Taming Name Spaces

Send comments to: bboyer@cincom.com

Introductory background

VisualWorks 5i introduced a much needed language feature, called a *name space*. By adding name spaces to VisualWorks, the system has become greatly more flexible in how it handles add-in components for a multiplicity of vendors.

However, the addition has also caused a significant amount of confusion. It was something new to a community, not all of whom have experienced the need. Further language modifications that were introduced (reasonably) at the same time, together with an awkwardness (to be kind) of the tools, and largely undocumented procedures for porting pre-5i code, unfortunately increased confusion and frustration.

In fact and practice, however, name spaces are not that big a deal, nor are they difficult to work with. This short document is an attempt to set the record straight, and provide some of the usage details that are missing from the *Application Developer's Guide*.

Before name spaces

Actually, the addition in 5i was more of a multiplication.

Smalltalk has always had name spaces, though there was only one: the monolithic Smalltalk pool. All globals (class names, global variable names, pool names) were resolved (their referents were determined) within that single context, the Smalltalk environment. Accordingly, each global name had to be unique to be identified from all others.

This worked fine as long as Smalltalk remained an environment of small, individual developers creating applications for their own use or in isolation from other applications. Each application developer or team knew exactly what globals would be in the system, and could provide the unique names required.

As Smalltalk went to the “enterprise,” and as component development and deployment became increasingly common, the luxury of isolation and control was lost. For example, a system integrator might want assemble a supply management system out of modules from multiple vendors. Each component may well have need to access records storing customer data, which each would quite reasonably represent as instances of a Customer class. In a single-vendor environment that class definition can be controlled and made consistent. In a multiple-vendor environment, however, that is much more difficult or impossible. The vendor, attempting to integrate the components from these vendors has a major problem. There are a variety of solutions available, none of them pretty.

A very real clash happened within the VisualWorks development team’s work. There was a desire to integrate SmallWalker, a web browser implemented in Smalltalk, into VisualWave, a VisualWorks add-in. But, both quite reasonably implemented an HTML class, and did so in incompatible ways. There were solutions, but they were very intrusive.

As long as all global names were resolved within the single Smalltalk name space, such naming collisions were inevitable, and increasingly frequent. This calls for a systemic solution rather than *ad hoc* work-arounds.

Enter: multiple name spaces

The general solution was not difficult, and had precedent in other programming environments. It was simply to restrict the global name resolution space, so that global names didn’t need to be unique in the whole Smalltalk environment, but only within a much smaller “name space.”

By restricting name resolution, references to vendor 1’s Customer class can cohabit the Smalltalk system with vendor 2’s Customer class, as long as they are in different name spaces. Each Customer class can be referred to unambiguously by identifying the containing name space.

There is a little more work in some cases, when both classes need to be referenced by the same application, or when an object in one name space needs to reference an object in another name space. References still need to be unambiguous. But, disambiguation is a relatively simple matter of specifying a name space, rather than changing all references to comply with a name change.

To accomplish this, VisualWorks 5i extended the system to support additional name spaces, providing for contexts more specific than just Smalltalk within which names are resolved. The universal Smalltalk name

space is retained as a “super name space.” Smalltalk is then divided into several other name spaces, each providing its own name resolution context. Additional name spaces can be defined within Smalltalk or within any of its sub-name spaces, to provide an appropriate separation of contexts.

Related changes

Items that had formerly be defined globally, by putting an object directly into the monolithic Smalltalk pool, had to be moved out of there and into name spaces. Specifically, these definitions are classes, global variables, and pools (pool dictionaries).

Class definitions needed to change rather little. A class is now defined by a message sent to a name space rather than to its superclass, and its superclass is specified as part of the definition.

Global variables are a bigger problem. They were created by explicitly adding an entry in the Smalltalk pool, usually by evaluating some object creation code in a workspace and entering that into the pool with a key. Accordingly, they had no explicit definition. And being global, they were assumed to be accessible to all objects in the system.

Pools contained shared variables, and the pool itself was defined globally. However, to access the contained variables, the pool had to be explicitly “imported” into a class in its definition.

Finally, class variables provided values shared by a class, its instances, its subclasses, and its subclasses’ instances. As such it was like a global variable except that references to it were restricted to those classes and instances. (A class variable could be accessed using a dotted-name notation, but this breaks encapsulation, so is a bad practice.)

Global variables, pool variables, and class variables have this in common: they are all shared by various objects in the system. It made sense to unify them in a single type of entity, called *Shared Variables*.

A brief terminological history and rationale

I should comment that we are now, starting with VW 5i.3, referring to these as “shared variables.” In fact, from 5i to 5i.2 we have called them “statics.” It wasn’t such a bad choice, but was disliked by many, including many VisualWorks developers.

The term “static” has history in C and Java, at least, where there is some affinity with what we called “statics” or “static variables” or “static bindings.” Essentially, their values are set independently of any runtime instance. So, that part made sense.

Regarding how to distinguish “constant” from “variable” instances of these things, we took the somewhat objectionable tack of generating a substantive term from an adjective, and made it plural. Hence, “statics.”

The choice of terms has generated controversy and dissatisfaction. Prior to every release there has been some discussion of changing the term. Only recently, during work on the 5i.3 release, has a truly viable alternative arisen, and something approaching consensus has formed around the term “shared variable.” Why this didn’t occur to us sooner is utterly unclear. Of course, there are objections to “shared” as well, but they seem relatively minor.

“Shared” suggests their referential scope; they can be referenced by any object in the name space in which they are defined or in a name space that imports that name space. And, they are variables, even though some may have rigid value assignments (be constant in a sense) and others flaccid assignments is acknowledged. (You may see occasional reference to “shareds,” but not often.)

We will refer to “shared variables” in this document, though we will follow 5i.2 when describing the UI, which refers to “statics”.

A quick and naive start

If you’re new to 5i, and especially if you’re new to VisualWorks in general, you really don’t want to be bothered with the intricacies of name spaces and their ilk. You’re wondering if there is a way to ignore all this stuff until you see a reason to bother with it (which you suspect won’t be until sometime after the next collapse and rebound of the universe).

The good news is that you can. At least to a point. As stated above, Smalltalk has always had one name space, namely Smalltalk. This approach will emulate that environment as much as possible, by creating everything in the Smalltalk name space.

I will give only quick descriptions of the process here, with minimal explanation at this time, since all you want to do is get some work done. When you’re curious, follow the links to get to the fuller explanation.

We’ll cover these issues:

- Defining a class category
- Defining a class
- Defining a class variable
- Defining a pool (pool dictionary)

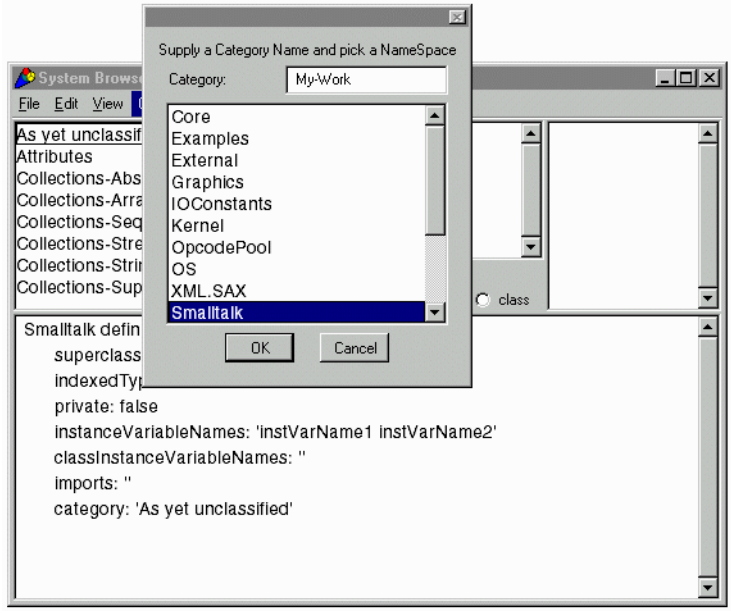
Defining a class category

If you use class categories, you will need to define one:

1. Open a simple Class Browser, by selecting **Browse → All classes** or clicking the equivalent tool bar button.

   This browser is the pre-5i category browser, and so will keep name spaces pretty much out of our view, though not entirely.

2. Select **Category → Add…**, to open category name prompter. Sorry, but you have to face name spaces, but just a little:



3. Enter a name for your category in the text field, and select **Smalltalk** in the name space list.

4. Click **OK**.

That's all. The category is added to the list.

Defining a class

As mentioned, at this point you will create all of your classes in the Smalltalk name space. This is precisely what happens in previous versions of VisualWorks, and in other Smalltalks in general, since class names are global variables in the system.

To create a class:

1. Open a simple Class Browser, by selecting **Browse → All classes** or clicking the equivalent tool bar button.
2. Select your class category, or select **Class → New class** and a class type (see Defining a class below for type descriptions).

   In either case, the class definition template is displayed in the code pane:

   ```
   Smalltalk defineClass: #NameOfClass
       superclass: #{NameOfSuperclass}
       indexedType: #none
       private: false
       instanceVariableNames: 'instVarName1 instVarName2'
       classInstanceVariableNames: ''
       imports: ''
       category: 'My-Work'
   ```

   The first word, Smalltalk, is the name space. If it's *not*, change it, because you want your class in the Smalltalk name space.
3. In the template:

   - Replace #NameOfClass with a symbol naming your class, such as #MyClass.
   - Replace NameOfSuperclass in #{NameOfSuperclass} with the name of your class's superclass. For example, if the superclass is Object, then make this expression #{Object}. (See "Referencing objects in name spaces" for this notation.)
   - Enter any instance variable or class instance variable names in the appropriate fields. (Note that class variables are not specified here anymore, because they are not part of the class definition.)
   - In the **imports:** field, enter private Smalltalk.* between the single-quotation marks. (See "Importing bindings" for more information.)

   At this point your definition looks like:

   ```
   Smalltalk defineClass: #MyClass
       superclass: #{Object}
       indexedType: #none
       private: false
       instanceVariableNames: ''
       classInstanceVariableNames: ''
       imports: ' private Smalltalk.* '
       category: 'My-Work'
   ```

4 Choose **Edit → Accept** to save the definition.

The definition undergoes a little formatting change and becomes:

```
Smalltalk defineClass: #MyClass
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            private Smalltalk.*
            '
    category: 'My-Work'
```

The class is now defined and added to the class list in the specified category, or in As yet unclassified if none was selected or specified.

Defining a class variable

Class variables are not defined in the class definition any longer, but are now objects with their own definitions, though those definitions are in the scope of a class. To define a class variable:

1 In a Class Browser, select your class (work class category and class).

2 Select the **Statics** radio button.

3 Create a shared variable category (protocol) using **Protocol → Add…**, if one doesn't already exist.

This is new in 5i, allowing you to categorize class variables as you do methods.

4 Select the category for the class variable.

The shared variable template is displayed in the code pane:

```
Smalltalk.MyClass defineStatic: #NameOfBinding
    private: false
    constant: false
    category: 'class vars'
    initializer: 'Array new: 5'.
```

5 In the template:

- Replace #NameOfBinding with a symbol naming your class variable, such as #MyClassVar.
- Replace the **initializer:** String with a String containing a Smalltalk expression that returns a value for the variable, or with nil. Set it

to nil if you want to set its value in a class method, as in pre-5i versions of VisualWorks.

6 Choose **Edit → Accept** to save the definition.

7 If you specified an initialization expression in step 5, select the variable in the browser and pick **Method → Initialize static**.

8 If you specified nil in step 5, define a class initialization method to set the variable's value, and send that message to the class.

The class variable is now defined and initialized. For more initialization options, see "Initializing shared variables" below.

Defining a pool (pool dictionary)

Pre-5i, a pool was defined as a global variable containing a dictionary. It was accessed by a class by being imported into the class, by including it on the poolDictionaries: line of the class definition. In this regard, little has changed, except that instead of a global you now use a name space as the pool, and shared variables as the pool variables.

To define a pool:

1 Open a Namespace Browser with a name space view (**Browse → All Namespaces** in the Visual Launcher).

Sorry, but you really do have to face name spaces for a moment here.

2 In the top-left pane, select the Smalltalk name space.

3 Select **Namespace → Add → Namespace** in the browser menu.

The name space definition template is displayed in the code pane:

```
Smalltalk defineNameSpace: #NameOfPool
    private: false
    imports: '
            OtherNameSpace.*
            private Smalltalk.*
            '

    category: 'As yet unclassified'
```

4 In the template:

- Replace #NameOfPool with a symbol specifying the pool name, e.g., #MyPool
- Remove OtherNameSpace.*
- Replace the **category:** string, so specify a meaningful category.

5 Choose **Edit → Accept** to save the definition.

The definition is saved, and the name space list is updated, with your new pool name space selected.

6 Construct a class method in an appropriate class that sends a series of at:put: messages to the new name space. For example:

```
initializePool
    Smalltalk.MyPool
        at: #first put: 'first';
        at: #second put: 'second'.
```

7 Send the defined message to the class to define the shared variables. Refresh the browser to see the definitions.

The pool and pool variables are now all defined and initialized.

To use the pool, go to the class definition of a class that will use it, and add it to the imports: line of the class definition. For example, in the class defined in Defining a class above, edit the definition to:

```
Smalltalk defineClass: #MyClass
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            private MyPool.*
            private Smalltalk.*
            '
    category: 'My-Work'
```

Save the class definition, and you are done.

Name spaces and their contents

In general terms, a name space is a context within which the referent of a term is determined.

For example, within the context of a gathering of my wife's family, the name "Bob," used without qualification, picks out one unique individual, while among my own family it picks out a different, though still unique individual. There is no confusion as long as these contexts are kept apart; our respective families serve as adequate name-resolution spaces, or name spaces.

Put our two families together, however, and the name "Bob" becomes ambiguous, and it's entirely possible for embarrassing confusions to occur. However, it is generally quite simple and straight-forward to avoid such confusions, and the resultant embarrassment, by explaining the scope more precisely. Including the family name is generally sufficient and not overly difficult.

In Smalltalk a name space works in the same way. Given a name of a variable, the object referred to by that name is identified within some naming scope. Traditionally, the name scope has been the either whole Smalltalk image in the case of global variables, an individual instance in the case of instance variables, or a class, its subclasses, and their instances in the case of class variables. To avoid confusion over the globals (class names, pools, and general globals), names were required to be unique within the system; you were only allowed to have one Bob.

With VisualWorks 5i, you are now allowed to have as many Bobs as you want, as long as each of them can be uniquely identified. Unique identification is possible by making sure that each Bob is defined and resolved in a single name space, and avoiding name space collisions.

Name space contents

A name space is a named object that represents the name resolution scope of a collection of shared variables. A name space is itself the value of a shared variable defined in another name space. A particular namespace, called Root, is the parent of all other name spaces, forming a name space hierarchy.

The Root name space initially contains two shared variables: Root, the value of which is the name space itself, and Smalltalk, the value of which is the Smalltalk name space.

To explore the structure of a name space, do an inspect on it. For example, evaluate this expression with doIt:

```
Root inspect
```

This opens a name space inspector (actually a PoolDictionaryInspector), showing the contents of the name space. Diving down through the Smalltalk entry, you observe additional shared variables whose values are the "top level" name spaces defined immediately in Smalltalk. Initially, the values of these are the name spaces that contain system code. As you create your own "top level" name spaces, shared variables for them are added to Smalltalk.

Continue the descent and you find definitions of name spaces, classes, and general shared variables.

To explore more deeply, seeing the structure of the entries, evaluate:

```
Root basicInspect
```

Doing this you see the representation of name spaces as a collection of bindings.

Shared variables

A shared variable is a variable that can be shared, or referenced, by multiple objects. In previous releases of VisualWorks, shared variables included class variables, pool variables, and global variables. In 5i, these various variable types were unified as a single type, currently called simply a "shared variable."

A shared variable's value is logically independent of any single instance of an object. Unlike instance variables, in which each object holds its individual state, and class instance variables, in which each class holds its state, shared variables can be shared among multiple objects.

Shared variables are implemented as *bindings*, which are instances of either class VariableBinding or its subclass InitializedVariableBinding. Accordingly, we sometimes refer to "a binding," and mean specifically an instance of one of these classes, rather than in the more general sense of a value assignment.

The value of a shared variable, or of the binding it refers to, is either a name space, a class, or an arbitrary object. In the third case, they serve the roles formerly served by globals, pools, and class variables

As class variables

A shared variable can be defined relative to a class, with the class serving as its name space. In this sense, shared variables replace the class variables of pre-5i releases.

For example, the class Date has a shared variable called MonthNames, which stores an array containing names for the 12 months. It would be wasteful to store the array in every instance that is cloned from it because the names are the same for all instances. Instead, the array is defined once in the shared variable. It is then accessible by instances of the class Date and its subclasses, and by instances of any other class that imports it.

Pre-5i, MonthNames was a class variable, and so shared by all instances of Date and its subclasses. Now it is defined as a shared variable under the Date class. Classes behave very much like name spaces, and in this capacity can include shared variable definitions.

Class variables are still inherited, and so are accessible to a class, its subclasses, and their instances. This is true even if the classes are in different name spaces and not imported.

As pool variables

Shared variables can also be defined directly in name spaces (non-class name spaces). For example, in the Graphics name space are defined a lot of classes, and two further name spaces: SymbolicPaintConstants and TextConstants. These name spaces exist solely as the name scopes for collections of shared variables.

Each shared variable is defined directly in the name space. Initialization values for the variables are provided either on the definition's initializer: line, as is done for most of the TextConstant variables, or in an appropriate class initialization method, as is done for the SymbolicPaintConstants variables.

For these variables to be accessed within a name space other than its defining name space, the variable must be imported, usually by a general import of its name space. (Refer to Importing bindings.)

As global variables

Globals are seldom used in VisualWorks, having been largely replaced by pool variables. Only a few "system globals" have remained in the system, such as Transcript and Processor, even before 5i. In general, they are a bad practice in object-oriented programming, because they break encapsulation, and so are to be avoided.

Instead of globals, these remaining system objects are defined as shared variables in a namespace that is almost certainly accessible to all name spaces. Transcript, for example, is defined as a shared variable in the Smalltalk.Core name space. To browse these definitions, select the Smalltalk name space in the Namespace Browser, and then select Core in the class/name space list, and browse the shared variables. You can do a search using **Protocol → Find method…** (a menu pick that is now badly named), and search for Transcript.

The resulting shared variables aren't truly "global" to the system, since it is easy to define a name space that doesn't import Core. It may not be clear why one would make such a name space, but it can be done, in which case Transcript, and similarly-defined shared variables, would not be visible to that name space.

As class and name spaces names

In pre-5i releases, class names were stored as global variables. In 5i, both class and name space names refer to shared variables whose values are classes and name spaces, respectively.

The template for defining classes is different in 5i as well, as is described below (see "Defining a class" below).

The name space hierarchy

VisualWorks name spaces are organized in a hierarchy. At the top of the hierarchy is a single name space, named Root.

Initially, it has a single sub-namespace, Smalltalk. For most practical purposes, the hierarchy starts with the Smalltalk name space, as the super-name space of all name spaces containing Smalltalk definitions. A fragment of the base VisualWorks name space tree, with a couple extra-base components added, looks like this:

```
Root
    Smalltalk
        Core
        OS
            IOConstants
        Graphics
            SymbolicPaintConstants
            TextConstants
        VisualWave
        XProgramming
            SUnit
```

In general, new name spaces should be contained within Smalltalk, either directly or indirectly, rather than directly in Root.

New “top-level” name spaces, those defined directly in Smalltalk, must be unique within the Smalltalk name space (there can only be one Smalltalk.Bob). The VisualWorks team and various vendors have reserved a number of top-level names. We maintain a Wiki site to allow you and others to reserve top-level name space names, and to see what names have been reserved, to help avoid name collisions at this level. (Go to Reserved Top-Level name spaces for VisualWorks 5i for the list).

The exception to keeping name spaces under Smalltalk would be a product that supports development and execution of another language, such as Java, within a Smalltalk image. Such a product might create a name space in Root, perhaps called JavaWorld, as well as various name spaces nested within it. The resulting name space hierarchy might look something like this:

```
Root
    Smalltalk
    JavaWorld
        java
            lang
            awt
        COM
            sun
            microsoft
```

If the Frost project were ever to be completed, it would probably take this approach.

What’s with Smalltalk.Root.Smalltalk anyway?

In the Root name space there are two shared variables defined: Root and Smalltalk. (To verify this, evaluate Root inspect.) Root refers to the name space itself, and Smalltalk refers to the Smalltalk name space.

It is sometimes convenient to be able to refer to the Root name space from Smalltalk, and so there is a shared variable defined in Smalltalk that refers to Root. This leads to a circularity that can be confusing, but need not.

When working in Smalltalk, references to named objects are assumed to start with Smalltalk, rather than Root. For most practical purposes, Root can be ignored.

If for any reason you do need to refer to Root, the circularity allows you to follow the same convention of starting with Smalltalk. So, to refer to the Root name space from within Smalltalk, the full path would be Root.Smalltalk.Root. But, because of the assumption of the Root.Smalltalk initial segment, you can refer to it simply as Root.

Working with name spaces

The biggest apparent change in 5i has been in the procedures for accessing the objects in Smalltalk: other name spaces, classes, and shared variables. These issues are covered in the following sections.

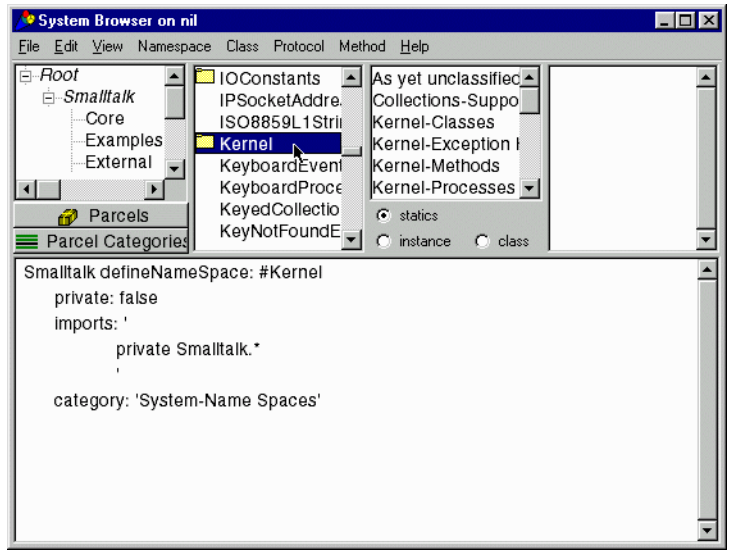
Browsing name spaces

The addition of name spaces and shared variables necessitated changes to the browsers. (In doing so, the browser architecture changed as well, but that's a subject for another time and place.)

The new System Browser introduced a selectable view in the top left pane. The current view is selected by clicking on an appropriate button to select, for example, a name space view, a parcel view, or the traditional category view. The three browser buttons on the default Visual Launcher (also available as menu picks on the Browse menu) open browsers that include more or fewer of these view options.

For working with name spaces, you want to open either the System Browser (aka the Namespace Browser), which has category and name space view options, or the Parcel Browser, which includes a parcel view as well. The Parcel Browser with the name space view is shown below. Both are available on the **Browse** menu of the Visual Launcher, and as buttons on the tool bar.

In all honesty, the browsers are not finished, and the current design makes them a little difficult to work with. Accordingly, the comments offered below are subject to change, as improvements are made, as they certainly will be.



Using the name space view

The first pane in this view shows the hierarchy of name spaces in an expandable tree view. The second pane is the traditional class list view, mostly. The difference is that it also shows name spaces, which are indicated by a folder icon.

When a name space is selected in the name space hierarchy view, the next pane shows only the name spaces and classes defined in that name space. Possibly shared variable definitions should be listed as well, but they are not, at least not now.

When a name space is selected in the class/name space view, the **statics** radio button is selected, and is the only button that is selectable. That's because name spaces only contain shared variable definitions. The next pane, the traditional method category, or protocol, view, lists the (class) categories of earlier versions, except that the categories now may contain name space and shared variable definitions as well.

When a class is selected in the class/name space view, the browser behaves more like the earlier browsers. You now can select the **instance** or **class** radio buttons, as well as the **statics** button. The method categories are the familiar things, containing method definitions as before. If any shared variables are defined in the class, selecting the **statics** radio button will show any categories, and selecting one of those shows its shared variables.

Using the category view

The category view interacts with the name space view, providing various filtering operations. These filterings can be useful for focusing on just certain parts of the system as you work.

If no name space is selected in the name space view, the category view works pretty much as it always has, except that name spaces are included in what had been only the class list (now a class/name space list). All categories are shown in the category list, and all classes and name spaces in a selected category are shown in the class/name space list. If no category is selected, then all classes and name spaces are shown.

If a name space is selected in the name space view, only those categories that are represented in the selected name space are shown in the category view. So, this filters the category view by the name space.

Conversely, if a category is selected in the category view, only those classes and name spaces defined in that category are shown in the class/name space list in the name space view.

To see all definitions in either view, make sure no name space is selected in the tree list in the name space view, and no category is selected in the category view.

Using the parcel view

The parcel view works independently of the other views, and very similarly to the Parcel Browser of previous releases.

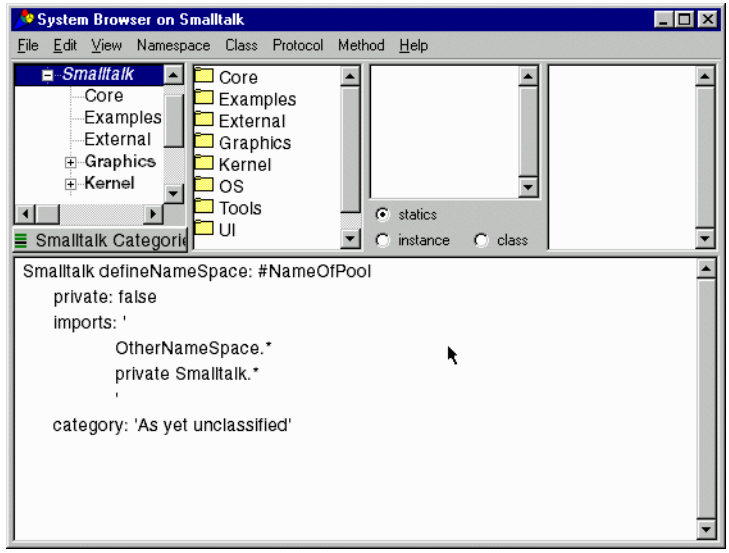
That it functions independently means that selections in other views have no effect on the display in parcel view mode. Selecting a name space or category, for example, does not filter the display in the parcel view, and selecting a parcel does not filter the display in the other views.

In parcel view mode, all classes and name spaces are displayed in the class/name space list pane. Various text formats are used, however, to indicate various states of code with respect to parcels. These are described in the *Application Developer's Guide*, and so are not repeated here.

Creating name spaces

Creating a name space is quite simple:

1. In a Namespace Browser, in the name space tree list, select the name space in which to create the new one. The selected name space's definition is shown in the code pane.
2. Select **Namespace → Add → Namespace**, to display a new name space definition template. Notice that the selected name space is named at the start of the template, and will be the super name space.



3. Replace #NameOfPool with a symbol giving the name for the new name space, such as #MyCompany.
4. Remove OtherNameSpace.* from the **imports:** line, but leave private Smalltalk.*. (See "Importing bindings" for more information.)
5. Change the **category:** string to something more descriptive.
6. Select **File → Accept** to save the definition and create the name space.

Your new name space is added to the list.

Naming a name space

There's no particular mystery to naming your name space(s). Most of your code will be application or add-ins, rather than extensions to the base system. So, your name space:

- needs to see a lot of the standard VisualWorks library
- does not need to be seen by the standard VisualWorks library
- needs to avoid name clashes with the VisualWorks and other 3rd party products.

The first of these is handled by imports, but is good to remember. The second point means that there is no reason, in general, for your code to be in an existing VisualWorks name space. The third suggests that you want a name space that will be clearly your own, separate from all others.

To deal with these points, we recommend that you create your own “top level” name space, immediately in the Smalltalk name space. To help keep it clear that this is yours, it is a good idea to use some form of your company name or similar designation, as suggested by the example in “Creating name spaces” above.

To help ensure that these top-level name space names are unique, we maintain a Reserved Top-Level NameSpaces wiki page. Instructions for reserving your top-level name are provided on that page. You reserve a name by adding it to the list. Make sure it hasn’t been taken by someone else, first, of course.

As long as your top-level name is unique, subsequent names you select for name spaces, classes, and shared variables under that top-level name are protected from clashes with those outside of that name space. So, you can name additional name spaces under your top-level name space in any way that makes sense to you.

When to create a new name space

You should always have at least one top-level name space for your own work. Beyond that, whether you need sub-name spaces depends on the name-access requirements of your products.

You may well have use for separate name spaces for each of your several products. Or, maybe not, depending on how tightly they interact.

In deciding, remember that all name spaces and classes created within a name space have access to all shared values defined in it. Consider that:

- If all of your classes need to see all of your other classes, then they all can reasonably be defined in a single name space.
- When you create classes that do not need access to some of your other classes, then it is time to consider creating further name spaces.
- If you create classes in one name space that need to access objects in another, you can import that other name space.

It's a judgement call that will become clear in practice.

Packaging/Parcelling a name space

Name space definitions can be saved out to archive files like other objects, using the usual procedures. They do not require special handling as pre-load actions or anything of that sort.

If you're using StORE and add a name space, you are prompted to select a package to place it in. Select the target package in the list and click **OK**. Presumably you've already created the package, but if not you can leave it unpackaged for the time being, by either selecting **(none)** as the package, or clicking **Cancel**.

If you do not use StORE, but instead need to save the name space in a parcel, you have a few options:

- To create a new parcel and add the name space, open a Namespace Browser. In a name space view, select the super-name space, then select the name space itself in the class/name space list. Next select **Class → Move to… → New parcel…** . A prompt will ask for a name for the new parcel.
- To move the name space to an existing parcel, open a Parcel Browser. Find and select the name space in the class/name space list (second pane), and select **Class → Parcel → Move to…**, and select the target parcel.

There are variations on both of these themes, and the precise procedures are subject to change as the tools develop.

Rearranging name spaces

Almost certainly you will need to move classes and name spaces around to other name spaces in the course of development. This is quite simple, using the Namespace Browser (that is, the name space view in the System Browser):

1. In the name space list, either select the containing name space of the object to be moved, or clear all selections.
2. In the class/name space list, find the class or name space to move.
3. Click and hold on the item, drag it to the target name space in the name space list, and drop it.

The class or name space is then moved, and the lists are updated to show the change.

To move a name space, you can also select it in the name space list, and select **Namespace → Move to…** . Select the target name space in the dialog that opens, and click **OK**.

Working with classes

Very little is changed with respect to classes except for the definition template, and the fact that class variables are now handled by shared variables.

Defining a class

To define a class:

1. In any system browser, select **Class → Add class**, and pick the class type:
   - **Fixed Size** (typical) for named instance variables
   - **Variable** for indexed instance variables holding objects
   - **Bytes** for indexed instance variables holding byte objects

   For more information on these instance variable types, refer to the VisualWorks *Application Developer's Guide*. These indexing types correspond to the older class definition method keywords subclass:, variableSubclass:, and variableByteSubclass:, respectively.

2 Complete the template that is displayed:

```
Smalltalk defineClass: #NameOfClass
    superclass: #{NameOfSuperclass}
    indexedType: #none
    private: false
    instanceVariableNames: ' instVarName1 instVarName2 '
    classInstanceVariableNames: ''
    imports: ''
    category: ' NameOfCategory '
```

3 Accept the definition (**Edit → Accept**)

The template is a message sent to a name space, Smalltalk by default. If you invoke the **Add class** command from a Namespace Browser with a name space selected, it is presented as the receiver, instead of Smalltalk. You can replace this with any name space name, to create the class in that name space, such as Smalltalk.MyNameSpace.

Provide the name for your class as a symbol following **defineClass:**. The name must begin with an upper-case letter.

Identify the superclass in the **superclass:** field using the literal binding reference notation shown. (This notation is described below, under "Binding references" below.)

The **indexedType:** field is filled based on the class type you selected: Fixes size = #none; Variable = #objects; Bytes = #bytes. You may change it here if you selected the wrong option.

Set **private:** to true to make the class unavailable for import by another class or namespace. (Refer to "Public and private shared variables".)

In the **imports:** field list any bindings you want to import, or make freely available to this class. (Refer to "Importing bindings".)

The **instanceVariableNames:**, **classInstanceVariableNames:** and **category:** fields are the same as in previous versions. Notice that there is no field for class variables, which are now represented as shared variables.

Classes as name spaces

We've already mentioned that in some situations classes can serve as name spaces. In fact, classes and name spaces are very similar, and efforts have been made to eliminate differences where possible.

The main difference between name spaces and classes is that classes are restricted as to the kinds of shared variables they can contain; they can contain only what we've called "general shared variables," which are its class variables. Classes cannot contain shared variables that have

name spaces or other classes as their primary reference. Note, however, that there is nothing to prevent a class or name space from being the value of one of its class variables.

What had formerly been a classes shared pools are now its imports, with all the same properties as the imports to a name space. An extension here is that a class can now import a single shared variable, by using a specific import, as well as being able to import the whole pool. Refer to "Importing bindings" for more information about general and specific imports.

A class's superclass is implicitly an import of the class that can never be declared private. This means that if *A* is a superclass of *B*, and *B* is a superclass of *C*, anything that *A* does not declare to be private will be visible to *C*, regardless of what *B* may declare private. This preserves from previous versions the rule that all class variables (assuming that they have not been declared private) are visible to all subclasses.

Working with shared variables

Name spaces are pools of shared variables, the primary reference of which are bindings. Each has a key and a value. In some cases the binding is a name space, in others a class, in yet others some arbitrary object. In this section we'll be primarily concerned with the latter case, since name spaces and classes have special importance, special definitions, and so on.

Defining shared variables

There are variations on this theme that you can find by experimenting with the browsers. Here are three standard procedures for adding shared variables that serve roles formerly played by global, class, and pool variables.

Defining a "global" shared variable

Shared variables defined in a name space do the service formerly provided by global variables and pool variables, except that the name scope is narrower. In VisualWorks, there are no longer any globals in the former sense; all variables are referenced within a specified scope.

To define a shared variable:

1 In the Namespace Browser, select a name space in the name space list to be the super-name space.

   For the widest availability, select the Smalltalk name space, though this is seldom recommended. Smalltalk.Core is usually as general as is needed, since it is imported by Smalltalk, and is the name space home for VisualWorks "globals." Select the most local name space that makes sense for the breadth of availability appropriate for this shared variable.

2 Select **Namespace → Add → Static** in the browser menu. The shared variable definition template is displayed in the code pane:

```
Smalltalk defineStatic: #NameOfBinding
    private: false
    constant: false
    category: 'As yet unclassified'
    initializer: 'Array new: 5'
```

3 Replace #NameOfBinding with a symbol specifying the shared variable name, such as #MySharedObject.

4 If the value is not to be exported for referencing from other name spaces, change the **private:** field to true; otherwise, leave it as false. (Refer to Public and private shared variables below.)

5 If the value of this variable should not be allowed to be changed by the application, change **constant:** to true; otherwise, leave it as false. (Refer to Constant and variable bindings below.)

6 Provide an appropriate category: string.

7 Enter an initialization expression, as a String, in the **initializer:** field, or enter nil. (Refer to Initializing shared variables below.)

8 Select **Edit → Accept** in the browser menu to save the definition and create the shared variable.

To see your new shared variable, open a Namespace Browser, select the **statics** radio button, select the variable's super-name space in the name space list, select its name space in the class/name space list, and select its category. You can also browse it in a Categories Browser.

Defining a "pool" and "pool variables"

Pools are collections of shared variables, and have existed in Smalltalk from the beginning. In most dialects, pools were implemented as instances of Dictionary, hence their commonly being referred to as "pool dictionaries."

In VisualWorks 5i, name spaces are, pools in this sense. An entity equivalent to a pre-5i pool, then, is a name space containing a set of shared variables.

For example, the pool TextConstants, which was implemented as a pool dictionary before 5i, is now implemented as a name space, TextConstants, containing a collection of shared variables. If you browse these definitions, as you can now do in a standard browser (you don't need to inspect the global TextConstants), the definitions can be further categorized. Whereas the pool dictionary contained a mix of character, emphasis, justification, and other constants, these are now separated within the TextConstants name space into categories.

You can define a pool by creating a name space, which is the pool, and then adding shared variables to it, which are the pool variables, as described above.

Alternatively, you can send at:put: messages to the pool name space, as was done with dictionaries in pre-5i release (refer to the SymbolicPaint class method initializeConstantPool for an example). To create a pool using this method:

1. In the Namespace Browser name space list, select the name space that will contain the pool.

   Select the most local name space that makes sense for the breadth of availability appropriate for this shared variable. While you can specify the Smalltalk name space, and so most approximate the "globalness" of pools in pre-5i releases, this is not generally recommended. A more restricted scope is usually better.

2. Select **Namespace → Add → Namespace** in the browser menu. The name space definition template is displayed in the code pane.

3. Complete the template, specifying the name of your pool as the name space name. (Refer to Creating name spaces for completing this template.)

4 Construct a class method in an appropriate class that sends a series of at:put: messages to the new name space. For example:

```
initializePool
    SmallTalk.MyNameSpace
        at: #first put: 'first';
        at: #second put: 'second'.
```

You can do this is a workspace, too, but then you won't have an easy way of reconstructing the pool.

5 Send the defined message to the class to define the shared variables. Refresh the browser to see the definitions.

At this point the pool variables are all defined and initialized. You may which to edit the definitions, however:

6 If any of the values are not to be exported for referencing from other name spaces, change their **private:** field to true; otherwise, leave it as false. Pool variables are usually intended for export. (Refer to Public and private shared variables below.)

7 If the values of any variables should be constant, not to be changed by the application, change **constant:** to true; otherwise, leave it as false. (Refer to Constant and variable bindings below.)

8 Provide an appropriate **category:** string.

9 Since you've already provided an initial value, leave the **initializer:** field set to nil. (Refer to Initializing shared variables below.)

10 Select **Edit → Accept** in the browser menu to save the definition and create the shared variable.

To see your new shared variables, open a Namespace Browser, select the **statics** radio button, select the pool's super-name space in the name space list, select the pool name space in the class/name space list, and select a category. You can also browse the pool in a Categories Browser.

Defining a "class variable"

Class variables in pre-5i releases stored values shared by a class, its subclasses, and all of their instances. In 5i, these are naturally replaced by shared variables whose name space is co-extensive with a class. In fact, classes can serve as name spaces in 5i, and so the replacement is quite exact.

To define a "class scoped" shared variable:

1 In any system browser, select the class that will serve as the name space for the variable, and select the **statics** radio button.

2 Select, or add and select, a category for the new shared variable, in the methods/shared variables list pane. The shared variable definition template is displayed in the code pane:

```
Smalltalk.MyNameSpace.MyClass defineStatic: #NameOfBinding
    private: false
    constant: false
    category: 'category description'
    initializer: 'Array new: 5'
```

3 Replace #NameOfBinding with a symbol specifying the shared variable name, such as #MySharedObject.

4 If the value is not to be exported for referencing from other name spaces, change the **private:** field to true; otherwise, leave it as false. (Refer to Public and private shared variables below.)

5 If the value of this variable should be allowed to be changed by the application, change **constant:** to true; otherwise, leave it as false. (Refer to Constant and variable bindings below.)

6 Enter an initialization expression, as a String, in the **initializer:** field, or enter nil. (Refer to Initializing shared variables below.)

7 Select **Edit → Accept** in the browser menu to save the definition and create the shared variable.

Your new shared variable is added to the list. It can be viewed in any class browser by selecting the **statics** radio button and its category

Constant and variable bindings

Sometimes it is desirable to set the value of a shared value and have it be immutable, or constant. The **constant:** field in the shared variable definition provides this option.

When set to false, the variable can be set and initialized by the usual means by any object in the system. (Refer to Initializing shared variables). When set to true, however, the value cannot be changed by the usual means.

For constant shared variables (which sounds odd, but they are still variables), changing the value requires rerunning the initializer, and so the variable is essentially protected from a runtime value change. The value is, for all intents and purposes, constant. Even a class initialization method that sets the variable will fail.

Note that you can change a shared variable's definition, and so change it from being variable to being constant. If you do so, be aware that methods that set the variable will now fail.

Public and private shared variables

Smalltalk has long lacked an enforceable distinction between public and private classes and methods. Variables have been either private (instance, class, and class instance variables) or public (global and pool variables), depending on the kind of variable. Name spaces and shared variables provide a way to fill some of this lack, by allowing you to control imports at two levels: definition and import.

At either its creation or when imported, a shared variable can be declared to be either public or private.

- If a binding is *public*, it is available for import by a name space or class.
- If a binding is *private*, it is not available for import by a name space or class.

Refer to "Importing bindings" for more information on importing.

Defining a binding as private or public

At one level, in its definition, each individual class, name space, and shared variable is declared as either public or private by setting the Boolean argument to the **private:** field. When set to false the binding is public, and so can be imported. When set to true the binding is private, and cannot be imported. At this level, privacy or publicity is set for the object itself, and so is absolute.

So, for example, a shared variable that is defined in MyNameSpace and declared as private is accessible only in the scope of MyNameSpace, and cannot be imported by any name space or class. It is hidden from anything that imports MyNameSpace.

Name spaces and classes are usually defined as public, since they should be imported by name spaces that need to access them. Pool variables also should be defined as public, since they also are meant to be imported. Class variables, shared variables that are defined within the scope of a class, are also usually defined as public, so they can be accessed by the class's subclasses, and their instances.

Defining a name space, class, or general shared variable as private is the exception, but an option if appropriate.

Importing a binding as public or private

At another level, when a binding is imported by a name space or class, it can be imported for either public or private access. In either case, the importing object has access to all public bindings in the imported object.

If the object is imported as “private,” the imported bindings are not exported by the importing object, and so are not further imported by an object that imports it. On the other hand if a binding is imported as “public,” then it is also imported into any object that imports that importing object.

A private import is indicated by putting the private keyword before the name space name:

```
Smalltalk defineNameSpace: #External
        private: false
        imports: '
                private Smalltalk.*
                '
        category: 'System-Name Spaces'
```

In most cases, a name space should import the contents of other name spaces as *private*. If another name space needs access to those imported bindings, it should gain access by importing their native name space, and not derivatively. This is not a hard rule, though, and your application structure may dictate other practices.

Don’t even think about it!

You will *never* have to do this. Doing so violates object-oriented design and is very dangerous. If a binding is defined or imported as private, it is meant not to be accessible by another importing name space. Those intentions should be respected.

However, it’s possible that rare circumstances in a development environment will really require that you get into a private name space. In this case you can prefix the name space name with drillDown, and commit all the horrors just warned about. This overrides the privacy specified either in a definition or import, giving free access to the imported binding.

The drillDown keyword implies private, so the bindings so imported are not re-exported.

Initializing shared variables

There are a variety of ways to initialize a shared variable.

By initialization string

In the shared variable definition you can specify an initialization string, which is any Smalltalk expression that returns an object. That object becomes the initial value when the variable is initialized.

To actually initialize the variable to the specified object, either:

- select the variable in a browser, and then select **Initialize static** in the <Operate> menu (or in the **Method** browser menu), or
- send the initialize method to a binding reference (see "Referencing objects in name spaces") of the variable, for example:

```
#{Smalltalk.MyNameSpace.MyBinding} initialize
```

These initialization methods work whether the variable is declared constant or not (whether the **constant:** field is true or false).

As part of class initialization

Especially in the case of class variables, initializing shared variables makes sense as part of class initialization. In this case, the value is set in the class initialize method, or in a method called by initialize.

For example, the Dummy class initialize method may simply set a value to a shared variable (DummyShared) defined in the class, like this:

```
initialize
    "Dummy initialize"
    DummyShared := String fromString: ' a b c d e'.
```

Note that to initialize a shared variable in this way, it must *not* be set as constant; the **constant:** field must be set to false.

By at:put: messages

We already noted that you can define a shared variable by sending an at:put: message to its containing name space, defining the variable as if it were an entry in a dictionary (see Defining a "pool" and "pool variables"). This method defines the variable and initializes its value at the same time.

Note that shared variables that are defined as constant cannot be set this way.

By lazy initialization

As long as the shared variable is not defined as constant, you can also allow its value to be set by “lazy initialization,” postponing initialization until the first access of the variable. This approach assumes that all accesses of the variable are via accessor methods, so that the variable is never directly referenced by, for example, its defining class or an instance of that class.

To use lazy initialization, set the **initializer:** field in the variable definition to nil. Then define accessor methods for the variable. You should define both set and get accessors, but the get accessors are where the variable’s default value is set, if it doesn’t already have a value.

For example, a get method for shared variable MyShared might be:

```
getMyShared

    ^MyShared isNil
        ifTrue:
            [MyShared := String new]
        ifFalse:
            [MyShared]
```

The initialization value, of course, will be whatever is appropriate for the application.

Since shared variables are accessible to both classes and their instances, such accessors probably will need to be defined for both the class and for instances (on the class and the instance side).

Note that shared variables that are defined as constant cannot be set this way.

Referencing objects in name spaces

Within the native naming scope of a binding, whether for a name space, a class, or a shared variable, the object can be referenced to by unqualified name. However, most objects will also have to reference objects that are not native to the same name space.

For example, within the VisualWorks system, virtually any object needs to reference objects in the Core name space, even though it is native to another name space. Your application objects, which will be native to your own name space(s), have to reference a wide range of objects in VisualWorks name spaces, and possibly objects from other vendors.

There are a variety of ways to reference these named objects, as described in the following sections.

Dotted names and name space paths

Binding names (names of name spaces, classes, and shared variables) use a dotted notation that describes the path through the name space hierarchy to the desired binding. While you seldom reference a binding using its full dotted name (except when specifying imports), in order to understand the other referencing methods you need to know about dotted names.

The full path a dotted name begins with the Root name space, continuing through the hierarchy to the target binding. For example, the full reference to the ButtonHilite constant (in its native name space) is:

```
Root.Smalltalk.Graphics.SymbolicPaintConstants.ButtonHilite
```

However, the VisualWorks system, when parsing a compound dotted-name, assumes the Root.Smalltalk initial segment. So, in practice, the above reference is shortened to:

```
Graphics.SymbolicPaintConstants.ButtonHilite
```

This is the form of reference used in import statements, providing the path starting immediately after Smalltalk.

If a binding is imported, the dotted name can specify the importing name space path, instead of the native name space path. So, for example, if Smalltalk.MyNameSpace imports ButtonHilite, the dotted name MyNameSpace.ButtonHilite would also be a legitimate dotted name, and would reach the variable.

Using dotted names in code to reference variables that are neither defined in nor imported into the current name space, is permitted but discouraged, because this use breaks encapsulation. There are, however, occasions when they are needed. In source code, it is sometimes necessary to refer to a variable that is not visible from the current name space. For example, if a developer is adding a method to a class that he does not own, and he may not have the freedom to add a new import to the class's environment. In future releases we intend to provide a better mechanism for extending classes, allowing extensions to use variables not normally visible to the class, but they are not currently available.

They have also been needed in a workspaces before 5i.3, to evaluate an expression that includes a variable from an arbitrary name space. In 5i.3, however, workspaces import name spaces, so this is no longer an issue.

Binding references

In an environment with name spaces, we need a way to reference a shared variable that makes no assumptions about which name space contains its definition. A *binding reference* provides this facility.

A binding reference is a named object that holds a starting point and a list of names. It can identify an arbitrary shared variable relative to an arbitrary name space, by identifying a navigation path from the name space to the shared variable.

Most of the protocol for binding references is defined in the class GenericBindingReference, with more specific protocol defined in BindingReference and LiteralBindingReference. The common protocol includes useful questions such as:

isDefined
: Does the variable exist in the system?

binding
: Answer the VariableBinding for the shared variable, or raise an error if it doesn't exist.

bindingOrNil
: Answer the VariableBinding for the shared variable, or nil if it doesn't exist.

value
: Answer the value of the shared variable, or raise an error if it doesn't exist.

valueOrDo: aBlock

Answer the value of the shared variable, or the value of aBlock if it doesn't exist.

A binding reference, when asked for its binding, iterates through its list of names. For each name, it asks the current name space for the variable of that name. If the name is the last in the list, it answers the shared variable. If the name is not last, it uses the value of the variable as the new current name space, and repeats the process with the next name in the list.

There are two forms of binding reference, distinguished by how their environment information is stored, corresponding to classes BindingReference and LiteralBindingReference. The environment is the name space scope within which the binding reference is evaluated.

Instances of BindingReference store their environment in their environment instance variable. Accordingly, each instance knows its compilation scope. Instances of LiteralBindingReference, on the other hand, store the *method* that created them in a method instance variable, and their environment is then determined from the compilation scope of the method.

A simple way of creating a BindingReference is by sending asQualifiedReference to a String, for example:

```
'MyBinding' asQualifiedReference
```

The syntax #{MyBinding} creates a LiteralBindingReference.

Inspect the results of each expression to compare their object structure. Be aware that although the printing representation of both is the same, they are not equal, being different classes of objects. (This inequality may change at some later time.)

Both of these allow referencing the shared variable without the programmer having to know or specify the path to the variable. The name resolution environment determines the object referenced. Consequently, it is not necessary to know whether the variable's environment is an import or native.

Note that the referenced binding does not need to exist when the binding reference is created. It's just a reference object, and is resolved at compile-time.

In both cases, name space path information can be included as well, using the dotted-name notation. Remember that compound dotted-names always go back to Smalltalk, so the entire path from that point must be given. For example:

```
'MyNameSpace.MyBinding' asQualifiedReference
```

or

```
#{MyNameSpace.MyBinding}
```

Other instance creation methods are available (browse class BindingReference and GenericBindingReference). For example:

```
BindingReference path: #(Core Object)
```

which creates a BindingReference to Core.Object. Providing the path is often necessary when specifying imports in name space and class definitions.

> **Note:** Class QualifiedName in VW 3.0 has been replaced by class BindingReference in 5i, so be aware of this if you referenced that class in your code.

Binding reference resolution

Binding reference are resolved in this order:

1. If a bindings is defined in the name space, the binding reference takes it.
2. Next, bindings imported by a specific import are selected.
3. Finally, bindings imported by a general import are used.

See Binding rules and errors below for restrictions on imports.

When to use BindingReference or LiteralBindingReference

The differences between BindingReference and LiteralBindingReference make these objects not fully interchangeable.

The #{...} syntax is appropriate for asking questions of binding references, such as isDefined, where the reference is short lived.

If a short-lived method (such as a DoIt) is used to create a reference for long-term storage (such as in a Dictionary), use asQualifiedReference or fullyQualifiedReference methods to create a BindingReference. Because a LiteralBindingReference holds a reference to the method that created it, putting this reference in long-term storage would prevent the creating method from being garbage collected.

If the reference will be stored in a long-term data structure, but the method which creates the reference is presumed to be equally long-lived, the choice is yours, but using asQualifiedReference, may be the better choice.

If the exact path of the binding reference is not known at compile time, but is partially or fully computed at runtime, then you will have to use a BindingReference, since #{} syntax is not an option.

Importing bindings

While it would be possible to require that you reference each object by explicitly describing the name space path from Root to the target object, that would be inconvenient, and would violate the object-orientation principle of encapsulation. Instead, it is preferred to import the bindings into the local object's name space so they can be referenced by unqualified name.

Name space and class definitions provide for importing bindings, by including the bindings in the imports list. The binding name is specified using the dotted-name notation, usually starting with the first name space in the path under Smalltalk (Smalltalk is assumed, see Dotted names and name space paths). For example, the XML name space imports its sub-name space like this:

```
Smalltalk defineNameSpace: #XML
    private: false
    imports: '
            private Smalltalk.*
            XML.SAX.*
            '
    category: 'XMLParsing'
```

This is a *general* import, using the asterisk (*) pattern matcher to import all bindings defined in the indicated name space. In this example, all bindings in the Smalltalk and in the Smalltalk.XML.SAX name spaces are imported. In particular, these lines import all name spaces defined under Smalltalk (it would import classes, too, if there were any), and all classes defined in the SAX name space are imported into the XML name space.

Note also that SAX is imported as *public*. Doing this has XML also export those imported bindings, so that they are also imported by any class or name space that imports XML. In this case this is the right thing to do since there's no reason for an application to have to import SAX separately from XML; if it needs XML, it will need SAX, too.

As explained in Public and private shared variables, including the private keyword in front of the Smalltalk.* import prevents XML from exporting those bindings. They can be reasonably expected to be imported by each name space. For this reason, private Smalltalk.* is included in the name space definition template.

On occasion a name space or class may need to import only a single binding from another name space. This is done using a *specific import*. For example, the TextConstants pool only needs access to one class in the Core name space, so it uses a specific import:

```
Smalltalk.Graphics defineNameSpace: #TextConstants
    private: false
    imports: '
        private Core.Character
        '
    category: 'As yet unclassified'
```

Once properly imported, the imported name can be used directly, without further path qualification.

Given this general explanation, the following specific cases may be helpful.

Importing classes and name spaces

When we mention “importing a name space,” we usually really mean importing the contents of the name space, rather than only the name space itself. The contents of a name space may include:

- class definitions
- other name space definitions
- general shared variable definitions

When defining a name space, you almost certainly want to import the VisualWorks system classes. To do this, include:

```
private Smalltalk.*
```

in the imports list. Smalltalk itself imports all of its sub-name spaces’ bindings publicly, so this one line, a general import, brings in all of the system classes, pools, and system variables (such as Transcript).

Importing class variables

It is seldom necessary to import a class variable explicitly. They are implicitly imported into the class in which they are defined, and inherited by its subclasses. Since they are used to store class state information, that is sufficient. If you do need to import a class variable, import it like a pool variable, with the class as its pool.

Importing pool variables

Pool variables are general shared variables defined in a common name space, which is their pool. Depending on circumstances, you will either want to import all of the pool variables, or only one or a few.

To import all pool variables in a pool, use a general import. So, for example, to import all of the TextConstants, use this general import in your class or name space definition:

```
imports: '
        private Graphics.TextConstants.*
        '
```

(See the definition of class TextAttributes.) This permits you to reference each text constant by unqualified name.

To import a single pool variable, use a specific import. For example, to import only the text constant Bold, use:

```
imports: '
        private Graphics.TextConstants.Bold
        '
```

This permits you to reference this one variable by unqualified name.

What about those “circular” system imports?

It may look funny that the Smalltalk name space definition imports all of the system name spaces:

```
Smalltalk.Root defineNameSpace: #Smalltalk
    private: false
    imports: '
            Core.*
            Kernel.*
            OS.*
            External.*
            Graphics.*
            UI.*
            Tools.*
            Database.*
            Lens.*
            '
    category: 'As yet unclassified'
```

while each of those name spaces’ definitions imports Smalltalk, e.g.:

```
Smalltalk defineNameSpace: #Kernel
    private: false
    imports: '
            private Smalltalk.*
            '
    category: 'System-Name Spaces'
```

What's happening is this? Smalltalk imports each of its sub-name spaces imports as public (for further export), so all of those bindings are accessible directly from Smalltalk. Each sub-name space in turn imports, privately, all of the bindings from Smalltalk, which includes all the bindings Smalltalk imported from their siblings.

Now, for example, an instance of External.CComposite can reference Core.Array by its unqualified name, Array. All of the base VisualWorks classes, pools, and such, are accessible directly from Smalltalk, as before.

For the most part, this also simplifies migrating to VisualWorks 5i from prior releases, by making sure all the system classes are available. When code is imported, it is loaded directly into the Smalltalk name space, where it has access to the essential system classes, and so mostly works without modification.

Binding rules and errors

Each imported binding name must be unique in the collection of names defined in and imported into the name space. Accordingly:

- If two specific imports refer to shared variables of the same name, the name space's definition is in error.
- If a specific import refers to a shared variable whose name is the same as a shared variable defined locally in the name space, this is an error.
- If two general imports bind the same name to different shared variables, and a local definition or specific import of that name does not exist, it is an error for a method to use that variable name. However, the name space may define a specific import that clarifies which of the two shared variables is desired.
- Local definitions of a shared variable and specific imports are searched before general imports when binding a name to a shared variable.

So, what's so hard about name spaces?

After this many pages it may be foolish to answer, "Nothing!" Still, I think that's essentially the case, even though they do, in practice, add a level of complexity to Smalltalk.

Hopefully, having this additional information about their rationale, structure, and use, much of the confusion surrounding name spaces and VisualWorks 5i will be alleviated, and their value recognized.