ECL Language

FACIO Processes
User Documentation

Project: FACIO processes
Author: Štěpán P. Nadrchal
Revision:
Created: 25.7.2006
Version, Status: 1.3; Valid
Pages: 15
Document ID: FACIO-UserDoc-ECL

Content

1. Document Purpose 4
1.1. Main ECL characteristics 4
2. ECL syntax 5
2.1. Keywords 5
2.2. Operators 7
2.3. Parenthesis 7
2.4. Comments 7
2.5. Strings 7
2.6. Number units 8
3. Syntax 8
3.1. Notation 8
3.2. Block and statements 8
Constant values 8
Variable declarations 9
If statement 9
While statement 9
For statement 9
Try ... catch statement 9
Try ... finally statement 9
Foreach statement 10
Case statement 10
SQL statement 10
SQL Expressions 11
4. Semantics 11
4.1. Operators 11
Boolean operators 11
Other operators 11
4.2. Statements 12
Simple Statements 12
If statement 12
While statement 12
For statement 12
foreach statement 12
Try ... catch statement 12
Try ... finally statement 12
Case statement 13
SQL statements 13
5. Using classes and objects 13
5.1. Class definition 13
5.2. Overwriting class methods by objects 14
Reasons 14
Specification 15
Changing class type 15

© PDQM, 1998 -2007

© PDQM, 1998 -2007

Versions

List of versions after the first official version:

Version	Date	Description of changes
1.1	10.7.2006	Several minor changes according to the ECL development.
1.3	11.7.2008	Adjustments for the FACIO Processes version 2008

Linked information

List of documents and other information sources that relate to the content of this document:

Document ID	Version	Description

1. Document Purpose

The document describes the ECL language used in P-D-Q. The document is intended for developers using ECL and administrators of the P-D-Q system.

1.1. Main ECL characteristics

ECL is a language proposed for the fast development of a user interface and for data management handling. ECL is:

- A structured and Object-oriented language
- A language with a strong control of variable types
- Integrated and optimized to handle data from the relational database
- Designed for defining data manipulation and for easy management of user interface

Since ECL is designed for rapid application development and database management handling, it does not contain some techniques common to general languages. For example, the language itself does not contain any functionality for:

- Definition of the classes. The object classes are defined out of the language syntax and tightly relate with database definition as well as with the internal functionality supported by the system that implements the language (P-D-Q in this case).
- Definition of the screens of the user interface (definition of windows are made out of the language and are accessible by functions)
- Pointer arithmetic
- Functions for manual memory management

ECL is supposed to be used together with persistent objects stored in a database. Such objects can have long-time validity and are often manually managed by the system users. Objects (i.e. class instances) describe not only information stored in the system but they also define system behaviour. ECL enhance common object programming approach and allows overlay class methods by objects.

© PDQM, 1998 -2007

2. ECL syntax

ECL syntax and semantics are based on the C language but it contains data types describing their content and operators for object manipulations, object method calling and allows direct use of basic SQL commands. ECL programs use objects represented by variables with the type “link to object of type …”.

Significant difference between ECL and common object languages is that object instances can override class methods[1].

2.1. Keywords

The language introduces the following keywords:

Keyword	Purpose
`if`	Introduce a condition
`else`	Variant part of the condition
`while`	Introduce a cycle
`foreach`	Introduce a cycle that passes all inputs (used in conjunction with SQL command or with array variable)
`for`	Introduce a for cycle
`try`	Starts a block catching the exceptions
`catch`	Starts a block that can manage the exception; part of try … catch
`finally`	Starts a block that is always passed; part of try … finally
`inherited`	Command calls a method of an ancestor with the same parameters.
`next`	Escapes a while / for / foreach command block and let the while statement continuing next pass
`continue`	Escapes a while / for / foreach command block and let the script continuing after the block
`exit`	Escapes the routine
`select`	SQL statement keyword
`from`	SQL statement keyword
`into`	SQL statement keyword
`where`	SQL statement keyword
`group by`	SQL statement keyword
`insert`	SQL statement keyword
`update`	SQL statement keyword
`delete`	SQL statement keyword

[1] The enhancement is not intended to promote self-modifying language in any case. The change follows from the fact that objects represents all data stored in the memory and database including system data describing system behaviour. Methods of objects as well as methods of classes and global procedures should be written manually by persons and cannot be modified in the runtime. ECL does not contain any instrument to allow script modification from another script or from the script itself.

ECL in general allows overriding of the methods by object for any object but P-D-Q utilizes overriding only for objects for which the enhancement has justness. Concretely, P-D-Q allows overriding of the methods for process and process task descriptions. if necessary, the overriding can be used for any class objects.

© PDQM, 1998 -2007

database	SQL statement - can be used at the end of regular SQL statement to redirect execution of the SQL statement to linked database.

The language defines types of variables:

Type name	Purpose
integer / int	An integer type
boolean	A Boolean (false / true) value
string	String
float	Real-type float (in double precision)
date	A date (without time)
time	A time (without a date)
datetime	Date and time
File	Representation of the file stored in the database
Link <classname>	Relation to class
ItemListVal	A field whose value is populated from a defined list
TimeInterval	A value representing the distance between two datetime stamps (Value is represented in seconds and one second is the smallest unit of interval)
Text	Text
RTF	A rich text format
HTML	HTML text
Bitmap	A picture in BMP format
JPEG	A picture in JPEG format
Icon	A picture in icon format
Metafile	A picture in Windows Metafile format
Sound	A sound in Wav format (not yet supported by the P-D-Q)
Blob	Unspecified block of data
Color	Definition of colour

Variables of all types can be used as arrays.
ECL introduce constants and variables:

Keyword	Purpose
true	
false	
null	An “empty” value
_ERROR_NAME_	Variable used in the catch block. It contains the name of the raised exception
_ERROR_DESCRIPTION_	Variable describing the error

© PDQM, 1998 -2007

2.2. Operators

The language supports the following operators:

Operator	Purpose
`:=`	Assignment
`.`	Connects a class variable and its field
`>, <, >=, <=`	Compartment
`!=, <>`	Compartment "is not equal" (both notation has the same meaning)
`!`	Negation
`&`	And
`|`	Or
`->`	Implication
`<->`	Equal (both are true or both are false)
`+, -, *, /, %`	Arithmetical operators (% means modulo operations)
`+=, *=, -=, /=, %=`	Arithmetical operators combined with the assignment

The semicolon ';' is not an operator but it is conventionally used to end an ECL statement.

2.3. Parenthesis

ECL uses the following parenthesis

Parenthesis	Purpose
`{ ... }`	A block of commands
`[ ... ]`	An index after the array variable
`( ... )`	Boundary of a condition or of a part of a regular expression, boundary of class type change statement or of the method parameters
`[]`	Square parenthesis one after another has special meaning. They are used together with the type of the variable and marks the variable to be of the array

2.4. Comments

ECL allows two types of comments:

One line comment (to the end of the line):

```
//  coment text
```

Multi line comment:

```
/* coment that can go over one or
more line. Is ended by the complementary symbol */
```

2.5. Strings

An ordinary string is written to apostrophes ‘:

```
‘ text ‘
```

A select statement can be marked by “:

```
“ select name from tablname into :a “
```

but it is not necessary and the SQL command can be placed directly to the code.

© PDQM, 1998 -2007

2.6. Number units

ECL allows using units that describe the time interval:

Unit	Purpose
s	Seconds
m	Minutes
h	Hours
d	Days
w	Weeks

The units can be used to identify time interval, i.e.:

```
2m+3s = 123 seconds
```

3. Syntax

The following paragraphs describe, how the ECL procedure is configured. The chapter does not describe common syntax e.g. numbers, conditions etc.

3.1. Notation

```
O(X) ::= X | empty
#(X) ::= any number of X (including none)
```

3.2. Block and statements

```
procedure ::= Block
Block ::= statement | { statement_sequence }
statement_sequence ::= statement #(; statement)
statement ::= simple_statement | composite_statement
simple_statement ::= variable_declaration | empty_statement |
              assignment_statement | function call | next | continue | exit
composite_statement ::= if_statement | while_statement | foreach_statement |
              try_catch_statement | try_finally_statement | case_statement |
              for_statement
  empty_statement ::= ;
assignment_statement ::= variable := expression
                       | variable_declaration := expression
BoolOperator ::= & | "|" | -> | <-> | ->
CompareOperator ::= > | < | = | >= | <> | <=
conditition ::= true | false | boolean variable
                | !condition
                | condition BoolOperator condition
                | expression CompareOperator expression
```

Constant values

Constant values are declared in ECL by the same way as how they are defined in common languages like C or Pascal with a few exceptions.

Constant text strings are delimited by apostrophes and unlike other languages can be at one or more lines of code.

Time intervals constants can use time units (see section 2.6) for definition of the interval size. Intervals are described by the number of seconds.

Variable declarations

```
variable_declaration ::= variable_type  variable_name #( , variable name )
variable_type ::= basic_type | link_type
basic_type ::=  vartype O( [] )
vartype ::= see table of types of variables
link_type ::= link O( [] ) classname
variable_name ::=
        any non keyword word from letters and digits (first must be letter)
```

Examples

```
int a, b;
string[] c;
link[] order;
int a := 5;
```

If statement

```
if_statement ::= if (condition) statement O(else statement)
```

Example

```
if (a > b) { ... } else { ... };
```

While statement

```
while_statement ::= while ( condition ) statement
```

Example

```
while (a > b)
{ ...
  next;
  ...
  continue;
  ... };
```

For statement

```
for_statement ::= for ( statement; condition; statement ) statement
```

Example

```
while (int i := 1; i <= 5; i += 1)
{ ...
  continue;
  ... };
```

Try ... catch statement

```
try_catch_statement ::= try statement catch statement
```

Example

```
try { ... }
catch showmesage(_ERROR_DESCRIPTION_);
```

Try ... finally statement

```
try_catch_statement ::= try statement finally statement
```

Example

```
try { ... }
```

© PDQM, 1998 -2007

```
finally command;
```

Foreach statement

```
foreach_statement ::= foreach (SQLcommand) statement
                    | foreach (non_array_varaible ; array_varaible) statement
```

Examples

Using foreach with SQL statement:

```
foreach ( select name from tablename into :name )
 showmessage(name);
```

Using foreach for listing variables from array variable:

```
foreach ( field; array_variable )
 sum := field;
```

Case statement

```
case_statement ::= case (expression) { O( (expression): statement )
                   #(else: statement ) }
```

ECL supports a general expression statement and does not limit the case statement to be used with fixed values only as common languages define it.

Examples

```
case (a) {
  1:  command;
  b:  command;
  else: command;
}

case true {
  x > 100: showmessage('Too much');
  x > 10: showmessage('Enough');
  x > 1: showmessage('A little');
  else: showmessage('Nothing');
}
```

SQL statement

SQL statements represent the statement written in the SQL language supported by the used database. Keywords and commands that are supported by the database can be used (e.g. top or first keyword can be used in dependence on what is supported by the used database.)

Select command

```
select ... from ... #( where ... )
    into :variable #(, :variable )
    #(database database_name)
select ... from ... #( where ... )
```

```
insert into table_name O( field_name #(, field_name)
          values ( (:SQLexpression #(, :SQLexpression) )
          #(database database_name)
update table_name set field_name = constant_value | :SQLexpression where ...
delete from table_name where ...
```

Whenever in the select command can be used *SQLexpression*:

© PDQM, 1998 -2007

```
SQLexpression ::= :variable | :{ complex_expression } |
                  ::variable | ::{ complex_expression }
```

Although ECL is not intended to manage changes of the database structure, other database statements like *alter table* etc. can also be executed from ECL but the whole statement must be placed to the quotation marks:

```
" SQL statement "
```

Let us note, that quotation marks does not delimit a string (apostrophe is used to delimit string) and the sql statement in the quotation marks can use SQL expressions as well as the supported select/insert/update/delete commands. SQL command in the quotation marks can be placed to one or more lines of code.

ECL script can work with objects and data from one or more databases. For each script one database is default and all SQL statements are performed with the default database if it is not redirected. The optional part *database <database_name>* of the ECL SQL statement redirects the statement to the named database. ECL does not support complex statements that join two or more databases and so only one database name can be referenced. The current ECL version does not allow using *SQLexpression* to define name of the database.

SQL Expressions

```
expression ::= simple_expression | complex_expression
simple_expression ::= variable | constant_value | function call
complex_expression ::= expression operator expression
```

4. Semantics

The following paragraphs describe semantics of ECL. Trivial semantics description is missed.

4.1. Operators

Boolean operators

```
a & b     ::= true iff a is true and b is true
a & b     ::= true iff a is true or b is true
a -> b    ::= true iff a is true and b is true or a is false
a <-> b   ::= true iff a and b are both true or false
a != b    ::= true iff value(a) is not equal to value(b)
a = b     ::= true iff value(a) is equal to value(b)
!a        ::= true iff a is false
```

Other operators

```
t.y where t is classtype      ::= call static method y
x.p where x is a link to object ::=
returns value of object field p or call object method p. (Static methods are
object methods as well and they can be with the object instance as well.)
x aop y where aop is arithemtical operator (+, -, *, /) ::=
returns result of arithmetical operation. ECL support the following operands:
```

- If both x and y are numerical values including time interval, are arithemtical operations are allowed.
- String parameters allows concatenation only (represented by operation '+'[2]) and both parameters must be of the string type.
- Datetime parameter and time interval parameter allows time shift (+ , -)

[2] Let us note that using of the string concatenation operator in the sql command should follow the database definition of the sql language. Some databases supports symbol '+' but others require using '|'. P-D-Q implementation of ECL does not translate operators used in the SQL commands.

x :aop y performs arithmetical operation with x and y parameters and assign result value to the variable x

4.2. Statements

Simple Statements

variable := expression ::= value(expression) is assigned to be a value(variable)

class_type_name (expression) ::= expression should by of the variable type "link to an object" and this expression change type of the link to the type link to the class_type_name object.

next ::= when used within the while_statement, for_statement or foreach_statement it causes breaking of the statement processing. If possible, block is processed for next value (foreach statement) or if the condition is still valid (while statement, for_statement)

continue ::= when used within the while_statement, for_statement or foreach_statement the scripts continue with the command next to the block

exit ::= breaks processing of a procedure

inherited ::= Calls overlaid method from the ancestor class. The parameters of the overlaid method must be a subset of the parameters of the descendant method. The ancestor class is called with the values of the parameters values that are actual in the moment of the call.

If statement

if (condition) statementA **else** statementB ::=
statementA is processed if condition is true otherwise StatementB is processed

While statement

while (condition) statement ::=
Statement is processed as long as condition is valid before the processing

For statement

for (initial_statement; condition; final_statement) statement ::=
intial_statement is processed at start. Statement processed as long as the condition is valid. After each processing od statement, final_statement is procesed.

foreach statement

foreach (SQL select) statement ::=
Statement is processed for all records fetched from database

foreach (field; array_variable) statement ::=
Statement is processed for all fields of the array variable; each of them is assigned to the field varaibles before processing the statement

Try ... catch statement

try statementA **catch** statementB ::= StatementA is processed and StatementB is processed in case that an exception raises in statementA.

Try and catch statement cannot be used to catch error in the source code structure that are not detected during compilation (e.g. wrong parameters passed to the method with free parameters etc.)

Try ... finally statement

try statementA **finally** statementB ::= StatementB is processed after the processing of the statemenetA regardless if exception raised or next or break expression had been used in the statementA.

© PDQM, 1998 -2007

Case statement

```
case (expressionC) { Expr1: statement1 ... ExprN: statmentN; else:
StatementE)} ::=
Evaluates the first statements from statement1 to statementN which ExprX is
equal to the expressionC. If any of ExpressionX is equal to the exrpessionC,
the exrepssionE is processed if exists.
```

SQL statements

SQL statement is passed to the database and is evaluated by the database except that SQLexpressions used within the SQL statement are processed and replaced by their result value. Meaning, syntax and processing of the statements are defined by the database upon which the language is used. It is valid also when the statement is in the quotation marks.

At the end of regular SQL statement can be used command

```
database <database_name>
```

i.e.:

```
select name from Customer where address = :address into :CustName database crm
```

where *Customer* is the name of the table that is in the database linked with the P-D-Q with the name *crm*. Value of the parameters are delimited by apostrophes if it is a string value so that *:address* parameter in the example is a correct. Is some cases such behaviour is interfering especially in the case when parameters are used to set name of the field or name of the table. In that case two colons can prefix parameter. It is presented by the following example:

```
delete from ::TableName where name is null
```

Parameter *TableName* (which can be a local variable but also a global static method of the class that returns name in which the class objects are stored) defines name of the table that are to be cleared. Since '::' prefixes the parameter, the command is correctly translated to the SQL command and name of the table will not be in the apostrophes.

5. Using classes and objects

5.1. Class definition

ECL does not yet have defined syntax for class declarations and definitions and they should be defined out of the ECL scripts. P-D-Q has windows and GUI for class definitions.

Each class can have as many ancestors as necessary. If class has more ancestors that have common ancestor, the common ancestor is ancestor of the class. Contrary to C++ the common ancestor is as a ancestor only once.

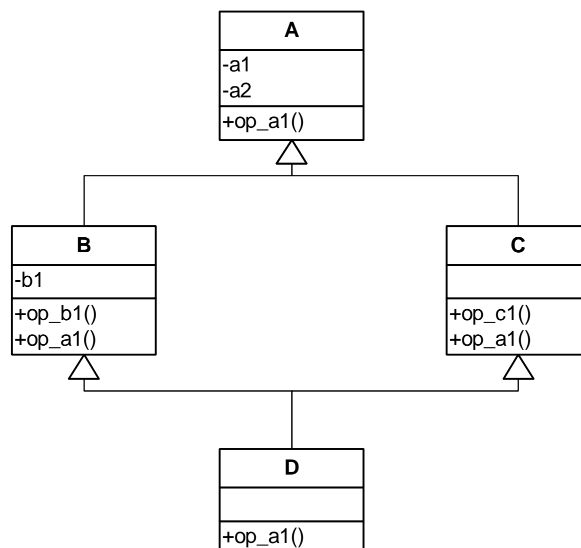

Figure 1 Class inheritance example

© PDQM, 1998 -2007

The example at the Figure 1 presents situation where class D have ancestors A, B and C. It has defined items a1, a3, b1 and methods op_a1, op_b1 and op_c1. All descendant classes overwrite method op_a1. The D.op_a1 can call inherited class method using the statement **inherited**, but it *cannot decide* if it calls b.op_a1 or c.op_a1. Statement **inherited** always calls the method of the first ancestor that defines it.

Class inheritance in ECL is not intended for complex class structures. Ancestors in the ECL has similar role as interfaces in the meaning how they are defined in the Java language.

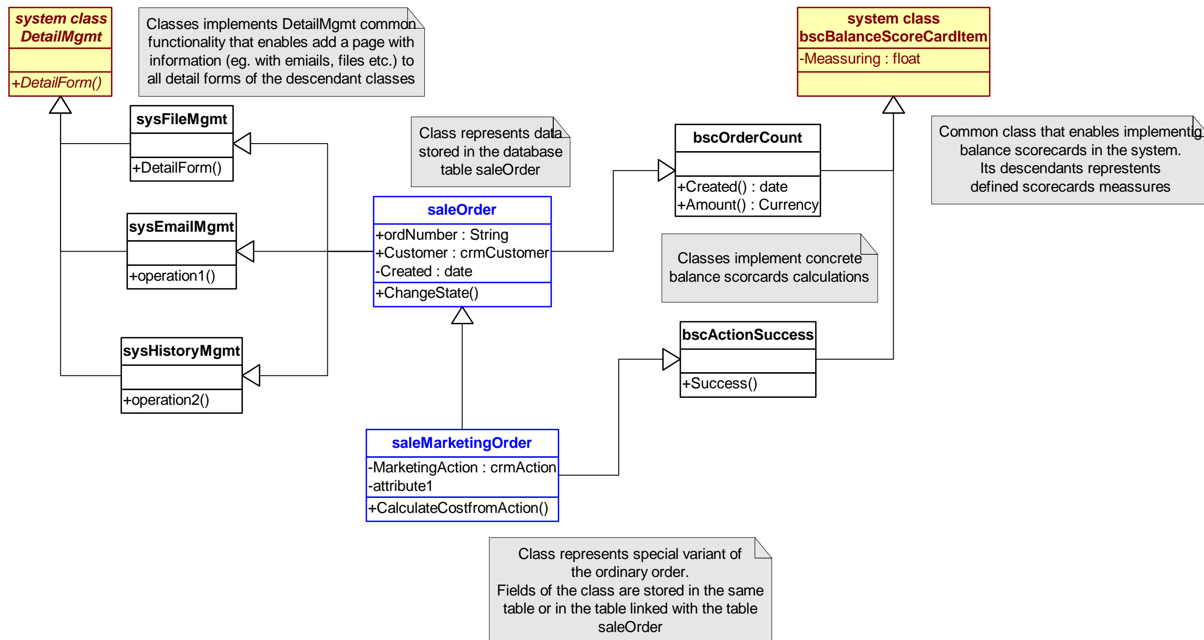

Figure 2 Inheritance utilisation

The Figure 2 presents a real example how to use class inheritance mechanism.

Blue marked classes are customer classes that represent database records. They encapsulate data from database together with functions of data management.

Yellow classes are internally implemented classes that make available the internal P-D-Q functionality. Classes inherited from these management classes provide special functionality or changes displaying of information in the system.

White marked classes utilize common functionality and implements concrete customer or general solution. They can store data to the database or not but if they stores data, they are in the separate table and can be linked potentially to any record in the database (weak records, eg. emails linked with the order) or stay independent on the data records (e.g. a result of a measure that had been made upon all orders)

5.2. Overwriting class methods by objects

Reasons

One of the major differences of ECL against common programming languages is the ability of objects (instances of classes) to overwrite class methods. Objects in the ECL meaning are mostly persistent and often represent a specific object, knowledge or activity in the real world. Many objects are identical - e.g. produced artefacts, bills etc. and there is no reason to define individual behaviour for them. On the other hand other elements are unique and have unique characteristics or behaviour although they are objects of the same type. An example might be tasks in process, working teams, metrics or even factory machines each of which can have different performance, features etc. If the behaviour and properties of each process task, metric or other unique objects had to be described by a separate class, it confuse the logic of the system because all tasks and metrics are instance of the same metric class. A descendant class that overrides parent class's methods can describe them, but it has two major drawbacks:

© PDQM, 1998 -2007

1. Classes are intended to be static and creation of a new class changes the system logic and structure
2. Each class would probably have only one instance object and the system would be twice more complicated: A real object would be described by two elements: special class and its unique instance object.

Specification

ECL class defines attributes and methods and these are valid for all instances. Instance cannot neither create globally visible attribute nor define a new method. Object instance can override methods defined by class or define internal attributes (that are persistent as the object is persistent).

Changing class type

In ECL, class descendant can be assigned to the variable of the ancestor. The ancestor should be retyped to declare its real class type. Object that is not either descendant nor ancestor cannot be assigned to the variable of a defined type.

© PDQM, 1998 -2007