# Title

**class** — Object-oriented programming (classes)

## Syntax

```
class classname [extends classname] {
        declaration(s)
}
```

Syntax is presented under the following headings:

*Description* and *Remarks and examples* follow that.

### Introduction

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

### Example

The following example is explained in detail in *Description*.

```
class coord {
        real scalar    x, y
        real scalar    length(), angle()
}

real scalar coord::length()
{
        return(sqrt(x^2 + y^2))
}

real scalar coord::angle()
{
         return(atan2(y, x)*360/(2*pi()))
}

class rotated_coord extends coord {
         real scalar     theta
         real scalar     angle()
         void            new()
}
```

```
        real scalar rotated_coord::angle()
        {
                return(super.angle() - theta)
        }

        void rotated_coord::new()
        {
                theta = 0
        }
```

One could use the class interactively:

```
        : b = rotated_coord()
        : b.x = b.y = 1
        : b.angle()                    // displayed will be 45
        : b.theta = 30
        : b.angle()                    // displayed will be 15
```

Note that executing the class as if it were a function creates an instance of the class. When using the class inside other functions, it is not necessary to create the instance explicitly as long as you declare the member instance variable to be a `scalar`:

```
        void myfunc()
        {
                class rotated_coord scalar   b

                b.x = b.y = 1
                b.angle()              // displayed will be 45
                b.theta = 30
                b.angle()              // displayed will be 15
        }
```

## Declaration of member variables

Declarations are of the form

```
        [exposure] [static]
        [final] matatype name [, name [, ...]]
```

where

```
        exposure := { public | protected | private }
        matatype := { eltype orgtype | eltype | orgtype }
          eltype :=  transmorphic          orgtype := matrix
                     numeric                          vector
                     real                             rowvector
                     complex                          colvector
                     string                           scalar
                     pointer
                     class classname
                     struct structname
```

For example,

```
        class S {
                 real matrix                 M
                 private real scalar         type
                 static real scalar          count
                 class coord scalar          c
        }
```

## Declaration and definition of methods (member functions)

Declarations are of the form

```
        [exposure] [static]
        [final] [virtual]
        matatype name() [, name() [...]]
```

For example,

```
        class T {
                 ...
                 real matrix                 inverse()
                 protected real scalar       type()
                 class coord scalar          c()
        }
```

Note that function arguments, even if allowed, are not declared.

Member functions (methods) and member variables may share the same names and no special meaning is attached to the fact. `type` and `c` below are variables, and `type()` and `c()` are functions:

```
        class U {
                 real matrix                 M
                 private real scalar         type
                 static real scalar          count
                 class coord scalar          c

                 real matrix                 inverse()
                 protected real scalar       type()
                 class coord scalar          c()
        }
```

Member functions are defined separately, after the class is defined. For example,

```
        class V {
                 real matrix                 M
                 private real scalar         type
                 static real scalar          count
                 class coord scalar          c

                 real matrix                 inverse()
                 protected real scalar       type()
                 class coord scalar          c()
        }
        real matrix V::inverse(...)
        {
                ...
        }
        real scalar V::type(...)
        {
                ...
        }
        class coord scalar V::c(...)
        {
                ...
        }
```

When you define member functions, they must be of the same *matatype* as they were previously declared to be, but you omit *exposure* (as well as static, final, and virtual).

## Default exposure in declarations

Variables and functions are public unless explicitly declared otherwise. (They are also not static, not final, and not virtual, but that is not part of exposure and so has nothing to do with this subsection.)

You may use any of the exposure modifiers public, protected, and private, followed by a colon, to create blocks with a different default:

```
        class V {
            public:
                real matrix                 M
                static real scalar          count
                class coord scalar          c
                real matrix                 inverse()
                class coord scalar          c()
            private:
                real scalar                 type
            protected:
                real scalar                 type()
        }
```

## Description

class provides object-oriented programming, also known as class programming, to Mata.

A class is a set of variables or related functions (methods) (or both) tied together under one name. Classes may be derived from other classes according to inheritance.

Let's look at the details of the example from the first page of this entry (under the heading *Example*):

1. First, we created a class called coord. When we coded

```
                class coord {
                        real scalar    x, y
                        real scalar    length(), angle()
                }
```

we specified that each element of a coord stores two real values, which we called x and y. coord also contains two functions, which we called length() and angle(). length() and angle() are functions because of the open and close parentheses at the end of the names, and x and y are variables because of the absence of parentheses. In the jargon, x and y are called member variables, and length() and angle() are called member functions.

The above, called the class's definition, defines a blueprint for the type coord.

A variable that is of type coord is called an instance of a coord. Say variables b and c are instances of coord, although we have not yet explained how you might arrange that. Then b.x and b.y would be b's values of x and y, and c.x and c.y would be c's values. We could run the functions on the values in b by coding b.length() and b.angle(), or on the values in c by coding c.length() and c.angle().

2. Next we defined coord's length() and angle() functions. The definitions were

```
real scalar coord::length()
{
        return(sqrt(x^2 + y^2))
}

real scalar coord::angle()
{
        return(atan2(y, x)*360/(2*pi()))
}
```

   These functions are similar to regular Mata functions. These functions do not happen to take any arguments, but that is not a requirement. It often happens that member functions do not require arguments because the functions are defined in terms of the class and its concepts. That is exactly what is occurring here. The first function says that when someone has an instance of a coord and they ask for its length, return the square root of the sum of the x and y values, individually squared. The second function says that when someone asks for the angle, return the arctangent of y and x, multiplied by 360/(2*pi()) to convert the result from radians to degrees.

3. Next we defined another new concept, a rotated_coord, by extending (inheriting from) class coord:

```
class rotated_coord extends coord {
        real scalar     theta
        real scalar     angle()
        void            new()
}
```

   So think of a coord and read the above as the additions. In addition to x and y of a regular coord, a rotated_coord has new variable theta. We also declared that we were adding two functions, angle() and new(). But wait, angle() already existed! What we are actually saying by explicitly mentioning angle() is that we are going to change the definition of angle(). Function new() is indeed new.

   Notice that we did not mention previously existing function length(). Our silence indicates that the concept of length() remains unchanged.

   So what is a rotated_coord? It is a coord with the addition of theta, a redefinition of how angle() is calculated, and the addition of a function called new().

This is an example of inheritance. `class rotated_coord` is an extension of `class coord`. In object-oriented programming, we would say this as "`class rotated_coord` inherits from `class coord`". A class that inherits from another class is known as a "subclass".

4. Next we defined our replacement and new functions. The definitions are

```
real scalar rotated_coord::angle()
{
        return(super.angle() - theta)
}

void rotated_coord::new()
{
        theta = 0
}
```

Concerning `angle()`, we stated that it is calculated by taking the result of `super.angle()` and subtracting `theta` from it. `super.angle()` is how one refers to the parent's definition of a function. If you are unfamiliar with object-oriented programming, parent may seem like an odd word in this context. We are inheriting concepts from `coord` to define `rotated_coord`, and in that sense, `coord` is the parent concept. Anyway, the new definition of an angle is the old definition, minus `theta`, the angle of rotation.

`new()` is a special function and is given a special name in the sense that the name `new()` is reserved. The `new()` function, if it exists, is a function that is called automatically whenever a new instance of the class is created. Our `new()` function says that when a new instance of a `rotated_coord` is created, initialize the instance's `theta` to 0.

Well, that seems like a good idea. But we did not have a `new()` function when we defined our previous class, `coord`. Did we forget something? Maybe. When you do not specify a `new()` function—when you do not specify how variables are to be initialized—they are initialized in the usual Mata way: missing values. `x` and `y` will be initialized to contain missing. Given our `new()` function for `rotated_coord`, however, `theta` will be initialized to 0.

`new()` is called a "constructor", because it is used to construct, or initialize, the class when a new instance of the class is created.

And that completes the definition of our two, related classes.

There are two ways to create instances of a `coord` or a `rotated_coord`. One is mostly for interactive use and the other for programming use.

If you interactively type `a=coord()` (note the parentheses), you will create a `coord` and store it in `a`. If you interactively type `b=rotated_coord()`, you will create a `rotated_coord` and store it in `b`. In the first example, typing `b=rotated_coord()` is exactly what we chose to do:

```
: b = rotated_coord()
```

Recall that a `rotated_coord` contains an `x`, `y`, and `theta`. At this stage, `x` and `y` equal missing, and `theta` is 0. In the example, we set `b`'s `x` and `y` values to 1, and then asked for the resulting `angle()`:

```
: b.x = b.y = 1

: b.angle()
  45
```

b-dot-x is how one refers to b's value of x. One can use b.x (and b.y) just as one would use any real scalar variable in Mata.

If we reset b's theta to be 30 degrees, then b's angle ought to change to being 15 degrees. That is exactly what happens:

```
: b.theta = 30

: b.angle()
  15
```

b-dot-angle() is how one specifies that member function angle() is to be run on instance b. Now you know why member functions so seldom have additional arguments: they are, in effect, passed an instance of a class and so have access to all the values of member variables of that class. We repeat, however, that a member function could take additional arguments. Had we coded

```
real scalar rotated_coord::angle(real scalar base)
{
        return(super.angle() - theta - base)
}
```

then angle() would have taken an argument and returned the result measured from base.

The difference between using rotated_coord interactively and using it inside a Mata program is that if we declare a variable (say, b) to be a class rotated_coord scalar, with the emphasis on scalar, then we do not need to bother coding b=rotated_coord() to fill in b initially. Coding class rotated_coord scalar b implies that b needs to be initialized because it is a scalar, and so that happens automatically. It would not hurt if we also coded b=rotated_coord(), but it would be a waste of our time and of the computer's once it got around to executing our program.

Now let's show you something we did not show in the first example. Remember when we defined length() for a coord? Remember how we did not define length() for a rotated_coord? Remember how we did not even mention length()? Even so, length() is a concept of a rotated_coord, because part of the definition of rotated_coord was inherited from coord, and that happened because when we declared rotated_coord, we said

```
class rotated_coord extends coord
```

The inheritance happened because we said extends. Let's test that length() works with our rotated_coord class instance, b:

```
: b.length()
  1.414213562
```

In the above, inheritance is what saved us from having to write additional, repeated code.

Let's review. First, we defined a coord. From that, we defined a rotated_coord. You might now define translated_and_rotated_coord using rotated_coord as a starting point. It would not be difficult.

Classes have lots of properties, features, and details, but it is the property of inheritance that is at the heart of object-oriented programming.

## Remarks and examples

Remarks are presented under the following headings:

## Notation and jargon

*:: (double colon)*

The double-colon notation is used as a shorthand in documentation to indicate that a variable or function is a member of a class and, in two cases, the double-colon notation is also syntax that is understood by Mata.

S::s indicates that the variable s is a member of class S. S::s is documentation shorthand.

S::f() indicates that function f() is a member of class S. S::f() is documentation shorthand. `S::f()` is something Mata itself understands in two cases:

1. Notation `S::f()` is used when defining member functions.
2. Notation `S::f()` can be used as a way of calling static member functions.

*be an instance of*

When we write "let s be an instance of S" we are saying that s is an instance of class S, or equivalently, that in some Mata code, s is declared as a `class S scalar`.

*class definition, class declaration*

A class definition or declaration is the definition of a class, such as the following:

```
class S {
        private real scalar     a, b
        real scalar             f(), g()
}
```

Note well that the declaration of a Mata variable to be of type `class S`, such as the line `class S scalar s` in

```
void myfunction()
{
        class S scalar  s
        ...
}
```

is not a class definition. It is a declaration of an instance of a class.

*class instance*, *instance*, *class A instance*

A class instance is a variable defined according to a class definition. In the code

```
void myfunction()
{
        class S scalar      s
        real scalar         b
        ...
}
```

s is a class instance, or, more formally, an instance of class S. The term “instance” can be used with all element types, not just with classes. Thus b is an instance of a real. The term “instance” is more often used with classes and structures, however, because one needs to distinguish definitions of variables containing classes or structures from the definitions of the classes and structures themselves.

*inheritance*, *extends*, *subclass*, *parent*, *child*

Inheritance is the property of one class definition using the variables and functions of another just as if they were its own. When a class does this, it is said to extend the other class. T extending S means the same thing as T inheriting from S. T is also said to be a subclass of S.

Consider the following definitions:

```
class S {
        real scalar     a, b
        real scalar     f()
}
class T extends S {
        real scalar     c
        real scalar     g()
}
```

Let s be an instance of S and t be an instance of T. It is hardly surprising that s.a, s.b, s.f(), t.c, and t.g() exist, because each is explicitly declared. It is because of inheritance that t.a, t.b, and t.f() also exist, and they exist with no additional code being written.

If U extends T extends S, then U is said to be a child of T and to be a child of S, although formally U is the grandchild of S. Similarly, both S and T are said to be parents of U, although formally S is the grandparent of U. It is usually sufficient to label the relationship parent/child without going into details.

*external functions*

An external function is a regular function, such as sqrt(), sin(), and myfcn(), defined outside of the class and, as a matter of fact, outside of all classes. The function could be a function provided by Mata (such as sqrt() or sin()) or it could be a function you have written, such as myfcn().

An issue arises when calling external functions from inside the code of a member function. When coding a member function, references such as sqrt(), sin(), and myfcn() are assumed to be references to the class’s member functions if a member function of the name exists. If one wants to ensure that the external-function interpretation is made, one codes ::sqrt(), ::sin(), and ::myfcn(). See *Accessing external functions from member functions* below.

*member*, *member variable*, *member function*, *method*

A variable or function declared within a class is said to be a member variable, or member function, of the class. Member functions are also known as methods. In what follows, let `ex1` be an instance of `S`, and assume class `S` contains member function `f()` and member variable `v`.

When member variables and functions are used inside member functions, they can simply be referred to by their names, `v` and `f()`. Thus, if we were writing the code for S::g(), we could code

```
real scalar S::g()
{
        return(f()*v)
}
```

When member variables and functions are used outside of member functions, which is to say, are used in regular functions, the references must be prefixed with a class instance and a period. Thus one codes

```
real scalar myg(class S scalar ex1)
{
        return(ex1.f()*ex1.v)
}
```

*variable and method overriding*

Generically, a second variable or function overrides a first variable or function when they share the same name and the second causes the first to be hidden, which causes the second variable to be accessed or the second function to be executed in preference to the first. This arises in two ways in Mata's implementation of classes.

In the first way, a variable or function in a parent class is said to be overridden if a child defines a variable or function of the same name. For instance, let `U` extend `T` extend `S`, and assume `S::f()` and `T::f()` are defined. By the rules of inheritance, instances of `U` and `T` that call `f()` will cause `T::f()` to execute. Instances of `S` will cause `S::f()` to be executed. Here `S.f()` is said to be overridden by `T.f()`. Because `T::f()` will usually be implemented in terms of `S::f()`, `T::f()` will find it necessary to call `S::f()`. This is done by using the `super` prefix; see *Using super to access the parent's concept*.

The second way has to do with stack variables having precedence over member variables in member functions. For example,

```
class S {
        real scalar     a, b
        void            f()
        ...
}

void S::f()
{
        real scalar     a

        a = 0
        b = 0
}
```

Let `s` be an instance of `S`. Then execution of `s.f()` sets `s.b` to be 0; it does not change `s.a`, although it was probably the programmer's intent to set `s.a` to zero, too. Because the programmer declared a variable named a within the program, however, the program's variable took precedence over the member variable a. One solution to this problem would be to change the a = 0 line to read this.a = 0; see *Referring to the current class using this*.

## Declaring and defining a class

Let's say we declared a class by executing the code below:

```
class coord {
        real scalar    x, y
        real scalar    length(), angle()
}
```

At this point, class coord is said to be declared but not yet fully defined because the code for its member functions length() and angle() has not yet been entered. Even so, the class is partially functional. In particular,

1. The class will work. Obviously, if an attempt to execute length() or angle() is made, an error message will be issued.

2. The class definition can be saved in an .mo or .mlib file for use later, with the same proviso as in 1).

Member functions of the class are defined in the same way regular Mata functions are defined but the name of the function is specified as *classname*::*functionname*().

```
real scalar coord::length()
{
        return(sqrt(x^2 + y^2))
}
real scalar coord::angle()
{
        return(atan2(y, x)*360/(2*pi()))
}
```

The other difference is that member functions have direct access to the class's member variables and functions. x and y in the above are the x and y values from an instance of class coord.

Class coord is now fully defined.

## Saving classes in files

The notation coord()—classname-open parenthesis-close parenthesis—is used to refer to the entire class definition by Mata's interactive commands such as mata describe, mata mosave, and mata mlib.

mata describe coord() would show

```
: mata describe coord()
      # bytes   type                        name and extent
------------------------------------------------------------
          344   classdef scalar             coord()
          176   real scalar                  ::angle()
          136   real scalar                  ::length()
------------------------------------------------------------
```

The entire class definition—the compiled coord() and all its compiled member functions—can be stored in a .mo file by typing

```
: mata mosave coord()
```

The entire class definition could be stored in an already existing .mlib library, lpersonal, by typing

```
: mata mlib add lpersonal coord()
```

When saving a class definition, both commands allow the additional option complete. The option specifies that the class is to be saved only if the class definition is complete and that, otherwise, an error message is to be issued.

```
: mata mlib add lpersonal coord(), complete
```

## Workflow recommendation

Our recommendation is that the source code for classes be kept in separate files, and that the files include the complete definition. At StataCorp, we would save coord in file coord.mata:

begin coord.mata

```
*! version 1.0.0 class coord
version 13
mata:
class coord {
         real scalar    x, y
         real scalar    length(), angle()
}
real scalar coord::length()
{
         return(sqrt(x^2 + y^2))
}
real scalar coord::angle()
{
        return(atan2(y, x)*360/(2*pi()))
}
end
```

end coord.mata

Note that the file does not clear Mata, nor does it save the class in a .mo or .mlib file; the file merely records the source code. With this file, to save coord in a .mo file, we need only to type

```
. clear mata
. do coord.mata
. mata mata mosave coord(), replace
```

(Note that although file coord.mata does not have the extension .do, we can execute it just like we would any other do-file by using the do command.) Actually, we would put those three lines in yet another do-file called, perhaps, cr_coord.do.

We similarly use files for creating libraries, such as

begin cr_lourlib.do

```
version 13
clear mata
do coord.mata
do anotherfile.mata
do yetanotherfile.mata
/* etc. */

mata:
mata mlib create lourlib, replace
mata mlib add *()
end
```

end cr_lourlib.do

With the above, it is easy to rebuild libraries after updating code.

## When you need to recompile

If you change a class declaration, you need to recompile all programs that use the class. That includes other class declarations that inherit from the class. For instance, in the opening example, if you change anything in the `coord` declaration,

```
class coord {
        real scalar    x, y
        real scalar    length(), angle()
}
```

even if the change is so minor as to reverse the order of `x` and `y`, or to add a new variable or function, you must not only recompile all of `coord`'s member functions, you must also recompile all functions that use `coord`, and you must recompile `rotated_coord` as well, because `rotated_coord` inherited from `coord`.

You must do this because Mata seriously compiles your code. The Mata compiler deconstructs all use of classes and substitutes low-level, execution-time-efficient constructs that record the address of every member variable and member function. The advantage is that you do not have to sacrifice run-time efficiency to have more readable and easily maintainable code. The disadvantage is that you must recompile when you make changes in the definition.

You do not need to recompile outside functions that use a class if you only change the code of member functions. You do need to recompile if you add or remove member variables or member functions.

To minimize the need for recompiling, especially if you are distributing your code to others physically removed from you, you may want to adopt the pass-through approach.

Assume you have written large, wonderful systems in Mata that all hinge on object-oriented programming. One class inherits from another and that one from another, but the one class you need to tell your users about is class `wonderful`. Rather than doing that, however, merely tell your users that they should declare a `transmorphic` named `wonderfulhandle`—and then just have them pass `wonderfulhandle` around. They begin using your system by making an initialization call:

```
wonderfulhandle = wonderful_init()
```

After that, they use regular functions you provide that require `wonderful` as an argument. From their perspective, `wonderfulhandle` is a mystery. From your perspective, `wonderful_init()` returned an instance of a class `wonderful`, and the other functions you provided receive an instance of a `wonderful`. Sadly, this means that you cannot reveal to your users the beauty of your underlying class system, but it also means that they will not have to recompile their programs when you distribute an update. If the Mata compiler can compile their code knowing only that `wonderfulhandle` is a `transmorphic`, then it is certain their code does not depend on how `wonderfulhandle` is constructed.

## Obtaining instances of a class

Declaring a class, such as

```
class coord {
        real scalar    x, y
        real scalar    length(), angle()
}
```

in fact creates a function, `coord()`. Function `coord()` will create and return instances of the class:

- `coord()`—`coord()` without arguments—returns a scalar instance of the class.
- `coord(3)`—`coord()` with one argument, here 3—returns a 1 × 3 vector, each element of which is a separate instance of the class.
- `coord(2,3)`—`coord()` with two arguments, here 2 and 3—returns a 2 × 3 matrix, each element of which is a separate instance of the class.

Function `coord()` is useful when using Mata interactively. It is the only way to create instances of a class interactively.

In functions, you can create scalar instances by coding `class coord scalar` *name*, but in all other cases, you will also need to use function `coord()`. In the following example, the programmer wants a vector of `coord`s:

```
void myfunction()
{
         real scalar              i
         class coord vector       v
         ...
         v = coord(3)
         for (i=1; i<=3; i++) v[i].x = v[i].y = 1
         ...
}
```

This program would have generated a compile-time error had the programmer omitted the line `class coord vector v`. Variable declarations are usually optional in Mata, but variable declarations of type class are not optional.

This program would have generated a run-time error had the programmer omitted the line `v = coord(3)`. Because `v` was declared to be `vector`, `v` began life as 1 × 0. The first thing the programmer needed to do was to expand `v` to be 1 × 3.

In practice, one seldom needs class vectors or matrices. A more typical program would read

```
        void myfunction()
        {
                 real scalar               i
                 class coord scalar        v
                 ...
                 v.x = v.y = 1
                 ...
        }
```

Note particularly the line `class coord scalar v`. The most common error programmers make is to omit the word `scalar` from the declaration. If you do that, then `matrix` is assumed, and then, rather than `v` being $1 \times 1$, it will be $0 \times 0$. If you did omit the word `scalar`, you have two alternatives. Either go back and put the word back in, or add the line `v = coord()` before initializing `v.x` and `v.y`. It does not matter which you do. In fact, when you specify `scalar`, the compiler merely inserts the line `v = coord()` for you.

## Constructors and destructors

You can specify how variables are initialized, and more, each time a new instance of a class is created, but before we get to that, let's understand the default initialization. In

```
        class coord {
                real scalar    x, y
                real scalar    length(), angle()
        }
```

there are two variables. When an instance of a `coord` is created, say, by coding `b = coord()`, the values are filled in the usual Mata way. Here, because `x` and `y` are scalars, `b.x` and `b.y` will be filled in with missing. Had they been vectors, row vectors, column vectors, or matrices, they would have been dimensioned $1 \times 0$ (vectors and row vectors), $0 \times 1$ (column vectors), and $0 \times 0$ (matrices).

If you want to control initialization, you may declare a constructor function named `new()` in your class:

```
        class coord {
                 real scalar    x, y
                 real scalar    length(), angle()
                 void           new()
        }
```

Function `new()` must be declared to be `void` and must take no arguments. You never bother to call `new()` yourself—in fact, you are not allowed to do that. Instead, function `new()` is called automatically each time a new instance of the class is created.

If you wanted every new instance of a `coord` to begin life with `x` and `y` equal to 0, you could code

```
        void coord::new()
        {
                 x = y = 0
        }
```

Let's assume we do that and we then inherit from coord when creating the new class rotated_coord, just as shown in the first example.

```
class rotated_coord extends coord {
        real scalar theta
        real scalar     angle()
        void            new()
}

void rotated_coord::new()
{
        theta = 0
}
```

When we create an instance of a rotated_coord, all the variables were initialized to 0. That is, function rotated_coord() will know to call both coord::new() and rotated_coord::new(), and it will call them in that order.

In your new() function, you are not limited to initializing values. new() is a function and you can code whatever you want, so if you want to set up a link to a radio telescope and check for any incoming messages, you can do that. Closer to home, if you were implementing a file system, you might use a static variable to count the number of open files. (We will explain static member variables later.)

On the other side of instance creation—instance destruction—you can declare and create void destroy(), which also takes no arguments, to be called each time an instance is destroyed. destroy() is known as a destructor. destroy() works like new() in the sense that you are not allowed to call the function directly. Mata calls destroy() for you whenever Mata is releasing the memory associated with an instance. destroy() does not serve much use in Mata because, in Mata, memory management is automatic and the freeing of memory is not your responsibility. In some other object-oriented programming languages, such as C++, you are responsible for memory management, and the destructor is where you free the memory associated with the instance.

Still, there is an occasional use for destroy() in Mata. Let's consider a system that needs to count the number of instances of itself that exists, perhaps so it can release a resource when the instance count goes to 0. Such a system might, in part, read

```
class bigsystem {
        static real scalar counter
        ...
        void                    new(), destroy()
        ...
}

void bigsystem::new()
{
        counter = (counter==. ? 1 : counter+1)
}

void bigsystem::destroy()
{
        counter--
}
```

Note that bigsystem::new() must deal with two initializations, first and subsequent. The first time it is called, counter is . and new() must set counter to be 1. After that, new() must increment counter.

If this system needed to obtain a resource, such as access to a radio telescope, on first initialization and release it on last destruction, and be ready to repeat the process, the code could read

```
void bigsystem::new()
{
        if (counter==.) {
                get_resource()
                counter = 1
        }
        else {
                ++counter
        }
}

void bigsystem::destroy()
{
        if (--counter == 0) {
                release_resource()
                counter = .
        }
}
```

Note that destroy()s are run whenever an instance is destroyed, even if that destruction is due to an abort-with-error in the user's program or even in the member functions. The radio telescope will be released when the last instance of bigsystem is destroyed, except in one case. The exception is when there is an error in destroy() or any of the subroutines destroy() calls.

For inheritance, child destroy()s are run before parent destroy()s.

## Setting member variable and member function exposure

Exposure specifies who may access the member variables and member functions of your class. There are three levels of exposure: from least restrictive to most restrictive, public, protected, and private.

- Public variables and functions may be accessed by anyone, including callers from outside a class.
- Protected variables and functions may be accessed only by member functions of a class and its children.
- Private variables and functions may be accessed only by member functions of a class (excluding children).

When you do not specify otherwise, variables and functions are public. For code readability, you may declare member variables and functions to be public with public, but this is not necessary.

In programming large systems, it is usually considered good style to make member variables private and to provide public functions to set and to access any variables that users might need. This way, the internal design of the class can be subsequently modified and other classes that inherit from the class will not need to be modified, they will just need to be recompiled.

You make member variables and functions protected or private by preceding their declaration with protected or private:

```
        class myclass {
                private real matrix      X
                private real vector      y
                void                     setup()
                real matrix              invert()
                protected real matrix    invert_subroutine()
        }
```

Alternatively, you can create blocks with different defaults:

```
        class myclass {
            public:
                void                     setup()
                real matrix              invert()
            protected:
                real matrix              invert_subroutine()
            private:
                real matrix              X
                real vector              y
        }
```

You may combine public, private, and protected, with or without colons, freely.

## Making a member final

A member variable or function is said to be final if no children define a variable or function of the same name. Ensuring that a definition is the final definition can be enforced by including final in the declaration of the member, such as

```
        class myclass {
            public:
                void                     setup()
                real matrix              invert()
            protected:
                final real matrix        invert_subroutine()
            private:
                real matrix              X
                real vector              y
        }
```

In the above, no class that inherits from myclass can redefine invert_subroutine().

## Making a member static

Being static is an optional property of a class member variable function. In what follows, let ex1 and ex2 both be instances of S, assume c is a static variable of the class, and assume f() is a static function of the class.

A static variable is a variable whose value is shared across all instances of the class. Thus, ex1.c and ex2.c will always be equal. If ex1.c is changed, say, by ex1.c=5, then ex2.c will also be 5.

A static function is a function that does not use the values of nonstatic member variables of the class. Thus `ex1.f(3)` and `ex2.f(3)` will be equal regardless of how `ex1` and `ex2` differ.

Outside of member functions, static functions may also be invoked by coding the class name, two colons, and the function name, and thus used even if you do not have a class instance. For instance, equivalent to coding `ex1.f(3)` or `ex2.f(3)` is `S::f(3)`.

## Virtual functions

When a function is declared to be virtual, the rules of inheritance act as though they were reversed. Usually, when one class inherits from another, if the child class does not define a function, then it inherits the parent's function. For virtual functions, however, the parent has access to the child's definition of the function, which is why we say that it is, in some sense, reverse inheritance.

The canonical motivational example of this deals with animals. Without loss of generality, we will use barnyard animals to illustrate. A class, `animal`, is defined. Classes `cow` and `sheep` are defined that extend (inherit from) `animal`. One of the functions defined in `animal` needs to make the sound of the specific animal, and it calls virtual function `sound()` to do that. At the `animal` level, function `animal::sound()` is defined to display "Squeak!". Because function `sound()` is virtual, however, each of the specific-animal classes is supposed to define their own sound. Class `cow` defines `cow::sound()` that displays "Moo!" and class `sheep` defines `sheep::sound()` that displays "Baa!" The result is that when a routine in `animal` calls `sound()`, it behaves as if the inheritance is reversed, the appropriate `sound()` routine is called, and the user sees "Moo!" or "Baa!" If a new specific animal is added, and if the programmer forgets to define `sound()`, then `animal::sound()` will run, and the user sees "Squeak!". In this case, that is supposed to be the sound of the system in trouble, but it is really just the default action.

Let's code this example. First, we will code the usual case:

```
class animal {
        ...
        void        sound()
        void        poke()
        ...
}

void animal::sound() { "Squeak!" }

void animal::poke()
{
        ...
        sound()
        ...
}

class cow extends animal {
        ...
}

class sheep extends animal {
        ...
}
```

In the above example, when an animal, cow, or sheep is poked, among other things, it emits a sound. Poking is defined at the animal level because poking is mostly generic across animals, except for the sound they emit. If c is an instance of cow, then one can poke that particular cow by coding c.poke(). poke(), however, is inherited from animal, and the generic action is taken, along with the cow emitting a generic squeak.

Now we make sound() virtual:

```
        class animal {
                ...
                virtual void  sound()
                void          poke()
                ...
        }
        void animal::sound() { "Squeak!" }
        void animal::poke()
        {
                ...
                sound()
                ...
        }
        class cow extends animal {
                ...
                virtual void  sound()
        }
        void cow::sound() { "Moo!" }
        class sheep extends animal {
                ...
                virtual void  sound()
        }
        void sheep::sound() { "Baa!" }
```

Now let's trace through what happens when we poke a particular cow by coding c.poke(). c, to remind you, is a cow. There is no cow::poke() function; however, c.poke() executes animal::poke(). animal::poke() calls sound(), which, were sound() not a virtual function, would be animal::sound(), which would emit a squeak. Poke a cow, get a "Squeak!" Because sound() is virtual, animal::poke() called with a cow calls cow::sound(), and we hear a "Moo!"

Focusing on syntax, it is important that both cow and sheep repeated virtual void sound() in their declarations. If you define function *S*::*f()*, then *f()* must be declared in *S*. Consider a case, however, where we have two breeds of cows, Angus and Holstein, and they emit slightly different sounds. The Holstein, being of Dutch origin, gets a bit of a U sound into the moo in a way no native English speaker can duplicate. Thus we would declare two new classes, angus extends cow and holstein extends cow, and we would define angus::sound() and holstein::sound(). Perhaps we would not bother to define angus::sound(); then an Angus would get the generic cow sound.

But let's pretend that, instead, we defined angus::sound() and removed the function for cow::sound(). Then it does not matter whether we include the line virtual void sound() in cow's declaration. Formally, it should be included, because the line of reverse declaration should not be broken, but Mata does not care one way or the other.

A common use of virtual functions is to allow you to process a list of objects without any knowledge of the specific type of object, as long as all the objects are subclasses of the same base class:

```
class animal     animals

animals = animal(3,1)

animals[1] = cow()
animals[2] = sheep()
animals[3] = holstein()

for(i=1; i<=length(animals); i++) {
        animals[i].sound()
}
```

Note the use of `animals = animal(3,1)` to initialize the vector of animals. This is an example of how to create a nonscalar class instance, as discussed in *Obtaining instances of a class*.

When the code above is executed, the appropriate sound for each animal will be displayed even though `animals` is a vector declared to be of type `class animal`. Because `sound()` is `virtual`, the appropriate sound from each specific animal child class is called.

## Referring to the current class using this

`this` is used within member functions to refer to the class as a whole. For instance, consider the following:

```
class S {
        real scalar     n
        real matrix     M
        void            make_M()
}

void S::make_M(real scalar n)
{
        real scalar    i, j

        this.n = n
        M = J(n, n, 0)
        for (i=1; i<=n; i++) {
                for (j=1; j<=i; j++) M[i,j] = 1
        }
}
```

In the above program, references to `M` are understood to be the class instance definition of `M`. References to `n`, however, refer to the function's definition of `n` because the program's `n` has precedence over the class's definition of it. `this.n` is how one refers to the class's variable in such cases. `M` also could have been referred to as `this.M`, but that was not necessary.

If, in function `S::f()`, it was necessary to call a function outside of the class, and if that function required that we pass the class instance as a whole as an argument, we could code `this` for the class instance. The line might read `outside_function(this)`.

## Using super to access the parent's concept

The `super` modifier is a way of dealing with variables and functions that have been overridden in subclasses or, said differently, is a way of accessing the parent's concept of a variable or function.

Let `T` extend `S` and let `t` be an instance of `T`. Then `t.f()` refers to `T::f()` if the function exists, and otherwise to `S::f()`. `t.super.f()` always refers to `S::f()`.

More generally, in a series of inheritances, `z.super.f()` refers to the parent's concept of `f()`—the `f()` the parent would call if no `super` were specified—`z.super.super.f()` refers to the grandparent's concept of `f()`, and so on. For example, let `W` extend `V` extend `U` extend `T` extend `S`. Furthermore, assume

| | | |
|---|---|---|
| `S::f()` | exists | |
| `T::f()` | | does not exist |
| `U::f()` | exists | |
| `V::f()` | | does not exist |
| `W::f()` | | does not exist |

Finally, let `s` be an instance of `S`, `t` be an instance of `T`, and so on. Then calls of the form `w.f()`, `w.super.f()`, ..., `w.super.super.super.super.f()`, `v.f()`, `v.super.f()`, and so on, result in execution of

| | Number of `super`s specified | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| `s.` | `S::f()` | | | | |
| `t.` | `S::f()` | `S::f()` | | | |
| `u.` | `U::f()` | `S::f()` | `S::f()` | | |
| `v.` | `U::f()` | `U::f()` | `S::f()` | `S::f()` | |
| `w.` | `U::f()` | `U::f()` | `U::f()` | `S::f()` | `S::f()` |

## Casting back to a parent

A class instance may be treated as a class instance of a parent by casting. Assume `U` extends `T` extends `S`, and let `u` be an instance of `U`. Then

```
(class T) u
```

is treated as an instance of `T`, and

```
(class S) u
```

is treated as an instance of `S`. `((class T) u)` could be used anywhere an instance of `T` is required and `((class S) u)` could be used anywhere an instance of `S` is required.

For instance, assume `S::f()` is overridden by `T::f()`. If an instance of `U` found it necessary to call `S::f()`, one way it could do that would be `u.super.super.f()`. Another would be

```
((class S) u).f()
```

## Accessing external functions from member functions

In the opening example, class coord contained a member function named length(). Had we been coding a regular (nonclass) function, we could not have created that function with the name length(). The name length() is reserved by Mata for its own function that returns the length (number of elements) of a vector; see [M-5] **rows()**. We are allowed to create a function named length() inside classes. Outside of member functions, c.length() is an unambiguous reference to c's definition of length(), and length() by itself is an unambiguous reference to length()'s usual definition.

There is, however, possible confusion when coding member functions. Let's add another member function to coord: ortho(), the code for which reads

```
class coord scalar coord::ortho()
{
        class coord scalar      newcoord
        real scalar             r, t

        r = length()
        t = angle()
        newcoord.x = r*cos(t)
        newcoord.y = r*sin(t)
        return(newcoord)
}
```

Note that in the above code, we call length(). Because we are writing a member function, and because the class defines a member function of that name, length() is interpreted as being a call to coord's definition of length(). If coord did not define such a member function, then length() would have been interpreted as a call to Mata's length().

So what do we do if the class provides a function, externally there is a function of the same name, and we want the external function? We prefix the function's name with double colons. If we wanted length() to be interpreted as Mata's function and not the class's, we would have coded

```
r = ::length()
```

There is nothing special about the name length() here. If the class provided member function myfcn(), and there was an external definition as well, and we wanted the externally defined function, we would code ::myfcn(). In fact, it is not even necessary that the class provide a definition. If we cannot remember whether class coord includes myfcn(), but we know we want the external myfcn() no matter what, we can code ::myfcn().

Placing double-colons in front of external function names used inside the definitions of member functions is considered good style. This way, if you ever go back and add another member function to the class, you do not have to worry that some already written member function is assuming that the name goes straight through to an externally defined function.

## Pointers to classes

Just as you can obtain a pointer to any other Mata variable, you can obtain a pointer to an instance of a class. Recall our example in *Virtual functions* of animals that make various sounds when poked.

For the sake of example, we will obtain a pointer to an instance of a cow() and access the poke() function through the pointer:

```
        void pokeacow() {
                class cow scalar                     c
                pointer(class cow scalar) scalar     p

                p = &c
                p->poke()
        }
```

You access the member variables and functions through a pointer to a class with the operator `->` just like you would for structures. See *Pointers to structures* in [M-2] **struct** for more information.

## Also see

[M-2] **declarations** — Declarations and types

[M-2] **struct** — Structures

[P] **class** — Class programming

[M-2] **intro** — Language definition