LISP - PACKAGES

In general term of programming languages, a package is designed for providing a way to keep one set of names separate from another. The symbols declared in one package will not conflict with the same symbols declared in another. This way packages reduce the naming conflicts between independent code modules.

The LISP reader maintains a table of all the symbols it has found. When it finds a new character sequence, it creates a new symbol and stores in the symbol table. This table is called a package.

The current package is referred by the special variable *package*.

There are two predefined packages in LISP:

- **common-lisp** - it contains symbols for all the functions and variables defined.
- **common-lisp-user** - it uses the common-lisp package and all other packages with editing and debugging tools; it is called cl-user in short

Package Functions in LISP

The following table provides most commonly used functions used for creating, using and manipulating packages:

SL No	Functions and Descriptions
1	**make-package** *package-name* &key :nicknames :use It creates and returns a new package with the specified package name.
2	**in-package** *package-name* &key :nicknames :use Makes the package current.
3	**in-package** *name* This macro causes *package* to be set to the package named name, which must be a symbol or string.
4	**find-package** *name* It searches for a package. The package with that name or nickname is returned; if no such package exists, find-package returns nil
5	**rename-package** *package new-name* &optional *new-nicknames* it renames a package.
6	**list-all-packages** This function returns a list of all packages that currently exist in the Lisp system.

7

delete-package *package*

it deletes a package

Creating a LISP Package

The **defpackage** function is used for creating an user defined package. It has the following syntax:

```
(defpackage :package-name
   (:use :common-lisp ...)
   (:export :symbol1 :symbol2 ...)
)
```

Where,

- package-name is the name of the package.
- The :use keyword specifies the packages that this package needs, i.e., packages that define functions used by code in this package.
- The :export keyword specifies the symbols that are external in this package.

The **make-package** function is also used for creating a package. The syntax for this function is:

```
make-package package-name &key :nicknames :use
```

the arguments and keywords has same meaning as before.

Using a Package

Once you have created a package, you can use the code in this package, by making it the current package. The **in-package** macro makes a package current in the environment.

Example

Create a new source code file named main.lisp and type the following code in it.

```
(make-package :tom)
(make-package :dick)
(make-package :harry)
(in-package tom)
(defun hello ()
   (write-line "Hello! This is Tom's Tutorials Point")
)
(hello)
(in-package dick)
(defun hello ()
   (write-line "Hello! This is Dick's Tutorials Point")
)
(hello)
(in-package harry)
(defun hello ()
   (write-line "Hello! This is Harry's Tutorials Point")
)
(hello)
(in-package tom)
(hello)
(in-package dick)
(hello)
(in-package harry)
(hello)
```

When you execute the code, it returns the following result:

```
Hello! This is Tom's Tutorials Point
Hello! This is Dick's Tutorials Point
Hello! This is Harry's Tutorials Point
```

Deleting a Package

The **delete-package** macro allows you to delete a package. The following example demonstrates this:

Example

Create a new source code file named main.lisp and type the following code in it.

```
(make-package :tom)
(make-package :dick)
(make-package :harry)
(in-package tom)
(defun hello ()
   (write-line "Hello! This is Tom's Tutorials Point")
)
(in-package dick)
(defun hello ()
   (write-line "Hello! This is Dick's Tutorials Point")
)
(in-package harry)
(defun hello ()
   (write-line "Hello! This is Harry's Tutorials Point")
)
(in-package tom)
(hello)
(in-package dick)
(hello)
(in-package harry)
(hello)
(delete-package tom)
(in-package tom)
(hello)
```

When you execute the code, it returns the following result:

```
Hello! This is Tom's Tutorials Point
Hello! This is Dick's Tutorials Point
Hello! This is Harry's Tutorials Point
*** - EVAL: variable TOM has no value
```