

Introduction to Shared Singleton Class: Dynamic Global Variables

By Nhat Do, Technical Services Engineer, 4D Inc.

Technical Note 25-04

Table of Contents

Table of Contents	2
Abstract	3
Introduction	3
Shared Objects, Classes & Singletons	4
What Is a Shared Object?	4
What Does a Shared Class Do?	4
How Does a Singleton Class Work?	5
The Shared Singleton Class	6
Dynamic Global Variables	6
Overview & Background	6
The Class Implementation	7
Demo Application	9
Startup Example	9
Scalars Example	9
Object Example	10
Collection Example	10
Conclusion	10

Abstract

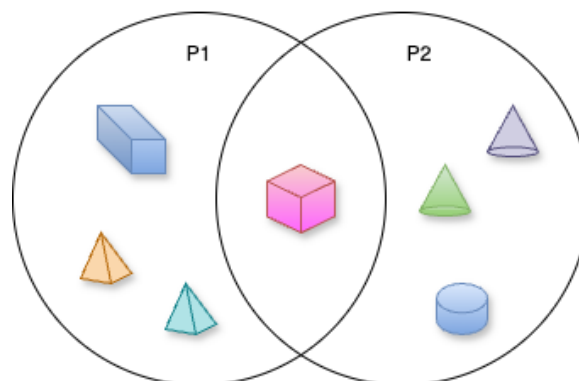
The 4D programming language is constantly evolving to provide a better coding experience for developers. Shared Classes and Singleton Classes were introduced in 4D 20 R5 to expand the scope of developer capabilities even greater. A shared class allows developers to instantiate a shared object with its own set of properties and functions; when combined with a singleton class, developers can ensure the shared object has a unique instance throughout every process. This technical note will discuss the fundamentals of the Shared Singleton Class and how to use it to effectively implement a store of dynamic global variables. These are values that can be accessed and modified safely across all processes.

Introduction

In 2018, the 4D programming language began to undergo a paradigm shift with the advent of ORDA—Object Relational Data Access—in version 17. Developers were now able to smoothly access and modify data in their applications through an abstraction layer called the datastore, using object notation. But the move towards a more streamlined, object-oriented approach to programming in 4D did not stop there. With the release of 4D 18 R3, the ability to create classes became available, allowing developers to define their own custom objects with common attributes and actions. This bolstered the concepts of encapsulation, reusability, and inheritance within the 4D programming language. Yet, the addition of Classes was only the beginning to the object-oriented programming evolution in 4D.

With the arrival of 4D 20 R5, creation of shared classes and singleton classes is now supported. Shared classes allow developers to instantiate shared objects, whose contents are communal between processes. Singleton classes give developers the ability to restrict the instantiation of a class to a singular instance. The two concepts can be married to create shared singleton classes—user classes that enable instantiation of a singular shared object for the class. Since it is certain to have a unique instance across all processes, a shared singleton class instance is particularly useful for storing dynamic global variables.

Shared Singleton



This technical note will go over the basics of shared objects, shared classes, singleton classes, and finally shared singleton classes. Then, it will discuss how to implement a shared singleton class to maintain dynamic global variables—values that are readable across all processes and

that can be modified between processes. The advantages of this implementation, especially over older approaches such as interprocess variables, will also be discussed. Additionally, this technical note includes a demo application that shows examples of the implementation in a few different scenarios. But first, foundational knowledge of shared objects, shared classes and singletons will be overviewed.

Shared Objects, Classes & Singletons

What Is a Shared Object?

To comprehend shared classes, the concept of shared objects must first be understood. A shared object has all the attributes of a regular object data type. It contains any amount of “property: value” pairs, and it can be handled by using “OB” commands or dot (object) notation. However, unlike a regular object, a shared object also has the characteristic of having its contents accessible and modifiable between processes, meaning that it is compatible with preemptive processes and can be used in writing thread-safe methods. This is because when a shared object needs to be modified, the “Use...End use” structure is mandatory. For example:

```
// Method: updateProps
#DECLARE($mySharedObj : Object; $prop1 : Text; $val1 : Variant; $prop2
: Text; $val2 : Variant)

Use ($mySharedObj)
    $mySharedObj[$prop1] := $val1
    $mySharedObj[$prop2] := $val2
End use
```

In the above example, “updateProps” is a method that could be called using the “New process” command or “CALL WORKER” command. The parameter “\$mySharedObj” is a shared object that is passed into “updateProps” along with some other parameters. In this method, “\$mySharedObj” is safely modified by protecting it with the “Use/End use” keywords, locking it from other processes that attempt to modify it while it is in “Use”. A shared object can be created by using the “New shared object” command, or it can be instantiated from a shared class.

What Does a Shared Class Do?

Like a regular class, a shared class has a class constructor, but with the addition of the “shared” keyword in front. This signifies that any instance created by calling the “new()” function on the shared class will be a shared object. As opposed to shared objects created by the “New shared object” command, an object instantiated from a shared class has the benefit of being able to use class functions. If functions need to modify an object of the shared class via the “This” command, since “This” would be referencing a shared object, the “Use/End use” keywords should be used. For example:

```
// Class: MySharedClass
shared Class constructor($prop : Text)
    This.prop := $prop
```

```
Function add($foo : Text)
  Use (This)
    This.prop+=$foo
  End use
```

In the example above, the “add” function of the shared class “MySharedClass” modifies the “prop” property of a shared object via dot notation on “This”. Therefore, “This” should be protected by using the “Use...End use” structure. However, the “Use/End use” keywords can be omitted entirely by adding the “shared” keyword before the “Function” keyword in a shared class. For example, the “add” function of “MySharedClass” above can be rewritten as the following:

```
shared Function add($foo : Text)
  This.prop+=$foo
```

Essentially, the “shared” keyword, in this case, functions as syntactic sugar to eliminate the need to write “Use...End use” every time the shared object of a shared class (“This”) needs to be modified inside a class function. This makes it quicker to write—as well as easier to read—code that is centered around shared class objects. Shared classes are not the only special types of user classes introduced recently. In addition to shared classes, user classes can also be designated as singleton classes.

How Does a Singleton Class Work?

A singleton class is denoted by the “singleton” keyword in front of the class constructor. Just as the name suggests, there can only be a single instance (for its originating process) of a class singleton at a time. A singleton can be instantiated by either calling the “.new()” method of the singleton class, or calling the “.me” property of the singleton. If the singleton has not been instantiated yet, calling the “.me” property is like calling “.new()” without any parameters. After it has been instantiated, calling “.me” just returns the singleton instance. For example, given the following singleton class named “MySingletonClass”:

```
// Class: MySingletonClass
singleton Class constructor($param : Text)
  This.foo:=$param
```

MySingletonClass’s instance can be created by either executing code like this:

```
$mySingleton:=cs.MySingletonClass.new("bar")
```

Or by calling its “.me” property like this:

```
$mySingleton:=cs.MySingletonClass.me
```

Initially, if both happen to be called consecutively, such as the following:

```
$mySingleton1:=cs.MySingletonClass.me
$mySingleton2:=cs.MySingletonClass.new("bar")
// cs.MySingletonClass.me.foo = ""
```

The first would always take precedence over the latter. That means that, in the above example, the singleton instance's "foo" property would become an empty string ("") value, instead of "bar". Furthermore, any subsequent updates to the singleton would be reflected in both variables, like in this follow-up example:

```
cs.MySingletonClass.me.foo:="baz"  
cs.MySingletonClass.me.qux:="plugh"  
// $mySingleton1.foo = "baz", $mySingleton1.qux = "plugh"  
// $mySingleton2.foo = "baz", $mySingleton2.qux = "plugh"
```

Thus, a singleton class prevents an instance from being created more than once in a particular process and centralizes its value(s) in one place. But what if a singleton is necessary for every process? That is where shared singleton classes come in.

Note: Besides basic (process) and shared singleton classes, there is another type called the session singleton class, which will not be covered in this technical note.

The Shared Singleton Class

To create a shared singleton class, simply prepend the "shared" keyword to the "singleton" keyword in front of the Class constructor. For example:

```
// Class: MySharedSingletonClass  
shared singleton Class constructor($foo : Text)  
  This.foo:=$foo  
  
shared Function myMethod($bar : Text)  
  This.bar:=$bar
```

This type of user class enables instantiation of a class shared object that functions as a singleton as well. From the above example, if "MySharedSingletonClass" is instantiated with a ".new()" or a ".me" call, a shared object would be created that is the sole instance for that class. Shared singleton classes essentially get the best of both worlds: the ability to instantiate an object that can be shared between processes, with access to shared functions; and the ability to restrict its presence to a single source. This is especially useful for implementing a store of dynamic global variables, where these types of values need to be shared and updated between processes, and where having a single source of truth makes them easier to control and maintain.

Dynamic Global Variables

Overview & Background

In many applications, it is not uncommon to maintain **variables** that are **globally** accessible and whose values can change **dynamically**. Common use cases of dynamic global variables include management of application-wide configuration settings, storage of session/user state and caching/memoization. Particularly, 4D programming sometimes requires variables to be shared and updated across several processes, such as when multiple forms are opened in concurrent processes. There are a few ways to implement

dynamic global variables in 4D. This technical note discusses one of the most recent and practical approaches: using a shared singleton class.

Historically, the use of interprocess variables was the main approach to dynamic global variables. However, as 4D technologies progressed, interprocess variables became deprecated. The main reason was that interprocess variables are not available from preemptive processes. This means that they cannot be used to write thread-safe methods. In a modern world where multi-core computers are widely in use to run applications with multi-threaded capabilities, another approach besides interprocess variables is necessary.

Fortunately, 4D applications can be developed nowadays using a shared singleton class for dynamic global variables. Since the instance of this type of class is technically a shared object, it has the benefit of being compatible with preemptive processes. Furthermore, being a singleton helps prevent different processes from accidentally manifesting more than a single source of truth. The shared singleton class also has the advantage of being able to use shared functions, in which additional features can be built in as well.

The Class Implementation

Since most of the heavy-lifting is already being done by 4D technology under-the-hood, it is quite simple to start implementing a shared singleton class for dynamic global variables, but it can always be built upon to add more features. At the bare minimum, it must have a class constructor, like the following:

```
// Class: GlobalVars
shared singleton Class constructor
```

In the example above, the shared singleton class “GlobalVars” is created with just a class constructor and the “shared singleton” keywords. That is all that is really needed to make a class for global variables. Then, “GlobalVars” can be used like this:

```
var $globals:=cs.GlobalVars.me
Use ($globals)
    $globals.foo:="bar"
End use
```

In the above example, the singleton was instantiated, and then the global variable “foo” was assigned the value “bar”, created as a property on the singleton object. Now, when the value of “foo” needs to be accessed, it can be read from “cs.GlobalVars.me.foo” in any process. Note that assigning “cs.GlobalVars.me” to a local variable, “\$globals”, helps to write the code a bit more quickly.

Of course, like any other class, the constructor can be fitted to take parameters and initialize its own internal properties:

```
// Class: GlobalVars
shared singleton Class constructor($foo : Text)
    This.foo:=$foo
```

So that the singleton can be instantiated with global variables immediately:

```
var $globals:=cs.GlobalVars.new("bar") // cs.GlobalVars.me.foo = "bar"
```

```
// assign new variable "num"
Use ($globals)
    $globals.num:=123
End use
```

Don't want to write out the entire "Use... End use" structure every time a global variable needs to be assigned? Let's create a class shared function called "assign" to make it easier:

```
// Class: GlobalVars
shared Function assign($name : Text; $value : Variant)
    This[$name]:=$value
```

Now, the singleton can use its shared function "assign" to set global values and bypass the need for "Use/End use" keywords:

```
$globals.assign("num"; 123) // cs.GlobalVars.me.num = 123
```

When a global variable needs to be updated, simply call the function with the variable name and pass in a new value:

```
$globals.assign("num"; 456) // cs.GlobalVars.me.num = 456
```

Note: This works for any case where the value to assign to a global variable is a scalar. Shared singletons can also store collections and objects, but those need to be treated carefully as shared collections/objects. The "Demo Application" section of this technical note provides details on how to manage global values that are collections or objects.

This completes a basic implementation of dynamic global variables using a shared singleton class. However, there is a lot more that can be added to improve the developer experience. As an example, let's say that every time an existing global variable is updated, a type validation should be performed to verify that the new value's data type is the same as that of the existing one. To implement this, the "assign" function can be modified like so:

```
// Class: GlobalVars
shared Function assign($name : Text; $value : Variant)
    // new variable
    If (Undefined(This[$name]))
        This[$name]:=$value
    Else
        // validates input type on existing variable
        If (Value type(This[$name])=Value type($value))
            This[$name]:=$value
        Else
            ALERT("Wrong input type for existing variable: "+$name)
        End if
    End if
```

Now, if the "num" variable from the above examples tried to be re-assigned like this:

```
$globals.assign("num"; "789")
// ALERT - Wrong input type for existing variable: num
```


The attempt would not work, and the developer would be alerted. This is useful for making sure that data types for created variables do not change unexpectedly, preventing potential debugging headaches down the line. Besides type validations, there are many other possibilities to improve the implementation of a dynamic global variables class. Another one that is mentioned in the next section (see “Scalars Example”) is the application of variadic parameters to allow the assignments of multiple global variables together in one line.

Demo Application

This section supplements the demo application that accompanies this tech note. Due to the nature of the topic, the demonstration of how the “GlobalVars” class works is shown mostly through inspection of expressions in the debugger. The demo application should be opened in 20 R5 or newer.

Startup Example

This example shows how values from the environment can be initialized as global variables in the shared singleton class. On startup of the application, the “GlobalVars” singleton is instantiated with values obtained from the JSON of the “info.txt” file located in the Resources folder:

```
// Method: __init_vars_from_env  
var $file:=File("/RESOURCES/info.txt")  
cs.GlobalVars.new($file)
```

Then, each of these values are read from the singleton and are displayed in the welcome message via another method. Modify the values in the JSON and restart the application to see how it changes the initial global variables.

Scalars Example

Run the “_create_vals” method to see how global variables with scalar values are managed. Notice that the “assign” function uses parameter indirection syntax to handle variadic parameters, like so:

```
// Class: GlobalVars, Function: assign  
For ($i; 1; Count parameters; 2)  
    This[${$i}] := ${$i+1}  
End for
```

This way, multiple values can be assigned to global variables in one line:

```
// Method: _create_vals  
$globals.assign("foo"; 123; "bar"; "hello"; "bool"; False)
```

Watch the expression “\$globals” in the expression pane while stepping through the debuggers to see how the scalar globals are created and updated in different processes. For example, step through the “_create_vals” process before finishing the “update_vals” process to get a different result.

Object Example

This example shows how object values are handled as global variables. In the “_build_car” method, a new car object is created as a shared object, because shared objects like the “GlobalVars” singleton do not support non-shared objects as property values:

```
// Method: _build_car
var $new_car:=New shared object("make"; "Fiat"; "model"; "Panda")
$globals.assign("car_obj"; $new_car)
```

When it is time to update properties of the car object, since it is a shared object, the “Use/End use” keywords are needed:

```
// Method: _mod_car
$car:=$globals.car_obj
Use ($car)
    $car[$prop]:=$val
End use
```

The “shared” keyword for the “assign” function only applies to the parent shared object (“This”) and does not allow omission of “Use...End use” when modifying child shared objects.

Collection Example

This example demonstrates how collection values are managed as global variables. Like an object value, a collection value must be created as a shared collection to be assigned as a property value on the shared singleton:

```
// Method: _pick_fruits
var $new_basket:=New shared collection("apple"; "orange")
$globals.assign("fruits_coll"; $new_basket)
```

However, since collection functions that modify shared collections automatically trigger “Use...End use” syntax internally, updating a collection global this way does not require it explicitly:

```
// Method: _pick_fruits
$fruits:=$globals.fruits_coll
$fruits.push("cherry")
```

Observe that for the Scalars/Object/Collection examples, after initializing some values, CALL WORKER is used to create another process. This is to show how global variables are updated across different processes.

Conclusion

This technical note described the general concepts and benefits of shared singleton classes. It demonstrated how a shared singleton class can be implemented to manage dynamic global

variables. This information should allow developers to write their own shared singleton class to handle variables that can be shared between processes.