

# Programming Style

## Quick Look

An effective programming style helps you write code that is easier to understand, debug, maintain, and port from system to system. This article discusses the general features of an effective style, and offers specific suggestions for how to develop one.

## Features of an Effective Style

There are a few general features of an effective style. Good programming style produces code that is:

- ❖ Explicit  
*Is always clear what is happening and why.*
- ❖ Consistent  
*Uses the same conventions throughout a given system.*
- ❖ Modular  
*Written, whenever possible, as if it will need to be moved into another, unrelated system.*

These general qualities lead to a handful of specific guidelines:

- ❖ Use clear, consistent naming conventions for tables, fields, variables, arrays, forms, methods, semaphores, named selections, and sets.
- ❖ Use a clear, consistent method structure.
- ❖ Include lots of comments.
- ❖ Use white space.
- ❖ Use parameters and few, if any, global variables.

## Naming Conventions

Using consistent names for your variables, arrays, tables, methods, forms and other objects is essential in order to simplify programming and maintenance. It doesn't matter too much what conventions you adopt, as long

as they make sense and you use them consistently. Choose conventions that pack the information you need into your object names. Here are some of the kinds of information you can store in object names:

- ❖ Data type of a variable or array.
- ❖ Membership in a code module.
- ❖ Programmer who created the object.
- ❖ Code to distinguish commands from functions.

## Method Structure

The 4D method editor does not impose any structure on your code. You can make your code easier to read and modify if you adopt a standard structure of your own. Here is a description of the structure we used for all the code in our book *Programming 4th Dimension: The Ultimate Guide*.

### **Method comments**

- Routine name
- Routine name and parameters
- Usage notes

### **Declarations**

- Parameter declarations
- Variable declarations

### **Initialization**

- Assignment of parameters to local variables
- Initialization of non-parameter local variables

### **Main code**

- Code of routine

This structure makes it easy to know where to find variable declarations, to figure out the parameters a routine uses, and to determine the routine's purpose.

## Comments

Writing comments as you work is a helpful way of developing your algorithm. If your code is too complicated to comment, it's too complicated. Almost *no one* goes back and comments later. If you do not comment your code, you make it needlessly difficult for another programmer to work with it. (Six months down the road, chances are *you'll* be that other programmer.)

Comments have no effect on execution speed in a compiled application, so there is no disadvantage to using them liberally. Consider this code:

```

If (Current date>$1)
    $0:=$2*$3
Else
    $0:=$2
End if

```

There is no way to determine what this code does, or what the data types of the parameters are. This is a five line routine and the problem is bad enough to make the code completely unreadable. Imagine how much the problem would be magnified in a 200 line routine! Adding comments is the first step in improving this code:

```

` Update_Accounts (Date;Real;Real) : Real
` Update_Accounts (Due date;Amount due;Interest rate) : New total
`
` This routine is applied to all accounts at the end of each month.
` If the account is past due the customer's interest rate is applied to their
` balance.

If (Current date>$1)
    $0:=$2*$3
Else
    $0:=$2
End if

```

## White Space

Dense, compact code is hard to read. Using white space makes routines much easier to read, and does not make them slower. Putting white space above and below major control structures like **If**, **Case of** and **For** makes the code easier on the eye. Breaking code into functional blocks visually distinguishes the method's functional subunits. The following routine adds customer records to a series of sets based on purchase levels and payments. We show two versions of the routine that differ only in their use of white space.

```

CREATE SET([Customers];"Low purchases")
CREATE SET([Customers];"Medium purchases")
CREATE SET([Customers];"High purchases")
CREATE SET([Customers];"No money received")
For ($i;1;Records in selection([Customers]))
  GOTO SELECTED RECORD([Customers];$i)
  If ([Customers]Total Sales=[Customers]Amount_due)
    ADD TO SET([Customers];"No money received")
  Else
    Case of
      : ([Customers]Total Sales<500)
        ADD TO SET([Customers];"Low purchases")
      : ([Customers]Total Sales<2500)
        ADD TO SET([Customers];"Medium purchases")
    Else
      ADD TO SET([Customers];"High purchases")
    End case
  End if
End for

```

This dense grouping of code is hard to read. It is difficult to determine where the **If**, **For**, and **Case** structures start and stop. Here is the same code with more white space:

```

CREATE SET([Customers];"Low purchases")
CREATE SET([Customers];"Medium purchases")
CREATE SET([Customers];"High purchases")
CREATE SET([Customers];"No money received")

For ($i;1;Records in selection([Customers]))

  GOTO SELECTED RECORD([Customers];$i)

  If ([Customers]Total Sales=[Customers]Amount_due)
    ADD TO SET([Customers];"No money received")
  Else

    Case of
      : ([Customers]Total Sales<500)
        ADD TO SET([Customers];"Low purchases")

      : ([Customers]Total Sales<2500)
        ADD TO SET([Customers];"Medium purchases")

    Else
      ADD TO SET([Customers];"High purchases")
    End case

  End if

End for

```

The white space makes the routine easier to read and the relevant control structures simpler to discern.

## Comment Control Structures

One of the easiest ways to make your code easier to follow is to comment the ends of control structures, like **End if**, **End for** and **End case**. When a routine contains nested control structures it becomes difficult to determine which End belongs with which starting statement. Without care you can end up with code like this:

```
For ($LoopCounter;1;Records in selection([Customers])  
    Imagine lots of code here.  
End if
```

This **For** ends with an **End if**. 4D will not complain about this problem interpreted, but it will never execute the code in your **For** loop. (Obviously *we* never make this kind of mistake. But it, uh... happened to, er... a *friend* of ours. Yeah, a friend.) Commenting the end of a control structure with the statement that starts it is a good way to make your code easier to read and debug. Here is the example routine we've been looking at commented in this way:

```
CREATE SET([Customers];"Low purchases")  
CREATE SET([Customers];"Medium purchases")  
CREATE SET([Customers];"High purchases")  
CREATE SET([Customers];"No money received")  
  
For ($i;1;Records in selection([Customers]))  
  
    GOTO SELECTED RECORD([Customers];$i)  
  
    If ([Customers]Total Sales=[Customers]Amount_due)  
        ADD TO SET([Customers];"No money received")  
    Else ` ([Customers]Total Sales=[Customers]Amount_due)  
  
        Case of  
        : ([Customers]Total Sales<500)  
            ADD TO SET([Customers];"Low purchases")  
  
        : ([Customers]Total Sales<2500)  
            ADD TO SET([Customers];"Medium purchases")  
  
        Else ` Case testing [Customers]Total Sales  
            ADD TO SET([Customers];"High purchases")  
        End case ` Case testing [Customers]Total Sales  
  
    End if ` ([Customers]Total Sales=[Customers]Amount_due)  
  
End for ` ($i;1;Records in selection([Customers]))
```

Even in this short routine the control structure comments help reinforce the structure of the routine. In a longer routine comments like these are essential.

## Labeled Parameters

We recommend that you *always* copy method parameters into local variables with meaningful names. 4D automatically assigns parameters to local variables \$1, \$2, and so on. These names are completely meaningless. It takes virtually no time, and in most cases, very little memory, to copy parameters into local variables:

```
$TablePtr:=$1
```

Apart from legibility, copying parameters to local variables serves two other purposes. First, it makes it *much* easier to change your parameter list. If you use explicit references to \$1, \$2, and so on in a routine and then need to modify the parameter list, modifying your existing routine is often tedious and error prone. If you copy parameters into local variables, all you need to do is change your parameter declarations and your assignments. There is no need to edit the code and take a chance on introducing errors. Let's look at the first example rewritten to use labeled parameters:

```
` Declare parameters and their label variables:
C_REAL($0;$New_total)
C_DATE($1;$Due_date)
C_REAL($2;$Amount_due)
C_REAL($3;$Interest_rate)

` Assign parameters to labels and initialize result variable:
$New_total:=0 ` This variable is returned in $0 as the result of this function.
$Due_date:=$1
$Amount_due:=$2
$Interest_rate:=$3

If (Current date>$Due_date)
    $New_total:=$Amount_due*$Interest_rate
Else
    $New_total:=$Amount_due
End if

$0:=$New_total
```

Even without introductory comments, this routine is highly readable.

## Declare Everything

4D Compiler can type your variables and arrays for you. This is a bug, not a feature, as far as we are concerned. Mistyped variables and misspelled variable names are by far the biggest cause of programming errors. They are also the least interesting errors you can imagine. So, you should explicitly declare *every single* variable, array, and parameter yourself using compiler directives. Don't leave the job of typing variables up to 4D Compiler. This cannot be stressed heavily enough. In fact, we're going to say it again.

4D Compiler can type your variables and arrays for you. This is a bug, not a feature, as far as we are concerned. Mistyped variables and misspelled variable names are by far the biggest cause of programming errors. They are also the least interesting errors you can imagine. So, you should explicitly declare *every single* variable, array, and parameter yourself using compiler directives. Don't leave the job of typing variables up to 4D Compiler.

## Avoid Global Variables

Avoid using global variables (process and interprocess variables) when possible. You can often use parameters instead. Global variables are the cause of many subtle and slippery bugs and make code hard to maintain and more error prone. Global variables are the right way to store “state” information, such as whether the current process is running in a Web connection. Also, given how 4D works, globals are the only way to work with variable data on forms.

On the other hand, globals are the *wrong* way to pass data between methods in the same process. This is what parameters are for. Use parameters instead of global variables. This is arguably the most important principle in this chapter. Parameters make it *much* easier to track the flow of stored values through the system. Using them will save you hours when debugging or updating a system. Also, using parameters is essential for creating modular code.

## Keep Methods Short

Long routines are difficult to read, hard to edit, hard to reuse, and take longer to load than several shorter routines of the same overall size. Whenever someone says, “I hope ACI fixes the 32K method limit” we say, “Yes, hopefully they'll make it 8K”. Short, modular routines are easy to understand, debug, and deploy for new purposes. Sometimes, however, it makes sense to write a long method. This doesn't mean they're easy to work with: long routines are harder to work with even if they are justified. Develop the habit of writing short routines that perform distinct tasks.

## Write Design Documentation

Method comments are essential, though often inadequate for expressing all the useful design information about a system. If you write formal system specifications, then you should also have full design documentation. Documentation also helps to clarify your thinking about your system, which often improves the design. Even modest programmer documentation is enormously helpful. The kinds of things you should document are:

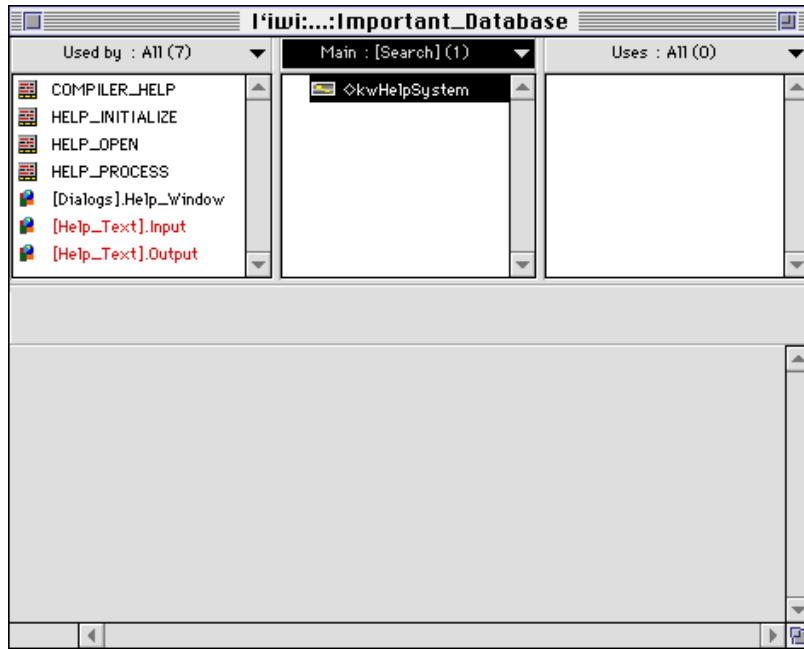
- ❖ *Special codes.* If you use any special codes in your system, *always* document these thoroughly. Few things are harder to figure out than someone else's codes.
- ❖ *Table rules.* Whatever rules you define for your tables need to be explicitly documented. Documenting these rules is actually a simple way of designing them. There is no need for anything complicated, just write out the rules narratively if you like. "Each customer record requires a unique ID, and cannot be purged unless all of its related invoices are also purged."
- ❖ *Structure choices.* Ideally, you should document every table, field, and relation in your system. 4D does not provide facilities for this internally, so you need to do it in external documentation. If documenting everything is too ambitious, at least document the complex and non-standard features.
- ❖ *Your conventions.* Document your naming conventions and other design standards. This makes them easier to understand and easier for a new programmer to begin using.

## Use Insider Keywords

4D Insider is a great cross-referencing tool, but it doesn't allow you to attach attributes to objects. One simple way to add "keywords" to code and forms is by using specially named variables. You can define variables used for no other purpose than cross-referencing with 4D Insider. Here, for example, are a few "keyword" variables used to identify a programmer and the last version of the database the object was modified:

```
<>kwSteveJobs:=True  
<>kwVersion321:=True
```

You can include these variables in any code segment, or place them on any form. When you locate one of these variables in 4D Insider, you'll see every place they're used. This is a simple, flexible, lightweight way of implementing simple version control and object keywords in your 4D systems.



*The <>kwHelpSystem variable and the objects that use it.*

