Object Pascal

For the Macintosh

Larry Tesler
February 14, 1985

Introduction

The specification of the language *Object Pascal* appears in the *Object Pascal Report*, attached. The present memo specifies differences between the current Macintosh implementation and the specification. In cases noted as "temporary deviations", we intend to bring the implementation into conformance shortly.

Multi-Level "`inherited`"

According to the standard, the statement "inherited Draw" activates a method of the immediate ancestor of the type whose method contains the statement. In the Macintosh version, if a more remote ancestral type, `T`, has a different implementation of the same method, then that method can be invoked by `T.Draw`. This feature is rarely used.

Object Type Declarations

According to the standard, an object type declaration may appear in a main program, in the interface of a unit, or in the implementation of a unit. In the Macintosh version, it can only appear in the interface of a unit. Method declarations can only appear in the implementation of the same unit. These deviations are temporary.

According to the standard, it is an error if a method is declared override in the type declaration and then not implemented, or if an override method is implemented but not declared override. In the Macintosh version, the compiler does not check for these errors. This is a temporary deviation.

According to the standard, the name of a method in a method header is qualified by the type name when the method body is introduced but not in the type declaration. In the Macintosh version, qualification is permitted in the type declaration.

Range Checking

According to the standard, it is an error to coerce a value to an object type if it is not in the domain of that type. In the Macintosh version, if and only if the compiler switch {$R+} is in effect, the compiler generates code to test the validity of object-type coercions.

Unsafe Use of Handles

According to the standard, the implementation of a reference is not specified, and may be machine dependent. In the Macintosh version, an object reference is implemented as a handle, i.e., as a pointer to a pointer to an object. The object itself can change its memory address during execution as part of a compaction process that is invoked to prevent heap fragmentation.

Because an object can change its location, it is unsafe to save a pointer to it in a variable or register and then later to access the object through the saved pointer. If a procedure call should intervene, then the heap might compact to make room for a newly allocated object or to swap the procedure itself into memory, invalidating the pointer.

Where possible, the compiler takes care to access objects only through handles and not through pointers. Examples:

```
type
    T = object
        F:  Integer;
        S:  String[10];
    end;
var Y:  T;
Y.F := Func;
Proc(Y.F);
with Y do
    begin
    F := Func;
    Proc(F);
    F := F + 1;
    DrawString(S);
end;
```

The assignments `Y.F := Func` and `F := Func` are safe even if activation of `Func` causes heap compaction.

The statements `Proc(Y.F)` and `Proc(F)` are safe unless the parameter is declared as `var` and activation of `Proc` causes compaction, in which case they are unsafe and the compiler will report an error.

The assignment `F := F + 1` is safe even if the preceding activation of `Proc(F)` caused compaction.

The statement `DrawString(S)` is unsafe if `DrawString` causes compaction, but the compiler will not report an error because `DrawString` does not declare its argument as a `var` parameter even though it treats it as such.

If you receive an error message from the compiler about an unsafe `var` parameter, it is best to pass a local variable instead and to follow the procedure call by an assignment from the local to the field. However, if you are certain that the called procedure and every procedure it will call are resident in memory and that none of them will compact the heap, or if you have temporarily locked the object whose field is the parameter, then you can suppress the complaint using the compiler switch `{$H-}`. Example:

```
{$H-} SetRect(Obj.bounds, 0. 0, 100, 200); {$H+}
```

Repetition of Parameter Lists

In Standard Pascal, it is not permitted to repeat the parameter list of a procedure when its body is introduced. In the Macintosh version, repetition of the parameter list is permitted for methods as well as for ordinary procedures and functions. The parameter names, types, and order (and function result type, if any) must agree exactly.

Word Symbols

According to the standard, `object` and `inherited` are word symbols and may not be redeclared. To protect the large body of software written in Workshop Pascal, the Macintosh version may be more forgiving. The compiler may allow a programmer to declare the identifiers `object` and `inherited` in a program or unit that does not have any object type declarations.

Names of Predeclared Functions

The following predeclared functions have non-standard names in the Macintosh version. `Member` is presently named `InClass`. `New` and `Dispose` when applied to object types are named `NewObject` and `DisposeObject`. These deviations are temporary.

Object Pascal vs. Lisa Clascal

by Larry Tesler
February 13, 1985

Object Pascal (described in *Object Pascal Report*, attached) is a revision of Lisa Clascal designed by Apple Computer's Macintosh Software Group with the help of Niklaus Wirth. Here I list the differences between the languages, and rationalize them.

An earlier set of memos described a proposed language, Clascal-85, that was a much larger departure from Lisa Clascal. Prof. Wirth convinced us that most of the changes proposed therein were in the opposite direction than desired.

Summary of Changes

The following syntactic constructs have changed:

Lisa Clascal (Old)		Object Pascal (New)
superself.< identifier >	->	**inherited** < identifier >
subclass of nil	->	**object**
subclass of c	->	**object** (c)
IsClass	->	**Member**
procedure <method name> ...	->	**procedure** c.<method name>
function <method name> ...	->	**function** c.<method name>

The last two changes above only apply in the implementation of a method.

The following are language additions (the first one has not been implemented yet):

- Classes may be declared in an **implementation** or in a **program**.
- **New** and **Dispose** are used to allocate and deallocate objects.
- **object** and **inherited** are reserved words.

The following are language deletions:

- **methods of** c and the matching **end**.
- **subclass, superself , abstract, default, thisclass**, and **Create**.

Also note new freedoms:

- The method implementations of a class need not be grouped together.
- "self." may be omitted.

and a new restriction:

- A name declared in any method of a type may not be the same as the name of a field or method of that type.

Rationale for Changes

The term *class* has been purged from the vocabulary.

- Niklaus Wirth urged this change to reduce the number of concepts: a class is simply an object type. I like the change because students used to confuse classes with objects.

One problem with the change is that Objective C and Smalltalk documentation use the term *class*. In Kurt Schmucker's book, *Object-oriented Programming for the Macintosh*, he can equate the term "class" with "object type".

We use *descendant* and *ancestor* instead of *subclass* and *superclass*.

- The term "class" is no longer used..
- Students seem to have difficulty with the *subclass/superclass* terminology. They often point out that a subclass has "more" properties than its superclass and yet "sub" implies it has less. Pointing out the analogy with set theory does not seem to straighten out their confusion.

In Object Pascal, "`with self do`" is implicit in every method.

- The ability to elide "self." is consistent with Smalltalk and Objective C.
- Dan Ingalls and Niklaus Wirth urged the change to emphasize that the method belongs to the scope of the type.
- Programs gets shorter.
- It is easier to convert Pascal code written with global variables and procedures to object-oriented code.

One problem with eliding "self." is that unqualified names are harder to identify. In MacApp, we have alleviated this problem by a naming convention. Globals start with "g", fields of objects start with "f", and fields of records start with "the". Still, for readability reasons, it is not recommended to "with" another object inside a method, especially one of the same type!

Qualification by type name is used in a method implementation's header.

- It clarifies the code and facilitates searches in the editor.
- It is more consistent to require it than to make it optional.
- It allows us to eliminate the "methods of...end" construct.

Abstract and default methods have been eliminated.

In Lisa Clascal, the qualifiers written after a regular method were **abstract**, **default**, and **override**. Unqualified methods were also permitted; they were semantically equivalent to **default** methods but were implemented more efficiently because the compiler could assume that overriding was unlikely. The purpose of **abstract** methods was also efficiency; no implementation was provided for them, because all concrete classes were expected to override them.

The reasons for eliminating abstract and default methods are:

- Our new implementation eliminates differences in efficiency.
- The user has fewer concepts to learn.

The special `Create` method has been eliminated.

In Object Pascal, the equivalent of Create"is to call New fand then to activate an initialization method of a user-chosen name. This change was urged by Niklaus Wirth. The reasons are:

- Create was an exception to and confused users.
- Activating an ancestor's Create was paradoxical because the object could only be allocated once (necessarily in TObject) but its size had to be known to do so (necessarily in the most specific Create). We tried various kludges to work around the paradox, all of which were awkward, inefficient, and confusing.

The syntax of the object-type declaration has changed.

In the declaration of an object type, the construct:

```
TPicWindow = object (TWindow)
```

replaces the old Lisa Clascal construct:

```
TPicWindow = subclass of TWindow
```

The new syntax was proposed by Niklaus Wirth. Its advantages are:

- To declare a top-level type we can simply omit the parenthesized type name. In Clascal, "`subclass of nil`" was a kludge.
- In Pascal, type identifiers are nouns that describe individuals, not sets, e.g., in:

  ```
  type Point = record v, h: integer end;
  var pt: Point;
  ```

  pt is a Point record, not the set of all possible Point records. Similarly, in:

  ```
  var w: TWindow
  ```

  w is [a reference to] a Point object, not the set of all possible Point objects.

The construct "`superself.Meth`" changed to "`inherited Meth`".

- The term *superclass* is no longer used.
- It is no longer required to use "self." so a "." seems redundant here.
- The word **inherited** is explicit, e.g., **inherited** Draw invokes the inherited Draw method ignoring any overrides.

The predeclared variable `thisClass` has been eliminated.

It was only used in Create methods. For systems programmers, our compiler will support POINTER(ObjTypeName).

The `methods-of` construct has been eliminated.

In Lisa Clascal, all methods of a class had to appear within `methods of...end` brackets.

- Listings were harder to read and illogical to indent.
- We may want to add a capability later that would let a unit add a method to a type declared in another unit.
- Some programmers may want to organize some units by method name instead of by type name.

An object type may be declared in an `implementation` or `program`.

In Lisa Clascal, an object type could only be declared in the interface of a unit. The change was urged by practically everybody. It is essential for a Think Technologies implementation. Allowing an object type to be declared in a procedure would be problemmatical so we have not.

Names declared in a method must differ from names declared in the type.

The parameters and local names within any method of a type may not conflict with the field and method names of that type. This is consistent with a rule of Smalltalk, and lets the compiler call the user's attention to a quite common bug

Object Pascal

Report

Larry Tesler
February 14, 1985

1. Introduction

Object-oriented programming is a technique in which a complex system is structured as a set of interacting *objects.* Each object defines its own data structure and algorithms. Consequently, a high degree of modularity and data abstraction can be achieved.

Strictly speaking, object-oriented programming does not require special language features, but programs written without the benefit of special constructs tend to be opaque and difficult to modify. Programming languages that support the object concept (e.g., Simula-67 and Smalltalk-80) make it easier to create maintainable programs.

The language *Object Pascal* facilitates object-oriented programming through an extension of the Pascal language. In doing so, it preserves the original aims of the Pascal language: (1) it is a language suitable for teaching; (2) it is capable of being implemented reliably and efficiently on currently available computers.

In designing Object Pascal, we were faced with many difficult tradeoffs. In the end, we were guided by Einstein's maxim: "Make everything as simple as possible, but no simpler." As a result, we sacrificed certain features found in other object-oriented languages because we felt their complexity outweighed the benefits they might have offered. These judgments were not based solely on theoretical notions. The language evolved for more than two years under the name "Clascal", during which hundreds of programmers learned and used it. Each time the language specification was revised, we had an opportunity to observe the effects of the changes on both programming convenience and ease of learning.

The specification reported here is based on a collaboration between Prof. Niklaus Wirth, the author of Pascal, and members of the group at Apple Computer, Inc. that developed and implemented Clascal. We have implemented the specifications in a version of our Pascal compiler for the Macintosh computer, and have created a significant object library and several interesting applications.

We hereby contribute the language specifications to the public domain, and encourage others to make compatible extensions to other Pascal compilers. It should be noted that, with certain adaptations, these extensions could be applied to the language Modula-2.

2. Summary of the Language

In standard Pascal, structured types employ one of the following four structuring methods: array structure, record structure, set structure, and file structure. In Object Pascal, a fifth structuring method is introduced: object structure. An object differs from a record in the following ways:

1. In addition to fields, an object may contain named components called *methods*. Unlike a field, a method is not a variable. A method is a procedure or function, and can be accessed only to activate it.
2. An object type can *inherit* from another object type. It acquires all the fields and methods of that type, can define additional fields and methods, and may *override* any of the acquired methods with its own implementations. Inheritance is transitive.
3. All objects are dynamic, i.e., they are created and destroyed during program execution. They are identified not by pointers, as dynamic records are, but by similar values called *references*.
4. A type identifier associated with an object type always denotes the *reference type* whose domain is that object type.
5. A *reference variable* is a variable whose type is declared to possess a reference type. Its value is never an actual object, but is instead a reference to an object. Any number of reference variables can reference the same object.

3. Notation and Terminology

The notation and terminology used herein follow the conventions of *Pascal User Manual and Report, Third Edition (ISO Pascal Standard)* by Jensen and Wirth, revised by Mickel, and Miner, 1985.

The section headings in this specification correspond to those in the *Report*. Unaffected sections are omitted. Consequently, the numbering of sections is not consecutive.

4. Symbols and Symbol Separators

In addition to the standard symbols of Pascal, Object Pascal adds:

WordSymbol = "object" | "inherited" .

6. Types

An object type definition introduces a type identifier that denotes the object-reference type associated with a new object type.

TypeDefinition = *ReferenceTypeIdentifier* "=" *ObjectType* | ... .

6.2. Structured Types

An object type cannot be packed. However, any of its fields can possess a packed type.

StructuredType = *ObjectType* | ... .

6.2.5. Object Types. The discussion of record types and fields in Section 6.2.2 of the *Report* applies equally well to object types, except that an object type can not have a variant part.

In addition to fields, an object can have method components.

An object type can inherit components from another object type. Inheritance is transitive, that is, if T3 inherits from T2, and T2 inherits from T1, then T3 inherits from T1. If one type inherits from another, the former is called a d*escendant* of the latter, and the latter an *ancestor* of the former. The *domain* of an object type consists of that type and all its descendants.

ObjectType = "object" [*Heritage*] [*FieldList* ";"] *MethodList* "end" .
Heritage = "(" *ReferenceTypeIdentifier* ")" .
MethodList = { *MethodHeading* [";" "override"] ";" } .
MethodHeading = (*ProcedureHeading* | *FunctionHeading*) .
MethodIdentifier = (*ProcedureIdentifier* | *FunctionIdentifier*) .

The scope of a component identifier extends over the domain of its object type, and encompasses component designators and with statements where it may be used. Its scope also extends over procedure and function blocks implementing methods of the object type and its descendants. Thus each component identifier spelling must be unique within an object type and its descendants.

The identifier of an override method must be spelled identically to the identifier of the method it overrides. The order, types, and names of the parameters, and the type of the function result, if any, must match exactly.

An object type can only be declared in the type definition part of a main program. It can not be declared in a variable definition part, in a formal parameter list, or within a procedure or function declaration block. (Note: In Pascal-related languages that support separate compilation, an object type can be declared in any module or unit, but only in the outermost scope.)

Examples of object types:

```
Employee = object
    FirstName, LastName: packed array [1..32] of Char;
    HourlyWage, HoursPaid: Integer;
    procedure Hire(Name1, Name2: packed array [1..32] of Char;
        Rate, HoursWorked: Integer);
    function RegularPay(HoursWorked: Integer): Integer;
    procedure IssuePaycheck(HoursWorked: Integer);
    end;

ExemptEmployee = object (Employee)
    function RegularPay(HoursWorked: Integer): Integer; override;
    end;

Executive = object (ExemptEmployee)
    WeeklyBonus: Integer;
    procedure SetBonus(PerformanceLevel: Integer);
    procedure IssuePaycheck(HoursWorked: Integer); override;
    end;
```

Type ExemptEmployee overrides the method RegularPay because supervisors are paid for HoursPaid hours regardless of the number they have actually worked. Type Executive overrides the method IssuePaycheck because WeeklyBonus must be added to the amount calculated by RegularPay.

It is often desirable to discourage assignment to fields of an object except from within methods of that object. This makes it possible for the object to guarantee its own internal consistency. To discourage uncontrolled assignment, the example above includes a Hire method to initialize the fields of an Employee object, and a SetBonus method to assign to the additional field of an Executive object.

6.2.5.1. Object Type Membership. During execution, an object is created as a specific type. It is considered a *member* of that type and of all ancestral types. In the above example, an object created as an ExemptEmployee is also a member of the type Employee. References to it may be assigned to reference variables of types ExemptEmployee and Employee

Conversely, if a variable Emp is declared to possess the reference type Employee, its value during execution may be either nil or a reference to a member of type Employee, ExemptEmployee, or Executive.

The method selected for activation by the designator Emp.RegularPay depends on the execution-time type of Emp. The method activated could be Employee.RegularPay or ExemptEmployee.RegularPay. (It could also be Executive.RegularPay, but Executive.RegularPay is the same as ExemptEmployee.RegularPay.)

In general, one can not determine from the program text which method a method designator will activate during execution. One can develop a procedure that activates Emp.RegularPay, and later, without modifying that procedure, can apply it to an employee of a new, unforeseen descendant type of Employee. When extensibility of this sort is desired, one should employ an object type with an open-ended set of descendant types in preference to a record type with a closed set of variants.

6.3.1. Reference Types. The discussion of pointer types in Section 6.3 of the *Report* applies equally to reference types, except that the domain type of a reference type is always an object type. There is no special syntax needed to declare reference types, because the name given in an object type definition becomes the name of the reference type.

ReferenceTypeIdentifier = *Identifier* .

6.5. Type Compatibility

The conditions for *assignment-compatibility* are extended as follows. A value possessing reference type T2 is assignment-compatible with a reference type T1 if T2 is in the domain of T1. In other words, a value of type Executive can be assigned to a variable of type Employee (a descendant of Executive) but not vice versa.

6.6. Type Coercion

If reference type T2 is in the domain of T1, a value possessing type T1 can be coerced to a value possessing type T2. It is an error if the value is not a member of type T2.

CoercedFactor = *ReferenceTypeIdentifier* "(" *Expression* ")" .

Example:

Executive(Emp)

7. Variables

7.2.3. Object Field Designators. An object field designator denotes a field of an object.

ObjectFieldDesignator = [*ReferenceVariable* "."] *FieldIdentifier* .

The rules are the same as for field designators of records (see Section 7.2.2 of the *Report*) with the following exceptions. (1) The remarks about variants do not apply. (2) Object fields are accessed through a reference variable instead of a record variable.

The reference variable and the "." may be omitted inside a with statement that lists the reference variable. They may also be omitted within any method block; when they are, the effect is the same as if "self." had been written (see Section 9.1.2).

7.5. Reference Variables.

A variable declared to possess a reference type is a *reference variable*. The "^" used to denote the identified variable of a pointer is forbidden after a reference variable; thus, there is no way to treat the referenced object as a variable in its own right. However, components of the object can be accessed through any reference to it.

ReferenceVariable = *VariableIdentifier* .

Example:

```
type T = object
       F, G: Integer
       end;
var X, Y: T;
New(X); {Create an object of type T; make X a reference to it}
Y := X; {Make Y be a second reference to the same object}
Y.F := X.G + 1; {Make its F field one greater than its G field}
```

8.1. Operands.

Function designators are extended in the same way as procedure statements (see Section 9.1.2).

FunctionDesignator = *MethodDesignator* *ActualParameterList* | ... .

Example:

```
function ExemptEmployee.RegularPay;
       begin
       RegularPay := inherited RegularPay(HoursPaid);
       end
```

For an explanation of "inherited," see the next section, 9.1.2.

9. Statements

9.1.2. Procedure Statements. The syntax of a procedure statement is extended in Object Pascal to allow a method designator denoting a procedure to replace the procedure identifier.

MethodDesignator = [*ReferenceVariable* "." | "inherited"] *MethodIdentifier* .

ProcedureStatement = *MethodDesignator* *ActualParameterList* | ... .

The object referenced by the reference variable plays two roles. First, the execution-time type of the object determines which implementation of the method is activated. Second, the object itself is an implicit actual parameter; it corresponds to a formal parameter named self that possesses the type corresponding to the activated method.

The reference variable and the "." may be omitted inside a with statement that lists the reference variable. They may also be omitted within any method block; when they are, the effect is the same as if "self." had been written.

The modifier "inherited" is generally used within an override method to activate the overridden method. It may only appear within a method declaration block. It must precede the identifier of a method that was inherited by the associated object type. It causes self to be the implicit actual parameter of the called procedure. The execution-time type of self does not determine which implementation is activated by "inherited." Instead, the inherited implementation is activated, ignoring any override.

9.2.4. With statements. The with statement is extended to permit reference variables as well as record variables.

WithStatement = "with" *WithVariableList* "do" *Statement* .

WithVariableList = *WithVariable* { "," *WithVariable* } .

WithVariable = *RecordVariable* | *ReferenceVariable* .

10. Blocks, Scope, and Activations

10.2. Scope

Within the type definition part in which the spelling of an object type-identifier is introduced, that identifier may occur before its introduction. (Note: In Pascal-related languages that support separate compilation, the introduction of the spelling may be required to precede all its occurrences if a module or unit whose scope includes the type definition in question has introduced a type-identifier of the same spelling.)

In a procedure or function declaration block that implements a method of an object type, the identifiers of the components of that type (and of its ancestors) are effective in the block. The interpretation is the same as if the statement list of the block had been embedded in a with statement of the form "with self do begin ... end".

The scope rules for field identifiers of object types is the same as for record types, except that the entire domain of the type is included.

11. Procedures and Functions

ProcedureAndFunctionDeclarationPart =

{ (*ProcedureDeclaration* | *FunctionDeclaration* | *MethodDeclaration*) ";" } .

11.3.5.1. Implicit Parameters. In a declaration of a method for an object type, there is always an implicit parameter, with the identifier self, that possesses the object type. The scope of self extends over the method declaration block. The value assigned to self when the variable is created is a reference to the object whose method component was designated to activate the method. Subsequent assignment to the variable is forbidden.

11.4.2. Dynamic allocation procedures. The predeclared procedures New and Dispose are enhanced to accept a single parameter that is a reference variable. New creates an object whose type corresponds to the reference type of that variable. It also creates a reference to the new object and assigns the reference to the variable.

11.5.2. Boolean functions. Let r be any object-reference expression, and let t be the type-identifier of a reference type that is in the domain of the reference type possessed by r.

Member(r, t)	is an error if r is undefined; otherwise, Member(r, t) yields true if r is non-nil and if the object r references is a member of the type denoted by t.

The Member function is useful for screening questionable coercions.

Example:

```
if Member(Emp, Executive) then
    begin
    Exec := Executive(Emp);
    Exec.SetBonus(Exec.WeeklyBonus * 1.1);
    end;
```

11.6. Method Declarations

A method declaration is like a procedure declaration or a function declaration with the following exception.

It is always declared in the style of a forward declaration: one declaration consists of the method heading and appears in the object type definition, and a second declaration consists of a method identification and the block. Both declarations must appear in the main program [or in the same compilation unit]: the first in its type definition part, and the second in its procedure and function declaration part. The directive "forward" is not used. In the second declaration, the method identifier must be qualified by the type name.

MethodDeclaration = *MethodIdentification* ";" *Block* .

MethodIdentification = ("procedure" | "function") *ReferenceTypeIdentifier* "." *MethodIdentifier* .

MethodIdentifier = *Identifier* .