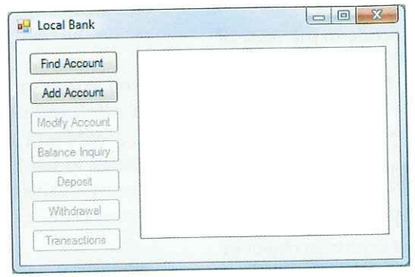
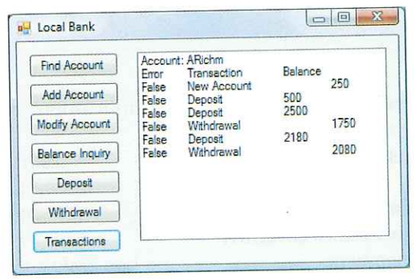
Exercises

Exercise 1 — LocalBank

Create a LocalBank application that is based on the AccountDemo reviews created in this chapter. LocalBank should use the Account class with some modifications. LocalBank should require that an account first be retrieved before any actions can be taken. Adding an account should also be allowed when the application starts. The first image below shows the application at start up. The second image below shows the application after adding at least one account, retrieving a valid account, and making transactions.





The Account class should:

- include the `Account` class, as mentioned above. However, there should not be a `DEFAULT_ACCT_NUM` member because every account number must be unique. The `Account` class should be modified to include just one constructor which accepts a first name, a last name, and an initial deposit. The constructor should set the account number to the first letter of the first name and the first 5 letters of the last name. The AcctNum property will need to be modified to be a read-only property.

The LocalBank Form1 code should:

- include two global declarations:

```
Dim activeAcct As Account        'current account to receive transactions
Dim accounts(-1) As Account      'array of accounts
```

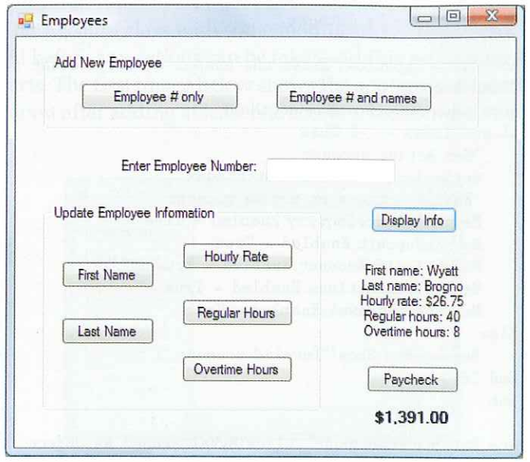
- have the btnModifyAccount_Click event procedure coded to change only the first and last names of the account holder.
- use the active account (`activeAcct`) when any of the other buttons on the interface are clicked.
- include a GetAcctNum() function that returns the index location in the array of the account with the indicated account number or –1 otherwise.

- include the following code for the btnFindAccount_Click and btnAddAccount_Click event procedures:

```
Private Sub btnFindAccount_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles btnFindAccount.Click
    Dim acctNum As String
    Dim acctIndex As Integer

    acctNum = InputBox("Enter the number of the account to retrieve:",
    "Find Account")
    acctIndex = GetAcctNum(acctNum)
    If acctIndex <> -1 Then
        'Set active account
        activeAcct = accounts(acctIndex)
        'Enable actions on active account
        Me.btnBalanceInquiry.Enabled = True
        Me.btnDeposit.Enabled = True
        Me.btnModifyAccount.Enabled = True
        Me.btnTransactions.Enabled = True
        Me.btnWithdrawal.Enabled = True
    Else
        MessageBox.Show("Invalid account.")
    End If
End Sub

Private Sub btnAddAccount_Click(ByVal sender As Object,
ByVal e As System.EventArgs) Handles btnAddAccount.Click
    Dim firstName, lastName As String
    Dim initDeposit As Double
    Dim newAcct As Account

    firstName = InputBox("Enter the account holder's first name:", "First Name")
    lastName = InputBox("Enter the account holder's last name:", "Last Name")
    initDeposit = InputBox("Enter the account holder's initial deposit:",
    "Initial Deposit")

    'Create new account and add to array of accounts
    newAcct = New Account(firstName, lastName, initDeposit)
    ReDim Preserve accounts(accounts.Length)  'increase array by 1
    accounts(accounts.Length - 1) = newAcct
End Sub
```

Exercise 2 — Employees

Create an Employees application that allows the user to enter data for as many employees as wanted and then displays information for a specific employee. The application interface should look similar to the following after adding several new employees and updating and displaying employee information:



Your program code should:

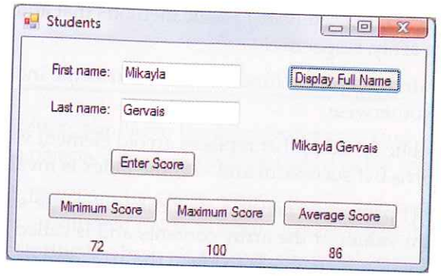
- include an Employee class, as described below.
- use a dynamic array of Employee objects to maintain a set of employees.
- display input boxes for user input.
- include a FindItemIndex() function that returns the index location in the array of the employee with the indicated employee number or –1 otherwise.
- display messages indicating "Not valid." if the employee number does not exist.
- display formatted output.

The Employee class should include the following members:

- an Employee data member that is type Double. PersonalInfo is a structure that contains employeeNum, firstName, and lastName string members.
- a Payroll data member that is type Double. PayrollInfo is a structure that contains hourlyRate, regularHours, and overtimeHours single members.
- properties that allow for getting and setting of each piece of Employee and Payroll data.
- a constructor that has a string parameter for the employee number to initialize EmployeeNum. The constructor should also include statements that initialize HourlyRate to 5 and RegularHours to 40.
- a constructor that accepts an employee number, first name, and last name string parameters to initialize EmployeeNum, FirstName, and LastName respectively. The constructor should also include statements that initialize HourlyRate to 5 and RegularHours to 40.
- Paycheck() public function that returns the paycheck amount. The paycheck amount is calculated by multiplying regular hours worked by hourly rate and adding the overtime hours multiplied by time and a half (hourly rate x 1.5).

Exercise 3 — Students

Create a Students application that allows the user to enter data for as many students as wanted and then display information for a specific student. The application interface should look similar to the following after entering a first and last name, clicking Display Full Name, clicking Enter Score several times to enter scores, and clicking each button to display information:

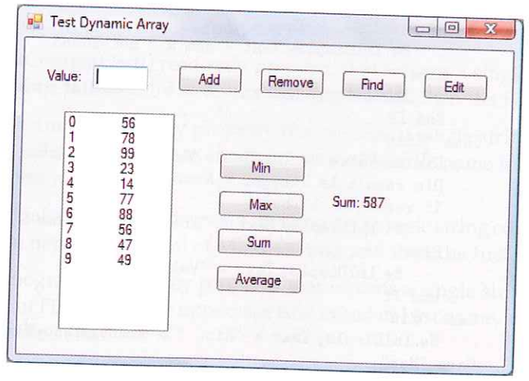


The Students application should contain a class named `Student` with the following members:

- FirstName and LastName properties.
- FullName read-only property for displaying a student's full name. Hint: Concatenate the first name and last name.
- Avg, Max, and Min read-only properties for getting the average, maximum, and minimum scores entered for the student.
- A NewScore() public method for adding a new score to the student's total score. The average, minimum, and maximum should be updated when a new score is added.

Exercise 4 — TestDynamicArray

Create a TestDynamicArray application that is similar to the DynamicArrayDemo application presented in Chapter 8. The application interface should look similar to the following after adding several values, removing values, and clicking Sum:



The TestDynamicArray application should contain a class named `DynamicArray` with the following members:

- a dynamic array data member with zero integer elements.
- Sum, Avg, Max, and Min read-only properties for getting the sum, average, max, and min of the numbers in the array.
- AddItem() and DeleteItem() public methods that add an item to or remove an item from the array, respectively.
- FindItemIndex() public function that returns the index of the item being searched for or –1 otherwise.
- Edit() public function that replaces an old element value with a new element value and returns 0 if successful and –1 if the index is invalid.
- GetStats() protected method that calculates the sum, average, maximum, and minimum values of the array contents and is called by AddItem, DeleteItem, and Edit every time a change is made to the array.
- DisplayContents() public method that displays the array contents in a list box.

The TestDynamicArray should contain the event procedure below:

```
Private Sub AButton _ Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles btnAdd.Click, btnRemove.Click, btnFind.Click,
btnEdit.Click, btnMax.Click, btnMin.Click, btnSum.Click, btnAverage.Click
    Static Numbers As New DynamicArray()
    Dim num As Integer
    Dim ButtonClicked As Button = sender
    If Me.txtValue.Text = Nothing And (ButtonClicked.Tag = "Add" Or
    ButtonClicked.Tag = "Remove" Or ButtonClicked.Tag = "Find" Or
    ButtonClicked.Tag = "Edit") Then
        MessageBox.Show("Type a value before selecting a button.")
    Else
        num = Val(Me.txtValue.Text)

        Select Case ButtonClicked.Tag
            Case "Add"
                Numbers.AddItem(num)
            Case "Remove"
                Numbers.DeleteItem(num)
            Case "Find"
                Dim result As Integer = Numbers.FindItemIndex(num)
                If result = -1 Then
                    Me.lblDisplay.Text = num & " not found."
                Else
                    Me.lblDisplay.Text = num & " found at index: " & result
                End If
            Case "Edit"
                Dim newValue As Integer = Val(InputBox("Enter new value "))
                Dim result As Integer = Numbers.Edit(num, newValue)
                If result = -1 Then
                    Me.lblDisplay.Text = "The value " & num & " not found."
                Else
                    Me.lblDisplay.Text = "Value changed."
                End If
            Case "Min"
                Me.lblDisplay.Text = "Min: " & Numbers.ShowMin
            Case "Max"
                Me.lblDisplay.Text = "Max: " & Numbers.ShowMax
            Case "Sum"
                Me.lblDisplay.Text = "Sum: " & Numbers.ShowSum
            Case "Average"
                Me.lblDisplay.Text = "Average: " & Numbers.ShowAverage
        End Select
        Numbers.DisplayContents(Me.lstOutput)
    End If
    Me.txtValue.Text = Nothing
    Me.txtValue.Select()
End Sub
```


Exercise 5 — MyPizzaShop

Create a MyPizzaShop application, similar to the PizzaOrder case study from Chapter 4. Use a `Pizza` class to instantiate new pizza objects. The `Pizza` class should have the following members:

- REGULAR_PRICE and LARGE_PRICE fields that store 6 and 10 respectively (for $6 and $10). Additional fields to store topping prices. Other data members will be needed for storing the base pizza price, the number of toppings, the toppings price, and the pizza size.
- A read-only PizzaPrice property for getting the price of the pizza and a Size property for getting and setting the size of the pizza.
- A constructor that takes no parameters and initializes the size to regular, the pizza price to the regular base price, the number of toppings to 0, and the toppings price to 0.
- A Public AddTopping() that increments the number of toppings and modifies the toppings price.

Exercise 6 — NameOptions

Create a NameOptions application that prompts the user for first, middle, and last names and provides several options for displaying the names in different ways. The project should include a `Name` class with the following members:

- a constructor with three parameters that initialize First, Middle, and Last data members.
- A FullName() read-only property that returns a single string containing First, Middle, and Last names separated by spaces.
- A LastCommaFirst() read-only property that returns a single string containing the Last name followed by a comma and a space and then the First name.
- A Signature() read-only property that returns a single string containing the First name followed by a space, the first initial of Middle name followed by a space, and then Last name.
- An Initials() read-only property that returns a single string containing the first initial of First name, first initial of Middle name, and then the first initial of Last name.
- A Monogram() read-only property that returns a single string containing the first initial of First name, the uppercase first initial of Last name, and then the first initial of Middle name.

Exercise 7 — MetricSystem

Create a MetricSystem application with options that demonstrates all the features of a `Metric` class, which should include:

- INCH_CENT_FACTOR, FEET_CENT_FACTOR, YARD_METER_FACTOR, and MILE_KILO_FACTOR field members that store the conversion factors 2.54, 30, .91, and 1.6, respectively.
- InchToCent() and CentToInch() public functions that return the unit of measurement from inches to centimeters or from centimeters to inches. Inches are converted to centimeters by multiplying the value in inches by the InchCentFactor conversion factor (2.54). Centimeters are converted to inches by dividing the value in centimeters by InchCentFactor.
- FeetToCents() and CentsToFeet() public functions that return the unit of measurement from feet to centimeters or from centimeters to feet. Feet is converted to centimeters by multiplying the value in feet by the FeetCentFactor conversion factor (30). Centimeters are converted to feet by dividing the value in centimeters by FeetCentFactor.
- YardsToMeters() and MetersToYards() public functions that return the unit of measurement from yards to meters or from meters to yards. Yards are converted to meters by multiplying the value in yards by the YardMeterFactor conversion factor (.91). Meters are converted to yards by dividing the value in meters by YardMeterFactor.
- MilesToKilos() and KilosToMiles() public functions that return the unit of measurement from miles to kilometers or from kilometers to miles. Miles are converted to kilometers by multiplying the value in miles by the MileKiloFactor conversion factor (1.6). Kilometers are converted to miles by dividing the value in kilometers by MileKiloFactor.