E box 11

NORSK REGNESENTRAL

NORWEGIAN COMPUTING CENTER

CLASS AND SUBCLASS DECLARATIONS

by

Ole-Johan Dahl

and

Kristen Nygaard

E2l2c

437079

Ebot 11

NORWEGIAN COMPUTING CENTER,
Forskningsveien 1 B,
Oslo 3 - Norway.

CLASS AND SUBCLASS DECLARATIONS

by

Ole-Johan Dahl
and
Kristen Nygaard

UTGÅTT AV
UNIVERSITETSBIBLIOTEKET
I BERGEN
Dato: 25/6 2015 Sign.: SMi

Presented at IFIP Working Conference on Simulation Languages, Oslo, May, 1967.

UNIVERSITAS BERGENSIS

CLASS AND SUBCLASS DECLARATIONS.

1. Introduction.

A central idea of some programming languages [1,2,3] is to provide protection for the user against (inadvertantly) making meaningless data references. The effects of such errors are implementation dependant and cannot be determined by reasoning within the programming language itself. This makes debugging difficult and impractical.

Security in this sense is particularly important in a list processing environment, where data are dynamically allocated and de-allocated, and the user has explicit access to data addresses (pointers, reference values, element values). To provide security it is necessary to have an automatic de-allocation mechanism (reference count, garbage collection). It is convenient to restrict operations on pointers to storage and retrieval. New pointer values are generated by allocation of storage space, pointing to the allocated space. The problem remains of correct interpretation of data referenced relative to user specified pointers, or checking the validity of assumptions inherent in such referencing. E.g. to speak of "A of X" is meaningful, only if there is an A among the data pointed to by X.

The record concept proposed by C.A.R.Hoare and N. Wirth [3] provides full security combined with good runtime efficiency. Most of the necessary checking can be performed at compile time. There is, however, a considerable expense in flexibility. The values of reference variables and procedures must be restricted by declaration to range over records belonging to a stated class. This is highly impractical.

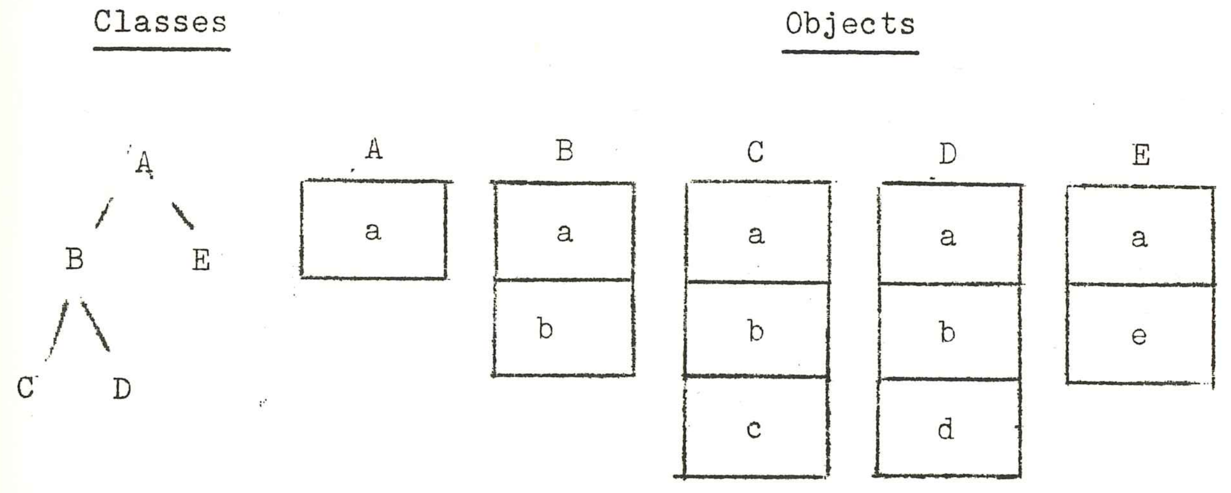
The connection mechanism of SIMULA combines full security with greater flexibility at a certain expense in convenience and run time efficiency. The user is forced, by the syntax of the connection statement, to determine at run time the class of a referenced data structure (process) before access to the data is possible.

The subclass concept of Hoare [4] is an attempt to overcome the difficulties mentioned above, and to facilitate the manipulation of data structures, which are partly similar, partly distinct. This paper presents another approach to subclasses, and some applications of this approach.

2. Classes.

The class concept introduced is a remodelling of the record class concept proposed by Hoare. The notation is an extension of the ALGOL 60 syntax. A prefix notation is introduced to define subclasses organized in a hierarchical tree structure. The members of a class are called objects. Objects belonging to the same class have similar data structures. The members of a subclass are compound objects, which have a prefix part and a main part. The prefix part of a compound object has a structure similar to objects belonging to some higher level class. It can itself be a compound object.

The figure below indicates the structure of a class hierarchy and of the corresponding objects. A capital letter denotes a class. The corresponding lower case letter denotes the data comprising the main part of an object belonging to that class.



B, C, D, E are subclasses of A; C and D are subclasses of B.

2.1 Syntax.

```
<class id.>::=<identifier>
<prefix>::=<class id.>
<class body>::=<statement>
<main part>::=class <class id.> <formal parameter part>;
              <specification part> <class body>
<class declaration>::=<main part>|<prefix> <main part>
```

2.2 Semantics.

An object is an instance of a class declaration. Different instances of the same declaration are said to belong to class C, where C is the class identifier. If the class body does not take the form of an unlabelled block, it acts as if enclosed in an implicit block. The parameters and the quantities declared local to the outermost block of the class body are called the attributes of an object. The attributes can be referenced locally from within the class body, or nonlocally by a mechanism called remote accessing (5).

The parameters are transmitted by value. One possible use of the statements of the class body may be to initialize attribute values.

A prefixed class declaration represents the result of concatenating the declaration referenced by the prefix and the main part. The concatenation is recursively defined by the following rules.

1) The formal parameter lists of the former and the latter are concatenated to form one parameter list.

2. The specification parts are juxtaposed.

3) A combined class body is formed, which is a block, whose block head contains the attribute declarations of the prefix body and the main body. The block tail contains the statements of the prefix body followed by those of the main body.

The attributes of the main part are not accessible from within the prefix body, except by remote accessing. The attributes of the prefix are accessible as ordinary local quantities from within the body of the main part.

The object class represented by a prefixed class declaration is a subclass of the class denoted by the prefix. Subclasses can be nested to any depth by using prefixed class identifiers as prefixes to other class declarations.

Let A_0 be any class. If A_0 is prefixed, we will denote this prefix by A_1. The prefix of A_1 (if any) will be denoted by A_2 etc.
The sequence

$$A_1, A_2, \ldots\ldots\ldots$$

will be called the "prefix sequence" of A_0. It follows from the syntax that if A_i and A_j both have A_k as prefix, they have identical prefix sequences.

It will be required that all prefix sequences are finite. (This excludes multiple occurrence of any class A_i in a prefix sequence.)
Let

$$A_1, A_2 \ldots, A_n$$

be the prefix sequence of A_0. We shall say that the class A_i is "included in A_j" if $0 \leq i \leq j \leq n$.

3. Object References.

Reference values in the sense of [4] are introduced, in a slightly modified form.

3.1 Reference types.

3.1.1 Syntax.

```
<type>::=<ALGOL type>|ref|ref <qualification>
<qualification>::=(<class id.>)
```

3.1.2 Semantics.

Associated with each object is a unique value of type ref, which is said to reference or point to the object. A reference value may, by qualifying a declaration or specification by a class identifier, be required to refer to objects belonging to either this class or any of its subclasses. In addition the value of any item of type reference is restricted to objects belonging to classes whose declarations are statically visible from the declaration or specification of the item.

The reference value none is a permissible value for any reference item, regardless of its qualification.

3.2 Reference Expressions.

3.2.1 Syntax.

```
<simple ref. expr.>::=none|<variable 1>|<function designator>|
                       <object designator>|<local reference>
<ref. expr.>::=<simple ref. expr.>|if <Boolean expr.> then
               <simple ref. expr.> else <ref. expr.>
<object designator>::=<class id.> <actual parameter part>
<local reference>::=this <class id.>
```

3.2.2 Semantics.

A reference expression is a rule for computing a reference value. Thereby reference is made to an object, except if the value is none, which is a reference to "no object".

3.2.2.1 Qualification.

A variable or function designator is qualified according to its declaration or specification. An object designator or local reference is qualified by the stated class identifier. The expression "none" is not qualified.

No qualification will be regarded as qualification by a universal class, which includes all declared classes.

3.2.2.2 Object generation.

As the result of evaluating an object designator an object of the stated class is generated. The class body is executed. The value of the object designator is a reference to the generated object. The life span of the object is limited by that of its reference value.

3.2.2.3 Local reference.

A local reference "<u>this</u> C" is a meaningful expression within the class body of the class C or of any subclass of C. Its value is a reference to the current instance of the class declaration (object).

Within a connection block (5.2) connecting an object of class C or a subclass of C the expression "<u>this</u> C" is a reference to the connected object.

The general rule is that a local reference refers to the object, whose attributes are local to the smallest enclosing block, and which belongs to a class included in the one specified. If there is no such object, the expression is illegal.

4. Reference operations.

4.1 Assignment.

4.1.1 Syntax.

<variable 1>::=<reference expr.>

4.1.2 Semantics.

Let the left and right hand sides be qualified by Cl and Cr, respectively, and let the value of the right hand side be a reference to an object of class Cv. The legality and effect of the statement depends on the relations that hold between these classes.

Case 1. Cl includes Cr: The statement is legal, and the assignment is carried out.

Case 2. Cl is a subclass of Cr: The statement is legal, and the assignment is carried out if Cl includes Cv, or if the value is none. If Cl does not include Cv, the effect of the statement is undefined (cf. 6.1).

Case 3. Cl and Cr satisfy neither of the above relations: The statement is illegal.

The following additional rule is considered for case 1: The statement is legal if and only if the declaration of the left hand item (variable, array or <type> procedure) is within the scope of the class identifier Cr and all its subclasses. (The scope is in this case defined after having effected all concatenations implied by prefixes.)

This rule would have the following consequences.

1) Accessible reference values are limited to pointers to objects, whose attributes are accessible by remote referencing (5).

2) Classes represented by declarations local to different instances of the same block are kept separate.

3) The security problems are simplified.

4.2. Relations.

4.2.1 Syntax.

```
<relation>::=<ALGOL relation>|<reference expr.>=<reference expr.>|
             <reference expr.>≠<reference expr.>|
             <reference expr.> is <class id.>
```

4.2.2 Semantics.

Two reference values are said to be equal if the point to the same object, or if both are none. A relation "X is C" is true if the object referenced by X belongs to the class C or to any of its subclasses.

4.3 For Statements.

4.3.1 Syntax.

<for list element>::= <ALGOL for list element>|<reference expr.>|
<reference expr.> while <Boolean expr.>

4.3.2 Semantics.

The extended for statement will facilitate the scanning of list structures.

5. Attribute Referencing.

An attribute of an object is identified completely by the following items of information:

1) the value of a <reference expr.> identifying an object,

2) a <class id.> specifying a class, which includes that of the object, and

3) the <identifier> of an attribute declared for objects of the stated class.

The class identification, item 2, is implicit at run time in a reference value, however, in order to obtain runtime efficiency it is necessary that this information is available to the compiler.

For a local reference to an attribute, i.e. a reference from within the class body, items 1 and 2 are defined implicitly. Item 1 is a reference to the current instance (i.e. object), and item 2 is the class identifier of the class declaration.

Non-local (remote) referencing is either through remote identifiers or through connection. The former is an adaptation of the technique proposed in [2], the latter corresponds to the connection mechanism of SIMULA [1].

5.1 Remote Identifiers.

5.1.1 Syntax.

<remote identifier>::=<reference expr.>.<identifier>
<identifier 1>::=<identifier>|<remote identifier>

Replace the meta-variable <identifier> by <identifier 1> at appropriate places of the ALGOL syntax.

5.1.2 Semantics.

A remote identifier identifies an attribute of an individual object. Item 2 above is defined by the qualification of the reference expression. If the latter has the value <u>none</u>, the meaning of the remote identifier is undefined (cf. 6.2).

5.2 Connection.

5.2.1 Syntax.

```
<connection block 1>::=<statement>
<connection block 2>::=<statement>
<connection clause>::= when <class id.> do <connection block 1>
<otherwise clause>::=<empty>| otherwise <connection block 2>
<connection part>::=<connection clause>|
                    <connection part> <connection clause>
<connection statement>::= inspect <reference expr.> do
                          <connection block 2>|
                          inspect <reference expr.>
                          <connection part> <otherwise clause>
```

5.2.2 Semantics.

The connection mechanism serves a double purpose:

1) To define item 1 above implicitly for attribute references within connection blocks. The reference expression of a connection statement is evaluated once and its value is stored. Within a connection block this value is said to reference the connected object. It can itself be accessed through a <local reference> (3.2.2.3).

2) To discriminate on class membership at run time, thereby defining item 2 implicitly for attribute references within alternative connection blocks. Within a <connection block 1>

item 2 is defined by the class identifier of the connection clause. Within a <connection block 2> it is defined by the qualification of the reference expression of the connection statement.

Attributes of a connected object are thus immediately accessible through their respective identifiers, as declared in the class declaration corresponding to item 2. These identifiers act as if they were declared local to the connection block. The meaning of such an identifier is undefined, if the corresponding <local reference> has the value none. This can only happen within a <connection block 2>.

6. Undefined Cases.

In defining the semantics of a programming language the term "undefined" is a convenient stratagem for postponing difficult decisions concerning special cases for which no obvious interpretation exists. The most difficult ones are concerned with cases, which can only be recognized by runtime checking.

One choice is to forbid offending special cases. The user must arrange his program in such a way that they do not occur, if necessary by explicit checking. For security the compiled program must contain implicit checks, which to some extent will duplicate the former. Failure of a check results in program termination and an error message. The implicit checking thus represents a useful debugging aid, and, subject to the implementor's foresight, it can be turned off for a "bugfree" program (if such a thing exists).

Another choice is to define ad hoc, but "reasonable" standard behaviours in difficult special cases. This can make the language much more easy to use. The programmer need not test explicitly for special cases, provided that the given ad hoc rule is appropriate in each situation. However, the language then has no implicit debugging aid for locating unforeseen special cases (for which the standard rules are not appropriate).

In the preceding sections the term undefined has been used three times in connection with two essentially different special cases.

6.1 Conflicting reference assignment.

Section 4.1.2, case 2, Cl does not include Cv: The suggested standard behaviour is to assign the value none.

6.2 Non-existing attributes.

Sections 5.1.2 and 5.2.2: The evaluation of an attribute reference, whose item 1 is equal to none, should cause an error printout and program termination. Notice that this trap will ultimately catch most unforeseen instances of case 6.1.

7. Examples.

The class and subclass concepts are intended to be general aids to data structuring and referencing. However, certain widely used classes might well be included as specialized features of the programming language.

As an example the classes defined below may serve to manipulate circular lists of objects by standard procedures. The objects of a list may have different data structures. The "element" and "set" concepts of SIMULA will be available as special cases in a slightly modified form.

```
class linkage; begin ref (linkage) suc, pred; end linkage;
linkage class link; begin
  procedure out; if suc ≠ none then
    begin pred. suc:= suc; suc. pred:= pred;
          suc:= pred:= none end out;
  procedure into (L); ref (list) L;
    begin if suc ≠ none then out;
          suc:= L; pred:= suc. pred;
          suc. pred:= pred. suc:= this linkage end into;
  end link;
linkage class list;
  begin suc:= pred:= this linkage end list;
```

Any object prefixed by "link" can go in and out of circular lists. If X is a reference expression qualified by link or a subclass of link, whose value is different from none, the statements

X. into (L) and X. out

are meaningful, where L is a reference to a list.

Examples of user defined subclasses are:

```
link class car (license number, weight);
       integer license number; real weight; .....;
car class truck (load); ref (list) load; .......;
car class bus (capicity); integer capacity;
       begin ref (person) array passenger [1 : capacity] ...end;
list class bridge; begin real load; .....end;
```

Multiple list memberships may be implemented by means of auxiliary objects.

```
link class element (X); ref  X ;;
```

A circular list of element objects is analogous to a "set" in SIMULA. The declaration "set S" of SIMULA is imitated by "ref (list) S" followed by the statement "S:= list".

The following are examples of procedures closely similar to the corresponding ones of SIMULA.

```
procedure include (X, S); value X; ref X; ref (list) S;
if X ≠ none then element (X). into (S);
ref (linkage) procedure suc (X); value X; ref (linkage) X;
       suc:= if X ≠ none then X. suc else none;
ref (link) procedure first (S); ref (list) S;
       first:= S. suc;
Boolean procedure empty (S); ref (list) S;
       empty:= S. suc = S;
```

Notice that for an empty list S "suc (S)" is equal to S, whereas "first (S)" is equal to none. This is a result of rule 6.1 and the fact that the two functions have different qualifications.

8. Extensions.

8.1 Prefixed Blocks.

8.1.1 Syntax.

```
<prefixed block>::=<block prefix> <main block>
<block prefix>::=<object designator>
<main block>::=<unlabelled block>
<block>::=<ALGOL block>|<prefixed block>|
          <label>:<prefixed block>
```

8.1.2 Semantics.

A prefixed block is the result of concatenating (2.2) an instance of a class declaration and the main block. The formal parameters of the former are given initial values as specified by the actual parameters of the block prefix. The latter are evaluated at entry into the prefixed block.

8.2 Concatenation.

The following extensions of the concepts of class body and concatenation give increased flexibility.

8.2.1 Syntax.

```
<class body>::=<statement>|<split body>
<split body>::=<block head>;<part 1> inner; <part 2>
<part 1>::=<empty>|<statement>;<part 1>
<part 2>::=<compound tail>
```

8.2.2 Semantics.

If the class body of a prefix is a split body, concatenation is defined as follows: the compound tail of the resulting class body consists of part 1 of the prefix body, followed by the statements of the main body, followed by part 2 of the prefix body. If the main body is a split body, the result of the concatenation is itself a split body.

For an object, whose class body is a split body, the symbol <u>inner</u> represents a dummy statement. A class body must not be a prefixed block.

8.3 Example.

As an example on the use of the extended class concept we shall define some aspects of the SIMULA concepts "process", "main program", and "SIMULA block".

Quasi-parallel sequencing is defined in terms of three basic procedures, which operate on a system variable SV. SV is an implied and hidden attribute of every object, and may informally be characterized as a variable of "type label". Its value is either null or a program point [5]. SV initially contains the "exit" information which refers back to the object designator. The three basic procedures are:

1) detach. The value of SV is recorded, and a new value, called a reactivation point, is assigned referring to the next statement in sequence. Control proceeds to the point referenced by the old value of SV. The effect is undefined if the latter is null.

2) resume(X); ref X. A new value is assigned to SV referring to the next statement in sequence. Control proceeds to the point referenced by SV of the object X. The effect is undefined if X.SV is null or if X is none. null is assigned to X.SV.

3) goto(X); ref X. Control proceeds to the point referenced by SV of the object X. The effect is undefined if X.SV is null or if X is none. null is assigned to X.SV.

```
class SIMULA; begin
  ref(process)current;
  class process; begin ref(process)nextev; real evtime;
        detach; inner; current:=nextev; goto(nextev)end;
  procedure schedule(X,T); ref(process)X; realT;
              begin X.evtime:=T; ------------ end;
  process class main program; begin
              L:resume(this SIMULA); go to L end;
  schedule(main program, 0)end SIMULA;
```

The "sequencing set" of SIMULA is here represented by a simple chain of processes, starting at "current", and linked by the attributes "nextev". The "schedule" procedure will insert the referenced process at the correct position of the chain, according to the assigned time value. The details have been omitted here.

The "main program" object is used to represent the SIMULA object within its own sequencing set.

Most of the sequencing mechanisms of SIMULA can, except for the special syntax, be declared as procedures local to the SIMULA class body.

Examples:

```
procedure passivate; begin current:=current.nextev;
                                 resume(current)end;
procedure activate(X); ref X; inspect X when process do
  begin nextev:=current; evtime:=current.evtime;
        current:=this process; resume(current)end;
procedure hold(T); real T; inspect current do
  begin current:=nextev; schedule(this process, evtime+T);
        resume(current)end;
```

Notice that the construction "process class" can be regarded as a definition of the symbol "activity" of SIMULA. This definition is not entirely satisfactory, because one would like to apply the prefix mechanism to the activity declarations themselves.

9. Conclusion.

The authors have for some time been working on a new version of the SIMULA language, tentatively named SIMULA 67. A compiler for this language is now being programmed and others are planned. The first compiler should be working by the end of this year.

As a part of this work the class concept and the prefix mechanism have been developed and explored. The original purpose was to create classes and subclasses of data structures and processes. Another useful possibility is to use the class concept to protect whole families of data, procedures, and subordinate classes. Such families can be called in by prefixes. Thereby language "dialects" oriented towards special problem areas can be defined in a convenient way. The administrative problems in making user defined classes generally available are important and should not be overlooked.

Some areas of application of the class concept have been illustrated in the preceding sections, others have not yet been explored. An interesting area is input/output. In ALGOL the procedure is the only means for handling I/O. However, a procedure is generated by the call, and does not survive this call. Continued existence, and existence in parallel versions is wanted for buffers and data defining external layout, etc. System classes, which include the declarations of local I/O procedures, may prove useful.

The SIMULA 67 will be frozen in June this year, and the current plan is to include the class and reference mechanisms described in sections 2 - 6. Class prefixes should be permitted for activity declarations. The "element" and "set" concepts of SIMULA will be replaced by appropriate system defined classes. Additional standard classes may be included.

SIMULA is a true extension of ALGOL 60. This property will very probably be preserved in SIMULA 67.

References.

1. O.-J. Dahl and K. Nygaard: "SIMULA - a Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual". Norwegian Computing Center, Oslo 1965.

2. C.A.R. Hoare: "Record Handling". ALGOL Bulletin 21, November 1965.

3. C.A.R. Hoare, N. Wirth: "A Contribution to the Development of ALGOL". Comm. of the ACM, June 1966.

4. C.A.R. Hoare: "Record Handling". Lectures delivered at the NATO Summer School, Vilard-de-Lans, September 1966.

5. P. Naur: "The Design of the GIER ALGOL Compiler" BIT. Vol. 3, no. 2,3.

6. P. Naur, ed: "Report on the Algorithmic Language ALGOL 60".

Class and Sub-Class Declarations.

Abstract

by O.-J. Dahl and K. Nygaard.

The remote referencing of attributes of objects (processes) in SIMULA is awkward whenever processes belong to similar, but not identical classes (activities). A class declaration is proposed as a remedy and a possible extension of SIMULA.

For processes with similar attributes the latter are grouped together by a class declaration. They are included among the process attributes by prefixing the activity declaration by the appropriate class identifier. A class idenrifier can also be used as a prefix to another class declaration.

Reference variables in the sense of C.A.R. Hoare are introduced. The element concept of SIMULA can normally be dispensed with. Standardized set operation can be achieved by using the (system defined) class identifier "link" as prefix to activity and class declarations, where

```
class link; begin reference suc, pred; end;
```

The concept of system defined classes seems to be an important means of enriching a general purpose language and also for structuring it towards specialized fields of application.