Organizing code and making it available for people and programs to use is a key part of making the best use of F#. In this book, you’ve already seen many of the constructs to help do this: functions, objects, type definitions, modules, namespaces, and assemblies. In some cases, however, you’ve encountered these only peripherally when using the libraries that come with F#. This chapter covers these constructs from the perspective of code organization and encapsulation.

Packaging code has four distinct but related meanings:

- **Organizing** code into sensible entities using namespaces, types, and modules.
- **Encapsulating** internal data structures and implementation details by making them private.
- **Assembling** code and data as one component, which for F# is called an *assembly*. An assembly is code packaged together with supporting resources as a single, logical unit of deployment.
- **Deploying** one or more assemblies, for example as a Web application or using a Web-based community packaging mechanism, such as NuGET.

The first two of these topics are associated with the F# language, and the last is more associated with the pragmatics of deploying, installing, configuring, and maintaining software. The third lies in between, because a .NET assembly can act as both a unit of encapsulation and a unit of deployment. In Chapter 19, we also consider some of the different kinds of software you may write with F# and how you can organize and package your code for these different cases.

**Hiding Things**

In all kinds of software, it’s common to hide implementation details of data structures and algorithms behind *encapsulation boundaries*. Encapsulation is a fundamental technique when writing software and is possibly the most important idea associated with object-oriented programming.

For this book’s purposes, *encapsulation* means hiding implementation details behind well-defined boundaries. This lets you enforce consistency properties and makes the structure of a program easier to manage. It also lets an implementation evolve over time. A good rule of thumb is to hide anything you don’t want used directly by client code.
Later, this chapter explains how encapsulation applies when you’re building assemblies, frameworks, and applications. In the extreme, you may even be ensuring that your code is secure when used in partial trust mode—in other words, that it can’t be inadvertently or deliberately used to achieve malicious results when used as a library by code that doesn’t have full permissions. The most important kind of encapsulation, however, is the day-to-day business of hiding the internal implementation details of functions, objects, types, and modules. The primary techniques used to do this are:

- Local definitions
- Accessibility annotations
- Explicit signatures

We cover the first two of these techniques next, and we cover explicit signatures in “Hiding Things with Signatures” later in this chapter.

**Hiding Things with Local Definitions**

The easiest way to hide definitions is to make them local to expressions or constructed class definitions using inner `let` bindings. These aren’t directly accessible from outside their scope. This technique is frequently used to hide state and other computed values inside the implementations of functions and objects. Let’s begin with a simple example. Here is the definition of a function that incorporates a single item of encapsulated state:

```fsharp
let generateTicket =
  let count = ref 0
  (fun () -> incr count; !count)
```

If you examine this definition, you see that the `generateTicket` function isn’t defined immediately as a function; instead, it first declares a local element of state called `count` and then returns a function value that refers to this state. Each time the function value is called, `count` is incremented and dereferenced, but the reference cell is never published outside the function implementation, and it is thus encapsulated.

Encapsulation through local definitions is a particularly powerful technique in F# when used in conjunction with object expressions. For example, Listing 7-1 shows the definition of an object interface type called `IPeekPoke` and a function that implements objects of this type using an object expression.

**Listing 7-1. Implementing Objects with Encapsulated State**

```fsharp
type IPeekPoke =
  abstract member Peek : unit -> int
  abstract member Poke : int -> unit

let makeCounter initialState =
  let state = ref initialState
  {new IPeekPoke with
    member x.Poke n = state := !state + n
    member x.Peek() = !state}
```

The type of the function `makeCounter` is:

```fsharp
val makeCounter : initialState:int -> IPeekPoke
```