Exam I Review

Exam Info

- Online Exam on Canvas
  - Timed 85-minute exam once you open it on Canvas. You'll be able to take the exam at any time within 48 hours after it becomes available on Canvas. No exceptions unless you have a medical excuse (**visit notices are not medical excuses**)
  - Exam will open on Thursday 10/19 12PM(noon) ET. Exam will close on Saturday 10/21 12PM(noon) ET.
  - The College of Engineering Honor Code strictly applies and should be followed (please complete the honor pledge when you finish your test). The honor pledge is in the quizzes section as well.
  - **Exam material should not be posted/discussed publicly or shared in any way (including Piazza Board). Do not post screenshots of any sort!**
- Exam recommendations:
  - Prepare a note sheet.
  - Have a notebook/blank paper for annotations and in case you need to submit them for re-grading.
  - Have a simple scientific calculator

Exam Format

- Section I, combination of true/false and multiple choice questions (30 points)
- Section II, 2 open ended questions – conceptual (20 points)
- Section III, 4-5 questions, you'll be asked to write code for queries based on the statements/methods covered in class (and HW) or you may be given code and a worksheet and may be asked to show the outcome (50 points)

Study Guide

- Lecture Slides
- Homework and Lab Assignments
- Practice Exam

Study Guide (Lec 1 – Lec 12)

- Terms and Concepts
- Database Design Process
- SQL
- VBA Logic

Terms and Concepts

- Terms and Concepts
  - ERD, entity, relationship
  - Relational Database, Table, attribute, record
  - Normal Forms
  - Primary key, foreign key

Database Intro

- Database Design Process
  - Identify Entities and Attributes
  - Primary keys
  - Relationships
  - Check dependencies
  - Normalization concepts

Normalization – key concepts

- Every entity (Table) should have a primary key (1NF)
- Every attribute should depend on the primary key (2NF)
- Every attribute should ONLY depend on the primary key (no transitive dependencies) (3NF)
  - All attributes must be attributes of the key only; the table should include no attributes of attributes.
  - Attributes of attributes are technically referred to as “transitive dependencies”.

SQL

- INSERT
- UPDATE
- DELETE
- SELECT
- Aggregate functions, GROUP BY… HAVING
- WHERE clause
- INNER JOIN
- LIKE
- Nested SELECT

INSERT Statement

- INSERT is used to add rows into a table
- Syntax:

  INSERT INTO table (field1, field2, …) **VALUES** (value1, value2, …)
- Example:

  INSERT INTO Students (UMID, SSN, Name, Email) **VALUES** (37339830, 334332190, 'Joseph Woods', 'joew@umich.edu')

UMID*	SSN	Name	Email
37339822	344021945	Edward Jones	edjones@umich.edu
37339823	342122843	Steven Hanks	shanks@umich.edu
37339824	564231347	Edward Jones	edjones2@umich.edu
37339829	473293828	Edward Jones	edwardj@umich.edu
37339830	334332190	Joseph Woods	joew@umich.edu

New Row

UPDATE Example

- Update a record in Student table:

  UPDATE Students (UMID, SSN, Name, Email)
SET
  Name='Joseph Woodson', SSN='333-23-3444'
  WHERE UMID=37339830

- Result:

UMID*	SSN	Name	Email
37339822	344-02-1945	Edward Jones	edjones@umich.edu
37339823	342-12-2843	Steven Hanks	shanks@umich.edu
37339824	564-23-1347	Edward Jones	edjones2@umich.edu
37339829	473-29-3828	Edward Jones	edwardj@umich.edu
37339830	333-23-3444	Joseph Woodson	

- Without a **WHERE** clause, **UPDATE** commands will change ALL records in a table!

  UPDATE Students SET Name='Amy'

  will make EVERY student have the name 'Amy'

- Whereas

  UPDATE Students SET Name='Amy' WHERE UMID=44403200

  Will only change the name of one student (4)

DELETE Example

- Delete a student with UMID 37339824

  DELETE FROM Students **WHERE** UMID=37339824
- This will delete only one record
- But if you use

  DELETE FROM Students **WHERE** Name='Edward Jones'
- This will delete all students named 'Edward Jones'

UMID*	SSN	Name	Email
37339822	344-02-1945	Edward Jones	edjones@umich.edu
37339823	342-12-2843	Steven Hanks	shanks@umich.edu
37339824	564-23-1347	Edward Jones	edjones2@umich.edu
37339829	473-29-3828	Edward Jones	edwardj@umich.edu
37339830	333-23-3444	Joseph Woodson	

Using SQL

- SELECT * FROM Students

Students			
UMID	Name	Email	Gender
10000001	Steve	steve@notexist.com	M
10000002	John	john@notexist.com	M
10000003	Mary	mary@notexist.com	F
10000004	Emily	emily@notexist.com	F
10000005	Mike	mike@notexist.com	M
10000006	James	james@notexist.com	M

How SELECT Statement Works

- SELECT UMID, Name FROM Students

UMID	Name	Email	Gender
10000001	Steve	steve@notexist.com	M
10000002	John	john@notexist.com	M
10000003	Mary	mary@notexist.com	F
10000004	Emily	emily@notexist.com	F
10000005	Mike	mike@notexist.com	M
10000006	James	james@notexist.com	M

UMID	Name
10000001	Steve
10000002	John
10000003	Mary
10000004	Emily
10000005	Mike
10000006	James

How SELECT Statement Works

- SELECT UMID, Name FROM Students WHERE Gender = 'F'

UMID	Name	Email	Gender
10000001	Steve	steve@notexist.com	M
10000002	John	john@notexist.com	M
10000003	Mary	mary@notexist.com	F
10000004	Emily	emily@notexist.com	F
10000005	Mike	mike@notexist.com	M
10000006	James	james@notexist.com	M

UMID	Name
10000003	Mary
10000004	Emily

SELECT Statement

1. Pick the columns you specified right after SELECT
2. For each row, check conditions in WHERE clause
3. Display results that show up in both 1 and 2

UMID	Name	Email	Gender
10000001	Steve	steve@notexist.com	M
10000002	John	john@notexist.com	M
10000003	Mary	mary@notexist.com	F
10000004	Emily	emily@notexist.com	F
10000005	Mike	mike@notexist.com	M
10000006	James	james@notexist.com	M

Review – GROUP BY

- For each region, show the region name and average GDP of all countries in this region.

```
SELECT Region, AVG(GDP) AS AvgGDP
FROM Countries GROUP BY Region
```

Name	Region	Area	Population	GDP
Afghanistan	South Asia	652225	26000000	
…	…	…	…	…
Zimbabwe	Africa	390759	12900000	6192000000

Region	AvgGDP
South Asia	69990000
…	…
Africa	23999900

Review – GROUP BY

- Note: Attributes in **SELECT** clause outside of aggregate functions must appear in **GROUP BY** list
- For example

WRONG!

```
SELECT Name, Region, AVG(GDP) AS AvgGDP
FROM Countries GROUP BY Region
```

- Would create an error, because Name is not in the **GROUP BY** clause

HAVING Clause

- **HAVING** specifies additional conditions for an aggregate function or **GROUP BY** statement
- Example:
- Show the region name and average GDP of region with average GDP > 20000000

```
SELECT Region, AVG(GDP) FROM Countries
GROUP BY Region HAVING AVG(GDP) > 20000000
```

Differences Between **HAVING** and **WHERE**

- **HAVING** clause are applied AFTER the formation of groups
- Conditions in the **WHERE** clause are applied BEFORE forming groups
- You cannot use aggregate functions in **WHERE** clause

HAVING Clause

- However, sometimes you can use both:

```
SELECT Region, AVG(GDP) FROM Countries
 WHERE region='Africa' OR region='Middle
   East'GROUP BY Region
```

- Will give you the same result as

```
SELECT Region, AVG(GDP) FROM Countries
 GROUP BY Region HAVING region='Africa'
  OR region='Middle East'
```

- Because no matter if you filter the region name before or after the aggregation, you will have the same result…"Region" is not aggregated, it's already a field in the table…

LIKE Operator

- LIKE operator is used to find values in a field that match the pattern you specify
- For example, select all countries names start with "B"

```
SELECT Name FROM Countries
WHERE Name LIKE 'B*'
```

- Wildcard character:
- '*' match zero or more characters
- '?' match a single character
- '#' match a single digit (0-9)

Example of **LIKE** Operator

- Select countries with name containing the word 'land'

```
SELECT Name FROM Countries
WHERE Name LIKE '*land*'
```

- Result:

CountryName
British Virgin Islands
Cayman Islands
Switzerland
Cocos Islands
...
Solomon Islands
Thailand
Turks and Caicos Islands
Virgin Islands
Christmas Island

DISTINCT Operator

- By default, the SQL result contains duplicated rows. **DISTINCT** operator can be used to remove duplicates
- For example, if we want to see all the regions listed in the *Countries* table:

```
SELECT DISTINCT Region FROM Countries
```

- If you don't use **DISTINCT**, it will show you all 200+ records, most of them duplicates

Query On Multiple Tables

- Now we know how to select data from one table.
- But sometimes we need to get data from multiple tables
- Use **INNER JOIN** clause to “join” two or more tables

Example

- Using the following database, for each person find the maximum work hours. Show the Person Name and corresponding Maximum Hours.

Employees					
EmployeeID	PersonName	Address	Email	Phone	Salary
1	Steve	1234 Fuller	steve@notexist.com	734-333-9999	2000
2	John	234 Huron St.	john@notexist.com	734-233-8777	3000
3	Mary	2489 Stone Road	mary@notexist.com	734-876-8888	4000
4	Emily	1254 Green Road	emily@notexist.com	734-233-9089	2500
5	Mike	333 Fifth	mike@notexist.com	734-344-0934	2900
6	James	255 Plymouth	james@notexist.com	734-333-4000	4500

EmployeeProject		
EmployeeID	ProjectID	WorkHours
1	1	5
2	2	4
2	3	7
3	3	10
4	2	5
5	1	3
5	2	3
6	1	8
6	3	2

SELECT

```
SELECT Employees.PersonName,
MAX(WorkHours) AS MaxHours FROM
Employees INNER JOIN EmployeeProject
ON Employees.EmployeeID =
EmployeeProject.EmployeeID
GROUP BY Employees.PersonName
```

PersonName	MaxHours
Emily	5
James	8
John	7
Mary	10
Mike	3
Steve	5

Other Ways of Joining Tables

- Sometimes INNER JOIN is not enough
- What if there are "mismatches" between two tables?

Ordinals

Number	OrdinalValue
1	First
2	Second
3	Third
4	Fourth

SpelledValues

Number	SpelledValue
1	One
2	Two
5	Five
6	Six

Other Ways of Joining Tables

- LEFT JOIN
- RIGHT JOIN

LEFT JOIN

- **LEFT JOIN** select all records from the left table even if there are no matching records in the right table

SELECT * **FROM** Ordinals **LEFT JOIN** SpelledValues
ON Ordinals.Number = SpelledValues.Number
ORDER BY Ordinals.Number

Ordinals

Number	OrdinalValue
1	First
2	Second
3	Third
4	Fourth

SpelledValues

Number	SpelledValue
1	One
2	Two
5	Five
6	Six

Number	Ordinal Value	Spelled Value
1	First	One
2	Second	Two
3	Third	NULL
4	Fourth	NULL

RIGHT JOIN

- **RIGHT JOIN** select all records from the right table even if there are no matching records in the left table

SELECT * **FROM** Ordinals **RIGHT JOIN** SpelledValues
ON Ordinals.Number = SpelledValues.Number
ORDER BY Ordinals.Number

Ordinals

Number	OrdinalValue
1	First
2	Second
3	Third
4	Fourth

SpelledValues

Number	SpelledValue
1	One
2	Two
5	Five
6	Six

Number	Ordinal Value	Spelled Value
1	First	One
2	Second	Two
5	NULL	Five
6	NULL	Six

Nested **SELECT** **(non-correlated subquery)**

- You can use **IN** operator to nest sub-queries:

```
SELECT CountryName, Region FROM Countries
  WHERE Region IN
    (SELECT Region FROM Countries
      WHERE CountryName='Brazil' OR
      Name='Mexico')
```

- This query shows each country and its region in the same region as 'Brazil' or 'Mexico'.
- The **SELECT** query inside the (…) is called sub-query or nested query.
- This is an example of a **non-correlated subquery**: The subquery executes only once

Nested **SELECT** **(non-correlated subquery)**

- You can use binary operator to test the values from sub-query:

```
SELECT Name FROM Countries
  WHERE Population >
      (SELECT Population FROM Countries
       WHERE Name='Russia')
```

- This shows each country name where the population is larger than 'Russia'

Nested SELECT (non-correlated subquery)

- Use the operator **ALL** or **ANY** when the sub-query have multiple values

```
SELECT Name FROM Countries
  WHERE Population > ALL
      (SELECT Population FROM Countries
       WHERE Region='Europe')
```

- Equivalent to

```
SELECT Name FROM Countries
  WHERE Population >
   (SELECT MAX(Population) FROM Countries
       WHERE Region='Europe')
```

Nested **SELECT** (non-correlated subquery)

- Where

```
SELECT Name FROM Countries
  WHERE Population > ANY
      (SELECT Population FROM Countries
       WHERE Region='Europe')
```

- Equivalent to

```
SELECT Name FROM Countries
  WHERE Population >
   (SELECT MIN(Population) FROM Countries
       WHERE Region='Europe')
```

Example of Nested **SELECT (correlated subqueries)**

- Find the largest country in each region, show the region, the name (of the country) and the population (of the country)

```
SELECT Region, CountryName, Population
  FROM Countries AS x
  WHERE Population >= ALL
    (SELECT Population FROM Countries AS y
        WHERE y.Region=x.Region)
  ORDER BY Population DESC
```

- Use alias (**AS** …) to distinguish tables with the same name.
- This is a **correlated subquery** because the subquery is executed multiple times

Same result (without the ALL clause)

```
SELECT Region, CountryName, Population
FROM Countries AS x
WHERE Population =
    (SELECT MAX(Population) FROM Countries
        AS y WHERE y.Region=x.Region)
ORDER BY Population DESC
```

Useful Queries In Data Mining Projects

- Customers Per Household
  - Frequency Tables
- Mean Time Between Purchases
- Recency
- Longevity

Customers in Household

```
SELECT householdid, COUNT(*) as numinhousehold
    FROM customer
    GROUP BY householdid
```

Distribution of Customers Per Household

```
SELECT numinhousehold, COUNT(*) as numhh
FROM (SELECT householdid, COUNT(*) as numinhousehold
      FROM customer
      GROUP BY householdid)
GROUP BY numinhousehold
ORDER BY numinhousehold
```

Payment Type by Amount of Transaction

```
SELECT paymenttype, SUM(IIF(0 <= totalprice AND totalprice < 10, 1, 0)) AS cnt_0_10_USD, SUM(IIF(10 <= totalprice AND totalprice < 100,1,0)) AS cnt_10_100USD, SUM(IIF(100 <= totalprice AND totalprice < 1000,1,0)) AS cnt_100_1000USD, SUM(IIF(totalprice >= 1000,1,0)) AS cnt_1000USD, COUNT(*) AS cnt, format(SUM(totalprice),"CURRENCY") AS revenue
FROM orders
GROUP BY paymenttype
ORDER BY paymenttype;
```

DATEDIFF Function

- Calculates the difference between 2 dates.
- Syntax:
  - DateDiff (interval, date1, date2)

DateDiff ("yyyy", #15/10/1998#, #22/11/2003#)
Result: 5

DateDiff ("m", #15/10/2003#, #22/11/2003#)
Result: 1

DateDiff ("d", #15/10/2003#, #22/11/2003#)
Result: 38

Interval	Explanation
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

Other useful functions

- DateValue: converts a string to a date.
  - Syntax: DateValue (string_date)

- Format
  - Syntax: Format(Number, "###.##")

Format Function

- Format (expression, [format])

Format	Explanation
General Number	Displays a number without thousand separators.
Currency	Displays thousand separators as well as two decimal places.
Fixed	Displays at least one digit to the left of the decimal place and two digits to the right of the decimal place.
Standard	Displays the thousand separators, at least one digit to the left of the decimal place, and two digits to the right of the decimal place.
Percent	Displays a percent value - that is, a number multiplied by 100 with a percent sign. Displays two digits to the right of the decimal place.
Scientific	Scientific notation.
Yes/No	Displays No if the number is 0. Displays Yes if the number is not 0.
True/False	Displays False if the number is 0. Displays True if the number is not 0.
On/Off	Displays Off if the number is 0. Displays On is the number is not 0.

https://msdn.microsoft.com/en-us/library/office/jj720239.aspx

Recency

```
SELECT householdid, MAX(orderdate) as maxdate,
DATEDIFF("d", maxdate, datevalue("09/30/2014")) as
recency_days
    FROM orders as o INNER JOIN customer as c ON
o.customerid = c.customerid
    WHERE orderdate < datevalue("09/30/2014")
    GROUP BY householdid
ORDER BY householdid;
```

Longevity

```
SELECT householdid, MIN(orderdate) as mindate,
DATEDIFF("d", mindate, datevalue("09/30/2014")) as longevity_days
    FROM orders as o INNER JOIN customer as c ON o.customerid = c.customerid
    WHERE orderdate < datevalue("09/30/2014")
    GROUP BY householdid
ORDER BY householdid;
```

Mean Time Between Purchases

```
SELECT householdid, MIN(orderdate) as mindate,
            MAX(orderdate) as maxdate, COUNT(*) as
numorders, DATEDIFF("d", mindate, maxdate) /
(numorders - 1) as MeanTimeBP_Days
    FROM orders as o INNER JOIN customer as c ON
o.customerid = c.customerid
    WHERE orderdate < datevalue("09/30/2014")
    GROUP BY householdid
    HAVING COUNT(*) > 1
ORDER BY householdid
```

Full Table (Create Final Table)

SELECT householdid, Max(z.hhmedincome) AS ZipHHMedIncome, MIN(orderdate) as mindate, MAX(orderdate) as maxdate, COUNT(*) as numorders, DATEDIFF("d", maxdate, datevalue("09/30/2014")) as recency_days, DATEDIFF("d", mindate, datevalue("09/30/2014")) as longevity_days, round(DATEDIFF("d", mindate, maxdate) / (numorders - 1), 0) as MeanTime_Days, SUM(IIF(o.campaignid=2173, 1, 0)) as NumCampaign2173, SUM(IIF(o.campaignid=2173, 1, 0))/Count(*)*100 AS PercentCampaign2173, MAX(IIF(o.campaignid=2173, 1, 0)) as Campaign2173IND INTO FinalTable

FROM (orders AS o INNER JOIN customer AS c ON o.customerid = c.customerid) INNER JOIN Zipcensus As z ON o.zipcode=z.zipcode

WHERE orderdate < datevalue("09/30/2014")

GROUP BY householdid

HAVING COUNT(*) > 1 And Count(*)<=10

ORDER BY c.householdid;

Exam I Review
Part 2

Why learn VBA?

- It comes with Excel, Excel is the most popular business application available
  - VBA works inside Excel, no special software needed
- Easy access to Excel controls
- Custom dialog boxes
- Custom worksheet functions
- Custom user interface and menus
- Add-in files that can be run directly in excel

Objects

- Objects are Excel elements that can be manually manipulated or via a macro. Here are some examples of Excel objects:
  - The Excel application
  - A workbook
  - A worksheet in a workbook
  - A range or a table in a worksheet
  - A ListBox control on a UserForm (a custom dialog box)
  - A chart in a worksheet
  - A chart series in a chart
  - A data point in a chart
  - …

Procedures

- Unit of computer code that performs an action
  - Sub procedures – series of statements executable in different ways. Example:

```
Sub One()
    Sum=1+2
    MsgBox "The Sum of 1+2=" & Sum
End Sub
```

  - Function procedures – return values (or an array). Can be also called from within other procedures or as an excel function in your worksheet. Example:

```
Function AddThree (arg1, arg2, arg3)
    AddThree=arg1+arg2+arg3
End Function
```

Object Hierarchy

- When referring to an object, we have to follow object hierarchy by using dots (.) as separators between hierarchies. For example when referring to a workbook named lecture6.xlsx, we would use:
  - Application.Workbooks("lecture6.xlsx")
- Similarly, if we want to refer to sheet1 inside lecture6.xlsx:
  - Application.Workbooks("lecture6.xlsx").Worksheets("sheet1")
- And if we refer to cell A10 in sheet1:
  - Application.Workbooks("lecture6.xlsx").Worksheets("sheet1").Range("A10")

Active Objects

- To simplify specific references to an object, we can use shortcuts.
- If lecture6-code.xlsm is the active workbook then we can simplify the reference to cell A10 as:
  - Worksheets("sheet1").Range("A10")
- Similarly, if sheet1 is the active sheet in the workbook, we can also just refer to the cell directly:
  - Range("A10")

Object Properties

- Each object has properties (or settings). Examples:
  - Range objects properties: ***Value***, ***Address***
    - We can directly refer to the value in cell A10 of sheet1 with this statement:
      Worksheets("sheet1").Range("A10").Value

VBA Variables

- Can assign values to variables directly or from cell values:
  - Var1=3000+2000
  - Var1=Worksheets("sheet1").Range("A10").Value
- Or can assign values to a cell from a variable
  - Worksheets("sheet1").Range("A10").Value=Var1

Object Methods

- Methods are actions performed with the object (similar to clicking a key or button in excel)
- For example: Worksheets("sheet1").Range("A10").ClearContents
  - This would clear the contents of cell A10 in sheet1

Events

- Applicable to some objects:
  - Opening a workbook triggers the event: Workbook_Open
  - Activating a workbook triggers the event: Workbook_Activate
  - Changing a cell in a worksheet triggers the event: Worksheet_Change

MsgBox - Syntax

- MsgBox function has five arguments (those in square brackets are optional):
- MsgBox(prompt[, buttons][, title][, helpfile, context])
  - prompt: (Required) The message displayed in the pop-up display.
  - buttons: (Optional) specifies which buttons and icons, if any, to appear in the message box. Use built-in constants (e.g. vbYesNo)
  - title: (Optional) text that appears in the message box's title bar. Default is Microsoft Excel.
  - helpfile: (Optional) You can have a Help file associated with the message box.
  - context: (Optional) The context ID of the Help topic.

MsgBox

- The value returned can be assigned to a variable, or use the function by itself. Examples:

```
Ans = MsgBox("Continue?", vbYesNo +
vbQuestion, "Tell me")
If Ans = vbNo Then Exit Sub
```

  - Notice the use of the sum of the two built-in constants (vbYesNo + vbQuestion) for the buttons argument. Using vbYesNo displays two buttons: Yes and No. Adding vbQuestion displays a question mark icon. When the first statement is executed, The variable "Ans" contains one of two values, represented by the constant vbYes or vbNo. If the user clicks the No button, the procedure ends.

Example 1 – Writing a simple Sub

- Insert a VBA Module into a project
- Type/Copy the following code:

```
Sub SayHello()
    Msg = "Is your username: " & Application.UserName & "?"
    Ans = MsgBox(Msg, vbYesNo)
    If Ans = vbNo Then
        MsgBox "Oops, My Bad"
        Range("A1") = "User Unknown!"
    Else if Ans=vbYes
        MsgBox "I knew it!"
        Range("A1") = "User: " & Application.UserName
    End If
End Sub
```

Application Properties

Property	Object Returned
ActiveCell	The active cell
ActiveChart	Active chart sheet or chart contained in a ChartObject or a worksheet
ActiveSheet	The active sheet, either a worksheet or a chart
ActiveWindow	The active window
ActiveWorkbook	The active workbook
Selection	Could be a Range object (cell or cells), Shape, ChartObject
ThisWorkbook	Workbook containing the VBA procedure in execution

Range Objects

- Two Common Syntaxes:
  - Object.Range(cell1)
  - Object.Range(cell1, cell2)
- Other examples:
  - Assigning a value to a specific cell:
    Worksheets("Sheet1").Range("A10").Value=15.2
  - Assigning a value to a named cell (e.g. Input):
    Worksheets("Sheet1").Range("Input").Value=150
  - Assigning a value to a specific range of cells:
    Worksheets("Sheet1").Range("A1:B10").Value=2

Cells Property

- Another way of referring to a range
- Cells property works like the Range property:
  - Object.Cells(rowIndex, columnIndex):
    - Worksheets("Sheet1").Cells(1,1)=10 (assigns a value of 10 to cell A1)
  - Object.Cells(rowIndex) – The argument in the parenthesis can vary from 1 to 17,179,869,184! In Excel 2010 each cell is numbered starting with cell A1 (1) going right and then down. So Cell A2 is the 16,385…
  - Object.Cells – This can be used when referring to a cell within a range: Range("D5:E7").Cells(2,1)=2 would assign a value of 2 to cell D6

Offset Property

- Only applies to a Range object…
- Syntax: object.Offset(rowOffset,columnOffset)
- The 2 arguments correspond to the relative position from the upper-left cell of the range object. For example:
  - ActiveCell.Offset(1,0).Value=15 – This enters a value of 15 into the cell directly below the active cell
  - ActiveCell.Offset(-1,-1).Value=15 – This enters a value of 15 into the cell above and to the left of the active cell

Data Types

Data Type	Bytes Used	Range of Values
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	–32,768 to 32,767
Long	4 bytes	–2,147,483,648 to 2,147,483,647
Single	4 bytes	–3.402823E38 to –1.401298E-45 (for negative values); 1.401298E-45 to 3.402823E38 (for positive values)
Double	8 bytes	–1.79769313486232E308 to –4.94065645841247E-324 (negative values);4.94065645841247E-324to 1.79769313486232E308 (for positive values)
Currency	8 bytes	–922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	12 bytes	+/–79,228,162,514,264,337,593,543,950,335 with no decimal point;+/–7.9228162514264337593543950335 with 28 places to the right of the decimal

More Data Types

Data Type	Bytes Used	Range of Values
Date	8 bytes	January 1, 0100 to December 31, 9999
Object	4 bytes	Any object reference
String (variable length)	10 bytes + string length	0 to approximately 2 billion characters
String (fixed length)	Length of string	1 to approximately 65,400 characters
Variant (with numbers)	16 bytes	Any numeric value up to the range of a double data type. It can also hold special values, such as Empty, Error, Nothing, and Null.
Variant (with characters)	22 bytes + string length	0 to approximately 2 billion
User-defined	Varies	Varies by element

Explicit Variable Declaration

- To force yourself to declare all the variables that you use, include the following as the first instruction in your VBA module:

  Option Explicit

- When this statement is present, VBA won't execute a procedure if it contains an undeclared variable name.

Local Variables

- Declared within a procedure.
  - Local variables can only be used in the procedure in which they're declared. When the procedure ends, the variable no longer exists
  - If you need the variable to retain its value when the procedure ends, declare it as a Static variable.
- Most common way to declare a local variable is to place a Dim statement between a Sub statement and an End Sub statement.
  - Dim statements usually are placed right after the Sub statement, before the procedure's code.

Local Variables

- Can also declare several variables with a single Dim statement. For example:
  - Dim x As Integer, y As Integer, z As Integer
  - Dim First As Long, Last As Double
- VBA doesn't allow declaration of a group of variables by separating the variables with commas.
  - The following statement, does not declare all the variables as integers: Dim i, j, k As Integer (only k)
  - Instead use: Dim i As Integer, j As Integer, k As Integer

Public Variables

- Public variables are available to all the procedures in all the VBA modules in a project
  - Declare by using the Public keyword rather than Dim:

    Public CurrentRate as Long
  - This makes CurrentRate variable available to any procedure in the VBA project,
  - Insert this statement before the first procedure in a module (any module).
  - This type of declaration must appear in a standard VBA module, not in a code module for a sheet or a UserForm.

Constants

- Declare constants with the Const statement. Here are some examples:

  Const NumQuarters as Integer = 4

  Const Rate = .0725, Period = 12

  Const ModName as String = “Budget Macros”

  Public Const AppName as String = “Budget Application”

- If you attempt to change the value of a constant in your code, you will get the error “Assignment to constant not permitted”.

Predefined Constants

- Predefined constants can be used without declaring
- Example - Page orientation uses built-in constants: xlLandscape or xlPortrait

```
Sub SetToLandscape()
ActiveSheet.PageSetup.Orientation = xlLandscape
End Sub
```

- The Object Browser can display a list of all Excel and VBA constants. In the VBE, press F2 to bring up the Object Browser.

Declaring arrays

- Use the Dim or Public statements (just as you declare a regular variable)
  - Specify the number of elements in the array.
    - First index number,
    - The keyword ”To”,
    - And the last index number — all inside parentheses.
  - For example, an array comprising exactly 150 integers:

```
Dim MyArray(1 To 150) As Integer
```

More on Array Declaration

- If specifying only the upper index, VBA assumes that 0 is the lower index. Example:

```
Dim MyArray(0 to 100) As Integer
Dim MyArray(100) As Integer
```

  - In both cases, the array consists of 100 elements.
- By default, VBA assumes zero-based arrays.
- If you would like VBA to assume that 1 is the lower index (for all arrays that declare only the upper index), use the following before any procedures in your module:

```
Option Base 1
```

Declaring multidimensional arrays

- VBA arrays can have up to 60 dimensions, although you'll rarely need more than three dimensions (a 3-D array). The following statement declares a 10x10 Array(Matrix) with two dimensions (2-D):

```
Dim MyArray(1 To 10, 1 To 10) As Integer
```

  - To refer to a specific element in a 2-D array, you need to specify two index numbers. We can assign a value to an element in the preceding array:

```
MyArray(3, 4) = 125
```

Object Variables

- An object variable represents an entire object, such as a range or a worksheet. Object variables are important for two reasons:
  - simplify your code significantly.
  - make your code execute more quickly.
- Object variables, like normal variables, are declared with the Dim or Public statement.

Object Variables

- For example, the following statement declares InputArea as a Rangeobject variable:

```
Dim InputArea As Range, MyArea2 as Range
```

- Use the Set keyword to assign an object to the variable. For example:

```
Set InputArea = Range("C16:E16")
Set MYArea2=Range("J8:O15")
```

Manipulating Objects and Collections

- Two important constructs that can simplify working with objects and collections:
  - With-End With constructs
  - For Each-Next constructs

With-End With

- We can rewrite using the With-End With construct. The following procedure performs exactly like the preceding one:

```
Sub ChangeFont2()
With Selection.Font
    .Name = "Cambria"
    .Bold = True
    .Italic = True
    .Size = 12
    .Underline = xlUnderlineStyleSingle
    .ThemeColor = xlThemeColorAccent1
End With
End Sub
```

For Each-Next constructs

- When using For Each-Next construct we don't have to know how many elements are in a collection.
- The syntax of the For Each-Next construct is:

```
For Each [element] In [collection]
    [instructions]
    [Exit For]
    [instructions]
Next [element]
```

For Each-Next Example

- VBA provides a way to exit a For-Next loop before all the elements in the collection are evaluated. Do this with an Exit For statement. The example that follows selects the first negative value in Row 1 of the active sheet:

```
Sub SelectNegative()
Dim Cell As Range
For Each Cell In Selection
    If Cell.Value < 0 Then
      Cell.Select
       Exit For
    End If
Next Cell
End Sub
```

  - If a cell is negative, it's selected, and then the loop ends when the Exit For statement is executed.

For-Next loops

- The simplest type of a good loop is a For-Next loop. Its syntax is:

```
For counter = start To end [Step stepval]
    [instructions]
    [Exit For]
    [instructions]
Next [counter]
```

Do While loops

- This section describes another type of looping structure available in VBA. Unlike a For-Next loop, a Do While loop executes as long as a specified condition is met.
- A Do While loop can have either of two syntaxes:

```
Do [While condition]
[instructions]
[Exit Do]
[instructions]
Loop
```

```
Do
[instructions]
[Exit Do]
[instructions]
Loop [While
condition]
```

  - The difference between is the point in time when the condition is evaluated. In the syntax to the left, the contents of the loop may never be executed. In the syntax to the right, the statements inside the loop are executed at least one time.

Controlling Code Execution

- Some VBA procedures start at the top and progress line by line to the bottom. (e.g. Macros).
- Often, we need to control the flow of your routines by skipping, executing multiple times, and testing conditions.
- Some additional ways of controlling the execution of your VBA procedures:
  - GoTo statements
  - If-Then constructs
  - Select Case constructs

GoTo statements

- The most straightforward way to change flow is to use a GoTo statement.
  - This statement simply transfers program execution to a new instruction, which must be preceded by a label (a text string followed by a colon, or a number with no colon).
  - VBA procedures can contain any number of labels, but a GoTo statement can't branch outside of a procedure.

If-Then constructs

- The most commonly used instruction grouping in VBA
- Decision-making capability. Good decision-making is the key to writing successful programs.
- Basic syntax of the If-Then construct is

  If condition Then true_instructions [Else false_instructions]

- The Else clause is optional.

If-Then

- In the previous example, every IF statement is evaluated (even if the first condition is satisfied) A more efficient procedure would end the routine when a condition is found to be True. This is the syntax:

```
If condition Then
[true_instructions]
[ElseIf condition-n Then
[alternate_instructions]]
[Else
[default_instructions]]
End If
```

VBA's IIf function

- Alternative to the If-Then construct: the IIf function.
- This function takes three arguments and works like Excel's IF worksheet function. The syntax is
- IIf(expr, truepart, falsepart)
  - expr: (Required) Expression you want to evaluate.
  - truepart: (Required) Value or expression returned if expr is True.
  - falsepart: (Required) Value or expression returned if expr is False.

Select Case constructs

- More useful for choosing among three or more options.
  - Also works with two options and is a good alternative to If-Then-Else.
- Syntax:

```
Select Case testexpression
[Case expressionlist-n
[instructions-n]]
[Case Else
[default_instructions]]
End Select
```

Working with Ranges

- Very important for many more complex programs. Examples:
  - Selecting/copying/moving a range
  - Identifying types of information in a range
  - Prompting for a cell value
  - Finding the first empty cell in a column
  - Pausing a macro to allow the user to select a range
  - Counting cells in a range

Copying a range

- A very simple copy-and-paste operation can be done in five lines of VBA code:

```
Sub Macro1()
Range("A1").Select
Selection.Copy
Range("B1").Select
ActiveSheet.Paste
End Sub
```

Copying a range

- Another way to approach this task is to use object variables to represent the ranges, as shown in the code that follows:

```
Sub CopyRange3()
Dim Rng1 As Range, Rng2 As Range
Workbooks.Open ThisWorkbook.Path & "\File1.xlsx"
Workbooks.Open ThisWorkbook.Path & "\File2.xlsx"
Set Rng1 =
Workbooks("File1.xlsx").Sheets("Sheet1").Range("A1")
Set Rng2 =
Workbooks("File2.xlsx").Sheets("Sheet2").Range("A1")
Rng1.Copy Rng2
End Sub
```

Moving a range

- Very similar to copying a range
- The difference is in the use of the Cut method instead of the Copy method.
  - Note that you need to specify only the upper-left cell for the destination range.
- The following example moves 18 cells (in A1:C6) to a new location, beginning at cell H1:

```
Sub MoveRange1()
Range("A1:C6").Cut Range("H1")
End Sub
```

Copying a variably sized range

- In many cases, you need to copy a range of cells, but you don't know the exact row and column dimensions of the range. For example, you might have a workbook that tracks weekly sales, and the number of rows changes weekly when you add new data.
- The following macro demonstrates how to copy this range from Sheet1 to Sheet5 (beginning at cell A1). It uses the CurrentRegion property, which returns a Range object that corresponds to the block of cells around a particular cell (in this case, A1).

```
Sub CopyCurrentRegion2()
Range("A1").CurrentRegion.Copy Sheets("Sheet5").Range("A1")
End Sub
```

- **Generally, the CurrentRegion property setting consists of a rectangular block of cells surrounded by one or more blank rows or columns.**

Selecting various types of ranges

- In addition to the CurrentRegion property, we can use the End method of the Range object.
  - The End method takes one argument, which determines the direction in which the selection is extended. The following statement selects a range from the active cell to the last non-empty cell:

    Range(ActiveCell, ActiveCell.End(xlRight)).Select
  - A similar example that uses a specific cell as the starting point:

    Range(Range("A20"), Range("A20").End(xlUp)).Select

Pausing a macro to get a user-selected range

- In some situations, you may need an interactive macro.
  - For example, a macro that pauses while the user specifies a range of cells. This example describes how to do this with Excel's InputBox method.
  - Don't confuse Excel's InputBox method with VBA's InputBox function. Although these two items have the same name, they're not the same.

Counting selected cells

- You can create a macro that works with the range of cells selected by the user. Use the Count property of the Range object to determine how many cells are contained in a range selection. For example:

```
MsgBox Selection.Count
```

- If the active sheet contains a range named data, the following statement assigns the number of cells in the data range to a variable named CellCount:

```
CellCount = Range("data").Count
```

- We can also determine how many rows or columns are contained in a range:

```
Selection.Rows.Count
```

- We can also use the Rows property to determine the number of rows in a range. The following statement counts the number of rows in a range named data and assigns the number to a variable named RowCount:

```
RowCount = Range("A1:B3").Rows.Count
```

Deleting all empty rows

- The following procedure deletes all empty rows in the active worksheet.

- This routine is fast and efficient because it doesn't check all rows. It checks only the rows in the used range, which is determined by using the UsedRange property of the Worksheet object.

```
Sub DeleteEmptyRows()
Dim LastRow As Long
Dim r As Long
Dim Counter As Long
Dim ans As Variant
Application.ScreenUpdating = False
 MsgBox "Number of used rows: " & ActiveSheet.UsedRange.Rows.Count
   LastRow = ActiveSheet.UsedRange.Rows.Count + _
    ActiveSheet.UsedRange.Rows(1).Row - 1
   MsgBox "First Row: " & ActiveSheet.UsedRange.Rows(1).Row
   MsgBox "Last Row: " & LastRow
   ans = MsgBox("Continue?", vbYesNo)
   If ans = vbNo Then Exit Sub
   For r = LastRow To 1 Step -1
       If Application.WorksheetFunction.CountA(Rows(r)) = 0 Then
           Rows(r).Delete
           Counter = Counter + 1
       End If
   Next r
   Application.ScreenUpdating = True
   MsgBox Counter & " Empty rows were deleted."
End Sub
```

Determining a cell's data type

- Excel provides a number of built-in functions that can help determine the type of data contained in a cell. These include ISTEXT, ISLOGICAL, and ISERROR. In addition, VBA includes functions such as IsEmpty, IsDate, and IsNumeric.
- The following function, named CellType, accepts a range argument and returns a string (Blank, Text, Logical, Error, Date, Time, or Number) that describes the data type

```
Function CellType(Rng) As String
` Returns the cell type of the upper left
` cell in a range
Dim TheCell As Range
Set TheCell = Rng.Range("A1")
Select Case True
Case IsEmpty(TheCell)
CellType = "Blank"
Case Application.IsText(TheCell)
CellType = "Text"
Case Application.IsLogical(TheCell)
CellType = "Logical"
Case Application.IsErr(TheCell)
CellType = "Error"
Case IsDate(TheCell)
CellType = "Date"
Case InStr(1, TheCell.Text, ":") <> 0
CellType = "Time"
Case IsNumeric(TheCell)
CellType = "Number"
End Select
End Function
```

Pivot Tables

- Creating a pivot table from a database or list enables you to summarize data in ways that otherwise would not be possible — and it's amazingly fast and requires no formulas. You also can write VBA code to generate and modify pivot tables.

Data appropriate for a pivot table

- A pivot table requires that your data is in the form of a rectangular database (Normalized).
  - You can store the database in either a worksheet range (which can be a table or just a normal range) or an external database file.
- Fields in a database table consist of two types:
  - Data: Contains a value or data to be summarized. For the bank account example, the Amount field is a data field.
  - Category: Describes the data. For the bank account data, the Date, AcctType, OpenedBy, Branch, and Customer fields are category fields because they describe the data in the Amount field.

Examining the recorded code for the pivot table

- VBA code that works with pivot tables can be confusing. To make any sense of the recorded macro, you need to know about a few relevant objects:
  - PivotCaches: A collection of PivotCache objects in a Workbook object (the data used by a pivot table is stored in a pivot cache).
  - PivotTables: A collection of PivotTable objects in a Worksheet object.
  - PivotFields: A collection of fields in a PivotTable object.
  - PivotItems: A collection of individual data items within a field category.
  - CreatePivotTable: A method that creates a pivot table by using the data in a pivot cache.

Adding calculated fields and filters

- Calculated fields:

```
pt.CalculatedFields.Add "Variance", "=Budget-Actual"
pt.PivotFields("Variance").Orientation = xlDataField
```

- Filter for column/row fields:

```
pt.PivotFields("Month").PivotFilters.Add_
Type:=xlCaptionEquals, Value1:="Jan"
```

If numerical value you could also use Type:=xlValueEquals, Value1:=number

- Filter for page filters:

```
pt.PivotFields("Division").CurrentPage = "N. America"
```

```
Sub CreatePivotTable()
Dim PTCache As PivotCache
Dim PT As PivotTable
' Create the cache
Set PTCache = ActiveWorkbook.PivotCaches.Create( _
SourceType:=xlDatabase, _
SourceData:=Range("A1").CurrentRegion)
' Add a new sheet for the pivot table
Worksheets.Add
' Create the pivot table
Set PT = ActiveSheet.PivotTables.Add( _
PivotCache:=PTCache, _
TableDestination:=Range("A3"))
' Specify the fields
With PT
.PivotFields("Region").Orientation = xlPageField
.PivotFields("Month").Orientation = xlColumnField
.PivotFields("SalesRep").Orientation = xlRowField
.PivotFields("Sales").Orientation = xlDataField
'no field captions
.DisplayFieldCaptions = False
End With
End Sub
```

User Forms

- User Forms are custom dialog boxes that give us a lot of flexibility to interface with application users.
- Toolbox Controls:
  - CommandButton
  - ListBox
  - RefEdit

Using CommandButtons in a UserForm

- Each CommandButton has its own event-handler procedure. For example, the following procedure is executed whenCommandButton1 is clicked:

```
Private Sub CommandButton1_Click()
Me.Hide
Call Macro1
Unload Me
End Sub
```

- This procedure hides the UserForm, calls Macro1, and then closes the UserForm. The other buttons have similar event-handler procedures.

Using a ListBox in a UserForm

- Before the UserForm is displayed, its Initialize event-handler procedure is called. This procedure, which follows, uses the AddItem method to add six items to the ListBox:

```
Private Sub UserForm_Initialize()
With ListBox1
.AddItem "Macro1"
.AddItem "Macro2"
.AddItem "Macro3"
.AddItem "Macro4"
.AddItem "Macro5"
.AddItem "Macro6"
End With
End Sub
```

Link values selected in list box to actions

- The Execute button also has a procedure to handle its Click event:

```
Private Sub ExecuteButton_Click()
Select Case ListBox1.ListIndex
Case -1
MsgBox "Select a macro from the list."
Exit Sub
Case 0: Call Macro1
Case 1: Call Macro2
Case 2: Call Macro3
Case 3: Call Macro4
Case 4: Call Macro5
Case 5: Call Macro6
End Select
Unload Me
End Sub
```

Selecting Ranges from a UserForm

- UserForms can allow you to select a range, thanks to the RefEdit control. The RefEdit control doesn't look exactly like the range selection control used in Excel's built-in dialog boxes, but it works in a similar manner
- Example UserForm contains a RefEdit control. This dialog box enables the user to perform a simple mathematical operation on all nonformula (and non-empty) cells in the selected range. The operation that's performed corresponds to the selected OptionButton.

- The RefEdit control returns a text string that represents a range address. You can convert this string to a Range object by using a statement such as
- Set UserRange = Range(RefEdit1.Text)
  - Initializing the RefEdit control to display the current range selection is good practice. You can do so in the UserForm_Initialize procedure by using a statement such as
    - RefEdit1.Text = ActiveWindow.RangeSelection.Address