21. Programming in the Large: Packages and Symbols

In Chapter 4 I discussed how the Lisp reader translates textual names into objects to be passed to the evaluator, representing them with a kind of object called a *symbol*. It turns out that having a built-in data type specifically for representing names is quite handy for a lot of kinds of programming.[1] That, however, isn't the topic of this chapter. In this chapter I'll discuss one of the more immediate and practical aspects of dealing with names: how to avoid name conflicts between independently developed pieces of code.

Suppose, for instance, you're writing a program and decide to use a third-party library. You don't want to have to know the name of every function, variable, class, or macro used in the internals of that library in order to avoid conflicts between those names and the names you use in your program. You'd like for most of the names in the library and the names in your program to be considered distinct even if they happen to have the same textual representation. At the same time, you'd like certain names defined in the library to be readily accessible--the names that make up its public API, which you'll want to use in your program.

In Common Lisp, this namespace problem boils down to a question of controlling how the reader translates textual names into symbols: if you want two occurrences of the same name to be considered the same by the evaluator, you need to make sure the reader uses the same symbol to represent each name. Conversely, if you want two names to be considered distinct, even if they happen to have the same textual name, you need the reader to create different symbols to represent each name.

How the Reader Uses Packages

In Chapter 4 I discussed briefly how the Lisp reader translates names into symbols, but I glossed over most of the details--now it's time to take a closer look at what actually happens.

I'll start by describing the syntax of names understood by the reader and how that syntax relates to packages. For the moment you can think of a package as a table that maps strings to symbols. As you'll see in the next section, the actual mapping is slightly more flexible than a simple lookup table but not in ways that matter much to the reader. Each package also has a name, which can be used to find the package using the function **`FIND-PACKAGE`**.

The two key functions that the reader uses to access the name-to-symbol mappings in a package are **FIND-SYMBOL** and **INTERN**. Both these functions take a string and, optionally, a package. If not supplied, the package argument defaults to the value of the global variable ***PACKAGE***, also called the *current package*.

FIND-SYMBOL looks in the package for a symbol with the given string for a name and returns it, or **NIL** if no symbol is found. **INTERN** also will return an existing symbol; otherwise it creates a new symbol with the string as its name and adds it to the package.

Most names you use are *unqualified*, names that contain no colons. When the reader reads such a name, it translates it to a symbol by converting any unescaped letters to uppercase and passing the resulting string to **INTERN**. Thus, each time the reader reads the same name in the same package, it'll get the same symbol object. This is important because the evaluator uses the object identity of symbols to determine which function, variable, or other program element a given symbol refers to. Thus, the reason an expression such as `(hello-world)` results in calling a particular `hello-world` function is because the reader returns the same symbol when it reads the function call as it did when it read the **DEFUN** form that defined the function.

A name containing either a single colon or a double colon is a package-qualified name. When the reader reads a package-qualified name, it splits the name on the colon(s) and uses the first part as the name of a package and the second part as the name of the symbol. The reader looks up the appropriate package and uses it to translate the symbol name to a symbol object.

A name containing only a single colon must refer to an *external* symbol--one the package *exports* for public use. If the named package doesn't contain a symbol with a given name, or if it does but it hasn't been exported, the reader signals an error. A double-colon name can refer to any symbol from the named package, though it's usually a bad idea--the set of exported symbols defines a package's public interface, and if you don't respect the package author's decision about what names to make public and which ones to keep private, you're asking for trouble down the road. On the other hand, sometimes a package author will neglect to export a symbol that really ought to be public. In that case, a double-colon name lets you get work done without having to wait for the next version of the package to be released.

Two other bits of symbol syntax the reader understands are those for keyword symbols and uninterned symbols. Keyword symbols are written with names starting with a colon. Such symbols are interned in the package named **KEYWORD** and automatically exported. Additionally, when the reader interns a symbol in the **KEYWORD**, it also defines a constant variable with the symbol as both its name and value. This is why you can use keywords in argument lists without quoting them--when they appear in a value position, they evaluate to themselves. Thus:

```
(eql ':foo :foo) ==> T
```

The names of keyword symbols, like all symbols, are converted to all uppercase by the reader before they're interned. The name doesn't include the leading colon.

```
(symbol-name :foo) ==> "FOO"
```

Uninterned symbols are written with a leading `#:`. These names (minus the `#:`) are converted to uppercase as normal and then translated into symbols, but the symbols aren't interned in any package; each time the reader reads a `#:` name, it creates a new symbol. Thus:

```
(eql '#:foo '#:foo) ==> NIL
```

You'll rarely, if ever, write this syntax yourself, but will sometimes see it when you print an s-expression containing symbols returned by the function **`GENSYM`**.

```
(gensym) ==> #:G3128
```

A Bit of Package and Symbol Vocabulary

As I mentioned previously, the mapping from names to symbols implemented by a package is slightly more flexible than a simple lookup table. At its core, every package contains a name-to-symbol lookup table, but a symbol can be made accessible via an unqualified name in a given package in other ways. To talk sensibly about these other mechanisms, you'll need a little bit of vocabulary.

To start with, all the symbols that can be found in a given package using **`FIND-SYMBOL`** are said to be *accessible* in that package. In other words, the accessible symbols in a package are those that can be referred to with unqualified names when the package is current.

A symbol can be accessible in two ways. The first is for the package's name-to-symbol table to contain an entry for the symbol, in which case the symbol is said to be *present* in the package. When the reader interns a new symbol in a package, it's added to the package's name-to-symbol table. The package in which a symbol is first interned is called the symbol's *home package*.

The other way a symbol can be accessible in a package is if the package *inherits* it. A package inherits symbols from other packages by *using* the other packages. Only *external* symbols in the used packages are inherited. A symbol is made external in a package by *exporting* it. In addition to causing it to be inherited by using packages, exporting a symbol also--as you saw in the previous section--makes it possible to refer to the symbol using a single-colon qualified name.

To keep the mappings from names to symbols deterministic, the package system allows only one symbol to be accessible in a given package for each name. That is, a package can't have a present symbol and an inherited symbol with the same name or inherit two different symbols, from different packages, with the same name. However, you can resolve conflicts by making one of the accessible symbols a *shadowing* symbol, which makes the other symbols of the same name

inaccessible. In addition to its name-to-symbol table, each package maintains a list of shadowing symbols.

An existing symbol can be *imported* into another package by adding it to the package's name-to-symbol table. Thus, the same symbol can be present in multiple packages. Sometimes you'll import symbols simply because you want them to be accessible in the importing package without using their home package. Other times you'll import a symbol because only present symbols can be exported or be shadowing symbols. For instance, if a package needs to use two packages that have external symbols of the same name, one of the symbols must be imported into the using package in order to be added to its shadowing list and make the other symbol inaccessible.

Finally, a present symbol can be *uninterned* from a package, which causes it to be removed from the name-to-symbol table and, if it's a shadowing symbol, from the shadowing list. You might unintern a symbol from a package to resolve a conflict between the symbol and an external symbol from a package you want to use. A symbol that isn't present in any package is called an *uninterned* symbol, can no longer be read by the reader, and will be printed using the `#:foo` syntax.

Three Standard Packages

In the next section I'll show you how to define your own packages, including how to make one package use another and how to export, shadow, and import symbols. But first let's look at a few packages you've been using already. When you first start Lisp, the value of **`*PACKAGE*`** is typically the `COMMON-LISP-USER` package, also known as `CL-USER`.[2] `CL-USER` uses the package `COMMON-LISP`, which exports all the names defined by the language standard. Thus, when you type an expression at the REPL, all the names of standard functions, macros, variables, and so on, will be translated to the symbols exported from `COMMON-LISP`, and all other names will be interned in the `COMMON-LISP-USER` package. For example, the name **`*PACKAGE*`** is exported from `COMMON-LISP`--if you want to see the value of **`*PACKAGE*`**, you can type this:

```
CL-USER> *package*
#<The COMMON-LISP-USER package>
```

because `COMMON-LISP-USER` uses `COMMON-LISP`. Or you can use a package-qualified name.

```
CL-USER> common-lisp:*package*
#<The COMMON-LISP-USER package>
```

You can even use `COMMON-LISP`'s nickname, `CL`.

```
CL-USER> cl:*package*
#<The COMMON-LISP-USER package>
```

But `*X*` isn't a symbol in `COMMON-LISP`, so you if type this:

```
CL-USER> (defvar *x* 10)
*X*
```

the reader reads **DEFVAR** as the symbol from the COMMON-LISP package and *X* as a symbol in COMMON-LISP-USER.

The REPL can't start in the COMMON-LISP package because you're not allowed to intern new symbols in it; COMMON-LISP-USER serves as a "scratch" package where you can create your own names while still having easy access to all the symbols in COMMON-LISP.[3] Typically, all packages you'll define will also use COMMON-LISP, so you don't have to write things like this:

```
(cl:defun (x) (cl:+ x 2))
```

The third standard package is the **KEYWORD** package, the package the Lisp reader uses to intern names starting with colon. Thus, you can also refer to any keyword symbol with an explicit package qualification of keyword like this:

```
CL-USER> :a
:A
CL-USER> keyword:a
:A
CL-USER> (eql :a keyword:a)
T
```

Defining Your Own Packages

Working in COMMON-LISP-USER is fine for experiments at the REPL, but once you start writing actual programs you'll want to define new packages so different programs loaded into the same Lisp environment don't stomp on each other's names. And when you write libraries that you intend to use in different contexts, you'll want to define separate packages and then export the symbols that make up the libraries' public APIs.

However, before you start defining packages, it's important to understand one thing about what packages do *not* do. Packages don't provide direct control over who can call what function or access what variable. They provide you with basic control over namespaces by controlling how the reader translates textual names into symbol objects, but it isn't until later, in the evaluator, that the symbol is interpreted as the name of a function or variable or whatever else. Thus, it doesn't make sense to talk about exporting a function or a variable from a package. You can export symbols to make certain names easier to refer to, but the package system doesn't allow you to restrict how those names are used.[4]

With that in mind, you can start looking at how to define packages and tie them together. You define new packages with the macro **DEFPACKAGE**, which allows you to not only create the package but to specify what packages it uses, what symbols it exports, and what symbols it imports from other packages and to resolve conflicts by creating shadowing symbols.[5]

I'll describe the various options in terms of how you might use packages while writing a program that organizes e-mail messages into a searchable database. The program is purely hypothetical, as are the libraries I'll refer to--the point is to look at how the packages used in such a program might be structured.

The first package you'd need is one to provide a namespace for the application--you want to be able to name your functions, variables, and so on, without having to worry about name collisions with unrelated code. So you'd define a new package with **DEFPACKAGE**.

If the application is simple enough to be written with no libraries beyond the facilities provided by the language itself, you could define a simple package like this:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp))
```

This defines a package, named `COM.GIGAMONKEYS.EMAIL-DB`, that inherits all the symbols exported by the `COMMON-LISP` package.[6]

You actually have several choices of how to represent the names of packages and, as you'll see, the names of symbols in a **DEFPACKAGE**. Packages and symbols are named with strings. However, in a **DEFPACKAGE** form, you can specify the names of packages and symbols with *string designators*. A string designator is either a string, which designates itself; a symbol, which designates its name; or a character, which designates a one-character string containing just the character. Using keyword symbols, as in the previous **DEFPACKAGE**, is a common style that allows you to write the names in lowercase--the reader will convert the names to uppercase for you. You could also write the **DEFPACKAGE** with strings, but then you have to write them in all uppercase, because the true names of most symbols and packages are in fact uppercase because of the case conversion performed by the reader.[7]

```
(defpackage "COM.GIGAMONKEYS.EMAIL-DB"
  (:use "COMMON-LISP"))
```

You could also use nonkeyword symbols--the names in **DEFPACKAGE** aren't evaluated--but then the very act of reading the **DEFPACKAGE** form would cause those symbols to be interned in the current package, which at the very least will pollute that namespace and may also cause problems later if you try to use the package.[8]

To read code in this package, you need to make it the current package with the **IN-PACKAGE** macro:

```
(in-package :com.gigamonkeys.email-db)
```

If you type this expression at the REPL, it will change the value of ***PACKAGE***, affecting how the REPL reads subsequent expressions, until you change it with another call to **IN-PACKAGE**. Similarly, if you include an **IN-PACKAGE** in a file that's loaded with **LOAD** or compiled with

COMPILE-FILE, it will change the package, affecting the way subsequent expressions in the file are read.[9]

With the current package set to the COM.GIGAMONKEYS.EMAIL-DB package, other than names inherited from the COMMON-LISP package, you can use any name you want for whatever purpose you want. Thus, you could define a new hello-world function that could coexist with the hello-world function previously defined in COMMON-LISP-USER. Here's the behavior of the existing function:

```
CL-USER> (hello-world)
hello, world
NIL
```

Now you can switch to the new package using **IN-PACKAGE**.[10] Notice how the prompt changes--the exact form is determined by the development environment, but in SLIME the default prompt consists of an abbreviated version of the package name.

```
CL-USER> (in-package :com.gigamonkeys.email-db)
#<The COM.GIGAMONKEYS.EMAIL-DB package>
EMAIL-DB>
```

You can define a new hello-world in this package:

```
EMAIL-DB> (defun hello-world () (format t "hello from EMAIL-DB package~%"))
HELLO-WORLD
```

And test it, like this:

```
EMAIL-DB> (hello-world)
hello from EMAIL-DB package
NIL
```

Now switch back to CL-USER.

```
EMAIL-DB> (in-package :cl-user)
#<The COMMON-LISP-USER package>
CL-USER>
```

And the old function is undisturbed.

```
CL-USER> (hello-world)
hello, world
NIL
```

Packaging Reusable Libraries

While working on the e-mail database, you might write several functions related to storing and retrieving text that don't have anything in particular to do with e-mail. You might realize that those functions could be useful in other programs and decide to repackage them as a library. You should define a new package, but this time you'll export certain names to make them available to other packages.

```
(defpackage :com.gigamonkeys.text-db
  (:use :common-lisp)
  (:export :open-db
           :save
           :store))
```

Again, you use the `COMMON-LISP` package, because you'll need access to standard functions within `COM.GIGAMONKEYS.TEXT-DB`. The `:export` clause specifies names that will be external in `COM.GIGAMONKEYS.TEXT-DB` and thus accessible in packages that `:use` it. Therefore, after you've defined this package, you can change the definition of the main application package to the following:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db))
```

Now code written in `COM.GIGAMONKEYS.EMAIL-DB` can use unqualified names to refer to the exported symbols from both `COMMON-LISP` and `COM.GIGAMONKEYS.TEXT-DB`. All other names will continue to be interned directly in the `COM.GIGAMONKEYS.EMAIL-DB` package.

Importing Individual Names

Now suppose you find a third-party library of functions for manipulating e-mail messages. The names used in the library's API are exported from the package `COM.ACME.EMAIL`, so you could `:use` that package to get easy access to those names. But suppose you need to use only one function from this library, and other exported symbols conflict with names you already use (or plan to use) in our own code.[11] In this case, you can import the one symbol you need with an `:import-from` clause in the **`DEFPACKAGE`**. For instance, if the name of the function you want to use is `parse-email-address`, you can change the **`DEFPACKAGE`** to this:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db)
  (:import-from :com.acme.email :parse-email-address))
```

Now anywhere the name `parse-email-address` appears in code read in the `COM.GIGAMONKEYS.EMAIL-DB` package, it will be read as the symbol from `COM.ACME.EMAIL`. If you need to import more than one symbol from a single package, you can include multiple names after the package name in a single `:import-from` clause. A **`DEFPACKAGE`** can also include multiple `:import-from` clauses in order to import symbols from different packages.

Occasionally you'll run into the opposite situation--a package may export a bunch of names you want to use and a few you don't. Rather than listing all the symbols you *do* want to use in an `:import-from` clause, you can instead `:use` the package and then list the names you *don't* want to inherit in a `:shadow` clause. For instance, suppose the `COM.ACME.TEXT` package exports a bunch of names of functions and classes used in text processing. Further suppose that most of these functions and classes are ones you'll want to use in your code, but one of the

names, `build-index`, conflicts with a name you've already used. You can make the `build-index` from `COM.ACME.TEXT` inaccessible by shadowing it.

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index))
```

The `:shadow` clause causes a new symbol named `BUILD-INDEX` to be created and added directly to `COM.GIGAMONKEYS.EMAIL-DB`'s name-to-symbol map. Now if the reader reads the name `BUILD-INDEX`, it will translate it to the symbol in `COM.GIGAMONKEYS.EMAIL-DB`'s map, rather than the one that would otherwise be inherited from `COM.ACME.TEXT`. The new symbol is also added to a *shadowing symbols list* that's part of the `COM.GIGAMONKEYS.EMAIL-DB` package, so if you later use another package that also exports a `BUILD-INDEX` symbol, the package system will know there's no conflict--that you want the symbol from `COM.GIGAMONKEYS.EMAIL-DB` to be used rather than any other symbols with the same name inherited from other packages.

A similar situation can arise if you want to use two packages that export the same name. In this case the reader won't know which inherited name to use when it reads the textual name. In such situations you must resolve the ambiguity by shadowing the conflicting names. If you don't need to use the name from either package, you could shadow the name with a `:shadow` clause, creating a new symbol with the same name in your package. But if you actually want to use one of the inherited symbols, then you need to resolve the ambiguity with a `:shadowing-import-from` clause. Like an `:import-from` clause, a `:shadowing-import-from` clause consists of a package name followed by the names to import from that package. For instance, if `COM.ACME.TEXT` exports a name `SAVE` that conflicts with the name exported from `COM.GIGAMONKEYS.TEXT-DB`, you could resolve the ambiguity with the following **`DEFPACKAGE`**:

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index)
  (:shadowing-import-from :com.gigamonkeys.text-db :save))
```

Packaging Mechanics

That covers the basics of how to use packages to manage namespaces in several common situations. However, another level of how to use packages is worth discussing--the raw mechanics of how to organize code that uses different packages. In this section I'll discuss a few

rules of thumb about how to organize code--where to put your **DEFPACKAGE** forms relative to the code that uses your packages via **IN-PACKAGE**.

Because packages are used by the reader, a package must be defined before you can **LOAD** or **COMPILE-FILE** a file that contains an **IN-PACKAGE** expression switching to that package. Packages also must be defined before other **DEFPACKAGE** forms can refer to them. For instance, if you're going to `:use COM.GIGAMONKEYS.TEXT-DB` in `COM.GIGAMONKEYS.EMAIL-DB`, then `COM.GIGAMONKEYS.TEXT-DB`'s **DEFPACKAGE** must be evaluated before the **DEFPACKAGE** of `COM.GIGAMONKEYS.EMAIL-DB`.

The best first step toward making sure packages exist when they need to is to put all your **DEFPACKAGE**s in files separate from the code that needs to be read in those packages. Some folks like to create a `foo-package.lisp` file for each individual package, and others create a single `packages.lisp` that contains all the **DEFPACKAGE** forms for a group of related packages. Either approach is reasonable, though the one-file-per-package approach also requires that you arrange to load the individual files in the right order according to the interpackage dependencies.

Either way, once all the **DEFPACKAGE** forms have been separated from the code that will be read in the packages they define, you can arrange to **LOAD** the files containing the **DEFPACKAGE**s before you compile or load any of the other files. For simple programs you can do this by hand: simply **LOAD** the file or files containing the **DEFPACKAGE** forms, possibly compiling them with **COMPILE-FILE** first. Then **LOAD** the files that use those packages, again optionally compiling them first with **COMPILE-FILE**. Note, however, that the packages don't exist until you **LOAD** the package definitions, either the source or the files produced by **COMPILE-FILE**. Thus, if you're compiling everything, you must still **LOAD** all the package definitions before you can **COMPILE-FILE** any files to be read in the packages.

Doing these steps by hand will get tedious after a while. For simple programs you can automate the steps by writing a file, `load.lisp`, that contains the appropriate **LOAD** and **COMPILE-FILE** calls in the right order. Then you can just **LOAD** that file. For more complex programs you'll want to use a *system definition* facility to manage loading and compiling files in the right order.[12]

The other key rule of thumb is that each file should contain exactly one **IN-PACKAGE** form, and it should be the first form in the file other than comments. Files containing **DEFPACKAGE** forms should start with `(in-package "COMMON-LISP-USER")`, and all other files should contain an **IN-PACKAGE** of one of your packages.

If you violate this rule and switch packages in the middle of a file, you'll confuse human readers who don't notice the second **IN-PACKAGE**. Also, many Lisp development environments, particularly Emacs-based ones such as SLIME, look for an **IN-PACKAGE** to determine the

package they should use when communicating with Common Lisp. Multiple **`IN-PACKAGE`** forms per file may confuse these tools as well.

On the other hand, it's fine to have multiple files read in the same package, each with an identical **`IN-PACKAGE`** form. It's just a matter of how you like to organize your code.

The other bit of packaging mechanics has to do with how to name packages. Package names live in a flat namespace--package names are just strings, and different packages must have textually distinct names. Thus, you have to consider the possibility of conflicts between package names. If you're using only packages you developed yourself, then you can probably get away with using short names for your packages. But if you're planning to use third-party libraries or to publish your code for use by other programmers, then you need to follow a naming convention that will minimize the possibility of name collisions between different packages. Many Lispers these days are adopting Java-style names, like the ones used in this chapter, consisting of a reversed Internet domain name followed by a dot and a descriptive string.

Package Gotchas

Once you're familiar with packages, you won't spend a bunch of time thinking about them. There's just not that much to them. However, a couple of gotchas that bite most new Lisp programmers make the package system seem more complicated and unfriendly than it really is.

The number-one gotcha arises most commonly when playing around at the REPL. You'll be looking at some library that defines certain interesting functions. You'll try to call one of the functions like this:

```
CL-USER> (foo)
```

and get dropped into the debugger with this error:

```
attempt to call `FOO' which is an undefined function.
   [Condition of type UNDEFINED-FUNCTION]

Restarts:
  0: [TRY-AGAIN] Try calling FOO again.
  1: [RETURN-VALUE] Return a value instead of calling FOO.
  2: [USE-VALUE] Try calling a function other than FOO.
  3: [STORE-VALUE] Setf the symbol-function of FOO and call it again.
  4: [ABORT] Abort handling SLIME request.
  5: [ABORT] Abort entirely from this (lisp) process.
```

Ah, of course--you forgot to use the library's package. So you quit the debugger and try to **`USE-PACKAGE`** the library's package in order to get access to the name `FOO` so you can call the function.

```
CL-USER> (use-package :foolib)
```

But that drops you back into the debugger with this error message:

```
Using package `FOOLIB' results in name conflicts for these symbols: FOO
   [Condition of type PACKAGE-ERROR]

Restarts:
  0: [CONTINUE] Unintern the conflicting symbols from the `COMMON-LISP-USER' package.
  1: [ABORT] Abort handling SLIME request.
  2: [ABORT] Abort entirely from this (lisp) process.
```

Huh? The problem is the first time you called `foo`, the reader read the name `foo` and interned it in `CL-USER` before the evaluator got hold of it and discovered that this newly interned symbol isn't the name of a function. This new symbol then conflicts with the symbol of the same name exported from the `FOOLIB` package. If you had remembered to **`USE-PACKAGE`** `FOOLIB` before you tried to call `foo`, the reader would have read `foo` as the inherited symbol and not interned a `foo` symbol in `CL-USER`.

However, all isn't lost, because the first restart offered by the debugger will patch things up in just the right way: it will unintern the `foo` symbol from `COMMON-LISP-USER`, putting the `CL-USER` package back to the state it was in before you called `foo`, allowing the **`USE-PACKAGE`** to proceed and allowing for the inherited `foo` to be available in `CL-USER`.

This kind of problem can also occur when loading and compiling files. For instance, if you defined a package, `MY-APP`, for code that was going to use functions with names from the `FOOLIB` package, but forgot to `:use FOOLIB`, when you compile the files with an `(in-package :my-app)` in them, the reader will intern new symbols in `MY-APP` for the names that were supposed to be read as symbols from `FOOLIB`. When you try to run the compiled code, you'll get undefined function errors. If you then try to redefine the `MY-APP` package to `:use FOOLIB`, you'll get the conflicting symbols error. The solution is the same: select the restart to unintern the conflicting symbols from `MY-APP`. You'll then need to recompile the code in the `MY-APP` package so it will refer to the inherited names.

The next gotcha is essentially the reverse of the first gotcha. In this case, you'd have defined a package--again, let's say it's `MY-APP`--that uses another package, say, `FOOLIB`. Now you start writing code in the `MY-APP` package. Although you used `FOOLIB` in order to be able to refer to the `foo` function, `FOOLIB` may export other symbols as well. If you use one of those exported symbols--say, `bar`--as the name of a function in your own code, Lisp won't complain. Instead, the name of your function will be the symbol exported by `FOOLIB`, which will clobber the definition of `bar` from `FOOLIB`.

This gotcha is more insidious because it doesn't cause an error--from the evaluator's point of view it's just being asked to associate a new function with an old name, something that's perfectly legal. It's suspect only because the code doing the redefining was read with a different value for **`*PACKAGE*`** than the name's package. But the evaluator doesn't necessarily know that. However, in most Lisps you'll get an warning about "`redefining BAR, originally defined in`?". You should heed those warnings. If

you clobber a definition from a library, you can restore it by reloading the library code with **LOAD**.[13]

The last package-related gotcha is, by comparison, quite trivial, but it bites most Lisp programmers at least a few times: you define a package that uses `COMMON-LISP` and maybe a few libraries. Then at the REPL you change to that package to play around. Then you decide to quit Lisp altogether and try to call `(quit)`. However, `quit` isn't a name from the `COMMON-LISP` package--it's defined by the implementation in some implementation-specific package that happens to be used by `COMMON-LISP-USER`. The solution is simple--change packages back to `CL-USER` to quit. Or use the SLIME REPL shortcut `quit`, which will also save you from having to remember that in certain Common Lisp implementations the function to quit is `exit`, not `quit`.

You're almost done with your tour of Common Lisp. In the next chapter I'll discuss the details of the extended **LOOP** macro. After that, the rest of the book is devoted to "practicals": a spam filter, a library for parsing binary files, and various parts of a streaming MP3 server with a Web interface.

---

[1]The kind of programming that relies on a symbol data type is called, appropriately enough, *symbolic* computation. It's typically contrasted to *numeric* programming. An example of a primarily symbolic program that all programmers should be familiar with is a compiler--it treats the text of a program as symbolic data and translates it into a new form.

[2]Every package has one official name and zero or more *nicknames* that can be used anywhere you need to use the package name, such as in package-qualified names or to refer to the package in a **DEFPACKAGE** or **IN-PACKAGE** form.

[3]`COMMON-LISP-USER` is also allowed to provide access to symbols exported by other implementation-defined packages. While this is intended as a convenience for the user--it makes implementation-specific functionality readily accessible--it can also cause confusion for new Lispers: Lisp will complain about an attempt to redefine some name that isn't listed in the language standard. To see what packages `COMMON-LISP-USER` inherits symbols from in a particular implementation, evaluate this expression at the REPL:

```
(mapcar #'package-name (package-use-list :cl-user))
```

And to find out what package a symbol came from originally, evaluate this:

```
(package-name (symbol-package 'some-symbol))
```

with `some-symbol` replaced by the symbol in question. For instance:

```
(package-name (symbol-package 'car)) ==> "COMMON-LISP"
(package-name (symbol-package 'foo)) ==> "COMMON-LISP-USER"
```

Symbols inherited from implementation-defined packages will return some other value.

[4]This is different from the Java package system, which provides a namespace for classes but is also involved in Java's access control mechanism. The non-Lisp language with a package system most like Common Lisp's packages is Perl.

[5]All the manipulations performed by **DEFPACKAGE** can also be performed with functions that man- ipulate package objects. However, since a package generally needs to be fully defined before it can be used, those functions are rarely used. Also, **DEFPACKAGE** takes care of performing all the package manipulations in the right order--for instance, **DEFPACKAGE** adds symbols to the shadowing list before it tries to use the used packages.

[6]In many Lisp implementations the `:use` clause is optional if you want only to `:use` `COMMON-LISP`--if it's omitted, the package will automatically inherit names from an implementation-defined list of packages that will usually include `COMMON-LISP`. However, your code will be more portable if you always explicitly specify the packages you want to `:use`. Those who are averse to typing can use the package's nickname and write `(:use :cl)`.

[7]Using keywords instead of strings has another advantage--Allegro provides a "modern mode" Lisp in which the reader does no case conversion of names and in which, instead of a **`COMMON-LISP`** package with uppercase names, provides a `common-lisp` package with lowercase names. Strictly speaking, this Lisp isn't a conforming Common Lisp since all the names in the standard are defined to be uppercase. But if you write your **`DEFPACKAGE`** forms using keyword symbols, they will work both in Common Lisp and in this near relative.

[8]Some folks, instead of keywords, use uninterned symbols, using the `#:` syntax.

```
(defpackage #:com.gigamonkeys.email-db
  (:use #:common-lisp))
```

This saves a tiny bit of memory by not interning any symbols in the keyword package--the symbol can become garbage after **`DEFPACKAGE`** (or the code it expands into) is done with it. However, the difference is so slight that it really boils down to a matter of aesthetics.

[9]The reason to use **`IN-PACKAGE`** instead of just **`SETF`**ing **`*PACKAGE*`** is that **`IN-PACKAGE`** expands into code that will run when the file is compiled by **`COMPILE-FILE`** as well as when the file is loaded, changing the way the reader reads the rest of the file during compilation.

[10]In the REPL buffer in SLIME you can also change packages with a REPL shortcut. Type a comma, and then enter `change-package` at the `Command:` prompt.

[11]During development, if you try to `:use` a package that exports a symbol with the same name as a symbol already interned in the using package, Lisp will signal an error and typically offer you a restart that will unintern the offending symbol from the using package. For more on this, see the section "Package Gotchas."

[12]The code for the "Practical" chapters, available from this book's Web site, uses the ASDF system definition library. ASDF stands for Another System Definition Facility.

[13]Some Common Lisp implementations, such as Allegro and SBCL, provide a facility for "locking" the symbols in a particular package so they can be used in defining forms such as **`DEFUN`**, **`DEFVAR`**, and **`DEFCLASS`** only when their home package is the current package.