Natural

First Steps

Version 9.3.2

April 2026

ADABAS & NATURAL

This document applies to Natural Version 9.3.2 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 1992-2026 Software GmbH, Darmstadt, Germany and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software GmbH product names are either trademarks or registered trademarks of Software GmbH and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software GmbH and/or its subsidiaries is located at https://softwareag.com/licenses.

Use of this software is subject to adherence to Software GmbH's licensing conditions and terms. These terms are part of the product documentation, located at https://softwareag.com/licenses and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software GmbH Products / Copyright and Trademark Notices of Software GmbH Products". These documents are part of the product documentation, located at https://softwareag.com/licenses and/or in the root installation directory of the licensed product(s).

Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided for in your License Agreement with Software GmbH.

Document ID: NATWIN-NNATFIRSTSTEPS-932-20260408

Table of Contents

Preface v
1 About this Documentation 1
Document Conventions 2
Online Information and Support 2
Data Protection 3
2 About this Tutorial 5
Prerequisites 6
About the Sample Application 6
3 Getting Started with Natural Studio 9
Invoking Natural Studio 10
Library Workspace 11
Issuing Commands 12
Creating a User Library 12
Programming Modes 12
4 Hello World! 15
Creating a Program 16
Running a Program 17
Correcting Program Errors 18
Stowing a Program 19
Setting the Workspace Options 20
5 Database Access 23
Starting the Demo Database 24
Saving Your Program Under a New Name 25
Defining the Required Data Using a View 25
Reading Data from a Database 29
Reading Selected Data from a Database 30
6 User Input 33
Allowing for User Input 34
Designing a Map for User Input 36
Invoking the Map from Your Program 46
Ensuring that an Ending Name is Always Used 47
7 Loops and Labels 51
Allowing Repeated Usage 52
Displaying a Message Indicating that Information was not Found 54
8 Inline Subroutines 57
Defining the Inline Subroutine 58
Performing the Inline Subroutine 59
9 Processing Rules and Helproutines 61
Defining a Processing Rule 62
Defining a Helproutine 64
10 Local Data Areas 67
Creating a Local Data Area 68
Defining Data Fields 69

Importing the Required Data Fields from a DDM 72
Referencing the Local Data Area from Your Program 73
11 Global Data Areas 77
Creating a Global Data Area from an Existing Local Data Area 78
Adapting the Local Data Area 80
Referencing the Global Data Area from Your Program 80
12 External Subroutines 83
Creating an External Subroutine 84
Referencing the External Subroutine from Your Program 85
13 Subprograms 89
Modifying the Local Data Area 90
Creating a Parameter Data Area from an Existing Local Data Area 92
Creating Another Local Data Area Containing a Different View 93
Creating a Subprogram 94
Referencing the Subprogram from Your Program 95

Preface

About this Tutorial	Prerequisites and what you will learn in the course of this tutorial.
Getting Started with Natural Studio	How to invoke Natural Studio. Information on the library workspace and the different ways of issuing commands. How to create the library that will be used in this tutorial. Information on Natural's programming modes and the mode that is required for this tutorial.
Hello World!	How to create, run and stow your first short program. How to display the content of the current library. Information on some options which control your workspace.
Database Access	Information on the demo database. How to read specific data from a database and display the output.
User Input	How to prompt the user for information and how to design a map for user input. How to ensure that a specific value is always used (here: an ending name), even if it has not been specified by the user.
Loops and Labels	How to define a repeat loop and labels for different loops. How to display a message when specific information (here: the starting name entered by the user) was not found.
Inline Subroutines	How to define and invoke an inline subroutine (that is: a subroutine which is coded directly in the program).
Processing Rules and Helproutines	How to define a processing rule (here: a message that is to appear when the user does not specify a starting name) and a helproutine (here: a help text for the field in which the user has to enter a starting name).
Local Data Areas	How to relocate the field definitions from the program to a local data area outside the program.
Global Data Areas	How to define a global data area which can be shared by multiple programs or routines.
External Subroutines	How to define and invoke an external subroutine (that is: a subroutine which is stored as a separate object outside the program).
Subprograms	How to define a parameter data area for a subprogram. How to define and invoke a subprogram.

1 About this Documentation

- Document Conventions 2
- Online Information and Support 2
- Data Protection 3

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *`folder.subfolder.service`*, APIs, Java classes, methods, properties.
`Italic`	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at **https://documentation.softwareag.com**.

Product Training

You can find helpful product training material on our Learning Portal at **https://learn.softwareag.com**.

Tech Community

You can collaborate with Software GmbH experts on our Tech Community website at **https://techcommunity.softwareag.com**. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software GmbH news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at **https://github.com/softwareag** and **https://hub.docker.com/publishers/softwareag** and discover additional Software GmbH resources.

Product Support

Support for Software GmbH products is provided to licensed customers via our Empower Portal at **https://empower.softwareag.com**. Many services on this portal require that you have an account. If you do not yet have one, you can request it at **https://empower.softwareag.com/register**. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

2 About this Tutorial

- Prerequisites 6
- About the Sample Application 6

This tutorial provides a very simple and brief introduction to Natural Studio (which is the development environment for Natural) and to programming with Natural.

As a first-time user, you are recommended to work through this tutorial to obtain a basic understanding of specific features of the Natural programming environment.

It is assumed that you have a basic understanding of Microsoft Windows.

Important: Read the following topics in the sequence indicated below, and that you work through all exercises in these topics in the same sequence as they appear in this tutorial. Problems may occur if you skip an exercise.

The layout of the example screens provided in the tutorial and the behavior of Natural described here can differ from your results. For example, the command or message line may appear in a different screen position, or the execution of a Natural command may be protected by security control. The default settings in your environment depend on the system parameters set by your Natural administrator.

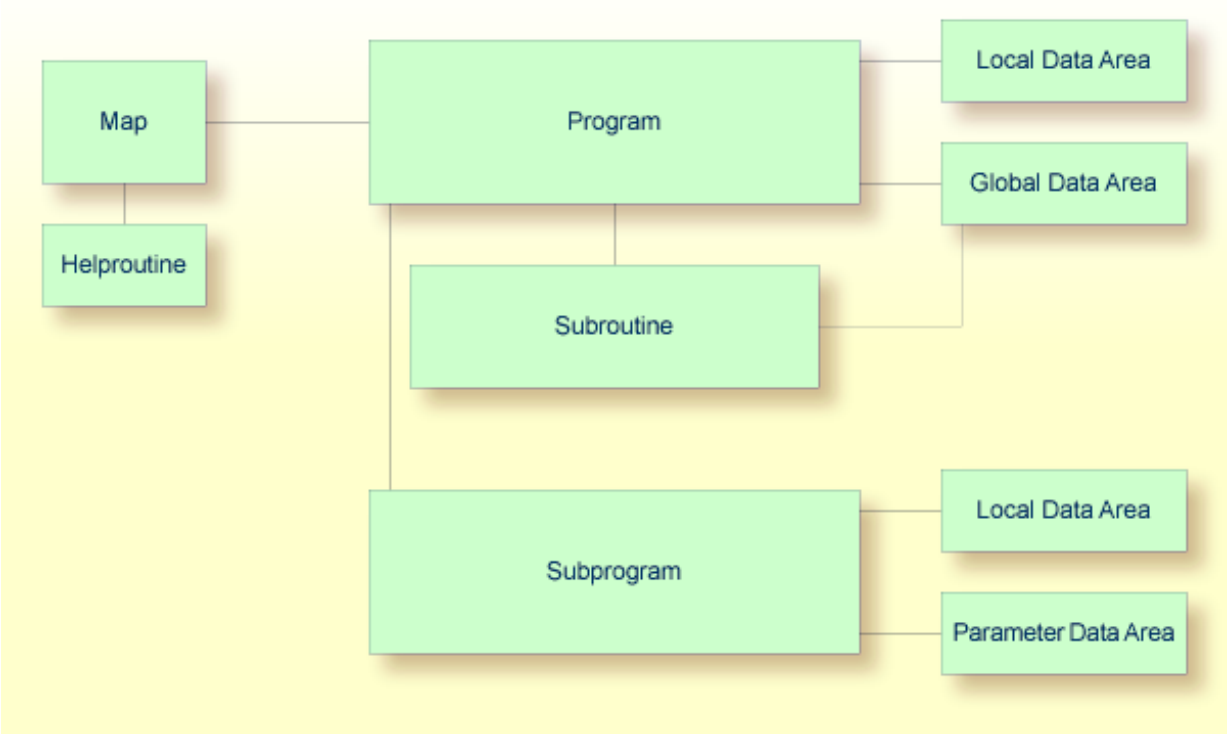
Prerequisites

To perform all steps of this tutorial, the demo database `SAG-DEMO-DB` must be installed (either locally under Windows or under Linux) and it must be active. This database is installed with Adabas; it is not automatically installed with Natural.

About the Sample Application

This tutorial illustrates how an application can be structured as a group of modules. It is not intended to provide an example of how an application should be built.

After you have written your first short Hello World program, you will write a program which reads employees information from a database and displays the output. The user will be prompted to enter a starting name and ending name for the output. You will enhance your program step by step by moving specific parts of your program to external modules. When you have completed all exercises of this tutorial, your application will be structured as follows:



Note: This tutorial describes how to create a map which is normally used in a character-oriented environment (such as a mainframe). For a graphical user interface, you would create a dialog. However, this is not part of this tutorial.

If you do not want to type in all program code yourself, you can copy the code examples from the tutorial into the Natural editors (for example, with the standard Windows key combinations CTRL+C and CTRL+V for copy and paste).

You can now proceed with your first exercise: *Getting Started with Natural Studio*.

3 Getting Started with Natural Studio

- Invoking Natural Studio 10
- Library Workspace 11
- Issuing Commands 12
- Creating a User Library 12
- Programming Modes 12

Invoking Natural Studio

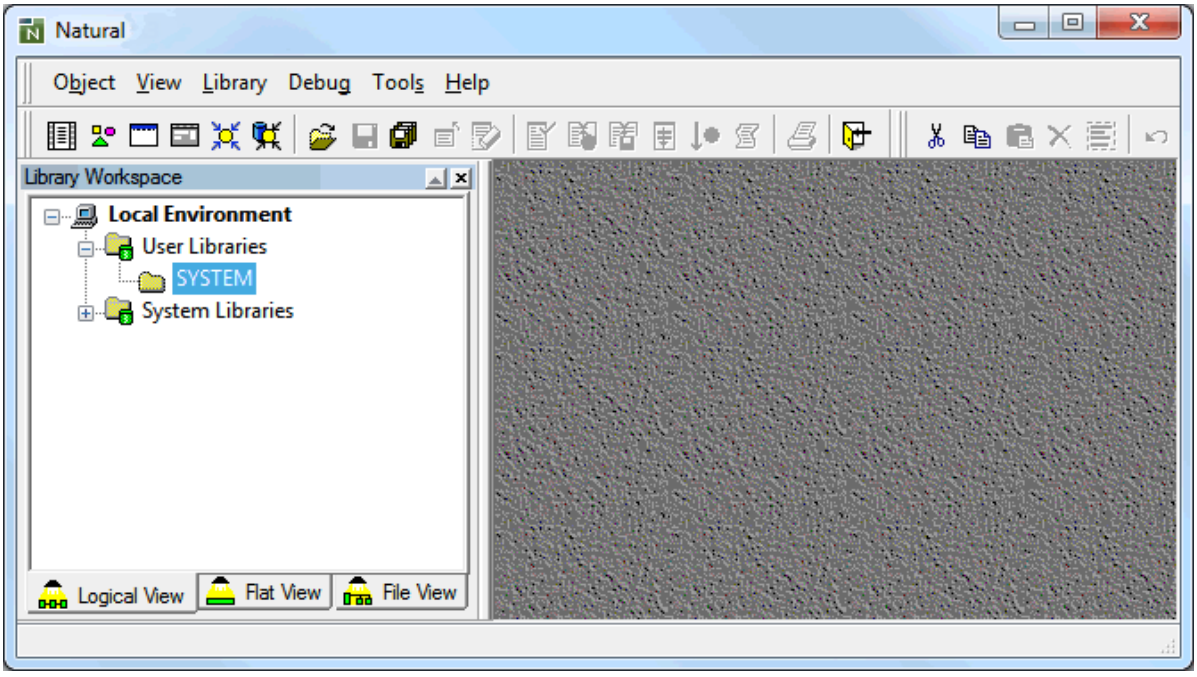
When Natural is installed, an entry for starting Natural is automatically added to the Windows **Start** menu.

To invoke Natural Studio

- From the Windows **Start** menu, choose **All Programs > Software AG > Tools > Natural** *n.n*.

  Note: The **Start** menu group name (by default, this is "Software AG") can be changed during the installation.

  The Natural Studio window appears.



When you start Natural Studio for the very first time, only your local environment containing the library workspace is shown in the window.

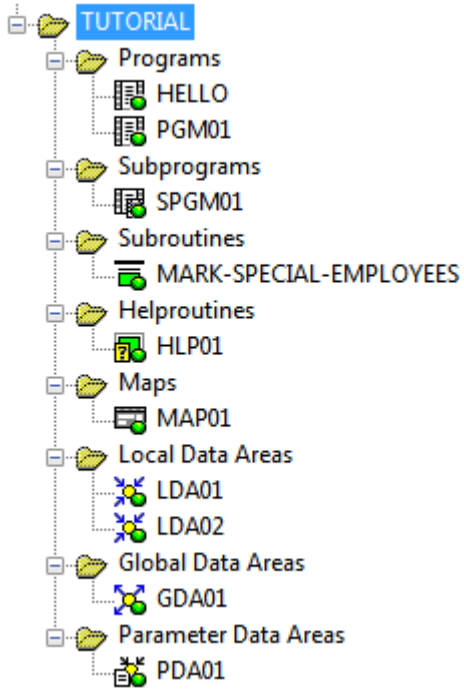
Library Workspace

All Natural objects required for creating an application are stored in Natural libraries in Natural system files. There is a system file for system programs (FNAT) and a system file for user-written programs (FUSER).

Natural thus distinguishes system libraries and user libraries. The system libraries, which start with the letters "SYS", are reserved for Software AG purposes only. A user library contains all user-defined objects (for example, programs and maps) which make up an application. The name of a user library must not start with the letters "SYS".

Different views are available in the library workspace. The exercises in this tutorial require that you work in logical view. In logical view, different nodes are provided for the libraries. The objects in a library are grouped into different folders, according to their types.

When you have completed all exercises of this tutorial, the node for your user library **TUTORIAL** will contain the following folders and objects:



Issuing Commands

As with other Windows applications, most commands in Natural Studio can be issued in a number of different ways: they can be chosen from the menu bar or from a context menu, or by choosing a toolbar button or pressing a key combination. This tutorial, however, does not mention all alternatives for issuing the same command. It just mentions commonly-used methods (in most cases: context menus and toolbar buttons).

To invoke a context menu (for example, for an object in the library workspace), you select the object and then click the right mouse button or press SHIFT+F10.

Several menus or toolbars are only shown in a specific context. For example, the **Program** menu is only shown in the menu bar when you are working with the program editor and the program editor window is active.

Creating a User Library

You will now create a user library with the name `TUTORIAL`. This library is to contain all Natural objects that you will create in the course of this tutorial.

To create a user library

1. In logical view, select the node named **User Libraries** which is located directly below the top node **Local Environment**.
2. From the context menu and choose **New**.

   A new library with the default name **USRNEW** is now shown in the tree.
3. Specify "TUTORIAL" as the name of the library and press ENTER.

Programming Modes

Natural provides two different programming modes:

- **Structured Mode**
  Structured mode is intended for the implementation of complex applications with a clear and well-defined program structure. It is recommended to use structured mode exclusively.

- **Reporting Mode**
  Reporting mode is only useful for the creation of adhoc reports and small programs which do not involve complex data and/or programming constructs.

Important: This tutorial requires that structured mode is active. If you try to run your program in reporting mode, `END-IF`, `END-READ` and `END-REPEAT` will cause errors.

To check whether structured mode is active

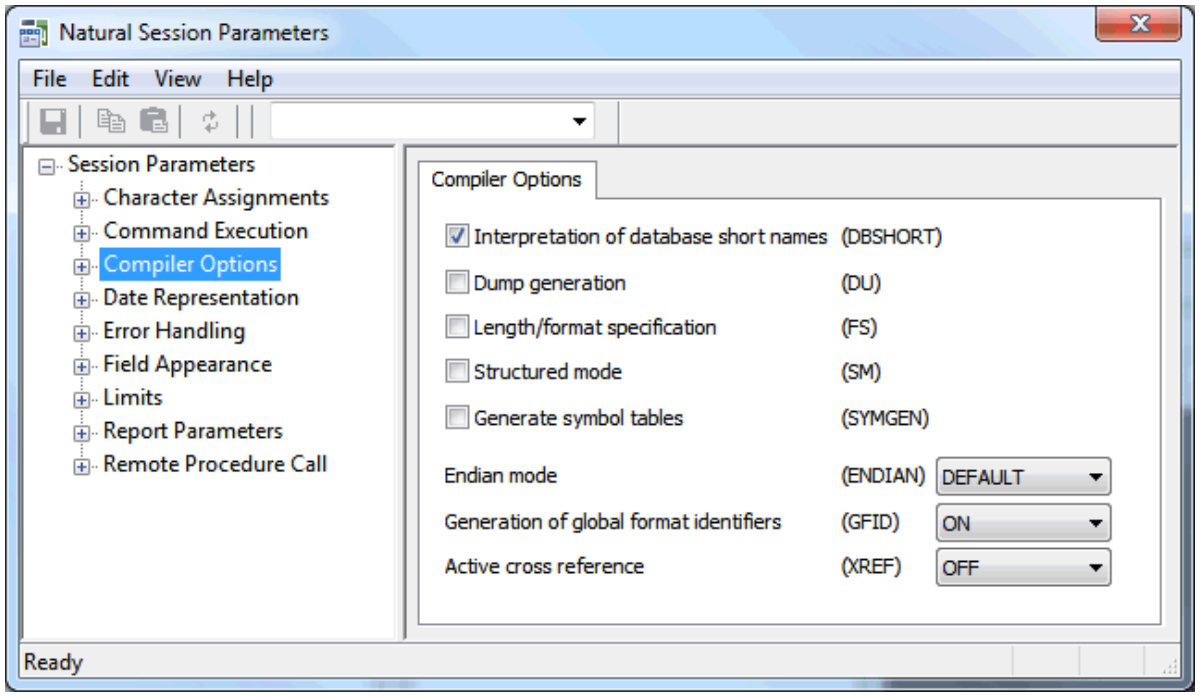
1 From the **Tools** menu, choose **Session Parameters**.

The **Natural Session Parameters** dialog box appears.

At the installation of Natural, the Natural administrator sets these parameters to default values which are then valid for all users of Natural. When you change them, they are only valid for current session.

2 In the tree on the left, select **Compiler Options**.

The compiler options are now shown on the right.



When the **Structured mode** check box is already selected, no further steps are required and you can close the dialog box. When it is not selected, proceed as described below.

3 Activate the **Structured mode** check box.

4 From the **File** menu, choose **Save**.

5 Close the dialog box.

You can now proceed with your first program: ***Hello World!***

4 Hello World!

- Creating a Program 16
- Running a Program 17
- Correcting Program Errors 18
- Stowing a Program 19
- Setting the Workspace Options 20

Creating a Program

You will now write your first short program which displays "Hello World!". It will be stored in the library you have created previously.

To create a new program

1 In the library workspace, select the library named TUTORIAL.

2 From the context menu, choose **New Source > Program**.

Or:

Choose the following toolbar button:

The program editor appears. It is currently empty.

3 Enter the following code in the program editor:

```
* The "Hello world!" example in Natural.
*
DISPLAY "Hello world!"
END /* End of program
```

Comment lines start with an asterisk (*) followed by at least one blank or a second asterisk. When you forget to enter the blank or second asterisk, Natural assumes that you have specified a system variable; this will result in an error.

If you want to insert empty lines in your program, you should define them as comment lines. This is helpful, if you want to access your program from different platforms (Windows, mainframe or Linux and Cloud). With the mainframe version of Natural, for example, the default is that empty lines are automatically deleted when you press ENTER.

You can also insert comments at the end of a statement line. In this case, the comment starts with a slash followed by an asterisk (/*).

The text that is to be shown in the output is defined with the DISPLAY statement. It is enclosed in quotation marks.

The END statement is used to mark the physical end of a Natural program. Each program must end with END.

Running a Program

The system command RUN automatically invokes the system command CHECK which checks the program code for errors. If no error is found, the program is compiled on the fly and then executed.

Notes:

1. The system commands are also available with the mainframe version of Natural. Under Windows, they are invoked by choosing the corresponding command from the **Object** menu.
2. CHECK is also available as a separate command in the **Object** menu.
3. Natural also provides the system command EXECUTE which uses the stowed version of your program (stowing a program is explained later in this tutorial). In contrast to this, the RUN command always uses your latest modifications to the program.

To run a program

1 From the **Object** menu, choose **Run**.

Or:

Choose the following toolbar button:

When your code is syntactically correct, the output contains the text you have defined.

```
Page      1                                                  13-05-16  13:27:42

Hello world!
```

2 Press ENTER to return to the program editor.

Correcting Program Errors

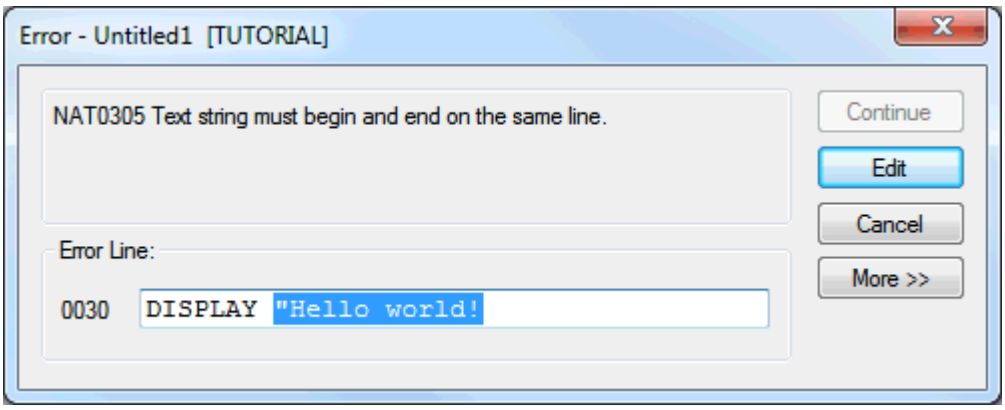
You will now create an error in your Hello World program and then run the program once more.

To correct an error

1 Delete the second quotation mark in the line containing the `DISPLAY` statement.

2 Run the program once more as described above.

When the error is found, a dialog box appears, providing information on the error.



3 Correct the error in the dialog box, that is: insert the missing quotation mark at the end of the line.

4 Choose the **Continue** button to find the next error.

In this case, no more errors are found and the output is shown.

5 Press ENTER to return to the program editor.

Note: Instead of choosing the **Continue** button, it is also possible to choose the **Edit** button. The dialog box is then closed and you can correct the error directly in the program editor.

Stowing a Program

When you stow a program, it is compiled and both source code and a generated program are stored in the Natural system file.

Like the RUN command, the system command STOW automatically invokes the CHECK command. A program is only stowed when it is syntactically correct.

Note: If you want to save the changes to your program, even if the program contains a syntactical error (for example, if you want to suspend your work until the next day), you can use the system command SAVE which can be invoked from the **Object** menu.

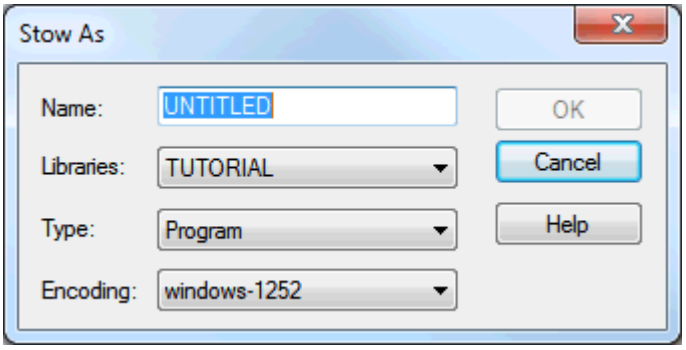
To stow a program

1 From the **Object** menu, choose **Stow**.

Or:

Choose the following toolbar button:

Since your program has not yet been saved, the **Stow As** dialog box appears.



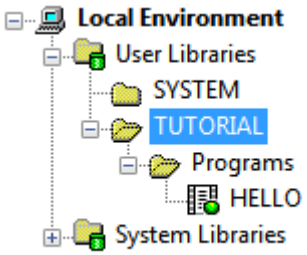
The name of the currently selected library is automatically provided in the corresponding drop-down list box.

2 Specify the name "HELLO" in the **Name** text box.

3 Choose the **OK** button.

A message is now shown, informing you that stowing was successful. This message is shown in the status bar and/or in a dialog box, depending on the setting of a specific workspace option (see below).

In the library workspace, a new node named **Programs** appears below the **TUTORIAL** node. This node contains the program you have just stowed.



The green dot on the program icon indicates that both source code and a generated program are available for the object.

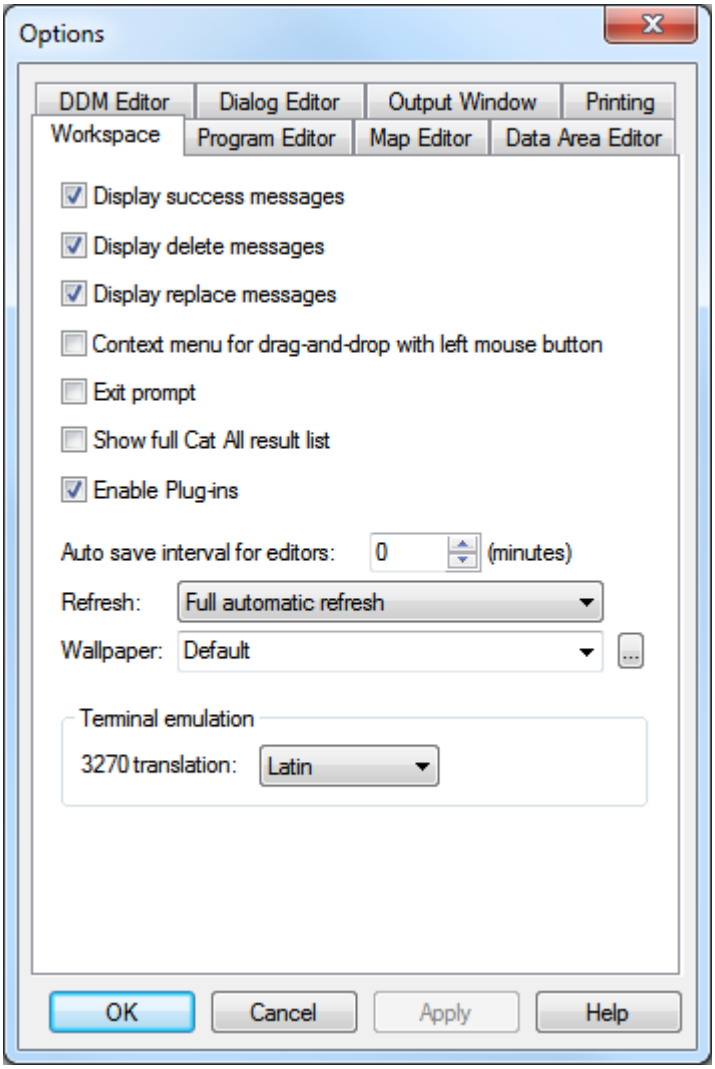
Setting the Workspace Options

You will now check the settings of your workspace options.

To check the workspace options

1 From the **Tools** menu, choose **Options**.

2 In the resulting **Options** dialog box, display the **Workspace** page.



3 When success messages are to be shown in a dialog box, make sure that the corresponding check box is selected.

Note: Whether line numbers are shown in the program editor is controlled by an option on the **Program Editor** page.

4 Choose the **OK** button to save your changes and to close the dialog box.

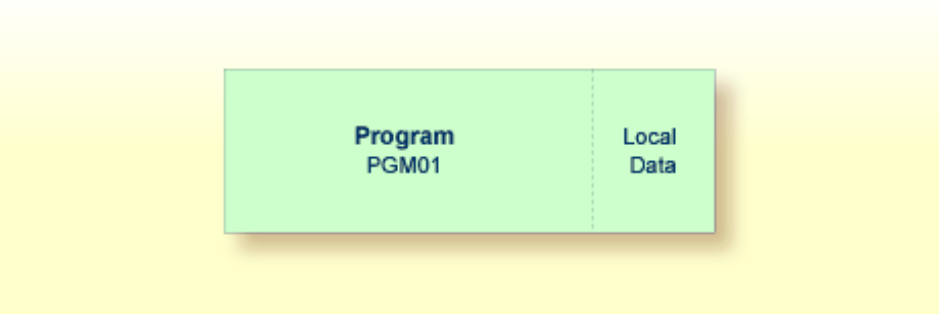
You can now proceed with the next exercises: *Database Access*.

5 Database Access

- Starting the Demo Database 24
- Saving Your Program Under a New Name 25
- Defining the Required Data Using a View 25
- Reading Data from a Database 29
- Reading Selected Data from a Database 30

You will now write a short program which reads specific data from a database file and displays the corresponding output.

When you have completed the exercises below, your sample application will consist of just one module (the data fields that are used by the program are defined within the program):



Starting the Demo Database

The demo database `SAG-DEMO-DB` is not started automatically. Before you can proceed with the exercises in this tutorial, you must make sure that it has been started. Otherwise, the examples will not work.

The following description applies when Adabas has been installed locally under Windows. If you want to work with the Linux and Cloud version and the demo database is not already running, ask your Linux and Cloud administrator to start it.

To start the demo database

1 From the **Start** menu, choose **All Programs > Software AG Adabas** *n.n* **> Adabas DBA Workbench**.

The status of the demo database is shown in the resulting database list. When the status is "Active", no further steps are required and you can close the DBA Workbench application window.

When the status is not "Active", proceed as described below.

2 Select **SAG-DEMO-DB** in the database list.

3 From the **Database** menu, choose **Start**.

A dialog box appears indicating that the database has been started.

4 Choose the **OK** button to close the dialog box.

5 Close the DBA Workbench application window.

Saving Your Program Under a New Name

You will now create a new program which will be used in the remainder of this tutorial. It will be created by saving your Hello World program under a new name.

To save the program under a new name

1 From the **Object** menu, choose **Save As**.

> **Tip:** Make sure that the program editor is selected. Otherwise the above command is not available.

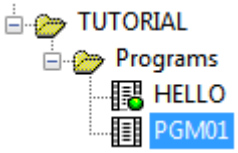
The **Save As** dialog box appears.

2 Specify "PGM01" as the new name for the program.

3 Choose the **OK** button.

The new name is now shown in the title bar of the program editor.

In the library workspace, the new program is shown in the **Programs** node. Since it has not yet been stowed, its program icon does not contain a green dot.



4 Delete all code in the program editor (for example, by pressing CTRL+A to select all text and then pressing the DEL key - this is standard Windows functionality).

Defining the Required Data Using a View

The database file and the fields that are to be used by your program have to be specified between `DEFINE DATA` and `END-DEFINE` at the top of the program.

For Natural to be able to access a database file, a logical definition of the physical database file is required. Such a logical file definition is called a data definition module (DDM). The DDM contains information about the individual fields of the file. DDMs are usually defined by the Natural administrator.

To be able to use the database fields in a Natural program, you must specify the fields from the DDM in a view. Sample DDMs are provided in the system library SYSEXDDM. For this tutorial, we will use the DDM for the EMPLOYEES database file.

You can import the fields, including the required format and length definitions, from the DDM into the program editor.

To specify the DEFINE DATA block

- Enter the following code in the program editor:

```
DEFINE DATA
LOCAL

END-DEFINE
*
END
```

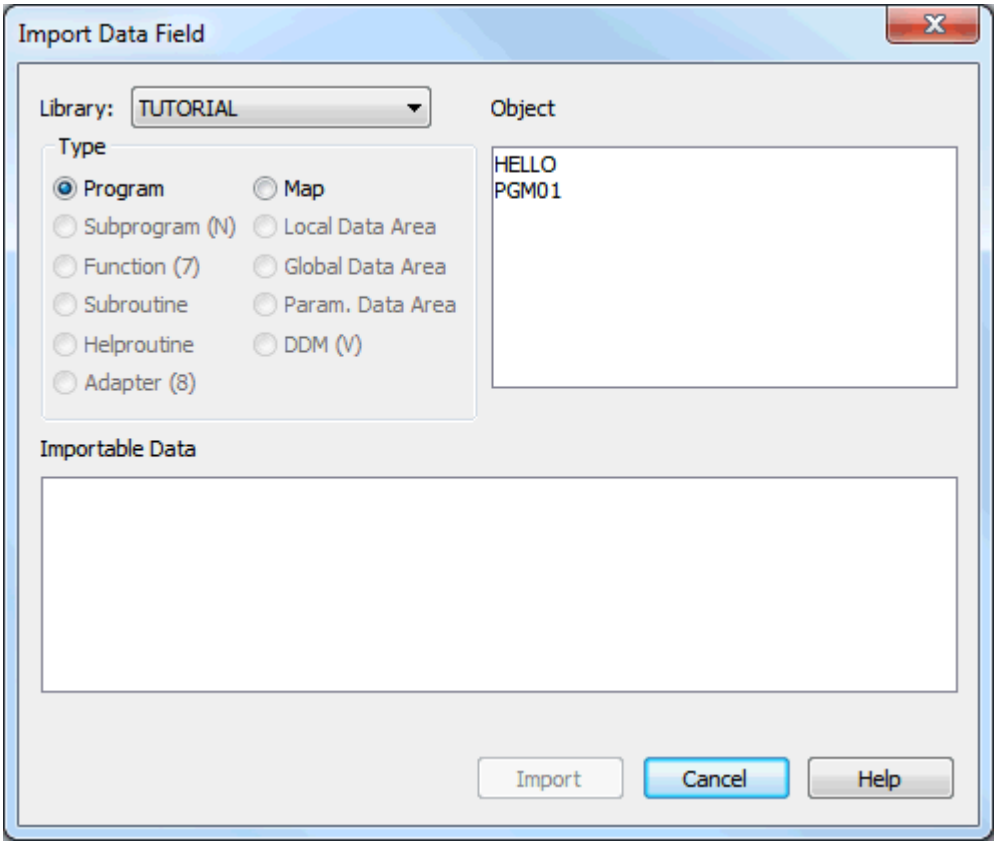
Tip: The Windows version of Natural does not distinguish between uppercase and lowercase letters. However, when working with the mainframe version of Natural, keywords and identifiers are always entered with uppercase letters; text constants may contain lowercase letters. Therefore, if you also want to edit your programs on a mainframe, it is recommended that you always enter your program code as you would on a mainframe.

LOCAL means that the variables that you will define with the next step are local variables which apply only to this program.

To import data fields from a DDM

1. Place the cursor in the line below LOCAL.
2. From the **Program** menu, choose **Import**.

   The **Import Data Field** dialog box appears.



3 From the **Library** drop-down list box, select **SYSEXDDM**.

When the **DDM** option button is selected, all defined DDMs are shown in the **Object** list box.

4 Select the sample DDM with the name `EMPLOYEES`.

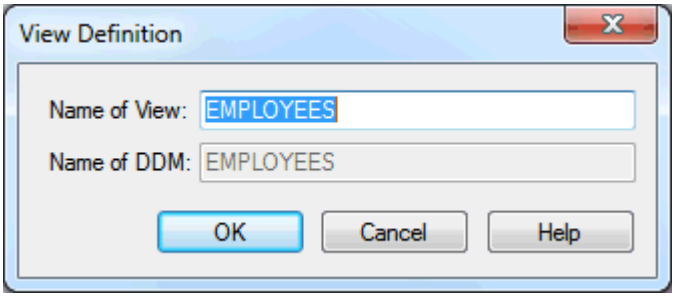
The importable data fields are now shown at the bottom of the dialog box.

5 Press CTRL and select the following fields:

```
FULL-NAME
NAME
DEPT
LEAVE-DATA
LEAVE-DUE
```

6 Choose the **Import** button.

The **View Definition** dialog box appears.



By default, the name of the DDM is proposed as the view name. You can specify any other name.

7 Enter "EMPLOYEES-VIEW" as the view name.

8 Choose the **OK** button.

The **Cancel** button in the **Import Data Field** dialog box is now labeled **Quit**.

9 Choose the **Quit** button to close the **Import Data Field** dialog box.

The following code has been inserted in the program editor:

```
1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
  2 FULL-NAME
    3 NAME (A20)
  2 DEPT (A6)
  2 LEAVE-DATA
    3 LEAVE-DUE (N2)
```

The first line contains the name of your view and the name of the database file from which the fields have been taken. This is always defined on level 1. The level is indicated at the beginning of the line. The names of the database fields from the DDM are defined at levels 2 and 3.

Levels are used in conjunction with field grouping. Fields assigned a level number of 2 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (this may also be only one field) by using the group name. This is a convenient and efficient method of referencing a series of consecutive fields.

Format and length of each field is indicated in parentheses. "A" stands for alphanumeric, and "N" stands for numeric.

Reading Data from a Database

Now that you have defined the required data, you will add a READ loop. This reads the data from the database file using the defined view. With each loop, one employee is read from the database file. Name, department and remaining days of vacation for this employee are displayed. Data are read until all employees have been displayed.

Note: It may happen that an error message is displayed indicating that the transaction has been aborted. This usually happens when the non-activity time limit which is determined by Adabas has been exceeded. When such an error occurs, you should simply repeat your last action (for example, issue the RUN command once more).

To read data from a database

1 Insert the following below END-DEFINE:

```
READ EMPLOYEES-VIEW BY NAME
*
  DISPLAY NAME 3X DEPT 3X LEAVE-DUE
*
END-READ
```

BY NAME indicates that the data which is read from the database is to be sorted alphabetically by name.

The DISPLAY statement arranges the output in column format. A column is created for each specified field and a header is placed over the column. 3X means that 3 spaces are to be inserted between the columns.

2 Run the program.

The following output appears.

```
Page      1                                                  09-06-30  16:06:49

        NAME           DEPARTMENT   LEAVE
                          CODE       DUE
--------------------   ----------   -----

ABELLAN                PROD04        20
ACHIESON               COMP02        25
ADAM                   VENT59        19
ADKINSON               TECH10        38
ADKINSON               TECH10        18
ADKINSON               TECH05        17
ADKINSON               MGMT10        28
ADKINSON               TECH10        26
```

```
ADKINSON                SALE30        36
ADKINSON                SALE20        37
ADKINSON                SALE20        30
AECKERLE                SALE47        31
AFANASSIEV              MGMT30        26
AFANASSIEV              TECH10        35
AHL                     MARK09        30
AKROYD                  COMP03        20
ALEMAN                  FINA03        20
```

As a result of the `DISPLAY` statement, the column headers (which are taken from the DDM) are underlined and one blank line is inserted between the underlining and the data. Each column has the same width as defined in the `DEFINE DATA` block (that is: as defined in the view).

The title at the top of each page, which contains the page number, date and time, is also caused by the `DISPLAY` statement.

3 Press ENTER repeatedly to display all pages.

You will return to the program editor when all employees have been displayed.

Tip: If you want to return to the program editor before all employees have been displayed, press ESC.

Reading Selected Data from a Database

Since the previous output was very long, you will now restrict it. Only the data for a range of names is to be displayed, starting with "Adkinson" and ending with "Bennett". These names are defined in the demo database.

To restrict the output to a range of data

1 Before you can use new variables, you have to define them. Therefore, insert the following below `LOCAL`:

```
1 #NAME-START        (A20) INIT <"ADKINSON">
1 #NAME-END          (A20) INIT <"BENNETT">
```

These are user-defined variables; they are not defined in demo database. The hash (#) at the beginning of the name is used to distinguish the user-defined variables from the fields defined in the demo database; however, it is not a required character.

`INIT` defines the default value for the field. The default value must be specified in pointed brackets and quotation marks.

2 Insert the following below the `READ` statement:

```
STARTING FROM #NAME-START
ENDING AT #NAME-END
```

Your program should now look as follows:

```
DEFINE DATA
LOCAL
  1 #NAME-START        (A20) INIT <"ADKINSON">
  1 #NAME-END          (A20) INIT <"BENNETT">
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 FULL-NAME
      3 NAME (A20)
    2 DEPT (A6)
    2 LEAVE-DATA
      3 LEAVE-DUE (N2)
END-DEFINE
*
READ EMPLOYEES-VIEW BY NAME
  STARTING FROM #NAME-START
  ENDING AT #NAME-END
*
  DISPLAY NAME 3X DEPT 3X LEAVE-DUE
*
END-READ
*
END
```

3 Run the program.

The output is shown. When you press ENTER repeatedly, you will notice that you will return to the program editor after a couple of pages (that is: when the data for the last employee named Bennett has been displayed).

4 Stow the program.

You can now proceed with the next exercises: ***User Input***.

6 User Input

- Allowing for User Input 34
- Designing a Map for User Input 36
- Invoking the Map from Your Program 46
- Ensuring that an Ending Name is Always Used 47

You will now learn how to prompt the user for data, that is: a starting name and an ending name for the output.

When you have completed the exercises below, your sample application will consist of the following modules:

Allowing for User Input

You will now modify your program so that input fields for the starting name and ending name will be shown in the output. This is done using the `INPUT` statement.

To define input fields

1 Insert the following below `END-DEFINE`:

```
INPUT (AD=MT)
  "Start:" #NAME-START /
  "End:   " #NAME-END
```

The session parameter `AD` stands for “attribute definition”, its value "M" stands for “modifiable output field”, and the value "T" stands for “translate lowercase to uppercase”.

The "M" value in `AD=MT` means that the default values defined with `INIT` (that is: "ADKINSON" and "BENNETT") will be shown in the input fields. Different values may be entered by the user. When the "M" value is omitted, the input fields will be empty even though default values have been defined.

The "T" value in `AD=MT` means that all lowercase input is translated to uppercase before further processing. This is important since the names in the demo database file have been defined completely in uppercase letters. When the "T" value is omitted, you have to enter all names completely in uppercase letters. Otherwise, the specified name will not be found.

`"Start:"` and `"End:"` are text fields (labels). They are specified in quotation marks.

#NAME-START and #NAME-END are data fields (input fields) in which the user can enter the desired starting name and ending name.

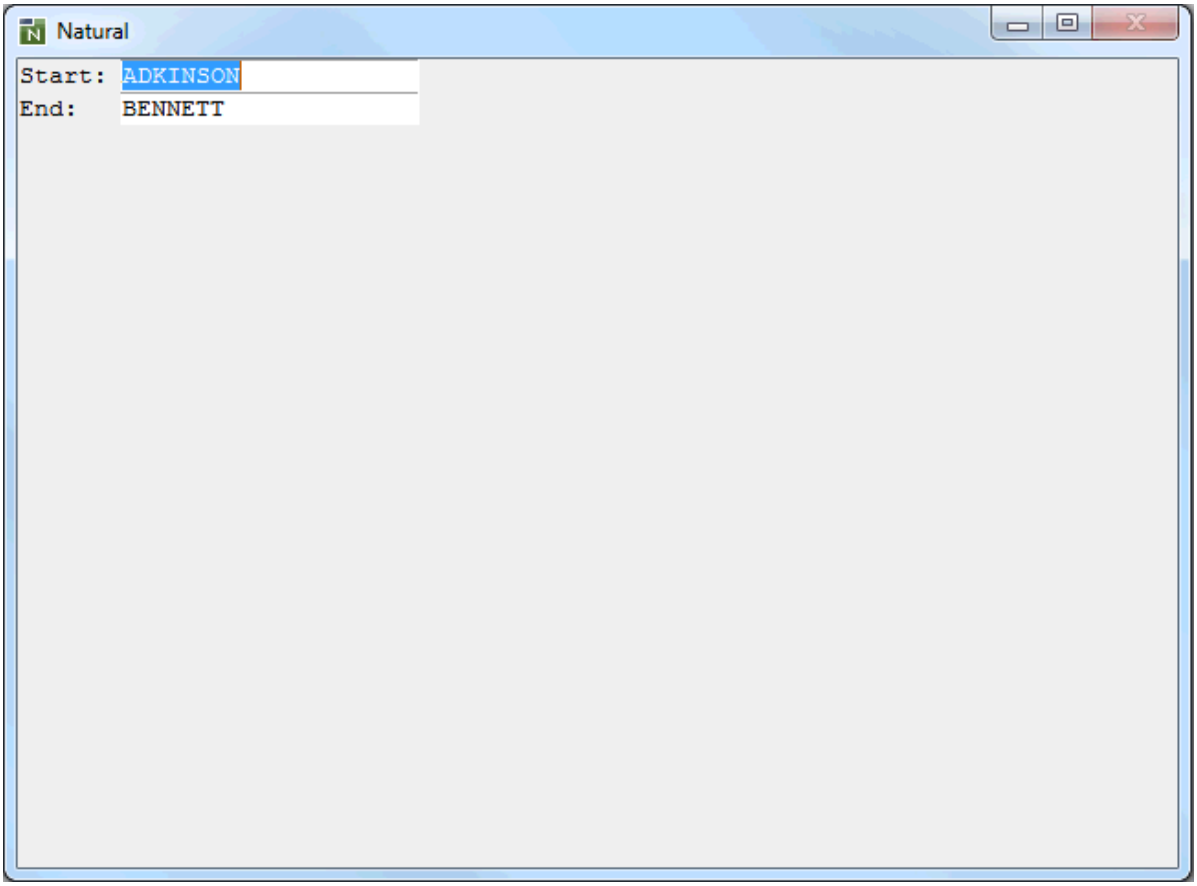
The slash (/) means that the subsequent fields are to be shown in a new line.

Your program should now look as follows:

```
DEFINE DATA
LOCAL
  1 #NAME-START        (A20) INIT <"ADKINSON">
  1 #NAME-END          (A20) INIT <"BENNETT">
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 FULL-NAME
      3 NAME (A20)
    2 DEPT (A6)
    2 LEAVE-DATA
      3 LEAVE-DUE (N2)
END-DEFINE
*
INPUT (AD=MT)
  "Start:" #NAME-START /
  "End:  " #NAME-END
*
READ EMPLOYEES-VIEW BY NAME
  STARTING FROM #NAME-START
  ENDING AT #NAME-END
*
  DISPLAY NAME 3X DEPT 3X LEAVE-DUE
*
END-READ
*
END
```

2 Run the program.

The output shows the fields you have just defined.



3 Use the default names and press ENTER.

The list of employees is now shown.

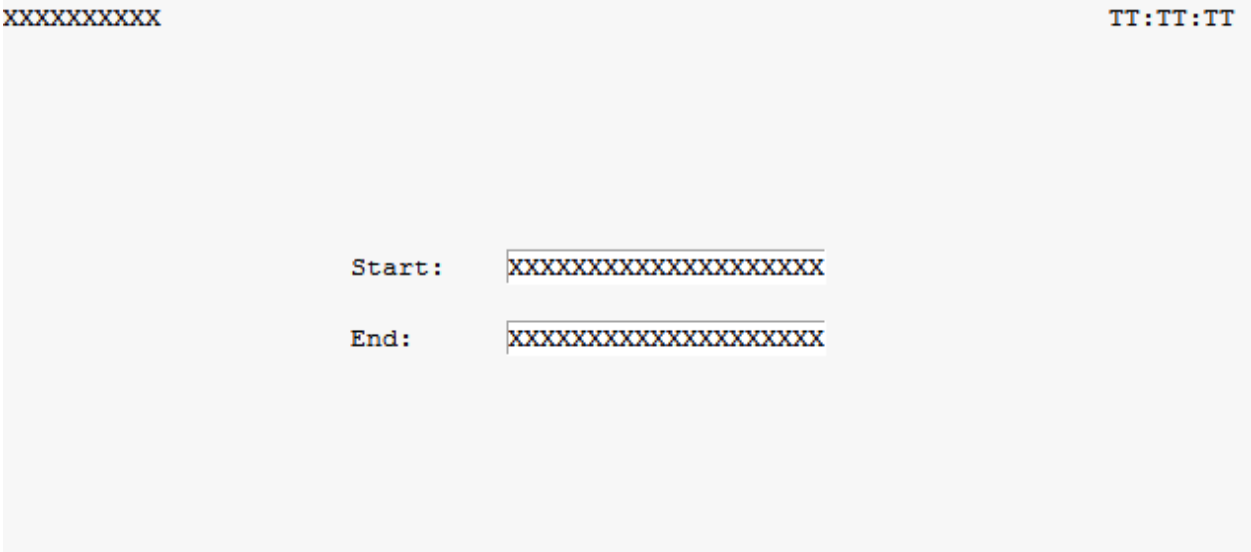
4 Press ENTER repeatedly until you return to the program editor, or press ESC.

5 Stow the program.

Designing a Map for User Input

You are now introduced to a different way of prompting the user for input. You will use the map editor to create a map which contains the same fields that you have previously defined in your program. A map is a separate object and is used to separate the user interface layout from the business logic of an application.

The map you will create now will look as follows:



The first line of the map contains system variables for the current date and time. There are two data fields (input fields) in which the user can specify a starting name and an ending name. The data fields are preceeded by text fields (labels).

The following steps are required for the above map:

- Creating a Map
- Defining Text Fields
- Specifying Labels for Text Fields
- Defining Data Fields
- Specifying Names and Attributes for Data Fields
- Adding System Variables
- Testing a Map
- Stowing a Map

Creating a Map

You will now invoke the map editor in which you will design your map.

Leave the program editor open in the background.

To create a map

1 In the library workspace, select the library which also contains your program (that is: select the **TUTORIAL** node).

2 From the context menu, choose **New Source > Map**.

Or:

Choose the following toolbar button:

An empty map editor window appears.

Defining Text Fields

You will now add two text fields (also called constants or labels) to the map.

To define the text fields

1 From the **Insert** menu, choose **Text Constant**.

Note: When the map editor is active, different menus are shown in the menu bar. The menus **Insert**, **Field** and **Map** are now shown (instead of the **Programs** menu which was visible when the program editor was active).

Or:

Choose the following toolbar button:

Note: By default, the toolbar containing this button is shown vertically to the right of the map editor window. A different toolbar was shown there previously when the program editor was active.

2 Move the mouse to the position in the map editor window at which you want to insert the text constant.

The mouse pointer changes showing a cross and the symbol for a text constant.

3 Press and hold down the mouse button.

4 Drag the mouse to the right until the field has the desired length. For our field a length of about 10 characters is sufficient.

While dragging the mouse, the current length is shown in the status bar of the application window. The format and the position in the map are also shown there. A text field always has the format `A` (alphanumeric).

5 Release the mouse button.

A text field with a default label is now shown. Handles indicate that the text field is selected. The mouse pointer still shows a cross and the symbol for a text constant and you can immediately draw another text field.

Note: If you want to exit this mode, just click any other position in the map editor.

6 Draw a second text field below the first text field.

If the field starts in the wrong column, you can move it by positioning the mouse pointer on this field, pressing and holding down the mouse button and then dragging the mouse to the desired position. When you release the mouse button, the normal mouse pointer is shown again.

Specifying Labels for Text Fields

The text fields you have just inserted do not yet have the correct labels. You will now specify them.

To specify the labels

1 Select the first text field and from the context menu, choose **Definition**.

Or:

Double-click the text field.

The existing text is selected.

2 Enter "Start:" as the label for the first text field and press ENTER.

The text field (for which you have previously defined a length of 10 characters) is automatically resized to the length of this string.

3 Repeat the above steps and enter "End:" as the label for the second text field.

Defining Data Fields

You will now add two data fields to the map. These are the input fields in which the user can specify the starting name and ending name.

You can define the data fields in two different ways: with the **Data Field** command where it is your responsibility to define the correct format and length of the data field, or with the **Import > Data Field** command where you simply select the data field from a list and where the correct format and length is automatically used. These two ways are described below.

To define a data field where you have to specify the length

1 From the **Insert** menu, choose **Data Field**.

Or:

Choose the following toolbar button:

2 Draw the data field at the right of the previously inserted text field for the starting name. Draw it in the same way as a text field. A length of 20 characters is required for the data field (if your field is too short or too long, a later exercise also explains how to modify the length). The data field is automatically filled with "X" characters.

To import a data field

1 From the **Insert** menu, choose **Import > Data Field**.

The **Import Data Field** dialog box appears.

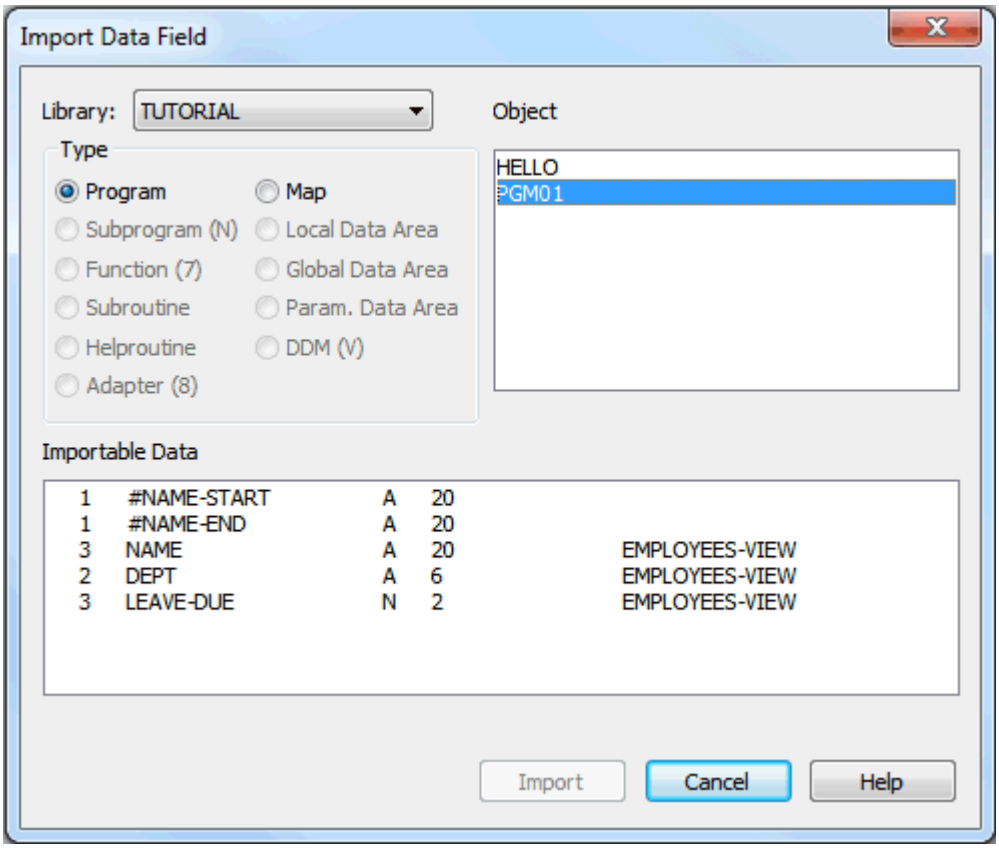
2 From the **Library** drop-down list box, select **TUTORIAL**.

3 Select the **Program** option button.

All programs that are currently defined in your library are now shown in the **Object** list box.

4 Select the program with the name `PGM01`.

The importable data fields are now shown at the bottom of the dialog box.



5 Select the field #NAME-END and choose the **Import** button.

The **Cancel** button in the **Import Data Field** dialog box is now labeled **Quit**.

6 Choose the **Quit** button to close the **Import Data Field** dialog box.

The data field is now shown at the top left of the map. It is filled with "X" characters. Handles indicate that the data field is selected.

7 Move the data field to the right of the previously defined text field for the end name. To do so position the mouse pointer on this field, press and hold down the mouse button, drag the mouse to the desired position and then release the mouse button.

Specifying Names and Attributes for Data Fields

The following applies only for the data field for the starting name which you have defined manually. It does not apply to the data field for the ending name which you have imported: When you create a new data field for a user-defined variable, Natural assigns a field name to it. This field name contains a number. You have to adjust the names of the newly created fields to the variable names defined in your program.

You will now make sure that the same names are used as in your program: #NAME-START and #NAME-END. The output of these fields (that is: the user input) will be passed to the corresponding user-defined variables in your program.

You will also make sure that the same attributes are defined for both #NAME-START and #NAME-END.

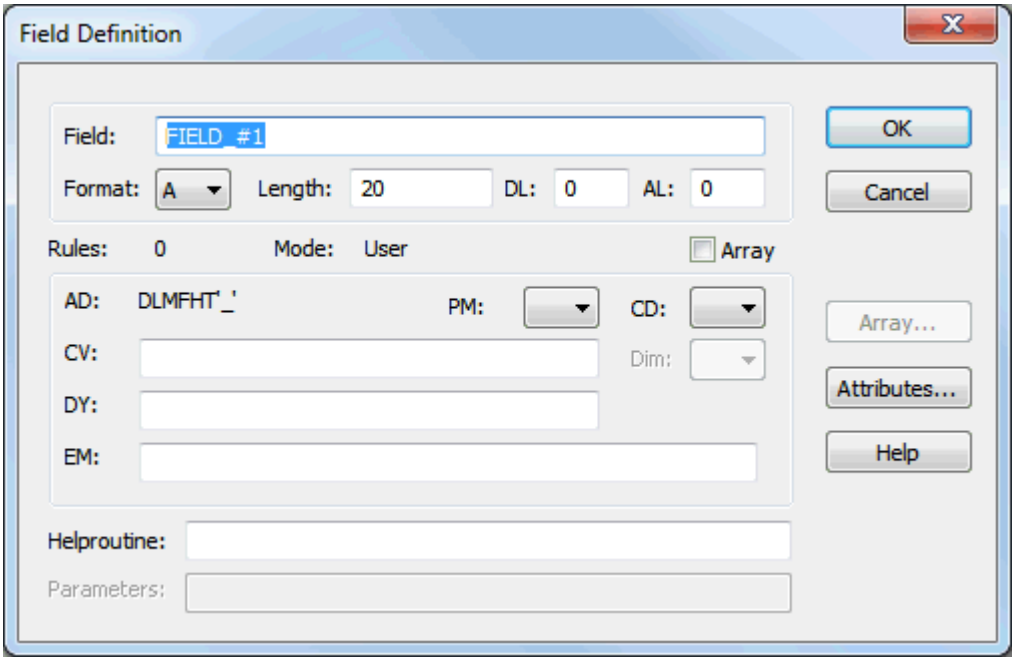
To define names and attributes for the data fields

1 Select the data field for the starting name and from the context menu, choose **Definition**.

Or:

Double-click the data field.

The **Field Definition** dialog box appears.



The **Field** text box shows the field name that has been assigned by Natural: `#FIELD_#1`.

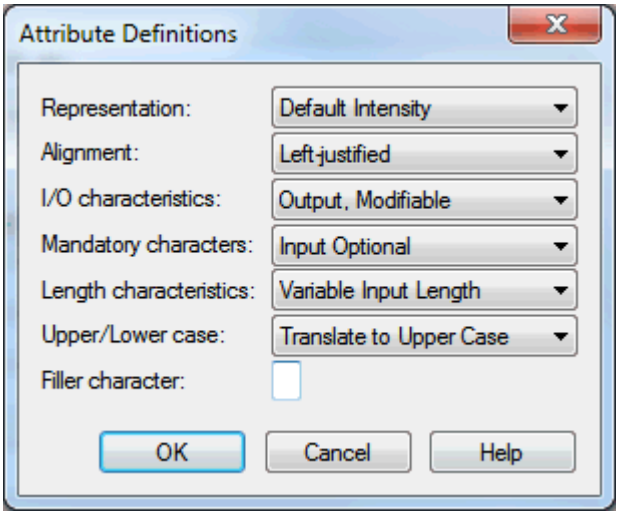
2 In the **Field** text box, enter "#NAME-START".

The format must be **A**. This is selected by default.

3 In the **Length** text box, enter "20" (if your field has a different length).

4 Choose the **Attributes** button.

The **Attribute Definitions** dialog box appears.



5 Make sure that **Output, Modifiable** has been selected in the **I/O characteristics** drop-down list box.

This defines the field as an output field which can be modified.

6 Make sure that **Translate to Upper Case** has been selected in the **Upper/Lower case** drop-down list box.

This enables the user to enter the name in lowercase letters. Until now, the names in the demo database could only be found when the names were specified completely in uppercase letters.

7 In the **Filler character** text box, enter an underscore (_) character.

A blank character is defined as the default filler character. Therefore, before you can enter an underscore in this field, you have to delete the blank.

The filler character is used to fill any empty positions in input fields in the map, allowing the user to see the exact position and length of a field when entering input.

8 Choose the **OK** button to close the **Attribute Definitions** dialog box.

9 Choose the **OK** button to close the **Field Definition** dialog box.

10 Repeat the above steps for the data field for the ending name. Make sure that the correct field name (#NAME-END), format (A) and length (20) are defined. Make sure that the same attribute definitions are used as for #NAME-START.

Adding System Variables

Natural system variables contain information about the current Natural session, such as the current library, user, or date and time. They may be referenced at any point within a Natural program. All system variables begin with an asterisk (*).

You will now add system variables for the date and time to the map. When the program is run, the current date and time will be displayed in the map.

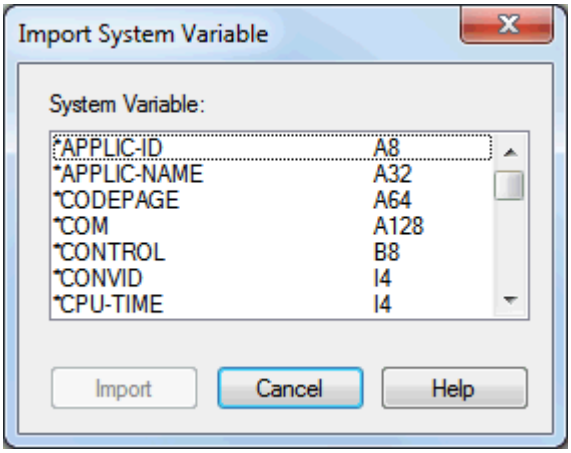
To add system variables

1 From the **Insert** menu, choose **Import > System Variable**.

Or:

Choose the following toolbar button:

The **Import System Variable** dialog box appears.



2 Scroll to `*DAT4I` and select it.

3 Scroll to `*TIMX`, press CTRL and select it.

4 Choose the **Import** button to import the selected variables.

The **Cancel** button in the **Import System Variable** dialog box is now labeled **Quit**.

5 Choose the **Quit** button to close the dialog box.

Both system variables have been placed at the top left of the map.

6 Select **TT:TT:TT** (that is: the system variable for the time) and move it to the end of the first line.

Note: You may have to resize the map editor window to see the end of the line.

Testing a Map

You will now test your map to check whether it works as intended.

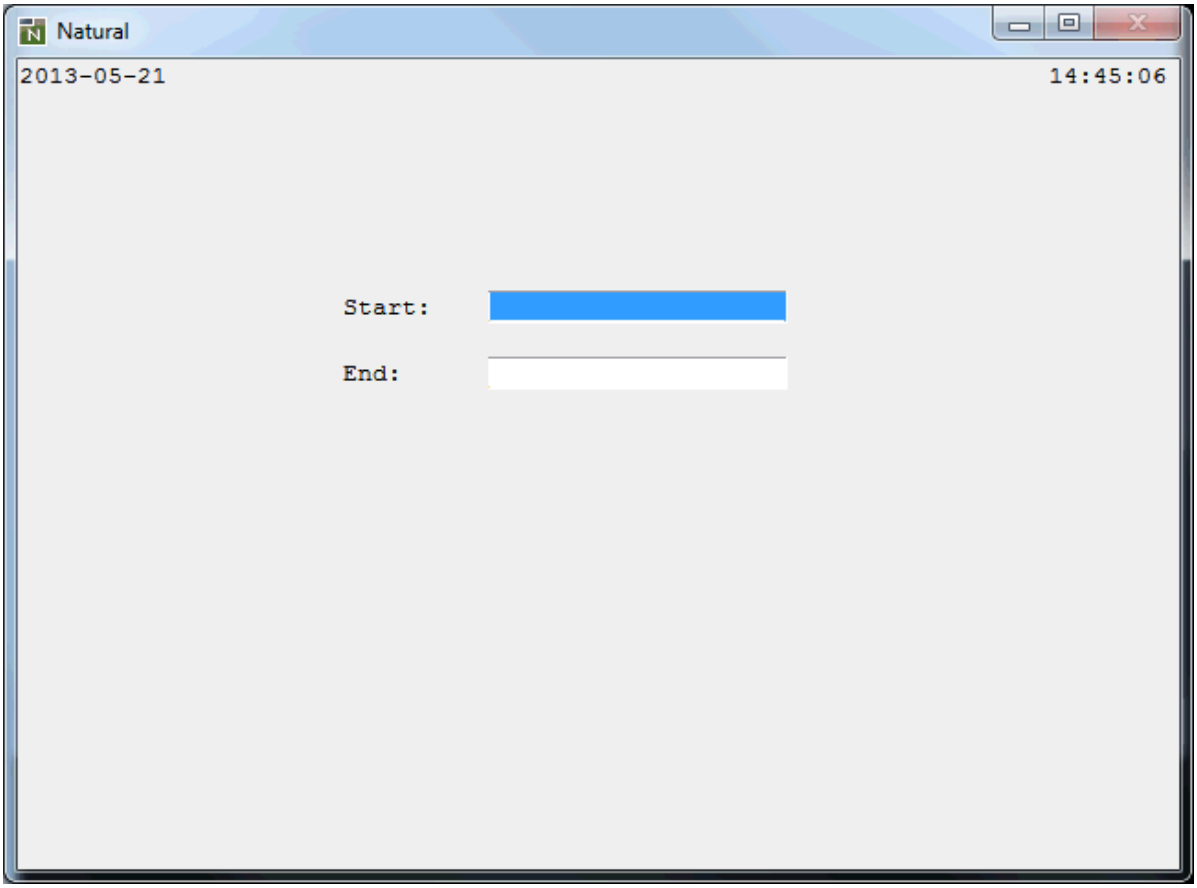
To test the map

1 From the **Object** menu, choose **Test**.

Or:

Choose the following toolbar button:

The following output is shown.



Note: You may have to resize the output window to see the time at the top right.

The input field for the starting name is automatically selected since it is the first input field in the map. Both input fields contain the filler character.

Note: When working in insert mode, the user has to delete the filler characters before it is possible to enter text. This is not necessary in overwrite mode.

2 Press ENTER to return to the map editor.

Stowing a Map

When the map has successfully been tested, you have to stow it so that it can be found by your program.

To stow the map

1 Stow the map in the same way as you stow a program.

2 When you are asked to specify a name for the map, enter "MAP01".

In the library workspace, a new node named **Maps** appears below the **TUTORIAL** node. This node contains the map you have just stowed.

Leave the map editor open for later modifications.

Invoking the Map from Your Program

Once a map has been stowed, it can be invoked by a Natural program using a `WRITE` or `INPUT` statement.

To invoke the map from your program

1 Return to the program editor.

If you cannot see the program editor (for example, because you have previously maximized the map editor window), you can return to an open program editor window by choosing the corresponding command for `PGM01` from the bottom of the **Window** menu.

You can also double-click the program `PGM01` in the library workspace (or when working with the keyboard, select it and press ENTER). When the program has previously been closed, it is opened again. When it it still open in the background, its editor window is brought back to the front.

2 Replace the previously defined `INPUT` lines with the following line:

```
INPUT USING MAP 'MAP01'
```

This will invoke the map you have just designed.

The map name must be enclosed in single quotation marks to distinguish the map from a user-defined variable.

Your program should now look as follows:

```
DEFINE DATA
LOCAL
  1 #NAME-START        (A20) INIT <"ADKINSON">
  1 #NAME-END          (A20) INIT <"BENNETT">
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 FULL-NAME
      3 NAME (A20)
    2 DEPT (A6)
    2 LEAVE-DATA
      3 LEAVE-DUE (N2)
END-DEFINE
*
INPUT USING MAP 'MAP01'
*
READ EMPLOYEES-VIEW BY NAME
  STARTING FROM #NAME-START
  ENDING AT #NAME-END
*
  DISPLAY NAME 3X DEPT 3X LEAVE-DUE
*
END-READ
*
END
```

3 Run the program.

Your map is now shown.

4 Press ENTER repeatedly until you return to the program editor, or press ESC.

5 Stow the program.

Ensuring that an Ending Name is Always Used

As your program is coded now, no data will not be found if an ending name is not specified.

You will now remove the initial values for the starting name and ending name; then the user always has to specify these names. To ensure that an ending name is always used, even if it has not been specified by the user, you will add a corresponding statement.

To use the ending name

1 In the `DEFINE DATA` block, remove the default values (`INIT`) for the fields `#NAME-START` and `#NAME-END` so that the corresponding lines look as follows:

```
1 #NAME-START        (A20)
1 #NAME-END          (A20)
```

2 Insert the following below `INPUT USING MAP 'MAP01'`:

```
IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
END-IF
```

When the `#NAME-END` field is blank (that is: when an ending name has not been entered by the user), the starting name is automatically used as the ending name.

Note: Instead of using the statement `MOVE #NAME-START TO #NAME-END` it is also possible to use the following variant of the `ASSIGN` or `COMPUTE` statement: `#NAME-END := #NAME-START`.

Your program should now look as follows:

```
DEFINE DATA
LOCAL
  1 #NAME-START        (A20)
  1 #NAME-END          (A20)
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 FULL-NAME
      3 NAME (A20)
    2 DEPT (A6)
    2 LEAVE-DATA
      3 LEAVE-DUE (N2)
END-DEFINE
*
INPUT USING MAP 'MAP01'
*
IF #NAME-END = ' ' THEN
  MOVE #NAME-START TO #NAME-END
END-IF
*
READ EMPLOYEES-VIEW BY NAME
  STARTING FROM #NAME-START
  ENDING AT #NAME-END
*
  DISPLAY NAME 3X DEPT 3X LEAVE-DUE
*
END-READ
*
END
```

3 Run the program.

4 In the resulting map, enter "JONES" in the field which prompts for a starting name and press ENTER.

Note: Since **Translate to Upper Case** has been specified for this field, it is now also possible to enter the name in lowercase letters.

In the resulting list, only the employees with the name "Jones" are now shown.

5 Press ENTER to return to the program editor.

6 Stow the program.

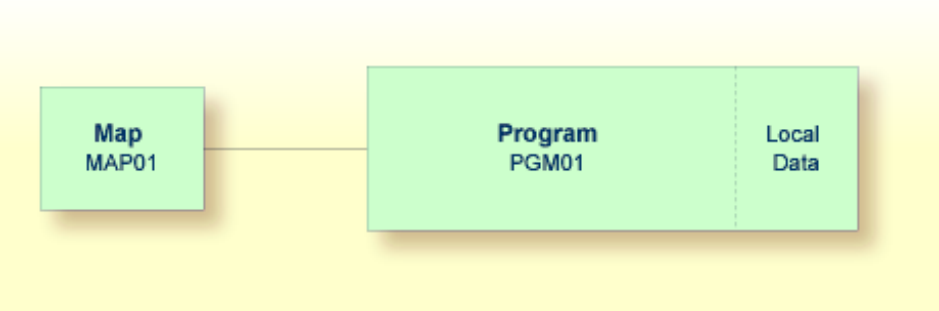
You can now proceed with the next exercises: ***Loops and Labels***.

7 Loops and Labels

- Allowing Repeated Usage 52
- Displaying a Message Indicating that Information was not Found 54

You will now enhance your program by adding loops and labels.

When you have completed the exercises below, your sample application will still consist of the same modules as in the previous chapter:



Allowing Repeated Usage

As it is now, the program terminates after it has displayed the map and has shown the list. So that the user can display a new employees list immediately, without restarting the program, you will now put the corresponding program code into a `REPEAT` loop.

To define a repeat loop

1 Insert the following below `END-DEFINE`:

```
RP1. REPEAT
```

`REPEAT` defines the start of the repeat loop. `RP1.` is a label which is used when leaving the repeat loop (this is defined below).

2 Define the end of the repeat loop by inserting the following before the `END` statement:

```
END-REPEAT
```

3 Insert the following below `INPUT USING MAP 'MAP01'`:

```
IF #NAME-START = '.' THEN
  ESCAPE BOTTOM (RP1.)
END-IF
```

The `IF` statement, which must be ended with `END-IF`, checks the content of the `#NAME-START` field. When a dot (.) is entered in this field, the `ESCAPE BOTTOM` statement is used to leave the loop. Processing will continue with the first statement following the loop (which is `END` in this case).

By assigning a label to the loop (here RP1.), you can refer to this specific loop in the ESCAPE BOTTOM statement. Since loops may be nested, you should specify which loop you want to leave. Otherwise, the program will only leave the innermost active loop.

Your program should now look as follows:

```
DEFINE DATA
LOCAL
  1 #NAME-START        (A20)
  1 #NAME-END          (A20)
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 FULL-NAME
      3 NAME (A20)
    2 DEPT (A6)
    2 LEAVE-DATA
      3 LEAVE-DUE (N2)
END-DEFINE
*
RP1. REPEAT
*
  INPUT USING MAP 'MAP01'
*
  IF #NAME-START = '.' THEN
    ESCAPE BOTTOM (RP1.)
  END-IF
*
  IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
  END-IF
*
  READ EMPLOYEES-VIEW BY NAME
    STARTING FROM #NAME-START
    ENDING AT #NAME-END
*
    DISPLAY NAME 3X DEPT 3X LEAVE-DUE
*
  END-READ
*
END-REPEAT
*
END
```

Note: For better readability, the content of the REPEAT loop has been indented.

4 Run the program.

5 In the resulting map, enter "JONES" in the field which prompts for a starting name and press ENTER.

In the resulting list, the employees with the name "Jones" are shown. Press ENTER. Due to the REPEAT loop, the map is shown again. Now you can also see that "JONES" has been entered as the ending name.

6 To leave the map, enter a dot (.) in the field which prompts for a starting name and press ENTER. Do not forget to delete the remaining characters of the name which is still shown in this field.

7 Stow the program.

Displaying a Message Indicating that Information was not Found

You will now define the message that is to be displayed when the user enters a starting name which cannot be found in the database.

To define the message that is to be displayed when the specified employee cannot be found

1 Add the label RD1. to the line containing the READ statement so that it looks as follows:

```
RD1. READ EMPLOYEES-VIEW BY NAME
```

2 Insert the following below END-READ:

```
IF *COUNTER (RD1.) = 0 THEN
   REINPUT 'No employees meet your criteria.'
END-IF
```

To check the number of records found in the READ loop, the system variable *COUNTER is used. If its contents equals 0 (that is: an employee with the specified name has not been found), the message defined with the REINPUT statement is displayed at the bottom of your map.

To identify the READ loop, you assign a label to it (here RD1.). Since a complex database access program can contain many loops, you have to specify the loop to which you refer.

Your program should now look as follows:

```
DEFINE DATA
LOCAL
  1 #NAME-START        (A20)
  1 #NAME-END          (A20)
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 FULL-NAME
      3 NAME (A20)
    2 DEPT (A6)
    2 LEAVE-DATA
      3 LEAVE-DUE (N2)
```

```
END-DEFINE
*
RP1. REPEAT
*
  INPUT USING MAP 'MAP01'
*
  IF #NAME-START = '.' THEN
    ESCAPE BOTTOM (RP1.)
  END-IF
*
  IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
  END-IF
*
  RD1. READ EMPLOYEES-VIEW BY NAME
    STARTING FROM #NAME-START
    ENDING AT #NAME-END
*
    DISPLAY NAME 3X DEPT 3X LEAVE-DUE
*
  END-READ
*
  IF *COUNTER (RD1.) = 0 THEN
    REINPUT 'No employees meet your criteria.'
  END-IF
*
END-REPEAT
*
END
```

3 Run the program.

4 In the resulting map, enter a starting name which is not defined in the demo database (for example, "XYZ") and press ENTER.

Your message should now appear in the map.

5 To leave the map, enter a dot (.) in the field which prompts for a starting name and press ENTER. Do not forget to delete the remaining characters of the name which is still shown in this field.

Or:

Press ESC.

6 Stow the program.

You can now proceed with the next exercises: ***Inline Subroutines***.

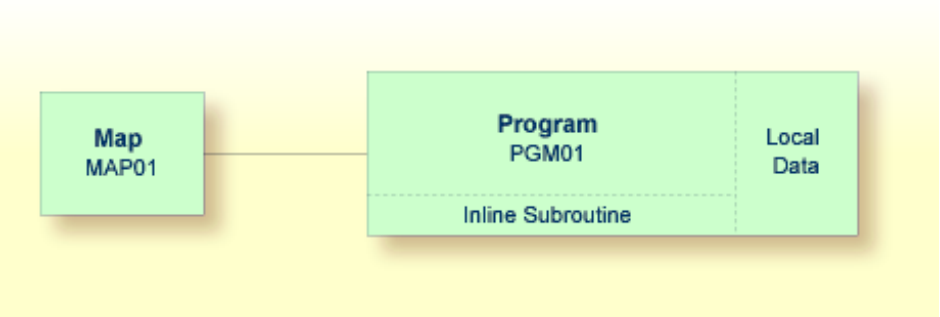
8 Inline Subroutines

- Defining the Inline Subroutine 58
- Performing the Inline Subroutine 59

Natural distinguishes two types of subroutines: inline subroutines which are defined directly in the program and external subroutines which are stored as separate objects outside the program (this is explained later in this tutorial).

You will now add an inline subroutine to your program which moves an asterisk (*) to the new user-defined variable named #MARK. This subroutine will be invoked when an employee has 20 days of leave or more.

When you have completed the exercises below, your sample application will be structured as follows:



Defining the Inline Subroutine

You will now add the subroutine to your program.

To define the subroutine

1 Insert the following below the user-defined variable #NAME-END:

```
1 #MARK               (A1)
```

This variable will be used by the subroutine. Therefore, it has to be defined first.

2 To define the subroutine, insert the following before the END statement:

```
DEFINE SUBROUTINE MARK-SPECIAL-EMPLOYEES
  MOVE '*' TO #MARK
END-SUBROUTINE
```

When performed, this subroutine moves an asterisk (*) to #MARK.

Note: Instead of using the statement MOVE '*' TO #MARK it is also possible to use the following variant of the ASSIGN or COMPUTE statement: #MARK := '*'.

3 Modify the DISPLAY statement as follows:

```
DISPLAY NAME 3X DEPT 3X LEAVE-DUE 3X '>=20' #MARK
```

This displays a new column in your output. Its heading is ">=20". The column will contain an asterisk (*) if the corresponding employee has 20 days of leave or more.

Performing the Inline Subroutine

Now that you have defined the inline subroutine, you can specify the corresponding code for performing it.

To perform the inline subroutine

1 Insert the following before the DISPLAY statement:

```
IF LEAVE-DUE >= 20 THEN
  PERFORM MARK-SPECIAL-EMPLOYEES
ELSE
  RESET #MARK
END-IF
```

When an employee is found who has 20 days of leave or more, the new subroutine named MARK-SPECIAL-EMPLOYEES is performed. When an employee has less than 20 days of leave, the content of #MARK is reset to blank.

Your program should now look as follows:

```
DEFINE DATA
LOCAL
  1 #NAME-START        (A20)
  1 #NAME-END          (A20)
  1 #MARK              (A1)
  1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
    2 FULL-NAME
      3 NAME (A20)
    2 DEPT (A6)
    2 LEAVE-DATA
      3 LEAVE-DUE (N2)
END-DEFINE
*
RP1. REPEAT
*
  INPUT USING MAP 'MAP01'
*
  IF #NAME-START = '.' THEN
```

```
    ESCAPE BOTTOM (RP1.)
  END-IF
*
  IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
  END-IF
*
  RD1. READ EMPLOYEES-VIEW BY NAME
    STARTING FROM #NAME-START
    ENDING AT #NAME-END
*
    IF LEAVE-DUE >= 20 THEN
      PERFORM MARK-SPECIAL-EMPLOYEES
    ELSE
      RESET #MARK
    END-IF
*
    DISPLAY NAME 3X DEPT 3X LEAVE-DUE 3X '>=20' #MARK
*
  END-READ
*
  IF *COUNTER (RD1.) = 0 THEN
    REINPUT 'No employees meet your criteria.'
  END-IF
*
END-REPEAT
*
DEFINE SUBROUTINE MARK-SPECIAL-EMPLOYEES
  MOVE '*' TO #MARK
END-SUBROUTINE
*
END
```

2 Run the program.

3 In the resulting map, enter "JONES" and press ENTER.

The list of employees should now contain the additional column.

4 Press ESC to close the output window.

5 Stow the program.

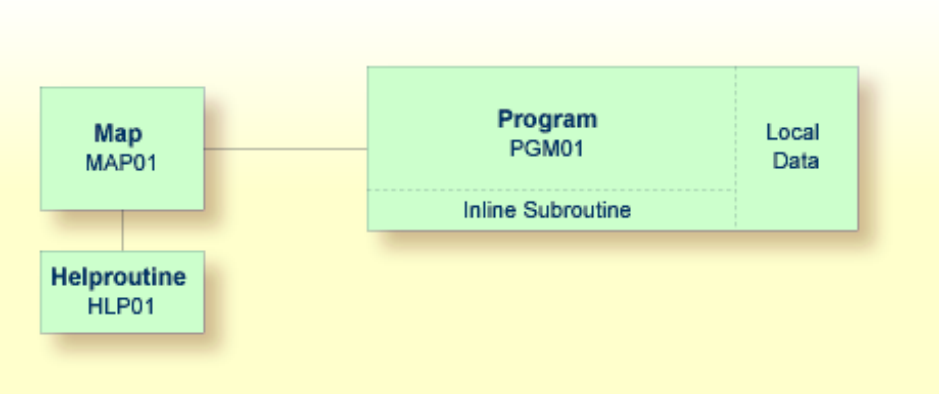
You can now proceed with the next exercises: ***Processing Rules and Helproutines***.

9 Processing Rules and Helproutines

- Defining a Processing Rule 62
- Defining a Helproutine 64

Processing rules and helproutines are defined for fields in a map.

When you have completed the exercises below, your sample application will consist of the following modules (a processing rule cannot be defined as a separate module; it is always part of a map):



Defining a Processing Rule

You will now define the message that is to be displayed when the user presses ENTER without specifying a starting name.

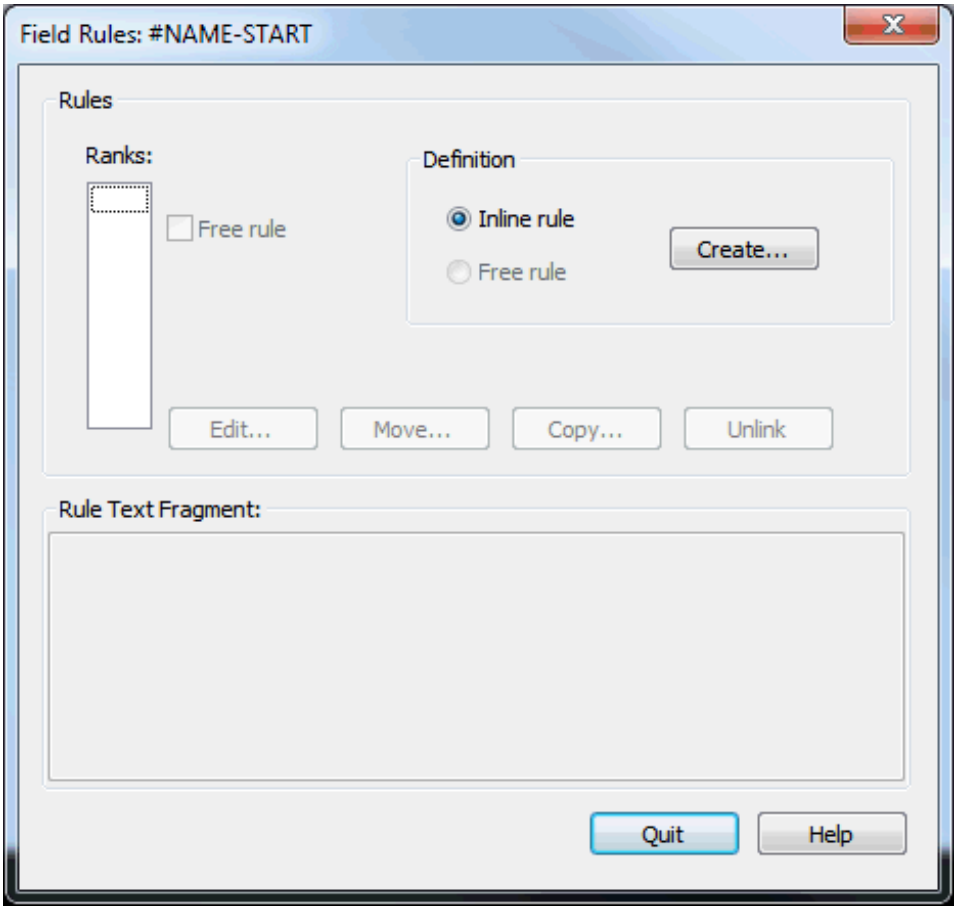
To define a processing rule

1 Return to the map editor.

2 Select the input field for the starting name.

3 From the context menu, choose **Rules**.

The **Field Rules** dialog box for the field #NAME-START appears.



4 Choose the **Create** button.

An empty editor window appears.

5 Enter the following processing rule:

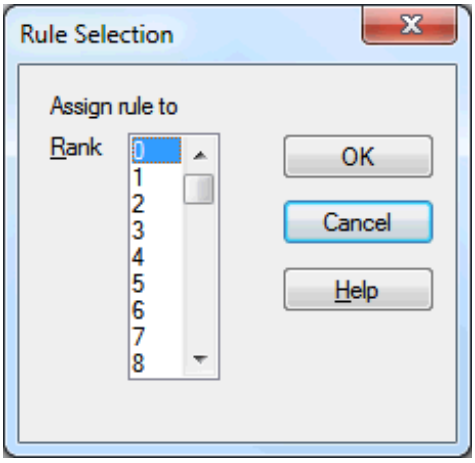
```
IF & = ' ' THEN
  REINPUT 'Please enter a starting name.'
  MARK *&
END-IF
```

The ampersand (&) in the processing rule will dynamically be replaced with the name of the field. In this case, it will be replaced with `#NAME-START`. If `#NAME-START` is blank, the message defined with the `REINPUT` statement is displayed.

`MARK` is an option of the `REINPUT` statement. Its syntax is `MARK *fieldname`. `MARK` specifies the field in which the cursor is to be placed when the `REINPUT` statement is executed. In this case, the cursor will be placed in the `#NAME-START` field.

6 Save the content of the editor window.

The **Rule Selection** dialog box appears.



7 In the **Rank** list box, select **1** and choose the **OK** button.

The rank defines the sequence in which the rules for the different fields are to be processed. All rules with rank 1 are processed first, followed by those with rank 2, etc.

8 Close the editor window in which you have entered the processing rule.

9 Test the map.

10 In the resulting output, enter any starting name and press ENTER.

The output window is closed.

11 Test the map once more. Do not enter a name and press ENTER.

The message defined with the processing rule should now appear in the map.

12 To leave the output window, enter a dot (.) in the field which prompts for a starting name and press ENTER.

13 Stow the map.

Defining a Helproutine

A helproutine is displayed when the user presses the help key when the cursor is on the input field for the starting name.

You will first define the helproutine and then associate it with a specific field.

To create a helproutine

1 In the library workspace, select the library which also contains your program (that is: select the **TUTORIAL** node).

2 From the context menu, choose **New Source > Helproutine**.

An empty editor appears.

3 Enter the following:

```
WRITE 'Type the name of an employee'
END
```

4 Stow the helproutine.

The **Stow As** dialog box appears.

5 Enter "HLP01" as the name of the helproutine.

6 Choose the **OK** button.

In the library workspace, a new node named **Helproutines** appears below the **TUTORIAL** node. This node contains the helproutine you have just stowed.

7 Close the editor window in which you have entered the helproutine.

To associate the helproutine with a field on the map

1 Return to the map editor.

2 Select the data field for the starting name.

3 From the context menu, choose **Definition**.

Or:

Double-click the data field.

The **Field Definition** dialog box appears.

4 In the **Helproutine** text box, enter "'HLP01'" (including the single quotation marks).

This is the name under which you have saved your helproutine.

5 Choose the **OK** button.

6 Test the map.

7 In the resulting output, enter a question mark (?) in the input field for the starting name.

The help text you have defined is shown in a separate window.

8 Press ENTER to close this window.

9 To leave the map, enter a dot (.) in the field which prompts for a starting name and press ENTER.

10 Stow the map.

11 Close the map editor window.

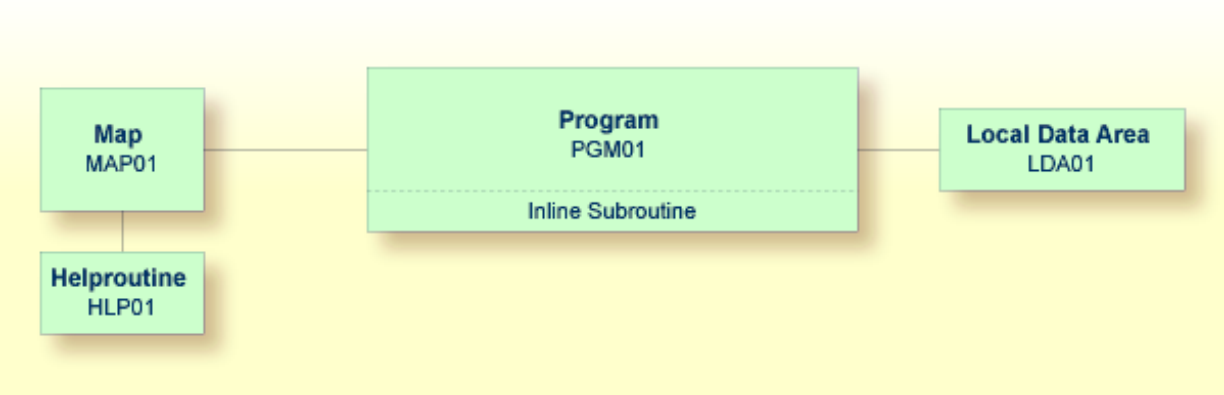
You can now proceed with the next exercises: ***Local Data Areas***.

10 Local Data Areas

- Creating a Local Data Area 68
- Defining Data Fields 69
- Importing the Required Data Fields from a DDM 72
- Referencing the Local Data Area from Your Program 73

Currently, the fields used by your program are defined within the `DEFINE DATA` statement in the program itself. It is also possible, however, to place the field definitions in a local data area (LDA) outside the program, with the program's `DEFINE DATA` statement referencing this local data area by name. For reusability and for a clear application structure, it is usually better to define fields in data areas outside the programs.

You will now relocate the information from the `DEFINE DATA` statement to a local data area. When you have completed the exercises below, your sample application will consist of the following modules:



Creating a Local Data Area

You will now invoke the data area editor in which you will specify the required fields.

To create a local data area

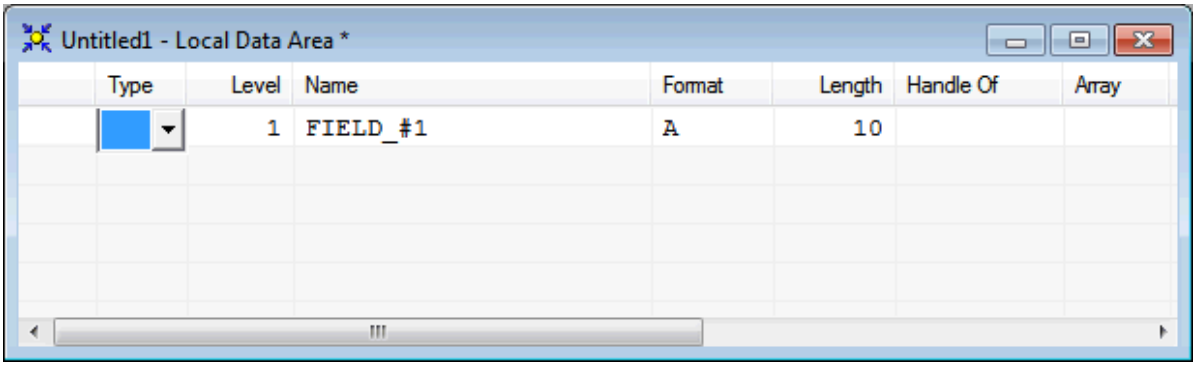
1 In the library workspace, select the library which also contains your program (that is: select the **TUTORIAL** node).

2 From the context menu, choose **New Source > Local Data Area**.

Or:

Choose the following toolbar button:

An editor window appears.



Level, name, format and length are automatically preset in the first row.

Defining Data Fields

You will now define the following fields:

Level	Name	Format	Length
1	#NAME-START	A	20
1	#NAME-END	A	20
1	#MARK	A	1

These are the user-defined variables which you have previously defined in the `DEFINE DATA` statement.

You can define the data fields in two different ways: manually in the editor window where it is your responsibility to define the correct format and length of the data field, or with the **Import** command where you simply select the data fields from a list and where the correct format and length is automatically used. These two ways are described below.

The setting of the following toolbar button in the Data Area Editor Insert toolbar indicates the insert position. When the toolbar button appears pressed, the new field is inserted after the selected field (this state is assumed in this tutorial); otherwise, the new field is inserted before the selected field.

To define a data field manually in the editor window

1 Make sure that the **Level** column in the first row contains the preset value "1".
2 Change the preset value in the **Name** column of the first row to "#NAME-START".
3 Make sure that the **Format** column in the first row contains the preset value "A".
4 Change the preset value in the **Length** column of the first row to "20".

To import data fields from a program

1 In the editor, select the row which has been created for `#NAME-START`.

Note: If required, press ENTER or ESC to switch from single-cell selection to full-row selection mode.

2 From the context menu, choose **Import**.

Or:

Choose the following toolbar button:

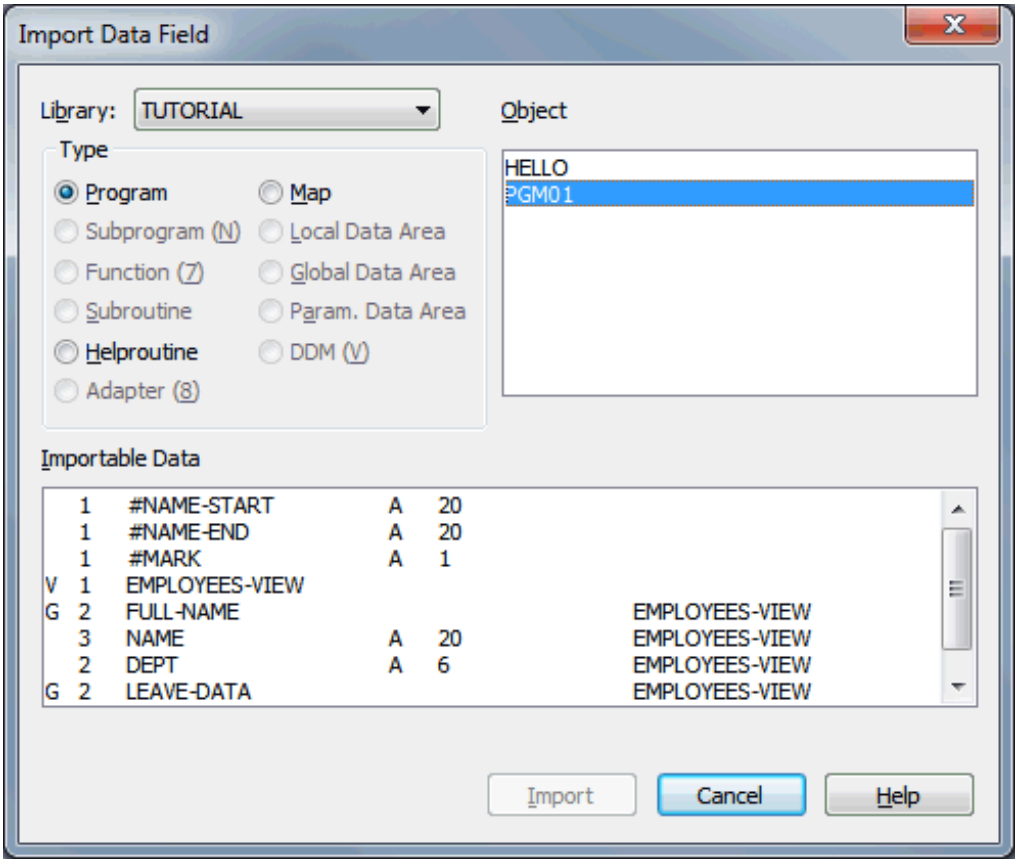
The **Import Data Field** dialog box appears.

3 Make sure that **TUTORIAL** is selected in the **Library** drop-down list box.
4 Select the **Program** check box.

All programs that are currently defined in your library are now shown in the **Object** list box.

5 Select the program with the name `PGM01`.

The importable data fields are now shown at the bottom of the dialog box.



6 Press CTRL and select the following fields in the **Importable Data** list box:

#NAME-END
#MARK

7 Choose the **Import** button.

The **Cancel** button in the **Import Data Field** dialog box is now labeled **Quit**.

8 Choose the **Quit** button to close the **Import Data Field** dialog box.

The fields #NAME-END and #MARK are now shown below the #NAME-START field in the editor window.

Importing the Required Data Fields from a DDM

You will now import the same data fields which you have previously defined in the program's `DEFINE DATA` statement. The fields are read directly from a Natural data view into the data area editor. A data view references database fields defined in a data definition module (DDM).

In the data area editor, the imported data fields will be inserted below the currently selected data field.

To import data fields from a DDM

1 In the editor, select the row containing `#MARK`.

2 From the context menu, choose **Import**.

The **Import Data Field** dialog box appears.

3 From the **Library** drop-down list box, select **SYSEXDDM**.

The **DDM** option button is selected.

4 In the **Object** list box, select the sample DDM with the name **EMPLOYEES**.

5 Press CTRL and select the following fields in the **Importable Data** list box:

```
PERSONNEL-ID
FULL-NAME
NAME
DEPT
LEAVE-DATA
LEAVE-DUE
```

Note: The field `PERSONNEL-ID` will be used later when you create the subprogram.

6 Choose the **Import** button.

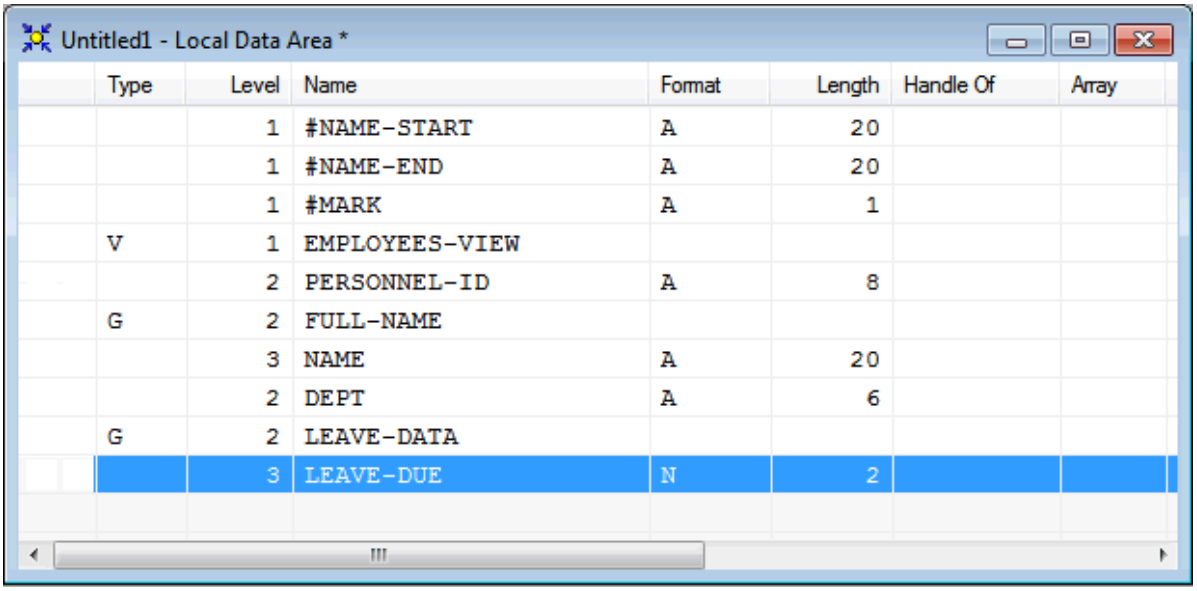
The **View Definition** dialog box appears.

7 Specify the same name that you have previously defined for your view (that is: `EMPLOYEES-VIEW`).

8 Choose the **OK** button.

9 Choose the **Quit** button to close the **Import Data Field** dialog box.

The local data area should now look as follows:



Type	Level	Name	Format	Length	Handle Of	Array
	1	#NAME-START	A	20		
	1	#NAME-END	A	20		
	1	#MARK	A	1		
V	1	EMPLOYEES-VIEW				
	2	PERSONNEL-ID	A	8		
G	2	FULL-NAME				
	3	NAME	A	20		
	2	DEPT	A	6		
G	2	LEAVE-DATA				
	3	LEAVE-DUE	N	2		

The Type column indicates the type of the variable. The view is indicated by a "V" and each group is indicated by a "G".

Note: When the **Expand/Collapse** check box has been selected in the data area editor options, expand/collapse toggles are shown in the first column (for the view and each group).

10 Stow the local data area.

11 When you are asked to specify a name for the local data area, enter "LDA01".

In the library workspace, a new node named **Local Data Areas** appears below the **TUTORIAL** node. This node contains the local data area you have just stowed.

Referencing the Local Data Area from Your Program

Once a local data area has been stowed, it can be referenced by a Natural program.

You will now change the `DEFINE DATA` statement your program so that it uses the local data area that you have just defined.

Leave the data area editor open in the background.

To use the local data area in your program

1 Return to the program editor.

2 In the `DEFINE DATA` statement, delete all variables between `LOCAL` and `END-DEFINE`.

3 Add a reference to your local data area by modifying the `LOCAL` line as follows:

```
LOCAL USING LDA01
```

Your program should now look as follows:

```
DEFINE DATA
  LOCAL USING LDA01
END-DEFINE
*
RP1. REPEAT
*
  INPUT USING MAP 'MAP01'
*
  IF #NAME-START = '.' THEN
    ESCAPE BOTTOM (RP1.)
  END-IF
*
  IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
  END-IF
*
  RD1. READ EMPLOYEES-VIEW BY NAME
    STARTING FROM #NAME-START
    ENDING AT #NAME-END
*
    IF LEAVE-DUE >= 20 THEN
      PERFORM MARK-SPECIAL-EMPLOYEES
    ELSE
      RESET #MARK
    END-IF
*
    DISPLAY NAME 3X DEPT 3X LEAVE-DUE 3X '>=20' #MARK
*
  END-READ
*
  IF *COUNTER (RD1.) = 0 THEN
    REINPUT 'No employees meet your criteria.'
  END-IF
*
END-REPEAT
*
DEFINE SUBROUTINE MARK-SPECIAL-EMPLOYEES
  MOVE '*' TO #MARK
END-SUBROUTINE
*
END
```

4 Run the program.

5 To confirm that the results are the same as before (when the `DEFINE DATA` statement did not reference a local data area), enter "JONES" as the starting name and press ENTER.

6 Press ESC to close the output window.

7 Stow the program.

You can now proceed with the next exercises: ***Global Data Areas***.

11 Global Data Areas

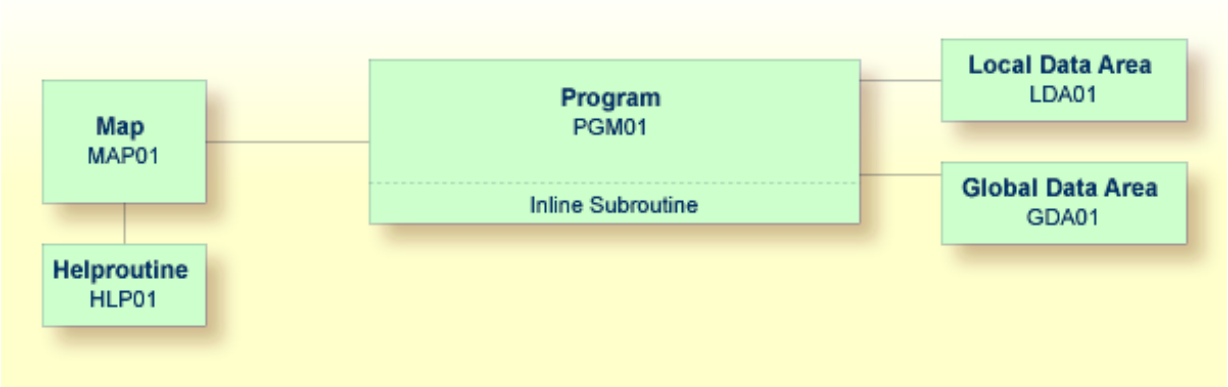
- Creating a Global Data Area from an Existing Local Data Area 78
- Adapting the Local Data Area 80
- Referencing the Global Data Area from Your Program 80

Data defined in a global data area (GDA) can be shared by multiple programs, external subroutines and helproutines.

Any modification of a data element value in a global data area affects all Natural objects that reference this global data area. Therefore, if you change the source of a global data area, you have to stow all previously created Natural objects that reference this global data area once more. The sequence in which objects are stowed is important. You must first stow the global data area and then the program. If you stow the program first and then the global data area, the program cannot be stowed because new elements in the global data area cannot be found.

You will now create a global data area which will be shared by your program and an external subroutine that you will create later. As the basis for your global data area, you will use some of the information from the local data area you have just created.

When you have completed the exercises below, your sample application will consist of the following modules:



Creating a Global Data Area from an Existing Local Data Area

You can create a new data area from an existing data area by editing it and saving it under a different name and with a different type. The original data area remains unchanged, and the new data area can be edited. Since the fields #NAME-START and #NAME-END are not required in the global data area, you will remove them.

Note: It is also possible to create a global data area by choosing **New > Global Data Area** from the **Object** menu.

To create the global data area

1 Return to your local data area.

2 From the **Object** menu, choose **Save As**.

The **Save As** dialog box appears.

3 Specify "GDA01" as the name for the global data area.

4 Make sure that the library is selected which also contains your program (that is: the **TUTORIAL** node).

5 From the **Type** drop-down list box, choose **Global**.

6 Choose the **OK** button.

The new name and type are now shown in the title bar of the editor window. In the library workspace, the new global data area is shown in the **Global Data Areas** node.

7 Press CTRL and select the following fields:

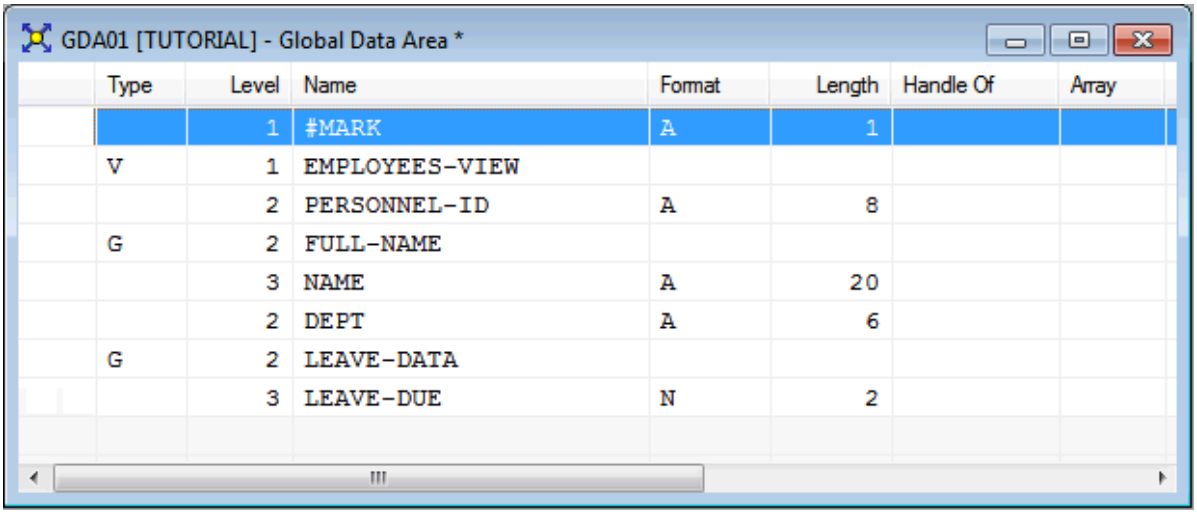
```
#NAME-START
#NAME-END
```

8 From the context menu, choose **Delete**.

Or:

Press DEL.

The global data area should now look as follows:



Type	Level	Name	Format	Length	Handle Of	Array
	1	#MARK	A	1		
V	1	EMPLOYEES-VIEW				
	2	PERSONNEL-ID	A	8		
G	2	FULL-NAME				
	3	NAME	A	20		
	2	DEPT	A	6		
G	2	LEAVE-DATA				
	3	LEAVE-DUE	N	2		

9 Stow the global data area.

The global data area can now be found by your program and by the external subroutine that you will define later.

10 Close the editor window for the global data area.

Adapting the Local Data Area

The fields contained in the global data area are no longer required in the local data area. Therefore, you will now remove all fields except #NAME-START and #NAME-END from the local data area.

To remove the fields

1 In the library workspace, select the local data area LDA01, and from the context menu, choose **Open.**

Or:

In the library workspace, double-click the local data area LDA01.

2 In the resulting data area editor, select all fields except #NAME-START and #NAME-END.

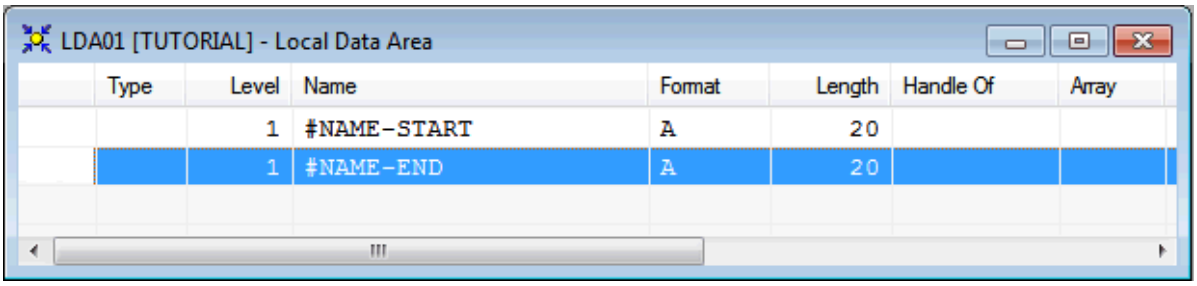
3 From the context menu, choose **Delete**.

Or:

Press DEL.

4 Stow the modified local data area.

The local data area should now look as follows:



Referencing the Global Data Area from Your Program

Once a global data area has been stowed, it can be referenced by a Natural program.

You will now change the DEFINE DATA statement in your program so that it also uses the global data area that you have just defined.

Leave the data area editor open in the background.

To use the global data area in your program

1 Return to the program editor.

2 Insert the following in the line above `LOCAL USING LDA01`:

```
GLOBAL USING GDA01
```

A global data area must always be defined before a local data area. Otherwise, an error occurs.

Your program should now look as follows:

```
DEFINE DATA
  GLOBAL USING GDA01
  LOCAL USING LDA01
END-DEFINE
*
RP1. REPEAT
*
  INPUT USING MAP 'MAP01'
*
  IF #NAME-START = '.' THEN
    ESCAPE BOTTOM (RP1.)
  END-IF
*
  IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
  END-IF
*
  RD1. READ EMPLOYEES-VIEW BY NAME
    STARTING FROM #NAME-START
    ENDING AT #NAME-END
*
    IF LEAVE-DUE >= 20 THEN
      PERFORM MARK-SPECIAL-EMPLOYEES
    ELSE
      RESET #MARK
    END-IF
*
    DISPLAY NAME 3X DEPT 3X LEAVE-DUE 3X '>=20' #MARK
*
  END-READ
*
  IF *COUNTER (RD1.) = 0 THEN
    REINPUT 'No employees meet your criteria.'
  END-IF
*
END-REPEAT
*
DEFINE SUBROUTINE MARK-SPECIAL-EMPLOYEES
  MOVE '*' TO #MARK
```

```
END-SUBROUTINE
*
END
```

3 Run the program.

4 To confirm that the results are the same as before (when the `DEFINE DATA` statement did not reference a global data area), enter "JONES" as the starting name and press ENTER.

5 Press ESC to close the output window.

6 Stow the program.

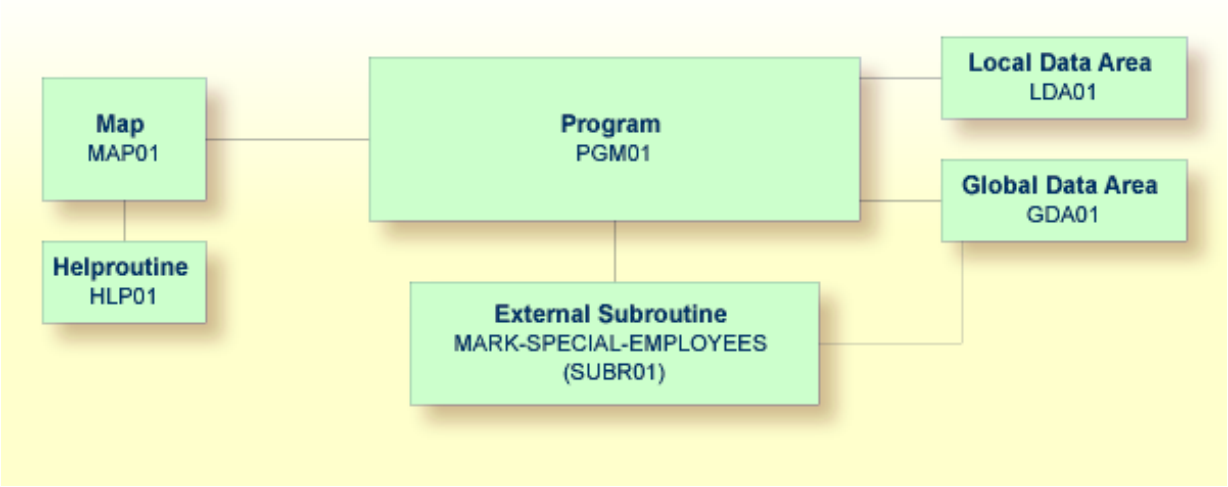
You can now proceed with the next exercises: ***External Subroutines***.

12 External Subroutines

- Creating an External Subroutine 84
- Referencing the External Subroutine from Your Program 85

Until now, the subroutine `MARK-SPECIAL-EMPLOYEES` has been defined within the program using a `DEFINE SUBROUTINE` statement. You will now define the subroutine as a separate object external to the program.

When you have completed the exercises below, your sample application will consist of the following modules:



Creating an External Subroutine

You will now invoke an editor in which you will specify the code for the external subroutine.

The `DEFINE SUBROUTINE` statement of the external subroutine is coded in the same way as the inline subroutine in the program.

To create an external subroutine

1 In the library workspace, select the library which also contains your program (that is: select the **TUTORIAL** node).

2 From the context menu, choose **New Source > Subroutine**.

An empty editor window appears.

3 Enter the following:

```
DEFINE DATA
  GLOBAL USING GDA01
  LOCAL USING LDA01
END-DEFINE
*
DEFINE SUBROUTINE MARK-SPECIAL-EMPLOYEES
  MOVE '*' TO #MARK
END-SUBROUTINE
*
END
```

4 Stow the subroutine.

The **Stow As** dialog box appears.

5 Enter "SUBR01" as the name of the external subroutine and choose the **OK** button.

In the library workspace, the new external subroutine is shown in the **Subroutines** node. In logical view, the name of the subroutine is shown as defined in the code: `MARK-SPECIAL-EMPLOYEES`. In all other views, the name `SUBR01` is shown.

6 Close the editor window in which you have entered the external subroutine.

Referencing the External Subroutine from Your Program

The `PERFORM` statement invokes both internal and external subroutines. When an internal subroutine is not found in the program, Natural automatically tries to perform an external subroutine with the same name. Note that Natural looks for the name that has been defined in the subroutine code (that is: the subroutine name), not for the name that you have specified when saving the subroutine (that is: the Natural object name).

Now that you have defined an external subroutine, you have to remove the inline subroutine (which has the same name as the external subroutine) from your program.

To use the external subroutine in your program

1 Return to the program editor.

2 Remove the following lines:

```
DEFINE SUBROUTINE MARK-SPECIAL-EMPLOYEES
  MOVE '*' TO #MARK
END-SUBROUTINE
```

Your program should now look as follows:

```
DEFINE DATA
  GLOBAL USING GDA01
  LOCAL USING LDA01
END-DEFINE
*
RP1. REPEAT
*
  INPUT USING MAP 'MAP01'
*
  IF #NAME-START = '.' THEN
    ESCAPE BOTTOM (RP1.)
  END-IF
*
  IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
  END-IF
*
  RD1. READ EMPLOYEES-VIEW BY NAME
    STARTING FROM #NAME-START
    ENDING AT #NAME-END
*
    IF LEAVE-DUE >= 20 THEN
      PERFORM MARK-SPECIAL-EMPLOYEES
    ELSE
      RESET #MARK
    END-IF
*
    DISPLAY NAME 3X DEPT 3X LEAVE-DUE 3X '>=20' #MARK

  END-READ
*
  IF *COUNTER (RD1.) = 0 THEN
    REINPUT 'No employees meet your criteria.'
  END-IF
*
END-REPEAT
*
END
```

3 Run the program.

4 Enter "JONES" as the starting name and press ENTER.

The resulting list should still show an asterisk for each employee who has 20 days of leave and more.

5 Press ESC to close the output window.

6 Stow the program.

You can now proceed with the next exercises: ***Subprograms***.

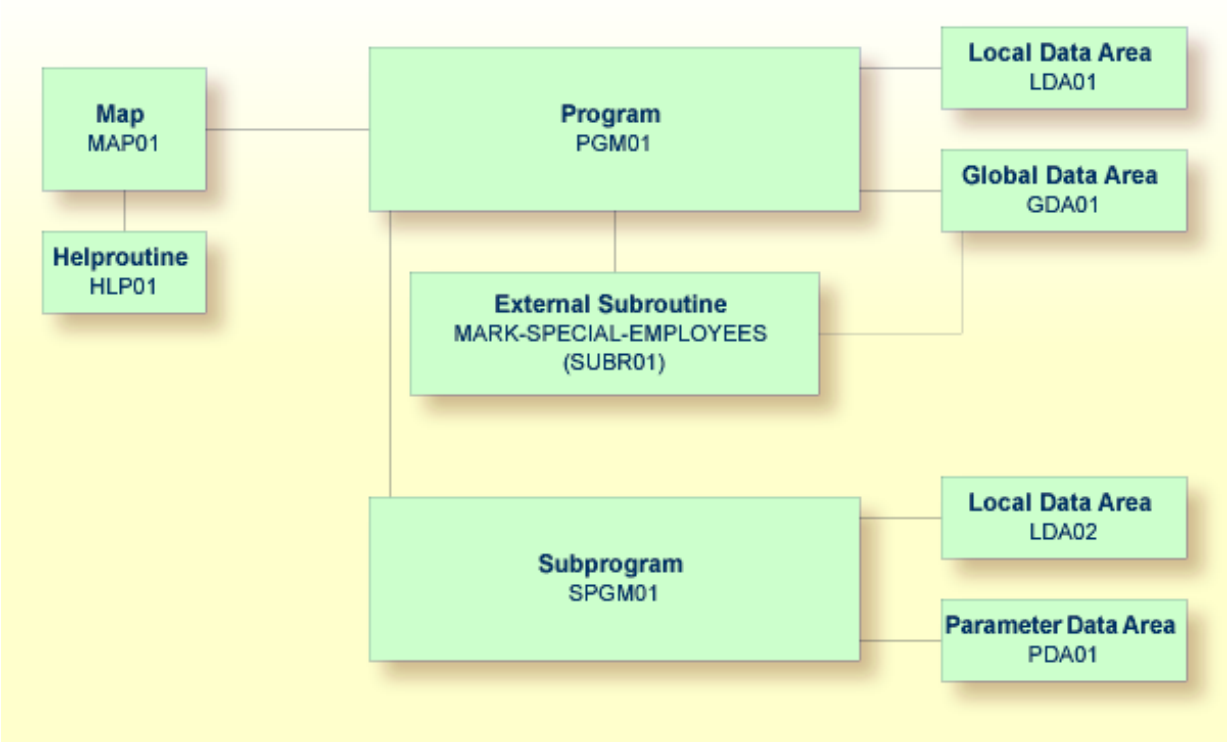
13 Subprograms

- Modifying the Local Data Area 90
- Creating a Parameter Data Area from an Existing Local Data Area 92
- Creating Another Local Data Area Containing a Different View 93
- Creating a Subprogram 94
- Referencing the Subprogram from Your Program 95

You will now expand your program to include a CALLNAT statement that invokes a subprogram. In the subprogram, the employees identified from the main program will be the basis of a FIND request to the VEHICLES file which is also part of the demo database. As a result, your output will contain vehicles information from the subprogram as well as employees information from the main program.

The new subprogram requires the creation of an additional local data area and a parameter data area.

When you have completed the exercises below, your sample application will consist of the following modules:



Modifying the Local Data Area

You will now add more fields to the local data area that you have previously created. These fields will be used by the subprogram that you will create later.

To add more fields to the local data area

1 Return to your local data area.

2 Select the row containing #NAME-END.

3 From the context menu, choose **Insert > Data Field**.

Or:

Choose the following toolbar button:

The **Data Field Definition** dialog box appears.

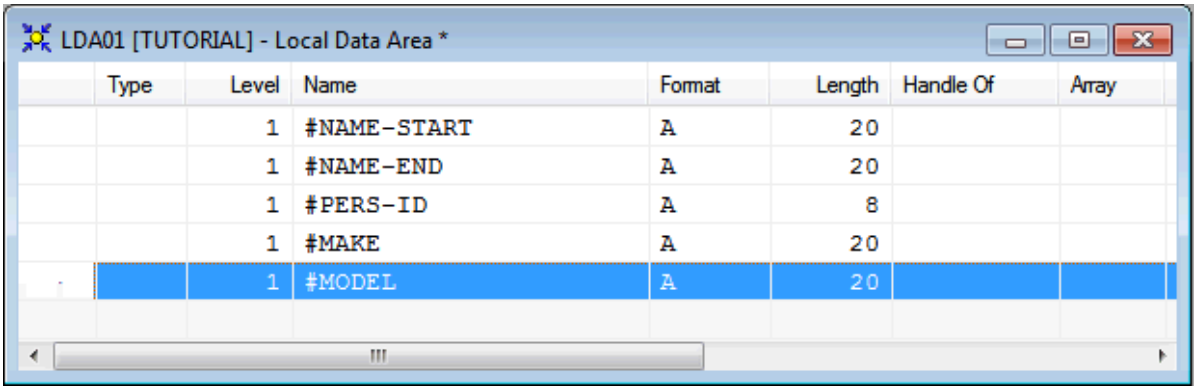
4 Define the following fields:

Level	Name	Format	Length
1	#PERS-ID	A	8
1	#MAKE	A	20
1	#MODEL	A	20

Choose the **Add** button after you have defined a field.

5 When all fields have been defined, choose the **Quit** button.

The local data area should now look as follows:



6 Stow the local data area.

Creating a Parameter Data Area from an Existing Local Data Area

A parameter data area (PDA) is used to specify the data parameters to be passed between your Natural program and the subprogram that you will create later. The parameter data area will be referenced in the subprogram.

With minor modifications, your local data area can be used to create the parameter data area: you will delete two of the data fields in in the local data area and then save the revised data area as a parameter data area. The original local data area remains intact.

Note: It is also possible to create a parameter data area by choosing **New > Parameter Data Area** from the **Object** menu.

To create the parameter data area

1 In the local data area, delete the fields #NAME-START and #NAME-END.

2 From the **Object** menu, choose **Save As**.

The **Save As** dialog box appears. Specify "PDA01" as the name for the parameter data area.

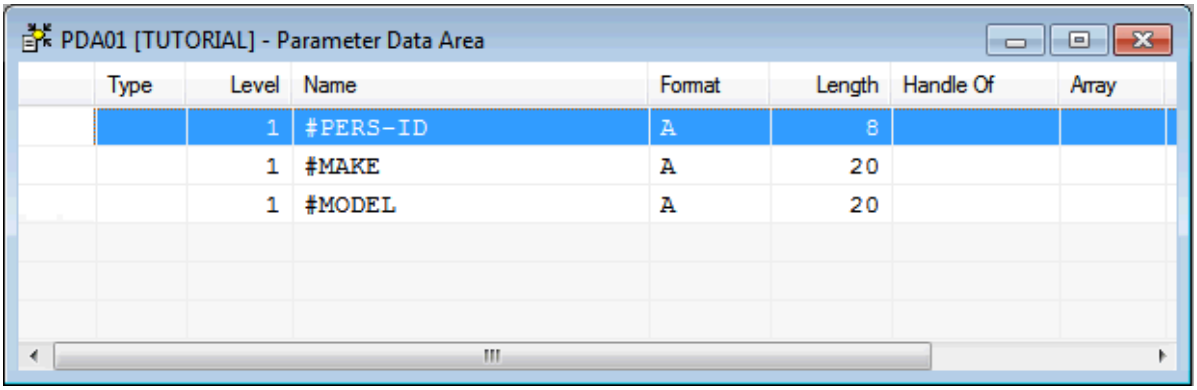
3 Make sure that the library is selected which also contains your program (that is: the **TUTORIAL** node).

4 From the **Type** drop-down list box, choose **Parameter**.

5 Choose the **OK** button.

The new name and type are now shown in the title bar of the editor window. In the library workspace, the new parameter data area is shown in the **Parameter Data Areas** node.

The parameter data area should now look as follows:



6 Stow the parameter data area.

7 Close the editor window for the parameter data area.

Creating Another Local Data Area Containing a Different View

You will now create a second local data area and import fields from the DDM for the `VEHICLES` database file. This DDM is also provided in the system library `SYSEXDDM`.

This local data area will be referenced in the subprogram.

To create the local data area

1 In the library workspace, select the library which also contains your program (that is: select the **TUTORIAL** node).

2 From the context menu, choose **New Source > Local Data Area**.

Or:

Choose the following toolbar button:

An editor window appears.

3 From the **Insert** menu, choose **Import**.

The **Import Data Field** dialog box appears.

4 From the **Library** drop-down list box, select **SYSEXDDM**.

Select the **DDM** option button.

In the **Object** list box, select the sample DDM with the name **VEHICLES**.

5 Press CTRL and select the following fields in the **Importable Data** list box:

```
PERSONNEL-ID
CAR-DETAILS
MAKE
MODEL
```

6 Choose the **Import** button.

The **View Definition** dialog box appears.

7 Specify the name "VEHICLES-VIEW" as the name for the view.

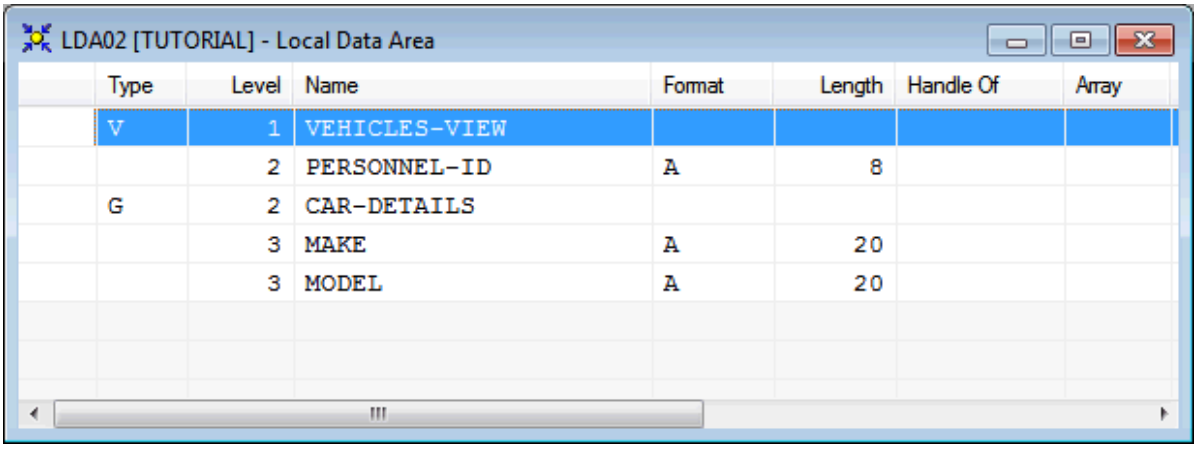
8 Choose the **OK** button.

9 Choose the **Quit** button to close the **Import Data Field** dialog box.

10 Stow the new local data area.

11 When you are asked to specify a name for the local data area, enter "LDA02" and choose the **OK** button.

The local data area should now look as follows:



12 Close the editor window for the local data area.

Creating a Subprogram

You will now create a subprogram that uses a parameter data area and a local data area to retrieve information from the `VEHICLES` file. The subprogram receives the personnel ID passed by the program `PGM01` and uses this ID as the basis for a search of the `VEHICLES` file.

To create the subprogram

1 In the library workspace, select the library which also contains your program (that is: select the **TUTORIAL** node).

2 From the context menu, choose **New Source > Subprogram**.

An editor window appears.

3 Enter the following:

```
DEFINE DATA
  PARAMETER USING PDA01
  LOCAL USING LDA02
END-DEFINE
*
FD1. FIND (1) VEHICLES-VIEW
  WITH PERSONNEL-ID = #PERS-ID
  MOVE MAKE (FD1.) TO #MAKE
  MOVE MODEL (FD1.) TO #MODEL
  ESCAPE BOTTOM
END-FIND
*
END
```

This subprogram returns to a given personnel ID the make and model of the employee's company car.

The `FIND` statement selects a set of records (here: one record) from the database based on the search criterion `#PERS-ID`.

In the field `#PERS-ID`, the subprogram receives the value of `PERSONNEL-ID` that has been passed by the program `PGM01`. The subprogram uses this value as the basis for a search of the `VEHICLES` file.

4 Stow the subprogram.

5 When you are asked to specify a name for the subprogram, enter "SPGM01" and choose the **OK** button.

6 Close the editor window for the subprogram.

Referencing the Subprogram from Your Program

A subprogram is invoked from the main program using a `CALLNAT` statement. A subprogram can only be invoked via a `CALLNAT` statement; it cannot be executed by itself. A subprogram has no access to the global data area used by the invoking object.

Data is passed from the main program to the specified subprogram through a set of parameters that are referenced in the `DEFINE DATA PARAMETER` statement of the subprogram.

The variables defined in the parameter data area of the subprogram do not have to have the same names as the variables in the `CALLNAT` statement. Since the parameters are passed by address, it is only necessary that they match in sequence, format, and length.

You will now modify your main program so that it can use the subprogram you have just defined.

To use the subprogram in your main program

1 Return to the program editor.

2 Insert the following directly above the DISPLAY statement:

```
RESET #MAKE #MODEL
CALLNAT 'SPGM01' PERSONNEL-ID #MAKE #MODEL
```

The RESET statement sets the values of #MAKE and #MODEL to null values.

3 Delete the line containing the DISPLAY statement and replace it with the following:

```
WRITE TITLE
      / '*** PERSONS WITH 20 OR MORE DAYS LEAVE DUE ***'
      / '***     ARE MARKED WITH AN ASTERISK        ***'//
*
DISPLAY   1X '//N A M E' NAME
          1X '//DEPT'    DEPT
          1X '/LV/DUE'   LEAVE-DUE
             ' '         #MARK
          1X '//MAKE'    #MAKE
          1X '//MODEL'   #MODEL
```

The text defined with the WRITE TITLE statement will appear at the top of each page in the output. The WRITE TITLE statement overrides the default page title: the information which was previously displayed at the top of each page (page number, date and time) is no longer shown. Each slash (/) causes the subsequent information to be shown in a new line.

Since the subprogram is now returning additional vehicles information, the columns in the output need to be resized. They receive shorter headers. The column in which the asterisk is to be shown (#MARK), does not receive a header at all. One space will be inserted between the columns (1X). Each slash in the header causes the subsequent information to be shown in a new line of the same column.

Your program should now look as follows:

```
DEFINE DATA
  GLOBAL USING GDA01
  LOCAL USING LDA01
END-DEFINE
*
RP1. REPEAT
*
  INPUT USING MAP 'MAP01'
*
  IF #NAME-START = '.' THEN
    ESCAPE BOTTOM (RP1.)
  END-IF
*
```

```
  IF #NAME-END = ' ' THEN
    MOVE #NAME-START TO #NAME-END
  END-IF
*
  RD1. READ EMPLOYEES-VIEW BY NAME
    STARTING FROM #NAME-START
    ENDING AT #NAME-END
*
    IF LEAVE-DUE >= 20 THEN
      PERFORM MARK-SPECIAL-EMPLOYEES
    ELSE
      RESET #MARK
    END-IF
*
    RESET #MAKE #MODEL
    CALLNAT 'SPGM01' PERSONNEL-ID #MAKE #MODEL
*
    WRITE TITLE
      / '*** PERSONS WITH 20 OR MORE DAYS LEAVE DUE ***'
      / '***     ARE MARKED WITH AN ASTERISK        ***'//
*
    DISPLAY   1X '//N A M E' NAME
              1X '//DEPT'    DEPT
              1X '/LV/DUE'   LEAVE-DUE
                 ' '         #MARK
              1X '//MAKE'    #MAKE
              1X '//MODEL'   #MODEL
*
  END-READ
*
  IF *COUNTER (RD1.) = 0 THEN
    REINPUT 'No employees meet your criteria.'
  END-IF
*
END-REPEAT
*
END
```

4 Run the program.

5 Enter "JONES" as the starting name and press ENTER.

The resulting list should look similar to the following:

```
              *** PERSONS WITH 20 OR MORE DAYS LEAVE DUE ***
              ***     ARE MARKED WITH AN ASTERISK        ***



                           LV
      N A M E         DEPT  DUE           MAKE                MODEL
-------------------- ------ --- - -------------------- --------------------

JONES                SALE30  25 * CHRYSLER             IMPERIAL
JONES                MGMT10  34 * CHRYSLER             PLYMOUTH
JONES                TECH10  11   GENERAL MOTORS       CHEVROLET
JONES                MGMT10  18   FORD                 ESCORT
JONES                TECH10  21 * GENERAL MOTORS       BUICK
JONES                SALE00  30 * GENERAL MOTORS       PONTIAC
JONES                SALE20  14   GENERAL MOTORS       OLDSMOBILE
JONES                COMP12  26 * DATSUN               SUNNY
JONES                TECH02  25 * FORD                 ESCORT 1.3
```

6 Press ESC to close the output window.

7 Stow the program.

You have successfully completed this tutorial.