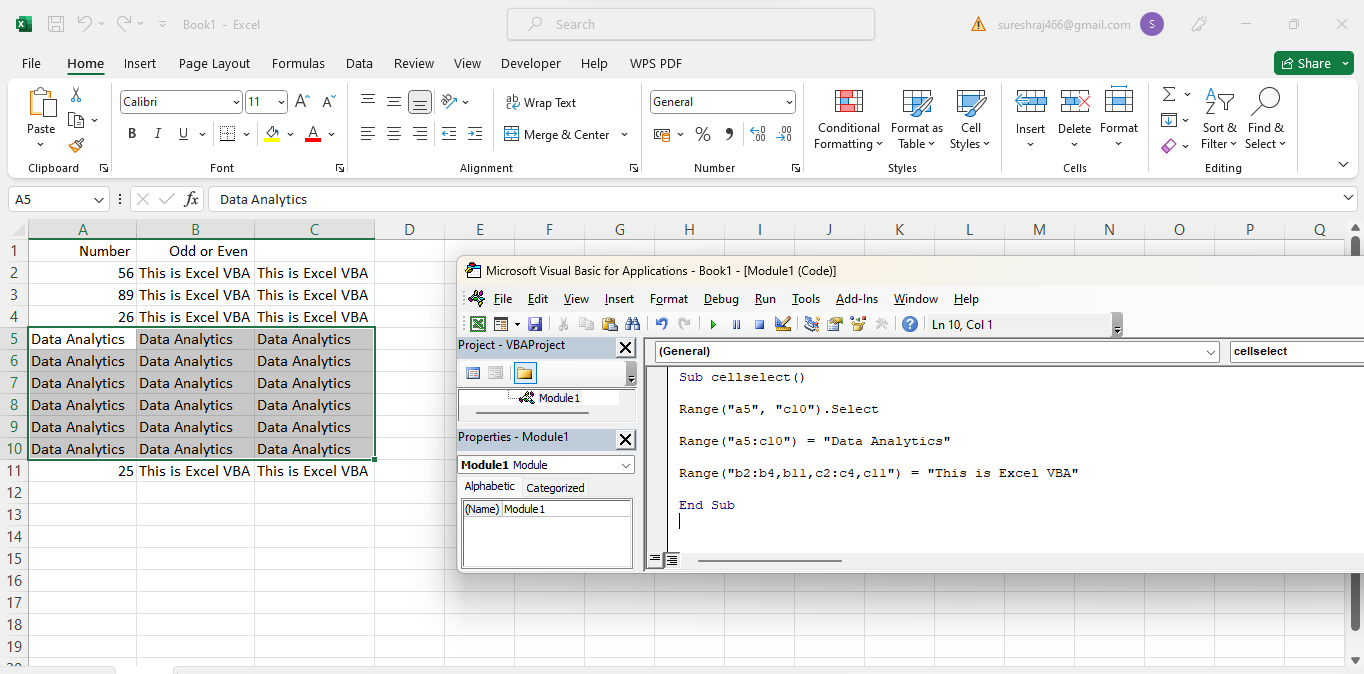


**Excel Assignment - 20**

1. Write a VBA code to select the cells from A5 to C10. Give it a name “Data Analytics” and fill the cells with the following cells “This is Excel VBA”

| Number | Odd or  even |
| --- | --- |
| 56 |  |
| 89 |  |
| 26 |  |
| 36 |  |
| 75 |  |
| 48 |  |
| 92 |  |
| 58 |  |
| 13 |  |
| 25 |  |

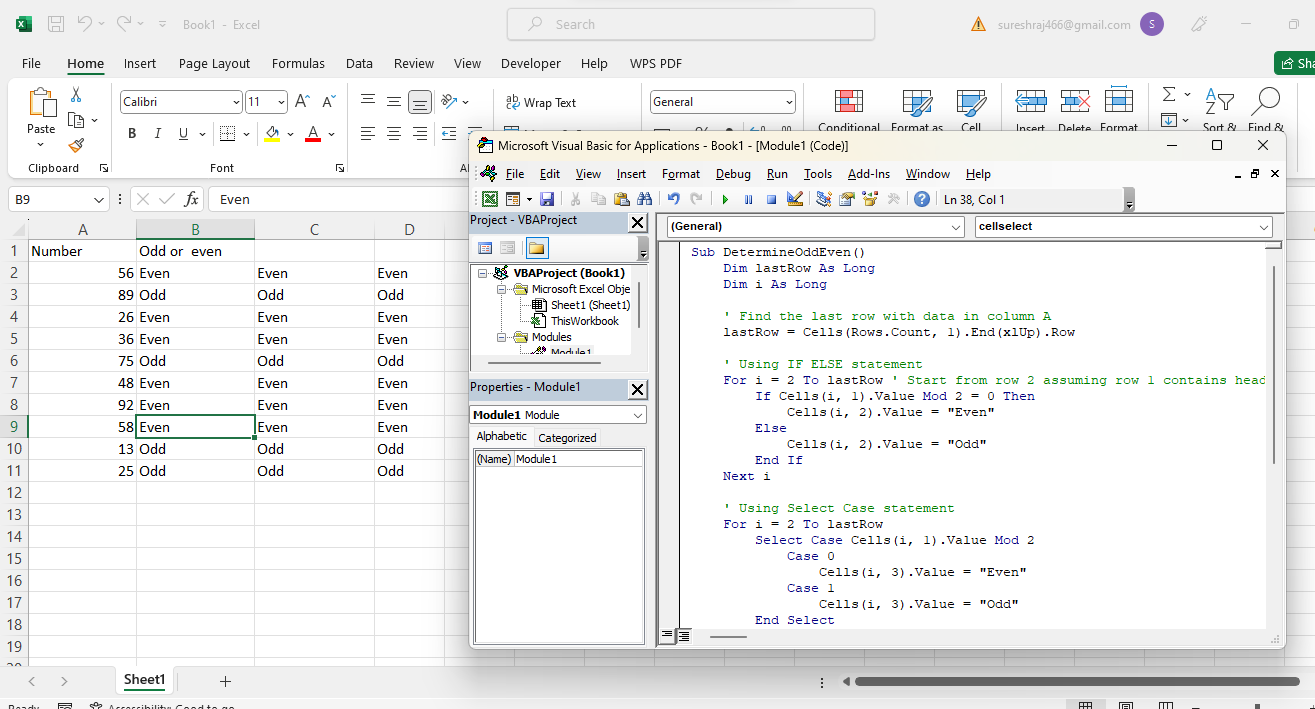
Answer: 

2. Use the above data and write a VBA code using the following statements to display in the next column if the number is odd or even

a. IF ELSE statement

b. Select Case statement

c. For Next Statement

Answer: 

3. What are the types of errors that you usually see in VBA?

Answer:

There are four types of errors in Excel VBA:

1. Syntax errors
2. Compilation errors
3. Runtime errors
4. Logical Errors

### Syntax Error

A syntax error, as the name suggests, occurs when VBA finds something wrong with the syntax in the code.

**Compile errors**

Compile errors occur when something is missing that is needed for the code to run.

### Run Time Errors

Runtime errors are those that occur when the code is running.

Run time errors will occur only when all the syntax and compile errors are being taken care of.

### Logical Errors

Logical errors would not make your code stop but can lead to wrong results. These could also be the most difficult types of errors to troubleshoot.

4. How do you handle Runtime errors in VBA?

Answer:

Handling Runtime Errors

Runtime errors occur as your macro runs, and typically result from specific conditions present at that time. For example, if you prompt the user for a host name, and attempt to connect to that host, but the host is not available, the Connect method fails and Visual Basic generates a runtime error.

You should always include some form of error handling in your macros to deal with runtime errors. Without any error handling, a runtime error causes a macro to stop immediately, and gives the user little information.

To deal with runtime errors, you'll need to trap (catch) the errors, handle them, and then resume execution after the error is handled:

[Trapping Runtime Errors](https://www.microfocus.com/documentation/infoconnect-desktop/17-1/vba-guide/HandlingRuntimeErrors.html#trapping)

[Handling Errors Inline](https://www.microfocus.com/documentation/infoconnect-desktop/17-1/vba-guide/HandlingRuntimeErrors.html#handling)

[Using Error Handling Routines](https://www.microfocus.com/documentation/infoconnect-desktop/17-1/vba-guide/HandlingRuntimeErrors.html#routines)

Notes:

Two other types of errors you may encounter as you program in InfoConnect include:

* Compiler errors which prevent your macros from running, and typically result from errors in syntax.
* Programming logic, which occurs when your macro does not function as expected — the programming syntax is correct, the macro compiles, but an error in logic causes the macro to produce unexpected or incorrect results. Use the debugging tools in the Visual Basic Editor to track down logic errors. (Search for debugging code in the Visual Basic Editor Help index for more information.)

Most of the illustrative examples in this guide do not include error handling because these samples are intended to show how to perform specific tasks or illustrate concepts. If you use these samples as part of your production macro, be sure to add error handling routines.

## Trapping Runtime Errors

The first step in dealing with runtime errors is to set a "trap" to catch the error. You do this by including an On Error statement in your macro. When a runtime error occurs, the On Error statement transfers control to an error-handling routine.

To trap errors, you must set your error trap above the point in the procedure where errors are likely to occur. A good practice is to place the error trap near the top of the procedure. To avoid having the error-handling routine execute even when no error occurs, you should include an Exit Sub or Exit Function statement just before the error-handling routine's label.

| **Trapping errors** | **Copy Code** |
| --- | --- |
| Sub ErrorDemo ()  On Error GoTo MyHandler  'Program code goes here. To avoid invoking the error handling routine after this  'section is executed, include the following line just above the error-handler.  Exit Sub  MyHandler:  'Error is handled and the macro terminates gracefully.  Exit Sub  End Sub | |

#### Resuming a Procedure After an Error

After your macro successfully traps and handles an error, you can choose whether to have the macro exit the procedure (by including an Exit Sub or Exit Function statement in the handler) or resume the procedure (by including a resume statement). When you resume a procedure, you can also have the macro correct the error condition automatically, in which case the end user may never know that an error occurred.

| Use this statement | To do this |
| --- | --- |
| Resume | Exit the error handler and resume the macro at the line that caused the error. Your macro must automatically correct the error condition or prompt the user to correct it before resuming; otherwise, the error will occur again. |
| Resume Next | Resume the macro at the line following the one that caused the error. |
| Resume Label | Exit the error handler and pass control to the statement identified by the label. |

Each of the following examples show the general structure of a procedure that includes an error trap and error-handling routine.

In this example, control resumes at the line following the statement that caused the runtime error.

| **Resuming control on line after error** | **Copy Code** |
| --- | --- |
| Sub SampleResumeNext ()  On Error GoTo MyHandler  'Normal program code goes here. If an error  'occurs, control is transferred to the handler.  'When the handler is done, control resumes  'at the next line here.  Exit Sub  MyHandler:  'Error is handled, and control resumes at the line  'after the statement that caused the error.  Resume Next  End Sub | |

In this example, control returns to the DoPrompt label after the error handler executes.

| **Resuming control at label** | **Copy Code** |
| --- | --- |
| Sub SampleResumeToLabel ()  On Error GoTo MyHandler  DoPrompt:  'Normal program code goes here. If an error occurs,  'control is transferred to the handler.  Exit Sub  MyHandler:  'Error is handled, and then control is transferred  'to the DoPrompt label.  Resume DoPrompt  End Sub | |

## Handling Errors Inline

To cause a procedure to handle relatively simple runtime errors without branching to a separate error-handling routine, you can use a form of the On Error statement that lets you deal with errors inline; that is, directly in the code that caused the error, rather than in a separate routine.

To handle an error inline, use the Resume Next statement with On Error. Any errors that occur during runtime cause InfoConnect to continue executing the macro at the next statement. If an error occurs, it is handled by opening a dialog box, passing control to another procedure or to a routine within the same procedure.

This example checks for errors when saving an IBM 3270 or 5250 session document. If the save operation fails, an error message appears. If the save operation succeeds (Err.Number returns 0), a "Save complete" message appears.

| **Handling an error inline** | **Copy Code** |
| --- | --- |
| Sub SaveSessionDemo()  Dim theError As Integer  On Error Resume Next  ThisIbmTerminal.Save  theError = Err.Number  Select Case theError  Case 0  MsgBox "Save complete."  Case Else  MsgBox Err.Description & "."  End Select  End Sub | |

## Using Error Handling Routines

After you set an error trap, write the error-handling routine that will deal with errors that are likely to occur — in addition to unanticipated errors — when your macro runs. You can identify an error-handling routine by its line label.

Observe the following guidelines:

* Place error-handling routines near the end of the procedure, just before the End Sub or End Function statement.
* Avoid executing error-handling code when no error occurs by placing an Exit Sub or Exit Function statement just before the error handler's label. A Resume statement is typically used to continue execution of the macro after the error handler is done.
* Use the Err object to return specific information about the error. (See the Visual Basic Help for more information about the Err object.)

The following sample shows a simple error handling routine.

This simple error handler displays a custom error message and then terminates the macro:

| **Terminating the macro on error** | **Copy Code** |
| --- | --- |
| Sub ExitOnError ()  On Error GoTo MyHandler  'Main body of procedure.  Exit Sub  MyHandler:  MsgBox "Error occurred. Cannot complete operation."  Exit Sub  End Sub | |

5. Write some good practices to be followed by VBA users for handling errors

Answer:

1. Use ‘On Error Go [Label]’ at the beginning of the code. This will make sure any error that can happen from there is handled.
2. Use ‘On Error Resume Next’ ONLY when you’re sure about the errors that can occur. Use it with expected error only. In case you use it with unexpected errors, it will simply ignore it and move forward. You can use ‘On Error Resume Next’ with ‘Err.Raise’ if you want to ignore a certain type of error and catch the rest.
3. When using error handlers, make sure you’re using Exit Sub before the handlers. This will ensure that the error handler code is executed only when there is an error (else it will always be executed).
4. Use multiple error handlers to trap different kinds of errors. Having multiple error handler ensures that an error is properly addressed. For example, you would want to handle a ‘type mismatch’ error differently than a ‘Division by 0’ run-time error.



6. What is UDF? Why are UDF’s used? Create a UDF to multiply 2 numbers in VBA

Answet:

A user-defined function (UDF) is a function you define so you can call it from SQL. As with built-in functions you can call from SQL, a UDF’s logic typically extends or enhances SQL with functionality that SQL doesn’t have or doesn’t do well. A UDF also gives you a way to encapsulate functionality so that you can call it repeatedly from multiple places in code.

You write a UDF’s logic – its handler – in one of the supported languages. Once you have a handler, you can create a UDF with a CREATE FUNCTION command, then call the UDF with a SELECT statement.

### Scalar and Tabular Functions

You can write a UDF that returns a single value (a scalar UDF) or that returns a tabular value (a user-defined table function, or UDTF).

* A *scalar* function (UDF) returns one output row for each input row. The returned row consists of a single column/value.
* A *tabular* function (UDTF) returns a tabular value for each input row. In the handler for a UDTF, you write methods that conform to an interface required by Snowflake. These methods will:
  + Process each row in a partition (required).
  + Initialize the handler once for each partition (optional).
  + Finalize processing for each partition (optional).

