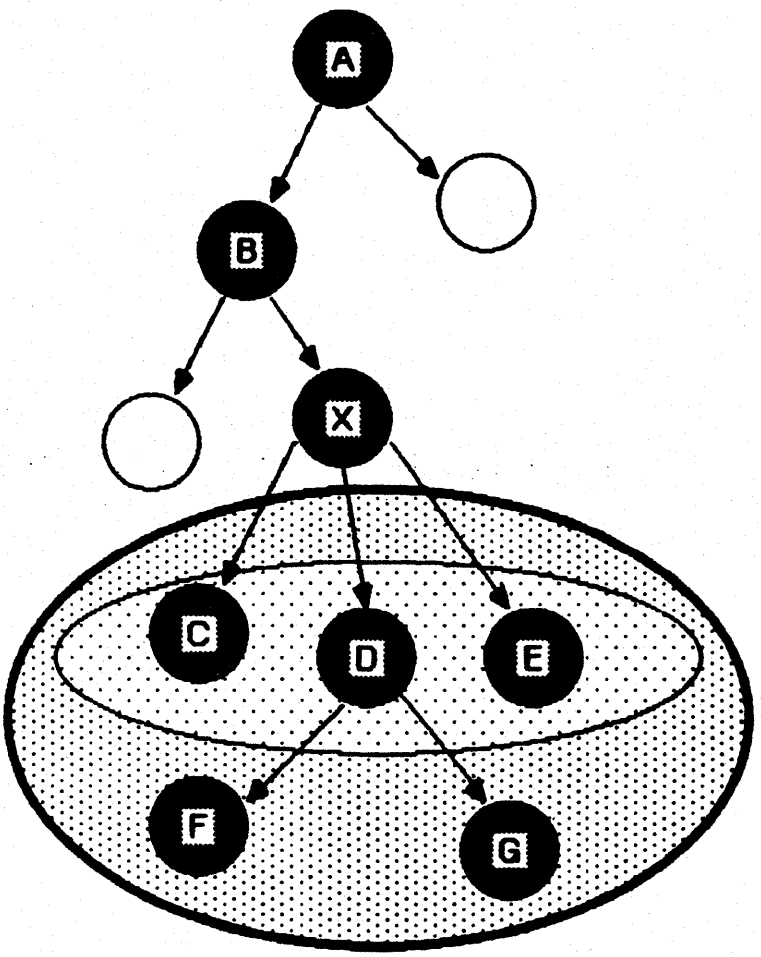
An Introduction to Clascal



Susan Keohan

Macintosh User Education

Apple Computer, Inc.

Preface

This is a conceptual introduction to Clascal, not a reference manual. Example programs and syntax diagrams follow the text.

The purpose of this document is to introduce experienced Pascal programmers to the concepts necessary to make the transition from a traditional procedure-oriented environment to Clascal's object-oriented environment.

This document assumes you are familiar with Lisa Pascal and QuickDraw.

All programs and program fragments are boldface in this document. For example, **thisShape.Erase** is a program fragment.

Table of Contents

An Introduction to Clascal

Introduction

Clascal is a set of extensions to Lisa Pascal that adds objects and classes of objects to the language. The semantic extensions of Clascal were inspired by the language Smalltalk 76. The syntactic extensions were influenced by the language Simula 67. It is not necessary to know Smalltalk or Simula before learning Clascal.

Clascal differs little from Pascal syntactically. The major difference between the languages is in programming technique.

When you write a Pascal program, your procedure and function specifications are separate from your data structure specifications. When a Pascal program is run, data is passed to a procedure or function, and the procedure or function acts on the data.

In Clascal the data structures of an object and its procedures and functions are specified together in a class declaration. When the program is run, objects are created. These objects use the functions and procedures that the programmer specified when defining the data structure. All the operations that an object can perform are defined by the object's class. As a result, program modularity is improved.

In Pascal, to add new variations to an old data type, you must either define new types incompatible with the old type, or edit existing code to add new cases to a variant record type and to procedures that act on that type. In Clascal, you can define subclasses of existing classes without editing existing code and without introducing incompatible types. As a result, program extensibility is improved.

Comparing Pascal and Clascal

Clascal is a superset of Lisa Pascal. Because the readers of this document are Pascal programmers, Clascal concepts are compared to Pascal concepts throughout this document. As a Pascal programmer, you should have little difficulty adjusting to Clascal, because most of what you know about Pascal is true for Clascal.

This document uses a Pascal program and a parallel Clascal program to give examples, and to contrast the two languages. The programs are listed under The Example Programs at the end of this document (see page 30). All examples within this document are from these programs.

The Pascal program and Clascal program, when run, do the same things. The difference between the programs is that the Clascal program, because of the structure of Clascal, can be easily extended, while the Pascal program cannot.

Pascal

PasExample, the Pascal program, is written in Lisa Pascal. When PasExample is run, in the Lisa WorkShop, it prints a command line on the top of the screen. The user chooses, from this command line, to draw and move arcs or rectangles with rounded corners. Only one shape appears on the screen at a time; when a shape is drawn or moved the old shape is erased.

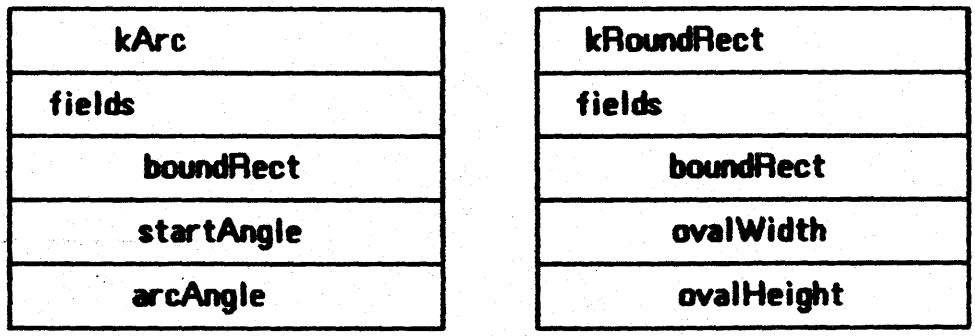
Note

PasExample uses QuickDraw to draw and erase shapes. It helps to have read the *Appendix E, QuickDraw* in the *Pascal Reference Manual* before reading this document.

In PasExample, arcs and rectangles with rounded corners are defined in a record-type definition:

```
AShape = RECORD
   boundRect: Rect;
   CASE kind: EShape of
      kArc:  (startAngle, arcAngle: INTEGER);
      kRoundRect: (ovalWidth, ovalHeight: INTEGER);
END;
```

This example is the record definition of AShape. Thus, both arcs and rounded rectangles have a boundRect field. Note that a variant record part is used to add startAngle and arcAngle to the kArc variant, and ovalWidth, and ovalHeight to the kRoundRect variant of AShape. The figure below illustrates the fields an arc or rounded rectangle have.



Data fields in kArc and kRoundRect

Aside from storage allocation procedures, PasExample defines six procedures and functions. The Run procedure controls the execution of the program. The other five procedures and functions act on arcs and rounded rectangles. The NewArc function creates an arc and the NewRoundRect function creates a rectangle with rounded corners. The DrawShape procedure draws an arc or a rectangle with rounded corners, and the EraseShape procedure erases an arc or a rectangle with rounded corners. The RandomRect procedure assigns the rectangle within which the arc or rounded rectangle is drawn.

PasExample was implemented using "handles" -- double-indirect pointers (^^) to records -- and a heap on which the records are stored. The heap is compactable. The type TShape is a handle on an AShape-type record.

Within Pascal programs, information is grouped by operations (procedures and functions). This grouping by operations scatters information about each type of data structure within a Pascal program. For example, if you want to see what can be done with arcs in PasExample, you have to look at NewArc, DrawShape, EraseShape, and RandomRect.

In PasExample, there is one Draw procedure for all shapes.

```
PROCEDURE DrawShape(SELF: TShape);
BEGIN
   CASE SELF^^.kind of
      kArc: FrameArc(SELF^^.boundRect, SELF^^.startAngle,
                     SELF^^.arcAngle);
      kRoundRect: FrameRoundRect(SELF^^.boundRect,
                     SELF^^.ovalWidth, SELF^^.ovalHeight);
   END;
END;
```

The CASE statement determines how to draw a particular shape. In PasExample, as in all Pascal programs, it would be easy to add a new operation because it is a procedure-oriented language and information is grouped by operation. It would, however, be difficult to add another variant of AShape. To do so, you have to edit the declaration of RECORD AShape in PasExample, and add to the case statements in all procedures and functions in the program. If the source code is unavailable, or if the object code is from a shared library, it is impossible to add another data structure.

Clascal

Clascal programs are structured around classes. A Clascal object is defined by its class. An object's class defines both the type of data structure the object has, and the operations (procedures and functions) that the object can perform. Classes belong to a hierarchy. This hierarchy makes it possible for classes to share characteristics belonging to classes above them in the class hierarchy. Figure 1 illustrates the basic Clascal hierarchy, and introduces some fundamental Clascal terms.

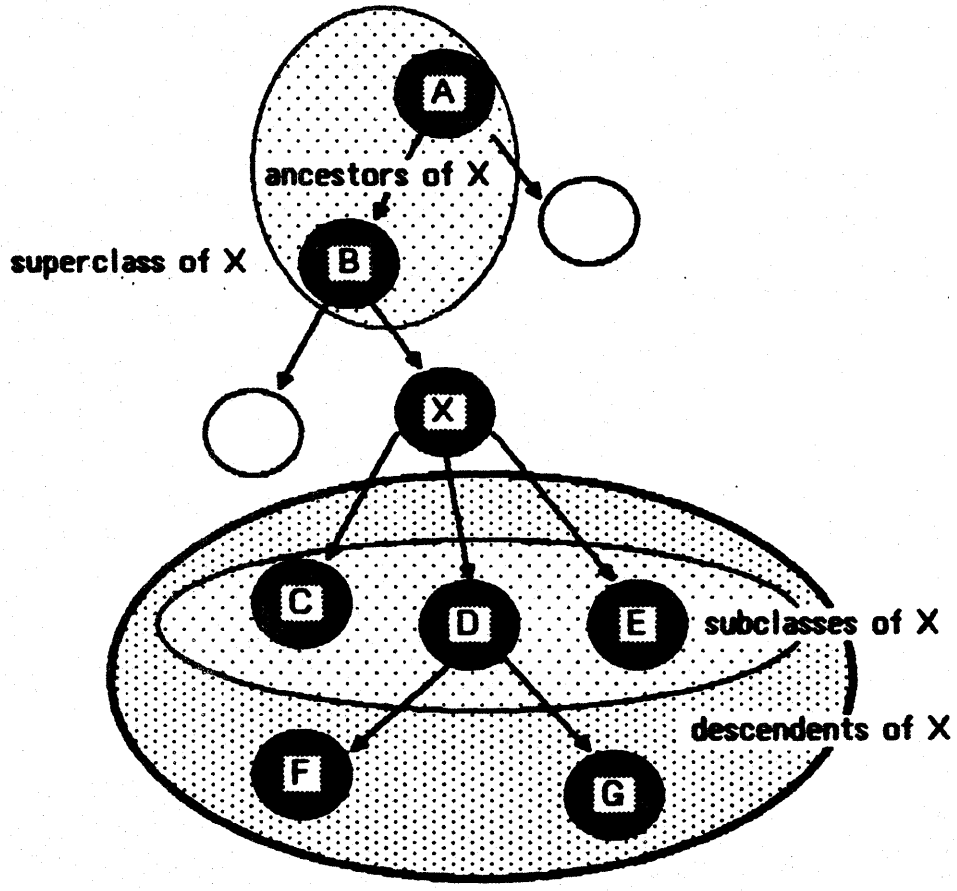


Figure 1. Clascal Hierarchy

Every circle in diagram above is a class. Within the hierarchy, classes have relationships. A *subclass* is a class that is one level below in the hierarchy -- C, D, and E are subclasses of X. A *superclass* is a class that is one level above a class in the hierarchy -- B is X's superclass. Ancestor classes are classes that are above a class on the hierarchy -- A and B are X's ancestors. Descendent classes are classes that are below a class in the hierarchy -- C, D,

E, F, and G are descendents of X. These concepts are discussed in more detail later in this document.

Objects are organized in classes. A class is a kind of data type that defines objects. A class is a *type* in the traditional Pascal sense; it is similar to a record type, but also associates operations with the class. These operations are procedures and functions. In Clascal terminology, procedures and functions are called *methods*, to indicate that they are associated with a class. An object uses the methods associated with its class. So, when you define a class, you define the data fields, and all the methods an object of that class can use.

A class type is like a Pascal record type, but more. Unlike a record type, a class has associated with it, in addition to data fields, methods (procedures and functions).

An object is an instance of a class, just as a record variable is an instance of a record type. An object is not the same as a class. A class defines the fields and methods an object of its type will have, but is not one of the objects it defines; just as a Pascal RECORD type defines a record, but is not one of the records it defines.

The standard Clascal unit, UObject, defines a class, TObject. TObject defines the most general characteristics of all Clascal objects. For example, TObject provides a general method for copying an object, and a method for discarding an object. Additional classes in a program are defined in unit UObject and other units.

A primary Clascal concept is that the data fields and methods from one class can be inherited by another class. Classes are organized in a tree-structured hierarchy, with TObject the ultimate ancestor class.

A new class is created by declaring it a subclass of another class. This establishes a place for the new class in the hierarchy. The subclass inherits the characteristics of the superclass, and new characteristics can be added.

An object of any class is an instance of its own class, and a member of all its ancestor classes. For example, all objects are members of the class TObject, as well as members of their own classes.

ClasExample has two subclasses of TObject: TShape and TControl. TShape has two subclasses: TArc and TRoundRect. Figure 2 shows the class hierarchy in ClasExample.

2/8

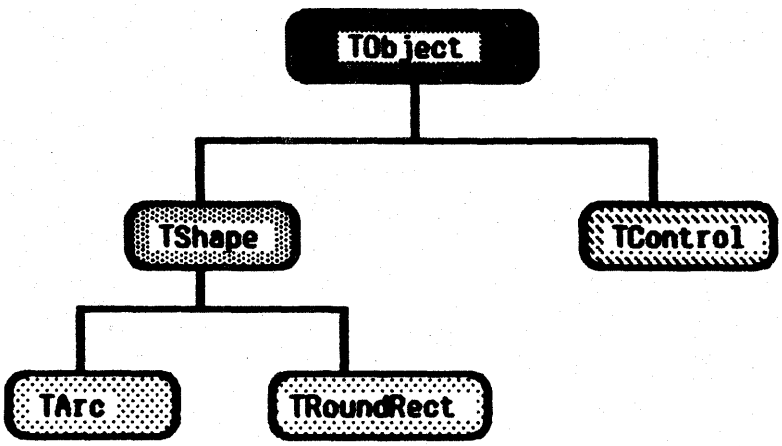


Figure 2. Class Hierarchy in ClasExample

Inherited data fields cannot be reimplemented in a subclass, but inherited methods can be reimplemented. Reimplementing a method is called "overriding a method".

For example, consider ClasExample. ClasExample draws and moves arcs and rounded rectangles. TShape defines the most general characteristics of shapes. These characteristics are a data field, boundRect and four non-CREATE methods:

- CREATE (Creates an object of class TShape)
- Draw (Draws an object of class TShape)
- Erase (Erases an object of class TShape)
- Move (Moves an object of class TShape)
- RandomRect (Initializes boundRect in an object of class TShape)

As subclasses of TShape, both TArc and TRoundRect inherit boundRect and the four non-CREATE methods defined by TShape. Note that TArc and TRoundRect inherit the definition of boundRect from TShape, but that each member of the class has its very own boundRect field. TArc adds two additional data fields, startAngle and arcAngle, overrides Draw and Erase, and adds its own CREATE function. Likewise, TRoundRect adds two additional data fields, ovalWidth and ovalHeight, overrides Draw and Erase, and adds its own CREATE function for the class. Since drawing and erasing arcs and rounded rectangles is different, TArc and TRoundRect define their own Draw and Erase methods. Initializing boundRect is the same for all shapes, so RandomRect and Move are inherited unchanged by both TArc and TRoundRect. Figure 3 illustrates this example.

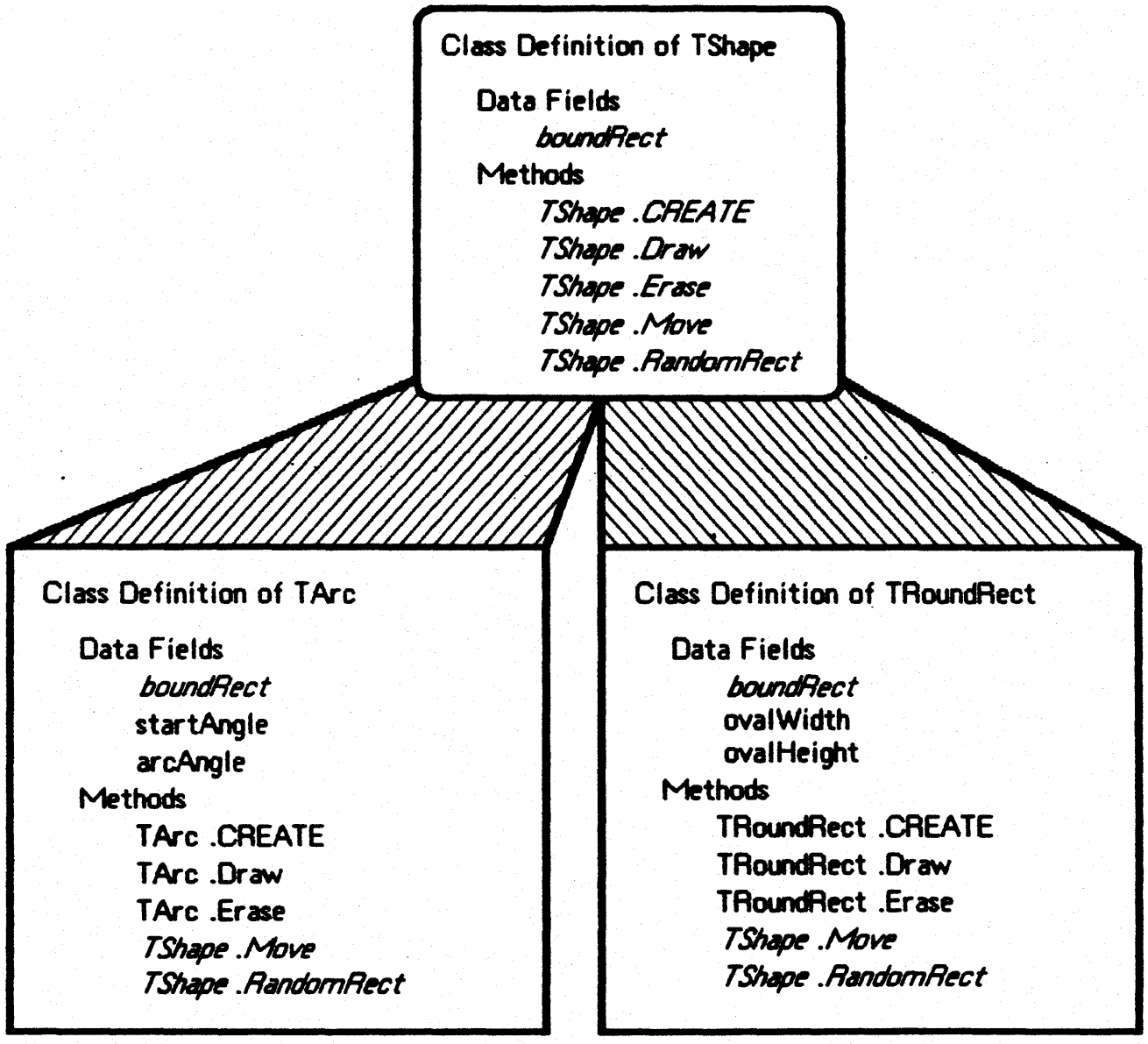


Figure 3. Inheritance in ClasExample

2/10

In Clascal, all object information is centralized within a class definition and implementation. An operation definition within a Clascal program, because it is organized within classes, is distributed throughout the program. For example, if you want to see how "Draw" is accomplished in ClasExample, you have to look in the method implementations for TShape, TRoundRect, and TArc. In PasExample, you had to look only in the one Draw procedure.

You can extend Clascal applications easily because of the class structure. In this document you will see how the class structure allows easy extension of Clascal programs, and how this ability is used by the ToolKit.

Class Types

Clascal adds a new kind of data type, *class*. A class type is like a Pascal record type, but has associated with it, in addition to data fields, methods (procedures and functions).

An *object-reference variable* is a special pointer-type variable that is used to reference an object. The type of an object-reference variable is always a class type. The value of an object-reference variable is either NIL or a reference to an object. The machine representation of an object reference is a *handle*, a pointer to a pointer to a block on the Clascal heap. In ClasExample:

```
control: TControl

thisShape: TShape
```

are object-reference variable definitions, control is of class TControl and thisShape is of class TShape.

Thus, an object-reference variable is not an object. When you refer to an object's field or a method with an object-reference variable, the handle is automatically dereferenced. This is different in syntax from Pascal. In PasExample, SELF refers to an object. To reference the arcAngle field in an arc, you would have to explicitly write

```
SELF^^.arcAngle
```

In Clascal, the double indirection is taken care of automatically. To reference the same field in an arc object, you would write

```
SELF.arcAngle
```

Objects

An object is an instance of a class. An object is stored in a block of dynamic memory on a Clascal heap. An object knows its class, and so knows which methods to use.

Data fields and associated methods make up an object. All objects that are instances of one class use the set of field names defined in the class definition.

An *object reference* is anything that refers to an object. An object reference stores a *handle* on an object. A handle is a double-indirect pointer an application uses to reference an object on the heap. A handle points to a *master pointer*, which points to the object's block on the heap.

Master pointers are maintained by the storage manager. When the space on the heap is relocated to make room for other objects, the storage manager changes the master pointer. Since the application uses the handle (which points to the master pointer), the application does not need to know about the change in the heap.

To reference a data field in an object, the format is *objectReference.variableName*. Note that no carets (^^) are used to resolve the handle -- the double indirection is implicit. In ClasExample:

```
INTERFACE

TYPE

   TArc = SUBCLASS OF TShape
      startAngle, arcAngle: INTEGER;

   END;

IMPLEMENTATION

   BEGIN

      SELF.arcAngle := randa;
```

assigns the value of randa to SELF.arcAngle. SELF is the object reference to the object which contains the field arcAngle. Note that in PasExample, SELF was declared explicitly as a variable. In ClasExample, it is predeclared by the compiler, as will be explained later in this document.

2/12

To maximize modularity, pure object-oriented programs make the restriction that fields of an object can only be accessed from the bodies of methods of that same object. In other words, all field references are of the form SELF.variableName. Access by other objects is forced to go through the method interface. For performance reasons, Clascal programs often compromise this principle, allowing read and sometimes write access by other objects. The fields that can be accessed from outside should be chosen with care, otherwise, future modifications to the program could have unexpected repercussions. Documentation of each class should indicate which fields, if any, permit external read and/or write access. Our documentation uses the letters *R* and *W* to the left of the field name for this purpose.

Figure 4 shows that object references in ClasExample are handles on objects in the heap. SELF is the object reference, ovalWidth is the field name, and that SELF.ovalWidth is a field designator that refers to the object's ovalWidth field.

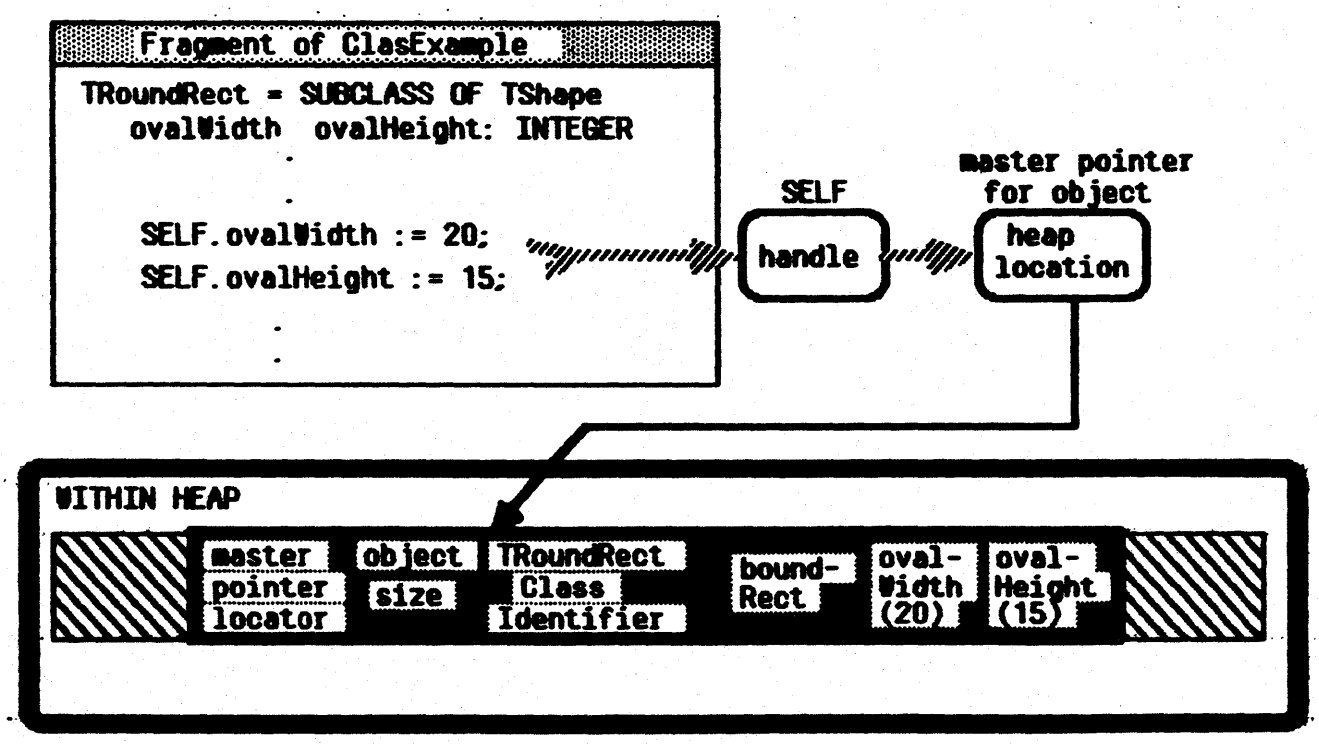


Figure 4. Object References in ClasExample.

2/13

An object can have more than one handle, but each object has only one master pointer. The master pointer is the direct link to objects on the heap. More than one object-reference variable can contain equal handles; they would all point to the same master pointer. Assigning the same handle to another object-reference variable creates a new path to the same object. Figure 5 illustrates the assignment of one object-reference variable to another in ClasExample. Note that thisShape and nextShape are object-reference variables, and so reference an object of class TShape, TArc, or TRoundRect.

Since the value of an object-reference variable is actually a handle on a block on the heap, when the value of one class-type variable is assigned to another, the two variables point to the same block on the heap. Assigning the value of one handle to another parallels assigning the value of one Pascal pointer-type variable to another.

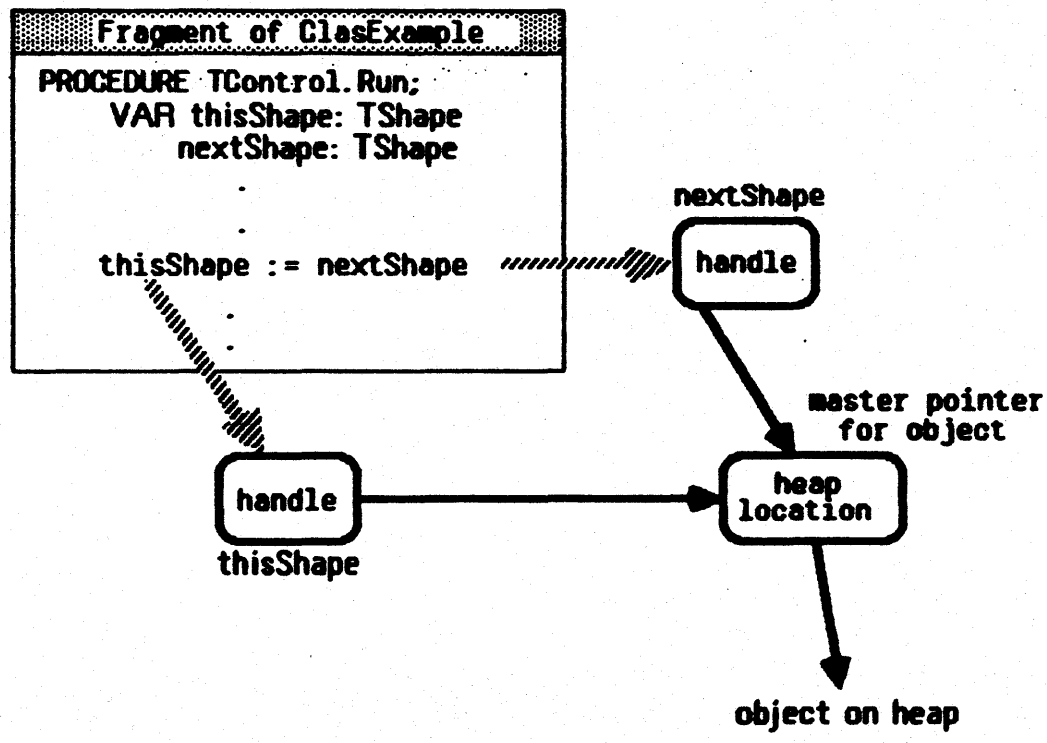


Figure 5. A New Path to the Same Object.

2/14

Methods

Methods are functions and procedures associated with a class. Methods define the operations an object can perform. All objects of the same class use the same methods.

A method declaration consists of a heading (name, arguments, and return type, if a function) and a body. The heading is specified in a SUBCLASS declaration in the TYPE section of a UNIT interface. The bodies are specified in a METHODS block in the IMPLEMENTATION section of the same unit. See ClasExample for illustrations.

An object's methods are defined in its class, and are associated with the object, as well as with all other objects of the same class. The data structure and methods together are one entity -- an object.

The association of an object with its methods makes it possible for the object to act on its own data. A *method call* tells an object to execute one of its methods. Whenever an object executes a method, the method acts on the data which that object stores. A method call is similar to a procedure or function call.

All objects of the same class respond to the same method calls. The format of a method call is *objectReference.Method(arguments).* *Method(arguments)* has the same syntax as a Pascal procedure or a function call and may, optionally, have arguments. From ClasExample

```
Program MClasExample

   .

VAR

   .

   control: TControl

   .

BEGIN

   .

control.Run;
```

control.Run is a method call with no arguments. control is an object-reference variable of type TControl. control.Run tells the object referenced by control to execute its Run method. Figure 6 illustrates these steps.

2/15

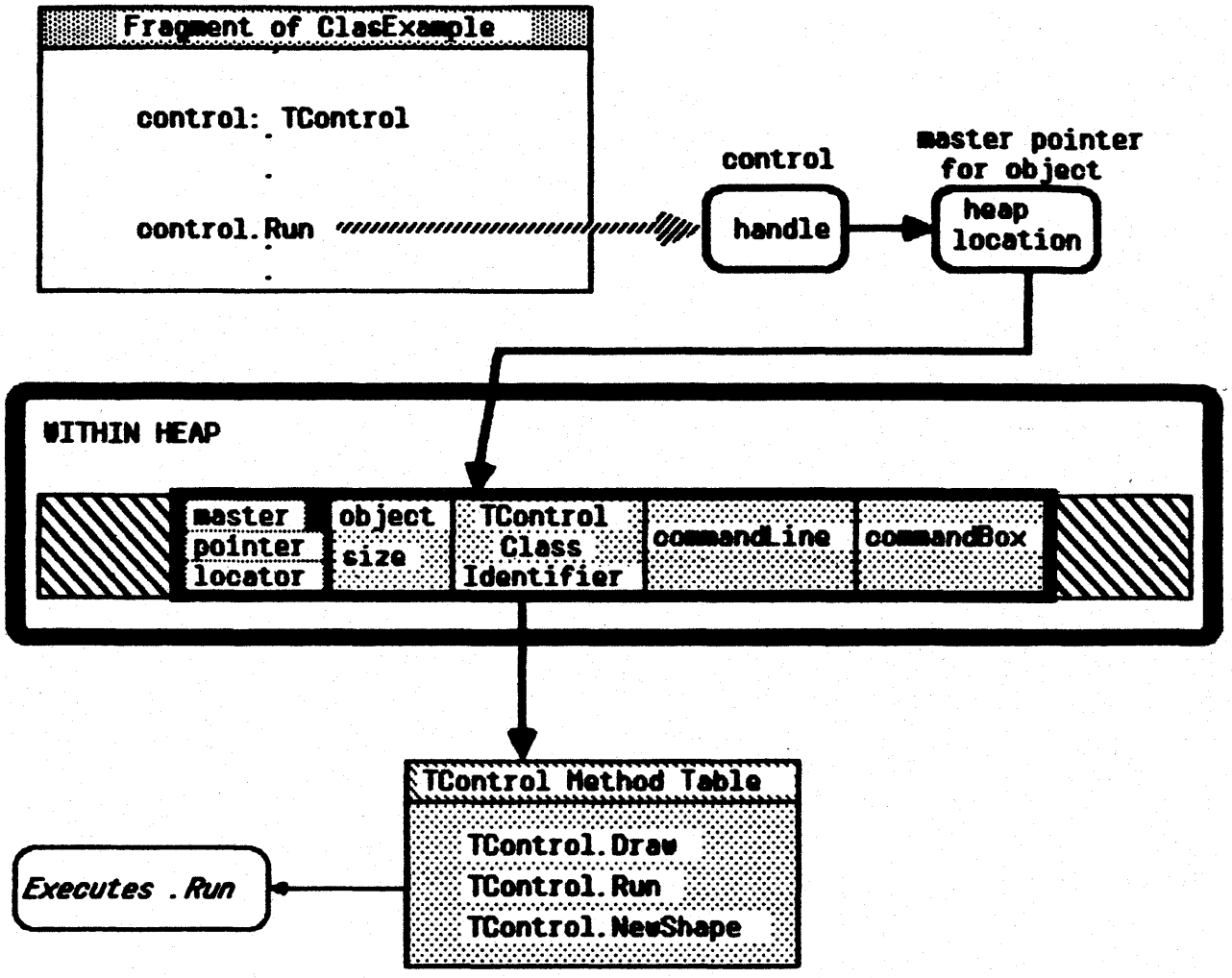


Figure 6. Method Call.

Classes and methods, combined with object references, allow you to write code with method calls upon objects, without knowing what object

2/16

will be referenced at run time. This is beneficial because it increases the extensibility of Clascal code. (Code extensibility is discussed later in this document.) For example, in ClasExample

```
IF (ch = 'a') or (ch = 'A') THEN
   thisShape := TArc.CREATE(NIL, SELF.heap)
ELSE
   thisShape := TRoundRect.CREATE(NIL, SELF.heap);
thisShape.Draw;
```

the value of thisShape is not determined until run time when the user chooses to draw an arc or a rounded rectangle. TArc and TRoundRect each have a Draw method. At run time, the correct Draw is executed.

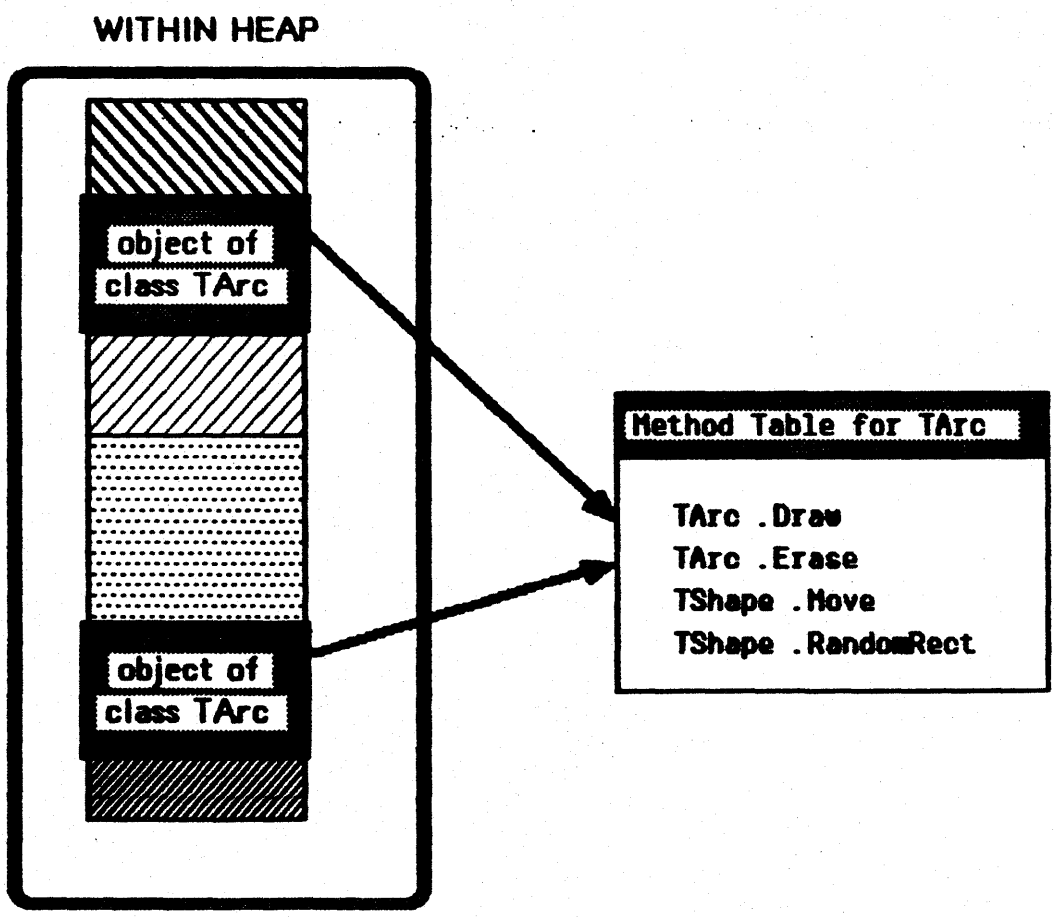


Figure 7. One Method Table per Class.

2/18

A run-time *method table* is set up for each class by the Clascal compiler. The method table for a particular class contains pointers to the methods that objects in that class use. All objects of the same class share the same method table. Figure 7 illustrates this. Notice that the method table includes methods defined specifically for the class as well as methods inherited from the superclass.

At compile time, thisShape.Draw is ambiguous. The method tables enable the correct implementation of Draw to be invoked at run time. Note that the method table does not include the CREATE method. The reason is that the call TArc.CREATE is unambiguous at compile time (see page 21).

In programming language theory, a procedure with multiple implementations is called generic. Draw is generic, with two specific implementations, TArc.Draw and TRoundRect.Draw. In languages like Ada, it is possible to determine at compile time which of several implementations a given generic procedure call will invoke. In languages like Clascal and Smalltalk, it is not possible to choose among implementations until run time. A generic procedure which can not be resolved until run time is sometimes called polymorphic.

You may have noticed that the syntax for a reference to a data field within an object and the syntax for a method call with no arguments are identical. Their syntax is:

- *objectReference.variableName* for a reference to a data field
- *objectReference.Method* for a method call

To help readers tell them apart, ToolKit typographical conventions begin variable names with a lowercase letter, whereas method names begin with an uppercase letter. Thus, control.Run is a method call, and SELF.arcAngle is a reference to a data field within an object.

SELF

SELF is an object reference. Any method can refer to SELF, and it always means *the object that is executing the method*. Every method has a SELF the compiler declares automatically. Its type is the class that contains the method. For example, in ClasExample

```
thisShape.Draw(gray);
```

calls the Draw method for the object. Within this method, SELF refers to the same object thisShape did. In other words, assuming that thisShape stores a handle on an object of class TArc, thisShape.Draw(gray) calls

```
PROCEDURE TArc.Draw(pat: pattern)
```

which behaves as if it were declared

```
PROCEDURE TArc.Draw(pat: pattern; SELF: TArc)
```

in which

```
pat := gray
SELF := thisShape
```

Note that it is illegal to declare SELF as a parameter for a method. It is automatically declared as a parameter by the compiler.

Like any other object-reference variable, SELF can be used to call a method, or to reference a data field, of an object that will be determined at run time. For example, in ClasExample, RandomRect is used to assign a value to the boundRect field of an object that is determined at run time.

An important fact to understand is that at run time, the object referenced by SELF in the method TShape.RoundRect is necessarily a member, but not necessarily an instance, of class TShape. SELF could reference an object of class TArc, TRoundRect, or any other descendent of TShape. On different calls to the method during a single program execution, the referents of SELF could be objects of different classes.

If you find SELF confusing, you can compare the RandomRect method in ClasExample with the corresponding procedure in PasExample, which does the same thing:

In PasExample:

```
PROCEDURE RandomRect(SELF: TShape);
   VAR rand1, rand2: INTEGER;
BEGIN
   rand1 := Abs(Random) MOD 600;
   SELF^^.boundRect.left := rand1;
   rand2 := Abs(Random) MOD 150;
   SELF^^.boundRect.top := rand2 + 75;
   SELF^^.boundRect.right := SELF^^.boundRect.left + 40;
   SELF^^.boundRect.bottom := SELF^^.boundRect.top + 40;
END;
```

In ClasExample, we include the class name in the heading, we omit the SELF parameter because it is implicitly declared as a TShape, and we write SELF.boundRect instead of SELF^^.boundRect:

```
PROCEDURE TShape.RandomRect;
   VAR rand1, rand2: INTEGER;
BEGIN
   rand1 := Abs(Random) MOD 600;
   SELF.boundRect.left := rand1;
   rand2 := Abs(Random) MOD 150;
   SELF.boundRect.top := rand2 + 75;
   SELF.boundRect.right := SELF.boundRect.left + 40;
   SELF.boundRect.bottom := SELF.boundRect.top + 40;
END;
```

Class Hierarchy and Inheritance

A method can be declared a default method in the interface by using the DEFAULT qualifier. The DEFAULT qualifier indicates to the compiler that it is likely that this method will be overridden in the subclasses. A DEFAULT method does not have to be reimplemented in a subclass, and a non-DEFAULT method can still be overriden. Using the DEFAULT qualifier on methods that are usually overriden saves space at run time.

The OVERRIDE qualifier must be used in the interface whenever an inherited method is reimplemented. Note that you must use OVERRIDE regardless of whether the DEFAULT qualifier was used.

Every class must declare a CREATE method. A CREATE method may never be declared either DEFAULT or OVERRIDE.

The type definitions for TShape, TArc, and TRoundRect are listed below; note the use of DEFAULT and OVERRIDE :

```
TShape = SUBCLASS OF TObject
   boundRect: Rect;
   FUNCTION  TShape.CREATE(object: TObject;
                           itsHeap: THeap):TShape;
   PROCEDURE TShape.Move;
   PROCEDURE TShape.RandomRect;
   PROCEDURE TShape.Draw(pat: pattern); DEFAULT;
   PROCEDURE TShape.Erase; DEFAULT;
TArc = SUBCLASS OF TShape
  startAngle, arcAngle: INTEGER;
  FUNCTION TArc.CREATE(object: TObject;
                           itsHeap: THeap): TArc;
  PROCEDURE TArc.Draw(pat: pattern); OVERRIDE;
  PROCEDURE TArc.Erase; OVERRIDE;
TRoundRect = SUBCLASS OF TShape
  ovalWidth, ovalHeight: INTEGER;
  FUNCTION TRoundRect.CREATE(object: TObject;
                           itsHeap: THeap): TRoundRect;
  PROCEDURE TRoundRect.Draw(pat: pattern); OVERRIDE;
  PROCEDURE TRoundRect.Erase; OVERRIDE;
```

According to the syntax of Clascal, the next-to-last line above could be abbreviated:

```
  PROCEDURE Draw; OVERRIDE;
```

i.e., the class name and dot are optional, and the argument list is also optional. When they are included, the compiler checks them for type agreement with the inherited method. Furthermore, including them improves readability and makes searches in the text editor far easier.

Figure 3 illustrates inheritance in ClasExample. TArc and TRoundRect inherit boundRect and add additional fields, startAngle and arcAngle for TArc and ovalWidth and ovalHeight for TRoundRect. TArc and TRoundRect reimplement the CREATE, Draw, and Erase methods, but the Move and RandomRect methods are inherited from TShape.

From ClasExample,

```
thisShape.Draw(gray)
```

is a method call. thisShape is declared as a variable of type TShape. This means that references to objects of classes TShape, TArc and TRoundRect can be assigned to it. In the example program, an object of type TShape is never created because TShape describes shapes in general, not particular shapes. Therefore, thisShape is a reference to an object of some subclass of TShape. As discussed above, the Draw method is implemented differently in each subclass. When the object is told to execute the Draw method, it uses the method defined for its particular subclass. Thus, if thisShape is an object of class TArc, it will execute PROCEDURE TArc.Draw. Likewise, if thisShape is an object of class TRoundRect, it will execute PROCEDURE TRoundRect.Draw.

When you create a new class, the Clascal compiler creates, in effect, a method table for that class that combines pointers to the methods from the superclass with pointers to the methods for the new class. The compiler reimplements superclass methods with the subclass' methods where applicable. Figure 8 lists the methods pointed to in the method tables for TArc and TRoundRect.

Method Table for TArc	Method Table for TRoundRect
TArc .Draw	TRoundRect .Draw
TArc .Erase	TRoundRect .Erase
TShape .Move	TShape .Move
TShape .RandomRect	TShape .RandomRect

Figure 8. Method Tables.

2/23

Assignment Checking and Typecasting

If a program declared variables arc1, arc2: TArc and variables shape1, shape2: TShape, then the following assignments would be legal and are always safe:

```
shape1 := shape2;

arc1 := arc2;

shape1 := arc1;
```

But the following assignment would be illegal, because the run-time value of shape1 could be a TRoundRect:

```
arc1 := shape1;
```

If, in the context of the assignment, you know that shape1 must be a member of class TArc, then write the following legal statement instead:

```
arc1 := TArc(shape1);
```

The construct TArc(...) is called a typecasting construct; it tells the compiler to treat the argument as if its type were TArc. When range-checking is on ({$R+}), this causes a run-time check that shape1 is indeed a member of class TArc. When range-checking is off ({$R-}), no code is generated, no checking is done, and a mismatch will cause anomalous behavior.

The following construct is legal. Constructs like it are often useful.

```
shape1 := TControl(object).NewShape(ch);
```

It is equivalent to the following three lines of code:

```
VAR control1: TControl;

...

control1 := TControl(object);

shape1 := control1 .NewShape(ch);
```

Note that NIL is a legal value of any object reference. However, an attempt to dereference NIL will cause a crash.

Creating and Freeing Objects

A new object, or instance of a class, is created by a special method, the CREATE function. A CREATE function *must* be declared in the class definition of every class. A CREATE function allocates space for an object on the heap, assigns the handle of this space to SELF, and should initialize all object fields. In ClasExample the CREATE function for TArc is:

```
FUNCTION TArc.CREATE(object: TObject; itsHeap: THeap): TArc;
   VAR rands, randa: INTEGER;
BEGIN
   IF object = NIL THEN
      object := NEWObject(itsHeap, THISCLASS);
   SELF :=TArc(TShape.CREATE(Object, itsHeap));
   rands := Abs (Random) MOD 270;
   SELF.startAngle := rands;
   randa := Abs (Random) MOD 270;
   SELF.arcAngle := randa;
END;
```

This CREATE function creates an object of class TArc.

Every class must have a CREATE method written specifically for it. Its parameters can be different from the parameters in the CREATE method of the superclass. Whenever you write a CREATE function, debugging is easier if you initialize all data fields.

Except for immediate subclasses of TObject, a CREATE function usually must initialize both the fields declared in its own class and fields inherited from ancestors. To do the latter, it can simply invoke the CREATE of the superclass. However, it would be wrong to allocate the object more or less than once. The first CREATE called must do the allocation, and the ancestors must not. By convention, the first argument of CREATE is object: TObject, and the second is itsHeap: THeap. A caller normally passes NIL as the value of object, but the CREATE of a subclass passes an already-created object to its superclass CREATE instead. In the NIL case only, CREATE is expected to allocate space for the object on itsHeap. In either case, CREATE is then supposed to initialize the fields declared by ancestors (by calling the CREATE of the next superclass up) and then to initialize the fields of its own class. See TArc.CREATE (above or in U1ClasExample) and TShape.CREATE (in U1ClasExample) for illustrations.

2/25

When an application no longer needs an object, you should remove that object from the heap, using the Free method, so the space it occupied can be reallocated. The Free method removes objects from the heap and reallocates the space on the heap. Free is a method of TObject. From ClasExample,

```
thisShape.Free
```

frees the object referenced by the handle contained in thisShape. Generally, each new subclass does not have to provide its own Free method; typically, it can just inherit it from TObject. However, if the subclass' CREATE function causes additional objects to be created, it is usually necessary for the Free method to free them, and to do so, the Free method must be overridden. The last statement of a Free method is generally SUPERSELF.Free, explained below.

SUPERSELF and Extensibility

Applications written in Clascal can be functionally extended with a minimum of difficulty because of Clascal's class structure. In Clascal, you can create a software library that does not account for an entire set of objects, just the most general. A program can then add new objects to the set in the library, without affecting the basic structure of the system, as long as the new objects have parallel methods. For example, to add a new object to ClasExample, so that it can draw and move ovals, as well as arcs and rounded rectangles, you would add a unit. The new unit would contain an additional subclass of TShape, called TOval, and a subclass of TControl, called TMyControl. This new hierarchy is shown in Figure 9.

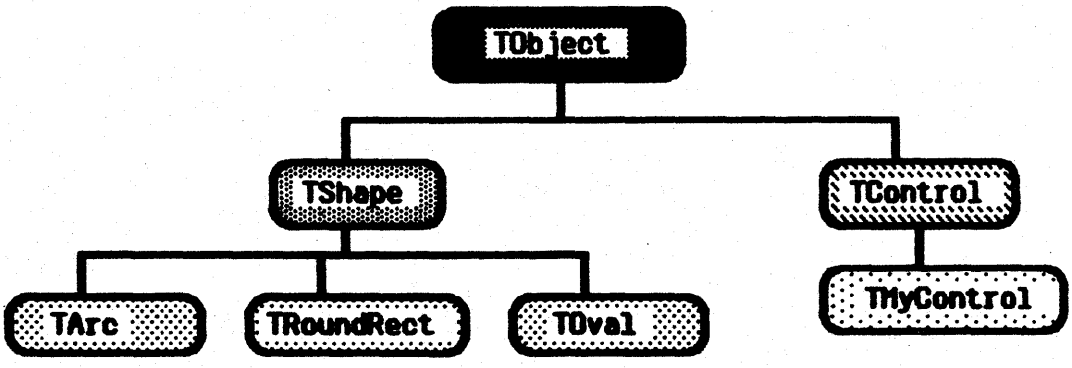


Figure 9. New Class Hierarchy in ClasExample.

2/26

In the main program, M2ClasExample, you would add the new shape to the command line. Salient parts of the new unit are listed below. The entire unit, U2ClasExample, is listed at the back of this document.

```
TYPE
 TOval = SUBCLASS OF TShape
    FUNCTION  TOval.CREATE(object: TObject; itsHeap: THeap):
                                                      TOval;
    PROCEDURE TOval.Draw(pat: pattern); OVERRIDE;
    PROCEDURE TOval.Erase; OVERRIDE;
 END;
 TMyControl = SUBCLASS of TControl
    FUNCTION  TMyControl.CREATE(object: TObject;
      itsHeap: THeap; itsLine: S255; itsBox: Rect): TMyControl;
    FUNCTION  TMyControl.NewShape(ch: CHAR): TShape; OVERRIDE;
 END;
IMPLEMENTATION
 METHODS OF TOval;
    FUNCTION  TOval.CREATE(object: TObject; itsHeap: THeap):
                                                      TOval;
    BEGIN
       IF object = NIL THEN
         object := NewObject(itsHeap, THISCLASS);
       SELF := TOval(TShape.CREATE(object, itsHeap);
    END;
    PROCEDURE TOval.Draw(pat: pattern);
    BEGIN
       FillOval(SELF.boundRect, pat);
    END;
    PROCEDURE TOval.Erase;
    BEGIN
       EraseOval(SELF.boundRect);
    END;
 END;
```


2/27

```
METHODS OF THyControl;
  FUNCTION  THyControl.CREATE(object: TObject; itsHeap: THeap;
        itsLine: S255; itsBox: Rect): THyControl;
   BEGIN
      IF object = NIL THEN
          object := NewObject(itsHeap, THISCLASS);
      SELF := THyControl(TControl.CREATE(object, itsHeap,
                itsLine, itsBox));
   END;
   FUNCTION  THyControl.NewShape(ch: CHAR): TShape;
   BEGIN
      IF (ch = 'o') OR (ch = 'O') THEN
         NewShape := TOval.CREATE(NIL, SELF.heap)
      ELSE
         NewShape := SUPERSELF.NewShape(ch);
   END;
END;
```

Note that FUNCTION THyControl.NewShape adds functionality and then calls the method from the superclass, TControl. FUNCTION THyControl.NewShape extends the functionality of the function by adding an 'o' command which creates an oval.

```
      IF (ch = 'o') OR (ch = 'O') THEN
         NewShape := TOval.CREATE(NIL, SELF.heap)
```

If the command the user enters is not an 'o', it must be one of the original commands that TControl.NewShape recognizes. The following method call is used to deal with that situation.

```
         NewShape := SUPERSELF.NewShape(ch);
```

This calls TControl.NewShape. SUPERSELF is usually used in an OVERRIDE method to allow an inherited method to be not completely overridden, but rather extended. At the point (or points) in the OVERRIDE method where you want the inherited method to be invoked, call it using SUPERSELF.

Rule

When SUPERSELF.Meth appears in the body of a method of class C, subclass of S, it tells SELF to invoke the implementation of Meth that was declared in METHODS OF S or that was inherited by S from its ancestors.

Note that a SUPERSELF call is unambiguous at compile time. The method called need not have the same name as the caller, but it usually does. The method name may not be CREATE.

The change required in MClasExample is

```
control := TControl.CREATE(NIL, mainHeap, 'R)ound Rectangle,
                    A)rc, M)ove, Q)uit', tempRect);
```

to

```
control := TMyControl.CREATE(NIL, mainHeap, 'R)ound Rectangle,
                    A)rc, O)val, M)ove, Q)uit', tempRect);
```

so that the user can choose to draw and move ovals. Note that by calling TMyControl.CREATE instead of TControl.CREATE, the control object is of class TMyControl, and adds the additional functionality to the program.

Clascal vs. Pascal -- What Does Clascal Really Do for You

Subclasses are the Clascal alternative to variant records. A subclass is a more specific definition of its superclass. In other words, the superclass defines general characteristics of a class, the subclass inherits these characteristics and adds additional data fields or methods, or reimplements inherited methods. In ClasExample, TShape contains boundRect: Rect as its only field. TArc, a subclass of TShape, contains the following fields:

```
boundRect: Rect

startAngle, arcAngle: INTEGER
```

Thus, TArc inherits boundRect from TShape and adds startAngle and arcAngle.

2/29

In Clascal, subclasses are used instead of variant record parts to handle special cases. In PasExample

```
AShape = RECORD
   boundRect: Rect;
   CASE kind: EShape of
      kArc:  (startAngle, arcAngle: INTEGER);
      kRoundRect: (ovalWidth, ovalHeight: INTEGER);
END;
```

is the record definition of AShape. Note that a variant record part is used to add the kArc variant to type AShape. Unlike the situation with variant records, in Clascal you can add a subclass like TArc and not affect the rest of your application. You can even add the subclass in a unit separate from the one in which the superclass is declared.

Clascal and the Lisa Applications ToolKit

Clascal was developed for use with the Lisa Applications ToolKit. The ToolKit is comprised of libraries of Clascal code with predefined classes that provide certain standard functions for an applications writer. The ToolKit defines the *Generic Application*. The Generic Application provides standard Lisa Application behavior. When you write a ToolKit program you add extensions to the Generic Application, by creating subclasses and methods to perform the work of your application. For more information, see The Lisa Applications ToolKit Reference Manual and the ToolKit Segments.

One of the most important features of Clascal is that method calls are used to tell an object to perform a method on itself. This means that a ToolKit library can tell one of your application's objects to invoke a method. For example, if a user pulls down a menu and selects Cut, the ToolKit will tell a selection object in your application to run a method of a certain name. You may have different selection classes for different kinds of selections (text, graphic, table row, etc.) and the appropriate cutting method will be invoked in every case.

When to Use the Clascal Extensions

As a general rule, you would use Clascal when you want to define abstract or standard behavior in one place, and concrete or special behavior in other places. The more complex a Pascal program is, the more Clascal typically saves you. You can do equivalent things with Pascal programs, but Clascal makes it much easier.

Whenever you write a program using the Lisa Applications ToolKit you must use Clascal to define the user interface portion of the program. In the rest of your program, you may use straight Pascal, if you wish, or you may define your own classes.

Advanced Topics

Squishing Production Code.

When a program has been completely debugged, you can reduce object code size by using the compiler switches {$R-} and {$D-}. The former turns off range checking for array subscripts and turns off run-time compatibility checking for class-typecasting expressions. The latter turns off emission of procedure, function, and method names into the object code and thus precludes symbolic debugging.

Abstract methods and classes.

An *abstract* method is one that is declared in a class C, but that can be implemented only in descendents of class C, not in C itself. An example is Draw in ClasExample. Note that TShape.Draw is declared DEFAULT in the interface of TShape and that it emits an error message in the implementation. After the program has been debugged and is ready to ship, one might want to reduce code size by deleting the implementation of TShape.Draw. The compiler would complain that the method is declared in the interface but not in the implementation. To suppress the error message, declare TShape.Draw to be ABSTRACT in the interface instead of DEFAULT. An abstract method may not be implemented in its own class, but only in descendent classes.

A CREATE method may be declared ABSTRACT. It has no implementation, and should not be invoked from anywhere in the program, even from a descendent's CREATE method. (Reminder: a CREATE method can not be declared DEFAULT or OVERRIDE.)

If an ABSTRACT method is invoked accidentally, it will cause a crash.

Classes like TShape that are not meant to be instantiated are called *abstract classes*. Usually--but not always--an abstract class has many abstract methods, often including CREATE. An abstract class defines an interface that is shared by all its descendent classes.

Conditional compilation is recommended for all of the above techniques, for example:

```
INTERFACE

...

PROCEDURE TShape.Draw(pat: Pattern);{$IFC myDebugFlag} DEFAULT
                                    {$ELSEC} ABSTRACT {$ENDC};

...

IMPLEMENTATION

...

{$IFC myDebugFlag}

PROCEDURE TShape.Draw(pat: Pattern);

BEGIN

   WriteLn('You can only draw in subclasses of TShape, not TShape');

END;

{$ENDC}
```

SuperduperSELF.

As explained earlier, SUPERSELF.Meth appearing in a method of class C will cause the Meth of the superclass of C to be invoked regardless of the run-time class of SELF. In the same spirit, Cls.Meth, where Cls is either C or any ancestor of C, will cause SELF to execute the Meth of Cls regardless of the run-time class of SELF. This construct is called the superduperSELF construct, because it is generally used to skip over the ancestor's implementation to the one at the next level. Be sure to use the actual class name, not "SuperduperSELF."

Suppose TShape had declared a Free method, and that TArc had overriden it, but for some reason TArc.Free wanted to invoke TObject.Free instead of TShape.Free. Instead of writing SUPERSELF.Free;--which would be equivalent to TShape.Free--one would write TObject.Free;.

The superduperself construct is rarely necessary, and is somewhat confusing, so most people avoid it whenever possible, and try to make SUPERSELF suffice.

Yanking the chair out from under you.

In TArc.CREATE, the following two lines appear:

```
rands := Abs(Random) MOD 270;
SELF.startAngle := rands;
```

One might expect to be able to write simply:

```
SELF.startAngle := Abs(Random) MOD 270;
```

However, this would generate a syntax error from the compiler alluding to "unsafe use of handle." The reason is that the code generated by the Pascal compiler leaves a pointer (not a handle) to SELF.startAngle on the stack before calling the function Random. If the function happened to cause a heap compaction, the pointer would become invalid. The compiler does not know which procedures and functions (including methods) can cause compaction. Therefore, to be safe, it assumes that all of them compact, and it always gives you an error message.

If you are sure and absolutely certain and fully convinced that the call will never allocate and never cause the heap to compact, you can tell the compiler and avoid breaking the statement into two. The way you tell it is to bracket the code section with {$H-} and {$H+}. All calls within that section will be assumed to be benign, and the error messages will be suppressed:

```
{$H-}  SELF.startAngle := Abs(Random) MOD 270; {$H+}
```

The following constructs generate error messages for similar reasons: passing an object field as a VAR parameter to a procedure or function, and calling a procedure or function within a WITH object.field block. There are also constructs that are dangerous for the same reasons but that do not generate error messages, e.g., passing @object.field as an argument to a procedure or function, calling a procedure or function in the subscript of an array-type field of an object, and doing anything mentioned in these paragraphs with a doubly-dereferenced handle instead of an object reference, e.g.:

```
h^^.fld := Random;
```

The Example Programs

A Clascal program and a Pascal program are listed in this section. These programs are referred to throughout this document. The two programs are parallel, except that the Clascal program has been extended, and the Pascal program has not.

The two programs use the ToolKit typographic and structural conventions. The recommended ToolKit structure contains a main program and units. Thus, the examples contain a main program and unit(s).

The Clascal example is comprised of

- **M1ClasExample** (The main program for the Clascal example -- before extensions)
- **M2ClasExample** (The main program for the Clascal example -- after extensions)
- **U1ClasExample** (A unit program for the Clascal example)
- **U2ClasExample** (A unit program for the Clascal example)

The Pascal example is comprised of

- **MPasExample** (The main program for the Pascal example)
- **UPasExample** (A unit program for the Pascal example)

Some other important ToolKit typographic conventions are that method names begin with a capital letter, whereas variable names begin with a lowercase letter. Thus, **boundRect** is a variable name, and **Erase** is a method name. Type names. including class names, also begin with an uppercase letter.

Within this document, **ClasExample** refers to **M1ClasExample**, **M2ClasExample**, **U1ClasExample**, and **U2ClasExample**. **PasExample** refers to **MPasExample**, **UPasExample**. Thus, an example from **ClasExample** could come from **M1ClasExample**, **M2ClasExample**, **U1ClasExample**, or **U2ClasExample**.

ClasExample and **PasExample** use QuickDraw procedures to draw shapes. These examples use QuickDraw to draw and erase shapes. It helps to have read *Appendix E, QuickDraw* in the *Pascal Reference Manual* before reading this document.

```
PROGRAM MPasExample;

USES
   {$U UObject}         UObject,
   {$U QuickDraw}       QuickDraw,
   {$U UPasExample}     UPasExample;
VAR
   error:          INTEGER;  {output parameter of FMOpen: positive if an
                             error}
   myPort:         GrafPort; {for use by QuickDraw}
   ctrlRect:       Rect;     {The area of the screen to be used for the
                             command line}
PROCEDURE FMOpen(VAR error: INTEGER); EXTERNAL;
          {EXTERNAL because not defined in any INTERFACE we can USE}

BEGIN

{In a ToolKit/32 program, UABC (the generic application) would have called
                        FMOpen & OpenPort automatically}
FMOpen(error);  {Gain access to the Lisa Font Manager; ignore errors for
                        this example}
OpenPort (@myPort);   {Provide QuickDraw with a GrafPort}

SetRect(ctrlRect, 20, 20, 500, 40);

Run('R)Round Rectangle, A)rc, M)ove, Q)uit', ctrlRect, mainHeap);
END.
```

```
UNIT UPasExample;

INTERFACE

USES
   {$U UObject}        UObject,
   {$U QuickDraw}      QuickDraw;

TYPE

   EShape = (kArc, kRoundRect);
   TCmdLine = string[80];
   TShape = ^PShape;
   PShape = ^AShape;

   AShape = RECORD
      boundRect:  Rect;
      CASE kind: EShape of
         kArc:    (startAngle, arcAngle: INTEGER);
         kRoundRect: (ovalWidth, ovalHeight: INTEGER);
      END;

PROCEDURE Run(commandLine: TCmdLine; commandBox: Rect; itsHeap:THeap);

IMPLEMENTATION
{EXTERNAL routines called by HFree; do not call these directly, instead
                        use HFree}
FUNCTION  HzFromH(h: Handle): THeap; EXTERNAL;
PROCEDURE FreeH(heap: THeap; h: Handle); EXTERNAL;
```

```
{Allocate a handle of a certain size; EXTERNAL because not defined in any
                        INTERFACE we can USE}
FUNCTION  HAllocate(heap: THeap; size: INTEGER): Handle; EXTERNAL;

{Free a handle that was allocated by HAllocate}
PROCEDURE HFree(h: Handle);
BEGIN
   FreeH(HzFromH(h), h);
END;

PROCEDURE DrawShape(pat: Pattern; SELF: TShape);
BEGIN
   CASE SELF^^.kind of
   kArc:
      FillArc(SELF^^.boundRect, SELF^^.startAngle, SELF^^.arcAngle, pat);
   kRoundRect:
      FillRoundRect(SELF^^.boundRect, SELF^^.ovalWidth, SELF^^.ovalHeight,
                        pat);
   END;
END;

PROCEDURE EraseShape(SELF: TShape);
BEGIN
   CASE SELF^^.kind of
      kArc: EraseArc(SELF^^.boundRect, SELF^^.startAngle,
                        SELF^^.arcAngle);
      kRoundRect: EraseRoundRect(SELF^^.boundRect, SELF^^.ovalWidth,
                        SELF^^.ovalHeight;
   END;
END;
```

```
PROCEDURE RandomRect(SELF: TShape);
   VAR rand1, rand2: INTEGER;
BEGIN
   rand1 := Abs(Random) MOD 600;
   SELF^^.boundRect.left := rand1;
   rand2 := Abs(Random) MOD 150;
   SELF^^.boundRect.top := rand2 + 75;
   SELF^^.boundRect.right := SELF^^.boundRect.left + 40;
   SELF^^.boundRect.bottom := SELF^^.boundRect.top + 40;
END;

FUNCTION NewArc(itsheap: THeap): TShape;
   VAR   SELF: TShape;
         rands, randa: INTEGER;
BEGIN
   SELF := POINTER(ORD(HAllocate(itsHeap, SIZEOF(AShape))));
   RandomRect(SELF);
   rands := Abs(Random) MOD 270;
   SELF^^.startAngle := rands;
   randa := Abs(Random) MOD 270;
   SELF^^.arcAngle := randa;
   SELF^^.kind := kArc;
   NewArc := SELF;
END;
```

```
FUNCTION NewRoundRect(itsheap: THeap): TShape;
   VAR SELF: TShape;
BEGIN
   SELF := POINTER(ORD(HAllocate(itsHeap, SIZEOF(AShape))));
   RandomRect(SELF);
   SELF^^.ovalWidth := 20;
   SELF^^.ovalHeight := 15;
   SELF^^.kind := kRoundRect;
   NewRoundRect:= SELF;
END;

PROCEDURE Run {commandLine: TCmdLine; commandBox: Rect; itsHeap: THeap};
   VAR   ch: char;
         shape: TShape;
         consoleFile:  TEXT; {used to allow Read(Ln) or Write(Ln) with the
                              main console}
BEGIN
   shape := NIL;
   Reset(consoleFile, '-mainconsole-dummyFileName');
   EraseRect(commandBox);
   MoveTo(commandBox.left, commandBox.bottom);
   DrawString(commandLine);
   REPEAT
      Read(consoleFile, ch);
      CASE ch OF
         'r','R','a','A':
            BEGIN
            IF shape <> NIL THEN
               BEGIN
               EraseShape(shape);
               HFree(Pointer(ORD(shape)));
               END;
```

```
            IF (ch = 'r') or (ch = 'R') THEN
               shape := NewRoundRect(itsHeap)
            ELSE
               shape := NewArc(itsHeap);
            DrawShape(gray, shape);
         END;
      'm','M':
         IF shape <> NIL THEN
            BEGIN
            EraseShape(shape);
            RandomRect(shape);
            DrawShape(gray, shape);
            END;
      END;
   UNTIL (ch = 'q') or (ch = 'Q');
END;
END.
```

```
PROGRAM M1ClasExample;   {Main Program}

USES
   {$U UObject} UObject,  {Needed to compile the U1ClasExample INTERFACE}
   {$U QuickDraw} QuickDraw,  {Needed to compile U1ClasExample INTERFACE}
   {$U U1ClasExample} U1ClasExample; {Declares classes TShape, TArc,
                                      TRoundRect, TControl}
VAR
   error:      INTEGER;  {output parameter of FMOpen: positive if an error}
   myPort:     GrafPort; {for use by QuickDraw}
   control:    TControl; {a reference (handle) to an instance of TControl}
   ctrlRect:   Rect;     {The area of the screen used for the command line}

PROCEDURE FMOpen(VAR error: INTEGER); EXTERNAL;
        {EXTERNAL because not defined in any INTERFACE we can USE}

BEGIN     {Main Program Statements}

{In a ToolKit/32 program, UABC (the generic application) would have called
                                      FMOpen & OpenPort automatically}
FMOpen(error);   {Gain access to the Lisa Font Manager; ignore errors for
                                      this example}
OpenPort (@myPort);   {Provide QuickDraw with a GrafPort}

{Create an instance of TControl to interact with the user}
SetRect(ctrlRect, 5, 20, 500, 60);
control := TControl.CREATE(NIL, mainHeap,
                    'R)ound Rectangle, A)rc, M)ove, Q)uit', ctrlRect);
{Run the command loop until the user types "Q" for Quit}
control.Run;
END.      {Main Program Statements}
```


2/41

```
UNIT U1ClasExample;

{Declares classes TShape, TArc, TRoundRect, TControl}

INTERFACE

USES
   {$U UObject} UObject, {Declares TYPE TObject and procedures such as
                                    NewObject}
   {$U QuickDraw} QuickDraw; {Declares TYPE Pattern and procedures such as
                                    FillArc}
TYPE

   TShape = SUBCLASS OF TObject
      boundRect: Rect;                    {Bounding box}

      FUNCTION  TShape.CREATE(object: TObject; itsHeap: THeap): TShape;
      PROCEDURE TShape.RandomRect; {Assign random rectangle to boundRect}
      PROCEDURE TShape.Move; {Assign new coordinates to boundRect}
      PROCEDURE TShape.Draw(pat: Pattern); DEFAULT; {The default is an
                                    error message}
      PROCEDURE TShape.Erase; DEFAULT; {The default is an error message}
   END;

   TArc = SUBCLASS OF TShape
      startAngle, arcAngle: INTEGER;  {Clockwise degrees from vertical to
                                    first radius & between radii}

      FUNCTION  TArc.CREATE(object: TObject; itsHeap: THeap): TArc;
      PROCEDURE TArc.Draw(pat: Pattern); OVERRIDE;
      PROCEDURE TArc.Erase; OVERRIDE;
   END;
```

```
TRoundRect = SUBCLASS OF TShape
   ovalWidth, ovalHeight: INTEGER; {Curvature of rounded corners}

   FUNCTION  TRoundRect.CREATE(object: TObject; itsHeap: THeap):
                                TRoundRect;
   PROCEDURE TRoundRect.Draw(pat: Pattern); OVERRIDE;
   PROCEDURE TRoundRect.Erase; OVERRIDE;
   END;

TControl = SUBCLASS OF TOBJECT
   commandLine: Str255; {Text displayed in the command line}
   commandBox: Rect; {Screen area where command line is displayed}

   FUNCTION  TControl.CREATE(object: TObject; itsHeap: THeap;
                             itsLine: Str255; itsBox: Rect):
                             TControl;
   PROCEDURE TControl.Draw; {Draw the menu}
   PROCEDURE TControl.Run; {Run the command loop}
   FUNCTION  TControl.NewShape(ch: CHAR): TShape; {Create the TShape
                             specified by user input}
   END;
```

```
IMPLEMENTATION

METHODS OF TShape;

FUNCTION  TShape.CREATE(object: TObject; itsHeap: THeap): TShape;
BEGIN
   IF object = NIL THEN  {If space is not already allocated}
      object := NewObject(itsHeap, THISCLASS);     { then allocate it}
   SELF := TShape(object); {Typecast from TObject to TShape}
   SELF.RandomRect; {Assign a random rectangle to boundRect}
END;

PROCEDURE TShape.Draw(pat: Pattern);
BEGIN
   WriteLn('You can only draw in subclasses of TShape, not TShape');
END;

PROCEDURE TShape.Erase;
BEGIN
   WriteLn('You can only erase in subclasses of TShape, not TShape');
END;

PROCEDURE TShape.Move;
BEGIN
   SELF.RandomRect;
END;
```

```
PROCEDURE TShape.RandomRect;
   VAR rand1, rand2: INTEGER;
BEGIN
   rand1 := Abs(Random) MOD 600;
   SELF.boundRect.left := rand1;
   rand2 := Abs(Random) MOD 150;
   SELF.boundRect.top := rand2 + 75;
   SELF.boundRect.right := SELF.boundRect.left + 40;
   SELF.boundRect.bottom := SELF.boundRect.top + 40;
END;

END;
```

```
METHODS OF TArc;

FUNCTION  TArc.CREATE(object: TObject; itsHeap: THeap): TArc;
   VAR rands, randa: INTEGER;
BEGIN
   IF object = NIL THEN {If space is not already allocated}
      object := NewObject(itsHeap, THISCLASS); { then allocate it}
   SELF := TArc(TShape.CREATE(object, itsHeap)); {Initialize inherited
                                        fields}
   rands := Abs(Random) MOD 270;
   SELF.startAngle := rands;
   randa := Abs(Random) MOD 270;
   SELF.arcAngle := randa;
END;

PROCEDURE TArc.Draw(pat: Pattern);
BEGIN
   FillArc(SELF.boundRect, SELF.startAngle, SELF.arcAngle, pat);
      {A QuickDraw call}
END;

PROCEDURE TArc.Erase;
BEGIN
   EraseArc(SELF.boundRect, SELF.startAngle, SELF.arcAngle);
      {A QuickDraw call}
END;

END;
```

```
METHODS OF TRoundRect;

FUNCTION  TRoundRect.CREATE(object: TObject; itsHeap: THeap): TRoundRect;
BEGIN
   IF object = NIL THEN {If space is not already allocated}
      object := NewObject(itsHeap, THISCLASS); { then allocate it}
   SELF := TRoundRect(TShape.CREATE(object, itsHeap)); {Initialize
                                        inherited fields}
   SELF.ovalWidth := 20;
   SELF.ovalHeight := 15;
END;

PROCEDURE  TRoundRect.Draw(pat: Pattern);
BEGIN
   FillRoundRect(SELF.boundRect, SELF.ovalWidth, SELF.ovalHeight, pat);
      {A QuickDraw call}
END;

PROCEDURE TRoundRect.Erase;
BEGIN
   EraseRoundRect(SELF.boundRect, SELF.ovalWidth, SELF.ovalHeight);
      {A QuickDraw call}
END;

END;
```

```
METHODS OF TControl;

FUNCTION  TControl.CREATE(object: TObject; itsHeap: THeap; itsLine:
                                  Str255; itsBox: Rect): TControl;
BEGIN
   IF object = NIL THEN {If space is not already allocated}
      object := NewObject(itsHeap, THISCLASS); { then allocate it}
   SELF := TControl(object); {Typecast from TObject to TControl}
   SELF.commandLine := itsLine;
   SELF.commandBox := itsBox;
END;

PROCEDURE  TControl.Draw;
BEGIN
   EraseRect(SELF.commandBox);
      {A QuickDraw call}
   MoveTo(SELF.commandBox.left, SELF.commandBox.bottom);
      {A QuickDraw call}
   DrawString(SELF.commandLine);
      {A QuickDraw call}
END;

PROCEDURE TControl.Run;
   VAR   ch: char;
         thisShape:    TShape;
         nextShape:    TShape;
         consoleFile:  TEXT;     {used to allow Read(Ln) or Write(Ln) with
                                    the main console}
BEGIN
   thisShape := NIL;
   nextShape := NIL;
```

```
   Reset(consoleFile, '-mainconsole-dummyFileName');
   SELF.Draw;                                  {Draw the command line}
   REPEAT
      Read(consoleFile, ch); {Accept one typed charcater from the user}
      CASE ch OF
         'm', 'M':
            IF thisShape <> NIL THEN
               BEGIN
               thisShape.Erase; {Erase the shape from its present location}
               thisShape.Move; {Assign it a new location}
               thisShape.Draw(gray); {Draw it in its new location}
               END;
            OTHERWISE
               BEGIN
               nextShape := SELF.NewShape(ch); {NIL if an unrecognized
                                    command, else a TShape}
               IF nextShape <> NIL THEN
                  BEGIN
                  IF thisShape <> NIL THEN  {Erase and deallocate any
                                    existing shape}
                     BEGIN
                     thisShape.Erase;        {Erase the old shape}
                     thisShape.Free;         {Deallocate it from the heap}
                     END;
                  thisShape := nextShape;
                  thisShape.Draw(gray);       {Draw the new shape}
                  END;
               END;
         END;
      UNTIL (ch = 'q') or (ch = 'Q');
END;
```

```
FUNCTION  TControl.NewShape(ch: CHAR): TShape;
BEGIN
   IF (ch = 'a') OR (ch = 'A') THEN
      NewShape := TArc.CREATE(NIL, SELF.heap)
   ELSE
   IF (ch = 'r') OR (ch = 'R') THEN
      NewShape := TRoundRect.CREATE(NIL, SELF.heap)
   ELSE
      NewShape := NIL;
END;

END;

END.
```

```
PROGRAM M2ClasExample;

{The only differences from M1ClasExample are:
-- Unit U2ClasExample is added to the USES chain;
-- O)val is added to the menu;
-- TMyControl is used instead of TControl.
}

USES
   {$U UObject} UObject,  {Needed to compile the U1ClasExample INTERFACE}
   {$U QuickDraw} QuickDraw,  {Needed to compile the U1ClasExample
                                 INTERFACE}
   {$U U1ClasExample}  U1ClasExample,  {Declares classes TShape, TArc,
                                 TRoundRect, TControl}
   {$U U2ClasExample}  U2ClasExample;  {Declares classes TOval,
                                 TMyControl}


VAR
   error:          INTEGER;  {output parameter of FMOpen: positive if an
                                 error}
   myPort:         GrafPort; {for use by QuickDraw}
   control:        TControl; {a reference to an instance of TControl (a
                                 handle)}
   ctrlRect:       Rect;     {the area of the screen to be used for the
                                 command line}


PROCEDURE FMOpen(VAR error: INTEGER); EXTERNAL;   {EXTERNAL because not
                                 defined in any INTERFACE we can USE}
```

```
BEGIN     {Main Program Statements}

{In a ToolKit/32 program, UABC (the generic application) would have called
                                    FMOpen & OpenPort automatically}
FMOpen(error);
OpenPort (@myPort);

{Create an instance of TControl to interact with the user}
SetRect(ctrlRect, 5, 20, 500, 60);

{Run the command loop until the user types "Q" for Quit}
control := TMyControl.CREATE(NIL, mainHeap, 'R)ound Rectangle, A)rc,
                                    O)val,
     M)ove, Q)uit', ctrlRect);
control.Run;

END.
```

```
UNIT U2ClasExample;

{Declares classes TOval (subclass of TShape), TMyControl (subclass of
                                        TControl)}

INTERFACE

USES
   {$U UObject} UObject,  {Needed to compile the U1ClasExample INTERFACE}
   {$U QuickDraw} QuickDraw,  {Declares TYPE Pattern and procedures such as
                                        FillArc}
   {$U U1ClasExample}  U1ClasExample;  {Declares classes TShape,
                                        TArc, TRoundRect,  TControl}

TYPE

   TOval = SUBCLASS OF TShape
      FUNCTION  TOval.CREATE(object: TObject; itsHeap: THeap): TOval;
      PROCEDURE TOval.Draw(pat: pattern); OVERRIDE;
      PROCEDURE TOval.Erase; OVERRIDE;
   END;

   TMyControl = SUBCLASS OF TControl
      FUNCTION  TMyControl.CREATE(object: TObject; itsHeap: THeap;
      itsLine: STR255; itsBox: Rect): TMyControl;
      FUNCTION  TMyControl.NewShape(ch: CHAR): TShape; OVERRIDE;
                                        {TControl.NewShape needn't be DEFAULT}
   END;

IMPLEMENTATION
```

```
METHODS OF TOval;

FUNCTION  TOval.CREATE(object: TObject; itsHeap: THeap): TOval;
BEGIN
   IF object = NIL THEN {If space is not already allocated}
      object := NewObject(itsHeap, THISCLASS); { then allocate it}
   SELF := TOval(TShape.CREATE(object, itsHeap)); {Initialize inherited
                                                   fields}
END;

PROCEDURE TOval.Draw(pat: pattern);
BEGIN
   FillOval(SELF.boundRect, pat);
      {A QuickDraw call}
END;

PROCEDURE TOval.Erase;
BEGIN
   EraseOval(SELF.boundRect);
      {A QuickDraw call}
END;
END;
```

```
METHODS OF TMyControl;

FUNCTION  TMyControl.CREATE(object: TObject; itsHeap: THeap;
                            itsLine: STR255; itsBox: Rect):
                            TMyControl;
BEGIN
   IF object = NIL THEN {If space is not already allocated}
      object := NewObject(itsHeap, THISCLASS); { then allocate it}
   SELF := TMyControl(TControl.CREATE(object, itsHeap, itsLine, itsBox));
                            {Init. inherited fields}
END;

FUNCTION  TMyControl.NewShape(ch: CHAR): TShape;
BEGIN
   IF (ch = 'o') OR (ch = 'O') THEN
      NewShape := TOval.CREATE(NIL, SELF.heap)
   ELSE
      NewShape := SUPERSELF.NewShape(ch); {Calls TControl.NewShape
                            regardless of SELF's class}
END;


END;


END.
```