beginner

COLLABORATORS

	TITLE : beginner	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 24, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 beginner **1**

1.1 Object Oriented E . . . 1
1.2 OOP Introduction . . . 1
1.3 Classes and methods . . . 2
1.4 Example class . . . 2
1.5 Inheritance . . . 3
1.6 Objects in E . . . 3
1.7 Methods in E . . . 4
1.8 Inheritance in E . . . 8
1.9 Data-Hiding in E . . . 14

Chapter 1

beginner

1.1 Object Oriented E

```
Object Oriented E
*****************

   The Object Oriented Programming (OOP) aspects of E are covered in this
chapter.  Don't worry if you don't know the OOP buzz words like 'object',
'method' and 'inheritance': these terms are explained in the OOP
introduction, below.  (For some reason, computer science uses strange
words to cloak simple concepts in secrecy.)


  OOP Introduction
  Objects in E
  Methods in E
  Inheritance in E
  Data-Hiding in E
```

1.2 OOP Introduction

```
OOP Introduction
================

   'Object Oriented Programming' is the name given to a collection of
programming techniques that are meant to speed up development and ease
maintenance of large programs.  These techniques have been around for a
long time, but it is only recently that languages that explicitly support
them have become popular.  You do not need to use a language that supports
OOP to program in an Object Oriented way; it's just a bit simpler if you
do!


  Classes and methods
  Example class
  Inheritance
```

1.3 Classes and methods

Classes and methods

The heart of OOP is the 'Black Box' approach to programming. The kind of black box in question is one where the contents are unknown but there is a number of wires on the outside which give you some way of interacting with the stuff on the inside. The black boxes of OOP are actually collections of data (just like the idea of variables that we've already met) and they are called objects (this is the general term, which is, coincidentally, connected with the OBJECT type in E). Objects can be grouped together in classes, like the types for variables, except that a class also defines what different kinds of wires protrude from the black box. This extra bit (the wires) is known as the interface to the object, and is made up of a number of methods (so a method is analogous to a wire). Each method is actually just like a procedure. With a real black box, the wires are the only way of interacting with the box, so the methods of an object ought to be the only way of creating and using the object. Of course, the methods themselves normally need to know the internal workings of the object, just like the way the wires are normally connected to something inside the black box.

There are two special kinds of methods: constructors and destructors. A constructor is a method which is used to initialise the data in an object, and a class may have several different constructors (allowing for different kinds of initialisation) or it may have none if no special initialisation is necessary. Constructors are normally used to allocate the resources (such as memory) that an object needs. The deallocation of such resources is done by the destructor, of which there is at most one for each class.

Protecting the contents of an object in the 'black box' way is known as data-hiding (the data in the object is visible only to its methods), and only allowing the contents of an object to be manipulated via its interface is known as data abstraction. By using this approach, only the methods know the structure of the data in an object and so this structure can be changed without affecting the whole of a program: only the methods would potentially need recoding. As you might be able to tell, this simplifies maintenance quite considerably.

1.4 Example class

Example class

A good example of a class is the mathematical notion of a set (of integers). A particular object from this class would represent a particular set of integers. The interface for the class would probably include the following methods:

1. Add -- adds an integer to a set object.

2. Member -- tests for membership of an integer in a set object.

3. Empty -- tests for emptiness of a set object.

4. Union -- unions a set object with a set object.

A more complete class would also contain methods for removing elements, intersecting sets etc. The important thing to notice is that to use this class you need to know only how to use the methods. The black box approach means that we don't (and shouldn't) know how the set class is actually implemented, i.e., how data is structured within a set object. Only the methods themselves need to know how to manipulate the data that represents a set object.

The benefit of OOP comes when you actually use the classes, so suppose you implement this set class and then use it in your code for some database program. If you found that the set implementation was a bit inefficient (in terms of memory or speed), then, since you programmed in this OOP way, you wouldn't have to recode the whole database program, just the set class! You can change the way the set data is structured in an object as much and as often as you like, so long as each implementation has the same interface (and gives the same results!).

1.5 Inheritance

Inheritance

The remaining OOP concept of interest is inheritance. This is a grand name for a way of building on classes that enables the derived (i.e., bigger) class to be used as if its objects were really members of the inherited, or base, class. For example, suppose class D were derived from class B, so D is the derived class and B is the base class. In this case, class D inherits the data structure of class B, and may add extra data to it. It also inherits all the methods of class B, and objects of class D may be treated as if they were really objects of class B.

Of course, an inherited method cannot affect the extra data in class D, only the inherited data. To affect the extra data, class D can have extra methods defined, or it can make new definitions for the inherited methods. The latter approach is only really useful if the new definition of an inherited method is pretty similar to the inherited method, differing only in how it affects the extra data in class D. This overriding of methods does not affect the methods in class B (nor those of other classes derived from B), but only those in class D and the classes derived from D.

1.6 Objects in E

```
Objects in E
============
```

Classes are defined using OBJECT in the same way that we've seen before (see OBJECT Type). So, in E, the terms 'object declaration' and 'class' may be used interchangeably. However, referring to an OBJECT type as a 'class' signals the presence of methods in an object.

The following example OBJECT is the basis of a set class, as described above (see Example class). This set implementation is going to be quite simple and it will be limited to a maximum of 100 elements.

```
OBJECT set
  elts[100]:ARRAY OF LONG
  size
ENDOBJECT
```

Currently, the only way to allocate an OOP object is to use NEW with an appropriately typed pointer. The following sections of code all allocate memory for the data of set, but only the last one allocates an OOP set object. Each one may use and access the set data, but only the last one may call the methods of set.

```
DEF s:set

DEF s:PTR TO set
s:=NewR(SIZEOF set)

DEF s:PTR TO set
s:=NEW s
```

OOP objects can, of course, be deallocated using END, in which case the destructor for the corresponding class is also called. Leaving an OOP object to be deallocated automatically at the end of the program is not quite as safe as normal, since in this case the destructor will not be called. Also, when using END to deallocate an object you do not need to use a pointer of exactly the same type as the object (like you would for normal NEW allocations). Instead you can use a pointer of any of the base classes' types. Constructors and destructors are described in more detail below.

1.7 Methods in E

```
Methods in E
============
```

The methods of E are very similar to normal procedures, but there is one, big difference: a method is part of a class, so must somehow be identified with the other parts of the class. In E this identification is done by relating all methods to the corresponding OBJECT type for the class, using the OF keyword after the description of the method's parameters. So, the methods of the simple set class would be defined as outlined below (of course, these examples have omitted the code of

methods).

```
PROC add(x) OF set
  /* code for add method */
ENDPROC

PROC member(x) OF set
  /* code for member method */
ENDPROC

PROC empty() OF set
  /* code for empty method */
ENDPROC

PROC union(s:PTR TO set) OF set
  /* code for union method */
ENDPROC
```

At first sight it might seem that the particular set object which would be manipulated by these methods is missing from the parameters. For instance, it appears that the empty method should need an extra PTR TO set parameter, and that would be the set object it tested for emptiness. However, methods are called in a slightly different way to normal procedures. A method is a part of a class, and is called in a similar way to accessing the data elements of the class. That is, the method is selected using . and acts (implicitly) on the object from which it was selected. The following example shows the allocation of a set object and the use of some of the above methods.

```
DEF s:PTR TO set
NEW s  -> Allocate an OOP object
s.add(17)
s.add(-34)
IF s.empty()
  WriteF('Error: the set s should not be empty!\n')
ELSE
  WriteF('OK: not empty\n')
ENDIF
IF s.member(0)
  WriteF('Error: how did 0 get in there?\n')
ELSE
  WriteF('OK: 0 is not a member\n')
ENDIF
IF s.member(-34)
  WriteF('OK: -34 is a member\n')
ELSE
  WriteF('Error: where has -34 gone?\n')
ENDIF
END s  -> Finished with s now
```

This is why the methods do not take that extra PTR TO set argument. If a method is called then it has been selected from an appropriate object, and so this must be the object which it affects. The slightly complicated method is union which adds another set object by unioning it. In this case, the argument to the method is a PTR TO set, but this is the set to be added, not the set which is being expanded.

So, how do you refer to the object which is being affected? In other words, how do you affect it? Well, this is the remaining difference from normal procedures: every method has a special local variable, self, which is of type PTR TO class and is initialised to point to the object from which the method was selected. Using this variable, the data and methods of object can be accessed and used as normal. For instance, the empty method has a self local variable of type PTR TO set, and can be defined as below:

```
PROC empty() OF set IS self.size=0
```

Constructors are simply methods which initialise the data of an object. For this reason they should normally be called only when the object is allocated. The NEW operator allows OOP objects to call a constructor at the point at which they are allocated, to make this easier and more explicit. The constructor will be called after NEW has allocated the memory for the object. It is wise to give constructors suggestive names like create and copy, or the same name as the class. The following constructors might be defined for the set class:

```
/* Create empty set */
PROC create() OF set
  self.size=0
ENDPROC

/* Copy existing set */
PROC copy(oldset:PTR TO set) OF set
  DEF i
  FOR i:=0 TO oldset.size-1
    self.elements[i]:=oldset.elements[i]
  ENDFOR
  self.size:=oldset.size
ENDPROC
```

They would be used as in the code below. Notice that the create constructor is, in this case, redundant since NEW will initialise the data elements to zero. If NEW does sufficient initialisation then you do not have to define any constructors, and even if you do have constructors you don't have to use them when allocating objects.

```
  DEF s:PTR TO set, t:PTR TO set, u:PTR TO set
  NEW s.create()
  IF s.empty THEN WriteF('s is empty\n')
  END s
  NEW t  /* This happens to be the same as using create */
  IF t.empty THEN WriteF('t is empty\n')
  t.add(10)
  NEW u.copy(t)
  IF u.member(10) THEN WriteF('10 is in u\n')
  END t, u
```

For each class there is at most one destructor, and this is responsible for clearing up and deallocating resources. If one is needed then it must be called end, and (as this might suggest) it is called automatically when an OOP object is deallocated using END. So, for OOP objects with a destructor, the (roughly) equivalent code to END using Dispose is a bit different. Take care to note that the destructor is not

called if END is not used to deallocate an OOP object (i.e., if deallocation is left to be done automatically at the end of the program).

```
END p

IF p
  p.end()  -> Call destructor
  Dispose(p)
  p:=NIL
ENDIF
```

The simple implementation of the set class needs no destructor. If, however, the elements data were a pointer (to LONG), and the array were allocated based on some size parameter to a constructor, then a destructor would be useful. In this case the set class would also need a maxsize data element, which records the maximum, allocated size of the elements array.

```
OBJECT set
  elements:PTR TO LONG
  size
  maxsize
ENDOBJECT

PROC create(sz=100) OF set  -> Default to 100
  DEF p:PTR TO LONG
  self.maxsize:=IF (sz>0) AND (sz<100000) THEN sz ELSE 100
  self.elements:=NEW p[self.maxsize]
ENDPROC

PROC end() OF set
  DEF p:PTR TO LONG
  IF self.maxsize=0
    WriteF('Error: did not create() the set\n')
  ELSE
    p:=self.elements
    END p[self.maxsize]
  ENDIF
ENDPROC
```

Without the destructor end, the memory allocated for elements would not be deallocated when END is used, although it would get deallocated at the end of the program (in this case). However, if AllocMem were used instead of NEW to allocate the array, then the memory would have to be deallocated using FreeMem, and this would best be done in the destructor, as above. (The memory would not be deallocated automatically at the end of the program if AllocMem is used.) Another solution to this kind of problem would be to have a special method which called FreeMem, and to remember to call this method just before deallocating one of these objects, so you can see that the interaction of END with destructors is quite useful.

Already, the above re-definition of set begins to show the power of OOP. The actual implementation of the set class is very different, but the interface can remain the same. The code for the methods would need to change to take into account the new maxsize element (where before the fixed size of 100 was used), and also to deal with the possibility the create constructor had not been used (in which case elements would be NIL

and maxsize zero). But the code which used the set class would not need to change, except maybe to allocate more sensibly sized sets!

Yet another, different implementation of a set was outlined above (see Binary Trees). In fact, remarkably few changes would be needed to convert the code from that section into another implementation of the set class. The new_set procedure is like a set constructor which initialises the set to be a singleton (i.e., to contain one element), and the add procedure is just like the add method of the set class. The only slight problem is that empty sets are not modelled by the binary tree implementation, so it wouldn't, as it stands, be a complete implementation. It would be straight-forward (but unduly complicated at this point) to add support for empty sets to this particular implementation.

1.8 Inheritance in E

Inheritance in E
==========

One class is derived from another using the OF keyword in the definition of the derived class OBJECT, in a similar way that OF is used with methods. For instance, the following code shows how to define the class d to be derived from class b. The class b is then said to be inherited by the class d.

```
OBJECT b
  b_data
ENDOBJECT

OBJECT d OF b
  extra_d_data
ENDOBJECT
```

The names b and d have been chosen to be somewhat suggestive, since the class which is inherited (i.e., b) is known as the base class, whilst the inheriting class (i.e., d) is known as the derived class.

The definition of d is the same as the following definition of duff, except for one major difference: with the above derivation the methods of b are also inherited by d and they become methods of class d. The definition of duff relates it in no way to b, except at best accidentally (since any changes to b do not affect duff, whereas they would affect d).

```
OBJECT duff
  b_data
  extra_d_data
ENDOBJECT
```

One property of this derivation applies to the data records built by OBJECT as well as the OOP classes. The data records of type d or duff may be used wherever a data record of type b were required (e.g., the argument to some procedure), and they are, in fact, indistinguishable from records of type b. Although, if the definition of b were changed (e.g., by changing the name of the b_data element) then data records of type duff

would not be usable in this way, but those of type d still would. Therefore, it is wise to use inheritance to show the relationships between classes or data of OBJECT types. The following example shows how procedure print_b_data can validly be called in several ways, given the definitions of b, d and duff above.

```
PROC print_b_data(p:PTR TO b)
  WriteF('b_data = \d\n', p.b_data)
ENDPROC

PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d, p_duff:PTR TO duff
  NEW p_b, p_d, p_duff
  p_b.b_data:=11
  p_d.b_data:=-3
  p_duff.b_data:=27
  WriteF('Printing p_b: ')
  print_b_data(p_b)
  WriteF('Printing p_d: ')
  print_b_data(p_d)
  WriteF('Printing p_duff: ')
  print_b_data(p_duff)
ENDPROC
```

So far, no methods have been defined for b, which means that it is just an OBJECT type. The procedure print_b_data suggests a useful method of b, which will be called print.

```
PROC print() OF b
  WriteF('b_data = \d\n', self.b_data)
ENDPROC
```

This definition would also define a print method for d, since d is derived from b and it inherits all the methods of b. However, duff would, of course, still be just an OBJECT type, although it could have a similar print method explicitly defined for it. If b has any methods defined for it (i.e., if it is a class) then data records of type duff cannot be used as if they were objects of the class b, and it is not safe to try! In this case, only objects of derived class d can be used in this manner. (If b is a class then d is a class, due to inheritance.)

```
PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d, p_duff:PTR TO duff
  NEW p_b, p_d, p_duff
  p_b.b_data:=11
  p_d.b_data:=-3;   p_d.extra_d_data:=3
  p_duff.b_data:=7; p_duff.extra_d_data:=-7
  WriteF('Printing p_b: ')
/* b explicitly has print method  */
  p_b.print()
  WriteF('Printing p_d: ')
/* d inherits print method from b */
  p_d.print()
  WriteF('No print method for p_duff\n')
/* Do not try to print p_duff in this way */
/*  p_duff.print() */
ENDPROC
```

Unfortunately, the print method inherited by d only prints the b_data element (since it is really a method of b, so cannot access the extra data added in d). However, any inherited method can be overridden by defining it again, this time for the derived class.

```
PROC print() OF d
  WriteF('extra_d_data = \d, ', self.extra_d_data)
  WriteF('b_data = \d\n', self.b_data)
ENDPROC
```

With this extra definition, the same main procedure above would now print all the data of d, but only the b_data element of b. This is because the new definition of print affects only class d (and classes derived from d).

Inherited methods are often overridden just to add extra functionality, as in the case above where we wanted the extra data to be printed as well as the data derived from b. For this purpose, the SUPER operator can be used on a method call to force the base class method to be used, where normally the derived class method would be used. So, the definition of the print method for class d could call the print method of class b.

```
PROC print() OF d
  WriteF('extra_d_data = \d, ', self.extra_d_data)
  SUPER self.print()
ENDPROC
```

Be careful, though, because without the SUPER operator this would involve a recursive call to the print method of class d, rather than a call to the base class method.

Just as data records of type d can be used wherever data records of type b were required, objects of class d can used in place of objects of class b. The following procedure prints a message and the object data, using the print method of b. (Of course, only the methods named by class b can be used in such a procedure, since the pointer p is of type PTR TO b.)

```
PROC msg_print(msg, p:PTR TO b)
  WriteF('Printing \s: ', msg)
  p.print()
ENDPROC

PROC main()
  DEF p_b:PTR TO b, p_d:PTR TO d
  NEW p_b, p_d
  p_b.b_data:=11
  p_d.b_data:=-3; p_d.extra_d_data:=3
  msg_print('p_b', p_b)
  msg_print('p_d', p_d)
ENDPROC
```

You can't use duff now, since it is not a class and b is, and msg_print expects a pointer to class b. The only other objects that can be passed to msg_print are objects from classes derived from b, and this is why p_d can be printed using msg_print. If you collect together the code and run the example you will see that the call to print in msg_print uses the

overridden print method when msg_print is called with p_d as a parameter. That is, the correct method is called even though the pointer p is not of type PTR TO d. This is called polymorphism: different implementations of print may be called depending on the real, dynamic type of p. Here's what should be printed:

```
Printing p_b: b_data = 11
Printing p_d: extra_d_data = 3, b_data = -3
```

Inheritance is not limited to a single layer: you can derive other classes from b, you can derive classes from d, and so on. For instance, if class e is derived from class d then it would inherit all the data of d and all the methods of d. This means that e would inherit the richer version of print, and may even override it yet again. In this case, class e would have two base classes, b and d, but would be derived directly from d (and indirectly from b, via d). Class d would therefore be known as the super class of e, since e is derived directly from d. (The super class of d is its only base class, b.) So, the SUPER operator is actually used to call the methods in the super class. In this example, the SUPER operator can be used in the methods of e to call methods of d.

The binary tree implementation above (see Binary Trees) suggests a good example for a class hierarchy (a collection of classes related by inheritance). A basic tree structure can be encapsulated in a base class definition, and then specific kinds of tree (with different data at the nodes) can be derived from this. In fact, the base class tree defined below is only useful for inheriting, since a tree is pretty useless without some data attached to the nodes. Since it is very likely that objects of class tree will never be useful (but objects of classes derived from tree would be), the tree class is called an abstract class.

```
OBJECT tree
  left:PTR TO tree, right:PTR TO tree
ENDOBJECT

PROC nodes() OF tree
  DEF tot=1
  IF self.left  THEN tot:=tot+self.left.nodes()
  IF self.right THEN tot:=tot+self.right.nodes()
ENDPROC tot

PROC leaves(show=FALSE) OF tree
  DEF tot=0
  IF self.left
    tot:=tot+self.left.leaves(show)
  ENDIF
  IF self.right
    tot:=tot+self.right.leaves(show)
  ELSEIF self.left=NIL
    IF show THEN self.print_node()
    tot++
  ENDIF
ENDPROC tot

PROC print_node() OF tree
  WriteF('<NULL> ')
ENDPROC
```

```
PROC print() OF tree
  IF self.left  THEN self.left.print()
  self.print_node()
  IF self.right THEN self.right.print()
ENDPROC
```

The nodes and leaves methods return the number of nodes and leaves of the tree, respectively, with the leaves method taking a flag to specify whether the leaves should also be printed. These methods should never need overriding in a class derived from tree, and neither should print, which traverses the tree, printing the nodes from left to right. However, the print_node method probably should be overridden, as is the case in the integer tree defined below.

```
OBJECT integer_tree OF tree
  int
ENDOBJECT

PROC create(i) OF integer_tree
  self.int:=i
ENDPROC

PROC add(i) OF integer_tree
  DEF p:PTR TO integer_tree
  IF i < self.int
    IF self.left
      p:=self.left
      p.add(i)
    ELSE
      self.left:=NEW p.create(i)
    ENDIF
  ELSEIF i > self.int
    IF self.right
      p:=self.right
      p.add(i)
    ELSE
      self.right:=NEW p.create(i)
    ENDIF
  ENDIF
ENDPROC

PROC print_node() OF integer_tree
  WriteF('\d ', self.int)
ENDPROC
```

This is a nice example of polymorphism at work: we can implement a tree which works with integers simply by defining the appropriate methods. The leaves method (of the tree class) will then automatically call the integer_tree version of print_node whenever we pass it an integer_tree object. The definitions of tree and integer_tree can even be in different modules (see Data-Hiding in E), and, using these OOP techniques, the module containing tree would not need to be recompiled even if a class like integer_tree is added or changed. This shows why OOP is good for code-reuse and extensibility: with traditional programming techniques we would have to adapt the binary tree functions to account for integers, and again for each new datatype.

Notice that the recursive use of the new method add must be called via an auxiliary pointer, p, of the derived class. This is because the left and right elements of tree are pointers to tree objects and add is not a method of tree (the compiler would reject the code as a syntax error if you tried to directly access add under these circumstances). Of course, if the tree class had an add method there would not be this problem, but what would the code be for such a method?

An add method does not really make sense for tree, but if almost all classes derived from tree are going to need such a method it might be nice to include it in the tree base class. This is the purpose of abstract methods. An abstract method is one which exists in a base class solely so that it can be overridden in some derived class. Normally, such methods have no sensible definition in the base class, so there is a special keyword, EMPTY, which can be used to define them. For example, the add method in tree would be defined as below.

```
PROC add(x) OF tree IS EMPTY
```

With this definition, the code for the add method for the integer_tree class could be simplified. (The auxiliary pointer, p, is still needed for use with NEW, since an expression like self.left is not a pointer variable.)

```
PROC add(i) OF integer_tree
  DEF p:PTR TO integer_tree
  IF i < self.int
    IF self.left
      self.left.add(i)
    ELSE
      self.left:=NEW p.create(i)
    ENDIF
  ELSEIF i > self.int
    IF self.right
      self.right.add(i)
    ELSE
      self.right:=NEW p.create(i)
    ENDIF
  ENDIF
ENDPROC
```

This, however, is not the best example of an abstract method, since the add method in every class derived from tree must now take a single LONG value as an parameter, in order to be compatible. In general, though, a class representing a tree with node data of type t would really want an add method to take a single parameter of type t. The fact that a LONG value can represent a pointer to any type is helpful, here. This means that the definition of add may not be so limiting, after all.

The print_node method is much more obviously suited to being an abstract method. The above definition prints something silly, because at that point we didn't know about abstract methods and we needed the method to be defined in the base class. A much better definition would make print_node abstract.

```
PROC print_node() OF tree IS EMPTY
```

```
It is quite safe to call these abstract methods, even for tree class
objects.  If a method is still abstract in any class (i.e., it has not
been overridden), then calling it on objects of that class has the same
effect as calling a function which just returns zero (i.e., it does very
little!).

   The integer_tree class could be used like this:

     PROC main()
       DEF t:PTR TO integer_tree
       NEW t.create(10)
       t.add(-10)
       t.add(3)
       t.add(5)
       t.add(-1)
       t.add(1)
       WriteF('t has \d nodes, with \d leaves: ',
              t.nodes(), t.leaves())
       t.leaves(TRUE)
       WriteF('\n')
       WriteF('Contents of t: ')
       t.print()
       WriteF('\n')
       END t
     ENDPROC
```

1.9 Data-Hiding in E

```
Data-Hiding in E
================

   Data-hiding is accomplished in E at the module level.  This means,
effectively, that it is wise to define classes in separate modules (or at
least only closely related classes together in a module), taking care to
EXPORT only the definitions that you need to.  You can also use the
PRIVATE keyword in the definition of any OBJECT to hide all the
elements following it from code which uses the module (although this does
not affect the code within the module).  The PUBLIC keyword can be used in
a similar way to make the elements which follow visible (i.e., accessible)
again, as they are by default.  For instance, the following OBJECT
definition makes x, y, a and b private (so only visible to the code within
the same module), and p, q and r public (so visible to code external to
the module, too).

     OBJECT rec
       p:INT
     PRIVATE
       x:INT
       y
     PUBLIC
       q
       r:PTR TO LONG
     PRIVATE
```

```
  a:PTR TO LONG, b
ENDOBJECT
```

For the set class you would probably want to make all the data private and all the methods public. In this way you force programs which use this module to use the supplied interface, rather than fiddling with the set data structures themselves. The following example is the complete code for a simple, inefficient set class, and can be compiled to a module.

```
OPT MODULE  -> Define class 'set' in a module
OPT EXPORT  -> Export everything

/* The data for the class */
OBJECT set PRIVATE  -> Make all the data private
  elements:PTR TO LONG
  maxsize, size
ENDOBJECT

/* Creation constructor */
/* Minimum size of 1, maximum 100000, default 100 */
PROC create(sz=100) OF set
  DEF p:PTR TO LONG
  self.maxsize:=IF (sz>0) AND (sz<100000) THEN sz ELSE 100 -> Check size
  self.elements:=NEW p[self.maxsize]
ENDPROC

/* Copy constructor */
PROC copy(oldset:PTR TO set) OF set
  DEF i
  self.create(oldset.maxsize)  -> Call create method!
  FOR i:=0 TO oldset.size-1  -> Copy elements
    self.elements[i]:=oldset.elements[i]
  ENDFOR
  self.size:=oldset.size
ENDPROC

/* Destructor */
PROC end() OF set
  DEF p:PTR TO LONG
  IF self.maxsize<>0  -> Check that it was allocated
    p:=self.elements
    END p[self.maxsize]
  ENDIF
ENDPROC

/* Add an element */
PROC add(x) OF set
  IF self.member(x)=FALSE  -> Is it new? (Call member method!)
    IF self.size=self.maxsize
      Raise("full")  -> The set is already full
    ELSE
      self.elements[self.size]:=x
      self.size:=self.size+1
    ENDIF
  ENDIF
ENDPROC
```

```
/* Test for membership */
PROC member(x) OF set
  DEF i
  FOR i:=0 TO self.size-1
    IF self.elements[i]=x THEN RETURN TRUE
  ENDFOR
ENDPROC FALSE

/* Test for emptiness */
PROC empty() OF set IS self.size=0

/* Union (add) another set */
PROC union(other:PTR TO set) OF set
  DEF i
  FOR i:=0 TO other.size-1
    self.add(other.elements[i])  -> Call add method!
  ENDFOR
ENDPROC

/* Print out the contents */
PROC print() OF set
  DEF i
  WriteF('{ ')
  FOR i:=0 TO self.size-1
    WriteF('\d ', self.elements[i])
  ENDFOR
  WriteF('}')
ENDPROC
```

This class can be used in another module or program, as below:

```
MODULE '*set'

PROC main() HANDLE
  DEF s=NIL:PTR TO set
  NEW s.create(20)
  s.add(1)
  s.add(-13)
  s.add(91)
  s.add(42)
  s.add(-76)
  IF s.member(1) THEN WriteF('1 is a member\n')
  IF s.member(11) THEN WriteF('11 is a member\n')
  WriteF('s = ')
  s.print()
  WriteF('\n')
EXCEPT DO
  END s
  SELECT exception
  CASE "NEW"
    WriteF('Out of memory\n')
  CASE "full"
    WriteF('Set is full\n')
  ENDSELECT
ENDPROC
```