
# *Creating Primitive Workflows*

**By Tomas Petricek and Jon Skeet, authors of *Real-World Functional Programming***

*Writing the code that performs asynchronous operations without blocking the calling thread is essential to avoid problems. In F#, this is largely simplified thanks to asynchronous workflows. In this article, based on chapter 13 of Real-World Functional Programming, the authors shows you how to create your own primitive workflow.*

You may also be interested in…

The F# PowerPack library contains asynchronous versions for many important I/O operations, but it can't include all of them. For that reason, the F# library also provides methods for building your own primitive workflows. If the operation you want to run inside the workflow uses a standard .NET pattern and provides `BeginOperation` and `EndOperation` methods, you can use the `Async.FromBeginEnd` method. If you give it these two methods as an argument, it'll return an asynchronous workflow.

Other operations are available that can be executed without blocking the thread. For example, we may want to wait for a particular event to occur and continue executing the workflow when it's triggered. Listing 1 creates a primitive that waits for the specified number of milliseconds using a timer and then resumes the workflow.

**Listing 1 Implementing asynchronous waiting (F# Interactive)**

```
> module MyAsync =
    let Sleep(time) =  #1
      Async.FromContinuations(fun (cont, econt, ccont) ->
        let tmr = new System.Timers.Timer(time, AutoReset = false)
        tmr.Elapsed.Add(fun _ -> cont())  #2
        tmr.Start()
    );;
(...)

> Async.RunSynchronously(async {
    printfn "Starting..."
    do! MyAsync.Sleep(1000.0)  #3
    printfn "Finished!"
  });;
Starting...
Finished!
val it : unit = ()
```

**#1 Suspends workflow for specified time B**
**#2 Resumes computation C**
**#3 Waits without blocking thread**

The same functionality is already available in the F# library, so it isn't only a toy example. It's implemented by `Async.Sleep`. Of course, we could block the workflow using the synchronous version, `Thread.Sleep`, but there's an important difference. This method would block the thread while our function creates a timer and returns the thread to the .NET thread pool. This means that when we use our primitive, the .NET runtime can execute workflows in parallel without any limitations.

The `Sleep` function (#1) takes the number of milliseconds for which we want to delay processing and uses the `Async.FromContinuations` method to construct the workflow.

This method reflects the internal structure of the workflow quite closely. The argument is a lambda function that will be executed when the workflow starts. The lambda takes a tuple of three continuations as an argument. The first function should be called when the operation completes successfully, and the second should be called when the operation throws an exception. Similarly to the declaration of the `Async<'T>` type from an earlier sidebar, there's a third continuation that can trigger cancellation of the workflow. In the body of the lambda, we create a timer and specify the handler for its `Elapsed` event. The handler simply runs the success continuation (#2).

Having created our new primitive, listing 1 shows a simple snippet that uses it. Because it returns a unit value, we're using the `do!` primitive rather than `let!` (#3). When the code is executed, it constructs the timer with the handler and starts it.

When the specified time elapses, the system takes an available thread from the thread pool and runs the event handler, which in turn executes the rest of the computation (in our case, printing to the screen).

### Asynchronous workflows in C#

There have been numerous attempts to simplify asynchronous programming in C#, but none of the available libraries works quite as neatly as the asynchronous workflow syntax. The F# syntax is extremely simple from the end-user point of view (just wrap the code in an `async` block), which is quite difficult to achieve in C#.

We've seen that LINQ queries roughly correspond to F# computation expressions, so you might be tempted to implement `Select` and `SelectMany` operations. In principle, it would be possible to write asynchronous operations using query expressions, but the syntax we can use inside queries is limited. Interestingly, C# iterators can be also used for this purpose. This approach is described in the article "Asynchronous Programming in C# Using Iterators" (available at http://tomasp.net/blog/csharpasync.aspx). The most real-world library that uses this technique is Jeffrey Richter's PowerThreading library [Richter, 2009].

One of the most complex libraries based on C# iterators is the Concurrency and Coordination Runtime (CCR) [Chrysanthakopoulos and Singh, 2005]. This library was developed as part of Microsoft Robotics studio, where responsiveness and asynchronous processing is essential for any application. You can find more information about this library in Jeffery Richter's "Concurrency Affairs" article [Richter, 2006].

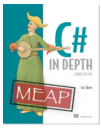
## Summary

Asynchronous workflows can be used for efficiently implementing I/O and other time consuming operations without blocking the caller thread and wasting resources. In this quick article, we showed you how to create a primitive workflow.

**Here are some other Manning titles you might be interested in:**


Dependency Injection in .NET
Mark Seemann


Windows Phone 7 in Action
Timothy Binkley-Jones, Massimo Perga and Michael Sync




C# in Depth, Third Edition
Jon Skeet

Last updated: 8/2/13