The Complete Idiot's Guide to Common Lisp Packages[1]

Erann Gat

1. Introduction

When coding a large project with multiple programmers two different programmers will often want to use the same name for two different purposes. It is possible to solve this problem using a naming convention, e.g. Bob prefixes all his names with "BOB-" and Jane prefixes all her names with "JANE-". This is in fact how Scheme addresses this problem (or fails to address it as the case may be).

Common Lisp provides a language mechanism called *packages* for segregating namespaces. Here's an example of how packages work:

```
? (make-package :bob)
#<Package "BOB">
? (make-package :jane)
#<Package "JANE">
? (in-package bob)
#<Package "BOB">
? (defun foo () "This is Bob's foo")
FOO
? (in-package jane)
#<Package "JANE">
? (defun foo () "This is Jane's foo")
FOO
? (foo)
"This is Jane's foo"
? (in-package bob)
#<Package "BOB">
? (foo)
"This is Bob's foo"
?
```

(NOTE: Code examples are cut-and-pasted from Macintosh Common Lisp (MCL). The command prompt in MCL is a question mark.)

Bob and Jane each have a function named FOO that does something different, and they don't conflict with each other.

[1] Version 1.2

What if Bob wants to use a function written by Jane? There are several ways he can do it. One is to use a special syntax to indicate that a different package is to be used:

```
? (in-package bob)
#<Package "BOB">
? (jane::foo)
"This is Jane's foo"
?
```

Another is to import what he wants to use into his own package. Of course, he won't want to import Jane's FOO function because then it would conflict with his own, but if Jane had a BAZ function that he wanted to use by simply typing (BAZ) instead of (JANE::BAZ) he could do it like this:

```
? (in-package jane)
#<Package "JANE">
? (defun baz () "This is Jane's baz")
BAZ
? (in-package bob)
#<Package "BOB">
? (import 'jane::baz)
T
? (baz)
"This is Jane's baz"
?
```

Alas, things don't always go quite so smoothly:

```
? (in-package jane)
#<Package "JANE">
? (defun bar () "This is Jane's bar")
BAR
? (in-package bob)
#<Package "BOB">
? (bar)
> Error: Undefined function BAR called with arguments () .
> While executing: "Unknown"
> Type Command-/ to continue, Command-. to abort.
> If continued: Retry applying BAR to NIL.
See the Restarts… menu item for further choices.
1 >
; Oops!  Forgot to import.
Aborted
? (import 'jane::bar)
> Error: Importing JANE::BAR to #<Package "BOB"> would conflict with
symbol BAR .
> While executing: CCL::IMPORT-1
> Type Command-/ to continue, Command-. to abort.
> If continued: Ignore attempt to import JANE::BAR to #<Package "BOB">.
See the Restarts… menu item for further choices.
1 >
; Huh?
```

To understand why this happened, what to do about it, and many of the other subtleties and surprises of packages, it is important to understand what packages actually do and how they work. For example, it is important to understand that when you type (import 'jane::foo) you are importing the *symbol* JANE::FOO, not the function associated with that symbol. It is important to understand the difference, and so we have to start with a review of some basic Lisp concepts.

2. Symbols, Values, and the READ-EVAL-PRINT Loop

Lisp operates in a READ-EVAL-PRINT loop. Most of the interesting stuff happens in the EVAL phase, but when it comes to packages interesting stuff happens in all three phases, and it's important to understand what happens when. In particular, some of the processing related to packages can change the state of the Lisp system at READ time, which can in turn result in some surprising (and often annoying) behavior, like the last example in the previous section.

A package is a collection of Lisp *symbols*, so to understand packages you first have to understand symbols. A symbol is a perfectly ordinary Lisp data structure, just as lists, numbers, strings, etc. are. There are built-in Lisp functions for creating and manipulating symbols. For example, there is a function called GENTEMP that creates new symbols:

```
? (gentemp)
T1
? (setq x (gentemp))
T2
? (set x 123)  ; Note the use of SET, not SETQ or SETF
123
? x
T2
? t2
123
?
```

(If you are not familiar with the SET function now would be a good time to look it up because if you don't you will be lost in short order.)

The symbols created by GENTEMP behave in all respects like symbols that you get just by typing in their names.

You have only limited control over the name of a symbol created by GENTEMP. You can pass it an optional prefix string that, but the system will add a suffix and you have to take whatever it decides to give you. If you want to make a symbol with a particular name you have to use a different function, MAKE-SYMBOL:

```
? (make-symbol "MY-SYMBOL")
#:MY-SYMBOL
?
```

Hm, that's odd. What's that funny-looking "#:" doing there?

To understand this we have to dig a little deeper into the guts of symbols.

```
? (setq symbol1 (make-symbol "MY-SYMBOL"))
#:MY-SYMBOL
? (setq symbol2 (make-symbol "MY-SYMBOL"))
#:MY-SYMBOL
? (setq symbol3 'my-symbol)
MY-SYMBOL
? (setq symbol4 'my-symbol)
MY-SYMBOL
? (eq symbol1 symbol2)
NIL
? (eq symbol3 symbol4)
T
?
```

As you see, MAKE-SYMBOL can make multiple distinct symbols that have the same name, whereas symbols that the reader gives you by typing the same name on two different occasions are the same symbol.

This property of symbol identity is very important. It is what insures that the FOO you type in one place is the same FOO as the FOO you type someplace else. If this were not so you could end up with some very weird results:

```
? (set symbol1 123)
123
? (set symbol2 456)
456
? (setq code-fragment-1 (list 'print symbol1))
(PRINT #:MY-SYMBOL)
? (setq code-fragment-2 (list 'print symbol2))
(PRINT #:MY-SYMBOL)
? (eval code-fragment-1)
123
123
? (eval code-fragment-2)
456
456
?
```

Contrast this with:

```
? (set symbol3 123)
123
? (set symbol4 456)
456
? (setq code-fragment-3 (list 'print symbol3))
(PRINT MY-SYMBOL)
? (setq code-fragment-4 (list 'print symbol4))
(PRINT MY-SYMBOL)
? (eval code-fragment-3)
456
456
? (eval code-fragment-4)
```

```
456
456
?
```

Symbols 1-4 all have the name "MY-SYMBOL" but symbols 1 and 2 are different symbols, while symbols 3 and 4 are the same symbol. How did this happen? Well, one obvious difference is that we called the function MAKE-SYMBOL to make symbols 1 and 2, while symbols 3 and 4 were made for us by the Lisp reader. Maybe the Lisp reader has a different way of making symbols than calling MAKE-SYMBOL. We can test this hypothesis:

```
? (trace make-symbol)
NIL
? 'foobaz
 Calling (MAKE-SYMBOL "FOOBAZ")
 MAKE-SYMBOL returned #:FOOBAZ
FOOBAZ
```

Nope, the reader apparently makes symbols the same way we did, by calling MAKE-SYMBOL. But wait, the symbol returned by MAKE-SYMBOL had that funny #: thing in front of it, but by the time the reader was done the #: prefix had vanished. What gives?

We can find the answer by trying the same experiment a second time with MAKE-SYMBOL still traced:

```
? 'foobaz
FOOBAZ
```

Aha! The second time we type FOOBAZ the reader doesn't call MAKE-SYMBOL. So the reader is apparently keeping a collection of all the symbols it has made, and before it makes a new one it checks that collection to see if there is already a symbol there by the same name. If there is, then it simply returns that same symbol instead of making a new one. And a symbol that is a member of such a collection loses its mysterious #: prefix.

That collection of symbols is called a *package*.

A package is a collection of Lisp symbols with the property that no two symbols in the collection have the same name.

Unfortunately, that is more or less the last aspect of packages that is simple and straightforward. From here on out things get rather hairier.

3. Interning

The act of putting a symbol into a package is called *interning* a symbol. A symbol that is a member of a package is said to be *interned* in that package. Symbols that are not members of any package are said to be *uninterned*. When an uninterned symbol is printed, it gets a #: prefix[2] to distinguish it from an interned symbol, and to warn you that just because it looks just like another symbol that you saw before it might not in fact be the same symbol.

Now, here's where things start to get a tad confusing.

There is a Lisp function called INTERN, which you might expect to add a symbol to a package, but it doesn't. That function is performed by a function called IMPORT.

```
? symbol1
#:MY-SYMBOL
? (import symbol1)
T
? symbol1
MY-SYMBOL
? (eq symbol1 'my-symbol)
T
?
```

As you can see, symbol1 has gone from being an uninterned symbol to an interned symbol. It has lost its #: prefix, and it is now EQ to the symbol MY-SYMBOL as produced by the Lisp reader.

Now, you might expect that to undo the effect of IMPORT you would call UNIMPORT, but there is no such function. (I warned you that things would not be straightforward.) To remove a symbol from a package you call UNINTERN:

```
? (unintern symbol1)
T
? symbol1
#:MY-SYMBOL
? (eq symbol1 'my-symbol)
NIL
?
```

Things are back to the way they were. Symbol 1 is now uninterned, and it is now a different symbol than the one the reader gives you. Let's put symbol1 back into our package:

```
? (import symbol1)
> Error: Importing #:MY-SYMBOL to #<Package "COMMON-LISP-USER"> would
conflict with symbol MY-SYMBOL .
> While executing: CCL::IMPORT-1
> Type Command-/ to continue, Command-. to abort.
> If continued: Ignore attempt to import #:MY-SYMBOL to #<Package
"COMMON-LISP-USER">.
See the Restarts… menu item for further choices.
```

[2] This is not quite accurate, but a good enough approximation for now. The real truth is that a #: prefix just means that a symbol has no home package, not necessarily that it is not interned in any package.

```
      1 >
```

Whoa! What happened? (It is a good exercise to try to figure out what's going on here before reading any further. You have all the information you need. Hint: try the experiment yourself with MAKE-SYMBOL traced.)

Here's what happened:

```
? (unintern 'my-symbol)
T
? (eq symbol1 'my-symbol
 Calling (MAKE-SYMBOL "MY-SYMBOL")
 MAKE-SYMBOL returned #:MY-SYMBOL
)
NIL
?
```

When we typed `(eq symbol1 'my-symbol)` the reader interned the symbol MY-SYMBOL. So when we tried to import symbol1 there was already a symbol with the same name in the package. Remember, packages maintain the invariant that there can only be one symbol with the same name in the package at any one time (that's the whole point) so when we tried to import symbol1 (whose name is also MY-SYMBOL) there was already a symbol there with the same name (having been quietly interned by the reader). This situation is called a *symbol conflict*, and it is, alas, very common.

Note, by the way, that the output from the trace of MAKE-SYMBOL above appears inside the S-expression `(eq symbol1 ‘my-symbol)`. This is because MCL rearranges the text on the screen to reflect the true order of events. Because MAKE-SYMBOL was called by the reader while processing the string "my-symbol" but before processing the close-paren, the output appears at that point. If we type: `(list 'x1 'x2 'x3 'x4)` the resulting output looks like this:

```
? (list 'x1
 Calling (MAKE-SYMBOL "X1")
 MAKE-SYMBOL returned #:X1
'x2
 Calling (MAKE-SYMBOL "X2")
 MAKE-SYMBOL returned #:X2
'x3
 Calling (MAKE-SYMBOL "X3")
 MAKE-SYMBOL returned #:X3
'x4
 Calling (MAKE-SYMBOL "X4")
 MAKE-SYMBOL returned #:X4
)
```

Here, the text formatting has been preserved to help illustrate what is happening. The text typed by the user is in **bold** type. (Note that most Lisp systems don't do this reformatting.)

There are a few more useful functions that we can go ahead and mention at this point. SYMBOL-NAME returns the name of a symbol as a string. FIND-SYMBOL takes a string and tells you if there is a symbol with that name already interned. And, finally, INTERN, which can be defined as follows:

```
(defun intern (name)
  (or (find-symbol name)
      (let ((s (make-symbol name)))
        (import s)
        s)))
```

In other words, INTERN is sort of the opposite of SYMBOL-NAME. SYMBOL-NAME takes a symbol and returns the string that is that symbol's name. INTERN takes a string and returns the symbol whose name is that string. INTERN is the function that READ uses to make symbols.

4. Which Package?

The functions that operate on packages like FIND-SYMBOL, IMPORT and INTERN by default operate on a package that is the value of the global variable *PACKAGE*, also known as the *current package*. Like symbols, packages have names, and no two packages can have the same name. Packages, like symbols, are also ordinary Lisp data structures. There are functions PACKAGE-NAME and FIND-PACKAGE which operate analogously to SYMBOL-NAME and FIND-SYMBOL, except of course, that FIND-PACKAGE doesn't take a package argument. (It's as if there is one global meta-package for package names.)

As we've already seen, we can make new packages by calling MAKE-PACKAGE, and set the current package by calling IN-PACKAGE. (Note that IN-PACKAGE is not a function but a macro that does not evaluate its argument.)

You can access symbols that are not in the current package by using a special syntax: the package name followed by two colons followed by the symbol name. This is handy if you want to use a symbol without importing it. For example, if Bob wanted to use Jane's FOO function he could type `JANE::FOO`.

4.1 Home packages

One of the invariants that Common Lisp tries to maintain is a property called print-read consistency. This property says that if you print a symbol, and then read the resulting printed representation of that symbol, the result is the same symbol, with two caveats: 1) this does not apply to uninterned symbols, and 2) it applies only if you refrain from certain "dangerous" actions. We'll cover what those are in a moment.

To maintain print-read consistency, some symbols need to be printed with their package qualifier. For example:

```
? (in-package jane)
#<Package "JANE">
? 'foo
```

```
FOO
? 'jane::foo
FOO
? (in-package bob)
#<Package "BOB">
? 'foo
FOO
? 'jane::foo
JANE::FOO
? 'bob::foo
FOO
?
```

Obviously one of the “dangerous actions” that allows print-read consistency to be violated is calling IN-PACKAGE.

Now, consider the following situation:

```
? (in-package bob)
#<Package "BOB">
? (unintern 'foo)
T
? (import 'jane::foo)
T
? (make-package :charlie)
#<Package "CHARLIE">
? (in-package charlie)
#<Package "CHARLIE">
? 'bob::foo
JANE::FOO
?
```

Here we have a symbol FOO which is interned in both the JANE and the BOB packages. That one symbol is therefore accessible as both JANE::FOO and BOB::FOO. How does the system decide which printed representation to use when the symbol is printed from the CHARLIE package, in which the symbol is not interned?

It turns out that every symbol keeps track of a single package called its *home package*, which is usually the first package in which that symbol was interned (but there are exceptions). When a symbol needs to be printed with a package qualifier, it uses the qualifier from its home package. You can query a symbol for its home package using the function SYMBOL-PACKAGE.

```
? (symbol-package 'bob::foo)
#<Package "JANE">
?
```

Note that it is possible to make symbols without home packages. Uninterned symbols, for examples, have no home packages. But it is also possible to make interned symbols with no home packages. For example:

```
? (in-package jane)
```

```
#<Package "JANE">
? 'weird-symbol
WEIRD-SYMBOL
? (in-package bob)
#<Package "BOB">
? (import 'jane::weird-symbol)
T
? (in-package jane)
#<Package "JANE">
? (unintern 'weird-symbol)
T
? (in-package bob)
#<Package "BOB">
? 'weird-symbol
WEIRD-SYMBOL
? (symbol-package 'weird-symbol)
NIL
? (in-package jane)
#<Package "JANE">
? 'bob::weird-symbol
#:WEIRD-SYMBOL
```

Such things are best avoided.

5. Export and Use-package

Suppose Jane and Bob are collaborating on a software development project, each working in their own package to avoid conflicts. Jane writes:

```
? (in-package jane)
#<Package "JANE">
? (defclass jane-class () (slot1 slot2 slot3))
#<STANDARD-CLASS JANE-CLASS>
```

Now, imagine that Bob wants to use JANE-CLASS. He writes:

```
? (in-package bob)
#<Package "BOB">
? (import 'jane::jane-class)
T
? (make-instance 'jane-class)
#<JANE-CLASS #x130565E>
?
```

So far so good. Now he tries:

```
? (setq jc1 (make-instance 'jane-class))
#<JANE-CLASS #x130BA96>
? (slot-value jc1 'slot1)
> Error: #<JANE-CLASS #x130BA96> has no slot named SLOT1.
> While executing: #<CCL::STANDARD-KERNEL-METHOD SLOT-MISSING (T T T T)>
```

What happened? Well, JANE-CLASS was defined in the JANE package, so the slot names are symbols interned in that package. But Bob tried to access an instance of that class using a symbol interned in the BOB package. In other words, JANE-CLASS has a slot named JANE::SLOT1, and Bob tried to access a slot name BOB::SLOT1, and there is no such slot.

What Bob would really like to do is import all the symbols that are "associated" with JANE-CLASS, that is, all the slot names, method names, etc. etc. How can he know what all those symbols are? He could examine Jane's code and try to figure it out for himself, but that would lead to many problems, not least of which is that he might decide to import a symbol that Jane didn't want Bob to mess with (remember, separating Jane's symbols from the effects of Bob's meddling is the whole point of having packages to being with).

A better solution would be for Jane to assemble a list of symbols that Bob should import in order to use her software. Then Jane could do something like:

```
? (defvar *published-symbols* '(jane-class slot1 slot2 slot3))
*PUBLISHED-SYMBOLS*
?
```

And Bob could then do `(import jane::*published-symbols*)`.

Common Lisp provides a standard mechanism for doing this. Every package maintains a list of symbols that are intended to be used by other packages. This list is called the *exported symbol list* of that package, and to add a symbol to that package you use the function EXPORT. To remove a symbol from a package's exported symbol list you use (as you might expect this time) UNEXPORT. To import all of the exported symbols in a package you call USE-PACKAGE. To unimport them all you call UNUSE-PACKAGE.

There are two things to note about exporting symbols.

First, a symbol can be exported from any package in which that symbol is interned, not just its home package. A symbol also does not have to be exported from its home package in order to be exported from some other package in which that symbol is interned.

Second, a symbol that is exported from its home package uses only one colon instead of two when printed with its package qualifier. This is to alert you to the fact that the symbol is exported from its home package (and so you might want to USE-PACKAGE the symbol's package instead of IMPORTing the symbol), and also to discourage you from using unexported symbols by forcing you to type an extra colon in order to access them. (No, I am not kidding.)

6. Shadowing

There is one last gotcha: using USE-PACKAGE is slightly different from IMPORTing all the exported symbols in the package. When you IMPORT a symbol you can undo the effect of the IMPORT using UNINTERN. You cannot use UNINTERN to (partially) undo the effect of a USE-PACKAGE. For example:

```
? (in-package jane)
#<Package "JANE">
? (export '(slot1 slot2 slot3))
T
? (in-package bob)
#<Package "BOB">
? (use-package 'jane)
T
? (symbol-package 'slot1)
#<Package "JANE">
? (unintern 'slot1)
NIL
? (symbol-package 'slot1)
#<Package "JANE">
?
```

This is a problem, because it would seem to make it impossible to use two different packages that export a symbol with the same name. For example, suppose you wanted to use two packages P1 and P2 in a third package named MYPACKAGE, and both P1 and P2 exported a symbol named X. If you just tried to USE-PACKAGE both P1 and P2 you would get a name conflict because Lisp would have no way of knowing whether X in MYPACKAGE should now be P1:X or P2:X.

To resolve such name conflicts every package maintains what is called a *shadowing symbols list*. The shadowing symbols list is a list of symbols that shadow or *override* any symbols would normally become visible in that package as a result of calling USE-PACKAGE.

There are two ways to add symbols to a package's shadowing symbols list, SHADOW and SHADOWING-IMPORT. SHADOW is used to add symbols that are in the package to the shadowing symbols list. SHADOWING-IMPORT is used to add symbols that are in some *other* package.

For example:

```
? (in-package p1)
#<Package "P1">
? (export '(x y z))
T
? (in-package p2)
#<Package "P2">
? (export '(x y z))
T
? (in-package bob)
T
? (use-package 'p1)
T
? (use-package 'p2)
> Error: Using #<Package "P2"> in #<Package "BOB">
>        would cause name conflicts with symbols inherited
>        by that package:
>        Z  P2:Z
>       Y  P2:Y
>       X  P2:X
. . .
1 >
Aborted
? (unuse-package 'p1)
T
? (shadow 'x)
T
? (shadowing-import 'p1:y)
T
? (shadowing-import 'p2:z)
T
? (use-package 'p1)
T
? (use-package 'p2)
T
? (symbol-package 'x)
#<Package "BOB">
? (symbol-package 'y)
#<Package "P1">
? (symbol-package 'z)
#<Package "P2">
?
```

To undo the effect of SHADOW or SHADOWING-IMPORT use UNINTERN.

Note that UNINTERN (and many, many other surprising things) can result in name conflicts unexpectedly appearing. The most common cause of an unexpected name conflict is usually inadvertently interning a symbol in a package by typing it at the reader without paying close attention to which package you were in at the time.

7. DEFPACKAGE

Now that you've learned all about the myriad functions and macros that can be used to manipulate packages you shouldn't really be using any of them. Instead, all of the functionality of IMPORT, EXPORT, SHADOW, etc. is all rolled up in a single macro called DEFPACKAGE, which is what you should use for real (non-prototype) code.

I'm not going to try to explain DEFPACKAGE here because now that you understand the basic concepts of packages you should just be able to read the documentation in the hyperspec and understand it. (There are also a lot of other goodies, like DO-SYMBOLS and WITH-PACKAGE-ITERATOR, in the hyperspec that you should now be able to understand on your own.)

One caveat about using DEFPACKAGE though: note that most of the arguments to DEFPACKAGE are *string designators*, not symbols. This means that they can be symbols, but if you choose to use symbols then those symbols will get interned in the current package, whatever it may be, at the time that the DEFPACKAGE form is read. This often leads to undesirable consequences, so it's a good idea to get into the habit of using keywords or strings in your DEFPACKAGE forms.

8. Final Thoughts

The most important thing to understand about packages is that they are fundamentally a part of the Lisp reader and not the evaluator. Once you wrap your brain around that idea, everything else will fall into place. Packages control how the reader maps strings onto symbols (and how PRINT maps symbols onto strings), nothing else. In particular, packages have nothing to do with the functions, values, property lists, etc. that might or might not be associated with any particular symbol.

Note in particular (this has nothing to do with packages but newcomers often get confused by this anyway) that both symbols and function objects can be used as functions, but they behave slightly differently. For example:

```
? (defun foo () "Original foo")
FOO
? (setf f1 'foo)
FOO
? (setf f2 #'foo)
#<Compiled-function FOO #xEB3446>
? (defun demo ()
     (list (funcall f1) (funcall f2)
           (funcall #'foo) (funcall #.#'foo)))
;Compiler warnings :
;   Undeclared free variable F1, in DEMO.
;   Undeclared free variable F2, in DEMO.
DEMO
? (demo)
("Original foo" "Original foo" "Original foo" "Original foo")
? (defun foo () "New foo")
FOO
? (demo)
("New foo" "Original foo" "New foo" "Original foo")
?
```

In this example we have two variables, F1 and F2. The value of F1 is the symbol FOO. The value of F2 is the function object that was in the symbol-function slot of the symbol FOO at the time F2 was assigned.

Note that when FOO is redefined, the effect of calling the symbol FOO is to get the new behavior, whereas calling the function object produces the old behavior.

Explaining the second and third result in the list is left as an exercise for the reader (no pun intended).