Programs, Functions, Subprograms and Subroutines

This document discusses those object types which can be invoked as routines; that is, as subordinate programs.

Helproutines and maps, although they are also invoked from other objects, are strictly speaking not routines as such, and are therefore discussed in separate documents; see *Helproutines* and *Maps*.

This chapter covers the following topics:

- A Modular Application Structure
- Multiple Levels of Invoked Objects
- Program
- Function
- 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, helproutines 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.

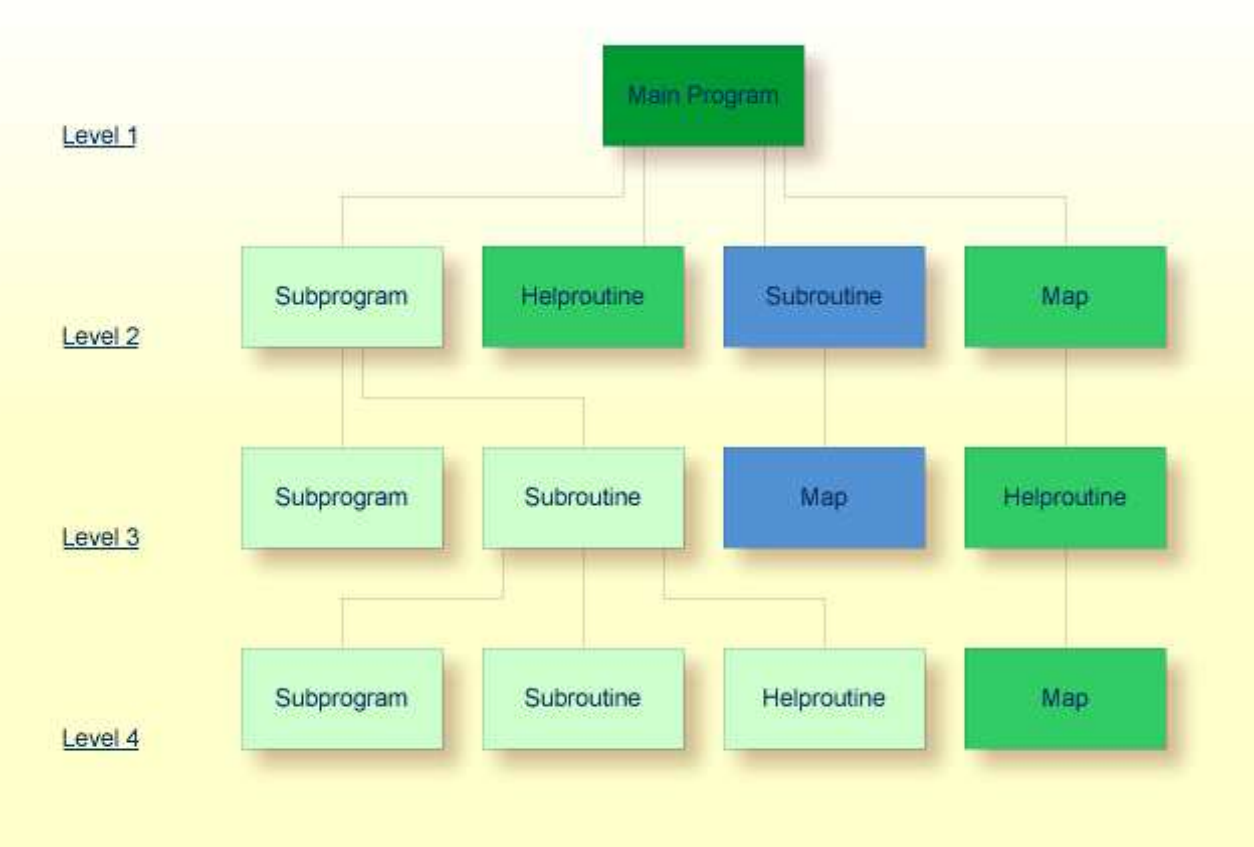
Multiple Levels of Invoked Objects

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 *System Variables* documentation).

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

- program
- function
- subroutine
- subprogram

Helproutines and maps, although they are also invoked from other objects, are strictly speaking not routines as such, and are therefore discussed in separate documents; see *Helproutines* and *Maps*.

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 another program, a subprogram, function, subroutine or helproutine.

- 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 `FETCH`ed 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 `FETCH`ed program will be activated as a *main program*. The invoking object will not be re-activated upon termination of the fetched program.

The following topics are covered below:

- Program Invoked with FETCH RETURN
- Program Invoked with FETCH

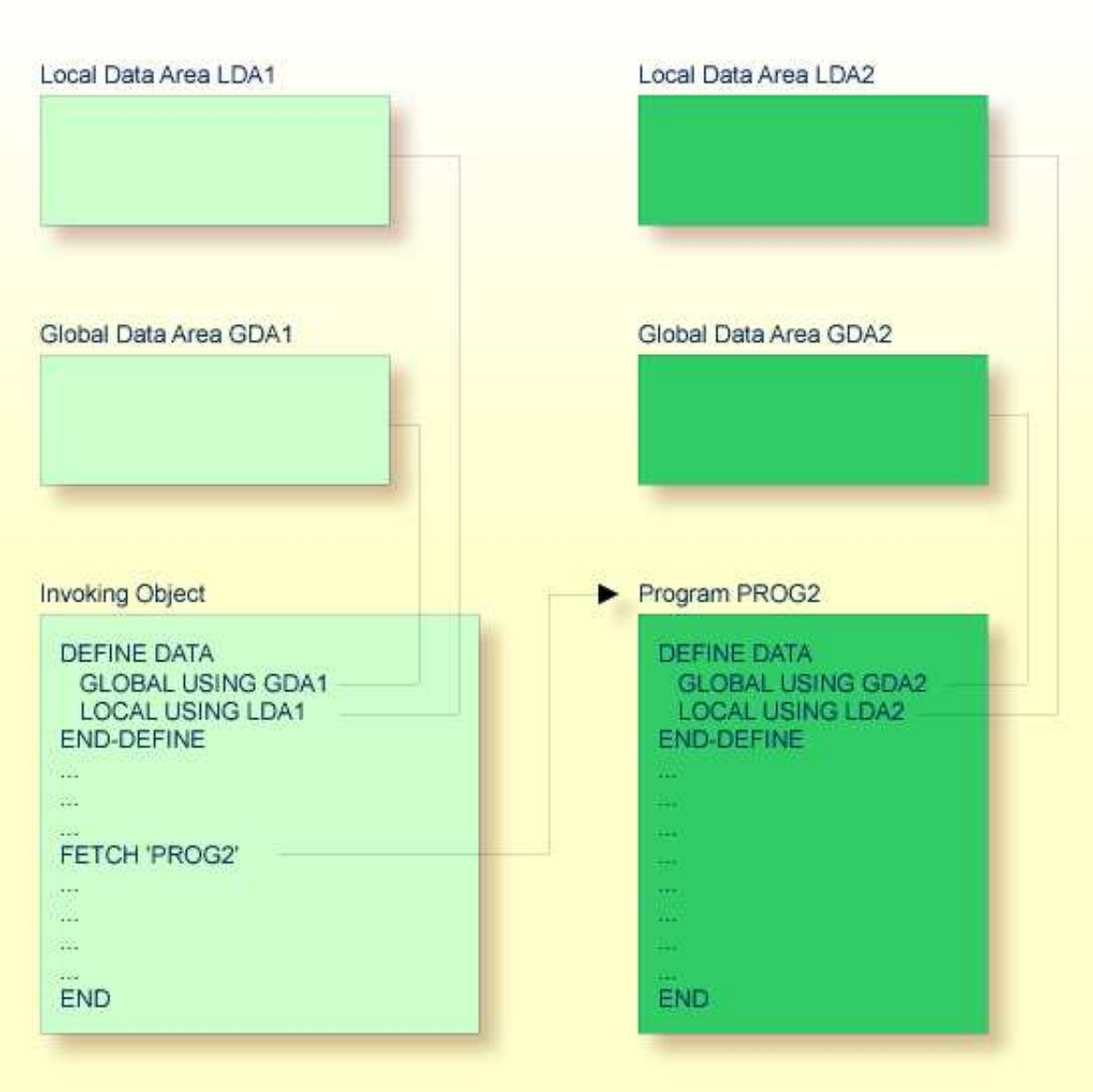
Program Invoked with FETCH RETURN

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 *Statements* documentation.

Function

An object of type "function" contains the definitions of a single function and may be structured as shown in the following code example:

```
DEFINE FUNCTION
   ...
   DEFINE SUBROUTINE
   ...
   END-SUBROUTINE
   ...
END-FUNCTION
```

The block of statements between `DEFINE FUNCTION` and `END-FUNCTION` must contain all those statements which are to be executed when the function is called.

Internal subroutines are allowed to be defined inside a function definition.

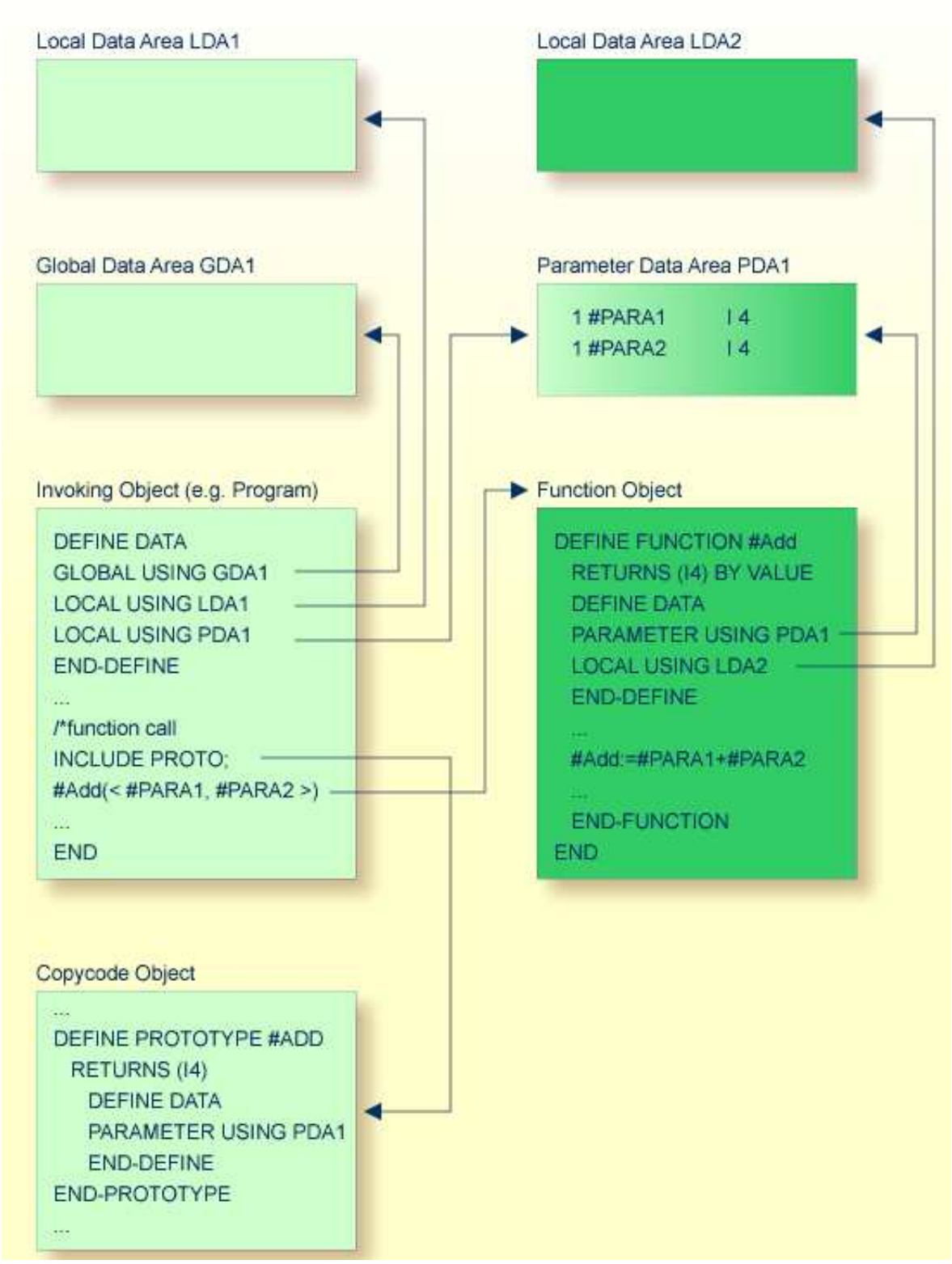
A function is invoked using the Function Call syntax.

If you have a block of code which is to be executed several times within the 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.

The global data area of the invoking object (for example, `GDA1`) cannot be referenced inside the function definition. Also, objects which will be invoked by a function cannot reference the global data area (GDA) of the object (`GDA1`) invoking the function, because entering a function causes a new global data area to be created by the runtime environment.

Parameter data areas (for example, `PDA1`) may be used to access parameters for function calls and function definitions in order to minimize the maintainance effort when changing parameters.

The copycode object containing the prototype definition is used at compilation time only in order to determine the type of the return variable for function call reference and to check the parameters, if this is desired.


Local Data Area LDA1
Local Data Area LDA2
Global Data Area GDA1
Parameter Data Area PDA1
1 #PARA1 I 4
1 #PARA2 I 4
Invoking Object (e.g. Program)
DEFINE DATA
GLOBAL USING GDA1
LOCAL USING LDA1
LOCAL USING PDA1
END-DEFINE
...
/*function call
INCLUDE PROTO;
#Add(< #PARA1, #PARA2 >)
...
END
Function Object
DEFINE FUNCTION #Add
RETURNS (I4) BY VALUE
DEFINE DATA
PARAMETER USING PDA1
LOCAL USING LDA2
END-DEFINE
...
#Add:=#PARA1+#PARA2
...
END-FUNCTION
END
Copycode Object
...
DEFINE PROTOTYPE #ADD
RETURNS (I4)
DEFINE DATA
PARAMETER USING PDA1
END-DEFINE
END-PROTOTYPE
...

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*:

- **Inline Subroutine**
  An inline subroutine is defined within the object which contains the `PERFORM` statement that invokes it.

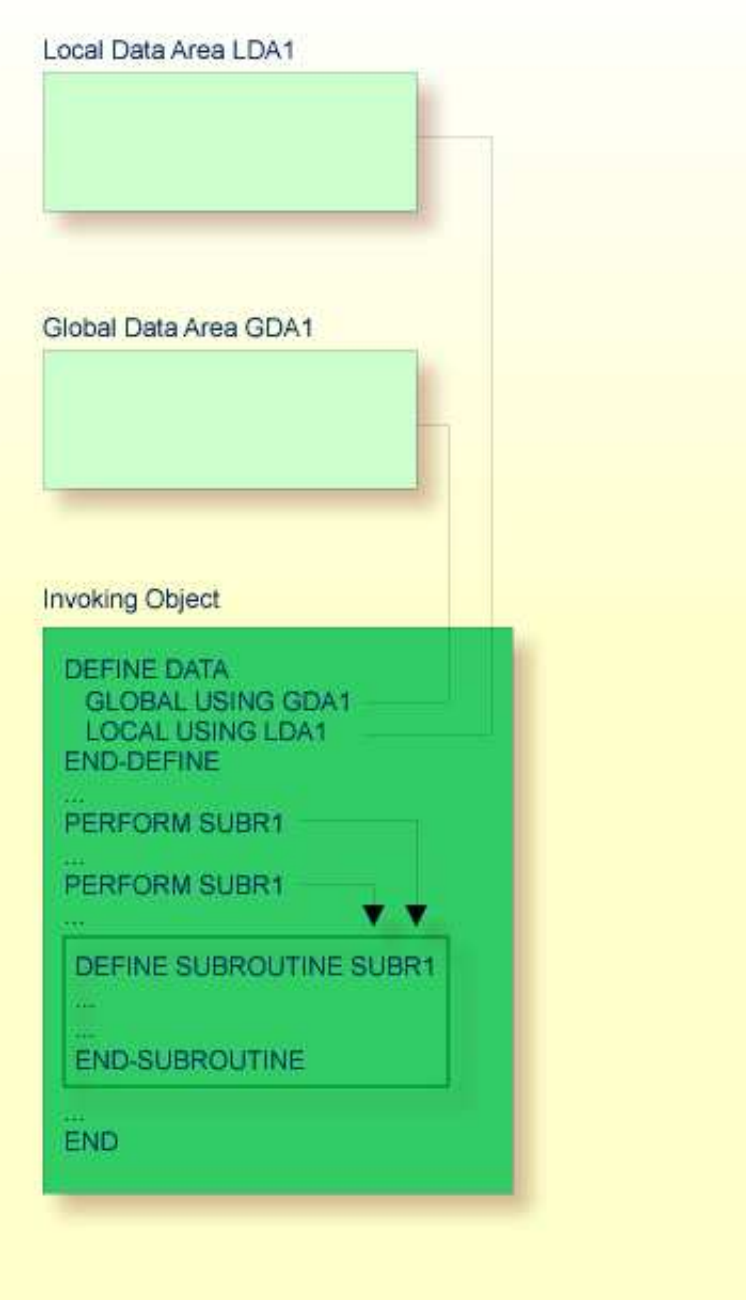
- **External Subroutine**
  An external subroutine is defined in a separate object - of type subroutine - outside the object which invokes it.

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.

The following topics are covered below:

- Inline Subroutine
- Data Available to an Inline Subroutine
- External Subroutine
- Data Available to an External Subroutine

Inline Subroutine



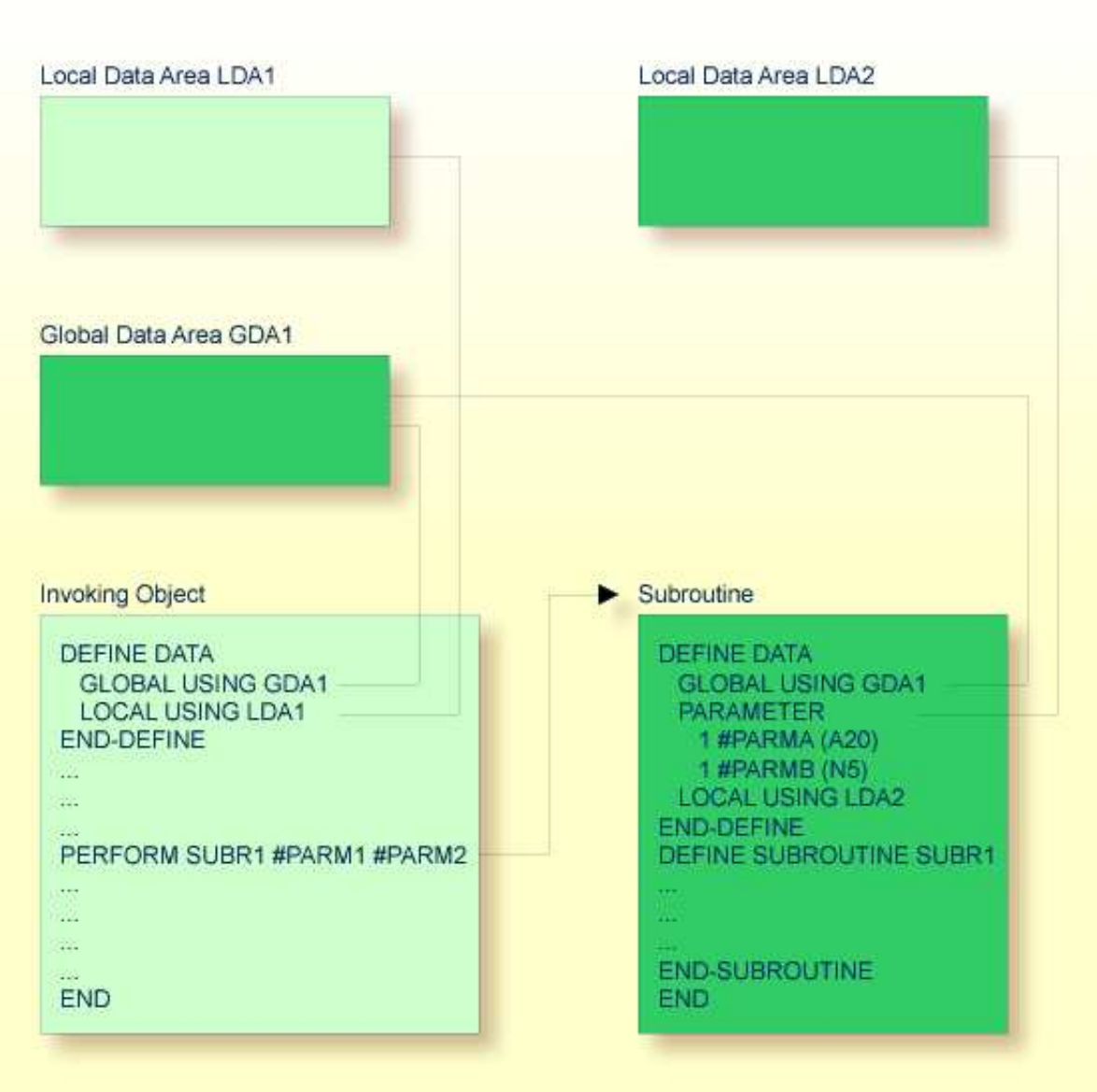
An inline subroutine can be contained within a programming object of type program, function, subprogram, subroutine or helproutine.

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.

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.

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, function, subprogram, subroutine or helproutine.

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 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.

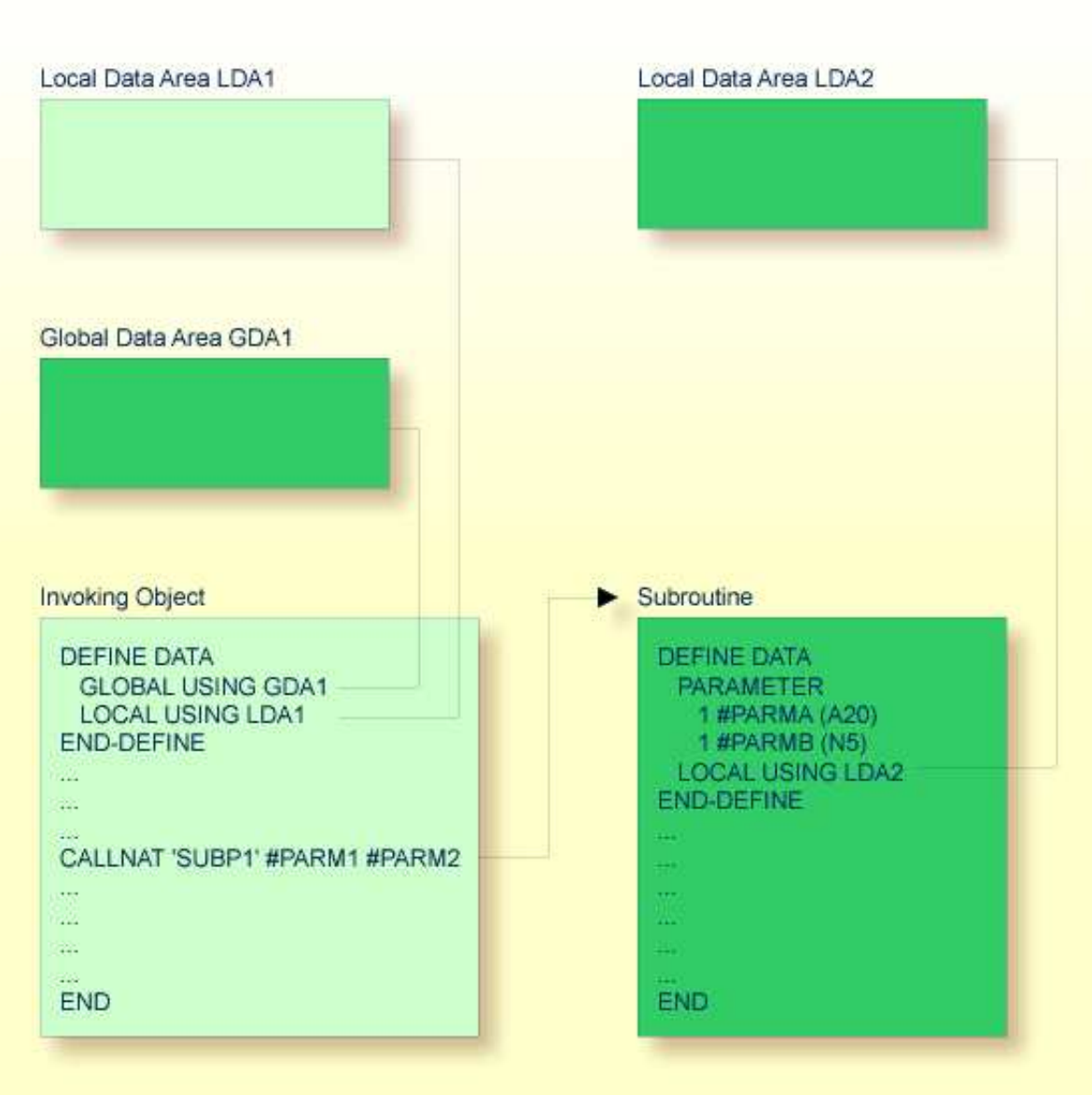
A subprogram cannot be executed by itself. It must be invoked from another object. The invoking object can be a program, function, subprogram, subroutine or helproutine.

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 helproutine, it can also establish its own global data area to be shared with the subroutine/helproutine.

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: