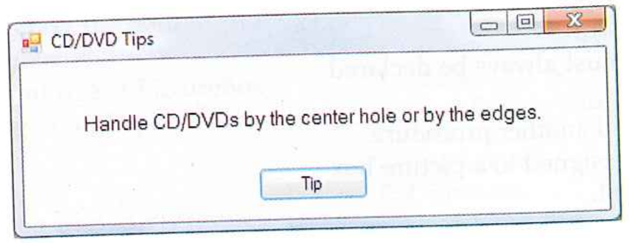
Exercises

Exercise 1 — CD-DVDTips

Create a CD-DVDTips application that displays one of the following messages when Tip is clicked:

Handle CD/DVDs by the center hole or by the edges.
Keep CD/DVDs away from extreme temperatures and moisture.
Store discs in a jewel case or sleeve to prevent scratches.

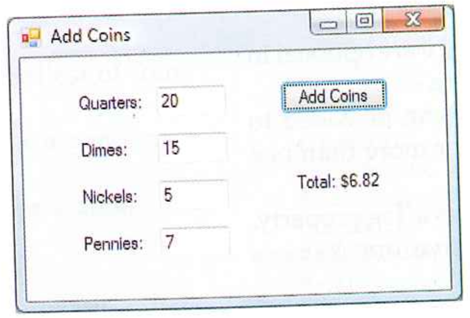
The program code should include a DisplayTip() procedure that uses RndInt() function from the TestRndIntFunction review in this chapter to randomly display one of the tips in a label. The application interface should look similar to the following after clicking Tip:



Handle CD/DVDs by the center hole or by the edges.

Exercise 2 — AddCoins

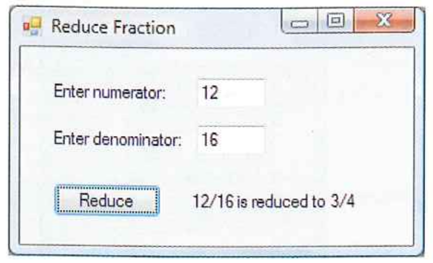
Create an AddCoins application that prompts the user for the number of quarters, dimes, nickels, and pennies and then displays the total dollar amount. The program code should include a TotalDollars() function with quarters, dimes, nickels, and pennies parameters. The application interface should look similar to:



Quarters: 20
Dimes: 15
Nickels: 5
Pennies: 7
Total: $6.82

Exercise 3 — ReduceFraction

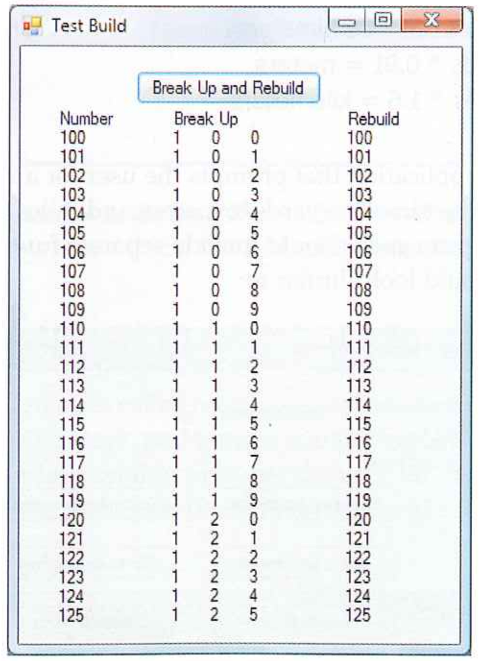
Create a ReduceFraction application that takes the integer numerator and denominator of a fraction and then displays the fraction reduced or a message stating the fraction cannot be reduced. A fraction may be reduced by finding the largest common factor and dividing both the numerator and denominator by this factor. The program code should include a Reduce() procedure with num and denom parameters that are changed, if possible, to the reduced values. The application interface should look similar to:



Enter numerator: 12
Enter denominator: 16
12/16 is reduced to 3/4

Exercise 4 — TestBuild

Create a TestBuild application that breaks up and then rebuilds numbers 100 through 125 and displays them in a label. The application interface should look similar to the following after clicking Break Up and Rebuild:



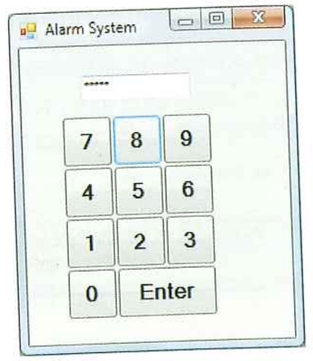
```
Number     Break Up           Rebuild
100        1   0   0          100
101        1   0   1          101
102        1   0   2          102
103        1   0   3          103
104        1   0   4          104
105        1   0   5          105
106        1   0   6          106
107        1   0   7          107
108        1   0   8          108
109        1   0   9          109
110        1   1   0          110
111        1   1   1          111
112        1   1   2          112
113        1   1   3          113
114        1   1   4          114
115        1   1   5          115
116        1   1   6          116
117        1   1   7          117
118        1   1   8          118
119        1   1   9          119
120        1   2   0          120
121        1   2   1          121
122        1   2   2          122
123        1   2   3          123
124        1   2   4          124
125        1   2   5          125
```

The program code should include:

- the ThreeDigits() and TwoDigits() procedures from the NumberBreakdown review.
- a Build() procedure that has firstDigit, SecondDigit, and thirdDigit parameters and returns, in a builtNumber parameter, a value that consists of the number represented by the three digits.

Exercise 5 AlarmSystem

An office building uses an alarm system that is turned off by entering a master code and then pressing Enter. The master code is 62498. Create an AlarmSystem application that displays a message box with an appropriate message after a code is typed and then Enter is clicked. The application interface should look similar to the following after clicking five number buttons:


Alarm System

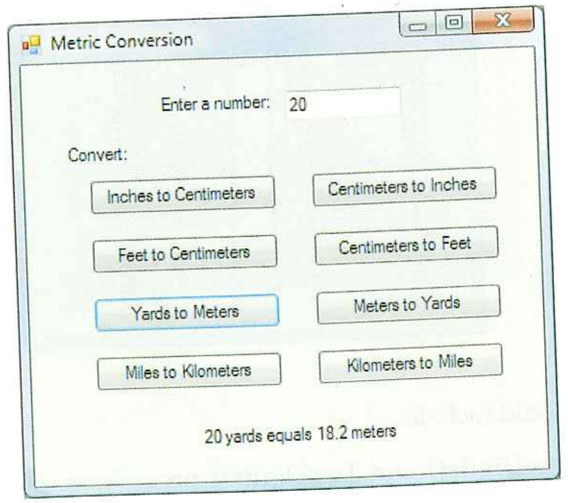
*****

Exercise 6 MetricConversion

The following formulas can be used to convert English units of measurement to metric units:

inches * 2.54 = centimeters
feet * 30 = centimeters
yards * 0.91 = meters
miles * 1.6 = kilometers

Create a MetricConversion application that prompts the user for a number and then converts it from inches to centimeters, feet to centimeters, yards to meters, and miles to kilometers and vice versa when a button is clicked. The program code should include separate functions to perform the conversions. The application interface should look similar to:

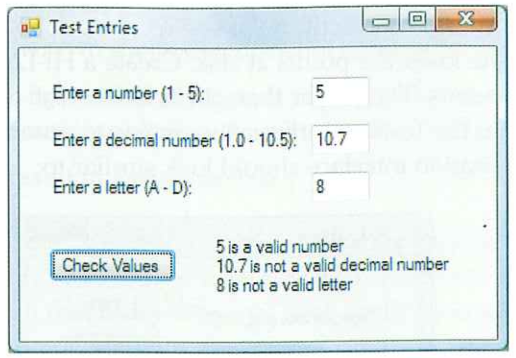

Metric Conversion

Enter a number: 20

20 yards equals 18.2 meters

Exercise 7 TestEntries

Validating user input is often required in programs. Create a TestEntries application that prompts the user for an integer, decimal number, and letter and then determines if the values are valid. The application interface should look similar to:


Test Entries

Enter a number (1 - 5): 5
Enter a decimal number (1.0 - 10.5): 10.7
Enter a letter (A - D): 8

5 is a valid number
10.7 is not a valid decimal number
8 is not a valid letter

The program code should include:

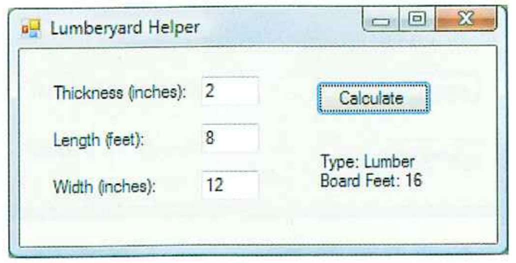
- a ValidInt() function that has highNum, lowNum, and number parameters and returns True if number is in the range lowNum to highNum, and False otherwise.
- a ValidDouble() function that has highNum, lowNum, and number parameters and returns True if number is in the range lowNum to highNum, and False otherwise.
- a ValidChar() function that has highChar, lowChar, character parameters and returns True if character is in the range lowChar to highChar, and False otherwise.

Exercise 8 LumberyardHelper

The basic unit of lumber measurement is the board foot. One board foot is the cubic content of a piece of wood 12 inches by 12 inches by 1 inch thick. For example, a board that is 1 inch thick by 8 feet long by 12 inches wide is 8 board feet:

((1 * (8 * 12) * 12) / (12 * 12 * 1)) = 8

Milled wood is cut to standardized sizes called board, lumber, and timber. A board is one-inch thick or less, timber is more than four inches thick, and lumber is anything between one and four inches thick. Create a LumberyardHelper application that prompts the user for the thickness, length, and width of a piece of wood and then displays the board feet and the classification of the cut. The application interface should look similar to:


Lumberyard Helper

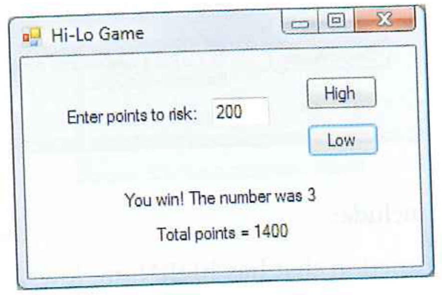
Thickness (inches): 2
Length (feet): 8
Width (inches): 12

Type: Lumber
Board Feet: 16

The program code should include a BoardFeet() function that has `thickness`, `length`, and `width` parameters and returns the number of board feet and a CutClassification() function that has a `thickness` parameter and returns the classification of the cut.

Exercise 9 — Hi-LoGame

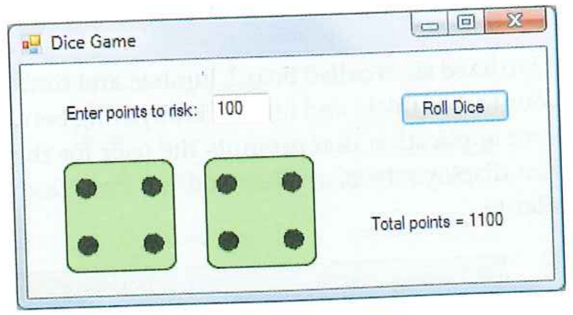
In the Hi-LoGame, the player begins with a score of 1000. The player enters the number of points to risk and chooses High or Low. The player's choice of high or low is compared to a random number between 1 and 13, inclusive. If the number is between 1 and 6 inclusive, it is considered "low." If it is between 8 and 13 inclusive, it is considered "high." The number 7 is neither high nor low, and the player loses the points at risk. If the player guesses correctly, he or she receives double the points at risk. If the player guesses incorrectly, he or she loses the points at risk. Create a Hi-LoGame application that prompts the user for the number of points. The player then clicks either High or Low to play. The program code should include RndInt() from the TestRndIntFunction review to generate the random number between 1 and 13 inclusive. The application interface should look similar to:



You win! The number was 3
Total points = 1400

Exercise 10 — DiceGame

Create a DiceGame application. The player begins with a score of 1000. After entering the number of points to risk, the player clicks Roll Dice. The points on each die is displayed. If the total is even, the player receives double the points at risk. If the total is odd, the player loses the points at risk. The program code should include a RollDice() procedure with picDie1, picDie2, and total parameters that generates a two random numbers in the range 1 to 6 to determine the die faces to display in the picture boxes. RollDie() then updates the total parameter. The `die1.gif`, `die2.gif`, `die3.gif`, `die4.gif`, `die5.gif`, and `die6.gif` data files for this text are required to complete this exercise. The program code should also include RndInt() from the TestRndIntFunction review. The DiceGame application should look similar to:



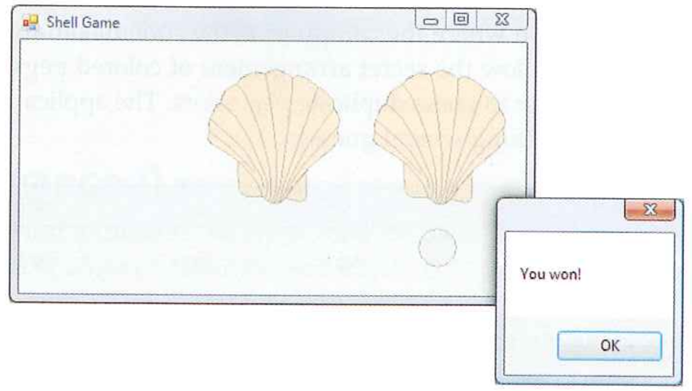
Total points = 1100

Exercise 11 — ShellGame

The ShellGame application from the reviews in this chapter gives the user just one chance at choosing the shell with the pearl. Modify the ShellGame application to give the user a better chance of finding the pearl:

1. After the user selects a shell but before the hidden pearl is displayed, remove (hide) one of the other two shells that does not contain the pearl.
2. Use an input box to ask the user if he or she wants to keep the original guess or choose the remaining shell as the new guess.
3. Display the result in a message box.

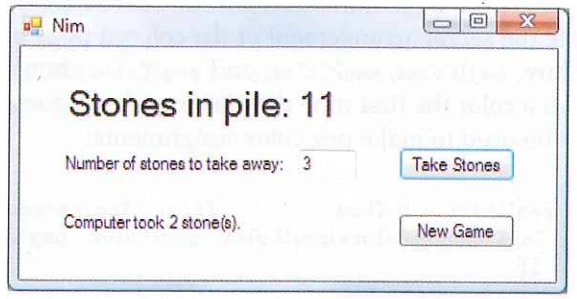
Improve the ShellGame application by appropriately separating tasks into procedures and functions, including using the RndInt() function from the TestRndIntFunction review to generate a random number from 1 to 3 for the shell that hides the pearl. The application interface should look similar to the following after playing one game:



You won!

Exercise 12 — Nim

The game of Nim starts with a random number of stones between 15 and 30. Two players alternate turns and on each turn may take either 1, 2, or 3 stones from the pile. The player forced to take the last stone loses. Create a Nim application that allows the user to play against the computer. In this version of the game, the application generates the number of stones to begin with, the number of stones the computer takes, and the user goes first. The application interface should look similar to:



Stones in pile: 11
Computer took 2 stone(s).

The program code should:

- prevent the user and the computer from taking an illegal number of stones. For example, neither should be allowed to take three stones when there are only 1 or 2 left.

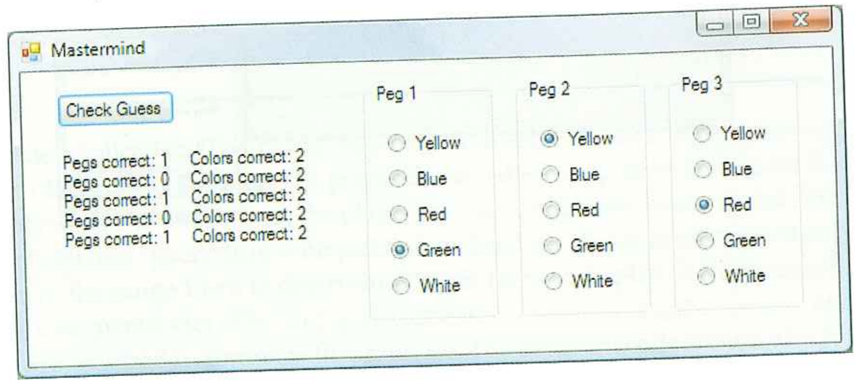
- include the ValidEntry() function presented in this chapter to check user input.
- include RndInt() from the TestRndIntFunction review to generate a random number from 1 to 3 for the computer's turn to remove stones from the pile.
- include separate procedures for the user's turn and the computer's turn.

Exercise 13 — Mastermind

The game of Mastermind is played as follows: one player (the codemaker) chooses a secret arrangement of colored pegs and the other player (the codebreaker) tries to guess it. After each guess, the codemaker reports two numbers:

1. The number of pegs that are the correct color in the correct position.
2. The number of pegs that are the correct color regardless of whether they are in the correct position.

Create a Mastermind application where the computer is the codemaker and the player is the codebreaker. For simplicity, do not allow the secret arrangement of colored pegs to have duplicate colors and do not allow the codebreaker to guess duplicate peg colors. The application interface should look similar to the following after making several guesses:


Mastermind

Check Guess

```
Pegs correct: 1   Colors correct: 2
Pegs correct: 0   Colors correct: 2
Pegs correct: 1   Colors correct: 2
Pegs correct: 0   Colors correct: 2
Pegs correct: 1   Colors correct: 2
```

Peg 1	Peg 2	Peg 3
Yellow	[✓ Yellow]	Yellow
Blue	Blue	Blue
Red	Red	[✓ Red]
[✓ Green]	Green	Green
White	White	White

The program code should:

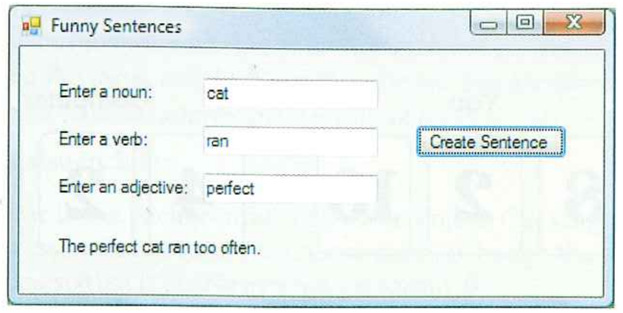
- include a ChooseColors() procedure that has `peg1Color`, `peg2Color`, and `peg3Color` parameters to generate unique colors for the secret arrangement of the colored pegs. Use numbers 1 through 5 to represent colors and use RndInt() from the TestRndIntFunction review to generate a random number.
- generate the secret arrangement of the colored pegs in the btnCheckGuess_Click() procedure. `peg1Color`, `peg2Color`, and `peg3Color` should be static variables that get assigned a color the first time the player makes a guess. Code similar to the following can be used to make peg color assignments:

```
If peg1Color = 0 Then          'first time variable used
   Call ChooseColors(peg1Color, peg2Color, peg3Color)
End If
```

- use separate functions to determine the number of correct colors and the number of correct pegs each time the player makes a guess.

Exercise 14 — FunnySentences

Create a FunnySentences application that prompts the user for a noun, verb, and adjective and then displays a sentence using these words. The program code should include a MakeSentence() procedure that has noun, verb, adjective, and lblLabel parameters and displays the sentence in the label. Use numbers 1 through 5 to represent five different sentences of your choosing and use RndInt() from the TestRndIntFunction review to generate a random number. The application interface should look similar to:


Funny Sentences

Enter a noun:	cat
Enter a verb:	ran
Enter an adjective:	perfect

Create Sentence

The perfect cat ran too often.

Exercise 15 (advanced) — ArithmeticDrill

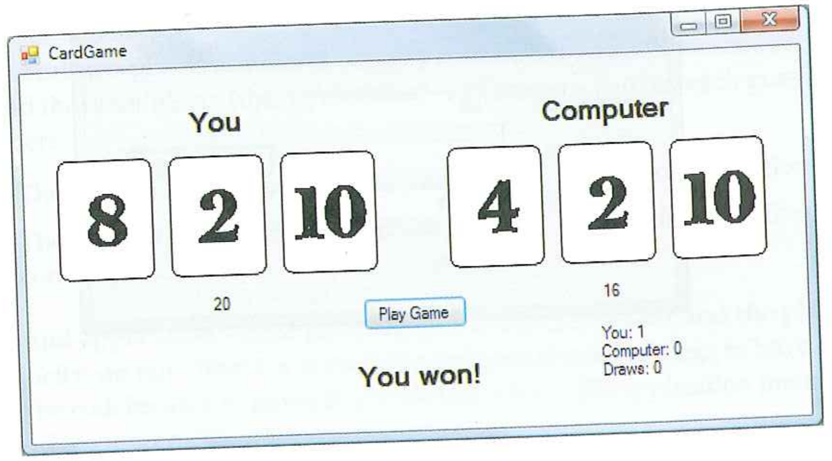
Computers are used to test a student's ability to solve arithmetic problems. Create an ArithmeticDrill application that tests a student on addition, subtraction, or multiplication using random integers between 1 and 100. The student begins by choosing the type of problem and is then asked 10 problems with 3 chances to answer each correctly. If after 3 chances the answer is still incorrect, the correct answer is displayed. A score is calculated by awarding 10 points for a correct answer on the first try, 5 points on the second try, 2 points on the third try, and 0 points if all three attempts are wrong. Your program code should contain procedures, functions, and static variables. RndInt() from the TestRndIntFunction review should be used as well.

Exercise 16 (advanced) — PetAdoption

A new pet adoption agency has opened up in your neighborhood and needs an application to keep track of pets in need of a home. The agency currently has 15 puppies, 10 kittens, 3 canaries, and 2 iguanas. Create a PetAdoption application that keeps a running total of how many pets are available by subtracting animals that have been adopted and adding new animals in need of a home to the total.

Exercise 17 (advanced) — CardGame

The CardGame application that deals three cards to the user (player) and three cards to the computer. The cards dealt are randomly selected and are in the range 1 to 10, inclusive. The winner is the one with the highest score. After each game, a message should be displayed (You won!, Computer won!, or It's a draw!) and a score updated and displayed (1 point for each win). The user can repeatedly play the game. The scores are maintained until the user quits the application. The application interface should look similar to the following after playing the game five times:



Use the pseudocode below when designing the application:

```
Sub btnPlayGame_Click()
    Deal 3 cards to player
    Deal 3 cards to computer

    If winner = player Then
        UpdateScore(playerScore)
        ShowScore
    ElseIf winner = computer Then
        UpdateScore(compScore)
        ShowScore
    Else
        UpdateScore(drawScore)
        ShowScore
    End If
End Sub
```

The interface should display six picture boxes that show the cards dealt. These picture box images should display cardback.gif when the application starts. The `cardback.gif`, `card1.gif`, `card2.gif`, `card3.gif`, `card4.gif`, `card5.gif`, `card6.gif`, `card7.gif`, `card8.gif`, `card9.gif`, and `card10.gif` data files for this text are required to complete this exercise.

The program code should include:

- a DealCard() procedure with picCard and intTotal parameters that generates a random number in the range 1 to 10 to determine the card to display in a picture box. DealCard() then updates the total and returns this value in a parameter.
- RndInt() from the TestRndIntFunction review.
- a Winner() function that compares the totals of the two hands and returns the winner. An UpdateScore() procedure adds 1 to the winner's score, and the ShowScore() procedure displays the current scores in a label.

Exercise 18 — CatchTheSquare

Create a CatchTheSquare application, which includes a label instructing the user to try to click the square and a square picture box object that moves when the user points to it. Objects have additional properties and can respond to events in addition to those discussed in the text. Use the properties and event listed below for this application.

Form properties also include:

- **Size** has properties Height and Width that correspond to the height and width of the Form object. Note: The Height and Width refer to the entire form. When positioning objects on the form, consider that the title bar has an approximate height of 32 and the borders have an approximate width of 6.

PictureBox properties also include:

- **BackColor** is the background color of an object. Clicking the arrow in this property displays a list of colors to choose from. (Change the background color of the PictureBox so that it visible against the form.)
- **Location** has properties X and Y, which can be changed by assigning Location a new Point structure that corresponds to the form location for the upper-left corner of the PictureBox object.
- **Size** has properties Height and Width that correspond to the height and width of the PictureBox object.

For example, the following statement changes the location of a PictureBox object:

```
Me.picSquare.Location = New Point(100, 100) 'upperleft of PictureBox at 100,100
```

A MouseEnter event can be coded for a picture box. A MouseEnter event occurs when the mouse pointer is over the PictureBox object. (This event can be used to move the square before the user has a chance to click.)