CLASCAL REFERENCE MANUAL
for the LISA ™

First Draft, March 9th, MCMLXXXIII

by
David Cásseres

CONTENTS

1 Introduction to Clascal Concepts

Clascal is a set of extensions to Pascal on the Lisa. These extensions support "object-oriented" programming in a style that somewhat resembles SIMULA and Smalltalk. The purpose is to provide a very high-level interface to code libraries, allowing the user program to perform highly complicated functions with simple calls, while still retaining flexibility.

1.1 Class-Types

Clascal is based on a new category of user-defined types called *class-types*. An individual class-type is referred to as a *class*.

A class-type is a kind of structured-type, resembling a record-type in that it contains named fields. A class can have two kinds of fields:

- *Data fields* are like the fields of a record; they contain variable data, and each data field has its own type.
- *Methods* are procedures and functions.

The fields are referenced like fields of a record, using a period and a field-identifier (or a with-statement that references a field-identifier). For example, if **area** identifies a field defined in class **Triangle**, and **crntTriangle** is declared by

```
var crntTriangle: Triangle;
```

then **crntTriangle.area** is a reference to the **area** field of **crntTriangle**. If **area** is a data field, then **crntTriangle.area** is a variable-reference; if **area** is a method, then **crntTriangle.area** is either a procedure-statement or a function-call.

A class-type is declared in the interface-part of a unit, and is supported by a *method-block* in the implementation-part of the same unit. Section 3 gives the syntax for class-types and method-blocks.

1.2 Objects

A class defines the behavior (data fields and methods) of its *objects*. Each object is an *instance* of the class that defines its behavior.

Each object is stored in a dynamically allocated, potentially relocatable data area within a heap. An object of a given class is created by the **new** method defined for that class; this method returns a newly created object of the class (see Section 4.6).

A variable of a class-type references an object (once it has been initialized). You can think of a class-type variable as being an object, if you bear the following in mind:

- When an object is assigned to a class-type variable, the variable does not become a new copy of the object: it becomes a new reference to the same object. Thus if class **Square** defines a data field named **side**, and **square1** and **square2** are two variables of class **Square**, and we make the assignments

```
square1 := Square.new;
square2 := square1;
```

  then the variable-references **square1.side** and **square2.side** refer to the very same data. (For more information, see Section 6.)

- The object referenced by a class-type variable is not necessarily an instance of the class of the variable: it may be an instance of a descendant of that class (see Section 1.3).

1.3 The Class Hierarchy

There is a predefined class named **Object**. Every class except **Object** is a *subclass* of exactly one other class, which is called its *superclass*. Any class can have any number of subclasses. Thus the classes form a tree hierarchy with **Object** as its root. Figure 1 shows how the tree might look after a few classes have been declared; note that all classes in the tree are strictly hypothetical except for class **Object**.

If **XyzClass** is some class, we can trace a chain of superclasses going from the superclass of **XyzClass** to the superclass's superclass and so forth up to class **Object**. The classes in the chain are the *ancestors* of **XyzClass**, and **XyzClass** is a *descendant* of each of its ancestors.

1.4 Inheritance

A class *inherits* the field definitions of its ancestors. Thus if class **Shape** (see Figure 1) defines a field named **center**, then **center** is also a field-identifier of **Triangle** even though **Triangle** does not explicitly define such a field. The "meaning" of **center** as a field of **Triangle** is given by its declaration as a field of **Shape**.

When a method is inherited, it can be *overridden* by the inheriting class; this is explained in Section 4.1.

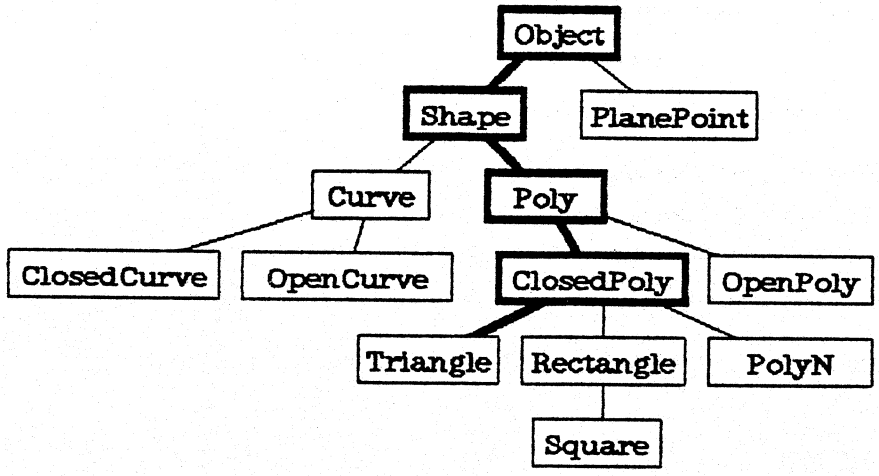


Fig. 1: A Hypothetical Class Tree.
(Dark lines show the ancestors of class **Triangle**.)

1.5 Assignment-Compatibility of Objects

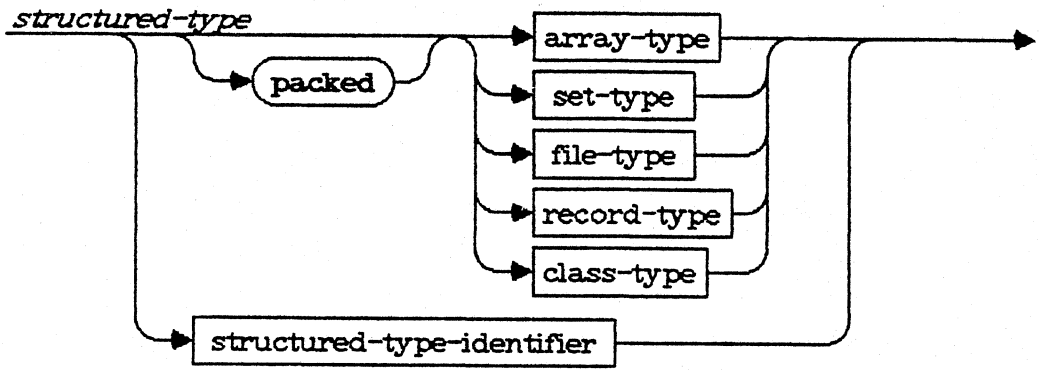
Suppose that **V** is a variable, parameter, or function-identifier of class-type **T**, and **expr** is an expression whose result is to be assigned to **V**. **V** and **expr** are assignment-compatible if either of the following is true:

- The result of **expr** is an object of class **T**.
- The result of **expr** is an object of a class descended from **T**.

2 Expansion of Existing Pascal Syntax

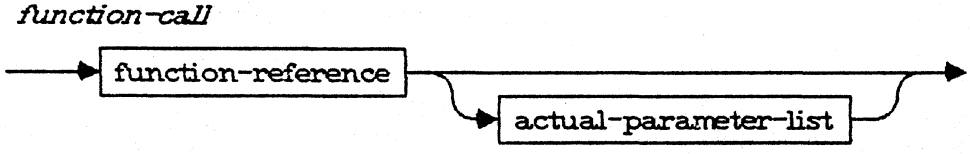
The following syntax diagrams are expansions of the conventional Pascal syntax. Note that they allow all the same constructions as the conventional syntax, and some additional constructions for Clascal.

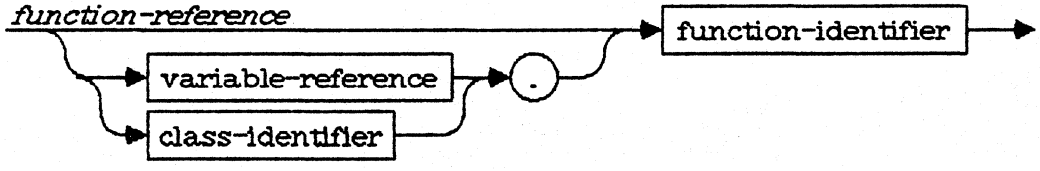
First, the syntax for a structured-type is redefined to allow a *class-type*.

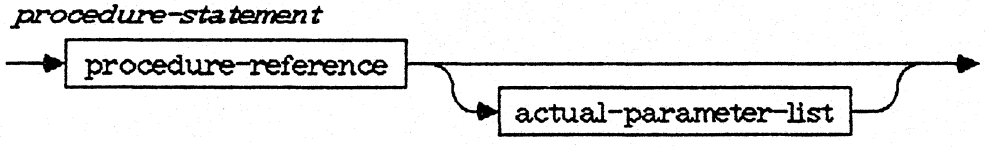


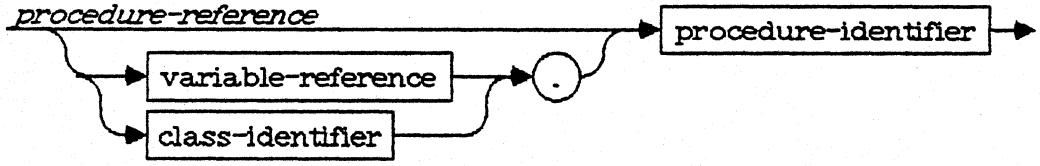
The syntax for a class-type is given in Section 3.

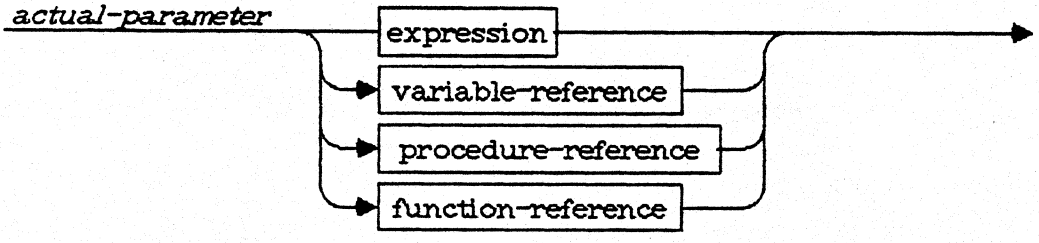
Function-calls, procedure-statements, and actual-parameters are redefined in terms of *function-references* and *procedure-references.* This allows reference to a method (procedure or function) that is defined as a field of a class.



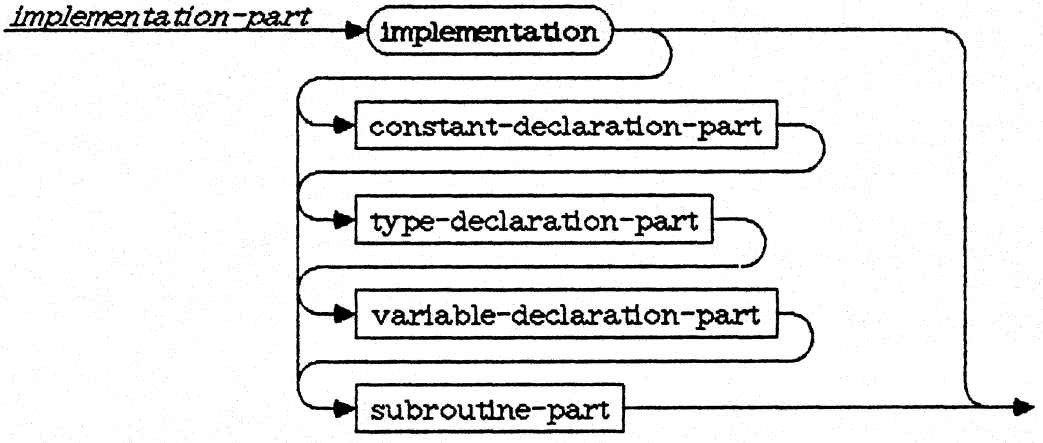


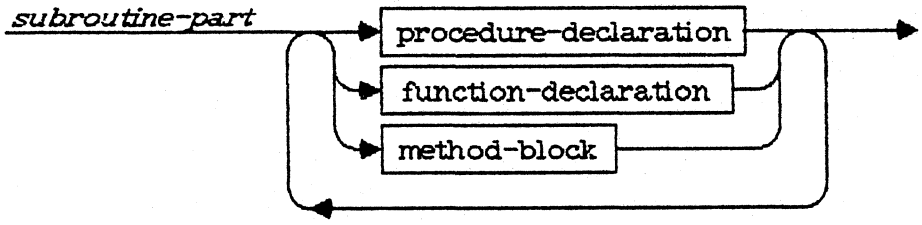






Finally, the syntax for an implementation-part is redefined by replacing the procedure-and-function-declaration-part with a more general construction called a *subroutine-part.*



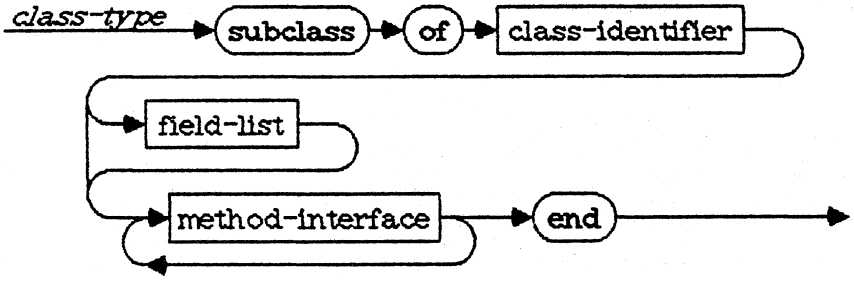


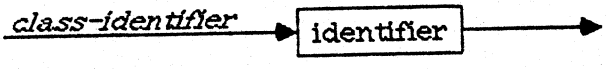
Like a procedure-and-function-declaration-part, a subroutine-part allows procedure-declarations and function-declarations. In addition, it allows *method-blocks*, which are discussed in Section 3.

3 Defining a Class

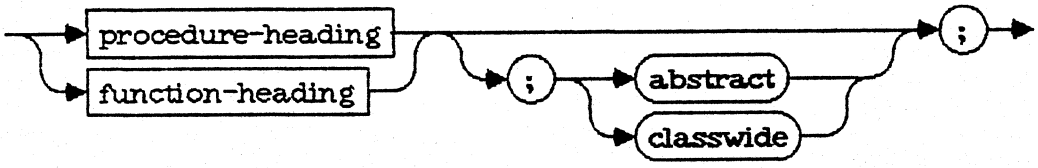
A class can only be defined within a unit (either a regular-unit or an intrinsic-unit). The specification of a class is in two parts. First the class-type itself is declared in the type-declaration-part of the unit's interface-part; then the class's methods are implemented in a method-block in the unit's implementation-part.

The syntax for a class-type is





method-interface



Example of a class-type declaration (must be in type-declaration-part within a unit's interface-part):

```
Triangle = subclass of ClosedPoly
  corner:  array[1..3] of PlanePoint;
  color:   TColor;
  {...other data fields...}
  function sides: integer; classwide;
  function new(c1, c2, c3: PlanePoint): Triangle;
  function area: real;
  procedure setCorners(c1, c2, c3: PlanePoint);
  procedure translate(vect: PlanePoint);
  procedure rotate(theta: real);
  {...other method-interfaces...}
end;
```

Notice that the class-type being declared can be referred to within its own declaration.

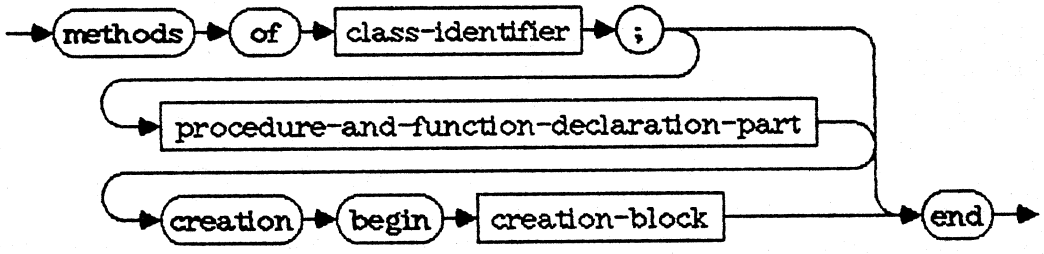
Each method-interface in the class-type defines the interface to a method of the class, i.e., the method's identifier, its formal-parameter-list (if any), and its result-type (if it is a function).

The identifiers of data fields and methods in a class-type must not conflict with those of any data fields and methods inherited from ancestor classes.

The **abstract** directive indicates a method with no implementation in this class, intended to be implemented by a subclass. The **classwide** directive indicates a method to be invoked via a reference to the class-type itself, instead of a reference to an object of the class. Abstract and classwide methods are discussed in more detail in Section 4.

For each class-type declared in the unit's interface, there is a method-block in the unit's implementation-part.

method-block



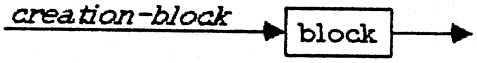
The method-block's procedure-and-function-declaration-part implements the class's non-abstract methods, including those that override inherited methods.

Each procedure or function is declared without any formal-parameter-list or result-type, since this information has already been given in the method's method-interface.

Example of a method-block (must be in the implementation-part of the unit):

```
methods of Triangle;
  function sides {: integer; classwide};
    begin sides := 3; end;
  function new {(c1, c2, c3: PlanePoint): Triangle};
    begin {code to implement new} end;
  function area {: real};
    begin {code to implement area} end;
  procedure setCorners {(c1, c2, c3: PlanePoint)};
    begin {code to implement setCorners} end;
  procedure translate {(vect: PlanePoint)};
    begin {code to implement translate} end;
  procedure rotate {(theta: real)};
    begin {code to implement rotate} end;
  {...other methods...};
end;
```

A creation-block is simply a conventional Pascal block:



If a creation-block is present, it will be executed before execution of any code in the host program. This allows initialization of the unit.

In general, there may be more than one creation-block to be executed before the host code is executed. The rule is that a creation block for a given class will not be executed until all creation-blocks declared for the class's ancestors have been executed.

4 Defining a Method

A method of a particular class is defined by two things:

- Its method-interface, which may appear in the class's class-type declaration or may be inherited from an ancestor class.
- A corresponding implementation (procedure-declaration or function-declaration) in the class's method-block.

4.1 Overriding Inherited Methods

Figure 2 (overleaf) shows how a method is looked up when it is called by referencing an object's identifier qualified with the method's identifier. Note that the search always begins in the object's own class.

If **doThis** is a method of **classL**, then any descendant of **classL** (such as **classN**) can *override* it by providing a different implementation of **doThis** in its method-block.

Note that in this case there is no corresponding method-interface in the class-type of **classN**; the overriding implementation inherits the original interface.

The original **doThis** method will still be inherited by any class that is between **classL** and **classN** in the chain (such as **classM**), but the overridden method will be inherited by descendants of **classN**. Any of these decendants can again override the method.

Note that you cannot override the method-interface of an inherited method; a compiler error will result. Likewise, you cannot override the declaration of an inherited data field.

4.2 Self-Reference via the Self Pseudovariable

Clascal provides a "pseudovariable" named **self**. When this identifier is used in a method, it refers (at execution time) to the object that was referred to in order to invoke the method. (In object-oriented programming parlance, this object said to be "executing" the method.)

This allows an object's methods to refer to the object's own fields, without knowing an identifier for the object itself.

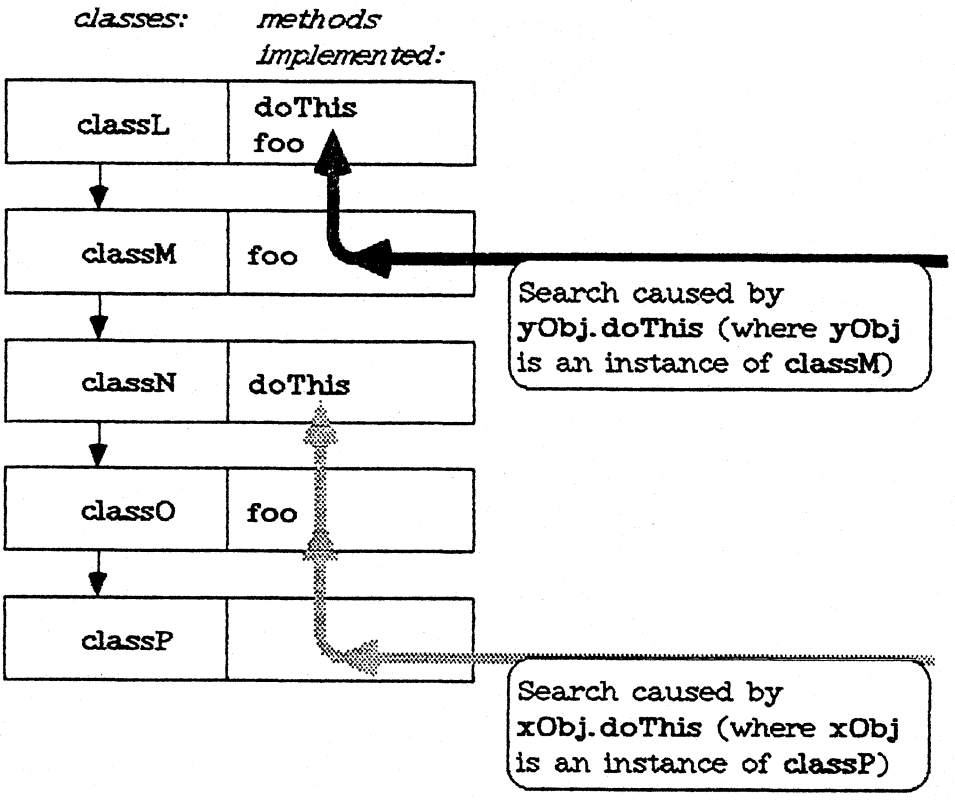


Fig. 2: Inheriting a Method
(Small arrows indicate subclassing; see listing, Appendix A.)

For example, suppose the following:

- Class **Triangle**, as indicated in Section 3, declares a method named **area** and a data field named **corner**.
- Method **area** contains a reference to **self.corner**.
- **tri1** and **tri2** are variables of type **Triangle**.

If **area** is invoked by **tri1.area**, then it will access the field **tri1.corner**. But if **area** is invoked by **tri2.area**, then it will access the field **tri2.corner**.

Another example is shown in Figure 3. An instance of **classP** inherits the **doThis** method from **classN**, and this **doThis** method contains the reference **self.foo**. This is a reference to the **foo** field of whatever object happens to be executing the **doThis** method (in the case illustrated, the object **xObj**). Thus the search for **foo** begins in **classP**,

the class of the object executing **doThis** — not in **classN**, where **doThis** is implemented.

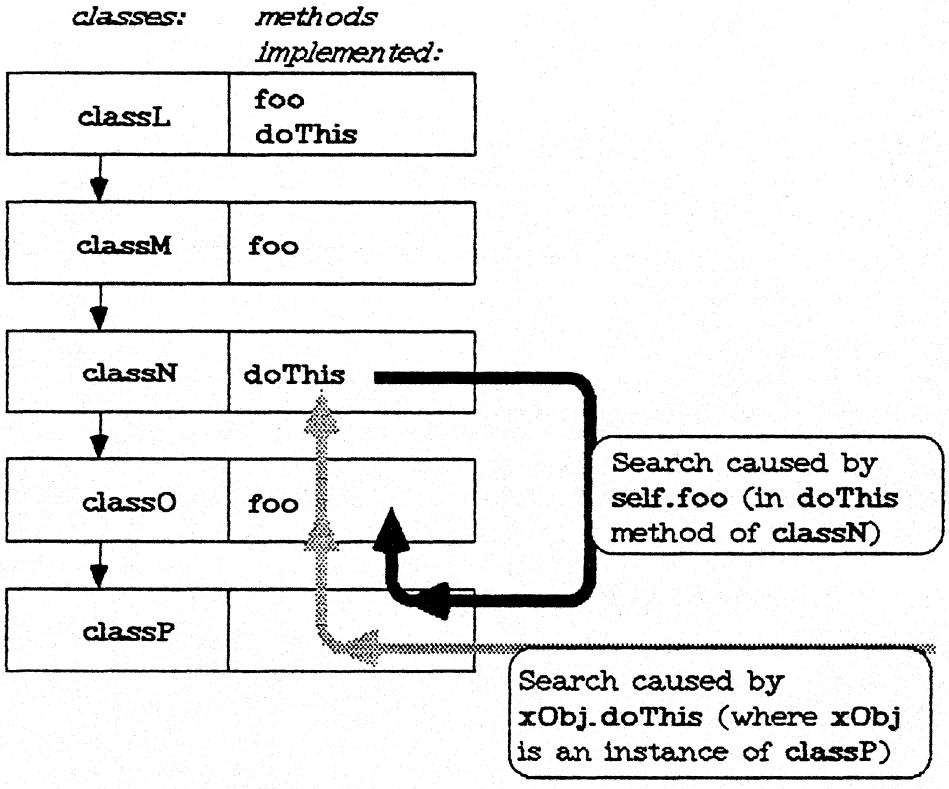


Fig. 3: Self-Reference via the Self Pseudovariable
(Small arrows indicate subclassing; see listing, Appendix A.)

Clascal does not allow you to assign anything to **self** (with one exception described in Section 4.6). However, you can make assignments to fields that are referenced via **self**. In other words, **self:=expr** is illegal but **self.color:=expr** is legal.

4.3 Self-Reference via a Class-Identifier

In addition to **self**, there is another mechanism for self-reference. A method can be referenced by using a class-identifier instead of **self**. Figure 4 (overleaf) illustrates this mechanism.

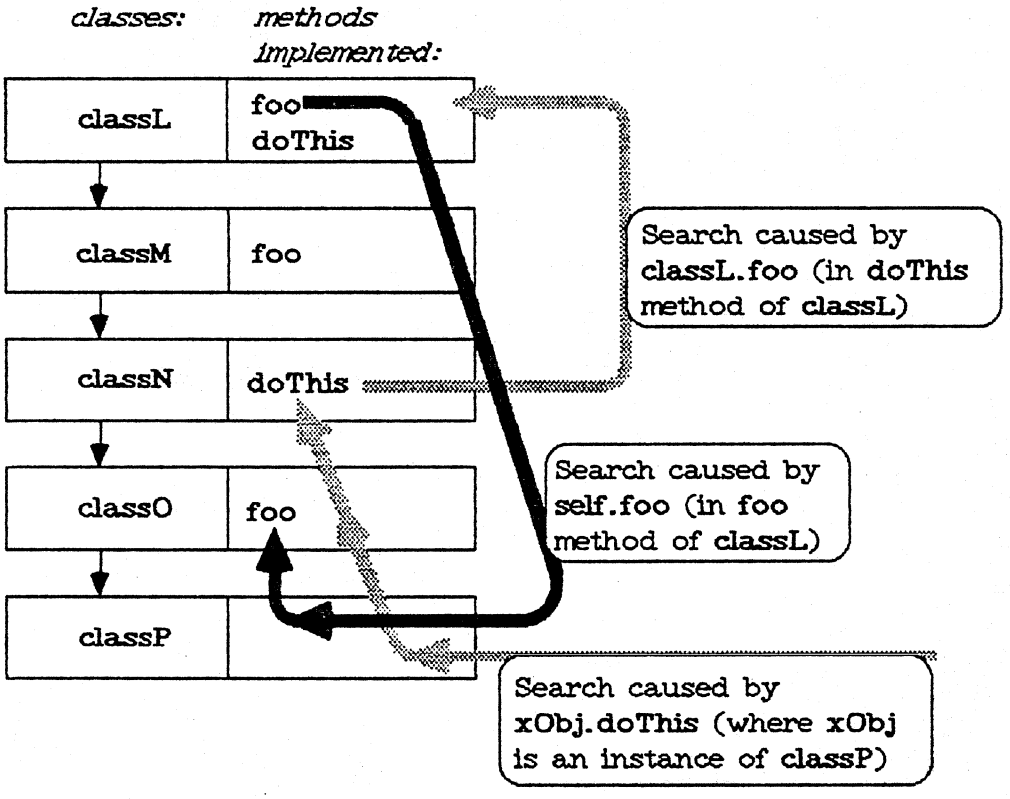


Fig. 4: Self-Reference via a Class-Identifier
(Small arrows indicate subclassing; see listing, Appendix A.)

The class-identifier must identify the class of the calling method (**classN** in the illustration) or an ancestor of the calling method's class (**classM** or **classL** in the illustration).

Note that the only difference between using a class-identifier in this way and using **self** is the following:

- In a method reference using **self**, the search for the referenced method always begins in the class of the object that is executing the calling method. This is shown for **self.foo** in Figures 3 and 4.
- In a method reference using a class-identifier, the search for the referenced method always begins in the specified class. This is shown for **classL.foo** in Figure 4.

Calling a method via a class-identifier allows a method to call another method that is overridden, as shown in Figure 4. If the **doThis** method of **classN** contained the call **self.foo**, this would invoke the **foo** method

of **classM**, which overrides the **foo** method of **classL**. The call **classL.foo**, being more explicit, avoids this.

Note that the call via a class-identifier is just as self-referential as a call via **self**. In Figure 4, the object **xObj** is executing the **doThis** method of **classN**, the **foo** method of **classL**, and the **foo** method of **class0** (since both the **classL.foo** and the **self.foo** calls are self-referential). If any of these methods accesses a data field, it will be a data field of **xObj**.

4.4 Classwide Methods

A *classwide method* is declared (in a class-type) by using the directive **classwide**. A classwide method is invoked by reference to the class itself, not by a reference to an object of the class.

For example, in the example of a class-type shown in Section 3 we have **sides** declared as a classwide method of class **Triangle**; it is a function that returns the **integer** value 3. This method is invoked by the function-reference **Triangle.sides**. If **tri** is an object of class **Triangle**, the function-reference **tri.sides** is an error.

No reference to **self** is allowed in a classwide method. There is a single exception to this rule: a **new** method is implicitly a classwide method (as explained in Section 4.6) but is allowed to refer to **self**.

4.5 Abstract Methods

An *abstract method* is declared (in a class-type) by using the directive **abstract**. An abstract method has no implementation in the class's method-block; it is intended to be overridden by descendants of the class.

A reason for declaring an abstract method is to allow other methods of the class to refer to it. For example, suppose that class **Shape** (see Figure 1) declares an abstract method named **boundary** and a non-abstract method named **inside**, which refers to **self.boundary**. The idea is that descendants of **Shape** will *inherit* **inside** and *override* **boundary**.

Suppose that class **Poly** overrides **boundary**, so that class **Square** inherits **boundary** from **Poly** and **inside** from **Shape**. If **nextSqr** is an object of class **Square**, then **nextSqr.inside** will invoke the **inside** method defined in **Shape**; when this method refers to **self.boundary**, it invokes the **boundary** method defined in **Poly**. Thus we have the useful phenomenon of a method inherited from a high level (**Shape**) invoking a method that is not concretely defined until a lower level (**Poly**).

If a class's **new** method (see Section 4.6) is abstract, the class is called an *abstract class*. An abstract class cannot have any instances, and

exists in order to define common properties to be inherited by its descendants.

4.6 The New Method

Every class-type must declare a method named **new**. Note that this means that a **new** method cannot be inherited, and that the standard Pascal **new** procedure is unavailable within the methods of any class-type.

The **new** method must be a function. It can have any desired formal-parameter-list (or none). The return-type must be the class-type within which the **new** method is being declared, as in the example of a class-type declaration shown earlier.

The **new** method is automatically a classwide method, although it is not declared with the **classwide** directive. Its purpose is to create and return a new object of the class-type in which it is declared.

Unlike other Pascal functions, the **new** method must not explicitly assign a return value to the identifier **new**. Instead, it assigns a value to **self**, and implicitly returns **self**. (This is the only case in which assignment to **self** is allowed, and the only case in which a classwide method can refer to **self**.)

The right-hand side of the assignment to **self** may be either of the following:

- An expression whose result is a *handle* for a newly created object of the class being implemented (the handle concept is discussed in Section 6)
- The constant **nil** (not just any expression with the value **nil**). This should be used in case the allocation of the object fails.

The ToolKit provides a function called **newObject**, which allocates space for an object on a heap and returns the object's handle; this function should be used in implementing **new** methods.

5 Class Object

The predefined class **Object** has no data fields or methods. To make it more useful, it can be redefined to provide a **new** method and a method for deallocating an object. The ToolKit provides such a redefinition; lacking the Toolkit, you can redefine class **Object** via the following maneuver (or something similar).

```
interface
  ...
  type  ObjectAlias = subclass of Object
          function new: ObjectAlias;
          procedure free;
        end;
        Object = ObjectAlias;
  ...
implementation
  ...
  methods of Object;
   function new {: Object};
     begin {code to implement new} end;
   procedure free;
     begin {code to implement free} end;
  end;
```

6 Objects as Handles

NOTE

To use Clascal, it is not necessary to understand the explanation given here.

Internally, a value of class-type is not really an object but a *handle* for an object.

Handles support relocatable dynamic allocation of storage. The pointer to a relocatable area is not relocatable, and the memory-management software automatically maintains it when the object is relocated. The handle points to this non-relocatable pointer, and thus does not need to be maintained. In conventional Pascal, a handle must be double-dereferenced in order to access the relocatable object. Thus if **hndl** is a handle for some relocatable variable, then **hndl^^** is a reference to the variable itself.

A declared variable of class-type is actually a pointer-type variable; a handle returned by the **new** method of the class can be assigned to it. Clascal provides automatic double-deferencing of object handles, as follows:

- A reference to a field (data or method) of a variable of class-type is automatically double-dereferenced, resulting in a reference to a field of the associated object as described in Section 1. If class

Square defines a data field called **edge**, and **aSquare** is an initialized variable of class **Square**, then **aSquare.edge** is a reference to the **edge** field of the object that **aSquare** is a handle for. (If **aSquare** were a conventional Pascal handle instead of a class-type variable, it would be necessary to write **aSquare^^.edge**.)

In other words, you can think of the variable as if it were the object itself, when referring to fields.

- However, a reference to a value of class-type, without reference to a particular field, is not dereferenced. If **square1** and **square2** are both variables of class **Square**, and **square1** has been initialized, then the assignment **square2 := square1** does not mean that a copy of the object is associated with **square2**; it means that the handle in **square1** is copied to **square2**. Thus **square1** and **square2** are now associated with the very same object, and the references **square1.edge** and **square2.edge** now refer to the same data.

7 $H+ and $H- Compiler Commands

Normally, objects are stored in heap zones and can be relocated; an object is relocated when the heap zone containing the object is compacted. As explained in Section 6, all references to objects are through *handles* (double-indirect pointers), so that the relocation is invisible to the user program.

It is important to avoid code that forms a direct reference to an object, then relocates the object, and then uses the direct reference -- which is invalid because of the relocation. For example, if **h** is an object and **h.a** references an **integer** data field of **h**, then

```
x := @h.a;    {direct pointer into the heap}
mumble;       {a procedure that might compact the heap}
x^ := 3;      {intended to store into h.a...}
```

is unsafe; if **foo** does compact the heap, the third statement will probably overwrite something that is not **h.a**.

Constructs like this one are obviously unsafe, and the programmer is responsible for avoiding them.

But because Clascal provides automatic double-dereferencing of object handles, the same problem can occur in some constructions that appear

safe. The compiler checks for these constructions if the $H+ command is in effect (the default). The unsafe constructions are

- Assigning the result of a function-call to a field of an object. *Example:*

```
h.a := foo; {h is an object, foo is a function}
```

- Calling a procedure or function in a with-statement that is controlled by a variable of class-type. *Example:*

```
with h do begin {h is an object}
  ...
  mumble; {a procedure}
  ...
end;
```

- Passing a field of an object as an actual variable parameter to a procedure or function. *Example:*

```
frob(h.a);  {h is an object, frob is a procedure that
             takes a variable parameter}
```

If you are certain that the procedure or function in one of these constructions will not compact the heap, you can turn off the compiler checking by using the $H- command.

Appendix A
Sample Listings

The following is skeleton source code for an example unit. This unit declares and implements the classes shown in Figures 2-4 of this manual.

```
unit Sample;
(* ------------------------------------------------------ *)
                        interface
(* ------------------------------------------------------ *)
    {Declare class-types for classL, classM, classN,
    classO, and classP.}
  type
        classL = subclass of object
         function new: classL; abstract;
         procedure doThis;
         procedure foo;
        end;
(* ------------------------------------------------------ *)
        classM = subclass of classL
         function new: classM; abstract;
        end;
(* ------------------------------------------------------ *)
        classN = subclass of classM
         function new: classN; abstract;
        end;
(* ------------------------------------------------------ *)
        classO = subclass of classN
         function new: classO; abstract;
        end;
(* ------------------------------------------------------ *)
        classP = subclass of classO
         function new: classP;
        end;
```

```
(* ------------------------------------------------------ *)
                        implementation
(* ------------------------------------------------------ *)

  methods of classL;
    procedure doThis;
      begin
      end;
    procedure foo;
      begin
        self.foo;
      end;
  end;
(* ------------------------------------------------------ *)
  methods of classM;
    procedure foo;
      begin
      end;
  end;
(* ------------------------------------------------------ *)
  methods of classN;
    procedure doThis;
      begin
        self.foo;
        classL.foo;
      end;
  end;
(* ------------------------------------------------------ *)
  methods of classO;
    procedure foo;
      begin
      end;
  end;
(* ------------------------------------------------------ *)
  methods of classP;
    function new;
      begin
        self := {expression to return a handle...}
      end;
  end;
(* ------------------------------------------------------ *)

end. {of unit}
```

CLASCHL COMPILER ERRORS

```
01    Super Class identifier missing.
02    Method New is not declared.
03    Subclass declaration not allowed here.
04    Method is not a procedure.
05    Method is not implemented.
06    Class is not implemented.
07    Super Class identifier is not a class.
08    Identifier is not a class.
09    'NEW' not allowed here.
10    'NEW' was expected here.
11    Illegal 'NEW' method.
12    Illegal use of Class identifier
13    $H+ h.a := p()
14    $H+ WITH h DO BEGIN ... p() END;
15    $H+ p(VAR h.a)
```