Clojure Namespaces and Vars

This guide covers:

- An overview of Clojure namespaces and vars
- How to define namespaces
- How to use functions in other namespaces
- `require`, `refer` and `use`
- Common compilation errors and typical problems that cause them
- Namespaces and their relation to code compilation in Clojure

What Version of Clojure Does This Guide Cover?

This guide covers Clojure 1.5.

Overview

Clojure functions are organized into *namespaces*. Clojure namespaces are very similar to Java packages and Python modules. Namespaces are basically maps (dictionaries) that map names to *vars*. In many cases, those vars store functions in them.

Defining a Namespace

Namespaces are usually defined using the `clojure.core/ns` macro. In its basic form, it takes a name as a symbol:

```
(ns superlib.core)
```

Namespaces can have multiple segments, separated by a dot:

```
(ns megacorp.service.core)
```

It is **highly recommended** to avoid using single segment namespaces (e.g. `superlib`) to avoid inconvenient conflicts other developers will have to work around. If a library or application belongs to an organization or a group of projects, the `[organization].[library|app].[group-of-functions]` pattern is recommended. For example:

```
(ns clojurewerkz.welle.kv)

(ns megacorp.search.indexer.core)
```

In addition, the `ns` macro takes a number of optional forms:

- `(:require ...)`
- `(:import ...)`
- `(:use ...)`
- `(:refer-clojure ...)`
- `(:gen-class ...)`

These are just slightly more concise variants of `clojure.core/import`, `clojure.core/require`, et cetera.

The :require Helper Form

The `:require` helper form is for setting up access to other Clojure namespaces from your code. For example:

```
(ns megacorp.profitd.scheduling
  (:require clojure.set))

;; Now it is possible to do:
;; (clojure.set/difference #{1 2 3} #{3 4 5})
```

This will make sure the `clojure.set` namespace is loaded, compiled, and available as `clojure.set` (using its fully qualified name). It is possible (and common) to make a namespace available under an alias:

```
(ns megacorp.profitd.scheduling
  (:require [clojure.set :as cs]))

;; Now it is possible to do:
;; (cs/difference #{1 2 3} #{3 4 5})
```

One more example with two required namespaces:

```
(ns megacorp.profitd.scheduling
  (:require [clojure.set  :as cs]
            [clojure.walk :as walk]))
```

The :refer Option

To make functions in `clojure.set` available in the defined namespace via short names (i.e., their unqualified names, without the `clojure.set` or other prefix), you can tell Clojure compiler to *refer* to certain functions:

```
(ns megacorp.profitd.scheduling
  (:require [clojure.set :refer [difference intersection]]))

;; Now it is possible to do:
;; (difference #{1 2 3} #{3 4 5})
```

The `:refer` feature of the `:require` form is new in Clojure 1.4.

It is possible to refer to all functions in a namespace (usually not necessary):

```
(ns megacorp.profitd.scheduling
  (:require [clojure.set :refer :all]))

;; Now it is possible to do:
;; (difference #{1 2 3} #{3 4 5})
```

The :import Helper Form

The `:import` helper form is for setting up access to Java classes from your Clojure code. For example:

```
(ns megacorp.profitd.scheduling
  (:import java.util.concurrent.Executors))
```

This will make sure the `java.util.concurrent.Executors` class is imported and can be used by its short name, `Executors`. It is possible to import multiple classes:

```
(ns megacorp.profitd.scheduling
  (:import java.util.concurrent.Executors
           java.util.concurrent.TimeUnit
           java.util.Date))
```

If multiple imported classes are in the same namespace (like in the example above), it is possible to avoid some duplication by using an *import list*. The first element of an import list is the package and other elements are class names in that package:

```
(ns megacorp.profitd.scheduling
  (:import [java.util.concurrent Executors TimeUnit]
           java.util.Date))
```

Even though *import list* is called a list, it can be any Clojure collection (typically vectors are used).

The Current Namespace

Under the hood, Clojure keeps **current namespace** a special var, *ns* (https://clojuredocs.org/clojure.core/*ns*). When vars are defined using the def (https://clojuredocs.org/clojure.core/def) special form, they are added to the current namespace.

The :refer-clojure Helper Form

Functions like `clojure.core/get` and macros like `clojure.core/defn` can be used without namespace qualification because they reside in the `clojure.core` namespace and Clojure compiler automatically *refers* all vars in it. Therefore, if your namespace defines a function with the same name (e.g. `find`), you will get a warning from the compiler, like this:

```
WARNING: find already refers to: #'clojure.core/find in namespace: megacorp.profi
```

This means that in the `megacorp.profitd.scheduling` namespace, `find` already refers to a value which happens to be `clojure.core/find`, but it is being replaced by a different value. Remember, Clojure is a very dynamic language and namespaces are basically maps, as far as the implementation goes. Most of the time, however, replacing vars like this is not intentional and Clojure compiler emits a warning.

To solve this problem, you can either rename your function, or else exclude certain `clojure.core` functions from being referred using the `(:refer-clojure ...)` form within the `ns`:

```
(ns megacorp.profitd.scheduling
  (:refer-clojure :exclude [find]))

(defn find
  "Finds a needle in the haystack."
  [^String haystack]
  (comment ...))
```

In this case, to use `clojure.core/find`, you will have to use its fully qualified name: `clojure.core/find`:

```
(ns megacorp.profitd.scheduling
  (:refer-clojure :exclude [find]))

(defn find
  "Finds a needle in the haystack."
  [^String haystack]
  (clojure.core/find haystack :needle))
```

The :use Helper Form

In Clojure versions before 1.4, there was no `:refer` support for the `(:require ...)` form. Instead, a separate form was used: `(:use ...)`:

```
(ns megacorp.profitd.scheduling-test
  (:use clojure.test))
```

In the example above, **all** functions in `clojure.test` are made available in the current namespace. This practice (known as "naked use") works for `clojure.test` in test namespaces, but in general not a good idea. `(:use ...)` supports limiting functions that will be referred:

```
(ns megacorp.profitd.scheduling-test
  (:use clojure.test :only [deftest testing is]))
```

which is a pre-1.4 alternative of

```
(ns megacorp.profitd.scheduling-test
  (:require clojure.test :refer [deftest testing is]))
```

It is highly recommended to use `(:require ...)` (optionally with `... :refer [...]`) on Clojure 1.4 and later releases. `(:use ...)` is a thing of the past and now that `(:require ...)` with `:refer` is capable of doing the same thing when you need it, it is a good idea to let `(:use ...)` go.

The :gen-class Helper Form

TBD: How to Contribute (https://github.com/clojure-doc/clojure-doc.github.io#how-to-contribute)

Documentation and Metadata

Namespaces can have documentation strings. You can add one with the optional `ns` macro parameter:

```
(ns superlib.core
  "Core functionality of Superlib.

   Other parts of Superlib depend on functions and macros in this namespace."
  (:require [clojure.set :refer [union difference]]))
```

or metadata:

```
(ns ^{:doc "Core functionality of Superlib.
            Other parts of Superlib depend on functions and macros in this namesp
      :author "Joe Smith"}
   superlib.core
  (:require [clojure.set :refer [union difference]]))
```

Metadata can contain any additional keys such as `:author` which may be of use to various tools (such as Codox (https://clojars.org/codox), Cadastre (https://clojars.org/cadastre), or lein-clojuredocs (https://clojars.org/lein-clojuredocs)).

How to Use Functions From Other Namespaces in the REPL

The `ns` macro is how you usually require functions from other namespaces. However, it is not very convenient in the REPL. For that case, the `clojure.core/require` function can be used directly:

```
;; Will be available as clojure.set, e.g. clojure.set/difference.
(require 'clojure.set)

;; Will be available as io, e.g. io/resource.
(require '[clojure.java.io :as io])
```

It takes a quoted *libspec (/articles/language/glossary/#libspec)*. The libspec is either a namespace name or a collection (typically a vector) of `[name :as alias]` or `[name :refer [fns]]`:

```
(require '[clojure.set :refer [difference]])

(difference #{1 2 3} #{3 4 5 6})  ; ⇒ #{1 2}
```

The `:as` and `:refer` options can be used together:

```
(require '[clojure.set :as cs :refer [difference]])

(difference #{1 2 3} #{3 4 5 6})  ; ⇒ #{1 2}
(cs/union #{1 2 3} #{3 4 5 6})    ; ⇒ #{1 2 3 4 5 6}
```

`clojure.core/use` does the same thing as `clojure.core/require` but with the `:refer` option (as discussed above). It is not generally recommended to use `use` with Clojure versions starting with 1.4. Use `clojure.core/require` with `:refer` instead.

Namespaces and Class Generation

TBD: How to Contribute (https://github.com/clojure-doc/clojure-doc.github.io#how-to-contribute)

Namespaces and Code Compilation in Clojure

Clojure is a compiled language: code is compiled when it is loaded (usually with `clojure.core/require`).

A namespace can contain vars or be used purely to extend protocols, add multimethod implementations, or conditionally load other libraries (e.g. the most suitable JSON parser or key/value store implementation). In all cases, to trigger compilation, you need to require the namespace.

Private Vars

Vars (and, in turn, functions defined with `defn`) can be private. There are two equivalent ways to specify that a function is private: either via metadata or by using the `defn-` macro:

```
(ns megacorp.superlib)

;;
;; Implementation
;;

(def ^{:private true}
  source-name "supersource")

(defn- data-stream
  [source]
  (comment ...))
```

Constant Vars

Vars can be constant. This is done by setting the `:const` metadata key to `true`. This will cause Clojure compiler to compile it as a constant:

```
(ns megacorp.epicgame)

;;
;; Implementation
;;

(def ^{:const true}
  default-score 100)
```

How to Look up and Invoke a Function by Name

It is possible to look up a function in particular namespace by-name with `clojure.core/ns-resolve`. This takes quoted names of the namespace and function. The returned value can be used just like any other function, for example, passed as an argument to a higher order function:

```
(ns-resolve 'clojure.set 'difference)  ; ⇒ #'clojure.set/difference

(let [f (ns-resolve 'clojure.set 'difference)]
   (f #{1 2 3} #{3 4 5 6}))  ; ⇒ #{1 2}
```

Compiler Exceptions

This section describes some common compilation errors.

ClassNotFoundException

This exception means that JVM could not load a class. It is either misspelled or not on the classpath (/articles/language/glossary/#classpath). Potentially your project has unsatisfied dependency (some dependencies may be optional).

Example:

```
user=> (import java.uyil.concurrent.TimeUnit)
ClassNotFoundException java.uyil.concurrent.TimeUnit  java.net.URLClassLoader$1.r
```

In the example above, `java.uyil.concurrent.TimeUnit` should have been `java.util.concurrent.TimeUnit`.

CompilerException java.lang.RuntimeException: No such var

This means that somewhere in the code a non-existent var is used. It may be a typo, an incorrect macro-generated var name or a similar issue. Example:

```
user=> (clojure.java.io/resouce "thought_leaders_quotes.csv")
CompilerException java.lang.RuntimeException: No such var: clojure.java.io/resouc
```

In the example above, `clojure.java.io/resouce` should have been `clojure.java.io/resource`. `NO_SOURCE_PATH` means that compilation was triggered from the REPL and not a Clojure source file.

Temporarily Overriding Vars in Namespaces

TBD: How to Contribute (https://github.com/clojure-doc/clojure-doc.github.io#how-to-contribute)

Getting Information About and Programmatically Manipulating Namespaces

TBD: How to Contribute (https://github.com/clojure-doc/clojure-doc.github.io#how-to-contribute)

Wrapping Up

Namespaces are basically maps (dictionaries) that map names to vars. In many cases, those vars store functions in them.

This implementation lets Clojure have many of its highly dynamic features at a very reasonable runtime overhead cost. For example, vars in namespaces can be temporarily altered for unit testing purposes.

Contributors

Michael Klishin michael@defprotocol.org (mailto:michael@defprotocol.org) (original author)

« Overview of clojure.core, the standard Clojure library (/articles/language/core_overview/) || Collections and Sequences in Clojure » (/articles/language/collections_and_sequences/)

Links

- About (/articles/about/)
- Table of Contents (/articles/content/)
- Clojure Community (/articles/ecosystem/community/)
- Getting Started with Clojure (/articles/tutorials/getting_started/)
- Introduction to Clojure (/articles/tutorials/introduction/)
- Clojure Editors (/articles/tutorials/editors/)
- Basic Web Development (/articles/tutorials/basic_web_development/)
- Parsing XML in Clojure (/articles/tutorials/parsing_xml_with_zippers/)
- Growing a DSL with Clojure (/articles/tutorials/growing_a_dsl_with_clojure/)
- Web Development (Overview) (/articles/ecosystem/web_development/)
- Data Structures (Help wanted) (/articles/cookbooks/data_structures/)
- Strings (/articles/cookbooks/strings/)
- Mathematics with Clojure (/articles/cookbooks/math/)
- Date and Time (Help wanted) (/articles/cookbooks/date_and_time/)
- Working with Files and Directories in Clojure (/articles/cookbooks/files_and_directories/)
- Middleware in Clojure (/articles/cookbooks/middleware/)