Tomas Petricek
with Jon Skeet

Functional Programming for the Real World

With examples in F# and C#

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

> We could add an additional primitive just like `wiggle` that would give us a location of the mouse cursor. We could implement this for example by adding the mouse location to the `BehaviorContext` type. This is quite interesting because it would allow us to create animations that depend on the mouse location. A more sophisticated extension could allow us to create non-linear dynamic systems. We could add a primitive that tells us how quickly is a certain behavior value changing and we could then use it to create system that depends on how quickly its state is changing.
>
> The animation library we created is indeed far from being a robust physical simulation, but it shows an interesting direction. Some of the ideas that I outlined in this sidebar will be available on the book web site after the book is published.

The animation library implemented as a domain specific language is an interesting example of a very useful functional programming style. This style can be of course used for developing a wide variety of applications, so in the next section we'll briefly sketch a domain specific language for a completely different area.

15.6 Developing financial modeling language

So far in this chapter, we've seen most of the ideas that you need to know if you plan to design your own domain specific language. To give you some idea how this could be done for a more business oriented problem, we'll briefly sketch a language that can be used for modeling financial contracts. This example is motivated by an article by Simon Peyton Jones et al. *Composing contracts: an adventure in financial engineering* [Jones, Eber, Seward, 2000]. In this section, we'll implement only the most basic parts of the language, so you can look at the article for more information.

15.6.1 Defining the primitives

Similarly as when creating the animation language, we'll need to start by defining the type of the values we're working with and by implementing a couple of primitives that can be later composed. Our primitive data type will be called `Contract` and it will represent trades that can occur at some particular date and time.

DECLARING THE CONTRACT TYPE

As you can see in listing 15.23, we're using similar technique as when declaring behaviors and we're creating a discriminated union with a single discriminator that contains a function that calculates the list of possible trades.

Listing 15.23 Type representing financial contracts (F# interactive)

```
> type Contract =
     | CF of (DateTime -> seq<int * string>);;                       #1
(...)
> let eval (CF f) dt = f(dt) |> List.of_seq;;                        #2
val eval : Contract -> DateTime -> (int * string) list
```

#1 Contract can calculate it's trades

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

#2 Gets a list of trades at particular date

The function that represents the actual contract takes a single argument and returns a sequence of tuples (#1). When we call it with a particular date as an argument, it will generate all trades that can occur at the given date. The trade is represented simply as a tuple containing a number of stocks that we want to buy or sell and the name of the stock. We'll use positive numbers to represent buying and negative values to represent selling of stocks.

The second part of the listings implements an `eval` function (#2) that evaluates the contract at some time and returns the list of trades. We're using a sequence to represent the trades in the contract, because that makes the code more general and we could in principle also represent infinite number of possible operations. However, the `eval` function returns a list, because we expect that the overall result will be finite.

IMPLEMENTING COMBINATORS

Once we have the data type representing values of our language, we need to implement a couple of primitive functions for creating and composing these values. In case of behaviors, we created primitive values such as `wiggle` and we declared lifted operators for composing them. In case of contracts, we'll start with a function `trade` that creates a contract representing a single purchase that can occur at any time. To compose contracts, we'll provide a function `combine`, which unions trades of the two provided contracts.

The listing 15.24 shows the implementation of these two functions as well as functions for restricting the dates when the contracts can occur and a function for creating trades where we're selling some stocks.

Listing 15.24 Combinators for creating and composing contracts (F# interactive)

```
> let trade amount what = CF(fun _ ->                            #1
     seq { yield amount, what })
  let combine (CF a) (CF b) = CF(fun now ->                      #2
     Seq.concat [ a(now); b(now) ])
  ;;
val trade : int -> string -> Contract
val combine : Contract -> Contract -> Contract

> let after dt (CF f) = CF(fun now ->                            #3
     seq { if now >= dt then yield! f(now) })
  let until dt (CF f) = CF(fun now ->                            #3
     seq { if now <= dt then yield! f(now) })
  let give (CF f) = CF(fun now ->                                #4
     seq { for am, itm in f(now) -> -am, itm })
  ;;
val after : DateTime -> Contract -> Contract
val until : DateTime -> Contract -> Contract
val give : Contract -> Contract
```

#1 Single trade of specified number of stocks
#2 Concatenate trades of two contracts
#3 Limit the date when contract is active
#4 Change sale to purchase and conversely

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

A single trade that can occur at any time is represented as a function that ignores its parameter (a date when we're evaluating the contract) and returns a sequence with a single element (#1). Composition is also easy (#2), because we simply concatenate all trades of the two underlying contracts that can occur at the given date.

The next two primitives let us limit the date when a contract is active (#3). We implemented them by creating a function that tests whether the date when we're evaluating the contract matches the condition of the primitive. When the test succeeds, it returns all underlying trades using the `yield!` primitive, otherwise it returns an empty sequence. Finally, the last primitive can be used to change whether a contract is sale or a purchase of the specified stocks. We implement it by iterating over all the underlying trades of a contract and changing positive amounts to negative and vice versa.

As I wrote earlier, the goal of this section is only to sketch how a language for describing financial contracts might look like. However, even with the very limited example that we've just implemented, we can describe many interesting things.

15.6.2 Using the modeling language

Perhaps the most valuable thing about domains specific languages is that we can use the basic primitives provided by the library designer to create more complicated functions for composing contracts. This makes the library quite flexible, because the users of the library (in our case financial experts) can create the primitives that they need. As the designers of the core library, we only need to provide primitives that are rich enough to allow that.

In the listing 15.25, we'll briefly look at two such functions that are defined in terms of the primitives we've seen in the previous section. It defines function for specifying time interval within which a trade can occur and a function that creates a trade valid at one specific date.

Listing 15.25 Implementing derived financial contract functions (F# interactive)

```
> let between dateFrom dateTo contract =
     after dateFrom (until dateTo contract);;
val between : DateTime -> DateTime -> Contract -> Contract

> let tradeAt date ammount what =
     between date date (trade ammount what);;
val tradeAt : DateTime -> int -> string -> Contract
```

The first function is composed from the two primitives that we defined for restricting the date of the contract. It takes the starting date and the ending date and a contract and returns a contract that can happen at any time within the specified interval. The second function creates a primitive trade that can occur only at the precisely specified date. It uses `trade` function to construct elementary trade and then limits it validity using the `between` function. Note that `after` and `until` function use operators that allow equality (>= and <=), so the use of between is reasonable.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

Equipped with these functions for creating and composing contracts, let's now try to write some contract and evaluate what trades can occur as part of the contract at two distinct dates. The listing 15.26 shows a contract where we're willing to sell 500 stocks of Google at one particular date and to buy 1000 stocks of Microsoft at any time within the specified 10 days.

Listing 15.26 Creating and evaluating sample contract (F# interactive)

```
> let dfrom, dto = DateTime(2009, 4, 10), DateTime(2009, 4, 20)
  let itstocks =
     combine (give (tradeAt (DateTime(2009, 4, 15)) 500 "GOOG"))        #1
             (between dfrom dto (trade 1000 "MSFT"));;                  #1
val itstocks : Contract = CF <fun:trade@6>

> eval itstocks (DateTime(2009, 4, 14));;                               #2
val it : (int * string) list = [(1000, "MSFT")]

> eval itstocks (DateTime(2009, 4, 15));;                               #2
val it : (int * string) list = [(1000, "MSFT"); (-500, "GOOG")]
```

#1 Describe contract using the DSL
#2 Get actual trades at two distinct dates

The listing starts by creating values that represent two dates between which we're willing to purchase Microsoft stocks. Then we define a value `itstocks` that represents our contract. We're using the `combine` primitive to merge two possible trades (#1). The first one is selling of the Google stocks. One way to construct sale is to create a contract that represents a purchase of the stocks (we construct that using the `tradeAt` function that we implemented in the previous listing) and then use the `give` primitive to change purchase into a sale. This way we can create reusable trades and then use them when writing both sales and purchases.

Once we've defined the contract, we can evaluate it. The contract in our language represents a specification of trades that can occur at some specified dates, so we can evaluate it to get the possible trades at some time. As you can see in the listing, for the first date, the result is only a purchase of Microsoft stocks, but for the second date, we'll get both of the trades.

Representing contracts as abstract values

In this example, we represented contracts in a way that is quite similar to how we earlier represented behaviors. We've essentially used a function that calculates the trades and then wrote combinators that compose these functions. This is one of the two basic techniques that I mentioned in the beginning of the chapter.

When working with contracts, we could use the second technique as well and we could design a discriminated union abstractly representing the contract. It would have options that roughly correspond to the basic primitives of the domain specific language:

```
type Contract =
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
| Exchange of int * string
| After of DateTime * Contract
| Until of DateTime * Contract
| Combine of Contract * Contract
```

As you can see the type is recursive, so we can compose the elementary value `Exchange` that represents a single trade with other trades using `Combine`, limit their validity using `After` and `Until` and so on.

The difference between these two techniques is that when using abstract value representations, we can write all sorts of processing functions for the language. We could for example easily add a function that takes `Contract` value and evaluates its overall risk and so on. On the other hand, when we use a function type under the hood, we cannot observe many properties of the value once it is created and we can only execute it. In reality, it would be probably better to represent contracts using abstract values, but I wanted to demonstrate how you can use the same technique we've seen earlier in the chapter for creating a language for two different domains.

Clearly, the domain specific language that we've sketched in this section was very limited and simplistic, but it demonstrated that the approach is very powerful and that it can be used for a wide variety of problem domains. It definitely isn't limited to describing animations and financial contracts and I'm sure you already have some ideas how you could use it for solving the problems that you're concerned with.

15.7 Summary

We started the chapter by talking about the language oriented programming style and in particular about various techniques for creating internal domain specific languages. I briefly mentioned techniques like literal expressions that can be used in both F# and C#, fluent interfaces that are particularly useful in C# and combinator libraries which are used in functional programming languages.

Later, we created a language for describing animations. We divided that into two unrelated concepts - behaviors and drawings. We provided a few primitives such as `wiggle`, `time` and `circle` and operations for composing them such as overloaded operators for behaviors or `moveXY` function for drawings. Using these primitives we could compose anything we wanted, so we don't need to know anything about the underlying representation and the user of our library can just think about problems using those simple primitives. Next, we've seen that well designed libraries can be nicely composed, because we could create an animation library just by composing two unrelated concepts - behaviors and drawings. Finally, we also briefly sketched a domain specific library for a completely different problem, which is modeling of financial contracts.

In the next chapter, we'll turn our attention back to asynchronous workflows that we've seen in chapter 13, but we're going to use them differently. We'll look at developing

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

applications that react to external events including events from the user interface. In general, we'll talk about writing *reactive applications* and the F# techniques that we can use.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

16 Developing reactive functional programs

In this chapter, we're look at a few techniques for creating user interfaces and dealing with the input from the user or other external events. We'll also discuss one interesting mechanism available in F# that can be used for creating concurrent programs. This sounds like somewhat unrelated topics, but we'll see many similarities. All of the libraries and examples we'll see in this chapter share a similar architecture, so let's first briefly look at the reactive architecture in general.

When implementing imperative or functional application with the usual architecture, the code we write drives the execution of the application and controls what happens in the next step. However, for some problems such as GUI applications, this architecture doesn't work very well. For example a windows application needs to handle a large number of various user interface events; it may need to respond to a completion of asynchronous web service requests or for example to a stat update from some background computation. The execution of this type of applications is controlled by the events and the application is concerned with *reacting* to them. For this reason, this principle is sometimes called *inversion of control* and is sometimes anecdotally referred to as *The Hollywood Principle*[§§§].

The standard .NET way for writing this kind of applications is to use event handlers. However, when using event handlers, we always need some local mutable state, which

[§§§] "Don't call us, we will call you".

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

means that it is in some way against the functional principles. On the other hand, this is the most straightforward way. We've already seen how to use it in chapters showing some graphical user interface, so we won't spend a long time discussing this programming style. Instead, we'll focus at some of the appealing alternatives that F# gives us.

We will start by looking at the declarative way to handle events, which is somewhat similar to the elegant declarative list processing that we've seen in some of the early chapters. Then we'll look at using asynchronous workflows for event handling, which gives us a way to revert back the inversion of control and again write the code in a way where we control (or at least appear to control) what the application is doing. Finally, we'll look at working with state in an application like that and we'll also briefly look at message passing concurrency, which is a powerful technique for writing multi-threaded applications.

16.1 Reactive programming using events

With no doubt, you already know how to write application that reacts to events in C# and we've seen that the same technique can be used in F# as well. The usual way is register a callback function (or a method) with the event. When the event occurs, the callback function is called and it can react to the event, for example by updating the state of the application or by doing changes in its user interface.

We'll shortly see that there are other ways for handling events, but let's first review the usual style using one example. The code in listing 16.1 monitors changes in the file system using the `FileSystemWatcher` class. Once initialized, the watcher triggers an event every time some file is created, renamed or deleted.

Listing 16.1 Monitoring file system events (F#)

```
open System.IO
let w = new FileSystemWatcher("C:\\Temp", EnableRaisingEvents = true) #A

let isNotHidden(fse:RenamedEventArgs) =                                 #1
   let hidden = FileAttributes.Hidden                                   #1
   (File.GetAttributes(fse.FullPath) &&& hidden) <> hidden              #1

w.Renamed.Add(fun fse ->                                                #2
   if isNotHidden(fse) then                                             #B
      printfn "%s renamed to %s" fse.OldFullPath fse.FullPath)          #B
```

#A Initialize the watcher
#1 Test attributes of the file
#2 Register the event handler
#B Report only visible files

The listing starts by initializing the `FileSystemWathcher` object and we also set the `EnableRaisingEvents` property during the construction, to activate the monitoring. The next few lines (#1) show a simple function that checks whether a file is not marked as hidden. The argument to this function is a class derived from `EventArgs` that carries information about the event triggered by the watcher.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

The last part of the code (#2) registers an event handler that will be called when a file is renamed. In F#, events are represented in a different way than in other .NET languages. In C#, event is a special member of the class and you can work with it only by using one of the operators for adding (+=) or removing (-=) event handlers. On the other hand, in F# events appear as standard members of type IEvent<'T> where the T parameter specifies the value carried by the event (derived from EventArgs). This type has an Add method that we can use for registering a callback function. The type representing events also has AddHandler and RemoveHandler methods, so you can still use delegates if you want to be able to remove the registered callback later.

The example above uses the Add method and gives it a lambda function as an argument (#2). The function reacts by printing information about the renamed file, but we don't want to react to every event. Instead we want to display the message only when the affected file is not marked as hidden. To do this, we simply write an if condition inside the callback function.

This of course works fine, but as we'll see in the next section, F# allows us to write the filtering of events in a more declarative way, which makes the program easier to read and also gives us better ways for factoring our code. Later we'll see that the same principles can be also to some extent applied in C#.

16.1.1 Introducing event functions

Working with events by directly providing callback function isn't very declarative. We're imperatively adding the event handler and the whole behavior is wrapped inside the callback function, so let's now think how we could write the same thing in a more declarative style. We've seen that one way for making code declarative is to use higher order functions. The best examples are functions for working with lists such as List.filter. If we had a list of events from the file system watcher (called fswList), we could factor the code into two parts. The first one would filter the events to select only those that we're interested in and the second part would print the information. The first part might look something like this:

```
let renamedVisible =
   fswList |> List.filter isNotHidden
```

The snippet uses the isNotHidden function as an argument to the function that filters the list. The second part could use the List.iter function to perform printing of every item in the list.

As we'll see in the listing 16.2, we can use exactly the same pattern when working with events. We can think of events in a similar way as we think of lists. Events also carry a sequence of values, with the difference that the values are not available immediately. A new value appears every time the event is triggered. This sequence of event arguments can be filtered in a similar way as collections. We can use Event.filter function to create an event that is triggered when the source event produces a value that matches provided predicate (a function returning bool).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

Listing 16.2 Filtering events using Event.filter function (F# interactive)

```
> let renamedVisible =                                          #1
     w.Renamed |> Event.filter isNotHidden                      #1
val renamedVisible : IEvent<RenamedEventArgs>                   #2

> renamedVisible |> Event.listen (fun fse ->                    #3
     printfn "%s renamed to %s" fse.OldFullPath fse.FullPath)
val it : unit
```

#1 Filter renames of hidden files
#2 Result is a filtered event
#3 Print file name when event occurs

The first command (#1) filters the event in a similar way in which we filtered a list of values. As you can see by looking at the type of the result (#2), the function creates a new event object. The returned event listens to the event of the file system watcher and when a file is renamed, it uses the provided filtering function to test whether the value carried by the event should be ignored or not. If the filtering function returns `false`, the resulting event is triggered, otherwise the current occurrence of the event is ignored.

The next line registers a function that prints information about the renamed file with the filtered event. We're using another function for working with events called `Event.listen`. This function does the same thing as the Add method that we were using earlier, but it allows us to write the whole event processing code in a more uniform way just using higher order functions.

Before we discuss benefits of this programming style, let's look at the table 16.1, which shows several of the most important functions for working with events, including those that we've used in the previous listing. As you can see, many of them very closely correspond to a function for working with sequences.

Event function	Type of the function and description
Event.filter	`('T -> bool) -> IEvent<'TDel,'T> -> IEvent<'T>` Returns event that is triggered only when the source event occurs and when the value carried by the event matches the predicate specified as the first argument. This function corresponds to `List.filter` for lists.
Event.map	`('T -> 'R) -> IEvent<'TDel,'T> -> IEvent<'R>` Returns an event that is triggered every time the source event is triggered. The value carried by the returned event is calculated from the source value using the function given as the first argument. This corresponds to the `List.map` function.
Event.listen	`('T -> unit) -> IEvent<'TDel,'T> -> unit` Registers a callback function for the specified event. The function provided as the first argument is called whenever the event given as the second argument occurs. This function is similar to `List.iter` function for lists.
Event.scan	`('S -> 'T -> 'S) -> 'S -> IEvent<'TDel,'T> ->`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

	IEvent<'S> This function creates event with internal state. The initial state is given as the second argument and it is updated every time the source event occurs using the function given as the first argument. The returned event reports the accumulated state every time the source event is triggered and state is recomputed.
Event.merge	IEvent<'TDel1,'T> -> IEvent<'TDel2,'T> -> IEvent<'T> Creates an event that is triggered when either of the events passed as arguments occurs. Note that the type of the values carried by the events (T) has to be same for both of the events given as arguments.

Table 16.1 Overview of some interesting higher order functions for working with events

The table shows a couple of things that are worth explaining. First of all, the type representing the event used as an input for all the functions is different than the result. The input type has two type parameters. The second one is the value carried by the event and the first one (named 'TDel) is a .NET delegate used when registering handlers for the event. The result type is a special type of events that is used in F# and uses a generic delegate named Handler<'T> that is available in the F# library. This means that the IEvent<'T> type is actually just a shortcut for a type IEvent<Handler<'T>, 'T>. In F#, we'll use the simplest type most of the time and the version with two type parameters is used only when accessing delegates declared in an existing .NET type.

The Event.scan function also deserves an explanation, because it looks a bit more complicated than the others. The signature looks a bit similar to the List.fold_left function. Both of the functions take an initial state and a function that knows how to calculate a new state from the original state and an element from the list or value carried by the event. The difference is that the fold_left function returns the result of accumulating all the elements of the list. This is of course impossible for events, because we don't know when the event will happen for the last time. So, instead of waiting for the last element, the Event.scan function returns an event that is triggered every time the internal state is recalculated. We'll see an example showing how useful the function is in the next section, but let me first return to our previous example for a second.

Probably the larger benefit of using higher order functions for working with events is that we can write the handling in a more declarative way. In the previous listing, we replaced an imperative if in the body of the event handler with a declarative filtering, but we can take the example even further. If we create a function that formats the information carried by the RenamedEventArgs (called for example formatFileEvent) then we can write the whole event handling as a single very succinct expression.

Listing 16.3 Declarative event handling (F#)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
w.Renamed
   |> Event.filter isNotHidden              #1
   |> Event.map formatFileEvent             #2
   |> Event.listen (printfn "%s")           #3
```

#1 Filter renames of hidden files
#2 Create event carrying formatted strings
#3 Output the carried message

The listing 16.3 implements the same functionality as our first listing, using two helper functions and the functions from the `Event` module. Once we know we can think of event as series of values, the code should be easy to read. Instead of imperatively specifying "what to do" when event occurs, we declaratively specify aspects of the required result. The first line specifies what kind of events we are interested in (#1), the second one specifies what information is important for us (#2) and the last line gives a way for displaying the formatted information (#3).

The declarative style is one of the benefits, but this way of working with events gives us a richer way to factor the code. For example, we could omit the last line to create an event that can be used in several other places of the application. Then we could for example use `Event.listen` with `MessageBox.Show` as an argument to display the notifications in a graphical form. To become more familiar with this concept, we'll look at another slightly more complicated example in the next section.

16.1.2 Creating simple reactive application

Let's now look how we can use the processing function when writing a simple Windows Forms application. The main form of the application is displayed in the figure 16.1 and you can probably already guess what it is supposed to do.

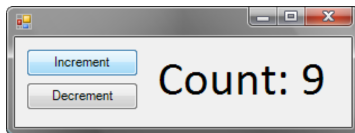

Figure 16.1 The number displayed in label is changed by clicking on the buttons.

If we implemented this application in the usual way, we'd create a mutable field (or mutable ref cell in F#). Then we'd write an event handler that would be called when either of the buttons is clicked. The event handler would test which of the buttons was clicked and it would increment or decrement the mutable state and display it on the label.

Now, how can we implement the same thing using the functions for working with events that we introduced in the previous section? One of the nice things of many declarative libraries is that the code written using them can be very nicely visualized. This is true for events as well, so you can see a diagram demonstrating our solution in figure 16.2.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

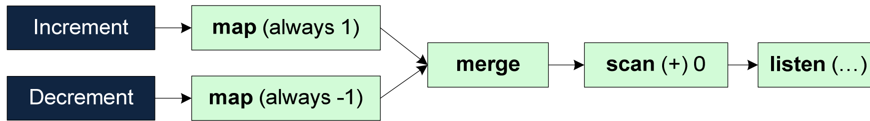

Figure 16.2 Event processing pipeline used in the sample application; boxes on the left represent source events and light boxes represent events created using processing functions.

The idea is that we'll take the click events and turn them into an event that carries an integer value. We'll do this using a helper function named `always`. It returns a function that ignores its argument and always returns the same value. We'll use it to create events that will carry either +1 or -1 depending on which of the buttons was clicked. Then we can merge these two events and use the `Event.scan` function to sum the values carried by the events.

The code needed to build the user interface isn't very interesting, so we'll look only at the part needed to setup the event processing. You can see the code that encodes the pipeline from the previous figure in the listing 16.4.

Listing 16.4 Pipeline for handling events (F#)

```
let always x = (fun _ -> x)                                 #A
let incEvt = (btnUp.Click |> Event.map (always 1))          #1
let decEvt = (btnDown.Click |> Event.map (always -1))       #1

Event.merge incEvt decEvt                                   #2
   |> Event.scan (+) 0                                      #3
   |> Event.listen (fun sum ->
      lbl.Text <- sprintf "Count: %d" sum)                  #B
```

#A Create function that always returns 'x'
#1 Create events that carry +1 or -1 values
#2 Merge the events
#3 Calculate summary of carried values
#B Display the result

To make the code more readable, we don't encode the whole pipeline as a single expression (even though it would be possible). Instead, we first declare two helper values that represent events (#1). The type of both `incEvt` and `decEvt` values is `IEvent<int>`, which means that they represent events. The value carried by the event raised by the "Increment" button is always +1 and the value of the other event is always -1. To generate the value, we're using a function `always` from the previous chapter that returns a function ignoring the argument and returning always the same value. The value that is ignored in the example above is `EventArgs` argument of the `Click` event.

As a next step, we merge these events together to create an event that will be triggered every time either of the buttons is clicked. The event carries integer values, so we can use `Event.scan` to sum the values starting with 0 as an initial value. We're using the plus

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

operator for aggregation, so in every click, the aggregation will add +1 or -1. Finally, we use the `Event.listen` function to specify a handler that displays the current sum of clicks.

The ability to work with events as if they were values of type `IEvent` is a special feature of the F# language, because F# automatically wraps .NET events into this type. Using the same principle in C# is a bit difficult, but it is possible and it nicely demonstrates the power of declarative programming style and LINQ, so we'll look at it at least briefly.

16.1.3 Declarative event processing in C# 3.0

To use events as first-class values in C#, we first need to create our own implementation of the `IEvent<T>` type for C#. This will be an interface containing two methods for doing the usual operations with events - one for attaching and one for removing an event handler. We will not discuss the full implementation in the book and we'll instead use existing project called Reactive LINQ. You can find more information about it in the series of articles starting with article *Introducing Reactive LINQ* [Petricek, 2008]. All source code needed to run the examples from this section is of course available on the book web site.

Let's now look how we could implement the demo with `FileSystemWatcher` in C#. The Reactive LINQ library gives us an `IEvent<T>` type and a couple of extension methods for doing the same things as `Event.filter`, `Event.map` and others. The library follows the standard C# naming, so the corresponding extension methods for working with event values are called `Where` and `Select`.

However, the problem that we have to workaround in C# is that events (such as `watcher.Renamed`) are not first-class values and so they cannot be passed as an argument to a method. This means that we have to first convert them into the `IEvent<T>` representation. The Reactive LINQ library provides a method `Reactive.Attach` that takes the name of the event as a string and creates an event value of type `IEvent`.

I mentioned that the methods for working with events are called `Where` and `Select`. This is very important, because it also means that we can use the syntactic sugar available in C# and use the LINQ query syntax to write the event processing code instead of calling these methods explicitly. The listing 16.5 uses Reactive LINQ to display notification when a visible file gets renamed.

Listing 16.5 Working with events using LINQ (C#)

```
var watcher = new FileSystemWatcher("C:\\Temp") {                       #A
   EnableRaisingEvents = true };
var watcherEvt = Reactive.Attach<RenamedEventArgs>                      #1
   (watcher, "Renamed");

var renamedEvt =
   from fse in watcherEvt                                               #2
   where IsNotHidden(fse)                                               #2
   select String.Format("{0} renamed to {1}",                           #2
      fse.OldFullPath, fse.FullPath);                                   #2

renamedEvt.Listen(Console.WriteLine);                                   #3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

#A Initialize the file system watcher
#1 Convert event to a value
#2 Filter events and yield string with file names
#3 Print the information when event occurs

After initializing the `FileSystemWatcher` object, we use the `Attach` method to turn the `watcher.Renamed` event into a first-class value (#1) represented using the `IEvent<RenamedEventArgs>` interface. The `Attach` method takes a single type argument that specifies the type of values carried by the event and a single argument which is the name of the event. It uses reflection under the hood, so we have to be careful to specify the name correctly.

Most of the processing code is implemented as a single LINQ query (#2) that uses a single helper method `IsNotHidden` to filter renames of hidden files. The C# compiler translates the query to ordinary calls to `Where` and `Select` extension methods, so there is nothing magical going on. Most of the code directly corresponds to what we've just seen in F#. Finally, the last line uses an extension method `Listen` to register a handler for the filtered event. We're using simply the `Console.WriteLine` method, so the string carried by the event will be printed to the screen.

In the last few sections, we've seen how to create events that are constructed from other events using higher order functions or using LINQ queries. However, we still haven't seen how to declare a new event. In C# this is done using the well known `event` keyword, but the technique used in F# differs, so we'll discuss it in the next section.

16.1.4 Declaring events in F#

When declaring a new event, we need two things. First of all, we need to create `IEvent<'T>` value that we could publish and that others could use for listening to our newly created event. As a second thing, we also need a way to trigger the event. In C#, the event can be triggered using the method invocation syntax, but only from the class where it was declared. When we create a new event in F#, we'll get a function value for triggering it.

Let's look at an example showing how this looks in practice. Probably the most common scenario for working with events is when we need to expose event as a member of some object in a similar way as for example Windows Forms controls. The listing 16.6 shows a simple concrete object type (a class) that exposes one event and one method that sometimes triggers it.

Listing 16.6. Declaring event as a class member (F# interactive)

```
> type Counter() =
     let mutable num = 0
     let ev = new Event<_>()                                              #1

     member x.SignChanged = ev.Publish                                    #2
     member x.Add(n) =
        num <- num + n
        if (sign(num - n) <> sign(num)) then
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
            ev.Trigger(num);;                                        #3

> let c = Counter()
  c.SignChanged |> Event.listen (printfn "Number: %d");;

> c.Add(10);;
Number: 10                                                           #A
> c.Add(10);;
> c.Add(-30);;
Number: -10                                                          #B
```

#1 Create a new event
#2 Publish the 'IEvent' value
#3 Trigger the event using provided member
#A Sign changed from 0 to 1
#B Sign changed to -1

The `Counter` class contains a single mutable field that stores the current number. The `Add` method can be used for changing the state of the object and we want to trigger the event `SignChanged` when the sign of the stored number changes. When declaring a new event, we use the `Event` class from the F# library. This object contains a `Publish` member that returns the corresponding `IEvent<'T>` value that can be listened to and a `Trigger` member for running the event.

The previous listing shows the typical way of working with events in a type declaration. We store the instance of the `Event` class as a local value (#1) and we expose the event value returned by the `Publish` member as a public member of the class, so that the users can listen to the event, but cannot trigger it. Finally, when the conditions of the event arise, we run it using the `Trigger` member (#3).

DECLARING C# COMPATIBLE EVENTS

The technique we've used in this section creates events that can be naturally used from F# however they won't appear as standard C# events. First of all, F# uses its own delegate type (`Handler<T>`). If you want to use some other delegate, you can create the event using a class `Event<'TDel, 'T>`, which allows you to specify the type of the delegate as the first argument. (…)

In the last few sections, we learned many things about events and we've seen some of the benefits of using events as first-class values. Most notably, the fact that we can use higher order functions for working with events. In the next section, we'll extend our example from the previous chapter with a useful function that will take event value as an argument, to demonstrate how a function like that could be designed and implemented.

16.2 Creating reactive animations

When implementing the library for creating animations in the previous chapter, I wrote that the library is largely influenced by functional reactive programming. However, we focused only on the part that implements animations, so the examples from the previous chapter couldn't react to events such as mouse clicks. Implementing a complete library for functional

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

reactive programming is outside of the scope of this book, but we can look at least at one example that shows the relation between behaviors (from the previous chapter) and events that we discussed in the previous sections. This will also show some of the possibilities that F# gives us by treating events as first-class values.

As you may remember from the previous chapter, behavior is a value that can vary in time. For example an ellipse whose location is changing depending on the time. In this section, we'll create a function named `switch`, which allows us to create behaviors that change when some external event occurs. We'll use it to create an animation that starts as a static image and becomes animating faster every time you click on the form.

16.2.1 Using the switch function

We'll start by looking at the example first and describe the implementation of the `switch` function later. As we've seen repeatedly in the previous chapters, a good way to understand what a function does is to look at its type, so let's examine the type first:

```
val switch : Behavior<'T> -> IEvent<'Del, Behavior<'T>> -> Behavior<'T>
```

The result of the function is a behavior that represents a value of `'T` varying in time. This means that the function somehow constructs a behavior using the first two arguments. The first argument represents an initial behavior. Before the event occurs, the returned behavior will be the same as the one provided as the first argument.

The most interesting thing is the second argument. It is an event that carries values of type `Behavior<'T>`. This means that every time the event is triggered, it will give us a new behavior that we can use instead of the initial behavior (or instead of the previous behavior). Every time the event occurs, the `switch` function will under the hood replace the returned behavior with the one obtained from the event. You may be thinking that event containing a behavior as a value looks a bit complicated. That's probably true, but we'll see shortly that events like this can be constructed quite easily.

If you look at the type of the `switch` function in Visual Studio or F# interactive, it will also print a `when` clause that specifies restriction for the `'Del` type. In particular, it specifies that the delegate type should take the value carried by the event (`Behavior<'T>`) as an argument, so that it's compatible with the event. However, this is just a technical detail that we don't have to worry about, because we'll use only created declared in F#.

Now that we know enough about the `switch` function, let's look how we can use it. The listing 16.7 first creates a simple rotating circle similar to those from the previous chapter. Then it constructs an event that is triggered when the user clicks on the form and carries a new behavior - the same animation running a bit faster. Finally, it uses the `switch` function to construct a behavior that's changing with every click.

Listing 16.7 Animation with changing speed (F#)

```
let af = new AnimationForm(ClientSize = Size(400, 400), Visible=true)
let greenCircle = circle (cns Brushes.OliveDrab) 100.0f.anim          #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
let rotatingCircle = rotate 100.0f 1.0f greenCircle               #A

let circleEvt =                                                   #1
  af.Click
  |> Event.map (always 0.1f)                                      #2
  |> Event.scan (+) 0.0f                                          #2
  |> Event.map (fun x -> faster x rotatingCircle)                 #3

let init = faster 0.0f rotatingCircle                             #B
af.Animation <- switch init circleEvt                             #4
```

#A Create rotating circle with a constant size
#1 Event carrying behaviors as a value
#2 Adds 0.1 to the initial speed 0.0
#3 Create a new faster animation
#B Initial animation is suspended
#4 Animation that speeds-up with clicks

The listing first creates a standard behavior `rotatingCircle` that represents an animated circle that's rotating using a constant speed. As a next step, it constructs the event that yields new behaviors (#1). We're using the same trick that we used when counting the number of clicks on a button to create an event that will yield a number specifying the speed, which increments with every click. The last call to `Event.map` in the pipeline (#3) turns the event carrying the speed into an event that carries a behavior. It changes the speed of the original rotating circle every time by calling the faster function with the new speed as an argument.

Once we have the event, we can finally use the `switch` function. First we create an initial behavior, which is the circle with the rotation speed set to zero. Then we use this behavior and the event declared earlier to create the final animation (#4). You can see how the final animation looks in the figure 16.3. The figure shows the animation after about 3 and 13 clicks.

Figure 16.3 Two forms showing the animation running using different speeds after several mouse clicks

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

Thanks to the combination of first-class events and behaviors, we can write quite interesting animations in a fully declarative way. In the next section, we'll look under the hood and we'll discuss the implementation of the `switch` function.

16.2.2 Implementing switch function

I already sketched how the `switch` function might work in the previous section, so let's now look at the full source in listing 16.8. The key idea is that the function will return a behavior that uses an actual behavior stored in a mutable variable. Every time the event occurs, we'll update the mutable variable, so the returned behavior will start behaving differently. Note that this use of mutable state is completely hidden from the user, so the code that we wrote as an end-user was declarative and free of any visible side-effects.

Listing 16.8 Implementing the switch function (F#)

```
let switch init evt =
   let current = ref init                                          #1
   evt |> Event.listen (fun arg -> current := arg)                 #2
   sample(fun ctx ->
      let (BH(f)) = !current                                       #3
      f(ctx))                                                      #3
```

#1 Store the actual behavior in a ref cell
#2 Update the behavior
#3 Get the current behavior and run it

The function first declares a mutable variable (using the F# ref cells that we discussed in chapter 8). The initial value of the ref cell is set to the initial behavior (#1). Next, we setup a handler for the event that can yield a new behavior (#2). When the event occurs, we set the value of the ref cell to the new behavior that we obtained from the event. We don't worry about the thread safety in this example, because when we use the switch function only with Windows Forms events, the state will be always accessed only from the (single) GUI thread. Finally, the behavior that's returned from the function is constructed using the `sample` primitive from the previous chapter. When the lambda function gets called to get the value of the behavior at the specified time, we simply dereference the current behavior and use it to process the request.

The functions from `Event` module are useful if the logic of the event handling isn't very complicated. If the reaction to an event is always the same and if you need to filter the event or combine it with other events, then the declarative style is very useful. However, describing more complex logic declaratively using events may not be easily possible. In the next section we'll look at another technique that uses asynchronous workflows from chapter 13 for handling of GUI events.

16.2 Programming user interface using workflows

When designing applications that don't react to external events, we have rich ways for describing the control flow of the application, such as `if-then-else` expressions, `for`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

loops and `while` loops in imperative languages or recursion and higher order functions in functional languages. Constructs like this make it very easy to describe what the application does. The control flow is clearly visible in the source code, so drawing a flowchart that describes it is a straightforward task.

Unfortunately, understanding reactive applications is much more difficult. A usual C# application or GUI control that needs to react to multiple events has some mutable state and when an event occurs, it updates the state and perhaps runs some action in response to the event, depending on the current state. In this encoding, it is quite difficult to understand what the states of the application and transitions between them are. Using asynchronous workflows, we can write the code in a way that makes the control flow of the application visible even for reactive applications.

16.2.1 Waiting for events asynchronously

The reason why we cannot use standard control flow constructs to drive reactive applications is that we don't have any way for waiting for an event to occur. Writing a function that runs in a loop and checks whether an event has occurred is difficult to implement, but more importantly, it is also a bad practice, because it would block the executing thread. As we've seen in chapter 13, asynchronous workflows allow us to write code that looks like sequential, can contain waiting for external events (such as completion of an asynchronous I/O operation), but is executed asynchronously without blocking the thread.

So far, we've seen only asynchronous methods that perform I/O operations, but there is also a primitive that stops the asynchronous workflow and resumes it when the specified event (of type `IEvent<'Del, 'T>`) occurs. The primitive is called `AwaitEvent` and is available as a member of the `Async` type. Currently, the primitive isn't a part of the F# libraries, so you can find it the full source code on the book web site. Let's start by looking at the type signature of the primitive:

```
val AwaitEvent : IEvent<'Del, 'T> -> Async<'T>
```

The type shows us that the function is quite simple. It takes event as an argument and returns a value that we can use inside an asynchronous workflow using the `let!` keyword. One important difference between events and `Async<'T>` values is that asynchronous workflow can be executed at most once, while events can be triggered multiple times. This means that the `AwaitEvent` function waits for the *first occurrence* of the event and then resumes the asynchronous workflow. Let's now look how to use this function in practice.

Counting mouse clicks

We'll start by looking at simple example that's similar to what we've seen in the beginning of the chapter and we'll implement a demo that counts the number of clicks and displays it on a label. This could be implemented using `Event.scan` and the source code would be shorter, but as we'll see later `AwaitEvent` is a far more powerful construct. You can see the source code in the listing 16.9.

Listing 16.9 Counting clicks using asynchronous workflows (F#)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
let frm, lbl = new Form(...), new Label(...)         #A

let rec loop(count) = async {                        #1
  let! args = Async.AwaitEvent(lbl.MouseDown)        #2
  lbl.Text <- sprintf "Clicks: %d" count
  return! loop(count + 1) }                          #3

do
   Async.Spawn(loop(1))                              #4
   Application.Run(frm)
```

#A Create the user interface (omitted)
#1 'Infinite' asynchronous loop
#2 Wait for the next click
#3 Loop with incremented count
#4 Start the loop without blocking

The essential part of the application that implements the counting is a single recursive function that's implemented as an asynchronous workflow (#1). The function appears to create an infinite loop, which may look suspicious for the first time. However, the construct is completely valid, because it starts by waiting for a `MouseDown` event (#2). This is done asynchronously, which means that the workflow will just install the event handler and the rest of it will be executed when you click on the label.

In the introduction, I wrote that the `AwaitEvent` primitive waits only for the first event, because asynchronous workflows can yield only a single value. As you can see in this example, if we want to handle every occurrence of the event, we can simply use recursive loop to setup the waiting again for the next occurrence. In addition, the `loop` function allows us to store the current state as parameters of the function. In fact, this way of expressing computations is very similar to primitive recursive functions that we've seen in the first sections of the book.

As I wrote earlier, the example we've just seen could be easily implemented using the `Event.scan` function, so let's look at a slightly more complicated problem now.

LIMITING THE SPEED OF CLICKS

Let's say that we'd like to limit the rate of clicks. For example, we want the count to stay the same at least for one second after it gets incremented by clicking on the label. One way for implementing this is to add another parameter to the `loop` function of type `DateTime` that will store the last time of a successful click. When the event occurs inside the loop, we could then check the difference from the current time and the last time and increase the count only when the difference is larger than the limit.

However, there is a much simpler way. In chapter 13 we implemented a primitive `Async.Sleep` that allows us to stop the workflow for a specified time. If we use it somewhere in the loop function, it will sleep for one second before reacting to the next event, which is exactly what we wanted. The method for sleeping the workflow for is also available in the F# library as an extension method for the `Thread` class (located in the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

System.Threading namespace), so everything we have to do is to add the following line before the line that last line that runs the recursion:

```
do! Thread.AsyncSleep(1000)
```

This is already something that would be quite difficult to do using the functions from the Event module. Just for curiosity, you can find the solution using Event functions in the online source code and it's about 8 lines long and a bit tricky to understand. However, the control flow of this example was still pretty simple. In the next section, we'll look at a more sophisticated example that better demonstrates the capabilities of using asynchronous workflows for GUI programming.

16.2.2 Drawing rectangles

A problem that's surprisingly difficult to solve in a functional way in F# is drawing of graphical objects on a Windows Forms form. When drawing a rectangle, the user starts by pressing the mouse button in one of the corners, then moves the cursor to the opposite corner and then releases the button. While moving the cursor with the button pressed, the application should draw the current shape of the rectangle and when the button is released it should be finally applied to a bitmap or stored in the list of vector shapes.

The usual implementation would use a mutable flag specifying whether we're currently drawing and a mutable variable for storing the last location specifying where the user pressed the mouse button. Then we'd handle MouseDown, MouseUp and MouseMove events and appropriately modify the state when one of them fires. However, if we think of the control flow of the application, we can see that it's actually quite simple. You can see a flowchart that shows it in the figure 16.4.

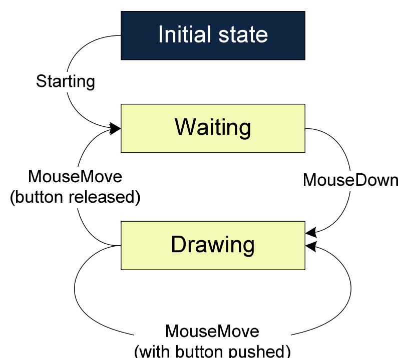

Figure 16.4 When the application is 'Waiting' we can press button to start 'Drawing'. In this state, we can either continue 'Drawing' by moving the mouse or complete the task and change the state of the application back to 'Waiting' by releasing the button.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

Before we'll look at encoding of the state machine from the figure above into actual F# program using asynchronous workflows, we'll need to write a single utility function to make the application complete.

IMPLEMENTING PROGRAM FUNDAMENTALS

We'll improve the application a little bit later, but let's start with just an empty form on which we can draw rectangles. The code in listing 16.10 shows the code necessary to create the form and a function `drawRectangle` that draws a rectangle on the form using the specified color and two of any corner points of the rectangle.

Listing 16.10 Creating user interface and drawing utility (F#)

```
open System
open System.Drawing
open System.Windows.Forms

let form = new Form(ClientSize=Size(800, 600))

let drawRectangle(clr, (x1, y1), (x2, y2)) =                           #1
   use gr = form.CreateGraphics()
   use br = new SolidBrush(clr)
   let left, top = min x1 x2, min y1 y2                                #A
   let width, height = abs(x1 - x2), abs(y1 - y2)                      #A
   gr.FillRectangle(Brushes.White, form.ClientRectangle)               #B
   gr.FillRectangle(br, Rectangle(left, top, width, height))
```

#1 Points are represented as tuples
#A Calculate upper left and lower right point
#B Clear the window using white color

The code in the listing is very straightforward. It is worth commenting that the function `drawRectangle` takes all the parameters as a tuple, so it can be used in a way that's consistent with calling .NET methods. In addition, its second and third parameters are nested tuples that represent X and Y coordinates of the point. As we'll see shortly, this makes the rest of the code a bit easier.

IMPLEMENTING THE DRAWING STATE MACHINE

Now that we have all the basics of the application, we can implement the drawing of rectangles. As we've seen in figure 16.4, the process can be represented as a state machine with two states ('Waiting' and 'Drawing') that have various transitions between them. When programming using asynchronous workflows, we can use a direct translation and create a single function for each of the states. The transitions between them can be encoded as function calls or as returning of a value from a function.

For our example this means that we'll have two functions called `drawingLoop` and `waitingLoop`. The first of them also needs to remember some state, which is done by passing parameters to the function. You can see the full source code that implements the drawing in listing 16.11.

Listing 16.11 Workflow for drawing rectangles (F#)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
let rec drawingLoop(clr, from) = async {
   let! move = Async.AwaitEvent(form.MouseMove)                           #1
   if (move.Button &&& MouseButtons.Left) = MouseButtons.Left then
      drawRectangle(clr, from, (move.X, move.Y))                          #2
      return! drawingLoop(clr, from)                                      #2
   else
      return (move.X, move.Y) }                                           #3

let waitingLoop() = async {
   while true do                                                          #4
      let! down = Async.AwaitEvent(form.MouseDown)
      let downPos = (down.X, down.Y)
      if (down.Button &&& MouseButtons.Left) = MouseButtons.Left then
         let! upPos = drawingLoop(Color.IndianRed, downPos)               #5
         do printfn "Drawn rectanlge (%A, %A)" downPos upPos }
```

#A Wait for the next mouse action
#2 Refresh rectangle and continue in the 'Drawing' state
#3 Return end location to the 'Waiting' state
#4 Repeat after drawing finishes
#5 Transition to the 'Drawing' state

The most direct way to encode the state machine would be to use only recursive calls between the two functions using the `return!` keyword. In the listing above, we did a minor change to this encoding, which makes the code a bit more readable. The `waitingLoop` function contains an infinite `while` loop (#4) that waits until the user pushes the left button and then transfers the control to the `drawingLoop` function. When it completes, it returns the end position of the rectangle (#3) and transfers the control back (#5). We can then print the information about the drawn rectangle and wait for another mouse-down event.

On the other hand, the function that's running while the user is drawing a rectangle is looping using recursive calls. It starts by waiting for the `MouseMove` event, which is also called when the button is released (#1). Then it tests whether the button is currently pressed and when that's the case, it refreshes the view of the form (#2). This transition is represented as the arc looping in the 'Drawing' state. When the button is released, it returns the last location as a result (#2), which is the transition back to the 'Waiting' state.

That's almost everything we need to run the application. The only remaining thing is to start the asynchronous workflow that handles drawing of rectangles and run the application:

```
[<STAThread>]
do Async.Spawn(waitingLoop())
   Application.Run(form)
```

In this simple application, we have only a single asynchronous workflow that handles all the interaction with the application. If we for example wanted to allow drawing of polygons by using the right mouse button, we could implement this without doing any changes to the code we wrote now. We would simply create another workflow for drawing of polygons and start it independently using `Async.Spawn`. This way of writing the user interface code gives us a very modular way for factoring the complex interactions into separate processes.

Waiting for events and the GUI thread

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

> The application we just implemented consists of a single running process, but it is important to realize that a process in the sense we're using doesn't correspond to a thread. In fact, even if we had multiple processes waiting for GUI events, the application would still be single threaded.
>
> We already discussed how asynchronous workflows work in chapter 13, but let's just briefly repeat the important point. When the workflow is waiting for an asynchronous operation it doesn't occupy any thread. Instead, it just registers a callback that will resume the workflow once the asynchronous operation completes. This means that the workflow will be executed on the thread which is used by the asynchronous operation to report that it completed. For I/O events this is a thread from the thread pool, however for GUI events, this is the GUI thread.
>
> In .NET applications (and in any Windows GUI applications in general), there is a single GUI thread that is used for processing all the incoming user interface events. For .NET this means that all the GUI events are triggered on this single thread. What does this mean for the example we just implemented? Because the only asynchronous operation we're using, the workflow will always run on the single GUI thread. This means that the technique we're using doesn't introduce any parallelism. It just gives us an easier way to write our single threaded GUI processing.
>
> We'll shortly see a technique that allows us to integrate this form of GUI processing with other processes that can possibly run in parallel however the usual code for user interface interaction like the one we've just seen should be simple and shouldn't perform any complicated computations, so there is no need for parallelism.

The code we wrote so far isn't really a drawing application, because it doesn't store the rectangles we draw in any way. Once the drawing is finished, it just prints some information to the console and forgets the rectangle. We could store a list of rectangles as a parameter of the `waitingLoop` function (if we changed it into a recursive function), but that wouldn't work very well, because the list would be private to the drawing loop and it couldn't be accessed from other parts of the application. To store the state that's global for the whole application, we need something better.

16.3 Storing state in reactive applications

The code we wrote in the previous section to handle drawing of rectangles can be viewed as a lightweight process that runs inside the application to handle certain task. In this chapter, the task was a GUI interaction, but it could as well perform asynchronous I/O operations as we've seen in chapter 13, such as download content of a web site. As I wrote earlier, there can be multiple processes like this running in parallel. Keep in mind that a process doesn't correspond to a thread, but some of the processes might be running in parallel.

Structuring the code as processes allows us to nicely factor the code, but we haven't discussed one essential aspect yet, which is how these processes can communicate. We

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

could of course use some global mutable variables, but that requires careful use of locking and it's generally discouraged in the functional programming. The technique we can use instead is called *message passing*. When writing application using message passing, the processes can send messages to each other and exchange all the needed state just by sending or replying to messages.

16.3.1 Creating mailbox processor

Let's now look what this means in practice. We'll extend the application from the previous section and we'll add a process that will store the current state of the application, which is the currently selected color (we'll add an option to change the color) and a list of all the created rectangles. It will handle messages that will be sent from the process for drawing rectangles or from other event handlers that we'll add to the application. In F#, the processes that can receive messages are also called *mailbox processors*. However, before we can start implementing the mailbox processor, we'll need to know what a message is.

IMPLEMENTING THE MESSAGE TYPE

Each process can handle messages of a single known type, so we'll start by declaring the type that represents message. As you can see in listing 16.12, discriminated union is the right F# type for this purpose.

Listing 16.12 The type representing messages (F#)

```
type RectData = Color * (int * int) * (int * int)                     #1

type DrawingMessage =
   | AddRectangle of RectData                                         #2
   | SetColor of Color                                                #2
   | GetRectangles of AsyncReplyChannel<list<RectData>>               #3
   | GetColor of AsyncReplyChannel<Color>                             #3
```

#1 Type alias for rectangle
#2 Messages for updating the state
#3 Messages for reading the state

The listing starts by declaring a type alias called `RectData` (#1) which is a tuple containing all information that we want to store about rectangles. The discriminated union itself then contains two types of messages. The first two messages (#2) are used for setting the current state and they carry the arguments for this operation. In case of `AddRectangle`, the processor will add the information about the newly created rectangle to an internal list and in case of `SetColor`, it will change the current color.

The next two messages (#3) look a bit trickier, because the value they carry has a type `AsyncReplyChannel<'T>`. This type allows us to create messages that send a reply back to the caller. In our case, it means that when the process receives one of these messages, it will send a reply back containing either a list of all rectangles in case of `GetRectangles` or the currently selected color in case of `GetColor`. We'll look at sending messages and waiting for the reply shortly, but let's start by implementing the mailbox processor.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

IMPLEMENTING THE PROCESSOR

In general, mailbox processors can be quite complicated. They can perform various calculations in reaction to the messages they receive; they can send messages to other processors and collect the replies, or they can even start new processors. However, the mailbox processor in our example is very simple and only stores the current state of the application and handles the messages to read or update the state.

The code, which you can see in listing 16.13, follows the same pattern as the code we wrote earlier. It is implemented as a recursive function written using asynchronous workflows that maintains the current state using function parameters.

Listing 16.13 Creating the mailbox processor (F#)

```
let state = MailboxProcessor.Start(fun mbox ->                        #1
   let rec loop(clr, rects) = async {
      let! msg = mbox.Receive()                                       #2
      match msg with
      | SetColor(newClr) ->
        return! loop(newClr, rects)                                   #3
      | AddRectangle(newRc) ->
        form.Invalidate()
        return! loop(clr, rects@[newRc])                              #3
      | GetColor(chnl) ->
        chnl.Reply(clr)                                               #4
        return! loop(clr, rects)
      | GetRectangles(chnl) ->
        chnl.Reply(rects)                                             #4
        return! loop(clr, rects) }
   loop(Color.IndianRed, []) )                                        #5
```

#3 and #5 appear intentionally two times; is it possible to do this?

#1 Starts by running the given function
#2 Asynchronously wait for the next message
#3 Update the state during recursive call
#4 Return the current state
#5 Start with initial color and an empty list

To create a mailbox processor, we use the `Start` member of the `MailboxProcessor` type. It initializes the mailbox for the messages and then runs the provided function (#1) to start the processing. The function returns an asynchronous workflow that can wait for messages using the `Receive` method (#2) of the mailbox that we get as an argument during the initialization.

We implemented the workflow using a recursive function called loop that takes two parameters. The parameter `clr` is the currently selected color and `rects` is a list of rectangles. When returning the workflow from the lambda function, we call the `loop` function with a red color and an empty list as the initial state (#5).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

Now, let's have a look at the body of the `loop` function. It starts by receiving the next message from the mailbox (#2). The mailbox internally stores a queue with messages, so if a message is already in the queue, the message will be returned immediately. On the other hand, if the queue is empty, the `Receive` method will block the workflow (without blocking the actual thread) and resume it once a message is sent to the processor. Once we receive a message, we use pattern matching to decide how to handle it. The first two messages just modify the state of the processor, so we recursively call the `loop` function using the `return!` keyword (#3) with the updated state. Note that when we get a new rectangle, we want to add it to the end of the list to make sure that it will be displayed on the top, so we use the @ operator for concatenating lists.

The last two messages (#4) are used for reading the state of the processor and they carry a reply channel as an argument. When the processor receives the message, it uses the `Reply` method of the channel to send the list of rectangles or the current color back as a result to the caller and then loops without altering the state.

MAILBOX PROCESSORS AND CONCURRENCY

When writing mailbox processors, it is important to understand how they're executed with respect to threads. The thread that's executing the body can change when the workflow waits for some asynchronous operation, but the body will never run on multiple threads concurrently. When a message is received during processing of the previous message, it is queued for later processing. The code we just wrote doesn't perform any complicated computations, so it will almost always process the message immediately. Thanks to this design decision we don't have to concern about any possible race conditions.

Now that we have the mailbox processor ready, we can look how to modify the rest of the application, to use and update the state stored in the processor by sending messages.

16.3.2 Communicating using messages

In the last listing of the previous section we created a mailbox processor called `state` which has a type `MailboxProcessor<DrawingMessage>`. Note that the `Start` method that we used to create it was a member of a non-generic class `MailboxProcessor`, which has a same name, but is overloaded by the number of type parameters. Before we start looking at more code, let's quickly look at the table 16.2, which shows some of the important instance methods that we can invoke on the of the mailbox processor.

Event function	Type of the function and description
Post	Sends a message to the mailbox processor without waiting until for any reply. If the mailbox processor is busy, the message is stored in the queue.
PostAndReply	Sends a message that expects `AsyncReplyChannel<'T>` as an argument to the mailbox processor and blocks the calling thread until the mailbox processor invokes the `Reply` method of the channel. Then it returns the value

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

	sent to the channel.
AsyncPostAndReply	Similar to the `PostAndReply` with the exception that it runs asynchronously. When we invoke it from an asynchronous workflow using `let!`, it doesn't block the calling thread and the result is returned asynchronously.
Receive	We used this method when creating the mailbox processor to asynchronously receive the next message from the queue, so that we can process it inside workflow. This method shouldn't be used outside of the mailbox processor.
Scan	Similarly to `Receive`, this method shouldn't be used outside of the mailbox processor. It can be used when the processor is in a state when it cannot process all types of messages, because it allows us to return `None` for messages that cannot be processed. The unprocessed messages remain in the queue for later processing.

Table 16.2 The most important methods provided by the MailboxProcessor<'Msg> type.

Note that the `Scan` and `Receive` methods should be used only from the code running inside mailbox processor. We've seen how to use `Receive` in the previous section and we'll talk about the `Scan` method briefly later in this chapter. The remaining 3 methods can be used from any thread. Sometimes you may want to write a processor that sends a message to itself, but a more typical scenario, which we'll see shortly, is when we're sending messages to the processor from outside.

IMPROVING THE DRAWING PROCESS

Let's now look at the changes that we need to do to the drawing process if we want to allow the user to change the color of rectangles before drawing and if we want to keep all the drawn rectangles on the screen. The first thing we have change a bit is the code for drawing. The drawRectangle function originally erased the screen, which isn't desirable if we want to draw multiple rectangles. After changing this behavior, we can implement a function in listing 16.14 that draws all rectangles in the given list.

Listing 16.14 Utility function for drawing rectanlges (F#)

```
let redrawWindow(rectangles) =
   use gr = form.CreateGraphics()
   gr.FillRectangle(Brushes.White, form.ClientRectangle)
   for r in rectangles do
      drawRectangle(r)
```

The function clears the content of the form and then iterates over all the elements of the given list and draws the individual rectangles using the `drawRectangle` function. Note that the list stores rectangles as tuple with three elements (color and two opposite corners), which is compatible with the tuple expected by the `drawRectangle` function.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

Now we're finally ready to modify the process that handles drawing of rectangles. As the whole code is implemented as an asynchronous workflow, we can use the asynchronous method `AsyncPostAndReply` when we need to get some information from the mailbox processor that stores the state. This is of course the preferred option when possible, because it doesn't block the calling thread. Most of the code in listing 16.15 stays the same, so I highlighted the lines that have changed.

Listing 16.15 Changes in the drawing process (F#)

```
let rec drawingLoop(clr, from) = async {
   let! move = Async.AwaitEvent(form.MouseMove)
   if (move.Button &&& MouseButtons.Left) = MouseButtons.Left then
      let! rects = state.AsyncPostAndReply(GetRectangles)              #1
      redrawWindow(rects)                                              #2
      drawRectangle(clr, from, (move.X, move.Y))                       #2
      return! drawingLoop(clr, from)
   else
      return (move.X, move.Y) }

let waitingLoop() = async {
   while true do
      let! down = Async.AwaitEvent(form.MouseDown)
      let downPos = (down.X, down.Y)
      if (down.Button &&& MouseButtons.Left) = MouseButtons.Left then
         let! clr = state.AsyncPostAndReply(GetColor)                  #3
         let! upPos = drawingLoop(clr, downPos)
         state.Post(AddRectangle(clr, downPos, upPos)) }               #4
```

#1 Get the list with existing rectangles
#2 Draw all rectangles including the new one
#3 Get the selected color
#4 Add the newly created rectangle

The first change that we have to do is in the `drawingLoop` function when updating the window to show the rectangle that the user is currently drawing. Originally, we only needed to erase the window and draw the new rectangle, but now we also need to draw all the rectangles that exist already. To do this, we obtain the list of rectangles from the mailbox processor by sending it the `GetRectangles` message (#1). The message takes an argument of type `AsyncReplyChannel<'T>` that will be used by the mailbox processor to reply to the caller, but we don't specify the channel explicitly in the code. This is possible, because the F# compiler treats the discriminated union constructor (`GetRectangles`) as a function that takes a single argument. We could write the same thing like this:

```
let! rects = state.AsyncPostAndReply(fun chnl -> GetRectangles(chnl))
```

If we write the code in this way, it is easier to see what is going on. The `AsyncPostAndReply` method creates a channel for the reply and uses the provided lambda function to create the message that carries the channel. The message is then sent to the mailbox processor and the workflow is suspended until a reply is sent to the channel. Once we receive the reply with a list of rectangles, we can draw them including the one that's being drawn right now (#2). Note that the reply can be sent on a background thread.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

This isn't a problem, because we're drawing using the `CreateGraphics` method, which doesn't have to be called from the GUI thread.

The second change we did is in the `waitingLoop` function. Once the user starts drawing the rectangle, we first read the currently selected color (#3). The color can be changed from the application user interface (we'll shortly see how), so it is important to get the color after the call to `AwaitEvent` completes. If we placed it before `AwaitEvent`, the user could change the color, but we wouldn't know that because the `AwaitEvent` primitive can block for a very long time. Once we get the color, we can call the `drawingLoop` function to handle the input of a rectangle and finally, we use the `Post` method to send all the information about the newly created rectangle to the mailbox processor (#4).

ADDING THE USER INTERFACE

The user interface of the application will be quite simple, but we'll need to call the mailbox processor from various places to work with the current application state. First of all, we'll add a handler for the `Paint` event, so the application redraws the rectangles when some part of the window is erased by Windows. Secondly, we'll add a toolbar with a single button that allows you to change the current color, so in the end you should be able to create drawings like the one in listing 16.5, which shows the running application.

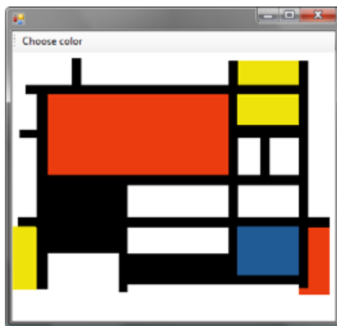

Figure 16.5 Running application with a drawing consisting only of rectangles.

You can find the code that creates the user interface in listing 16.16. The details of the code that creates the toolbar using `ToolStrip` and `ToolStripButton` controls are omitted, but you can find them in the source code available on the book web site.

Listing 16.16 Implementing the user interface (F#)

```
let tools, btnColor = new ToolStrip(...), new ToolStripButton(...)     #A

btnColor.Click.Add(fun _ ->
   use dlg = new ColorDialog()
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
      if (dlg.ShowDialog() = DialogResult.OK) then                  #B
         state.Post(SetColor(dlg.Color)) )                          #1

   form.Paint.Add(fun e ->
      let rects = state.PostAndReply(GetRectangles)                 #2
      redrawWindow(rects) )

   [<STAThread>]
   do Async.Spawn(waitingLoop())                                    #C
      Application.Run(form)
```

#A Create the GUI controls
#B Show dialog for color selection
#1 Send the selected color to mailbox processor
#2 Get a list with current rectangles
#C Start the process for drawing rectangles

Most of the code should be fairly straightforward. It creates the user interface and then registers a handler for two events. We don't need to do any filtering or other processing of the events, so we're using directly the `Add` method instead of using functions from the `Event` module. The first handler displays the `ColorDialog`, so the user can select a new color and if a color is selected, it posts a message with the new color to the mailbox processor (#1). We don't need to wait for any reply to this message, so the operation is done without blocking the thread.

The second event handler is for the `Paint` event and it needs to obtain the list with currently displayed rectangles first. To do this, we can use the `PostAndReply` method (#2), which constructs the message with a reply channel and then waits until the mailbox sends a reply. This method blocks the thread, so it should be used only rarely, in cases where we cannot complete the operation asynchronously. Updating the window of the application is definitely one of these situations, so this use is correct.

So far, we've been using the mailbox processor object directly. This is all right in the earlier phase of the development, but once the application becomes larger or if we want to turn a part of the application into a separate library, it is better to encapsulate the mailbox processor in an object. In the next section, we'll look how to do this.

16.3.3 Encapsulating mailbox processors

When encapsulating the mailbox processor, we'll change the global value representing the processor into a local field of an object and we'll add methods that send the messages to the private mailbox. This also has the benefit that we don't have to expose all of the messages if some of them are intended only for an internal use.

When doing a change like this, we don't need to modify the message processing code in any way. You can see the declaration of the concrete object type in the listing 16.17. Because the processing code stays the same, most of it is omitted in the listing.

Listing 16.17 Encapsulating mailbox processor into a type (F#)

```
   type DrawingState() =
      let mbox = MailboxProcessor.Start(fun mbox ->                #1
         let rec loop(clr, rects) = async {
```

Download at Boykma.Com


```
            let! msg = mbox.Receive()
            // Message processing code
         }
         loop(Color.Black, []) )

      member x.Setcolor(clr) =
         mbox.Post(SetColor(clr))                                   #2
      member x.AddRectangle(rc) =
         mbox.Post(AddRectangle(rc))                                #2
      member x.AsyncGetRectangles() =
         mbox.AsyncPostAndReply(GetRectangles)                      #3
      member x.AsyncGetColor() =
         mbox.AsyncPostAndReply(GetColor)                           #3
      member x.GetRectangles() =
         mbox.PostAndReply(GetRectangles)                           #4

   let state = new DrawingState()
```

#1 Private mailbox processor value
#2 Non-blocking operations without return value
#3 Asynchronous operations for reading the state
#4 Blocking call for obtaining rectangle list

To create a local mailbox processor inside the class declaration, we use a local `let` binding (#1). This becomes a part of the constructor of the class, which means that the mailbox will be started when the instance is created. Values declared using local `let` bindings are turned into local fields, so they are accessible from anywhere inside the class.

The members of the type are mostly boilerplate code. Members that update the state of the mailbox processor and don't wait for any return value (#2) send the message using the `Post` method. The second group of members (#3) that read the state is implemented using the `AsyncPostAndReply` method. Note that we're using the `Async` prefix in the name of these members. This is a standard notation used across the entire F# library to denote members that can be accessed only form asynchronous workflows. Finally, the last method (#4) is the single blocking member of the class.

Once we encapsulate the mailbox processor inside a class, we of course have to modify the rest of the code where it is accessed. Instead of sending a message explicitly, we can simply call one of the methods. The following snippet shows two sample calls from inside of an asynchronous workflow:

```
let! clr = state.AsyncGetColor()
state.Setcolor(clr)
```

We'll look at one more improvement shortly, so you'll see a few examples from the actual application in a second. Once we encapsulate the mailbox processor in a class, it can be compiled into an F# library and distributed as a reusable component. Using methods that return asynchronous workflow (the type `Async<'T>`) from C# is unfortunately difficult so if you want to create a component usable from C#, it is better to also provide methods that take a delegate as an argument and run it when the asynchronous operation completes.

In the next section we'll add one more feature to our application to show another aspect of event handling using `AwaitEvent` primitive inside asynchronous workflows.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

16.3.4 Waiting for multiple events

In all the examples of using AwaitEvent so far, we've been waiting only for a single event. The rectangle drawing application first waits for the MouseDown event and then repeatedly waits for MouseMove. However, what if we wanted to wait either for MouseMove event or for some other event that could be used to cancel the drawing?

In this section, we'll look at an example showing how to wait for multiple events. We'll keep the existing code and add the ability to cancel the drawing process. When the user hits the Esc key, we'll stop the drawingLoop returning None as the result. To do this, we need to wait for the MouseMove or the KeyDown event and handle the one that occurs first. You can find the modified code for the drawingLoop function in the listing 16.18.

Listing 16.18 Drawing rectangle with cancelation using 'Esc' key (F#)

```
let rec drawingLoop(clr, from) = async {
   let! args = Async.AwaitEvent(form.MouseMove, form.KeyDown)            #1
   match args with
   | Choice1Of2(move) when                                               #2
        (move.Button &&& MouseButtons.Left) = MouseButtons.Left ->       #2
     let! rects = state.AsyncGetRectangles()                             #A
     redrawWindow(rects)
     drawRectangle(clr, from, (move.X, move.Y))
     return! drawingLoop(clr, from)
   | Choice1Of2(move) ->                                                 #3
     return Some(move.X, move.Y)
   | Choice2Of2(key) when key.KeyCode = Keys.Escape ->                   #4
     form.Invalidate();
     return None
   | _ -> return! drawingLoop(clr, from) }                               #5
```

#1 Wait for any of the specified events
#2 Continue drawing
#A Obtain list of rectangles using method
#3 Button released, return the rectangle
#4 Esc key pressed, return 'None'
#5 Otherwise wait for another event

In all the previous examples, we used the AwaitEvent method only with a single event as an argument. However, the method is overloaded and allows us to specify multiple events. In that case, the method will wait until the first of the provided events occur and it will ignore any other occurrences. In our case, this means that the call (#1) will block until either a mouse is moved or a key is pressed and then it will run the processing code. If the processing ends with a recursive call, then the AwaitEvent will be called again to wait for the next event, but in other case, the next occurrences will be ignored.

When the AwaitEvent returns, we want to know which of the events occurred first and what argument it carried. Also, the values carried by the events can be different for all the provided events. In this situation, the method cannot simply return the carried argument, so let's look what is the type of the returned value. The F# library contains a generic discriminated union type Choice, which can represent one of several choices. The type is overloaded by the number of type parameters. In the example above, we have two

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

different choices, so the type of the args value is Choice<MouseEventArgs, KeyEventArgs>.

When the MouseMove event occurs first, the returned value will use the discriminated union constructor Choice1Of2 carrying information about the mouse event, otherwise the constructor Choice2Of2 will be used with a value of type KeyEventArgs. When waiting for multiple events, the names of the cases would be Choice1Of3 and so on.

The code that chooses how to react to the event when waiting for multiple events can be nicely written using pattern matching. The first branch (#2) is called when mouse moves while still holding the button pressed. In that case we update the window and continue drawing. If the mouse moves and the button is no longer pressed, the next case (#3) will be called. This means that the user finished drawing, so we can return the end location of the rectangle.

Finally, the last two cases specify reaction to the KeyDown event. We're again using the when clause to determine whether the pressed key is the Esc key. If that's the case, we cancel the drawing process and return None as the result, otherwise we ignore the keyboard event and continue waiting for another event. Note that we changed the return type of the function. Previously it was Async<int * int>, which is an asynchronous workflow returning a location, but now that we return either Some or None, the return type is Async<option<int * int>>. This means that we'll also have to do a minor adjustment to the waitingLoop function, so that it sends the AddRectangle message only when a rectangle is actually drawn. This is quite a simple change, so you can find it in the full source code on the book web site.

We started this section by discussing how to use mailbox processor to store the state of the application in a scenario where we need to handle various events. In all the examples, we limited ourselves only to events coming from the user interface. However, an important feature of mailbox processors is that they can be also used in scenarios involving concurrency. We'll briefly take a look at this topic in the next section.

16.4 Message passing concurrency

When talking about the development of concurrent programs in chapter 14, we focused mostly on techniques where we avoid using mutable state. Without mutable state, we can then run several parts of a computation in parallel, because they cannot interfere with each other. This works very well for many data processing problems that can be implemented in functional way, but there are also problems where the processes need to exchange information more frequently.

The most widely known solution is using the *shared memory* and protecting the access to the shared state using locks. The problem with this technique is that using locks correctly is quite difficult. You have to make sure that all the shared memory is properly locked (to avoid *race conditions* when multiple threads write to the same location). Another difficulty is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

that when acquiring locks not carefully we can cause a *deadlock*, which means that two threads become blocked, waiting for the other to complete, and can never resume.

The MailboxProcessor<'Msg> type in F# can be used for implementing concurrent programs using so called *message passing* concurrency. This approach isn't as widely known, but has been successfully used in a functional language called Erlang [Armstrong, 1996]. We've seen this approach already when storing the state of our rectangle drawing application, but we haven't in detail discussed how the technique can be used in a truly concurrent scenario.

In this section, we'll look at using mailbox processor from multiple threads to demonstrate this approach. We'll use an example with a single mailbox processor and multiple asynchronous workflows (running on multiple threads) that access it. More sophisticated programs that use message passing concurrency often use multiple mailbox processors that communicate with each other.

16.4.1 Creating state machine processor

The mailbox processor we created earlier for storing the state of the rectangle drawing application was quite simple. It was able to process 4 different messages and it maintained some local state, but regardless of the state, it was always able to process any message that it received immediately. However, this may not always be the case. For example, if a single mailbox processor sends a message to two other processors, it may need to collect the replies from these processors before reacting to any other message.

As we'll see, we can write mailbox processors that represent a state machine in a very similar way to what we used when implementing the state machine for handling events when drawing rectangles using asynchronous workflows. Let's first look at the messages that the processor will handle and then we'll talk about its possible states:

```
type Message =
   | ModifyState of int
   | Block
   | Resume
```

The mailbox processor will store an integer value and the ModifyState message can be used for updating it. For simplicity, we don't have any message for reading it and the processor will just print the number to the console every time it is updated. The two other messages are quite interesting. If the process is in the initial state and it receives Block, it stops processing all the ModifyState messages and it waits for Resume. As we'll see shortly, messages that are sent to the processor when it is in the blocked state aren't lost. The processor internally has a queue where the messages are stored, so once we resume it again, it will process all the messages it received while it was blocked.

Let's now look at the listing 16.19, which shows the implementation of the mailbox processor. Similarly to the earlier example, we're encoding the state machine using two recursive functions (using asynchronous workflows) that call each other.

Listing 16.19 Mailbox processor using state machine (F#)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com


```
let mbox = MailboxProcessor.Start(fun mbox ->
   let rec processing(n) = async {                              #1
      printfn "Processing: %d" n
      let! msg = mbox.Receive()                                 #2
      match msg with
      | ModifyState(by) -> return! processing(n + by)
      | Resume -> return! processing(n)
      | Block -> return! blocked(n) }
   and blocked(n) =                                             #3
      printfn "Blocking"
      mbox.Scan(fun msg ->                                      #4
        match msg with
        | Resume -> Some(async {                                #5
            printfn "Resuming"
            return! processing(n) })
        | _ -> None)                                            #6
   processing(0) )
```

#1 Represents the active state
#2 Process any message
#3 Represents the blocked state
#4 Only process the 'Resume' message
#5 Return workflow to continue with
#6 Other messages cannot be processed now

The implementation of the mailbox processor consists of two functions and both of them return an asynchronous workflow. The processor is started by calling the `processing` function (#1) with zero as the initial state. In this state, we can handle all the messages, so we can simply use the `Receive` primitive (#2) that asynchronously returns the next message. If the message is `ModifyState`, then we update the number and continue in the `processing` state. The `Resume` message doesn't make much sense in this state (because we haven't received the `Block` message yet), so we can ignore it. Finally, when we receive the `Block` message, we need to do something to stop processing all messages other than `Resume`, so we call the `blocked` function (#3) that represents the second state.

When the processing is blocked, we have to use the `Scan` primitive (#4), because it allows us to specify what messages we can handle and what messages should remain in the queue for later processing. The `Scan` member takes a function as an argument and the function specifies what to do when a message is received. In our example, when the message is `Resume`, we return an asynchronous workflow (#5) that the `Scan` member will run. The workflow prints a message to the console and then continues by executing the `processing` function and switching back to the active state. When the processor receives any other message in the `blocked` state, the `Scan` primitive will run the provided lambda function and will get `None` as the result. This means that it cannot process the message, so it adds the message to the queue and waits for another one.

Note that the mailbox processor as we implemented it doesn't work well in the case when we have multiple threads sending the `Block` and `Resume` messages, because if it receives a `Block` message when it's already blocked, it doesn't handle it and instead

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

continues processing once it receives the first Resume message. To solve this more properly, we'd have to handle Block messages in the blocked state and increment some number representing the count of Block messages. The Resume message would decrement it and we'd resume the processing only after the number reached zero again. However, this won't be a problem in the example we'll look at now, because we'll create only a single thread that will repeatedly block and resume the processor.

16.4.2 Accessing mailbox concurrently

The mailbox processor handles only a single message at time, but it can be safely accessed from multiple threads. All the methods for posting message to the processor (such as Post and PostAndReply) are thread-safe. Let's now look at an example showing how we can use the mailbox processor we just implemented from three different threads.

In the listing 16.20 we create two threads that repeatedly perform some computation and once they finish computing, they send a state update to the mailbox processor (in our simplified example the threads will just sleep for some time and then generate a random number). Next, we create a single thread that repeatedly sends the Block and Resume messages to the processor.

Listing 16.20 Sending messages from multiple threads (F#)

```
let modifyThread() =                                         #A
   let rnd = new Random()
   while true do
      Thread.Sleep(500)
      mbox.Post(ModifyState(rnd.Next(11) - 5))               #1

let blockThread() =
   while true do
      Thread.Sleep(2000)
      mbox.Post(Block)                                       #2
      Thread.Sleep(1000)                                     #2
      mbox.Post(Resume)                                      #2

for proc in [ blockThread; modifyThread; modifyThread ] do
   Async.Spawn(async { proc() })
```

#A Thread performing calculations
#1 Send an update to the mailbox
#2 Block the processing for one second

The code for the threads is quite simple. Both of them contain an infinite loop that would perform some computation in a real application and both of them occasionally send messages to the mailbox to synchronize. The first function uses only the ModifyState message and the second one first sends the message to block the thread, then waits for some time and then unblocks it (#2). Finally, we're using the Async.Spawn method to start executing the functions in a thread pool threads. We create a list of function values representing the processes to run and then start each of them in a for loop. Note that the list contains two times the modifyThread function, so we'll have two threads sending updates to the state.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

Let's now briefly analyze the behavior of the application when we execute it (either as a standalone application or in F# interactive). It will start by processing the incoming `ModifyState` messages for about 2 seconds. Then the blocking thread sends the `Block` message, so nothing will happen for the next 1 second. After that, the mailbox processor will be resumed and it'll process all the queued `ModifyState` messages, so it'll almost immediately update the state several times. Then it'll continue running, processing messages as they arrive for the next 2 seconds until the next `Block` message is received.

Even though this example doesn't implement any particularly useful behavior, it should give you a pretty good idea how to use mailbox processors in a real-world application that needs to synchronize the state using message passing concurrency.

16.5 Summary

In this chapter, we covered various aspects of development of reactive applications in the functional style. We started by talking about first-class events in F#, which is the ability to use event as a standard value that can be passed as an argument or returned from a function. This allows us to use higher order functions (such as `Event.filter` or `Event.map`) when writing code that processes events, which in turn makes the code more declarative in a same way as processing of lists using higher order functions or LINQ queries. The relation with LINQ is quite interesting and we briefly mentioned that in principle, we could use LINQ queries in C# for event processing as well.

However, for more dynamic types of behavior, the declarative programming using higher order functions doesn't work that well, so we looked at another technique. We've seen that we can use workflows introduced chapter 13 for asynchronously waiting until an event occurs, which allows us to write complex event handling without the inversion of control, which means that the control flow is managed by our application. This also makes it much easier to encode control structures where the process can transition between several states, because the code directly corresponds to a state machine diagram that you may draw.

Finally, we faced the problem how to store state in an application that is encoded asynchronous workflows that handle GUI events. We've seen that this can be done by using message passing techniques and we introduced the `MailboxProcessor<'T>` type that implements this programming model in F#. This type can be also used in concurrent scenarios, so we wrapped up with an example showing how to use it from multi-threaded application.

Unfortunately, most of the examples we've seen in this chapter rely on asynchronous workflows, so they can't be directly implemented in C#. In the chapter 13 I mentioned various projects that attempt to bring similar concepts to C# such as the Concurrency and Coordination Runtime [Richter, 2006], but none of them provides the same clarity as F#. The message passing concurrency techniques from the later of the chapter exist in many different forms. The implementation that's available in F# is very close to the Erlang style message passing (see for example Concurrent Programming in Erlang [Armstrong, 1996]),

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460

Download at Boykma.Com

but there are other alternatives. One of them is also available as a library for C# 2.0, so you can also take a look at the Joins Concurrency Library [Russo, 2007].

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=460