srd

Kopi Application Generator : User guide

Table of Contents

Preface

The Kopi Project is a Java software project from DMS which provides a framework for developing database applications using Java, JDBC and SWING.

Kopi contains a set of tools which allow you to edit and generate classfiles: dis (Java disassembler), KSM (Java assembler) and KJC. KJC compiles Java source code to bytecode, with all the same plus even more features as commercial compilers. KJC is available for free under the terms of the GNU Public License.

The Kopi Project also includes XKJC, a compiler for embedded SQL in Java. Built over JDBC, it allows the execution of Java-typed SQL statements and the mixing of expressions from both SQL and Java.

The last tool provided by DMS is Visual Kopi. Visual Kopi is an application framework using JDBC and JFC/Swing which lets you create database applications in a high level specification language. It also provides the ability to write triggers and commands in Java with seamlessly embedded SQL statements.

Organization of This Document

This documentation includes two main Chapters:

- CHAPTER 1 : XKJC - The EXtended Kopi Java language+ XKJC is a compiler for embedded SQL in Java. This language is a super set of Java; i.e., it is compatible with Java source code. This allows the execution of Java-typed SQL statements and the mixing of expressions from both SQL and Java.
- CHAPTER 2 : VLIB - The Visual Kopi application Framework+ Visual Kopi is an application framework using SWING which lets you create database applications in a high level specification language. In this chapter, we will explore the capabilities of this framework.

Getting Started

The source code of the KOPI project along with the Kopi Suite binaries, is available on SourceForge http://sourceforge.net/projects/kopi/.

Chapter 1. XKJC - The Extended Kopi Java language

This language extends the java language's capabilities and offer a set of additional tools. The most important additions of the xkjc are the new types introduced and the SQL integration to java code. To sum up, the extended kopi Java language is layer over the standard java language, meaning that all java statements are understood by the Kopi language, plus some new features and tools. The kopi files extension is (.k).

1.1. XKJC : The new Types

Kopi introduces 6 types to the basic types available in the java language :

- Date : date values that can be null, you can also use "date" for not null Date type values.
- Month : Years months values and can be null, you can use "month" for not null Month type values.
- Week : Years weeks values and can be null, you can use "week" for not null Month type values.
- Time : this type can be used to contain time values (hh:mm:ss), "time" for not null values.
- Timestamp : can contain timestamp values and null, "timestamp" for not null values.

1.2. XKJC : Java with embedded SQL

This language allows developer to define triggers and actions in the XKJC language that is a superset of Java with embedded SQL. XKJC allows the mixing of Java-typed expression from both SQL and Java.

Since SQL statements are parsed, it allows one to write a very clear code as compared to JDBC (see XKJC vs JDBC section). It allows one to make requests with embedded Java expressions computed at runtime. It comes with a support for BLOB and Java serialization mechanism, allowing to store Java objects in any JDBC database and to retrieve them both nicely and efficiently.

As in SQL all types have a NULL value, the distinction between primitive types and Object types of the Java language are problematic. To handle this, XKJC contains an operator overloading capability that allows the mixing of primitive types with objects.

The mechanism of overloading looks like the one provided by C+.+ compiler except that only expression operator are overridable (`,` `-`, `*`, `/`, `%`, `==`, `!=`, `&`, `|`, `<<` `>>`, `>>>`, `^`, `~`, `+`, –) and the cast operation. There is no implicit conversion, and that make it easy to understand.

There is an example of a typical XKJC program:

```
#cursor(int id) {
  SELECT Vector Obj
  FROM   BLOBS
  WHERE  ID = :id
} typed;
typed.open(6);
if (typed.next())
  for (int i = 0; i < typed.Obj.size(); i++)
    System.out.println("vect(" + i + "): " + typed.Obj.elementAt(i));
typed.close();
```

By providing an easy way to include SQL statement inside your Java code (not between double quote), XKJC made it more easy to read and maintain. With explicit typing the compiler check that expression are correct and involve the overloading mechanism to execute expression such as:

1.2.1. Connections

By providing explicit connection to SQL statement you are able to work on more than one database at the same time. But since common applications work with only one database, we provide a mechanism of implicit transaction that allows to save time by wrinting:

```
#execute{SELECT COUNT(*) FROM Cars INTO count};
```

instead of:

```
#execute[Main.getDatabase().getFreeConnection()] {
  SELECT COUNT(*) FROM Cars INTO count
};
```

1.2.2. Cursors

The cursor are the common way to get more than one row of data from the database. It's like ResultSet but it's typed:

1.2.3. SQL SELECT INTO Statement

The only SQL statement in XKJC is the #execute that allow to fetch data from the database with a "SELECT INTO".

Examples:

```
1. #execute {
   SELECT String name, Image image, BigDecimal speed
   FROM   Cars
   WHERE  ID = 1
   INTO   theName, theImage, theSpeed
 };


2.  #execute {
   SELECT String name
   FROM   Cars
   WHERE  ID =:id
   INTO   names[id]
 };


3. #execute[conn] {
   SELECT COUNT(*)
   FROM Cars
   INTO count
 };
```

The last example use an explicit connection while the others use an implicit connection (ie, the context should inherit from DBContextHandler and its default connection will be used).

1.2.4. SQL Expressions

SQL expression returns an int value that correspond to the number of row modified by the expression. An expression can use a cursor to identifie a row of the database (WHERE CURRENT OF) with fully implemented JDBC drivers.

Examples:

```
1. #update {
  UPDATE Cars
  WHERE  ID =:id
  SET    name = :(javaName + '-' + id)
};

2. #update(cursor) {
  UPDATE Cars
  SET    name = :(javaName + '-' + id)
};


3. #update [conn] {
  DELETE FROM Cars
  WHERE :id > 10
};
```

The last example use an explicit connection while the others use an implicit connection (ie, the context should inherit from DBContextHandler and its default connection will be used).

1.2.5. Transactions

Every SQL statements should be executed within transaction. This allows to re-execute statements interrupted by deadlock (after asking the user if he wants to) or abort a whole transaction if something is wrong (like if a Java exception is thrown). The syntax is:

```
#protected([``an optional message''']){
  ...
  ... // a list of Java and SQL statements
  ...
};
```

Chapter 2. VLIB - The Visual Kopi application framework

The visual Kopi Framework is an application framework based using the XKJC language in order to create database applications easily. In this chapter we will se how to create kopi powered forms,reports and print pages.

2.1. Generalities

In this section we will provide the definitions you may need in order to fully understand the syntax explanations in the rest of the chapter. We will also provide the standard types used in the visual kopi framework.

2.1.1. Syntax terms

During this chapter, you will have many **Syntax** sections containing different keywords and special characters :

- **SimpleName and QualifiedName**

When setting up your own application, you will often have to enter a name in order to define either a field, a block, a page or a form. You will then have to make use of either a simple or a qualified name.

A SimpleName consists of a character row in which neither spaces nor points are permitted.

Example

```
StudentGroup2
```

A QualifiedName is a row of characters in which points are allowed.

Example

```
StudentGroup2.ViennaUniversity
```

- **Parameters**

You can also find parameters with types in the syntax blocks;

Example

```
Integer x
String name
Fixed y
```

- **Special Characters**

These characters may be found ins syntax block and defines the rules of the syntax:

[ ] : if an element in put between [ ] that means it's optional.
* : if an element or expression is followed by *, then it can be inserted zero or more times
+ : if an element or expression is followed by +, then this element need to be inserted at least one time, or more
| : means Or
" " : if and element or expression is put between " ", then it's a keyword of the language.

Example

```
ItemDefinition  : "FORM" String formTitle ["IS" QualifiedName]
                  ["IMPLEMENTS" QualfiedName [,QualifiedName]* ]
```

2.1.2. Predefined Field Types

Every form or report field In visual kopi have field type that can be one of the 12 ready to use predefined types :

```
- FIXED     // used to insert fixed values
- IMAGE      // used to insert images
- INTEGER    // used to insert integer values
- STRING     // used to insert string values
- TEXT       // used to insert text values
- DATE       // used to insert a date value
- BOOL       // used to insert a true or false value
- COLOR      // used to set the fields color
- MONTH      // used to insert years months value
- TIME       // used to insert a hours:minutes time value
- TIMESTAMP  // used to insert a timestamp value
- WEEK       // used to insert years weeks  value
```

2.1.2.1. The STRING Field Type

A STRING is used to enter characters which can be either letters, numbers or both. The width has always to be given. Moreover, you can optionaly indicate how many rows it will contain and how many will finally be displayed on the form. If these optional arguments are used, you have to indicate the carraige return method by specifying either the FIXED ON or the FIXED OFF option to avoid compilation errors.
There are also three other options you can use in order to modify the String's case :

- CONVERT NAME (Converts the first letter of each word to capital letter)
- CONVERT UPPER (Convert the whole text to capital letters)
- CONVERT LOWER (Converts the whole text to normal letters)

Syntax:

```
StringType   : "STRING" (Integer width ,Integer height ,Integer visible)
               [FixOption]
               [StringFormat]

FixOption    : "FIXED" "ON"
               "FIXED" "OFF"

StringFormat : "CONVERT" "UPPER"
               "CONVERT" "LOWER"
               "CONVERT" "NAME"
```

Example

```
STRING (40,10,4)
FIXED ON
CONVERT UPPER
```

In this example, the text inserted will contain up to 40 characters and up to 10 rows. However, only the first 4 rows will be displayed on the form. Moreover, All the letters in the text will be converted to capital letters.

2.1.2.2. The TEXT Field Type

A Text and a String are similar apart from the fact that in a text, two parameters have always to be given: namely the width and the height of the field whereas you only need to determine the width in a string.

Syntax:

```
TextType      : "TEXT" (Integer width ,Integer height ,Integer visible)
                [FixOption]

[FixOption]   : "FIXED" "ON"
                "FIXED" "OFF"
```

For example, you can write STRING (40, 10, 4) or STRING (40) but you have to write TEXT (40, 10) or TEXT (40, 10, 4).

2.1.2.3. The IMAGE Field Type

This field type is used to insert an illustration or a picture. When introducing an IMAGE, you have to determine its width and height. These values have to be integers and are measured in pixel. In this case, the two attributes are compulsory.The field will look like a file chooser that lets you choose and image file to show in the field.

Syntax:

```
ImageType : "IMAGE" (Integer width , Integer height)
```

Example:

```
IMAGE(20,10)
```

In this field, the image will have a width of 20 pixel and a height of 10 pixel.

2.1.2.4. The FIXED Field Type

A FIXED is used to insert numbers, integers, fixed point numbers. Fraction numbers are entered with the field type FRACTION. The maximal width has to be determined for all them. In case of a fixed point number FIXNUM, the maximal scale i.e the number of characters standing after the comma has also to be defined. Also the comma has to be counted as a character. Only the width is to be defined in a FRACTION. You can also set the minimum and the maximum values for the FIXNUM field with the optional parameters MINVAL and MAXVAL.

Syntax:

```
FixedType   :  "FIXNUM" (Integer width ,Integer scale)
               ["MINVAL" Fixed minValue]
               ["MAXVAL" Fixed maxValue]

FractionType : "FRACTION" (Integer width)
               ["MINVAL" Fixed minValue]
               ["MAXVAL" Fixed maxValue]
```

Example:

```
FIXED(4,2)      // for 1,25
  MINVAL 0
  MAXVAL 3.0


FRACTION(7)    //  for 1 35/64
  MINVAL 0
  MAXVAL 25
```

2.1.2.5. The INTEGER Field Type

Integer field type is LONG is used to insert integers. Only the text width is to be defined. The MINVAL and MAXVAL options are also available for this type.

Syntax:

```
IntegerType:   "LONG" (Integer width)
               ["MINVAL" Integer minValue]
               ["MAXVAL" Integer maxValue]
```

Example

```
LONG(6)
  MINVAL 0
  MAXVAL 100
```

2.1.3. Code Field Types

In addition to the predefined field types already available in visual kopi, you can define more specific types which are the ENUM type and the CODE type. Unlike the predefined field types, these types have to be set in the Type definition type of the form before you can use them in the fields.

2.1.3.1. The ENUM Type

The ENUM type means enumeration or listing. An enumeration definition is made up of one or several strings and only the strings you have listed can be entered in the field.

Syntax:

```
EnumType:   "ENUM" (EnumList)

EnumList:   String member [,EnumList]
```

Example

```
ENUM ("X-Small","Small","Medium", *  Large", *  X-Large" )
```

2.1.3.2. The CODE types

There are four different sorts of CODE : * The CODE BOOL * The CODE LONG * The CODE FIXED * The CODE STRING

These types enable you to have a list of item-value pairs, the items will be displayed in the field and the values will be assigned instead.

- The Type CODE BOOL

In a CODE BOOL or BOOLEAN you have to assign a Boolean value to the item you have entered. Boolean values are values like "True" or "False" and "Yes" or "No".

Syntax:

```
CodeBooleanType : "CODE" "BOOL" "IS"
                  CodeBooleanList
                  "END" "CODE"

CodeBooleanList : CodeBoolean  [CodeBooleanList]

CodeBoolean     : String code "=" Boolean value
```

Example

```
CODE BOOL IS
  "married" = true
  "single"  = false
END CODE
```

- The Type CODE LONG

In a CODE LONG, you assign to each String item you have entered a LONG value.

Syntax:

```
CodeIntegerType : "CODE" "LONG" "IS"
                  CodeIntegerList
                  "END" "CODE"

CodeIntegerList : CodeInteger  [CodeIntegerList]

CodeInteger     : String code "=" Integer value
```

Example

```
CODE LONG IS
  "Monday" = 1
  "Tuesday" = 2
  "Wednesday" = 3
  "Thursday" = 4
  "Friday" = 5
  "Saturday" = 6
  "Sunday" = 7
END CODE
```

- The Type CODE FIXED

In a CODE FIXED, each item you have entered will get a FIXED value, i.e integers, fixed point numbers and fraction numbers.

Syntax:

```
CodeFixedType : "CODE" "FIXED" "IS"
                [CodeFixedList]
                "END" "CODE"
```

```
CodeFixedList : CodeFixed  [CodeFixedList]

CodeFixed     : String code "=" Fixed value
```

Example

```
CODE FIXED IS
  "piece" = 1.00
  "per cent" = 0.01
END CODE
```

- The Type CODE STRING

In a CODE STRING, each item you have entered will get a STRING value, this can be useful for shortcut of long strings for example.

Syntax:

```
CodeStringType : "CODE" "STRING" "IS"
                 [CodeStringType]
                 "END" "CODE"

CodeStringType : CodeString  [CodeStringType]

CodeString     : String code "=" String value
```

Example

```
CODE STRING IS
  "JDK" = "Java Development Kit"
  "JRE" = "Java Runtime Environment"
END CODE
```

2.1.4. The LIST command

Once you have defined a field type, you can make use of the LIST command in order to refer the user to a list or a table in the database which will help him when filling in the form in question.
If you refer a field type to a certain table, the field will get an icon on which you can click in order to retrieve this table. As this command connects the user with a certain table, you have to enumerate all columns of the table which information could be helpful for the user.
In so doing, you will then have to enter at least one column. The information contained in the first element of the list must have the same standard type as the defined type as it is the one which will be entered in the field in question. In addition, you may choose to refer to an existing form (that should extend the VDictionaryForm class) using either the NEW command to get a button on the bottom of the list allowing you to get to the referred form, or the ACCESS command to bypass the list and get directly to the referred form.

Syntax:

```
List        : "LIST" String tableName ["NEW" | "ACESS" QualifiedName ]
              "IS" ListColumns
              "END" "LIST"

ListColumns : ListColumn  [ListColumns]

ListColumn  : [String field "="] SimpleName ":" Predefined Field Type
```

This command is usually used when defining a new type, here is an example where we define the Lecturer Type as a STRING(8) value from the Symbol field of the "Lecturer" table on the database, when clicking on a field of Type Lecturer, you will have a list with three columns (Symbol, Surname and Lesson) retrieved from the "Lecturer" table. Selecting a row from this list will put the symbol value in the field.

Example

```
TYPE Lecturer IS
  STRING (8)
  CONVERT UPPER

  LIST "Lecturer" IS
    "Symbol" = Symbol : STRING (8)
    "Name" = Name : STRING (40)
    "Surname" = Surname : STRING (40)
    "Lesson" = Lesson : STRING (20)
  END LIST
...
END TYPE
```

2.1.4.1. The SELECT command

As you had the possibility to call up a list or a table from the database with the option LIST, you now can make Kopi sort out information from a whole list or a table which the option SELECT and this, according to criteria you have to define. Let's say you want to view the degrees which can be achieved in a certain year. For this, you have to use the SELECT command as you see in the following rows:

Example

```
TYPE CurrentDegree (Integer year) IS
     STRING (8)
       CONVERT UPPER
  LIST{(
        SELECT Symbol, Description
        FROM   Degree
        WHERE Year = : (year)
        )} IS

        "Symbol" = Symbol : STRING (8)
        "Description" = Description : STRING (40)
  END LIST
END TYPE
```

"Degree" is the database table to which we have to access in order to select the information. The command SELECT is used in order to enter the columns in which Kopi has to make its research. After FROM, you have to enter the table from which these columns are to be selected and after WHERE, you have to enter the conditions according to which this selection has to be carried out.

The sign "=" means the value of the data delivered after the selection must correspond to the conditions you have entered. The sign ":" inserts one Java expression. After this Java expression, you can insert a SQL expression again. (See JLS 15.27) Finally, you have to define for each of them the field type.

As a result of your selection, you will then have a table with two columns which will contain the different sorts of degrees achievable in the year you have entered.

2.2. Visual Kopi Forms

The aim of Kopi is to enable you to create applications. An application is made up of forms. A **form** is a set of **pages** consisting of **blocks**. A **block** is nothing else than a table which is divided into columns (vertical) and rows (horizontal). A row is made up of **fields**, and each **field** contains a data value at the intersection of a row and a column that can be of different **Types**. By the end of this chapter, you will learn how to create a form, but before, you need to learn how to create a field and how to form blocks.

2.2.1. Creating Form Fields

In Visual Kopi, a fields are containers in which you can define different types of data. A field definition begins with an access modifier followed by an optional SimpleName and ends with END FIELD.

Syntax

```
FieldDeclaration  : AccessModifier [FieldHeader] [HelpText] FieldType
                    FieldBody
                    "END" "FIELD"

FieldHeader       : [(Integer multiField)] [SimpleName] [FieldPosition]
                    [FieldLabel]
FieldBody         : "IS" QualifiedName [FieldColumns] | [FieldAligement]
                    [FieldDropList]     [FieldOptions]    [FieldColumns]
                    [FieldCommands] [FieldTriggers]
```

2.2.1.1. Field Access Modifiers

An access modifier determines the way the application user will have to handle each field in a form as well as the field accessibility. This command is compulsory. There are 4 sorts of access modifier:

Syntax:

```
AccessModifier:   "MUSTFILL"
                  |
                  "VISIT"
                  |
                  "SKIPPED"
                  |
                  "HIDDEN"
```

- MUSTFILL : As indicated, a MUSTFILL field has to be filled by the user. This field will be colored in blue.
- VISIT : A VISIT field is a field which the user can fill if he wants to. In Visual Kopi, this field will be colored in green.
- SKIPPED : A SKIPPED field is a field which is displayed on the form but which the user can not overwrite. A field is skipped if the user needs to know the information written in it. This field is always colored in black.
- HIDDEN : A HIDDEN field that is not displayed on the form as the information it provides is of no importance for the user, this kind of fields is usualy used for database table join operations or for ID fields.

As the user moves from a field to another when filling the form, the current field is always colored in red.

Example

```
MUSTFILL Year
  ...
END FIELD

HIDDEN ID
  ...
END FIELD
```

2.2.1.2. Multiple fields and Single fields

After specifying the field's Access modifier, you can use an integer to transform the field to a multiple field, meaning that the field is displayed as much times as the integer you defined. The fields created will have the same name plus the number of the field. **Example**

```
VISIT (3) Email
  ....
END FIELD
```

This means that the Email field will be displayed 3 times on the form with the names Email1 Email2 and Email3.

2.2.1.3. Field Position

This entry defines the position of the field in the current block. HIDDEN fields does not have positions, and in multiple block (Defined in the Blocks creation Section of this document) you may use the NO DETAIL Block option to pass positioning fields. There are two possibilities to define the field position.

Syntax:

```
FieldPosition : "AT" "<"Integer row  [,Integer column [-Integer multi]]">"
                | "FOLLOW" SimpleName
```

- Absolute Position With AT

  You can define it according to one integer or more:

  - AT <Integer row>
  - AT <Integer row , Integer column>
  - AT <Integer row, Integer column-Integer multifield>

Example

```
VISIT (4) Customer AT <1, 1-4>
  ...
END FIELD
```

The first integer indicates the row number while the second defines the column. In fact, Kopi automatically divides up the window in rows and columns when setting up a form.
When defining the width of a column, thus, it always adopts the width of the longest field. Since the form wide is not unlimited, you can put a long field in two or more columns in order to spare place as it is the case in the example above and it is usually used for multiple fields.

- Relative Position With FOLLOW

You can also make use of the following structure : FOLLOW SimpleName

Example

```
VISIT Email
  FOLLOW Name
  ...
END FIELD
```

If you enter a new field with this option (in this case it would be the field "Email"), it means that this field will be placed directly next to the first field defined by the FOLLOW command (field "Name") on the same row. The two fields will then form one single column.

2.2.1.4. Field label

The field label is optional, you declare it with the LABEL command that comes just after the position definition, the syntax is the following :

Syntax

```
FieldLabel : "NO" "LABEL" | "LABEL" [ String label [,String labels]* ]
```

If you don't want your field to have a label, you can use the NO LABEL command, otherwise use the LABEL command, in this case, either you specify a list of String separated by a comma so the first one will be the field's label, or you do not specify any String and the SimpleName that comes after the field's access modifier will be the field's label.

Example

```
VISIT firstName  AT <1 , 1-2>
  LABEL
  ...
END FIELD

SKIPPED secondName
  FOLLOW firstName
  LABEL "The second Name","another Label for this field"
  ...
END FIELD
```

2.2.1.5. Field Help Text

HELP is used to insert an explanation text for the application user in order to help him filling in the form. This Help text will then be displayed when the user places the mouse on the field.

Syntax:

```
HelpText:   String HelpText
```

Example

```
MUSTFILL Lesson
  LABEL
  HELP   "The lesson you would like to attend"
  ...
END FIELD
```

2.2.1.6. Field Types

The field type is a required entry, you may enter a predefined field type :

Example

```
MUSTFILL
  NO LABEL
  LONG(10)
END FIELD


VISIT Name
  STRING(20)
END FIELD
```

You can also use self defined field types that you have previously defined in the type definition section of your form.

Example

```
SKIPPED objectList
  LABEL
  TYPE Objects
END FIELD
```

In this example, Objects is a type you should have defined previously in type definition section of the form using standard types, CODE types, ENUM types, LIST and SELECT command ...

2.2.1.7. Field Alignment

This command is used to define the localization of the field's content inside the field. There are three types of alignment.

- ALIGN RIGHT the value is displayed at the right inside the field
- ALIGN LEFT the value is displayed at the left inside the field
- ALIGN CENTER the value is centered in the field

Example

```
VISIT field
  LABEL
  FIXNUM(9, 4)
  ALIGN CENTER
  ...
END FIELD
```

2.2.1.8. Field Drop files

This command is used to make a field accept to drop files into it, meaning you can drag files and drop them in your field.

Syntax:

```
DroppableDefinition : "DROPPABLE" ExtensionList
ExtentionList       : String extension [,ExtenstionList]
```

Example

```
VISIT images
  LABEL
  STRING(20)
  DROPPABLE("pdf", "jpeg", "tif", "tiff")
END FIELD
```

In this example, you can drag image files and drop them in the field named "images".

2.2.1.9. Field Options

In this part of the field definition, you can use one or more option from the 10 options available for fields in Kopi, here is the a list of these field options.

- NOECHO : If this option is used, characters typed in the field will not be displayed and a star(*) will be displayed instead, this option is useful for password fields.
- NOEDIT : This option makes it impossible to change the data of the field or to overwrite it.
- SORTABLE : This option adds two opposed arrows icons(up and down) just before the field, clicking on the icon changes the way data are sorted in the field, you can click the icon three times to have ascending sort, descending sort and default sort,
- TRANSIENT : This option make the field transient, meaning that the system can no trail it, if a transaction calls this field and then this transaction is aborted, the field will not be able to backup or roll-back to its original value, besides this option makes changes ignored for this field.
- NO DELETE ON UPDATE : If the field is a lookup is a column of a lookup table, using this option prevent the system to clear the field when inserting new rows or updating rows.
- NO DETAIL : If the block is in detailed, using this option on a field make it invisible in the detail.
- NO CHART : If the block is multiple, using this option on a field exclude it from the chart.
- QUERY UPPER : Whatever the string you input, this option will make kopi transform it to capital letters.
- QUERY LOWER : the opposite of the previous option it transform strings to lower case.

Example

```
VISIT name
  LABEL
  STRING(10)
  QUERY LOWER
  NOECHO
END FIELD
```

2.2.1.10. Field Columns

The COLUMNS field option is used to establish a connection between a certain column in the database with the field.Once such connection established, the field will have a direct access to the database column allowing insertions, modification ...+ You may enter this option in order to specify which table and which column the field refers.
The same field can refer to more than one column. You can also use the KEY option to specify a key column in the database or the NULLABLE option to specify an outer joint.+ Two more option are available with the COLUMNS command, the index and the priority options.

Syntax:

```
FieldColumns : "COLUMNS" (ColumnList) [ColumnIndex] [ColumnPriority]
ColumnList   : Column  [, Column ]
Column       : ["KEY"] ["NULLABLE"] QualifiedName
```

Example

```
VISIT Invoice AT <1, 2>
  LABEL "Invoice Number"
  STRING(10)
  COLUMNS(I.NumInvoice) //I is the invoices table for example
  ....
END FIELD
```

that a field can be connected to more than one table and column. In this case, the formula you will type will be:

Example

```
VISIT Invoice AT <1, 2>
  LABEL "Invoice Number"
  STRING(10)
  COLUMNS(KEY I.NumInvoice, ID.Invoice) //
  ....
END FIELD
```

In this example, I is the invoices table and IP is the invoice details table and they have an outer join connection throw columns NumInvoice and Invoice.

- Indexes

This option is used to define a value in the database which is to remain unique so that it can not appear anymore in another field of the same column. An INDEX is always to be followed by one or more integers. This integer can take a value between 0 and 31,both minimal and maximal values are also included.

Let's sum up with the following syntax:

Syntax:

```
ColumnIndex : "INDEX" Integers
Integers    : Integer index  [Integers]
```

If two or more fields are given the same index value, it means that two similar combinations of these field values will not be accepted.

For example, two different lessons cannot be given in the same room. In this case, the three fields, namely the fields "professor", "time" and "lesson" are to be attributed the same index. Thus, at least one of the three values needs to be changed so that the combination can be accepted by the machine.

Example

```
MUSTFILL "Lesson"
  LABEL
  HELP "The lesson you have to attend to"
```

```
  LONG (11)
  COLUMN (LEC.Lesson)
  INDEX 0
END FIELD


VISIT "Lecturer"
  LABEL
  LONG(11)
  COLUMN (LES.Lecturer)
  INDEX 0
END FIELD


MUSTFILL "Time"
  STRING(11)
  COLUMN (LES.Time)
  INDEX 0
END FIELD
```

However, this example would implicate a professor can give two different lessons at the same time. In order to avoid such errors, you can attribute one field two or more indexes. So you can associate the two fields "professor" and "time" together. Thus, you will have:

Example

```
MUSTFILL "Lesson"
  LABEL
  HELP "The lesson you have to attend to"
  LONG (11)
  COLUMN (LEC.Lesson)
INDEX 0
END FIELD

VISIT "Lecturer"
  LABEL
  LONG
  COLUMN (T.Lecturer)
  INDEX 0 1
END FIELD

MUSTFILL "Time"
  STRING
  COLUMN (LEC.Time)
  INDEX 1
END FIELD
```

In this case, notice that the "Lecturer" field has been associated with two indexes: 0 and 1.

The index value is ascendant. When attributing an index value to a field combination, you shall always take the value nexting that you have taken the last.

- **Priority**

Syntax:

```
ColumnPriority:   "PRIORITY" ["-"] Integer priority
```

This option is used in order to define the column order within a list when this list is displayed. A PRIORITY is always followed by an integer according to the structure given above. The column with the biggest priority value will appear on the extreme left side of the table and the one with the least value will be on the extreme right side.

We shall notice that negative values are also permitted in this option. However, the minus sign ("-") standing before the number does not have any influence over its value but simply indicates the way all the information will be sorted out within a column. Actually, the different fields are always sorted in the ascending way, i.e from A to Z in case of an alphabetical text and from 1 to x+1 for numbers. Now, if the integer is preceded by a minus, the column content will be sorted in the other way round.

Example

If columns "Surname", "Name" and "Date of Birth" are respectively given the priorities 3,4 and 1, "Name" will come first and will be followed by "Surname" and "Date of Birth". The same order applies with the values 3, -4 and 1, with the only difference that the names will be sorted out from Z to A.

Moreover, two columns with the same priority will be displayed according to the same order in which the user has listed them.

2.2.1.11. Field Commands

Once you have defined the columns , you may define the field commands. There are three types of field command:

- Standard Field Commands (no Modes)
- Commands using Modes
- Access modifiers using Modes

Syntax:

```
ModeAndCommands : ModeList  AccessModifier | [ModeList] Command

Command         : "COMMAND" QualifiedName | "COMMAND" CommandBody
                  "END" "COMMAND"

CommandBody     : "ITEM" SimpleName  CommandAction

CommandAction   : "ACTION" [(Parameter)] { KopiJavaCode  }
                  |
                  "EXTERN" QualifiedName | "CALL" SimpleName

ModeList        : "ON" Modes

Modes           : Mode  [, Modes ]

Mode            : "QUERY" | "INSERT" | "UPDATE"
```

2.2.1.12. Standard Field Command

The command QualifiedNames that we will can be kopi predefined commands or you can make new Actors and commands you can use in the field command definition. There are five ways in calling a standard command

```
"COMMAND" QualifiedName

Example :
COMMAND FormReset
```

Or:

```
"COMMAND" "ITEM" SimpleName "CALL" SimpleName "END" "COMMAND"

Example
COMMAND
  ITEM Autofill
  CALL function
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "EXTERN" QualifiedName "END" "COMMAND"

Example
COMMAND
  ITEM Autofill
  EXTERN filepath.function
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "ACTION" (VField SimpleName) {Kopi-Java Code}
"END" "COMMAND"

Example
COMMAND
  ITEM Clear
  ACTION (VField f) {
    f.setDefault ();
  }
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "ACTION" {Kopi-Java Code}
"END" "COMMAND"
```

2.2.1.13. Field Command using Modes

All the previous command definition ways can be preceded by one mode ore more. There are three sorts of mode which are to be sorted according to the following hierarchy:

- QUERY to start an inquiry within the database
- INSERT to create a new row in the database
- UPDATE to enter new information within a row

So you will have: ON Mode COMMAND, ON MODE QualifiedName ...

Example

```
ON QUERY, UPDATE COMMAND InsertMode
```

If you have entered a mode before the COMMAND, it means this COMMAND can only be invoked if the block is in the mode you have determined.

2.2.1.14. Field Access modifiers using Modes

This command type is used to change the access to the field using the block Mode. In fact the access is not changed directly but the higher access possible is set to the indicated access.

Example

```
ON QUERY,INSERT HIDDEN
```

In this example, we have determined that the field will be invisible if the block is in the QUERY or the INSERT mode.

2.2.1.15. Field Triggers

Triggers are events that you can use to execute actions when they occur, there are field triggers, block triggers and form triggers that you can use following this syntax :

Syntax

```
Trigger      : EventList TrigerAction

EventList    : Event [,EventList]

TrigerAction : SimpleName | "EXTERN" QualifiedName | {KOPI_JAVA code}
               | (VField SimpleNAme){KOPI_JAVA code}
```

Field Triggers are events that concern the fields behavior, here is a list of all kopi field triggers available :

- PREFLD : is executed upon entry of field
- POSTFLD : is executed upon exit of field
- POSTCHG : is executed on field content change
- PREVAL : is executed before validating any new entry
- VALFLD : is executed after field change and validation
- VALIDATE : this is the same trigger as VALFLD
- DEFAULT : Defines the default value of the field to be set if the setDefault() method is called (this method is automatically called when the user choose the insert command)
- FORMAT : Not defined actually
- ACCESS : ACCESS is a special trigger that defines how a field can be accessed. This trigger must return one of these values ACS_SKIPPED, ACS_HIDDEN, ACS_VISIT or ACS_MUSTFILL.
- VALUE : equates the value of two fields
- AUTOLEAVE : must return a boolean value, if "true" the cursor will move to the next field
- PREINS : is executed before inserting a row of the database
- PREUPD : is executed before updating a row of the database

- PREDEL : is executed before deleting a row of the database
- POSTINS : is executed after inserting a row of the database
- POSTUPD : is is executed after updating a row of the database

Examples

```
VISIT EMail
  LABEL "Email"
  HELP "The electronic address of the lecturer"
  STRING 40
  COLUMNS (B.Mail)
  DEFAULT {
    @N.EMail = "@";
  }
END FIELD
```

the "@N.EMail" expression means : bloc N, field EMail

```
MUSTFILL Lecturer
  LABEL
  FIXED (5,2)
  MINVAL 0
  COLUMNS (T.Lecturer)
  PRIORITY 1
  ACCESS {
    return (T.Art.getInt()==0? ACS_HIDDEN; ACS_MUSTFILL);
  }
END FIELD
```

2.2.1.16. Field Definition from another Field

You can also create a field without declaring any field alignment, field option, mode, command or field trigger. However, the column, index and priority definitions are permitted.

Example

```
HIDDEN Firm
  LONG (11)
  IS F.ID
END FIELD
```

IS is used to equate a field with another one. In this example, the field "Firm" is equated with the field "F.ID".That means that the Firm field will always have the same value as the F.ID field.

2.2.2. Creating Form Blocks

As you already know, a form is composed of blocks. A block is a set of data which are stocked in the database and shown on a form. A block is created in order to either view the content of a database, to insert new data in the database or to update existing data in the database. A Block will always begin with BLOCK and end with BLOCK END, when defining a block, you have to proceed according to the following structure.

Syntax

```
BlockDefinition : "BLOCK" (Integer buffer, Integer rows)
```

```
                  SimpleName[:SimpleName] [String blockTitle]
                  ["IS" QualifiedName] ["IMPLEMENTS"  [,QualifiedName]*]
                  BlockBody
                  "END" "BLOCK"

BlockBody       : [blockBoder] [blockAlignement] [blockHelp]
                  [blockOptions] [blockTables]
                  [blocIndices] [blockCommands] [blockTriggers] blockFields
                  [blockContextFooter]
```

2.2.2.1. Block Types

There are actually two types of blocks in Visual Kopi, the only difference between them in the definition syntax is the buffer Integer.

- single blocks

A single block is a block in which only one single row of a database table is displayed on the form. Each time, the computer will memorize only one entire row and a certain quantity of ID numbers through which it will retrieve another rows from the database if you want to view them.

Example

```
BLOCK (1,5)
  ...
END BLOCK
```

The first integer indicates the block type. In case of a single block, the first integer will always be 1. The second integer indicates the maximal number of the automatically memorized IDs.

- Multiple Blocks

A multiple block is a block in which more than one row are displayed on the form. These rows are retrieved all at once from the database and are memorized by the computer. Actually, you can define the number of the displayed rows which can be less than this of the memorized rows. In this case, there will be no need anymore to retrieve the hidden rows from the database when you want to view them.

Example

```
BLOCK (10,5)
  ...
END BLOCK
```

The first integer indicates the number of rows to be automatically memorized while the second defines the number of displayed rows. Notice the first integer value must always be greater than 1.

2.2.2.2. Block Names

The name of the block is composed SimpleName name optionally followed by a colon and a SimpleName shortcut, finally, you can specify a String for the Block title but it's not compulsory.

Example

```
BLOCK (10,5) Degree:D "Degree Block"
```

2.2.2.3. Block superClass and Interface

A Block may extend a superClass and implements one or more Interface.

Example

```
BLOCK (10,5) Degree:D "Degrees" IS VImportedBlock IMPLEMENTS VConstants
```

2.2.2.4. Block Border

After the Block name and implementation clause, you can insert the optional border statement that defines the Block's frame type. Besides, the Title of the block will appear only if the Block's Border type has been specified. There are actualy four Border options :

- BORDER LINE to frame the block with lines.
- BORDER RAISED to enhance a block by setting it on the foreground.
- BORDER LOWERED to put it at the background.
- BORDER ETCHED to carve a frame in the form.

Example

```
BLOCK (10,5) Degree:D "Degree Block" IS VImportedBlock
BORDER RAISED
```

2.2.2.5. Block Alignment

Alignment statements are useful to align a block(source block) referring to another one(target block), after the keyword ALIGN, you have to specify the referred block name followed by one or many sets of two integers separated by a minus, the sets are separated by a comma. As for the integers signification, the one in the left of the minus is the source block column number while the other one is for the target block column number. For Example, let us suppose we have a multiple block Prices with 4 columns, with column 3 filled with Quantities and Column 4 with Prices, we also have a single block TotalPrices with two fields totalQuantity and TotalPrices, we want these fields to be aligned with the correct columns of the Prices block:

Example

```
BLOCK (10,5) Prices : P "Prices Block"
 BORDER row
...
END BLOCK

BLOCK (1,1) TotalPrices : TP "TotalPrice"
  BORDER row
  ALIGN Prices <1-3, 2-4>
...
END BLOCK
```

2.2.2.6. Block Help

This optional command is used to define the help text for each field of the block. The different texts are to be entered after the word HELP.

Example

```
BLOCK (10,5) Degree: D "Degree Block"
  BORDER row
  HELP "The degree the student will target at the end of a year"
```

```
  ...
END BLOCK
```

2.2.2.7. Block Options

In this optional section you can insert commands with restriction functions towards the users, the available commands that you can use are :

- NO CHART : Disables the chart(grid) rendering of a multiple bloc to make it look like a single block.Only possible on multiple blocks displaying only one row, Besides the fields must be positioned with the AT command.
- NO DETAIL : Disables the positioning of fields and displays the block as a chart (grid), Only possible on multiple blocks, the fields should not be positioned with the AT command.
- NO DELETE : Prevent the user from removing fields content.
- NO INSERT : Prevent the user from inserting data in fields.
- NO MOVE : Prevent the user from moving between records.
- ACCESS ON SKIPPED : Makes the block accessible even if or its fields have SKIPPED access.
- UPDATE INDEX : If used, saving a block would delete all its rows and reinsert them one by one, by doing so, you can update the table rows even when you change the index fields without worrying about the "row already exist exception".

2.2.2.8. Block Tables

When making use of this section, you have to type the command TABLE and enter the table name followed by a comma and by the shortcut to this table name. This shortcut will further be used as a shorthand in place of the complete table name in order to access to the table. These names refer to certain tables in the database whereby the first table is the one on which the user will work. The remaining tables are the so-called "look-up tables", i.e tables that are associated with the first one.

Syntax:

```
BlockTables:   "TABLE"  "<"SimpleName  , SimpleName > [BlockTables]*
```

Example

```
TABLE <Lessons,L>
TABLE <Lecturers, P>
TABLE <Rooms, R>
```

The user will make use of these look-up tables as references when bringing in changes in the principal table.

2.2.2.9. Block Indexes

If you plan to enter one index or more when defining your fields, you also have to define one index text or more which will appear if you make a mistake by entering an indexed data or a data combination twice. This command can be followed by an error message contained in a string.

Syntax:

```
BlockIndices:   "INDEX" String message  [BlockIndices]*
```

Example

```
BLOCK (10,5) Degree : D "Degree"
 BORDER row
 INDEX "This data already exists"
 ...
END BLOCK
```

2.2.2.10. Block Commands

Block commands are commands accessible only from the block where they are called. There are Three ways to call block commands:

- Calling A Standard Command

There are 5 possible structures:

```
"COMMAND" QualifiedName

Example
COMMAND ResetBlock
```

Or:

```
"COMMAND" "ITEM" SimpleName "CALL" SimpleName "END" "COMMAND"

Example
COMMAND
  ITEM Save
  CALL Save
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "EXTERN" QualifiedName "END" "COMMAND"

Example
COMMAND
  ITEM SharesOfExpenses
  EXTERN Costing
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "ACTION" (VBlock SimpleName) {Kopi-Java Code}
"END" "COMMAND"

Example
COMMAND
  ITEM End
  ACTION (VBlock b) {
    b.validate ();
    close (CDE_VALIDATE)
   }
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "ACTION" {Kopi-Java Code} "END" "COMMAND"

Example
COMMAND
  ITEM Validate
  ACTION {
    validate ();
  }
END COMMAND
```

- Calling The Block Commands With Modes

Blocks have 3 possible Modes:

- QUERY : When querying the database
- INSERT : When inserting a new row in the database
- UPDATE : When updating rows in the database

You can combine these modes with the previous block command structure to have more control over your command

Syntax

```
ON BlockMode SimpleName
```

This means that the command called is only accessible when the block is in the specified Mode.

Example

```
ON QUERY, UPDATE COMMAND InsertMode
```

- Calling The Block Commands With Modes And An Access Modifier

in Kopi, a field can have different access types or modifiers, here is the definition of the four available access modifiers listed by ascending level.

- HIDDEN : HIDDEN field are invisible in the form, they are used to store hidden operations and database joins.
- SKIPPED : SKIPPED fields are read only fields, you can read the value but you can't modify it.
- VISIT : fields with this access type are accessible, can be modified but not necessary.
- MUSTFILL : MUSTFILL fields are accessible fields that the user must fill with a value.

In the block command section, you can set the highest access level for the block fields according to the mode in wich the block would beording to the mode in which the block would be.

Example

```
ON QUERY, UPDATE SKIPPED
ON UPDATE HIDDEN
ON QUERY VISIT
```

In the first example, all fields in the block will be either SKIPPED or HIDDEN in the query and update modes and can neither be MUSTFILL nor VISIT. For the second example, all the fields in the block

will be HIDDEN when the block is in update mode. In the last example, all the fields in the block will be either VISIT, SKIPPED or HIDDEN in the query mode and can not be MUSTFILL.

2.2.2.11. Block Triggers

The block triggers are the same as form triggers on the block level. There are actually 20 block triggers you can use to execute actions once they are fired.

Syntax

```
BlocTrigger: BlocEventList TriggerAction

BlocEventList: BlockEvent [,BlockEvent]*
```

Concerning the trigger action, which is the action to execute when the trigger is activated they can by :

- a local function : SimpleName
- an external function : EXTERN QualifiedName
- {KOPI_JAVA code}
- (VBlock SimpleName){KOPI_JAVA code}

Here is a list of all available block triggers or block events in Kopi.

- PREQRY : executed before querying the database
- POSTQRY : executed after querying the database
- PREDEL : executed before a row is deleted
- POSTDEL : executed after a row is deleted
- PREINS : executed before a row is inserted
- POSTINS : executed after a row is inserted
- PREUPD : executed before a row is updated
- POSTUPD : executed after a row is updated
- PRESAVE : executed before saving a row
- PREREC : executed upon record entry
- POSTREC : executed upon record exit
- PREBLK : executed upon block entry
- POSTBLK : executed upon block exit
- VALBLK : executed upon block validation
- VALREC : executed upon record validation
- DEFAULT : is executed when the block is in the InsertMode. This trigger becomes active when the user presses the key F4. It will then enable the system to load standard values which will be proposed to the user if he wishes to enter new data.

- INIT : executed upon block initialization
- RESET : executed upon Reset command (ResetForm)
- CHANGED : a special trigger that returns a boolean value of wether the block have been changed or not, you can use it to bypass the system control for changes this way :

```
CHANGED {
  return false;
}
```

- ACCESS : defines whether a block can or not be accessed, it must always return a boolean value

```
 ACCESS{
   return Block.getMode == MOD_QUERY
   // Tests if the block is in query mode,
   this block is only accessible on query mode
 }
```

Examples

```
BLOCK (10,5) Degree : D "Degree"
  BORDER row
  INDEX "This data already exists"
  COMMAND ResetForm

  INIT,PREBLK{
   KOPI_JAVA code
  }
  PREINS{
   System.out.print("Inserting date");
   ...
  }
  ...
END BLOCK
```

2.2.2.12. Block Fields Declaration

In this section, all you have to do is to write at least on block field definition that begins with an access modifier and ends with END FIELD, you can enter as much fields as you may need following the field definition we saw in the previous chapter.

2.2.2.13. Block Context Footer

In this section of the block, you can write local functions, define inner classes, override command actions, define your data members. The elements you write in the block Context Footer are accessible in the concerned block and are written in KOPI_JAVA code.

2.2.3. Creating Forms

There are actually two types of forms in kopi, normal forms and BLOC INSERT forms which are special forms containing blocks that you may insert in other forms.
A form begins always with FORM and ends with END FORM, while a BLOC INSERT form begins with BLOC INSERT and ends with END INSERT. When creating a form, you will have to include the block and the field definitions. Moreover, you will have to define the menus as well as the different commands. Finally, you can also define some form triggers. Concretely, the structure is the following:

Syntax:

```
FormDef : ["LOCALE" String formLocalization]
          "FORM" String formTitle
          ["IS" QualifiedName "IMPLEMENTS" QualifiedName[,QualifiedName]*]
          [FormHeader]  [MenuDefinition]
          [ActorDefinition] [TypeDefinition]
          [CommandDefinition] [InsertDefinition]
          "BEGIN"
          [FormCommands] [FormsTriggers] [BlocksDefinition]
          [ContextFooter]
          "END" "FORM"
```

2.2.3.1. Form Localization

This is an optional step in which you may define the language of your forms menus and messages, the latter have to be defined in xml files.

Syntax

```
"LOCALE" String formLocalization
```

Example:

```
LOCALE "en_EN"
```

2.2.3.2. Form Title

Every form have to begin with the keyword FORM that you can follow with a title (optional).

Syntax

```
"FORM" String formTitle
```

Example

```
FORM  "Lecturers List"
  ...
END FORM
```

2.2.3.3. Form Superclass And Interfaces

- Superclass:

Syntax:

```
SuperForm:   "IS" QualifiedName
```

Every form is a class that may extend another Java class by using the optional keyword IS. Otherwise, Kopi will automatically take over the java.lang.Object QualifiedName.

Example

```
FORM "Service Register" IS VReportSelectionForm
```

In other words, the class declaration you have just made specifies the direct superclass of the current class.

- Interfaces

You can also specify interfaces which the form may implement by using the IMPLEMENTS option. When used in a form declaration, this option then lists the names of interfaces that are direct superInterfaces of the class being declared. Thus, you will have the following structure:

Syntax

```
Interface :  "IMPLEMENTS" QualifiedName [,QualifiedName]*
```

Example:

```
FORM "Record registering" IS VPrintSelectionForm
  IMPLEMENTS VConstants
  ...
END FORM
```

2.2.3.4. Form Header

Syntax:

```
ContextHeader:   { PackageDeclaration  ImportDeclarations  }

PackageDeclaration:   <As defined in JLS 7.4>
ImportDeclarations:   <As defined in JLS 7.5>
```

The package definition is contained in the header. It consists in defining where this form belongs to i.e which application, which firm or which person it is related to.

Example

```
FORM "List of the Lecturers"
  {
  package at.dms.apps.timetable;
  }
  ...
END FORM
```

In this example, the form belongs to the DMS firm and is a part of the application called "timetable". In addition to this Java package declaration, you can make a Java import declaration in order to import some classes from other packages. You can add an unlimited number of imports.

Example

```
FORM "List of the Lecturers"
  {
    package at.dms.apps.tb;
    import java.util.Vector
  }
   ...
END FORM
```

2.2.3.5. Menus Definition

Defining a menu means adding an entry to the menu bar in the top of the form, you can add actors to this menu later by specifying the menu name in the actor definition. In the menu definition, the LABEL is optional.

Syntax:

```
MenuDefinition:   "MENU" SimpleName
                  ["LABEL" String label]
                  "END" "MENU"
```

Example

```
FORM "List of the Lecturers"

  {
    package at.dms.apps.tb;
  }

  MENU File
    LABEL "file"
  END MENU
...
END FORM
```

2.2.3.6. Actors Definition

An Actor is an item to be linked with a command, if its ICON is specified, it will appear in the icon_toolbar located under the menu bar, otherwise, it will only be accecible from the menu bar. ICON,LABEL and KEY are optional, the KEY being the keyboard shortcut to assign to the actor.

Syntax:

```
ActorDefinition:  "ACTOR" SimpleName
                  "MENU"  SimpleName
                  ["LABEL" String label]
                  "HELP"  String helpText
                  ["KEY"   String key]
                  ["ICON"  String icon]
                  "END" "ACTOR"
```

Example

```
FORM "List of the Lecturers"

  {
    package at.dms.apps.tb;
  }

  MENU File
    LABEL "file"
  END MENU


  ACTOR cut
    MENU  File
    LABEL "cut"
    HELP  "cut element"
    ICON        "CutIcon"
  END ACTOR
```

```
...
END FORM
```

2.2.3.7. Types Definition

After having defined your menus and actor, you can enter different field types definitions based on the standard field types or code field types, you can also use the LIST and SELECT commands to customize these new types.

Syntax:

```
TypeDefinition:  "TYPE" SimplName "IS" FieldType
                 [TypeList]
                 "END" "TYPE"
```

Example

```
FORM "List of the Lecturers"
  {
    package at.dms.apps.tb;
  }
  TYPE Name IS
    STRING (20,10,4)
    CONVERT UPPER
  END TYPE

  TYPE Lesson IS
    LONG (11)
    MINVAL 0
    MAXVAL 100
  END TYPE

  TYPE Answer IS
    CODE BOOL IS
        "Yes"   = TRUE
        "No"  = FALSE
    END CODE
  END TYPE

  TYPE Months IS
    ENUM("January", "February", "March", "April")
  END TYPE

TYPE Lecturer IS
  STRING (8)
  CONVERT UPPER

  LIST "Lecturer" IS
    "Symbol" = Symbol : STRING (8)
    "Name" = Name : STRING (40)
    "Surname" = Surname : STRING (40)
    "Lesson" = Lesson : STRING (20)
  END LIST
...
END TYPE
```

```
  ...
END FORM
```

2.2.3.8. Commands Definition

In this section you may want to define new commands, to do so, all you need is an already defined Actor from which you will call the command in order to execute an Action on the form. There are 3 ways to define this Action: every command have a effective ray of action (VField, VBlock, VForm)

- Calling a function with the CALL command
- Calling a function outside of the form using the EXTERN command
- Simply writing the body of the action using the ACTION command, the parameters are optional and can be VField, VBlock, VForm.

Command Defined in this section can be form level commands, block level commands or field level commands, this will depend on the action called by the command and where the command is actualy called.

Syntax

```
cmdDef  : "COMMAND" QualifiedName
          "ITEM" SimpleName commandBody
          "END" "COMMAND"

cmddBody: "CALL" SimpleName | "EXTERN" QualifiedName
     |"ACTION" [(VField SimpleName | VBlock SimpleName | VForm SimpleName)]
           {KOPI_JAVA statements}
```

Example

Calling a local action :

```
COMMAND Preview
  ITEM  Preview
  CALL  previewForm
END COMMAND
```

Example Calling an external action :

```
COMMAND SaveBlock
  ITEM Save
  EXTERN com.company.vlib.form.Commands.SaveBlock
END COMMAND
```

Example Writing the action's body :

```
COMMAND PrintBlock
  ITEM  PrintBlock
  ACTION (VBlock b){
    b.validate();
    b.getForm().close (VForm.CDE-Validate);
  }
END COMMAND
```

2.2.3.9. Insert Definition

This command allows you to actually define your types, menus, actors, commands in another file, and then insert it in any form you want, thus avoiding rewriting recurrent definitions. You still can add definition before or after using the INSERT command.

Syntax

```
Insert Definition : "INSERT" String filePath
```

Example

```
FORM "List of Students"

  {
  package at.dms.app.application
  }

  INSERT "Global.vf"

  TYPE Lesson IS
    LONG(11)
  END TYPE

...
END FORM
```

After the different definitions we have made (types, menus, actors,commands), we need to begin the declaration of our form.Here, we will set up the structure and the behaviour of the form throw 4 sections :

- Form Options
- Form Commands
- Form Triggers
- Blocks

2.2.3.10. Form Commands Declaration

In this section you may call the commands you want your form to have.You can call commands you already defined in the command definition section :

Example

```
BEGIN

  COMMAND Preview
  COMMAND PrintBlock
```

You can also make use of Kopi's ready command such as :

```
BEGIN

  COMMAND HelpForm
  COMMAND QuitForm
```

Moreover, you can define your command in this section the same way you did in the command definition section, but with no command name, it will be both a definition and a declaration.

There still one more think to know about form command declaration, in fact, you can also control the accessibility to a command by the Mode of the form.+ A form have 3 possible Modes :

- QUERY : When inquiring the database
- INSERT : When inserting a row in the database
- UPDATE : When updating a row in the database

The Syntax to use command with modes is the following :

Syntax

```
"ON" ModeName [,ModeName]* "COMMAND" SimpleName
```

2.2.3.11. Form Triggers Definition

Form Triggers are special events that once switched on you can execute a set of actions defined by the following syntax :

Syntax

```
FormTrigger  :    FormEventList TriggerAction
FormEventList:  FormEvent [,FormEvent]*
```

Kopi actually defines 6 Form Triggers or Form Events :

- INIT : executed when initializing the form and before the PREFORM Trigger, also executed at ResetForm command
- PREFORM : executed before the form is displayed and after the INIT Trigger, not executed at ResetForm command
- POSTFORM : executed when closing the form
- QUITFORM : actually not available
- RESET : executed upon ResetForm command
- CHANGED : a special trigger that returns a boolean value of whether the form have been changed or not, you can use it to bypass the system control for changes this way :

```
 CHANGED {
   return false;
 }
```

Examples

```
INIT{
  System.out.println("Trigger INIT activated");
  // KOPI_JAVA statements
}

INIT,PREFORM{
  //KOPI_JAVA statements
}
```

2.2.3.12. Form Pages

You can create Pages in your form using the NEW PAGE command after the trigger declaration section,this is optional and will create a Tab for each page you create under the form's toolbar. You can put as much blocks you want in each page, the same goes for form without pages.

Example

```
NEW PAGE "Page1"
```

2.2.3.13. Form Blocks

You can have several blocks in one form, you just have to stack them in your form, using the definition presented in the Second chapter of this document, after the form triggers definition, you can also group them in pages.

You can also use the block import command in order to import block units (predefined blocks) in your form :

Syntax

```
BlockImport: "INSERT" QualifiedName
```

As for the Block Unit definition, you can create one by creating a new form file using with this Syntax:

Syntax

```
BlockUnit: "BLOCK" "INSERT" ContextHeader
          [MenuDefinition]
          [ActorDefinition] [TypeDefinition]
          [CommandDefinition] [InsertDefinition] BlockDeclaration
          "END" "INSERT"
```

You can see that a Block Unit has the same structure of a normal Form, with the absence of the form declaration Section, replaced by a Block Declaration Section. Moreover, a Block Unit does not have a Context Footer.

Example

```
BLOCK INSERT
{
  package at.dms.apps.tb;
}

//Here you can add Types, commands, actors definitions

  INSERT "Global.vf"

  BLOCK (10, 5) Lecturer.L "Lecturers"
    BORDER LINE
    ALIGN Name<1-3>
    TABLE <Lecturers, L>
    TABLE <Lessons, C>

      VISIT lecturers
        LABEL
        STRING(10)
```

```
        COLUMNS(L.lecturers)
      END FIELD


      VISIT lessons
        LABEL
        STRING(10)
        COLUMNS(C.lessons)
      END FIELD

  END BLOCK
...

END INSERT
```

2.2.4. Form Context Footer

In this section of the form, you can write your local functions, define inner classes, override command actions, define your data members. The elements you write in the forms Context Footer are accessible in all the form and are writen in KOPI_JAVA code.

2.3. Visual Kopi Reports

Visual Kopi allows you to create dynamic reports. These are files with the ".vr" extension that have a very simple structure, in fact, all you have to do to create such reports is to list the definitions of all the fields you need, then you will have to write a constructor that will load data into these fields. The so created reports will have dynamic functions such as sorting and group making, you will also be able to print it or export to different file formats.You can even add other fields to the report after it's generated, do more calculus, customize the report columns... The Visual Kopi Reports are generated from Visual Kopi form files with the ".vf" extension by different methods that we will see in this chapter, along with the process of making a dynamic report.

2.3.1. Creating Report Fields

As we said in the introduction, the report structure is based on report fields, these fields begin with FIELD and a SimpleName and ends with END FIELD and have the following syntax :

Syntax

```
ReportFieldDef: "FIELD" [Integer MultiField] SimpleName [FieldLabel]
                [HelpText] FieldType [FieldAlign] [FieldOptions]
                [GroupCommand] [FieldCommands] [FieldTriggers]
                "END" "FIELD"
GroupCommands : "GROUP" SimpleName
```

2.3.1.1. Single Field and Multiple Fields

When you define a news report field, you have the choice to make it multiple by adding an integer (i) just after the FIELD command, doing so will create i fields. The generated fields will have the SimpleName indicated after the FIELD command plus a number from 1 to i.

Example

```
FIELD(3) Name
  ...
```

```
END FIELD
```

In this example we have created 3 fields (Name1, Name2 and Name3).

2.3.1.2. Report Field Label

The field label is the name that will be shown in the dynamic report, you can define a label by entering the LABEL command followed by String separated with a comma. If you omit the Strings after the LABEL command, the field will have its SimpleName as label. You can also specify NO LABEL if you want your field to have no label at all. If you specify more than one String after the LABEL command, then if your field is Single it will have the first String as label, otherwise, if it's a multiple field each generated field will have a label from the Strings by order.

Example

```
FIELD Name
  LABEL "Person Name"
  ...
END FIELD


FIELD(3) LastName
  LABEL "Name1", "Name2", "Name3"
  ...
END FIELD


FIELD Age
  LABEL
  ...
END FIELD


FIELD ID
  NO LABEL
  ...
END FIELD
```

2.3.1.3. Report Field Help Text

HELP is used to insert an explanation text for the application user . This Help text will then be displayed when the user places the mouse on the report field.

Example

```
FIELD Name
  LABEL
  HELP *"The Person's First Name"
  ...
END FIELD
```

2.3.1.4. Report Field Type

The field type is a required entry, you may enter a predefined field type :

Example

```
FIELD Name
  LABEL
  STRING(10)
END FIELD


FIELD Age
  LONG(3)
END FIELD
```

You can also use self defined field types that you have previously defined in the type definition section of your report.

Example

```
FIELD objectList
  LABEL
  TYPE Objects
END FIELD
```

In this example, Objects is a type you should have defined previously in type definition section of the report using standard types, CODE types, ENUM types, LIST and SELECT command ...

2.3.1.5. Report Field Alignment

This command is used to define the localization of the field's content inside the field. There are three types of alignment.

- ALIGN RIGHT the value is displayed at the right inside the field
- ALIGN LEFT the value is displayed at the left inside the field
- ALIGN CENTER the value is centered in the field

Example

```
FIELD Name
  LABEL
  STRING(10)
  ALIGN CENTER
END FIELD
```

2.3.1.6. Report Field Options

There is actually only one option for the dynamic report fields in Kopi: the HIDDEN OPTION , if this option is used on a field, he will not be visible the the report. This options have to be used on the last visible field of the report to avoid rendering bugs.

Example

```
FIELD Name
  LABEL
  STRING(10)
  HIDDEN
END FIELD
```

The field Name will not be visible on the report.

2.3.1.7. Report Field Group

You can create clickable groups in your report by using the keyword GROUP in you field followed by the field you want to be grouped by the actual field.

Example

```
FIELD Customers
  LABEL
  STRING(5)
END FIELD

FIELD Articles
  LABEL
  STRING(10)
END FIELD

FIELD InvoiceNum
  LABEL
  STRING(10)
  GROUP Articles
  GROUP Customers
END FIELD
```

In this report, you can click on the InvoiceNum field to group customers and articles.

2.3.1.8. Report Field Command

In report Fields, you can call commands with one of the following syntaxes :

```
"COMMAND" QualifiedName

Example
COMMAND ExportCsv
```

Or:

```
"COMMAND" "ITEM" SimpleName "CALL" SimpleName "END" "COMMAND"

Example
COMMAND
  ITEM Export
  CALL function
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "EXTERN" QualifiedName "END" "COMMAND"

Example
COMMAND
  ITEM Export
  EXTERN function
END COMMAND
```

Or:

```
"COMMAND" "ITEM" SimpleName "ACTION"
(VReportColulmn SimpleName) {Kopi-Java Code}
"END" "COMMAND"
```

Or:

```
"COMMAND" "ITEM" SimpleName "ACTION" {Kopi-Java Code}
"END" "COMMAND"
```

2.3.1.9. Report Field Triggers

Report field triggers are special events that you can catch to execute other actions.

Syntax

```
Trigger      : EventList TrigerAction

EventList    : Event [,EventList]*

TrigerAction : SimpleName | "EXTERN" QualifiedName | {KOPI_JAVA code} |
               (VReportColulmn SimpleNAme){KOPI_JAVA code}
```

Here is the two triggers available for report fields :

- FORMAT : Actually not available
- COMPUTE : executed when the report is displayed and can be used to compute expressions on the report columns and show the result.

Example

```
FIELD Price
  LABEL
  FIXNUM(9, 4)
  COMPUTE  EXTERN ReportTriggers.sumFixed
END FIELD

FIELD
  LABEL
  STRING(10)
  FORMAT
END FIELD
```

2.3.2. Creating Reports

Visual Kopi Dynamic reports have a unique structure that begin with REPORT and ends with END REPORT as described by the following syntax :

Syntax

```
ReportDefinition  :[ReportLocalization] "REPORT" ReportTitle
                   [IS QualifiedName]
                   ["IMPLEMENTS"  [,QualifiedName]*]
                   [ContextHeader] [ReportHelp] [ReportDefinitions]
                   "BEGIN" [ReportCommands] [ReportTriggers] (ReportFields)
                   [ContextFooter]
                   "END" "FIELD"
```

```
ReportTitle       :  String Title

ReportDefinitions : [MenuDefinition] [ActorDefinition] [TypeDefinition]
                    [CommandDefinition]
                    [InsertDefinition]
```

2.3.2.1. Report Localization

This is an optional step in which you may define the language of your forms menus and messages, the latter have to be defined in xml files.

Example:

```
LOCALE "en_EN"
```

2.3.2.2. Report Title

Every Report have to begin with the keyword REPORT that you have to follow with a title.

Example

```
REPORT "Invoices"
  ...
END REPORT
```

2.3.2.3. Report Superclass And Interfaces

- Superclass:

Syntax:

```
SuperClass:   "IS" QualifiedName
```

Every Report is a class that may extend another Java class by using the optional keyword IS. Otherwise, Kopi will automatically take over the java.lang.Object QualifiedName.

Example:

```
REPORT  "Orders Report" IS VReport
```

In other words, the class declaration you have just made specifies the direct superclass of the current class.

- Interfaces:

You can also specify interfaces which the report may implement by using the IMPLEMENTS option.

Syntax

```
"IMPLEMENTS" QualifiedName [,QualifiedName]*
```

Example:

```
REPORT "Orders Report" IS VReport
  IMPLEMENTS UReport
  ...
END REPORT
```

2.3.2.4. Report Header

Syntax:

```
ContextHeader:   { PackageDeclaration  ImportDeclarations  }

PackageDeclaration:   <As defined in JLS 7.4>
ImportDeclarations:   <As defined in JLS 7.5>
```

The package definition is contained in the header. It consists in defining where this report belongs to i.e which application, which firm or which person it is related to.

Example

```
REPORT "List of the Lecturers"
  {
  package at.dms.apps.timetable;
  }
  ...
END REPORT
```

In this example, the report belongs to the DMS firm and is a part of the application called "timetable". In addition to this Java package declaration, you can make a Java import declaration in order to import some classes from other packages. You can add an unlimited number of imports.

Example

```
REPORT "List of the Lecturers"
  {
    package at.dms.apps.tb;

    import java.util.Hashtable
    import java.util.Vector
  }
   ...
END REPORT
```

2.3.2.5. Report Help Text

After the report header, you can enter a help text for the report using the following syntax:

Syntax

```
HELP String helpText
```

Actually every report has a help menu that tries to describe the structure of the report by giving information about its commands and fields in a document, the help text will be on the top of this help menu document.

Example

```
REPORT "Orders Report"

  {
    package at.dms.apps.tb;

    import java.util.Hashtable
```

```
    import java.util.Vector
  }

HELP "This report lists purchase orders"
...

END REPORT
```

2.3.2.6. Report Menus Definition

Defining a menu means adding an entry to the menu bar in the top of the report, you can add actors to this menu later by specifying the menu name in the actor definition. In the menu definition, the LABEL is optional.

Syntax:

```
MenuDefinition:   "MENU" SimpleName ["LABEL" String label]   "END" "MENU"
```

Example

```
REPORT "List of the Lecturers"

  {
    package at.dms.apps.tb;
  }

  MENU newMenu
    LABEL "newMenu"
  END MENU
...
END REPORT
```

2.3.2.7. Report Actors Definition

An Actor is an item to be linked with a command, if its ICON is specified, it will appear in the icon_toolbar located under the menu bar, otherwise, it will only be accessible from the menu bar. ICON,LABEL and KEY are optional, the KEY being the keyboard shortcut to assign to the actor.

Syntax:

```
ActorDefinition:  "ACTOR" SimpleName
                  "MENU"  SimpleName
                  ["LABEL" String label]
                  "HELP"  String helpText
                  ["KEY"   String key]
                  ["ICON"  String icon]
                  "END" "ACTOR"
```

Example

```
FORM "List of the Lecturers"

  {
    package at.dms.apps.tb;
  }
```

```
  MENU newMenu
    LABEL "newMenu"
  END MENU


  ACTOR printReport
    MENU  newMenu
    LABEL "Print"
    HELP  "Print the report"
    ICON  "printerIcon"
  END ACTOR
...
END REPORT
```

2.3.2.8. Report Types Definition

After having defined your menus and actor, you can enter different field types definitions based on the standard field types or code field types, you can also use the LIST and SELECT commands to customize these new types.

Syntax:

```
TypeDefinition:  "TYPE" SimplName "IS" FieldType  [TypeList] "END" "TYPE"
```

Example

```
REPORT "List of the Lecturers"
  {
    package at.dms.apps.tb;
  }
  TYPE Name IS
    STRING (20,10,4)
    CONVERT UPPER
  END TYPE

  TYPE Lesson IS
    LONG (11)
    MINVAL 0
    MAXVAL 100
  END TYPE

  TYPE Answer IS
    CODE BOOL IS
        "Yes"   = TRUE
        "No"  = FALSE
    END CODE
  END TYPE

  TYPE Months IS
    ENUM("January", "February", "March", "April")
  END TYPE

TYPE Lecturer IS
  STRING (8)
  CONVERT UPPER
```

```
  LIST "Lecturer" IS
    "Symbol" = Symbol : STRING (8)
    "Name" = Name : STRING (40)
    "Surname" = Surname : STRING (40)
    "Lesson" = Lesson : STRING (20)
  END LIST
...
END TYPE

  ...
END REPORT
```

2.3.2.9. Report Commands Definition

In this section you may want to define new commands, to do so, all you need is an already defined Actor from which you will call the command in order to execute an Action on the form. There are 3 ways to define this Action: every command have an effective ray of action (VRField, VReport)

- Calling a function with the CALL command
- Calling a function outside of the report using the EXTERN command
- Simply writing the body of the action using the ACTION command, the parameters are optional and can be VRField or VReport.

Syntax

```
cmdDef: "COMMAND" QualifiedName
        "ITEM" SimpleName
        commandBody
        "END" "COMMAND"

cmdBody: "CALL" SimpleName | "EXTERN" QualifiedName
         |"ACTION"
         [(VField SimpleName | VBlock SimpleName | VForm SimpleName)]
         {KOPI_JAVA statements}
```

Example

Calling a local action :

```
COMMAND print
  ITEM  printActor
  CALL  printReport
END COMMAND
```

Example Calling an external action :

```
COMMAND doAction
  ITEM Save
  EXTERN com.company.vkopi.lib.report.Actions
END COMMAND
```

Example Writing the action's body :

```
COMMAND PrintReport
```

```
  ITEM  PrintReport
  ACTION (VReport report){
    // KOPI_JAVA code
  }
END COMMAND
```

2.3.2.10. Report Insert Definition

This command allows you to actually define your types, menus, actors, commands in another file, and then insert it in any report you want, thus avoiding rewriting recurrent definitions. You still can add definition before or after using the INSERT command.

Syntax

```
Insert Definition : "INSERT" String filePath
```

Example

```
REPORT "List of Students"

  {
  package at.dms.app.application
  }

  INSERT "ReportDefault.vr"

  TYPE Lesson IS
    LONG(11)
  END TYPE

...
END REPORT
```

2.3.2.11. Report Commands Declaration

After the Definitions section of the report, you have to enter the BEGIN keyword in order to begin the report declaration part where you can optionally call report commands and triggers, and where you have to enter the report's fields. Concerning the commands declaration, you can start by the REPORT COMMAND statement that add all the default report commands to your report that include printing and exporting the report. Now you may call other commands using the COMMAND keyword followed by a defined command name from those who already exist or the ones you defined in the command definition section.

Example

```
COMMAND Sort
COMMAND PrintReport
```

You can also define your command when declaring it instead of defining it in the command definition section of the report, simply write the command definition like explained before, but with no name, in the command declaration section of the report.

Example

```
COMMAND
  ITEM actorName
```

```
  ACTION {
    KOPI_JAVA code
  }
END COMMAND


COMMAND
  ITEM actorName
  CALL localFunction
END COMMAND
```

2.3.2.12. Report Triggers Declaration

Report Triggers are special events that once switched on you can execute a set of actions defined by the following syntax :

Syntax

```
ReportTrigger  :    ReportEventList TriggerAction
ReportEventList:  ReportEvent [,FormEvent]*
```

Kopi actually defines 2 report Triggers or report Events :

- PREREPORT : executed before the report is displayed.
- POSTREPORT : executed after the report is closed.

Example

```
REPORT "Orders Report"
{
  package at.dms.app.application
}

INSERT "ReportDefault.vr"

BEGIN

  REPORT COMMAND

  PREREPORT{
    System.out.println("This message is displayed before the report appears");
  }
  ...
END REPORT
```

2.3.2.13. Report Fields Declaration

As you already know, a dynamic report is based on field that will be shown as report columns, in this section you have to write at least on field definition or more following the definition and the structure we saw in the previous chapter.

2.3.2.14. Report Context Footer:

This section should follow the report fields declaration and have to be placed between curly braces, here you may define all the functions, data, classes you need in your report, written in KOPI_JAVA

code. But most important, you need to define the constructor of you report, this constructor will be responsible of filling the report's lines or rows. All you need to do is importing your data (a List or vector parameter, database query cursors ...) , declaring a row in the report then add the Add(); statement to add the row to the report. The constructor may have different parameters but a DBContextHandler object is compulsory (a form most of the time) For example here is the constructor of a dynamic report named UserList with 3 fields (FirstName, LastName, Age).We will retrieve data from the User table on the database.

Example

```
void UserList(DBContextHandler context) throws VException {
  super(context);
  #protected () {
    try {
      #cursor (){
        SELECT int U.age,
               String U.firstname,
               String U.lastname,

        FROM    User U

      } cursor;

      cursor.open();

      while (cursor.next())
      {
        FirstName = cursor.U.firstname;
        LastName  = cursor.U.lastname;
        Age       = cursor.U.age;
        add();
      }
      cursor.close();
   } catch (DBNoRowException) {
        System.out.println("DBNoRowException");
      }
  }
}
```

2.3.3. Calling reports

A report is always called from a form, if the caller form extends from the VDictionaryForm class you have to do the following steps :

- Change VDictionaryForm to VReportSelectionForm
- Add the CreateReport command to the caller form
- Implement the createReport abstract method :

Example

```
protected VReport createReport() throws VException{
  return new USerList(this);
}
```

Otherwise you can create a normal form or block command that executes the following code :

```
WindowController.getWindowController().doNotModal(new UserList(this));
```

2.4. Visual Kopi Print Pages

Visual Prints are static reports that you can create with the Kopi framework, such reports are written in files with the (.vp) extension and have a lot of differences from the dynamic reports we have seen earlier. In fact, prints are more complicated to create since you have to take care of the design, the positions and the style details of the report. The result is what we call a static report that you cannot modify or interact with in the PDF format, but you can print, save send .. We also call these printable Pages.

2.4.1. Print Page Structure

When writing a static report, you have to define the data structures, then you insert the data in the report using the defined structures, finally you need to write the print constructor. A print always begins with PAGE and ends with END PAGE. Here is the global syntax of a page :

Syntax

```
PageDefinition :  "PAGE" "IS" QualfiedName
                  "IMPLEMENTS" QualifiedName[,QualifiedName]* ]
                  {
                    PackageDeclaration;
                    [ImportDeclaration];
                  }
                  [InsertDefinition]
                  StylesDefinition
                  "BEGIN"
                  PageFormat
                  PageBlocks
                  [PageHeaderFooter]
                  [ContextFooter]
                  "END" "PAGE"
```

2.4.1.1. Page Superclass And Interfaces

- **Superclass:**

Syntax:

```
SuperPage:   "IS" QualifiedName
```

Every Page is a class that may extend another Java class by using the optional keyword IS. Otherwise, Kopi will automatically take over the java.lang.Object QualifiedName.

Example:

```
PAGE IS PProtectedPage
```

- **Interfaces**

You can also specify interfaces which the Page may implement by using the IMPLEMENTS option.
Syntax

```
"IMPLEMENTS" QualifiedName[,QualifiedName]*
```

2.4.1.2. Page Header

Syntax:

```
ContextHeader:   { PackageDeclaration  ImportDeclarations  }

PackageDeclaration:   <As defined in JLS 7.4>
ImportDeclarations:   <As defined in JLS 7.5>
```

The package definition is contained in the header. It consists in defining where this Page belongs to i.e which application, which firm or which person it is related to.

Example

```
PAGE IS PProtectedPage
  {
  package at.dms.apps.timetable;

  import java.util.Hashtable
  import java.util.Vector
  }
  ...
END PAGE
```

In this example, the Page belongs to the DMS firm and is a part of the application called "timetable". In addition to this Java package declaration, you can make a Java import declaration in order to import some classes from other packages. You can add an unlimited number of imports.

2.4.1.3. Insert Definitions

You can user other files to write your print pages styles and then have them inserted in your print page definition file (.vp) using the INSERT command.

Example

```
INSERT "PrintDefault.vp"
```

2.4.1.4. Styles Definitions

In this part of the Page, you have to define the structures and the styles of your page, you can use the following 3 types of styles to define as much structures as you need.

The Style types are :

- BLOCK STYLES
- TEXT STYLES
- PARAGRAPH STYLES
- Block Styles

Block styles are simple structures that only have Border and Background parameters to set : Syntax

```
BlockStyle : "BLOCK" "STYLE" SimpleName ["IS" SimpleName]
             ["BORDER" Integer border
             |
             "BACKGROUND" Integer r Integer g Integer b]*
```

```
            "END" "STYLE"
```

the BORDER statement sets the thickness of the block's border, while the BACKGROUND sets the color of the block's background by setting the RGB color parameters.

Example

```
    BLOCK STYLE whiteBlock
        BORDER 5
    END STYLE
```

- **Text Styles**

Text styles are structures that set text data properties : **Syntax**

```
TextStyle  : "TEXT" "STYLE" SimpleName ["IS" SimpleName]
             ["BACKGROUND" Integer r Integer g Integer b
             |
             "FOREGROUND" Integer r Integer g Integer b
             |
             "BOLD"
             |
             "ITALIC"
             |
             "SUBSCRIPT"
             |
             "SUPERSCRIPT"
             |
             "UNDERLINE"
             |
             "FONT" String font
             |
             "SIZE" Integer size
             |
             "STRIKETHROUGH"]*

             "END" ""STYLE
```

Example

```
TEXT STYLE text
  BOLD
  SIZE 22
  FONT "Courier"
  UNDERLINE
END STYLE
```

- **Paragraph Styles**

Paragraph styles are structures for table columns like data entries with :

Syntax

```
ParagraphStyle : "PARAGRAPH" "STYLE" SimpleName ["IS" SimpleName]
                 ["ALIGN" PositionAlign
                  |
                 "IDENT" PositionIdent
```

```
                  |
                 "FIRST" "LINE" "IDENT" Integer firstlineIdent
                  |
                 "LINE" "SPACING" Fixed spaces
                  |
                 "ORIENTATION"
                  |
                 "SPACE" PositionSpace
                  |
                 "MARGIN" PositionMargin
                  |
                 "BORDER" Integer border  [TOP | BOTTOM | LEFT | RIGHT]*
                  |
                 "BACKGROUND" Integer r Integer g Integer b
                  |
                 "NO BACKGROUND"
                  |
                 TABSET PrintTabset]*

                 "END" "STYLE"



PositionAlign  : "LEFT"  | "RIGHT"  | "CENTER"  | "JUSTIFIED"
PositionIdent  : "LEFT" Integer left   |  "RIGHT" Integer right
PositionMargin : "LEFT" Integer left   |  "RIGHT" Integer right
PositionSpace  : "ABOVE" Integer above | "BELOW" Integer below
PrintTabset    : (["TAB"] SimpleName "AT" Integer pos "ALIGN" alg)*
alg            : LEFT | CENTER | RIGHT | DECIMAL
```

Example

```
PARAGRAPH STYLE table
  ALIGN LEFT
  INDENT LEFT 5
  TABSET
    Article AT 10 ALIGN LEFT
    Quantity AT 140 ALIGN RIGHT
    Price AT 260 ALIGN RIGHT
    Discount AT 360 ALIGN RIGHT
    Total AT 490 ALIGN RIGHT
  BORDER 3 TOP
END STYLE


PARAGRAPH STYLE total IS posTabs
  BORDER 5
  BACKGROUND 200 190 210
END STYLE
```

2.4.1.5. Page Format Definition

This section is optional, here you can define the page format :

Syntax

```
FormatDefinition : [  "LANDSCAPE" | "PORTRAIT"  [String formatValue]   ]

formatValue      : (Integer width  Integer height  Integer border )
                   | A5 | A4 | A3 | legal | letter
```

All you have to do is to insert the keyword that sets the orientation of the page (PORTRAIT or LANDSCAPE), this keyword can be followed by a string that defines the format of the page. This String can be one of the defined Strings (A5,A4,A3,letter,legal) or you can define the width, the height and the border of the page in a string with the 3 parameters in the mentioned order. If you don't define this section, the default values will be PORTRAIT "A4"

Example

```
PORTRAIT "A4"

LANDSCAPE "A5"

PORTRAIT "letter"

PORTRAIT "592 842 25"
```

2.4.1.6. Page Blocks Definition

This section of the page is where you will insert your data in the form of blocks that can inherit the styles you have defined in the styles definition section of the page. There are 5 types of blocks you can use to insert you data:

- Text Blocks
- Recursive Blocks
- Horizontal Blocks
- List Blocks
- Rectangle Blocks

There is also another method that consist on inserting already defined Blocks with the INSERT command. Before explaining every block type, there are some common properties that are used in most of these blocks which are :

BlockPosition

```
BlockPosition   : [ "POS" Integer posx (Integer posy | "BELOW")]
                  [ "SIZE" (Integer width | String WidthString)
                   (Integer height| String HeightString) ]
                  [ "SHOW" "IF" KOPI_JAVA code]

WidthString     : "PAGE_WIDTH"  | "MAX"
HeightString    : "PAGE_HEIGHT" | "MAX"
```

The block position property sets the block's position by specifying the x and y Integer values, the y value can be replaced by the keyword "BELOW" and the text will be placed in the first available position in the y axis. You can also specify the size of the text block using the SIZE command followed by the width and the height integers, there are also special String values that you can use for the size, the MAX String can be used for the width or the height, it is equal to 1000. Width also can be

PAGE_WIDTH and height can be PAGE_HEIGHT. After the size definition you can enter a display condition with the command SHOW IF followed by a condition statement, for example SHOW IF getCurrentPage() == 1; means that the text will be displayed only on the first page.

BlockTriggers

```
PrintTriggers       : SimpleName [(ParameterList)] { TrigSource }

TrigSource          : [String line | <TAB StyleName> | <StyleName>
                      | "PAGE_COUNT" | (KOPI_JAVA code)
                      | {( (KOPI_JAVA code) => (TrigSource)* }]*
```

To insert the data into the block, you need to create what we call print triggers, a kind of method that defines the way you fill your block. These triggers can optionally have a parameter list like any JAVA method and must have a body that can contain various elements:

- Simple String texts
- Styles : to use a style all you need to do is to insert it between "<" and ">", if the style has a TABSET , you have to insert TAB before the name of its elements.
- Kopi_Java code between parentheses
- kopi_java condition between parentheses followed by "#" then one or more trigger, all this statement shave to be inserted into "{}" and means that the the condition have to be fulfilled in order to activate the trigger placed after the "#".

BlockBody

```
BlockBody  : "BODY" { KOPI_JAVA code }
```

After the triggers, you can have a body block that have to implement the print() abstract method in kopi_java code in which you can use the triggers defined in the same block, it is optional in a text block but you will have to implement it in some of the other blocks like the recursive block.

- **Text Blocks Definition**

Text blocks are data blocks that can be used to show simple text on the print page : **Syntax**

```
TextBlockDefinition : ["REC"] "TEXT" SimpleName
                      BlockPosition
                      ["STYLE" SimpleName]
                      (PrintTriggers)*
                      [BlockBody]
                      "END" "BLOCK"
```

A text Block begins by "TEXT" and ends by "END BLOCK", it can have the optional keyword "REC" before "TEXT" to make the text block recurrent.

Example Using only block triggers :

```
REC TEXT Invoice
  POS 60 160
  SIZE 290 MAX
  SOURCE {
    <alignLeft><helvetica9> <bold> "Invoice N° " (inv.NumInvoice) "\n"
  }
```

```
END BLOCK
```

Example Using Body section:

```
REC TEXT Details
  POS 60 190
  SIZE 200 100

  line(String title, Object description)
  {
    <details><helvetica7><!bold> (title) ":"
    <TAB TAB1><helvetica8><!bold> (description)
  }

  boldline(String title, Object description)
  {
    <details><helvetica7><bold> (title) ":"
    <TAB TAB1><helvetica8><bold> (description)
  }

  newLine
  {
    <helvetica4> "\n"
  }

  BODY
  {
    Remise
    protected void print() throws Exception
      {
        line("Page", ""+getCurrentPage());
        boldline("Date", inv.DateInvoice);
        boldline("Client", inv.Customer );
        line("Adress", inv.Adress);
        line("Payment Mode", inv.PaymentMode);
        line("Delivery Mode", inv.DeliveryMode);
      }
  }
END BLOCK
```

- **List Blocks Definition**

List blocks are used to insert tables of based on TABSET styles in the page:

Syntax

```
ListBlockDefinition  : "LIST" [SimpleName]
                       [BlockPosition]
                       "STYLE" SimpleName
                       (PrintTriggers)*
                       [BlockBody]
                       "END" "BLOCK"
```

The STYLE section only accepts block styles.

Example

```
 //STYLES DEFINITIONS


 PARAGRAPH STYLE posTabs
   ALIGN LEFT
   INDENT LEFT 5
   TABSET
     Article AT 10 ALIGN LEFT
     Quantity AT 140 ALIGN RIGHT
     Price AT 260 ALIGN RIGHT
     Discount AT 360 ALIGN RIGHT
     Total AT 490 ALIGN RIGHT
   BORDER 3 TOP
 END STYLE


 PARAGRAPH STYLE posTabsHead IS posTabs
   BORDER 5
   BACKGROUND 200 190 210
 END STYLE

//INSERTING DATA

 LIST Loop1
 SIZE 500 480

 LISTHEADER
 {
   <posTabsHead> <helvetica9><bold>
   <TAB Article> ("Article")
   <TAB Quantity> ("Quantity")
   <TAB Price> ("Price")
   <TAB Discount> ("Discount")
   <TAB Total> ("Total")
  }
 line(InvoicePos invpos)
 {
   <posTabs>
   <helvetica4> "\n" <helvetica9>
   <TAB Article> (invpos.Article)
   <TAB Quantity> (invpos.Quantity)
   <bold>
   <TAB Price> (invpos.Price)
   <TAB Discount> (invpos.Discount) " %"
   <TAB Total> (invpos.Quantity * (invpos.Price -
   ((invpos.Price * invpos.Discount)/100)))
   <helvetica4> "\n"
 }
 BODY
 {
   protected void print() throws Exception
     {
       invpos.setDefaultConnection(getDBContext().getDefaultConnection());
       invpos.open(id);
```

```
        while (invpos.next())
        {
          line(invpos);
          addBreak();
        }
        invpos.close();
    }
  }
  END BLOCK
```

- Recursive Blocks Definition

Also called vertical blocks, they are used to contain multiple blocks and displaying them vertically, you can use one of the block styles that you have defined in this block definition after the keyword STYLE, but no text styles and paragraph styles are allowed.

Syntax

```
RecursiveBlockDefinition : "VERTICAL" "BLOCK" [SimpleName]
                           [BlockPosition]
                           ["STYLE" SimpleName]
                           (PageBlocks).
                           "END" "BLOCK"
```

Example

```
VERTICAL BLOCK Loop
  POS 60 280
  SIZE 150 400
  STYLE darkGreyBlock

  LIST Loop1
  SIZE 500 480

  LISTHEADER
  {
    <posTabsHead> <helvetica9><bold>
    <TAB Article> ("Article")
    <TAB Quantity> ("Quantity")
    <TAB Price> ("Price")
    <TAB Discount> ("Discount")
    <TAB Total> ("Total")
   }
  line(InvoicePos invpos)
  {
    <posTabs>
    <helvetica4> "\n" <helvetica9>
    <TAB Article> (invpos.Article)
    <TAB Quantity> (invpos.Quantity)
    <bold>
    <TAB Price> (invpos.Price)
    <TAB Discount> (invpos.Discount) " %"
    <TAB Total> (invpos.Quantity * (invpos.Price -
    ((invpos.Price * invpos.Discount)/100)))
    <helvetica4> "\n"
```

```
  }
  BODY
  {
    protected void print() throws Exception
      {
        invpos.setDefaultConnection(getDBContext().getDefaultConnection());
        invpos.open(id);
        while (invpos.next())
        {
          line(invpos);
          addBreak();
        }
        invpos.close();
    }
  }
  END BLOCK
END BLOCK
```

- Horizontal Blocks Definition

Horizontal Blocks are the same as vertical blocks (recursive blocks), they do contain other blocks but they display them horizontally, you can use one of the block styles that you have defined in this block definition after the keyword STYLE, but no text styles and paragraph styles are allowed.

Syntax

```
RecursiveBlockDefinition : "HORIZONTAL" "BLOCK" [SimpleName]
                           [BlockPosition]
                           ["STYLE" SimpleName]
                           (PageBlocks).
                           "END" "BLOCK"
```

Example

```
HORIZONTAL BLOCK texts
  POS 60 280
  SIZE 150 400
  TEXT text1
  ...
  END BLOCK

  TEXT text2
  ...
  END BLOCK
END BLOCK
```

- Rectangle Blocks Definition

This Type of blocks just insert a rectangle with the parameters of position, size and block style to set.

Syntax

```
RectangleBlockDefinition : "RECT" SimpleName
                           [BlockPosition]
                           "STYLE" SimpleName
                           "END" "BLOCK"
```

Example

```
RECT Rectangle
  POS     50 120
  SIZE    100 150
  STYLE   greyBlock
END BLOCK
```

- Imported Blocks Definition

You can import already defined blocks using the INSERT keyword

Syntax

```
ImportedBlockDefinition : "INSERT" BlockType QualifiedName

BlockType               : "RECT" | "VERTICAL" "BLOCK"
                          | "HORIZONTAL" "BLOCK" | "TEXT"
                          | "LIST"
```

Example

```
INSERT TEXT OtherReport.BlockText
```

2.4.1.7. Page Header and Footer

After defining the page blocks , you can define the header and the footer,to do so you can just create a trigger named PAGEFOOTER or PAGEHEADER, or you can import a block as a header/footer.
Examples

```
PAGEFOOTER {
  <landscape> <helvetica8> "Invoice N° " (numInvoice)
  "  -  Page " (getCurrentPage()) " / " (getPageCount()) "\n"
     <helvetica8> (Date.now()) " " (Time.now())
}


PAGEFOOTER AnotherPageFile.definedBlock
```

2.4.1.8. Page Context Footer

In this section is inserted between curly brackets after you will have to define the page's constructor and initializers, the data members and the types definitions and other methods.

Here is an example named InvoicePage, notice the use of cursor for types definitions, the initPage() method sets the connection to the database. This static report prints a page of an invoice by its id:
Example

```
//TYPES DEFINITION
#cursor Invoice (int id)
{
  SELECT int NumInvoice,
         String Customer,
         String Adress,
         String PaymentMode,
         String DeliveryMode,
```

```
        date DateInvoice,
        fixed Amount
  FROM  Invoices I
  WHERE I.ID = :(id)
}

#cursor InvoicePos (int id)
{
  SELECT String Article,
        int Quantity,
        fixed Price,
        nullable int Discount
  FROM  InvoicesPos IP
  WHERE IP.Invoice = :(id)
}

//INITIALIZERS

public InvoicePage(DBContextHandler handler, int id)
{
  super(handler);
  this.id = id;
}
public void initPage() throws SQLException
{
  inv.setDefaultConnection(getDBContext().getDefaultConnection());
  inv.open(id);
  inv.next();

}
public void closePage() throws SQLException
{
inv.close();
}

// DATA MEMBERS

private int id;
private Invoice inv = new Invoice();
private InvoicePos invpos = new InvoicePos();
```

2.4.2. Calling Print Pages

Print pages are only available when called from a form that extends the VPrintSelectionForm class, besides, this class have to to implement the abstract method createReport and the command print.

Example

```
FORM IS VPrintSelectionForm
...

{
  public PProtectedPage createReport(DBContextHandler handler){
  return new InvoicePage (handler , @!I.ID);
```

```
  }
}
END FORM
```

Chapter 3. The Kopi Project Future

After reading this document, you should be able to create efficient database applications using the Kopi framework, you can also create dynamic reports and print pages, the all using java embedded SQL with kopi java extended language. Actually, the project owners are working on a web version of the kopi framework called "KOPI WEB", this version features the same capabilities of the SWING based version and is based on the Vaadin framework which is an open source web application framework for rich Internet applications Vaadin, uses java as programming language to create web content and Google web toolkit to render the resulting web page.