Outsource Laboratories

Introduction to Programming with Microsoft® Visual Basic® .NET

Outsource Laboratories Press™

Printed in the United States of America

ISBN 0-9725199-7-1

1 2 3 4 5 6 7 8 9 10 09 08 07 06 05 04

OUTSOURCE LABORATORIES PRESS
PO Box 187, Matawan, NJ 07747-0187 USA

For more information about Outsource Laboratories Press' textbooks and services, contact us:

Toll free: 888-GO-OLABS (888-466-5227)
Email: training@olabs.com
http://www.olabs.com

Technical Support: Outsource Laboratories will offer limited technical support to instructors who have purchased classroom sets and have registered with Outsource Laboratories. Contact your representative for more information.

OL300 11194

About Olabs' Text

This text is organized into units, and within a unit it is organized by up to three levels of headings. The unit title and first level heading are shown in the header of each page following the first page of a unit. This allows you to quickly relate lower level headings to the main topic simply by looking at the page header.

The figures in the text coordinate with presentation files used by the instructor. If your instructor is showing a slide labeled Figure 2-20, you will be able to find this in your text labeled with the same number (the figure number 2-20 references the 20th figure in Unit 2).

All the units conclude with review questions.

Exercises are placed throughout the units, close to relevant topics. Exercises may require coding, however some require analysis only. Icons next to the exercise heading indicate what type of exercise it is. Icons are used in other situations within the text as well.

The following icons are used in the text, however each one may not be used in every manual:

This icon represents a hands-on exercise that requires compiling a program and running it.

This icon represents a hands-on exercise which is done on paper or as a discussion; it does not require editing, compiling or running a program.

This icon represents a running code sample that is to be studied.

This icon represents that the material is related to a sample application.

Accompanying this text is a a set of electronic files which contain exercise templates and solutions organized to follow the unit structure of the text, sample programs referenced in the text, as well as solutions to analysis type exercises. Your instructor maintains files required for examples and exercises.

1 Basics of Programming

Introduction

In this unit we provide some background on computer programming. We begin by explaining what programming is about and what it involves. We then take a brief look at the history of programming to see how programming evolved into what it is today. In our coverage of history, you will learn about the architecture of a stored program computer, on which modern computer architectures are based. You will also become aware of the three levels of programming languages that have emerged. We will then start you thinking about programming by describing the six steps involved in program development. Finally, you will learn how to outline program solutions using flowcharts and pseudocode.

Figure 1-1 lists the objectives of this unit.

At the end of this unit, you will be able to:

1. Understand what programming is.
2. Understand how programming evolved into what it is today.
3. Understand how a stored program computer works.
4. Explain the three levels of programming languages.
5. List the steps in developing a program.
6. Analyze programming problems.
7. Write algorithms using flowcharts and pseudocode.

Figure 1-1: Unit Objectives

What is Programming?

Programming has been around for over a century. However, it did not really become popular as a field of study until the last 40 years. During this time, we saw the widespread rise in popularity of the computer due to the dramatic decrease in its cost and the significant improvements in computer technology that improved its performance.

Today, computers are an important part of every day life. We see them in action when we withdraw money from bank ATMs, when we make calls using a cell phone, when we nuke our food using a microwave. Also, the advent of the Internet and the World Wide Web has given rise to a plethora of online businesses and the adoption of email as a form of communication.

The use of computers in so many different areas has generated increased interest in programming as a field of study and as a career. So what exactly does programming involve? Programming involves creating programs. This begs the question: What is a **program**? A program is a sequence of instructions that can be understood by a computer. The purpose of a program is to make a computer perform some task or solve some problem. We also use the terms **software** or **application** to refer to a program.

The instructions that make up a program are not written in English. They are written in a **programming language** that can be understood by a computer. In this course, we will be working with the Visual Basic .NET programming language but examples of others include C, C++ and Java.

Figure 1-2 provides a summary of what programming is about.

- Programming involves creating programs.
- A program is a sequence of instructions that can be understood by a computer.
- Programs are also referred to as software or applications.
- Programs are written in a programming language.

Figure 1-2: What is Programming?

Steps for Developing a Program

Figure 1-12 lists the six steps that you would typically take in designing and writing a computer program.

1. Analyze the problem.
2. Design an algorithm to solve the problem.
3. Implement the algorithm using a programming language.
4. Compile and run the program.
5. Test and debug the program.
6. Maintain the program.

Figure 1-12: Steps for Developing a Program

We will now discuss each of the steps above in more detail.

STEP 1: ANALYZE THE PROBLEM

Once a programming problem has been defined and specified, it is necessary to analyze the problem and understand what it is about. In analyzing a problem, we need to consider that there are three main stages to a program as listed in Figure 1-13.

A program consists of three main stages:

1. Takes input.
2. Process the input.
3. Presents the output.

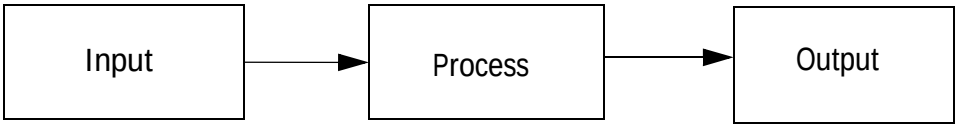


Figure 1-13: Three Main Stages of a Program

We therefore need to determine, from the problem specification, what input, output and process information is needed by the program.

Problem analysis may also require consideration of the interface to be used for data input and output. Most real world problems will require a program to interface with a user via a GUI.

A problem specification will usually provide details on what graphical components are to be included in the GUI and how these are to be laid out. Some problems, however, may only specify what the GUI should do (i.e. the input, output and process information). It is then left to the developer, during the problem analysis phase, to identify suitable GUI components and layout.

The GUI should include, at the very least, graphical components that do the following:

- Allow the user to enter input.
- Allow the user to tell the program when to process the input.
- Present the output back to the user.

When designing a GUI, it is good to keep in mind that it should be simple, well-organized and easy to use.

An example of a GUI is provided in Figure 1-14.

Figure 1-14: Example of a GUI

The GUI features a list component that contains three image file names. When an item in the list is selected, the image saved under the selected file name will be displayed along with a corresponding label that describes the list item that has been selected. A button also features in the GUI that enables a user to quit the program.

ANALYZING AN EXAMPLE PROBLEM

The Problem:

You and your friend from Europe are watching the weather forecast. The temperature is given in Fahrenheit, but your friend is used to temperature measures in Celsius. You would like to build a GUI for your friend that will convert a temperature from Fahrenheit into Celsius. The formula to convert Fahrenheit into Celsius is: Celsius = 5/9 * (Fahrenheit - 32).

Consider the Input, Output and Process Information:

The three main stages of the temperature conversion program would be:

1. Input: The temperature in Fahrenheit.
2. Process: Apply the formula to convert the Fahrenheit value into a Celsius value.
3. Output: The temperature in Celsius.

Programs should be designed to be as general as possible. In other words, they are not written to solve just one specific problem. They should be able to solve many problems that are similar in nature. Given this, the temperature conversion program should be able to convert any Fahrenheit value into a Celsius value.

Consider the GUI components and layout:

Figure 1-15 presents a sketch of a possible GUI for the temperature conversion program.

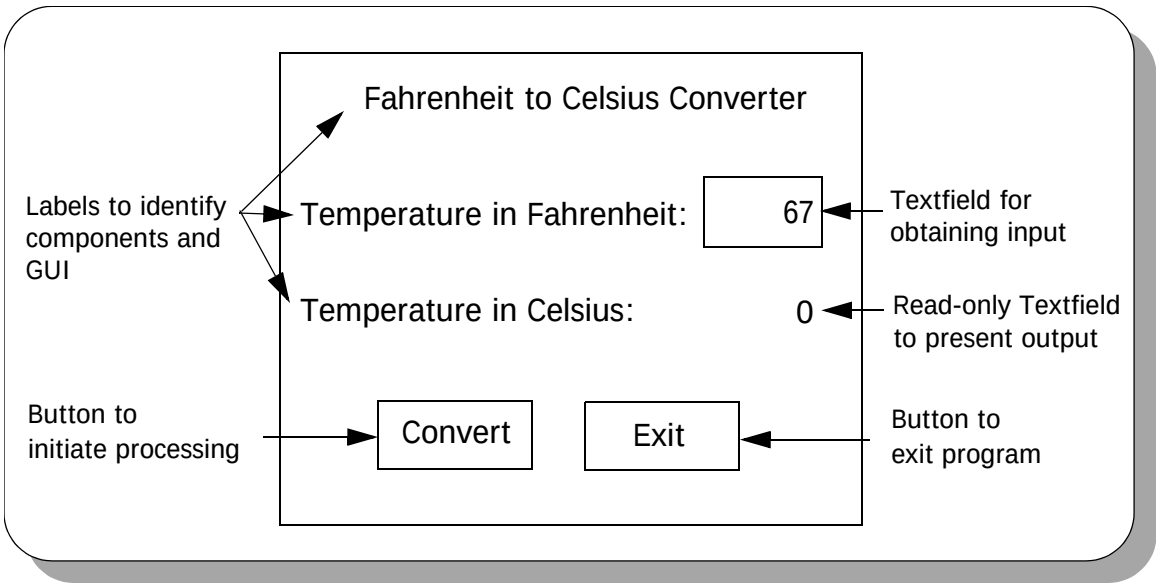


Figure 1-15: Sketch of a GUI for Temperature Conversion

The GUI in the sketch above consists of the following graphical components:

- A textfield to enable the user to enter a Fahrenheit temperature.
- A button to enable the user to initiate the processing of the input.
- A button to enable the user to exit the program.
- A read-only textfield to display the Celsius temperature back to the user.
- Labels for identifying components and GUI.

Exercise 1-1: Analyzing Simple Programming Problems

Purpose: Learn to analyze simple programming problems.

Instructions: Follow the instructions in Figure 1-16 to complete this exercise.

Identify the input, process and output stages of each of the following programming problems:

1. An online store allows its shoppers to buy a maximum of 3 products. It needs a program to add up the prices of these products and calculate the total including an 8% sales tax.
2. Given a number between 1 and 12, provide the name of the month corresponding to that number.
3. Each month, a US exchange student living in Australia receives rent, gas and electricity bills as well as a US credit card bill. The student would like a program that will give him the total of these four bills in Australian dollars so that he can keep track of how much he is spending. Since the USD to AUD exchange rate is not a constant value, the student must provide this information to the program.

Figure 1-16: "Analyzing Simple Programming Problems" Exercise

Note: Unit01\Exercises\Solutions\Exercise1-1.pdf contains the solution.

Review Questions

1. Describe the three levels of programming languages.
2. What is a stored program computer?
3. What are the three main parts of most programs?
4. What is the difference between a compiler and an interpreter?
5. Below are the steps for developing a program. They are jumbled up. Arrange them into the correct sequence.

   Test and debug the program.
   Design an algorithm to solve the problem.
   Analyze the problem.
   Compile and run the program.
   Maintain the program.
   Implement the algorithm using a programming language.
6. What is a flowchart?
7. What is pseudocode?
8. Match the programming problems on the left hand side of the table below to the pseudocode on the right hand side.

Programming Problems	**Pseudocode**

Output the result of a calculation	Cookies = 15
	WHILE (cookies > 0) DO
	Eat one cookie
	cookie - 1
	ENDWHILE

Eat cookies until they are all gone given 15 cookies.	Print result

Input the name of a person	IF age < 21
	Print “Sorry you can’t enter this place”
	ELSE
	Print “Please come in”
	ENDIF

Store the price of an item you buy in a shop	price = 67.80
Check if the age is less than 21. If it is, print “Sorry, you can’t enter this place”, otherwise print “Please come in”.	result = 7 * 45
Multiply 7 by 45 and store the result.	Read name

9. Describe the two types of bugs that can exist in a program.

2 Visual Basic .NET Programming

Introduction

In this unit, we introduce you to the Visual Basic .NET programming language. We explain the .NET Framework with which Visual Basic .NET is tightly integrated. We then describe some of the different types of programs that can be written using Visual Basic .NET.

You will become familiar with Visual Studio .NET, a development tool that provides an integrated development environment (IDE) for building Visual Basic .NET applications. Using Visual Studio .NET, you will learn to compile and run simple Visual Basic .NET applications, design graphical user interfaces, implement event handlers and write your first Visual Basic .NET application. Following this, we step you through the basic structure of a Visual Basic .NET program and show you how to document your programs using comments.

Figure 2-1 lists the objectives of this unit.

At the end of the unit, you will be able to:

1. Understand some facts about the Visual Basic .NET language
2. Understand the concept of the .NET Framework.
3. Explain the different types of programs that can be developed using Visual Basic .NET.
4. Use Visual Studio .NET to create a graphical user interface.
5. Understand the concept of an event handler and implement them using Visual Studio .NET.
6. Understand the basic structure of a Visual Basic .NET program.
7. Use comments to document a program.
8. Create a simple Visual Basic .NET application.

Figure 2-1: Unit Objectives

3 Variables, Data Types and Operators

Introduction

Before you are able to write a program that performs a simple calculation, you need to learn the basic building blocks of a program: variables, data types, and operators.

In this unit, we will discuss how to use variables to store data. We look at how data types are used to inform the computer of the type of data it is dealing with. You will learn the main operators used in Visual Basic .NET to perform calculations on data.

At the end of this unit you will be able to write your own simple programs using these basic building blocks. In the early sections of this unit, you will be working with console applications so that we can focus on simple programming logic rather than GUI design. Towards the end of the unit, when you have a better understanding of Visual Basic .NET programming logic, you will return to working with GUI applications.

Figure 3-1 lists the objectives of this unit.

At the end of this unit, you will be able to:

1. Perform variable declaration and assignment.
2. Understand why there are different data types.
3. Distinguish and use different data types.
4. Understand variables and literals.
5. Understand and use operators.
6. Write console applications as well as Windows applications.

Figure 3-1: Unit Objectives

The Add2Numbers Program

Let's start by looking at an example program that we will be using at many points throughout this unit. Our example program is a simple Visual Basic .NET console application that sums up two numbers and prints out the result.

The problem we are solving with this program is as follows:

1. Declare three variables of type integer.
2. Assign numbers to two of those variables.
3. Add these two numbers together and assign the value to the third variable.
4. Print the two numbers and the result.

The program would be written in pseudocode as follows:

```
1     Add2Numbers
2        Read number1
3        Read number2
4        result = number1 + number2
5        Print number1, number2 and result
6     END
```

The lines 2 and 3 in the pseudocode above involve the declaration of variables and assigning of values to the variables. Line 4 involves an arithmetic operation and assignment of value to a variable. Line 6 outputs values.

In Figure 3-2 we show the complete code of the Add2Numbers program. Recall that we explored the basic structure of a Visual Basic .NET program in Unit 2. Concentrate on the structure of the Add2Numbers program for the moment.

Can you answer the following three questions:

1. What type of program is the Add2Numbers program? A class or a module?
2. What is the name of the program?
3. In which line does the `Sub Main` procedure start?
4. In which line does the `Sub Main` procedure end?

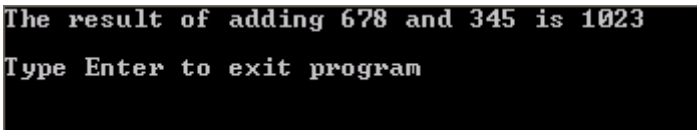
We have used the keyword 'Module' rather than 'Class' to create the Add2Numbers program. This is because we do not intend to deal with the Add2Number program as an object. It has no special attributes or behaviors, apart from running as a program that adds two numbers.

Notice also that the program does not end immediately after printing out the results to the console. It goes on to perform an additional task at lines 22 to 24 which keeps the console window open so that output of the program can be viewed. Without these lines, the console window would close immediately after the printing out of the results.

```
1  Module Add2Numbers
2
3      Sub Main()
4          ' Declare three variables of type integer.
5          Dim number1 As Integer
6          Dim number2 As Integer
7          Dim result As Integer
8
9          'Assign values to variables number1 and number2
10         number1 = 678
11         number2 = 345
12
13         ' Sum the two numbers together and store in result.
14         result = number1 + number2
15
16         ' Print out the result.
17         Console.WriteLine("The result of adding " _
18                         & number1 & " and " _
19                         & number2 _
20                         & " is " & result)
21
22         ' Wait for user to exit.
23         Console.WriteLine()
24         Console.WriteLine("Type Enter to exit program")
25         Console.ReadLine()
26     End Sub
27
28 End Module
```

Figure 3-2: Add2Numbers Program

Open the project Unit03\Examples\Add2Numbers\Add2Numbers.vbproj. Compile and run the program. Below is a screen capture that shows the program output in the console window:



Statements and Expressions

A **statement** is an instruction in a high-level programming language that tells the computer what operations need to be performed. Each statement can represent a sequence of several machine instructions.

In Visual Basic .NET, a statement typically takes up a single line of code. However, it is not uncommon for very long statements to feature in code. Long statements can be spread out over multiple lines. This is done by using a space followed by an underscore character (_) at the end of a line of code that is to be continued onto the next line.

Figure 3-3 shows some examples of statements that are part of the Add2Numbers program which we just looked at.

- Example statements from TempConverter Program:

```
line 5: Dim number1 As Integer

line 10: number1 = 678

line 17: Console.WriteLine("The result of adding " _
line 18:                 & number1 & " and " _
line 19:                 & number2 _
line 20:                 & " is " & result)
```

- Example expressions:

```
example 1: number1 + number2
example 2: “Type enter to exit program”
example 3: 678
```

Figure 3-3: Statements and Expressions

Statements are often made up of **expressions**. An expression is a combination of symbols that represent a value. Figure 3-3 provides some examples of expressions.

The symbols that combine to form expressions are called **operands** and **operators**. An operand is an entity that has a value like a number. An operator is a symbol that represents an action like subtraction.

In the first example expression listed in Figure 3-3, `number1` and `number2` are operands, and the addition symbol (`+`) symbol is an operator.

Variables

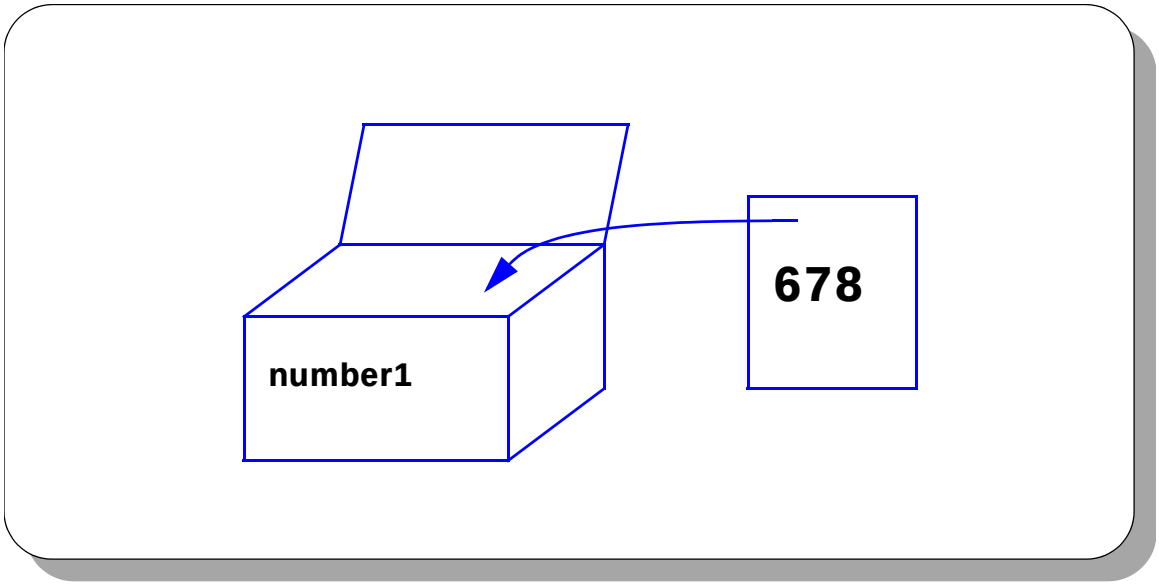


Figure 3-4: The Variable - A Storage Box for Values

Variables are like storage boxes for values. Imagine a variable as a little box with a name as shown in Figure 3-4. Imagine a piece of paper with a number or a word printed on it being placed into the box. Whenever we use the name of the box in our program, we open the box and look at the number or word on the piece of paper that is contained inside. We can then use the contents of the box. For example, the number in the box can be used in a calculation or be output to the screen.

In the Add2Numbers program that we looked at earlier in this unit, we used three variables: `number1`, `number2` and `result`. The variables `number1` and `number2` each held a number and the variable `result` held the result of adding the two numbers.

DECLARING VARIABLES

When we want to use variables in a program we have to tell the program the names of our variables. This process is known as performing a **variable declaration**. All variables must be declared before they are used in a program. In the Add2Numbers program, three variables are declared. The statements in the Add2Numbers program that declare these variables are shown in Figure 3-5.

```
5   Dim number1 As Integer
6   Dim number2 As Integer
7   Dim result As Integer
```

Figure 3-5: Declaring Variables

In every programming language there are strict rules about how everything has to be written. We call this the **syntax** of a programming language. The syntax to declare a variable in the Visual Basic .NET programming language is shown in Figure 3-6.

Syntax to declare a variable in Visual Basic .NET:

```
Dim <variable name> As <data type>
```

The following line of code declares a variable of **type** *integer* with the **name** *number1*.

```
Dim number1 As Integer
```

Figure 3-6: Declaring Variables (cont'd)

A variable declaration statement in Visual Basic .NET begins with the keyword '`Dim`' followed by the name of the variable. After the variable name is the keyword 'As' followed by the **data type** of the variable. We will discuss the possible data types which a variable can be declared as in the next section. The variables that we are using in our example program are of type integer. This means that those variables can only store whole numerical values (e.g., 1, 34, 189). In Visual Basic .NET you indicate a variable of type integer by writing `As Integer` after the name of the variable.

4 Flow Control - Selection

Introduction

So far, we know a program to be a set of instructions or statements, each of which is executed in sequence by a computer. This notion works well when the solution to a problem or task consists of a list of steps that needs to be performed in sequence.

However, in most real world problems, solutions cannot be broken down into a set of sequential steps. In many cases, you will want to perform a certain action only if a certain condition occurs. In other cases, you may want to perform a certain action repeatedly until some condition is met. We need some way to control the flow of statements in a program.

There are two types of flow control constructs in programming. The **Selection** constructs allow for a set of statements to be executed only if a certain condition is true or false. The **Repetition** constructs allow for a set of statements to be executed repeatedly while a certain condition is true. This unit focuses on **Selection** constructs. **Repetition** constructs are explored in Unit 5.

Figure 4-1 lists the objectives of this unit.

At the end of this unit, you will be able to:

1. Understand flow control.
2. Give examples of the use of selection in a program.
3. Distinguish various types of selection structures.
4. Flowchart the Selection constructs.
5. Use and understand the different types of selection.

Figure 4-1: Unit Objectives

What is Flow Control?

It is often not the case that a task which we want to program consists of a straightforward set of sequential actions. Flow control enables a program to deviate from the sequence of execution. Two types of flow control constructs can be employed in our programs.

The first type, **Selection**, is used when we want to select a course of action depending on a particular condition. For instance, an airline reservation system needs to check the availability of a flight before allowing a reservation to be made. If the flight is full, it needs to notify the user that there are no seats. Otherwise, it will let the user continue to make his\her reservation.

The other flow control construct is **Repetition**. It is used when we want an action to be performed repeatedly while a certain condition is true. An example is a photocopier that prints multiple copies of a page. The action of printing a particular page is repeated until the specified number of copies has been produced.

Figure 4-2 provides a summary of our discussion on flow control.

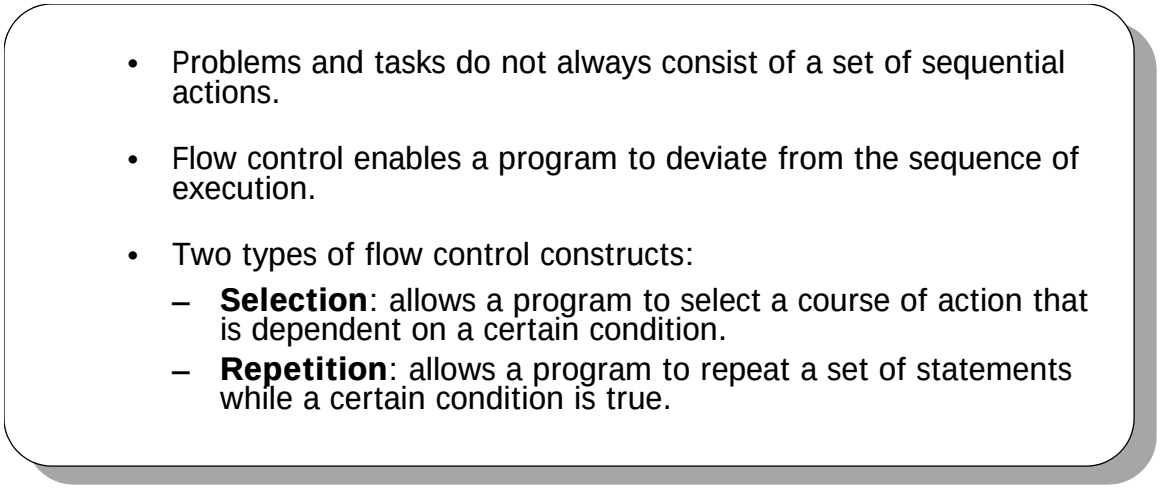


Figure 4-2: What is Flow Control?

Selection Statements

Selection statements provide us with a mechanism for writing programs that can choose between a set of statements to execute. The set of statements chosen depends on a particular condition.

Consider a very simplistic AirlineReservations program which checks flight availability and prints out "Reservation rejected" if the flight is full, or "Reservation accepted" if the flight is not full. Recall from Unit 1 that the pseudocode for choosing between alternative actions is:

```
IF ... THEN
...
ELSE
...
ENDIF
```

The pseudocode to implement this Airline Reservations program is shown in Figure 4-3.

- Selection statements - mechanism for a program to choose between a set of statements, depending on a certain condition.
- Example: AirlineReservations program.

```
Read fullFlight
IF fullFlight THEN
   Print "Reservation rejected."
ELSE
   Print "Reservation accepted."
ENDIF
```

Condition: Whether flight is full.

Two courses of action:

- Print "Reservation rejected." if flight is full.
- Print "Reservation accepted." if flight is not full.

Figure 4-3: If-Statement - AirlineReservations Example

The AirlineReservations program has two alternative courses of action: reject the reservation request or accept the reservation request. The course of action taken depends on the condition of whether or not the flight is full.

The pseudocode for a selection statement is very similar in form to its syntactical representation. Figure 4-4 shows the syntax of a selection statement.

Syntax	Example: AirlineReservations program

If <*condition*> Then
  <*true stmts*>
Else
  <*false stmts*>
End If
	If fullFlight = true THEN
	  Console.WriteLine("Reservation rejected")
	Else
	  System.out.println("Reservation accepted")
	End If

Figure 4-4: If-Statement - Syntax

A selection statement is commonly known as an **If-statement** because of the keyword '`If`' that starts it. The '`If`' keyword is followed by a condition, i.e (fullFlight == true). This condition always evaluates to either true or false. Next comes the set of statements that is executed if the condition evaluates to true. Following that is the keyword '`Else`'. After the keyword '`Else`' comes the set of statements that are executed if the condition evaluates to false.

Types of If Statements

There are three types of If statements.

1. The If-Then-Else statement that tests a condition and then has actions for the true and the false case.

2. An If-statement without an else condition for scenarios in which an action exists for the true condition but not for the false condition.

3. A multiple If-statement for scenarios in which there are not two, but multiple courses of action that can be taken depending on a particular condition.

Let's look at each of these cases in more detail.

THE IF-THEN-ELSE STATEMENT

As the logic of our programs gets more complicated, it can be helpful to represent the logic in a flowchart, a concept that we introduced in Unit 1. The flowchart in Figure 4-9 represents the decision point in our AirlineReservations program.

The diamond represents making a decision in our code. In this case we are deciding whether the variable `fullFlight` is true or not. If the flight is full (fullFlight = True) the reservation is rejected. However, if it is not full (fullFlight = False), we accept the reservation. After that, the program continues on with any subsequent statements.

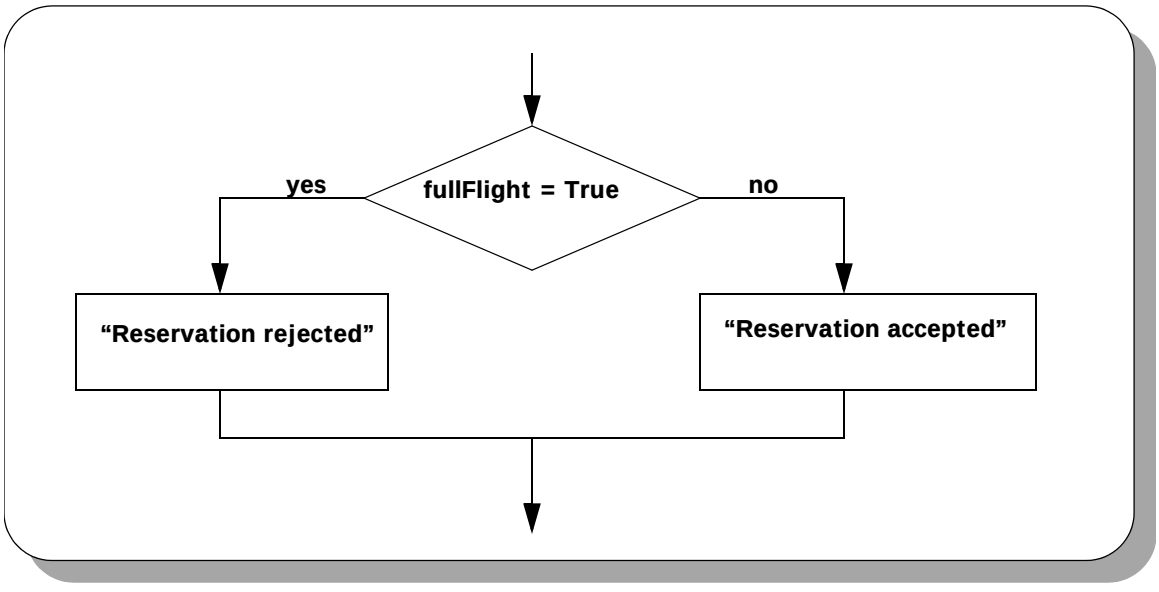


Figure 4-9: Flowchart: If-Then-Else Statement - AirlineReservations

An example that uses the If-Then-Else statement is the following weather program. Let's study the flowchart of the program first and then the pseudocode. The flowchart is depicted in Figure 4-10.

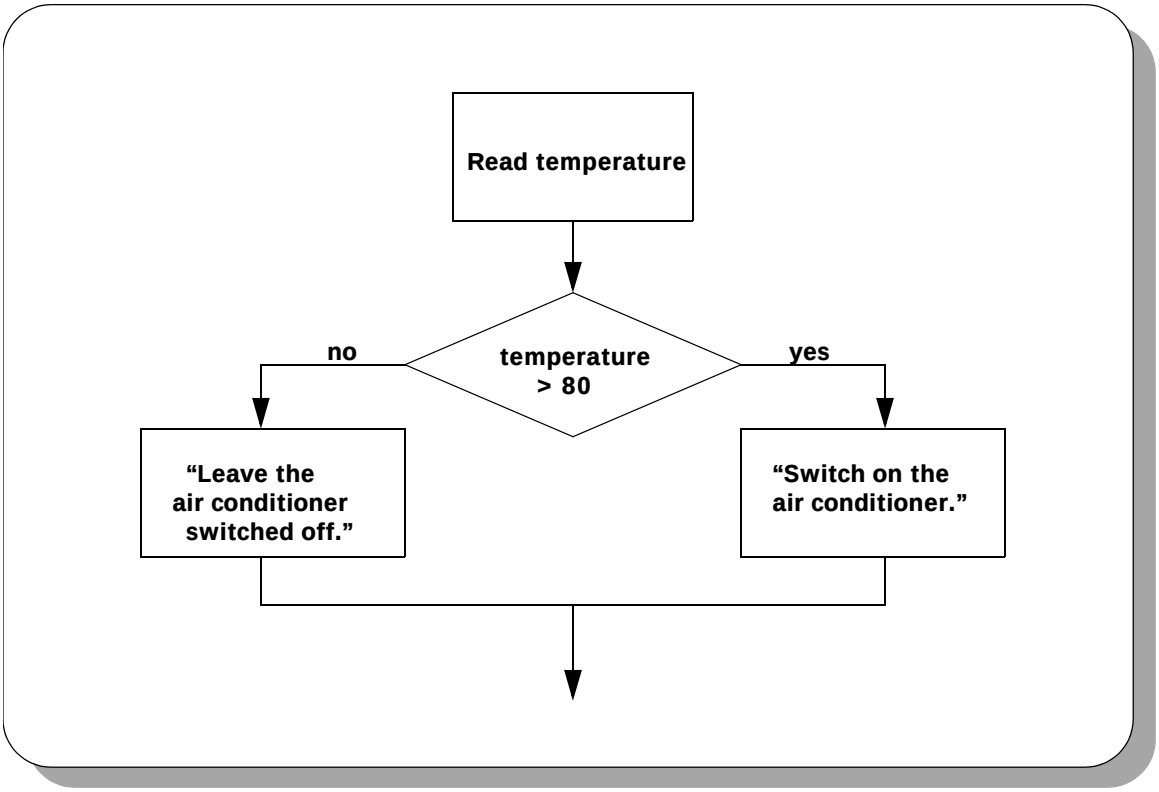


Figure 4-10: Flowchart: If-Then-Else Statement - WeatherData

The pseudocode for the WeatherData program is as follows:

```
WeatherData
   Read temperature
   IF temperature > 80 THEN
      Print "Switch on the air conditioner."
   ELSE
      Print "Leave the air conditioner switched off."
   ENDIF
END
```

Let us study the Visual Basic .NET code for this program that is displayed in Figure 4-11.

7 Arrays

Introduction

In Unit 3, you learned how to use variables in a program. A variable can only be used to store a single value. There are situations in which you may need to store a number of values of the same type. You could store each value in a different variable, however, Visual Basic .NET and other programming languages provide a data structure called an array that enables a set of values of the same type to be stored conveniently.

In this unit you will learn how to use arrays and where they can be applied in your program.

Figure 7-1 lists the objectives of this unit.

At the end of this unit, you will be able to:

1. Declare, create and use arrays.
2. Identify scenarios in which arrays would be used.
3. Search through the elements of an array for a value.
4. Use command-line parameters in a Visual Basic .NET console application.

Figure 7-1: Unit Objectives

What is an Array?

An array is a data structure used in programming to store and manipulate a collection of elements of the same data type. More formally, an array is a linear sequence of one or more variables with the same variable name, but distinguished by a positional number called an **index**. The term commonly used to describe a variable of an array is **element**.

Figure 7-2 shows a graphical representation of an array and some of its more important characteristics.

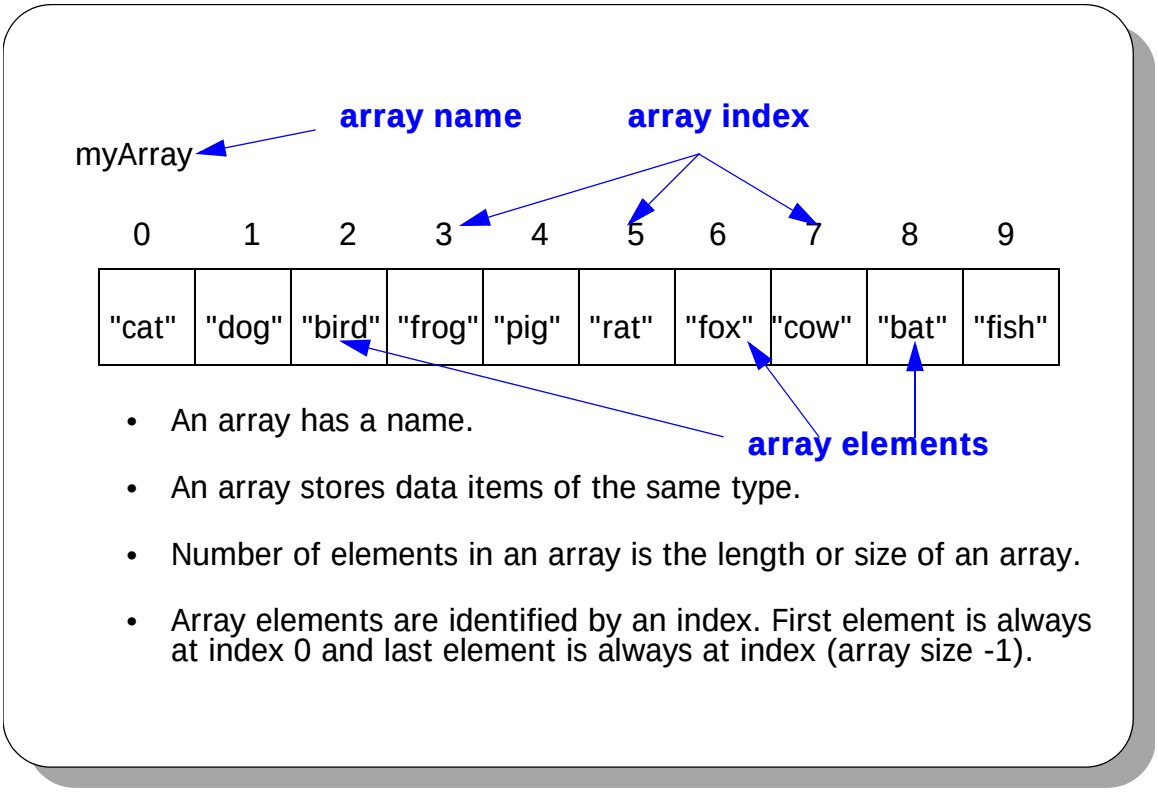


Figure 7-2: What is an Array?

Let's look at an example in which the use of an array would be appropriate. Consider a program that keeps track of the names of the people per seat in the rows of a movie theatre. To identify each person you need to store the name of the person associated with the seat number. You could create a variable for each seat. But that would make the handling of the data in the program difficult. Instead you can use an array to store the names of the people in one row. Then you would have an array for each row of the theatre.

We will use row number 7 as an example. In this row there are 10 seats; each seat is occupied by one person. Let's call the array 'row7'. It will have a length of 10 with the index starting at 0 and ending at 9. The elements of the array will be of type `String`.

The graphical representation of our example is shown in Figure 7-3.

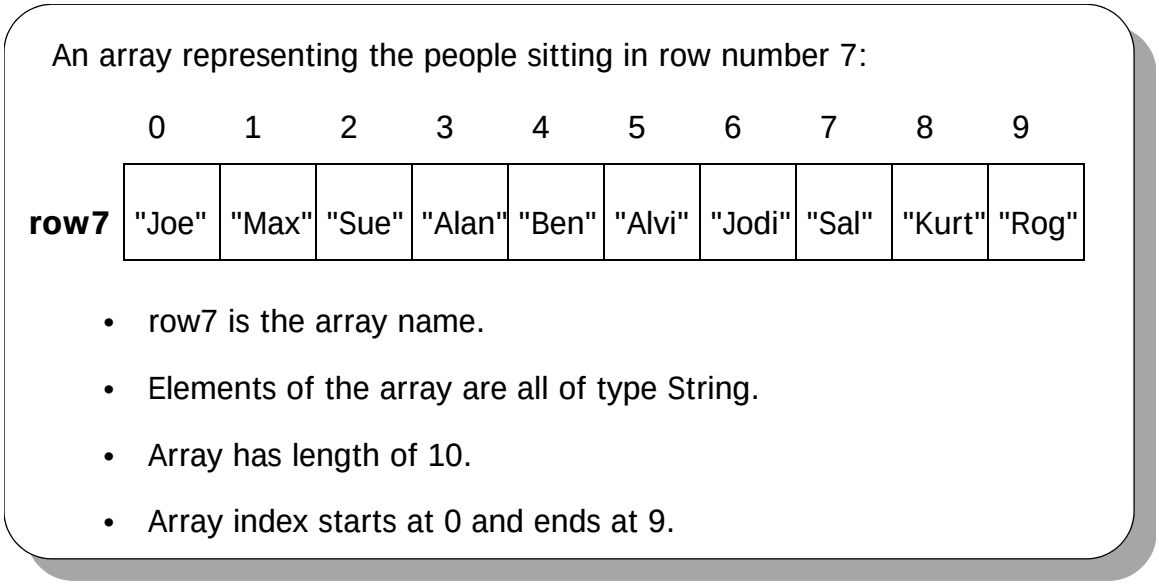


Figure 7-3: The Row of a Movie Theater as an Array

Other element types like Integer, Double, Boolean and types that we have not yet discussed can also be stored in an array. However, an array can only store a set of data of the same type, e.g. an array can either store strings or integers but not a mix of the two data types.

Just like a variable, an array has a name. With this name you refer to the whole set of data items. An index to the array allows each element of the array to be accessed individually. A common mistake in programming is to try to access the first element of the array using an index of 1. Remembering that an array starts at index 0 is not intuitive and takes some getting use to.

Now that we have seen our first array, what can we do with it? In most cases you store data in an array so that you can retrieve it at some later point in a program. The array acts a little like a storage container. It lives as long as the program runs and has to be created anew every time we run the program.

One of the movie attendants might want to address the person in seat number 6 (for our purposes lets assume that the movie theatre starts counting seats at 0 just like programmers do). Without your program, they wouldn't know the name of the person in that seat, but now they can look it up to address the person by name. They could now say for example "Jodi in row 7, seat 6, please stop throwing popcorn at the other people."

Apart from accessing individual items, you may also want to access all array elements, change one of the elements stored, or search for a specific item in the array. Figure 7-4 summarizes the processes that are most likely to be performed on an array.

Example Array Scenarios

To help you better understand the different ways in which arrays can be used, we present a number of real life scenarios that use arrays. Figure 7-20 provides a summary of the examples presented in this section.

1. Representing Rows of People in a Theatre.
2. Weekday Translation - array values known at compile time.
3. Summing Up an Array of Prices - summing array elements.
4. Finding Lowest Price in an Array of Prices - comparing array elements.
5. Storing Student Info - storing values in an array.

Figure 7-20: Example Array Scenarios

SCENARIO 1: REPRESENTING ROWS OF PEOPLE IN A THEATER

Although we have already looked at the movie theater row example, we include it here to show the complete development process of the program, from problem description to final solution.

Problem Description: You need to write a program that handles the information about people sitting on the seats in a row in the movie theater. You want to identify the people sitting on the seats by their names. As we are dealing with a set of data of the same type the array seems to be an appropriate means. Every row in the theater can be represented by an array. Every element of the array represents one of the seats. The program stores the data (names) of the people sitting in the row and prints out who sits where.

Let us look at the flowchart in Figure 7-21 first, then the pseudocode.

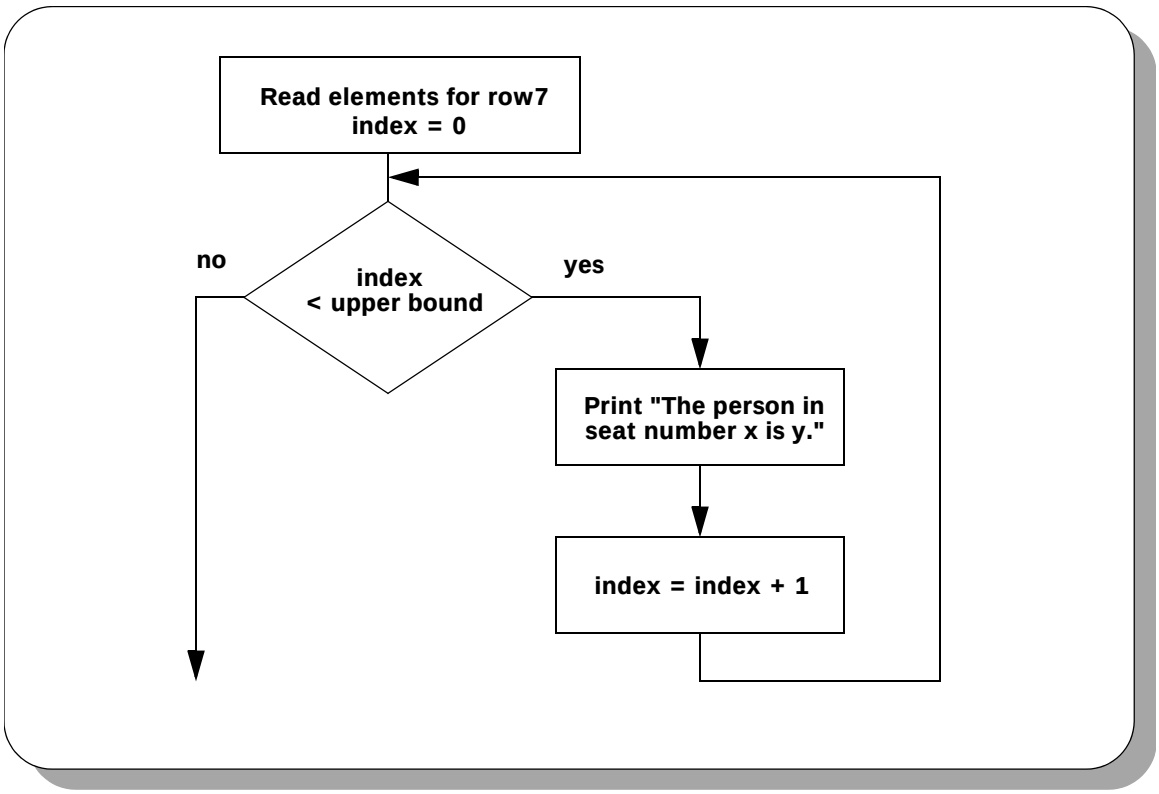


Figure 7-21: Flowchart - TheaterRow Program

The pseudocode for such a program could be expressed as follows:

```
TheaterRow
   Fill array row with elements
   index = 0
   WHILE index < upper bound DO
      Print "The person in seat number x is y."
      index = index + 1
   ENDWHILE
END
```

In Figure 7-22 we show the code for the TheaterRow program.

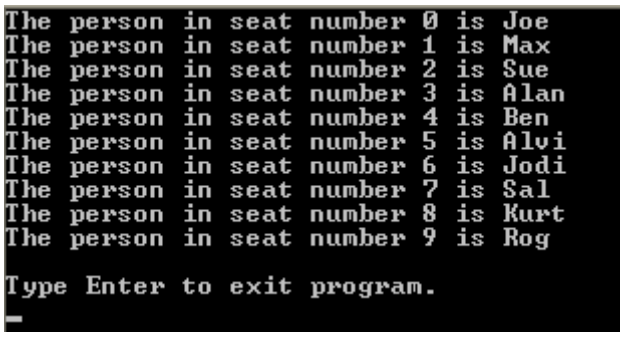
```
1  Module TheaterRow
2      Sub Main()
3        Dim row7() As String = {"Joe", "Max", "Sue","Alan", _
4                    "Ben", "Alvi", "Jodi", "Sal", "Kurt", "Rog"}
5
6        For index As Integer = 0 To row7.GetUpperBound(0)
7           Console.WriteLine("The person in seat number " _
8                             & index & " is " & row7(index))
9        Next index
10
11       ' Wait for user to exit.
12       Console.WriteLine()
13       Console.WriteLine("Type Enter to exit program.")
14       Console.ReadLine()
15     End Sub
16 End Module
```

Figure 7-22: TheaterRow Program

At lines 3 and 4 of the program code, we store the data in an array of strings with the name `row7`.

The For loop at lines 6 to 9 goes through the array element by element and prints out the current index (i.e. the seat number) and the contents (i.e the name of the person) at that index.

Compile and run the program Unit07\Examples\TheaterRow\TheaterRow.vbproj. The program output is shown in the screen capture below:



SCENARIO 2: WEEKDAY TRANSLATION PROGRAM

Problem Description: A program is needed that provides the names of the days of the week in different languages. It stores the days of the week in an array, each day is represented by one array element. For each language we have one array. You can choose in which language the program should print out the days of the week. The program checks your selection and prints out the days in that language.

9 Introduction to Object Oriented Programming and UML

Introduction

In this course, we have been examining, writing, compiling and executing Visual Basic .NET programs written in a procedural style. Within each program, we focused on a sequential set of steps from beginning to end, sometimes breaking some of the steps out into a method. Visual Basic .NET however is an object-oriented (OO) programming language, designed to be structured in an object-oriented rather than a purely procedural approach.

In this unit, we introduce you to several key concepts of OO programming, like encapsulation, inheritance and interfaces. The goal of this unit is for you to understand OO concepts, *not* to understand how to write an OO program on your own. That is left for later units.

Further, the pieces of an OO program can be represented graphically using a notation called the Unified Modeling Language (UML). So, when we present the OO concepts, we will use the UML notation. At the end of this unit, you should be able to read a simple UML diagram and explain what the UML symbols stand for in the context of OO programming.

Figure 9-1 lists the objectives of this unit.

At the end of this unit, you will be able to:

1. Explain differences between a class and an object.
2. Define attributes and methods of a class, given a short description of the class.
3. Explain encapsulation.
4. Explain inheritance.
5. Explain abstract classes and interfaces.
6. Read a simple UML diagram.
7. Use UML notation to represent a class and to express relationships between classes.

Figure 9-1: Unit Objectives

What is Object Oriented Programming?

In Unit 6, we examined a program written using the Visual Basic .NET programming language that helped with the company payroll. This program took a procedural approach and used methods to organize some of the code. Even though the program was written with an object-oriented programming language, it didn't use object-oriented programming techniques.

Object-oriented programming techniques allow you to modularize your program using classes. The use of classes allows you to have smaller, more maintainable units of code. Further, within your program, the classes (for which you write the code) are used to create objects (for which you provide names). The objects interact with each other using messages. The interacting objects achieve the functionality of your program.

A class defines a template for all objects created from it. You write the code that defines the class; you don't write code for an object. You use the Visual Basic .NET language to create an object from the class; the object is always given a name. The CLR, which we studied in Unit 2, creates the object using the class as a template and associates it with the name you specified.

Figure 9-2 illustrates these points with a diagram. On the left-hand side, we see a rectangle that represents our procedural program named CompanyPayroll. It consists of Visual Basic .NET statements that may be organized functionally using methods. In comparison, on the right-hand side we see a rectangle that represents an object-oriented version of the procedural program; it is called `OOPayrollApp`. This object-oriented program contains statements that are organized into two classes and other statements that use these classes.

The two classes that make up our `OOPayrollApp` program are called `OOPayrollApp` and `Employee`. In our OO application, we use `OOPayrollApp` and `Employee` to create two objects called `payroll` and `employee` respectively. The `payroll` and `employee` objects interact in the program to produce the output you would have seen with the CompanyPayroll program. The `payroll` object interacts with the `employee` object by calling its methods `ComputeRaise`, `ComputeNewSalary`, `ComputeBonus`, `PrintEmployeeInfo`. You may remember some of these methods from our procedural payroll program. Note that procedural programming is still important despite the fact that we are using OO techniques. We will be using the procedural approach to implement the methods of each class.

A class is a stand-alone unit of code. It can be compiled and small test programs can be written to interact with the class to test its methods. Now imagine that you have a large application. By being able to design the application as a set of classes, you are able to have smaller, testable units of code. This way when an error occurs in your unit of code, there is a smaller amount of code to debug.

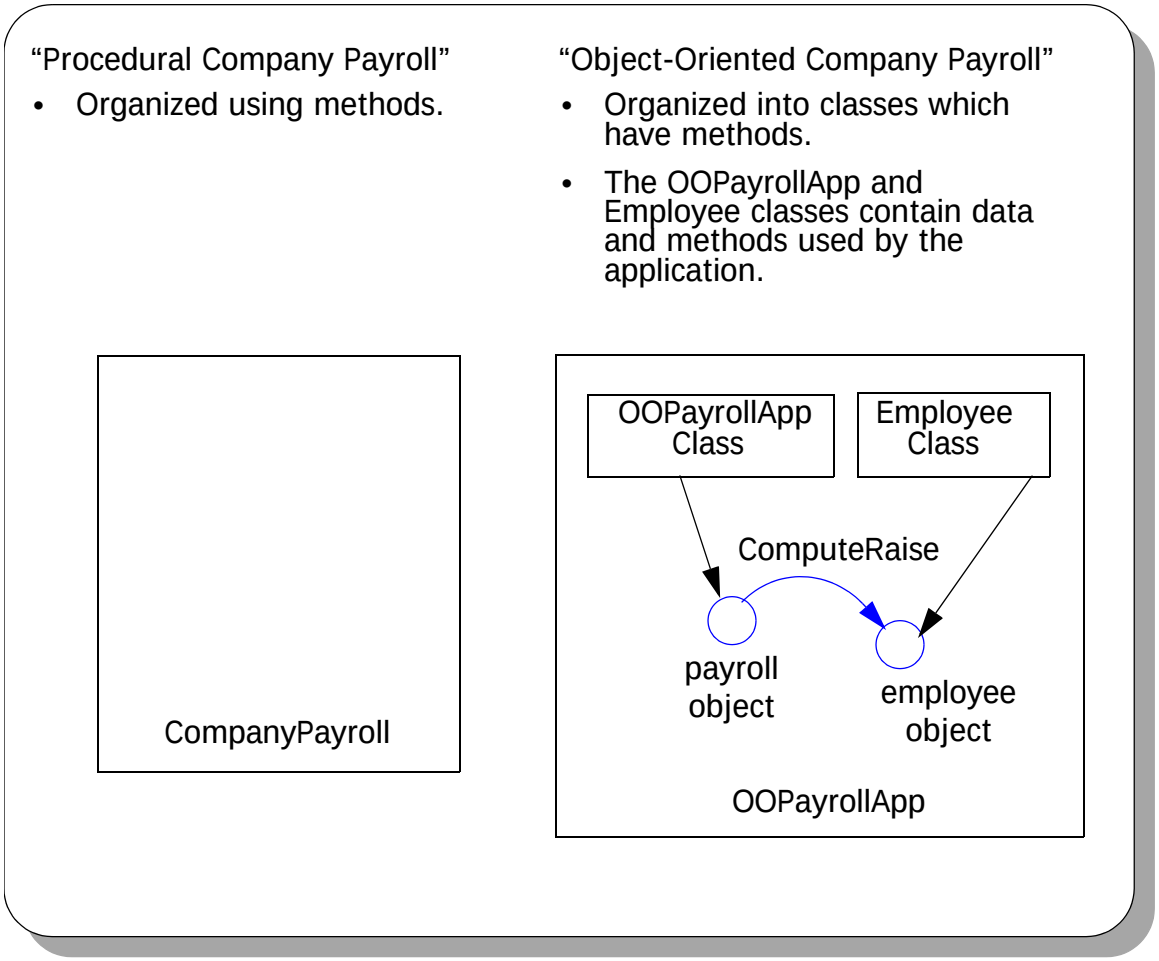


Figure 9-2: Procedural vs. Object-Oriented

Classes

Classes and objects are at the heart of object-oriented programming. Recall from previous discussion that a class serves as a template for creating an object.

An important characteristic of a class and thus an object is that it "encapsulates" data and behavior. A class defines a set of attributes and operations. Each object created from the class will have a value for each attribute defined in the class and will be capable of performing the operations defined by the class. The object encapsulates a specific set of data and the operations that can be performed on that data. The object and class provide the means to treat the data and operations as a unit, which helps when building, testing and maintaining the code.

Further, many classes are based on the real world entities that they represent. Hence your program can model real world things that interact, thus making it easier to design and understand. We'll see this shortly.

If you consider a company's payroll, you might think of payroll and employees as real world things with which you interact during the payroll process. This real world view can be incorporated into your programming using OO techniques. There are techniques that may be used as a part of an object-oriented analysis and design process which help you decide on and define the classes and objects to use in a programming application. This type of process, which helps you to figure out how to use OO programming design and coding techniques to your best advantage, is beyond the scope of this course.

In the previous figure, we showed a `payroll` object created from the `OOPayrollApp` class and an `employee` object created from the `Employee` class. The `payroll` object interacts with the `employee` object by calling the employee's `ComputeRaise` method. The `payroll` object contains all the attributes and operations of its template, the `OOPayrollApp` class. Likewise, the `employee` object contains all the attributes and operations of its template, the `Employee` class.

These points are summarized in Figure 9-3.

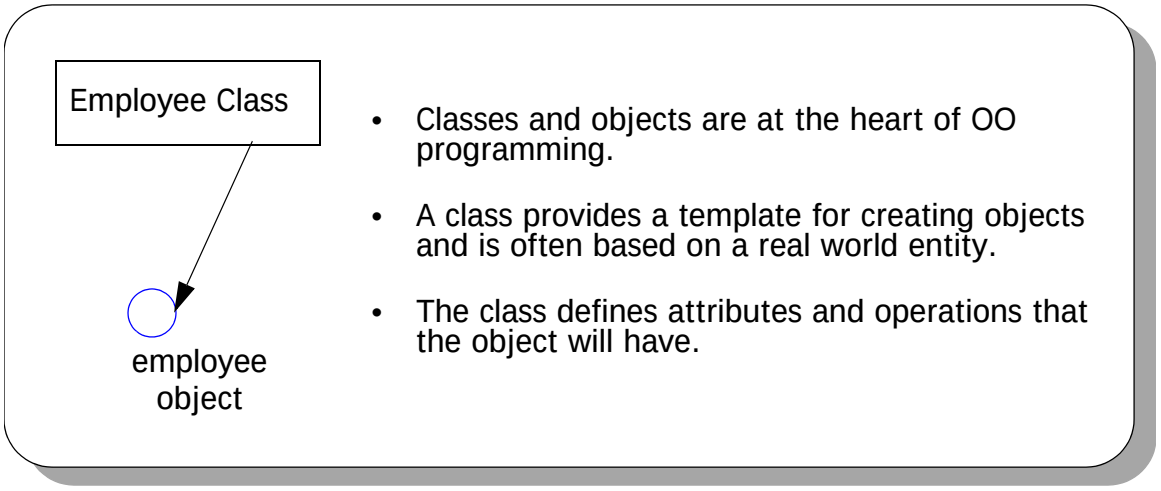


Figure 9-3: Classes and Objects

Defining a Class

What are the attributes and operations of a class, and how do we define them? The first thing we need to understand is how to define a class. We'll look at how classes are represented using UML diagrams and then see how they are defined using Visual Basic .NET code.

Figure 9-4 shows a UML diagram that represents a Visual Basic .NET class called `Employee`.

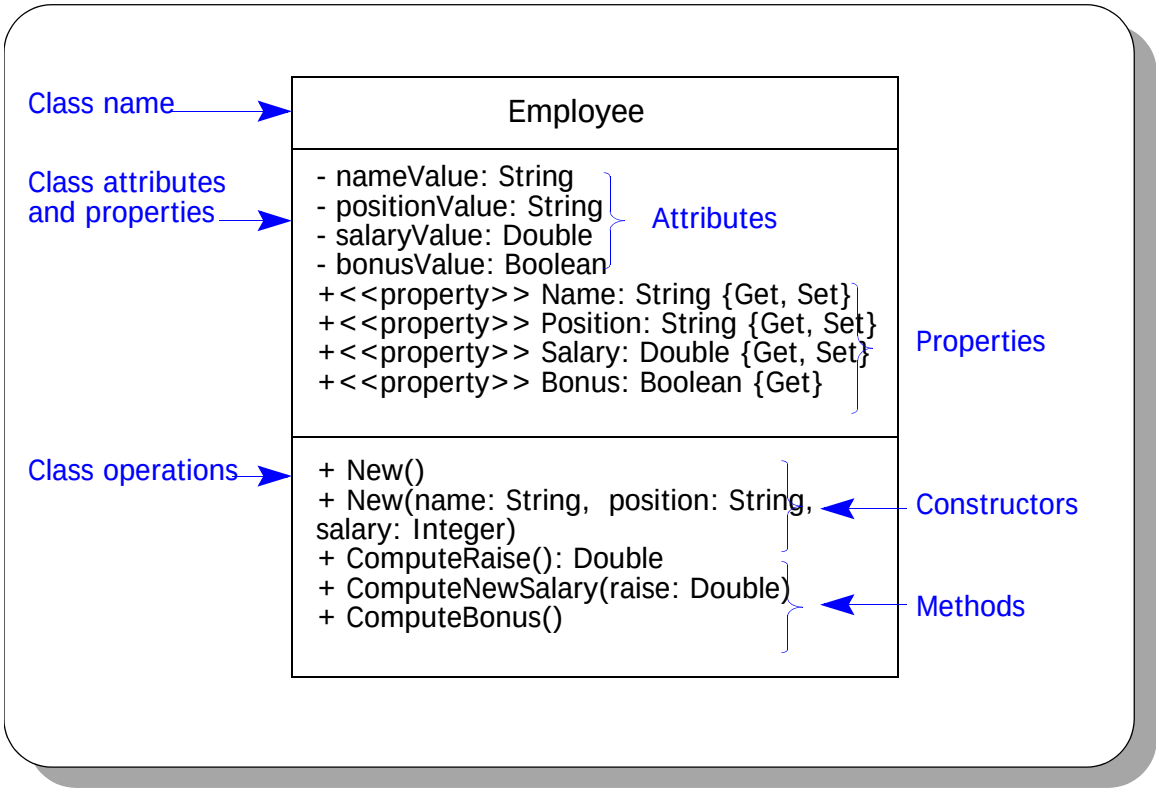


Figure 9-4: The Employee Class in UML Notation - with Properties

The class name is shown in the first compartment. Notice that it begins with a capital letter. This is the convention for naming a class in Visual Basic .NET. Although starting a class name with a capital letter is not enforced by the Visual Basic .NET compiler, it is standard coding practice that should be followed.

The second compartment contains the class attributes which, in this case, are `nameValue`, `positionValue`, `salaryValue` and `bonusValue`. Each attribute name is followed by a colon and then its type. For example, "nameValue: String" means that the attribute `nameValue` is of type `String`. Notice that each attribute name is preceded by a minus sign meaning that it is private to the class. This means that another class interacting with the `Employee` class cannot access these attributes directly. Private attributes must be accessed through **accessors** which, depending on the programming language used, are either public methods or methods of public properties that enable a private attribute to be set or retrieved. Visual Basic .NET treats accessors as methods of public properties.

11 Developing Classes

Introduction

Classes and objects are central to the development of real-world applications. They are the "building blocks" of Visual Basic .NET applications; the "blue print" from which objects are created. Objects provide the only means for encapsulating the data and behavior of an entity.

So far, we have covered how to create and use objects from *predefined* or *supplied* classes. Now we will cover how to define our own classes and use them to create objects within an application.

Figure 11-1 lists the objectives of this unit.

At the end of the unit, you will be able to:

1. Understand how to develop classes in Visual Basic .NET.
2. Create a class step-by-step.
3. Create constructors.
4. Employ inheritance.
5. Understand basic polymorphism.
6. Understand access modifiers.

Figure 11-1: Unit Objectives

Steps to Create a Class

Objects have two characteristics: the information they store (attributes/properties) and the operations that they perform (methods). Before you create a class, it is important to know in detail what information you want your objects to store and what actions they should perform.

Figure 11-2 outlines three steps to be taken in creating a class. We will walk through these steps in detail in the following sections, providing an example at each step and also an exercise so that you can practice these steps.

1. **Name the Class.**

   The name of the class is the type name used to declare and create objects of the class. E.g., if we are creating bank account objects, then a good name for the class is BankAccount.

2. **Determine and declare the Attributes and Properties** of the desired objects.

   For bank account objects, we might be interested in the account holder's name, address and current balance.

3. **Determine, define, and implement the Methods** that operate on the objects.

   For bank account objects, we might be interested in methods for performing deposits and withdrawing funds.

Figure 11-2: Steps to Create a Class

CASE STUDY: A BANKING APPLICATION

For illustrative purposes, the remainder of this unit will focus on the incremental development of a simple banking application.

In our simple banking application, a bank teller will be able to open an account of one of two types associated with a customer. For each account, the bank teller will be able to deposit money and withdraw money. The two accounts are a standard/generic account and a money market account. In our application, we will develop the following classes that will provide the functionality we need for our banking application:

- A class representing the *customer* of the bank who will be associated with a bank account. This class will let us set and get the name of the customer as well as their address and age.
- A class representing a *generic bank account*. This class will provide the functionality to create a new account for a specified customer, deposit into the account and withdraw from the account.
- A class representing a special *money market bank account*. This class will provide the same functionality as the bank account, however, the withdrawal amount cannot be less than $1000 and there will be additional functionality to calculate the interest on the account and add it to the balance.

12 Data Structures

Introduction

The purpose of this unit is to introduce the student to classic programming data structures. We explore the concept of a stack, queue, linked list and binary tree and cover the implementation of each data structure. We also discuss some of the commonly used data structures that are implemented in the Visual Basic .NET programming language. The student will become familiar with the use of Visual Basic .NET's `ArrayList` and `Hashtable` classes.

Figure 12-1 lists the objectives of this unit.

At the end of the unit, you will be able to:

1. Understand the concept of a data structure.
2. Understand stacks and queues.
3. Understand linked lists.
4. Understand binary trees.
5. Understand how to use Visual Basic .NET's ArrayList and Hashtable classes.
6. Enumerate over the elements of a Visual Basic .NET data structure.

Figure 12-1: Unit Objectives

What are Data Structures?

A data structure is a mechanism that provides a means for storing and manipulating a collection of data. There are many different ways in which data may be organized and operated on. Hence, there are many different types of data structures to represent these methods of organization. One data structure that you should already be familiar with is the array, which we covered in Unit 8. Other classic programming data structures are the stack, queue, linked list and binary tree. We will be studying these in the following sections. We will see how each structure stores data, it's behavior and then corresponding implementation.

In programming today, the construction of classic data structures has become less important for application developers. The reason is that programming languages such as Visual Basic .NET provide most of the data structures that a developer might need. Later in the unit, we will explore some of the commonly used data structures that are available in the Visual Basic .NET Class Library.

See Figure 12-2 for the main points about data structures.

Data structures:

- In general terms, a data structure is a mechanism that provides a means for storing and manipulating a collection of data (items).
- In Visual Basic .NET terms, a data structure is typically an object that manages the storage of some collection of objects (items).
- As there are many different ways in which data may be organized and operated upon, correspondingly there are many different data structures to represent such methods of organization.
- Examples: array, linked list, stack, queue, binary tree, hashtable

Figure 12-2: What are Data Structures?

Stacks

Stacks can be thought of as a linear data structure whose items are stacked one on top of the other. The top of the stack is its single point of access; new items are always inserted at the top of the stack and only the most recently inserted item can be accessed. For this reason, a stack is said to have **Last-In-First-Out (LIFO)** ordering.

Figure 12-3 shows a diagram of a stack that contains integers.

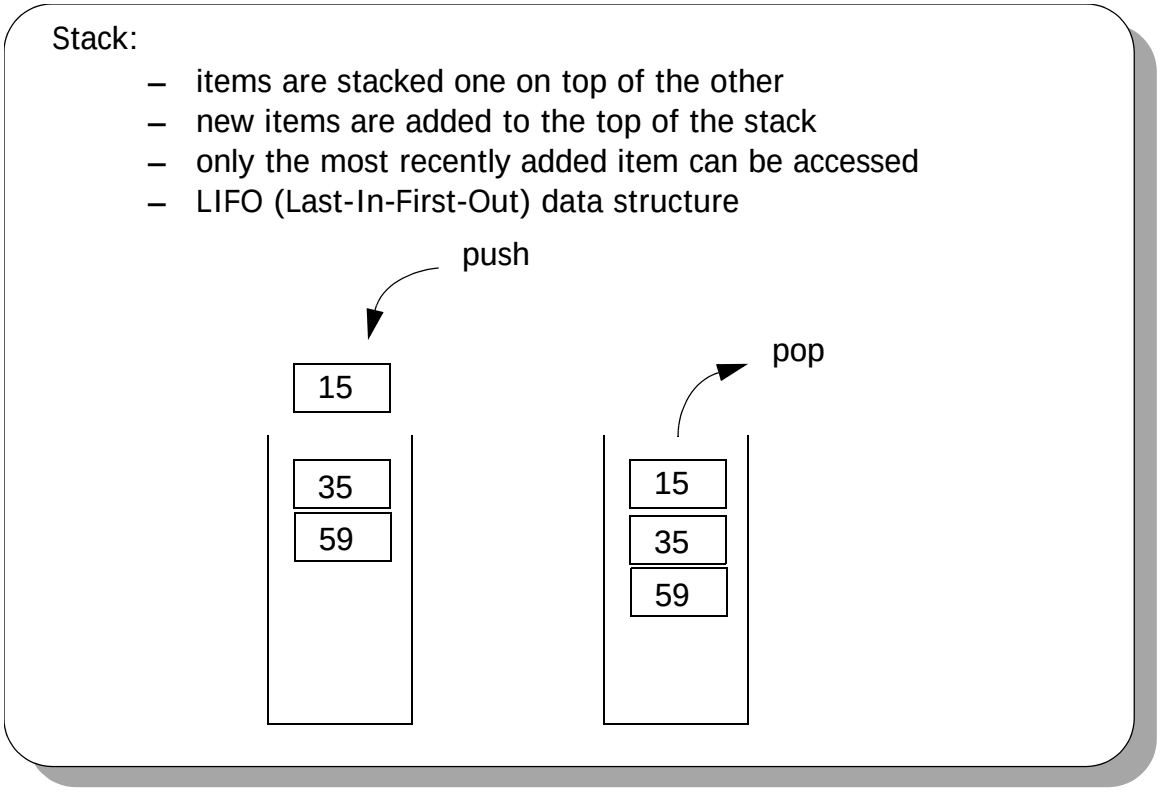


Figure 12-3: The Stack Data Structure

A good example of the way a stack works is the stack of plates that you might find in a buffet restaurant. When a plate is needed, a customer will take the top-most plate from the stack. When new clean plates are available, a waiter or waitress will add these to the top of the stack.

The stack data structure has many uses in programming. For example, it is used for evaluating expressions, keeping track of method calls and passing parameters to a method.

STACK OPERATIONS

The standard operations that are associated with a stack are listed below:

- `Push` - inserts an item onto the top of the stack
- `Pop` - removes the top-most item from the stack
- `Peek` (sometimes called `top`) - retrieves the top element without removing it
- `IsEmpty` - checks if the stack is empty

The following are other operations which are sometimes associated with a stack:

- `IsFull` - checks if the stack is full
- `Size` - returns the number of elements in the stack

IMPLEMENTING A STACK USING AN ARRAY

There are a number of ways in which a stack can be implemented. In this section, we look at an array-based implementation in which all items in the stack are kept in an array.

In our implementation, we define a class called `StackArray` which stores items of type `String`. Figure 12-4 shows the instance variables and constructors of our `StackArray` class. The class is contained in the file Unit12\Examples\StackArray\StackArray.vb.

```
1  Public Class StackArray
2     Private Dim MAXSIZE As Integer = 100
3     Private Dim stack() As String
4     Private Dim top As Integer = 0
5
6     Public Sub New()
7     ' No argument constructor.
8        stack = New String(MAXSIZE) {}
9     End Sub
10
11    Public Sub New(ByVal size As Integer)
12    ' Constructor that takes size of stack as input.
13       MAXSIZE = size
14       stack = New String(MAXSIZE) {}
15    End Sub
16    ...
17 End Class
```

Figure 12-4: StackArray.vb - Instance Variables and Constructors

A `String` array called `stack` is declared in line 3 to represent the stack. At line 4, an integer variable `top` is declared to keep track of the top of the stack. This value of `top` is always the array index where the next new item is to be added.

Binary Tree

A data structure that is commonly used in programming is a **tree**. The tree data structure, like a linked list, is made up of nodes. Tree nodes are organized in a hierarchical structure, similar to that of a family tree.

The node at the top of the hierarchy is known as the **root node**. Each node of a tree must have two or more pointers. A pointer of a tree node is also known as a **branch**. The node that a branch leads to is known as a **child node**. A tree node that has children is called a **parent node**. A tree node that has no children is called a **leaf node**.

There are different types of tree data structures. The type of a tree is determined by the number of pointers that belong to each node of the tree. The simplest and most commonly used type of tree is one whose nodes have two pointers. Such a tree is known as a **binary tree**. Binary trees are good for storing data that needs to be ordered. They are also good for efficient searching, inserting and deleting of items. You will see why this is so later in the section.

Figure 12-34 shows a diagram of a binary tree.

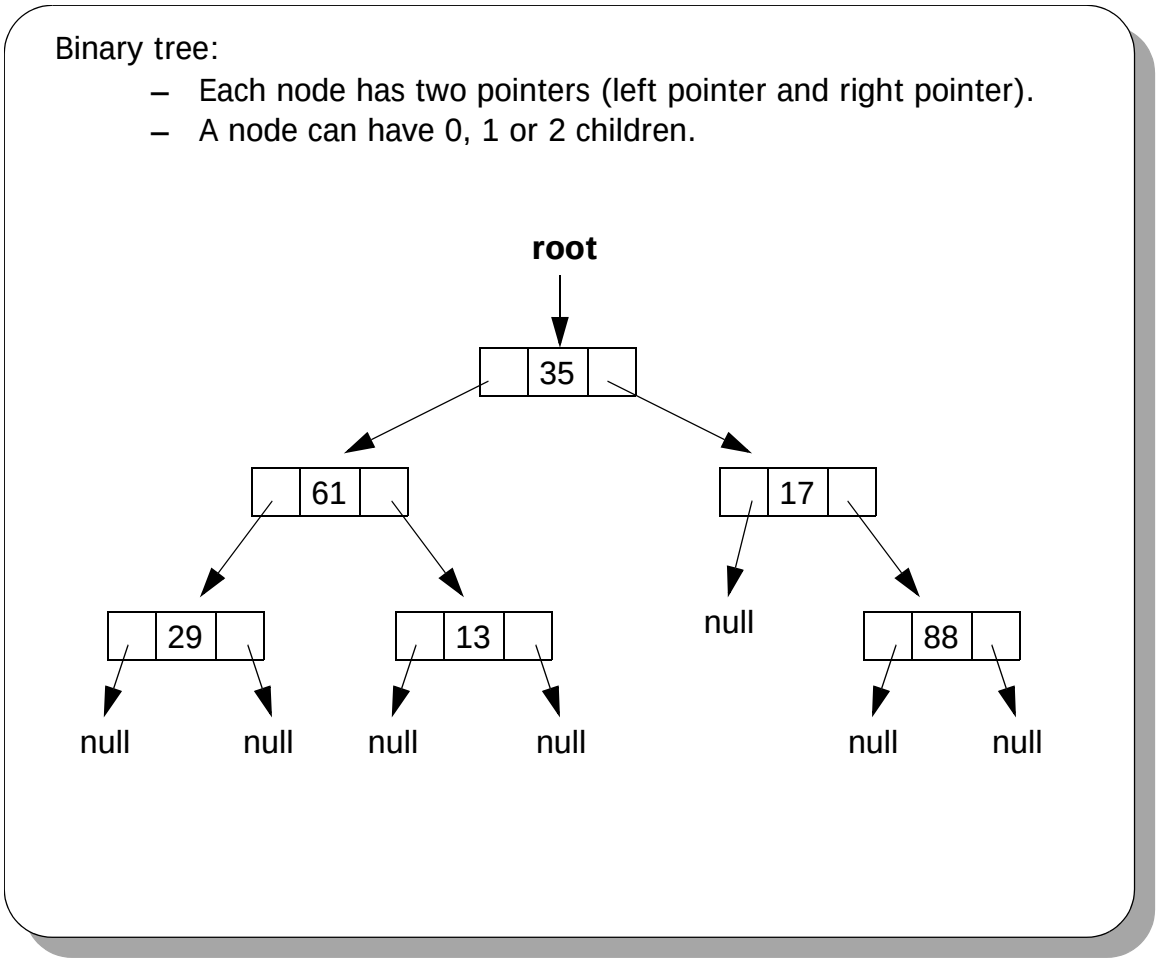


Figure 12-34: Binary Tree

The binary tree in the figure above is accessed via a pointer to the root node (35). This pointer is typically called the **root pointer**. Each node of the binary tree consists of three parts: a data element (that stores positive integer values), a left pointer and a right pointer. A pointer can either reference another node or it can be a null reference. Hence, a node can have 0, 1 or 2 children.

The binary tree shown in Figure 12-34 is an example of an unordered tree. In such a tree, the order in which items appear in the tree is of no importance. A search for an item in an unordered tree can be slow since the item could be located anywhere in the tree. To improve the efficiency of the search we need to use a **binary search tree**.

BINARY SEARCH TREE

A binary search tree is a special case of a binary tree that places significance on the ordering of items. The data value in a parent node is always greater than all data values in the nodes of its left subtree and always less than all data values of nodes in its right subtree.

An example of a binary search tree is shown in Figure 12-35.

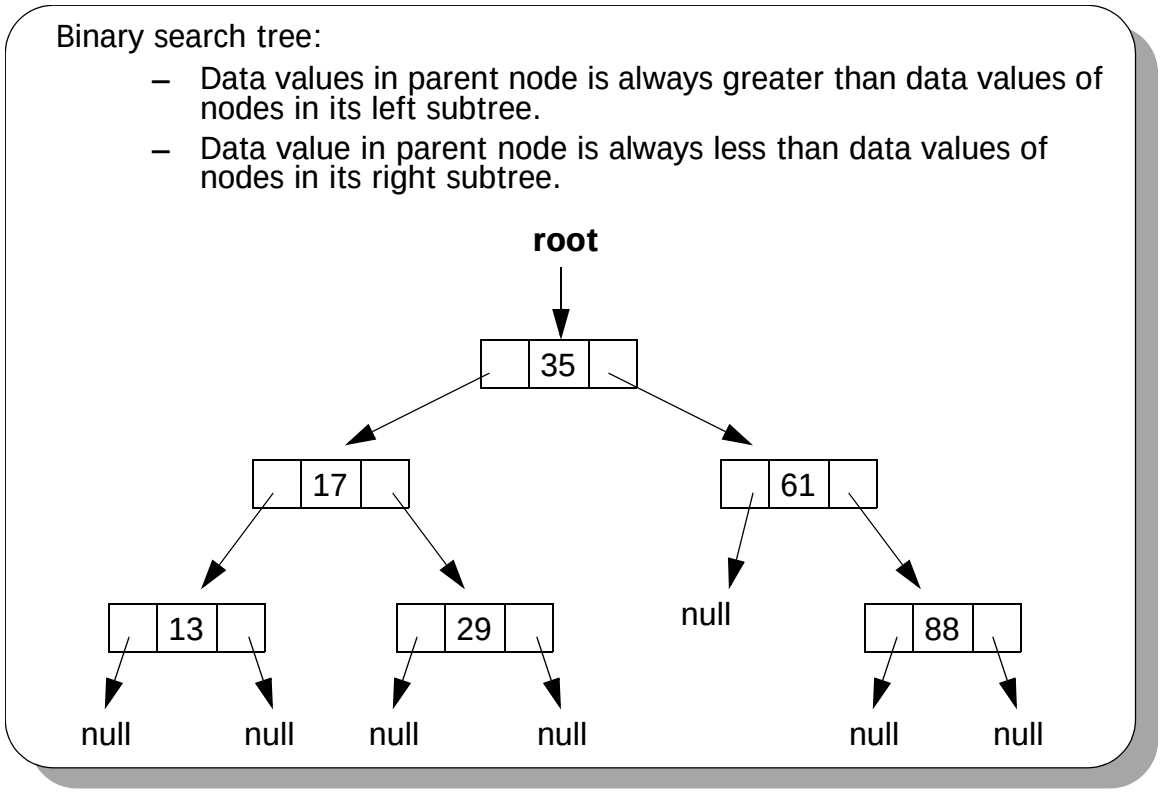


Figure 12-35: Binary Search Tree

In the binary search tree diagram above, the data value of the root node is 35. Notice that all items in the left subtree of the root node are less than 35 and all items in the right subtree are greater than 35. The same condition also features within subtrees. The root node of the left subtree is 17. All items to the left of this node are less than 17 and all items to the right of it are greater.

The ArrayList Class

We discussed one of the most frequently used data structures, the array, earlier in Unit 8. Recall that arrays are **static data structures** that are built into the Visual Basic .NET language. Arrays are useful for storing data when the number of data elements that we want to store is known. However, there are situations where this number cannot be determined.

To get around this problem, Visual Basic .NET provides a number of classes that represent **dynamic data structures**. One of these is a class called `ArrayList` which is a resizable array-like data structure. This class is found in the `System.Collections` namespace.

Like arrays, items in an `ArrayList` are stored in a linear sequence and are accessed by a numerical index. In fact, the underlying implementation of `ArrayList` is an `Object` array. Unlike arrays, the size of an `ArrayList` object will automatically expand or shrink as items are added or removed.

All elements of an `ArrayList` must be of type `Object`. Recall from Unit 10 that in Visual Basic .NET, all class instances inherit from the `Object` class. Therefore any class instance can be added to an `ArrayList` because it will be implicitly up-cast to the type `Object`. Numerical types like `Integer` and `Double`, also inherit from the `Object` class and hence can be added to an `ArrayList`.

An `ArrayList` is said to be a **polymorphic data structure** because it is able to store elements of different data types.

Figure 12-62 lists the features of an `ArrayList`.

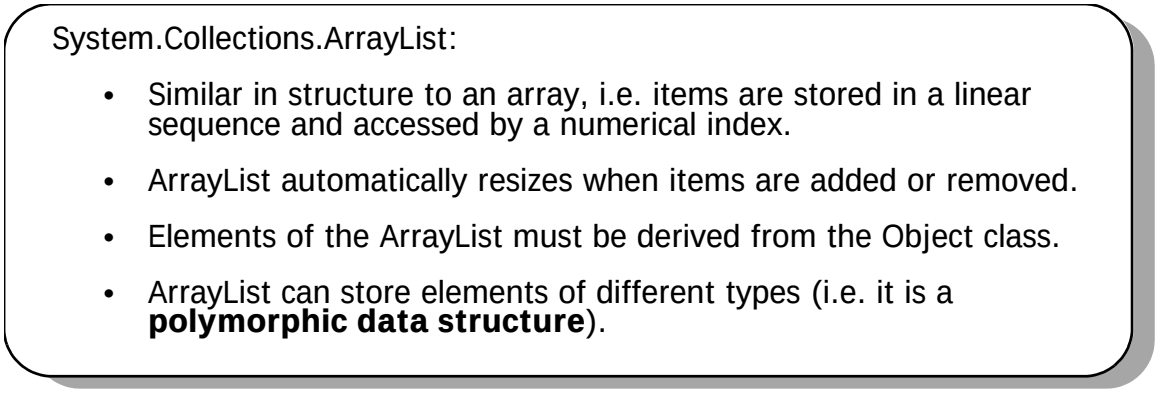


Figure 12-62: The ArrayList class

15 The First Bank of VB GUI: A Case Study

Introduction

So far in this course, we have used simple GUI and console applications to introduce you to the features of the Visual Basic .NET IDE and programming language. In this unit, we will study a more complex application that is closer to the applications you would find in the real world. Using the knowledge of programming concepts gained in earlier units, you will be developing parts of a GUI application for an imaginary bank called The First Bank of VB.

This is a very simplistic banking application, intended for use by bank tellers at the First Bank of VB. The application is limited to creating new customer records, creating new accounts for customers, searching for existing records of customers and accounts, depositing money into an account and withdrawing money from an account. You will be responsible for assembling the controls that make up three of the GUI screens in the application, as well as handling the events associated with each of the screens.

A problem specification, commonly referred to as "requirements", for the First Bank of VB GUI will be provided. We also provide descriptions of the screens that make up the GUI, in particular the controls in each screen and the events that are handled. We describe both the screens that you will be developing as well as other screens of the GUI that have already been implemented.

Figure 15-1 lists the objectives of this unit.

At the end of this unit, you will be able to:

1. Create more complex, real-world type applications using the knowledge of programming concepts gained in earlier units.
2. Create a GUI with multiple forms.
3. Add images to a form.
4. Add a menu bar to a form.
5. Use panels in a form.

Figure 15-1: Unit Objectives

First Bank of VB GUI Requirements: General

The first section of the First Bank of VB GUI requirements is a general description of the software that is needed by the bank. These are as follows:

- The GUI should enable the management of customers and accounts.
  - *The bank teller must be able to create a new customer.*
  - *The bank teller must be able to create a new account for a customer.*
  - *The bank teller must be able to search for existing customer records.*
  - *The bank teller must be able to search for existing account records.*
- The GUI should enable deposits and withdrawals from an account.
  - *The bank teller must be able to deposit money into an account.*
  - *The bank teller must be able to withdraw money from an account.*

Your assignment is to develop several parts of the GUI application, in particular:

1. Develop the login screen.
2. Develop the main screen which controls the GUI navigation.
3. Develop the 'Create Customer' screen.
4. Implement the code to handle navigation between screens (i.e. handle button and pop-up menu item events).
5. Implement the code to handle deposits to and withdrawals from an account.

First Bank of VB GUI Requirements: Use Cases

The general requirements for an application are then typically converted, by a software engineer, into a UML (Unified Modeling Language) use case diagram to summarize the functionality that needs to be developed in the GUI application. The use case diagram for the portions of the GUI application that you will be developing is shown in Figure 15-2 below:

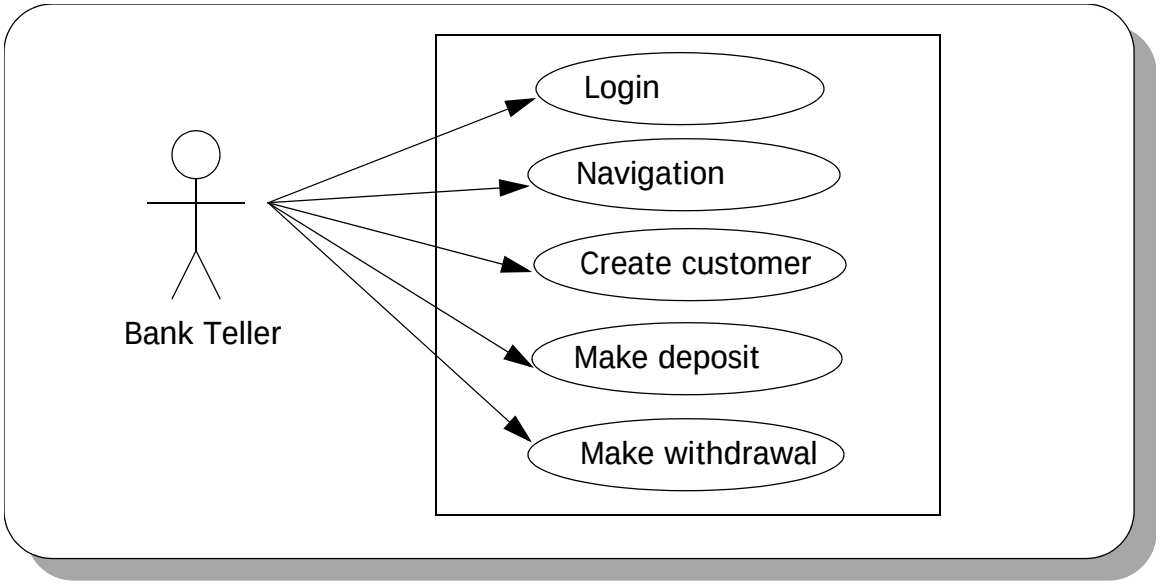


Figure 15-2: Use Cases for First Bank of VB GUI

The user of the system, represented by the stick figure in the diagram, is the bank teller. The box depicts the boundary of the system. The ovals depict functionality in the system. The ovals represent "use cases" and represent a course of events that the system must perform in response to a trigger from the actor, in this case the bank teller.

The following are the descriptions of the use cases which document the functional requirements for the parts of the application that you will be responsible for building:

Use Case: Login
Actor: Bank teller
Description: This use case begins when the teller starts up the application which causes the login screen to be displayed. The window prompts for the teller's user ID and password. The system then checks the validity of the user ID and password combination against the contents of a user ID/ password file. If valid, the system proceeds to the main window of the GUI. If invalid, an error dialog is displayed.

Use Case: Navigation
Actor: Bank teller
Description: This use case begins when the teller successfully logs into the First Bank of VB GUI. The main screen of the GUI is displayed. It is from this screen that the teller can navigate to other screens of the GUI. This is done via pull-down menus. From the main screen, the teller can terminate the program using the Exit menu item of the File menu. The teller can also visit the 'Create Customer', 'Create Account' and 'Search' screens by selecting the appropriate menu item in the Task menu.

Use Case: Create customer
Actor: Bank teller
Description: This use case begins when the teller selects the Create Customer option from the Task menu of the main window. The 'Create Customer' panel is displayed. The panel prompts for the title, name, address, address type, email address and promotional information preferences of the customer. If any of the entries are invalid (i.e. customer name, address and email have not been entered), an error dialog is displayed notifying the user of the error so that they can correct it. If all entries are valid, the system creates a new customer record and adds it to the system's storage. It then displays the customer details in the 'Customer Details' panel.

Use Case: Make deposit
Actor: Bank teller
Description: This use case begins when the teller enters a deposit amount in the 'Account Details' window and then clicks on the 'Deposit' button in the window. If the deposit amount entered is valid (i.e. is a valid decimal value), the account balance in the customer's account is increased by the deposit amount and the 'Account Details' window is updated with the new balance. If the entry is invalid then an error dialog is displayed.

Use Case: Make withdrawal
Actor: Bank teller
Description: This use case begins when the teller enters a withdrawal amount in the 'Account Details' window and then clicks on the 'Withdrawal' button in the window. If the withdrawal amount entered is valid (i.e. it is a numerical value and will not cause the account balance to be a negative value), the account balance in the customer's account is decreased by the withdrawal amount and the 'Account Details' window is updated with the new balance. If the entry is invalid then an error dialog is displayed.

First Bank of VB GUI Requirements: The User Interface

The user interface is to be a Visual Basic .NET GUI. The project requires that you create three screens of the GUI: Login, Main and 'Create Customer' screens. This will involve setting up the components within each screen and also handling the events that can occur in each screen. Let's take a look at each of the screens that you'll be creating as well as other screens of the application that have already been implemented.

THE LOGIN SCREEN

GUI applications that store sensitive information, like a someone's financial details, will typically have a login screen to restrict access to authorized users.

The login screen of the First Bank of VB GUI features two text boxes that allow a user to enter their user ID and password. Labels are used to instruct the user on what to do and also to identify the text boxes. To submit the user ID and password, the user clicks on a Login button. The Cancel button is used for cancelling the login and exiting the program.

Once completed, the screen should resemble the screenshot in Figure 15-3:

Figure 15-3: Login Screen for First Bank of VB GUI

If a click event on the Login button occurs, the relevant event handler needs to check the validity of the login. A method called `IsValidLogin` is supplied to do this. The method compares the user ID and password entered via the login screen to the user ID and passwords contained in the file "Unit15\<exercise folder>\Datafiles\users.txt". You can add your own userID/password combinations to the file. However, the userID/password combinations that you add must be in the format that is specified in the file.

If the login is valid, the `IsValidLogin` method returns a boolean value of true and the GUI will navigate to the main screen. If the login is invalid, the method returns false and a dialog box will pop-up with an appropriate error message. You do not need to know any more about how this method is implemented but if you are curious, you can take a look at the source code of the `LoginForm` class.

If a click event on the Cancel button occurs, the relevant event handler will terminate the application using the `End` statement.

Exercise 15-1: Creating the Login Screen

Purpose: Learn to create a GUI with multiple forms.

Background: This exercises involves adding a new Windows Form to a project in which a form already exists. You will be laying out components in the new form and setting form/control properties using the Visual Basic .NET IDE to create a Login screen for the First Bank of VB GUI. The form that already exists in the project is the partly implemented Main Application screen which will be completed in a later exercise.

The layout of the Login screen is shown below:

Instructions: Figure 15-12 details the steps required to complete the exercise.

1. Open Unit15\Exercises\FirstBankVB_Layout\FirstBankVB.vbproj.
2. Add a new Windows Form called LoginForm.vb to the project.
3. Set the LoginForm as the new Startup Object of the project.
4. Layout the GUI components according to the sketch of the GUI.
5. Set the properties of the GUI components according to the values listed in Table 12-1.
6. Lock the GUI when you are satisfied with the layout.
7. Save, compile and run the program. The Login screen should be displayed.
8. Exit the program using the Stop Debugging option from the Debug menu in the Visual Basic .NET IDE.

Figure 15-12: "Creating the Login Screen" Exercise

Note: Unit15\Exercises\Solutions\FirstBankVB_Layout\FirstBankVB.vbproj contains a solution.

Table 12-1: Form/Control Properties for Login Screen

Component Description	Property Name	Property Category	Value
Form	Text	Appearance	First Bank of VB Login
	StartPosition	Layout	CenterScreen
Mascot picture box	Image	Appearance	C:\ol300\Unit15\Images\BugEye.gif
	SizeMode	Behavior	CenterImage
	Name	Design	LoginPB
Instruction label	Text	Appearance	Enter user ID and password.
	Font[Bold]	Appearance	True
	Name	Design	InstructionL
User ID label	Text	Appearance	User ID:
	Font[Bold]	Appearance	True
	Name	Design	UserIDL
User ID textbox	Text	Appearance	<blank>
	Name	Design	UserIDTB
Password label	Text	Appearance	Password:
	Font[Bold]	Appearance	True
	Name	Design	PasswordL
Password textbox	Text	Appearance	<blank>
	PasswordChar	Behavior	*
	Name	Design	PasswordTB
Login button	Text	Appearance	Login
	Name	Design	LoginB
Cancel button	Text	Appearance	Cancel
	Name	Design	CancelB