Title

stata.com

class — Class programming

Description Remarks and examples Also see

Description

Stata's two programming languages, ado and Mata, each support object-oriented programming. This manual entry explains object-oriented programming in ado. Most users interested in object-oriented programming will wish to do so in Mata. See [M-2] **class** to learn about object-oriented programming in Mata.

Ado *classes* are a programming feature of Stata that are especially useful for dealing with graphics and GUI problems, although their use need not be restricted to those topics. Ado class programming is an advanced programming topic and will not be useful to most programmers.

Remarks and examples

stata.com

Remarks are presented under the following headings:

1. Introduction

A *class* is a collection of member variables and member programs. The member programs of a class manipulate or make calculations based on the member variables. Classes are defined in `.class` files. For instance, we might define the class `coordinate` in the file `coordinate.class`:

begin `coordinate.class`

```
version 15.0
class coordinate {
                double  x
                double  y
}
program .set
                args x y
                .x = `x'
                .y = `y'
end
```

end `coordinate.class`

The above file does not create anything. It merely defines the concept of a "coordinate". Now that the file exists, however, you could create a "scalar" variable of type `coordinate` by typing

```
.coord = .coordinate.new
```

`.coord` is called an *instance of* `coordinate`; it contains `.coord.x` (a particular x coordinate) and `.coord.y` (a particular y coordinate). Because we did not specify otherwise, `.coord.x` and `.coord.y` contain missing values, but we could reset `.coord` to contain (1,2) by typing

```
.coord.x = 1
.coord.y = 2
```

Here we can do that more conveniently by typing

```
.coord.set 1 2
```

because `coordinate.class` provides a member program called `.set` that allows us to set the member variables. There is nothing especially useful about `.set`; we wrote it mainly to emphasize that classes could, in fact, contain member programs. Our `coordinate.class` definition would be nearly as good if we deleted the `.set` program. Classes are not required to have member programs, but they may.

If we typed

```
.coord2 = .coordinate.new
.coord2.set 2 4
```

we would now have a second instance of a `coordinate`, this one named `.coord2`, which would contain (2,4).

Now consider another class, `line.class`:

begin `line.class`

```
version 15.0
class line {
                coordinate c0
                coordinate c1
}
program .set
                args x0 y0 x1 y1
                .c0.set `x0' `y0'
                .c1.set `x1' `y1'
end
program .length
                class exit sqrt((`.c0.y'-`.c1.y')^2 + (`.c0.x'-`.c1.x')^2)
end
program .midpoint
                local cx = (`.c0.x' + `.c1.x')/2
                local cy = (`.c0.y' + `.c1.y')/2
                tempname b
                .`b'=.coordinate.new
                .`b'.set `cx' `cy'
                class exit .`b'
end
```

end `line.class`

Like `coordinate.class`, `line.class` has two member variables—named `.c0` and `.c1`—but rather than being numbers, `.c0` and `.c1` are `coordinate`s as we have previously defined the term. Thus the full list of the member variables for `line.class` is

`.c0`	first `coordinate`
`.c0.x`	x value (a `double`)
`.c0.y`	y value (a `double`)
`.c1`	second `coordinate`
`.c1.x`	x value (a `double`)
`.c1.y`	y value (a `double`)

If we typed

```
.li = .line.new
```

we would have a `line` named `.li` in which

`.li.c0`	first `coordinate` of `line .li`
`.li.c0.x`	x value (a `double`)
`.li.c0.y`	y value (a `double`)
`.li.c1`	second `coordinate` of `line .li`
`.li.c1.x`	x value (a `double`)
`.li.c1.y`	y value (a `double`)

What are the values of these variables? Because we did not specify otherwise, `.li.c0` and `.li.c1` will receive default values for their type, `coordinate`. That default is `(.,.)` because we did not specify otherwise when we defined `line`s or `coordinate`s. Therefore, the default values are `(.,.)` and `(.,.)`, and we have a missing line.

As with `coordinate`, we included the member function `.set` to make setting the line easier. We can type

```
.li.set 1 2 2 4
```

and we will have a line going from (1,2) to (2,4).

line.class contains the following member programs:

.set	program to set .c0 and .c1
.c0.set	program to set .c0
.c1.set	program to set .c1
.length	program to return length of line
.midpoint	program to return coordinate of midpoint of line

.set, .length, and .midpoint came from line.class. .c0.set and .c1.set came from coordinate.class.

Member program .length returns the length of the line.

```
.len = .li.length
```

would create .len containing the result of .li.length. The result of running the program .length on the object .li. .length returns a double, and therefore, .len will be a double.

.midpoint returns the midpoint of a line.

```
.mid = .li.midpoint
```

would create .mid containing the result of .li.midpoint, the result of running the program .midpoint on the object .li. .midpoint returns a coordinate, and therefore, .mid will be a coordinate.

2. Definitions

2.1 Class definition

Class *classname* is defined in file *classname*.class. The definition does not create any instances of the class.

The *classname*.class file has three parts:

begin *classname*.class

```
version ...              // Part 1:  version statement
class  classname {       // Part 2:  declaration of member variables
                ...
}
program ...              // Part 3:  code for member programs
                ...
end
program ...
        ...
end
...
```

end *classname*.class

2.2 Class instance

To create a "variable" *name* of type *classname*, you type

```
.name = .classname.new
```

After that, .*name* is variously called an identifier, class variable, class instance, object, object instance, or sometimes just an instance. Call it what you will, the above creates new .*name*—or replaces existing .*name*—to contain the result of an application of the definition of *classname*. And, just as with any variable, you can have many different variables with many different names all the same type.

.*name* is called a first-level or top-level identifier. .*name1*.*name2* is called a second-level identifier, and so on. Assignment into top-level identifiers is allowed if the identifier does not already exist or if the identifier exists and is of type *classname*. If the top-level identifier already exists and is of a different type, you must drop the identifier first and then re-create it; see *11. Object destruction*.

Consider the assignment

```
.name1.name2 = .classname.new
```

The above statement is allowed if .*name1* already exists and if .*name2* is declared, in .*name1*'s class definition, to be of type *classname*. In that case, .*name1*.*name2* previously contained a *classname* instance and now contains a *classname* instance, the difference being that the old contents were discarded and replaced with the new ones. The same rule applies to third-level and higher identifiers.

Classes, and class instances, may also contain member programs. Member programs are identified in the same way as class variables. .*name1*.*name2* might refer to a member variable or to a member program.

2.3 Class context

When a class program executes, it executes in the context of the current instance. For example, consider the instance creation

```
.mycoord = .coordinate.new
```

and recall that `coordinate.class` provides member program `.set`, which reads

```
program .set
                args x y
                .x = `x'
                .y = `y'
end
```

Assume that we type "`.mycoord.set 2 4`". When `.set` executes, it executes in the *context* of `.mycoord`. In the program, the references to `.x` and `.y` are assumed to be to `.mycoord.x` and `.mycoord.y`. If we typed "`.other.set`", the references would be to `.other.x` and `.other.y`.

Look at the statement "`.x = `x'`" in `.set`. Pretend that `` `x' `` is 2 so that, after macro substitution, the statement reads "`.x = 2`". Is this a statement that the first-level identifier `.x` is to be set to 2? No, it is a statement that .*impliedcontext*`.x` is to be set to 2. The same would be true whether `.x` appeared to the right of the equal sign or anywhere else in the program.

The rules for resolving things like `.x` and `.y` are actually more complicated. They are resolved to the implied context if they exist in the implied context, and otherwise they are interpreted to be in the global context. Hence, in the above examples, `.x` and `.y` were interpreted as being references to .*impliedcontext*`.x` and .*impliedcontext*`.y` because `.x` and `.y` existed in .*impliedcontext*. If, however, our program made a reference to `.c`, that would be assumed to be in the global context (that is, to be just `.c`), because there is no `.c` in the implied context. This is discussed at length in *9. Prefix operators*.

If a member program calls a regular program—a regular ado-file—that program will also run in the same class context; for example, if `.set` included the lines

```
move_to_right
.x = r(x)
.y = r(y)
```

and program `move_to_right.ado` had lines in it referring to `.x` and `.y`, they would be interpreted as `.`*impliedcontext*`.x` and `.`*impliedcontext*`.y`.

In all programs—member programs or ado-files—we can explicitly control whether we want identifiers in the implied context or globally with the `.Local` and `.Global` prefixes; see *9. Prefix operators*.

3. Version control

The first thing that should appear in a `.class` file is a `version` statement; see [P] **version**. For example, `coordinate.class` reads

begin `coordinate.class`

```
version 15.0
[ class statement defining member variables omitted ]
program .set
                args x y
                .x = `x'
                .y = `y'
end
```

end `coordinate.class`

The `version 15.0` at the top of the file specifies not only that, when the class definition is read, it be interpreted according to version 15.0 syntax, but also that when each of the member programs runs, it be interpreted according to version 15.0. Thus you do not need to include a `version` statement inside the definition of each member program, although you may if you want that one program to run according to the syntax of a different version of Stata.

Including the `version` statement at the top, however, is of vital importance. Stata is under continual development, and so is the class subsystem. Syntax and features can change. Including the `version` command ensures that your class will continue to work as you intended.

4. Member variables

4.1 Types

The second thing that appears in a `.class` file is the definition of the member variables. We have seen two examples:

begin `coordinate.class`

```
version 15.0
class coordinate {
                double  x
                double  y
}
[ member programs omitted ]
```

end `coordinate.class`

and

begin line.class

```
version 15.0
class line {
                coordinate c0
                coordinate c1
}
[ member programs omitted ]
```

end line.class

In the first example, the member variables are `.x` and `.y`, and in the second, `.c0` and `.c1`. In the first example, the member variables are of type `double`, and in the second, of type `coordinate`, another class.

The member variables may be of *type*

`double`	double-precision scalar numeric value, which includes missing values `.`, `.a`, ..., and `.z`
`string`	scalar string value, with minimum length 0 (`""`) and maximum length the same as for macros, in other words, long
	The class `string` type is different from Stata's `str#` and `strL` types. It can hold much longer string values than can the `str#` type, but not as long of string values as the `strL` type. Additionally, unlike `strL`s, class `string`s cannot contain binary 0.
classname	other classes, excluding the class being defined
`array`	array containing any of the *types*, including other `array`s

A class definition might read

begin todolist.class

```
version 15.0
class todolist {
                double  n               // number of elements in list
                string  name            // who the list is for
                array   list            // the list itself
                actions x               // things that have been done
}
```

end todolist.class

In the above, `actions` is a class, not a primitive type. Somewhere else, we have written `actions.class`, which defines what we mean by `actions`.

`array`s are not typed when they are declared. An `array` is not an array of `double`s or an array of `string`s or an array of `coordinate`s; rather, each array element is separately typed at run time, so an array may turn out to be an array of `double`s or an array of `string`s or an array of `coordinate`s, or it may turn out that its first element is a `double`, its second element is a `string`, its third element is a `coordinate`, its fourth element is something else, and so on.

Similarly, `array`s are not declared to be of a predetermined size. The size is automatically determined at run time according to how the array is used. Also arrays can be sparse. The first element of an array might be a `double`, its fourth element a `coordinate`, and its second and third elements left undefined. There is no inefficiency associated with this. Later, a value might be assigned to the fifth element of the array, thus extending it, or a value might be assigned to the second and third elements, thus filling in the gaps.

4.2 Default initialization

When an instance of a class is created, the member variables are filled in as follows:

`double`	`.` (missing value)
`string`	`""`
classname	as specified by class definition
`array`	empty, an array with no elements yet defined

4.3 Specifying initialization

You may specify in *classname*`.class` the initial values for member variables. To do this, you type an equal sign after the identifier, and then you type the initial value. For example,

begin `todolist.class`

```
version 15.0
class todolist {
                double  n    = 0
                string  name = "nobody"
                array   list = {"show second syntax", "mark as done"}
                actions x    = .actions.new arguments
}
```

end `todolist.class`

The initialization rules are as follows:

`double` *membervarname* `=` ...

After the equal sign, you may type any number or expression. To initialize the member variable with a missing value (`.`, `.a`, `.b`, ..., `.z`), you must enclose the missing value in parentheses. Examples include

```
double n = 0
double a = (.)
double b = (.b)
double z = (2+3)/sqrt(5)
```

Alternatively, after the equal sign, you may specify the identifier of a member variable to be copied or program to be run as long as the member variable is a `double` or the program returns a `double`. If a member program is specified that requires arguments, they must be specified following the identifier. Examples include

```
double n = .clearcount
double a = .gammavalue 4 5 2
double b = .color.cvalue, color(green)
```

The identifiers are interpreted in terms of the global context, not the class context being defined. Thus `.clearcount`, `.gammavalue`, and `.color.cvalue` must exist in the global context.

`string` *membervarname* `=` ...

After the equal sign, you type the initial value for the member variable enclosed in quotes, which may be either simple (`"` and `"`) or compound (`‘"` and `"’`). Examples include

```
string name = "nobody"
string s = ‘"quotes "inside" strings"’
string a = ""
```

You may also specify a string expression, but you must enclose it in parentheses. For example,

```
string name = ("no" + "body")
string b    = (char(11))
```

Or you may specify the identifier of a member variable to be copied or a member program to be run, as long as the member variable is a `string` or the program returns a `string`. If a member program is specified that requires arguments, they must be specified following the identifier. Examples include

```
string n = .defaultname
string a = .recapitalize "john smith"
string b = .names.defaults, category(null)
```

The identifiers are interpreted in terms of the global context, not the class context being defined. Thus `.defaultname`, `.recapitalize`, and `.names.defaults` must exist in the global context.

`array` *membervarname* `= {...}`

After the equal sign, you type the set of elements in braces (`{` and `}`), with each element separated from the next by a comma.

If an element is enclosed in quotes (simple or compound), the corresponding array element is defined to be `string` with the contents specified.

If an element is a literal number excluding `.`, `.a`, ..., and `.z`, the corresponding array element is defined to be `double` and filled in with the number specified.

If an element is enclosed in parentheses, what appears inside the parentheses is evaluated as an expression. If the expression evaluates to a string, the corresponding array element is defined to be `string` and the result is filled in. If the expression evaluates to a number, the corresponding array element is defined to be `double` and the result is filled in. Missing values may be assigned to array elements by being enclosed in parentheses.

An element that begins with a period is interpreted as an object identifier in the global context. That object may be a member variable or a member program. The corresponding array element is defined to be of the same type as the specified member variable or of the same type as the member program returns. If a member program is specified that requires arguments, the arguments must be specified following the identifier, but the entire syntactical elements must be enclosed in square brackets (`[` and `]`).

If the element is nothing, the corresponding array element is left undefined.

Examples include

```
array mixed = {1, 2, "three", 4}
array els   = {.box.new, , .table.new}
array rad   = {[.box.new 2 3], , .table.new}
```

Note the double commas in the last two initializations. The second element is left undefined. Some programmers would code

```
array els   = {.box.new, /*nothing*/, .table.new}
array rad   = {[.box.new 2 3], /*nothing*/, .table.new}
```

to emphasize the null initialization.

classname membervarname = . . .

After the equal sign, you specify the identifier of a member variable to be copied or a member program to be run, as long as the member variable is of type *classname* or the member program returns something of type *classname*. If a member program is specified that requires arguments, they must be specified following the identifier. In either case, the identifier will be interpreted in the global context. Examples include

```
box mybox1 = .box.new
box mybox2 = .box.new 2 4 7 8, tilted
```

All the types can be initialized by copying other member variables or by running other member programs. These other member variables and member programs must be defined in the global context and not the class context. In such cases, each initialization value or program is, in fact, copied or run only once—at the time the class definition is read—and the values are recorded for future use. This makes initialization fast. This also means, however, that

- If, in a class definition called, say, `border.class`, you defined a member variable that was initialized by `.box.new`, and if `.box.new` counted how many times it is run, then even if you were to create 1,000 instances of `border`, you would discover that `.box.new` was run only once. If `.box.new` changed what it returned over time (perhaps because of a change in some state of the system being implemented), the initial values would not change when a new border object was created.
- If, in `border.class`, you were to define a member variable that is initialized as `.system.curvals.no_of_widgets`, which we will assume is another member variable, then even if `.system.curvals.no_of_widgets` were changed, the new instances of `border.class` would always have the same value—the value of `.system.curvals.no_of_widgets` current at the time `border.class` was read.

In both of the above examples, the method just described—the prerecorded assignment method of specifying initial values—would be inadequate. The method just described is suitable for specifying constant initial values only.

4.4 Specifying initialization 2, .new

Another way to specify how member variables are to be initialized is to define a `.new` program within the class.

To create a new instance of a class, you type

`.` *name* `=.` *classname*`.new`

`.new` is, in fact, a member program of *classname*; it is just one that is built in, and you do not have to define it to use it. The built-in `.new` allocates the memory for the instance and fills in the default or specified initial values for the member variables. If you define a `.new`, your `.new` will be run after the built-in `.new` finishes its work.

For example, our example `coordinate.class` could be improved by adding a `.new` member program:

begin coordinate.class

```
version 15.0
class coordinate {
                double  x
                double  y
}
program .new
                if "‘0’" != "" {
                        .set ‘0’
                }
end
program .set
                args x y
                .x = ‘x’
                .y = ‘y’
end
```

end coordinate.class

With this addition, we could type

```
.coord = .coordinate.new
.coord.set 2 4
```

or we could type

```
.coord = .coordinate.new 2 4
```

We have arranged `.new` to take arguments—optional ones here—that specify where the new point is to be located. We wrote the code so that `.new` calls `.set`, although we could just as well have written the code so that the lines in `.set` appeared in `.new` and then deleted the `.set` program. In fact, the two-part construction can be desirable because then we have a function that will reset the contents of an existing class as well.

In any case, by defining your own `.new`, you can arrange for any sort of complicated initialization of the class, and that initialization can be a function of arguments specified if that is necessary.

The `.new` program need not return anything; see *6. Member programs' return values*.

`.new` programs are not restricted just to filling in initial values. They are programs that you can code however you wish. `.new` is run every time a new instance of a class is created with one exception: when an instance is created as a member of another instance (in which case, the results are prerecorded).

4.5 Another way of declaring

In addition to the syntax

type *name* [`=` *initialization*]

where *type* is one of `double`, `string`, *classname*, or `array`, there is an alternative syntax that reads

name `=` *initialization*

That is, you may omit specifying *type* when you specify how the member variable is to be initialized because, then, the type of the member variable can be inferred from the initialization.

4.6 Scope

In the examples we have seen so far, the member variables are unique to the instance. For example, if we have

```
.coord1 = .coordinate.new
.coord2 = .coordinate.new
```

then the member variables of `.coord1` have nothing to do with the member variables of `.coord2`. If we were to change `.coord1.x`, then `.coord2.x` would remain unchanged.

Classes can also have variables that are shared across all instances of the class. Consider

begin `coordinate2.class`

```
version 15.0
class coordinate2 {
                classwide:
                        double x_origin = 0
                        double y_origin = 0
                instancespecific:
                        double x = 0
                        double y = 0
}
```

end `coordinate2.class`

In this class definition, `.x` and `.y` are as they were in `coordinate.class`—they are unique to the instance. `.x_origin` and `.y_origin`, however, are shared across all instances of the class. That is, if we were to type

```
.ac = .coordinate2.new
.bc = .coordinate2.new
```

there would be only one copy of `.x_origin` and of `.y_origin`. If we changed `.x_origin` in `.ac`,

```
.ac.x_origin = 2
```

we would find that `.bc.x_origin` had similarly been changed. That is because `.ac.x_origin` and `.bc.x_origin` are, in fact, the same variable.

The effects of initialization are a little different for classwide variables. In `coordinate2.class`, we specified that `.origin_x` and `.origin_y` both be initialized as 0, and so they were when we typed "`.ac = .coordinate2.new`", creating the first instance of the class. After that, however, `.origin_x` and `.origin_y` will never be reinitialized because they need not be re-created, being shared. (That is not exactly accurate because, once the last instance of a `coordinate2` has been destroyed, the variables will need to be reinitialized the next time a new first instance of `coordinate2` is created.)

Classwide variables, just as with instance-specific variables, can be of any type. We can define

begin `supercoordinate.class`

```
version 15.0
class supercoordinate {
                classwide:
                        coordinate  origin
                instancespecific:
                        coordinate  pt
}
```

end `supercoordinate.class`

The qualifiers `classwide:` and `instancespecific:` are used to designate the scope of the member variables that follow. When neither is specified, `instancespecific:` is assumed.

4.7 Adding dynamically

Once an instance of a class exists, you can add new (instance-specific) member variables to it. The syntax for doing this is

name `.Declare` *attribute_declaration*

where *name* is the identifier of an instance and *attribute_declaration* is any valid attribute declaration such as

```
double    varname
string    varname
array     varname
classname varname
```

and, on top of that, we can include `=` and initializer information as defined in *4.3 Specifying initialization* above.

For example, we might start with

```
.coord = .coordinate.new
```

and discover that there is some extra information that we would like to carry around with the particular instance `.coord`. Here we want to carry around some color information that we will use later, and we have at our fingertips `color.class`, which defines what we mean by `color`. We can type

```
.coord.Declare color mycolor
```

or even

```
.coord.Declare color mycolor = .color.new, color(default)
```

to cause the new class instance to be initialized the way we want. After that command, `.coord` now contains `.coord.color` and whatever third-level or higher identifiers `color` provides. We can still invoke the member programs of `coordinate` on `.coord`, and to them, `.coord` will look just like a `coordinate` because they will know nothing about the extra information (although if they were to make a copy of `.coord`, then the copy would include the extra information). We can use the extra information in our main program and even in subroutines that we write.

❑ Technical note

Just as with the declaration of member variables inside the `class {}` statement, you can omit specifying the *type* when you specify the initialization. In the above, the following would also be allowed:

```
.coord.Declare mycolor = .color.new, color(default)
```

❑

4.8 Advanced initialization, .oncopy

Advanced initialization is an advanced concept, and we need concern ourselves with it only when our class is storing references to items outside the class system. In such cases, the class system knows nothing about these items other than their names. We must manage the contents of these items.

Assume that our coordinates class was storing not scalar coordinates but rather the names of Stata variables that contained coordinates. When we create a copy of such a class,

```
.coord = .coordinate.new 2 4
.coordcopy = .coord
```

`.coordcopy` will contain copies of the names of the variables holding the coordinates, but the variables themselves will not be copied. To be consistent with how all other objects are treated, we may prefer that the contents of the variables be copied to new variables.

As with `.new` we can define an `.oncopy` member program that will be run after the default copy operation has been completed. We will probably need to refer to the source object of the copy with the built-in `.oncopy_src`, which returns a key to the source object.

Let's write the beginnings of a coordinate class that uses Stata variables to store vectors of coordinates.

begin `varcoordinate.class`

```
version 15.0
class varcoordinate {
        classwide:
                n = 0
        instancespecific:
                string x
                string y
}
program .new
                .nextnames
                if "`0'" != "" {
                        .set `0'
                }
end
program .set
                args x y
                replace `.x' = `x'
                replace `.y' = `y'
end
program .nextnames
                .n = `.n' + 1
                .x = "__varcorrd_vname_`.n'"
                .n = `.n' + 1
                .y = "__varcorrd_vname_`.n'"
                generate `.x' = .
                generate `.y' = .
end
program .oncopy
        .nextnames
        .set `.`.oncopy_src'.x' `.`.oncopy_src'.y'
end
```

end `varcoordinate.class`

This class is more complicated than what we have seen before. We are going to use our own unique variable names to store the x- and y-coordinate variables. To ensure that we do not try to reuse the same name, we number these variables by using the classwide counting variable `.n`. Every

time a new instance is created, unique x- and y-coordinate variables are created and filled in with missing. This work is done by `.nextnames`.

The `.set` looks similar to the one from `.varcoordinates` except that now we are holding variable names in ‘.x’ and ‘.y’, and we use `replace` to store the values from the specified variables into our coordinate variables.

The `.oncopy` member function creates unique names to hold the variables, using `.nextnames`, and then copies the contents of the coordinate variables from the source object, using `.set`.

Now, when we type

```
.coordcopy = .coord
```

the x- and y-coordinate variables in `.coordcopy` will be different variables from those in `.coord` with copies of their values.

The `varcoordinate` class does not yet do anything interesting, and other than the example in the following section, we will not develop it further.

4.9 Advanced cleanup, destructors

We rarely need to concern ourselves with objects being removed when they are deleted or replaced.

When we type

```
.a = .classname.new
.b = .classname.new
.a = .b
```

the last command causes the original object, `.a`, to be destroyed and replaces it with `.b`. The class system handles this task, which is usually all we want done. An exception is objects that are holding onto items outside the class system, such as the coordinate variables in our `destructor` class.

When we need to perform actions before the system deletes an object, we write a `.destructor` member program in the class file. The `.destructor` for our `varcoordinate` class is particularly simple; it drops the coordinate variables.

begin `varcoordinate.class` -- `destructor`

```
program .destructor
                capture drop ‘.x’
                capture drop ‘.y’
end
```

end `varcoordinate.class` -- `destructor`

5. Inheritance

One class definition can inherit from other class definitions. This is done by including the `inherit(`*classnamelist*`)` option:

begin *newclassname*`.class`

```
version 15.0
class newclassname {
                ...
}, inherit(classnamelist)
program ...
                ...
end
...
```

end *newclassname*`.class`

newclassname inherits the member variables and member programs from *classnamelist*. In general, *classnamelist* contains one class name. When *classnamelist* contains more than one class name, that is called *multiple inheritance*.

To be precise, *newclassname* inherits all the member variables from the classes specified except those that are explicitly defined in *newclassname*, in which case the definition provided in *newclassname*`.class` takes precedence. It is considered bad style to name member variables that conflict.

For multiple inheritance, it is possible that, although a member variable is not defined in *newclassname*, it is defined in more than one of the "parents" (*classnamelist*). Then it will be the definition in the rightmost parent that is operative. This too is to be avoided, because it almost always results in programs' breaking.

newclassname also inherits all the member programs from the classes specified. Here name conflicts are not considered bad style, and in fact, redefinition of member programs is one of the primary reasons to use inheritance.

newclassname inherits all the programs from *classnamelist*—even those with names in common—and a way is provided to specify which of the programs you wish to run. For single inheritance, if member program `.zifl` is defined in both classes, then `.zifl` is taken as the instruction to run `.zifl` as defined in *newclassname*, and `.Super.zifl` is taken as the instruction to run `.zifl` as defined in the parent.

For multiple inheritance, `.zifl` is taken as the instruction to run `.zifl` as defined in *newclassname*, and `.Super(`*classname*`).zifl` is taken as the instruction to run `.zifl` as defined in the parent *classname*.

A good reason to use inheritance is to "steal" a class and to modify it to suit your purposes. Pretend that you have `alreadyexists.class` and from that you want to make `alternative.class`, something that is much like `alreadyexists.class`—so much like it that it could be used wherever `alreadyexists.class` is used—but it does one thing a little differently. Perhaps you are writing a graphics system, and `alreadyexists.class` defines everything about the little circles used to mark points on a graph, and now you want to create `alternate.class` that does the same, but this time for solid circles. Hence, there is only one member program of `alreadyexists.class` that you want to change: how to draw the symbol.

In any case, we will assume that `alternative.class` is to be identical to `alreadyexists.class`, except that it has changed or improved member function `.zifl`. In such a circumstance, it would not be uncommon to create

begin `alternative.class`

```
version 15.0
class alternative {
}, inherit(alreadyexists)
program .zifl
                ...
end
```

end `alternative.class`

Moreover, in writing `.zifl`, you might well call `.Super.zifl` so that the old `.zifl` performed its tasks, and all you had to do was code what was extra (filling in the circles, say). In the example above, we added no member variables to the class.

Perhaps the new `.zifl` needs a new member variable—a `double`—and let's call it `.sizeofresult`. Then we might code

begin `alternative.class`

```
version 15.0
class alternative {
                double   sizeofresult
}, inherit(alreadyexists)
program .zifl
                ...
end
```

end `alternative.class`

Now let's consider initialization of the new variable, `.sizeofresult`. Perhaps having it initialized as missing is adequate. Then our code above is adequate. Suppose that we want to initialize it to 5. Then we could include an initializer statement. Perhaps we need something more complicated that must be handled in a `.new`. In this final case, we must call the inherited classes' `.new` programs by using the `.Super` modifier:

begin `alternative.class`

```
version 15.0
class alternative {
                double   sizeofresult
}, inherit(alreadyexists)
program .new
                ...
                .Super.new
                ...
end
program .zifl
                ...
end
```

end `alternative.class`

6. Member programs' return values

Member programs may optionally return "values", and those can be `double`s, `string`s, `array`s, or class instances. These return values can be used in assignment, and thus you can code

```
.len    = .li.length
.coord3 = .li.midpoint
```

Just because a member program returns something, it does not mean it has to be consumed. The programs `.li.length` and `.li.midpoint` can still be executed directly,

```
.li.length
.li.midpoint
```

and then the return value is ignored. (`.midpoint` and `.length` are member programs that we included in `line.class`. `.length` returns a `double`, and `.midpoint` returns a `coordinate`.)

You cause member programs to return values by using the `class exit` command; see [P] **class exit**.

Do not confuse returned values with return codes, which all Stata programs set, even member programs. Member programs exit when they execute.

Condition	Returned value	Return code
`class exit` with arguments	as specified	0
`class exit` without arguments	nothing	0
`exit` without arguments	nothing	0
`exit` with arguments	nothing	as specified
`error`	nothing	as specified
command having error	nothing	as appropriate

Any of the preceding are valid ways of exiting a member program, although the last is perhaps best avoided. `class exit` without arguments has the same effect as `exit` without arguments; it does not matter which you code.

If a member program returns nothing, the result is as if it returned `string` containing `""` (nothing).

Member programs may also return values in `r()`, `e()`, and `s()`, just like regular programs. Using `class exit` to return a class result does not prevent member programs from also being r-class, e-class, or s-class.

7. Assignment

Consider `.coord` defined

```
.coord = .coordinate.new
```

That is an example of assignment. A new instance of class `coordinate` is created and assigned to `.coord`. In the same way,

```
.coord2 = .coord
```

is another example of assignment. A copy of `.coord` is made and assigned to `.coord2`.

Assignment is not allowed just with top-level names. The following are also valid examples of assignment:

```
.coord.x = 2
.li.c0 = .coord
.li.c0.x = 2+2
.todo.name = "Jane Smith"
.todo.n = 2
.todo.list[1] = "Turn in report"
.todo.list[2] = .li.c0
```

In each case, what appears on the right is evaluated, and a copy is put into the specified place. Assignment based on the returned value of a program is also allowed, so the following are also valid:

```
.coord.x = .li.length
.li.c0 = .li.midpoint
```

`.length` and `.midpoint` are member programs of `line.class`, and `.li` is an instance of `line`. In the first example, `.li.length` returns a `double`, and that `double` is assigned to `.coord.x`. In the second example, `.li.midpoint` returns a `coordinate`, and that `coordinate` is assigned to `li.c0`.

Also allowed would be

```
.todo.list[3] = .color.cvalue, color(green)
.todo.list = {"Turn in report", .li.c0, [.color.cvalue, color(green)]}
```

In both examples, the result of running `.color.cvalue, color(green)` is assigned to the third array element of `.todo.list`.

7.1 Type matching

All the examples above are valid because either a new identifier is being created or the identifier previously existed and was of the same type as the identifier being assigned.

For example, the following would be invalid:

```
.newthing = 2              // valid so far . . .
.newthing = "new"          // . . . invalid
```

The first line is valid because `.newthing` did not previously exist. After the first assignment, however, `.newthing` did exist and was of type `double`. That caused the second assignment to be invalid, the error being "type mismatch"; r(109).

The following are also invalid:

```
.coord.x = .li.midpoint
.li.c0 = .li.length
```

They are invalid because `.li.midpoint` returns a `coordinate`, and `.coord.x` is a `double`, and because `.li.length` returns a `double`, and `.li.c0` is a `coordinate`.

7.2 Arrays and array elements

The statements

```
.todo.list[1] = "Turn in report"
.todo.list[2] = .li.c0
.todo.list[3] = .color.cvalue, color(green)
```

and

```
.todo.list = {"Turn in report", .li.c0, [.color.cvalue, color(green)]}
```

do not have the same effect. The first set of statements reassigns elements 1, 2, and 3 and leaves any other defined elements unchanged. The second statement replaces the entire array with an array that has only elements 1, 2, and 3 defined.

After an element has been assigned, it may be unassigned (cleared) using `.Arrdropel`. For example, to unassign `.todo.list[1]`, you would type

```
.todo.list[1].Arrdropel
```

Clearing an element does not affect the other elements of the array. In the above example, `.todo.list[2]` and `.todo.list[3]` continue to exist.

New and existing elements may be assigned and reassigned freely, except that if an array element already exists, it may be reassigned only to something of the same type.

```
.todo.list[2] = .coordinate[2]
```

would be allowed, but

```
.todo.list[2] = "Clear the coordinate"
```

would not be allowed because `.todo.list[2]` is a `coordinate` and `"Clear the coordinate"` is a `string`. If you wish to reassign an array element to a different type, you first drop the existing array element and then assign it.

```
.todo.list[2].Arrdropel
.todo.list[2] = "Clear the coordinate"
```

7.3 lvalues and rvalues

Notwithstanding everything that has been said, the syntax for assignment is

lvalue `=` *rvalue*

lvalue stands for what may appear to the left of the equal sign, and *rvalue* stands for what may appear to the right.

The syntax for specifying an *lvalue* is

`.`*id*[`.`*id*[...]]

where *id* is either a *name* or *name*`[`*exp*`]`, the latter being the syntax for specifying an array element, and *exp* must evaluate to a number; if *exp* evaluates to a noninteger number, it is truncated.

Also an *lvalue* must be assignable, meaning that *lvalue* cannot refer to a member program; that is, an *id* element of *lvalue* cannot be a program name. (In an *rvalue*, if a program name is specified, it must be in the last *id*.)

The syntax for specifying an *rvalue* is any of the following:

`"`[*string*]`"`
`‘"`[*string*]`"’`
`#`
exp
`(`*exp*`)`
`.`*id*[`.`*id*[...]] [*program_arguments*]
`{}`
`{`*el*[`,`*el*[`,`...]]`}`

The last two syntaxes concern assignment to arrays, and *el* may be any of the following:

nothing
`"`[*string*]`"`
`‘"`[*string*]`"’`
`#`
`(`*exp*`)`
`.`*id*[`.`*id*[...]]
`[.`*id*[`.`*id*[...]] [*program_arguments*]`]`

Let's consider each of the syntaxes for an *rvalue* in turn:

`"`[*string*]`"` and `‘"`[*string*]`"’`
If the *rvalue* begins with a double quote (simple or compound), a `string` containing *string* will be returned. *string* may be long—up to the length of a macro.

#

If the *rvalue* is a number excluding missing values `.`, `.a`, ..., and `.z`, a `double` equal to the number specified will be returned.

exp and `(`*exp*`)`

If the *rvalue* is an expression, the expression will be evaluated and the result returned. A `double` will be returned if the expression returns a numeric result and a `string` will be returned if expression returns a string. Expressions returning matrices are not allowed.

The expression need not be enclosed in parentheses if the expression does not begin with simple or compound double quotes and does not begin with a period followed by nothing or a letter. In the cases just mentioned, the expression must be enclosed in parentheses. All expressions may be enclosed in parentheses.

An implication of the above is that missing value literals must be enclosed in parentheses: *lvalue* `= (.)`.

.id[*.id*[...]] [*program_arguments*]

If the *rvalue* begins with a period, it is interpreted as an object reference. The object is evaluated and returned. *.id*[*.id*[...]] may refer to a member variable or a member program.

If *.id*[*.id*[...]] refers to a member variable, the value of the variable will be returned.

If *.id*[*.id*[...]] refers to a member program, the program will be executed and the result returned. If the member program returns nothing, a `string` containing `""` (nothing) will be returned.

If *.id*[*.id*[...]] refers to a member program, arguments may be specified following the program name.

`{}` and `{`*el*[`,`*el*[`,`...]]`}`

If the *rvalue* begins with an open brace, an `array` will be returned.

If the *rvalue* is `{}`, an empty array will be returned.

If the *rvalue* is `{`*el*[`,`*el*[`,`...]]`}`, an array containing the specified elements will be returned.

If an *el* is nothing, the corresponding array element will be left undefined.

If an *el* is `"`[*string*]`"` or `‘"`[*string*]`"’`, the corresponding array element will be defined as a `string` containing *string*.

If an *el* is *#* excluding missing values `.`, `.a`, ..., `.z`, the corresponding array element will be defined as a `double` containing the number specified.

If an *el* is `(`*exp*`)`, the expression is evaluated, and the corresponding array element will be defined as a `double` if the expression returns a numeric result or as a `string` if the expression returns a string. Expressions returning matrices are not allowed.

If an *el* is *.id*[*.id*[...]] or `[`*.id*[*.id*[...]] [*program_arguments*]`]`, the object is evaluated, and the corresponding array element will be defined according to what was returned. If the object is a member program and arguments need to be specified, the *el* must be enclosed in square brackets.

Recursive array definitions are not allowed.

Finally, in *4.3 Specifying initialization*—where we discussed member variable initialization—what actually appears to the right of the equal sign is an *rvalue*, and everything just said applies. The previous discussion was incomplete.

7.4 Assignment of reference

Consider two different identifiers, *.a.b.c* and *.d.e*, that are of the same type. For example, perhaps both are `doubles` or both are `coordinates`. When you type

```
.a.b.c = .d.e
```

the result is to copy the values of *.d.e* into *.a.b.c*. If you type

```
.a.b.c.ref = .d.e.ref
```

the result is to make *.a.b.c* and *.d.e* be the same object. That is, if you were later to change some element of *.d.e*, the corresponding element of *.a.b.c* would change, and vice versa.

To understand this, think of member values as each being written on an index card. Each instance of a class has its own collection of cards (assuming no classwide variables). When you type

```
.a.b.c.ref = .d.e.ref
```

the card for *.a.b.c* is removed and a note is substituted that says to use the card for *.d.e*. Thus both *.a.b.c* and *.d.e* become literally the same object.

More than one object can share references. If we were now to code

```
.i.ref = .a.b.c.ref
```

or

```
.i.ref = .d.e.ref
```

the result would be the same: *.i* would also share the already-shared object.

We now have *.a.b.c*, *.d.e*, and *.i* all being the same object. Say that we want to make *.d.e* into its own unique object again. We type

```
.d.e.ref = anything evaluating to the right type not ending in .ref
```

We could, for instance, type any of the following:

```
.d.e.ref = .classname.new
.d.e.ref = .j.k
.d.e.ref = .d.e
```

All the above will make *.d.e* unique because what is returned on the right is a copy. The last of the three examples is intriguing because it results in *.d.e* not changing its values but becoming once again unique.

8. Built-ins

`.new` and `.ref` are examples of built-in member programs that are included in every class. There are other built-ins as well.

Built-ins may be used on any object except programs and other built-ins. Let *.B* refer to a built-in. Then

- If *.a.b.myprog* refers to a program, *.a.b.myprog.B* is an error (and, in fact, *.a.b.myprog.anything* is also an error).
- *.a.b.B.anything* is an error.

Built-ins come in two forms: built-in functions and built-in modifiers. Built-in functions return information about the class or class instance on which they operate but do not modify the class or class instance. Built-in modifiers might return something—in general they do not—but they modify (change) the class or class instance.

Except for `.new` (and that was covered in *4.4 Specifying initialization 2, .new*), built-ins may not be redefined.

8.1 Built-in functions

In the documentation below, *object* refers to the context of the built-in function. For example, if *.a.b.F* is how the built-in function *.F* was invoked, then *.a.b* is the object on which it operates.

The built-in functions are

`.new`
: returns a new instance of *object*. `.new` may be used whether the *object* is a class name or an instance, although it is most usually used with a class name. For example, if `coordinate` is a class, `.coordinate.new` returns a new instance of `coordinate`.

  If `.new` is used with an instance, a new instance of the class of the object is returned; the current instance is not modified. For example, if *.a.b* is an instance of `coordinate`, then *.a.b*`.new` does exactly what `.coordinate.new` would do; *.a.b* is not modified in any way.

  If you define your own `.new` program, it is run after the built-in `.new` is run.

`.copy`
: returns a new instance—a copy—of *object*, which must be an instance. `.copy` returns a new object that is a copy of the original.

`.ref`
: returns a reference to the object. See *7.4 Assignment of reference*.

`.objtype`
: returns a `string` indicating the type of *object*. Returned is one of `"double"`, `"string"`, `"array"`, or "*classname*".

`.isa`
: returns a `string` indicating the category of *object*. Returned is one of `"double"`, `"string"`, `"array"`, `"class"`, or `"classtype"`. `"classtype"` is returned when *object* is a class definition; `"class"` is returned when the object is an instance of a class *(sic)*.

`.classname`
: returns a `string` indicating the name of the class. Returned is "*classname*" or, if *object* is of type `double`, `string`, or `array`, returned is `""`.

`.isofclass` *classname*
: returns a `double`. Returns 1 if *object* is of class type *classname* and 0 otherwise. To be of a class type, *object* must be an instance of *classname*, inherited from the class *classname*, or inherited from a class that inherits anywhere along its inheritance path from *classname*.

`.objkey`
: returns a `string` that can be used to reference an object outside the implied context. See *12.1 Keys*.

`.uname`
: returns a `string` that can be used as a *name* throughout Stata that corresponds to the object. See *12.2 Unames*.

`.ref_n`
: returns a `double`. Returned is the total number of identifiers sharing *object*. Returned is 1 if the object is unshared. See *7.4 Assignment of reference*.

`.arrnels`
: returns a `double`. `.arrnels` is for use with `arrays`; it returns the largest index of the array that has been assigned data. If *object* is not an array, it returns an error.

`.arrindexof "`*string*`"`
: returns a `double`. `.arrindexof` is for use with `arrays`; it searches the array for the first element equal to *string* and returns the index of that element. If *string* is not found, `.arrindexof` returns 0. If *object* is not an array, it returns an error.

`.classmv`
: returns an `array` containing the `.refs` of each classwide member variable in *object*. See *12.3 Arrays of member variables*.

`.instancemv`
: returns an `array` containing the `.refs` of each instance-specific member variable in *object*. See *12.3 Arrays of member variables*.

`.dynamicmv`
: returns an `array` containing the `.refs` of each dynamically allocated member variable in *object*. See *12.3 Arrays of member variables*.

`.superclass`
: returns an `array` containing the `.refs` of each of the classes from which the specified object inherited. See *12.3 Arrays of member variables*.

8.2 Built-in modifiers

Modifiers are built-ins that change the object to which they are applied. All built-in modifiers have names beginning with a capital letter. The built-in modifiers are

`.Declare` *declarator*
: returns nothing. `.Declare` may be used only when *object* is a class instance. `.Declare` adds the specified new member variable to the class instance. See *4.7 Adding dynamically*.

`.Arrdropel #`
: returns nothing. `.Arrdropel` may be used only with `array` elements. `.Arrdropel` drops the specified array element, making it as if it was never defined. `.arrnels` is, of course, updated. See *7.2 Arrays and array elements*.

`.Arrdropall`
: returns nothing. `.Arrdropall` may be used only with `arrays`. `.Arrdropall` drops all elements of an array. `.Arrdropall` is the same as `.`*arrayname* `= {}`. If *object* is not an array, `.Arrdropall` returns an error.

`.Arrpop`
: returns nothing. `.Arrpop` may be used only with `arrays`. `.Arrpop` finds the top element of an array (largest index) and removes it from the array. To access the top element before popping, use `.`*arrayname*`['.`*arrayname*`.arrnels']`. If *object* is not an array, `.Arrpop` returns an error.

`.Arrpush "`*string*`"`
: returns nothing. `.Arrpush` may be used only with `arrays`. `.Arrpush` pushes *string* onto the end of the array, where end is defined as `.arrnels`+1. If *object* is not an array, `.Arrpush` returns an error.

9. Prefix operators

There are three prefix operators:

```
.Global
.Local
.Super
```

Prefix operators determine how object names such as *.a*, *.a.b*, *.a.b.c*, ... are resolved.

Consider a program invoked by typing `.alpha.myprog`. In program `.myprog`, any lines such as

```
.a = .b
```

are interpreted according to the implied context, if that is possible. `.a` is interpreted to mean `.alpha.a` if `.a` exists in `.alpha`; otherwise, it is taken to mean `.a` in the global context, meaning that it is taken to mean just `.a`. Similarly, `.b` is taken to mean `.alpha.b` if `.b` exists in `.alpha`; otherwise, it is taken to mean `.b`.

What if `.myprog` wants `.a` to be interpreted in the global context even if `.a` exists in `.alpha`? Then the code would read

```
.Global.a = .b
```

If instead `.myprog` wanted `.b` to be interpreted in the global context (and `.a` to be interpreted in the implied context), the code would read

```
.a = .Global.b
```

Obviously, if the program wanted both to be interpreted in the global context, the code would read

```
.Global.a = .Global.b
```

`.Local` is the reverse of `.Global`: it ensures that the object reference is interpreted in the implied context. `.Local` is rarely specified because the local context is searched first, but if there is a circumstance where you wish to be certain that the object is not found in the global context, you may specify its reference preceded by `.Local`. Understand, however, that if the object is not found, an error will result, so you would need to precede commands containing such references with `capture`; see [P] **capture**.

In fact, if it is used at all, `.Local` is nearly always used in a macro-substitution context—something discussed in the next section—where errors are suppressed and where nothing is substituted when errors occur. Thus in advanced code, if you were trying to determine whether member variable `.addedvar` exists in the local context, you could code

```
if "`Local.addedvar.objtype'" == "" {
                /* it does not exist */
}
else {
                /* it does */
}
```

The `.Super` prefix is used only in front of program names and concerns inheritance when one program occults another. This was discussed in *5. Inheritance*.

10. Using object values

We have discussed definition and assignment of objects, but we have not yet discussed how you might use class objects in a program. How do you refer to their values in a program? How do you find out what a value is, skip some code if the value is one thing, and loop if it is another?

The most common way to refer to objects (and the returned results of member programs) is through macro substitution; for example,

```
local x = `.li.c0.x'
local clr "`.color.cvalue, color(green)'"
scalar len = `.coord.length'
forvalues i=1(1)`.todo.n' {
                Mysub "`todo.list[`i']'"
}
```

When a class object is quoted, its printable form is substituted. This is defined as

Object type	Printable form
`string`	contents of the string
`double`	number printed using `%18.0g`, spaces stripped
`array`	nothing
classname	nothing or, if member program `.macroexpand` is defined, then `string` or `double` returned

Any object may be quoted, including programs. If the program takes arguments, they are included inside the quotes:

```
scalar len = `.coord.length'
local clr "`.color.cvalue, color(green)'"
```

If the quoted reference results in an error, the error message is suppressed, and nothing is substituted.

Similarly, if a class instance is quoted—or a program returning a class instance is quoted—nothing is substituted. That is, nothing is substituted, assuming that the member program `.macroexpand` has not been defined for the class, as is usually the case. If `.macroexpand` has been defined, however, it is executed, and what `macroexpand` returns—which may be a `string` or a `double`—is substituted.

For example, say that we wanted to make all objects of type `coordinate` substitute `(#,#)` when they were quoted. In the class definition for `coordinate`, we could define `.macroexpand`,

begin coordinate.class

```
version 15.0
class coordinate {
                [ declaration of member variables omitted ]
        }
[ definitions of class programs omitted ]
program .macroexpand
                local tosub : display "(" `.x' "," `.y' ")"
                class exit "`tosub'"
end
```

end coordinate.class

and now `coordinate`s will be substituted. Say that `.mycoord` is a `coordinate` currently set to (2,3). If we did not include `.macroexpand` in the `coordinate.class` file, typing

```
...`.mycoord'...
```

would not be an error but would merely result in

```
......
```

Having defined `.macroexpand`, it will result in

```
...(2,3)...
```

A `.macroexpand` member function is intended as a utility for returning the printable form of a class instance and nothing more. In fact, the class system prevents unintended corruption of class-member variables by making a copy, returning the printable form, and then destroying the copy. These steps ensure that implicitly calling `.macroexpand` has no side effects on the class instance.

11. Object destruction

To create an instance of a class, you type

.name `=` *.classname*`.new` [*arguments*]

To destroy the resulting object and thus release the memory associated with it, you type

`classutil drop` *.name*

(See [P] **classutil** for more information on the `classutil` command.) You can drop only top-level instances. Objects deeper than that are dropped when the higher-level object containing them is dropped, and classes are automatically dropped when the last instance of the class is dropped.

Also any top-level object named with a name obtained from `tempname`—see [P] **macro**—is automatically dropped when the program concludes. Even so, `tempname` objects may be returned by `class exit`. The following is valid:

```
program .tension
                ...
                tempname a b
                .`a' = .bubble.new
                .`b' = .bubble.new
                ...
                class exit .`a'
end
```

The program creates two new class instances of `bubbles` in the global context, both with temporary names. We can be assured that `.‘a’` and `.‘b’` are global because the names `‘a’` and `‘b’` were obtained from `tempname` and therefore cannot already exist in whatever context in which `.tension` runs. Therefore, when the program ends, `.‘a’` and `.‘b’` will be automatically dropped. Even so, `.tension` can return `.‘a’`. It can do that because, at the time `class exit` is executed, the program has not yet concluded and `.‘a’` still exists. You can even code

```
program .tension
                ...
                tempname a b
                .‘a’ = .bubble.new
                .‘b’ = .bubble.new
                ...
                class exit .‘a’.ref
end
```

and that also will return `.a` and, in fact, will be faster because no extra copy will be made. This form is recommended when returning an object stored in a temporary name. Do not, however, add `.ref`s on the end of “real” (nontemporary) objects being returned because then you would be returning not just the same values as in the real object but the object itself.

You can clear the entire class system by typing `discard`; see [P] **discard**. There is no `classutil drop _all` command: Stata’s graphics system also uses the class system, and dropping all the class definitions and instances would cause `graph` difficulty. `discard` also clears all open graphs, so the disappearance of class definitions and instances causes `graph` no difficulty.

During the development of class-based systems, you should type `discard` whenever you make a change to any part of the system, no matter how minor or how certain you are that no instances of the definition modified yet exist.

12. Advanced topics

12.1 Keys

The `.objkey` built-in function returns a `string` called a key that can be used to reference the object as an *rvalue* but not as an *lvalue*. This would typically be used in

```
local k = ‘.a.b.objkey’
```

or

```
.c.k = .a.b.objkey
```

where `.c.k` is a `string`. Thus the keys stored could be then used as follows:

`.d = .‘k’.x`	meaning to assign `.a.b.x` to `.d`
`.d = .‘.c.k’.x`	(same)
`local z = ‘.‘k’.x’`	meaning to put value of `.a.b.x` in `‘z’`
`local z = ‘.‘.c.k’.x’`	(same)

It does not matter if the key is stored in a macro or a string member variable—it can be used equally well—and you always use the key by macro quoting.

A key is a special string that stands for the object. Why not, you wonder, simply type `.a.b` rather than `.‘.c.k’` or `.‘k’`? The answer has to do with implied context.

Pretend that .myvar.bin.myprogram runs .myprogram. Obviously, it runs .myprogram in the context .myvar.bin. Thus .myprogram can include lines such as

```
.x = 5
```

and that is understood to mean that .myvar.bin.x is to be set to 5. .myprogram, however, might also include a line that reads

```
.Global.utility.setup `.x.objkey'
```

Here .myprogram is calling a utility that runs in a different context (namely, .utility), but myprogram needs to pass .x—of whatever type it might be—to the utility as an argument. Perhaps .x is a coordinate, and .utility.setup expects to receive the identifier of a coordinate as its argument. .myprogram, however, does not know that .myvar.bin.x is the full name of .x, which is what .utility.setup will need, so .myprogram passes `.x.objkey'. Program .utility.setup can use what it receives as its argument just as if it contained .myvar.bin.x, except that .utility.setup cannot use that received reference on the left-hand side of an assignment.

If myprogram needed to pass to .utility.setup a reference to the entire implied context (.myvar.bin), the line would read

```
.Global.utility.setup `.objkey'
```

because .objkey by itself means to return the key of the implied context.

12.2 Unames

The built-in function .uname returns a *name* that can be used throughout Stata that uniquely corresponds to the object. The mapping is one way. Unames can be obtained for objects, but the original object's name cannot be obtained from the uname.

Pretend that you have object .*a*.*b*.*c*, and you wish to obtain a name you can associate with that object because you want to create a variable in the current dataset, or a value label, or whatever else, to go along with the object. Later, you want to be able to reobtain that name from the object's name. .*a*.*b*.*c*.uname will provide that name. The name will be ugly, but it will be unique. The name is not temporary: you must drop whatever you create with the name later.

Unames are, in fact, based on the object's .ref. That is, consider two objects, .*a*.*b*.*c* and .*d*.*e*, and pretend that they refer to the same data; that is, you have previously executed

```
.a.b.c.ref = .d.e.ref
```

or

```
.d.e.ref = .a.b.c.ref
```

Then .*a*.*b*.*c*.uname will equal .*d*.*e*.uname. The names returned are unique to the data being recorded, not the identifiers used to arrive to the data.

As an example of use, within Stata's graphics system sersets are used to hold the data behind a graph; see [P] **serset**. An overall graph might consist of several graphs. In the object nesting for a graph, each individual graph has its own object holding a serset for its use. The individual objects, however, are shared when the same serset will work for two or more graphs, so that the same data are not recorded again and again. That is accomplished by simply setting their .refs equal. Much later in the graphics code, when that code is writing a graph out to disk for saving, it needs to figure out which sersets need to be saved, and it does not wish to write shared sersets out multiple times. Stata finds out what sersets are shared by looking at their unames and, in fact, uses the unames to help it keep track of which sersets go with which graph.

12.3 Arrays of member variables

Note: The following functions are of little use in class programming. They are of use to those writing utilities to describe the contents of the class system, such as the features documented in [P] **classutil**.

The built-in functions `.classmv`, `.instancemv`, and `.dynamicmv` each return an `array` containing the `.refs` of each classwide, instance-specific, and dynamically declared member variables. These array elements may be used as either *lvalues* or *rvalues*.

`.superclass` also returns an `array` containing `.refs`, these being references to the classes from which the current object inherited. These array elements may be used as *rvalues* but should not be used as `lvalues` because they refer to underlying class definitions themselves.

`.classmv`, `.instancemv`, `.dynamicmv`, and `.superclass`, although documented as built-in functions, are not really functions, but instead are built-in member variables. This means that, unlike built-in functions, their references may be followed by other built-in functions, and it is not an error to type, for instance,

```
... .li.instancemv.arrnels ...
```

and it would be odd (but allowed) to type

```
.myarray = .li.instancemv
```

It would be odd simply because there is no reason to copy them because you can use them in place.

Each of the above member functions are a little sloppy in that they return nothing (produce an error) if there are no classwide, instance-specific, and dynamically declared member variables, or no inherited classes. This sloppiness has to do with system efficiency, and the proper way to work around the sloppiness is to obtain the number of elements in each array as `0‘.classmv.arrnels’`, `0‘.instancemv.arrnels’`, `0‘.dynamicmv.arrnels’`, and `0‘.superclass.arrnels’`. If an array does not exist, then nothing will be substituted, and you will still be left with the result `0`.

For example, assume that `.my.c` is of type `coordinate2`, defined as

begin `coordinate2.class`

```
version 15.0
class coordinate2 {
                classwide:
                        double x_origin = 0
                        double y_origin = 0
                instancespecific:
                        double x = 0
                        double y = 0
}
```

end `coordinate2.class`

Then

referring to ...	is equivalent to referring to ...
`.my.c.classmv[1]`	`.my.c.c.x_origin`
`.my.c.classmv[2]`	`.my.c.c.y_origin`
`.my.c.instancemv[1]`	`.my.c.c.x`
`.my.c.instancemv[2]`	`.my.c.c.y`

If any member variables were added dynamically using `.Dynamic`, they could equally well be accessed via `.my.c.dynamicmv[]` or their names. Either of the above could be used on the left or right of an assignment.

If `coordinate2.class` inherited from another class (it does not), referring to `.coordinate2.superclass[1]` would be equivalent to referring to the inherited class; `.coordinate2.superclass[1].new`, for instance, would be allowed.

These "functions" are mainly of interest to those writing utilities to act on class instances as a general structure.

Appendix A. Finding, loading, and clearing class definitions

The definition for class *xyz* is located in file *xyz*`.class`.

Stata looks for *xyz*`.class` along the ado-path in the same way that it looks for ado-files; see **[U] 17.5 Where does Stata look for ado-files?** and see [P] **sysdir**.

Class definitions are loaded automatically, as they are needed, and are cleared from memory as they fall into disuse.

When you type `discard`, all class definitions and all existing instances of classes are dropped; see [P] **discard**.

Appendix B. Jargon

built-in: a member program that is automatically defined, such as `.new`. A **built-in function** is a member program that returns a result without changing the object on which it was run. A **built-in modifier** is a member program that changes the object on which it was run and might return a result as well.

class: a name for which there is a class definition. If we say that `coordinate` is a class, then *coordinate*`.class` is the name of the file that contains its definition.

class instance: a "variable"; a specific, named copy (instance) of a class with its member values filled in; an identifier that is defined to be of *type classname*.

classwide variable: a member variable that is shared by all instances of a class. Its alternative is an instance-specific variable.

inheritance: the ability to define a class in terms of one (single inheritance) or more (multiple inheritance) existing classes. The existing class is typically called the base or super class, and by default, the new class inherits all the member variables and member programs of the base class.

identifier: the name by which an object is identified, such as `.mybox` or `.mybox.x`.

implied context: the instance on which a member program is run. For example, in `.`*a*`.`*b*`.myprog`, `.`*a*`.`*b* is the implied context, and any references to, say, `.`*x* within the program, are first assumed to, in fact, be references to `.`*a*`.`*b*`.`*x*.

instance: a class instance.

instance-specific variable: a member variable that is unique to each instance of a class; each instance has its own copy of the member variable. Its alternative is a classwide variable.

lvalue: an identifier that may appear to the left of the `=` assignment operator.

member program: a program that is a member of a class or of an instance.

member variable: a variable that is a member of a class or of an instance.

object: a class or an instance; this is usually a synonym for an instance, but in formal syntax definitions, if something is said to be allowed to be used with an object, that means it may be used with a class or with an instance.

polymorphism: when a system allows the same program name to invoke different programs according to the class of the object. For example, `.draw` might invoke one program when used on a star object, `.mystar.draw`, and a different program when used on a box object, `.mybox.draw`.

reference: most often the word is used according to its English-language definition, but a `.ref` reference can be used to obtain the data associated with an object. If two identifiers have the same reference, then they are the same object.

return value: what an object returns, which might be of type `double`, `string`, `array`, or *classname*. Generally, return value is used in discussions of member programs, but all objects have a return value; they typically return a copy of themselves.

rvalue: an identifier that may appear to the right of the `=` assignment operator.

scope: how it is determined to what object an identifier references. *.a.b* might be interpreted in the global context and literally mean *.a.b*, or it might be interpreted in an implied context to mean *.impliedcontext.a.b*.

shared object: an object to which two or more different identifiers refer.

type: the type of a member variable or of a return value, which is `double`, `string`, `array`, or *classnam*.

Appendix C. Syntax diagrams

Appendix C.1 Class declaration

`class` [*newclassname*] `{`

[`classwide:`]

[*type mvname* [`=` *rvalue*]]

[*mvname* `=` *rvalue*]

[...]

[`instancespecific:`]

[*type mvname* [`=` *rvalue*]]

[*mvname* `=` *rvalue*]

[...]

`}` [`, inherit(`*classnamelist*`)`]

where

mvname stands for member variable name;

rvalue is defined in *Appendix C.2 Assignment*; and

type is { *classname* | `double` | `string` | `array` }.

The .Declare built-in may be used to add a member variable to an existing class instance,

id[.*id*[...]] .Declare *type newmvname* [= *rvalue*]
id[.*id*[...]] .Declare *newmvname* = *rvalue*

where *id* is {*name* | *name*[*exp*]}, the latter being how you refer to an array element; *exp* must evaluate to a number. If *exp* evaluates to a noninteger number, it is truncated.

Appendix C.2 Assignment

lvalue = *rvalue*
lvalue.ref = *lvalue*.ref *(sic)*
lvalue.ref = *rvalue*

where

lvalue is .*id*[.*id*[...]]

rvalue is

"[*string*]"
‘"[*string*]"’
#
exp
(*exp*)
.*id*[.*id*[...]]
[.*id*[.*id*[...]]].*pgmname* [*pgm_arguments*]
[.*id*[.*id*[...]]].Super[(*classname*)].*pgmname* [*pgm_arguments*]
{}
{*el* [,*el* [,...]]}

When *exp* evaluates to a string, the result will contain at most 2045 characters and will be terminated early if it contains a binary 0.

The last two syntaxes concern assignment to arrays; *el* may be

nothing
"[*string*]"
‘"[*string*]"’
#
(*exp*)
.*id*[.*id*[...]]
[.*id*[.*id*[...]]].*pgmname*
[[.*id*[.*id*[...]]].*pgmname* [*pgm_arguments*]]
[[.*id*[.*id*[...]]].Super[(*classname*)].*pgmname* [*pgm_arguments*]]

id is {*name* | *name*[*exp*]}, the latter being how you refer to an array element; *exp* must evaluate to a number. If *exp* evaluates to a noninteger number, it is truncated.

Appendix C.3 Macro substitution

Values of member variables or values returned by member programs can be substituted in any Stata command line in any context using macro quoting. The syntax is

...`‘`*.id*[*.id*[...]]`’`...

...`‘`[*.id*[*.id*[...]]] *.pgmname*`’`...

...`‘`[*.id*[*.id*[...]]] *.pgmname pgm_arguments*`’`...

...`‘`[*.id*[*.id*[...]]]`.Super`[`(`*classname*`)`] *.pgmname*`’`...

...`‘`[*.id*[*.id*[...]]]`.Super`[`(`*classname*`)`] *.pgmname pgm_arguments*`’`...

Nested substitutions are allowed. For example,

...`‘.‘tmpname’.x’`...

...`‘‘ref’’`...

In the above, perhaps local `tmpname` was obtained from `tempname` (see [P] **macro**), and perhaps local `ref` contains `‘‘.myobj.cvalue’’`.

When a class object is quoted, its printable form is substituted. This is defined as

Object type	Printable form
`string`	contents of the string
`double`	number printed using `%18.0g`, spaces stripped
`array`	nothing
classname	nothing or, if member program `.macroexpand` is defined, then `string` or `double` returned

If the quoted reference results in an error, the error message is suppressed and nothing is substituted.

Appendix C.4 Quick summary of built-ins

Built-ins come in two forms: 1) built-in functions—built-ins that return a result but do not change the object on which they are run, and 2) built-in modifiers—built-ins that might return a result but more importantly modify the object on which they are run.

Built-in functions (may be used as *rvalues*)

.object`.id`	creates new instance of *.object*
.instance`.copy`	makes a copy of *.instance*
.instance`.ref`	for use in assignment by reference
.object`.objtype`	returns "`double`", "`string`", "`array`", or "*classname*"
.object`.isa`	returns "`double`", "`string`", "`array`", "`class`", or "`classtype`"
.object`.classname`	returns "*classname*" or ""
.object`.isofclass` *classname*	returns 1 if *.object* is of class type *classname*
.object`.objkey`	returns a string that can be used to refer to an object outside the implied context
.object`.uname`	returns a string that can be used as *name* throughout Stata; *name* corresponds to *.object*'s `.ref`.
.object`.ref_n`	returns number (`double`) of total number of identifiers sharing object
.array`.arrnels`	returns number (`double`) corresponding to largest index of the array assigned
.array`.arrindexof "`*string*`"`	searches array for first element equal to *string* and returns the index (`double`) of element or returns 0
.object`.classmv`	returns `array` containing the `.ref`s of each classwide member of *.object*
.object`.instancemv`	returns `array` containing the `.ref`s of each instance-specific member of *.object*
.object`.dynamicmv`	returns `array` containing the `.ref`s of each dynamically added member of *.object*
.object`.superclass`	returns `array` containing the `.ref`s of each of the classes from which *.object* inherited

Built-in modifiers

.instance`.Declare` *declarator*	returns nothing; adds member variable to instance; see *Appendix C.1 Class declaration*
.array`[`*exp*`].Arrdropel #`	returns nothing; drops the specified array element
.array`.Arrpop`	returns nothing; finds the top element and removes it
.array`.Arrpush "`*string*`"`	returns nothing; adds string to end of array

Also see

[P] **class exit** — Exit class-member program and return result

[P] **classutil** — Class programming utility

[P] **sysdir** — Query and set system directories

[M-2] **class** — Object-oriented programming (classes)

[U] 17.5 Where does Stata look for ado-files?