5 Files, Modules, and Programs

We've so far experienced OCaml largely through the toplevel. As you move from exercises to real-world programs, you'll need to leave the toplevel behind and start building programs from files. Files are more than just a convenient way to store and manage your code; in OCaml, they also correspond to modules, which act as boundaries that divide your program into conceptual units.

In this chapter, we'll show you how to build an OCaml program from a collection of files, as well as the basics of working with modules and module signatures.

5.1 Single-File Programs

We'll start with an example: a utility that reads lines from `stdin`, computes a frequency count of the lines, and prints out the ten most frequent lines. We'll start with a simple implementation, which we'll save as the file *freq.ml*.

This implementation will use two functions from the `List.Assoc` module, which provides utility functions for interacting with *association lists*, *i.e.*, lists of key/value pairs. In particular, we use the function `List.Assoc.find`, which looks up a key in an association list; and `List.Assoc.add`, which adds a new binding to an association list, as shown here:

```
# open Base;;
# let assoc = [("one", 1); ("two",2); ("three",3)];;
val assoc : (string * int) list = [("one", 1); ("two", 2); ("three",
    3)]
# List.Assoc.find ~equal:String.equal assoc "two";;
- : int option = Some 2
# List.Assoc.add ~equal:String.equal assoc "four" 4;;
- : (string, int) Base.List.Assoc.t =
[("four", 4); ("one", 1); ("two", 2); ("three", 3)]
# List.Assoc.add ~equal:String.equal assoc "two"  4;;
- : (string, int) Base.List.Assoc.t = [("two", 4); ("one", 1);
    ("three", 3)]
```

Note that `List.Assoc.add` doesn't modify the original list, but instead allocates a new list with the requisite key/value pair added.

Now we can write `freq.ml`.

```
open Base
open Stdio
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
let build_counts () =
  In_channel.fold_lines In_channel.stdin ~init:[] ~f:(fun counts line
    ->
      let count =
        match List.Assoc.find ~equal:String.equal counts line with
        | None -> 0
        | Some x -> x
      in
      List.Assoc.add ~equal:String.equal counts line (count + 1))

let () =
  build_counts ()
  |> List.sort ~compare:(fun (_, x) (_, y) -> Int.descending x y)
  |> (fun l -> List.take l 10)
  |> List.iter ~f:(fun (line, count) -> printf "%3d: %s\n" count line)
```

The function `build_counts` reads in lines from `stdin`, constructing from those lines an association list with the frequencies of each line. It does this by invoking `In_channel.fold_lines` (similar to the function `List.fold` described in Chapter 4 (Lists and Patterns)), which reads through the lines one by one, calling the provided `fold` function for each line to update the accumulator. That accumulator is initialized to the empty list.

With `build_counts` defined, we then call the function to build the association list, sort that list by frequency in descending order, grab the first 10 elements off the list, and then iterate over those 10 elements and print them to the screen. These operations are tied together using the `|>` operator described in Chapter 3.2.4 (Prefix and Infix Operators).

Where Is `main`?

Unlike programs in C, Java or C#, programs in OCaml don't have a unique `main` function. When an OCaml program is evaluated, all the statements in the implementation files are evaluated in the order in which they were linked together. These implementation files can contain arbitrary expressions, not just function definitions. In this example, the declaration starting with `let () =` plays the role of the `main` function, kicking off the processing. But really the entire file is evaluated at startup, and so in some sense the full codebase is one big `main` function.

The idiom of writing `let () =` may seem a bit odd, but it has a purpose. The `let` binding here is a pattern-match to a value of type `unit`, which is there to ensure that the expression on the right-hand side returns `unit`, as is common for functions that operate primarily by side effect.

If we weren't using `Base` or any other external libraries, we could build the executable like this:

```
$ ocamlopt freq.ml -o freq
File "freq.ml", line 1, characters 5-9:
1 | open Base
         ^^^^
Error: Unbound module Base
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
[2]
```

But as you can see, it fails because it can't find `Base` and `Stdio`. We need a somewhat more complex invocation to get them linked in:

```
$ ocamlfind ocamlopt -linkpkg -package base -package stdio freq.ml -o
    freq
```

This uses `ocamlfind`, a tool which itself invokes other parts of the OCaml toolchain (in this case, `ocamlopt`) with the appropriate flags to link in particular libraries and packages. Here, `-package base` is asking `ocamlfind` to link in the `Base` library; `-linkpkg` asks ocamlfind to link in the packages as is necessary for building an executable.

While this works well enough for a one-file project, more complicated projects require a tool to orchestrate the build. One good tool for this task is `dune`. To invoke `dune`, you need to have two files: a `dune-project` file for the overall project, and a `dune` file that configures the particular directory. This is a single-directory project, so we'll just have one of each, but more realistic projects will have one `dune-project` and many `dune` files.

At its simplest, the `dune-project` just specifies the version of the `dune` configuration-language in use.

```
(lang dune 3.0)
```

We also need a `dune` file to declare the executable we want to build, along with the libraries it depends on.

```
(executable
  (name      freq)
  (libraries base stdio))
```

With that in place, we can invoke `dune` as follows.

```
$ dune build freq.exe
```

We can run the resulting executable, `freq.exe`, from the command line. Executables built with `dune` will be left in the `_build/default` directory, from which they can be invoked. The specific invocation below will count the words that come up in the file `freq.ml` itself.

```
$ grep -Eo '[[:alpha:]]+' freq.ml | ./_build/default/freq.exe
  5: line
  5: List
  5: counts
  4: count
  4: fun
  4: x
  4: equal
  3: let
  2: f
  2: l
```

Conveniently, `dune` allows us to combine the building and running an executable into a single operation, which we can do using `dune exec`.

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
$ grep -Eo '[[:alpha:]]+' freq.ml | dune exec ./freq.exe
  5: line
  5: List
  5: counts
  4: count
  4: fun
  4: x
  4: equal
  3: let
  2: f
  2: l
```

We've really just scratched the surface of what can be done with `dune`. We'll discuss `dune` in more detail in Chapter 1.1.3 (The OCaml Platform).

Bytecode Versus Native Code

OCaml ships with two compilers: the `ocamlopt` native code compiler and the `ocamlc` bytecode compiler. Programs compiled with `ocamlc` are interpreted by a virtual machine, while programs compiled with `ocamlopt` are compiled to machine code to be run on a specific operating system and processor architecture. With `dune`, targets ending with `.bc` are built as bytecode executables, and those ending with `.exe` are built as native code.

Aside from performance, executables generated by the two compilers have nearly identical behavior. There are a few things to be aware of. First, the bytecode compiler can be used on more architectures, and has some tools that are not available for native code. For example, the OCaml debugger only works with bytecode (although `gdb`, the GNU Debugger, works with some limitations on OCaml native-code applications). The bytecode compiler is also quicker than the native-code compiler. In addition, in order to run a bytecode executable, you typically need to have OCaml installed on the system in question. That's not strictly required, though, since you can build a bytecode executable with an embedded runtime, using the `-custom` compiler flag.

As a general matter, production executables should usually be built using the native-code compiler, but it sometimes makes sense to use bytecode for development builds. And, of course, bytecode makes sense when targeting a platform not supported by the native-code compiler. We'll cover both compilers in more detail in Chapter 27 (The Compiler Backend: Bytecode and Native code).

5.2 Multifile Programs and Modules

Source files in OCaml are tied into the module system, with each file compiling down into a module whose name is derived from the name of the file. We've encountered modules before, such as when we used functions like `find` and `add` from the `List.Assoc` module. At its simplest, you can think of a module as a collection of definitions that are stored within a namespace.

Let's consider how we can use modules to refactor the implementation of `freq.ml`.

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

Remember that the variable `counts` contains an association list representing the counts of the lines seen so far. But updating an association list takes time linear in the length of the list, meaning that the time complexity of processing a file is quadratic in the number of distinct lines in the file.

We can fix this problem by replacing association lists with a more efficient data structure. To do that, we'll first factor out the key functionality into a separate module with an explicit interface. We can consider alternative (and more efficient) implementations once we have a clear interface to program against.

We'll start by creating a file, `counter.ml`, that contains the logic for maintaining the association list used to represent the frequency counts. The key function, called `touch`, bumps the frequency count of a given line by one.

```
open Base

let touch counts line =
  let count =
    match List.Assoc.find ~equal:String.equal counts line with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add ~equal:String.equal counts line (count + 1)
```

The file *counter.ml* will be compiled into a module named `Counter`, where the name of the module is derived automatically from the filename. The module name is capitalized even if the file is not. Indeed, module names are always capitalized.

We can now rewrite `freq.ml` to use `Counter`.

```
open Base
open Stdio

let build_counts () =
  In_channel.fold_lines In_channel.stdin ~init:[] ~f:Counter.touch

let () =
  build_counts ()
  |> List.sort ~compare:(fun (_, x) (_, y) -> Int.descending x y)
  |> (fun l -> List.take l 10)
  |> List.iter ~f:(fun (line, count) -> printf "%3d: %s\n" count line)
```

The resulting code can still be built with `dune`, which will discover dependencies and realize that `counter.ml` needs to be compiled.

```
$ dune build freq.exe
```

5.3 Signatures and Abstract Types

While we've pushed some of the logic to the `Counter` module, the code in `freq.ml` can still depend on the details of the implementation of `Counter`. Indeed, if you look at the definition of `build_counts`, you'll see that it depends on the fact that the empty set of frequency counts is represented as an empty list. We'd like to prevent this kind

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

of dependency, so we can change the implementation of `Counter` without needing to change client code like that in `freq.ml`.

The implementation details of a module can be hidden by attaching an *interface*. (Note that in the context of OCaml, the terms *interface*, *signature*, and *module type* are all used interchangeably.) A module defined by a file `filename.ml` can be constrained by a signature placed in a file called `filename.mli`.

For `counter.mli`, we'll start by writing down an interface that describes what's currently available in `counter.ml`, without hiding anything. `val` declarations are used to specify values in a signature. The syntax of a `val` declaration is as follows:

```
val <identifier> : <type>
```

Using this syntax, we can write the signature of `counter.ml` as follows.

```
open Base

(** Bump the frequency count for the given string. *)
val touch : (string * int) list -> string -> (string * int) list
```

Note that `dune` will detect the presence of the `mli` file automatically and include it in the build.

To hide the fact that frequency counts are represented as association lists, we'll need to make the type of frequency counts *abstract*. A type is abstract if its name is exposed in the interface, but its definition is not. Here's an abstract interface for `Counter`:

```
open Base

(** A collection of string frequency counts *)
type t

(** The empty set of frequency counts *)
val empty : t

(** Bump the frequency count for the given string. *)
val touch : t -> string -> t

(** Converts the set of frequency counts to an association list. A
    string shows up at most once, and the counts are >= 1. *)
val to_list : t -> (string * int) list
```

We added `empty` and `to_list` to `Counter`, since without them there would be no way to create a `Counter.t` or get data out of one.

We also used this opportunity to document the module. The `mli` file is the place where you specify your module's interface, and as such is a natural place to put documentation. We started our comments with a double asterisk to cause them to be picked up by the `odoc` tool when generating API documentation. We'll discuss `odoc` more in Chapter 22.2.2 (Browsing Interface Documentation).

Here's a rewrite of `counter.ml` to match the new `counter.mli`:

```
open Base

type t = (string * int) list
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
let empty = []
let to_list x = x

let touch counts line =
  let count =
    match List.Assoc.find ~equal:String.equal counts line with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add ~equal:String.equal counts line (count + 1)
```

If we now try to compile `freq.ml`, we'll get the following error:

```
$ dune build freq.exe
File "freq.ml", line 5, characters 53-66:
5 |   In_channel.fold_lines In_channel.stdin ~init:[] ~f:Counter.touch
                                                         ^^^^^^^^^^^^^
Error: This expression has type Counter.t -> Export.string ->
    Counter.t
       but an expression was expected of type
         'a list -> Export.string -> 'a list
       Type Counter.t is not compatible with type 'a list
[1]
```

This is because `freq.ml` depends on the fact that frequency counts are represented as association lists, a fact that we've just hidden. We just need to fix `build_counts` to use `Counter.empty` instead of `[]` and to use `Counter.to_list` to convert the completed counts to an association list. The resulting implementation is shown below.

```
open Base
open Stdio

let build_counts () =
  In_channel.fold_lines
    In_channel.stdin
    ~init:Counter.empty
    ~f:Counter.touch

let () =
  build_counts ()
  |> Counter.to_list
  |> List.sort ~compare:(fun (_, x) (_, y) -> Int.descending x y)
  |> (fun counts -> List.take counts 10)
  |> List.iter ~f:(fun (line, count) -> printf "%3d: %s\n" count line)
```

With this implementation, the build now succeeds!

```
$ dune build freq.exe
```

Now we can turn to optimizing the implementation of `Counter`. Here's an alternate and far more efficient implementation, based on `Base`'s `Map` data structure.

```
open Base

type t = int Map.M(String).t

let empty = Map.empty (module String)
let to_list t = Map.to_alist t
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
let touch t s =
  let count =
    match Map.find t s with
    | None -> 0
    | Some x -> x
  in
  Map.set t ~key:s ~data:(count + 1)
```

There's some unfamiliar syntax in the above example, in particular the use of `int Map.M(String).t` to indicate the type of a map, and `Map.empty (module String)` to generate an empty map. Here, we're making use of a more advanced feature of the language (specifically, functors and first-class modules, which we'll get to in later chapters). The use of these features for the Map data-structure in particular is covered in Chapter 15 (Maps and Hash Tables).

5.4 Concrete Types in Signatures

In our frequency-count example, the module `Counter` had an abstract type `Counter.t` for representing a collection of frequency counts. Sometimes, you'll want to make a type in your interface *concrete*, by including the type definition in the interface.

For example, imagine we wanted to add a function to `Counter` for returning the line with the median frequency count. If the number of lines is even, then there is no single median, and the function would return the lines before and after the median instead. We'll use a custom type to represent the fact that there are two possible return values. Here's a possible implementation:

```
type median =
  | Median of string
  | Before_and_after of string * string

let median t =
  let sorted_strings =
    List.sort (Map.to_alist t) ~compare:(fun (_, x) (_, y) ->
        Int.descending x y)
  in
  let len = List.length sorted_strings in
  if len = 0 then failwith "median: empty frequency count";
  let nth n = fst (List.nth_exn sorted_strings n) in
  if len % 2 = 1
  then Median (nth (len / 2))
  else Before_and_after (nth ((len / 2) - 1), nth (len / 2))
```

In the above, we use `failwith` to throw an exception for the case of the empty list. We'll discuss exceptions more in Chapter 8 (Error Handling). Note also that the function `fst` simply returns the first element of any two-tuple.

Now, to expose this usefully in the interface, we need to expose both the function and the type `median` with its definition. Note that values (of which functions are an example) and types have distinct namespaces, so there's no name clash here. Adding the following two lines to `counter.mli` does the trick.

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
(** Represents the median computed from a set of strings. In the case
    where there is an even number of choices, the one before and after
    the median is returned. *)
type median =
  | Median of string
  | Before_and_after of string * string

val median : t -> median
```

The decision of whether a given type should be abstract or concrete is an important one. Abstract types give you more control over how values are created and accessed, and make it easier to enforce invariants beyond what is enforced by the type itself; concrete types let you expose more detail and structure to client code in a lightweight way. The right choice depends very much on the context.

5.5 Nested Modules

Up until now, we've only considered modules that correspond to files, like `counter.ml`. But modules (and module signatures) can be nested inside other modules. As a simple example, consider a program that needs to deal with multiple identifiers like usernames and hostnames. If you just represent these as strings, then it becomes easy to confuse one with the other.

A better approach is to mint new abstract types for each identifier, where those types are under the covers just implemented as strings. That way, the type system will prevent you from confusing a username with a hostname, and if you do need to convert, you can do so using explicit conversions to and from the string type.

Here's how you might create such an abstract type, within a submodule:

```
open Base

module Username : sig
  type t

  val of_string : string -> t
  val to_string : t -> string
  val ( = ) : t -> t -> bool
end = struct
  type t = string

  let of_string x = x
  let to_string x = x
  let ( = ) = String.( = )
end
```

Note that the `to_string` and `of_string` functions above are implemented simply as the identity function, which means they have no runtime effect. They are there purely as part of the discipline that they enforce on the code through the type system. We also chose to put in an equality function, so you can check if two usernames match. In a real application, we might want more functionality, like the ability to hash and compare usernames, but we've kept this example purposefully simple.

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

The basic structure of a module declaration like this is:

```
module <name> : <signature> = <implementation>
```

We could have written this slightly differently, by giving the signature its own top-level `module type` declaration, making it possible to create multiple distinct types with the same underlying implementation in a lightweight way:

```
open Base
module Time = Core.Time

module type ID = sig
  type t

  val of_string : string -> t
  val to_string : t -> string
  val ( = ) : t -> t -> bool
end

module String_id = struct
  type t = string

  let of_string x = x
  let to_string x = x
  let ( = ) = String.( = )
end

module Username : ID = String_id
module Hostname : ID = String_id

type session_info =
  { user : Username.t
  ; host : Hostname.t
  ; when_started : Time.t
  }

let sessions_have_same_user s1 s2 = Username.( = ) s1.user s2.host
```

The preceding code has a bug: it compares the username in one session to the host in the other session, when it should be comparing the usernames in both cases. Because of how we defined our types, however, the compiler will flag this bug for us.

```
$ dune build session_info.exe
File "session_info.ml", line 29, characters 59-66:
29 | let sessions_have_same_user s1 s2 = Username.( = ) s1.user
    s2.host

    ^^^^^^^
Error: This expression has type Hostname.t
       but an expression was expected of type Username.t
[1]
```

This is a trivial example, but confusing different kinds of identifiers is a very real source of bugs, and the approach of minting abstract types for different classes of identifiers is an effective way of avoiding such issues.

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

5.6 Opening Modules

Most of the time, you refer to values and types within a module by using the module name as an explicit qualifier. For example, you write `List.map` to refer to the `map` function in the `List` module. Sometimes, though, you want to be able to refer to the contents of a module without this explicit qualification. That's what the `open` statement is for.

We've encountered `open` already, specifically where we've written `open Base` to get access to the standard definitions in the `Base` library. In general, opening a module adds the contents of that module to the environment that the compiler looks at to find the definition of various identifiers. Here's an example:

```
# module M = struct let foo = 3 end;;
module M : sig val foo : int end
# foo;;
Line 1, characters 1-4:
Error: Unbound value foo
# open M;;
# foo;;
- : int = 3
```

Here's some general advice on how to use `open` effectively.

5.6.1 Open Modules Rarely

`open` is essential when you're using an alternative standard library like `Base`, but it's generally good style to keep the opening of modules to a minimum. Opening a module is basically a trade-off between terseness and explicitness—the more modules you open, the fewer module qualifications you need, and the harder it is to look at an identifier and figure out where it comes from.

When you do use `open`, it should mostly be with modules that were designed to be opened, like `Base` itself, or `Option.Monad_infix` or `Float.O` within `Base`..

5.6.2 Prefer Local Opens

It's generally better to keep down the amount of code affected by an `open`. One great tool for this is *local open*s, which let you restrict the scope of an open to an arbitrary expression. There are two syntaxes for local opens. The following example shows the `let open` syntax;

```
# let average x y =
    let open Int64 in
    (x + y) / of_int 2;;
val average : int64 -> int64 -> int64 = <fun>
```

Here, `of_int` and the infix operators are the ones from the `Int64` module.

The following shows off a more lightweight syntax which is particularly useful for small expressions.

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
# let average x y =
    Int64.((x + y) / of_int 2);;
val average : int64 -> int64 -> int64 = <fun>
```

5.6.3 Using Module Shortcuts Instead

An alternative to local opens that makes your code terser without giving up on explicitness is to locally rebind the name of a module. So, when using the `Counter.median` type, instead of writing:

```
let print_median m =
  match m with
  | Counter.Median string -> printf "True median:\n   %s\n" string
  | Counter.Before_and_after (before, after) ->
    printf "Before and after median:\n   %s\n   %s\n" before after
```

you could write:

```
let print_median m =
  let module C = Counter in
  match m with
  | C.Median string -> printf "True median:\n   %s\n" string
  | C.Before_and_after (before, after) ->
    printf "Before and after median:\n   %s\n   %s\n" before after
```

Because the module name `C` only exists for a short scope, it's easy to read and remember what `C` stands for. Rebinding modules to very short names at the top level of your module is usually a mistake.

5.7 Including Modules

While opening a module affects the environment used to search for identifiers, *including* a module is a way of adding new identifiers to a module proper. Consider the following simple module for representing a range of integer values:

```
# module Interval = struct
    type t = | Interval of int * int
             | Empty

    let create low high =
      if high < low then Empty else Interval (low,high)
  end;;
module Interval :
  sig type t = Interval of int * int | Empty val create : int -> int
    -> t end
```

We can use the `include` directive to create a new, extended version of the `Interval` module:

```
# module Extended_interval = struct
    include Interval
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
    let contains t x =
      match t with
      | Empty -> false
      | Interval (low,high) -> x >= low && x <= high
  end;;
module Extended_interval :
  sig
    type t = Interval.t = Interval of int * int | Empty
    val create : int -> int -> t
    val contains : t -> int -> bool
  end
# Extended_interval.contains (Extended_interval.create 3 10) 4;;
- : bool = true
```

The difference between `include` and `open` is that we've done more than change how identifiers are searched for: we've changed what's in the module. If we'd used `open`, we'd have gotten a quite different result:

```
# module Extended_interval = struct
    open Interval

    let contains t x =
      match t with
      | Empty -> false
      | Interval (low,high) -> x >= low && x <= high
  end;;
module Extended_interval :
  sig val contains : Extended_interval.t -> int -> bool end
# Extended_interval.contains (Extended_interval.create 3 10) 4;;
Line 1, characters 29-53:
Error: Unbound value Extended_interval.create
```

To consider a more realistic example, imagine you wanted to build an extended version of the `Option` module, where you've added some functionality not present in the module as distributed in `Base`. That's a job for `include`.

```
open Base

(* The new function we're going to add *)
let apply f_opt x =
  match f_opt with
  | None -> None
  | Some f -> Some (f x)

(* The remainder of the option module *)
include Option
```

Now, how do we write an interface for this new module? It turns out that `include` works on signatures as well, so we can pull essentially the same trick to write our `mli`. The only issue is that we need to get our hands on the signature for the `Option` module. This can be done using `module type of`, which computes a signature from a module:

```
open Base

(* Include the interface of the option module from Base *)
include module type of Option
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
(* Signature of function we're adding *)
val apply : ('a -> 'b) t -> 'a -> 'b t
```

Note that the order of declarations in the mli does not need to match the order of declarations in the ml. The order of declarations in the ml mostly matters insofar as it affects which values are shadowed. If we wanted to replace a function in Option with a new function of the same name, the declaration of that function in the ml would have to come after the include Option declaration.

We can now use Ext_option as a replacement for Option. If we want to use Ext_option in preference to Option in our project, we can create a file of common definitions, which in this case we'll call import.ml.

```
module Option = Ext_option
```

Then, by opening Import, we can shadow Base's Option module with our extension.

```
open Base
open Import

let lookup_and_apply map key x = Option.apply (Map.find map key) x
```

5.8 Common Errors with Modules

When OCaml compiles a program with an ml and an mli, it will complain if it detects a mismatch between the two. Here are some of the common errors you'll run into.

5.8.1 Type Mismatches

The simplest kind of error is where the type specified in the signature does not match the type in the implementation of the module. As an example, if we replace the val declaration in counter.mli by swapping the types of the first two arguments:

```
(** Bump the frequency count for the given string. *)
val touch : string -> t -> t
```

and we try to compile, we'll get the following error.

```
$ dune build freq.exe
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface
    .freq.eobjs/byte/dune__exe__Counter.cmi:
       Values do not match:
         val touch :
           ('a, int, 'b) Base.Map.t -> 'a -> ('a, int, 'b) Base.Map.t
       is not included in
         val touch : string -> t -> t
       File "counter.mli", line 16, characters 0-28: Expected
    declaration
       File "counter.ml", line 8, characters 4-9: Actual declaration
[1]
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

5.8.2 Missing Definitions

We might decide that we want a new function in `Counter` for pulling out the frequency count of a given string. We could add that to the `mli` by adding the following line.

```
(** Returns the frequency count for the given string *)
val count : t -> string -> int
```

Now if we try to compile without actually adding the implementation, we'll get this error.

```
$ dune build freq.exe
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface
    .freq.eobjs/byte/dune__exe__Counter.cmi:
       The value `count' is required but not provided
       File "counter.mli", line 15, characters 0-30: Expected
    declaration
[1]
```

A missing type definition will lead to a similar error.

5.8.3 Type Definition Mismatches

Type definitions that show up in an `mli` need to match up with corresponding definitions in the `ml`. Consider again the example of the type `median`. The order of the declaration of variants matters to the OCaml compiler, so the definition of `median` in the implementation listing those options in a different order:

```
(** Represents the median computed from a set of strings. In the case
    where there is an even number of choices, the one before and after
    the median is returned. *)
type median =
  | Before_and_after of string * string
  | Median of string

val median : t -> median
```

will lead to a compilation error.

```
$ dune build freq.exe
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface
    .freq.eobjs/byte/dune__exe__Counter.cmi:
       Type declarations do not match:
         type median = Median of string | Before_and_after of string
    * string
       is not included in
         type median = Before_and_after of string * string | Median
    of string
       Constructors number 1 have different names, Median and
    Before_and_after.
       File "counter.mli", lines 21-23, characters 0-20: Expected
    declaration
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

```
       File "counter.ml", lines 17-19, characters 0-39: Actual
    declaration
[1]
```

Order is similarly important to other type declarations, including the order in which record fields are declared and the order of arguments (including labeled and optional arguments) to a function.

5.8.4 Cyclic Dependencies

In most cases, OCaml doesn't allow cyclic dependencies, i.e., a collection of definitions that all refer to one another. If you want to create such definitions, you typically have to mark them specially. For example, when defining a set of mutually recursive values (like the definition of `is_even` and `is_odd` in Chapter 3.2.3 (Recursive Functions)), you need to define them using `let rec` rather than ordinary `let`.

The same is true at the module level. By default, cyclic dependencies between modules are not allowed, and cyclic dependencies among files are never allowed. Recursive modules are possible but are a rare case, and we won't discuss them further here.

The simplest example of a forbidden circular reference is a module referring to its own module name. So, if we tried to add a reference to `Counter` from within `counter.ml`.

```
let singleton l = Counter.touch Counter.empty
```

we'll see this error when we try to build:

```
$ dune build freq.exe
File "counter.ml", line 18, characters 18-31:
18 | let singleton l = Counter.touch Counter.empty
                       ^^^^^^^^^^^^^
Error: The module Counter is an alias for module Dune__exe__Counter,
    which is the current compilation unit
[1]
```

The problem manifests in a different way if we create cyclic references between files. We could create such a situation by adding a reference to `Freq` from `counter.ml`, e.g., by adding the following line.

```
let _build_counts = Freq.build_counts
```

In this case, `dune` will notice the error and complain explicitly about the cycle:

```
$ dune build freq.exe
Error: Dependency cycle between the following files:
   _build/default/.freq.eobjs/freq.impl.all-deps
-> _build/default/.freq.eobjs/counter.impl.all-deps
-> _build/default/.freq.eobjs/freq.impl.all-deps
[1]
```

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

5.9 Designing with Modules

The module system is a key part of how an OCaml program is structured. As such, we'll close this chapter with some advice on how to think about designing that structure effectively.

5.9.1 Expose Concrete Types Rarely

When designing an `mli`, one choice that you need to make is whether to expose the concrete definition of your types or leave them abstract. Most of the time, abstraction is the right choice, for two reasons: it enhances the flexibility of your design, and it makes it possible to enforce invariants on the use of your module.

Abstraction enhances flexibility by restricting how users can interact with your types, thus reducing the ways in which users can depend on the details of your implementation. If you expose types explicitly, then users can depend on any and every detail of the types you choose. If they're abstract, then only the specific operations you want to expose are available. This means that you can freely change the implementation without affecting clients, as long as you preserve the semantics of those operations.

In a similar way, abstraction allows you to enforce invariants on your types. If your types are exposed, then users of the module can create new instances of that type (or if mutable, modify existing instances) in any way allowed by the underlying type. That may violate a desired invariant *i.e.*, a property about your type that is always supposed to be true. Abstract types allow you to protect invariants by making sure that you only expose functions that preserve your invariants.

Despite these benefits, there is a trade-off here. In particular, exposing types concretely makes it possible to use pattern-matching with those types, which as we saw in Chapter 4 (Lists and Patterns) is a powerful and important tool. You should generally only expose the concrete implementation of your types when there's significant value in the ability to pattern match, and when the invariants that you care about are already enforced by the data type itself.

5.9.2 Design for the Call Site

When writing an interface, you should think not just about how easy it is to understand the interface for someone who reads your carefully documented `mli` file, but more importantly, you want the call to be as obvious as possible for someone who is reading it at the call site.

The reason for this is that most of the time, people interacting with your API will be doing so by reading and modifying code that uses the API, not by reading the interface definition. By making your API as obvious as possible from that perspective, you simplify the lives of your users.

There are many ways of improving readability of client code. One example is labeled arguments (discussed in Chapter 3.2.6 (Labeled Arguments)), which act as documentation that is available at the call site.

https://doi.org/10.1017/9781009129220.007 Published online by Cambridge University Press

You can also improve readability simply by choosing good names for your functions, variant tags and record fields. Good names aren't always long, to be clear. If you wanted to write an anonymous function for doubling a number: `(fun x -> x * 2)`, a short variable name like `x` is best. A good rule of thumb is that names that have a small scope should be short, whereas names that have a large scope, like the name of a function in a module interface, should be longer and more descriptive.

There is of course a tradeoff here, in that making your APIs more explicit tends to make them more verbose as well. Another useful rule of thumb is that more rarely used names should be longer and more explicit, since the cost of verbosity goes down and the benefit of explicitness goes up the less often a name is used.

5.9.3 Create Uniform Interfaces

Designing the interface of a module is a task that should not be thought of in isolation. The interfaces that appear in your codebase should play together harmoniously. Part of achieving that is standardizing aspects of those interfaces.

`Base`, `Core` and related libraries have been designed with a uniform set of standards in mind around the design of module interfaces. Here are some of the guidelines that they use.

- *A module for (almost) every type.* You should mint a module for almost every type in your program, and the primary type of a given module should be called `t`.
- *Put `t` first.* If you have a module `M` whose primary type is `M.t`, the functions in `M` that take a value of type `M.t` should take it as their first argument.
- Functions that routinely throw an exception should end in `_exn`. Otherwise, errors should be signaled by returning an `option` or an `Or_error.t` (both of which are discussed in Chapter 8 (Error Handling)).

There are also standards in Base about what the type signature for specific functions should be. For example, the signature for `map` is always essentially the same, no matter what the underlying type it is applied to. This kind of function-by-function API uniformity is achieved through the use of *signature includes*, which allow for different modules to share components of their interface. This approach is described in Chapter 11.2.4 (Using Multiple Interfaces).

Base's standards may or may not fit your projects, but you can improve the usability of your codebase by finding some consistent set of standards to apply.

5.9.4 Interfaces Before Implementations

OCaml's concise and flexible type language enables a type-oriented approach to software design. Such an approach involves thinking through and writing out the types you're going to use before embarking on the implementation itself.

This is a good approach both when working in the core language, where you would write your type definitions before writing the logic of your computations, as well as at the module level, where you would write a first draft of your `mli` before working on the `ml`.

Of course, the design process goes in both directions. You'll often find yourself going back and modifying your types in response to things you learn by working on the implementation. But types and signatures provide a lightweight tool for constructing a skeleton of your design in a way that helps clarify your goals and intent, before you spend a lot of time and effort fleshing it out.