Data Areas

This chapter covers the following topics:

- Use of Data Areas
- Local Data Area
- Global Data Area
- Parameter Data Area

Use of Data Areas

As explained in *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.

A separate data area is a Natural object that can be used by multiple Natural programs, subprograms, subroutines, helproutines or classes. A data area contains data element definitions, such as user-defined variables, constants and database fields from a data definition module (DDM).

All data areas are created and edited with the data area editor.

Natural supports three types of data area:

- Local Data Area
- Global Data Area
- Parameter Data Area

Local Data Area

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

- Define local data within a program.
- Define local data outside a program in a separate Natural programming object, a local data area (LDA).

  Such a local data area is initialized when a program, subprogram or external subroutine that uses this local data area starts to execute.

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

Example 1 - Fields Defined Directly within a DEFINE DATA Statement:

In the following example, the fields are defined directly within the DEFINE DATA statement of the program.

```
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:

In the following example, the same fields are not defined in the DEFINE DATA statement of the program, but in an LDA, named LDA39, and the DEFINE DATA statement in the program contains only a reference to that data area.

Program:

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

Local Data Area LDA39:

```
I T L  Name                            F Length     Miscellaneous
All -- ------------------------------- - ---------- ------------------------>
  V  1 VIEWEMP                                      EMPLOYEES
     2 PERSONNEL-ID                    A          8
     2 FIRST-NAME                      A         20
     2 NAME                            A         20
     1 #VARI-A                         A         20
     1 #VARI-B                         N        3.2
     1 #VARI-C                         I          4
```

Global Data Area

The following topics are covered below:

- Creating and Referencing a GDA
- Creating and Deleting GDA Instances
- Data Blocks

Creating and Referencing a GDA

GDAs are created and modified with the Natural data area editor. For further information, refer to *Data Area Editor* in the *Editors* documentation.

A GDA that is referenced by a Natural programming object must be stored in the same Natural library (or a steplib defined for this library) where the object that references this GDA is stored.

Note:
Using a GDA named COMMON for startup:
If a GDA named COMMON exists in a library, the program named ACOMMON is invoked automatically when you LOGON to that library.

Important:
When you build an application where multiple Natural programming objects reference a GDA, remember that modifications to the data element definitions in the GDA affect all Natural programming objects that reference that data area. Therefore these objects must be recompiled by using the CATALOG or STOW command after the GDA has been modified.

To use a GDA, a Natural programming object must reference it with the GLOBAL clause of the DEFINE DATA statement. Each Natural programming object can reference only one GDA; that is, a DEFINE DATA statement must not contain more than one GLOBAL clause.

Creating and Deleting GDA Instances

The first instance of a GDA is created and initialized at runtime when the first Natural programming object that references it starts to execute.

Once a GDA instance has been created, the data values it contains can be shared by all Natural programming objects that reference this GDA (DEFINE DATA GLOBAL statement) and that are invoked by a PERFORM, INPUT or FETCH statement. All objects that share a GDA instance are operating on the same data elements.

A new GDA instance is created if the following applies:

- A subprogram that references a GDA (*any* GDA) is invoked with a CALLNAT statement.
- A subprogram that does *not* reference a GDA invokes a programming object that references a GDA (*any* GDA).

If a new instance of a GDA is created, the current GDA instance is suspended and the data values it contains are stacked. The subprogram then references the data values in the newly created GDA instance. The data values in the suspended GDA instance or instances is inaccessible. A programming object only refers to one GDA instance and cannot access any previous GDA instances. A GDA data element can only be passed to a subprogram by defining the element as a parameter in the CALLNAT statement.

When the subprogram returns to the invoking programming object, the GDA instance it references is deleted and the GDA instance suspended previously is resumed with its data values.

A GDA instance and its contents is deleted if any of the following applies:

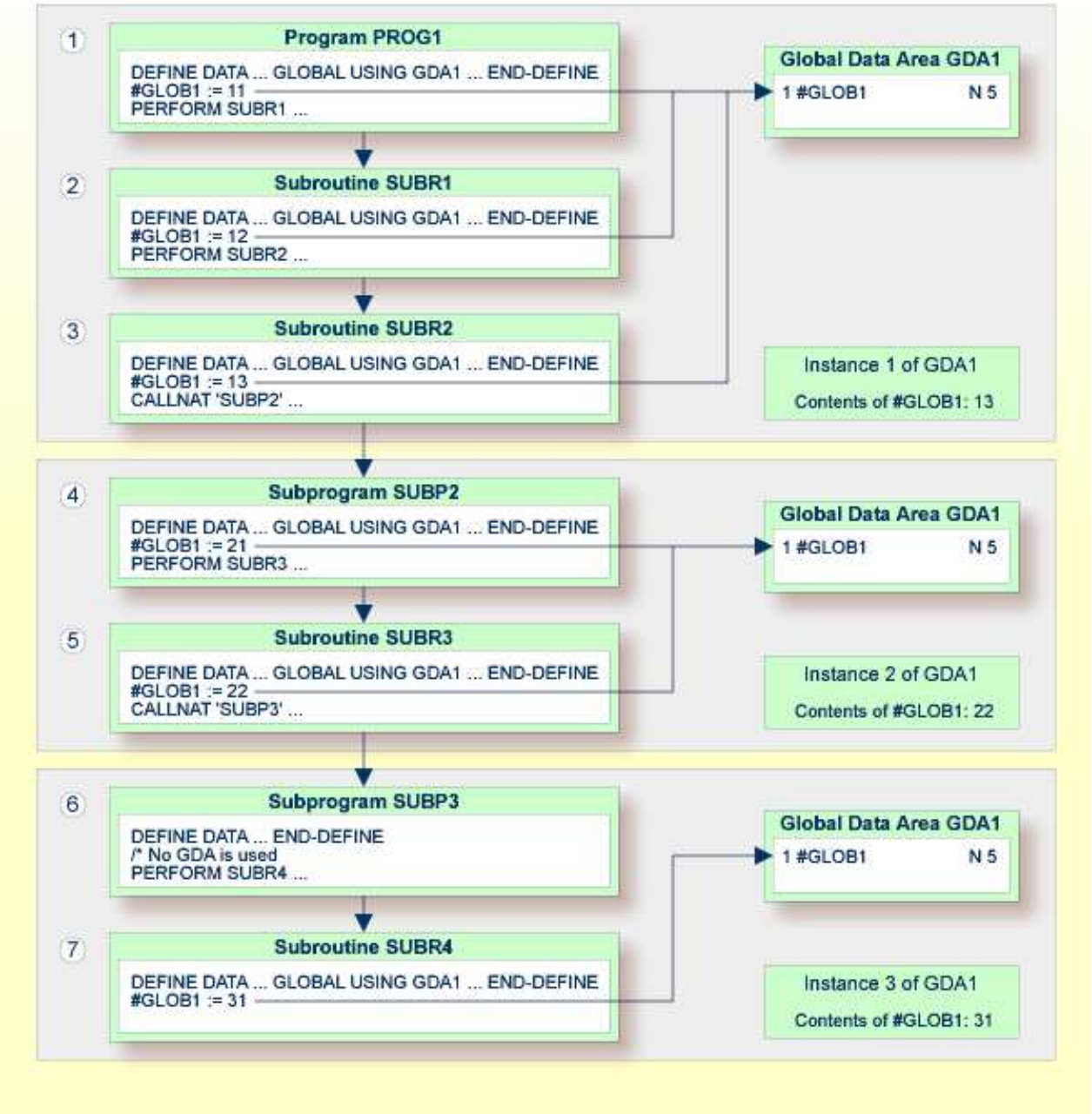
- The next LOGON is performed.
- Another GDA is referenced on the same level (levels are described later in this section).
- A RELEASE VARIABLES statement is executed. In this case, the data values in a GDA instance are reset either when a program at the level 1 finishes executing, or if the program invokes another program via a FETCH or RUN statement.

The following graphics illustrate how programming objects reference GDAs and share data elements in GDA instances.

Sharing GDA Instances

The graphic below illustrates that a subprogram referencing a GDA cannot share the data values in a GDA instance referenced by the invoking program. A subprogram that references the same GDA as the invoking program creates a new instance of this GDA. The data elements defined in a GDA that is referenced by a subprogram can, however, be shared by a subroutine or a helproutine invoked by the subprogram.

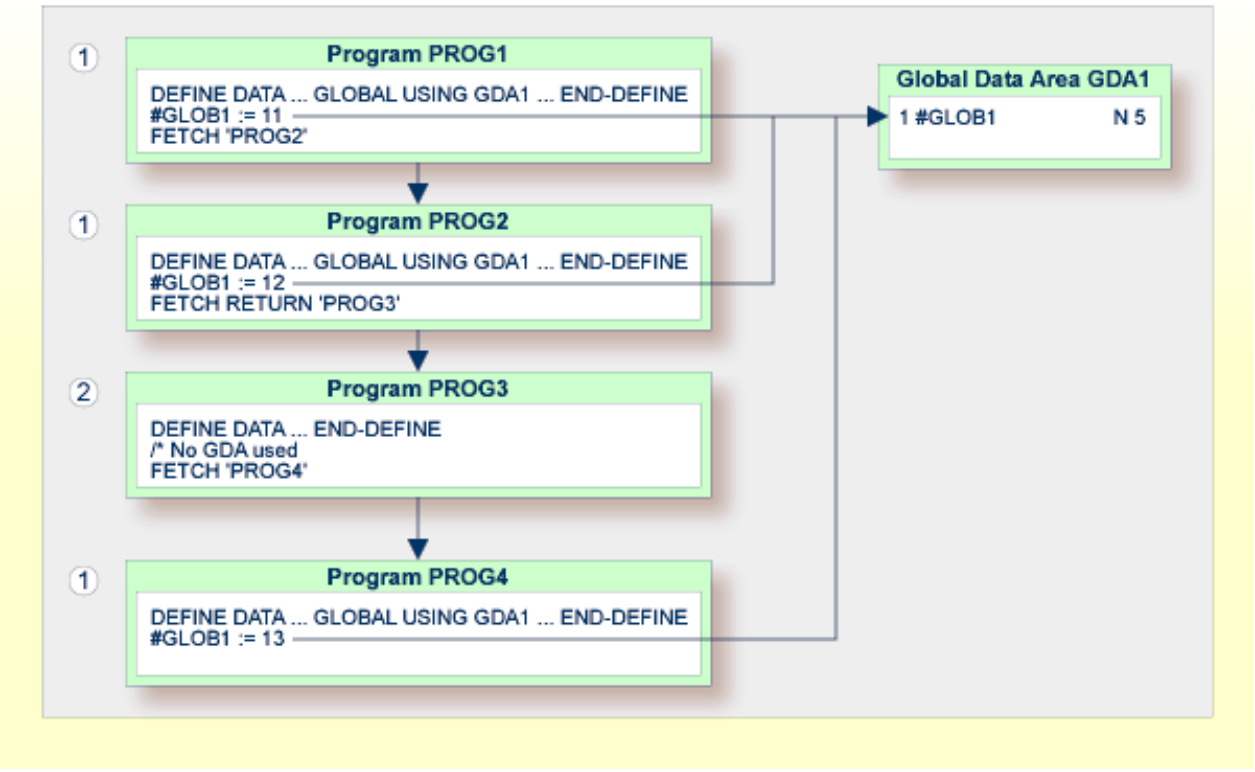
The graphic below shows three GDA instances of `GDA1` and the final values each GDA instance is assigned by the data element `#GLOB1`. The numbers (1) to (7) indicate the hierarchical levels of the programming objects.



Using FETCH or FETCH RETURN

The graphic below illustrates that programs referencing the same GDA and invoking one another with the `FETCH` or `FETCH RETURN` statement share the data elements defined in this GDA. If any of these programs does not reference a GDA, the instance of the GDA referenced previously remains active and the values of the data elements are retained.

The numbers ① and ② indicate the hierarchical levels of the programming objects.



Using FETCH with different GDAs

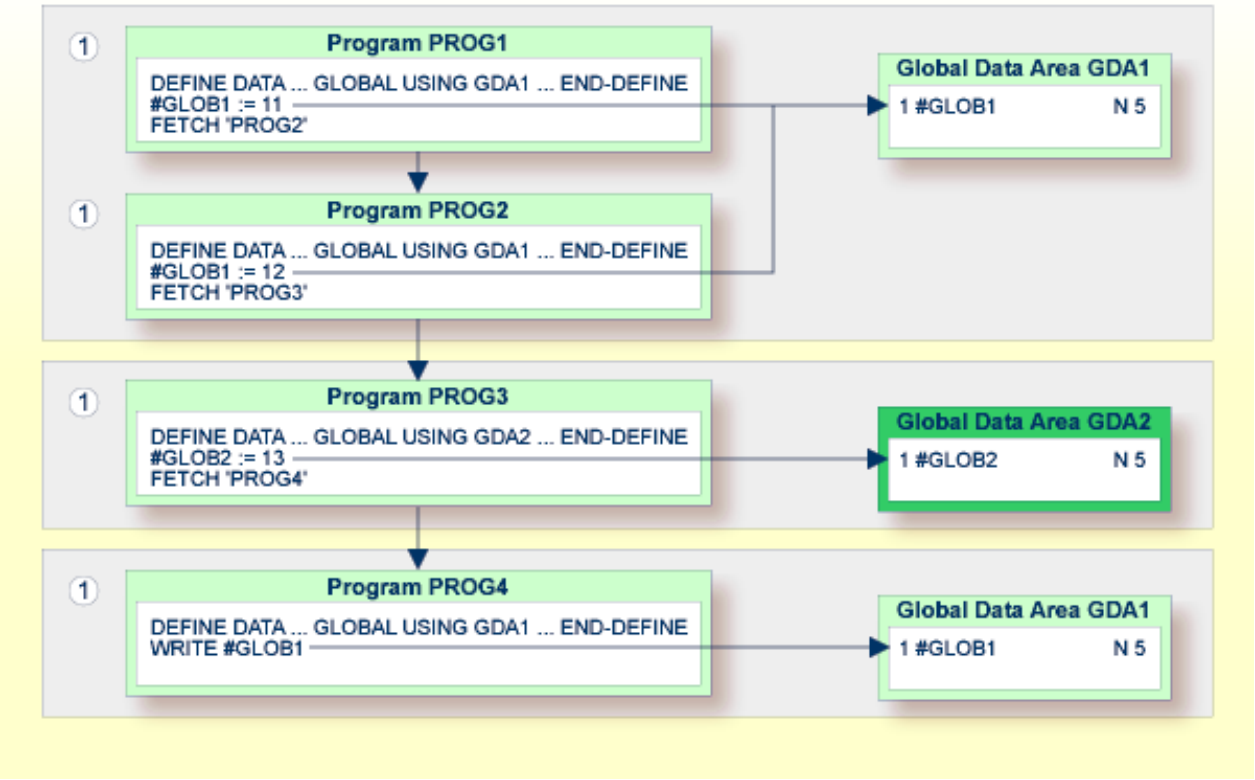
The graphic below illustrates that if a program uses the `FETCH` statement to invoke another program that references a different GDA, the current instance of the GDA (here: `GDA1`) referenced by the invoking program is deleted. If this GDA is then referenced again by another program, a new instance of this GDA is created where all data elements have their initial values.

You cannot use the `FETCH RETURN` statement to invoke another program that references a different GDA.

The number (1) indicates the hierarchical level of the programming objects.

The invoking programs `PROG3` and `PROG4` affect the GDA instances as follows:

- The statement `GLOBAL USING GDA2` in `PROG3` creates an instance of `GDA2` and deletes the current instance of `GDA1`.
- The statement `GLOBAL USING GDA1` in `PROG4` deletes the current instance of `GDA2` and creates a new instance of `GDA1`. As a result, the `WRITE` statement displays the value zero (0).



Data Blocks

To save data storage space, you can create a GDA with data blocks.

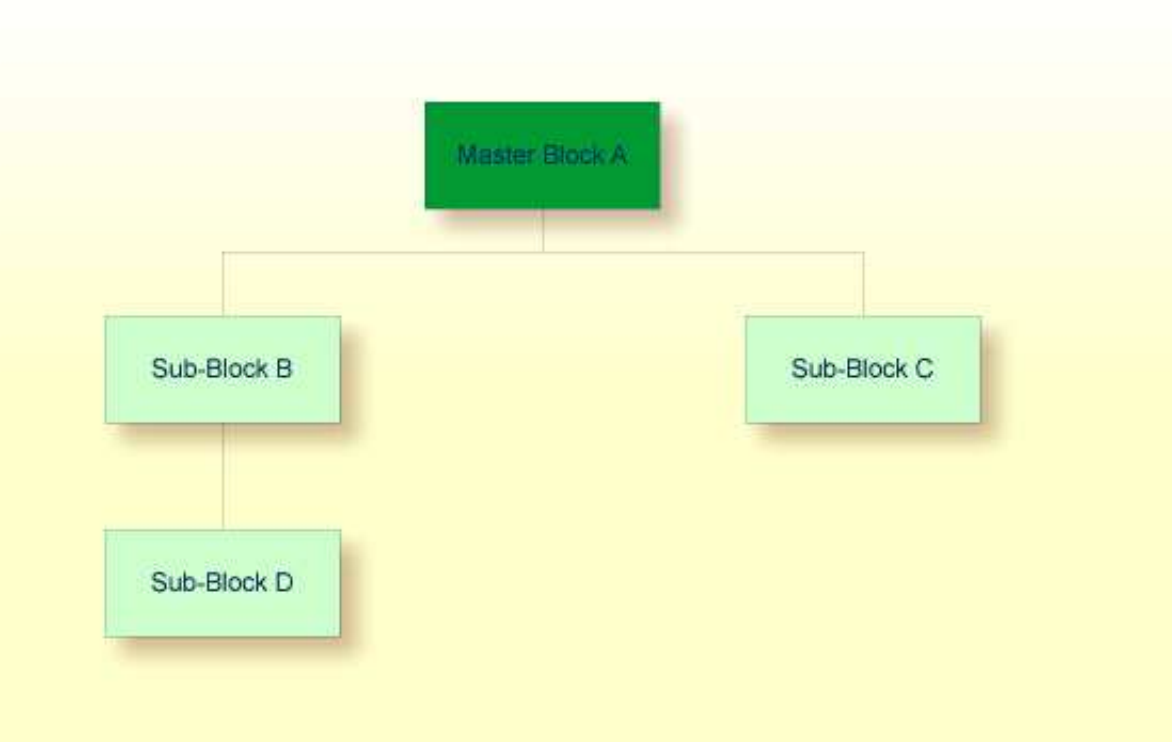
The following topics are covered below:

- Example of Data Block Usage
- Defining Data Blocks
- Block Hierarchies

Example of Data Block Usage

Data blocks can overlay each other during program execution, thereby saving storage space.

For example, given the following hierarchy, Blocks B and C would be assigned the same storage area. Thus it would not be possible for Blocks B and C to be in use at the same time. Modifying Block B would result in destroying the contents of Block C.



Defining Data Blocks

You define data blocks in the data area editor. You establish the block hierarchy by specifying which block is subordinate to which: you do this by entering the name of the "parent" block in the comment field of the block definition.

In the following example, `SUB-BLOCKB` and `SUB-BLOCKC` are subordinate to `MASTER-BLOCKA`; `SUB-BLOCKD` is subordinate to `SUB-BLOCKB`.

The maximum number of block levels is 8 (including the master block).

Example:

Global Data Area `G-BLOCK`:

```
I T L Name                            F Leng Index/Init/EM/Name/Comment
- - - -------------------------------- - ---- --------------------------------
  B   MASTER-BLOCKA
    1 MB-DATA01                       A   10
  B   SUB-BLOCKB                          MASTER-BLOCKA
    1 SBB-DATA01                      A   20
  B   SUB-BLOCKC                          MASTER-BLOCKA
    1 SBC-DATA01                      A   40
  B   SUB-BLOCKD                          SUB-BLOCKB
    1 SBD-DATA01                      A   40
```

To make the specific blocks available to a program, you use the following syntax in the `DEFINE DATA` statement:

Program 1:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA
END-DEFINE
```

Program 2:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
```

Program 3:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKC
END-DEFINE
```

Program 4:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB.SUB-BLOCKD
END-DEFINE
```

With this structure, Program 1 can share the data in MASTER-BLOCKA with Program 2, Program 3 or Program 4. However, Programs 2 and 3 cannot share the data areas of SUB-BLOCKB and SUB-BLOCKC because these data blocks are defined at the same level of the structure and thus occupy the same storage area.

Block Hierarchies

Care needs to be taken when using data block hierarchies. Let us assume the following scenario with three programs using a data block hierarchy:

Program 1:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
MOVE 1234 TO SBB-DATA01
FETCH 'PROGRAM2'
END
```

Program 2:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA
END-DEFINE
*
FETCH 'PROGRAM3'
END
```

Program 3:

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
WRITE SBB-DATA01
END
```

Explanation:

- Program 1 uses the global data area G-BLOCK with MASTER-BLOCKA and SUB-BLOCKB. The program modifies a field in SUB-BLOCKB and fetches Program 2 which specifies only MASTER-BLOCKA in its data definition.
- Program 2 resets (deletes the contents of) SUB-BLOCKB. The reason is that a program on Level 1 (for example, a program called with a FETCH statement) resets any data blocks that are subordinate to the blocks it defines in its own data definition.
- Program 2 now fetches Program 3 which is to display the field modified in Program 1, but it returns an empty screen.

For details on program levels, see *Multiple Levels of Invoked Objects*.

Parameter Data Area

A subprogram is invoked with a CALLNAT statement. With the CALLNAT statement, parameters can be passed from the invoking object to the subprogram.

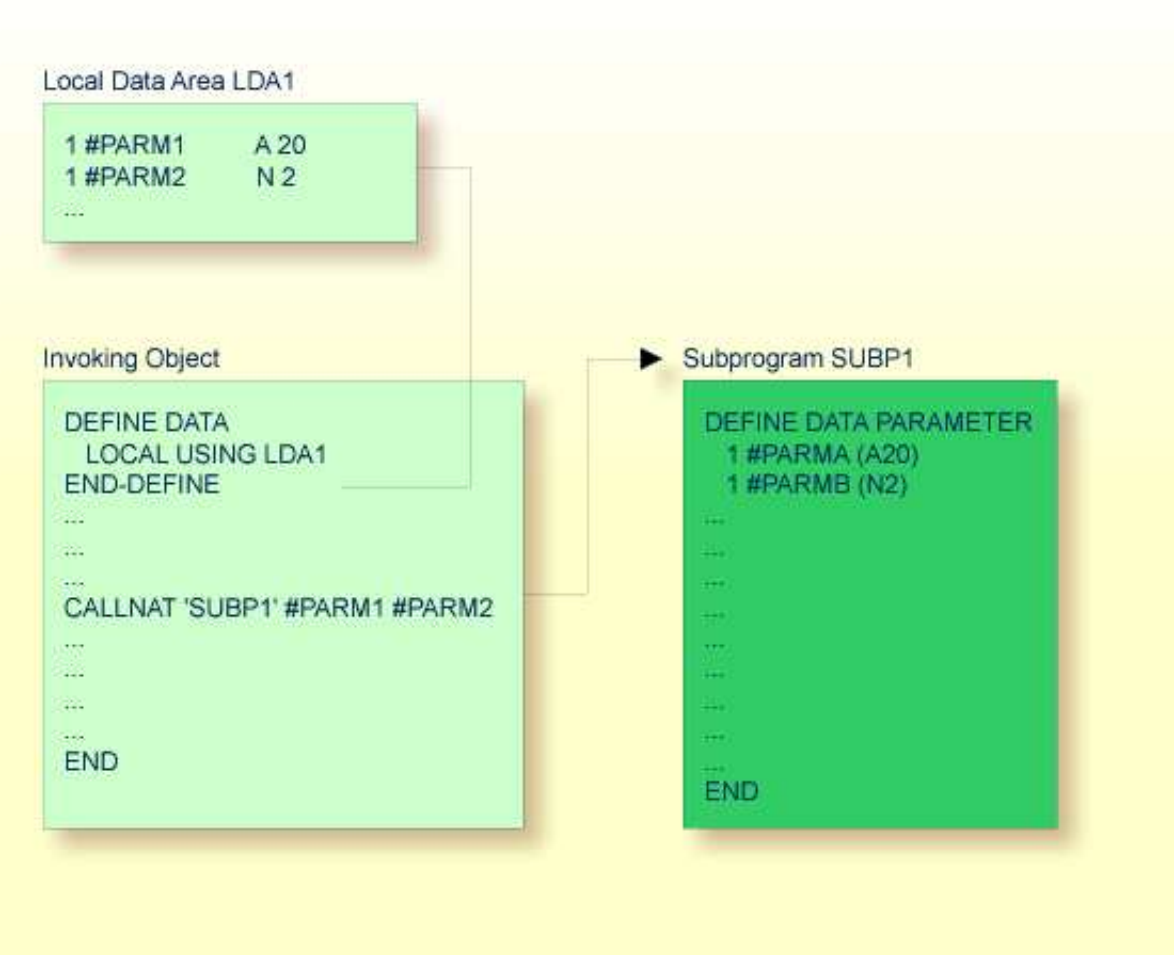
These parameters must be defined with a DEFINE DATA PARAMETER statement in the subprogram:

- 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 PDA.

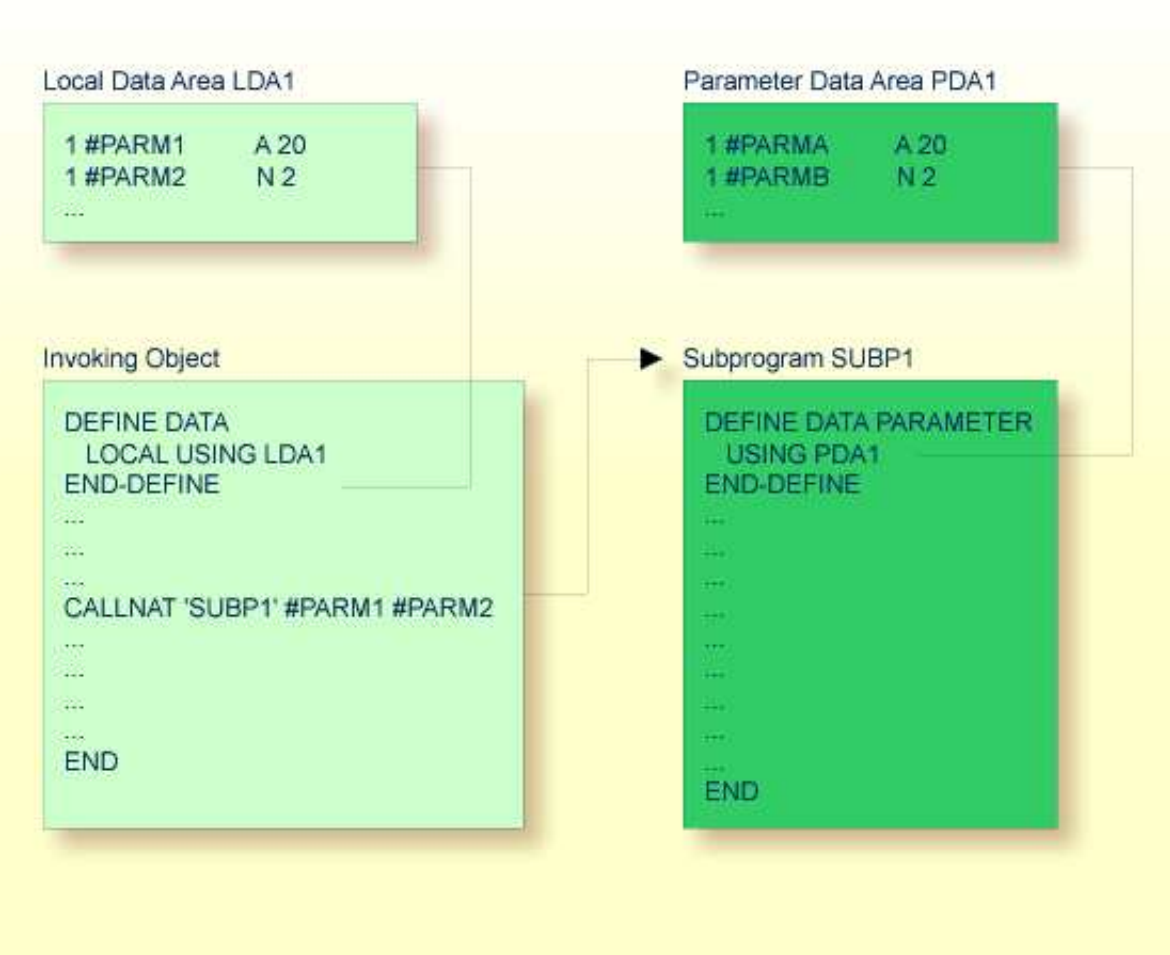
The following topics are covered below:

- Parameters Defined within DEFINE DATA PARAMETER Statement
- Parameters Defined in Parameter Data Area

Parameters Defined within DEFINE DATA PARAMETER Statement



Parameters 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 PDA; in the illustrations above, they are defined in the LDA used by the invoking object (but they could also be defined in a GDA).

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).

To guarantee that the data element definitions used in the invoking program are identical to the data element definitions used in the subprogram or external subroutine, you can specify a PDA in a `DEFINE DATA LOCAL USING` statement. By using a PDA as an LDA you can avoid the extra effort of creating an LDA that has the same structure as the PDA.