Understanding Core Programming

Lesson Objectives

By the end of this lesson, you will be able to:

- ☐ Explain how computers store programs and data in memory.
- ☐ Demonstrate computer decision structures, including flowcharts and pseudo-code.
- ☐ Identify and explain the best ways to handle repetition.

Computer Storage and Data Types

Computer Memory

Computers store programs and their instructions in various types of memory. This section of the lesson will acquaint you with those types of memory, their functions and key terms that describe their use.

Primary Storage (Main Memory)

Primary storage is the only type of memory that is directly accessible by the *central processing unit (CPU)*. The CPU continuously reads instructions stored there and carries those instructions out as required. Any data actively operated on is also stored there.

Historically, primary storage in computers used tubes, or magnetic drums. By the mid-1950s, those unpredictable methods were almost entirely replaced by still unwieldy, magnetic core memory.

The invention of the solid-state (no moving parts) transistor changed everything. The transistor led to unprecedented miniaturization and increased reliability.

Modern *random access memory (RAM)* has taken over as the chief type of primary storage. RAM is physically small, light and much less expensive than its predecessors. There are many sub-types of RAM (DIMM, SIMM, DDR – now DDR2, and DDR3), but they behave in similar ways. They are *volatile* (they need electricity to retain data).

Virtual Memory

When the CPU needs to store information from a program, it will use the available main memory. However, there are times when not enough memory is available to store all the information that the CPU is trying to save. ***Virtual memory*** is hard drive disk space set aside to act like physical main memory. Although much slower than main memory, virtual memory can expand main memory so that programs that otherwise could not run can run.

Processor Registers

Processor registers are physically located inside the processor. Each register holds about a *word* of data. “Word” here means a chunk of data 32 bits in size, not a *word* like the one you find in dictionaries; so a dword (double word) would be 64 bits. You may encounter a 32-bit boundary in x86 home computers. The CPU instructs the *arithmetic and logic unit (ALU)* to perform calculations or logical operations on this data (or with the help of it). Registers provide the fastest response of all data storage.

Processor Cache (SRAM)

Cache is slower than registers, but is faster than other forms of memory further down the hierarchy. It is an extension of the memory of the registers; however, it only stores and does not compute.

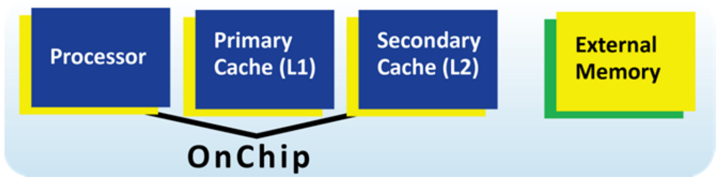


Figure 1-1

Multi-level cache setup is often used:

- *Primary cache* is the smallest and fastest, and located inside the CPU.
- *Secondary cache* is somewhat larger and slower, and is located just off the CPU.

The CPU communicates to the primary storage. If the information requested by the CPU is not in the registers or cache, the request is sent through a *memory bus* via a *memory management unit (MMU).*

The MMU determines if the information is in the RAM or needs to be retrieved from the mass storage device. The MMU does this with the use of its Translation Look-aside Buffer (TLB), which holds a table matching virtual addresses to physical addresses. Although the MMU is usually integrated into the CPU, it can sometimes be a separate chip.

BIOS (Basic Input/ Output System)

If all memory were volatile, computers would have to be taught how to start up every time they were turned on. Unlike RAM, registers and cache, BIOS is non-volatile. BIOS is primary storage containing a small startup program used to *bootstrap* the computer. Bootstrap, which comes from the phrase "pull yourself up by your bootstraps," means to read a larger program from non-volatile secondary storage to RAM and start to execute it.

The BIOS is often referred to as *Read-Only Memory (ROM)*. However, not only can it be accessed randomly (like RAM), but it can be altered – as in "update your BIOS," or "flash your BIOS" (delete the current instructions).

Secondary Storage

Secondary storage is not directly accessible by the CPU. The computer accesses it via input/output channels. Like the BIOS, it is non-volatile (does not lose the data when the device is powered down) and is much less expensive than primary storage.

Hard disk drives (HDDs) are usually used as secondary storage.

Access time (the length of time it takes to read data) varies based on the type of memory:

- HDD typically requires a few thousandths of a second (milliseconds).
- RAM needs billionths of a second (nanoseconds).

The physical structure and location of each type of the memory causes the variation in the access time. Primary memory is very close to the CPU and is solid state (it has no moving parts), whereas HDDs are farther away and are comprised of spinning media, which increases access time.

Some other examples of secondary storage technologies are:

- External HDDs
- Flash memory (e.g., USB flash drives/ keys/ thumb drives)
- Floppy disks (old, seldom used)
- Magnetic tape, (generally used only for backup)
- Paper tape (obsolete)
- Punched cards (obsolete)
- DVD/CD/Blu-Ray RAM disks

These other devices are often formatted according to a file system format (e.g., FAT, FAT32, NTFS and CDFS).

Tertiary Storage (Tertiary Memory)

Tertiary storage is often handled by robotic retrieval of HDDs or tape drives from a large physical array of disks.

The method of retrieving data, albeit slowly, is as follows:

1. The computer requests the information.
2. A database tells the robot where the data is.
3. The robot physically moves to the location in the warehouse.
4. It “picks” the media (usually tape backup).
5. It takes the media to the reader/ writer.
6. The reader accesses/writes the information requested.
7. The robot returns the media to its position in the warehouse.

Memory Stacks

The *stack* is a working area of memory that grows and shrinks dynamically with the demands of your executing program.

Memory stacks are regions of memory where data is added or removed in a last-in-first-out manner. As a function executes, it can add data to the top of the stack and be accessed quickly. As the function finishes, it will eliminate that data from the stack.

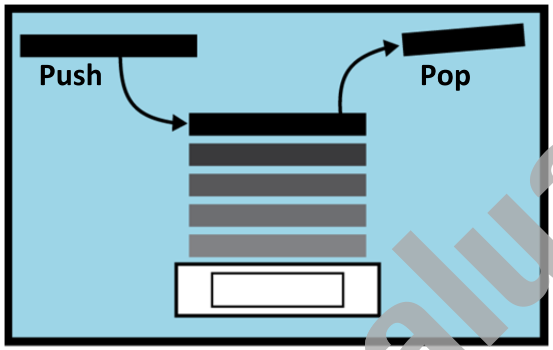


Figure 1-2

This method of storage is very efficient; however, data is transient and must be copied to other memory if it is needed after the function finishes.

The actual data stored on the stack is the address of the function call that allows the return statement to return to the correct location. Memory under control of the stack is said to have been allocated on the stack.

The stack’s size depends on the microprocessor. It can be as small as a few dozen kilobytes, or up to 64 KB, with 8 KB (or 2 pages) reserved for overflow error control. You can override the stack limit at compile time Assigning more data to the stack than is available can cause a “stack overflow” or “system access violation” error.

Heaps

Applications need to allocate specific amounts of memory to store their data. That allocated memory is called a heap. The application can make the heap any size, but is usually less than 16 KB. A heap, therefore, allows for memory optimization and isolation, and is independent of microprocessor page size restrictions.

Heap memory is the level above virtual memory.

When a process is created, a default heap is also created for the process. An application can use the process heap for its memory allocations, and the heap grows and shrinks accordingly. However, performance can suffer if the amount and type of memory allocations in the default heap cause the heap to become fragmented.

Note that the total size of objects allocated on the heap is limited only by your system's available virtual memory.

Data Types

Overview

When you are going to store a piece of data in the computer, you need to store it in a specific container. That container is called a *variable*. The different kinds of variables are called *data types*. Different data types are needed to store different kinds of data. Each programming language controls its own data types. In general, data types fall into three broad categories: Numeric, Text and Other.

Most programming languages have similar data types, even if the names differ slightly.

Historically, programmers used code such as the Dim statement to reserve space for the data that needed to be stored. The Dim statement shows very simply how the computer handles user data.

When Visual Basic programmers write the code: ***Dim A as Integer,*** they are telling the computer to "Reserve a Dimension of 4 bytes of space in memory, call it A, and allow it to hold a whole number between 2,147,483,648 through 2,147,483,647 (signed)."

When declaring a variable, you must remember two things:

- the type of data you want to store in it, and
- the size of data you want to store in it.

Although the term "Data Type" is most commonly used, "Data Type and Size" is actually a better term.

Type of Data

A variable can be compared to a physical container. For example, if you want to store water, a bucket is a much better container type than a three-ring binder, but if you were storing your autobiography, the binder would be preferred. We store numbers in numeric data types and text in text data types

Size of Data

For efficiency, you should reserve as little space as possible to hold your data. Just as you would not rent a 40-foot shipping container to hold a single phone book, you would not want to store the contents of a four-bedroom house in a shopping cart. You need the right size container for the data you want to store.

Data Types to Store Numbers

Certain factors must be considered when using numeric data types:

- Are decimals involved? On the other hand, will the stored number always be a whole number (integer)?
- Is a sign involved (such as the temperature is -3)?
- How big is the number?
- What level of accuracy is needed?

Integers (whole numbers) can be stored in variables of the following types:

- **Integer** (or **int32**), for regular-sized whole numbers (up to about 2 billion)
- **Long** (or **int64**), for large whole numbers (9.2 x 10^{18})
- **Short** (or **Int16**), for small whole numbers (up to about 33 thousand)
- **Byte,** for very small whole numbers (0 to 256)
- **SByte,** a byte with a positive or negative signed attached (from -127 to +127)

Decimals can be stored in variables of the following types:

- **Decimal**, for numbers requiring a high level of precision (up to about 28 digits)
- **Single**, for "regular-sized" floating-point numbers (up to 8 decimal accuracy and about 1 x 10^{40} digits in size)
- **Double**, for very large floating-point numbers (up to 18 decimal accuracy and about 1 x 10^{310} digits in size)

The memory size requirements for these data types are as follows:

Common Data type Names	Nominal storage allocation	Value range
Byte	1 byte	0 through 255 (unsigned)
Decimal	16 bytes	0 through +/-79,228,162,514,264,337,593,543,950,335 (+/-7.9...E+28) † with no decimal point; 0 through +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/-0.0000000000000000000000000001 (+/-1E-28) †
Double (double-precision floating-point)	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324 † for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 † for positive values
Integer / Int32	4 bytes	-2,147,483,648 through 2,147,483,647 (signed)
Long (long integer)/ Int64/ long long	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (9.2...E+18 †) (signed)
SByte	1 byte	-128 through 127 (signed)
Short (short integer)/ Int16	2 bytes	-32,768 through 32,767 (signed)
Single (single-precision floating-point)	4 bytes	-3.4028235E+38 through -1.401298E-45 † for negative values; 1.401298E-45 through 3.4028235E+38 † for positive values
UInteger/ UInt32	4 bytes	0 through 4,294,967,295 (unsigned)
Ulong/ UInt64	8 bytes	0 through 18,446,744,073,709,551,615 (1.8...E+19 †) (unsigned)
UShort/ UInt16	2 bytes	0 through 65,535 (unsigned)

Data Types to Store Text

Most programs also deal with text, whether displaying information or capturing text entered by the user. Text is usually stored in the *String* data type, which can contain a series of letters, numbers, spaces and other characters. A String can be of any length, from a sentence or a paragraph to a single character to nothing at all (a *null string*).

For a variable that will always represent just one character, there is also a Char data type. If you only need to hold one character in a single variable, you can use the Char data type instead of a String.

A character type (often called "char") may contain a single letter, digit, punctuation mark or control character.

Some languages have two or more character types:

- ASCII - single-byte type for characters.
- Unicode - multi-byte type for characters.

Characters can be combined into a string of characters. The string can include numbers and other numerical symbols but these will be treated as text. You cannot perform numeric calculations on strings. You can ask the computer to calculate the length of the string; thereby it can participate in a "count" calculation, but you cannot tell the computer to add two strings. You can, however, combine two strings using the process of concatenation which means "joining together" (see the examples below). You can store number characters in a string (like "6"). This data type is used to store numbers such as a phone number when numerical calculations will never be performed on the data.

Good programming practice involves using comments, or remarks, throughout the program to allow other programmers to understand what is happening and why you coded a certain way. Although most programming languages allow remarks, they use different methods of putting them into code. The code in this book follows the Visual Basic method, which is with a single quotation mark, a ‘. The computer does not read anything on the line after the quote. In older Visual Basic code, you will see the remark started with the “REM” statement; which is why you will hear programmers say, for example, “Just REM out that code,” meaning “Convert the code to remarks so the computer will not run it, but do not delete it in case you need to run the code later.”

Example 1:

```
Dim A As Integer
Dim B As Integer
Dim C As Integer
A = 6
B = 3
C = A + B
Print C ‘ this code would yield 9
```

Example 2:

```
Dim A As String
Dim B As String
Dim C As String
A = “6” ‘ string variables require quotes
B = “3”
‘C = A + B ‘ this code will cause an error, so I have REMed it out
C = A & B ‘ this code is valid, the ampersand (&) is the concatenation symbol
Print C ‘ this code would yield “63”
```

In most languages, a string is equivalent to an array of characters; however, Java uses distinct types (java.lang.String and char[ ]).

Literals for characters and strings are usually surrounded by quotation marks: sometimes, single quotation marks (') are used for characters, and double quotation marks (") are used for strings.

Examples of character literals in C syntax are:

'3'

'B'

'$'

'\t' (tab character)

Examples of string literals in C syntax are:

"B"

"My Dog"

Remarks in C are made by using a forward slash/asterisk combination (/*) as in /* blah blah blah */

Each character will require either 1 or 2 bytes to store the data, depending on the programming language used and the system being used. Character strings and string variables are simply multiple characters attached together; therefore, the memory size requirement is simply the number of characters multiplied by the size of each character (1 or 2 bytes).

Data Types to Store Other Information

In addition to text and numbers, programs sometimes need to store other types of information, such as a true or false value, a date or data that has a special meaning to the program.

Boolean

For values that can be represented as true/false, yes/no, or on/off, Visual Basic has the *Boolean* data type. A Boolean variable can hold one of two possible values: True or False.

Date

Although you can represent dates or times as numbers, the Date data type makes it easy to calculate dates or times, such as the number of days until your birthday or the number of minutes until lunch.

Object

In some cases, the type of data you need to store may be different at different times. The *Object* data type allows you to declare a variable and then define its data type later.

Structures: Your Own Data Types

A *structure* is a generalization of the user-defined type (UDT) supported by earlier versions of Visual Basic.

Structures are useful when you want a single variable to hold several related pieces of information. For example, you might want to keep an employee's name, telephone extension and salary together. You could use several variables for this information, or you could define a structure and use it for a single employee variable. The advantage of the structure becomes apparent when you have many employees and therefore many instances of the variable.

Composite

You can combine data items of different types to create a structure. A structure associates one or more elements with each other and with the structure itself. When you declare a structure, it becomes a composite data type, and you can declare variables of that type.

When you need to store more than one type of data in a single variable, you can use a composite data type. Composite data types include arrays, structures and classes.

The memory size requirements for these data types are as follows:

Common Data type Names	Nominal storage allocation	Value range
Boolean	Depends on implementing platform (often 1 byte)	True or False
DateTime/Date	8 bytes	0:00:00 (midnight) on January 1, 0001 through 11:59:59 PM on December 31, 9999
Object (class)	4 bytes on 32-bit platform 8 bytes on 64-bit platform	Any type can be stored in a variable of type Object
User-Defined (structure)/ (inherits from ValueType)	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members

Understand Computer Decision Structures

Overview

Computer programs can run in a linear fashion. That is, they can follow step-by-step procedures and reach the same conclusion every time. However, doing so would lead to very simplistic applications. Programs become powerful when we allow them to make decisions based on a certain dynamic set of circumstances.

Before we get into any Decision Structure (in which the computer determines the correct path to take), we will first examine some simple graphic methods to represent what is going on inside various computer programs. Those methods are flowcharts and pseudo code.

Flowcharts

A *flowchart* is a visual representation of a step-by-step solution to a problem.

Flowcharts are used in analyzing, designing, documenting or managing a process or program. They are not restricted to computer programming; any process can be flow-charted.

A flowchart is a common way to represent an algorithm (i.e., process). Each step in the process is represented by a box. The function of the step determines the shape of the box. Arrows linking the boxes show the order and potential paths. A flowchart truly shows the “flow” of the process and therefore the flow of the programming.

The flowchart can be considered the most important part of the program because all the important decisions are made there. It might be easier to get sign-off from a non-technical manager.

Flowcharts are programming language-independent, so they can be created first, even before a language is selected.

The following example demonstrates the simplest type of flowchart, a linear progression (for the example we will chart washing a dog):

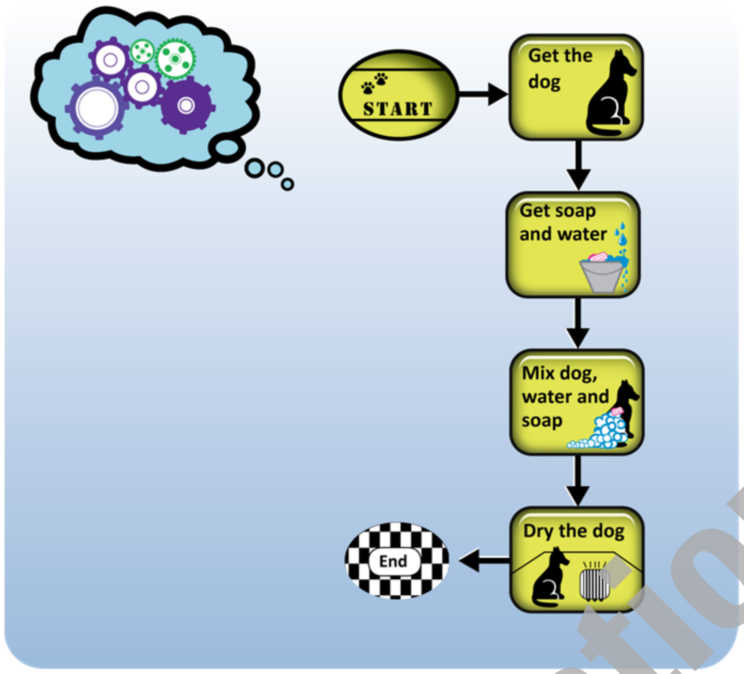
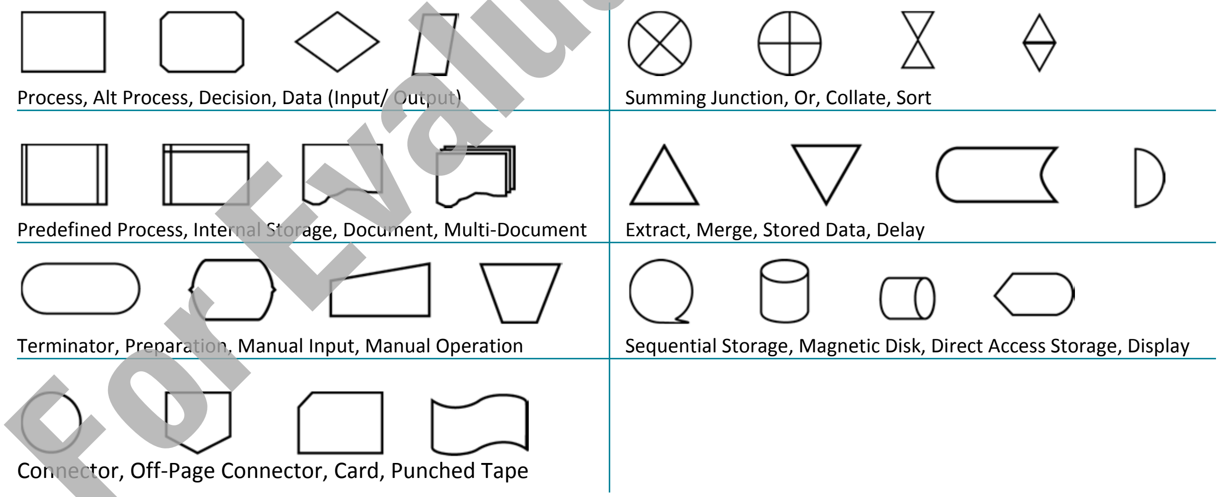


Figure 1-3

Symbols



Process, Alt Process, Decision, Data (Input/ Output)

Summing Junction, Or, Collate, Sort

Predefined Process, Internal Storage, Document, Multi-Document

Extract, Merge, Stored Data, Delay

Terminator, Preparation, Manual Input, Manual Operation

Sequential Storage, Magnetic Disk, Direct Access Storage, Display

Connector, Off-Page Connector, Card, Punched Tape

All processes should flow from top to bottom or left to right. All arrow lines must be straight (curved lines are not allowed in flowcharting).

Typical flowchart symbols include:

- Terminator
  - Ovals, usually containing the word “start” or “end.”
  - May contain another phrase signaling the start or end of a process, such as "submit enquiry" or "receive product.”
- Arrows
  - Direction of travel when the current process is completed.
  - Direction of flow.

- Processing steps
  - Rectangles.
  - Something for the computer to perform, usually a calculation.
- Data (Input/Output)
  - Parallelogram.
  - Example: Get X from the user (Input); display X (Output).
- Decision
  - Represented as a diamond (rhombus).
  - Boolean (Yes/No or True/False test).
  - Two and only two arrows leave this symbol:
    - Always mark these arrows “True” and “False” (or the like).
- A number of other symbols that have less universal currency, such as:
  - A *Document* represented as a rectangle with a wavy base.
  - A *Manual input* represented by parallelogram, with the top irregularly sloping up from left to right. An example would be to signify data entry from a form.
  - A *Manual operation* represented by a trapezoid with the longest parallel side at the top, to represent an operation or adjustment to process that can only be made manually.
  - A *Data File* represented by a cylinder.
  - Connectors
    - Circles.
    - Converging paths or connect to another page (usually not enough room on the current page to display).

Pseudo-Code

Pseudo-code is a way of expressing information from a flowchart in a text. It is an intermediate step before coding. Although it is generally not programming language-specific, it can be written with a “bias” toward a certain language.

For the dog wash example, the pseudo-code could be written:

```
1. Get the dog.
2. Get soap.
3  Get water.
4. Mix dog, water and soap.
5. Dry the dog.
```

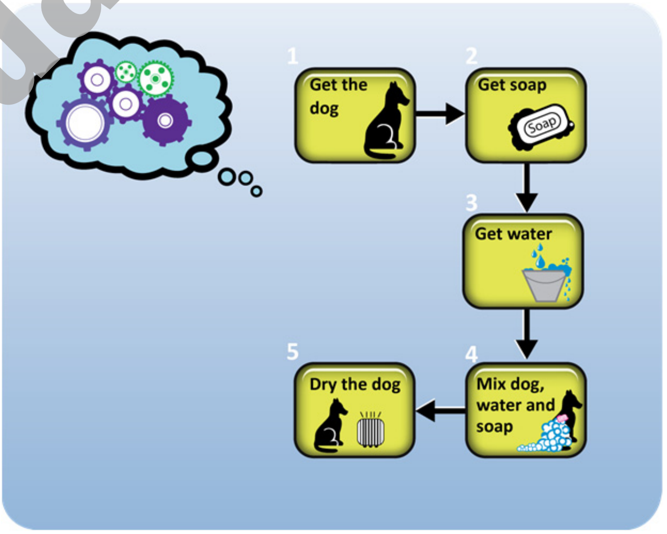


Figure 1-4

Syntax

Syntax is a way of showing the structure of a piece of code without showing a specific example. It lets you see how the code works and the arguments needed to use the code/structure successfully.

The syntax for the sum function in Excel is:

```
SUM(number1,number2, ...)
```

where number1,number2, ... are 1 to 30 arguments for which you want the total value or sum. It is then easy to use the code yourself; for example:

```
=sum(A2,C6)
```

If Decision Structures

The *IF* decision structure is common to many programming languages. It is the basic building block of any decision structure. Additional functionality can be added to the IF statement to make it more powerful; these additions come in the form of the **If-Then Else** and **If-Then ElseIf** structures.

If-Then

The same flowchart from earlier, with an added decision structure, would look like this:

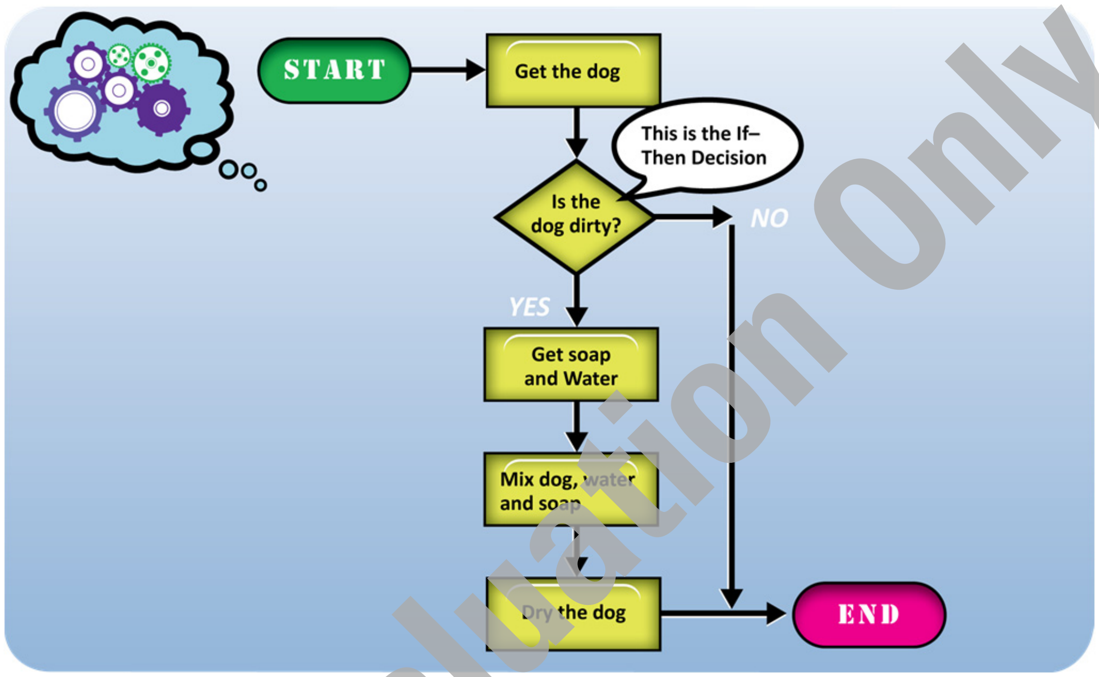


Figure 1-5

Although the syntax varies, the if-then structure is common to many programming languages, including VB, VBA, VB.Net, C++, C#, Java and more.

Syntax:

```
If (Condition A) Then
        (statement block B)
End If
```

"statement block B" is only performed if "condition A" evaluates to "true"; if it is false, go straight to the "End If."

If-Then Else

Used when you want to perform a block of code if the condition is true, and you want to perform a different block of code if the condition is false.

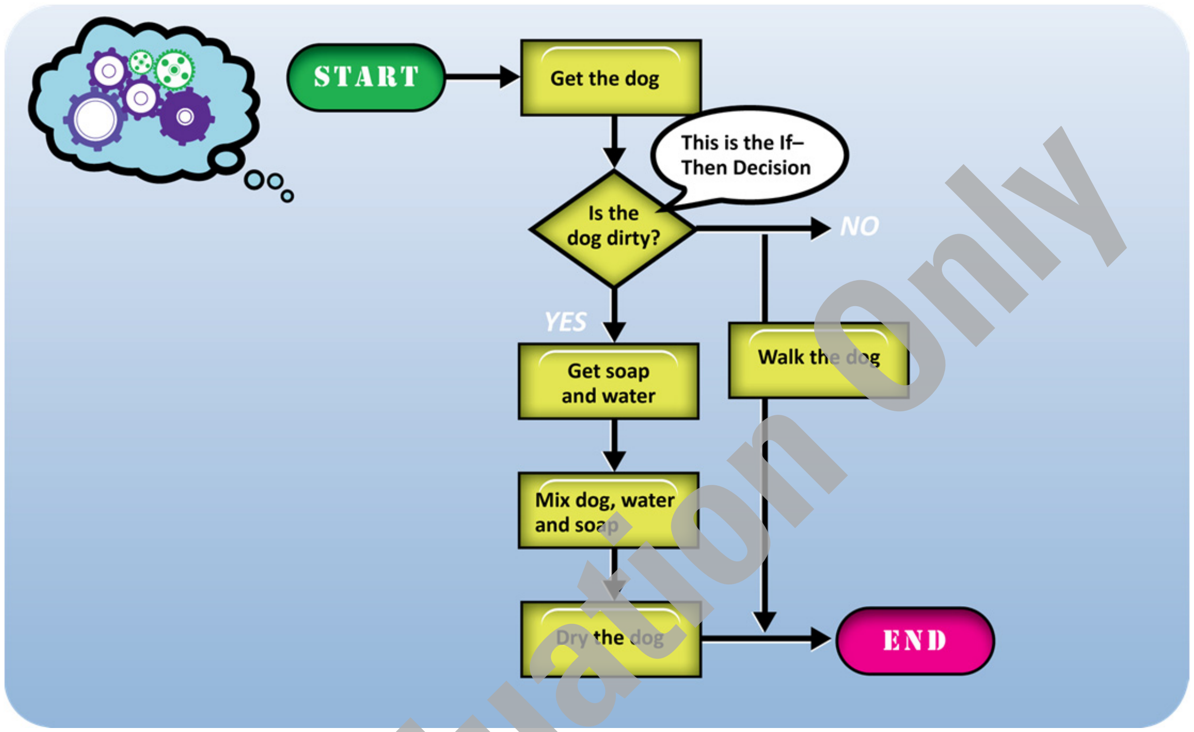


Figure 1-6

```
If (condition C) Then
        (statement block D)
Else
        (statement block E)
End If
```

"statement block D" is only performed if "condition C" evaluates to "true". When completed, go to "End If."

"statement block D" is only performed if "condition C" evaluates to "true". When completed, go to "End If."

If-Then ElseIf

For an extra choice, you can select from among three answers, you can use the If-Then-ElseIf statement:

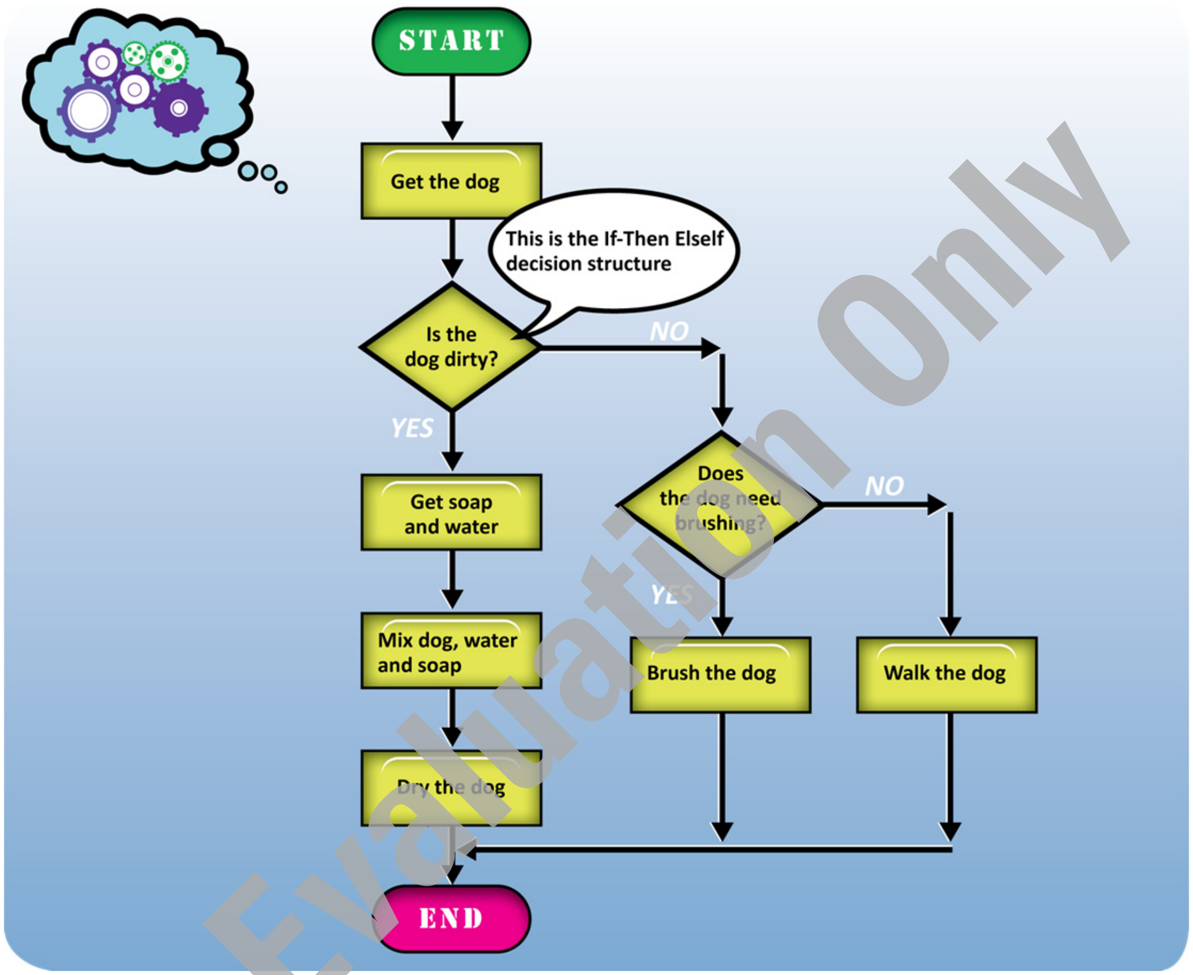


Figure 1-7

```
If number<0 Then
   Print "Your number is negative"
ElseIf number>0 Then
   Print "Your number is positive"
Else
   Print "Your number is zero"
End If
```

Multiple Decision Structures Such As Switch/Select Case

Sometimes a Boolean expression (true/ false) is too limiting. Even the If Then ElseIf statement might not provide enough choice. Perhaps you need to decide among several (or even many) answers. You could perform “nested If” statements, but they require a lot of code and are confusing to read and troubleshoot.

As a flowchart, a "nested If" would look like this:

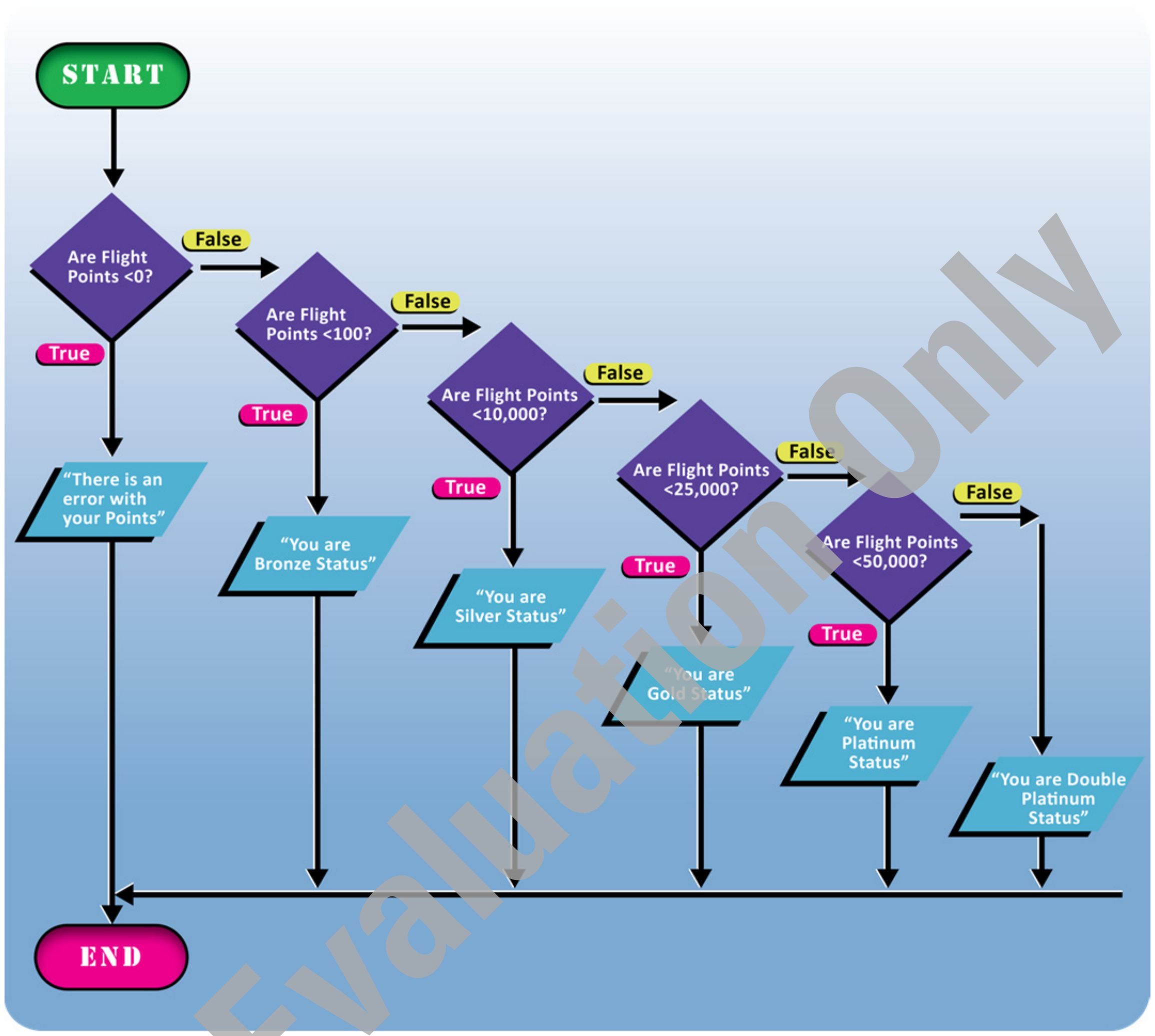

START
Are Flight Points <0?
False
True
"There is an error with your Points"
Are Flight Points <100?
False
True
"You are Bronze Status"
Are Flight Points <10,000?
False
True
"You are Silver Status"
Are Flight Points <25,000?
False
True
"You are Gold Status"
Are Flight Points <50,000?
False
True
"You are Platinum Status"
"You are Double Platinum Status"
END

Figure 1-8

A nested If-Then Else structure in pseudo-code, for the preceding flowchart:

```
If Flight Points < 0 then
     Message "There is an error with your points"
Else if Flight Points < 100 then
     Message "You are Bronze Status"
     Else if Flight Points < 10,000 then
          Message "You are Silver Status"
          Else if Flight Points < 25,000 then
               Message "You are Gold Status"
               Else if Flight Points < 50,000 then
                    Message "You are Platinum Status"
                    Else
                         Message "You are double Platinum Status"
                    End If
               End If
          End If
     End If
End If
```

In most programming, indents of code do not affect the running of the program. Indentation is used to help people read, edit and troubleshoot the code.

Instead, a Select Case (Visual Basic) structure looks like this:

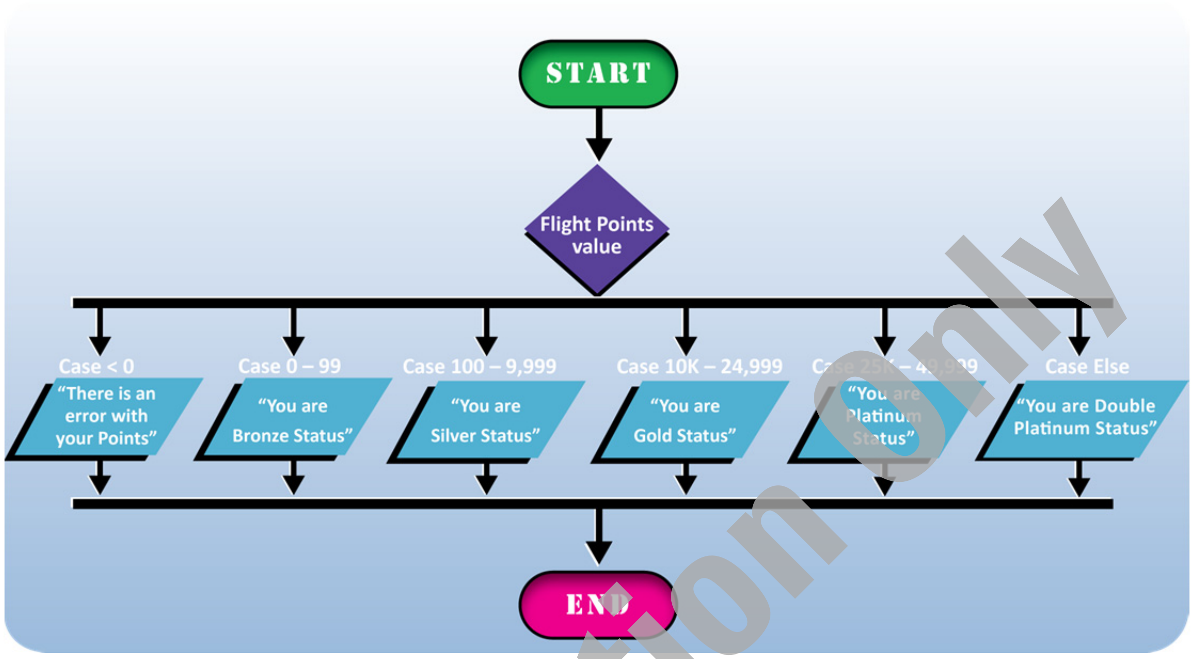


Figure 1-9

Some rules about the Select Case statement are as follows:

- The code executes from the top to the bottom.
- If you match the Case, run the code there, and then go directly to the End Case Statement.
- The Case Else statement will run if no Case is matched.
- There is only one possible answer from this statement.

In Pseudo-Code (close to Visual Basic):

```
Select Case Flight Points
    Case < 0
        "There is an error with your Points"
    Case < 100
        "You are Bronze Status"
    Case < 1,000
        "You are Silver Status"
    Case < 10,000
        "You are Gold Status"
    Case < 25,000
        "You are Gold Status"
    Case < 50,000
        "You are Platinum Status"
    Case Else
        "You are Double Platinum Status"
End Case
```

In Pascal, C, C++, C#, and Java, this same structural element (although with a slightly different syntax) is called a *Switch*.

Decision Tables

Decision tables are a precise and compact way to model complicated logic.

Like if-then-else and switch-case statements, decision tables associate conditions with actions to perform and allow for logic testing. However, decision tables are more compact, and are easier to edit but not as easy to translate into code.

Decision tables are used to show all the possible paths in a simple matrix.

Structure

The syntax is as follows:

Table Name	
Conditions	Condition Answers (T/F)
Actions	Perform Action (Y/N)

Example (Troubleshoot a Local Printer)

		Answers			
Conditions	Is there a warning light on the printer?	N	N	N	Y
	Do you see the printer in your "Printers" folder on your computer?	N	Y	Y	Y
	Are any lights lit on the printer?	Y	N	Y	Y
	Perform Actions				
Actions	Check for printer power cable plugged into wall and printer.		✓		
	Check for printer USB cable plugged into computer and printer.			✓	
	Install/ reinstall the print driver.	✓			
	Check/replace toner.				✓
	Check/ remove paper jam.				✓
	Check/add paper.				✓

The answers to the conditions that occur eventually lead to the decision found in the Actions area. For instance, using the first Answer column, there is no warning light on the printer and there is no Printers folder but there are lights on the printer. With these conditions met, we are led to the decision to install or reinstall the print driver to fix the printer problem.

This is an oversimplified example, with the intent to illustrate the concept of decision tables.

Evaluating Expressions

Evaluation Vs Comparison

In languages such as Visual Basic, there is an inherent ambiguity to the equal "=" operator. The symbol can mean "Evaluate the right side of the equation and assign the value to the left side" such as A = 1 + 3. This meaning is sometimes referred to as the Assignment Operator. On the other hand, the = symbol can mean, "Compare the right side to the left side of the equation and see if it is true" as with 4 = 1 + 3, which would be true.

This ambiguity is solved in languages like C# which uses = only for evaluation, with the == operator for comparison.

Mathematical Formulas (Evaluation)

Computers evaluate mathematical formulas in the proper order (also known as order of operations or precedence).

Automatically evaluate expressions using the following order:

B.E.D.M.A.S.

Brackets "()", then **E**xponents "2" (including roots "√"), then **D**ivision "/" and **M**ultiplication "*", then **A**ddition "+" and **S**ubtraction "-"

Example: 4+2*3

If you (**incorrectly**) evaluated from left to right…

4+2 = 6 then *3 = 18

However, you must (and the computer will automatically) use the correct order of operations, therefore: **M**ultiplication before **A**ddition

2*3=6, then +4 = **10**, which is the **correct** evaluation

Hierarchy	Operator	In written mathematical expression	In common computer syntax
1st (highest)	brackets	(4+2)	(4+2)
2nd	exponents	4^2	4^2
2nd	roots (e.g. cube root)	$\sqrt[3]{4}$	4^(1/3)
3rd	division	4÷2	4/2
3rd	multiplication	4X2	4*2
4th	addition	4+2	4+2
4th	subtraction	4-2	4-2

Comparison Operators

The following table summarizes the comparison operators in Visual Basic:

Operator	Description	Examples
= (equal)	Returns **True** if the number on the left side is equal to the number on the right side.	5 = 4 (false) 4 = 5 (false) 4 = 4 (true)
<> (not equal to)	Returns **True** if the number on the left is not equal to the number on the right.	5 <> 4 (true) 4 <> 5 (true) 4 <> 4 (false)
> (greater than)	Returns **True** if the number on the left is greater than the number on the right.	5 > 4 (true) 4 > 5 (false) 4 > 4 (false)
< (less than)	Returns **True** if the number on the left is less than the number on the right.	5 < 4 (false) 4 < 5 (true) 4 < 4 (false)
>= (greater than or equal to)	Returns **True** if the number on the left is greater than or equal to the number on the right.	5 >= 4 (true) 4 >= 5 (false) 4 >= 4 (true)
<= (less than or equal to)	Returns **True** if the number on the left is less than or equal to the number on the right.	5 <= 4 (false) 4 <= 5 (true) 4 <= 4 (true)

Exercise

In this exercise, you will create charts as well as work with code in Excel.

1. Create a flowchart for getting juice from the refrigerator.
2. Create a flowchart for going to the store in a car.
3. Create a decision structure for obedience training a dog.
4. Create a flowchart to compare expressions.

The following code is entered for you in Excel using Visual Basic for Applications

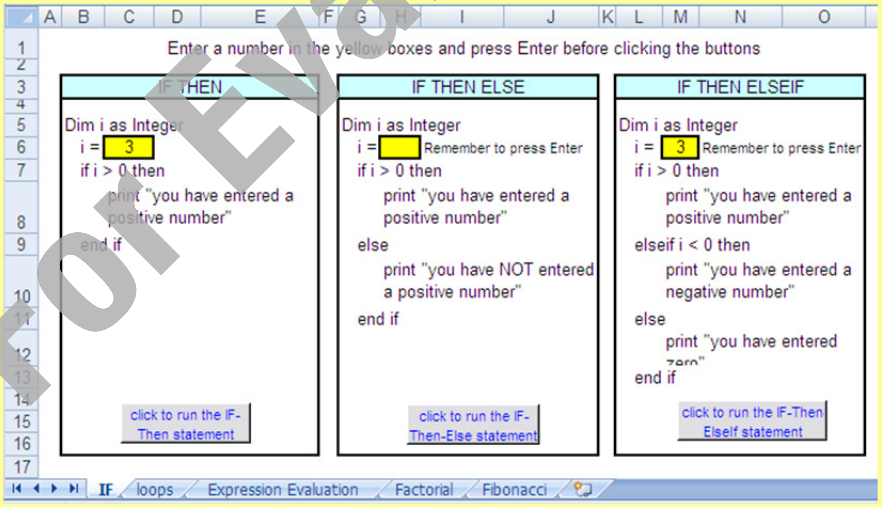
```
Dim A As Double
Dim B As Double

A = CDbl(Range("B4").Value)
B = CDbl(Range("B5").Value)

MsgBox A & " > " & B & " Is " & (A > B)
MsgBox A & " < " & B & " Is " & (A < B)
MsgBox A & " = " & B & " Is " & (A = B)
```

The first two lines declare the variables A and B, which will hold the numeric values used in this program; they use the CDbl statement to convert the text from Excel cell B4 and Excel cell B5 into numeric values. Finally, the last three lines create expressions to compare the two variables using three basic comparison operators, and display the results of those expressions in three message boxes.

5. Open Excel 2007.
6. From the Office Button, click **Excel Options**.
7. Click the **Trust Center** tab to the left and then click **Trust Center Settings**.
8. From the Macro Settings menu, click **Enable all macros (not recommended; potentially dangerous code can run** and **Trust access to the VBA project object model**. Click **OK**.
9. In the **Popular** category, click **Show Developer tab in the Ribbon** and click **OK**.
10. Open the *Lesson 1 Samples.xlsm* file from your student data files.
11. From the first tab called IF, type a number in each of the yellow cells (C6, H6 and M6) and click each button, noticing the result of each:



Enter a number in the yellow boxes and press Enter before clicking the buttons

IF THEN

```
Dim i as Integer
   i = 3
   if i > 0 then
      print "you have entered a
      positive number"
   end if
```

IF THEN ELSE

```
Dim i as Integer
   i =        Remember to press Enter
   if i > 0 then
      print "you have entered a
      positive number"
   else
      print "you have NOT entered
      a positive number"
   end if
```

IF THEN ELSEIF

```
Dim i as Integer
   i = 3  Remember to press Enter
   if i > 0 then
      print "you have entered a
      positive number"
   elseif i < 0 then
      print "you have entered a
      negative number"
   else
      print "you have entered
      zero"
   end if
```

click to run the IF-Then statement
click to run the IF-Then-Else statement
click to run the IF-Then Elself statement
IF
loops
Expression Evaluation
Factorial
Fibonacci

The first message box will display **True** if A (the number you entered in the first text box) is greater than B (the number you entered in the second text box); otherwise it will display **False**. The second message box will display **True** if A is less than B, and the third message box will display **True** if both numbers are the same.

12. Try typing different numbers into the text boxes to see how the results change.
13. Keep *Lesson 1 Samples.xlsm* open.

Identify the Appropriate Method for Handling Repetition

Overview

Left unregulated, a program proceeds through its statements from beginning to end. Simple programs can be written with only this unidirectional flow (as you saw in the Decision Structures section). However, much of the power and utility of any programming language comes from its ability to change execution order with control statements and loops.

Control structures allow you to regulate the flow of your program's execution. Using control structures, you can write Visual Basic code that makes decisions or repeats actions. Other control structures let you guarantee disposal of a resource or run a series of statements on the same object reference.

Loops

Loops are programming structures used to repeat the running of code until a certain condition (or set of conditions) is met.

There are different kinds of loops, whose syntax does vary with the programming language being used. Here, we will focus on Visual Basic because it tends to be the most verbose, which can help new programmers:

- For
- While
- Do… While
- Do… Until
- Repeat Until

For Loops

A *for* loop gets its name from the code that is often used. It enables you to repeat code a specified number of times and is available in most programming languages.

Here is a **for loop** in a partial flowchart:

In Basic (this example will run the "Statements" 10 times; no matter what the values of any variable or user input, the statements will run 10 times):

```
Dim I As Integer
For I = 1 To 10
    [Statements]
Next I
```

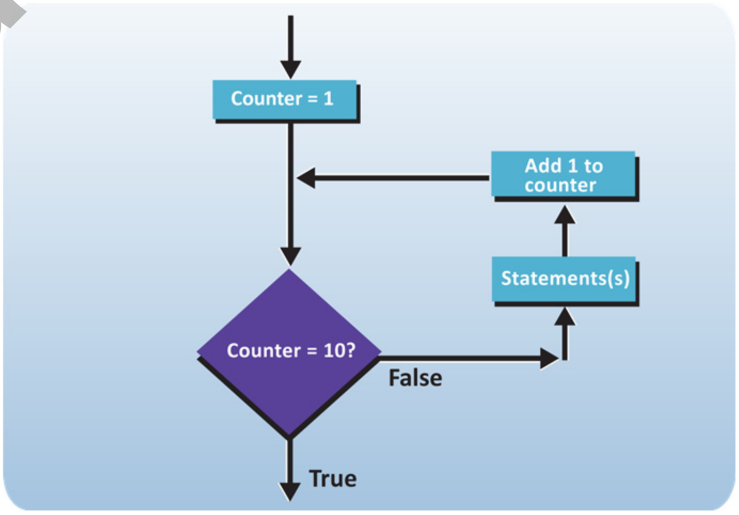
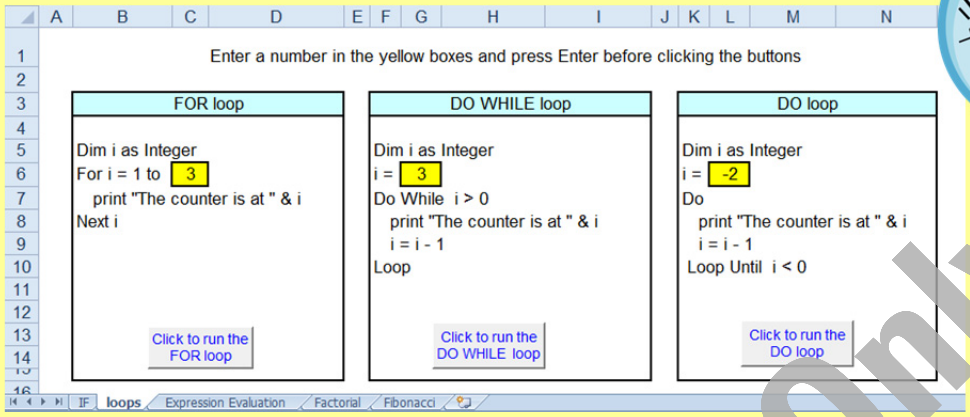


Figure 1-10

1. From the *Lesson 1 Samples.xlsm* file, select the tab called **Loops**.



Enter a number in the yellow boxes and press Enter before clicking the buttons

FOR loop	DO WHILE loop	DO loop
Dim i as Integer	Dim i as Integer	Dim i as Integer
For i = 1 to 3	i = 3	i = -2
print "The counter is at " & i	Do While i > 0	Do
Next i	print "The counter is at " & i	print "The counter is at " & i
	i = i - 1	i = i - 1
	Loop	Loop Until i < 0

2. Change the number in cell **C6** and click **Click to run the FOR loop.**
3. Repeat as necessary.
4. Keep the file open.

While Loops

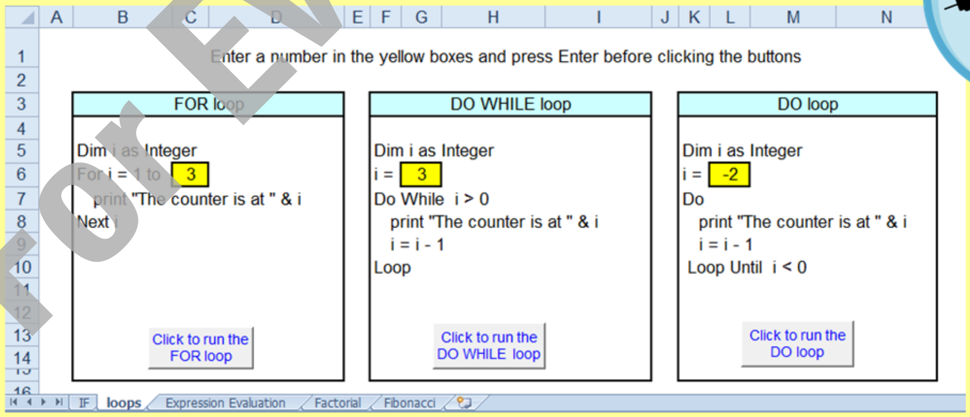
Most programming languages also have constructions for repeating a loop until some condition(s) is/are met.

The terminology may be confusing; this structure is called a While Loop (even though the code uses **Do While**).

Because the code within a While Loop might never be run (if the expression is false to begin with), it is known as a pre-test, where the condition is evaluated before the statements are run.

```
    Do While (Condition)
        [Statements]
Loop
```

1. From the *Lesson 1 Samples.xlsm* file, select the tab called **Loops**.



Enter a number in the yellow boxes and press Enter before clicking the buttons

FOR loop	DO WHILE loop	DO loop
Dim i as Integer	Dim i as Integer	Dim i as Integer
For i = 1 to 3	i = 3	i = -2
print "The counter is at " & i	Do While i > 0	Do
Next i	print "The counter is at " & i	print "The counter is at " & i
	i = i - 1	i = i - 1
	Loop	Loop Until i < 0

2. Change the number in cell **G6** and click **Click to run the DO WHILE loop**.
3. Repeat as necessary.
4. Keep the file open.

Do...While Loops

Unlike the While Loop (whose statements might never run if the condition is false to begin with), the statements inside the "Do… While Loop" must be run at least once. Known as a post-test, where the statements are run once before the condition is evaluated, this structure is often used when you need to get user input for each iteration.

Syntax:

```
Do
     [statements]
While (condition is true)
```

The same concept is used for **Do… Until Loop**.

```
Do
     [statements]
Until (condition is true)
```

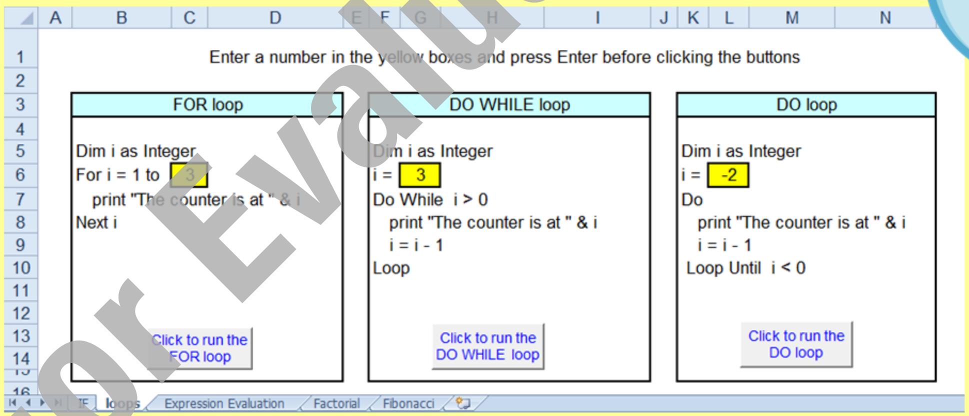
Example in pseudo-code

```
Do
     Get item number from User
     Calculate how much the product costs
     Total = Total + new cost
Until (user enters "done")
Print "You owe" & Total
```

Some languages may use a different syntax for this type of loop; for example, Pascal uses a **repeat until** loop.

Exercise

1. From the *Lesson 1 Samples.xlsm* file, select the tab called **Loops**.



Enter a number in the yellow boxes and press Enter before clicking the buttons

FOR loop

```
Dim i as Integer
For i = 1 to  3
   print "The counter is at " & i
Next i
```

Click to run the FOR loop

DO WHILE loop

```
Dim i as Integer
i =  3
Do While  i > 0
   print "The counter is at " & i
   i = i - 1
Loop
```

Click to run the DO WHILE loop

DO loop

```
Dim i as Integer
i =  -2
Do
   print "The counter is at " & i
   i = i - 1
 Loop Until  i < 0
```

Click to run the DO loop

2. Change the number in cell **L6** and click **Click to run the DO loop**.
3. Repeat as necessary.
4. Keep the file open.

Recursion (Collection-Controlled Loops) Versus Iterative

The two methods of performing loops are iteration and recursion. In simple terms, iteration means performing a test on an item to see if certain conditions are met, and if they are not, moving to the next item disregarding the one you just examined. Recursion, on the other hand, keeps track of the previous element(s). In fact, it uses the previous values(s) to create the new value. The advantage to iteration is that it rarely causes an overflow error because it only tracks one thing at a time. The disadvantage to iteration is that it does not remember the previous values, so evaluation must begin all over every time. Conversely, recursion keeps track of all of the values it has dealt with, which can cause overflow errors. However, recursion can also solve more complex problems that require information from the previous data to be used by the current data.

In each recursion, there needs to be a "base case," which is the starting point.

Two common examples often used to show recursion are factorials and the Fibonacci Sequence.

Factorials

In simple mathematical terms, a factorial is a positive number multiplied by all the numbers between itself and the digit one.

For example, 6 factorial = 6! = 6*5*4*3*2*1 = 720 (zero is not used in the factorial series).

The following paragraph is a function, defined in words, that calculates a factorial.

- If the number is less than zero, reject it.
- If it is not an integer, reject it.
- If the number is zero, its factorial is one.
- If the number is larger than zero, multiply it by the factorial of the next smaller number

To calculate the factorial of a positive whole number (that is not zero) – call it n. You can calculate the factorial of n-1, then multiply it by n. This leads to an internal loop where in order to calculate the number, the function must call itself for the next smaller number before it can execute on the current number. This is an example of recursion. In math, this appears as: 6! = 6 * 5!

Recursion and iteration (looping) are closely related — a function can return the same results with either recursion or iteration. Certain computations will lend themselves better to one technique or the other; you choose the most natural or logical approach.

You must be careful because even with the usefulness of recursion, you can easily create a recursive function that never returns a result and cannot reach an endpoint. Such a recursion (and iteration looping as well) causes the computer to execute an *infinite* loop. For instance, if you were to allow negative numbers, you would get 6! = 6*5!=… then 4! 3! 2! 1! 0! -1! -2! …and you would never reach the end. It would calculate for infinity.

Computer resources are another concern when you deal with recursion. Because you are calling the same function from within the function itself, the computer has to track all the numbers as you go. It effectively has to remember all phases of the recursion at once. This restriction can cause stack overflows or system memory overflows.

Therefore, you must be very careful in creating recursive functions. It is a good idea to follow any code by hand to make certain you call the recursive function excessively (or infinitely). One method you can employ is to set up a counter to count the number of times the functions calls itself, so you can limit the calls to a certain predefined count, then you can stop it from running if it passes the threshold. There is no absolute maximum number of internal function calls; it depends on the function itself.

Example: Recursive Factorial Function (n! = n * (n-1)!)

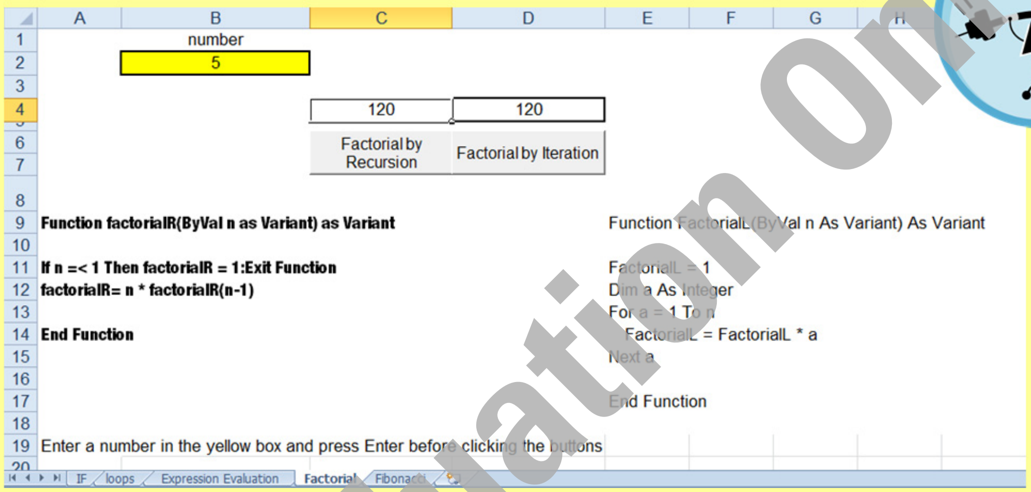
```
Function factorial(ByVal n As Integer) As Integer
    If n =< 1 Then
        factorial = 1:
        Exit Function
    End If
    factorial = n * factorial(n-1)
End Function
```

Example: Iteration Factorial Function n! = (n-1) * (n-2) * (n-3)… (1)

```
Function factorial (ByVal n As Integer) As Integer
     factorial = 1
     Dim a as Integer
     For a = 1 To n
     factorial = factorial * a
     Next 'a
End Function
```

Exercise

1. From the *Lesson 1 Samples.xlsm* file, select the tab called **Factorial**.



```
number
5
120    120
Factorial by Recursion    Factorial by Iteration

Function factorialR(ByVal n as Variant) as Variant

If n =< 1 Then factorialR = 1:Exit Function
factorialR= n * factorialR(n-1)

End Function

Function FactorialL(ByVal n As Variant) As Variant

FactorialL = 1
Dim a As Integer
For a = 1 To n
  FactorialL = FactorialL * a
Next a

End Function

Enter a number in the yellow box and press Enter before clicking the buttons
```

2. Change the number in cell **B2** and click **Factorial by Recursion**.
3. Repeat as necessary.
4. Change the number in cell **B2** and click **Factorial by Iteration**.
5. Repeat as necessary.
6. Keep the file open.

Fibonacci numbers:

In mathematics, the Fibonacci numbers are the numbers in the following sequence:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

The first two Fibonacci numbers are 0 and 1 (although some people skip the 0, so the first two numbers are 1 and 1). Each subsequent number is the sum of the previous two.

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2},$$

With seed (base) values

$$F_0 = 0 \quad \text{and} \quad F_1 = 1.$$

The Fibonacci sequence is named after Leonardo of Pisa, who was known as Fibonacci (a contraction of filius Bonaccio, "son of Bonaccio"). Fibonacci's 1202 book *Liber Abaci* introduced the sequence to Western European mathematics. The sequence was reported earlier in Indian mathematics. This sequence produces complex and interesting visual effects, as shown in the figure below.

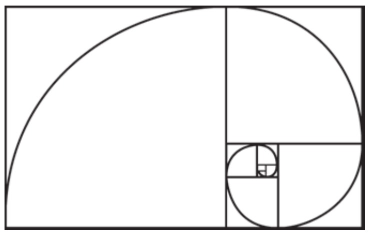


Figure 1-11

A Fibonacci spiral is created by drawing arcs connecting the opposite corners of squares in the Fibonacci tiling; this one uses squares of sizes 1, 1, 2, 3, 5, 8, 13, 21, and 34;

1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
8 + 5 = 13
13 + 8 = 21
21 + 13 = 34 and so on.

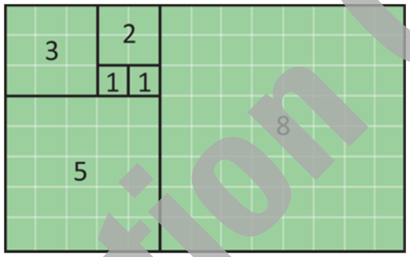
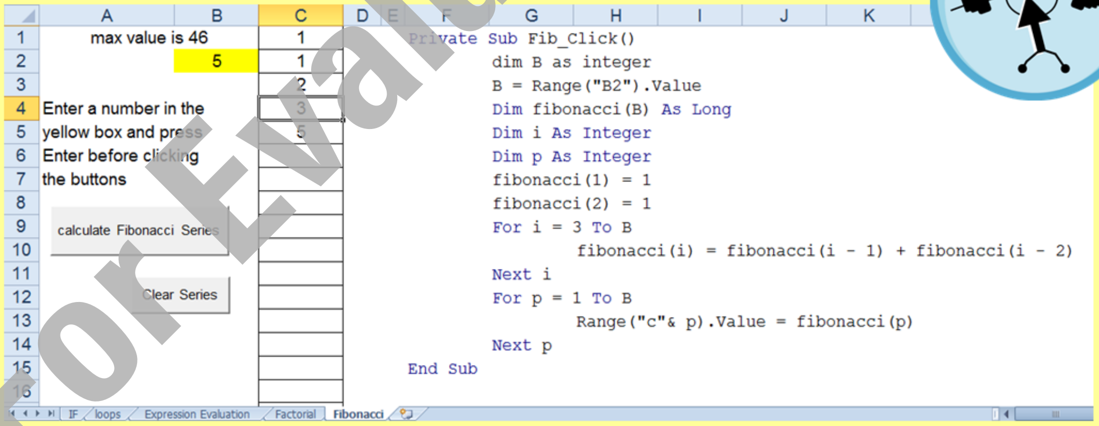


Figure 1-12

The preceding Fibonacci area diagram shows a way to see graphically the effect of the sequence.

Exercise

1. From the *Lesson 1 Samples.xlsm* file, select the tab called **Fibonacci**.



	A	B	C
1	max value is 46		1
2		5	1
3			2
4	Enter a number in the		3
5	yellow box and press		5
6	Enter before clicking		
7	the buttons		

```
Private Sub Fib_Click()
    dim B as integer
    B = Range("B2").Value
    Dim fibonacci(B) As Long
    Dim i As Integer
    Dim p As Integer
    fibonacci(1) = 1
    fibonacci(2) = 1
    For i = 3 To B
        fibonacci(i) = fibonacci(i - 1) + fibonacci(i - 2)
    Next i
    For p = 1 To B
        Range("c"& p).Value = fibonacci(p)
    Next p
End Sub
```

2. Change the number in cell B2 and click **calculate Fibonacci Series**.
3. Click Clear Series.
4. Repeat as necessary.
5. Change the number in cell **B2** and click **Factorial by Iteration**.
6. Repeat as necessary.
7. Keep the file open.

Understand Error (Exception) Handling

In a perfect world, no program would ever contain any errors in code, and every possible input from a user would be resolved and not crash the program. However, in practical terms, you need to know not only what errors can occur, but also how to handle them so they do not crash your programs. By definition, an exception can be thought of as a special condition that changes the normal flow of program execution. The word “special” is ambiguous, but can be thought of as something that does not help the program run. Frequently used terms are “throwing an exception” or “an exception throw.”

Types of Errors

Basically, errors can be broken down into three types:

- Syntax errors
- Runtime errors
- Logic errors

Syntax Errors

These errors occur when a mistake occurs in the spelling or punctuation of code. They are often discovered by the software almost immediately (often as soon as the programmer leaves the line he or she was working on).

Example:

```
A = "Dogs
```

- Here, the computer will detect an error almost as soon as it is typed.
- There is a missing closing quotation mark ” after Dogs.

Syntax errors are the most common type of errors. You can fix them easily in the coding environment as soon as they occur.

Runtime Errors

Runtime errors are usually only found when the code is executed during its running (hence the name runtime).

Runtime errors can be reduced, but not eliminated completely if the program is *compiled* (the computer can do a thorough check and “dry-run” of your program to see if any runtime errors will occur).

For example, you might correctly write a line of code to open a file. However, if the file is corrupted or already open by another process, the application cannot carry out the **Open** function, and it stops running. You can fix most run-time errors by rewriting the faulty code, then recompiling and rerunning it.

Logic Errors

Logic errors are those that appear after the application is in use. They should be caught at design time if the coder is careful, but there are usually so many lines of code that to expect perfection is unrealistic. Logic errors are generally the hardest type to fix, because it is often difficult to find where they originate.

Example:

```
A = CountOfDogs
B = CountOfCats
Print "The count of Cats is " & A
```

- Here, there will be no notification from the computer that there is an error; it “thinks” everything is fine.
- However, the count of cats is stored in “B”, not “A.”

Exception Handling

Visual Basic supports both structured and unstructured exception (error) handling. By inserting exception handling code in your application, you can handle most of the errors users may encounter and enable the application to continue running properly. You can use structured and unstructured error handling to plan for possible errors, thereby preventing them from crashing your application or producing inaccurate results.

You should insert exception handling code in any method that uses operators that may generate an exception, or that calls into or accesses other procedures that may generate an exception. Often, programmers will code functions to call other functions. If you do not code for error handling, you will often not know in which function the error occurred.

Structured Exception Handling

In structured exception handling, blocks of code are separated from one another, with each block having at least one associated handler. Each handler is specific to the type of exception it handles. When an exception is thrown (occurs), the corresponding handler's code is executed. A single method can have multiple structured exception handling blocks, and the blocks can be nested within each other.

Unstructured Exception Handling

The **On Error** statement is used for unstructured exception handling. **On Error** is placed at the beginning of a block of code. It handles any errors occurring within that block. If you use more than one **On Error** statement, the most recent statement takes precedence.

Choosing When to Use Structured and Unstructured Exception Handling

Structured exception handling is the use of a control structure containing exceptions and filters to create a mechanism to handle exceptions. This feature allows your code to distinguish between different types of errors and react appropriately. In unstructured exception handling, an **On Error** statement handles all exceptions.

Structured exception handling is significantly more robust and flexible than unstructured exception handling. When possible, use structured exception handling; however, you might use unstructured exception handling under the following circumstances:

- You are upgrading an application written in an earlier version of Visual Basic (where structured error handling did not exist).
- You are developing a beta version of an application and you are not concerned if the program fails to shut down gracefully.
- You know in advance exactly what will cause the exception (so the On Error can be coded specifically for that exception).
- A deadline is pressing, and you are willing to sacrifice flexibility for speed.
- Code is short enough that you only need to test the branch of code generating the exception.
- You need to use the **Resume Next** statement, which is not supported in structured exception handling. This statement simply states, "if you reach an error, go to the next line without crashing."

Apart from what you choose to handle exceptions within your code, always take an objective look at your (and the code's) assumptions. For example, when your application asks the user to input a telephone number, you must be aware of the following assumptions:

- The user will input numbers rather than letters.
- The number will have a certain format (xxx-xxx-xxxx).
- The user will not input a null string.
- The user has a single telephone number.

User input might breach any of these assumptions. Robust code requires adequate exception handling, which allows your application to recover from such a breach.

You should use informative exception handling. Beyond stating "an error has occurred," messages resulting from exception handling should indicate exactly why and where things went wrong.

The Try...Catch...Finally Block

In newer versions of Visual Basic (not Excel), the **Try...Catch...Finally** is an example of structured exception handling. You can **Try** a segment of code; if an exception is thrown, it jumps to, and runs the code in the **Catch** block. After that code has finished, any code in the **Finally** block is run. The **Try** statement represents the beginning of the block, while the **End Try** statement represents the end of the block.

Example Syntax

```
Try
 ' Code here attempts to do something.
Catch
 ' If an error occurs, code here will run.
Finally
 ' Code in this block will always run.
End Try
```

First, the code in the **Try** block is executed. If it runs without an exception, the program skips the **Catch** block and runs the code in the **Finally** block. If the code in the **Try** block does throw an exception, the code in the **Catch** block is run immediately; then the code in the **Finally** block is run.

Lesson Summary

You are now able to:

☑ Explain how computers store programs and data in memory.

☑ Demonstrate computer decision structures, including flowcharts and pseudo-code.

☑ Identify and explain the best ways to handle repetition.

MMM
Go online for Additional Review and Case Scenarios

Review Questions

1. What does RAM stand for?
   a. Read All Memory
   b. Random Access Memory
   c. Ready All Memory
   d. Refresh Access to Memory

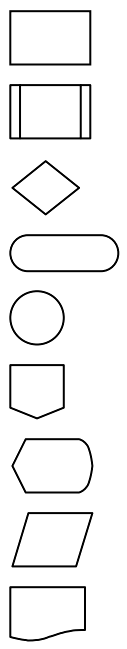
2. Volatile means:
   a. Electricity must be flowing for retention
   b. Some memory can "steal" data from other memory
   c. Memory can lose data easily
   d. The memory vapors must not be breathed

3. Unlike RAM, BIOS is:
   a. Inexpensive
   b. Not available at startup
   c. Not on the motherboard
   d. Non-volatile

4. A memory stack:
   a. Is an example of first in, first out storage
   b. Holds memory addresses
   c. Has its size controlled by the application
   d. Is stored in the BIOS

5. A String data type can hold a number:
   a. True
   b. False

6. Identify the name of each flowchart symbol:



Off-page connector ____________________

Data ____________________

Process ____________________

Decision ____________________

Display ____________________

Document ____________________

Pre-defined Process ____________________

Connector ____________________

Terminator ____________________

7. Syntax is:
   a. The structure to format code
   b. A way to understand programming using real-world examples
   c. A generic representation of a piece of code or function
   d. Not used in modern programming

8. What is correct about loops?
   a. There is only one correct way to code them
   b. They are efficient at reusing code
   c. They will run the same way regardless of user input
   d. You do not need to test them because they are easy to write

9. List the three types of errors:
   a. ____________________
   b. ____________________
   c. ____________________

10. Which is an example of Structured Error Handling?
   a. IF
   b. Loop
   c. OnError
   d. Try...Catch...Finally