the `atomic` statement. The value of the `clen` variable is used in order to allocate a character array of an appropriate size. We then read the `urls` list and assign its characters to the array in a `for` loop. Again, the implementation of the `getUrlArray` method looks surprisingly similar to the corresponding single-threaded implementation:

```
def getUrlArray(): Array[Char] = atomic { implicit txn =>
  val array = new Array[Char](clen())
  for ((ch, i) <- urls().map(_ + "\n").flatten.zipWithIndex) {
    array(i) = ch
  }
  array
}
```

This time, there is no danger of seeing inconsistent values of the `clen` and `urls` variables. When used in a transaction, the two values are always consistent with each other, as shown in the following program:

```
object AtomicHistorySTM extends App {
  Future {
    addUrl("http://scala-lang.org")
    addUrl("https://github.com/scala/scala")
    addUrl("http://www.scala-lang.org/api")
    log("done browsing")
  }
  Thread.sleep(25)
  Future {
    try { log(s"sending: ${getUrlArray().mkString}") }
    catch { case e: Exception => log(s"Ayayay... $e") }
  }
  Thread.sleep(5000)
}
```

Note that we added the `sleep` statement in the main program, as this sets the timing of the two asynchronous computations to occur approximately at the same time. You can tweak the duration of the `sleep` statement in order to observe the various interleavings of the two asynchronous computations. Convince yourself with the fact that dumping the browsing history to the `log` file always observes some prefix of the three `addUrl` calls, and does not throw an exception.

TIP

When encoding a complex program state, use multiple transactional references. To atomically perform multiple changes on the program state, use the `atomic` statement.

Having seen basic way of using the `atomic` statement with transactional references, we will proceed to show more advanced examples and study the STM semantics in more detail.

Composing transactions

When used correctly, transactional memory is a powerful tool for building concurrent applications that modify shared data. Nevertheless, no technology is a silver bullet, and neither is STM. In this section, we will study how to compose transactions in larger programs and learn how transactional memory interacts with other features of Scala. We investigate some of the caveats of STM, and go beyond transactional references and the `atomic` statement blocks to show how to use STM more effectively.

The interaction between transactions and side effects

Previously, we learned that an STM may roll back and retry a transaction. An attentive reader might notice that retrying a transaction means re-executing its side effects. Here, the side effects are arbitrary reads and writes to regular `object` fields and variables.

Sometimes, side effects are not a problem. Transactional references cannot be modified outside a transaction, and inside a transaction their modifications are aborted when retrying. Still, the other kinds of side effect are not rolled back. Consider the following program:

```
object CompositionSideEffects extends App {`
  val myValue = Ref(0)
  def inc() = atomic { implicit txn =>
    log(s"Incrementing ${myValue()}")
    myValue() = myValue() + 1
  }
  Future { inc() }
  Future { inc() }
  Thread.sleep(5000)
}
```

The preceding program declares a `myValue` transactional reference, and an `inc` method that increments `myValue` inside an `atomic` block. The `inc` method also contains a `log` statement which prints the current value of the `myValue` reference. The program asynchronously calls the `inc` method twice. Upon executing this program, we get the following output:

```
ForkJoinPool-1-worker-1: Incrementing 0
ForkJoinPool-1-worker-3: Incrementing 0
ForkJoinPool-1-worker-3: Incrementing 1
```

The two asynchronous computations call the `inc` method at the same time, and both start a transaction. One of the transactions adds the `myValue` reference to its read set, calls the `log` statement with the `0` value, and proceeds to increment the `myValue` reference by adding the `myValue` reference to its write set. In the meantime, the other transaction first logs the `0` value, then attempts to read `myValue` again, and detects that `myValue` is in a write set of another active transaction. The second transaction is rolled back, and retried after the first transaction commits. The second transaction reads the `myValue` reference once more, prints `1`, and then increments `myValue`. The two transactions commit, but the side-effecting `log` call is executed three times as a result of the rollback.

It might not be harmful to execute a simple `log` statement multiple times, but repeating arbitrary side effects can easily break the correctness of a program. Avoiding side effects in transactions is a recommended practice.

Recall that an operation is idempotent if executing it multiple times has the same effect as executing it once, as discussed in `Chapter 6`, *Concurrent Programming with Reactive Extensions*. You might conclude that, if a side-effecting operation is idempotent, then it is safe to execute it in a transaction. After all, the worst thing that can happen is that the idempotent operation gets executed more than once, right? Unfortunately, this reasoning is flawed. After a transaction is rolled back and retried, the values of the transactional references might change. The second time a transaction is executed, the arguments to the idempotent operation might be different, or the idempotent operation might not be invoked at all. The safest way to avoid such situations is to avoid external side effects altogether.

TIP

Avoid external side effects inside the transactions, as transactions can be re-executed multiple times.

In practice, we usually want to execute a side effect only if the transaction commits, that is, after we are sure that the changes to the transactional references are visible to other threads. To do this, we use the `Txn` singleton object, which can schedule multiple operations that execute after the transaction commits or rolls back.

After a rollback, these operations are removed, and potentially re-registered when retrying the transaction. Its methods can only be called from inside an active transaction. In the following code, we rewrite the `inc` method to call the `Txn` object's `afterCommit` method, and schedule the `log` statement to execute after the transaction commits:

```
def inc() = atomic { implicit txn =>
  val valueAtStart = myValue()
  Txn.afterCommit { _ =>
    log(s"Incrementing $valueAtStart")
  }
  myValue() = myValue() + 1
}
```

Note that we read the `myValue` reference inside the transaction and assign the value to a local variable `valueAtStart`. The value of the `valueAtStart` local variable is later printed to the standard output. This is different from reading the `myValue` reference inside the `afterCommit` block:

```
def inc() = atomic { implicit txn =>
  Txn.afterCommit { _ =>
    log(s"Incrementing ${myValue()}") // don't do this!
  }
  myValue() = myValue() + 1
}
```

Calling the last version of `inc` fails with an exception. Although the transactional context `txn` exists when the `afterCommit` method is called, the `afterCommit` block is executed later, after the transaction is already over and the `txn` object is no longer valid. It is illegal to read or modify transactional references outside a transaction. Before using it in an `afterCommit` block, we need to store the value of the transactional reference into a local variable in the transaction itself.

Why does accessing a transactional reference inside the `afterCommit` block only fail at runtime, when the transaction executes, instead of failing during compilation? The `afterCommit` method is in the **static scope** of the transaction, or, in other words is statically nested within an `atomic` statement. For this reason, the compiler resolves the `txn` object of the transaction, and allows you to access the transactional references, such as `myValue`. However, the `afterCommit` block is not executed in the dynamic scope of the transaction. In other words, the `afterCommit` block is run *after* the `atomic` block returns.

By contrast, accessing a transactional reference outside of the `atomic` block is not in the static scope of a transaction, so the compiler detects this and reports an error.

In general, the `InTxn` objects must not escape the transaction block. For example, it is not legal to start an asynchronous operation from within the transaction, and use the `InTxn` object to access transactional references.

> **TIP:** Only use the transactional context within the thread that started the transaction.

In some cases, we want to execute some side-effecting operations when a rollback occurs. For instance, we would like to log each rollback to track the contention in our program. This information can help us restructure the program and eliminate potential performance bottlenecks. To achieve this, we use the `afterRollback` method:

```
def inc() = atomic { implicit txn =>
  Txn.afterRollback { _ =>
    log(s"rollin' back")
  }
  myValue() = myValue() + 1
}
```

Importantly, after a rollback, the transaction is no longer under way. Just as in the `afterCommit` blocks, it is illegal to access the transactional references in the `afterRollback` blocks.

> **TIP:** Use the `Txn` object's `afterCommit` and `afterRollback` methods to perform side-effecting operations in transactions without the danger of executing them multiple times.

Not all side-effecting operations inside the transactions are bad. As long as the side effects are confined to mutating objects that are created inside the transaction, we are free to use them. In fact, such side effects are sometimes necessary. To demonstrate this, let's define the `Node` class for a transactional linked list collection. A transactional list is a concurrent, thread-safe linked list that is modified using memory transactions. Similar to a functional cons list, represented by the `List` class in Scala, the transactional `Node` class contains two fields that we call `elem` and `next`. The `elem` field contains the value of the current node. To keep things simple, the `elem` field is a value field and can only contain integers.

The `next` field is a transactional reference containing the next node in the linked list. We can read and modify the `next` field only inside memory transactions:

```
case class Node(elem: Int, next: Ref[Node])
```

We now define a `nodeToString` method, which takes a transactional linked list node `n`, and creates a `String` representation of the transactional list starting with the `n` node:

```
def nodeToString(n: Node): String = atomic { implicit txn =>
  val b = new StringBuilder
  var curr = n
  while (curr != null) {
    b ++= s"${curr.elem}, "
    curr = curr.next()
  }
  b.toString
}
```

In the preceding code snippet, we were careful to confine the side effects to objects that were created inside the transaction, in this case, the `StringBuilder` object `b`. Had we instantiated the `StringBuilder` object before the transaction started, the `nodeToString` method would not work correctly:

```
def nodeToStringWrong(n: Node): String = {
  val b = new StringBuilder // very bad
  atomic { implicit txn =>
    var curr = n
    while (curr != null) {
      b ++= s"${curr.elem}, "
      curr = curr.next()
    }
  }
  b.toString
}
```

If the transaction gets rolled back in the `nodeToStringWrong` example, the contents of the `StringBuilder` object are not cleared. The second time a transaction runs, it will modify the already existing, non-empty `StringBuilder` object and return a string representation that does not correspond to the state of the transactional list.

When mutating an object inside a transaction, make sure that the object is created inside the transaction and that the reference to it does not escape the scope of the transaction.

Having seen how to manage side effects inside transactions, we now examine several special kinds of transactions and study how to compose smaller transactions into larger ones.

Single-operation transactions

In some cases, we only want to read or modify a single transactional reference. It can be cumbersome to type the `atomic` keyword and the implicit `txn` argument just to read a single `Ref` object. To alleviate this, ScalaSTM defines single-operation transactions on transactional references. Single-operation transactions are executed by calling a single method on a `Ref` object. This method returns a `Ref.View` object, which has the same interface as a `Ref` object, but its methods can be called from outside a transaction. Each operation on a `Ref.View` object acts like a single-operation transaction.

Recall the `Node` class for transactional linked lists from the previous section, which stored integers in an `elem` field, and the reference to the next node in the transactional reference called `next`. Let's augment `Node` with two linked list methods. The `append` method takes a single `Node` argument `n`, and inserts `n` after the current node. The `nextNode` method returns the reference to the next node, or `null` if the current node is at the end of the list:

```
case class Node(val elem: Int, val next: Ref[Node]) {
  def append(n: Node): Unit = atomic { implicit txn =>
    val oldNext = next()
    next() = n
    n.next() = oldNext
  }
  def nextNode: Node = next.single()
}
```

The `nextNode` method does a single-operation transaction. It calls single on the `next` transactional reference, and then calls the `apply` method in order to obtain the value of the next node. This is equivalent to the following definition:

```
def nextNode: Node = atomic { implicit txn =>
  next()
}
```

We can use our transactional `Node` class to declare a linked list called `nodes`, initially containing values `1`, `4`, and `5`, and then concurrently modify it. We start two futures `f` and `g`, which call `append` to add nodes with the values `2` and `3`, respectively. After the futures complete, we call `nextNode` and print the value of the next node. The following code snippet will print the node with either the value `2` or `3`, depending on which future completes last:

```
val nodes = Node(1, Ref(Node(4, Ref(Node(5, Ref(null))))))
val f = Future { nodes.append(Node(2, Ref(null))) }
val g = Future { nodes.append(Node(3, Ref(null))) }
for (_ <- f; _ <- g) log(s"Next node is: ${nodes.nextNode}")
```

We can also use the `single` method to invoke other transactional reference operations. In the following code snippet, we use the `transform` operation to define an `appendIfEnd` method on the `Node` class, which appends a node `n` after the current node only if the current node is followed by `null`:

```
def appendIfEnd(n: Node) = next.single.transform {
  oldNext => if (oldNext == null) n else oldNext
}
```

The `transform` operation on a `Ref` object containing the values of type `T` takes a transformation function of type `T => T`. It atomically performs a read of the transactional reference, applies the transformation function to the current value, and writes the new value back. Other single-operation transactions include `update`, `compareAndSet`, and `swap` operations. We refer the readers to the online documentation to learn their precise semantics.

> TIP: Use single-operation transactions for single read, write, and CAS-like operations in order to avoid the syntactic boilerplate associated with the `atomic` blocks.

Single-operation transactions are convenience methods that are easier to type, and are possibly more efficient, depending on the underlying STM implementation. They can be useful, but as programs grow, we are more interested in building larger transactions from simple ones. We will investigate how to do this in the next section.

Nesting transactions

Recall from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, that a `synchronized` statement can be nested inside other `synchronized` statements. This property is essential when composing programs from multiple software modules. For example, a money transfer module in a banking system must call operations from a logging module to persist the transactions. Both the modules might internally use arbitrary sets of locks, without the knowledge of other modules. An unfortunate disadvantage of arbitrarily nested `synchronized` statements is that they allow the possibility of a deadlock.

Separate `atomic` statements can also nest arbitrarily. The motivation for this is the same as with the `synchronized` statement. A transaction inside a software module must be able to invoke operations inside other software modules, which themselves might start the transactions. Not having to know about the transactions inside an operation allows a better separation between different software components.

Let's illustrate this with a concrete example. Recall the `Node` class from the previous section, which was used for transactional linked lists. The `Node` class was somewhat low-level. We can only call the `append` method to insert new nodes after the specified node, and call `nodeToString` on a specific node to convert its elements to a `String` object.

In this section, we define the transactional sorted list class, represented by the `TSortedList` class. This class stores integers in ascending order. It maintains a single transactional reference `head`, which points to the head of the linked list of the `Node` objects. We define the `toString` method on the `TSortedList` class to convert its contents into a textual representation. The `toString` method needs to read the transactional reference `head`, so it starts by creating a new transaction. After reading the value of the `head` transactional reference into a local value `headNode`, the `toString` method can reuse the `nodeToString` method that we defined earlier:

```
class TSortedList {
  val head = Ref[Node](null)
  override def toString: String = atomic { implicit txn =>
    val h = head()
    nodeToString(h)
  }
}
```

Recall that the `nodeToString` method starts another transaction to read the next references in each node. When the `toString` method calls `nodeToString`, the second transaction becomes *nested* in the transaction started by `toString`. The `atomic` block in the `nodeToString` method does not start a new, separate transaction. Instead, the nested transaction becomes a part of the existing transaction. This has two important consequences. First, if the nested transaction fails, it is not rolled back to the start of its `atomic` block in the `nodeToString` method. Instead, it rolls back to the start of the `atomic` block in the `toString` method. We say that the start of the transaction is determined by the dynamic scope, rather than the static scope. Similarly, the nested transaction does not commit when it reaches the end of the `atomic` block in the `nodeToString` method. The changes induced by the nested transaction become visible when the initial transaction commits. We say that the scope of the transaction is always that of the top-level transaction.

> Nested `atomic` blocks result in a transaction that starts when the top-level `atomic` block starts, and can commit only after the top-level `atomic` block completes. Similarly, rollbacks retry the transaction starting from the top-level `atomic` block.

We now study another example of using nested transactions. Atomically converting transactional sorted lists to their string representation is useful, but we also need to insert elements in the list. We define the `insert` method, which takes an integer and inserts it into a proper position in the transactional list.

Since `insert` can modify both the transactional reference `head` and the nodes in the list, it starts by creating a transaction. It then checks for two special cases. A list can be empty, in this case we set `head` to a new node containing `x`. Likewise, the `x` integer might be smaller than the first value in the list; in which case, the `head` reference is set to a new node containing `x`, and its `next` field is set to the previous value of the `head` reference. If neither of these conditions applies, we call a tail-recursive, nested method `insert` to process the remainder of the list:

```
import scala.annotation.tailrec
def insert(x: Int): this.type = atomic { implicit txn =>
  @tailrec def insert(n: Node): Unit = {
    if (n.next() == null || n.next().elem > x)
      n.append(new Node(x, Ref(null)))
    else insert(n.next())
  }
  if (head() == null || head().elem > x)
    head() = new Node(x, Ref(head()))
  else insert(head())
  this
}
```

The nested `insert` method traverses the linked list in order to find the correct position for the `x` integer. It takes the current node `n` and checks if the node is followed by `null`, indicating the end of the list, or if the next element is greater than `x`. In both cases, we call the `append` method on the node. If the node following `n` is not `null`, and its `elem` field is less than or equal to `x`, we call `insert` recursively on the next node.

Note that the tail-recursive, nested method `insert` uses the transactional context `txn` of the enclosing `atomic` block. We can also define a separate tail-recursive method `insert` outside the scope of the transaction. In this case, we need to encode the transactional context `txn` as a separate implicit parameter:

```
@tailrec
final def insert(n: Node, x: Int)(implicit txn: InTxn): Unit = {
  if (n.next() == null || n.next().elem > x)
    n.append(new Node(x, Ref(null)))
  else insert(n.next(), x)
}
```

Alternatively, we can omit the implicit `txn` transactional context parameter, but then we have to start a nested transaction inside the tail-recursive `insert` method. This might be slightly less efficient than the previous approach, but it is semantically equivalent:

```
@tailrec
final def insert(n: Node, x: Int): Unit = atomic { implicit txn =>
  if (n.next() == null || n.next().elem > x)
    n.append(new Node(x, Ref(null)))
  else insert(n.next(), x)
}
```

We test our transactional sorted list with the following snippet. We instantiate an empty transactional sorted list and insert several integers concurrently from the asynchronous computations `f` and `g`. After both the corresponding futures complete execution, we print the contents of the sorted list:

```
val sortedList = new TSortedList
val f = Future { sortedList.insert(1); sortedList.insert(4) }
val g = Future { sortedList.insert(2); sortedList.insert(3) }
for (_ <- f; _ <- g) log(s"sorted list - $sortedList")
```

Running the preceding snippet always outputs the elements `1`, `2`, `3`, and `4` in the same sorted order, regardless of the execution schedule of the futures. We created a thread-safe transactional sorted list class, and the implementation is almost identical to the corresponding sequential sorted list implementation. This example shows the true potential of STM. It allows you to create concurrent data structures and thread-safe data models without having to worry too much about concurrency.

There is one more aspect of transactions that we have not yet considered. What happens if a transaction fails due to an exception? For example, the tail-recursive `insert` method can get called with a `null` value instead of a valid `Node` reference. This results in throwing a `NullPointerException`, but how does it affect the transaction? We will explore the exception semantics of the transactions in the following section.

Transactions and exceptions

From what we've learned about transactions so far, it is not clear what happens with a transaction if it throws an exception. An exception could roll back the transaction, or it could commit its changes. ScalaSTM does a rollback, by default, but this behavior can be overridden.

Let's assume that the clients of our transactional sorted list want to use it as a concurrent priority queue. A *priority queue* is a collection that contains ordered elements, such as integers. An arbitrary element can be inserted into a priority queue using the `insert` method. At each point, we can retrieve the smallest element currently in the priority queue using the `head` method. The priority queue also allows you to remove the smallest element with the `pop` method.

The transactional sorted list is already sorted and supports element insertion with the `insert` method, however, once added, elements cannot be removed. To make our transactional sorted list usable as a priority queue, we define a `pop` method, which removes the first `n` elements from a transactional list `xs`. We start a transaction inside the `pop` method, and declare a local variable `left`, initializing it with the number of removed elements `n`. We then use a `while` loop to remove nodes from `head` and decrease the `left` variable until it becomes 0:

```
def pop(xs: TSortedList, n: Int): Unit = atomic { implicit txn =>
  var left = n
  while (left > 0) {
    xs.head() = xs.head().next()
    left -= 1
  }
}
```

To test the `pop` method, we declare a new transactional list `lst`, and insert integers `4`, `9`, `1`, and `16`. The list is sorted, so the integers appear in the list in the order `1`, `4`, `9`, and `16`:

```
val lst = new TSortedList
lst.insert(4).insert(9).insert(1).insert(16)
```

Next, we start an asynchronous computation that removes the first two integers in the list by calling `pop`. After the asynchronous computation is successfully completed, we print the contents of the transactional list to the standard output:

```
Future { pop(lst, 2) } foreach {
  case _ => log(s"removed 2 elements; list = $lst")
}
```

So far, so good. The `log` statement outputs the list with the elements `9` and `16`. We proceed by starting another asynchronous computation, which removes the first three elements from the transactional list:

```
Future { pop(lst, 3) } onComplete {
  case Failure(t) => log(s"whoa $t; list = $lst")
}
```

However, when we call the `pop` method again, it throws a `NullPointerException`; there are only two elements left in the transactional list. As a result, the reference `head` is eventually assigned `null` during the transaction. When the `pop` method tries to call `next` on `null`, an exception is thrown.

In the `onComplete` callback, we output the name of the exception and the contents of the transactional list. It turns out that the transactional list still contains the elements `9` and `16`, although the `head` reference of the transactional list had been set to `null` in the transaction. When an exception is thrown, the effects of the transaction are reverted.

> When an exception is thrown inside a transaction, the transaction is rolled back and the exception is rethrown at the point where the top-level `atomic` block started.

Importantly, the nested transactions are also rolled back. In the following code snippet, the nested `atomic` block in the `pop` method completes successfully, but its changes are not committed. Instead, the entire transaction is rolled back when the `sys.error` call throws a `RuntimeException` in the enclosing top-level `atomic` block:

```
Future {
  atomic { implicit txn =>
    pop(lst, 1)
    sys.error("")
  }
} onComplete {
  case Failure(t) => log(s"oops again $t - $lst")
}
```

Unlike ScalaSTM, some other STM implementations do not roll back transactions when an exception is thrown; instead, they commit the transaction. STM experts have not yet reached a consensus on what the exception semantics should be. ScalaSTM uses a hybrid approach. Most exceptions roll back the transaction, but Scala's **control exceptions** are excluded from this rule. Control exceptions are exceptions that are used for control flow in Scala programs. They extend the `ControlThrowable` trait from the `scala.util.control` package, and are sometimes treated differently by the Scala compiler and runtime. When a control exception is thrown inside a transaction, ScalaSTM does not roll back the transaction. Instead, the transaction is committed.

Control exceptions are used to support the `break` statement in Scala, which is not a native language construct. The `break` statement throws a control exception, which is then caught by the enclosing breakable block. In the next example, we define a breakable block for the `break` statement and start a transaction that calls `pop` in a `for` loop with the values `1`, `2`, and `3`. After the first iteration, we break the loop. The example shows that the changes in the first `pop` statement are committed. The transactional list now contains only the element `16`:

```
import scala.util.control.Breaks._
Future {
  breakable {
    atomic { implicit txn =>
      for (n <- List(1, 2, 3)) {
        pop(lst, n)
        break
      }
    }
  }
  log(s"after removing - $lst")
}
```

Furthermore, it is possible to override how a specific transaction handles exceptions by calling the `withControlFlowRecognizer` method on the atomic block. This method takes a partial function from `Throwable` to `Boolean`, and uses it to decide whether a particular exception is to be considered as a control exception or not. If the partial function is not defined for particular exception, the decision is deferred to the default control flow recognizer.

In the following example, the `atomic` block overrides the default control flow recognizer. For this specific transaction, subclasses of the `ControlThrowable` trait are considered as regular exceptions. The `pop` call removes the last element of the transactional list as part of this transaction, but when we call `break`; the transaction is rolled back. The `log` statement at the end of the asynchronous computation shows that the list still contains the number `16`:

```
import scala.util.control._
Future {
  breakable {
    atomic.withControlFlowRecognizer {
      case c: ControlThrowable => false
    } { implicit txn =>
      for (n <- List(1, 2, 3)) {
        pop(lst, n)
        break
      }
    }
  }
  log(s"after removing - $lst")
}
```

Note that the exceptions thrown inside the transactions can also be intercepted using the `catch` statement. In this case, the effects of the nested transactions are aborted, and the execution proceeds from the point where the exception was caught. In the following example, we catch the exception thrown by the second `pop` call:

```
val lst = new TSortedList
lst.insert(4).insert(9).insert(1).insert(16)
atomic { implicit txn =>
  pop(lst, 2)
  log(s"lst = $lst")
  try { pop(lst, 3) }
  catch { case e: Exception => log(s"Houston... $e!") }
  pop(lst, 1)
}
log(s"result - $lst")
```

The second `pop` method call should not remove any elements from the list, so we expect to see the element `16` at the end. Running this code snippet results in the following output:

```
run-main-26: lst = 9, 16,
run-main-26: lst = 9, 16,
run-main-26: Houston... java.lang.NullPointerException!
run-main-26: result - 16,
```

Interestingly, the output reveals that the first `log` statement is invoked twice. The reason is that, when the exception is thrown the first time, both the nested and the top-level transactions are rolled back. This is an optimization in the ScalaSTM implementation, since it is more efficient to flatten the nested and the top-level transaction during the first execution attempt. Note that, after the transactional block is executed the second time, the exception from the nested transaction is correctly handled.

These examples are useful in understanding the semantics of exceptions inside the transactions. Still, the clients of our transactional sorted list want more than an exception when they call the `pop` method on an empty sorted list. In some cases, like the producer-consumer pattern from `Chapter 3`, *Traditional Building Blocks of Concurrency*, a thread has to wait and repeat the transaction when the sorted list becomes non-empty. This is called retrying, and is the topic of the next section.

Retrying transactions

In sequential computing, a single thread is responsible for executing the program. If a specific value is not available, the single thread is responsible for producing it. In concurrent programming, the situation is different. When a value is not available, some other thread, called a **producer**, might eventually produce the value. The thread consuming the value, called a **consumer**, can either block the execution until the value becomes available, or temporarily execute some other work before checking for the value again. We have seen various mechanisms for achieving this relationship, ranging from monitors and the `synchronized` statement from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*, concurrent queues from `Chapter 3`, *Traditional Building Blocks of Concurrency*; futures and promises in `Chapter 4`, *Asynchronous Programming with Futures and Promises*; to event-streams in `Chapter 6`, *Concurrent Programming with Reactive Extensions*.

Syntactically, the `atomic` statement best corresponds to the `synchronized` statement. Recall that the `synchronized` statement support the guarded block pattern, in which the thread acquires a monitor, checks for some condition, and then calls `wait` on the monitor. When some other thread fulfills this condition, it calls the `notify` method on the same monitor, indicating that the first thread should wake up and continue its work. Although sometimes fragile, this mechanism allows us to circumvent busy-waiting.

From what we have learned about STMs so far, monitors and the `notify` method have no direct counterpart in the `atomic` statement. Without them, busy-waiting is the only option when a transaction needs to wait for a specific condition to proceed. To illustrate this, let's consider the transactional sorted lists from the last section. We would like to augment the transactional sorted lists with the `headWait` method which takes a list and returns the first integer in the list if the list is non-empty. Otherwise, the execution should block until the list becomes non-empty:

```
def headWait(lst: TSortedList): Int = atomic { implicit txn =>
  while (lst.head() == null) {} // never do this
  lst.head().elem
}
```

The `headWait` method starts a transaction, and busy-waits until the `head` reference of the transactional list `lst` becomes different from `null`. To test this method, we create an empty transaction sorted list, and start an asynchronous computation that calls the `headWait` method. After one second, we start another asynchronous computation that adds the number `1` to the list. During the one-second delay, the first asynchronous computation repetitively busy-waits:

```
object RetryHeadWaitBad extends App {
  val myList = new TSortedList
  Future {
    val headElem = headWait(myList)
    log(s"The first element is $headElem")
  }
  Thread.sleep(1000)
  Future { myList.insert(1) }
  Thread.sleep(1000)
}
```

The first time we ran this example, it completed successfully after one second and reported that the first element of the list is `1`. However, this example is likely to fail. ScalaSTM will eventually detect that there is a conflict between the transaction in the `headWait` method and the transaction in the `insert` method, and will serialize the two transactions. In the case where the STM chooses the `headWait` method to execute first, number `1` is never inserted into `myList` value. Effectively, this program ends up in a deadlock. This example illustrates that busy-waiting in a transaction is just as bad as busy-waiting inside a `synchronized` statement.

TIP

Avoid long-running transactions whenever possible. Never execute an infinite loop inside a transaction, as it can cause deadlocks.

An STM is more than just support for executing isolated memory transactions. To fully replace monitors and the `synchronized` statement, an STM must provide an additional utility for transactions that block until a specific condition is fulfilled. ScalaSTM defines the `retry` statement for this purpose. When the execution inside the transaction reaches a `retry` statement, the transaction is rolled back to the enclosing top-level `atomic` block with a special exception, and the calling thread is blocked. After the rollback, the read set of the transaction is saved.

Values from the transactional references in the read set are the reason why the transaction decides to call the `retry` method. If and when some transactional reference in the read set changes its value from within another transaction, the blocked transaction can be retried.

We now reimplement the `headWait` method so that it calls the `retry` method if the `head` value of the transactional list is `null`, indicating that the list is empty:

```
def headWait(lst: TSortedList): Int = atomic { implicit txn =>
  if (lst.head() != null) lst.head().elem
  else retry
}
```

We rerun the complete program. Calling the `headWait` method is a potential blocking operation, so we need to use the `blocking` call inside the asynchronous computation. The transaction in `headWait` reads the transactional reference `head`, and puts it into the read set after calling the `retry` method. When the reference `head` later changes, the transaction is automatically retried:

```
object RetryHeadWait extends App {
  val myList = new TSortedList
  Future {
    blocking {
      log(s"The first element is ${headWait(myList)}")
    }
  }
  Thread.sleep(1000)
  Future { myList.insert(1) }
  Thread.sleep(1000)
}
```

This time, the program runs as expected. The first asynchronous computation is suspended until the second asynchronous computation adds `1` to the list. This awakens the first asynchronous computation and repeats the transaction.

TIP

Use the `retry` statement to block the transaction until a specific condition is fulfilled, and retry the transaction automatically once its read set changes.

In some cases, when a specific condition is not fulfilled and the transaction cannot proceed, we would like to retry a different transaction. Assume that there are many producer threads in the program, and a single consumer thread. To decrease contention between the producers, we decide to introduce two transactional sorted lists called `queue1` and `queue2`. To avoid creating contention by simultaneously accessing both lists, the consumer thread must check the contents of these transactional sorted lists in two separate transactions. The `orAtomic` construct allows you to do this.

The following snippet illustrates how to use `orAtomic` in this situation. We instantiate two empty transactional sorted lists: `queue1` and `queue2`. We then start an asynchronous computation that represents the consumer and starts a transaction that calls the `headWait` method on the `queue1` list. We call the `orAtomic` method after the first transaction. This specifies an alternative transaction if the first transaction calls `retry`. In the `orAtomic` block, we call the `headWait` method on the `queue2` list. When the first `atomic` block calls the `retry` method, the control is passed to the `orAtomic` block, and a different transaction starts.

Since both the transactional lists, `queue1` and `queue2`, are initially empty, the second transaction also calls the `retry` method, and the transaction chain is blocked until one of the transactional lists changes:

```
val queue1 = new TSortedList
val queue2 = new TSortedList
val consumer = Future {
  blocking {
    atomic { implicit txn =>
      log(s"probing queue1")
      log(s"got: ${headWait(queue1)}")
    } orAtomic { implicit txn =>
      log(s"probing queue2")
      log(s"got: ${headWait(queue2)}")
    }
  }
}
```

We now simulate several producers that call the `insert` method 50 milliseconds later:

```
Thread.sleep(50)
Future { queue2.insert(2) }
Thread.sleep(50)
Future { queue1.insert(1) }
Thread.sleep(2000)
```

The consumer first prints the "probing queue1" string, calls the retry method inside the headWait method, and proceeds to the next transaction. It prints the "probing queue2" string in the same way and then blocks its execution. After the first producer computation inserts 2 into the second transactional list, the consumer retries the chain of transactions again. It attempts to execute the first transaction and prints the "probing queue1" string again before finding that the queue1 list is empty. It then prints the "probing queue2" string and successfully outputs the element 2 from the queue2 list.

Retrying with timeouts

We have seen that it is useful to suspend a transaction until a specific condition gets fulfilled. In some cases, we want to prevent a transaction from being blocked forever. The wait method on the object monitors comes with an overload that takes the timeout argument. When the timeout elapses without a notify call from some other thread, an InterruptedException is thrown. The ScalaSTM withRetryTimeout method is a similar mechanism for handling timeouts.

In the following code snippet, we create a message transactional reference that initially contains an empty string. We then start an atomic block whose timeout is set to 1000 milliseconds. If the message transactional reference does not change its value within that time, the transaction fails by throwing an InterruptedException:

```
val message = Ref("")
Future {
  blocking {
    atomic.withRetryTimeout(1000) { implicit txn =>
      if (message() != "") log(s"got a message - ${message()}")
      else retry
    }
  }
}
Thread.sleep(1025)
message.single() = "Howdy!"
```

We deliberately set the timeout to 1025 milliseconds to create a race condition. This program will either print the "Howdy!" message or fail with an exception.

We use the `withRetryTimeout` method when timing out is an exceptional behavior. Shutting down the application is one example of such a behavior. We want to avoid having a blocked transaction that prevents the program from terminating. Another example is waiting for a network reply. If there is no reply after some duration of time, we want to fail the transaction.

In some cases, a timeout is a part of a normal program behavior. In this case, we wait for a specific amount of time for conditions relevant to the transaction to change. If they do, we roll back and retry the transaction, as before. If the specified amount of time elapses without any changes, the transaction should continue. In ScalaSTM, the method that does this is called `retryFor`. In the following code snippet, we rewrite the previous example using the `retryFor` method:

```
Future {
  blocking {
    atomic { implicit txn =>
      if (message() == "") {
        retryFor(1000)
        log(s"no message.")
      } else log(s"got a message - '${message()}'")
    }
  }
}
Thread.sleep(1025)
message.single() = "Howdy!"
```

This time, the transaction inside the asynchronous computation does not throw an exception. Instead, the transaction prints the `"no message."` string if a timeout occurs.

TIP

When a timeout represents exceptional program behavior, use the `withRetryTimeout` method to set the timeout duration in the transaction. When the transaction proceeds normally after a timeout, use the `retryFor` method.

The different `retry` variants are the ScalaSTM powerful additions to the standard STM model. They are as expressive as the `wait` and `notify` calls, and much safer to use. Together with the `atomic` statement, they unleash the full potential of synchronization.

Transactional collections

In this section, we take a step away from transactional references, and study more powerful transactional constructs, called, transactional collections. While transactional references can only hold a single value at once, transactional collections can manipulate multiple values. In principle, the `atomic` statements and transactional references are sufficient to express any kind of transaction over shared data. However, ScalaSTM's transactional collections are deeply integrated with the STM. They can be used to express shared data operations more conveniently and execute the transactions more efficiently.

Transaction-local variables

We have already seen that some transactions need to create a local mutable state that exists only during the execution of the transaction. Sometimes, we need to re-declare the same state over and over again for multiple transactions. In such cases, we would like to declare the same state once, and reuse it in multiple transactions. A construct that supports this in ScalaSTM is called a **transaction-local variable**.

To declare a transaction-local variable, we instantiate an object of the `TxnLocal[T]` type, giving it an initial value of type `T`. In the following code, we instantiate a `myLog` transaction-local variable. We will use `myLog` inside the transactional sorted list operations to log the flow of different transactions:

```
val myLog = TxnLocal("")
```

The value of the `myLog` transaction-local variable is seen separately by each transaction. When a transaction starts, the value of `myLog` is equal to an empty string, as specified when `myLog` was declared. When the transaction updates the value of the `myLog` variable, this change is only visible to that specific transaction. Other transactions behave as if they have their own separate copies of `myLog` variable.

We now declare a `clearList` method that atomically removes all elements from the specified transactional sorted list. This method uses the `myLog` variable to log the elements that were removed:

```
def clearList(lst: TSortedList): Unit = atomic { implicit txn =>
  while (lst.head() != null) {
    myLog() = myLog() + "\nremoved " + lst.head().elem
    lst.head() = lst.head().next()
  }
}
```

Usually, we are not interested in the contents of the `myLog` variable. However, we might occasionally want to inspect the `myLog` variable for debugging purposes. Hence, we declare the `clearWithLog` method that clears the list and then returns the contents of `myLog`. We then call the `clearWithLog` method on a non-empty transactional list from two separate asynchronous computations. After both asynchronous computations complete execution, we output their logs:

```
val myList = new TSortedList().insert(14).insert(22)
def clearWithLog(): String = atomic { implicit txn =>
  clearList(myList)
  myLog()
}
val f = Future { clearWithLog() }
val g = Future { clearWithLog() }
for (h1 <- f; h2 <- g) log(s"Log for f: $h1\nLog for g: $h2")
```

Since the `clearList` operation is atomic, only one of the transactions can remove all the elements. The contents of the `myLog` object reflect this. Depending on the timing between the asynchronous computations, elements `14` and `22` both appear either in the log of the `f` future or in the log of the `g` future. This shows that each of the two transactions sees a separate duplicate of the `myLog` variable.

TIP

Transaction-local variables are syntactically more lightweight than creating transactional references and passing them between different methods.

Transaction-local variables are used while logging or gathering statistics on the execution of the program. The `TxnLocal` constructor additionally allows you to specify the `afterCommit` and `afterRollback` callbacks, invoked on the transaction-local variable when the transaction commits or rolls back, respectively. We refer the reader to the online documentation to find out how to use them. To build more complex concurrent data models, we use transactional arrays and maps, which we will study in the next section.

Transactional arrays

Transactional references are a handy way to encapsulate a transactional state, but they come with certain overheads. First, a `Ref` object is more heavyweight than a simple object reference and consumes more memory. Second, every access to a new `Ref` object needs to add an entry in the transaction's read set. When we are dealing with many `Ref` objects, these overheads can become substantial. Let's illustrate this with an example.

Assume that we are working in the marketing department of a company that does Scala consulting. We are asked to write a program that updates the content of the company website with the marketing information about the Scala 2.10 release. Naturally, we decide to use ScalaSTM for this task. The website consists of five separate pages, each represented with a string. We declare the contents of the website in a sequence called `pages`. We then assign the content of the pages to an array of transactional references. If some page changes later, we can update its transactional reference in a transaction:

```
val pages: Seq[String] = Seq.fill(5)("Scala 2.10 is out, " * 7)
val website: Array[Ref[String]] = pages.map(Ref(_)).toArray
```

This solution is not satisfactory. We created a lot of transactional reference objects, and the definition of `website` is not easily understandable. Luckily, ScalaSTM has an alternative called a **transactional array**. A transactional array, represented with the `TArray` class, is similar to an ordinary Scala array, but can be accessed only from within a transaction. Its modifications are only made visible to the other threads when a transaction commits. Semantically, a `TArray` class corresponds to an array of transactional references, but it is more memory-efficient and concise:

```
val pages: Seq[String] = Seq.fill(5)("Scala 2.10 is out, " * 7)
val website: TArray[String] = TArray(pages)
```

Scala development proceeds at an amazing pace. Not long after Scala 2.10 was announced, the 2.11 release of Scala became available. The marketing team asks us to update the contents of the website. All occurrences of the `"2.10"` string should be replaced with the `"2.11"` string. We write a `replace` method that does this:

```
def replace(p: String, s: String): Unit = atomic { implicit txn =>
  for (i <- 0 until website.length)
    website(i) = website(i).replace(p, s)
}
```

Using the `TArray` class is much nicer than storing transactional references in an array. Not only does it spare us from a parenthesis soup resulting from calling the `apply` operation on the transactional references in the array, but it also occupies less memory. This is because a single contiguous array object is created for the `TArray[T]` object, whereas an `Array[Ref[T]]` object requires many `Ref` objects, each of which has a memory overhead.

TIP

Use the `TArray` class instead of arrays of transactional references to optimize memory usage and make programs more concise.

Let's test the `TArray` class and the `replace` method in a short program. We first define an additional method, `asString`, which concatenates the contents of all the website pages. We then replace all occurrences of the `2.10` string with the `2.11` string. To test whether `replace` works correctly, we concurrently replace all occurrences of the `out` word with `"released"`:

```
def asString = atomic { implicit txn =>
  var s: String = ""
  for (i <- 0 until website.length)
    s += s"Page $i\n======\n${website(i)}\n\n"
  s
}
val f = Future { replace("2.10", "2.11") }
val g = Future { replace("out", "released") }
for (_ <- f; _ <- g) log(s"Document\n$asString")
```

The `asString` method captured all the entries in the transactional array. In effect, the `asString` method atomically produced a snapshot of the state of the `TArray` object. Alternatively, we could have copied the contents of `website` into another `TArray` object, instead of a string. In either case, computing the snapshot of a `TArray` object requires traversing all its entries, and can conflict with the transactions that modify only a subset of the `TArray` class.

Recall the transactional conflict example from the beginning of this chapter. A transaction with many reads and writes, as in the `asString` method, can be inefficient, because all the other transactions need to serialize with the `asString` method when a conflict occurs. When the array is large, this creates a scalability bottleneck. In the next section, we will examine another collection capable of producing atomic snapshots in a much more scalable manner, namely, the transactional maps.

Transactional maps

Similar to transactional arrays, transactional maps avoid the need to store transactional reference objects inside a map. As a consequence, they reduce memory consumption, improve the transaction performance, and provide a more intuitive syntax. In ScalaSTM, transactional maps are represented with the `TMap` class.

ScalaSTM's `TMap` class has an additional advantage. It exposes a scalable, constant-time, atomic `snapshot` operation. The `snapshot` operation returns an immutable `Map` object with the contents of the `TMap` object at the time of the snapshot. Let's declare a transactional map, `alphabet`, which maps character strings to their position in the alphabet:

```
val alphabet = TMap("a" -> 1, "B" -> 2, "C" -> 3)
```

We are unsatisfied with the fact that the letter `A` is in lowercase. We start a transaction that atomically replaces the lowercase letter `a` with the uppercase letter `A`. Simultaneously, we start another asynchronous computation that calls the `snapshot` operation on the `alphabet` map. We tune the timing of the second asynchronous computation so that it creates a race condition with the first transaction:

```
Future {
  atomic { implicit txn =>
    alphabet("A") = 1
    alphabet.remove("a")
  }
}
Thread.sleep(23)
Future {
  val snap = alphabet.single.snapshot
  log(s"atomic snapshot: $snap")
}
Thread.sleep(2000)
```

In this example, the `snapshot` operation cannot interleave with the two updates in the `atomic` block. We can run the program several times to convince ourselves of this. The second asynchronous computation prints either the map with the lowercase letter `a`, or the map with the uppercase letter `A`, but it can never output a map with both the lowercase and the uppercase occurrence of the letter `A`.

TIP

Use `TMap` (instead of maps of transactional references) to optimize memory usage, make programs more concise, and efficiently retrieve atomic snapshots.

Summary

In this chapter, we learned how STM works and how to apply it in concurrent programs. We saw the advantages of using STM's transactional references and `atomic` blocks over the `synchronized` statements, and investigated their interaction with side effects. We studied the semantics of exception handling inside transactions and learned how to retry and conditionally re-execute transactions. Finally, we learned about transactional collections, which allow us to encode shared program data more efficiently.

These features together enable a concurrent programming model in which the programmer can focus on expressing the meaning of the program, without having to worry about handling lock objects, or avoiding deadlocks and race conditions. This is especially important when it comes to modularity. It is hard or near impossible to reason about deadlocks or race conditions in the presence of separate software components. STM exists to liberate the programmer from such concerns, and is essential when composing large concurrent programs from simpler modules.

These advantages come with a cost, however, as using an STM for data access is slower than using locks and the `synchronized` statement. For many applications, the performance penalty of using an STM is acceptable. When it is not, we need to revert to simpler primitives, such as locks, atomic variables, and concurrent data structures.

To learn more about STMs, we recommend reading the related chapter in the book *The Art of Multiprocessor Programming*, *Maurice Herlihy and Nir Shavit*, *Morgan Kauffman*. There are many different STM implementations in the wild, and you will need to study various research articles to obtain an in-depth understanding of STMs. An extensive list of STM research literature is available at `http://research.cs.wisc.edu/trans-memory/biblio/index.html`. To learn more about the specifics of ScalaSTM, consider reading the doctoral thesis entitled *Composable Operations on High-Performance Concurrent Collections*, *Nathan G. Bronson*.

In the next chapter, we will study the actor programming model, which takes a different approach to achieving memory consistency. As we will see, separate computations never access each other's regions of memory in the actor model, and communicate mainly by exchanging messages.

Exercises

In the following exercises, you will use ScalaSTM to implement various transactional programming abstractions. In most cases, their implementation will closely resemble a sequential implementation, while using transactions. In some cases, you might need to consult external literature or ScalaSTM documentation to correctly solve the exercise.

1. Implement the transactional pair abstraction, represented with the `TPair` class:

```
class TPair[P, Q](pinit: P, qinit: Q) {
  def first(implicit txn: InTxn): P = ???
  def first_=(x: P)(implicit txn: InTxn): P = ???
  def second(implicit txn: InTxn): Q = ???
  def second_=(x: Q)(implicit txn: InTxn): Q = ???
  def swap()(implicit e: P =:= Q, txn: InTxn): Unit = ???
}
```

   In addition to getters and setters for the two fields, the transactional pair defines the `swap` method that swaps the fields, and can only be called if types `P` and `Q` are the same.

2. Use ScalaSTM to implement the mutable location abstraction from Haskell, represented with the `MVar` class:

```
class MVar[T] {
  def put(x: T)(implicit txn: InTxn): Unit = ???
  def take()(implicit txn: InTxn): T = ???
}
```

   An `MVar` object can be either full or empty. Calling `put` on a full `MVar` object blocks until the `MVar` object becomes empty, and adds an element. Similarly, calling `take` on an empty `MVar` object blocks until the `MVar` object becomes full, and removes the element. Now, implement a method called `swap`, which takes two `MVar` objects and swaps their values:

```
def swap[T](a: MVar[T], b: MVar[T])(implicit txn: InTxn)
```

   Contrast the `MVar` class with the `SyncVar` class from `Chapter 2`, *Concurrency on the JVM and the Java Memory Model*. Is it possible to implement the `swap` method for `SyncVar` objects without modifying the internal implementation of the SyncVar class?

3. Implement the `atomicRollbackCount` method, which is used to track how many times a transaction was rolled back before it completed successfully:

```
def atomicRollbackCount[T](block: InTxn => T): (T, Int)
```

4. Implement the `atomicWithRetryMax` method, which is used to start a transaction that can be retried at most `n` times:

```
def atomicWithRetryMax[T](n: Int)(block: InTxn => T): T
```

   Reaching the maximum number of retries throws an exception.

   TIP: Use the `Txn` object.

5. Implement a transactional **First In First Out** (**FIFO**) queue, represented with the `TQueue` class:

```
class TQueue[T] {
  def enqueue(x: T)(implicit txn: InTxn): Unit = ???
  def dequeue()(implicit txn: InTxn): T = ???
```

   The `TQueue` class has similar semantics as `scala.collection.mutable.Queue`, but calling `dequeue` on an empty queue blocks until a value becomes available.

6. Use ScalaSTM to implement a thread-safe `TArrayBuffer` class, which extends the `scala.collection.mutable.Buffer` interface.
7. The `TSortedList` class described in this chapter is always sorted, but accessing the last element requires traversing the entire list, and can be slow. An AVL tree can be used to address this problem. There are numerous descriptions of AVL trees available online. Use ScalaSTM to implement the thread-safe transactional sorted set as an AVL tree:

```
class TSortedSet[T] {
  def add(x: T)(implicit txn: InTxn): Unit = ???
  def remove(x: T)(implicit txn: InTxn): Boolean = ???
  def apply(x: T)(implicit txn: InTxn): Boolean = ???
}
```

   The `TSortedSet` class has similar semantics as `scala.collection.mutable.Set`.

8. Use ScalaSTM to implement a banking system that tracks amounts of money on user accounts. Different threads can call the `send` method to transfer money from one account to another, the `deposit` and `withdraw` methods which deposit to or withdraw money from a specific account, respectively, and the `totalStock` method which returns the total amount of money currently deposited in the bank. Finally, implement the `totalStockIn` method that returns the total amount of money currently deposited in the specified set of banks.

9. Implement the generic transactional priority queue class, represented with the type `TPriorityQueue`, used to sort elements. Then implement a method called `scheduleTask`, which adds a task to the priority queue. Each task has a priority level. A set of workers must wait for the queue to become non-empty, at which point they repetitively remove tasks with the highest priority, and execute them.
10. Implement a generic transactional directed graph data structure, whose nodes are represented with the `Node` class. Then implement a method `scheduleTask`, which adds a task to into the graph. Each task has the list of dependencies – other tasks in the graph that must be executed before it begins; and this list represents the directed edges in the graph. A set of workers repetitively queries the graph, and schedules tasks for execution. A task can only be executed after its dependencies are done executing.

8

Actors

> *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."*
>
> *– Leslie Lamport*

Throughout this book, we have concentrated on many different abstractions for concurrent programming. Most of these abstractions assume the presence of shared memory. Futures and promises, concurrent data structures, and software transactional memory, are best suited to shared-memory systems. While the shared-memory assumption ensures that these facilities are efficient, it also limits them to applications running on a single computer. In this chapter, we consider a programming model that is equally applicable to a shared-memory machine or a distributed system, namely, the **actor model**. In the actor model, the program is represented by a large number of entities that execute computations independently, and communicate by passing messages. These independent entities are called **actors**.

The actor model aims to resolve issues associated with using shared memory, such as data races or synchronization, by eliminating the need for shared memory altogether. *Mutable* state is confined within the boundaries of one actor, and is potentially modified when the actor receives a message. Messages received by the actor are handled serially, one after another. This ensures that the mutable state within the actor is never accessed concurrently. However, separate actors can process the received messages concurrently. In a typical actor-based program, the number of actors can be orders of magnitude greater than the number of processors. This is similar to the relationship between processors and threads in multi-threaded programs. The actor model implementation decides when to assign processor time to specific actors, to allow them to process messages.

The true advantage of the actor model becomes apparent when we start distributing the application across multiple computers. Implementing programs that span across multiple machines and devices that communicate through a computer network is called **distributed programming**. The actor model allows you to write programs that run inside a single process, multiple processes on the same machine, or on multiple machines that are connected to a computer network. Creating actors and sending messages is oblivious to, and independent of, the location of the actor. In distributed programming, this is called **location transparency**. Location transparency allows you to design distributed systems without having the knowledge about the relationships in the computer network.

In this chapter, we will use the Akka `actor` framework to learn about the actor concurrency model. Specifically, we cover the following topics:

- Declaring actor classes and creating actor instances
- Modeling actor state and complex actor behaviors
- Manipulating the actor hierarchy and the lifecycle of an actor
- The different message-passing patterns used in actor communication
- Error recovery using the built-in actor supervision mechanism
- Using actors to transparently build concurrent and distributed programs

We will start by studying the important concepts and terminology in the actor model, and learning the basics of the actor model in Akka.

Working with actors

In the actor programming model, the program is run by a set of concurrently executing entities called actors. Actor systems resemble human organizations, such as companies, governments, or other large institutions. To understand this similarity, we consider the example of a large software company.

In a software company such as Google, Microsoft, Amazon, or Typesafe, there are many goals that need to be achieved concurrently. Hundreds or thousands of employees work toward achieving these goals, and are usually organized in a hierarchical structure. Different employees work at different positions. A team leader makes important technical decisions for a specific project, a software engineer implements and maintains various parts of a software product, and a system administrator makes sure that the personal workstations, servers, and various equipment are functioning correctly. Many employees, such as the team leader, delegate their own tasks to other employees who are lower in the hierarchy than themselves. To be able to work and make decisions efficiently, employees use e-mails to communicate.

When an employee comes to work in the morning, he inspects his e-mail client and responds to the important messages. Sometimes, these messages contain work tasks that come from his boss, or requests from other employees. When an e-mail is important, the employee must compose the answer right away. While the employee is busy answering one e-mail, additional e-mails can arrive, and these e-mails are enqueued in his e-mail client. Only once the employee is done with one e-mail is he able to proceed to the next one.

In the preceding scenario, the workflow of the company is divided into a number of functional components. It turns out that these components closely correspond to different parts of an actor framework. We will now identify these similarities by defining the parts of an actor system, and relating them to their analogs in the software company.

An **actor system** is a hierarchical group of actors that share common configuration options. An actor system is responsible for creating new actors, locating actors within the actor system, and logging important events. An actor system is an analog of the software company itself.

An **actor class** is a template that describes a state internal to the actor, and how the actor processes the messages. Multiple actors can be created from the same actor class. An actor class is an analogy for a specific position within the company, such as a software engineer, a marketing manager, or a recruiter.

An **actor instance** is an entity that exists at runtime and is capable of receiving messages. An actor instance might contain mutable state, and can send messages to other actor instances. The difference between an actor class and an actor instance directly corresponds to the relationship between a class and an object instance of that class in object-oriented programming. In the context of the software company example, an actor instance is analogous to a specific employee.

A **message** is a unit of communication that actors use to communicate. In Akka, any object can be a message. Messages are analogous to e-mails sent within the company. When an actor sends a message, it does not wait until some other actor receives the message. Similarly, when an employee sends an e-mail, he does not wait until the e-mail is received or read by the other employees. Instead, he proceeds with his own work; an employee is too busy to wait. Multiple e-mails might be sent to the same person concurrently.

The **mailbox** is a part of memory that is used to buffer messages, specific to each actor instance. This buffer is necessary, as an actor instance can process only a single message at a time. The mailbox corresponds to an e-mail client used by an employee. At any point, there might be multiple unread e-mails buffered in the e-mail client, but the employee can only read and respond to them one at a time.

An **actor reference** is an object that allows you to send messages to a specific actor. This object hides information about the location of the actor from the programmer. An actor might run within separate processes or on different computers. The actor reference allows you to send a message to an actor irrespective of where the actor is running. From the software-company perspective, an actor reference corresponds to the e-mail address of a specific employee. The e-mail address allows us to send an e-mail to an employee, without knowing anything about the physical location of the employee. The employee might be in his office, on a business trip, or on vacation, but the e-mail will eventually reach him no matter where he goes.

A **dispatcher** is a component that decides when actors are allowed to process messages, and lends them computational resources to do so. In Akka, every dispatcher is, at the same time, an execution context. The dispatcher ensures that actors with non-empty mailboxes eventually get run by a specific thread, and that these messages are handled serially. A dispatcher is best compared to the e-mail answering policy in the software company. Some employees, such as the technical support specialists, are expected to answer e-mails as soon as they arrive. Software engineers sometimes have more liberty-they can choose to fix several bugs before inspecting their e-mails. The janitor spends his day working around the office building, and only takes a look at his e-mail client in the morning.

To make these concepts more concrete, we start by creating a simple actor application. This is the topic of the following section, in which we learn how to create actor systems and actor instances.

Creating actor systems and actors

When creating an object instance in an object-oriented language, we start by declaring a class, which can be reused by multiple object instances. We then specify arguments for the constructor of the object. Finally, we instantiate an object using the `new` keyword and obtain a reference to the object.

Creating an actor instance in Akka roughly follows the same steps as creating an object instance. First, we need to define an actor class, which defines the behavior of the actor. Then, we need to specify the configuration for a specific actor instance. Finally, we need to tell the actor system to instantiate the actor using the given configuration. The actor system then creates an actor instance and returns an actor reference to that instance. In this section, we will study these steps in more detail.

An actor class is used to specify the behavior of an actor. It describes how the actor responds to messages and communicates with other actors, encapsulates actor state, and defines the actor's startup and shutdown sequences. We declare a new actor class by extending the `Actor` trait from the `akka.actor` package. This trait comes with a single abstract method, `receive`. The `receive` method returns a partial function object of the type `PartialFunction[Any, Unit]`. This partial function is used when an actor receives a message of the `Any` type. If the partial function is not defined for the message, the message is discarded.

In addition to defining how an actor receives messages, the actor class encapsulates references to objects used by the actor. These objects comprise the actor's state. Throughout this chapter, we use Akka's `Logging` object to print to the standard output. In the following code, we declare a `HelloActor` actor class, which reacts to a `hello` message specified with the `hello` constructor argument. The `HelloActor` class contains a `Logging` object, `log`, as part of its state. The `Logging` object is created using the `context.system` reference to the current actor system, and the `this` reference to the current actor. The `HelloActor` class defines a partial function in the `receive` method, which determines if the message is equal to the `hello` string argument, or to some other object called `msg`.

When an actor defined by the `HelloActor` class receives a `hello` string message, it prints the message using the `Logging` object `log`. Otherwise, it prints that it received an unexpected message, and stops by calling the `context.stop` method on the actor reference `self`, which represents the current actor. This is shown in the following code snippet:

```
import akka.actor._
import akka.event.Logging
class HelloActor(val hello: String) extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case `hello` =>
      log.info(s"Received a '$hello'... $hello!")
    case msg     =>
      log.info(s"Unexpected message '$msg'")
      context.stop(self)
  }
}
```

Declaring an actor class does not create a running actor instance. Instead, the actor class serves as a blueprint for creating actor instances. The same actor class can be shared by many actor instances. To create an actor instance in Akka, we need to pass information about the actor class to the actor system. However, an actor class such as `HelloActor` is not sufficient for creating an actor instance; we also need to specify the `hello` argument. To bundle the information required for creating an actor instance, Akka uses objects called **actor configurations**.

An actor configuration contains information about the actor class, its constructor arguments, mailbox, and dispatcher implementation. In Akka, an actor configuration is represented with the `Props` class. A `Props` object encapsulates all the information required to create an actor instance, and can be serialized or sent over the network.

To create `Props` objects, it is recommended practice to declare Factory methods in the companion object of the actor class. In the following companion object, we declare two Factory methods, called `props` and `propsAlt`, which return `Props` objects for the `HelloActor` class, given the `hello` argument:

```
object HelloActor {
  def props(hello: String) = Props(new HelloActor(hello))
  def propsAlt(hello: String) = Props(classOf[HelloActor], hello)
}
```

The `props` method uses an overload of the `Props.apply` factory method, which takes a block of code by creating the `HelloActor` class. This block of code is invoked every time an actor system needs to create an actor instance. The `propsAlt` method uses another `Props.apply` overload, which creates an actor instance from the `Class` object of the actor class, and a list of constructor arguments. The two declarations are semantically equivalent.

The first `Props.apply` method overload takes a closure that calls the actor class constructor. If we are not careful, the closure can easily catch references to the enclosing scope. When this happens, these references become a part of the `Props` object. Consider the `defaultProps` method in the following utility class:

```
class HelloActorUtils {
  val defaultHi = "Aloha!"
  def defaultProps() = Props(new HelloActor(defaultHi))
}
```

Sending the `Props` object that is returned by the `defaultProps` method over the network requires sending the enclosing `HelloActorUtils` object captured by the closure, incurring additional network costs.

Furthermore, it is particularly dangerous to declare a `Props` object within an actor class, as it can catch a `this` reference to the enclosing actor instance. It is safer to create the `Props` objects exactly as they were shown in the `propsAlt` method.

> TIP
> Avoid creating the `Props` objects within actor classes to prevent accidentally capturing the actor's `this` reference. Wherever possible, declare `Props` inside factory methods in top-level singleton objects.

The third overload of the `Props.apply` method is a convenience method that can be used with actor classes with zero-argument constructors. If `HelloActor` defines no constructor arguments, we can write `Props[HelloActor]` to create a `Props` object.

To instantiate an actor, we pass an actor configuration to the `actorOf` method of the actor system. Throughout this chapter, we will use our custom actor system instance called `ourSystem`. We define the `ourSystem` variable using the `ActorSystem.apply` factory method:

```
lazy val ourSystem = ActorSystem("OurExampleSystem")
```

We can now create and run the `HelloActor` class by calling the `actorOf` method on the actor system. When creating a new actor, we can specify a unique name for the actor instance with the `name` argument. Without explicitly specifying the `name` argument, the actor system automatically assigns a unique name to the new actor instance. The `actorOf` method does not return an instance of the `HelloActor` class. Instead, it returns an actor reference object of the `ActorRef` type.

After creating a `HelloActor` instance `hiActor`, which recognizes the `hi` messages, we send it a message, `hi`. To send a message to an Akka actor, we use the `!` operator (pronounced as *tell* or *bang*). For clarity, we then pause the execution for one second by calling `sleep`, and give the actor some time to process the message. We then send another message, `hola`, and wait one more second. Finally, we terminate the actor system by calling its `shutdown` method. This is shown in the following program:

```
object ActorsCreate extends App {
  val hiActor: ActorRef =
    ourSystem.actorOf(HelloActor.props("hi"), name = "greeter")
  hiActor ! "hi"
  Thread.sleep(1000)
  hiActor ! "hola"
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

Upon running this program, the `hiActor` instance first prints that it received a `hi` message. After one second, it prints that it received a `hola` string as a message, an unexpected message, and terminates.

Managing unhandled messages

The `receive` method in the `HelloActor` example was able to handle any kind of message. When the message was different from the pre-specified `hello` argument, such as `hi`, used previously, the `HelloActor` actor reported this in the default case. Alternatively, we could have left the default case unhandled. When an actor receives a message that is not handled by its `receive` method, the message is wrapped into an `UnhandledMessage` object and forwarded to the actor system's event stream. Usually, the actor system's event stream is used for logging purposes.

We can override this default behavior by overriding the `unhandled` method in the actor class. By default, this method publishes the unhandled messages on the actor system's event stream. In the following code, we declare a `DeafActor` actor class, whose `receive` method returns an empty partial function. An empty partial function is not defined for any type of message, so all the messages sent to this actor get passed to the `unhandled` method. We override it to output the `String` messages to the standard output. We pass all other types of message to the actor system's event stream by calling the `super.unhandled` method. The following code snippet shows the `DeafActor` implementation:

```
class DeafActor extends Actor {
  val log = Logging(context.system, this)
  def receive = PartialFunction.empty
  override def unhandled(msg: Any) = msg match {
    case msg: String => log.info(s"I do not hear '$msg'")
    case msg         => super.unhandled(msg)
  }
}
```

Let's test a `DeafActor` class in an example. The following program creates a `DeafActor` instance named `deafy`, and assigns its actor reference to the value `deafActor`. It then sends the two messages, `deafy` and `1234`, to `deafActor`, and shuts down the actor system:

```
object ActorsUnhandled extends App {
  val deafActor: ActorRef =
    ourSystem.actorOf(Props[DeafActor], name = "deafy")
  deafActor ! "hi"
  Thread.sleep(1000)
  deafActor ! 1234
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

Running this program shows that the first message, the `deafy` string, is caught and printed by the `unhandled` method. The `1234` message is forwarded to the actor system's event stream, and is never shown on the standard output.

An attentive reader might have noticed that we could have avoided the `unhandled` call by moving the case into the `receive` method, as shown in the following `receive` implementation:

```
def receive = {
  case msg: String => log.info(s"I do not hear '$msg'")
}
```

This definition of the `receive` method is more concise, but is inadequate for more complex actors. In the preceding example, we have fused the treatment of unhandled messages together with how the actor handles regular messages. Stateful actors often change the way they handle regular messages, and it is essential to separate the treatment of unhandled messages from the normal behavior of the actor. We will study how to change the actor behavior in the following section.

Actor behavior and state

When an actor changes its state, it is often necessary to change the way it handles incoming messages. The way that the actor handles regular messages is called the **behavior** of the actor. In this section, we will study how to manipulate actor behavior.

We have previously learned that we define the initial behavior of the actor by implementing the `receive` method. Note that the `receive` method must always return the same partial function. It is not correct to return different partial functions from the `receive` method depending on the current state of the actor. Let's assume we want to define a `CountdownActor` actor class, which decreases its `n` integer field every time it receives a `count` message, until it reaches zero. After the `CountdownActor` class reaches zero, it should ignore all subsequent messages. The following definition of the `receive` method is not allowed in Akka:

```
class CountdownActor extends Actor {
  var n = 10
  def receive = if (n > 0) { // never do this
    case "count" =>
      log(s"n = $n")
      n -= 1
  } else PartialFunction.empty
}
```

To correctly change the behavior of the `CountdownActor` class after it reaches zero, we use the `become` method on the actor's `context` object. In the correct definition of the `CountdownActor` class, we define two methods, `counting` and `done`, which return two different behaviors. The `counting` behavior reacts to the `count` messages and calls `become` to change to the `done` behavior once the `n` field is zero. The `done` behavior is just an empty partial function, which ignores all the messages.

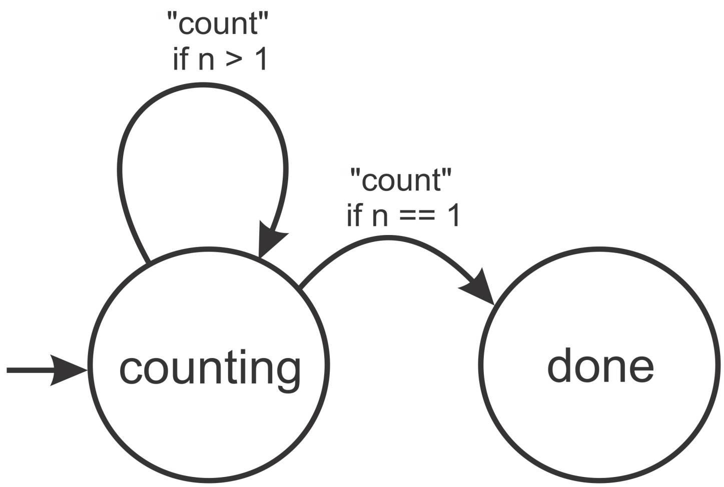
This is shown in the following implementation of the `CountdownActor` class:

```
class CountdownActor extends Actor {
  val log = Logging(context.system, this)
  var n = 10
  def counting: Actor.Receive = {
    case "count" =>
      n -= 1
      log.info(s"n = $n")
      if (n == 0) context.become(done)
  }
  def done = PartialFunction.empty
  def receive = counting
}
```

The `receive` method defines the initial behavior of the actor, which must be the `counting` behavior. Note that we are using the type alias `Receive` from the Actor companion object, which is just a shorthand for the `PartialFunction[Any, Unit] type`.

When modeling complex actors, it is helpful to think of them as **state machines**. A state machine is a mathematical model that represents a system with some number of states and transitions between these states. In an actor, each behavior corresponds to a state in the state machine. A transition exists between two states if the actor potentially calls the `become` method when receiving a certain message. In the following figure, we illustrate the state machine corresponding to the `CountdownActor` class. The two circles represent the states corresponding to the behaviors `counting` and `done`. The initial behavior is **counting**, so we draw an arrow pointing to the corresponding state. We represent the transitions between the states with arrows starting and ending at a state.

When the actor receives the **count** message and the **n** field is larger than **1**, the behavior does not change. However, when the actor receives the **count** message and the **n** field is decreased to `0`, the actor changes its behavior to **done**:



The following short program tests the correctness of our actor. We use the actor system to create a new `countdown` actor, and send it 20 `count` messages. The actor only reacts to the first 10 messages, before switching to the `done` behavior:

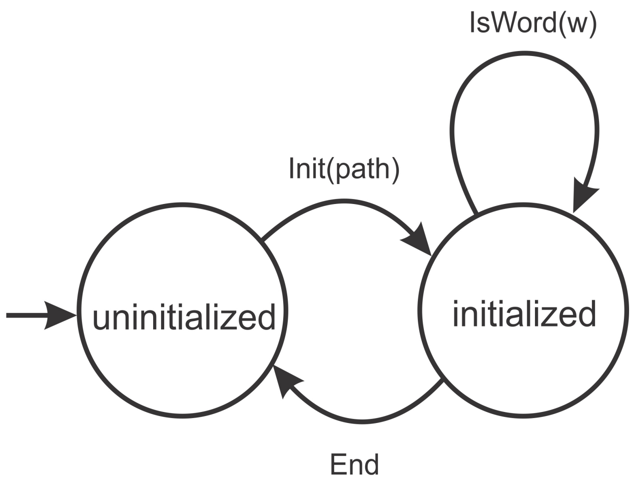
```
object ActorsCountdown extends App {
  val countdown = ourSystem.actorOf(Props[CountdownActor])
  for (i <- 0 until 20) countdown ! "count"
  Thread.sleep(1000)
  ourSystem.shutdown()
}
```

Whenever an actor responds to the incoming messages differently depending on its current state, you should decompose different states into partial functions and use the `become` method to switch between states. This is particularly important when actors get more complex, and ensures that the actor logic is easier to understand and maintain.

> TIP: When a stateful actor needs to change its behavior, declare a separate partial function for each of its behaviors. Implement the `receive` method to return the method corresponding to the initial behavior.

We now consider a more refined example, in which we define an actor that checks if a given word exists in a dictionary and prints it to the standard output. We want to be able to change the dictionary that the actor is using during runtime. To set the dictionary, we send the actor an `Init` message with the path to the dictionary. After that, we can check if a word is in the dictionary by sending the actor the `IsWord` message. Once we're done using the dictionary, we can ask the actor to unload the dictionary by sending it the `End` message. After that, we can initialize the actor with some other dictionary.

The following state machine models this logic with two behaviors, called `uninitialized` and `initialized`:



It is a recommended practice to define the datatypes for the different messages in the companion object of the actor class. In this case, we add the case classes `Init`, `IsWord`, and `End` to the companion object of the `DictionaryActor` class:

```
object DictionaryActor {
  case class Init(path: String)
  case class IsWord(w: String)
  case object End
}
```

We next define the `DictionaryActor` actor class. This class defines a private `Logging` object `log`, and a `dictionary` mutable set, which is initially empty and can be used to store words. The `receive` method returns the `uninitialized` behavior, which only accepts the `Init` message type. When an `Init` message arrives, the actor uses its `path` field to fetch the dictionary from a file, load the words, and call `become` to switch to the `initialized` behavior. When an `IsWord` message arrives, the actor checks if the word exists and prints it to the standard output. If an `End` message arrives, the actor clears the dictionary and switches back to the `uninitialized` behavior. This is shown in the following code snippet:

```
class DictionaryActor extends Actor {
  private val log = Logging(context.system, this)
  private val dictionary = mutable.Set[String]()
  def receive = uninitialized
  def uninitialized: PartialFunction[Any, Unit] = {
    case DictionaryActor.Init(path) =>
      val stream = getClass.getResourceAsStream(path)
      val words = Source.fromInputStream(stream)
      for (w <- words.getLines) dictionary += w
      context.become(initialized)
  }
  def initialized: PartialFunction[Any, Unit] = {
    case DictionaryActor.IsWord(w) =>
      log.info(s"word '$w' exists: ${dictionary(w)}")
    case DictionaryActor.End =>
      dictionary.clear()
      context.become(uninitialized)
  }
  override def unhandled(msg: Any) = {
    log.info(s"cannot handle message $msg in this state.")
  }
}
```

Note that we have overridden the `unhandled` method in the `DictionaryActor` class. In this case, using the `unhandled` method reduces code duplication, and makes the `DictionaryActor` class easier to maintain, as there is no need to list the `default` case twice in both the `initialized` and `uninitialized` behaviors.