Chapter 2

Designing a Program

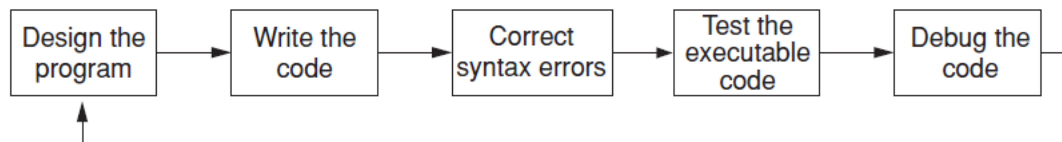
When programmers begin a new project, they never jump right in and start writing code as the first step. They begin by creating a design of the program. After designing the program, the programmer begins writing code in a high-level language.

A language's syntax rules dictate things such as how key words, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.

If the program contains a syntax error, or even a simple mistake such as a misspelled key word, the compiler or interpreter will display an error message indicating what the error is.

Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A ***logic error*** is a mistake that does not prevent the program from running, but causes it to produce incorrect results.

If there are logic errors, the programmer *debugs* the code. This means that the programmer finds and corrects the code that is causing the error. Sometimes during this process, the programmer discovers that the original design must be changed. This entire process, which is known as the *program development cycle,* is repeated until no errors can be found in the program.



The process of designing a program:

- Understand the task that the program is to perform.
The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements.
A *software requirement* is simply a single function that the program must perform to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.

- Determine the steps that must be taken to perform the task.
Once you understand the task that the program will perform, you begin by breaking down the task into a series of logical steps (***Algorithm***) using ***Pseudo-code*** or ***Flowchart***.

Pseudo-code:

Pseudo-code (fake code) is an informal language that has no syntax rules and is not meant to be compiled or executed.

Example: Pay Calculating Program

Display "Enter the number of hours the employee worked."
Input hours
Display "Enter the employee's hourly pay rate."
Input payRate
Set grossPay = hours * payRate
Display "The employee's gross pay is $", grossPay

NOTE: Each statement in the pseudo-code represents an operation that can be performed in any high-level language.

Flowcharts:

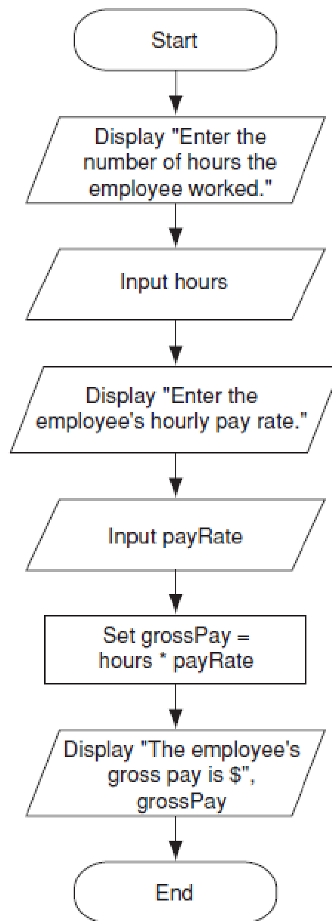
A *flowchart* is a diagram that graphically depicts the steps that take place in a program.

There are three types of symbols in the flowchart:
The ***ovals*** (terminal symbols); the *Start* terminal symbol marks the program's starting point and the *End* terminal symbol marks the program's ending point.

The ***parallelograms***; are used for both *input symbols* and *output symbols*.

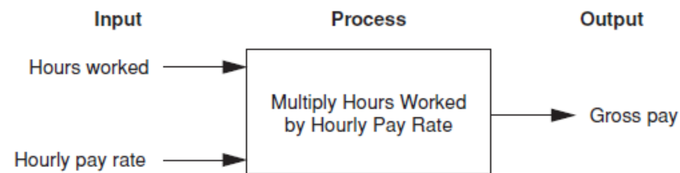
The ***rectangles***; are used as *processing symbols*.

NOTE: Each of these symbols represents a step in the program. The symbols are connected by arrows that represent the "flow" of the program.



Output, Input, and Variables

Computer programs typically perform the following three-step process:
- Input is received
- Some process is performed on the input
- Output is produced



Displaying Screen Output:

Display "Kate Austen", "1234 Walnut Street", "Asheville, NC 28899"
Display "Kate Austen"
Display "1234 Walnut Street"
Display "Asheville, NC 28899"
Flowcharts?!!!

NOTE: In programming terms, a sequence of characters that is used as data is called a *string*.

Display "January", "February", "March"
Display "January ", "February ", "March"

Variables and Input:

Programs use variables to store data in memory. A ***variable*** is a storage location in memory that is represented by a name.

Most programming languages require that you *declare* all the variables that you intend to use in a program. A *variable declaration* is a statement that typically specifies two things about a variable:
- The variable's name
- The variable's data type

Data Types:

- A variable of the Integer data type can hold whole numbers.
- A variable of the Real data type can hold either whole numbers or numbers with a fractional part.
- A variable of the String data type can hold any string of characters.

NOTE: In addition to a String data type, many programming languages also provide a Character data type. The difference between a String variable and a Character variable is that a String variable can hold a sequence of characters of virtually any length, and a Character variable can hold only one character.

Declare Integer length
Declare Real grossPay
Declare String name
Declare Integer length, width, height

Example:

```
Declare Integer age
Display "What is your age?"
Input age
Display "Here is the value that you entered:"
Display age
```

Example:

```
Declare Real test1
Declare Real test2
Declare Real test3
Declare Real average

Set test1 = 88.0
Set test2 = 92.5
Set test3 = 97.0
Set average = (test1 + test2 + test3) / 3

Display "Your average test score is ", average
```

Variable Initialization:

```
Declare Real price = 49.95
Declare Integer length = 2, width = 4, height = 8
```

What about?!!!

```
Declare Integer i
Set i = 3.7

Declare Integer i
Set i = 3.0

Declare Real dollars
Set 99.95 = dollars
```

Performing Calculations:

Symbol	Operator	Description
+	Addition	Adds two numbers
-	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the quotient
MOD	Modulus	Divides one number by another and gives the remainder
^	Exponent	Raises a number to a power

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	`6 * B`
$(3)(12)$	3 times 12	`3 * 12`
$4xy$	4 times x times y	`4 * x * y`

Algebraic Expression	Pseudocode Statement
$y = 3\frac{x}{2}$	`Set y = x / 2 * 3`
$z = 3bc + 4$	`Set z = 3 * b * c + 4`
$a = \frac{x+2}{a-1}$	`Set a = (x + 2) / (a - 1)`

Be careful: Integer Division

In Java, C++, C, and Python, the / operator throws away the fractional part of the result when both operands are integers. In these languages the result of the expression 3/2 would be 1. In Visual Basic, the / operator does not throw away the fractional part of the answer. In Visual Basic, the result of the expression 3/2 would be 1.5.

Named Constants:

A named constant is a name that represents a value that cannot be changed during the program's execution.
Constant Real INTEREST_RATE = 0.069

Block Comments and Line Comments

Block comments take up several lines. They often appear at the beginning of a program, explaining what the program does, listing the name of the author, giving the date that the program was last modified, and any other necessary information.

```
// This program calculates an employee's gross pay.
// Written by Matt Hoyle.
// Last modified on 12/14/2010
```

Line comments are comments that occupy a single line and explain a short section of the program.

```
// Calculate the interest.
Set interest = balance * INTEREST_RATE
```

```
Input age // Get the user's age.
```

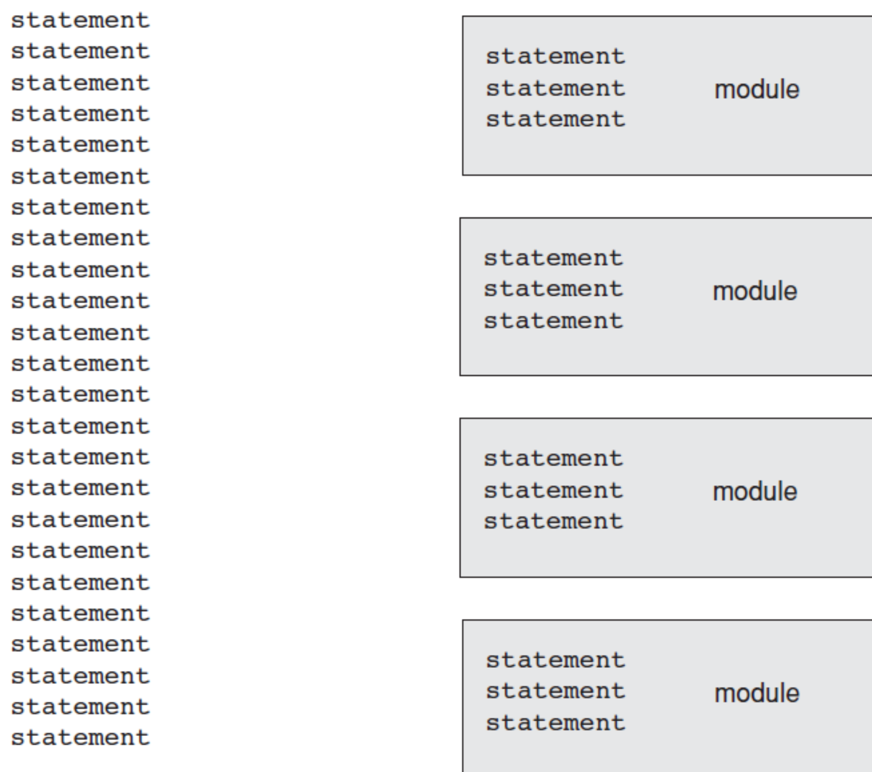
Chapter 3

Remember: a program is a set of instructions that a computer follows to perform a task.

Most programs perform tasks that are large enough to be broken down into several subtasks. As a result, programmers usually break down their programs into modules.

A ***module*** (procedure, subroutine, subprogram, method, function*)* is a group of statements that exist within a program for the purpose of performing a specific task.

So, instead of writing a large program as one long sequence of statements, it can be written as several small modules, each one performing a specific part of the task. These small modules can then be executed in the desired order to perform the overall task.



Benefits of Using Modules

Simpler Code
Several small modules are much easier to read than one long sequence of statements.

Code Reuse
If a specific operation is performed in several places in a program, a module can be written once to perform that operation, and then be executed any time it is needed.

Better Testing
Programmers can test each module in a program individually, to determine whether it correctly performs its operation.

Faster Development
Modules can be written for the commonly needed tasks, and those modules can be incorporated into each program that needs them.

Easier Facilitation of Teamwork
When a program is developed as a set of modules that each performs an individual task, then different programmers can be assigned the job of writing different modules.

Defining & Calling a Module

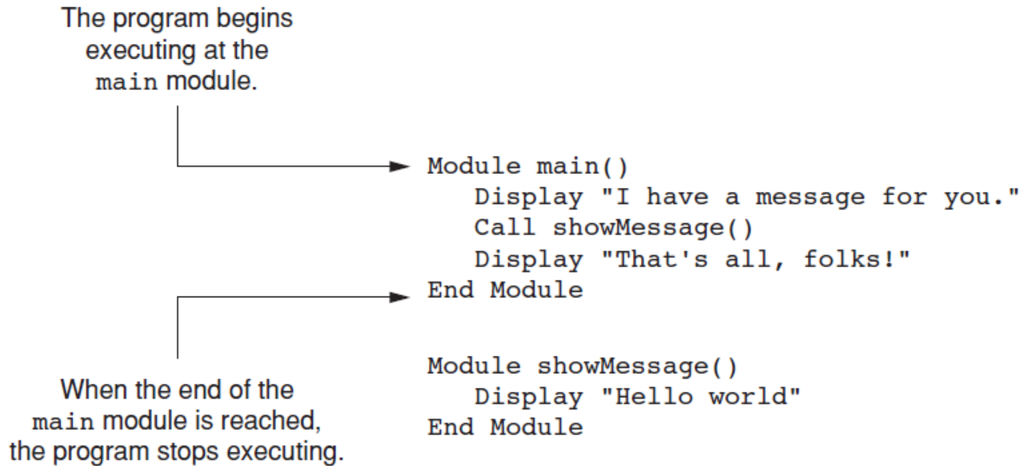
A module's name should be descriptive enough so that anyone reading your code can reasonably guess what the module does.

Module names cannot contain spaces, cannot typically contain punctuation characters, and usually cannot begin with a number.

A module definition has two parts: a header and a body. The *header* indicates the starting point of the module, and the *body* is a list of statements that belong to the module.

```
Module name()
   statement
   statement      } These statements are the body of the module.
   etc.
End Module
```

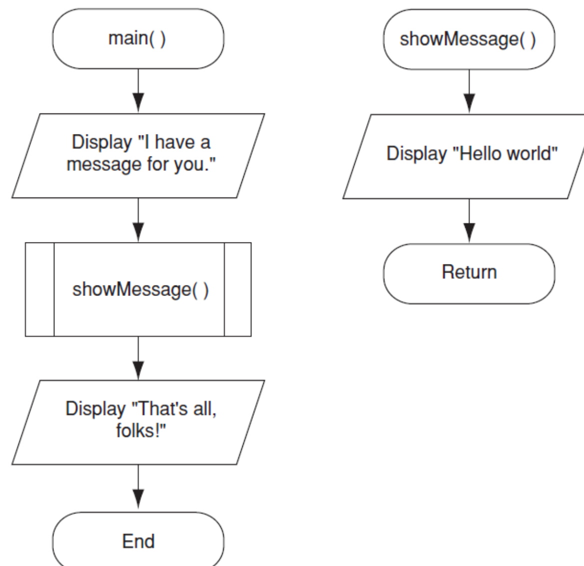
When a module is called, the computer jumps to that module and executes the statements in the module's body. Then, when the end of the module is reached, the computer jumps back to the part of the program that called the module, and the program resumes execution at that point.



NOTE: The main module is the program's starting point, and it generally calls other modules. When the end of the main module is reached, the program stops executing.

Flowcharting a Program with Modules

In a flowchart, a module call is shown with a rectangle that has vertical bars at each side. The name of the module that is being called is written on the symbol.

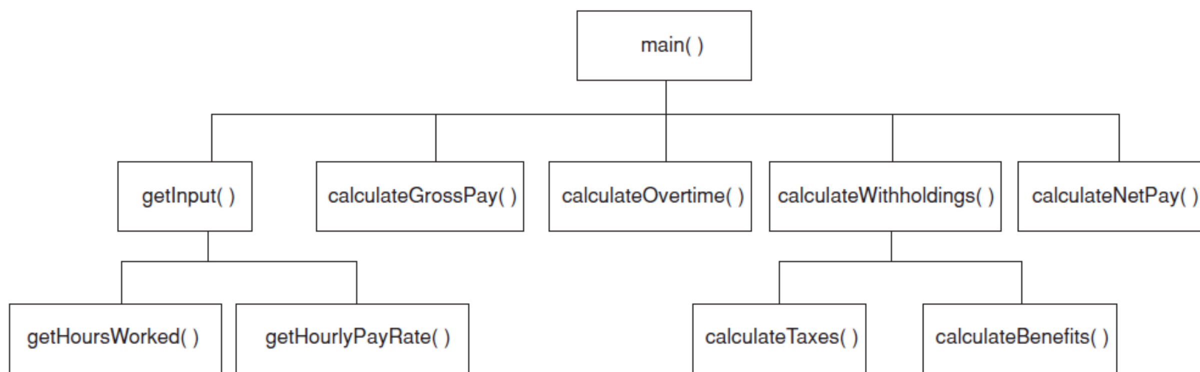


Programmers commonly use a technique known as *top-down design* to break down an algorithm into modules.

The process of top-down:
- The overall task that the program is to perform is broken down into a series of subtasks.
- Each of the subtasks is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified.
- Once all the subtasks have been identified, they are written in code.

Hierarchy Charts

A hierarchy chart (*structure chart*) shows boxes that represent each module in a program. The boxes are connected in a way that illustrates their relationship to one another.



Local Variables

In most programming languages, a variable that is declared inside a module is called a *local variable*. A local variable belongs to the module in which it is declared, and only statements inside that module can access the variable.

```
1  Module main()
2     Call getName()
3     Display "Hello ", name        <-- This will cause an error!
4  End Module
5
6  Module getName()
7     Declare String name           <-- This is local variable.
8     Display "Enter your name."
9     Input name
10 End Module
```

```
Module getTwoAges()
   Declare Integer age
   Display "Enter your age."
   Input age

   Declare Integer age          <-- This will cause an error!
   Display "Enter your pet's age."   A variable named age has
   Input age                         already been declared.
End Module
```

Passing Arguments to Modules

Sometimes it is useful to call a module and send one or more pieces of data into the module. Pieces of data that are sent into a module are known as *arguments*. The module can use its arguments in calculations or other operations.

```
1 Module main()
2    Call doubleNumber(4)
3 End Module
4
5 Module doubleNumber(Integer value)
6    Declare Integer result
7    Set result = value * 2
8    Display result
9 End Module
```

```
Module main()
   Declare Integer number
   Display "Enter a number and I will display"
   Display "that number doubled."
   Input number
   Call doubleNumber(number)
End Module

Module doubleNumber(Integer value)
   Declare Integer result
   Set result = value * 2
   Display result
End Module
```

20

The contents of the number variable are copied into the value parameter variable.

Passing Multiple Arguments

```
 1 Module main()
 2    Display "The sum of 12 and 45 is:"
 3    Call showSum(12, 45)
 4 End Module
 5
 6 Module showSum(Integer num1, Integer num2)
 7    Declare Integer result
 8    Set result = num1 + num2
 9    Display result
10 End Module
```

```
 1 Module main()
 2    Declare Integer number = 99
 3
 4    // Display the value stored in number.
 5    Display "The number is ", number
 6
 7    // Call the changeMe module, passing
 8    // the number variable as an argument.
 9    Call changeMe(number)
10
11    // Display the value of number again.
12    Display "The number is", number
13 End Module
14
15 Module changeMe(Integer myValue)
16    Display "I am changing the value."
17
18    // Set the myValue parameter variable
19    // to 0.
20    Set myValue = 0
21
22    // Display the value in myValue.
23    Display "Now the number is ", myValue
24 End Module
```

```
 1 Module main()
 2    // Declare and initialize some variables.
 3    Declare Integer x = 99
 4    Declare Integer y = 100
 5    Declare Integer z = 101
 6
 7    // Display the values in those variables.
 8    Display "x is set to ", x
 9    Display "y is set to ", y
10    Display "z is set to ", z
11
12    // Pass each variable to setToZero.
13    Call setToZero(x)
14    Call setToZero(y)
15    Call setToZero(z)
16
17    // Display the values now.
18    Display "----------------"
19    Display "x is set to ", x
20    Display "y is set to ", y
21    Display "z is set to ", z
22 End Module
23
24 Module setToZero(Integer Ref value)
25    Set value = 0
26 End Module
```

Global Variables & Constants

A *global variable* is a variable that is visible to every module in the program. In most programming languages, you create a global variable by at the top of the program.

```
1 // The following declares a global Integer variable.
2 Declare Integer number
3
4 // The main module
5 Module main()
6    // Get a number from the user and store it
7    // in the global variable number.
8    Display "Enter a number."
9    Input number
10
11   // Call the showNumber module.
12   Call showNumber()
13 End Module
14
15 // The showNumber module displays the contents
16 // of the global variable number.
17 Module showNumber()
18   Display "The number you entered is ", number
19 End Module
```

A *global constant* is a named constant that is available to every module in the program.

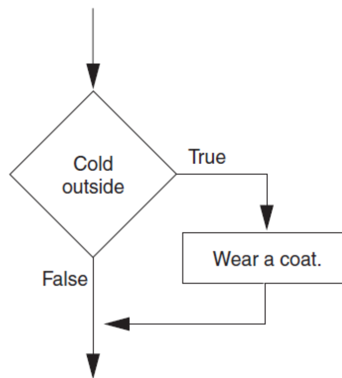
Most programmers agree that you should restrict the use of global variables, or not use them at all.
- Global variables make debugging difficult.
- Modules that use global variables are usually dependent on those variables. If you want to use such a module in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- Global variables make a program hard to understand. A global variable can be modified by any statement in the program.

Chapter 4

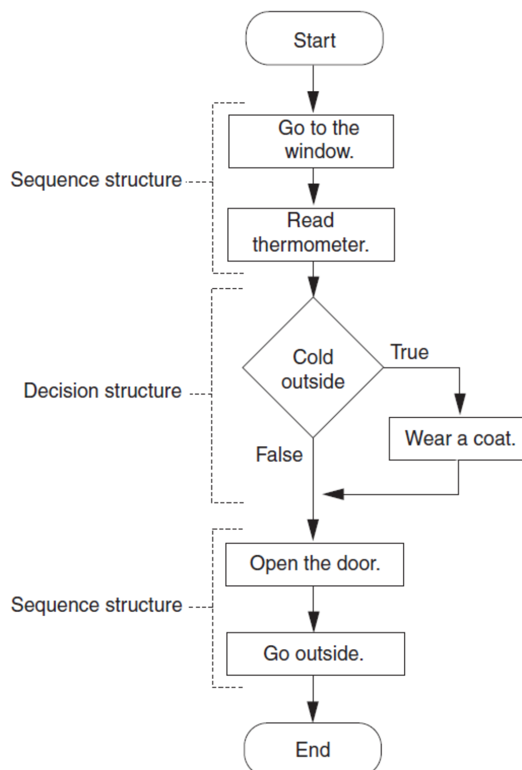
A control structure is a logical design that controls the order in which a set of statements executes.

A specific action is performed only if a certain condition exists. If the condition does not exist, the action is not performed.

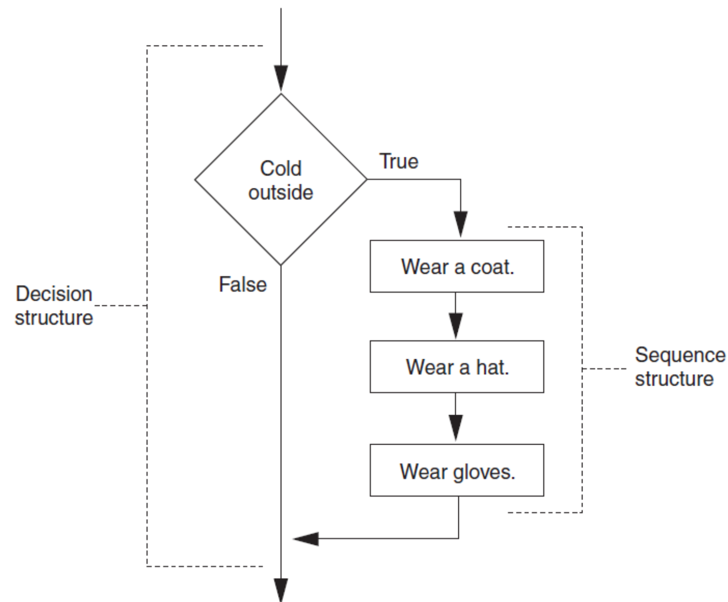
A single alternative decision structure: it provides only one alternative path of execution.



Combining decision structure:



A sequence structure nested inside a decision structure:



In Pseudocode:

In pseudocode we use the If-Then statement to write a single alternative decision structure.

```
If condition Then
    statement
    statement
    etc.
End If
```

These statements are conditionally executed.

Boolean Expressions and Relational Operators

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Expression	Meaning
x > y	Is x greater than y?
x < y	Is x less than y?
x >= y	Is x greater than or equal to y?
x <= y	Is x less than or equal to y?
x == y	Is x equal to y?
x != y	Is x not equal to y?

Example:

```
If sales > 50000 Then
   Set bonus = 500.0
   Set commissionRate = 0.12
   Display "You've met your sales quota!"
End If
```

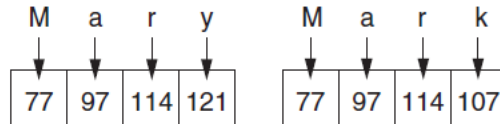
NOTE: Make sure the If clause and the End If clause are aligned. Indent the conditionally executed statements that appear between the If clause and the End If clause.

A *dual alternative decision structure* has two possible paths of execution - one path is taken if a condition is true, and the other path is taken if the condition is false.

```
If condition Then
   statement
   statement       } These statements are executed if the condition is true.
   etc.
Else
   statement
   statement       } These statements are executed if the condition is false.
   etc.
End If
```

NOTE: Most programming languages allow you to compare strings (case sensitive). This allows you to create decision structures that test the value of a string.

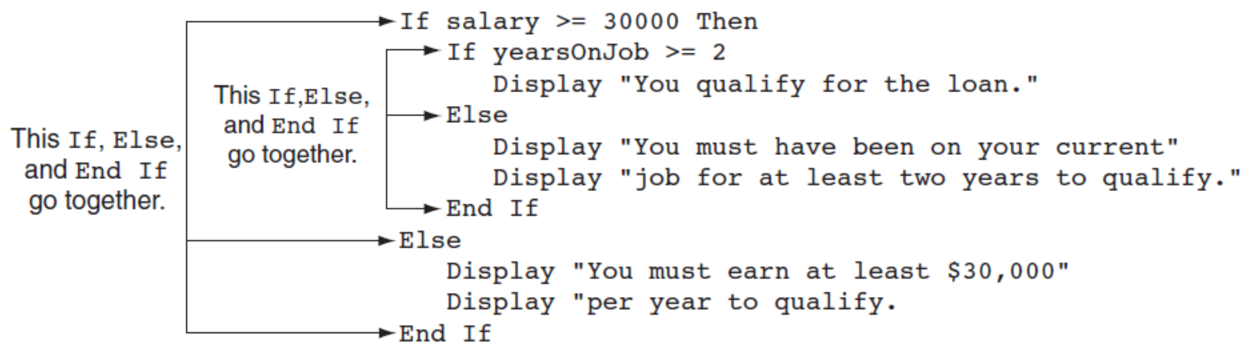
The uppercase characters "A" through "Z" are represented by ASCII values 65 through 90.
The lowercase characters "a" through "z" are represented by ASCII values 97 through 122.
0-9 as digits / characters [different ASCII values 48-57]
" " [ASCII value 32]



When you use relational operators to compare these strings, they are compared character-by-character. For example, look at the following pseudocode:

```
Declare String name1 = "Mary"
Declare String name2 = "Mark"
If name1 > name2 Then
   Display "Mary is greater than Mark"
Else
   Display "Mary is not greater than Mark"
End If
```

To test more than one condition, a decision structure can be nested inside another decision structure.



The Case Structure

The *case structure* is a *multiple alternative decision structure*. It allows you to test the <u>value</u> of a variable or an <u>expression</u> and then use that value to determine which statement or set of statements to execute.

The first line of the structure starts with the word <u>Select</u>, followed by a *testExpression.* The *testExpression* is usually a variable, but in many languages, it can also be anything that gives a value (such as a math expression).

Inside the structure there is one or more blocks of statements that begin with a Case statement. The word <u>Case</u> is followed by a value.

If the *testExpression* does not match any of the Case values, the program branches to the <u>Default</u> statement and executes the statements that immediately follow it.

```
Select testExpression        ← This is a variable or an expression.
   Case value_1:
      statement              } These statements are executed if the
      statement              } testExpression is equal to value_1.
      etc.
   Case value_2:
      statement              } These statements are executed if the
      statement              } testExpression is equal to value_2.
      etc.
   Insert as many Case sections as necessary
   Case value_N:
      statement              } These statements are executed if the
      statement              } testExpression is equal to value_N.
      etc.
   Default:
      statement              } These statements are executed if the testExpression
      statement              } is not equal to any of the values listed after the Case
      etc.                   } statements.
End Select                   ← This is the end of the structure.
```

Logical Operators

The logical **AND** operator and the logical **OR** operator allow you to connect multiple Boolean expressions to create a compound expression. The logical **NOT** operator reverses the truth of a Boolean expression.
Truth Tables

Short-Circuit Evaluation (AND, OR)

```
1  // Declare variables
2  Declare Real salary, yearsOnJob
3
4  // Get the annual salary.
5  Display "Enter your annual salary."
6  Input salary
7
8  // Get the number of years on the current job.
9  Display "Enter the number of years on your ",
10         "current job."
11 Input yearsOnJob
12
13 // Determine whether the user qualifies.
14 If salary >= 30000 AND yearsOnJob >= 2 Then
15    Display "You qualify for the loan."
16 Else
17    Display "You do not qualify for this loan."
18 End If
```

What about?!!!

```
If x < 20 AND x > 40 Then
Display "The value is outside the acceptable range."
End If
```

Boolean Variables

A Boolean variable can hold one of two values: true or false. Boolean variables are commonly used as flags, which indicate whether specific conditions exist.

NOTE: Most programming languages have key words such as True and False that can be assigned to Boolean variables.

Chapter 5

A repetition structure (*loop*) causes a statement or set of statements to execute repeatedly.

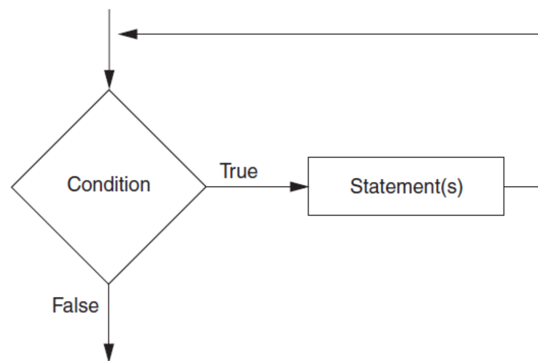
A *condition-controlled loop: (While, Do-While, Do-Until)*

It uses a true/false condition to control the number of times that it repeats.

The While Loop

A *pretest* loop, which means it tests its condition *before* performing an iteration.

While a condition is true, do some tasks.




```
While condition
    statement
    statement
    etc.
End While
```

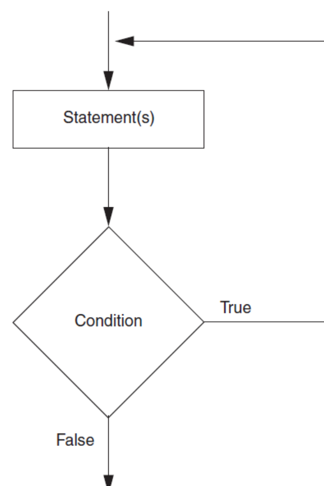
These statements are the body of the loop. They are repeated while the condition is true.

```
 1 // Variable declarations
 2 Declare Real sales, commission
 3 Declare String keepGoing = "y"
 4
 5 // Constant for the commission rate
 6 Constant Real COMMISSION_RATE = 0.10
 7
 8 While keepGoing == "y"
 9     // Get the amount of sales.
10     Display "Enter the amount of sales."
11     Input sales
12
13     // Calculate the commission.
14     Set commission = sales * COMMISSION_RATE
15
16     // Display the commission
17     Display "The commission is $", commission
18
19     Display "Do you want to calculate another"
20     Display "commission? (Enter y for yes.)"
21     Input keepGoing
22 End While
```

NOTE: An *infinite loop* continues to repeat until the program is interrupted. Infinite loops usually occur when the programmer ***forgets*** to write code inside the loop that makes the test condition false.

The Do-While Loop

A <u>*posttest*</u> loop, which means it performs an iteration before testing its condition. As a result, the Do-While loop always performs at least <u>one iteration</u>, even if its condition is false to begin with.

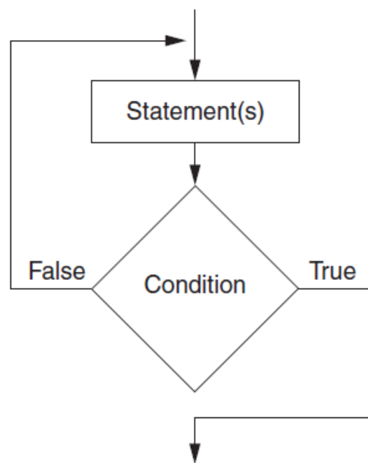


```
Do
    statement
    statement
    etc.
While condition
```

These statements are the body of the loop. They are always performed once, and then repeated while the condition is true.

The Do-Until Loop

A *posttest* loop, which means it iterates as long as a condition is false, and then stops when the condition becomes true.




```
Do
    statement
    statement
    etc.
Until condition
```

These statements are the body of the loop. They are always performed once, and then repeated until the condition is true.

```
1 // Declare a variable to hold the password.
2 Declare String password
3
4 // Repeatedly ask the user to enter a password
5 // until the correct one is entered.
6 Do
7    // Prompt the user to enter the password.
8    Display "Enter the password."
9    Input password
10
11    // Display an error message if the wrong
12    // password was entered.
13    If password != "prospero" Then
14       Display "Sorry, try again."
15    End If
16 Until password == "prospero"
17
18 // Indicate that the password is confirmed.
19 Display "Password confirmed."
```

For Loop

It repeats a specific number of times.

```
For counterVariable = startingValue To maxValue
     statement
     statement      These statements are the body of the loop.
     statement
     etc.
End For
```

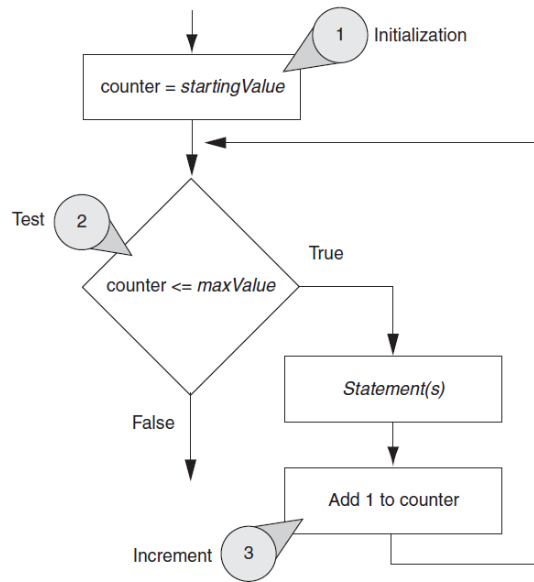
```
1 Declare Integer counter
2 Constant Integer MAX_VALUE = 5
3
4 For counter = 1 To MAX_VALUE
5    Display "Hello world"
6 End For
```

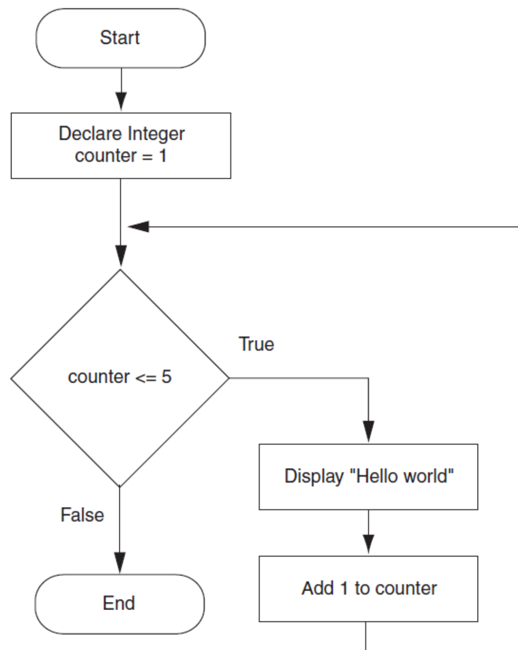
```
 1 // Variables
 2 Declare Integer counter, square
 3
 4 // Constant for the maximum value
 5 Constant Integer MAX_VALUE = 10
 6
 7 // Display table headings.
 8 Display "Number", Tab, "Square"
 9 Display "-----------------------"
10
11 // Display the numbers 1 through 10 and
12 // their squares.
13 For counter = 1 To MAX_VALUE
14    // Calculate number squared.
15    Set square = counter^2
16
17    // Display number and number squared.
18    Display counter, Tab, square
19 End For
```

NOTE: The amount by which the counter variable is incremented in a For loop is known as the *step amount*. By default, the step amount is 1. Most languages provide a way to change the step amount (+ve, -ve).

```
 1 // Declare a counter variable
 2 Declare Integer counter
 3
 4 // Constant for the maximum value
 5 Constant Integer MAX_VALUE = 11
 6
 7 // Display the odd numbers from 1 through 11.
 8 For counter = 1 To MAX_VALUE Step 2
 9    Display counter
10 End For
```

What about a *count-controlled loop using a While Loop*?!!!





WARNING! If you forget to increment the counter variable in a count-controlled `While` loop, the loop will iterate an infinite number of times.

Can we count Backwards?!!!

Nested For Loop

```
Declare Integer hours, minutes, seconds
For hours = 0 To 23
   For minutes = 0 To 59
      For seconds = 0 To 59
         Display hours, ":", minutes, ":", seconds
      End For
   End For
End For
```

Chapter 6

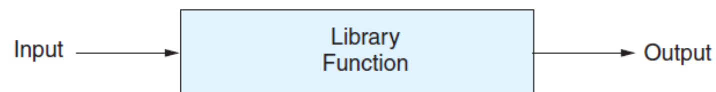
A *function* is a module that returns a value back to the part of the program that called it.

Like a regular module:
- A function is a group of statements that perform a specific task
- When you want to execute a function, you call it

Library Functions

Most programming languages come with library functions that have already been written (built into the programming language).

The library functions are used to manipulate numbers and perform various math operations, to convert data from one type to another, to manipulate strings … etc.



random function

The random function returns an integer value.

Some number

```
Set number = random(1, 100)
```

A random number in the range of 1 through 100 will be assigned to the `number` variable.

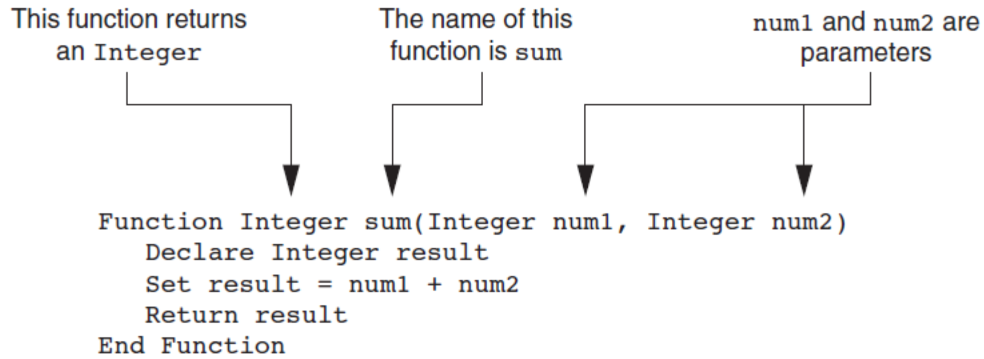
```
 1 // Declare a variable to control the
 2 // loop iterations.
 3 Declare String again
 4
 5 Do
 6    // Roll the dice.
 7    Display "Rolling the dice..."
 8    Display "Their values are:"
 9    Display random(1, 6)
10    Display random(1, 6)
11
12    // Do this again?
13    Display "Want to roll them again? (y = yes)"
14    Input again
15 While again == "y" OR again == "Y"
```

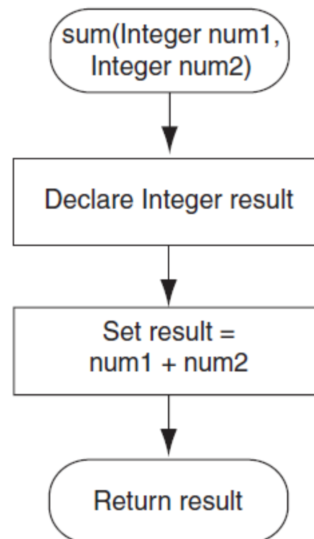
```
 1 // Declare a counter variable.
 2 Declare Integer counter
 3
 4 // Constant for the number of flips.
 5 Constant Integer NUM_FLIPS = 10
 6
 7 For counter = 1 To NUM_FLIPS
 8    // Simulate the coin flip.
 9    If random(1, 2) == 1 Then
10       Display "Heads"
11    Else
12       Display "Tails"
13    End If
14 End For
```

Writing Functions

```
Function DataType FunctionName(ParameterList)
   statement
   statement
   etc.
   Return value
End Function
```

A function must have a `Return` statement. This causes a value to be sent back to the part of the program that called the function.





Returning Strings

```
Function String getName()
   // Local variable to hold the user's name.
   Declare String name

   // Get the user's name.
   Display "Enter your name."
   Input name

   // Return the name.
   Return name
End Function
```

Returning Boolean

```
Function Boolean isEven(Integer number)
   // Local variable to hold True or False.
   Declare Boolean status

   // Determine whether number is even. If it is, set
   // status to True. Otherwise, set status to False.
   If number MOD 2 == 0 Then
      Set status = True
   Else
      Set status = False
   End If

   // Return the value in the status variable.
   Return status
End Function
```

Example,
```
If isEven(number) Then
    Display "The number is even."
Else
    Display "The number is odd."
End If
```

sqrt function

The sqrt function accepts an argument and returns the square root of the argument.

```
 1 // Variable declarations
 2 Declare Real a, b, c
 3
 4 // Get the length of side A.
 5 Display "Enter the length of side A."
 6 Input a
 7
 8 // Get the length of side B.
 9 Display "Enter the length of side B."
10 Input b
11
12 // Calculate the length of the hypotenuse.
13 Set c = sqrt(a^2 + b^2)
14
15 // Display the length of the hypotenuse.
16 Display "The length of the hypotenuse is ", c
```

pow function

The purpose of the pow function is to raise a number to a power.

Example,
Set area = pow (4, 2)

`abs`	Returns the absolute value of the argument. *Example*: After the following statement executes, the variable `y` will contain the absolute value of the value in `x`. The variable `x` will remain unchanged. `y = abs(x)`
`cos`	Returns the cosine of the argument. The argument should be an angle expressed in radians. *Example*: After the following statement executes, the variable `y` will contain the cosine of the angle stored in the variable `x`. The variable `x` will remain unchanged. `y = cos(x)`
`round`	Accepts a real number as an argument and returns the value of the argument rounded to the nearest integer. For example, `round(3.5)` will return 4, and `round(3.2)` will return 3. *Example*: After the following statement executes, the variable `y` will contain the value of the variable `x` rounded to the nearest integer. The variable `x` will remain unchanged. `y = round(x)`
`sin`	Returns the sine of the argument. The argument should be an angle expressed in radians. *Example*: After the following statement executes, the variable `y` will contain the sine of the angle stored in the variable `x`. The variable `x` will remain unchanged. `y = sin(x)`
`tan`	Returns the tangent of the argument. The argument should be an angle expressed in radians. *Example*: After the following statement executes, the variable `y` will contain the tangent of the angle stored in the variable `x`. The variable `x` will remain unchanged. `y = tan(x)`

Formatting Functions

i.e.
Declare Real amount = 6450.879
Display currencyFormat(amount)

NOTE: Many programming languages today support *localization,* which means they can be configured for a specific country. In these languages a function such as currencyFormat would display the correct currency symbol for the country that the program is localized for.

length function

The length function returns the length of a string. It accepts a string as its argument and returns the number of characters in the string – Integer.

```
Display "Enter your new password."
Input password
If length(password) < 6 Then
   Display "The password must be at least six characters long."
End If
```

append function (concatenation)

The append function accepts two strings as arguments, *string1* and *string2*. It returns a third string that is created by appending string2 to the end of string1.

```
Declare String lastName = "Conway"
Declare String salutation = "Mr. "
Declare String properName
Set properName = append(salutation, lastName)
Display properName
```

toUpper and toLower functions

The toUpper and toLower functions convert the case of the alphabetic characters in a string.

```
Declare String str = "Hello World!"
Display toUpper(str)
```

```
Declare String str = "WARNING!"
Display toLower(str)
```

```
Declare String again
Do
   Display "Hello!"
   Display "Do you want to see that again? (Y = Yes)"
   Input again
While toUpper(again) == "Y"
```

substring function

The substring function typically accepts three arguments: (1) a string that you want to extract a substring from, (2) the beginning position of the substring, and (3) the ending position of the substring.

NOTE: The first character in a string is at position 0

```
Declare String str = "New York City"
Declare String search
Set search = substring(str, 5, 7)
Display search
```

```
1 // Declare a variable to hold a string.
2 Declare String str
3
4 // Declare a variable to hold the number
5 // of Ts in a string.
6 Declare Integer numTs = 0
7
8 // Declare a counter variable.
9 Declare Integer counter
10
11 // Get a sentence from the user.
12 Display "Enter a string."
13 Input str
14
15 // Count the number of Ts in the string.
16 For counter = 0 To length(str)
17    If substring(str, counter, counter) == "T" Then
18       numTs = numTs + 1
19    End If
20 End For
21
22 // Display the number of Ts.
23 Display "That string contains ", numTs
24 Display "instances of the letter T."
```

contains function

The contains function accepts two strings as arguments. It returns True if the first string contains the second string; otherwise, the function returns False.

```
Declare string1 = "four score and seven years ago"
Declare string2 = "seven"
If contains(string1, string2) Then
   Display string2, " appears in the string."
Else
   Display string2, " does not appear in the string."
End If
```

Chapter 7

Garbage In, Garbage Out

If a program reads bad data as input, it will produce bad data as output. Programs should be designed to reject bad data that is given as input.

```
// Get a test score.
Display "Enter a test score."
Input score
// Validate the test score.
While score < 0 OR score > 100
   Display "ERROR: The score cannot be less than 0"
   Display "or greater than 100."
   Display "Enter the correct score."
   Input score
End While
```

```
// Get the answer to the question.
Display "Is your supervisor an effective leader?"
Input answer
// Validate the input.
While toLower(answer) != "yes" AND toLower(answer) != "no"
   Display "Please answer yes or no. Is your supervisor an"
   Display "effective leader?"
   Input answer
End While
```

Chapter 8

Remember: A Variable can only hold one value at a time.

So, what do you need to store and process a list of data of the same type?!!!

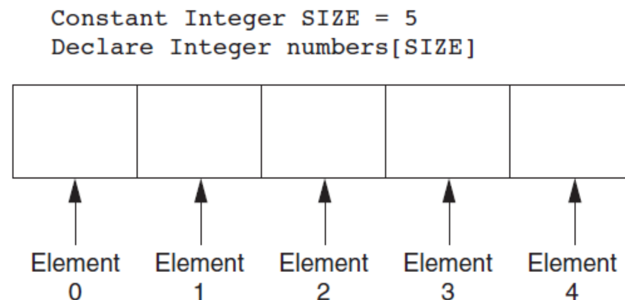
An array allows you to store a group of items of the same data type together in memory (i.e. names, grades, salaries, addresses, date of births …etc.).

Processing a large number of items in an array is usually easier than processing a large number of items stored in separate variables.

Most programming languages allow you to create *arrays*, which are specifically designed for storing and processing lists of data.

An array is a named storage location in memory.

NOTE: You cannot store a mixture of data types in an array.



Example:
Declare Integer units[10]
Declare Real salesAmounts[7]
Declare String names[50]

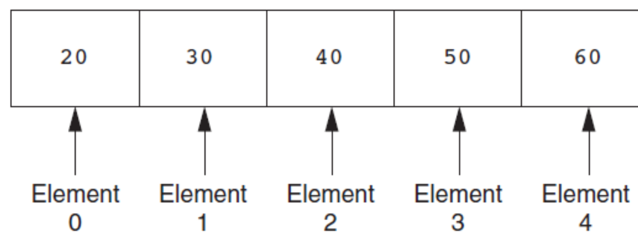
OR,

Constant Integer SIZE = 10
Declare Integer units[SIZE]

The units array has 10 elements. Each element is assigned a subscript that starts from 0 to SIZE-1.

Assigning Values to Array Elements

Example:
Constant Integer SIZE = 5
Declare Integer numbers[SIZE]
Set numbers[0] = 20
Set numbers[1] = 30
Set numbers[2] = 40
Set numbers[3] = 50
Set numbers[4] = 60




```
1  // Create a constant for the size of the array.
2  Constant Integer SIZE = 3
3
4  // Declare an array to hold the number of hours
5  // worked by each employee.
6  Declare Integer hours[SIZE]
7
8  // Declare a variable to use in the loops.
9  Declare Integer index
10
11 // Get the hours for each employee.
12 For index = 0 To SIZE - 1
13    Display "Enter the hours worked by"
14    Display "employee number ", index + 1
15    Input hours[index]
16 End For
17
18 // Display the values entered.
19 Display "The hours you entered are:"
20 For index = 0 To SIZE - 1
21    Display hours[index]
22 End For
```

```
1 // Declare an Integer array with 10 elements.
2 Declare Integer series[10]
3
4 // Declare a variable to use in the loop.
5 Declare Integer index
6
7 // Set each array element to 100.
8 For index = 0 To 9
9    Set series[index] = 100
10 End For
```

Array Initialization

Most languages allow you to initialize an array with values when you declare it.

```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE] = 10, 20, 30, 40, 50
```

```
Constant Integer SIZE = 7
Declare String days[SIZE] = "Sunday", "Monday", "Tuesday",
                            "Wednesday", "Thursday", "Friday",
                            "Saturday"
```

off-by-one Error

// This code has an off-by-one error.
Constant Integer SIZE = 100;
Declare Integer numbers[SIZE]
Declare Integer index
For index = 1 To SIZE - 1
Set numbers[index] = 0
End For

// ERROR!
For index = 0 To SIZE
Set numbers[index] = 0
End For

The For Each Loop

Several programming languages provide a specialized version of the For loop that is known as the For Each loop.

The *For Each* loop can simplify array processing when your task is simply to step through an array, retrieving the value of each element.

```
For Each var In array
   statement
   statement
   statement
   etc.
End For
```

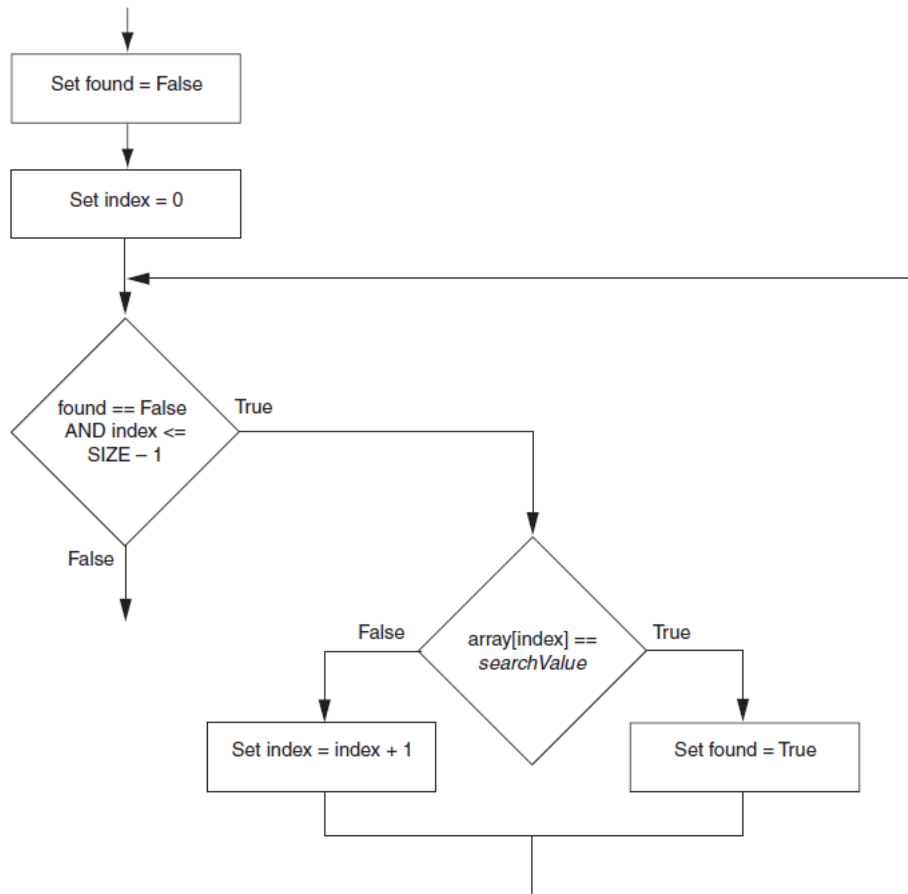
The first time the loop iterates, *var* will contain the value of *array*[0], the second time the loop iterates *var* will contain the value of *array*[1] … etc.

Example:
Constant Integer SIZE = 5
Declare Integer numbers[SIZE] = 5, 10, 15, 20, 25
Declare Integer num

```
For Each num In numbers
    Display num
End For
```

Sequentially Searching an Array

The sequential search algorithm is a simple technique for finding an item in an array. It steps through the array, beginning at the first element, and compares each element to the item being searched for. The search stops when the item is found or the end of the array is reached.




```
Set found = False
Set index = 0
While found == False AND index <= SIZE – 1
   If array[index] == searchValue Then
      Set found = True
   Else
      Set index = index + 1
   End If
End While
```

Copying an Array

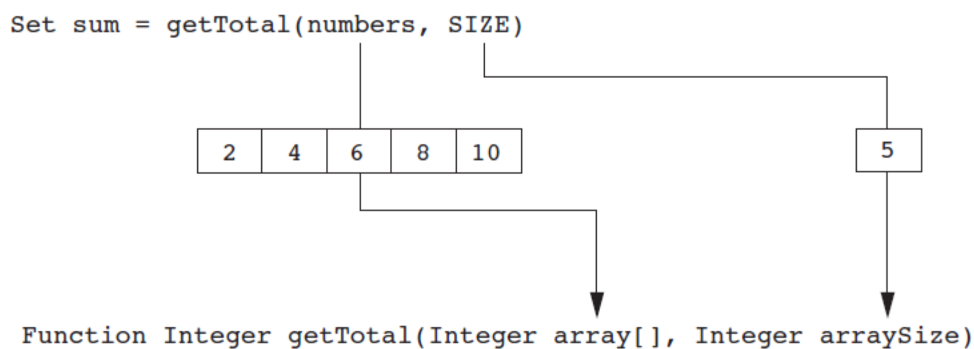
```
Constant Integer SIZE = 5
Declare Integer firstArray[SIZE] = 100, 200, 300, 400, 500
Declare Integer secondArray[SIZE]
Declare Integer index

For index = 0 To SIZE – 1
    Set secondArray[index] = firstArray[index]
End For
```

Passing an Array as an Argument to a Module or Function

Most languages allow you to pass an array as an argument to a module or a function.

Passing an array as an argument typically requires that you pass two arguments:
(1) the array itself, and
(2) an integer that specifies the number of elements in the array.

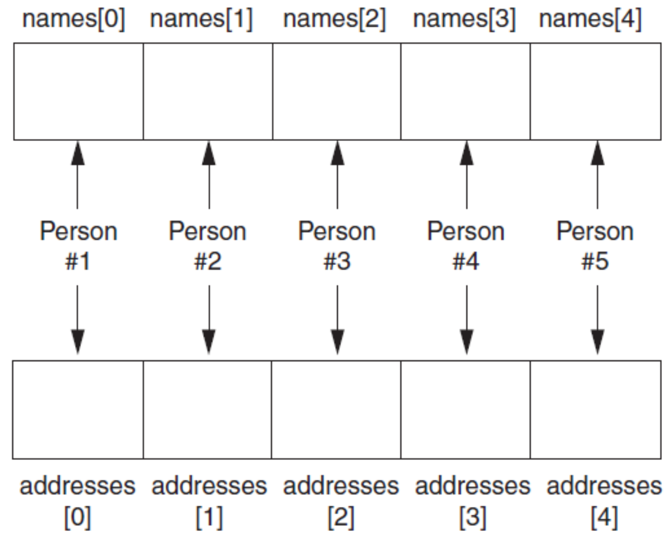


Parallel Arrays

By using the same subscript, you can establish relationships between data stored in two or more arrays.

```
Constant Integer SIZE = 5
Declare String names[SIZE]
Declare String addresses[SIZE]

Declare Integer index
For index = 0 To SIZE – 1
   Display names[index]
   Display addresses[index]
End For
```



Two-Dimensional (2D) Arrays

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column.

Example:
Constant Integer ROWS = 3
Constant Integer COLS = 4
Declare Integer values[ROWS][COLS]

	Column 0	Column 1	Column 2	Column 3
Row 0	values[0][0]	values[0][1]	values[0][2]	values[0][3]
Row 1	values[1][0]	values[1][1]	values[1][2]	values[1][3]
Row 2	values[2][0]	values[2][1]	values[2][2]	values[2][3]

```
1 // Create a 2D array.
2 Constant Integer ROWS = 3
3 Constant Integer COLS = 4
4 Declare Integer values[ROWS][COLS]
5
6 // Counter variables for rows and columns.
7 Declare Integer row, col
8
9 // Get values to store in the array.
10 For row = 0 To ROWS - 1
11    For col = 0 To COLS - 1
12       Display "Enter a number."
13       Input values[row][col]
14    End For
15 End For
16
17 // Display the values in the array.
18 Display "Here are the values you entered."
19 For row = 0 To ROWS - 1
20    For col = 0 To COLS - 1
21       Display values[row][col]
22    End For
23 End For
```

Most languages allow you to initialize a two-dimensional array with data when you declare the array.

```
Declare Integer testScores[3][4] = 88, 72, 90, 92,
                                   67, 72, 91, 85,
                                   79, 65, 72, 84
```

Arrays of 3D or More Dimensions

Most languages allow you to create arrays with multiple dimensions.

Example:
Declare Real seats [3][5][8]