

BOOK 3

COMM1E - Database Systems

ORACLE

An Introduction

To

Forms

Part II

© Sue Stirk

School of Computing, Engineering & Technology

University of Sunderland, 2002

Version 1

Contents

1.	Introduction to Triggers.....	4
1.1	Introduction.....	4
1.2	Objectives.....	4
1.3	Basic Terms and Concepts	4
2.	Using a Simple Trigger to write a Menu.....	6
2.1	Introduction.....	6
2.2	Objectives.....	6
2.3	Building a menu.....	6
3.	Writing Simple Triggers	10
3.1	Introduction.....	10
3.2	Objectives.....	10
3.3	Creating a Trigger to Select Data from a Table	10
3.4	Exception Handling	12
3.4.1	What happens when an exception occurs?	12
3.4.2	Exception Handling for Select Statements	12
3.4.3	Trigger code to handle exceptions	12
3.4.4	User Defined Exceptions.....	13
3.4.5	Why use them?	14
3.4.6	Defining your own exceptions	14
3.4.7	Key Triggers and GUI Applications	15
4.	Alerts	16
4.1	Introduction.....	16
4.2	Objectives.....	16
4.3	Creating an Alert	16
4.4	Using one alert to display different messages.	19
5.	Using triggers to calculate the value of a field	21
5.1	Introduction.....	21
5.2	Objectives.....	21
5.3	Write a trigger to calculate annual charge	21
5.4	Field Referencing in PL/SQL	23
5.5	The Assignment Operator	23
6.	More data manipulation commands within triggers.....	24
6.1	Introduction.....	24
6.2	Objectives.....	24
6.3	Preparation.....	24
6.4	Using the UPDATE command	26
6.5	Using the DELETE command.....	28
6.5.1	What The Trigger Will Do	28
6.6	Using the INSERT command	31
6.6.1	What The Trigger Will Do	32
7.	Variables	33
7.1	Introduction.....	33
7.2	Objectives.....	33
7.3	Screen Items (Fields)	33
7.4	Local (PL/SQL) Variables	33
7.5	Size of Variables	35

7.6	Global Variables	35
7.6.1	<u>Automatic Creation of Global Variables</u>	36
7.6.2	<u>Initialising Globals With Default Values</u>	37
7.6.3	<u>How To Use</u>	37
7.6.4	<u>Alert Handling Using Global Variables</u>	37
7.7	System Variables	39
CASE STUDY		42

1. Introduction to Triggers

1.1 Introduction

As you have seen, Forms already has a lot of default functionality built in, such as the ability to query tables. However, it is often the case that this default processing is not sufficient to give exactly the functionality that your user might require. Most of the real power of ORACLE Forms comes through its ability to further customise forms by linking small programs, called triggers, to events which will occur during the processing of a form.

There are a large number of pre-defined events, such as clicking a button, to which triggers can be attached and this unit will gradually introduce some of these.

1.2 Objectives

When you have completed this section you will be able to:

- Describe the basic concepts of triggers

1.3 Basic Terms and Concepts

Trigger

Triggers are a group of programming instructions which are executed (triggered) when a particular event occurs within the processing of a form. In trigger programming, the user is in control and triggers may be fired by many different actions by the user. For example, a key may be pressed, or a mouse button clicked, or a record may be inserted. This type of programming is called **event driven programming**. Triggers are written using a programming language called PL/SQL.

The name of a trigger defines what it does. For example, the KEY-NEXT-ITEM trigger is executed when the cursor moves to the next item on the form (this happens when the TAB or ENTER key is pressed). Usually when you press either of these keys the cursor will move to the next item on the form. However, if you use this trigger, the programming statements that you associate with it will be executed instead.

PL/SQL

PL/SQL can be thought of as an extension to SQL. It combines standard SQL statements such as select, insert, and delete, with procedural language capabilities. PL/SQL provides programming facilities which SQL alone does not. Triggers can be used to validate and to enhance what users see when they use a form.

PL/SQL Block

A PL/SQL block is simply a block of code. It consists of three main sections followed by an **end;** The three main sections are

1. A declarative section, in which variables are declared. **This section is optional.**
2. A main body section, which contains executable statements
3. An exception section, which contains statements to handle non-routine events. **This section is optional.** However, if you use it, it must contain some exception code.

A block therefore will usually contain the following sections:

```
declare  
begin  
exception  
end;
```

Each statement within a block should end with a semi-colon.

EXERCISE

1. What is PL/SQL?.
2. What are the main elements of a PL/SQL block?

2. Using a Simple Trigger to write a Menu

2.1 Introduction

Most applications contain several different transactions and so require a method to enable users to select the one they require. This is usually achieved through a menu of options. Triggers, which you build into a form, can be used to provide such a menu system. The method of doing this is very similar to building any other form, but with the addition of a few simple programming statements.

2.2 Objectives

When you have completed this section you will be able to:

- Write a simple trigger to allow a user to select an option from a menu
- Compile and test a trigger that you have written.
- List one common error when writing a trigger

2.3 Building a menu

We will use the property database system to demonstrate how this can be done. Before you start re-run the property.sql file.

First of all, you will need to build a new form, and place some text on the form to represent the options that are available. The Menu that we will build will look like the one below:

The screenshot shows a window titled "WINDOW1" with a menu bar containing "Action", "Edit", "Query", "Block", "Record", "Field", and "Help". Below the menu bar is a toolbar with various icons. The main area of the window displays the text "Property Rental Services" and "MAIN MENU". Below this text are four buttons stacked vertically: "Branch Details", "Owner Details", "Inspection Details", and "Exit". At the bottom of the window, there is a status bar showing "Record: 1/1" and an "Insert" button.

Before you start to create a menu you need to have some forms to call from the menu.

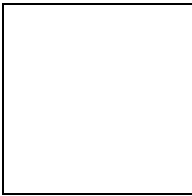
Create three separate default forms:

- a) branch - based on the branch table using all columns
 - b) owner - based on the owner table using all columns
 - c) inspection - based on the inspection table using all columns.
- (You can look back at Unit 2 (Section 2) if you have forgotten how to do this)

You must ensure that you compile and save your three forms so that there is an executable version (i.e. the **.fmx** must exist as this is the version that will be called by the menu)

Now let us follow the steps below to create the Menu:

- 1 First we will create a new form called MENU1. Note however, that when you create the New Block you are not using any columns from any database table. Therefore we need to create this form without using the Wizard. To do this, first give your form its name (MENU1), then highlight *data blocks* in the Object navigator and then click on the create icon (the one with the green plus sign). This will give you a dialogue box that gives you the option to use the wizard or to create the block manually.



'Use the Data Block Wizard' is automatically selected as the default. You need to select 'Build a new data block manually'. A default block name will appear in the Object navigator. Change the name of the block to b1. Go to the properties of the block (by double clicking on the icon at the side of the block name) and ensure that the property called Database Data Block is set to NO. This tells Oracle that you are not using a database table to base your block on. Close the properties window.

N.B. From this point on it will be assumed that you close windows and forms when you have finished with them.

2. Go to the Layout Editor. At this stage the form will be blank. Use the Layout editor to create the title text on the form and to create 4 buttons on the form - one for each Option plus an Exit button. Give meaningful names to the buttons and to the labels on the buttons. To do this double click on the button and you will be taken to the property sheet where you can change the names.

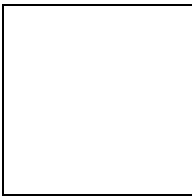
N.B. If you have forgotten how to use the layout editor or how to create buttons you can go back to Unit 2, section 5 to remind yourself.

3. Save and run your form to check out what it looks like. Save your form as Menu1.fmb.

Of course, when you run your form and click on a button, nothing will happen because you have not created any code yet.

Now that you have created your form, and it looks OK you need to write a trigger for each button so that it does something.

4. In the Object Navigator, click on the small plus sign at the left side of the new BRANCH button (you will find this under Items) to expand the node. You should see a triggers node. Double click on this (or highlight it and then select the create icon - which looks like a green plus sign) and you will see a list of pre-defined triggers. The trigger we want to use is the WHEN-BUTTON-PRESSED trigger so scroll down the list until you find it. Then select it by double clicking. You should see the PL/SQL editor.



5. Type in the PL/SQL code as shown below.

```
begin
    call ('branch');
end;
```

This is the code to call the first form - **branch**

N.B. If you have placed your forms in a sub-directory then you will need to type the full path name into your code.

Then compile the trigger. If you have done this correctly you will see a message at the bottom right of the screen telling you that your code has been successfully compiled.

You will need to repeat this for each button on the form, amending your code to call the appropriate form in each case.

The exit button, of course, will have slightly different code. The code to use is:

```
begin
    exit_form;
end;
```

Now save and run your form. Test that all of the options work.

N.B. A common error is to call a form that does not exist. If you do this, you will not have any compilation errors, but when you run your form you will see an error message at the bottom of the screen which tells you that Oracle cannot read the form. If this happens to you it is usually because you have typed the name of your form incorrectly, or because you do not have the executable version of the form.

3. Writing Simple Triggers

3.1 Introduction

There is a vast array of triggers that can be used within Forms. Triggers can be classified in various ways. For example there are key triggers, navigational triggers, transaction triggers etc. Key triggers are executed when the user presses a key. Navigational triggers are executed as the user moves the cursor around the form e.g. from one block to another. Transaction triggers are executed when data is changed, for example when a record is inserted. Validation triggers can be used to check that the data entered by users is correct. This section will introduce you to some of the triggers that can be used.

3.2 Objectives

When you have completed this section you will be able to:

- Be more familiar with the steps involved in trigger definition
- Write a simple trigger to select data automatically from a table
- Compile and test a trigger that you have written.
- Recognise what an exception is
- Understand how processing is affected by an exception
- Be able to write a user defined exception

3.3 Creating a Trigger to Select Data from a Table

When a form is used, you will often want to display information from the database. There can be many reasons for this, for example, to ensure that the correct record is being dealt with, or perhaps to ease the burden of typing. This section will show you how to write a trigger to automatically display information selected from the database.

The information we will display will be the name details of a tenant when the tenant number on a form is entered.

EXERCISE

We will use the default **lease.fmb** form that you created in Unit 2 for this exercise. Before you start rename your form **lease1.fmb**.

Add two new non-database table text items to your form - one for the forename (call it t_forenames) and one for the surname (call it t_surname) of the tenant.

If you can't remember how to do this go back to Section 5.4.7 of the Forms Guide (Part I)

Remember to set the 'Database Item' property to NO in the property palette of each of these two items.

Now BEFORE writing the trigger, save and run your form. Execute a query. What is displayed in the tenant name fields? Write down your answer here.

Why do you think this is? Write down your answer here.

Now we will write the trigger. The trigger we will use for this is a KEY-NEXT-ITEM trigger. This trigger needs to be placed on the *tenant_no* item.

1. Go to the Object Navigator and find the *tenant_no* item. You will see that there is already one trigger on this item. This has automatically been placed there by Forms to maintain integrity constraints. We will add another trigger to the *tenant_no* item.
2. Highlight the *tenant_no* and then click your right mouse button. You will see a menu. Select Smart Triggers from the menu. Smart triggers is a facility which cuts down the possible list of triggers to the most likely ones that you might want to use in any particular place. You should see that here there are several triggers that could reasonably be used. However, this does not give the full list, so select 'Other' from the Smart Triggers menu and you will see a list of all triggers.
3. Find and select KEY_NEXT-ITEM. Click on this trigger and you will be taken to the PL/SQL editor. Type in the code as follows:

```
begin
    select t_forenames, t_surname
    into :lease.t_forenames, :lease.t_surname
    from tenant
    where tenant.tenant_no = :lease.tenant_no;
end;
```

4. This statement is quite straightforward. It is just like the SQL you have already used but with an additional line *into* to display the items in the fields on the form. Compile the code by pressing the compile button in the PL/SQL editor, and if it compiles successfully close the editor. If you receive an error when you compile the trigger, check your typing!
5. Now save and run your form so that you can check that the trigger works. Type in any lease number, property_no of 1001 and a tenant no of 3003. Move to the next field by

pressing the ENTER or TAB key. You should see that now the name of the tenant has been retrieved and is displayed in your new fields.

3.4 Exception Handling

Whenever an SQL statement in a trigger is executed, there is a possibility that something will go wrong. An exception in PL/SQL is quite simply an exceptional circumstance occurring. This exceptional occurrence could be an error, or it could be an event that the programmer defines as 'exceptional'. For this reason, triggers contain a special section which is specifically used for handling this type of event, or as they are called in PL/SQL, **exceptions**. There are two kinds of exception in PL/SQL. These are **pre-defined** exceptions and **user-defined** exceptions.

3.4.1 What happens when an exception occurs?

When an error is found, or in PL/SQL terminology, when an exception is raised, then processing of the block stops and control passes to the special section which is written to handle, or trap, such exceptions. When this happens, the program does not go back to the original piece of code that produced the error but stays in the special section unless there are instructions to do something else. Any further processing must therefore be catered for in the exception section. This means that additional code must be written.

3.4.2 Exception Handling for Select Statements

There are two particular types of exception that can occur when you write a select statement in a trigger.

The most frequently found exceptions are NO_DATA_FOUND and TOO_MANY_ROWS. These are both pre-defined exceptions and are reasonably self evident. The first means that a select statement has been unable to retrieve any rows. The second occurs if the select statement retrieves more than one row when only a single one was expected. If this occurs, the select statement would fail and you would see an unhelpful error message on your screen. However, we can add some PL/SQL to our trigger to handle the exception and to do something sensible with it, such as displaying a message to the user, so that they know what is happening.

3.4.3 Trigger code to handle exceptions

Let us take the select statement that you have written in the previous exercise and add some code to give the user a message if the tenant number entered does not exist.

EXERCISE

1. Go back to your KEY-NEXT-ITEM trigger in the **Lease1.fmb** form and add the NO-DATA-FOUND exception statement.

(N.B. You do not need to use capitals - They have been used in the code for emphasis.)

begin

```
select t_forenames, t_surname
into :lease.t_forenames, :lease.t_surname
from tenant
where tenant.tenant_no = :lease.tenant_no;
```

exception

```
when NO_DATA_FOUND then
message('This tenant number does not exist.
Please try another tenant');
```

end;

2. Save your form as **Lease2.fmb**. Run and test your form by typing in relevant details. Ensure that the tenant number you type doesn't exist. You should see your error message at the bottom of the screen. Close your form.

There are several more pre-defined exceptions that could be raised owing to a SELECT error, which you could potentially come across when writing triggers. These are:-

INVALID_NUMBER :	when converting a character field (CHAR) to a numeric field (NUMBER) and the conversion fails.
DUP_VAL_ON_INDEX :	an INSERT/UPDATE command tried to create two rows in a table with duplicate values in a unique value (indexed) column.
ZERO_DIVIDE :	when you try to divide a number by zero.

In all of these cases you could write a statement to display an error message in the exception section to warn the user of the problem encountered.

3.4.4 User Defined Exceptions

So far, we have looked at pre-defined exceptions, i.e. exceptions which are provided by ORACLE for you to use. However, it is also possible to write your own exceptions. These are called user defined exceptions

3.4.5 Why use them?

You may want to write your own exception in order to:

- Transfer control to the exception handler for exceptions that are considered too 'abnormal' in the context of a trigger.
- Make your code more descriptive.

3.4.6 Defining your own exceptions

User-defined exceptions are defined in the **DECLARE** section of the **PL/SQL** block, using the keyword **EXCEPTION**.

Within the executable section the exception can be **RAISED**, thus transferring control to the exception section.

If an exception handler for the named exception exists then the associated statements will be executed. Control is then passed out of the PL/SQL block.

If a handler for that exception is not found, then an unhandled exception error occurs and processing stops. An unhandled exception causes a trigger to fail and a message is displayed to the user. The following PL/SQL code shows an example of a user defined exception:

```
declare
    no_rentpm exception;
begin
    if :lease.rent_pm is null then
        raise no_rentpm;
    end if;
exception
    when no_rentpm then
        message ('No monthly rent amount entered - please re-
        enter');
end;
```

EXERCISE

1. Think about the previous piece of code and try to work out what it is doing. Write your ideas in the space below.

- 2 Using your **lease2.fmb** create a user defined exception that will only be raised if the **STDATE (start date)** field on your lease form is left blank during data entry.. Call the exception `no_startdate`.
Use a KEY-NEXT-ITEM trigger, and consider carefully which item you need to place the trigger on.
3. Save your form as **lease3.fmb**. Run your form and test your trigger.

3.4.7 Key Triggers and GUI Applications

As you have seen, the key-next-item trigger 'fires' when a key is pressed - the TAB or ENTER key. However, it is important to remember that in GUI applications users may use the mouse instead of pressing keys. In our example above if the user had used the mouse to move to the next item on the form then our trigger would not have fired. This may mean writing another additional trigger to perform the same processing. In order to reduce the amount of coding that you have to do we will assume for the purposes of this Unit that all actions to move from field to field will be performed by the user pressing the TAB or ENTER key. You should be aware though that this may not necessarily be the case in a 'live' application.

4. Alerts

4.1 Introduction

In Oracle Forms, an **Alert** is a window that you can display to give the user a message and to require a choice, such as Yes or No to be made before proceeding. Oracle provides three types of alert:

- Stop
- Caution
- Information

Each type of Alert represents a different level of severity. Alerts are created through the Object navigator.

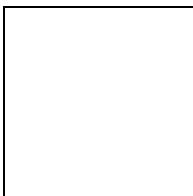
4.2 Objectives

When you have completed this section you will be able to:

- Create an Alert
- Create trigger code to display an alert
- Change a message displayed in an alert through programming

4.3 Creating an Alert

Creating an Alert is quite straightforward. We will create an alert on our **lease3.fmb** form. The Alert will look like the one below



The Alert will ask the user if the details typed in are correct and allow the user to answer Yes or No by clicking on a button.

EXERCISE

To create an Alert we will use the **lease3.fmb** form.

STEP 1

In the Object Navigator double click on the Alert node (towards the top of the Object navigator). This will create an Alert with a default name.

STEP 2

Double click on the Flag icon at the side of the Alert. This will display the property sheet for the Alert. Set the following properties:

Name:	CORRECT_ALERT
Alert Style:	Stop
Button1 Label:	Yes
Button2 Label:	No
Message	Are the Details Correct?

NOTE: If you double click on the message property you will see a message box where you can type in your message.

Close the properties window.

STEP 3

Save your form as **lease4.fmb**, generate and run your form. Type in a new lease record. You should notice that the alert does not appear when you have finished entering the details. This is because it still needs some code to 'trigger' the Alert.

Next we will add a button, and some code, to the form so that when the user clicks on the button the Alert is displayed.

EXERCISE

STEP 1

Add the button. Call the button **Save**. (Remember that you added buttons when you created the menu).

STEP 2

Go to the Object Navigator and create a trigger on the button.

What do you think the trigger should be? (Use smart triggers to help you)

STEP 3

Write the code below to display the Alert.

```
DECLARE
    alert_button number;
BEGIN
    alert_button := Show_Alert('Correct_Alert'); /*See Comment 1*/
    if alert_button = ALERT_BUTTON2 then /*The button marked 'No'*/
        go_item('lease.lease_no'); /* See Comment 2 */
    elsif alert_button = ALERT_BUTTON1 then /*The button marked 'Yes'*/
        commit;
        exit_form;
    end if;
END;
```

Comment 1

The action that is performed when a button is pressed depends on the PL/SQL code that you write. To display an alert, you must use the built in function `SHOW_ALERT`. This function returns a number that indicates to Oracle which button has been pressed. The above trigger code declares a variable called `alert_button` so that different actions can be taken depending on which button is pressed. This variable holds the number returned by the `SHOW_ALERT` function.

Comment 2

Go-item is a built in function that allows you to move directly to the item specified in the brackets. The item must be enclosed in single quotes.

However, many of these built in functions or procedures are restricted. This means that there are some triggers that you cannot use them in. For example you cannot use *go-item* in a `WHEN-VALIDATE-ITEM` trigger.

STEP 4

Save your form as (**lease5.fmb**) and then write down here how you will test it.
Now test your trigger.

4.4 Using one alert to display different messages.

As you can probably imagine it is quite common for applications to have several different alerts to aid users. Rather than having to create numerous alert boxes it is possible to have one alert and to change the message it displays at various points. You can only do this though if the buttons are the same. For example an alert with two buttons 'Yes' and 'No' is quite common. You display different messages by using a built in function of PL/SQL called SET_ALERT_PROPERTY instead of using the property window for the alert

EXERCISE

1. Open the form **lease5.fmb**. Rename this form **lease6.fmb**
2. Delete the alert that you created in this form by highlighting the alert and pressing the delete key. Now create a new alert call ALL_ALERT. This single alert box will be used to display different messages. Create the alert buttons through the property palette of the alert. Button 1 will be 'Yes' and button 2 will be 'No'. Do **NOT** create any message in the property window.
3. Now we will change the code in the when-button-pressed trigger

```
declare
    alert_button number;
    alert_id alert;

begin
    alert_id := find_alert('all_alert');
    set_alert_property (alert_id,
        alert_message_text, 'Are the Details Correct?');
    alert_button := Show_Alert(alert_id);
    if alert_button = alert_button2 then
        go_item('lease.lease_no');
    elsif
        alert_button = alert_button1 then
        set_alert_property (alert_id,
            alert_message_text, 'Do you wish to enter another record?');
        alert_button := Show_Alert(alert_id);
        if alert_button = Alert_button1 then
            commit;
            next_record;
        elsif
            alert_button = alert_button2 then
            commit;
            exit_form;
        end if;
    end if;
end if;

end;
```

Remember to compile your code .

4. Write down here what you think this piece of code will do.
5. How do you think you will test it?
6. Save and run your form and test the trigger. If it works OK then close your form. If not, go back to your PL/SQL and try to work out why not.

5. Using triggers to calculate the value of a field

5.1 Introduction

This section will give you further experience in writing triggers. You will create a trigger which calculates a value using PL/SQL. In order to do this we will again use the property database. We will begin by creating a default form. From there we will customise it to enable us to carry out our task.

5.2 Objectives

When you have completed this section you will be able to:

- Be more familiar with the steps involved in trigger definition
- Write a trigger to calculate the value of a field
- Compile and test a trigger that you have written.

5.3 Write a trigger to calculate annual charge

STEP 1 - CREATE A DEFAULT FORM

Create a new default form based on the **prop_for_rent table**. Call the form prop1.fmb. Include all of the columns from the table.

STEP 2 - CUSTOMISE THE FORM

Customise the form to enhance its appearance.

Add a new item on the form. Call it 'Annual_Charge' and set its properties appropriately.

Because we want the value in this field to be calculated automatically and we don't want the user to be able to type into it, this should be a display item.

Once you have improved its appearance, and added the new item, run your form and perform a query to ensure that your form works OK.

STEP 3 - DETERMINE WHICH TRIGGER SHOULD BE USED TO CALCULATE THE ANNUAL CHARGE

The next step is to write the trigger.

Problem:

First, we need to consider what we want to achieve with our trigger. The first point to note is that this form will be used as an **INPUT** form - it is not just for querying the database. We know that we want to calculate the annual charge.

The annual charge is composed of the monthly rent, multiplied by 12, and the poll tax amount.

Next we need to think about what actions the user will perform when using our form, and at what point we are able to calculate the annual charge

The sequence of actions would be as follows:

- 1) The user will type in details on the form.
- 2) Each time a value is typed into a field, the user will press the TAB or ENTER key to move to the next field.
- 3) However, after typing in a value for the monthly rent and then the poll tax and pressing the TAB key, the cursor will move to the next field, which is annual rent - but that it is all it will do. The user would therefore have to type in a value for the annual charge. However, we want to calculate it automatically as soon as we have the information to do so, i.e. as soon as we know the value for the poll tax field.

Solution:

The trigger we want to use in order to do this is the KEY-NEXT-ITEM trigger.

What we need to do is to replace the current function of this trigger when it reaches the poll tax field (which simply moves the cursor to the next field), with an automatic calculation when it moves to the next field. We need a KEY-NEXT-ITEM trigger placed on the poll tax field.

EXERCISE

1. Write the trigger to calculate the total rent

You have already written several triggers in earlier sections. Follow the same steps again, to write this trigger. Use the code shown below.

Code for KEY-NEXT-ITEM Trigger placed on the Prop_poll_tax field:

```
declare
begin
    :prop_for_rent.annual_charge := (:prop_rent_pm *12) +
    :prop_for_rent.prop_poll_tax;
end;
```

2. Generate, and execute your form. Test it. The annual charge should be automatically calculated when you press return after entering a value in the poll tax field. Does it work?

The trigger you have written is executed (or triggered) whenever the user presses the TAB key, i.e. whenever the user moves to the next field. However, you will notice when you test your form that the cursor stays in the poll tax field and **does not move to the next field**. This is because you have replaced the normal next item function with your trigger. In order to allow the cursor to move to another field you need to add a line of code to your trigger.

Type the change to the trigger as indicated in below.

```
declare
begin
    :prop_for_rent.annual_charge := (:prop_rent_pm *12) +
    :prop_for_rent.prop_poll_tax;
go_item('prop_for_rent.owner_no');
end;
```

Re run your form and test it. The annual charge should be automatically calculated and the cursor should move to the owner_no field.

5.4 Field Referencing in PL/SQL

Field referencing simply refers to how you distinguish between the **values** in items, and the **names** of items. For example, it is the difference between the item name prop_rent_pm, and the value that might be contained in it such as 200.00. When you write triggers it is necessary to distinguish between them - and in fact you have already done this in the trigger you have just written. When you calculate the annual charge, you multiply the **value** in prop_rent_pm by 12, and add the **value** in prop_poll_tax to arrive at a total figure to be displayed on the screen. The way that you do this is by typing :prop_for_rent.prop_rent_pm*12 etc So, to refer to the **value** in a field you always use this format (**:blockname.itemname**).

In your code you have used **go_item('prop_for_rent.owner_no');** This simply moves the cursor to the owner_no field in the block prop_for_rent. This is how you reference the **name** of an item.

5.5 The Assignment Operator

The assignment operator is simply the symbol := and means becomes equal to. Your statement in the above trigger therefore means 'the value in the annual charge field in block prop_for_rent becomes equal to the rent per month * 12 added to the poll tax.'

6. More data manipulation commands within triggers

6.1 Introduction

As well as being able to use the SELECT statement in a trigger other Data Manipulation Language commands that you are familiar with from your use of SQL can also be used. This section will look at the use of the INSERT, UPDATE and DELETE statements within the context of a trigger.

It is important to note that use of INSERT, UPDATE and DELETE commands within a trigger is only relevant if you intend to manipulate data in another table, i.e. a table that the current form is NOT based upon, or a form that does not allow direct data manipulation.

6.2 Objectives

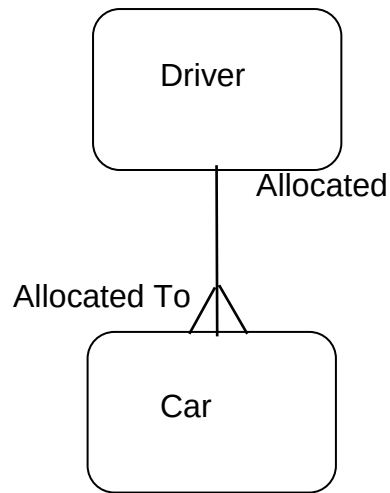
When you have completed this section you will be able to:

- write update statements within a trigger
- write delete statements within a trigger
- write insert statements within a trigger

6.3 Preparation

In order to demonstrate the use of these commands within triggers you are required to set up a new, small database. The logical data model for the Drivacar database can be found below:

DrivaCar Data Model



Using a text editor, create the two tables below. If you are not sure how to do this, go back to Section 9 of your SQL Guide. When you have done this, insert some data of your own choosing into the tables, i.e. invent some. Ensure that the driver_id values are the same in both tables and that the primary key/foreign key relationship is specified when you create the tables.

<u>DRIVER</u>		
DRIVER_ID	VARCHAR2 (6)	NOT NULL,
NAME	VARCHAR2(30)	NOT NULL,
ADDRESS1	VARCHAR2(25)	NOT NULL,
ADDRESS2	VARCHAR2(25),	
ADDRESS3	VARCHAR2(25),	
PCODE	VARCHAR2(8)	NOT NULL,
TEL_NO	VARCHAR2(12),	
NO_CARS	NUMBER(2).	

<u>CAR</u>		
CAR_ID	VARCHAR2(6)	NOT NULL,
DRIVER_ID	VARCHAR2(6)	NOT NULL,
CAR_MAKE	VARCHAR2(20)	NOT NULL,
COLOUR	VARCHAR2(10)	NOT NULL,
DATE_PURCHASED	DATE.	

6.4 Using the UPDATE command

We will look at a the use of the update command to solve a simple problem:

The Problem:

We wish to show an incremented **No_Cars** value on the Driver table whenever a new car is inserted for a specified driver. However, we do **NOT** want to manually update the relevant Driver row, i.e. by typing an actual value into the form field. Instead, we want the trigger to make the necessary changes to the data automatically

STEP 1

Create a Master-Detail form based upon the **two tables**. Call this form **drivacar.fmb**.

Hint : The Master Block is based upon **DRIVER** and the Detail block based upon **CAR**. In order to prevent us from being able to manually type a figure for the number of cars the driver block should have no updateable fields so you should ensure that this is the case through changing the 'update allowed' property.

STEP 2

Create a **WHEN-NEW-FORM-INSTANCE** trigger that executes a query and moves the cursor to the **CAR** block when the form is run. The trigger text for this can be seen below.

A When-New-Form-Instance trigger is always the **first** trigger to 'fire' when a form is executed. A common use for When-New-Form-Instance is to force the form to Execute a Query when the form is opened, rather than go into its default Insert mode.

The When-New-Form-Instance trigger must be defined at Form Level, i.e. it does not belong to a specific block or field.

Code for WHEN-NEW-FORM-INSTANCE trigger placed at form level:

```
declare
begin
    execute_query;
    go_block('car');
    execute_query;
end;
```

Comment

When the **When-New-Form-Instance** trigger fires, the first record in the Driver table will be displayed in the DRIVER block. This is done by using the **EXECUTE_QUERY** packaged function within the trigger.

The cursor is then sent to the CAR block, where another query is executed. The query in the CAR block will return associated Car records for the current driver record displayed in the DRIVER block. The user can now insert new CAR records for that driver.

We wish to insert another Car record for the driver, and, when inserting the record into the database (When you Commit it), increment the No_Cars field in the Driver table.

STEP 3

The next step is to code the trigger that will actually update the Driver table.

A **KEY-COMMIT** trigger, with the code below, is needed for this that will only allow the update as part of the commit time process.

Code for KEY-COMMIT trigger placed on the CAR block:

```
declare
begin
    update driver
    set no_cars = no_cars + 1
    where :car.driver_id = driver_id;
    commit;
    go_block('driver');
    execute_query;
exception
    when no_data_found then message ('driver details not found -
        error on update');
    raise form_trigger_failure;
end;
```

This trigger fires every time the **ACCEPT/COMMIT** key/button is invoked. (This is Ctrl and F7 on the Sun's)

The trigger updates the Driver Row based upon the matching key value retrieved in the query, i.e. the DRIVER_ID.

STEP 4

Test the form. When you run your form, a query will be executed and the first record displayed. Add a new car for the driver displayed. Press the appropriate key to accept/commit the new record. The number of cars should have been updated on the screen.

6.5 Using the DELETE command

Having looked at updating rows within a trigger, the next stage is to look at deleting records.

It has to be said at this stage that Updates are by far the most commonly used DML statement within triggers. There is always a danger that using the Delete statement can sometimes cause loss of referential integrity within the database and so should only be used with discretion.

Problem : You need to remove the cars allocated to a particular driver. However, you have a form which only shows rows from the driver table.

STEP 1

1. Create a form called Driver based only upon the Driver table.
2. Create a non database item called deallocate on the form (1 Character in length, with the Upper Case property set)
3. Generate & Execute the form.
4. Execute a query. The first Driver Record will appear. This is fine as far as it goes. However, as yet there is no way by which Forms can perform the delete operation. As stated previously, the only way to do this is via a Trigger.

6.5.1 What The Trigger Will Do

The trigger needs to check what the user types in the **Deallocate** field. If the response is a 'Y', i.e. the associated records in the Car table are to be deleted, the deletion will take place.

As well as the Delete, the **No_Cars** field will be set to zero, the transaction Committed, and the form cleared.

If the deallocate field has any other value input, or if it is blank, the cursor will move to the Driver_id field.

Finally, if the user tries to deallocate car records from a driver that has no current allocations, then an appropriate method of handling the resultant error and informing the user must be put in place.

STEP 2

Read through the the trigger code to delete the car records – but do not create it yet. You will do this in the next exercise.

Code for KEY-NEXT-ITEM trigger placed on the deallocate Item

```
declare
    alert_button number;
    no_alloc exception;                                /*See comment 1 */
begin
    if :driver.deallocate = 'Y' then
        delete from car
        where :driver.driver_id = driver_id;
            if sql%notfound then                        /*See comment 2 */
                raise no_alloc;
            end if;
        :driver.no_cars :=0;
        commit;
        clear_form;
    else
        go_item('driver.driver_id');
    end if;
exception
    when no_alloc then
        alert_button := show_alert('no_alloc_alert');
        if alert_button = alert_button1 then
            go_item ('driver.driver_id');
        elsif
            alert_button = alert_button2 then
                exit_form;
            end if;
end;
```

This trigger is not as complex as it looks. Indeed it uses some of the features that you have already learnt about, such as exceptions and alerts.

The trigger checks what the user has entered into the deallocate field and, if it is a 'Y', does the following :

- (a) Deletes Car Records associated with that driver.
- (b) Resets the No of cars field in the driver block to 0.
- (c) Commits the changes to the database.
- (d) Clears the form.
- (e) Handles exceptions caused by having no car records to deallocate.
If anything else is entered into the deallocate field (or if it is left blank) the cursor will go to the Driver_id field.

- (f) If the user tries to deallocate (remove) cars from a driver that has no current cars then an appropriate error message using an alert box will be displayed.

Comment 1

Here, a user-defined exception has been declared. If you are unsure about what exceptions are and what they do, then go back to the previous section of this guide to reacquaint yourself.

The exception No_Alloc will be used to generate an error message to the user upon trying to deallocate a zero car allocation from a driver.

Comment 2

When using Data Manipulation Language (DML) commands within triggers, it is often necessary to test the outcome of the command. PL/SQL has several inbuilt 'attributes' that can be used to test for the result of a DML command. This trigger is using the Sql%Notfound attribute.

The Sql%Notfound Attribute

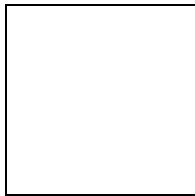
Here, when the DML statement executes, i.e. the 'DELETE from car' statement, it returns a boolean value of TRUE or FALSE, depending on whether any data was found and subsequently deleted. If the DML statement fails (i.e. there are no car rows to deallocate from the specified driver) then the value of sql%notfound will be TRUE. If the deletion is successful, then the value of sql%notfound will be FALSE.

In our trigger, if the value of sql%notfound is TRUE, we want processing to stop and an appropriate error message to be sent to the user via the raising of an exception.

If sql%notfound is FALSE, we want the changes to be committed to the database and processing to continue.

Comment 3

Remember, that processing will only drop into the exception section if sql%notfound is TRUE. Here, the alert is processed, giving the user a pop-up alert box on screen that contains an appropriate error message.



Note the two buttons on the alert box. The result of the user clicking either of these buttons is also handled in the exception section of the key-next-item trigger

EXERCISE

1. Build this trigger into your Driver form. (REMEMBER to create the Alert to display the message in and to call it NO-ALLOC_ALERT)
2. Compile the trigger and then generate and run your form.
Execute a query. The first driver record will appear.
Move to the deallocate field.
Enter 'Y' into the field and hit the TAB or ENTER key.
If the driver has any Car allocation records, they will be deleted. If not, a message will appear instructing the user that there are no Car rows to delete.
3. Test whether the Trigger has succeeded by looking at the Car table rows within SQL*PLUS

You must ensure that at least one Driver record has associated Car records. If this is not the case then Insert some relevant data via SQL*PLUS or a default form.

6.6 Using the INSERT command

The final DML statement that can be hard coded into a trigger is INSERT.
Again, as with Deletes, Inserts should be used with discretion.

Problem :

We need to hold details of Driver ID, Driver Name and Number of Cars in a separate table for administration purposes. This information will be inserted into the new table as a by-product of inserting a new driver record into the Driver table, i.e. it will happen at the same time. No forms are available in the system to put this information directly into the new table, therefore the Insert needs to be handled by a trigger.

STEP 1

Create a new table to hold the three relevant columns that you need, i.e. Driver_id, Name, No_cars. Call this table Driver_admin.
The column definitions are the same as in the driver table.

The form that you created in the previous exercise on Deleting can be used for this exercise, i.e. driver.fmb

One trigger is needed to handle the insert.

6.6.1 What The Trigger Will Do

When a new Driver record is entered into the form and, at the point of insertion of this record into the driver table, a subset of details will be used as the data to create a NEW record in the Driver_Admin table automatically.

STEP 2

Create the trigger code to insert records automatically into the new table.

Code for Key-Commit Trigger to be placed on the driver block

```
declare
begin
    insert into driver_admin
    values (:driver.driver_id, :driver.name, :driver.no_cars);
    Commit;
end;
```

What The Trigger Does

This trigger simply Inserts a new record into the Driver_admin table, using a subset of the data entered into the driver form, i.e. Driver_id, Name, No_cars. Both records, i.e. the new Driver record and the Triggered Insertion into Driver_admin are committed.

STEP 3

Test your trigger by running the form, inserting a new record and then pressing the Accept/commit key.

7. Variables

7.1 Introduction

There are four kinds of variable in PL/SQL. These are

- Screen Items
- Local (PL/SQL) Variables
- Global Variables
- System Variables

7.2 Objectives

When you have completed this section you will be able to:

- Recognise different variable types
- Use global variables to display messages to the user

7.3 Screen Items (Fields)

These are defined as part of the form customisation process. For example, the annual charge field that you set up in your prop1 form is a screen item.

7.4 Local (PL/SQL) Variables

Local variables need to be declared in the declare section of a PL/SQL Block before being referenced, e.g.

```
declare
    annual_charge number;
begin
    Your Trigger Code Goes Here ...
end;
```

You can also assign a value to a variable when it is declared, e.g.

```
declare
annual_charge number := 5000;
begin
    Your Trigger Code Goes Here ...
end;
```

The assignment is achieved through use of the symbol `:=` which simply means 'becomes equal to'. For example

```
new_rent := 10
new_rent := :prop_for_rent.rent_pw*1.05
```

Assignment can also be achieved through a select statement, e.g.

```
select prop_rent_pm * 1.05
into new_rent
from prop_for_rent
where property_no = :prop_for_rent.property_no;
```

EXERCISE

The object of this exercise is to calculate the new rent per month for a given property, taking into account a **5%** rent increase.

1. Create a new default form based on the `prop_for_rent` table . Name your form **prop2.fmb**.
2. The increased rent value will be displayed in a non database table field called **rent_inc**. Create this new item and an appropriate label.
3. Write a trigger to ensure that when a query is executed the increased rent per month is automatically shown in the new **rent_inc** field, as well as the `prop_rent_pm` field.
4. Remember that the annual charge will still need to be calculated.

Write down here which type of trigger would be best suited to achieving the above functionality before writing it. Remember that this will be **AFTER** a query is executed.

N.B. You might find it helpful if you check some values of `prop_rent_pm` through an SQL query first.

Code for POST-QUERY trigger placed at Block level

```
declare
    new_rent number;
begin
    select prop_rent_pm * 1.05
    into new_rent
    from prop_for_rent
    where :prop_for_rent.property_no = property_no;
    :prop_for_rent.rent_inc := new_rent;
    :prop_for_rent.prop_rent_pm =new_rent;
    :prop_for_rent.annual_charge := (new_rent *12)+
        :prop_for_rent.prop_poll_tax;
end;
```

This trigger fires once for each row returned by the query.

7.5 Size of Variables

When declaring a variable of type VARCHAR or NUMBER, you may specify the size, e.g.

```
declare
    name VARCHAR2(25);
    rent_pw NUMBER (7,2);
    startdate DATE;
begin
    Your Trigger Code Goes Here ...
end;
```

You may set the data type and size of a variable to be the same as that of a column in a table. Therefore, a change in the size or data type of the referenced column will result in a corresponding change of the variable. For example, the variable in the code below is assigned the data type of the surname data item in the tenant table.

```
declare
    ten_sname tenant.surname%TYPE;
begin
    Your Trigger Code Goes Here ...
end;
```

7.6 Global Variables

A Global variable is a variable that is active within **any** Trigger within a form, and is active throughout an SQL*Forms session. This variable type can store a character string of up to 255 characters in length.

Global variables are used to store data values that you do not want to store inside a block or that you want to share between several forms in a run form session.

Before a global variable can become 'active', it must first be initialised via a trigger statement. The variable is initialised the first time you assign a value to it.

An advantage of global variables is that once they are initialised they can be used by any form within a run form session (thus the term GLOBAL). This 'sharing' can speed up processing enormously if a number of forms can access the same variable.

Global variables are assigned within the main section of a trigger (usually after the BEGIN statement), and are always preceded by **:global**. Note that global variables are not declared in the 'declare' section.

```
e.g.
begin
    :global.tmess1 := 'Do you want to input another record?';

    Your Trigger Code Goes Here...
end;
```

This sets up a global variable called **tmess1** that has the value of the string: **'Do you want to input another record?'** assigned to it. The value of this global can be utilised at various times during a forms session, whether simply as informative user messages or within an alert (see later in this section).

Any references to global variables must be in the same format, e.g.

```
if :b1.tmess1 = :global.tmess1
    then create_record;
end if;
```

7.6.1 Automatic Creation of Global Variables

There is no hard and fast rule forcing global variables to be declared in advance of their use. Oracle Forms will always create the global if it comes across a statement that references a previously undefined variable.

```
e.g.
:global.surname := :tenant.t_surname;
```

Here, Forms creates the global **surname** and assigns the value of the block item **t_surname** to it.

7.6.2 Initialising Globals With Default Values

Global variables can be initialised by using the **DEFAULT_VALUE** built-in procedure. This procedure is used to assign a value to a global whose initial value would normally be NULL. If the variable, to which the value is being assigned is an undefined global variable, Oracle Forms will automatically create the variable for you.

e.g.

```
Default_Value('ENGLAND', 'GLOBAL.country');
```

This creates the global variable **country** and initialises it to the value **ENGLAND**.

7.6.3 How To Use

Global Variables can be used anywhere within the begin/end section of PL/SQL code.

They can be used in Select statements, e.g.

```
select t_surname
into :tenant.surname
from tenant
where :global.holdval = tenant.tenant_no;

select property_no
into :global.prop
from prop_for_rent
where prop_for_rent.property_no = property_no;
```

They can be used in IF - END IF statements, in order to control processing within a trigger,

e.g.

```
if :global.ten_sname = 'SMITH' then
    Your Trigger Code Goes Here...
elsif
    :global.ten_sname = 'BROCK' then
    Your Trigger Code Goes Here...
end if;
```

7.6.4 Alert Handling Using Global Variables

You should be familiar with the concepts of Oracle Forms alerts from an earlier section of this guide. If you are unsure of the concepts involved then simply reread the section.

A good use of global variables is in defining alert messages to users. The example given below shows how you can use global variables to achieve this.

Code for KEY-NEXT-ITEM trigger placed on the COMMENTS item
Do NOT type this yet. You will create this trigger in the next exercise

```
declare
    alert_id alert;
    alert_button number;
begin
    alert_id := find_alert ('universal_alert');    /*comment 1*/
    set_alert_property
        (alert_id, alert_message_text, :global.message1); /*comment 2*/
    alert_button := show_alert(alert_id);
    if alert_button = alert_button1 then
        set_alert_property
            (alert_id, alert_message_text, :global.message2); /*comment 3*/
        alert_button := show_alert(alert_id);
        if alert_button = alert_button1 then
            commit;
            create_record;
        elsif
            alert_button = alert_button2 then
                commit;
                exit_form;
            end if;
        elsif alert_button = alert_button2 then
            next_item;
        end if;
    end if;
end;
```

What This Trigger Does

This trigger is used for checking that the information the user has entered into the form (in this case, a new property record) is correct. The user is prompted for a response via a series of alerts. Their responses determine subsequent form processing.

The trigger only uses one alert, however the messages that the alert displays are set within the trigger by assigning the values of several pre-defined global variables. The global variables would normally have been set up in a **WHEN-NEW-FORM-INSTANCE** trigger, and would resemble the following:

```
:global.message1 := 'Are These Details Correct?';
:global.message2 := 'Do You Wish To Enter Another Property?';
```

Comment 1

Here, the object identifier of the alert: **Universal_Alert** is located (using the Find_Alert function) and stored in the local variable: **Alert_Id**.

Comment 2

Here, the **SET_ALERT_PROPERTY** procedure is used to set the message that the alert will display to that of the value held in the global variable **:global.message1**.

This checks the user response to the alert . There are two options: YES and NO.

If the response is YES, the processing moves to the inner next statement.

If the response is NO, the processing moves to the ELSIF construct of the outer IF -- END IF statement.

Comment 3

Processing will reach this point as a result of the user responding to the alert with a YES, i.e. the details entered into the form are correct.

At this point, the message in the alert is changed to the value held in the second global variable **:global.message2**. The alert is then redisplayed to the user.

Control now enters the inner IF -- END IF construct, where the user response to the new alert message will be evaluated.

EXERCISE

Create a new default form based on the prop_for_rent table. Call the form prop3.fmb
You are required to do the following:

1. Create an alert. Call it **Universal_Alert**. The alert will have two buttons, labelled YES and NO.
2. Create a When-New-Form-Instance trigger, and set up your two global messages.
3. Create the Key-Next-Item trigger exactly as it appears above
4. Generate and run the form.
5. Test your form by entering some details to see if the messages are displayed.

7.7 System Variables

There are a number of system variables that are predefined and can be used to monitor the status of current objects. For example there are system variables which can be used to tell you which field and block the cursor is in, or which will tell you the value of the current time

and date. You have already come across these when you placed the current time and date on one of your customised forms in Unit 2 (\$\$DATE\$\$).

A System variable is an SQL*Forms variable that keeps track of some internal SQL*Forms state. System variables can be referenced in order to control the way in which applications behave. (Unlike global variables, the values of system variables are form dependant). There are a number of system variables which are described in the table below.

Although you will not necessarily have to use all of the system variables, some examples are provided below.

These are provided **for reference purposes only** to demonstrate the type of system variables available.

For complete descriptions of the system variables, it is recommended that you read the online forms documentation.

System.Block Status

key-clrblk (*This is the trigger name*)

```
if :system.block_status = 'changed'
    then commit;
end if;
clear_block;
```

System.Cursor Block

key-nxtblk

```
if
    :system.cursor_block = 'orders'
then go_block ('items');
elsif
    :system.cursor_block = 'items'
then go_block ('customers');
elsif
    :system.cursor_block = 'customers'
then go_block ('orders');
end if;
```

System.Cursor Record

key-prvfld

```
if :system.cursor_record = '1'
    then go_item ('orders.total');
```

```
else
    previous_item;
end if;
```

System.Cursor Value

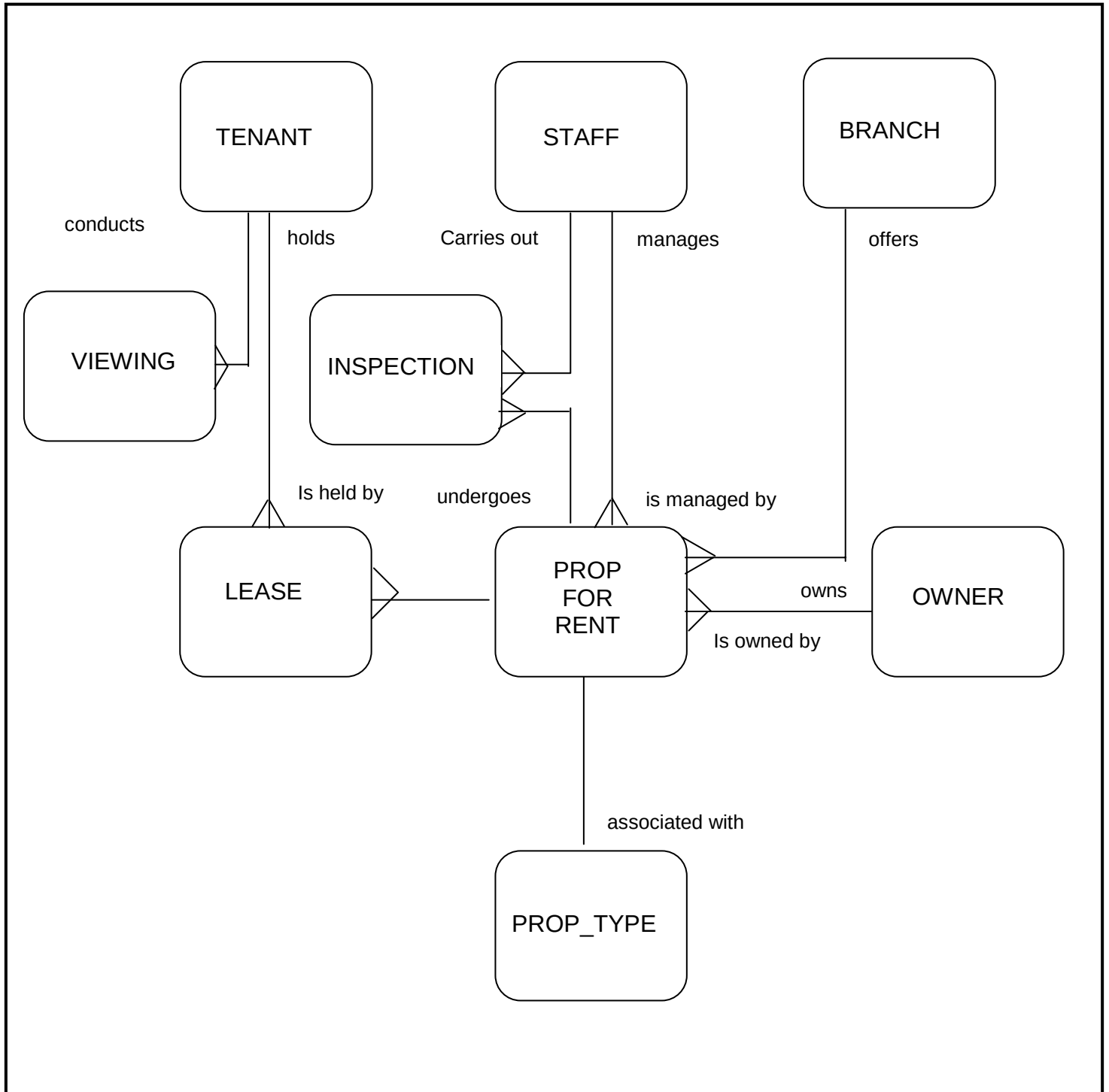
```
key-next-item

begin
    if :system_cursor_value < 100
    then next_item
    else message ('We do not sell any items less than £100');
end;
```

Try to work out what each of these would do.

CASE STUDY

PROPERTY RENTAL SERVICES



LEASE

Name	Null?	Type
-----	-----	-----
LEASE_NO	NOT NULL	NUMBER(5)
*PROPERTY_NO	NOT NULL	NUMBER(5)
*TENANT_NO	NOT NULL	NUMBER(5)
RENT_PM		NUMBER(6,2)
PAYMENT_METHOD	NOT NULL	CHAR(1)
DEPOSIT_AMOUNT		NUMBER(6,2)
DEPOSIT_PAID		CHAR(1)
STDATE	NOT NULL	DATE
ENDDATE		DATE

VIEWING

Name	Null?	Type
-----	-----	-----
PROPERTY_NO	NOT NULL	NUMBER(5)
TENANT_NO	NOT NULL	NUMBER(5)
DATE_VIEW		DATE
COMMENTS		VARCHAR2(50)

TENANT

Name	Null?	Type
-----	-----	-----
TENANT_NO	NOT NULL	NUMBER(5)
T_SURNAME	NOT NULL	VARCHAR2(25)
T_FORENAMES	NOT NULL	VARCHAR2(25)
T_STREET	NOT NULL	VARCHAR2(25)
T_AREA		VARCHAR2(20)
T_TOWN	NOT NULL	VARCHAR2(20)
T_POSTCODE	NOT NULL	VARCHAR2(8)
T_TELNO		VARCHAR2(12)
T_PREF_TYPE		CHAR(1)
T_MAX_RENT		NUMBER(6,2)

INSPECTION

Name	Null?	Type
-----	-----	-----
PROPERTY_NO	NOT NULL	NUMBER(5)
STAFF_NO	NOT NULL	NUMBER(5)
INSPECT_DATE	NOT NULL	DATE
COMMENTS		VARCHAR2(50)

PROP_FOR_RENT

Name	Null?	Type
-----	-----	-----
PROPERTY_NO	NOT NULL	NUMBER(5)
PROP_STREET	NOT NULL	VARCHAR2(25)
PROP_AREA		VARCHAR2(20)
PROP_TOWN	NOT NULL	VARCHAR2(20)
PROP_PCODE	NOT NULL	VARCHAR2(8)
PROP_TYPE	NOT NULL	CHAR(1)
PROP_ROOMS	NOT NULL	NUMBER(2)
PROP_RENT_PM	NOT NULL	NUMBER(7,2)
PROP_POLL_TAX		NUMBER(6,2)
*OWNER_NO	NOT NULL	NUMBER(5)
*STAFF_NO		NUMBER(5)

*BRANCH_NO	NOT NULL	NUMBER(3)
AVAILABLE		CHAR(1)
COMMENTS		VARCHAR2(150)

PROP_TYPE

Name	Null?	Type
-----	-----	-----
PROP_TYPE	NOT NULL	CHAR(1)
TYPE_DESC	NOT NULL	VARCHAR2(15)

OWNER

Name	Null?	Type
-----	-----	-----
OWNER_NO	NOT NULL	NUMBER(5)
OW_SURNAME	NOT NULL	VARCHAR2(25)
OW_FORENAMES	NOT NULL	VARCHAR2(25)
OW_STREET	NOT NULL	VARCHAR2(25)
OW_AREA		VARCHAR2(20)
OW_TOWN	NOT NULL	VARCHAR2(20)
OW_PCODE	NOT NULL	VARCHAR2(8)
OW_FEE		NUMBER(6,2)

STAFF

Name	Null?	Type
-----	-----	-----
STAFF_NO	NOT NULL	NUMBER(5)
*BRANCH_NO		NUMBER(3)
STAFF_SURNAME	NOT NULL	VARCHAR2(25)
STAFF_FORENAMES	NOT NULL	VARCHAR2(25)
STAFF_STREET	NOT NULL	VARCHAR2(25)
STAFF_AREA		VARCHAR2(20)
STAFF_TOWN	NOT NULL	VARCHAR2(20)
STAFF_PCODE	NOT NULL	VARCHAR2(8)
STAFF_TELNO		VARCHAR2(12)
STAFF_GENDER		CHAR(1)
STAFF_SALARY		NUMBER(8,2)

BRANCH

Name	Null?	Type
-----	-----	-----
BRANCH_NO	NOT NULL	NUMBER(3)
BR_STREET	NOT NULL	VARCHAR2(25)
BR_AREA		VARCHAR2(20)
BR_TOWN	NOT NULL	VARCHAR2(20)
BR_PCODE	NOT NULL	VARCHAR2(8)
BR_TELNO		VARCHAR2(12)