

Object Types

This section covers the following topics:

- What Types of Programming Objects Are There?
 - Data Areas
 - Programs, Subprograms and Subroutines
 - Maps
 - Helproutines
 - Multiple Use of Source Code - Copycode
 - Documenting Natural Objects - Text
 - Creating Event Driven Applications - Dialog
 - Creating Component Based Applications - Class
 - Using Non-Natural Files - Resource
-

What Types of Programming Objects Are There?

Within a Natural application, several types of programming objects can be used to achieve an efficient application structure.

There are the following types of Natural programming objects:

- Local Data Area
- Global Data Area
- Parameter Data Area
- Program
- Subprogram
- Subroutine
- Helproutine
- Map
- Copycode
- Text
- Dialog
- Class

To create and maintain all these objects, you use the Natural editors:

- Local data areas, global data areas and parameter data areas are created/maintained with the *data area editor*.
- Maps are created/maintained with the *map editor*.
- Dialogs are created/maintained with the *dialog editor*.
- Classes are created/maintained with the *Class Builder*.
- All other types of objects listed above are created/maintained with the *program editor*.

The editors are described in your Natural User's Guide.

Data Areas

As explained in the section Defining Fields, all fields that are to be used in a program have to be defined in a DEFINE DATA statement.

The fields can be defined within the DEFINE DATA statement itself; or they can be defined outside the program in a separate data area, with the DEFINE DATA statement referencing that data area.

Natural supports three types of data areas:

- **Local Data Area**
In a local data area, you define the data elements that are to be used by a single Natural module in an application.
- **Global Data Area**
In a global data area, you define the data elements that are to be used by more than one Natural program, routine, etc. in an application.
- **Parameter Data Area**
In a parameter data area, you define the fields that are passed as parameters to a subprogram, external subroutine or help routine.

Local Data Area

Variables defined as local are used only within a single Natural module. There are two options for defining local data:

- You can define the data within the program.
- You can define the data in a local data area outside the program.

In the first example, the fields are defined within the DEFINE DATA statement of the program. In the second example, the same fields are defined in a local data area, and the DEFINE DATA statement only contains a reference to that data area.

Example 1 - Fields Defined within a DEFINE DATA Statement:

```
DEFINE DATA LOCAL
  1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
  1 #VARI-A (A20)
  1 #VARI-B (N3.2)
  1 #VARI-C (I4)
END-DEFINE
...
```

Example 2 - Fields Defined in a Separate Data Area:

Program:

```
DEFINE DATA LOCAL
  USING LDA39
END-DEFINE
...
```

Local Data Area "LDA39":

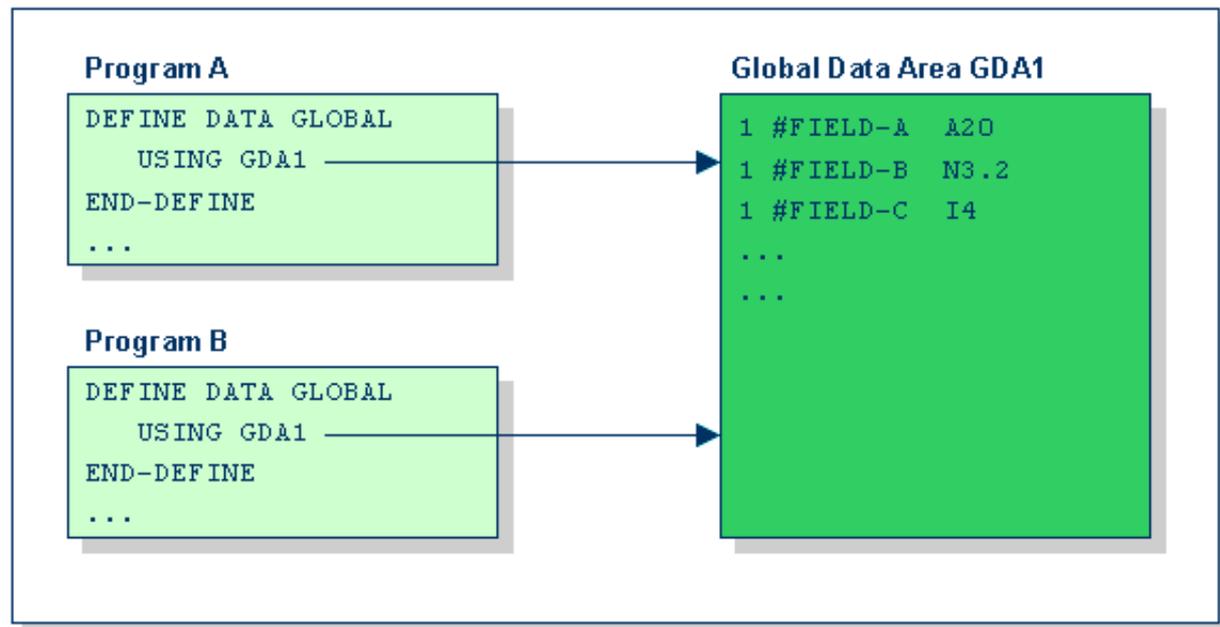
I	T	L	Name	F	Leng	Index/Init/EM/Name/Comment
V	1		VIEWEMP			EMPLOYEES
	2		NAME	A	20	
	2		FIRST-NAME	A	20	
	2		PERSONNEL-ID	A	8	
	1		#VARI-A	A	20	
	1		#VARI-B	N	3.2	
	1		#VARI-C	I	4	

For a clear application structure, it is usually better to define fields in data areas outside the programs.

Global Data Area

In a global data area, you define the data elements that are to be used by more than one program, routine, etc. in an application.

Variables defined in a global data area may be referenced by several objects in an application.



The global data area and the objects which reference it must be in the same library (or a steplib).

Global data areas must be defined with the data area editor, and a program using that data area must reference it in the DEFINE DATA statement. Any number of main programs, external subroutines and help routines can share the same global data area.

Each object can reference only one global data area; that is, a DEFINE DATA statement must not contain more than one GLOBAL clause.

Note:

When you build an application where multiple objects share a global data area, remember that modifications to a global data area affect all programs or routines that reference that data area. Therefore these objects must be STOWed again after the global data area has been modified.

When are Global Data Areas Initialized?

A global data area is initialized when it is used for the first time. It remains active in the current Natural session (that is, the variables in the global data area retain their contents) until:

- the next LOGON, or
- another global data area is used on the same level (levels are described later in this section), or
- a RELEASE VARIABLES statement is executed. In this case, the variables in the global data area are reset when either the execution of the level 1 program is finished, or the program invokes another program via a FETCH or RUN statement.

Note:

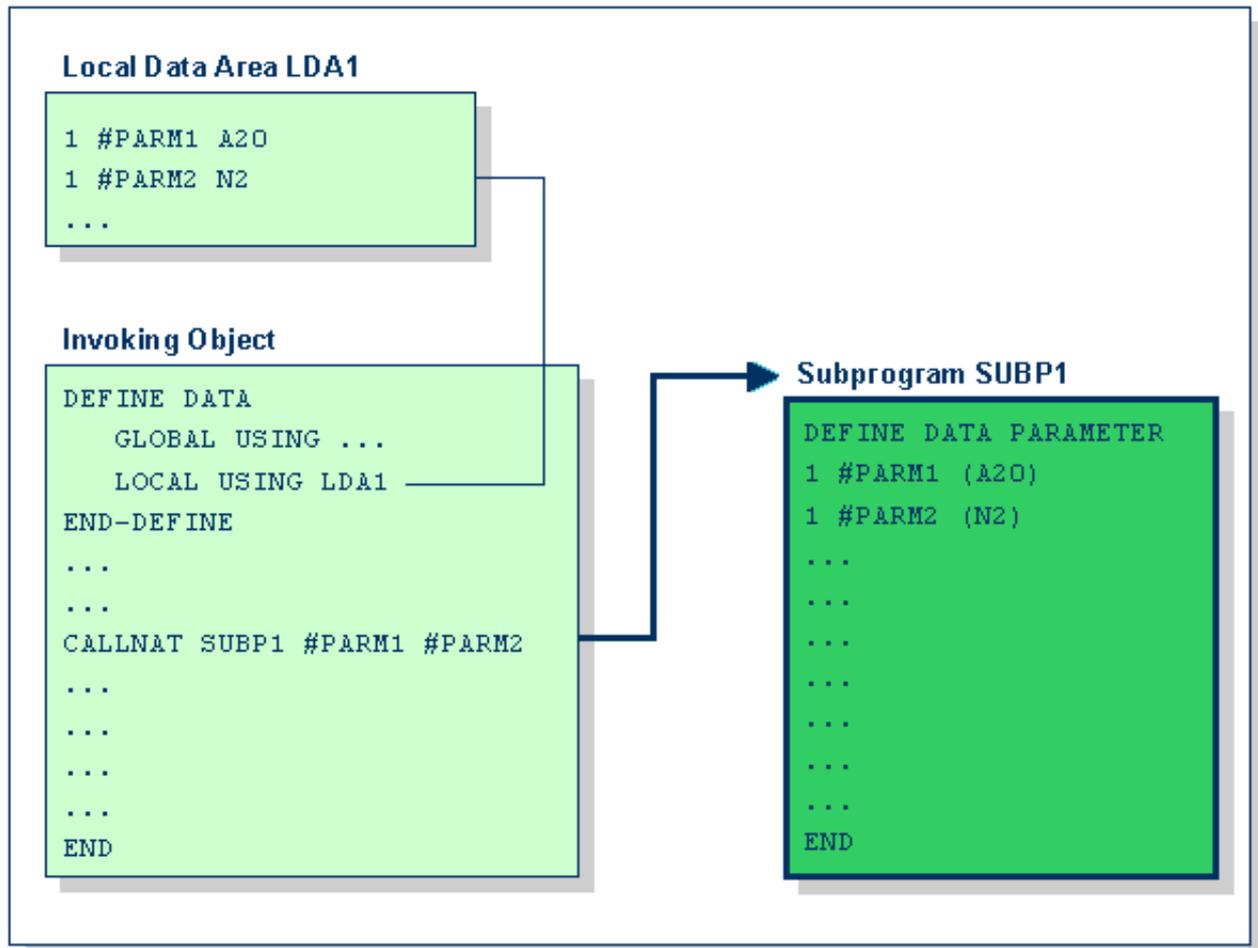
If a GDA named "COMMON" exists in a library, the program named ACOMMON is invoked automatically when you LOGON to that library.

Parameter Data Area

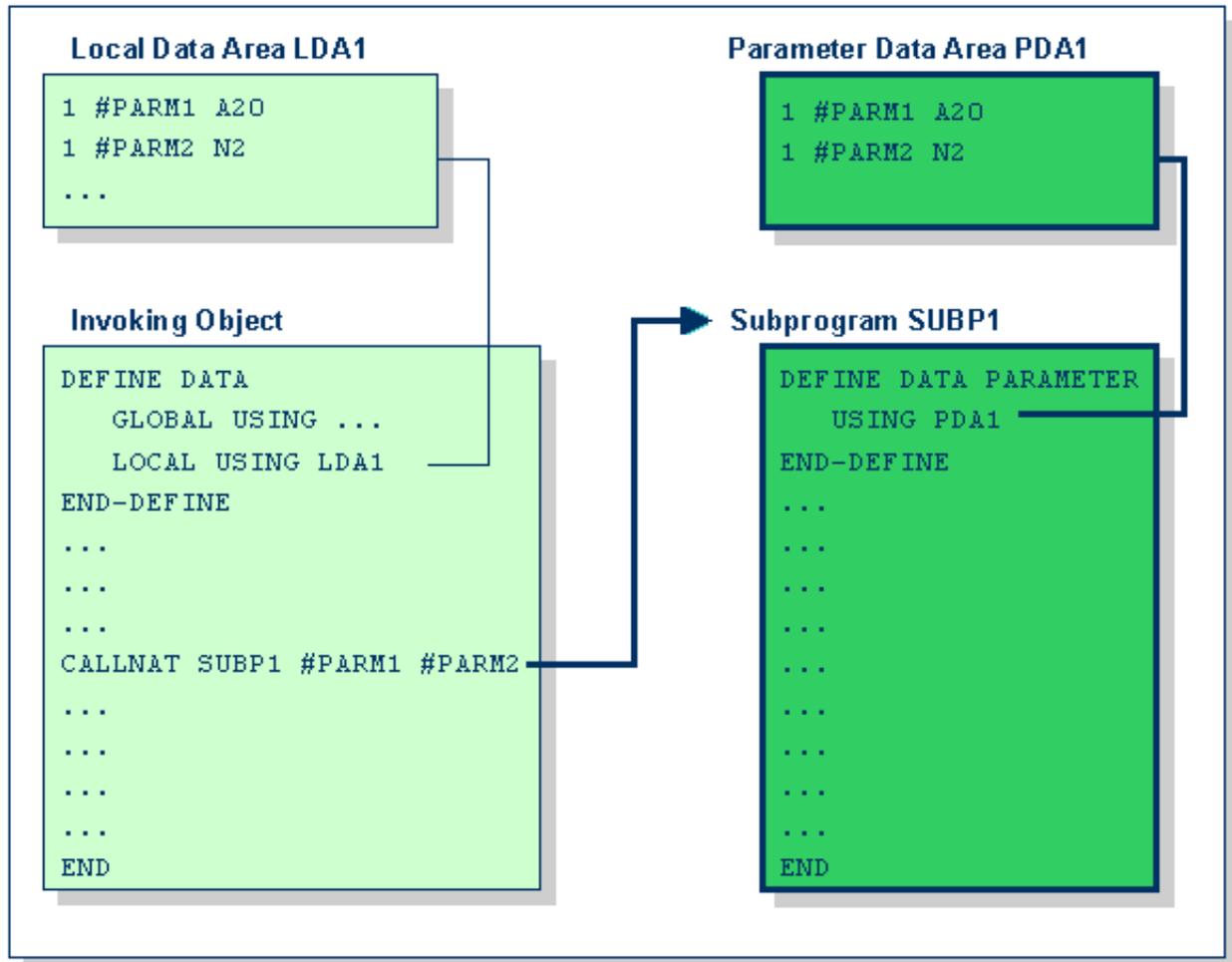
Parameter data areas are used by subprograms and external subroutines.

A subprogram is invoked with a CALLNAT statement. With the CALLNAT statement, parameters can be passed from the invoking object to the subprogram. These parameters must be defined with a DEFINE DATA PARAMETER statement in the subprogram: they can be defined in the PARAMETER clause of the DEFINE DATA statement itself; or they can be defined in a separate parameter data area, with the DEFINE DATA PARAMETER statement referencing that parameter data area.

Parameter Defined Within DEFINE DATA PARAMETER Statement:



Parameter Defined in Parameter Data Area:



In the same way, parameters that are passed to an external subroutine via a PERFORM statement must be defined with a DEFINE DATA PARAMETER statement in the external subroutine.

In the invoking object, the parameter variables passed to the subprogram/ subroutine need not be defined in a parameter data area; in the illustrations above, they are defined in the local data area used by the invoking object (but they could also be defined in a global data area).

The sequence, format and length of the parameters specified with the CALLNAT/ PERFORM statement in the invoking object must exactly match the sequence, format and length of the fields specified in the DEFINE DATA PARAMETER statement of the invoked subprogram/subroutine. However, the names of the variables in the invoking object and the invoked subprogram/subroutine need not be the same (as the parameter data are transferred by address, not by name).

Programs, Subprograms and Subroutines

The following topics are covered below:

- A Modular Application Structure
- Multiple Levels of Invoked Objects
- Program
- Subroutine
- Subprogram
- Processing Flow when Invoking a Routine

A Modular Application Structure

Typically, a Natural application does not consist of a single huge program, but is split into several modules. Each of these modules will be a functional unit of manageable size, and each module is connected to the other modules of the application in a clearly defined way. This provides for a well structured application, which makes its development and subsequent maintenance a lot easier and faster.

During the execution of a main program, other programs, subprograms, subroutines, help routines and maps can be invoked. These objects can in turn invoke other objects (for example, a subroutine can itself invoke another subroutine). Thus, the modular structure of an application can become quite complex and extend over several levels.

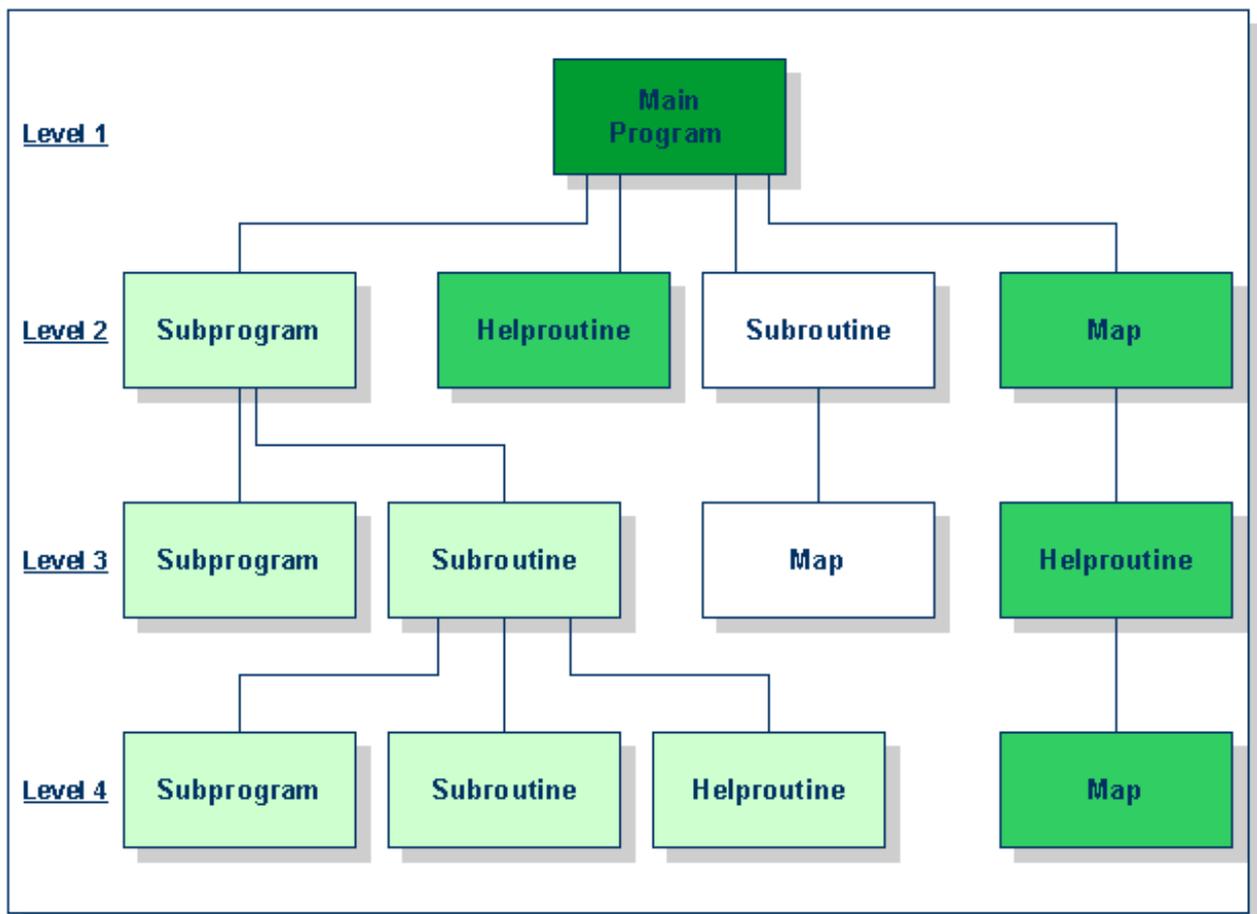
Multiple Levels of Invoked Objects

Each invoked object is one level below the level of the object from which it was invoked; that is, with each invocation of a subordinate object, the level number is incremented by 1.

Any program that is directly executed is at level 1; any subprogram, subroutine, map or helproutine directly invoked by the main program is at level 2; when such a subroutine in turn invokes another subroutine, the latter is at level 3.

A program invoked with a FETCH statement from within another object is classified as a main program, operating from level 1. A program that is invoked with FETCH RETURN, however, is classified as a subordinate program and is assigned a level one below that of the invoking object.

The following illustration is an example of multiple levels of invoked objects and also shows how these levels are counted:



If you wish to ascertain the level number of the object that is currently being executed, you can use the system variable *LEVEL (which is described in the Natural Reference documentation).

This section discusses the following Natural object types, which can be invoked as routines (that is, subordinate programs):

- program
- subroutine
- subprogram

Help routines and maps, although they are also invoked from other objects, are strictly speaking not routines as such, and are therefore discussed in later sections of this section.

Basically, programs, subprograms and subroutines differ from one another in the way data can be passed between them and in their possibilities of sharing each other's data areas. Therefore the decision which object type to use for which purpose depends very much on the data structure of your application.

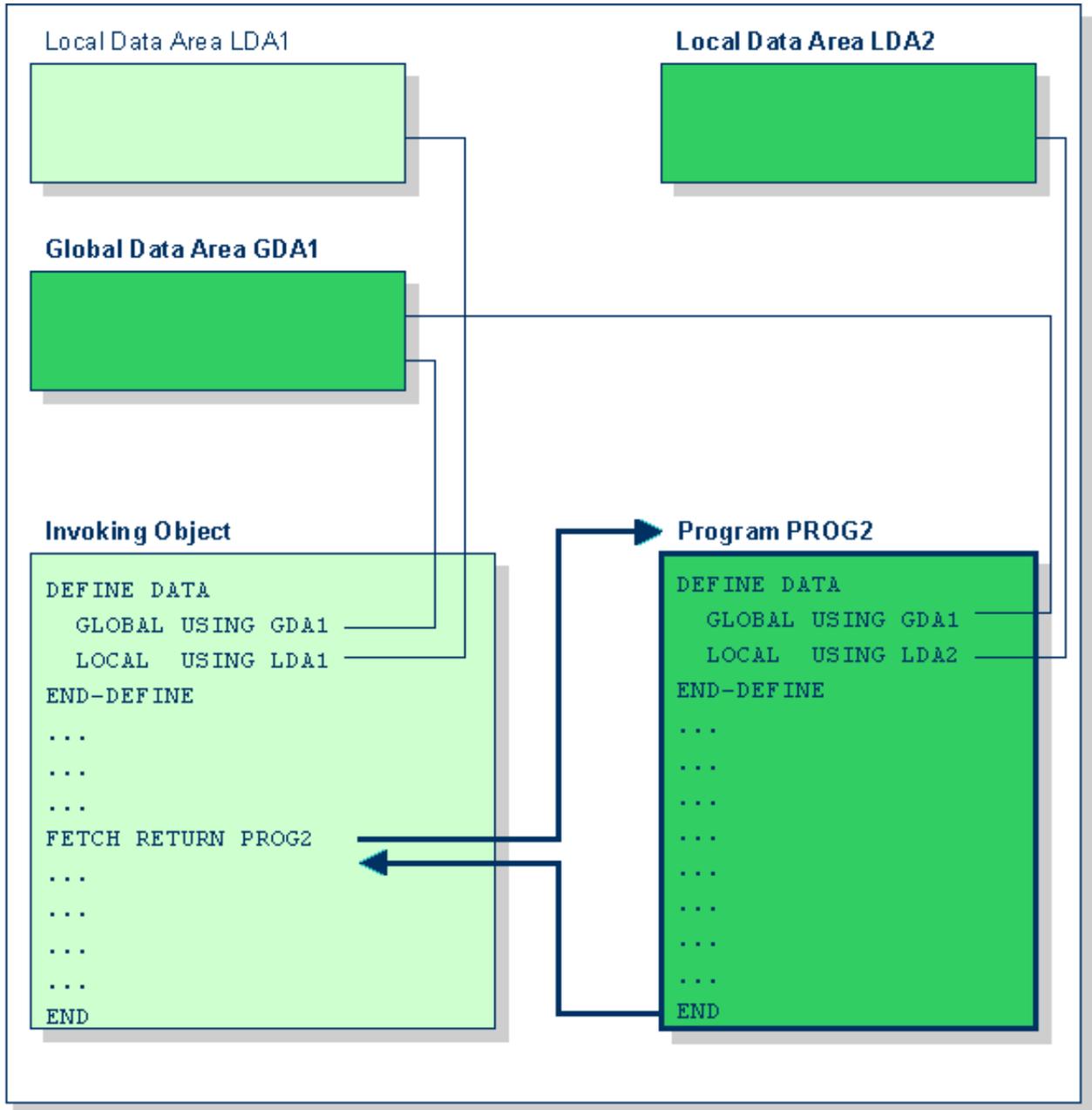
Program

A program can be executed - and thus tested - by itself. To compile and execute a source program, you use the system command RUN. To execute a program that already exists in compiled form, you use the system command EXECUTE.

A program can also be invoked from another object with a FETCH or FETCH RETURN statement. The invoking object can be a program, subprogram, subroutine or help routine.

- When a program is invoked with FETCH RETURN, the execution of the invoking object will be suspended - not terminated - and the FETCHed program will be activated as a *subordinate program*. When the execution of the FETCHed program is terminated, the invoking object will be re-activated and its execution continued with the statement following the FETCH RETURN statement.
- When a program is invoked with FETCH, the execution of the invoking object will be terminated and the FETCHed program will be activated as a *main program*. The invoking object will not be re-activated upon termination of the FETCHed program.

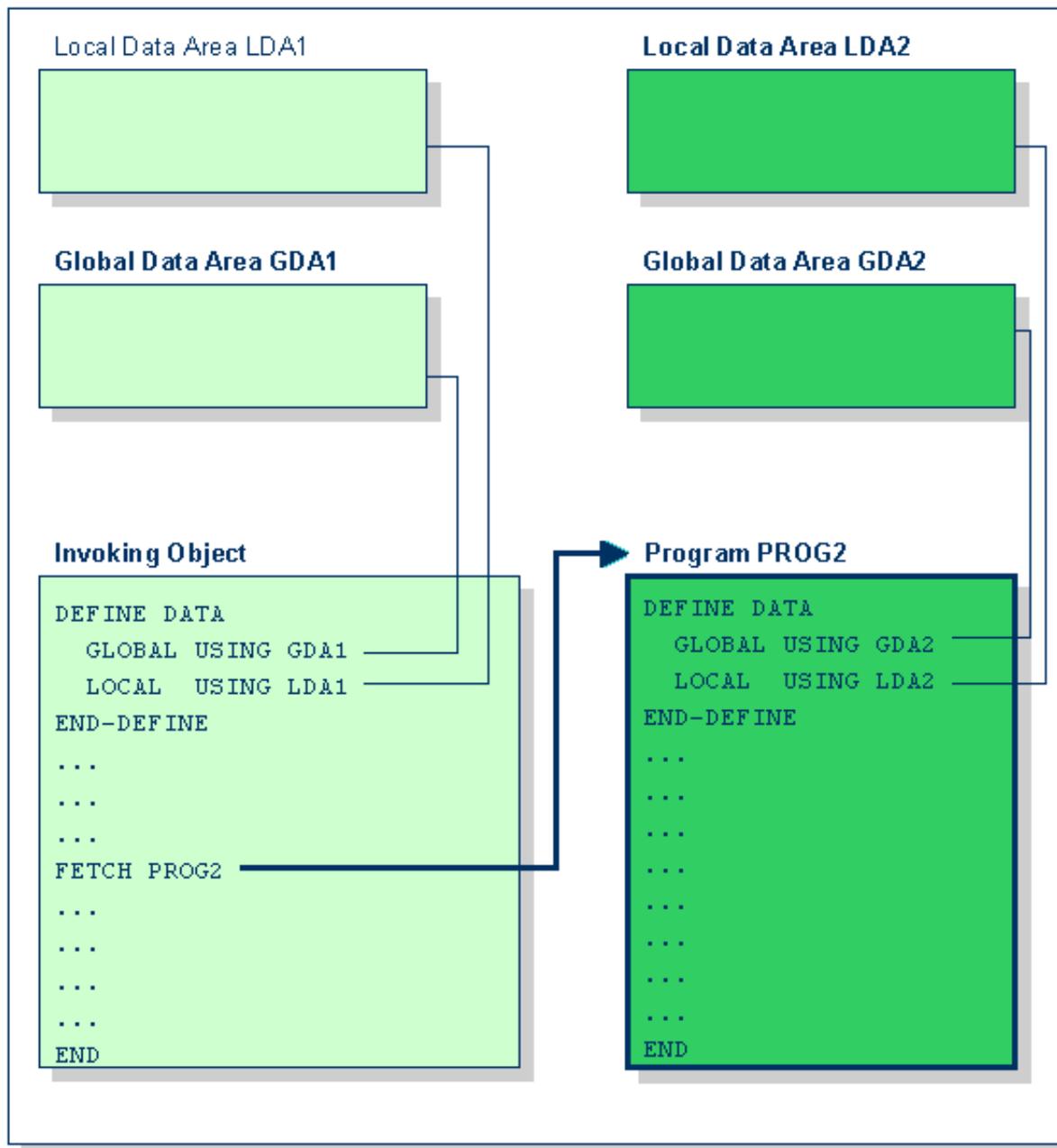
Program Invoked with FETCH RETURN:



A program invoked with `FETCH RETURN` can access the global data area used by the invoking object.

In addition, every program can have its own local data area, in which the fields that are to be used only within the program are defined.

However, a program invoked with `FETCH RETURN` cannot have its own global data area.

Program Invoked with FETCH:

A program invoked with FETCH as a main program usually establishes its own global data area (as shown in the illustration above). However, it could also use the same global data area as established by the invoking object.

Note:

A source program can also be invoked with a RUN statement; see the RUN statement in the Natural Statements documentation.

Subroutine

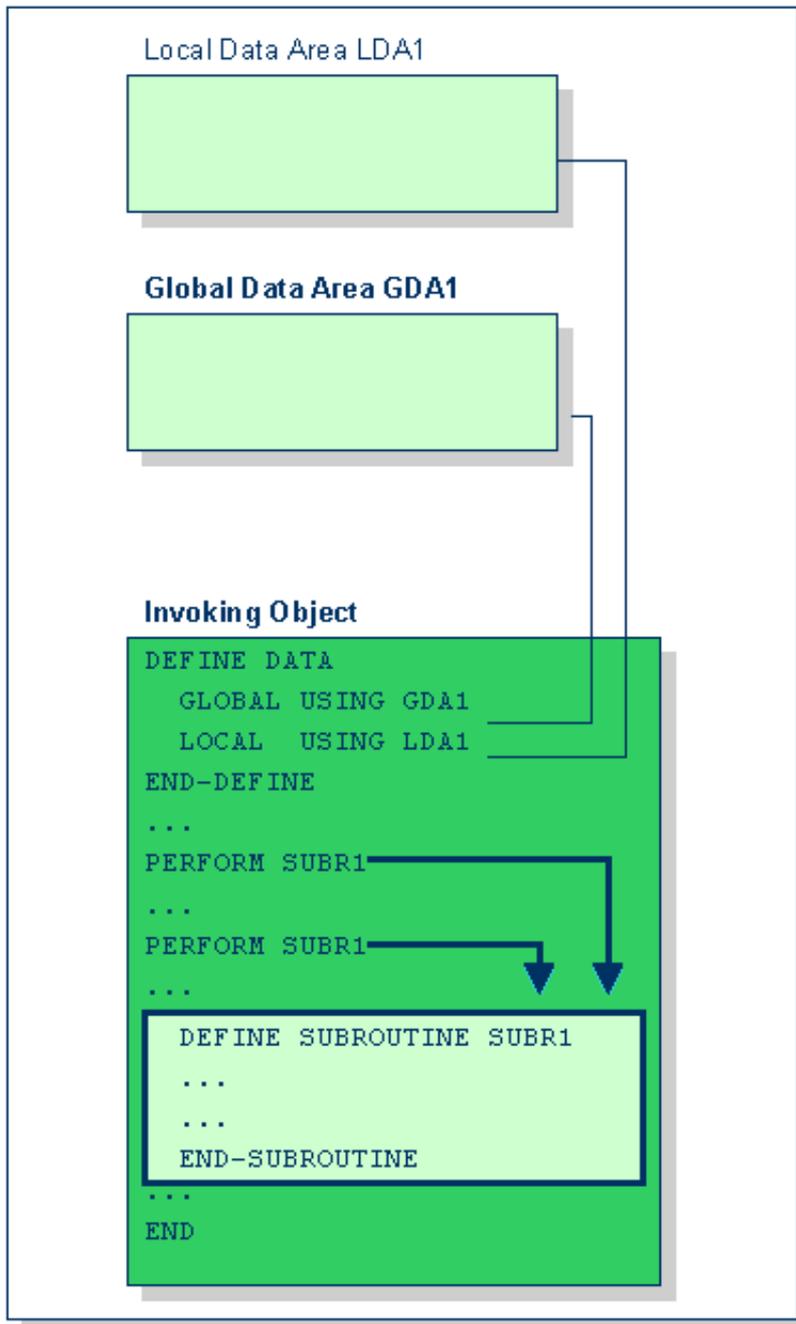
The statements that make up a subroutine must be defined within a DEFINE SUBROUTINE ... END-SUBROUTINE statement block.

A subroutine is invoked with a PERFORM statement.

A subroutine may be an *inline subroutine* or an *external subroutine*:

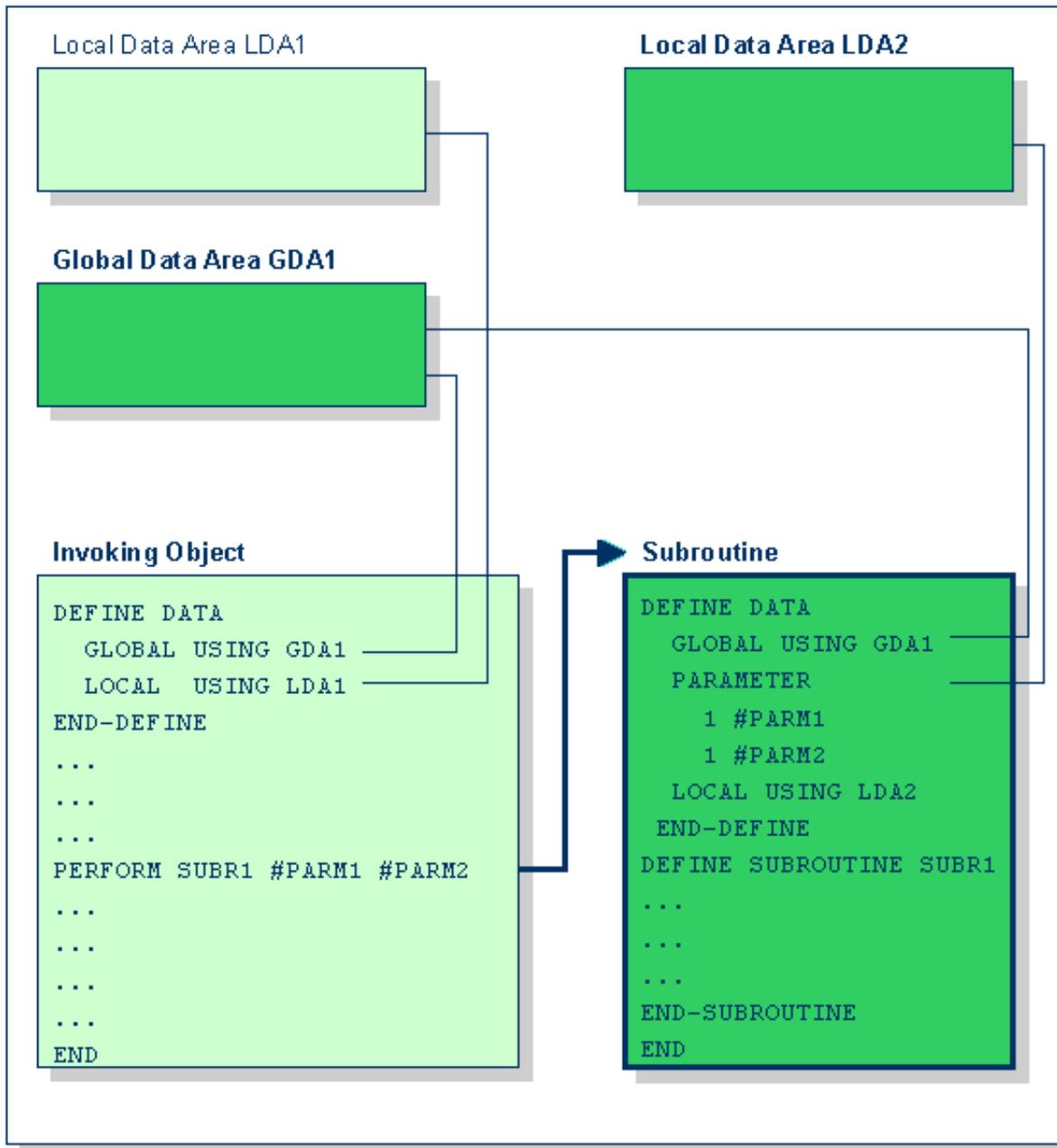
- An *inline subroutine* is defined within the object which contains the PERFORM statement that invokes it.
- An *external subroutine* is defined in a separate object - of type subroutine - outside the object which invokes it.

If you have a block of code which is to be executed several times within an object, it is useful to use an inline subroutine. You then only have to code this block once within a DEFINE SUBROUTINE statement block and invoke it with several PERFORM statements.

Inline Subroutine:

An inline subroutine can be contained within a programming object of type program, subprogram, subroutine or help routine.

If an inline subroutine is so large that it impairs the readability of the object in which it is contained, you may consider putting it into an external subroutine, so as to enhance the readability of your application.

External Subroutine:

An external subroutine - that is, an object of type subroutine - cannot be executed by itself. It must be invoked from another object. The invoking object can be a program, subprogram, subroutine or help routine.

Data Available to an Inline Subroutine

An inline subroutine has access to the local data area and the global data area used by the object in which it is contained.

Data Available to an External Subroutine

An external subroutine can access the global data area used by the invoking object.

Moreover, parameters can be passed with the `PERFORM` statement from the invoking object to the external subroutine. These parameters must be defined either in the `DEFINE DATA PARAMETER` statement of the subroutine, or in a parameter data area used by the subroutine.

In addition, an external subroutine can have its own local data area, in which the fields that are to be used only within the subroutine are defined.

However, an external subroutine cannot have its own global data area.

Subprogram

Typically, a subprogram would contain a generally available standard function that is used by various objects in an application.

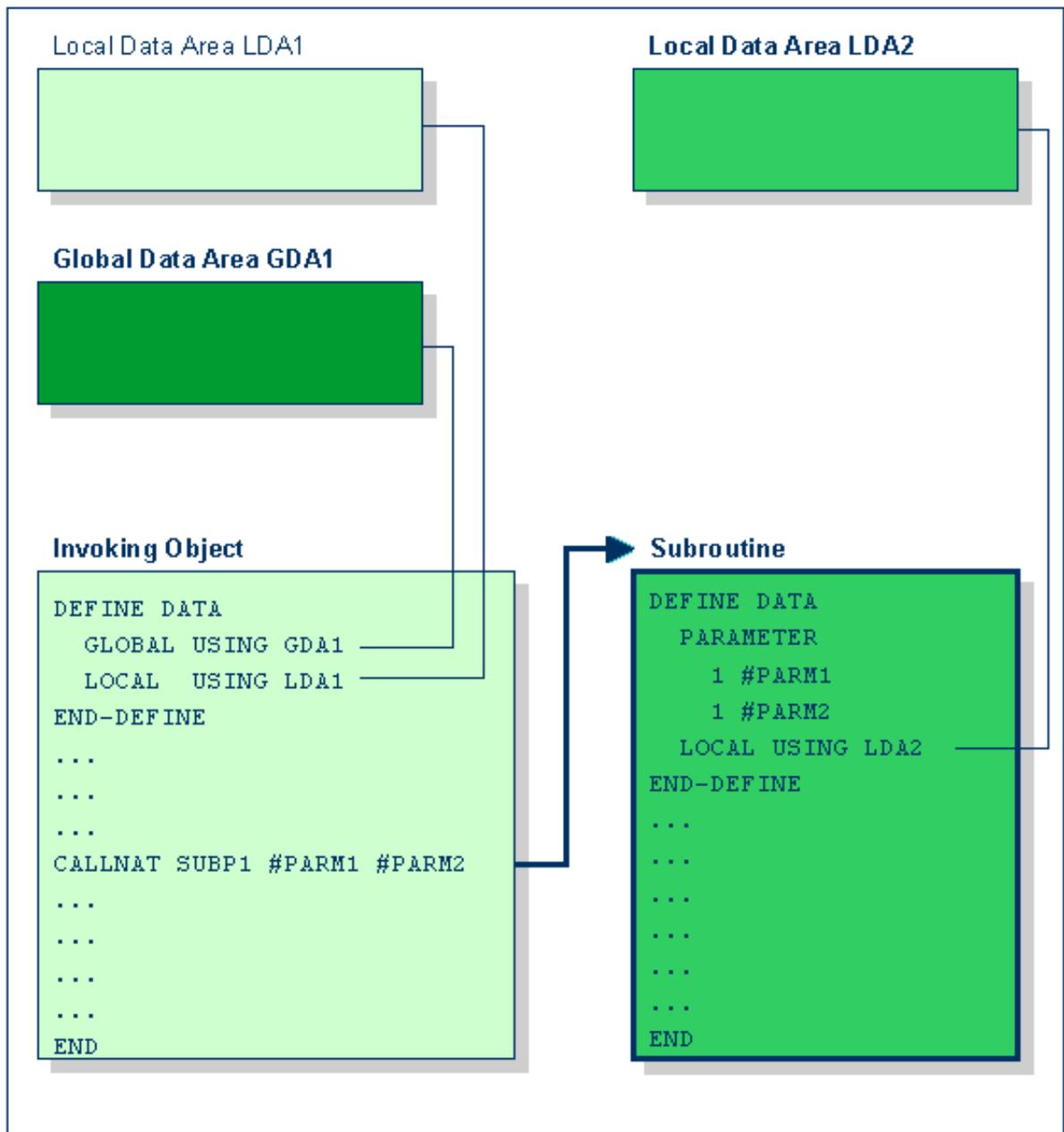
A subprogram cannot be executed by itself. It must be invoked from another object. The invoking object can be a program, subprogram, subroutine or help routine.

A subprogram is invoked with a `CALLNAT` statement.

When the `CALLNAT` statement is executed, the execution of the invoking object will be suspended and the subprogram executed. After the subprogram has been executed, the execution of the invoking object will be continued with the statement following the `CALLNAT` statement.

Data Available to a Subprogram

With the CALLNAT statement, parameters can be passed from the invoking object to the subprogram. These parameters are the only data available to the subprogram from the invoking object. They must be defined either in the DEFINE DATA PARAMETER statement of the subprogram, or in a parameter data area used by the subprogram.



In addition, a subprogram can have its own local data area, in which the fields to be used within the subprogram are defined.

If a subprogram in turn invokes a subroutine or helpoutine, it can also establish its own global data area to be shared with the subroutine/helpoutine.

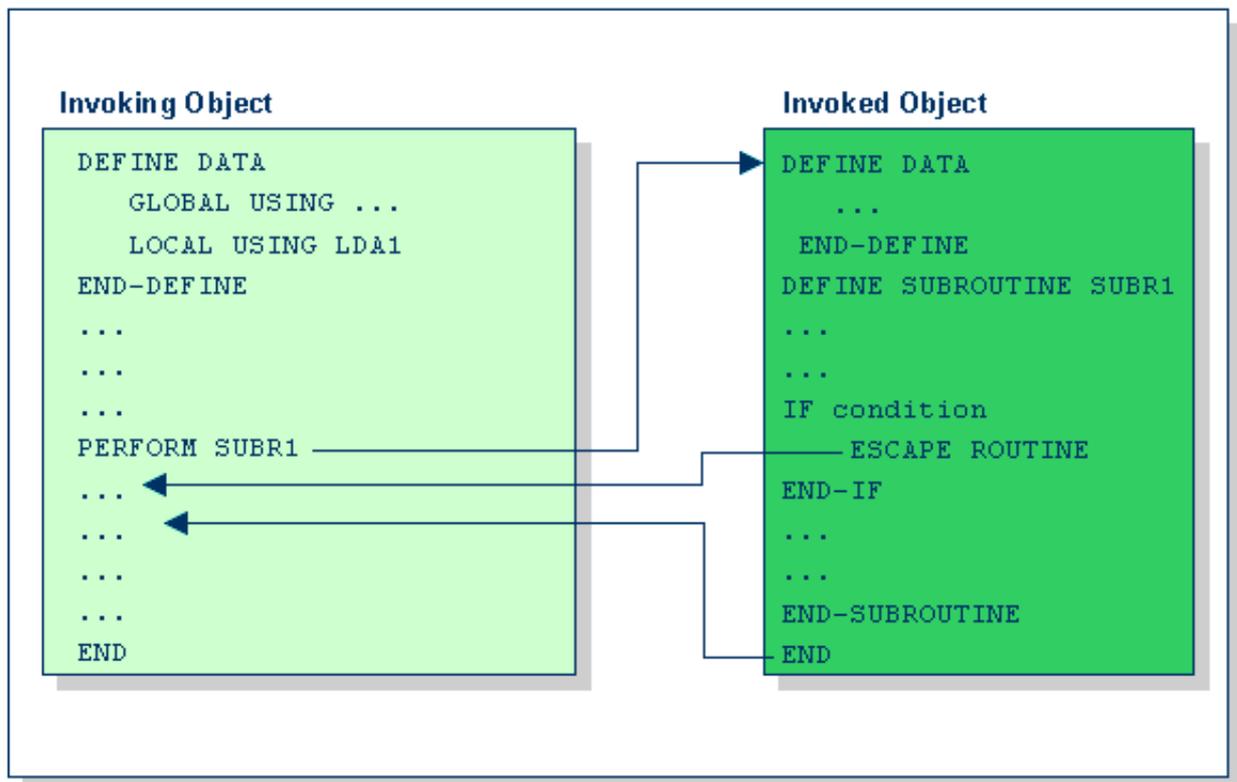
Processing Flow when Invoking a Routine

When the CALLNAT, PERFORM or FETCH RETURN statement that invokes a routine - a subprogram, an external subroutine, or a program respectively - is executed, the execution of the invoking object is suspended and the execution of the routine begins.

The execution of the routine continues until either its END statement is reached or processing of the routine is stopped by an ESCAPE ROUTINE statement being executed.

In either case, processing of the invoking object will then continue with the statement following the CALLNAT, PERFORM or FETCH RETURN statement used to invoke the routine.

Example:



Maps

Maps are those parts of an application which the users see on their screens.

The dialogue with the user is done via input maps. An *input map* is invoked with an INPUT USING MAP statement.

If an application produces any output report, this report can be displayed on the screen by using an *output map*. An output map is invoked with a WRITE USING MAP statement.

Maps are created with the map editor, which is described in your Natural User's Guide.

Processing of a map can be stopped with an ESCAPE ROUTINE statement in a processing rule.

Help maps are, in principle, like any other maps, but when they are assigned as help, additional checks are performed to ensure their usability for help purpose. Help maps are created with the map editor.

Helproutines

Helproutines have specific characteristics to facilitate the processing of help requests.

Helproutines are created with the program editor. They may be used to implement complex and interactive help systems.

The following topics are covered below:

- Invoking Help
- Specifying Helproutines
- Programming Considerations for Helproutines
- Passing Parameters to Helproutines
- Help as a Window

Invoking Help

A Natural user can invoke a Natural helproutine either by entering the help character (the default character is "?") in a field, or by pressing the help key (usually PF1).

Note 1:

- The help character must be entered only once.
- The help character must be the only character modified in the input string.
- The help character must be the first character in the input string.

Note 2:

If a helproutine is specified for a numeric field, Natural will allow a question mark to be entered for the purpose of invoking the helproutine for that field. Natural will still check that valid numeric data are provided as field input.

If not already specified, the help key may be specified with the SET KEY statement:

```
SET KEY PF1=HELP
```

A helproutine can only be invoked by a user if it has been specified in the program or map from which it is to be invoked.

Specifying Helproutines

A helproutine may be specified:

- in a program: at statement level and at field level;
- in a map: at map level and at field level.

If a user requests help for a field for which no help has been specified, or if a user requests help without a field being referenced, the helproutine specified at the statement or map level is invoked.

A helproutine may also be invoked by using a REINPUT USING HELP statement (either in the program itself or in a processing rule). If the REINPUT USING HELP statement contains a MARK option, the helproutine assigned to the MARKed field is invoked. If no field-specific helproutine is assigned, the map helproutine is invoked.

A REINPUT statement in a helproutine may only apply to INPUT statements within the same helproutine.

The name of a helproutine may be specified either with the session parameter HE of an INPUT statement:

```
INPUT (HE='HELP2112')
```

or using the extending field editing facility of the map editor (as described in your Natural User's Guide).

The name of a helproutine may be specified as an alphanumeric constant or as an alphanumeric variable containing the name. If it is a constant, the name of the helproutine must be specified within apostrophes.

Programming Considerations for Help routines

Processing of a help routine can be stopped with an ESCAPE ROUTINE statement.

Be careful when using END OF TRANSACTION or BACKOUT TRANSACTION statements in a help routine, because this will affect the transaction logic of the main program.

Passing Parameters to Help routines

A help routine can access the currently active global data area (but it cannot have its own global data area). In addition, it can have its own local data area.

Data may also be passed from/to a help routine via parameters. A help routine may have up to 20 explicit parameters and one implicit parameter. The explicit parameters are specified with the "HE" operand after the help routine name:

```
HE= 'MYHELP' , '001'
```

The implicit parameter is the field for which the help routine was invoked:

```
INPUT #A (A5) (HE='YOURHELP' , '001')
```

where "001" is an explicit parameter and "#A" is the implicit parameter/the field.

This is specified within the DEFINE DATA PARAMETER statement of the help routine as:

```
DEFINE DATA PARAMETER
  1 #PARM1 (A3)          /* explicit parameter
  1 #PARM2 (A5)          /* implicit parameter
END-DEFINE
```

Please note that the implicit parameter (#PARM2 in the above example) may be omitted. The implicit parameter is used to access the field for which help was requested, and to return data from the help routine to the field. For example, you might implement a calculator program as a help routine and have the result of the calculations returned to the field.

Note:

When help is called, the help routine is called before the data are passed from the screen to the program data areas. This means that help routines cannot access data entered within the same screen transaction.

Once help processing is completed, the screen data will be refreshed: any fields which have been modified by the help routine will be updated - excluding fields which had been modified by the user before the help routine was invoked, but including the field for which help was requested.

(Exception: If the field for which help was requested is split into several parts by dynamic attributes (DY parameter), and the part in which the question mark is entered is *after* a part modified by the user, the field content will not be modified by the help routine.)

Note:

Control variables are not evaluated again after the processing of the help routine, even if they have been modified within the help routine.

Equal Sign Option

The equal sign (=) may be specified as an explicit parameter:

```
INPUT PERSONNEL-NUMBER (HE='HELPROUT',=)
```

This parameter is processed as an internal field (A65) which contains the field name (or map name if specified at map level). The corresponding helproutine starts with:

```
DEFINE DATA PARAMETER
  1 FNAME (A65)           /* contains 'PERSONNEL-NUMBER'
  1 FVALUE (N8)          /* value of field (optional)
END-DEFINE
```

This option may be used to access one common helproutine which reads the field name and provides field-specific help by accessing the application online documentation or the Predict data dictionary.

Array Indices

If the field selected by the help character or the help key is an array element, its indices are supplied as implicit parameters (1 - 3 depending on rank, regardless of the explicit parameters). The format/length of these parameters is I2.

```
INPUT A(*,*) (HE='HELPROUT',=)
```

The corresponding helproutine starts with:

```
DEFINE DATA PARAMETER
  1 FNAME (A65)           /* contains 'A'
  1 FVALUE (N8)          /* value of selected element
  1 FINDEX1 (I2)         /* 1st dimension index
  1 FINDEX2 (I2)         /* 2nd dimension index
END-DEFINE
...

```

Help as a Window

The size of a help to be displayed may be smaller than the screen size. In this case, the help appears on the screen as a window, enclosed by a frame:

```

*****
                                PERSONNEL INFORMATION
*****
PLEASE ENTER NAME: ? _____
PLEASE ENTER CITY:  _____
TYPE IN . TO STOP  !-----+
                   !
                   ! Type in the name of an   !
                   ! employee in the first   !
                   ! field and press ENTER.  !
                   ! You will then receive  !
                   ! a list of all employees !
                   ! of that name.          !
                   !                         !
                   ! For a list of employees !
                   ! of a certain name who   !
                   ! live in a certain city, !
                   ! type in a name in the  !
                   ! first field and a city  !
                   ! in the second field   !
                   ! and press ENTER.      !
***** !                               ! *****
                   !-----+

```

Within a helproutine, the size of the window may be specified as follows:

- by a FORMAT statement (for example, FORMAT PS=15 LS=30);
- by an INPUT USING MAP statement; in this case, the size defined for the map (in its map settings) is used;
- by a DEFINE WINDOW statement; this statement allows you to either explicitly define a window size or leave it to Natural to automatically determine the size of the window depending on its contents.

The position of a help window is computed automatically from the position of the field for which help was requested. Natural places the window as close as possible to the corresponding field without overlaying the field. With the DEFINE WINDOW statement, you may bypass the automatic positioning and determine the window position yourself.

For further information on window processing, please refer to the DEFINE WINDOW statement in the Natural Statements documentation and the terminal command %W in the Natural Reference documentation.

Multiple Use of Source Code - Copycode

Copycode is a portion of source code which can be included in another object via an INCLUDE statement.

So, if you have a statement block which is to appear in identical form in several objects, you may use copycode instead of coding the statement block several times. This reduces the coding effort and also ensures that the blocks are really identical.

The copycode is included at compilation; that is, the source-code lines from the copycode are not physically inserted into the object that contains the INCLUDE statement, but they will be included in the compilation process and are thus part of the resulting object module.

Consequently, when you modify the source code of copycode, you also have to newly compile (STOW) all objects which use that copycode.

Copycode cannot be executed on its own. It cannot be STOWed, but only SAVED.

For further information on copycode, please refer to the description of the INCLUDE statement in the Natural Statements documentation.

Documenting Natural Objects - Text

The Natural object type "text" is used to write text rather than programs. You can write any text you wish (there is no syntax check). You can use this type of object to document Natural objects in more detail than you can, for example, within the source code of a program. "Text" objects may also be useful at sites where Predict is not available for program documentation purposes.

You write the text using the Natural program editor. The only difference in handling as opposed to writing programs, is that the text you write stays as it is, that is, there is no lower to upper case translation or empty line suppression (provided in your editor profile Empty Line Suppression is set to "N" and Editing in Lower Case is set to "Y", see your Natural User's Guide for Windows for more details).

"Text" objects can only be SAVED, they cannot be STOWed. They cannot be RUN, only displayed in the editor.

Creating Event Driven Applications - Dialog

Dialogs are used in conjunction with event-driven programming when creating Natural applications for graphical user interfaces (GUIs).

For information on dialogs and event-driven programming, please refer to the Natural User's Guide for Windows.

Creating Component Based Applications - Class

Classes are used in conjunction with NaturalX when creating component based applications to be used in a client/server environment.

For information on classes, please refer to the NaturalX documentation.

Using Non-Natural Files - Resource

Resources are only available with Natural under Windows 98 and Windows NT/2000.

Natural distinguishes two kinds of resources:

- **Shared Resources**
A shared resource is any non-Natural file that is used in a Natural application and is maintained in the Natural library system.
- **Private Resources**
A private resource is a file that is assigned to one and only one Natural object and is considered to be part of that object. An object can have at most one private resource file. At the moment, only Natural dialogs have private resources.

Both shared and private resources belonging to a Natural library are maintained in a subdirectory named `..\RES` in the directory that represents the Natural library in the file system.

Shared Resources

A shared resource is any non-Natural file that is used in a Natural application and is maintained in the Natural library system. A non-Natural file that is to be used as a shared resource must be contained in the subdirectory named `..\RES` of a Natural library.

Example - Using a shared resource:

The bitmap `MYPICTURE.BMP` is to be displayed in a Bitmap control in a dialog `MYDLG`, contained in a library `MYLIB`. First the bitmap is put into the Natural library `MYLIB` by moving it into the directory `..\MYLIB\RES`. The following code snippet from the dialog `MYDLG` shows how it is then assigned to the Bitmap control:

```
DEFINE DATA LOCAL
01 #BM-1 HANDLE OF BITMAP
...
END-DEFINE
* (Creation of the Bitmap control omitted.)
...
#BM-1.BITMAP-FILE-NAME := "MYPICTURE.BMP" ...
```

The advantages of using the bitmap as a shared resource are:

- The file name can be specified in the Natural dialog without a path name.
- The file can be kept in a Natural library together with the Natural object that uses it.

Note:

In previous Natural versions non-Natural files were usually kept in a directory that was defined with the environment variable `NATGUI_BMP`. Existing applications that use this approach will work in the same way as before, because Natural always searches for a shared resource file in this directory, if it was not found in the current library.

Private Resources

Private resources are used internally by Natural to store binary data that is part of Natural objects. These files are recognized by the file name extension `NR*`, where `*` is a character that depends on the type of the Natural object. Natural maintains private resource files and their contents automatically. A Natural object can have a maximum of one private resource file. Currently, only Natural dialogs have a private resource file. This file is used to store the configuration of ActiveX controls that are defined in a dialog and are configured with their own property pages. See [ActiveX Control Property Pages](#) on how to configure an ActiveX control.

Example - Private resources:

The name of the private resource file of the dialog `MYDLG` is `MYDLG.NR3`. Natural creates, modifies and deletes this file automatically as needed, when the dialog is created, modified, deleted etc. The private resource file is used to store binary data related to the dialog `MYDLG`.