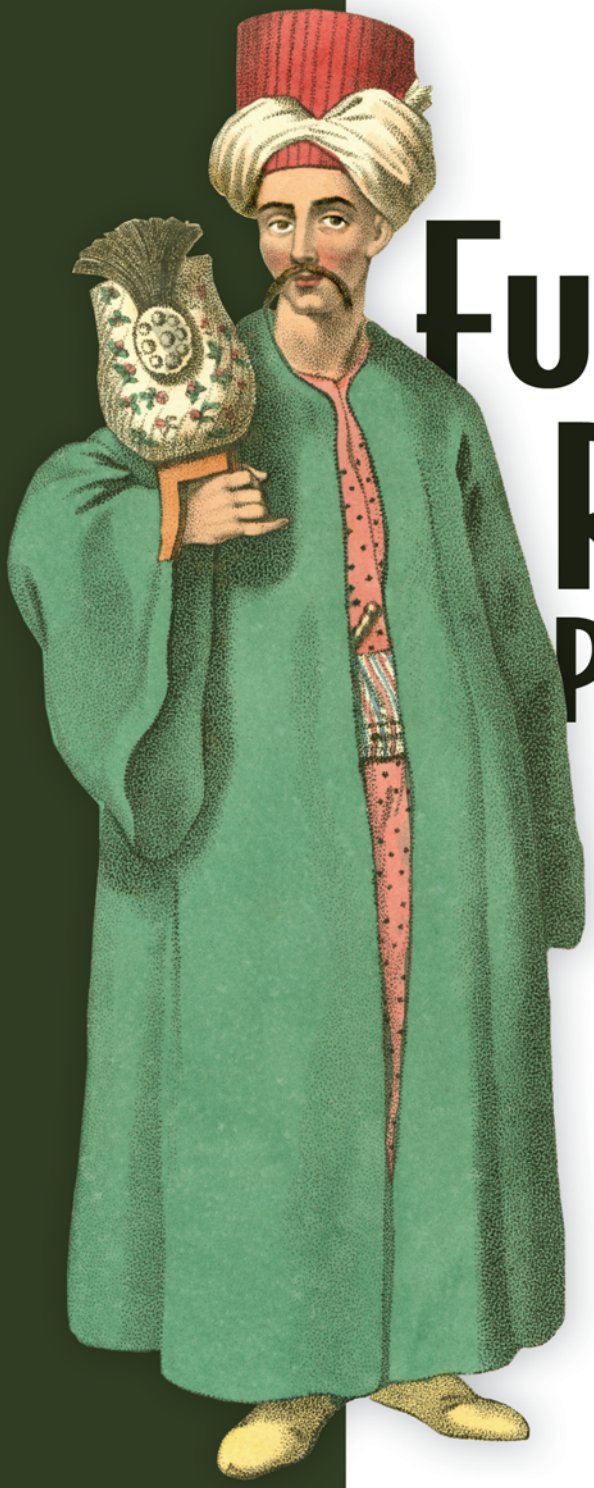
SAMPLE CHAPTER



Functional Reactive Programming

Stephen Blackheath
Anthony Jones

FOREWORD BY Heinrich Apfelmus

MANNING

Functional Reactive Programming

by Stephen Blackheath, Anthony Jones

Sample Chapter 1

brief contents

1 Stop listening!

This chapter covers

- What FRP is
- What events are, and how they cause trouble
- What FRP is for: the problem we're trying to solve
- The benefits of FRP
- How an FRP system works
- A different way of thinking that underlies FRP

Welcome to our book! We love *functional reactive programming* (FRP). Many people like the idea too, yet they aren't entirely clear what FRP is and what it will do for them. The short answer: it comes in the form of a simple library in a standard programming language, and it replaces listeners (also known as *callbacks*) in the widely used *observer pattern*, making your code cleaner, clearer, more robust, and more maintainable—in a word, *simpler*.

It's more than this: FRP is a very different way of doing things. It will improve your code and transform your thinking for the better. Yet it's surprisingly compatible with the usual ways of writing code, so it's easy to factor into existing projects in stages. This book is about the concepts of FRP as they apply to a range of FRP systems and programming languages.

FRP is based on ideas from functional programming, but this book doesn't assume any prior knowledge of functional programming. Chapter 1 will lay down some underlying concepts, and in chapter 2 we'll get into the coding. So stop listening, and start reacting!

1.1 Project, meet complexity wall

It seemed to be going so well. The features weren't all there yet, but development was swift. The boss was happy, the customers were impressed, the investors were optimistic. The future was bright.

It came out of nowhere … Software quality crumbled. The speed of development went from treacle to molasses. Before long, there were unhappy customers and late nights. What happened?

Sooner or later, many big projects hit the complexity wall. The complexities in the program that seemed acceptable compound exponentially: At first you hardly notice, and then—BAM! It hits broadside. The project will then typically go one of four ways:

- It's shelved.
- It's rewritten from scratch, and a million dollars later, it hits the same wall again.
- The company staffs up. As the team expands, its productivity shambles off into the realm of the eternal quagmire. (Often the company has been acquired around this time.)
- It undergoes major refactoring, leading eventually to maintainable code.

Refactoring is the only way forward. It's your primary tool to save a project that has hit the wall, but it's best used earlier, as part of a development methodology, to prevent disaster before it happens.

But this book isn't about refactoring. It's about *functional reactive programming* (FRP), a programming style that works well with refactoring because it can prevent or repair out-of-control complexity. FRP isn't a methodology, and—apologies if you bought this book under false pretenses—it won't solve all of your problems. FRP is a specific programming technique to improve your code in an area that just happens to be a common source of complexity (and therefore bugs): event propagation.

Simple things taking too long

I joined a team that was developing a Java-based configuration tool for an embedded system. The software was difficult to modify to the point where a request for adding a check box to one of the screens was estimated as a two-week job.

This was caused by having to plumb the Boolean value through layers of interfaces and abstraction. To solve this, we put together what we'd later discover was a basic FRP system. Adding a check box was reduced to a one-line change.

We learned that every piece of logic, every listener, and every edge case you need to write code for is a potential source of bugs.

1.2 What is functional reactive programming?

FRP can be viewed from different angles:

- It's a replacement for the widely used *observer pattern*, also known as *listeners* or *callbacks*.
- It's a composable, modular way to code event-driven logic.
- It's a different way of thinking: the program is expressed as a reaction to its inputs, or as a flow of data.
- It brings order to the management of program state.
- It's something fundamental: we think that anyone who tries to solve the problems in the observer pattern will eventually invent FRP.
- It's normally implemented as a lightweight software library in a standard programming language.
- It can be seen as a complete embedded language for stateful logic.

If you're familiar with the idea of a domain-specific language (DSL), then you can understand FRP as a minimal complete DSL for stateful logic. Aside from the I/O parts, an arbitrarily complex video game (for example) can be written completely in FRP. That's how powerful and expressive it is. Yet it isn't all-or-nothing—FRP can be easily introduced into an existing project to any extent you like.

1.2.1 A stricter definition

Conal Elliott is one of the inventors of FRP, and this book is about FRP by his definition. We'll call this *true FRP* as a shorthand. What is and isn't FRP? Here's part of Elliott's reply to a Stack Overflow post, "Specification for a Functional Reactive Programming language" (http://mng.bz/c42s):

> *I'm glad you're starting by asking about a specification rather than implementation first. There are a lot of ideas floating around about what FRP is. For me it's always been two things: (a) denotative and (b) temporally continuous. Many folks drop both of these properties and identify FRP with various* implementation *notions, all of which are beside the point in my perspective.*
>
> *By "denotative," I mean founded on a precise, simple, implementation-independent, compositional semantics that exactly specifies the meaning of each type and building block. The compositional nature of the semantics then determines the meaning of all type-correct combinations of the building blocks.*

A true FRP system has to be specified using denotational semantics.

> **DEFINITION** *Denotational semantics* is a mathematical expression of the formal meaning of a programming language. For an FRP system, it provides both a formal specification of the system and a proof that the important property of *compositionality* holds for all building blocks in all cases.

Compositionality is a mathematically strong form of the concept of composability that is often recommended in software design. We'll describe it in detail in chapter 5.

This book emphasizes the practice of FRP as expressed through FRP systems you can use right away. Some of the systems we'll cover aren't true FRP. As we go, we'll point out what's specifically lacking and why it's so important that an FRP system should be based on denotational semantics. We'll cover continuous time in chapter 9.

1.2.2 Introducing Sodium

The primary vehicle for FRP in this book is the authors' BSD-licensed Sodium library, which you can find at https://github.com/SodiumFRP. It's a system with a denotational semantics that we give in appendix E. It's a practical system that has passed through the crucible of serious commercial use by the authors.

We're using Sodium because it's a practically useful, simple, true FRP system. At the time of writing, there aren't many systems like this available in nonfunctional languages. There's minimal variation between FRP systems, so the lessons learned from Sodium are applicable to all systems. To aid in understanding, we'll use Sodium as a common reference point when discussing other systems. This book is about FRP, and Sodium is the best means to that end available to us.

Like anything, Sodium is the product of design decisions. It isn't perfect, and we don't wish to promote its use over any other system. We intend Sodium to be four things:

- A production-ready library you can use in commercial and non-commercial software across a range of programming languages
- A vehicle to promote the true definition of FRP
- A reference and benchmark for future innovation
- A solid learning platform, due to its minimalist design philosophy

1.3 Where does FRP fit in? The lay of the land

NOTE This book assumes knowledge of general programming, but not *functional programming*. Further, to use FRP, you only need a subset of the concepts from functional programming, and we'll explain what you need to know along the way. FRP gives you many of the benefits of functional programming with a shorter learning curve, and you can use it in your existing language.

It may sound oversimplified, but it turns out that FRP is the intersection of *functional programming* and *reactive programming*—see figure 1.1. Here's what these technologies are:

- *Functional programming*—A style or paradigm of programming based on functions, in the mathematical sense of the word. It deliberately avoids shared mutable state, so it implies the use of immutable data structures, and it emphasizes *compositionality*.

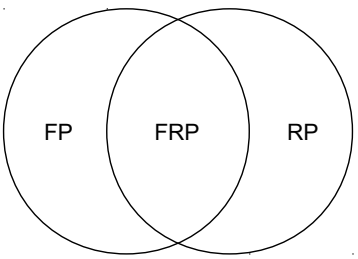


Figure 1.1 FRP is a subset of both functional and reactive programming

Compositionality turns out to be a powerful idea, as we'll explain. It's the reason why FRP can deal with complexity so effectively.

- *Reactive programming*—*A* broad term meaning that a program is 1) event-based, 2) acts in response to input, and 3) is viewed as a flow of data, instead of the traditional flow of control. It doesn't dictate any specific method of achieving these aims. Reactive programming gives looser coupling between program components, so the code is more modular.
- *Functional reactive programming*—A specific method of reactive programming that enforces the rules of functional programming, particularly the property of compositionality.

Typically, systems described as *reactive programming* emphasize distributed processing, whereas FRP is more fine-grained and starts with strong consistency. Consistency must be relaxed to achieve scalability in a distributed system. (We explain why in section 11.3.) FRP and reactive programming take different approaches to this question. FRP can be useful for distributed processing, but it isn't designed specifically for it.

The *Akka* system is classified as reactive programming. It's designed for distributed processing and is largely based on the *actor model*. (We'll contrast FRP against actors in chapter 10.)

Microsoft's *Reactive Extensions* (Rx) isn't true FRP at the time of writing. It sits somewhere between Akka and FRP. There's a difference in design goals between Rx and FRP. Rx is mostly concerned with chaining event handlers, and it gives you many options for how you do it. FRP controls what you do more tightly and gives you strong guarantees in return. Most of what you'll learn in this book can be applied to Rx. We'll cover the FRP-like parts of Rx in chapter 6.

1.4 Interactive applications: what are events?

Most applications are architected around one of two programming models, or a mix of the two:

- Threads
- Events

They're both aimed at managing state changes in response to input, but they achieve it in different ways. Which one to choose depends mainly on the nature of the problem you're trying to solve:

- *Threads* model state transitions as a control flow. They tend to be a good fit for I/O or for any situation where the state transitions fall into a clearly defined sequence. We put actors and generators in this category, too.
- *Events* are discrete, asynchronous messages that are propagated around the program. They're a suitable model where a sequence is less obvious, especially where the interactions between components are more complex. Typical applications include graphical user interfaces (GUIs) and video games.

People have debated which model is the best over the years. We don't think one is better than the other; rather, we consider each good for its proper purpose. When a

thread is best, you should use a thread. But this book is about the second programming model: events. Often they're the best choice, and when they are, this book will teach you how to stay out of trouble.

1.5 State machines are hard to reason about

The term *state machine* refers to any system that works in the following way:

1. An input event comes into the system.
2. The program logic makes decisions based on the input event and the current program state.
3. The program logic changes the program state.
4. The program logic may also produce output.

We've drawn this in figure 1.2. The arrows depict the flow of data.

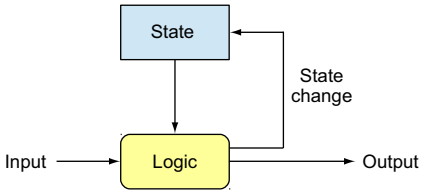


Figure 1.2 The flow of data in a generalized state machine

We normally use the term *state machine* to describe programs or, more commonly, parts of programs, that directly reflect the structure just described. In fact, any program that does anything useful is functionally equivalent to a state machine because it's possible to rewrite any program as a state machine and have it function the same way.

We could say that all programs are fundamentally state machines. But code written in a traditional state-machine style tends to be unreadable and brittle. (Any embedded C programmer will attest to this.) It also tends to be extremely efficient, which is the usual excuse for using this style. The job of the programmer could be seen as finding ways to organize state machines so they're maintainable. Of course, a programmer must express the program so that a computer can run it, but their responsibility doesn't end there. It's not possible to keep all the code in your head at once, unless the code is small or your head is especially large, so a programmer's main task is to structure the code so as to make the program easy to modify. Or, we can say that a programmer's primary focus is managing complexity.

We argue that all programs are state machines, and state machines are inherently difficult to reason about, and this is why programming is difficult. Programmers achieve their task of transforming chaos into order by using a bag of tricks, or a set of abstractions they have learned, which they add to over the years through both study and creativity. Threads and events are two abstractions you'll find rattling about in there. There are many others, and they all have their advantages and disadvantages for different problem domains. This book is about a powerful and very general

abstraction you can add to your toolbox that directly addresses the problem of managing the complexity of state machines.

1.6 Interactive applications without the bugs

The problems we're trying to solve have inherent difficulties; this is true. In spite of this, most of our problems come from the way we're doing things.

Hikers often say that there's no such thing as bad weather, only bad equipment. We say that there's no such thing as bad code, only bad infrastructure.

A large portion—the majority—of bugs in event-based programs are preventable. That's the message of this book.

1.7 Listeners are a mainstay of event handling, but …

Listeners or callbacks—also called the observer pattern—are the dominant way of propagating events in software today. But it wasn't always this way.

In the old days, when the walls were orange, the mice living in them weren't the event sources we know today but were small animals, and list boxes hadn't been invented yet. If you wanted to propagate some value around your program, you got the value and called all the places where that value was going to be used. Back then, the producer had a dependency on its consumers. If you wanted to add a new consumer of your events, then you made the producer call it, too. Programs were monolithic, and if you wanted to reuse some code that produced events (such as a list box), it was a bit of work, because it was wired into the rest of the program.

The idea of a list box as a reusable software component doesn't work well if it has to know in advance what all its consumers are. So the observer pattern was invented: if you want to start observing an event producer, you can come along at any time and register a new consumer (or listener) with it, and from then on, that consumer is called back whenever an event occurs. When you want to stop observing the producer, you deregister the consumer from it, as shown in the following listing.

Listing 1.1 Listeners: the observer pattern

```
public class ListBox {
    public interface Listener {
        void itemSelected(int index);
    }

    private List<Listener> listeners = new ArrayList<>();
    public void addListener(Listener l) {
        listeners.add(l);
    }
    public void removeListener(Listener l) {
        listeners.remove(l);
    }
    protected void notifyItemSelected(int index) {
        for (l : listeners) l.itemSelected(index);
    }
}
```

In this way, listeners invert the natural dependency. The consumer now depends on the producer, not the other way around. This makes the program extensible and gives you modularity through a looser coupling between components.

1.8 Banishing the six plagues of listeners

What could possibly go wrong with the wonderful observer pattern? Uh…yeah. We've identified six sources of bugs with listeners; see figure 1.3. FRP banishes all of them. They are as follows:

- *Unpredictable order*—In a complex network of listeners, the order in which events are received can depend on the order in which you registered the listeners, which isn't helpful. FRP makes the order in which events are processed not matter by making it completely nondetectable.
- *Missed first event*—It can be difficult to guarantee that you've registered your listeners before you send the first event. FRP is transactional, so it's possible to provide this guarantee.
- *Messy state*—Callbacks push your code into a traditional state-machine style, which gets messy fast. FRP brings order.
- *Threading issues*—Attempting to make listeners thread-safe can lead to deadlocks, and it can be difficult to guarantee that no more callbacks will be received after deregistering a listener. FRP eliminates these issues.

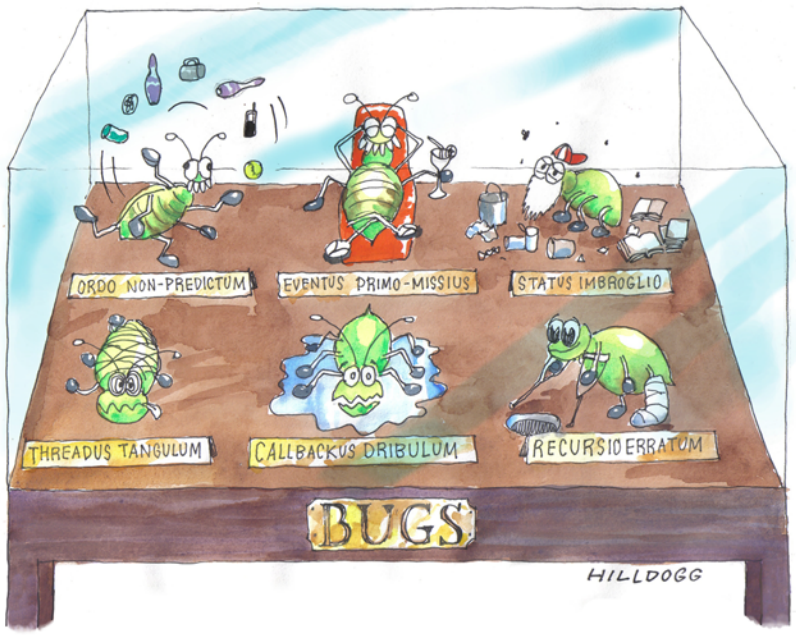


Figure 1.3 The six plagues of listeners

- *Leaking callbacks*—If you forget to deregister your listener, your program will leak memory. Listeners reverse the natural data dependency but don't reverse the keep-alive dependency as you'd like them to. FRP does this.
- *Accidental recursion*—The order in which you update local state and notify listeners can be critical, and it's easy to make mistakes. FRP eliminates this issue.

You'll find a detailed explanation of these problems with examples in appendix B.

1.9 Why not just fix listeners?

We think that if you fix the problems with listeners, you'll invent FRP. In fact, many people in industry have done exactly this, usually for a specific problem domain, and usually without calling it FRP.

We took the research of Conal Elliott, Paul Hudak, and others, and our own experience, and developed a general-purpose, open source (BSD3-licensed) FRP library called Sodium for multiple programming languages with an emphasis on minimalism and practicality. We and Heinrich Apfelmus, the developer of another FRP system called *Reactive Banana*, have compared his system against Sodium, and even though they were developed independently, they turn out to be equivalent apart from naming. We weren't as surprised by this as you might expect.

We believe that FRP isn't so much a clever idea as something fundamental: that when motivated people independently try to fix the observer pattern, after much stumbling, they will eventually converge on similar solutions; and that there are only a few possible variations in that "perfect" design. We'll discuss these. FRP is a discovery, not an invention.

Implementing an FRP system turns out to be surprisingly difficult. We had the work of others to steal from, yet it took more than six person-months of work to understand the issues and develop an FRP system. And the library is only 1,000 lines of code!

We think the reason is that event handling is an inherently hard problem. With listeners, we deal with the frustrations in small doses every day, but to develop an FRP library, you need to deal with them all at once.

We highly recommend that you choose an existing library and not reinvent the wheel. If there isn't one available for your language of choice, we recommend that you consider porting an existing implementation.

1.10 "Have you tried restarting it?" or why state is problematic

We've all had the same experience. The software you're using gets into a bad state. *Something* inside the program hasn't been updated properly, and it won't work anymore. That's right, you know what to do, and there are lots of internet memes about it:

- KEEP CALM and REBOOT.
- CTRL ALT DELETE fixes everything.
- If all else fails, RESTART.
- How I fix stuff working on IT: Restart whatever isn't working 88%. Quick Google search 10%. Weird IT voodoo 2%.

- WHAT IF I TOLD YOU A restart will fix your computer.
- [And the best one:] Restart the world!

We think programming needs a reboot. In most languages, you can declare a variable like this

```
int x = 10;
```

and then modify its value, like this:

```
x = x + 1;
```

In FRP, we don't use normal mutable variables because they're sensitive to changes in execution sequence. A common bug is to read the variable before or after it was updated when you intended to do the opposite.

FRP keeps state in containers called *cells*. They solve the sequence problem because they automatically come with update notification and sequence management: the state is always up-to-date. State changes happen at predictable times.

The strange case of program configuration

Have you worked on a program with a complex configuration that affects many parts of the code? What happens in your project when the configuration changes while the program is running?

Each module that uses the configuration has to register a listener to catch the updates, or it won't catch the updates. This complicates the code, and it's easy to make mistakes in propagating the configuration changes to the right places.

And how is it tested? That's right: it isn't. This sort of code is difficult to put into a form suitable for unit testing.

How many software projects have given up entirely and require a program restart to pick up the new configuration? When you change something on your ADSL router, does it take effect immediately, or are you required to reboot?

Why is program configuration, which should be simple, such an intractable problem in practice? Could this be indicative of something fundamental we've gotten wrong in the way we program?

1.11 The benefit of FRP: dealing with complexity

We all know from experience that the complexity of a program can get out of control. This is so common that it's considered normal in industry. When complex parts interact in complex ways, the complexity can compound.

FRP deals with complexity in a specific way. It enforces a mathematical property called *compositionality*. This enables software components to be composed without unexpected side effects. As the program gets larger and more complex, compositionality becomes more and more important. It makes software more scalable in a fundamental

way. The reasons behind this will be easier to explain when you've grasped the fundamentals, so we'll cover compositionality in detail in chapter 5.

1.12 How does FRP work?

We'll illustrate how FRP works with a simplified flight-booking example, shown in figure 1.4. Here's the specification:

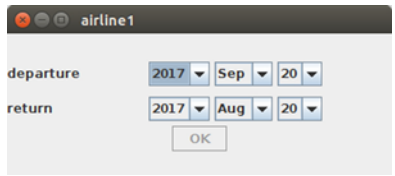


Figure 1.4 Simplified flight-booking example

- The user can enter their departure and return dates using two date field widgets.
- While the user is using the mouse and keyboard to enter the dates, some business logic continuously makes decisions about whether the current selection is valid. Whenever the selection is valid, the OK button is enabled, and when it's not, the button is disabled.
- The business rule we're using is this: *valid if departure date <= return date.*

In the figure, we're trying to depart in September and return in August of the same year, which is the wrong order. So the business rule returns `false`, and you can see that the OK button is disabled (grayed out).

A conceptual view of this application is shown in figure 1.5. We're representing the GUI widgets as clouds to indicate that their internal structure is hidden. In other words, they're *black boxes.*

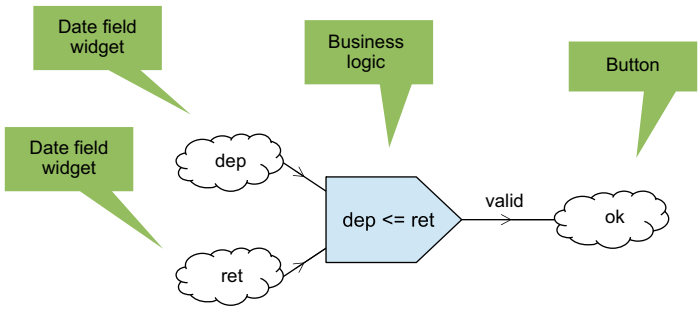


Figure 1.5 Conceptual view of the flight-booking example

A central idea is this: *departure and return dates, as well as the "valid" status from the business logic, all change dynamically in response to user input events.* The lines show a *flow of data* from the two dates, through some logic, and into the OK button. As the user changes the dates, the OK button is enabled or disabled dynamically according to the decision made by the logic.

The next listing gives the code to construct the widgets and the logic, with the Java GUI setup left out. Don't expect to understand every detail right now; we'll return to this code in section 1.16.2.

Listing 1.2 Flight-booking example using FRP

```
SDateField dep = new SDateField();
SDateField ret = new SDateField();
Cell<Boolean> valid = dep.date.lift(ret.date,
             (d, r) -> d.compareTo(r) <= 0);
SButton ok = new SButton("OK", valid);
```

We're using Java and the authors' Sodium FRP library. We'll branch out into other FRP systems and languages later in the book. What we use doesn't matter much for the teaching of FRP. Apart from surface differences, FRP is much the same in any language or FRP system.

FRP uses two fundamental data types:

- *Cells* represent *values that change over time*. Your programming language already has variables that allow you to represent changing values. This book is about the advantages that FRP abstractions give you if you use them, instead.
- *Streams* represent *streams of events*. We'll introduce streams in chapter 2.

The key idea we want you to get is that the code directly reflects the conceptual view in figure 1.5.

We're using a toy library that we wrote with GUI widgets that all start with *S*: `SDateField` and `SButton` are like normal widgets, except we've added an FRP-based external interface. This allows us to avoid some mechanics we don't want to cover yet.

`SDateField` exports this public field:

```
Cell<Calendar> date;
```

`Calendar` is the Java class that represents a date. We said a cell is a value that changes over time, so `Cell<Calendar>` represents a date that changes over time.

As you can see, `Cell` takes a type parameter that tells us the type of value the cell contains, and we do this in Java with generics in the same way we do with lists and other data structures. For instance, you might have a list of dates, and that would have the type `List<Calendar>`. `Cell<Calendar>` is the same concept, but it represents a single date that can change, instead of a list of dates.

The `date` field of `SDateField` gives the date as it appears on the screen at any given time while the user is manipulating the `SDateField` widget. In the code you can see the two dates being used with a `lift()` method.

> **NOTE** The expression `(d, r) -> d.compareTo(r) <= 0` in listing 1.2 is lambda syntax, which is new in Java 8. If this confuses you, don't worry. We're only talking about concepts at this stage. We'll explain how this all works in chapter 2.

You can check out and run this example with the following commands. You'll need Java 8, which means at least version 1.8 of the Java Development Kit (JDK). We've written scripts for both Maven and Ant, which are two popular build systems in Java, both from the Apache Software Foundation. Windows users might find Maven easiest. The Windows version of Ant is called *WinAnt*. Here are the commands:

```
git clone https://github.com/SodiumFRP/sodium
cd sodium/book/swidgets/java
mvn test -Pairline1    or    ant airline1
```

Look in other directories. You may find that the examples have been translated into other languages.

NOTE Sodium can be found on Maven's Central Repository with a groupId of `nz.sodium`.

1.12.1 Life cycle of an FRP program

Figure 1.6 shows the mechanics of how FRP code is executed. In most FRP systems, this all happens at application runtime. There are two stages:

- *Stage 1: Initialization*—Typically during program startup, FRP code statements are converted into a directed graph in memory.
- *Stage 2: Running*—For the rest of the program execution you feed in values and turn the crank handle, and the FRP engine produces output.

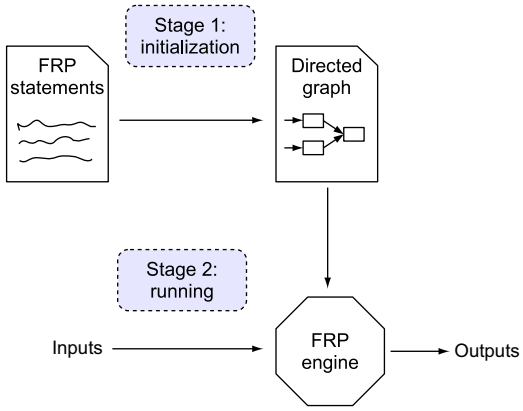


Figure 1.6 The stages of execution of an FRP program

NOTE In practice, the program spends most of its time in the *running* stage with an already-constructed directed graph. But the graph can also be dynamically modified during the running stage. We cover this in chapter 7.

A major task of the FRP engine is to ensure that things are processed in the order specified by the dependencies in the directed graph that's maintained in memory. In spreadsheets, this is referred to as *natural order recalculation*. It could be better described as the "correct order" to distinguish it from any other order, which could give the wrong result.

This separation between *initialization* and *running* stages is similar to the way GUI libraries normally work. You construct all your widgets first (*initialization*), and afterward an event loop handles events from the user (*running*). The Java GUI framework Swing, which we're using here, works this way.

During the initialization stage, the flight-booking example executes this FRP statement:

```
Cell<Boolean> valid = dep.date.lift(ret.date,
            (d, r) -> d.compareTo(r) <= 0);
```

This code expresses a relationship between the widgets, and nothing else.

NOTE *Lifting* is a general functional programming concept. We'll return to it in chapter 2.

Once the initialization is over, we enter the running stage. Java creates a window and processes incoming mouse and keyboard events from the user. The FRP engine's job is to maintain the relationship we expressed, ensuring that the value of `valid` is always up to date.

We could write a spreadsheet to do this, as shown in figure 1.7. In fact, FRP works the same way a spreadsheet does. Our choice of the class name `Cell` to represent dynamically changeable values was partially influenced by this.

Can you really write arbitrarily complex application logic in the style of a spreadsheet? Yes, you can. That's exactly what FRP allows you to do. But you'll have to think a bit differently.

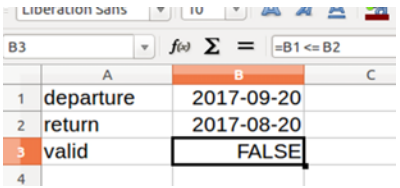


Figure 1.7 We can express the flight-booking example as a spreadsheet.

1.13 Paradigm shift

In his 1962 book *The Structure of Scientific Revolutions,* Thomas Kuhn argued that science progresses in leaps, which he called *paradigm shifts*, rather than in a slow, linear progression as people often thought.

DEFINITION A *paradigm* is a way of thinking, a world view, a philosophical framework, or a frame of reference. A paradigm usually applies to a particular area of knowledge. It's based on a set of underlying assumptions.

FRP belongs to a new paradigm. Although you can use FRP to make incremental improvements to an existing code base, there's a different way of thinking underlying it. If you embrace that way of thinking, you'll get the most out of FRP.

1.13.1 Paradigm

Everyone's way of thinking is based on a set of assumptions, mostly at the subconscious level, which are taken to be true. They may or may not be true. They underlie everything that person knows. No matter how clever someone is, there are always some assumptions that have gone unquestioned.

Most people share most of their assumptions about the world around them, and these shared assumptions provide a frame of reference that enables us to communicate with each other. When there's a significant difference in the assumptions between two

people, communication becomes more difficult. We say that people are operating in different paradigms. A good example is the culture shock you can experience when you visit another country.

In certain areas of knowledge, people's assumptions can be very different indeed. When this happens, the experience can be jarring. A statement made in terms of frame of reference *A* can be nonsensical with respect to frame of reference *B*. Each person may even think the other is insane. Thomas Kuhn described this situation by saying that the two ways of thinking are *incommensurable*.

By way of example, on Christmas, we like to eat ice cream at the beach and then jump into the sea. This may seem like strange behavior, but what else would you do on a hot summer day?

1.13.2 Paradigm shift

A person can change their paradigm, either slowly or all at once through an epiphany. You usually need a crisis to bring about an epiphany, so we prefer the first method.

If you're used to object-oriented programming (OOP), then you'll be entering a new paradigm. FRP will appear a bit strange. We can claim all day that FRP is a simple idea, and it is. But *simple* and *easy to understand* aren't the same thing. The reality is that until your thinking slots into place, you will encounter some challenges.

FRP rests on certain notions about what's important in programming that may go against your current understanding. Without these ideas, FRP is just a way of taking something that should be straightforward and doing it in an eccentric, limiting way. The claimed benefits will be remote.

We're talking about standard functional programming ideas. If you've done functional programming, it will be easier to learn FRP. If not, that's no problem. You don't need to know everything about functional programming to use FRP, and we'll teach you what you need to know.

> **NOTE** FRP made it easier for one of the authors to learn functional programming. The other author learned these the other way around.

We're asking you to question some of your assumptions. This can be intriguing, challenging, and liberating. Through this process, you either change your beliefs or strengthen them. Either way, it's beneficial, but it's never easy.

Next we'll start laying the foundation for thinking in FRP. We'll continue throughout the book.

1.14 Thinking in terms of dependency

Traditionally, software is expressed as a sequence of steps, written by the programmer and executed by the machine. Each step has a relationship with the steps that came before. Some steps will depend on the previous step, and some may depend on things from much earlier. Consider these steps:

- Comb hair
- Wash face

There is no dependency between these two statements. They can be executed in any order or simultaneously if you have enough hands.

Here's another classic example that functional programmers use:

1. Open silo doors
2. Fire missiles

In this case, the dependency implied by the sequence is critical.

If we weren't using FRP, we'd write the flight-booking example much like in listing 1.3. We assume the existence of a `JDateField` widget, which doesn't exist in reality. Notice that there are certain places where the order of statements is critical—things will break if it isn't right.

Listing 1.3 Flight-booking example in a traditional non-FRP style

```
public class BookingDialog {
    public BookingDialog() {
        JDateField startField = new JDateField(...);        <-- GUI
        JDateField endField = new JDateField(...);
        this.ok = new JButton("OK");
        ...;
        this.start = startField.getDate();        <-- Logic
        this.end = endField.getDate();
        update();
        startField.addDateFieldListener(new DateFieldListener() {
            public void dateUpdated(Calendar date) {
                BookingDialog.this.start = date;
                BookingDialog.this.update();
            }
        });
        endField.addDateFieldListener(new DateFieldListener() {
            public void dateUpdated(Calendar date) {
                BookingDialog.this.end = date;
                BookingDialog.this.update();
            }
        });
    }
    private JButton ok;
    private Calendar start;
    private Calendar end;
    private void update() {
        boolean valid = start.compareTo(end) <= 0;
        ok.setEnabled(valid);
    }
}
```

Order is critical

Order is critical

Order is critical

Threads allow you to express sequence; events allow you to express dependency. In different situations, both are needed. A lot of problems come from trying to express dependency with threads, or sequences with events.

Given a conceptual diagram like the one we drew for the flight-booking example, we can extract the dependency relationships easily, as shown in figure 1.8. All we need to

do is remove the unnecessary bits and reverse the data-flow arrows to turn them into a "depends on" relationship.

The FRP engine knows all these relationships, so it can automatically determine the dependencies. From that, the correct sequence is guaranteed.

The example we've given is simple, but sequence-dependent code can get complex. The problem with representing dependencies as a sequence comes when you go to change the code. To make something happen earlier or later, you need to make sure you fully understand the dependencies implicit in the existing sequence. In FRP, you express dependencies directly, so you can just add or remove the dependencies, and the sequence is automatically updated. It's impossible to make a sequence mistake.

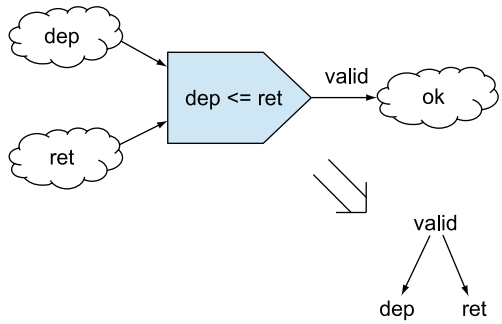


Figure 1.8 Extracting "depends on" relationships from a conceptual diagram: reverse the data-flow arrows.

With regular listener-based event handling, you can express dependency, but it's still difficult to maintain a reliable sequence. The order of processing depends on when the code propagates events and also on the order in which listeners were registered. (This is the plague we call *unpredictable order*.) It's easy to change this inadvertently, resulting in unwanted surprises. When this happens in a complex program, it can take some time to unravel.

We naturally think of our problems in terms of dependency. Programming has largely been concerned with translating that into a sequence. You've no doubt become good at this. FRP does away with this aspect of programming so you stay in dependency-land, and you can program in a way that's closer to the problem description.

1.15 Thinking declaratively: what the program is, not what it does

In FRP we talk about working in the *problem space* rather than working in the *machine space*. Decades of software development have made the authors lazy. We don't want to add sequence information to our code if we don't have to. We'll only end up having to debug it.

The sequence can be derived from dependencies, so you can write less code by leaving the sequence out altogether. You end up with a lot more "what" and a lot less "how." This style is referred to as *declarative programming*: you tell the machine what the program is, not what it does. You directly describe *things* and the *relationships between them*.

We wanted to demonstrate that the "what" of programming is easier to combine, reason about, and understand than the "how"—but we didn't want to do it on an empty stomach. So we decided to cook some lasagna. But when we looked up the recipe, we were horrified to see this:

1. Heat the oil in a large pan.
2. Fry the onions until golden.
3. Add ground beef and tomato.

A sequence! We'll spare you any more of this torture.

This recipe is a list of tiny sequence-dependent details with no overview. Imagine if you were asked what lasagna is by someone from Australia, where lasagna is practically unknown. If you gave them the full monologue of the recipe, they'd have a hard time understanding what they were going to get. This is no way to write a cookbook: it's an *operational definition* of lasagna, defined in terms of the steps needed to create it. Declarative programming instead uses a *conceptual definition*—see figure 1.9.

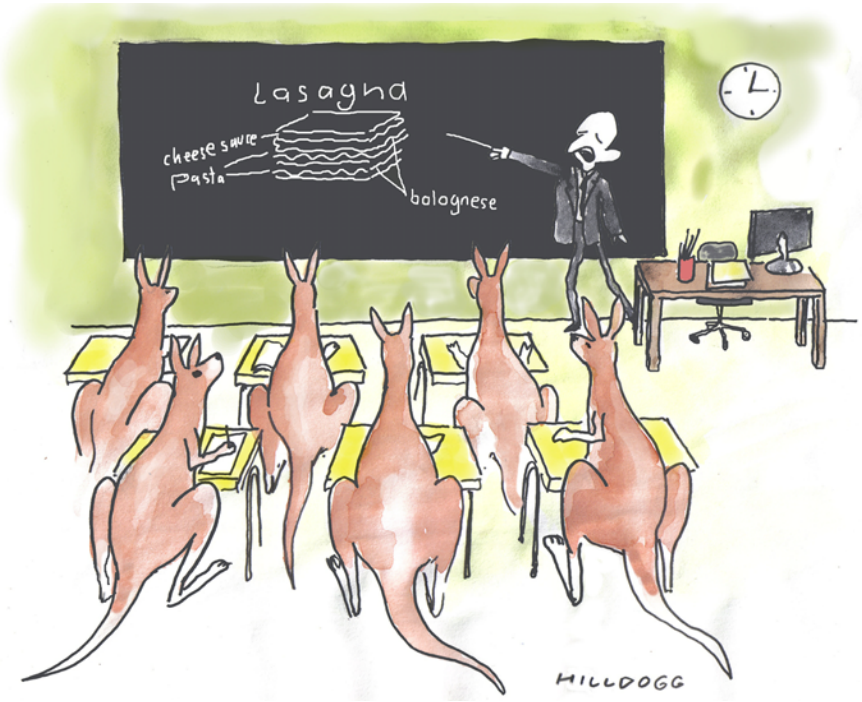


Figure 1.9 A conceptual definition is easier to grasp than a long list of detailed instructions.

This is how our cookbook would be written:

- *Lasagna* is grated cheese on cheese sauce on flat pasta on cheese sauce on Bolognese on flat pasta on cheese sauce on Bolognese on flat pasta on cheese sauce baked for 45 minutes.
- *Bolognese* is onion and oil fried until golden mixed with ground beef mixed with tomato simmered for 20 minutes.
- *Cheese sauce* is milk and cheese added progressively to roux while frying it until the sauce thickens.
- *Roux* is flour and butter fried briefly.

- *Baked* is put in an oven dish in a hot oven.
- *Fried* is put in a pan on high and mixed frequently.
- *Simmered* is put in a pan on low and mixed infrequently.

We're fond of code reuse, so we thought we'd include some further recipes:

- *Spaghetti Bolognese* is Bolognese on boiled and drained spaghetti.
- *Macaroni and cheese* is cheese sauce on boiled and drained macaroni.

We want everyone to enjoy functional cooking, so some may find this alternative useful:

- *Cheese sauce* is tomato paste, tahini, oil, rice flour, red miso, soy sauce, soy milk, and nutritional yeast mixed.

Notice a few things:

- We express dependencies directly. The sequence is derived from them. We don't care whether the cheese sauce is made before or after the Bolognese or simultaneously, and neither should anyone else.
- It's closer to a conceptual view of the food, so it's easy to understand.
- It's short, so our cookbook will need to be padded out with a lot of pictures.
- We can compose the parts into new recipes easily.

As we go through, we'll give concrete examples of what a program *is*, not what it *does*. For now, we'll leave you with this thought that is the basis of the philosophy behind FRP:

- A program is a transformation from inputs to outputs.

1.16 Conceptual vs. operational understanding of FRP

There are two ways to understand how FRP works: *operationally* or *conceptually*, like we did with the lasagna. A large part of the point of FRP is that it's *conceptually* simple.

Most programmers are accustomed to operational thinking, and we want to turn you away from that. With FRP, operational thinking not only is a more complex way to approach it, but also pollutes your mind with unnecessary details. Pay no attention to the man behind the curtain, as it were.

A few sections back, we presented some code for a flight-booking example. Later, we showed how you'd implement this in a traditional style, using listeners or callbacks. It probably won't surprise you to learn that under the covers of most FRP systems, everything is done with listeners.

> **NOTE** Not all FRP systems use listeners internally. Push-based FRP systems typically do, but there are also pull-based systems—where values are calculated on demand—and hybrids of the two. How a system works internally affects performance only. It makes no difference to the meaning of the FRP code you write. Sodium is a push-based system.

You may well ask, "What's really going on when the code runs?" We'll tell you in a moment, but first, we want to emphasize the importance of conceptual thinking.

1.16.1 Opening your mind to FRP

We ask you to clear your mind so it's receptive, like an empty rice bowl. Picture data flowing through a limitless stream of code (see figure 1.10). Are you feeling it?

Figure 1.10 A stream of code

What is the essential nature of the `Cell` class we introduced to you? FRP is said to be *reactive* because in writing the code, you're always responding to input. When working in FRP, we need you to view cells only as sources of information, and as the only sources of information. You never say, "And now, the code will push this value." Banish such thoughts from your mind.

FRP logic is a flow of data. Data flows into your logic through *streams* and *cells*. Data flows out the output side. The bit in the middle is also a flow of data. FRP code is a reaction to its input. Data flows from input to output. FRP is fundamentally *a declarative description of the output in terms of the input*.

In FRP, you're conceptually working at the level of the data flow, not the individual event value. We've found from experience that when you introduce people to FRP, they inevitably ask one question:

"How do you get the value?"

"In the far north," we reply, "a stream of values flows over rocks. Seeking a moment's rest, a value alights on a willow."

When working with FRP, try to stay conceptual in your thinking. Try to stay at the level of the relationships between things, not the mechanics of their interaction.

1.16.2 What's really going on when the code runs?

In the flight-booking example we gave earlier, we presented some code. Listing 1.4 gives the same code with more of the ancillary detail, but we've left out some ultra-verbose layout-related tomfoolery. The `nz.sodium.*` import gives the Sodium FRP system including `Cell`.

The `swidgets.*` import gives our "toy library" for GUI widgets. The `SDateField` and `SButton` widgets are like normal widgets, but jazzed up with an FRP interface. We did this because otherwise we would have needed to tell you how to feed data into `Cell` and how to get it out.

Interfacing FRP to the outside world isn't difficult, but we consider it of the highest importance at this early stage to keep you away from operational thinking. Our experience in teaching FRP is that people gravitate toward the things they know. When someone new to FRP sees a listener-like interface, they'll say, "I know how to use this." They then slip back into established habits and ways of thinking. We'll tell you how to interface FRP to the rest of your program later in the book.

Listing 1.4 More detail for the flight-booking example

```
import javax.swing.*;
import java.awt.*;
import java.util.Calendar;
import swidgets.*;
import nz.sodium.*;

public class airline1 {
    public static void main(String[] args) {
        JFrame view = new JFrame("airline1");
        view.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        SDateField dep = new SDateField();
        SDateField ret = new SDateField();
        Cell<Boolean> valid = dep.date.lift(ret.date,
            (d, r) -> d.compareTo(r) <= 0);
        SButton ok = new SButton("OK", valid);

        GridBagLayout gridbag = new GridBagLayout();
        view.setLayout(gridbag);
        GridBagConstraints c = new GridBagConstraints();
        ...
        view.add(new JLabel("departure"), c);
        view.add(dep, c);
        view.add(new JLabel("return"), c);
        view.add(ok, c);
        view.setSize(380, 140);
        view.setVisible(true);
    }
}
```

What goes on when this code runs? During *initialization*, a push-based FRP system typically constructs a network of listeners like the one we've drawn in figure 1.11. We're showing these in a style more akin to unified modeling language (UML), where each box represents a Java object in memory with a list of listeners that are currently registered. Objects that listen to other objects are called back on their `update(..)` method.

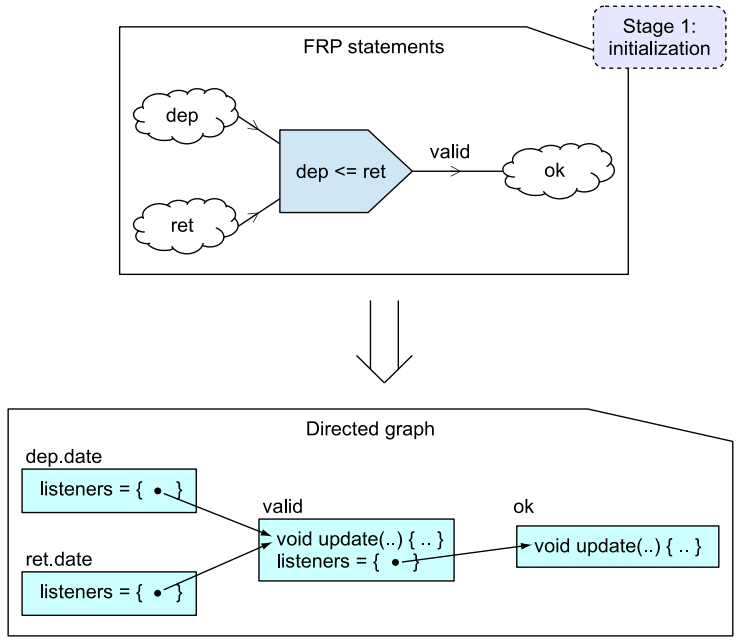


Figure 1.11 Behind the scenes, the FRP system translates FRP statements into a directed graph of listeners.

During the *running* stage, nothing happens until the user changes the departure or return date. Then this sequence of events occurs:

1. `dep.date` or `ret.date` notifies its listeners of the change.
2. The `update(..)` method for `valid` is called, and the logic for the business rule is recalculated with the latest values. `valid` then notifies its listeners.
3. The `update(..)` method for `ok` is called, which causes the button widget to be enabled or disabled.

1.17 Applying functional programming to event-based code

A lot of the power of FRP comes from the fact that FRP *cells* and *streams* follow the rules of functional programming in a way that listener/callback code can never hope to do.

It becomes possible to manipulate event-based code using functional programming. FRP allows functional programming to become a *meta-language* for event-based logic.

DEFINITION *Meta-language*—A language used to manipulate the code of a second language

With FRP, functional programming isn't manipulating logic directly—it's manipulating the statements of a logic language. We want to give you a taste of that.

NOTE This final section of chapter 1 will be difficult for people who are completely new to functional programming. It's just to show you what FRP can do, and it isn't necessary for learning the material. If this example gives you a dose of cataplexy, skip it for now. It'll be easy to follow after you've finished chapter 2.

We're going to encapsulate business rules with a class called `Rule` so we can manipulate rules as a concept. We can rewrite the rule from the first example (return can't precede departure) like this:

```
Rule r1 = new Rule((d, r) -> d.compareTo(r) <= 0);
```

Here we're writing the code of the rule using Java 8 lambda syntax and passing that as the argument to `Rule`'s constructor.

In China, the numbers 4, 14, and 24 are considered unlucky. You can define a new business rule that doesn't allow travel on unlucky dates. Given a function

```
private static boolean unlucky(Calendar dt) {
    int day = dt.get(Calendar.DAY_OF_MONTH);
    return day == 4 || day == 14 || day == 24;
}
```

the rule is expressed as

```
Rule r2 = new Rule((d, r) -> !unlucky(d) && !unlucky(r));
```

You also need a way to combine rules, such that this rule

```
Rule r = r1.and(r2);
```

returns `true` if rules `r1` and `r2` are both satisfied.

Listing 1.5 gives the code for the `Rule` class. It's a container class for a function that takes two dates (`Calendar`s) and returns `true` if the rule deems the given dates to be valid. The `Lambda2` class comes from Sodium.

DEFINITION *Reify*—A functional programming term meaning to convert an abstract representation of something into real code

In line with this definition, the `reify()` method "compiles" the abstract rule into real FRP code. You use `Rule` first to manipulate rules as an abstract concept, and then you *reify* the result into executable code. In this example, `reify()` takes the cells

representing the departure and return dates and returns a cell representing whether the supplied dates are valid according to that rule.

`and()` is a method for manipulating existing rules. It combines two rules to give a new rule that is satisfied if both the input rules are satisfied.

Listing 1.5 Encapsulating a business rule

```
class Rule {
    public Rule(Lambda2<Calendar, Calendar, Boolean> f) {
        this.f = f;
    }
    public final Lambda2<Calendar, Calendar, Boolean> f;
    public Cell<Boolean> reify(Cell<Calendar> dep, Cell<Calendar> ret) {
        return dep.lift(ret, f);
    }
    public Rule and(Rule other) {
        return new Rule(
            (d, r) -> this.f.apply(d, r) && other.f.apply(d, r)
        );
    }
}
```

NOTE Old programming text books may frown on nondescriptive variable names like `f` for *function*, but functional programmers do this because they always want their code to be general unless it absolutely has to be related to a specific problem space. The class name `Rule` says what it is. The contained function's name doesn't need to add to this.

The following listing is the logic of the main program. It implements the two business rules described using the new `Rule` class.

Listing 1.6 Manipulating abstract business rules

```
private static boolean unlucky(Calendar dt) {
    int day = dt.get(Calendar.DAY_OF_MONTH);
    return day == 4 || day == 14 || day == 24;
}
...
    SDateField dep = new SDateField();
    SDateField ret = new SDateField();
    Rule r1 = new Rule((d, r) -> d.compareTo(r) <= 0);
    Rule r2 = new Rule((d, r) -> !unlucky(d) && !unlucky(r));
    Rule r = r1.and(r2);
    Cell<Boolean> valid = r.reify(dep.date, ret.date);
    SButton ok = new SButton("OK", valid);
```

Return can't precede departure.

Can't travel on unlucky dates

To run this example, check out the code with git if you haven't done so already, and then run it. These are the commands:

```
git clone https://github.com/SodiumFRP/sodium
cd sodium/book/swidgets/java
mvn test -Pairline2    or    ant airline2
```

This is a small example of an approach that becomes powerful as the problem gets more complex. In chapter 12, we'll take this concept further and present an implementation of a GUI system done this way.

1.18 Summary

- *Listeners* or *callbacks* have a set of problems that we call the six plagues.
- FRP replaces state machines and listeners or callbacks.
- FRP deals with complexity through a mathematical property called *compositionality*.
- Thinking in terms of dependency is better than thinking in terms of sequence.
- FRP code has a structure like a *directed graph*, and an FRP engine derives execution order automatically from it.
- FRP uses a *declarative programming* style, meaning you think of the program in terms of what it is, not what it does.
- FRP is something fundamental: it's a discovery, not an invention.
- FRP allows functional programming to be used as a *meta-language* for writing event-based logic.

PROGRAMMING

Functional Reactive Programming

Blackheath • Jones

Today's software is shifting to more asynchronous, event-based solutions. For decades, the Observer pattern has been the go-to event infrastructure, but it is known to be bug-prone. Functional reactive programming (FRP) replaces Observer, radically improving the quality of event-based code.

Functional Reactive Programming teaches you how FRP works and how to use it. You'll begin by gaining an understanding of what FRP is and why it's so powerful. Then, you'll work through greenfield and legacy code as you learn to apply FRP to practical use cases. You'll find examples in this book from many application domains using both Java and JavaScript. When you're finished, you'll be able to use the FRP approach in the systems you build and spend less time fixing problems.

What's Inside

- Think differently about data and events
- FRP techniques for Java and JavaScript
- Eliminate Observer one listener at a time
- Explore Sodium, RxJS, and Kefir.js FRP systems

Readers need intermediate Java or JavaScript skills. No experience with functional programming or FRP required.

Stephen Blackheath and **Anthony Jones** are experienced software developers and the creators of the Sodium FRP library for multiple languages.

Illustrated by Duncan Hill

"A gentle introduction to the necessary concepts of FRP."
—From the Foreword by Heinrich Apfelmus, author of the Reactive-banana FRP library

"Highly topical and brilliantly written, with great examples."
—Ron Cranston, Sky UK

"A comprehensive reference and tutorial, covering both theory and practice."
—Jean-François Morin
Laval University

"Your guide to using the merger of functional and reactive programming paradigms to create modern software applications."
—William E. Wheeler
West Corporation

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/functional-reactive-programming

ISBN-13: 978-1-63343-010-5
ISBN-10: 1-63343-010-3
54999
9 781633 430105

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]