Time to talk about namespaces, which are an interesting concept and powerful tool for modularizing in Shiny, but they do also have some nuance. This is important because without understanding some of that nuance, there might be some designs you want to do that appear impossible. The piercing of namespaces in particular is one that requires understanding this nuance, but will be in its own section. I should also say that while nuance is accurate, it is also an ironic word to use here as the way namespaces work in Shiny is quite simple, at least in appearance, but a desire to abstract the concept makes the borderline obvious implementation nebulous and potentially confusing.

The problem namespaces are intended to solve is to allow multiple objects to have the same ID. Object IDs must be unique, but sometimes, such as when modularizing code, you want an ID to be reused, though the value changes with each use. One way to solve this is present with custom functions, as each creates a temporary environment to execute its contents within, but these are temporary and thus destroyed when the function exits. As Shiny has both UI and server elements that you may want values to go between, the objects must be preserved, and namespaces enable this. Perhaps a scheme of creating persistent environments would be viable, but that is not the solution here and might be overkill as well.

The way namespaces work in Shiny is actually quite obvious, as I suggested earlier, because the namespace name is simply attached as a prefix to the object ID, with the \*\*ns.sep\*\* character separating them. This special character is apparently just a dash, but is left more abstract in the documentation for the \*\*NS\*\* function that handles this manipulation.

The \*\*NS\*\* function takes two arguments, appropriately named \*\*name\*\* and \*\*id\*\*. The former is the namespace name, and is always required. The latter is the object ID and is optional, with a default value of \*\*NULL\*\* and, except for an advanced use I will cover in the appropriate section, is almost unnecessary. If you give \*\*NS\*\* both of these arguments, it will return a string that is the namespaced object ID. If you only provide the \*\*name\*\* argument, however, then a function is returned. That function then has a singular argument of \*\*id\*\* and so will generate the namespaced ID for whatever namespace you identified when making this function.

```{r echo = TRUE, eval=TRUE}

NS(“test”, “id”)

ns <- NS(“test”)

ns(“id”)

```

This creation of the namespacing function is convenient as you will potentially find yourself needing to create multiple namespaced object IDs for a single namespace. The more explicit use of \*\*NS\*\* does have its uses though, if very uncommon.

Coming back just a little, notice the values returned by the two namespacing functions are, as I described earlier, the object ID with a prefix of the namespace name. While that suggests a simplicity to namespacing that is valid, there is more to it because if you were to try to list all the objects from a Shiny applet, such as with \*\*ls\*\*, the result might not include any of the namespaced objects. It seems Shiny segregates these objects, even to the point of not listing them without first getting them outside of the namespace. This can prove frustrating when troubleshooting, as you cannot use all of the same tricks to narrow down an issue as when working with base R (or at least this is the case for me). Fortunately super-assignment with `<<-` remains valid.

There is another method but it might not be as useful for troubleshooting, so I am going to save it for the appropriate Advanced section. Still, I will share now that it has to do with exploiting the fact the \*\*input\*\* and \*\*output\*\* objects are actually \*\*reactiveValues\*\*. Another of my Advanced sections is also specifically relevant to namespaces, as it describes how to pierce into them, such as to observe when a namespaced input changes from outside a module. The need for such piercing is rare, but I have encountered a scenario where that was the optimal solution. (Alternatively a mess of nested modules might have solved it, but was hardly ideal.)