Appendix: Packages

Packages are Common Lisp's way of grouping code into modules. Early dialects of Lisp contained a symbol-table, called the *oblist*, which listed all the symbols read so far by the system. Through a symbol's entry on the oblist, the system had access to things like its value and its property list. A symbol listed in the oblist was said to be *interned*.

Recent dialects of Lisp have split the concept of the oblist into multiple *packages*. Now a symbol is not merely interned, but interned in a particular package. Packages support modularity because symbols interned in one package are only accessible in other packages (except by cheating) if they are explicitly declared to be so.

A package is a kind of Lisp object. The current package is always stored in the global variable `*package*`. When Common Lisp starts up, the current package will be the user package: either `user` (in CLTL1 implementations), or `common-lisp-user` (in CLTL2 implementations).

Packages are usually identified by their names, which are strings. To find the name of the current package, try:

```
> (package-name *package*)
"COMMON-LISP-USER"
```

Usually a symbol is interned in the package that was current at the time it was read. To find the package in which a symbol is interned, we can use `symbol-package`:

```
> (symbol-package 'foo)
#<Package "COMMON-LISP-USER" 4CD15E>
```

The return value here is the actual package object. For future use, let's give `foo` a value:

```
> (setq foo 99)
99
```

By calling `in-package` we can switch to a new package, creating it if necessary:[1]

```
> (in-package 'mine :use 'common-lisp)
#<Package "MINE" 63390E>
```

At this point there should be eerie music, because we are in a different world: `foo` here is not what it used to be.

```
MINE> foo
>>Error: FOO has no global value.
```

Why did this happen? Because the `foo` we set to `99` above is a distinct symbol from `foo` here in `mine`.[2] To refer to the original `foo` from outside the user package, we must prefix the package name and two colons:

```
MINE> common-lisp-user::foo
99
```

So different symbols with the same print-name can coexist in different packages. There can be one `foo` in package `common-lisp-user` and another `foo` in package `mine`, and they will be distinct symbols. In fact, that's partly the point of packages: if you're writing your program in a separate package, you can choose names for your functions and variables without worrying that someone will use the same name for something else. Even if they use the same name, it won't be the same symbol.

Packages also provide a means of information-hiding. Programs must refer to functions and variables by their names. If you don't make a given name available outside your package, it becomes unlikely that code in another package will be able to use or modify what it refers to.

In programs it's usually bad style to use package prefixes with double colons. By doing so you are violating the modularity that packages are supposed to provide. If you have to use a double colon to refer to a symbol, it's because someone didn't want you to.

[1] In older implementations of Common Lisp, omit the `:use` argument.

[2] Some implementations of Common Lisp print the package name before the toplevel prompt whenever we are not in the user package. This is not required, but it is a nice touch.

Usually one should only refer to symbols which have been *exported*. By exporting a symbol from the package in which it is interned, we cause it to be visible to other packages. To export a symbol we call (you guessed it) `export`:

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T
> (setq bar 5)
5
```

Now when we return to `mine`, we can refer to `bar` with only a single colon, because it is a publicly available name:

```
> (in-package 'mine)
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

By *importing* `bar` into `mine` we can go one step further, and make `mine` actually share the symbol `bar` with the user package:

```
MINE> (import 'common-lisp-user:bar)
T
MINE> bar
5
```

After importing `bar` we can refer to it without any package qualifier at all. The two packages now share the same symbol; there can't be a distinct `mine:bar`.

What if there already was one? In that case, the call to `import` would have caused an error, as we see if we try to import `foo`:

```
MINE> (import 'common-lisp-user::foo)
>>Error: FOO is already present in MINE.
```

Before, when we tried unsuccessfully to evaluate `foo` in `mine`, we thereby caused a symbol `foo` to be interned there. It had no global value and therefore generated an error, but the interning happened simply as a consequence of typing its name. So now when we try to import `foo` into `mine`, there is already a symbol there with the same name.

We can also import symbols *en masse* by defining one package to *use* another:

```
MINE> (use-package 'common-lisp-user)
T
```

Now all symbols exported by the user package will automatically be imported by `mine`. (If `foo` had been exported by the user package, this call would also have generated an error.)

As of CLTL2, the package containing the names of built-in operators and variables is called `common-lisp` instead of `lisp`, and new packages no longer use it by default. Since we used this package in the call to `in-package` which created `mine`, all of Common Lisp's names will be visible here:

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

You're practically compelled to make any new package use `common-lisp` (or some other package containing Lisp operators). Otherwise you wouldn't even be able to get out of the new package.

As with compilation, operations on packages are not usually done at the toplevel like this. More often the calls are contained in source files. Generally it will suffice to begin a file with an `in-package` and a `defpackage`. (The `defpackage` macro is new in CLTL2, but some older implementations provide it.) Here is what you might put at the top of a file containing a distinct package of code:

```
(in-package 'my-application :use 'common-lisp)

(defpackage my-application
            (:use common-lisp my-utilities)
            (:nicknames app)
            (:export win lose draw))
```

This will cause the code in the file—or more precisely, the names in the file—to be in the package `my-application`. As well as `common-lisp`, this package uses `my-utilities`, so any symbols exported thence can appear without any package prefix in the file.

The `my-application` package itself exports just three symbols: `win`, `lose`, and `draw`. Since the call to `in-package` gave `my-application` the nickname `app`, code in other packages will be able to refer to them as e.g. `app:win`.

The kind of modularity provided by packages is actually a bit odd. We have modules not of objects, but of names. Every package that uses `common-lisp` imports the name `cons`, because `common-lisp` includes a function with that name. But in consequence a variable called `cons` would also be visible every package that used `common-lisp`. And the same thing goes for Common Lisp's other name-spaces. If packages are confusing, this is the main reason why; they're not based on objects, but on names.

Things having to do with packages tend to happen at read-time, not runtime, which can lead to some confusion. The second expression we typed:

```
(symbol-package 'foo)
```

returned the value it did because *reading the query created the answer.* To evaluate this expression, Lisp had to read it, which meant interning `foo`.

As another example, consider this exchange, which appeared above:

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
```

Usually two expressions typed into the toplevel are equivalent to the same two expressions enclosed within a single `progn`. Not in this case. If we try saying

```
MINE> (progn (in-package 'common-lisp-user)
             (export 'bar))
>>Error: MINE::BAR is not accessible in COMMON-LISP-USER.
```

we get an error instead. This happens because the whole `progn` expression is processed by `read` before being evaluated. When `read` is called, the current package is `mine`, so `bar` is taken to be `mine:bar`. It is as if we had asked to export this symbol, instead of `common-lisp-user:bar`, from the user package.

The way packages are defined makes it a nuisance to write programs which use symbols as data. For example, if we define `noise` as follows:

```
(in-package 'other :use 'common-lisp)
(defpackage other
            (:use common-lisp)
            (:export noise))

(defun noise (animal)
  (case animal
    (dog 'woof)
    (cat 'meow)
    (pig 'oink)))
```

then if we call `noise` from another package with an unqualified symbol as an argument, it will usually fall off the end of the `case` clauses and return `nil`:

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise 'pig)
NIL
```

That's because what we passed as an argument was `common-lisp-user:pig` (no offense intended), while the `case` key is `other:pig`. To make `noise` work as one would expect, we would have to export all six symbols used within it, and import them into any package from which we intended to call `noise`.

In this case, we could evade the problem by using keywords instead of ordinary symbols. If `noise` had been defined

```
(defun noise (animal)
  (case animal
    (:dog :woof)
    (:cat :meow)
    (:pig :oink)))
```

then we could safely call it from any package:

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise :pig)
:OINK
```

Keywords are like gold: universal and self-evaluating. They are visible everywhere, and they never have to be quoted. A symbol-driven function like `defanaph` (page 223) should nearly always be written to use keywords.

Packages are a rich source of confusion. This introduction to the subject has barely scratched the surface. For all the details, see CLTL2, Chapter 11.