Date:	30 January 1998
To:	J3
From:	Werner W Schulz, email: `wws20@cam.ac.uk`
Subject:	Object-oriented Programming in F2000: Inheritance and Polymorphism
References:	97/196, 97/230

Contents

1 An Overview **2**
1.1 Goals and Constraints . . . 2
1.2 A Look at other OOP Languages . . . 3
1.3 Which route for OOP in F2000? . . . 3
1.4 Notation . . . 4

2 Class: A new Programme Unit **4**
2.1 Rules and Comments . . . 4
2.2 Optional additions to the `CLASS` syntax . . . 8
2.3 Usage and Semantics of Classes and Objects . . . 8
2.4 Extending OOP to existing Fortran . . . 9
2.5 Other Issues . . . 9

3 Inheritance and Polymorphism **10**
3.1 The subtyping problem . . . 10
3.2 Mytype . . . 12
3.3 Bounded matching . . . 12
3.4 Hash-types . . . 13

4 Critical Comments **14**
4.1 Criticism of `CLASS` . . . 14
4.2 Proposals 97/196 and 97/230 . . . 14
4.3 The Point Class Example . . . 16

5 The Module: Problems and Solutions **18**
5.1 A Fortran Environment . . . 18
5.2 Module Extensions . . . 18

6 Conclusion **19**

A Glossary of Terms **19**

List of Figures

1 The proposed `CLASS` construct. . . . 5
2 Animal class and herbivore subclass where `plant` is a subclass of `food`. . . . 10
3 Singly-linked Node class with `MYTYPE`. . . . 11
4 Doubly-linked Node class. . . . 11
5 Animal class and herbivore subclass revisited. . . . 12
6 List class without binary methods. . . . 13
7 The extensible `TYPE` as proposed in [1] (in shortened form). . . . 14
8 The `Point2D` class in the proposed `CLASS` syntax. . . . 16
9 The `Point3D` class in the proposed `CLASS` syntax. . . . 17
10 Using Point classes with homogeneous and heterogeneous datastructures. . . . 17

1 An Overview

Object-oriented programming (OOP) has become a powerful method of software engineering. Construction, extension, *and* maintenance of large programmes benefit from reusability and stability of code as well as better real-world modelling. The main techniques of OOP are known as data abstraction and encapsulation (information hiding), inheritance and polymorphism. The addition of OOP facilities is arguably the most important and complex requirement for F2000.

Fortran has been and still is the language of choice for many scientists and engineers but a language like C++ has made some inroads despite the complexity and the lack of a standard.[1] Fortran is rather late in adopting an OOP approach, a standard including OOP and first compliant compilers will only appear in about four years time. F2000 must come up with a convincing design of OOP to make up for this lost time, to reverse some of the trends of recent years and to continue the direction set by Fortran 90 towards a more modern, flexible and safer language.

In the following chapters I want to present a comprehensive OOP proposal for F2000. One of the reasons for writing this proposal is my concern that F2000 might only offer a slimmed-down version of OOP while later revisions will complete the task. In my view there is no need for such a half-hearted approach and it could be detrimental to Fortran. I could only establish that polymorphism and inheritance are required but there seem to be no detailed specifications outlining any minimal requirements and options for OOP in Fortran as one would expect for such a complex issue. Only a 'white paper' by W. Clodius [2] lists and discusses some of the important topics and possible choices.

After reviewing some guiding principles the syntax for a `CLASS` is proposed while detailed comments justify the choices. Various related issues and options are discussed, and other OOP proposals and options are compared with the proposed `CLASS` construct. Some ideas about modules and an 'environment' are discussed. The reader is referred to the appendix for the definitions of terms that will help to avoid possible misunderstandings.

1.1 Goals and Constraints

OOP will be a completely new feature in Fortran though object-based programming can be achieved using `MODULE`, `TYPE` and overloading via `INTERFACE`. The challenge facing F2000 is to implement a well-designed OOP construct under the constraint of backward compatibility and the limited schedule for discussing and drafting a new standard.

A good design must reflect the underlying concepts of OOP, should be simple to understand and easy to use, promote software quality and good programming habits, enforce safety, and pay attention to efficiency. As a result F2000 should be able to present itself and be recognized as a first-rate modern, flexible and safe programming language that is useful as a teaching language as well as a practical tool for large-scale software engineering. The design should convince people who left or are about to leave Fortran for languages like C++ to stay with Fortran.

OOP is often discussed in the message passing model. This model reflects the encapsulation properties well and is used in other areas like parallel programming,

Polymorphism is the crucial feature of OOP as it determines the usefulness and flexibility of the OOP design but this requires a careful discussion and implementation. Research into OOP over the last years has clarified many issues and proposed new solutions to handle polymorphism well. F2000 is in the fortunate position to implement OOP at this stage and should take full advantage of these results in its design [3].

A simple and transparent design is very important. OOP can be difficult to grasp in all its implications, hence syntax and semantics of the OOP features should work together and ease the burden of learning and applying OOP. The syntax should clearly mark that a new and distinct construct is present; the proper choice of names in the syntax is equally important as a close correspondence to everyday usage and practice in other OOP languages will make life easier. A clearly structured syntax will also enhance the readability of codes. Measures for automatic code documentation should be taken.

The default behaviour should reflect and support forcefully the concepts of OOP, e.g. data encapsulation means that access to objects is through a rather small and well-defined interface and the internal details are completely hidden. Explicit memory management should be kept minimal. The design should also eliminate features that are easily abused or circumvented. The ability to write generic programmes is very useful, and OOP without genericity lacks an important ingredient for abstraction. These qualities will also enhance the safety of the codes.

[1] A draft for a C++ standard has recently been submitted.

Since errors at run-time may be difficult to debug, OOP in F2000 should be strongly typed to discover all errors associated with the types of operands at compile time. This should also allow better code optimisation. Strong typing will impose restrictions on polymorphism and hence the flexibility of OOP. Here one should strive to go as far as possible to allow an expressive form of polymorphism that is both easy to grasp and can be type-checked. Strong typing is also a time-saving feature since it is well-known that the cost of fixing an error rises strongly when detection is delayed.

The constraint of backward compatibility should not be used to compromise the OOP design since this will have serious negative repercussions for the acceptance of F2000 and may hinder any serious applications. F95 has already lost some compatiblity with F77/F90 and this trend is to continue with further revisions. To some extent this problem will be circumvented in my proposal by either excluding the use of certain older Fortran elements or by advocating a style more in line with the one indicated by the modern form of F90.

The time constraint is a very limiting factor. OOP in F2000 has to borrow from existing languages rather than starting from scratch. A selection of the most common ones will be discussed next. Fortunately, recent research has cleared the difficult area of inheritance and polymorphism enough to allow Fortran to make a very clear and informed choice that was not available to nearly all the commonly used OOP languages around and which led to some serious shortcomings.

As there is no time for beta-testing of any of the OOP features before the release of the F2000 standard a cautious approach must be chosen in difficult areas which imposes rather more than less restrictions on some features. It will be much easier later on to relax these restrictions than to impose them.

F90 added the `MODULE` to the language, however, its current form is too restrictive for programming in the large. Since OOP will affect the solution the future role for modules must be part of the discussion, even if no particular steps are taken at this stage. Care should be taken not to overload the role of modules with features that are better left to others. In my view this is particularly important and urgent in the case of OOP. Several options how to extend modules will be discussed later.

1.2 A Look at other OOP Languages

C++ is arguably the most widespread OOP language though whether it is just used as a better C or mostly for OOP remains unclear. OOP is supported by a class construct, multiple inheritance is allowed, and genericity is provided through templates. The shortcomings of C++ are wellknown, and C++ is certainly an example that one should not follow though some ideas concerning forms of visibility (private,protected,friends, public) and some aspects of inheritance should be kept in mind. Java derives from the C/C++ family but with a simpler syntax. Java is not type-safe, has no genericty but provides modules called packages. Efficiency is low due to its interpreted code; the great plus is the portability of Java. Polymorphism in C++ and Java is quite restricted.

Eiffel [4] and its offspring, Sather [5], are, like Java, pure OOP languages based on classes. Eiffel's syntax is very elegant and supports a number of new ideas summarized as 'design by contract'. Multiple inheritance is allowed as are generic classes. Eiffel allows covariant changes of arguments and is therefore not fully type-safe.[2] All of them know a construct *self* (or *this*) which refers to the current object. C++ and Sather are quite efficient in their OOP implementations, Sather can actually be faster than C++.

An interesting alternative is Beta [6] which unifies class, type and method into a so-called pattern, thereby bringing procedural programming and OOP together. Beta also supports concurrency and modules (fragments) but its type system is currently not completely safe.

ADA95 [7] provides OOP in a very different way, including the notation. ADA95 is a complex language with strengths, e.g. it supports programming in the large through packages (modules) with separate interfaces and was designed with safety in mind. OOP is supported through `packages`, child packages, tagged (or extensible) and generic types. ADA95's tagged types are wrapped inside packages mimic classes. A notion of *self* does not exist. Inheritance and polymorphism are more restricted than is necessary; ADA95 is not fully type-safe. Modula-3 is close to ADA95 with a module construct and record types to simulate classes.

Several other approaches to OOP (e.g. Self) are possible but these languages are often dynamically typed.

1.3 Which route for OOP in F2000?

All of the above languages have their specific deficiencies; in the context of OOP the important topics are the conflicting ones of type-safety and polymorphism.

[2] A proposal for remedy is made but its effect is not yet clear.

The ADA/Modula family started as a module language that added OOP by extending their record types. Since F90/95 is in a similar position this would be one possible path to OOP for F2000, and the current workproposal (CWP, [1]) seems to lean in this direction. The alternative is to use a distinct class construct. This approach is taken by most pure and hybrid OOP languages, and I will take this route which in my view has advantages which will be discussed in the various chapters; the syntax is borrowed to some extent from Eiffel.

1.4 Notation

The following notation is used throughout this paper:

```
< ... >    user-supplied names and statements
[ ... ]    optional declarations
|          mutually exclusive options
!          comments
```

In `[[...]]` with out spaces between equal brackets the outer bracket denotes an optional declaration while the inner one is a syntactical part of the declaration. The syntax names will always be written in capital letters such as `CLASS` while variables and other names appear in lower or mixed case. A more general and unified version of F90 notation, including a simplified operator declaration, is used throughout and should be adopted by F2000. Variables, procedures, called methods here, classes, etc. will use a very similar syntax as in:

```
<type-declaration> [, attributes] :: <name>

! Examples:
REAL, PUBLIC, TARGET :: var
REAL, OPERATOR, DIMENSION(3) :: ".X." (vec1,vec2)    ! vector product
CLASS, ABSTRACT :: person
```

Declarations that can be followed by a list of names, e.g. `ONLY`, contain a colon, ":".

From now on when using the word "type" I am using the definition as given in the glossary; the F90/95 construct of the same name is written `TYPE`.

The word parameter is sometimes used as a synonym for argument. Here I will not follow this convention but I denote as parameter such types that parametrize classes and denote as arguments the values that are passed to methods.

2 Class: A new Programme Unit

The concept of OOP centres around the concept of a class, i.e. an entity that contains data (variables defining a state) and methods (methods describing behaviour). Objects are instances of classes and work together by exchanging messages. Internal details of a class are completely hidden to the outside. The notion of a class has no equivalent in Fortran though some similarities exist with the current `TYPE`s and `MODULE`s.

I want to keep the proposed `CLASS` construct separate from `TYPE` for various reasons, some are listed in a later chapter; the reason in this chapter is simply to concentrate on the features of `CLASS` that I would like to see implemented in F2000 without worrying about `TYPE` at all. A discussion of `CLASS` vs. `TYE` is postponed till later.

The syntax for the `CLASS` construct is given in Figure 1. The public interface of `CLASS`, i.e. its type, consists of (none, some, all of) its variables, some of its methods (at least one) including their arguments, and the attributes of `CLASS`.

2.1 Rules and Comments

The various components of the syntax in Figure 1 are defined, rules stated and comments (*Note*) provided for the underlying reasons, some of the comments reflect my personal preferences. Some of the rules may not be obvious at first since some important aspects such as polymorphism will be dealt with later though the *Notes* try to give a hint.

1. A `CLASS` is a programme unit with a name and optional attributes and can be compiled separately; it can be part of a `MODULE`; in this case it can have the usual attributes `PUBLIC` or `PRIVATE`.

   Note: A declaration of `PUBLIC` or `PRIVATE` refers to the visibility of the whole class outside the module, not the visibility of class variables or methods inside the module.

```
CLASS [, FROZEN | ABSTRACT ]   &
      [, PRIVATE | PUBLIC ]    :: <class_name>[[<parameters>]]

   [ USE <module_name> [,ONLY: <only_list>] ]
   IMPLICIT NONE

   [ INHERIT :  <superclass>[[parameters]]
     [ RENAME   : <renaming list>]
     [ REDEFINE : <superclass_methods> ] ]

   [ CREATE : <init_method(s)> ]

   [ <datatype> [,<attributes>] :: <class_variable> [=<value>] ]

   [ [<datatype>,] <class_method> [,<attributes>] :: <method_name> [(<arguments>)]
                [argument_declarations]
                [<method body>]
     END class_method <method_name> ]

     ! datatype:      INTEGER, REAL, ..., <classname>, MYTYPE, #<hash-type>
     ! class_method:  FUNCTION, SUBROUTINE, OPERATOR, ASSIGNMENT
     ! attributes:    INTEGER, REAL, MYTYPE, etc.
     !                PUBLIC, PRIVATE, ABSTRACT, etc.

END CLASS <class_name>
```

Figure 1: The proposed `CLASS` construct.

2. The body of a `CLASS` consists of variables, methods and instructions to inherit other `CLASS`es, use `MODULE`s, and further declarations; `IMPLICIT NONE` is mandatory to avoid any undeclared class variable. Global attribute statements should not be used, rather all attributes should be declared with the variables and methods.

3. `CLASS`es define a tree hierarchy of `CLASS`es through inheritance; the top of a (sub)tree is called superclass; the branches subclasses. `CLASS` inherits from the superclass named in the `INHERIT` statement all variables and methods, public, readonly or private. `USE` and `CREATE` statements are not inherited.

   Note: The hierarchy defines an order where < denotes the *subclass* < *superclass* relationship.

4. A class can use a module to access definitions, data and methods. The `USE` of modules is private to the class, i.e. neither subclasses nor other units that have objects of the class can access the module unless by explicit `USE`. `USE` appears before `INHERIT`.

   Note: This restrictive use for `MODULE` is necessary to remain true to the principles of OOP, encapsulation and information hiding.

5. A `CLASS` can be inherited by default unless prevented by the attribute `FROZEN`.

6. A class can be made abstract (deferred) by using the attribute `ABSTRACT`. `FROZEN` and `ABSTRACT` are mutually exclusive attributes. An abstract class cannot be used to create an instance of that class, i.e.
   ```
        CLASS(A_abstract) :: A_object
   ```
   is illegal. Compilers must issue an error message.

7. The definition of a method can be delayed through the `ABSTRACT` attribute until it is defined in a subclass. Any class that contains an `ABSTRACT` method either by explicit declaration or through inheritance is abstract and must contain the attribute `ABSTRACT`. An abstract method must explicitly declare its interface, i.e. the arguments, only the body of the method is omitted. A concrete subclass derived from an abstract superclass must redefine all abstract methods of its superclass.

Note: These rules allow to change any abstract superclass into a concrete one without effecting the subclass.

8. A `CLASS` can be parametrized as in this linked list:
```
     CLASS :: list[T]
          T       :: value
          list[T] :: next
          ...
     END CLASS linklist
     list[REAL] :: head  ! linked list with real values
```
   where `T` stands for any allowed type declaration in Fortran. `T` can be constrained by a bound, e.g. `T<#Comparable` (see below). A subclass of a parametrized class must be parametrized as well with a variable or explicit type.

   Note: Square brackets (`[]`) are introduced to provide a distinction between parametrized classes and declaration of objects.

   Note: Whether or not the `POINTER` attribute has to be added in dynamical structures will depend on the way objects are implemented.

9. `CREATE` names a specific method (`MYTYPE, FUNCTION :: <name>`) which must be called when an object is created (see below). The named methods are available for this purpose even if declared `PRIVATE`.

   Note: Sometimes the only possible way of properly initializing an object is by call of a method and not by initial values for variables. A subclass is different from its superclass, its internal state is probably different as well and therefore does not inherit the `CREATE` statement. The method in `CREATE` is still available as an ordinary method but should be `PRIVATE` to avoid misuse. This feature is a must for good and safe software development.

   Note: This has to fit together with item R.7 (Constructors/Destructors) of the F2000 Workplan.

10. Inherited methods and variables from any superclass can be renamed in the subclass in the usual way (using `=>`) to avoid name clashes or give more descriptive names. The new name must be used from now on everywhere, e.g. in subclasses of this class. The `RENAME` statement can only follow immediately after the `INHERIT` statement.

11. Inherited methods can be redefined, i.e. a body of executable statements is provided for the first time (for abstract classes) or a new body for methods already defined in the superclass. The name of these methods must appear in the `REDEFINE` list, and the redefined methods must appear within the scope. The `REDEFINE` statement can only follow immediately after the `INHERIT` statement (see Figure 1). Redefined methods must have a conforming interface with the superclass method; for more details see the later chapter on polymorphism.

    Note: Strictly speaking, `REDEFINE` is not necessary, but it reminds the programmer and enhances readability.

12. Inherited variables cannot be redefined, except changed to `READONLY` from `PRIVATE` or, when declared with `PARAMETER`, the actual value can be reset.

13. The class variables can have various attributes. Variables have as default the new attribute `READONLY` with `PRIVATE` as an alternative while `PUBLIC` is forbidden. Variables can have initial values that will be used to initialize objects.

    Note: One basic principle of OOP is to work only with objects and a small interface to inquire about or change the object. A `PUBLIC` variable violates this principle and is therefore excluded. This is no burden as `READONLY` is more than adequate in nearly all cases. `READONLY` is a practical solution that avoids the burden of writing many access function if only `PRIVATE` were available. See also below on referential invariance.

    Note: `READONLY` will obviously also be useful within `MODULE`; the same name serves in a similar fashion as a qualifier for file access, and the `INTENT(IN)` again provides a similar functionality. It is also an obvious help for optimisations.

14. Procedures are by default `PUBLIC`, this can be overridden using `PRIVATE`.

    Note: Procedures are the only (recommended) way to change objects, hence they should be `PUBLIC` by default, in contrast to the variables of a class; see also the next item.

15. `FUNCTION`s without arguments should be declared and used without parentheses (as is possible with `SUBROUTINE`s).

    Important Note: Users of a `FUNCTION` do (and need) not know whether they access a variable or a `FUNCTION`. In other words, all variables are shortcuts for get-functions which explains the `READONLY` default before, and consequently the visible interface of a `CLASS` consists only of methods (or can be treated as such). Later modifications of the class implementation will not affect code that uses the class which adds to the stability of the software. This property comes also under the name referential invariance. This is also satisfying as it forms a bridge between practice and theory of OOP, since the latter usually treats only the case of an interface consisting purely of methods.

16. A `FUNCTION` without arguments can be redefined to become a variable but not vice versa.

    Note: If the class inherits a method that assigns a value to a variable then a redefinition of this variable as a `FUNCTION` causes havoc. A simple example is a class `Polygon` with a function `NrCorners`; the subclass `Rectangle` can turn this function into a variable since the number of corners is now fixed to four.

17. `PRIVATE` variables and methods are only visible within one instance of a `CLASS`, i.e. objects of the same class cannot see the `PRIVATE` variables and methods of each other.

    Note: This restriction is necessary though it causes problems for software design. A solution can be provided by an extension of `PRIVATE`.

18. Variables within each class method are local to the method and cannot be `SAVE`d.

    Note: The whole idea of an object runs contrary to the notion of saving local variables in class methods and may cause serious problems for compilers. All properties should be derivable from the class variables. If necessary, an extra (private) class variable will provide the needed facility in a much cleaner way. Fortran's rule of saving initialized variables does not apply in this case.[3]

19. Within the `CLASS` methods one can refer to the current object using the special word `SELF`. This makes it possible to refer to object within the class methods if it has to appear as an argument in a method call inside the class. The use of `SELF` for any other purpose is forbidden.

    Note: This makes `SELF` effectively a reserved word in F2000 but it is the only clean solution.

20. In a similar fashion the word `SUPER` refers to the superclass. Similar rules and recommendations as in the case of `SELF` apply.

    Note: When `REDEFINE`'ing methods the common situation occurs that only some extra statements have to be added to the method of the superclass.

21. A word `MYTYPE` is introduced that allows to refer to the type of `SELF` as well as hash-types which can refer to polymorphic variables. See below for reasons and examples.

22. Rules about order of evaluation and precedence must be defined. E.g. `X%add(Y)%add(Z)` should mean `(X%add(Y))%add(Z)` using standard Fortran rules though parentheses are recommended for readability.

23. Some features are disallowed in `CLASS`es. They are `BLOCK DATA`, `COMMON` blocks, `EQUIVALENCE`, `ENTRY`, `SEQUENCE`, `SAVE`, `INCLUDE`. In general, an approach similar to F or Elf of eliminating all duplicate and unsafe forms should be used (enforced).

24. The root of the class hierarchy is called `ANY` or similar.

A clarification:
`READONLY` variables have been introduced to reduce the burden of writing safe classes. Some pure OOP languages do not allow access to variables at all but require get-functions which are often trivial. To avoid these trivial functions variables are made `READONLY` and are equivalent to public functions (without argument list). In the remainder of this proposal I will not distinguish between these variables and public functions (without arguments). The *object type* (or just *type*) of a class instance consist of all public methods which includes the readonly variables. This equivalence will be important in two cases: a) when discussing matching and polymorphism in a later chapter, and b) when creating composite classes the meaning of `READONLY` can be inferred (recursively) from this equivalence: the visible variables and methods of a `READONLY` object remain visible.

[3]This rule is rather debatable anyway since saving and initializing are very different actions.

2.2 Optional additions to the `CLASS` syntax

Some OOP languages have added some useful features to their classes. Some are listed here:

1. Assertions (post- and pre-conditions, invariants) are useful for documentation and especially during code development and add to the reliablity of software. Recovery from unexpected results via exception-handling is important in modern systems.
2. Multiple inheritance by inheriting several classes but only when the need of this feature becomes apparent and can be implemented well. I have tried to design my proposal with this possible addition in mind. Few additions or changes would be necessary to accomodate multiple inheritance.
3. Instructions supporting concurrent computing are a natural extension to OOP. ADA95, Eiffel and Beta already support it.
4. Sometimes it is useful/necessary for performance that objects can have access to the private components of other objects. One could extend the syntax of `PRIVATE` to `PRIVATE(<access_list>)` where `<access_list>` contains a list of privileged classes including the current class itself whose objects have access to the variable or method.
5. To allow for pure OOP the definition of `CLASS` could be extended to be a standalone programme with no `PROGRAM` unit needed. This is quite useful in event-based programming, and nothing needs to be added to the syntax.

2.3 Usage and Semantics of Classes and Objects

Classes and their instances, objects, are used in the following way:

- defining an object in a programme unit:

```
 <class_name>[[<parameters>]] [, attributes] :: <object>
#<class_name>[[<parameters>]] [, attributes] :: <object>
```

- initializing an object:

```
<object> = <class>       ! no CREATE method defined
<object> = <class>%<createmethod>[(<arguments>)]
```

- use of `CLASS` variables and methods:

```
<result>  = <object>%<variable>
<result>  = <object>%<function> [(arguments)]
CALL <object>%<subroutine>[(arguments)]
```

A class is simply accessed by its name; if it is contained within a module, the usual `USE` statement must be made. Methods of a class are always qualified by the name of the object which must be declared, hence there is no need to import a class. In the declaration of variables I have not followed the current usage of `TYPE` since the class name makes perfectly clear what is meant and the extra `CLASS` is superfluous. I also find this syntax more in line with the declaration of the intrinsic types (`INTEGER, REAL`, etc.) which can be viewed as special classes (see below). A different reason is genericity which will be more difficult to achieve if types do not have a uniform syntax. No other language to my knowledge uses Fortran's notation, see e.g. ADA.

The essential steps of initialisation are creating an instance of the class, intialize with default values, attach instance to object and, if defined, one of the creation methods must executed. The initialization has to allocate memory if needed but there is no explicit reference to `ALLOCATE`. Details remain open since they depend on the representation of objects and have to be harmonized with item R.7 of the workplan.

Some important points I just present in the form of questions since it would lead to far to spell out the detailed rules. They are also not essential for the parts that follow. These matters are mostly important with respect to performance and memory management. Whatever solution is chosen the user should not have to take care of memory management of objects.

What is the default representation of objects? Should objects be references to values or contain their values? Does a composite object contain a subobject or only refer to it? Since OOP is characterized by a very dynamical run-time structure with objects created on demand the preferred solution should be the reference default. An attribute for objects that contain their values should then be added. (Eiffel uses the name expanded; Fortran's intrinsic types like `REAL` are expanded objects.) Hash-types are always references (see below for definition). Any reference should have the default value `NULL`. (The reference model is used in most of my later examples.)

What form should assignment take? What does equality mean? Should there be different notations for assignment and refering? Clodius [2] lists three different possibilities for assignment with their consequences for the semantics of an object: by reference, by value, and by functional access. Most OOP languages use the first, by reference. Using references is the standard mechanism to allow for subsumption (or its alternative) and sharing of objects. The danger with references is that the user can accidentally modify an object. The alternatives are safer but suffer from other problems, mostly performance related. I personally prefer the reference solution. The notation for assignments should use the familiar equal sign as in `a=b` while for copies (assignment by value) the more verbose `CALL a%COPY(b)` should be used.[4] For objects that are references one needs mechanisms for copying, cloning and deep-cloning (for recursive cloning).

Similar questions apply to the checking of equality which could mean that equal objects refer to the same object or their values are equal. A deep-equal would recursively follow all references to check for value equality.

2.4 Extending OOP to existing Fortran

The class construct must be extended to all intrinsic data types of Fortran, i.e. `INTEGER`, `REAL`, `COMPLEX`, `CHARACTER` and `LOGICAL`, at least conceptually, to put them on the same basis as user-defined classes. The language-intrinsic attributes and inquiry functions such as `SIZE`, `KIND`, `RANGE`, etc. are viewed as methods of these predefined classes, e.g.

```
REAL(KIND=kind)      :: x
REAL, DIMENSION(20) :: b
kind_of_x = x%KIND
size_of_B = b%SIZE
```

This is an obvious property that F2000 must provide for seamless OOP, and it becomes essential for generic classes. Several rather general classes are needed, for example, a class `Numeric` that contains the standard arithmetic operations, a class `Comparable` with operations like `<`, `==`, etc. This topic has not been addressed by any other proposal as far as I know.

2.5 Other Issues

1. Access to class names: I have left out a way to compare an object's class with a given class name or another object's class. This is usually not needed and considered bad programming.

2. Type conversion: I have avoided ways of converting an object into an object of a different class. This is usually not needed and can easily be abused (see C++). The access methods, the initializing statement and the constructs presented later on should be sufficient. Another reason against casts is

3. Memory management: A proper implementation of OOP will require garbage collection. The user should be saved from the explicit management of this chore as it complicates programming and is too often impossible to achieve. A strongly-typed F2000 will allow garbage collection while type casts are detrimental.

4. A class library: Introducing a `CLASS` syntax is just the first step to give Fortran users access to OOP. A minimal library of classes is certainly necessary and should become part of Fortran as a separate standard.

5. Documentation: When dealing with large programmes and a large number of classes (and modules) tools to abstract and subsequently find them become important. It is useful to provide a non-executable construct `INDEX: <list of keys>` to support additional tools with the documentation chores.

6. Support of IO for objects: This is important for real applications since one must be able to save objects to file and retrieve them.

[4] This seems to me to be more in line with the assignment rules in F90/95 for `TYPE`s with `POINTER` components.

```
CLASS :: animal
      SUBROUTINE eat( meal )
          food :: meal
      END SUBROUTINE eat
END CLASS animal

CLASS :: herbivore
      INHERIT  : animal
      REDEFINE : eat
      SUBROUTINE eat( meal )
          plant :: meal
      END SUBROUTINE eat
END CLASS herbivore
```

Figure 2: Animal class and herbivore subclass where `plant` is a subclass of `food`.

3 Inheritance and Polymorphism

Polymorphism is the ability of a variable name to become attached to objects of different types,[5] which are usually related as members of a subtree of the class hierarchy. This allows polymorphism to handle heterogeneous datastructures in a way that is impossible with Fortran. The method for a particular item in a polymorphic datastructure is determined at run-time via dynamic binding.

F2000 should be a statically type-checked (strongly typed) language, i.e. a compiler is able to assign a type to every expression and check operations for type correctness with obvious benefits for safety and optimisation. But strong-typing is by nature conservative, and this requires constraints upon the possible form of polymorphism and hence upon the expressiveness of the language as compared with dynamically typed ones. The exact interplay between inheritance and polymorphism under strong typing was not well understood until recently. The only known strongly typed forms of polymorphism were thus much more restrictive than is really necessary which explains the limitations of some popular OOP languages like C++. Bruce et al. in a series of papers have developed a much more satisfying solution that is very expressive and can still be completely type-checked [8, 3, 9]. This solution will also form the basis of my proposal for polymorphism in F2000. This chapter is largely drawn form these papers. For a number of useful definitions the reader is referred to the glossary.

3.1 The subtyping problem

I will illustrate the consequences of strong typing on a well-known example: while animals eat food the subclass herbivores should only eat plants. The `animal` class in Figure 2 is a typical example of how a user would want to programme this problem in an OOP language; it is, however, at odds with the subtyping rule (see Appendix A) which does not allow a covariant change of arguments (here `plant <: food`) in the subclass methods. A common problem is the case of binary methods, e.g. addition of objects or checking the equality of two objects.[6] Additional problems occur in an attempt to programme a linked list and to reuse the code for a doubly-linked list [3]. A more detailed discussion of binary methods can be found in [8].

Because binary methods and other examples of covariant argument change are so ubiquitous it is obvious that different rules have to be found to overcome this restriction. Essentially three solutions to the problem exist which to do not avoid binary methods: more precise typings, multimethods or matching. The advantages and disadvantages are discussed quite extensively in [8], and I want to concentrate on the third solution. Matching [3] is the most promising and useful method in terms of both expressiveness and simplicity and will be used in this proposal. To implement matching two new constructs are needed, a `MYTYPE` construct and so-called hashed types, denoted `#<type>`, together with a syntax (`<#`) to express the hierarchical relationship of bounded matching to indicate the intention of the programmer. One should note that ADA95 requires a similar decision by the programmer to mark variables for 'Class-wide Programming'. These new constructs are probably best introduced by example.

[5] The semantic version of polymorphism is treated here, not the syntactical one known as overloading.
[6] Note that it is the type of the *argument* that causes the problem not the type of the method.

```
CLASS :: Node
    IMPLICIT NONE
    Integer :: Value = 0
    MYTYPE  :: next  = NULL

    FUNCTION GetNext
        MYTYPE :: GetNext
        GetNext = next
    END FUNCTION GetNext
    SUBROUTINE SetNext( NewNext )
        MYTYPE :: NewNext
        next = NewNext
    END SUBROUTINE SetNext

    SUBROUTINE AttachRight( NewNext )
        MYTYPE :: NewNext
        CALL SELF%SetNext( NewNext )
    END SUBROUTINE AttachRight

END CLASS Node
```

Figure 3: Singly-linked Node class with `MYTYPE`.

```
CLASS :: DblNode
    IMPLICIT NONE
    INHERIT  : Node
    REDEFINE : AttachRight
    MYTYPE  :: prev = NULL

    FUNCTION GetPrev
        MYTYPE :: GetPrev
        GetPrev = prev
    END FUNCTION GetPrev
    SUBROUTINE SetPrev( NewPrev )
        MYTYPE :: NewPrev
        prev = NewPrev
    END SUBROUTINE SetPrev

    SUBROUTINE AttachRight( NewNext )
        MYTYPE :: NewNext
        CALL SELF%SetNext( NewNext )
        CALL NewNext%SetPrev( SELF )
    END SUBROUTINE AttachRight

END CLASS Node
```

Figure 4: Doubly-linked Node class.

```
CLASS :: animal[myfoodtype<#foodtype]  ! parameter myfoodtype
      SUBROUTINE eat( meal )           ! constrained by foodtype
          myfoodtype :: meal
          ...
      END SUBROUTINE eat
END CLASS animal

CLASS :: herbivore[myfoodtype<#planttype]
      INHERIT : animal[myfoodtype]
      ...
END CLASS herbivore
```

Figure 5: Animal class and herbivore subclass revisited.

3.2 Mytype

We need a way to indicate the automatic covariant change of types in subclasses. This is provided by the `MYTYPE` construct.[7] An example will illustrate this in Figures 3 and 4, other examples will follow later.

In the class `Node` `MYTYPE` is interpreted as `Node` while in the doubly-linked list as `DblNode`, and this includes the inherited methods from `Node`. This means all the code from the linked-list is reused, only `AttachRight` needs a modification and `SetPrev` has to be added. Without a construct like `MYTYPE` one could not reuse any of the code from `Node`. The binary methods `AttachRight, SetPrev, SetNext` do not pose problems unless one tries to work with a mixture of singly and doubly-linked objects which does not occur in practical work. `DblNode` is, however, not a subtype of `Node` anymore due to the presence of `MYTYPE`, but `DblNode` still matches `Node` since it contains at least the methods of `Node` with the 'same' types where any appearance of `MYTYPE` is treated as the 'same' type. This example should have made obvious that the `MYTYPE` construct is extremely useful and reflects very accurately certain relationships when going from a superclass to a subclass.

3.3 Bounded matching

Dealing with heterogeneous datastructures is an important application for polymorphism. Usually these structures are a collection of objects related with each other, e.g. a list of various windows on a computer screen, or the content of a library with books, theses, journals, etc. Sometimes certain operations on these collections are needed: set operations, binary search trees, etc. In the case of search trees the objects must have the quality of being comparable, i.e. methods `LessThan` (or `<`) and `IsEqual` (or `==`) returning logical results are needed. Again matching allows us to express exactly this quality. We can give a bound on the type of the objects, like `T <# Comparable`, where `Comparable` consists of the two mentioned methods and `T` is any type that has at least these methods, i.e. `T` matches `Comparable`. In other words, a type parameter is restricted via matching, also called match-bounded polymorphism or bounded matching [3]. To see how bounded matching solves the `animal` class problem in Figure 2 it is recoded in Figure 5. If the body of `eat` remains unchanged no further action is necessary. A `herbivore[planttype]` is now allowed to eat `grass`, `fruit`, etc. as long as these plants are matching `planttype`. We could also declare a `herbivore[fruit]` that is only allowed to eat `fruit`, etc. The new matching type in the subclass herbivore is applied to all occurences in the class and its inherited methods and variables. Note that under subtyping polymorphism this example would not be possible due to the subtyping rule for methods. In other words, `myfoodtype` has become a parameter that can be defined differently for each subclass without invalidating the matching of types. One should note that `herbivore[planttype]` is a subclass of `animal[planttype]` but not of `animal[foodtype]`. Bounded matching provides the programmer with a powerful syntax that allows to write some form of generic methods quite easily. Returning to the earlier case one can define a `CLASS :: Comparable` which has the methods `LessThan` and `IsEqual`, programme a sorting routine whose argument matches `Comparable`, and one can now sort any (heterogeneous) collection of objects that match `Comparable`, e.g. this could be applied to integer, real and character types.

[7]A more general form could be defined via `LIKE(<var>)` with `MYTYPE` as the special case `LIKE(SELF)`.

```
CLASS :: List[T]
    T      :: val
    MYTYPE :: next = NULL     ! next automatically becomes List[T]

    SUBROUTINE Insert( newval )
        T :: newval
        ...
    END SUBROUTINE Insert
    SUBROUTINE Delete( oldval )
        T :: oldval
        ...
    END SUBROUTINE Delete
    FUNCTION, LOGICAL :: Find( someval )
        T :: someval
        ...
    END FUNCTION Find
END CLASS List
```

Figure 6: List class without binary methods.

3.4 Hash-types

One can go one step further and introduce a new type construct, a so-called hash-type, written `#<type>`. An object will have type `#T` if it has any type that *matches* `T`. If `T` is an object type, then `#T` is also a type, and if an object `a` has type `S` with `S<#T`, then `a` has also type `#T`. We also have a form of subsumption with hash-types; if `S<#T` and `a` has type `#S`, then `a` has type `#T`. Hash-types can be used in type declarations as in

```
#T  :: x
#S  :: y     ! S matches T: S<#T
x = y        ! a valid assignment
```

ADA95 introduces a similar type (`T'Class`) that denotes a class `T` and its subclasses.

There is no need to pass around parameters as in bounded matching, an advantage in situations where one knows which type is to be matched, e.g. when dealing with various window types one can just write

```
FUNCTION Foo( w )
    #Window  :: w
    ...
END FUNCTION Foo
```

and `Foo` can only apply methods to `w` that match those of `Window`.

A parametrized class `List[T]`, defined with methods as in Figure 6 could receive the actual type `S`, making it a homogeneous list, or a hash-type `#S`, making it a heterogeneous list. Since hash-types are not compatible with binary methods this obviously requires that `List` has none, in particular `MYTYPE` cannot be present, otherwise only the homogeneous list is possible.

The examples in this chapter have shown that matching offers a great deal of expressiveness and flexibility in designing classes in a way that most programmers would choose intuitively. The notion of matching (see glossary) seems restrictive in that the type of the methods cannot be changed but `MYTYPE` is allowed and provides the flexibilty that is most often needed. The current definition of a `CLASS` makes sure that subclasses will always match the superclasses, i.e. inheritance is useful unlike under subtyping where useful subclasses may be impossible. Binary methods still require some care but at least they can be programmed in many cases of practical importance where subtyping would not allow them. Hash-types provide an easy way of handling heterogeneous data structures while bounded matching together with parametrized classes provides the tools to write generic classes. The proposed constructs, matching including bounds and hash-types, are quite elegant and simple, provide strong styping and still offer great expressiveness to any user. In summary, this would make F2000 a safe and practical tool for programming with more capabilities in the field of polymorphism than any widely used OOP language today can offer.

4 Critical Comments

Only one proposal (CWP with actually two parts, 97/196 and 97/230) has been presented to and already discussed by J3. It is actually this proposal that inspired me to present an alternative. Some criticisms that can be applied to the CWP are already directly or indirectly given in earlier chapters. In this chapter I will outline my main points of critique of the CWP and discuss the proposed `CLASS` construct.

4.1 Criticism of `CLASS`

1. A likely objection to `CLASS` is the introduction of a new word, `CLASS`, instead of using the available `TYPE`. One reason for this choice was to introduce the proposal without making reference to `TYPE` and prejudice it. F90/95 has already introduced some overlapping constructs like `SELECT` and `WHERE` with `IF` blocks, `NULL` with `NULLIFY`, so that this argument has lost its strength. A second reason is the use of names in the field of OOP. *Class* is the established name and the acceptance of object-oriented Fortran also outside the Fortran community will be much easier when the names are familiar, an argument that should not be overlooked. Thirdly, in a hybrid language like F2000 it is useful to keep procedural and object-oriented programming separate.
2. A class in OOP is such an important concept that this should be reflected in the language with its own name. Replacing `CLASS` with the pair `MODULE` and `TYPE` is a bad linguistic feature. A class is different from a module; a class has behaviour and an internal state which cannot be separated. Modules on the other hand are a tool of *physically* organizing code which includes limiting the visibility of its content. There is no particular meaning attached to a module unlike a class (one could merge two modules into one without any effect, the same process would change the meaning of a class).
3. The proposed `CLASS` has strong features. Incorporating them into `TYPE` will conflict with backward compatibility which would probably require to cut down on the proposed properties and hence weaken the OOP features in F2000. A strong `CLASS` construct is certainly preferable to a weak `TYPE`.
4. Why using the hash-notation instead of pointers? A hash-type has pointer properties but the main function is to indicate that it can only refer to objects of matching types while the word pointer is much too closely associated with actual memory handling. Secondly the notation is concise and will work as a parameter to generic classes which is impossible with a syntax that uses an explicit `POINTER` attribute. On the proposed `OBJECT` see below.

```
MODULE modA
    TYPE, EXTENSIBLE :: Atype
        ! data declarations
    CONTAINS
        PROCEDURE procA => procB
    END TYPE Atype
CONTAINS
    SUBROUTINE procB( btype, other_arguments )
        ! body
    END SUBROUTINE procB
END MODULE modA

USE modA
TYPE(Atype) :: x
CALL x%procA( other_arguments )
```

Figure 7: The extensible `TYPE` as proposed in [1] (in shortened form).

4.2 Proposals 97/196 and 97/230

The CWP [1] proposes as a basic building block to extend `TYPE` with the syntax as given by the example in Figure 7.

1. The current proposals does not promote an extensible `TYPE` into a programme unit, thus requiring always the tandem `MODULE` and `TYPE`, as in ADA95, with the resulting duplication of hierarchies as each sub-`TYPE` may be placed in a different `MODULE`. This lack of a proper separation between a `MODULE` and an extensible `TYPE` is a bad linguistic feature.

2. I have some serious reservations about the syntax of the CWP. Two names have to be given to name a method[8] for an extensible type (`procA` pointing to `procB`). The doubling of names is more confusing than helpful. It is argued that this syntax allows the user/programmer to gain a quick view of the structure of the `TYPE`. This neglects, however, that the argument lists of the public methods of a `TYPE` are also needed. Inheritance will (or should) usually lead to small classes that are extended bit by bit, and modern Fortran support already shorter constructs. Hence the above argument looses much of its strength. Similarities with ADA95 are quite obvious though ADA95's syntax seems to be stronger and more consistent to me, e.g. the interfaces of methods are present in the `TYPE` scope.

3. The second name of the method (`nameB`) is not protected against misuse unless the programmer explicitly declares it private. This design enables shortcuts that are generally considered unsafe. This unsafe design is exactly what should be avoided by default in any new feature in F2000.

4. The user can use the overriden methods of the superclass explicitly via type conversions, see example of `x%vector_2d%length()` in 97-230 where `x` is of `TYPE(vector_3d)` inherited from `TYPE(vector_2d)`. This is a special case of the general features introduced in items 7 (Type Enquiry) and 8 (Access to Extended Components) in 97-196. They hinder safe type-checking, obscure proper OOP and are very similar to constructs much criticized in C++. There is hardly a case when they are needed. The examples later on show a better solution using hash-types.

5. The definition of methods also introduces an asymmetry in their argument list, since in the *implementation* of a method the first argument has to be the object itself (`btype` in Fig. 7) while in the *call* of the method this argument is not present. This is at least inconsistent and in contrast with the rest of Fortran. The ADA95 solution, `CALL <class_method>( obj, args )`, is at least consistent though it denies the connection between object and method while retaining it for object and variable. The only consistent solution that does not treat variables and methods differently is to introduce the `SELF` construct.

6. Besides the extensible `TYPE` a construct `OBJECT` is introduced, and a variable `Var`, as in
   ```
   OBJECT(atype) :: Var,
   ```
   can have values that are subtypes of `atype`. This notion is confusing since variables declared with `TYPE` and `OBJECT` are actually all objects in OOP terminology. The term `OBJECT` here is intended primarily to declare polymorphic objects. In addition, `TYPE` can have the attribute `POINTER` which can only lead to more confusion. The hash-type introduced earlier for the `CLASS` version is similar but more general and avoids the confusion. I want to remind that genericity is hampered by a nonuniform type syntax.

7. The declarations could be made less verbose by dropping the words `TYPE` and `OBJECT` since any name different from the intrinsic Fortran types automatically implies a `TYPE`.[9] The name `OBJECT` could then be replaced by explicitly adding the `POINTER` attribute.

8. The distinction between `TYPE`, and `TYPE, EXTENSIBLE` is unnecessary; extension should be the default. The current `TYPE` is either completely public or completely private. I have argued above for a more differentiated and flexible syntax with a different default.

9. It not clear to me what form of polymorphism (subtyping, matching, multimethods) is actually intended. Genericity does not appear among the features. OOP without it is of limited use.

10. A complication for `TYPE` might arise from the current effort to parametrize `TYPE`. I do not know whether the current proposals harmonize or how they could affect a generic `TYPE`.

[8] I continue using the names like method as earlier on.

[9] Or are there parsing problems? Certainly not in F90 notation.

```
CLASS :: Point2D

    IMPLICIT NONE
    CREATE : Cart          ! Though PRIVATE Cart can be used for creation
    REAL  :: x=0.0, y=0.0

    REAL, FUNCTION :: Len
       Len = SQRT( SELF%x**2 +SELF%y**2 )   ! Example of SELF
    END FUNCTION Len
    REAL, FUNCTION :: Dist( p )
       MYTYPE :: p
       Dist = SELF%SubDist2( p )
       ! here equivalent to Dist = SQRT( (x-p%x)**2 +(y-p%y)**2 )
    END FUNCTION Dist
    REAL, FUNCTION :: SubDist2( p )
       #MYTYPE :: p
       SubDist2 = SQRT( (x -p%x)**2 +(y -p%y)**2 )
    END FUNCTION SubDist2
    MYTYPE, FUNCTION, PRIVATE :: Cart( X0, Y0 )  ! Could also specify
        REAL :: X0, Y0                           ! a Subroutine using
        x = X0 ; y = Y0                          ! polar Coordinates
        Cart = SELF
    END FUNCTION Cart
END CLASS Point2D
```

Figure 8: The `Point2D` class in the proposed `CLASS` syntax.

4.3 The Point Class Example

The examples of Figures 8, 9 and 10 are presented here to show the effect of `MYTYPE` and hash-types on the point example that is being discussed in J3 (see CWP and 98/103).

In my view the examples in CWP and 98/103 show still some confusion over polymorphism and inheritance. One source of the problem is that a short description of what properties the `Point` type should have is lacking, more concrete what should a method like `Dist` compute? Obviously the distance between two points (of the same dimension). But this becomes ambiguous in CWP and 98-103. Actually the Java example in 98-103 is not a proper version of inheritance since in Java the so-called signature of methods cannot be changed, i.e. restricted subtyping rules are used for polymorphism; overloading is used in 98-103 for class `Point_3d`. None of this is necessary. To discuss and demonstrate polymorphism the examples must be more clearly defined as will be done here.

I believe that the presented features of `MYTYPE` and hash-types allow to achieve a solution that demonstrates the intentions quite clearly. As a reminder, polymorphism means heterogeneous datastructures, however, this means that binary methods are excluded since the exact type of the object is not available before run-time, and testing for the actual type of an item in a heterogeneous list does not solve the problem posed by inheritance. This is one of the reasons why I did not provide for type conversions and enquiries. Homogeneous data present no problems in this respect since all data have the same type.

My definition of the point class example is given in Figures 8 and 9. `Dist` computes the distance between two points of the *same* type, a sensible definition. For two points with different types a method like `Dist` is not allowed. This makes sense as the distance between a `Point_2D` and a `Point_3D` is not defined. We can, however, define a subspace distance, i.e. project any `Point_3D` onto the xy-plane. This operation can be done for any n-dimensional point with $n \geq 2$. Note the difference between the argument declarations in `Dist` and `SubDist2`.

In Figure 10 the polymorphic array `q` is set up and all the operations are executed as one might expect. When `Len` is called the method called depends on the run-time type of `q` but there is no problem here.

```
CLASS :: Point3D
    IMPLICIT NONE
    INHERIT  : Point_2D
    REDEFINE : Dist, Len, Cart
    CREATE   : Cart
    REAL    :: z=0.0

    FUNCTION, REAL :: Len
       Len = SQRT( x**2 +y**2 +z**2 )
    END FUNCTION Len
    FUNCTION, REAL :: Dist( p )
       MYTYPE :: p
       Dist = SQRT( (x-p%x)**2 +(y-p%y)**2 +(z-p%z)**2 )
    END FUNCTION Dist
    MYTYPE, FUNCTION, PRIVATE :: Cart( X0, Y0, Z0 )
        REAL :: X0, Y0, Z0
        x = X0 ; y = Y0 ; z = Z0
        Cart = SELF
    END FUNCTION Cart
END CLASS Point3D
```

Figure 9: The `Point3D` class in the proposed `CLASS` syntax.

```
CLASS :: PointMain    ! or PROGRAM PointMain
    IMPLICIT NONE
    Integer   :: i
    Point_2D  :: p(3)
    #Point_2D :: q(3)
    Point_3D  :: r(3)

    p(1) = Point_2D%Cart( 1.0, 1.0 )
    p(2) = Point_2D%Cart( 2.0, 2.0 )
    p(3) = Point_2D%Cart( 1.0, 3.0 )
    q(1) = Point_2D%Cart( 1.0, 1.0 )          ! legal
    q(2) = Point_3D%Cart( 2.0, 2.0, 1.0 )     ! legal
    q(3) = Point_3D%Cart( 1.0, 2.0, 1.0 )     ! legal
    r(1) = Point_3D%Cart( 1.0, 1.0, 1.0 )
    r(2) = Point_3D%Cart( 2.0, 2.0, 2.0 )
    r(3) = Point_3D%Cart( 1.0, 2.0, 3.0 )

    DO i=1,2
       Write(*,*)  p(i+1)%Dist( p(i-1) )
       Write(*,*)  r(i+1)%Dist( r(i-1) )
       Write(*,*)  r(i+1)%SubDist2( p(i) ), p(i)%Len
       Write(*,*)  p(i+1)%SubDist2( q(i) ), q(i)%Len
       Write(*,*)  q(i+1)%SubDist2( r(i) ), r(i)%Len
       ! Write(*,*)  p(i+1)%Dist( q(i) )  ! illegal
       ! Write(*,*)  q(i+1)%Dist( p(i) )  ! illegal
    END DO
END CLASS PointMain
```

Figure 10: Using Point classes with homogeneous and heterogeneous datastructures.

5 The Module: Problems and Solutions

5.1 A Fortran Environment

There have been various critical comments on the current `MODULE` construct. The title for this section is a better indication of the underlying issue. Most of these comments have been concerned with

1. how to organize large codes (includes use of directory structure)
2. how to implement different versions with the same interface
3. how to combine efficiency with hiding of implementation details
4. how to compile efficiently (separation of interface and implementation)
5. how to find information (indexing, documentation)

Answers to these questions are important, but the language cannot provide it all. Certain tools must be provided that ease or even remove these problems. Unfortunately very little help so far has come from the Fortran vendors. At least three approaches are possible:

1. extend the `MODULE` syntax
2. add a small management syntax
3. leave it to the compiler vendors to provide integrated tools

Advantages and disadvantages of these solutions:

- The leave-it-to-the-vendor solution will leave users with different versions of variable quality. A rather uniform appearance across platforms is certainly much more appealing. No particular action from the Fortran committees is required.
- Extending the `MODULE` syntax including separation of interface and implementation. This requires a certain amount work from the committees (and has to be approved?). It also will add to the workload of programmers, especially since the interface information is already determined through attributes. Visibility can be made more fine-grained, which in turn allows code to be split over several modules. It does not address all questions, in particular how to find information about a module, class or method and cannot take advantage of the directory structure. It is very questionable whether syntax elements for purely organisational purposes should be part of the language at all.
- A management syntax added to the language as a separate part. This can take advantage of the file structure of the platform so that files from different directories can be accessed easily. An extension of the `PRIVATE` construct can be used for fine-grained access. The compiler will read a control file with the necessary information. Again, this needs approval and as a completely new feature will receive objections. Compilers have to be smart enough, the notorious quality of implementation.

Whatever path is taken a certain uniformity should be achieved. Many high quality tools already exist on many platforms that can help in this respect, e.g. GUI, syntax-highlighting editors (emacs,improved vi) and hypertext browsers.[10] The Fortran standard committees have not concerned themselves with these issues of an 'enviroment' but I personally believe this attitude should be reconsidered. Some languages provide more sophisticated tools (compilers for ADA95, environments for Eiffel and Beta). It would be useful to gain more information about the practical experiences from users of these languages on this topic.

5.2 Module Extensions

Van Snyder has recently proposed a syntax for modules that extends modules by adding junior and child modules [10]. I am concerned about the close ties that are made between `MODULE` and OOP on the one hand and providing a more fine grained visibility meachanism on the other hand. The `MODULE` becomes loaded with so many different features that the modular structure of the language suffers while the learning curve becomes steeper and steeper.

There are easier ways to achieve the same effects. First, a more fine-grained visiblity can be gained by extending `PRIVATE` to include an access list of privileged `MODULE`s that can see internal details of other `MODULE`s. This would be a very simple and compatible solution which is at least as flexible as child and junior parts. Secondly, compilers can and *must* automatically extract the interface including the privileged access statements and use this for an efficient compilation. This has advantages for the programmer: all code is kept in one place since there is no physical separation of code and interface, which

[10] See e.g. `http://daisy.uwaterloo.ca/ eddemain/f90doc/`.

will be less error-prone due to forgotten updates of the interface. A parser/compiler can this much more efficiently and safely. The same procedure can be used to provide library-style interface documentation accessible to browsers or the like.

Fortran is the only language with modules (or packages) that does not qualify the names of variables and methods by the name of the module. This policy should be reconsidered very seriously since the clash of names can only be avoided by renaming, which compares rather unfavourably with the mechanism, for example, in ADA95.

6 Conclusion

I have presented a rather comprehensive proposal for OOP in F2000 in form of a `CLASS` construct that is safe via strong typing, is simple and at least as expressive as most OOP language with some form of type checking. The important OOP principles are observed and their abuse made difficult. The proposal goes beyond what is currently considered but provides a much more satisfying and consistent solution that leaves room for further development of the `CLASS` construct and does not overload the `MODULE` with syntax and semantics. The extension of OOP syntax to existing elements has been proposed to make OOP as seamless as possible with current Fortran. Some proposals for a simplified syntax have been added, and the possible extension of modules has been discussed. The current proposal for OOP in Fortran has many merits that should again defy the so often predicted imminent demise of Fortran. Programmers can develop their software along the more traditional structured programming lines or along the newer OOP method, using whatever is most appropriate for the problem at hand.

PS: I would appreciate if one of the committee members could present me with a summary of the discussion of my proposal since I will be unable to attend the meeting.

References

[1] M. Cohen, J3/97-196 and J3/97-230 and references therein.
`ftp://ftp.dfrc.nasa.gov/pub/x3j3/ncsa/doc/standing/010.html`

[2] W. B. Clodius, *Critical Issues for Object Orientation in Fortran*, J3/97-119, 1997.

[3] K. Bruce, *Typing in Object-oriented Languages: Achieving Expressiveness and Safety*, `http://www.cs.williams.edu/~kim/README.html`

[4] B. Meyer, *Object-oriented Software Construction*, New York, Prentice-Hall 1997.

[5] language in the public domain. `http://www.icsi.berkeley.edu/~sather/`

[6] L. O. Madsen, B. Møller-Pdersen, K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*, Wokingham, Addison-Wesley, 1993.

[7] N. H. Cohen, *ADA as a second Language*, New York, McGraw-Hill, 1996.

[8] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, B. Pierce, *On Binary Methods*, `http://www.cs.williams.edu/~kim/README.html`

[9] L. E. Petersen, *A Module System for $\mathcal{LOOM}$*, Thesis, Williams College, Williamstown 1997.
`http://www.cs.williams.edu/~kim/Theses.html`

[10] V. Snyder, *Enhancing Modules*, J3/98-104,105,106.
`ftp://ftp.dfrc.nasa.gov/pub/x3j3/ncsa/doc/meeting/144/`

A Glossary of Terms

The terminology in the field of object-oriented programming (design, analysis) can be quite confusing which is due to the independent development of ideas and only a slow emergence of a standard notation. The following definitions were mostly collected from [3] with $m : \tau$ denoting a method m of type τ.

class:	code defining variables and methods

subclass: code that extends a *class* (called superclass) and inherits the methods and variables from the superclass.

object: instance of a *class*; *object* and method are sometimes also called receiver and message.

self: name for the receiver of a message (i.e. the current *object*), only used inside a class; meaning of *self* changes with *subclass*

type: short for *object type* or *interface type*; contains the names of an *object*'s methods and the *types* of each method's arguments and results; a *type* is always the most specific one for an *object*.[11]

subtype: (informal) a *type* σ is a *subtype* of a *type* τ, written $\sigma <: \tau$, and for *object types*:

$$\begin{array}{l} \mathit{ObjectType}\{m_j : \sigma_j\}_{1\le j\le k} <: \mathit{ObjectType}\{m_i : \tau_i\}_{1\le i\le n} \ \text{ if } \ n \le k \\ \text{with } \ \sigma_i <: \tau_i \ \text{ for each } \ i \le n. \end{array}$$

An expression of *type* σ can be used in any context that expects an expression of *type* τ. In other words, any expression of *type* σ can masquerade as an expression of *type* τ. This usually expressed as the rule of

subsumption: (subtype polymorphism); if $\sigma <: \tau$ and a programme fragment has *type* σ it also has *type* τ

subtyping: (of methods) rule: $\sigma \to \tau <: \sigma' \to \tau'$ if and only if $\sigma' <: \sigma$ and $\tau <: \tau'$; also called the contravariance rule. Note that the **argument** changes in a contravariant way, and hence both functions and subroutines are affected.

mytype: denotes the *type* of the receiver of a message with the understanding that its meaning is variable in *subclasses* in accordance with *self*

matching: short for match-bounded polymorphism; an *object type* matches another, written $\sigma <\# \tau$, if the first *type* has at least the methods of the second considering *mytype* in one to be "the same" as *mytype* in the other. A formal definition is:

$$\mathit{ObjectType}\{m_i : \tau_i\}_{1\le i\le k} \ <\# \ \mathit{ObjectType}\{m_i : \tau_i\}_{1\le i\le n} \ \text{ iff } \ n \le k$$

where the first n method types in *ObjectTypes* may not be changed , but occurences of *mytype* mark places where a *type* changes meaning automatically.

binary method: a method of *type* τ that has an argument of *type* τ; a wider definition includes also other arguments of same or different *type*

[11] Note that methods or `SUBROUTINE`s have a type *unit* that is supertype to all types.