A Simple Implementation of Name Spaces for R

Luke Tierney
Department of Statistics and Actuarial Science
University of Iowa

May 29, 2003

1 Introduction

This document presents the implementation of name space support provided in R 1.7. Some minor changes in the current development verison have also been incorporated. A pdf version[1] of this document is also available.

Name spaces provide a means for packages to control the way global variables in their function definitions are resolved and to control which local definitions are to be available outside the package. Packages with name spaces export variables with certain values. A package with a name space can import variables exported by other packages with name spaces. Functions defined within a name space are defined in an environment consisting of the internal name space frame, which is enclosed in a (set of) imported frames.[2] All name spaces import the `base` name space. The `base` name space is a special name space that is enclosed by the global environment. For example, a function defined in a name space `bar` that imports `foo` will be defined in an environment that looks like this:

```
 ---------------
| bar internals |
 ---------------
|  foo exports  |
 ---------------
| base exports  |
 ---------------
|  .GlobalEnv   |
 ---------------
| package:pkg1  |
 ---------------
      ...
 ---------------
| package:base  |
 ---------------
```

The variables in `base` appear twice: once as a statically determined import (static in the sense that its position in the environment is fixed) and once at the end of the dynamic global environment (where the search index of base varies as packages are attached and detached).[3]

[1] See URL `morenames.pdf`.

[2] The implementation fuses all explicit imports into a single frame. This means that a non-function in a more recent import will mask a function in an earlier import. I think I consider this a feature, not a bug.

[3] In his comments on the first draft of this proposal, John Chambers suggested that it might be cleaner to not include `.GlobalEnv`, so that all globals must be found in explicit imports or in base. I agree with this in principle. Unfortunately the need to

Name spaces are sealed once they are created. Sealing means that imports and exports cannot be changed and that internal variable bindings cannot be changed. Sealing is important if a compiler is to be able to clearly identify what a global variable refers to in order, for example, to handle a reference to certain functions in base in a special way. Sealing also allows a simpler implementation strategy for this name space mechanism.

2 Creating a Package With a Name Space

There are currently two different ways to create a package with a name space. The primary approach is to use a `NAMESPACE` file with directives describing the name space. An alternative is based on including name space directives in the package code. Eventually we will settle on one of these approaches and eliminate the other. But for the moment both are supported.

2.1 Using a NAMESPACE File

A package has a name space if it has a `NAMESPACE` file in its root directory. This file specifies the imports and exports of the name space. These examples should make the syntax used in `NAMESPACE` files clear.

Suppose we want to create a package `foo` with internal definitions for a variable `x` and a function `f`. The code file is

2a ⟨*foo/R/foo.R* 2a⟩≡ 3d▷
```
x <- 1
f <- function(y) c(x,y)
```

If we want to export only `f`, then the `NAMESPACE` file is just

2b ⟨*foo/NAMESPACE* 2b⟩≡ 3e▷
```
export(f)
```

A second package `bar` has a code file that looks like

2c ⟨*bar/R/bar.R* 2c⟩≡
```
c <- function(...) sum(...)
g <- function(y) f(c(y, 7))
h <- function(y) y+9
```

The definition of `c` masks the function in base. `f` is not defined in `bar` but is imported from `foo`. Only `g` and `h` are exported. Thus the `NAMESPACE` file looks like

2d ⟨*bar/NAMESPACE* 2d⟩≡
```
import(foo)
export(g, h)
```

Finally, a third package `baz` has an empty code file `baz.R`

2e ⟨*baz/R/baz.R* 2e⟩≡
```
# empty file
```

The purpose of `baz` is to import some of the exports of `foo` and `bar` and re-export them, using renaming in one case: `bar`'s export `g` is imported under the internal name `hh`, and the internal variable `hh` is exported under the name `gg`.[4]

accommodate `UseMethod` dispatch means, I think, that this is not possible for now. Some additional discussion of the interactions of name spaces with `UseMethod` dispatch is given in Section 4.

[4]Renaming seems like a useful option, but it may turn out to create to many complications and need to be dropped.

2f ⟨*baz/NAMESPACE* 2f⟩≡
```
import(foo)
importFrom(bar, hh = g)
export(f, gg = hh)
```

A user accesses a package with a name space like any other package by calling `library` to load it and attach it to the search path. This recursively loads any packages required to satisfy import specifications, but these implicitly loaded packages will not be attached to the search path. So for the `baz` package,

3a ⟨*R session* 3a⟩≡ 3b▷
```
> library(baz)
> search()
 [1] ".GlobalEnv"      "package:baz"     "package:methods" "package:ctest"
 [5] "package:mva"     "package:modreg"  "package:nls"     "package:ts"
 [9] "Autoloads"       "package:base"
> loadedNamespaces()
[1] "bar"  "base" "baz"  "foo"
```

Loading `baz` with `library` causes it to be loaded and its exports attached. In addition, `foo` and `bar` are loaded but not attached. Only the exports of `baz` are available in the attached frame. Their printed representations show the name spaces in which they were defined.

3b ⟨*R session* 3a⟩+≡ ◁3a 3c▷
```
> ls("package:baz")
[1] "f"  "gg"
> f
function (y)
c(x, y)
<environment: namespace:foo>
> gg
function (y)
f(c(y, 7))
<environment: namespace:bar>
```

Calling `gg` produces a result consistent with the definitions of `c` in the two settings: in `bar` the function `c` is defined to be equivalent to `sum`, but in `foo` the variable `c` refers to the standard function `c` in base.

3c ⟨*R session* 3a⟩+≡ ◁3b 4d▷
```
> gg(6)
[1]  1 13
```

A name space file can also register a method for S3 method dispatch. If `foo` includes the definition

3d ⟨*foo/R/foo.R* 2a⟩+≡ ◁2a
```
print.foo <- function(x, ...) cat("<a foo>\n")
```

and the `NAMESPACE` file includes

3e ⟨*foo/NAMESPACE* 2b⟩+≡ ◁2b 3f▷
```
S3method(print,foo)
```

then the `print.foo` function is registered as the `print` method for class `foo`. It is not necessary to export the method. The need for this is discussed in Section 4.

Finally, a shared library can be registered for loading by adding a directive of the form

3f ⟨*foo/NAMESPACE* 2b⟩+≡ ◁3e
```
useDynLib(foo)
```

to the `NAMESPACE` file. The name space loading mechanism will load this library with `library.dynam` when the name space is loaded. This eliminates the need for most load hook functions.

Loading and attaching are separate processes for packages with name spaces: if package `foo` is loaded to satisfy the import request from `bar` then `foo` is not attached to the global search path. As a result, instead of the single hook function `.First.lib` two hook functions are needed, `.onLoad` and `.onAttach`. Most packages will need at most `.onLoad`. These variables should not be exported.

2.2 Specifying Exports and Imports in Package Code

The second approach to adding a name space to a package `foo` is to add the line

```
Namespace: foo
```

to the `DESCRIPTION` file. The name specified must match the package name.[5] Then calls to the functions `.Import`, `.ImportFrom`, `.Export`, and `.S3method` can be placed directly in the package code.[6] The code files for three packages `foo1`, `bar1`, and `baz1` analogous to the three example packages of the previous section would be

4a ⟨*foo1/R/foo1.R* 4a⟩≡
```
x <- 1
f <- function(y) c(x,y)
print.foo <- function(x, ...) cat("<a foo>\n")
.S3method(print,foo)
.Export(f)
.onLoad <- function(lib, pkg) library.dynam("foo", pkg, lib)
```

4b ⟨*bar1/R/bar1.R* 4b⟩≡
```
.Import(foo1)
c <- function(...) sum(...)
g <- function(y) f(c(y, 7))
h <- function(y) y+9
.Export(g, h)
```

4c ⟨*baz1/R/baz1.R* 4c⟩≡
```
.Import(foo1)
.ImportFrom(bar1, hh = g)
.Export(f, gg = hh)
```

This approach may allow us to use `.Import` in base, thus allowing some functionality to be moved out of base into packages and allowing those packages to use private functions in their definitions.[7]

One issue that has not been resolved is how to track documentation of functions that have been imported and then re-exported.

3 Using Name Spaces With Existing Packages

To use name spaces with an existing package a `NAMESPACE` file has to be added to the package. For a package that does not use other packages the `NAMESPACE` file will only contain export directives. The directive can be constructed using `ls`. For example, for `ctest` a reasonable directive can be built with

[5] `R CMD check` will check for this in R 1.8.

[6] Perhaps a `.UseDynLib` function should also be provided.

[7] For implementation reasons all variabls in base are exported.

4d ⟨*R session* 3a⟩+≡ ◁3c 6▷
```
> as.call(c(as.name("export"), ls("package:ctest")))
export("ansari.test", "ansari.test.default", "ansari.test.formula",
    "bartlett.test", "bartlett.test.default", "bartlett.test.formula",
...
    "wilcox.test.default", "wilcox.test.formula")
```
This leaves out only one of the internal variables: `.First.lib`, which should be replaced by `.onLoad` or a `useDynLib` directive.

To support adding name spaces to existing packages with many public variables it is useful to have an export directive that allows variables to be exported as patterns. The `exportPattern` directive can be used as

```
exportPattern("^trellis\\.")
```

for example. The arguments are patterns that are processed by calling `ls` on the internal name space environment with `all=TRUE`; this means imports are not picked up by the pattern. This is just one of many possible approaches. Using this approach, a package that wants to export all variables except those beginning with a dot could use

5 ⟨*NAMESPACE file to export varialbles not beginning with a period* 5⟩≡
```
exportPattern("^[^\\.]")
```
as its `NAMESPACE` file, together with a `useDynLib` directive or a definition for `.onLoad`.

A package that needs other packages that have name spaces should remove calls to `require` from the sources and replace them with `import` directives in the `NAMESPACE` file.

It may also be necessary to make some changes to the content of a `.First.lib` function, in addition to renaming it as `.onLoad`, when converting to using a name space. Many existing `.First.lib` functions do something like

```
pkgEnv <- pos.to.env(match(paste("package:",pkgname,sep=""), search()))
assign("foo", bar, envir = pkgEnv)
```

to obtain the package environment. This can be replaced by

```
pkgEnv <- topenv()
```

The function `topenv` returns the environment to be used for top level definitions—either the first name space internal environment found searching from the environment where `topenv` is called, or `.GlobalEnv`.

One other issue that may need to be addressed is the registration of methods for `UseMethod` dispatch. Some tools for locating these would be useful.

For now, packages that use name spaces must not be installed with `--save`. Section 5 discusses this issue, which should not be too hard to resolve.

4 UseMethod Dispatching and Method Registration

There is no hope, as far as I can see, of having a notion of private classes with `UseMethod` dispatch—the name-pasting that goes on in dispatch makes this impossible. So class names are globally scoped. The issue is making sure that methods are reasonably scoped, in particular that methods that are defined can be found.

When a generic function uses `UseMethod` to dispatch to an appropriate method the environment searched for methods is the environment in which the generic is called. This means that methods are found if they are defined in the local environment of the call or in the global search path. Without name spaces, essentially all methods are going to be accessible from `.GlobalEnv` (i.e. they are in base, in loaded packages, or in the top level `.GlobalEnv` frame itself).

Suppose a package/name space `p1` defines and exports a print method `print.C` for a class `C`. Suppose a package/name space `p2` imports `p1` and exports a function `f` that returns an object of class `C` and the user executes

```
library(p2)
print(f())
```

The `library` call loads and attaches the package/name space `p2`. It also loads `p1` as a dependency, but `p1` is *not* attached. Hence the `print.C` method of `p1` is not visible at the call site of `print`. The only way around this I can see is an explicit method registration mechanism.

A very simple method registration mechanism has been developed to overcome this problem. After searching the call environment, the top frame of the definition environment for the generic is examined for the existence of a variable containing a methods table. If this exists and is an environment, then it is searched for a method definition. For internal functions the defining environment is taken to be `.BaseNamespaceEnv`. For example,

6 ⟨*R session* 3a⟩+≡ ◁4d

```
> .S3method(print, C, function(x) cat("<C>\n"))
> .S3method(as.character, C, function(x) "<--C-->")
> x<-structure(list(1), class="C")
> x
<C>
> as.character(x)
[1] "<--C-->"
```

This approach associates methods with their generics and insures that methods are visible whenever their generics are. The reason for searching the calling environment first is to minimize changes from the current behavior.

This registration mechanism is only intended to be used within a name space, and packages with name spaces currently do not work with `--save`. Figuring out how to support registration and saving is I think going to be very similar to figuring out how to integrate `methods` and name spaces. Some discussion is given in Section 5.

5 Saving And Loading Name Spaces

The new `saveload` code for R 1.4 includes support for name spaces: If an internal name space environment is to be saved, then instead of saving the entire environment a marker is saved along with a character vector describing the name space. For now that character vector contains just the name and the version, and the version is currently ignored, but it could eventually be expanded. When a work space with such a reference is loaded into an R process, then the character vector is passed to `getNamespace`, which calls `loadNamespace` to load the name space and then return the loaded name space's internal environment.

One thing that has not been addressed yet is how to handle installation with `--save`. I don't think there are conceptual difficulties, it just has to be done. For now, R 1.7 signals an error if `--save` is used and the package has a `NAMESPACE` file.

6 Some Open Issues

6.1 NAMESPACE File or Special Syntax

Java, Tcl, Perl and many other languages use some form of special syntax or declaration mechanism for specifying name space organization within source code. Something along the lines of

```
namespace(foo) {
   ...
}
```

Other languages, such as Ada, Modula 3, mzScheme, and ML separate out the declaration of modules from the implementation source code. Whether they go so far as to require that source and module declarations be in separate files varies, but some do.

Both approaches have merits. Interweaving name space declarations with code makes it easier to see how a particular definition fits into the public interface of the module. Declaring a Java method as `public`, for example, makes clear that it is part of the public interface. On the other hand, separating out the module structure makes it easier to see the public interface at a glance and to think about plugging in different implementations of the same interface.

The main reasons I chose the approach of using a `NAMESPACE` file are:

- It seemed the easiest to implement, since it avoids any parser changes.
- It makes adding name spaces to packages very easy, so it is easy to experiment with using name spaces with existing packages.
- It is declarative and so provides reliable information about the dependencies of a package that can be used by other tools.

Neither of these reasons is compelling for the longer term. We should think through what we want to do here before making a final commitment.

6.2 Name Space Granularity

The current approach requires that there is (at most) one name space per package. It could at times be useful to have a single name space that covers several packages, or a single package that provides several name spaces.

Having a name space that spans several packages could be useful to allow bits of a larger implementation to easily be replaced by new implementations. This can be achieved with the existing mechanism by organizing code into several implementation packages and then defining one interface package that imports the implementations and re-exports them.

One scenario where it might be useful for a single package to provide two name spaces is when the package can be used directly by end users or as a building block for extension by programmers. One set of features could then be exported for the end user and a different set for extension purposes.

A perhaps related issue is whether we should allow nested name spaces or packages in some form.

6.3 Internal Variable Access and Documentation

The internal variables of a name space are meant to be hidden from users of the name space. Thus, from a user's point of view, it makes sense to only document exported variables. But for large packages it might be useful to be able to document functions used internally so someone maintaining the package has some help. One way to deal with this, in principle, is to break large packages into smaller units that have their key functions exported and documented, and for the main package to import these subsidiary packages but only re-exports some of those features. Making sure that re-exported variables also export their documentation is another issue.

Should there be a formal syntax for accessing internal variables? Exported variables can be accessed with `foo::f`. Internal variable can be accessed with something like

```
get("x", env = getNamespace("foo"))
```

or, using the convenience function `getFromNamespace`,

```
getFromNamespace("x", "foo")
```

but should it also be possible to do some sort of special syntax access the internal value of `x` in `foo`? In the absence of renaming, `foo::x` *could* be made to work; whether it *should* is not clear.

6.4 Compilation

Name spaces are useful for compilation for several reasons. Ons is that they allow the compiler to determine accurately which variables refer to variables in `base`. However, this determination will have to be made on the basis of the name space structure in place at compile time. If the name space structure at load time is different, in particular if the name space structure at load time shadows certain `base` variables that were not shadowed at compile time, then assumptions made at compile time may be invalid.

To deal with this we would need to keep enough information about name spaces in a saved work space to be able to determine whether assumptions the compiler makes remain valid. What happens if they do not is another issue—the compiled version could be discarded or an error could be signaled. One option would be for the sealing process to do a semantic analysis that determines exactly which variables the name space uses from each imported name space and to save this information as a signature of the imports. At load time the name space structure could then be checked for consistency with this signature. Version information could also help.

6.5 The `methods` Package

Here is a brief outline of the issues. I have not yet looked at the internals of `methods` to know how complicated this is going to be.

The defining functions in `methods` (`setClass`, `setMethod`, and friends) take a `where` argument. The current default is `.GlobalEnv`. With name spaces the default will need to be the top definition environment for the environment of the call, `topenv(parent.frame())`.

Generic functions are stored in ordinary variables, so in principle nothing additional should be needed. Classes are not directly stored as ordinary variables. They seem to use a name mangling mechanism, with `setClass("track",...)` producing a variable `".__C__track"` containing some form of class object. To support classes in name spaces we would just need a mechanism of exporting and importing that is aware of this name mangling, say directives of the form

```
exportClass(track)
exportClass(trace = track)
importClassFrom(foo, trace)
importClassFrom(foo, mytrack = trace)
```

Within a single run of R little more should be needed. A call to `setMethod` of the form

```
setMethod("plotData", signature(x="track", y="missing"),...)
```

will resolve plotData as a reference to the function by that name visible at the call site by name space rules, and track will refer to the class by that name visible at the call site by name space rules. Where things get tricky is figuring out how to deal with saved work spaces. If user code executes a setMethod and then save.image, then some mechanism is needed to make sure the method is installed in the right place when the saved work space is loaded. The mechanism needed is clear: some form of hook on the load process. A special hook for methods is one approach, but as something similar is needed for registered UseMethod methods, which might prove useful, perhaps a general mechanism for triggering actions on load is worth considering. The tricky issue is finding the right place.

There are two distinct scenarios. One is saved work spaces created by --save installations of packages. Here the original setMethod is known to have occurred in a particular name space context with a known set of classes and generics visible by name space rules. These same classes and generics should be visible at load time, so just saving the raw setMethod call and executing it at load time should be pretty close to sufficient. Things are different if the setMethod is evaluated at top level. Here the visible classes and generics are determined by the dynamic search path consisting of the packages that happen to be loaded. If a top level call

```
  setMethod("gen", signature(x = "foo"), ...)
```

is evaluated and at evaluation time gen is found in package A and foo in package B, then it is not entirely clear what should happen if the work space is saved after this definition and then restored. Should the method be installed against whatever context happens to be found when the restore occurs? Or should an attempt be made to get the ”right” packages loaded?

If the packages have name spaces a reasonable case can be made that the right packages should be loaded if possible. This can be done by saving the method definition in a way that records the source name space of the generic as well as the source name space of the class that were used when the original setMethod call was evaluated. With this information, the appropriate name spaces could be loaded with loadNamespace (i.e. loaded but not attached). In a sense this requires that objects carry what might be called fully qualified class identifiers.

A similar issue arises if class foo is defined in a name space A and an object of class foo is created at top level and saved. Should the class information attached to the object allow the name space A to be loaded when the object is loaded? The new save/load code should allow us to do this if we decide it is the right way to go. Loading A is likely to provide many important methods, such as print methods. But methods for foo defined in a different name space B cannot easily loaded automatically. One approach that would cover this case is to maintain a library-level directory of all methods.

6.6 Sealing

Sealing now locks down more than is strictly necessary. For a compiler to be able to arrange for efficient variable lookup all that is needed is sealing of environments, not bindings. That is, it is sufficient to prevent adding and removing bindings. Preventing the bindings from changing their values is only needed if the compiler wants to make assumptions about those values, such as using the value of pi or inlining a call to log.

We could consider a name space implementation in which bindings are imported and exported rather than values. If the bindings are read-only then you get the current behavior. If they are writable, then assigning to an imported binding (the <<- kind of assignment) should change the value of the corresponding variable in the exporting name space and in all other importing name spaces. Importing bindings requires a more complex implementation, in particular some fairly heavy changes in envir.c, but it is in principle possible.

Whether having writable exports is desirable or not is debatable. Exporting values rather than bindings and sealing all bindings leads to a conceptually simpler model and eliminates the possibility of certain kinds of mischief. If we choose to go that route one option that might minimize changes needed in `envir.c` would be to use special promises.

6.7 Loading Name Spaces

With manual loading via `library` it is convenient to be able to use a short name like `nls` to specify the library. Any version issues can be resolved by specifying the `lib.loc` argument. With name spaces there is a need for implicit loading of imports. Manual intervention is at the very least inconvenient if not impossible. It would therefore be nice to be able to specify an import directive that makes sure the right package and perhaps the right version within a package is loaded.

One option might be to allow packages to register full names that make a conflict less likely. The `NAMESPACE` file for `nls` for example might contain a directive like

```
namespace.fullname(r-project.org/r-core/nls)
```

A package depending on `nls` might then import it as

```
import(nls=r-project.org/r-core/nls)
```

or something along those lines. This would require some form of support in how packages are registered at installation and how package search works.

A hierarchical package structure might be another way to address these issues.

Packages already have a version mechanism; perhaps import directives should allow a minimal version to be specified in some form.

6.8 Organization of Base Name Space

Base is currently one enormous package. It might be worth dividing it into `core`, `models`, etc. A given site could then perhaps define its own base as

10 ⟨*site base name space* 10⟩≡
```
importAndExport(core)
importAndExport(models)
importAndExport(ctest)
```

or something along these lines. If we did this then the default name space included in all name spaces should probably be `core`, not `base`.

One advantage of this approach would be to make `base` easier maintain by allowing some large chunks of code that is repeated in several places to be split out into utility functions that can be re-used internally but need not be exported.

6.9 Other Issues

Several things may still need to be made aware of name spaces. The utilities in the `tools` package seem mostly compatible with name spaes at this point but may need another look once integration of name spaces and S4 dispatch begins.

The `fix` function may be somewhat less useful now since with name spaces you cannot mask functions in packages using name spaces (or base if defined in a name space) with definitions in `.GlobalEnv`. As an interim measure, Brian Ripley has developed a `fixInNamespace` function that may be useful.

One option to consider to avoid having some packages with and some without name spaces is to implicitly define a name space for a package when it is loaded. On the other hand, pure data packages are probably better served by the current approach.

Many more advanced module systems, such as the Scheme48 and MzScheme ones and the ML one allow incomplete or parameterized modules. This sort of thing seems to be used in two different ways. In strongly typed languages like ML the parameters are often types and allow for generic modules. In languages like MzScheme with single inheritance object systems the parameters would often be classes and the parameterized modules would provide a way to specify mixin functionality that can then easily be added to any class, thus providing similar benefits to multiple inheritance. At this point I'm not convinced the benefits for R would outweigh the complexity, but this is the time to think through the issues.

7 Implementation

The implementation consists of R code in `namespace.R` and C code in the base distribution.

11a ⟨*namespace.R* 11a⟩≡
  ⟨*public name space functions* 11b⟩
  ⟨*internal name space functions* 12b⟩

7.1 The Name Space Data Base

Name spaces are registered in a global data base. This insures that the name space name uniquely identifies the name space. The data base is stored internally as an environment. The query function for the data base is `getNamespace`.

11b ⟨*public name space functions* 11b⟩≡ (11a) 11c▷

```
getNamespace <- function(name) {
    ns <- .Internal(getRegisteredNamespace(as.name(name)))
    if (! is.null(ns)) ns
    else loadNamespace(name)
}
```

A listing of the registered name spaces is returned by `loadedNamespaces`.

11c ⟨*public name space functions* 11b⟩+≡ (11a) ◁11b 12d▷

```
loadedNamespaces <- function()
    ls(env = .Internal(getNamespaceRegistry()), all = TRUE)
```

7.2 Name Space Representation

Name spaces are represented as an environment with a specially named variable containing a second environment to store name space data; this provides for mutable state. The sequence of frames of a name space looks like this:

```
 --------------
|   internals   |
 --------------
|    imports    |
 --------------
| base exports  |
 --------------
|  .GlobalEnv   |
 --------------
```

The structure is intended to be private; all access to name space internals is through a set of accessor functions.

The `.__NAMESPACE__.` variable contains an environment to store onformation about the name space. The `spec` field is used by internal C code; if its name is changed here then the C code must be changed as well. The `spec` is a character vector. Its first element is the name space name. The second element is the name space version; the version is ignored for now. Additional elements may be added later. The internal serialization code writes out the entire `spec` vector as an identifier for the name space to load when unserializing. The internal unserialize code passes that character vector to `getNamespace`, which currently ignores the version component.

7.2.1 Creating Name Spaces

Name space objects are created by `makeNamespace`. For bootstrapping reasons the `spec` field needs to be installed using `assign`, not `setNamespaceInfo`.

12a ⟨`makeNamespace` *definition* 12a⟩≡ (20)
```
makeNamespace <- function(name, version = NULL, lib = NULL) {
    impenv <- new.env(parent = .BaseNamespaceEnv, hash = TRUE)
    env <- new.env(parent = impenv, hash = TRUE)
    name <- as.character(as.name(name))
    version <- as.character(version)
    info <- new.env(hash = TRUE, parent = NULL)
    assign(".__NAMESPACE__.", info, env = env)
    assign("spec", c(name=name,version=version), env = info)
    setNamespaceInfo(env, "exports", new.env(hash = TRUE, parent = NULL))
    setNamespaceInfo(env, "imports", list("base"=TRUE))
    setNamespaceInfo(env, "path", file.path(lib, name))
    setNamespaceInfo(env, "dynlibs", NULL)
    setNamespaceInfo(env, "S3methods", NULL)
    .Internal(registerNamespace(name, env))
    env
}
```

Predicates for recognizing name space objects and identifying the base name space are provided by

12b ⟨*internal name space functions* 12b⟩≡ (11a) 12c▷
```
isNamespace <- function(ns) .Internal(isNamespaceEnv(ns))
isBaseNamespace <- function(ns) identical(ns, .BaseNamespaceEnv)
```

The functions `getNamespaceInfo` and `setNamespaceInfo` are used for accessing and assigning values in the auxiliary information environment.

12c ⟨*internal name space functions* 12b⟩+≡ (11a) ◁12b 15b▷
```
getNamespaceInfo <- function(ns, which) {
    ns <- asNamespace(ns, base.OK = FALSE)
    info <- get(".__NAMESPACE__.", env = ns, inherits = FALSE)
    get(which, env = info, inherits = FALSE)
}
setNamespaceInfo <- function(ns, which, val) {
    ns <- asNamespace(ns, base.OK = FALSE)
    info <- get(".__NAMESPACE__.", env = ns, inherits = FALSE)
    assign(which, val, env = info)
}
```

7.2.2 Accessor Functions

The name and version of a name space are returned by

12d ⟨*public name space functions* 11b⟩+≡ (11a) ◁11c 13b▷

```
getNamespaceName <- function(ns) {
    ns <- asNamespace(ns)
    if (isBaseNamespace(ns)) "base"
    else getNamespaceInfo(ns, "spec")["name"]
}
getNamespaceVersion <- function(ns) {
    ns <- asNamespace(ns)
    if (isBaseNamespace(ns))
        c(version = paste(R.version$major, R.version$minor, sep="."))
    else getNamespaceInfo(ns, "spec")["version"]
}
```

7.2.3 Exports

The internal name corresponding to an export name is computed by

13a ⟨`getInternalExportName` *definition* 13a⟩≡ (16a 17b)

```
getInternalExportName <- function(name, ns) {
    exports <- getNamespaceInfo(ns, "exports")
    if (! exists(name, env = exports, inherits = FALSE))
        stop(paste(name, "is not an exported object"))
    get(name, env = exports, inherits = FALSE)
}
```

The currently registered exports are returned by `getNamespaceExports`.

13b ⟨*public name space functions* 11b⟩+≡ (11a) ◁12d 13d▷

```
getNamespaceExports <- function(ns) {
    ns <- asNamespace(ns)
    if (isBaseNamespace(ns)) ls(NULL, all = TRUE)
    else ls(getNamespaceInfo(ns, "exports"), all = TRUE)
}
```

`addExports` registers new export specifications.

13c ⟨`addExports` *definition* 13c⟩≡ (17a)

```
addExports <- function(ns, new) {
    exports <- getNamespaceInfo(ns, "exports")
    expnames <- names(new)
    intnames <- new
    for (i in seq(along = new)) {
        if (exists(expnames[i], env = exports, inherits = FALSE))
            warning("replacing previous export:", expnames[i])
        assign(expnames[i], intnames[i], env = exports)
    }
}
```

7.2.4 Imports

The current imports are returned by

13d ⟨*public name space functions* 11b⟩+≡ (11a) ◁13b 14b▷

```
getNamespaceImports <- function(ns) {
    ns <- asNamespace(ns)
    if (isBaseNamespace(ns)) NULL
    else getNamespaceInfo(ns, "imports")
}
```

`addImports` registers new import specifications.

14a ⟨`addImports` *definition* 14a⟩≡ (15d)

```
addImports <- function(ns, from, what) {
    imp <- structure(list(what), names = getNamespaceName(from))
    imports <- getNamespaceImports(ns)
    setNamespaceInfo(ns, "imports", c(imports, imp))
}
```

The function `getNamespaceUsers` provides a possible fairly inefficient function for determining all loaded name spaces that import a given name space.

14b ⟨*public name space functions* 11b⟩+≡ (11a) ◁13d 17b▷

```
getNamespaceUsers <- function(ns) {
    nsname <- getNamespaceName(asNamespace(ns))
    users <- character(0)
    for (n in loadedNamespaces()) {
        inames <- names(getNamespaceImports(n))
        if (match(nsname, inames, 0))
            users <- c(n, users)
    }
    users
}
```

7.2.5 Sealing Name Spaces

The `sealNamespace` function seals a name space by locking its internal enveronment and its imports frame.

14c ⟨`sealNamespace` *definition* 14c⟩≡ (20)

```
sealNamespace <- function(ns) {
    ⟨namespaceIsSealed definition 14d⟩
    ns <- asNamespace(ns, base.OK = FALSE)
    if (namespaceIsSealed(ns)) stop("already sealed")
    lockEnvironment(ns, TRUE)
    lockEnvironment(parent.env(ns), TRUE)
}
```

The predicate `namespaceIsSealed` just checks whether the internal environment is locked.

14d ⟨`namespaceIsSealed` *definition* 14d⟩≡ (14c 15d 17a)

```
namespaceIsSealed <- function(ns)
   environmentIsLocked(ns)
```

7.2.6 Registering S3 Methods and Dynamic Librarys

14e ⟨`addNamespaceS3method` *definition* 14e⟩≡ (23b)

```
addNamespaceS3method <- function(ns, generic, class, method) {
    regs <- getNamespaceInfo(ns, "S3methods")
    regs <- c(regs, list(list(generic, class, method)))
    setNamespaceInfo(ns, "S3methods", regs)
}
```

15a ⟨addNamespaceDynLibs *definition* 15a⟩≡ (20)
```
addNamespaceDynLibs <- function(ns, newlibs) {
    dynlibs <- getNamespaceInfo(ns, "dynlibs")
    setNamespaceInfo(ns, "dynlibs", c(dynlibs, newlibs))
}
```

7.2.7 Utilities

The function `asNamespace` is used to allow most higher level functions to be called with either a name space object or a character string naming a registered name space.

15b ⟨*internal name space functions* 12b⟩+≡ (11a) ◁12c 15c▷
```
asNamespace <- function(ns, base.OK = TRUE) {
    if (is.character(ns) || is.name(ns))
        ns <- getNamespace(ns)
    if (! isNamespace(ns))
        stop("not a name space")
    else if (! base.OK && isBaseNamespace(ns))
        stop("operation not allowed on base name space")
    else ns
}
```

7.3 Importing Into Name Spaces

The `namespaceImport` function accepts any number of name spaces as arguments but defers the actual work to `namespaceImportFrom`.

15c ⟨*internal name space functions* 12b⟩+≡ (11a) ◁15b 15d▷
```
namespaceImport <- function(self, ...) {
    for (ns in list(...))
        namespaceImportFrom(self, asNamespace(ns))
}
```

The `namespaceImportFrom` function imports the values of the specified variables into the import frame and it records the import request. The record maintained in the name space could be used to restore a saved name space. This definition allows importing into base and into non-namespace environments.

15d ⟨*internal name space functions* 12b⟩+≡ (11a) ◁15c 16a▷
```
namespaceImportFrom <- function(self, ns, vars) {
    ⟨addImports definition 14a⟩
    ⟨namespaceIsSealed definition 14d⟩
    ⟨makeImportExportNames definition 16b⟩
    if (is.character(self))
        self <- getNamespace(self)
    ns <- asNamespace(ns)
    if (missing(vars)) impvars <- getNamespaceExports(ns)
    else impvars <- vars
    impvars <- makeImportExportNames(impvars)
    impnames <- names(impvars)
    if (any(duplicated(impnames))) {
        stop("duplicate import names ",
             paste(impnames[duplicated(impnames)], collapse=", "))
    }
    if (isNamespace(self) && isBaseNamespace(self)) {
        impenv <- self
```

```
            msg <- "replacing local value with import:"
            register <- FALSE
        }
        else if (isNamespace(self)) {
            if (namespaceIsSealed(self))
                stop("cannot import into a sealed namespace")
            impenv <- parent.env(self)
            msg <- "replacing previous import:"
            register <- TRUE
        }
        else if (is.environment(self)) {
            impenv <- self
            msg <- "replacing local value with import:"
            register <- FALSE
        }
        else stop("invalid import target")
        for (n in impnames)
            if (exists(n, env = impenv, inherits = FALSE))
                warning(paste(msg, n))
        importIntoEnv(impenv, impnames, ns, impvars)
        if (register) {
            if (missing(vars)) addImports(self, ns, TRUE)
            else addImports(self, ns, impvars)
        }
    }
```

The function `importIntoEnv` is responsible for transferring bindings from one environment to another. The internal version, insures that promises are not forced and that active bindings are transferred properly.

16a ⟨*internal name space functions* 12b⟩+≡ (11a) ◁15d 17a▷

```
    importIntoEnv <- function(impenv, impnames, expenv, expnames) {
        ⟨getInternalExportName definition 13a⟩
        expnames <- unlist(lapply(expnames, getInternalExportName, expenv))
        .Internal(importIntoEnv(impenv, impnames, expenv, expnames))
    }
```

The variables to be imported are specified as a character vector. If entries in the vector are named then the values are imported under the specified name. Thus

```
    namespaceImportFrom(bar, foo, y="x")
```

means the value of the variable `x` exported by `foo` will be imported into `bar` under the name `y`. The function `makeImportExportNames` takes a possibly named character vector and adds names for all elements. This function is also used to allow renaming of exports.

16b ⟨`makeImportExportNames` *definition* 16b⟩≡ (15d 17a)

```
    makeImportExportNames <- function(spec) {
        old <- as.character(spec)
        new <- names(spec)
        if (is.null(new)) new <- old
        else new[new==""] <- old[new==""]
        names(old) <- new
        old
    }
```

7.4 Exporting From Name Spaces

The namespaceExport function accepts a character vector of names to export. If the elements of the argument are named, then the names are used as the export names.

17a ⟨*internal name space functions* 12b⟩+≡ (11a) ◁16a 18d▷

```
namespaceExport <- function(ns, vars) {
    ⟨namespaceIsSealed definition 14d⟩
    if (namespaceIsSealed(ns))
        stop("cannot add to exports of a sealed namespace")
    ns <- asNamespace(ns, base.OK = FALSE)
    if (length(vars) > 0) {
        ⟨addExports definition 13c⟩
        ⟨makeImportExportNames definition 16b⟩
        new <- makeImportExportNames(vars)
        if (any(duplicated(new)))
            stop("duplicate export names ",
             paste(new[duplicated(new)], collapse=", "))
        undef <- new[! sapply(new, exists, env = ns)]
        if (length(undef) != 0) {
           undef <- do.call("paste", as.list(c(undef, sep=", ")))
            stop(paste("undefined exports:", undef))
        }
        addExports(ns, new)
    }
}
```

7.5 Evaluation and Environments

The values of exported variables can be obtained with the getExportedValue. The name provided to getExportedValue is first translated to its internal name, and then the value of the internal name is looked up in the internal frame and returned.

17b ⟨*public name space functions* 11b⟩+≡ (11a) ◁14b 17c▷

```
getExportedValue <- function(ns, name) {
    ⟨getInternalExportName definition 13a⟩
    ns <- asNamespace(ns)
    if (isBaseNamespace(ns)) get(name, env = ns)
    else get(getInternalExportName(name, ns), env = ns)
}
```

The :: operator provides a shorthand for getExportedValue:

17c ⟨*public name space functions* 11b⟩+≡ (11a) ◁17b 18a▷

```
"::" <- function(pkg,name){
    pkg <- as.character(substitute(pkg))
    name <- as.character(substitute(name))
    getExportedValue(pkg, name)
}
```

7.6 Attaching Name Spaces

The function attachNamespace attaches a NULL list and then transfers the values of the exported variables of a name space into the resulting environment. The attached frame is locked. The .onAttach hook function, if present in the internal environment, is run after the variables have been installed. It is not likely that this mechanism will get much use; the `onLoad` hook in `loadNamespace` is much more useful.

18a ⟨*public name space functions* 11b⟩+≡ (11a) ◁17c 20▷
```
  attachNamespace <- function(ns, pos = 2) {
      ⟨runHook definition 18b⟩
      ns <- asNamespace(ns, base.OK = FALSE)
      nsname <- getNamespaceName(ns)
      nspath <- getNamespaceInfo(ns, "path")
      attname <- paste("package", nsname, sep=":")
      if (attname %in% search())
          stop("name space is already attached")
      env <- attach(NULL, pos = pos, name = attname)
      on.exit(detach(pos = pos))
      attr(env, "path") <- nspath
      exports <- getNamespaceExports(ns)
      importIntoEnv(env, exports, ns, exports)
      runHook(".onAttach", ns, dirname(nspath), nsname)
      lockEnvironment(env, TRUE)
      on.exit()
      invisible(env)
  }
```

18b ⟨`runHook` *definition* 18b⟩≡ (18a 20 23a)
```
  runHook <- function(hookname, env, ...) {
      if (exists(hookname, envir = env, inherits = FALSE)) {
          fun <- get(hookname, envir = env, inherits = FALSE)
          if (! is.null(try({ fun(...); NULL})))
              stop(paste(hookname, "failed"))
      }
  }
```

7.7 Parsing the NAMESPACE File

The convention for locating the `NAMESPACE` file of a package is encoded in the function `namespace-FilePath`. Changing this convention means just changing this function.

18c ⟨`namespaceFilePath` *definition* 18c⟩≡ (18)
```
  namespaceFilePath <- function(package, package.lib)
      file.path(package.lib, package, "NAMESPACE")
```

The test of whether a package has a name space is handled by `packageHasNamespace`

18d ⟨*internal name space functions* 12b⟩+≡ (11a) ◁17a 18e▷
```
  packageHasNamespace <- function(package, package.lib) {
      ⟨namespaceFilePath definition 18c⟩
      file.exists(namespaceFilePath(package, package.lib)) ||
      ! is.na(read.dcf(file.path(package.lib, package, "DESCRIPTION"),
                                 fields="Namespace"))
  }
```

The function `parseNamespaceFile` is responsible for reading in a `NAMESPACE` file using `parse` and collecting the directives into a structure.

18e ⟨*internal name space functions* 12b⟩+≡ (11a) ◁18d 23b▷

```
parseNamespaceFile <- function(package, package.lib, mustExist = TRUE) {
    ⟨namespaceFilePath definition 18c⟩
    ⟨sQuote definition 19⟩
    nsFile <- namespaceFilePath(package, package.lib)
    if (file.exists(nsFile))
        directives <- parse(nsFile)
    else if (mustExist)
        stop(paste("package", sQuote(package), "has no NAMESPACE file"))
    else directives <- NULL
    exports <- character(0)
    exportPatterns <- character(0)
    imports <- list()
    dynlibs <- character(0)
    S3methods <- list()
    for (e in directives)
        switch(as.character(e[[1]]),
               export = {
                   exp <- e[-1]
                   exp <- structure(as.character(exp), names=names(exp))
                   exports <- c(exports, exp)
               },
               exportPattern = {
                   pat <- as.character(e[-1])
                   exportPatterns <- c(pat, exportPatterns)
               },
               import = imports <- c(imports,as.list(as.character(e[-1]))),
               importFrom = {
                   imp <- e[-1]
                   ivars <- imp[-1]
                   inames <- names(ivars)
                   imp <- list(as.character(imp[1]),
                               structure(as.character(ivars), names=inames))
                   imports <- c(imports, list(imp))
               },
               useDynLib = {
                   dyl <- e[-1]
                   dynlibs <- c(dynlibs, as.character(dyl))
               },
               S3method = {
                   spec <- e[-1]
                   if (length(spec) != 2 && length(spec) != 3)
                       stop(paste("bad S3method directive:", deparse(e)))
                   S3methods <- c(S3methods, list(as.character(e[-1])))
               },
               stop(paste("unknown namespace directive:", deparse(e))))
    list(imports=imports, exports=exports, exportPatterns = exportPatterns,
         dynlibs=dynlibs, S3methods = S3methods)
}
```

19 ⟨sQuote *definition* 19⟩≡ (18e 20)

```
sQuote <- function(s) paste("’", s, "’", sep = "")
```

7.8 Loading Name Spaces

The code for loading a name space is quite similar to, and is mostly borrowed from, the corresponding code in `library`. An `on.exit` action is installed to unregister the name space if the load fails; no attempt is made to roll back any successfully loaded imports in this case. A crude dynamic variable is used to check for circular dependencies. Currently `cacheMetaData` is called if it looks like methods have been defined, but this is not likely to work yet and a warning is issued.

20 ⟨*public name space functions* 11b⟩+≡ (11a) ◁18a 22c▷

```
loadNamespace <- function (package, lib.loc = NULL,
                           keep.source = getOption("keep.source.pkgs")) {
    # eventually allow version as second component; ignore for now.
    package <- as.character(package)[[1]]

    # check for cycles
    ⟨dynGet definition 22a⟩
    loading <- dynGet("__NameSpacesLoading__", NULL)
    if (match(package, loading, 0))
        stop("cyclic name space dependencies are not supported")
    "__NameSpacesLoading__" <- c(package, loading)

    ns <- .Internal(getRegisteredNamespace(as.name(package)))
    if (! is.null(ns))
        ns
    else {
        ⟨runHook definition 18b⟩
        ⟨sQuote definition 19⟩
        ⟨makeNamespace definition 12a⟩
        ⟨sealNamespace definition 14c⟩
        ⟨addNamespaceDynLibs definition 15a⟩
        # **** FIXME: test for methods
        hadMethods <- "package:methods" %in% search()

        # find package and check it has a name space
        pkgpath <- .find.package(package, lib.loc, quiet = TRUE)
        if (length(pkgpath) == 0)
            stop(paste("There is no package called", sQuote(package)))
        package.lib <- dirname(pkgpath)
        if (! packageHasNamespace(package, package.lib))
            stop(paste("package", sQuote(package),
                       "does not have a name space"))

        # create namespace; arrange to unregister on error
        nsInfo <- parseNamespaceFile(package, package.lib, mustExist = FALSE)
        version = read.dcf(file.path(package.lib, package, "DESCRIPTION"),
                           fields="Version")
        ns <- makeNamespace(package, version = version, lib = package.lib)
        on.exit(.Internal(unregisterNamespace(package)))

        # process imports
        for (i in nsInfo$imports) {
            if (is.character(i))
                namespaceImport(ns, loadNamespace(i, c(lib.loc, .libPaths()),
                                                  keep.source))
            else
                namespaceImportFrom(ns,
                                    loadNamespace(i[[1]],
                                                  c(lib.loc, .libPaths()),
                                                  keep.source), i[[2]])
        }

        # load the code
        env <- asNamespace(ns)
        codeFile <- file.path(package.lib, package, "R", package)
        if (file.exists(codeFile))
            sys.source(codeFile, env, keep.source = keep.source)
        else warning(paste("Package ", sQuote(package), "contains no R code"))

        # save the package name in the environment
        assign(".packageName", package, envir = env)

        # register any S3 methods
        for (spec in nsInfo$S3methods) {
            generic <- spec[1]
            class <- spec[2]
            if (length(spec) == 3) mname <- spec[3]
            else mname <- paste(generic, class, sep=".")
            registerS3method(spec[1], spec[2], mname, env = env)
        }

        # load any dynamic libraries
        for (lib in nsInfo$dynlibs)
            library.dynam(lib, package, package.lib)
        addNamespaceDynLibs(env, nsInfo$dynlibs)

        # run the load hook
        runHook(".onLoad", env, package.lib, package)

        # process exports, seal, and clear on.exit action
        exports <- nsInfo$exports
        for (p in nsInfo$exportPatterns)
            exports <- c(ls(env, pat = p, all = TRUE), exports)
        namespaceExport(ns, exports)
        sealNamespace(ns)

        # **** FIXME: process methods but warn of possible problems
        if (! exists(".noGenerics", envir = env, inherits = FALSE) &&
            length(objects(env, pattern="^\\.__M", all=TRUE)) != 0 &&
            hadMethods &&
            ! identical(package, "package:methods")) {
            warning("method code may not work in a name space")
            cacheMetaData(env, TRUE)
        }
        on.exit()

        ns
    }
```

```
}
```

22a ⟨`dynGet` *definition* 22a⟩≡ (20)

```
dynGet <- function(name, notFound = stop(paste(name, "not found"))) {
    n <- sys.nframe()
    while (n > 1) {
        n <- n - 1
        env <- sys.frame(n)
        if (exists(name, env = env, inherits = FALSE))
            return(get(name, env = env, inherits = FALSE))
    }
    notFound
}
```

7.9 Modified Library Function

The modified `library` library function just adds a small bit of code to check whether the package to be loaded is a name space package. If it is, it is loaded using `loadNamespace`; otherwise the code falls through the standard `library` code.

22b ⟨*modification to* `library` *function in base package* 22b⟩≡

```
# if the name space mechanism is available and the package
# has a name space, then the name space loading mechanism
# takes over.
if (exists("packageHasNamespace") &&
    packageHasNamespace(package, which.lib.loc)) {
    tt <- try({
        ns <- loadNamespace(package, c(which.lib.loc, lib.loc))
        env <- attachNamespace(ns)
    })
    if (inherits(tt, "try-error"))
        if (logical.return)
            return(FALSE)
        else stop("package/namespace load failed")
    else {
        on.exit(do.call("detach", list(name = pkgname)))
        nogenerics <- checkNoGenerics(env)
        if(warn.conflicts &&
           !exists(".conflicts.OK", envir = env, inherits = FALSE))
           checkConflicts(package, pkgname, pkgpath, nogenerics)
        on.exit()
        if (logical.return)
            return(TRUE)
        else
            return(invisible(.packages()))
    }
}
```

7.10 Finding the Top Level Environment

The function `topenv` locates the nearest “top level” environment to its argument.

22c ⟨*public name space functions* 11b⟩+≡ (11a) ◁20 23a▷

```
topenv <- function(envir = parent.frame()) {
    while (! is.null(envir)) {
        if (! is.null(attr(envir, "name")) ||
            identical(envir, .GlobalEnv) ||
            .Internal(isNamespaceEnv(envir)))
            return(envir)
        else envir <- parent.env(envir)
    }
    return(.GlobalEnv)
}
```

7.11 Unloading Name Spaces

During debugging it may be useful to be able to unload name spaces. This requires that the name space not be used for imports by any other loaded name space.

23a ⟨*public name space functions* 11b⟩+≡ (11a) ◁22c 24▷

```
unloadNamespace <- function(ns) {
    ⟨runHook definition 18b⟩
    ns <- asNamespace(ns, base.OK = FALSE)
    nsname <- getNamespaceName(ns)
    pos <- match(paste("package", nsname, sep=":"), search())
    if (! is.na(pos)) detach(pos = pos)
    users <- getNamespaceUsers(ns)
    if (length(users) != 0)
        stop(paste("name space still used by:", paste(users, collapse = ", ")))
    nspath <- getNamespaceInfo(ns, "path")
    try(runHook(".onUnload", ns, nspath))
    .Internal(unregisterNamespace(nsname))
}
```

7.12 Registering S3 Methods

The S3 methods table is stored in the `.__S3MethodsTable__.` variable. (This is just a crude hack for now—something a bit more sophisticated would be useful.) The function `registerS3method` registers a method. This definition uses a promise if the method is specified by name in order to work well with data base storage.

23b ⟨*internal name space functions* 12b⟩+≡ (11a) ◁18e

```
registerS3method <- function(genname, class, method, envir = parent.frame()) {
    ⟨addNamespaceS3method definition 14e⟩
    genfun <- get(genname, envir = envir)
    if (typeof(genfun) == "closure")
        defenv <- environment(genfun)
    else defenv <- .BaseNamespaceEnv
    if (! exists(".__S3MethodsTable__.", envir = defenv, inherits = FALSE))
        assign(".__S3MethodsTable__.", new.env(hash = TRUE, parent = NULL),
               envir = defenv)
    table <- get(".__S3MethodsTable__.", envir = defenv, inherits = FALSE)
    if (is.character(method)) {
        wrap <- function(method, home) {
            method <- method  # force evaluation
            home <- home      # force evaluation
            delay(get(method, env = home), env = environment())
        }
        assign(paste(genname, class, sep = "."), wrap(method, envir),
               envir = table)
    }
    else if (is.function(method))
        assign(paste(genname, class, sep = "."), method, envir = table)
    else stop("bad method")
    if (isNamespace(envir) && ! identical(envir, .BaseNamespaceEnv))
        addNamespaceS3method(envir, genname, class, method)
}
```

7.13 Functions for Importing and Exporting from Code Files

These functions support the alternate interface to name space creation described in Section 2.2.

24 ⟨*public name space functions* 11b⟩+≡ (11a) ◁23a

```
.Import <- function(...) {
    envir <- parent.frame()
    names <- as.character(substitute(list(...)))[-1]
    for (n in names)
        namespaceImportFrom(envir, n)
}
.ImportFrom <- function(name, ...) {
    envir <- parent.frame()
    name <-  as.character(substitute(name))
    names <- as.character(substitute(list(...)))[-1]
    namespaceImportFrom(envir, name, names)
}
.Export <- function(...) {
    ns <- topenv(parent.frame())
    if (identical(ns, .BaseNamespaceEnv))
        warning("all objects in base name space are currently exported.")
    else if (! isNamespace(ns))
        stop("can only export from a name space")
    else {
        names <- as.character(substitute(list(...)))[-1]
        namespaceExport(ns, names)
    }
}
.S3method <- function(generic, class, method) {
    generic <- as.character(substitute(generic))
    class <- as.character(substitute(class))
    if (missing(method)) method <- paste(generic, class, sep=".")
    registerS3method(generic, class, method, envir = parent.frame())
    invisible(NULL)
}
```

Indices

Chunks

Identifiers