CHAPTER 4

Understanding Types

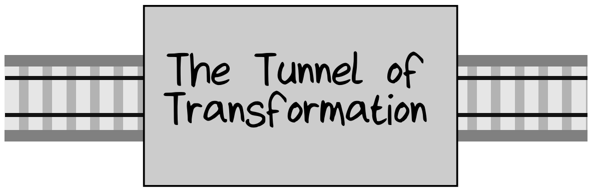
In the second chapter, we captured the domain-driven requirements for a single workflow of the order-taking system. The next challenge is to convert those informal requirements into compilable code.

The approach we are going to take is to represent the requirements using F#'s "algebraic type system." In this chapter we'll learn what algebraic types are, how they are defined and used, and how they can represent a domain model. Then, in the next chapter, we'll use what we've learned to accurately model the order-placing workflow.

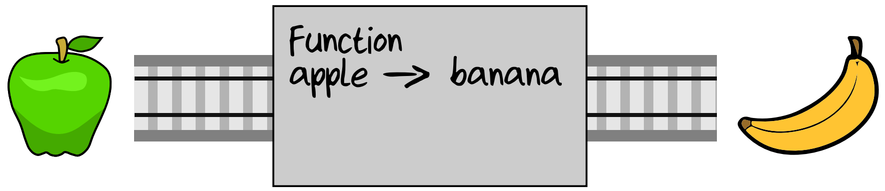
Understanding Functions

Before we can understand types, we need to understand the most basic concept in functional programming—a function.

If you remember your high-school mathematics, a function is a kind of black box with an input and an output. You can imagine it as a bit of railroad track, with a Tunnel of Transformation sitting on it. Something goes in, is transformed somehow, and comes out the other side.



For example, let's say that this particular function turns apples into bananas. We describe a function by writing down the input and output, separated by an arrow, as shown in the figure on page 60.



Type Signatures

The apple -> banana description is called a *type signature* (also known as a function signature). This particular signature is simple, but type signatures can get very complicated. Understanding and using type signatures is a critical part of coding with F#, so let's make sure we understand how they work.

Here are two functions: add1 adds 1 to its single input x, and add adds its two inputs, x and y:

```
let add1 x = x + 1   // signature is: int -> int
let add x y = x + y  // signature is: int -> int -> int
```

As you can see, the let keyword is used to define a function. The parameters are separated by spaces, without parentheses or commas. Unlike C# or Java, there is no return keyword. The last expression in the function definition is the output of the function.

Even though F# cares about the types of the inputs and outputs, you rarely need to explicitly declare what they are, because in most cases the compiler will infer the types for you automatically.[1]

- For add1, the inferred type of x (before the arrow) is int and the inferred type of the output (after the arrow) is also int, so the type signature is int -> int.
- For add, the inferred type of x and y is int and the inferred type of the output (after the last arrow) is also int. add has two parameters, and each parameter is separated by an arrow, so the type signature is int -> int -> int.

If you are using an IDE such as Visual Studio, hovering over the definition of a function will show you its type signature, but since this is a book, we'll put the type signature in a comment above the definition when we need to make it clear. It's just a comment and isn't used by the compiler.

Functions that consist of more than one line are written with an indent (like Python). There are no curly braces. Here's an example:

1. https://fsharpforfunandprofit.com/posts/type-inference/

```
// squarePlusOne : int -> int
let squarePlusOne x =
  let square = x * x
  square + 1
```

This example also shows that you can define subfunctions within a function (let square = ...) and again, that the last line (square + 1) is the return value.

Functions with Generic Types

If the function will work with *any* type, then the compiler will automatically infer a *generic* type, as in this areEqual function.

```
// areEqual : 'a -> 'a -> bool
let areEqual x y =
  (x = y)
```

For areEqual the inferred type of x and y is 'a. A tick-then-letter is F#'s way of indicating a generic type. And it's true, since x and y could be *any* type as long as they are the *same* type.

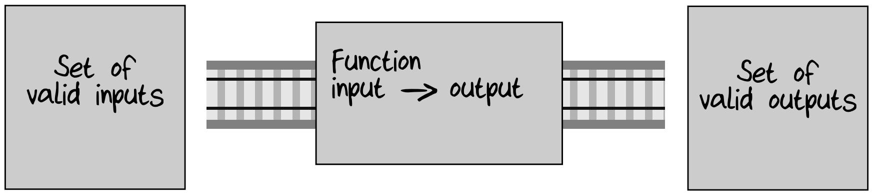
And by the way, this code shows that the equality test is = in F#, not == like in C-like languages. For comparison, the code for areEqual in C#, using generics, might look something like this:

```
static bool AreEqual<T>(T x, T y)
{
  return (x == y);
}
```

Types and Functions

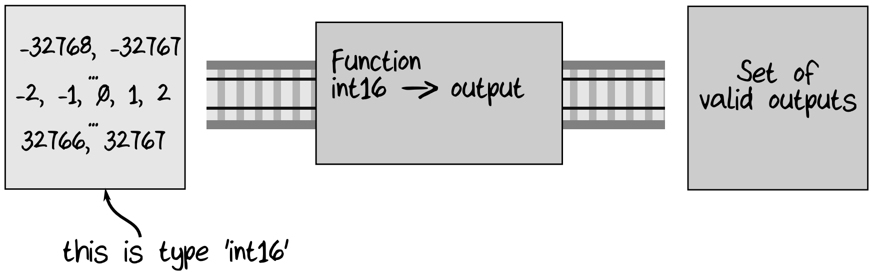
In a programming language like F#, types play a key role, so let's look at what a functional programmer means by *type*.

A *type* in functional programming is not the same as a *class* in object-oriented programming. It's much simpler. In fact, a type is just the name given to the set of possible values that can be used as inputs or outputs of a function:



For example, we might take the set of numbers in the range -32768 to +32767 and give them the label int16. There is no special meaning or behavior to a type beyond that.

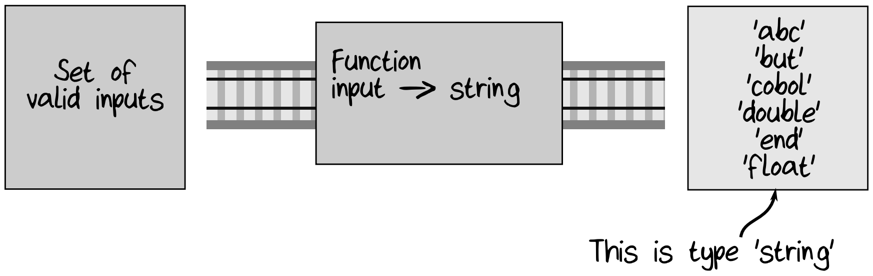
Here is an example of a function with an int16 input:



The type is what determines the function's signature, so the signature for this function might look like this:

```
int16 -> someOutputType
```

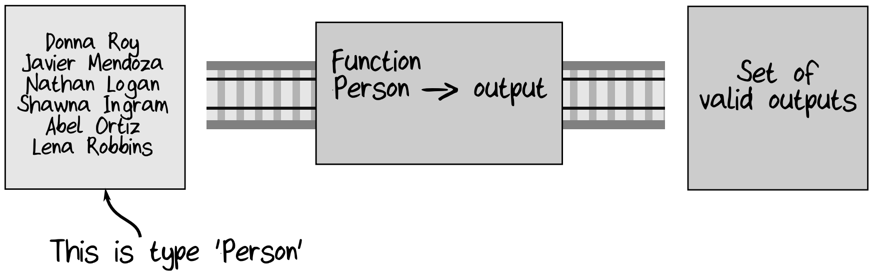
Here is an example of a function with an output consisting of the set of all possible strings, which we will call the string type:



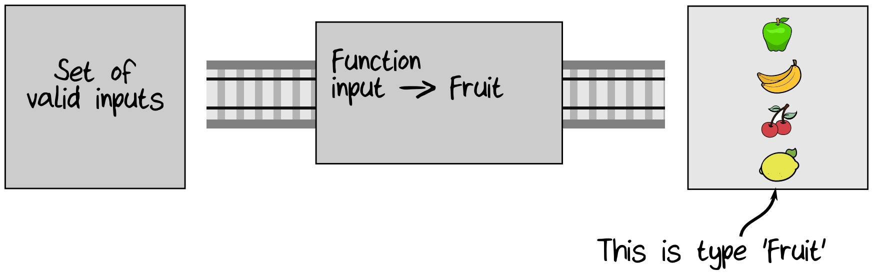
The signature for this function would be this:

```
someInputType -> string
```

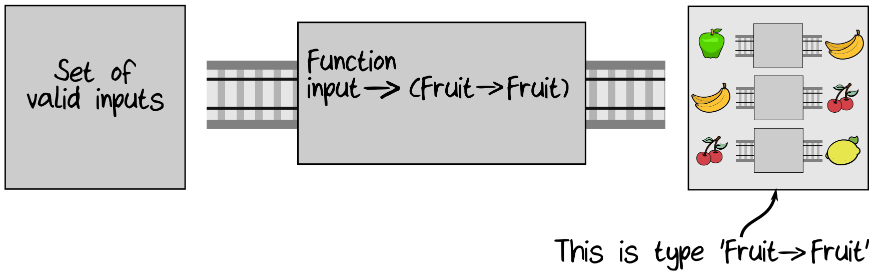
The set of things in a type do not have to be primitive objects. For example, we may have a function that works with a set of objects that collectively we call Person:



From a conceptual point of view, the things in the type can be *any* kind of thing, real or virtual. The figure on page 63 shows a function that works with "Fruit." Whether these are real fruit or a virtual representation isn't important right now.



And finally, functions are things too, so we can use sets of functions as a type as well. The function below outputs something that is a Fruit-to-Fruit function:



Each element in the output set is a `Fruit -> Fruit` function, so the signature of the function as a whole is this:

```
someInputType -> (Fruit -> Fruit)
```

Jargon Alert: "Values" vs. "Objects" vs. "Variables"

In a functional programming language, most things are called "values." In an object-oriented language, most things are called "objects." So what is the difference between a "value" and an "object"?

A value is just a member of a type, something that can be used as an input or an output. For example, `1` is a value of type `int`, `"abc"` is a value of type `string`, and so on.

Functions can be values too. If we define a simple function such as `let add1 x = x + 1`, then `add1` is a (function) value of type `int->int`.

Values are immutable (which is why they are not called "variables"). And values do not have any behavior attached to them, they are just data.

In contrast, an object is an encapsulation of a data structure *and* its associated behavior (methods). In general, objects are expected to have state (that is, be mutable), and all operations that change the internal state must be provided by the object itself (via "dot" notation).

So in the world of functional programming (where objects don't exist), you should use the term "value" rather than "variable" or "object."

Composition of Types

You'll hear the word "composition" used a lot in functional programming—it's the foundation of functional design. Composition just means that you can combine two things to make a bigger thing, like using Lego blocks.

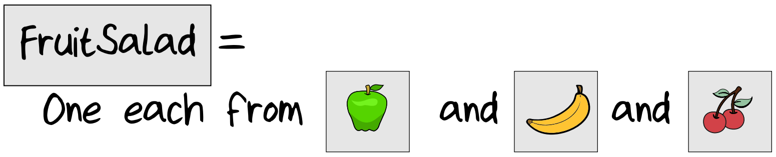
In the functional programming world, we use composition to build new functions from smaller functions and new types from smaller types. We'll talk about composition of types right now, and we'll talk about function composition later, in Chapter 8, *Understanding Functions*, on page 147.

In F#, new types are built from smaller types in two ways:

- By _AND_ing them together
- By _OR_ing them together

"AND" Types

Let's start with building types using *AND*. For example, we might say that to make fruit salad you need an apple *and* a banana *and* some cherries:



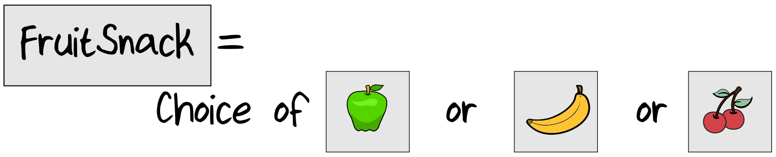
In F# this kind of type is called a *record*. Here's how the definition of a FruitSalad record type would be written in F#:

```
type FruitSalad = {
  Apple: AppleVariety
  Banana: BananaVariety
  Cherries: CherryVariety
}
```

The curly braces indicate that it is a record type, and the three fields are Apple, Banana, and Cherries.

"OR" Types

The other way of building new types is by using *OR*. For example, we might say that for a fruit snack you need an apple *or* a banana *or* some cherries:



These kinds of "choice" types will be incredibly useful for modeling (as we will see throughout this book). Here is the definition of a FruitSnack using a choice type:

```
type FruitSnack =
  | Apple of AppleVariety
  | Banana of BananaVariety
  | Cherries of CherryVariety
```

A choice type like this is called a *discriminated union* in F#. It can be read like this:

- A FruitSnack is either an AppleVariety (tagged with Apple) *or* a BananaVariety (tagged with Banana) *or* a CherryVariety (tagged with Cherries).

The vertical bar separates each choice, and the tags (such as Apple and Banana) are needed because sometimes the two or more choices may have the same type and so tags are needed to distinguish them.

The varieties of fruit are themselves defined as *OR* types, which in this case is used similarly to an enum in other languages.

```
type AppleVariety =
  | GoldenDelicious
  | GrannySmith
  | Fuji

type BananaVariety =
  | Cavendish
  | GrosMichel
  | Manzano

type CherryVariety =
  | Montmorency
  | Bing
```

This can be read as:

- An AppleVariety is either a GoldenDelicious *or* a GrannySmith *or* a Fuji,

and so on.

Jargon Alert: "Product Types" and "Sum Types"

The types that are built using *AND* are called *product types*.

The types that are built using *OR* are called *sum types* or *tagged unions* or, in F# terminology, *discriminated unions*. In this book I will often call them *choice types*, because I think that best describes their role in domain modeling.

Simple Types

We will often define a choice type with only *one* choice, such as this:

```
type ProductCode =
  | ProductCode of string
```

This type is almost always simplified to this:

```
type ProductCode = ProductCode of string
```

Why would we create such a type? Because it's an easy way to create a "wrapper"—a type that contains a primitive (such as a string or int) as an inner value.

We'll be seeing a lot of these kinds of types when we do domain modeling. In this book I will label these single-case unions as "simple types," as opposed to compound types like records and discriminated unions. More discussion of them is available in the section on Simple Types on page 79.

Algebraic Type Systems

Now we can define what we mean by an "algebraic type system." It's not as scary as it sounds—an algebraic type system is simply one where every compound type is composed from smaller types by *AND*-ing or *OR*-ing them together. F#, like most functional languages (but unlike OO languages), has a built-in algebraic type system.

Using *AND* and *OR* to build new data types should feel familiar—we used the same kind of *AND* and *OR* to document our domain. We'll see shortly that an algebraic type system is indeed an excellent tool for domain modeling.

Working with F# Types

In F#, the way that types are defined and the way that they are constructed are very similar.

For example, to define a record type, we use curly braces and then name:type definitions for each field, like this:

```
type Person = {First:string; Last:string}
```

To construct a value of this type, we use the same curly braces but use = to assign a value to a field, like this:

```
let aPerson = {First="Alex"; Last="Adams"}
```

And to deconstruct a value of this type using pattern matching, we use the same syntax but this time on the *left* side of the equation, like this:

```
let {First=first; Last=last} = aPerson
```

This code says that the values first and last will be set to the corresponding fields in the record. With records, we can also use the more familiar dot syntax as well. So the code above is equivalent to this:

```
let first = aPerson.First
let last = aPerson.Last
```

The symmetry between construction and deconstruction applies to discriminated unions as well. To define a choice type, we use the vertical bar to separate each choice, with each choice defined as caseLabel of type, like this:

```
type OrderQuantity =
  | UnitQuantity of int
  | KilogramQuantity of decimal
```

A choice type is constructed by using any one of the case labels as a constructor function, with the associated information passed in as a parameter, like this:

```
let anOrderQtyInUnits = UnitQuantity 10
let anOrderQtyInKg = KilogramQuantity 2.5
```

Cases are *not* the same as subclasses—UnitQuantity and KilogramQuantity are not types themselves, just distinct cases of the OrderQuantity type. In the example above, both these values have the *same* type: OrderQuantity.

To deconstruct a choice type, we must use pattern matching (the match..with syntax) with a test for each case, like this:

```
let printQuantity aOrderQty =
  match aOrderQty with
  | UnitQuantity uQty ->
    printfn "%i units" uQty
  | KilogramQuantity kgQty ->
    printfn "%g kg" kgQty
```

As part of the matching process, any data associated with a particular case is also made available. In the example above, the uQty value will be set if the input matches the UnitQuantity case.

Here's the result of the pattern matching when we pass in the two values we defined above:

```
printQuantity anOrderQtyInUnits // "10 units"
printQuantity anOrderQtyInKg    // "2.5 kg"
```

Building a Domain Model by Composing Types

A composable type system is a great aid in doing domain-driven design because we can quickly create a complex model simply by mixing types together in different combinations. For example, say that we want to track payments for an e-commerce site. Let's see how this might be sketched out in code during a design session.

First, we start with some wrappers for the primitive types, such as CheckNumber. These are the "simple types" we discussed above. Doing this gives them meaningful names and makes the rest of the domain easier to understand.

```
type CheckNumber = CheckNumber of int
type CardNumber = CardNumber of string
```

Next, we build up some low-level types. A CardType is an *OR* type—a choice between Visa *or* Mastercard, while CreditCardInfo is an *AND* type, a record containing a CardType *and* a CardNumber:

```
type CardType =
  Visa | Mastercard        // 'OR' type

type CreditCardInfo = {    // 'AND' type (record)
  CardType : CardType
  CardNumber : CardNumber
  }
```

We then define another *OR* type, PaymentMethod, as a choice between Cash or Check or Card. This is no longer a simple "enum" because some of the choices have data associated with them: the Check case has a CheckNumber and the Card case has CreditCardInfo:

```
type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo
```

We can define a few more basic types, such as PaymentAmount and Currency:

```
type PaymentAmount = PaymentAmount of decimal
type Currency = EUR | USD
```

And finally, the top-level type, Payment, is a record containing a PaymentAmount *and* a Currency *and* a PaymentMethod:

```
type Payment = {
  Amount : PaymentAmount
  Currency:  Currency
  Method:  PaymentMethod
  }
```

So there you go. In about 25 lines of code, we have defined a pretty useful set of types already.

Of course, there is no behavior directly associated with these types because this is a functional model, not an object-oriented model. To document the actions that can be taken, we instead define types that represent functions.

So, for example, if we want to show there is a way to use a Payment type to pay for an unpaid invoice, where the final result is a paid invoice, we could define a function type that looks like this:

```
type PayInvoice =
  UnpaidInvoice -> Payment -> PaidInvoice
```

Which means this: Given an UnpaidInvoice and then a Payment, we can create a PaidInvoice.

Or, to convert a payment from one currency to another:

```
type ConvertPaymentCurrency =
  Payment -> Currency -> Payment
```

where the first Payment is the input, the second parameter (Currency) is the currency to convert to, and the second Payment—the output—is the result after the conversion.

Modeling Optional Values, Errors, and Collections

While we are discussing domain modeling, let's talk about some common situations and how to represent them with the F# type system, namely:

- Optional or missing values
- Errors
- Functions that return no value
- Collections

Modeling Optional Values

The types that we have used so far—records and choice types—are not allowed to be null in F#. That means that every time we reference a type in a domain model, it's a *required* value.

So how can we model missing or optional data?

The answer is to think about what missing data means: it's either present or absent. There's something there, or nothing there. We can model this with a choice type called Option, defined like this:

```
type Option<'a> =
  | Some of 'a
  | None
```

The Some case means that there is data stored in the associated value 'a. The None case means there is no data. Again, the tick in 'a is F#'s way of indicating a generic type—that is, the Option type can be used to wrap *any* other type. The C# or Java equivalent would be something like Option<T>.

You don't need to define the Option type yourself. It's part of the standard F# library, and it has a rich set of helper functions that work with it.

To indicate optional data in the domain model then, we wrap the type in Option<..>, just as we would in C# or Java. For example, if we have a PersonalName type and the first and last names are required but the middle initial is optional, we could model it like this:

```
type PersonalName = {
  FirstName : string
  MiddleInitial: Option<string>  // optional
  LastName : string
  }
```

F# also supports using the option label *after* the type, which is easier to read and more commonly used:

```
type PersonalName = {
  FirstName : string
  MiddleInitial: string option
  LastName : string
  }
```

Modeling Errors

Let's say we have a process with a possible failure: "The payment was made successfully, or it failed because the card has expired." How should we model this? F# does support throwing exceptions, but we'll often want to *explicitly* document in the type signature the fact that a failure can happen. This calls out for a choice type with two cases, so let's define a type Result:

```
type Result<'Success,'Failure> =
  | Ok of 'Success
  | Error of 'Failure
```

We'll use the Ok case to hold the value when the function succeeds and the Error case to hold the error data when the function fails. And of course we want this type to be able to contain any kind of data, hence the use of generic types in the definition.

If you are using F# 4.1 and above (or Visual Studio 2017), then you don't need to define the Result type yourself, since it's part of the standard F# library. If you are using an earlier version of F#, you can easily define it and its helper functions in a few lines.

To indicate that a function can fail, we wrap the output with a Result type. For example, if the PayInvoice function could fail, then we might define it like this:

```
type PayInvoice =
  UnpaidInvoice -> Payment -> Result<PaidInvoice,PaymentError>
```

This shows that the type associated with the Ok case is PaidInvoice and the type associated with the Error case is PaymentError. We could then define PaymentError as a choice type with a case for each possible error:

```
type PaymentError =
  | CardTypeNotRecognized
  | PaymentRejected
  | PaymentProviderOffline
```

This approach to documenting errors will be covered in detail in Chapter 10, *Implementation: Working with Errors*, on page 191.

Modeling No Value at All

Most programming languages have a concept of void, used when a function or method returns nothing.

In a functional language like F#, every function must return *something*, so we can't use void. Instead we use a special built-in type called unit. There is only one value for unit, written as a pair of parentheses: ().

Let's say that you have a function that updates a customer record in a database. The input is a customer record, but there's no useful output. In F#, we would write the type signature using unit as the output type, like this:

```
type SaveCustomer = Customer -> unit
```

(In practice it would be more complex than this, of course! See Chapter 12, *Persistence*, on page 239, for a detailed discussion of working with databases.)

Alternatively, let's say you have a function that has no input yet returns something useful, such as a function that generates random numbers. In F#, you would indicate "no input" with unit as well, like this:

```
type NextRandom = unit -> int
```

When you see the unit type in a signature, that's a strong indication that there are side effects. Something somewhere is changing state, but it's hidden from you. Generally, functional programmers try to avoid side effects, or at least limit them to restricted areas of code.

Modeling Lists and Collections

F# supports a number of different collection types in the standard libraries:

- list is a fixed-size immutable collection (implemented as a linked list).
- array is a fixed-size mutable collection, where individual elements can be fetched and assigned to by index.
- ResizeArray is a variable size array. That is, items can be added or removed from the array. It is the F# alias for the C# List<T> type.
- seq is a lazy collection, where each element is returned on demand. It is the F# alias for the C# IEnumerable<T> type.
- There are also built-in types for Map (similar to Dictionary) and Set, but these are rarely used directly in a domain model.

For domain modeling, I suggest always using the list type. Just like option, it can be used as a suffix after a type (which makes it very readable), like this:

```
type Order = {
  OrderId : OrderId
  Lines : OrderLine list // a collection
  }
```

To create a list, you can use a list literal, with square brackets and semicolons (not commas!) as separators:

```
let aList = [1; 2; 3]
```

or you can prepend a value to an existing list using the :: (also known as "cons") operator:

```
let aNewList = 0 :: aList  // new list is [0;1;2;3]
```

To deconstruct a list in order to access elements in it, you use similar patterns. You can match against list literals like this:

```
let printList1 aList =
  // matching against list literals
  match aList with
  | [] ->
    printfn "list is empty"
  | [x] ->
    printfn "list has one element: %A" x
  | [x;y] ->       // match using list literal
    printfn "list has two elements: %A and %A" x y
  | longerList ->  // match anything else
    printfn "list has more than two elements"
```

Or you can match using the "cons" operator, like this:

```
let printList2 aList =
  // matching against "cons"
  match aList with
  | [] ->
    printfn "list is empty"
  | first::rest ->
    printfn "list is non-empty with the first element being: %A" first
```

Organizing Types in Files and Projects

There's one last thing you should know. F# has strict rules about the order of declarations. A type higher in a file cannot reference another type further down in a file. And a file earlier in the compilation order cannot reference a file later in the compilation order. This means that when you are coding your types, you have to think about how you organize them.

A standard approach is to put all the domain types in one file, say Types.fs or Domain.fs, and then have the functions that depend on them be put later in the compilation order. If you have a lot of types and you need to split them across multiple files, put the shared ones first and the subdomain-specific ones after. Your file list might look something like this:

```
Common.Types.fs
Common.Functions.fs
OrderTaking.Types.fs
OrderTaking.Functions.fs
Shipping.Types.fs
Shipping.Functions.fs
```

Within a file, that rule means you need to put the simple types at the top and the more complex types (that depend on them) further down, in dependency order:

```
module Payments =
  // simple types at the top of the file
  type CheckNumber = CheckNumber of int

  // domain types in the middle of the file
  type PaymentMethod =
    | Cash
    | Check of CheckNumber // defined above
    | Card of ...

  // top-level types at the bottom of the file
  type Payment = {
    Amount: ...
    Currency: ...
    Method: PaymentMethod  // defined above
    }
```

When you are developing a model from the top down, the dependency order constraint can sometimes be inconvenient, because you often will want to write the lower-level types below the higher-level types. In F# 4.1 you can use the "rec" keyword at the module or namespace level to solve this. The rec keyword allows types to reference each other anywhere in the module.

```
module rec Payments =
  type Payment = {
    Amount: ...
    Currency: ...
    Method: PaymentMethod  // defined BELOW
    }

  type PaymentMethod =
    | Cash
    | Check of CheckNumber // defined BELOW
    | Card of ...

  type CheckNumber = CheckNumber of int
```

For earlier versions of F# you can use the "and" keyword to allow a type definition to reference a type directly underneath it.

```
type Payment = {
    Amount: ...
    Currency:  ...
    Method:  PaymentMethod // defined BELOW
    }

and PaymentMethod =
  | Cash
  | Check of CheckNumber   // defined BELOW
  | Card of ...

and CheckNumber = CheckNumber of int
```

This out-of-order approach is fine for sketching, but once the design has settled and is ready for production, it's generally better to put the types in the correct dependency order. This makes it consistent with other F# code and makes it easier for other developers to read.

For a real-world example of how to organize types in a project, see the code repository for this book.

Wrapping Up

In this chapter, we looked at the concept of *type* and how it relates to functional programming, and we also saw how the composition of types could be used to create larger types from smaller types using F#'s algebraic type system. We were introduced to record types, built by *AND*-ing data together, and choice types (also known as discriminated unions), built by *OR*-ing data together, as well as other common types based on these, such as `Option` and `Result`.

Now that we understand how types work, we can revisit our requirements and document them using what we've learned.