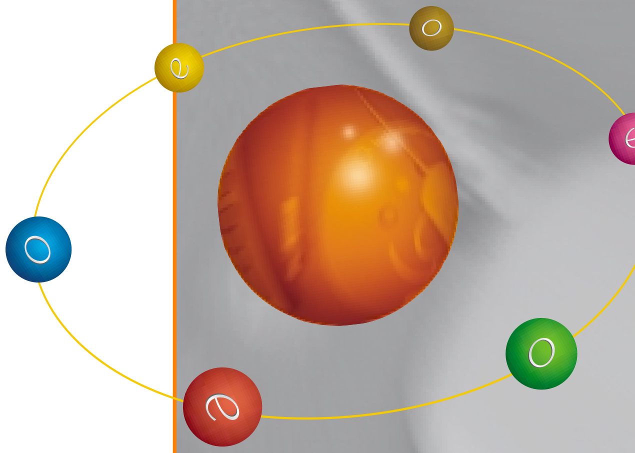
Objecteering/UML

Objecteering/J Language User Guide

Version 5.2.2



Objecteering Software

www.objecteering.com

Taking object development one step further

Objecteering/UML version 5.2.2 - CODOBJ 001/001

Contents

Chapter 1: Overview

Definition and objectives

Purpose of this manual

Welcome to the *Objecteering/The J language* user guide!

This user guide explains the notions, syntax and use of the J language. It is aimed at users wishing to adapt the Objecteering/UML CASE tool to different sorts of model usage, including:

- ♦ specific code generation
- ♦ model transformation to assist technical design
- ♦ definition of additional consistency checks
- ♦ adding model requests

J language

The J language is the language which supports *Objecteering/UML Profile Builder*, and which allows the Objecteering/UML tool to be parameterized and driven. J is an object language dedicated to handling models. It introduces unique features designed and developed for this purpose.

Prerequisites

J is a "*simple*" object language. It is assumed that readers of this user guide are already familiar with this type of language (*Smalltalk*, *Eiffel*, *C++*, *Java*, etc.), since this user guide does not explain in detail the notions of class, generalization, message sending, attribute and operation.

J programs are developed and executed using the *Objecteering/UML Profile Builder* tool, described in the *Objecteering/UML Profile Builder* user guide.

J is based on the Objecteering/UML metamodel, described in the *Objecteering/Metamodel* user guide. The main part of the J library is described in the *Objecteering/J Libraries User Guide* user guide.

J characteristics

Java-like syntax

J is an object language with a Java-like syntax. It is dedicated to handling the metamodel (browsing capacities) and the UML Profile mechanism (double look up mechanism). Several Java features, such as interface, package and many Java libraries cannot be handled, but J does have some specific unique features which are very convenient when using the metamodel.

Interpreted language

J is an interpreted language. Its code can thus be modified and tested rapidly. Instructions are immediately taken into account and their impact is immediately apparent.

As a consequence, J provides the "*eval*" method, which makes it possible to dynamically execute a String containing J code (see the "*eval statement*" section of chapter 5 of this user guide).

Language for handling the metamodel

J is run at metamodel level and handles model elements created by the user. For example, if the user creates a "*client*" class with "*name*" and "*age*" attributes, a J program will be able to access the "*client*" object, ask for its attributes and then handle the "*name*" and "*age*" objects.

The classes provided in J are, therefore, those defined by the metamodel, which are all described in the *Objecteering/Metamodel* user guide.

UML profiles

UML profiles are essential to the structuring of a J program and to making it customizable. For the J interpreter, the current UML profile is a fundamental piece of information, used to identify the service (operation or attribute) that has been called by a message. The class of the destination instance is just as important. The J programmer uses UML profiles to:

- structure processing by theme (C++ generation, documentation generation, etc.)
- redefine the method of a parent UML profile (generation parameterization)
- define a service that can be shared by several independent J programs (for example, the directory that produces generated files)

Composition of a J program

J classes are predefined - basic classes (*int*, etc.) plus the Objecteering/UML metamodel ("*Class*", "*Association*", etc.). The J programmer can declare new methods on them, but cannot define new classes. The programmer uses the objects created by users in Objecteering/UML (classes, attributes, etc.).

Note: Objecteering/UML work products are an exception. The user can define "*work product classes*" that can be handled in J.

Accessing information

There are several ways to access or handle information:

- through simple attributes, which are predefined by the Objecteering/UML metamodel
- through "*class*" attributes, which can be defined by the J programmer
- through method parameters
- through local variables

Sets

One of the most important aspects of the J language is its capacity to handle sets. Sets are found in:

- elements linked to a given model element (for example, the attributes of a class or the parameters of an operation, etc.)
- variables, attributes or parameters

Browsing is made easier

The J language is used to browse a model, in order to manage all the related information (for example, the classes of a package, the operations of these classes, etc.). This browsing is handled through:

- the sending and diffusing of messages
- the notion of context
- control messages

The sending and diffusion of messages

J provides two major control structures:

- the sending of a message, used to apply a method to an object, is written as follows:

  ```
  object.method_name (parameters) ;
  ```

- the diffusion of messages applies either to a set of objects, or to an object reference. Message diffusion is used to send the message to all set occurrences, or to avoid the processing of an empty reference, and is written as follows:

  ```
  ReferenceSet.<method_name (parameters) ;
  ```

Anonymous methods are a special case of diffusion. They work in the same way, but instead of applying a method to a set or to an object reference, the whole block of instructions is applied according to the set or the reference.

```
ReferenceSet
{
   J processing
}
```

The "if" instruction

The "if" instruction is the conditional control structure within J programs. It applies to boolean expressions:

```
if (condition1)
{
   instructions
}
else if (condition2)
{
   instructions
}
else
{
   instructions
}
```

Control messages

J provides the following control messages, which make browsing easier:

- the "select" message, which is used to filter certain elements (selection)
- the "while" message, which can limit the evaluation of an expression

These messages are particularly well adapted to sets.

Comments

The "//" characters indicate comments, which are valid until the end of the current line.

```
if (condition) { // a comment
```

The "/*...*/" characters surround contained comments.

Executing J programs

Editing a UML profiling project

J programs are developed using the UML profiling project editor. To use this tool, you must have a license for *Objecteering/UML Profile Builder*.

Running programs

There are two ways to execute J programs:

- through commands: commands correspond to menu entries for the final user. They are directly associated with a J method, and are run on the original object of the command.
- through the J language on line, in "*batch*" mode. This mode is explained in chapter 7 of this user guide.

J and the metamodel

Overview

The metamodel defined for Objecteering/UML is accessible through the J language. J uses metaclasses, meta-associations, meta-roles and meta-attributes to navigate within a model and access model information.

J can also transform the metamodel, for example, change attribute values, create or determine model elements or modify links).

As we have already seen, J is used to define methods at metaclass level.

Handling rules

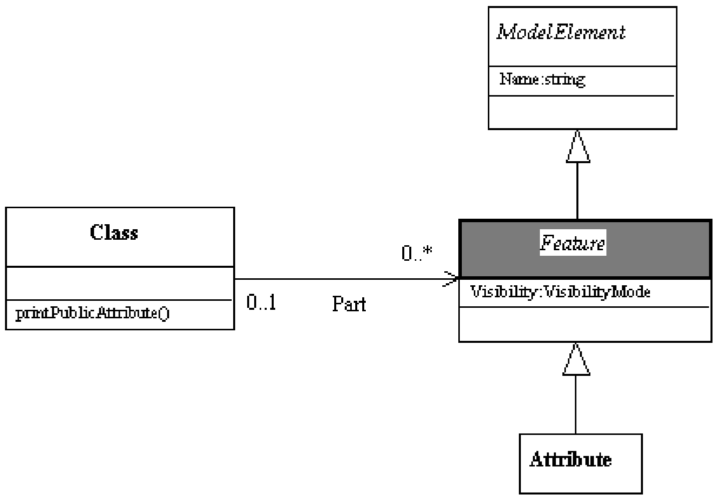


Figure 1-1. Part of the metamodel (simplified)

In Figure 1-1, the names of all the attributes whose visibility is "*public*" ("*printPublicAttributes()*" J method) can be displayed.

Note: This example can be edited in the example UML profiling project and can be executed by running the "*printPublicAttributes*" command on classes.

```
Class:printPublicAttributes()
{
   PartAttribute
   {
      if (Visibility==Public)
         StdOut.write (Name, NL);

   }
}
```

Note: "*NL*" indicates the return. "*StdOut*" is the standard output which will be the Objecteering/UML console most of the time. "*write*" is a method of writing, which allows an unlimited number of parameters, which can be character *strings* or other basic types (*integer*, *boolean*, etc). "*PartAttribute*" is the concatenation of the "*Part*" role name, and the "*Attribute*" metaclass name. For J, and from a "*Class*" instance perspective, it designates all attributes belonging to the current class. "*Name*" is the attribute of the "*Attribute*" metaclass.

Our first example

Aims of the program example

We are going to develop a first program which lists the names of the classes of the package ("*Package*" metaclass), and for each class, provides a list of public methods.

Realization principle

The J method, which we will call "*listClasses*", must be defined on the "*Package*" metaclass. This method will run through all the classes of the package that is being processed, and for each class, it will list methods with public visibility.

Note: This example is for the "*listClasses*" command on packages.

J program

```
Package:listClasses () Method definition
{
    OwnedElementClass  // Going through the Package
                          classes
   {
      StdOut.write(NL, "CLASS:", Name);
      PartOperation.<select (Visibility == Public)
      // Going through the method's public classes
      {
         StdOut.write(NL, " Public method:", Name);
      }
   }
}            // End of "listClasses
```

Description of the terms used

Term	Nature	Definition
Package	Metamodel class	Package
NL	Constant	Return (New Line)
OwnedElementClass	object (Class[ ])	Classes of the current package
StdOut	Predefined variable	Standard output
PartOperation	object(Operation[ ])	The operations of the class
Visibility	object - (String attribute)	Visibility of an operation
Name	object - (String attribute)	Name of the current object
listClasses	J method	Defined method
String	basic class	Character string
write	Predefined method	Writing in a file

Execution

If a "*Human_Society*" package is composed of "*Man*" and "*Woman*" classes with the "*walk*" and "*eat*" methods, the "*listClasses*" message sent to the "*Human_Society*" object will give the following result:

```
CLASS: Man
Public method :eat
Public method :walk
CLASS: Woman
Public method :eat
Public method :walk
```

Chapter 2: J classes

Root classes

Overview

Root classes are top level classes which generalize all other classes.

The "Object" class

The "*Object*" class is the basic class of all the classes present in the J language. All classes, including basic classes (*integer*, *String*, etc.), specialize the "*Object*" class.

The ... operator	corresponds to ...
==	reference equality
!=	reference inequality

The Object[ ] class

The *Object[]* class represents a set of objects, whatever their nature. It specializes "*Object*".

A constructor exists with the set type written "*SetOf*" that can be associated to any J class. The type built in this way specializes *Object[]*. It is thus possible to handle *String[]* or *Attribute[]*.

Note: Sets whose elements are sets are not implemented at present.

See chapter 4 of this user guide for set handling services.

The "Metaclass" class

J classes, either primitive classes like "*int*" or UML metaclasses like "*Association*", are all instances of the "*Metaclass*" class. Thus, without knowing which is the element handled, it is possible to find out its "*Metaclass*", and then to discover its metaclass name, followed by its parent metaclasses. All the classes present in the J language are instances of the "*Metaclass*" class.

Purpose of the "Metaclass" class

Having access to the "*meta metamodel*" allows the realization of more generic processing using the J language. This mechanism is similar to the Java introspection mechanism.

Changes to the metamodel in J are not permitted (only changes to the model). Introspection services are, therefore, limited.

ClassOf

```
MetaClass Object: ClassOf()
```
"*ClassOf*" returns an object's metaclass.

Name

```
String MetaClass: Name()
```
"*Name*" returns the metaclass name.

```
String Object: metaclassName()
```
Short cut to get the metaclass name of an element.

Symbolic access to metamodel features

Syntax:

```
boolean Object:getFeature (in String feature, out
Object value)
```

Returns the value of "*feature*" in "value" for the current object. "*feature*" is either an attribute, or a dependency with a destination class. The return is true if "*feature*" is defined on the current object.

Example: We are going to use "*getFeature*" to recover the name and features of a class:

```
String name;
Feature[] features;
Object o;

this.getFeature("Name", o);
name ?= o;

getFeature("PartFeature", o);
features ?= o;
```

ParentOf

```
MetaClass MetaClass: ParentOf() return MetaClass
```

"*ParentOf*" returns the parent metaclass of the current class or returns empty if no parent class exists.

Example:

In this example, we are going to display the name of the metaclass and the name of the parent metaclass of any object.

Note: This example can be executed with the "*dump*" command on any model element ("*Object*")..

```
Object:dump()
{
    MetaClass Mclass = ClassOf();
    StdOut.write("Metaclass : ", Mclass.Name, NL);
    Mclass.ParentOf()
    {
       StdOut.write("Parent metaclass: ", Name, NL);
    }
}
```

Basic classes

Overview

Basic J classes, like all the others, specialize the "*Object*"root class. Any J method created at "*Object*" class level will, therefore, be available for any J class.

Five basic types are available:

- int
- float
- boolean
- String
- enumerate

The "int" class

The "*int*" class indicates integer values. Specific operators are described below.

The ... method or operator	**corresponds to ...**
<	"less than" integer comparison.
<=	less than or equal to integer comparison.
>	greater than integer comparison.
>=	"greater than or equal to" integer comparison.
+	the sum of integers.
-	the subtraction of integers.
*	the multiplication of integers.
/	the division of integers.
%	the obtaining of the modulo.
float toFloat()	the conversion to float.
String toString()	the conversion to String.

Note 1: If there is a processing error, a message is displayed and the J interpreter stops.

Note 2: There is an implicit conversion of the integers into reals.

The "float" class

The "*float*" class gives real values. Specific operators are described below.

The ... method or operator	**corresponds to ...**
+	the sum of floats.
-	the subtraction of floats.
*	the multiplication of floats.
/	the division of floats.
<	"less than" real comparison.
<=	"less than or equal to" real comparison.
>	"greater than" real comparison.
>=	"greater than or equal to" real comparison.
float sqrt()	the square root extraction.
float pow(Exponent : in float)	the raising to the "Exponent" power.
float log()	the obtaining of the Napierian logarithm.
float exp()	the obtaining of the exponential.
int toInt()	the conversion to integer.
String toString()	the conversion to String.

```
float u = 1.0;
float e;
e = u.exp() ; // e == 2.71...
u = e.log() ; // u == 1.0
```

The "boolean" class

The "*boolean*" class gives boolean values. These are used by classic operators (and, or and not). There are two possible values:

- true
- false

Specific operators are detailed below.

The ... method or operator	**corresponds to ...**
String toString()	the conversion to String.
||	a logical "or" between boolean expressions. If the left-hand term of a "||" is true, the right-hand term is not evaluated.
&&	a logical "and" between boolean expressions. If the left-hand term of a "&&" is false, the right-hand term is not evaluated.
!	a logical "negation" of a boolean expression.

```
boolean b = true ;
String s;
s = b.toString(); // s = "true"
```

The "String" class

The "*String*" class gives character Strings. Character strings have an unlimited size. Their specific operators are described below.

The ... method or operator	corresponds to ...
<	"less than" String comparison.
<=	"less than or equal to" String comparison.
>	"greater than" String comparison.
>=	"greater than or equal to" String comparison.
+	the conversion to String and the concatenation of the parameters.
int size()	the obtaining of the String size.
space (in int SpacesNumber	the completion of the String by an indicated number of spaces.
substitute (in String ToSubstitute, in String NewValue)	the replacement of all the occurrences of "ToSubstitute", by "NewValue" in the String.
toUpper()	the conversion to Upper Case.
toLower()	the conversion to Lower Case.
int findFirst (in String Pattern, in int StartIndex)	the position of the first occurrence of "Pattern" from "StartIndex" or -1 if "Pattern" is not present.
int findLast(in String Pattern)	the position of the last occurrence of "Pattern" or -1 if "Pattern" is not present.
String[] segment (in String Separator)	all the segments between separators. Each occurrence of "Separator" precisely defines 2 segments that can be empty.
String[] findToken(in String SeparatorsSet)	all the lexemes separated by one or more occurrences of one of the characters of "SeparatorsSet".
eraseBefore(in int StartIndex)	the deletion of the characters placed before "StartIndex".
eraseAfter(in int StartIndex)	the deletion of the characters placed after "StartIndex".

The ... method or operator	**corresponds to ...**
erase(in int StartIndex, in int StopIndex)	the deletion of the characters placed between "StartIndex" and "StopIndex ", including the indexes.
assign(in String Origin, in int StartIndex, in int StopIndex)	the assignment with the "Origin" part placed between "StartIndex" and "StopIndex ", including the indexes.
boolean strnequal (in String ToCompare, in int Length)	the comparison with "ToCompare" on a "Length" length.
strcat (...)	the concatenation of the list of strings indicated as parameters of the current string.
concat (...)	the conversion into a string and the concatenation of the list of strings indicated as parameters of the current string.
strncpy (in String Origin, in int Length)	the assignment with the first "Length" characters of "Origin".
strncat(in String ToAdd, in int Length)	the concatenation of the first characters of "ToAdd" with the "Length".
strip (in String BorderSet)	the deletion of all the occurrences of the characters of "BorderSet" at the beginning and at the end.
prepend(in String ToAdd)	the adding of "ToAdd" at the beginning.
insertStr(in StringToAdd, in int Position)	the insertion of "ToAdd" in the "Position" position.
overwrite(in String ToWrite, in int Position)	the overwriting of "ToWrite" from "Position".
float toFloat()	the conversion to float.
int toInt()	the conversion to integer.
boolean toBoolean()	the conversion to Boolean.
String toString()	the conversion to String (identity).

Note on + operator

If one of the + operators is not a *String*, it must be converted. This conversion is predefined for basic classes. For other classes, the "*String toString ()*" message is sent and its return is used as operator.

Example of a String

```
String s1 = "/usr/bin:";
String s2 = ":/bin";
String s3 = ":/usr/ucb:";
String[] ss ;
s1 = s1+s2+s3; // s1 = "/usr/bin::/bin:/usr/ucb:"
ss = s1.segment(":");
// ss == {"/usr/bin", "", "/bin", "/usr/ucb", ""}
ss = s1.findToken(":");
// ss == {"/usr/bin", "/bin", "/usr/ucb"}
```

Escape character in a constant String

The "~" character is used to avoid interpreting the character that follows. Generally, the "~" symbol is used to insert the "" character in a character String.

```
String s1 = "the symbol ~"~~~" is used to insert the
character ~"~"~"";
```

enumeration type

In J, new enumeration elements may not be created. However, several enumerations are predefined in the Objecteering/UML metamodel. For example, the "*Visibility*" enumerate has the (*Public*, *Private*, *Protected*) values. (For further information, please see the *enumerated type* used in the metamodel).

The ... method	**corresponds to ...**
int toInt()	the conversion of an enumeration into "*int*", according to the literal order, starting with "according to".
String toString()	converts an enumeration into String.

Values are handled in J purely by literal names.

Input/output classes

Overview

The family of stream classes (*stream*, *outStream*, *inStream*) is used to manage input/output. Three predefined instances of the "*outStream*" class correspond to the standard exits (*stdOut*, *stdErr*, *stdFile*).

The "Stream" class

The "*Stream*" class is abstract. It provides no services.

The "outStream" class

The "*outStream*" class represents output flows. Its methods are as follows:

The ... method	**is used to ...**
boolean open(in String pFileName, in boolean pAppendMode)	open the file indicated by "name". If "pAppendMode" is true, this operation does not reset the file to zero and prepares to write in append mode. If pAppendMode is false, this operation overwrites the previous contents and prepares to write in the file. If the file does not exist, it is created. If the file cannot be opened, false is returned. The "*pFileName*" parameter is an absolute or relative path. The second parameter (pAppendMode) is optional. Its default value is false.
write (in basic_class p1 in basic_class p2, ...)	write the list of primitive values supplied as parameters.
close ()	close the file.
existFile (in String FileName, out boolean answer)	test the existence of the file by its name.

The "inStream" class

The "*inStream*" class represents in files. It is used to read the dataflow stored in the file. Its methods are as follows:

The ... method	is used to ...
boolean open(in String FileName)	open the file with the indicated name. The file must exist.
read (in String buffer)	read the file in the "buffer".
close ()	close the file.
existFile (in String FileName,out boolean answer)	test the existence of a file, using its name.

Example

We are going to write different values in a file.

```
boolean b=true;
int i= 23;
outStream MyFile;
MyFile.open("EXIT");
MyFile.write ("i == ",i, " b == ", b, NL);
MyFile.close();
```

The result in the "*EXIT*" file is as follows:

```
i == 23 b == true
```

Note: The "*inStream*" class combined with "eval" is used to load the formatted data.

Predefined output

The default output linked to the predefined output corresponds to the console if J execution is launched in Objecteering/UML.

The ... exit	corresponds to ...
StdOut	the predefined output for messages (screen or console).
StdErr	the predefined output for error messages.

Metamodel classes

Overview

The classes of the metamodel (as well as basic classes and work products) are the only classes available from J. Their definition can be extended by the addition of J methods or class attributes, but not by the addition of new instance attributes. New classes may not be created.

Access to a model's information is deduced from the graphical representation of the metamodel.

Note: The work product class (*MpGenProduct*) is an exception.

Accessing attributes

Access to an object's attributes is gained through the "." notation, by simply using their name:

```
Class  c = ...;
String n;

n = c.Name;
```

or in the context of a Class:

```
String n;
n = Name; // equivalent to : n = this.Name
```

Accessing associations

Access to associations is achieved by concatenating the role name with the association destination class name or one of its derived classes.

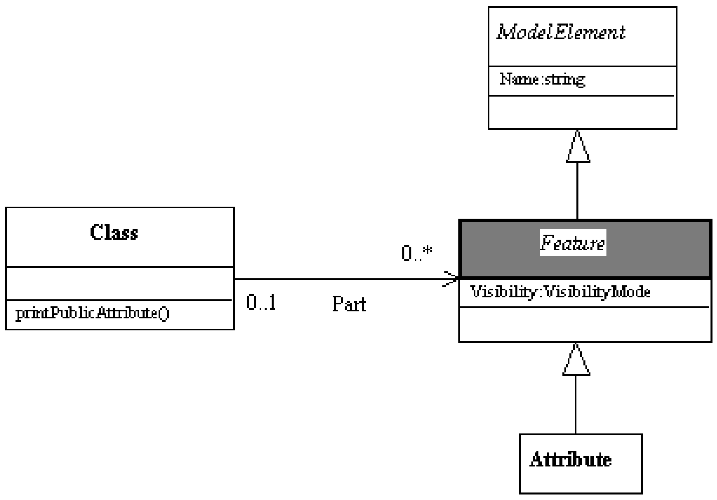


Figure 2-1. Extract of the metamodel used

```
Class c= ...
Feature[] theFeature;
Attribute[] theAttributes;

// Access to the class c features :
// role == Part , destination class == Feature
theFeatures = c.PartFeature;

// Access to the class c attributes
// (the attributes are features)
// role == Part, destination class == Attribute
theAttributes = c.PartAttribute;
```

The type of objects obtained also depends on the association's maximum multiplicity:

- If it is equal to 1, a simple object, possibly empty, is returned.
- If it is greater than 1, an object set, possibly empty, is returned.

Chapter 3: Methods, attributes and variables

Notion of reference

Definition

Local variables, operation parameters and class and instance attributes whose type is neither a basic class nor a set, are references towards the objects concerned. A reference is empty if it does not refer to any object.

The notVoid service can be used to test whether or not a reference is empty.

Example:

```
if (notVoid (ref))
   //non empty reference
...else
   //empty reference
```

Note: A void service, which carries out the opposite operation, also exists.

J methods

Definition

J methods support the definition of all J program processing. They are linked to J classes, presented in chapter 2 of this user guide.

Declaration

A J method called "*method_name*", linked to a J class called "*class_name*", in the default UML profile, will be defined as follows:

```
class_name: default#method_name
{
// J instructions
}
```

Note: The metamodel editor produces this syntax, and the J programmer need only enter the instructions which constitute the method body.

Example

We are going to write a specific "*print*" method, linked to the "*Class*" class and that displays the class name.

```
Class:print()
{
// the name of a class is contained in the Name
attribute
StdOut.write("PRINT ", Name);
}
```

Polymorphism/Access to the super class method definition

Like any other object language, the sending of a message supports polymorphism. We are going to use the previous example to distinguish attributes from other features ("*format*" example).

```
Class:printFeature()
{
   PartFeature.<print();
}

Feature:print()
{
   // The current object is a Feature
   StdOut.write("Feature : ", format(), NL);
}

String Feature:format()
{
   return Name;  // Very simple formatting...
}

String Attribute:format()
{
   // The formatting consists in prefixing
   // the name of '(Attribute) '
   return "(Attribute) " + super.format;
}
```

The "*super*" pseudo-variable, used in the "*format*" method of Attribute, is used to send a message to the current object (here it is "*format*") defined in its parent class. This characteristic authorizes code factorization in the parent classes.

Variables and parameters

Local variables

Local variables can be defined at the beginning of the J "*block*" (method body or anonymous methods). Their scope and life span are limited to their method definition. Their definition syntax takes the following form:

```
class_name variable_name; // or
class_name variable_name =initial_value;
```

A variable defined without any explicit initial value is initialized according to its type.

Type	Initial value
int	0
float	0.0
boolean	false
String	""
inStream, outStream	a stream in the closed state
other (non primitive)	empty

An explicit initial value can be a literal value, a variable already defined (local variable, attribute parameter) or an expression:

```
int i = 5;
int j = i;
String s = i.toString();
String n = Name;
Class c = this;
Feature[] som = PartFeature;
```

Predefined variables

Predefined variables are as follows:

The ... variable	represents ...
this	the current object.
StdOut, StdFile	the "*standard screen*" file for normal messages
StdErr	the "*error standard exit*" file.
NL	(New Line) String representing the passing to a new line.
Tab	(Tabulation) String representing the tabulation character.

Method parameters

Methods can have parameters. Parameters behave in the same way as local variables, and are defined as follows:

```
return_class class_name:method_name (in class1_name
Parameter1_name,
            inout class2_name parameter2_name);
```

In general, their form is:

```
Passing_mode Class_name parameter_name
```

Passing modes

Passing modes indicate whether the caller (the emitter of messages) should provide parameter values or whether he must wait for the resulting value.

The ... mode	defines a parameter as ...
in	input (provided by the caller).
inout	input/output (received by the caller).

Note: Techniques such as Java "wrappers" are not necessary with J.

Actual parameter types

Passing modes determine the actual parameter types that are compatible with those of formal parameters.

With the ... mode	the type of actual parameter must ...
in	be compatible (like the simple assignment) with the type of the formal parameter
inout	be identical to the formal parameter type.

Return instruction

As in Java, "*return*" exits the current method, with a value as parameter.

```
boolean Package : Generate()
{
   boolean Result;//false by default
   ...
 return Result;
```

"*return*" is also used to exit a method which returns nothing. In this case, there is no expression after "*return*".

Attributes

Overview

J provides the notion of attribute, which is the essential means of information storage and access. There are two families of attributes:

- ♦ instance attributes
- ♦ class attributes

Instance attributes

Instance attributes are attributes specific to each of the handled objects. The definition is imposed by the metamodel and is fixed. The J programmer cannot define new instance attributes (for example, the "*Name*" attribute), except for the work product classes (see *MpGenProduct* metaclass).

Class attributes

Class attributes are attributes whose value is shared by all the class instances. A J programmer can define new class attributes, but cannot modify those UML profiles delivered with Objecteering/UML.

Example of declaration

A class attribute is declared outside any method:

```
Attribute_Class_Name Class_name:Attribute_name
Attribute_Class_Name;
//For example
String Object:Environment
String Schema:Environment
String Class:Environment
```

Note: The *Objecteering/UML Profile Builder* tool produces this syntax.

Access to attributes

The "." and ".<" operators allow access to an object's attributes. When the current object's attributes are indicated, they are directly named.

For class attributes, the "." operator is used to explicitly indicate the attribute class. If the class attribute is defined on the class (or a parent class) of the current object, it is also possible to indicate the attribute directly.

The ... expression	**defines access to ...**
MyPackage.Name	the "*Name*" instance attribute of the "*MyPackage*" class or one of its parent classes.
Class:Environment	the "*Environment*" class attribute of the "*Class*" class or of one of its parent classes.

To access class attributes, it is recommended that you use the ":" operator systematically, even if the current context allows a simpler designation.

Anonymous methods

Definition

An anonymous method is a method without a name or parameters. Its body is embedded in a named method or in another anonymous method. The beginning and the end of the body are indicated respectively by "*{" and "}*".

Local variables can be defined before the first instruction.

Instructions can access local variables and method parameters. However, there is no direct access to the instance or class attributes of embedding objects. The use of "*super*" is not permitted.

Diffusion to an object

An anonymous method is always distributed to the object that comes syntactically before the "{" at the beginning of the body. In the body of the anonymous method, the "*this*" variable is initialized with this object or successively with each of these elements if it is a set.

Example 1

We are now going to concatenate all the attributes of a class in a character *String*.

Note: This program can be executed with the "*printAttribute*" command on a class.

```
Class:printAttribute()
{
   String line;
   PartAttribute  // diffusion to the attributes
   {  // beginning of the anonymous method
      //We are here in the context of an attribute
      String buffer = "," + Name;
      line = line + buffer
      //access to an embedded variable
    }//end of the anonymous method
      //return to a Class context
   StdOut.write("the ", Name, " class's attributes
                 are ",
                 line, NL);
}
```

Example 2

Let's take the previous example and apply the method to the Attribute's name directly, i.e. a character *String*, instead of applying it to the Attribute.

```
Class:printAttribute2()
{
   String line;
   PartAttribute.<Name  // diffusion to the attribute
                             // names
   {
      // this == a character String
      line = line + "," + this;
      // no more need for Name
   }
   StdOut.write("The attributes of the class ", Name,
                " are ",line,NL);
}
```

Chapter 4: J sets

Overview of J sets

Purpose

J programs use sets extensively in their processing. Sets facilitate the browsing of a model and allow the easy handling of groups of objects.

The metamodel has numerous cases of n-ary associations between classes (for example, package classes, class operations or operation parameters). Their access is easily processed through sets.

Notation

Sets are defined with the "*class of the elements[]*"class, such as *Object[]*, *String[]*, *Class[]*, etc.

Declaration

The following declaration is used to obtain an empty set of objects "*Object*":

```
Object[] E;
```

Order of the elements

Set elements are put in a certain order. The order of insertion in the set will be the order of the scanning of these elements when anonymous methods are diffused or applied. The order is very important in the metamodel. For example, the order in which class operations are accessed in the "*PartOperation*" object is the order in which these operations appear in the model, as well as in the parameters of an operation (*IOParameter*), etc.

Assignment between sets

Sets are considered as basic classes. The assignment between sets, therefore, copies the value of a set into another set. The example below shows a trick used to empty a set, given that a declared set is empty by default.

```
Feature[] E1;
Feature[] Emptied;
E1 = PartFeature;
...
E1 = Emptied; // E1 becomes an empty set
```

Example

The example below is used to list the names of all the parameters of all the operations of the current class. Firstly, an anonymous method is used, and then a diffused message carries out the same processing.

```
// First case : anonymous method
PartOperation
{
   IOParameter
   {
      StdOut.write(Name, ", ");
   }
}
// Second case : message diffusion
Parameter:Print()
{
   StdOut.write(Name, ", ");
}//End of "Print" method

//PartOperation
{
   IOParameter.<Print(); // message diffusion "Print()"
}
```

Empty set

The "*notVoid*" service can be used to test whether or not a set is empty.

```
If (notVoid (mySet))
   //non empty set
else
   //empty set
```

Note: A void service, which carries out the opposite operation, also exists.

Messages to sets (spreading)

Access to the i element

J sets do not currently allow instructions such as "*E[i]=V;*" or "*V=E[i];*" to be executed. The specific services below must be used:

```
Object:getItemSet (inObject[] pSet, in int pIndex, out
Object pElt)
```

Returns in "*pElt*", the "*pIndex*" rank element of the "*pSet*" set. The first element is at the 0 index. "*pSet*" and "*pElt*" types can be more precise (*String[]* and *String*, for example).

```
Object:setItemSet (inout Object[] pSet, in int pIndex,
in Object pElt)
```

Replaces the "*pIndex*" rank element by "*pElt*" in "*pSet*". The set must have an element at the "*pIndex*" index. The first element is at the 0 index. "*pSet*" and "*pElt*" types can be more precise (*String[]* and *String*, for example).

```
Object:sort(in boolean pAscendingSort, in String
pMetaAttributeName)
```

Sorts a set in ascending or descending order, according to the value of pAscendingSort. The sort criterion is provided by the name of a meta-attribute for metamodel objects, and by the value of the objects which make up the set for the basic classes (int, float, boolean, String, enumerate). For the latter, pMetaAttributeName is ignored and becomes optional.

Note: To directly sort modeling elements, the "*sortSemanticAssociation*" service should be used (please refer to the "*Model transformation primitives*" section in chapter 2 of the *Objecteering/J Libraries* user guide).

```
boolean Object:contains (in Object pElement)
```

Indicates whether pElement belongs to the set in question. If the set contains model elements, a reference to pElement is searched for. If a basic class is concerned (int, float, boolean, String, enumerate), an identical value is searched for.

```
Object:remove (in Object pElement)
```

Removes all references of pElement in the set for metamodel objects, and removes all basic classes (int, float, boolean, String, enumerate) of the same value for a set of basic classes.

Note: To remove a modeling element from a relationship, the "*erase*" service should be used (please refer to the "*Model transformation primitives*" in chapter 2 of the *Objecteering/J Libraries* user guide).

`Object:clear`
Empties the set in question.

"Size" message

The "*size*" or "*length*" operator is used to find out the number of elements in the current set.

Example:

Here, we are displaying a text according to the number of methods of the current class:

```
if (PartOperation.size() != 0)
{
   StdOut.write("The current class is composed of ",
      PartOperation.size(), " operations ");
else
   StdOut.write("The current class has no operation ");
}
```

"addElement" message

"*addElement*" is used to insert elements into a set. When a set is being scanned (diffusion, anonymous method, etc.), elements are scanned in their initial order of insertion. The example below shows another way to create the E set of public operations of a class.

```
PartOperation.<select(Visibility == Public)
{
   E.addElement (this);
}
```

"Add" message

The "*add*" operator is used to cumulate two sets by concatenating the operating set with the receiver of the message:

```
E1.add(E2); // the elements of E2 are added to E1.
```

"Retract" message

When the "*retract*" message is applied to a set, the last element of this set is withdrawn from it.

```
E1.retract();//the last element is retracted from the
set
```

"Select" message

The "*select*" operator is used to carry out a selection amongst the occurrences of a set, according to the boolean expression supplied in the parameter. The result is a sub-set of the initial set. On this result, it is possible to apply either an anonymous method, or a new "*select*" operator, or to carry out a message diffusion.

When the boolean expression is evaluated, the current object is the set's current element to which "*select*" is applied.

"*select*" filters the set before passing the resulting set to the rest of the expression; the filter boolean expression must not contain any term modified in the part of expression that follows the "*select*".

"Select" message: Example

You will find displayed here three different techniques for displaying a class's public operations.

```
//1: Anonymous method
PartOperation.<select(Visibility == Public)
{
   display ();
}
//2: Message diffusion (method "display")
PartOperation.<select(Visibility == Public).<display
();
//3: Passing by an intermediary variable
Operation [] E;
E = PartOperation.<select(Visibility == Public);
E.<display();
```

"While" message

The "*while*" operator is used to scan a set, while a certain condition is fulfilled. It scans all the occurrences until the stop scanning condition occurs.

When the boolean expression is evaluated, the current object is the set's current element to which "*while*" is applied.

The boolean expression is evaluated for each element. If it equals "*true*", the part of expression that follows "*while*" is evaluated. It is, therefore, possible to change a term of the boolean expression in the part that follows the "*while*".

"While" message: Example

In the example below, the attributes of a class are displayed until an attribute named "*Size*" is found or until all the attributes have been scanned.

```
PartAttribute.<while(Name != "Size")
{
   StdOut.write(Name);
}
```

Another way of writing it shows the modification of the boolean expression term in the anonymous method that follows.

```
boolean found;
PartAttribute.<while(found)
{
   found = Name == "Size";
   if (not(found) )
   {
      StdOut.write(Name);
   }
}
```

Generalization of while and select

The "*while*" and "*select*" messages are not only applied to sets, but may also be used with simple objects.

So the following example:

```
Operation: printIfPublic()
{
   if (Visibility == Public)
   {
      printMethod();
   }
}
```

can be written with a select (or a while):

```
Operation: printIfPublic()
{
   this.select(Visibility == Public).<printMethod();
}
```

Chapter 5: Statements

Flow of control

Overview

The J language has very few control structures. Those it does have are as follows:

- ♦ the conditional structure (if)
- ♦ the sending of a message structure
- ♦ set structures (*select, while, diffusion* and *anonymous methods*)

Conditional structure (if)

Conditional structure syntax takes a classic form:

```
if (condition1)
{
// instructions
}
```

or with the "*else*" instruction:

```
if (condition1)
{
// instructions
}
else
{
// instructions
}
```

or with the "*else if*" instruction:

```
if (condition1)
{
// instructions
}
else if (condition2)
{
// instructions
}
else
{
// instructions
}
```

Example

For any class of a given package and for any navigable association from this class, the example below is used to generate a descriptive sentence for document purposes. The sentence built depends especially on the multiplicity of the associations.

Note: The "*describeAssociations*" command on packages executes this program.

```
Package:describeAssociations()
{
   OwnedElementClass Scanning of all the package's classes
   {
      PartAssociationEnd.<describeAssociation(Name);
   }
}  End of example method

AssociationEnd:describeAssociation (in String className)
{
AssociationEnd currentAssociation = this;
String oppositeClassName;

// Describe only navigable associations
if (! IsNavigable()) return;

RelatedAssociation.<ConnectionAssociationEnd
{
    if (currentAssociation != this)
    {
        oppositeClassName = OwnerClass.Name;
    }
}
StdOut.write(NL, "Thanks to the", Name, "association, a
representative of", className, "is linked");
if (MultiplicityMin == "0")
{
    StdOut.write(" optionally to");
}
else if (MultiplicityMin == "1")
{
    if (MultiplicityMax != "1")
       StdOut.write ("to at least one representative and to
                      at most");
}
```

```
else
    StdOut.write(" to at least ", MultiplicityMin,
    "representatives and to at most");

if (MultiplicityMax == "1")
    if (MultiplicityMin == "1")
        StdOut.write ("to one and only one representative
                       of",
oppositeClassName,".");
    else
        StdOut.write ("at most one representative of",
oppositeClassName,".");
else if (MultiplicityMax == "*")
        StdOut.write("an unlimited number of representatives
                      of",
oppositeClassName,".");
else
    StdOut.write(" ",MultiplicityMax, "representatives of",
                  oppositeClassName,".");
```

Result

When this processing is applied to the classes of a package, the following sentences are produced.

```
Thanks to the ContractualLink association, a
representative of Insurance is linked to one and only
one representative of Man.
Thanks to the Direction association, a representative
of Man is linked optionally to an unlimited number of
representatives.
etc.
```

Boolean expressions

Boolean expressions are all sorts of expressions combining boolean values (variables, attributes, or a method's return value).

For more details on the boolean class, please refer to the "*Basic classes*" section of chapter 3 of this user guide.

Assignments

Simple assignment

Assignment is used to change the value of a variable (local variable, parameters, class attributes). Syntax is as follows:

```
variable = evaluated_ expression;
```

The type of the term on the right must be the same as the type of the variable, in other words:

- it is identical to the type of the variable, or
- it is the variable child class, or
- there is an implicit conversion of the type of term on the right to the variable type.

The effect of an assignment depends on the variable class type:

- for primitive or basic classes (int, String, etc.), the assignment copies the value of the origin in the destination.
- for non primitive classes, (classes of the metamodel, stream, etc.), the assignment only references the same object.

```
int i= 23;
int j;
Object[] E1;
Object[] E2;
Class CurrentClass;
Class MainClass;
j = i;
// Assignment of the content of i to j. i and j
reference two
// different objects
MainClass = CurrentClass;
// These two variables reference the same object, which
is
// one of the classes of the processed model.
E2 = E1;
// E2 contains all the elements of E1. They are copied
// if they are primitive objects, otherwise they are
referenced.
```

Assignment attempt

Simple assignment does not allow the assignment of a variable with a reference to a type that is parent to the variable's type without risking errors, even if the referenced object has a compatible type. To carry out this type of assignment safely, J provides an operator that attempts assignments. Syntax is as follows:

```
variable ?= evaluated_ expression;
```

With this operator, the variable takes the value of the term on the right if the type of this term is compatible with those of the variable. If there is type incompatibility, the variable is empty.

Example of an assignment attempt

In this example, we are in the context of a *ModelElement* and if the current object is a class, we want to display its features by calling the "*Class:printFeature*" method of the previous example.

```
ModelElement:printFeature()
{
   Class c ?= this; // 'c' is assigned if 'this' is
                    // a 'Class'
   c.<printFeature();  // 'printFeature' is only called
                    // if 'c' is a class.
}
```

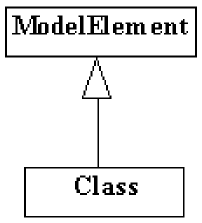


Figure 5-1. Extract of the metamodel used

Message sending and diffusion

Message sending

J methods are executed when messages are sent to managed objects. Messages are sent in the following way:

```
Object_name.method_Name();
```

We are going to send, for example, the "print" message to an instance of "Class".

```
Class c= ...
c.print();
```

Diffusion

Diffusion is a special case of message sending. It has the following characteristics:

- ♦ syntactic representation is ".<"
- ♦ the diffusion of a message to an empty reference or to an empty set is equivalent to a null operation (noop)
- ♦ the diffusion of a message to a set provokes the sending of a message to all its elements, instead of the destination set when a message is sent
- ♦ an expression containing a diffusion has the empty value (convertible into all types) if the diffusion cannot be carried out

Diffusion and sending a message to a non empty reference on a simple object are equivalent.

This mechanism avoids using "*for*" loops and empty reference tests.

Example of diffusion

We are going to create a specific "print" method, on the *Feature* class and then diffuse it to the features (members) of a class in the "*printFeature*" method. The "*print*" method uses a "*format*" method to format the name of the feature.

Note: This example can be executed on the classes using the "*printFeature*" command.

```
Class:printFeature()
{
   PartFeature.<print();
}

Feature:print();
{
   // The current object is a Feature
   StdOut.write("Feature : ", format(), NL);
}

String Feature:format () return
{
   return Name;  // Very simple formatting...
}
```

Diffusion can also be used in simple expressions:

```
String n = c.<Name; // n is worth "" if c is an empty
reference
```

General view of the "message concept"

A message can be generalized in anything that can provide a result:

- ♦ getting an attribute value (for example, "*Name*")
- ♦ getting associated elements (for example, "*PartAttribute*")
- ♦ calling a J method

As a result, a J method call, or access to an attribute or to a related element, can be used uniformly in the same kind of expressions.

"eval" statement

Overview

Certain J services provide control functions on the J interpreter. The "*eval*" service is a powerful feature of an interpreted language.

Other services exist, which can help debug a J program, stop the execution of a J program or obtain information on the current execution context.

eval

The "*eval*" service is used to calculate a character String dynamically, by considering its contents as being composed of J instructions.

When the String is evaluated, the current object does not change, it is the one in use when the eval is called.

<u>Note 1</u>: Declarations of local variables are authorized in the String which is to be evaluated.

<u>Note 2</u>: A return statement in the evaluated String provokes a return from the method (non anonymous) which contains the eval statement.

eval example

We are going to use "*eval*" to read the variable values previously stored in a file (Storage example):

```
Object:Storage();
{
outStream Fic;
Fic.open ("Example");
Fic.write ("i1 = 15; s1 = "hello";
            b1=false");
Fic.close();
Recover();
} - End of Storage

Object:Recover();
{
String read_buffer;

inStream Fic;
int i1;
String s1;
boolean b1;

Fic.open ("Example");
Fic.read (read_buffer);
eval (read_buffer);
//at this stage, the variables i1, s1, b1 are assigned.

StdOut.write ("i1=",i1,NL,
              "s1=",s1,NL
              "b1=",b1,NL);
}
```

Other services

J provides services which deal with the interpreter status or its invocation context.

The ... service	**is used to...**
`Object : exit(in int status)`	Exit the program execution underway. "*status*" indicates the output code and may only be recovered in "*batch*" mode.
`boolean Object:setTrace (in boolean mode)`	Activate or inhibit the J trace mode according to the "*mode*" value. Return the previous mode. The trace mode consists of displaying called methods and line numbers executed in the console, or in the execution window in "*batch*" mode.
`boolean Object:isStandaloneInvocation()`	Indicate whether or not the interpreter is launched from the command line (objingcl).
`Project Object:getCurrentProject()`	Return the current UML modeling project.
`String Object:getCurrentProfileName()`	Return the complete name of the interpreter's current UML profile.
`String Object:getStartUpProfileName()`	Return the complete name of the UML profile initially used upon the launching of the interpreter.
`Object Object:getStartUpObject()`	Return the destination object of the first message of the J program which is being interpreted.
`String Object:getCurrentModuleName ()`	Return the name of the module that the launching command belongs to may be empty.

Chapter 6: UML profiles

Overview of UML profiles

Definition

UML profiles allow you to consider just one part of the J class methods, according to your field of interest. They appear as prefixes of the methods which limit their visibility space. These prefixes are defined in the *Objecteering/UML Profile Builder* tool. They are never textually declared by the user.

Organizing J methods

The classes provided with the J language have a great number of methods, which all provide different features. These J methods allow you to produce C++ program generation of relational databases, documentation, etc. UML profiles are used to organize these methods, by taking into account their specific objectives.

First example

"*documentation*" and "*GenCpp*" are two UML profiles, to which the "*generate()"* methods belong. According to the UML profile in question, "*generate()*" indicates a different action:

```
Package:default#documentation#generate()
{ ...
}
Package:default#Cxx#generate()
{ ...
}
```

UML profile hierarchy

UML profiles are organized hierarchically, the root of the hierarchy being the "*default*" UML profile. This hierarchy is used to define a new UML profile, "*UMLprofile2*", which specifies a previously defined UML profile "*UMLprofile1*". UML profiles are, in fact, described in the form of a path which takes the following form:

```
default#UMLProfile1#UMLProfile2
```

Generation adaptation mechanism

Applied to all Objecteering/UML generation provided with the tool (C++, Java, etc.), the UML profile mechanism is used to redefine generation methods, in order to adapt them to a specific requirement. Here are the steps to follow:

1 - Define a UML profile in the UML profile dedicated to the generation in question (for example, *Cxx#Specific*).

2 - Redefine the method you wish to adapt (for example, *Attribute:Generate*) in the new UML profile.

3 - Run generation from this new UML profile.

Note: The *Objecteering/UML Profile Builder* module is required to run these actions.

Usage

Declaring UML profiles

UML profiles are declared in the *Objecteering/UML Profile Builder* tool.

Absolute or relative UML profile notation in programming

Absolute UML profile notation is carried out independently of the current UML profile. It starts from the root (default), either with a notation written "*default#P1#P2*", or by using the "#" character (*#P1#P2*). Relative UML profile notation can use the "-" symbol to move up the UML profile hierarchy, starting from the current UML profile.

Redefining methods

By default, a "*child*" UML profile owns all the features that are defined for its "*parent*" UML profiles. However, it is also possible to redefine in the child UML profile features present in the parent UML profile. In this case, the child's features mask those of the parent. For example, "*V3_2*" uses the "*Cxx*" generation rules, but redefines some of its methods.

Navigation example

In the "*Cxx*" UML profile, the "*generate*" method in "*V3_2*" can be accessed in several ways:

```
#external#Code#Cxx#V3_2#generate(); //Absolute notation
```
or

```
V3_2#generate(); // Relative notation
```

The "generate" method in "design" can be accessed in several ways:

```
#documentation#design#generate(); //Absolute notation
//or (Relative notation)
-#-#documentation#design#generate();
```

Current UML profile

The current UML profile is evaluated during execution. It corresponds to the last explicitly specified UML profile and can be:

- the UML profile that triggers the J program (that of the program's user).
- a UML profile explicitly specified, in an absolute or relative way during the method call. When the method returns, the previous UML profile becomes current again.

Relative UML profile

A relative UML profile is evaluated according to the declaration UML profile of the embedding method. UML profiles expressed in a relative way are not evaluated according to the current UML profile, which is known dynamically. This does not allow you to know, during the writing of the J method, the UML profile which is finally chosen.

Example

```
#external#Code#Cxx#generate() package; (1)
#external#Code#generate() class; (2)
#external#Code#Cxx#V3_2#generate() class; (3)
#documentation#print() class; (4)
#documentation#generate() class; (5)
```

If a user in the "*#external#Code#Cxx#V3_2#*" UML profile calls "*generate()*" on a package, (1) will be triggered. If the code of (1) triggers *"-#documentation#generate()*" on a class, the (5) method is called (declaration viewpoint = "*Cxx*"). If the code of (1) triggers "*generate()*" on a class, method (3) is triggered (current UML profile = v3_2). If this last method explicitly calls "*#documentation#print()*", then "*print()*" is called with the new current "*#documentation"* UML profile. In this case, a call of *"generate()*" will trigger method (5) available in this UML profile.

Accessing features of a UML profile without changing current UML profile

The "##" notation, placed between the UML profile and the feature, is used to specify a UML access profile, which allows access to the feature without changing the current UML profile.

Example

In the external#Code#Cxx UML profile, the "*init*" method calls "*init2*". In the #external#Code#Cxx#V3_2 UML profile, we want these methods to be redefined and the redefined version of init2 to be called by init of the #external#Code#Cxx UML profile.

```
//in the external#Code#Cxx UMLProfile
Class:#external#Code#Cxx#init()
{
   ...
   init2();
   ...
}
Class:#external#Code#Cxx#init2();
{
   ...
}
// the redefinitions in #external#Code#Cxx#V3_2
Class:#external#Code#Cxx#V3_2#init2();
{
   ...
}
Class:#external#Code#Cxx#V3_2#init();
{
   ...
   // Calling of init of the superior UML profile
   // without changing the current UML profile so that
   // the init2 of V3_2 be called by init of Cxx
   -##init();
   ...
}
```

Navigation rule

If, for example, an M method is called without any specified UML profile, J will search M firstly on:

- the current class and in the current UML profile
- a class parent to the current class and in the current UML profile
- the current class and in a parent UML profile
- a class parent to the current class and in a parent UML profile

Example

With the following method definitions:

```
Object:#default#Cxx#V3_2#generate();
Class:#default#Cxx#generate();
```

If the "*generate()*" message is sent to the "*Class*" class in the "*V3_2*" UML profile, the following method will be called:

```
"Object:"#default#Cxx#V3_2#generate();"
```

Structuring UML profiles

Overview

For obvious reasons related to structure stability and backward compatibility, Objecteering/UML is delivered with a preliminary fixed hierarchy of UML profiles. This hierarchy will be developed when new UML profiles are created, but the structure paradigm will be respected.

UML profile structure

Figure 6-1 presents the default UML profile hierarchy delivered with Objecteering/UML.

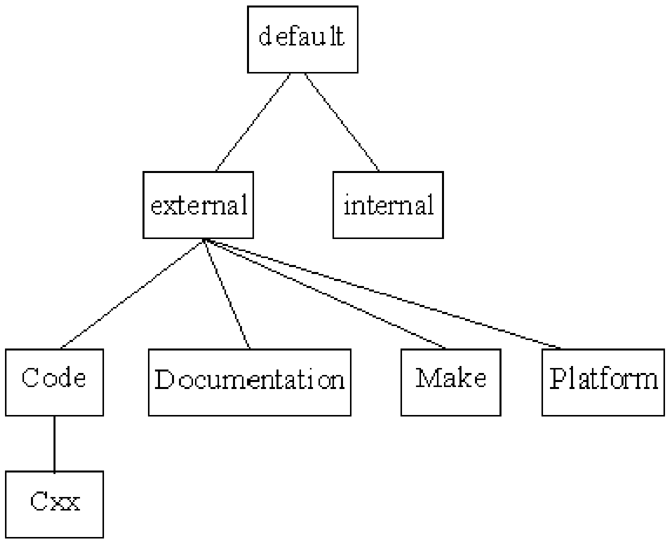


Figure 6-1. The UML profiles delivered by default in a UML profiling project

UML profiles

The ... UML profile	represents ...
internal	J rules developed internally, not accessible by the user.
external	generators and rules the user can access. The new user UML profiles must be created in "external".
Code	all code generators (C++, Java, SQL, etc.) which must be defined in this UML profile. Tagged values of general interest (NoCode, external) are defined at this level.
documentation	documentation generation.
Make	the generation of Makefiles (used with "Cxx" by the "GenC++" module).
Cxx	The C++ generator. The adaptation of the generator for new rules must be in this UML profile.
Platform	The characteristics of the platforms on which Objecteering/UML is available. The production String generation (Make) especially uses this UML profile. "PC", "UNIX", etc. UML profiles are found there.

Chapter 7: Executing J online

Executing J online - Overview

Purpose

As a command line, J can be used to launch a J program, such as *C++ generation* or *documentation generation*, directly from the operating system (a shell command line, an MS-DOS window, etc.) without having to run "*objing*".

There are two different modes according to the different operations that can be applied:

1 - The mode used to launch J, by running the UML Modeler tool. This mode is used to launch J methods on a model in the context of a UML modeling project, such as code generation, for example.

2 - The administration mode

End of execution

At the end of the execution of the J program, a backup is launched if:

- ♦ the interpreter has not encountered any errors in the J program
- ♦ the exit code has the value "*0*"

Administration services

The "*J on line command*" mode is useful for writing automated administration scripts. Using the "*baseadm*" command, a script can, for example, configure a database. Using a "*J on line command*", the script can then configure UML modeling projects, or carry out data exchanges between UML modeling projects.

Command line syntax

"project" mode

Syntax is as follows:

```
objingcl [-display <display>][-noDisplay] -db <base>
[-prj <project> -mdl <module> -cmd <commandName>
[-delim <delim>] <metaClass::objectName>
[-- {parameter}]
```

"administration" mode

Syntax is as follows:

```
objingcl [-display<display>] [-noDisplay]-admin -db
<database>  -mdl <module> -cmd <commandName> [--
{parameter}]
```

Example 1

To launch C++ generation ("generate"command) on the "A_cxx" work product, use the following syntax:

```
objingcl -db myBase -prj myProject -mdl CxxModule -cmd
generate::*::MpcCodeCxx:A_cxx
```

Note: The recognized name of the C++ generation module is CxxModule. The name of the metaclass which represents the C++ work product is MpcCodeCxx.

Example 2

This example presents the "run" command in a "*myModule*" module, for a given work product.

```
objincl -db myBase -prj myProjet -mdl myModule -cmd
default#external#mygenerator#run
MyTypeOfWorkProduct::*::foo -- -MyFileOption myFile.log
```

The -display/-noDisplay argument (UNIX only)

Some generation, such as documentation generation, require an X11 display to produce graphics. By default `objingcl` opens the predefined display (`:0.0` or the content of `DISPLAY` environment variable) and fails if it cannot open it. The `-display` argument indicates another display (for example, `-display zeus:0`) and the `-noDisplay` argument indicates that `objingcl` will not open any display. In the latter case, any graphical use will fail.

The -db database argument

This indicates the database name. The complete access path to the file is not allowed.

The -prj project argument

This indicates the working UML modeling project. The UML modeling project must exist in the database. If it is not specified, the J interpreter uses a UML modeling project with the same name as the database.

The -mdl module argument

This indicates the module which owns the command. The module must be present in the database and be selected in the UML modeling project. If required by the module, a license token is reserved during the execution of J.

Example: For C++, the module name is *CxxModule.*

The -cmd commandName argument

This indicates the name of the command or the J method to be triggered. If this is the name of a command, it must exist in the module. It is the name and not the label which appears in the menus. If it is a J method, it must be prefixed by the launching UML profile which respects the J syntax. The UML profile can be derived from the definition UML profile of the J method. It must be referenced by the module.

Example:

Considering the *generate* method defined in the UML profile:

```
default#external#Code#Cxx
```

for which a derived viewpoint exists:

```
default#external#Code#Cxx#MyCxx
```

For a generation which has this UML profile as the starting point, you simply have to indicate:

```
default#external#Code#Cxx#MyCxx#generate
```

The -admin argument

This indicates the administration mode. In this case, no objects are required.

The metaClass::objectName argument

The indication of the object(s) on which the interpreter is launched (target objects). *MetaClass* indicates the type of objects (*Class* for a class). It must not be a basic class, but the object real type. (For example, *ModelElement* cannot be used as Class instances).

Objects must all belong to the working UML modeling project. If several objects (with the same type) have the same name, the J program is triggered on each instance.

The object designation syntax can deal with nested and homonymous objects.

Syntax is as follows:

```
<metaClass>::<completeObjectName>
```

`<metaClass>` is the type of the object(s) (Class, Package, etc)

`<completeObjectName>` is the object access path followed by its name.

Example:

To designate the C1 class which belongs to P2 package, itself nested in the P1 package:

```
Class::P1::P2::C1
```

To shorten the designation, the whole or a part of the access path can be replaced by the metacharacter '*'. It can only be placed just after the metaclass.

Example:

To designate C1 more quickly:

```
(1) Class::*::C1
(2) (2) Class::*::P2::C1
(3) (3) Class::P1::*::C1 (error : * is not just after
  the metaclass)
```

A name containing '*' may indicate several objects ((1) all C1 classes, (2) all C1 classes belonging to all P2 packages). In this case, the J program is triggered for each object. In (3) C1 class could be found, if it exists a package named '*' in P1 package.

The `-delim` argument indicates a separator other than '::'.

Example:

To designate C1::1 class belonging to P1 package, the default separator can be replaced with '.':

```
objingcl ... -delim . Class.P1.C::1
```

Internal and external names of the modules

Modules are recognized by J through an internal name, which is different from the external name. For example, the *Objecteering/Documentation* module has the following internal name: "*GenDocModule*".

The ... module's external name	has for internal name ...
Documentation	GenDocModule
C++	CxxModule
Java	JavaModule

Name of the work product metaclasses

Generation work products have a different metaclass according to the generators and their types. For Objecteering/UML modules, the names are resumed in the table below.

The ... work products	correspond to the metaclass ...
Documentation	Document
C++	MpcCodeCxx, MpcMakeCxx
Java	Java

Parameters

It is possible to pass parameters to a J program from the command line. The list of parameters is prefixed by --. They are accessible through *getInvocationParameters* whose signature is:

```
String[] Object:getInvocationParameters()
```

Parameters generally appear in the same order as in the command line. They have the character type, and are divided by the surrounding shell. If there is a call from the modeler, the set is empty.

Example:

```
objingcl -db myBase -prj myProject -mdl myModule -cmd
myCommand Class::*::MyClass -- -verbose -tmp /var/tmp
'My result.log'
```

The example presents 4 parameters which are:

```
'-verbose', '-tmp', '/var/tmp', 'Myresult.log'
```

Chapter 8: J syntax

BNF form

Overview

J syntax is presented in Backus Naur Form (BNF), by applying the characteristics detailed below.

Case sensitive

Lower case characters are considered different from upper case characters.

Example: The "*StArt*" word is different from the "start" keyword.

Essential syntactical elements

Element	Description
Syntactic categories	In lower case. Linked by the underscore character (_). Example: method_definition, attribute_definition.
Specific words	In quotes. Example: 'this'
Optional elements	In brackets. Example: [UML profile]
Repetitive elements	In cokebottles ({}). Can appear one or more times. From left to right.

Alternatives	They are separated by a vertical bar.
	Example:
	`values:=`
	`<name |  figure name`
	`| '('expression [','expression]')'>`

Category name	In italics. Equivalent with the name of the categories not in italics.

Note: BNF would be more legible if the key words (if, while, select) were in bold instead of between inverted commas.

J syntax

J program

With J syntax, UML profiles, class attributes and methods have to be declared in the right order.

```
Program::=
         {attribute_declaration}
         {method_definition}
```

Class attributes

J class attributes can be declared with an initial value.

```
attribute_declaration::=
         class ':' [profile] attribute_name ':'
              class [assignment]';'
```

Method declaration

Method declaration is made up of the class which owns the method, its UML profile and its name, followed by its parameters and its possible return value.

```
method_definition::=
   class ':' [profile] method_specification
method_specification::=
   method_name parameter_designation_list
   [return class]
parameter_designation_list::=
         '(' [ <parameter_designation>
         {',' <parameter_designation>}])'
parameter_designation::=
   passing_mode class parameter_name
passing_mode::= 'in'|'inout'
```

Designating UML profiles

The designation of the UML profile is useful, either to declare a method, or to define explicitly the UML profile during the message sending. The designation of the UML profile is in absolute mode (starting from "default" or "#"), or relative mode (starting from the current UML profile and possibly using the moving up "-").

```
UML profile::=
         [header_profile] [{profile_item}]
         profile_name profile_mode
header_profile::= <default |'#'>;
profile item::= profile_name '#';
profile_name::= <name | '-'>;
profile_mode::= <'#' | '##' >;
```

Designating a class

Classes are predefined, and will be either basic classes (integer, etc.), or metamodel classes (see class). The "[]" prefix is used to designate sets.

```
Class::= class | class'[]'
```

Defining blocks of instructions

Blocks of instructions appear either to define the content of the methods, or to define anonymous methods, linked to the sets resulting from a "select" or "while" command. At the head of each block, local variables can be defined.

```
Block::=
   '{'
            {variable_declaration}
            {statement}
   '}'
```

Declaring local variables

Local variables are declared at the head of each block. They can have an initial value.

```
variable_declaration::=
   class variable_name [assignment]
```

Instructions

Instructions can be:

- a condition (if, else, etc.)
- an assignment (=, ?=)
- the sending of a message (object, method (parameter))
- a message diffusion (object.<method (parameter))
- the calling of an anonymous method

```
Statement::=
   if_statement
   | anonymous_statement
   | message_call ';'
   | [assign_statement] ';'
```

"if" Instruction

The "if" instruction owns the "else" or " else if" operators, which allow you to easily combine several possible cases ("case" not being present).

```
if_statement::=
   'if' '('expression')'
         statement | compound_if_statement
   ['else' statement|compound_if_statement
compound_if_statement::=
   '{'
         {statement}
   '}'
```

Assignment

```
assign_statement::= assignable assignment-operator
expression
assignable::=
variable_name|parameter_name|attribute_name
assignment operator::= '='|'?='
```

Message sending/Message diffusion

The sending of a message consists of accessing an object, then designating a method name and its parameters. Dynamic evaluation will launch the concerned method for the given object.

```
message_call::=
        instance_access call_operator
        [UML profile] method_name argument_list
instance_access::=[message_call | instance | iterator ]
                    {call_operator [message_call |
instance |
                                    iterator ]}
instance::= attribute_name | variable_name |
            MetaModelInstance |
            'this' | 'super' | 'StdOut' | 'StdErr'
iterator::= 'select' '(expression)' | 'while'
'('expression')'
call_operator::= '.' | '.<'
argument_list::=
         '('[<argument> {','argument}]')'
argument::= instance_access | integer_value |
String_value
| boolean_value | method_statement | set_access |
'this'
```

Anonymous methods

An anonymous method always comes after an expression.

```
anonymous_statement::= message_call
```

Expressions

```
expression::= argument | boolean_expression |
arithmetic_expression
   | '('expression')'
boolean_expression::= expression '||' expression
                  | expression '&&' expression
                  | expression '==' expression
                  | expression '!=' expression
                  | expression relational_operator
expression
                  | '!' expression
relational_operator::= '<' | '<=' | '>' | '>='
arithmetic_expression::= expression arith_op expression
arith_op::= '+' | '-' | '/' | '*' | '%'
```

J classes

J classes are basic classes, "stream" utility classes and metamodel classes.

```
basic_class::=
   MetaModelClass | 'Object' | 'String' | 'boolean' |
'int'
   | 'float' | 'inStream' | 'outStream' | 'stream'
```

Basic values

```
boolean value::= 'true' | 'false'
String_value::= Any symbol between " and "
int_value::= Any integer value
```

Index