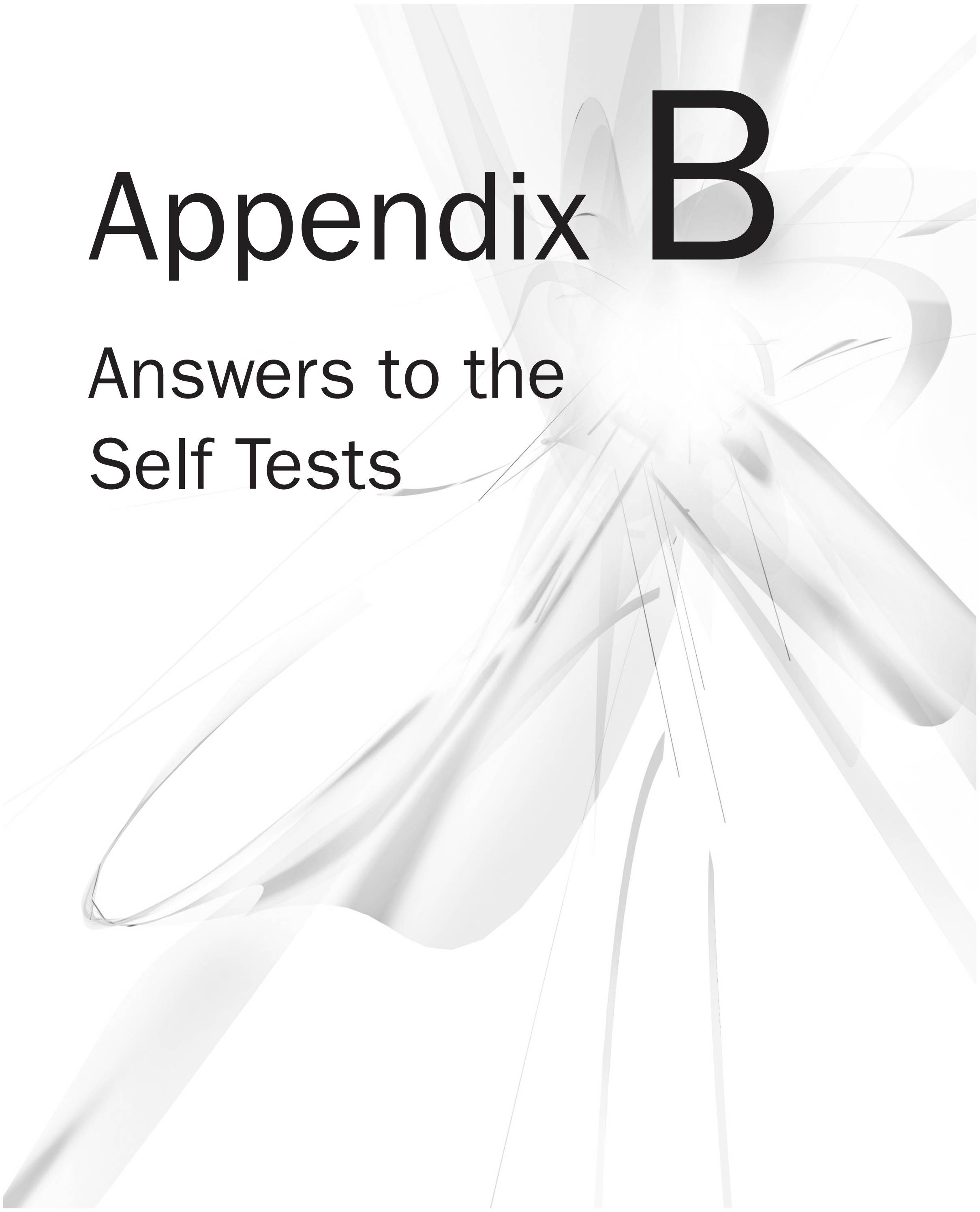
Appendix B

Answers to the Self Tests



Chapter 1

1. **A.** method; **B.** Classes; **C.** Namespaces; **D.** Variables
2. True
3. **A.** SHIFT-F5; **B.** F9 (or clicking beside the instruction); **C.** F11; **D.** F10; **E.** F5
4.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            Console.WriteLine("Pat");
            Console.ReadLine();
        }
    }
}
```

5. Use indentation, use descriptive naming, and add comments.
6. Ten debugging techniques are
   - **Errors and warnings** Visual Studio identifies syntax errors with red wavy lines and identifies warnings about questionable sections of code with green wavy lines. The Error List window summarizes these errors and warnings with descriptions to help you fix them. Clicking an error or warning in the list automatically moves your cursor to the corresponding line of code in the program.
   - **Breakpoints** Breakpoints allow you to stop and inspect your variable values at run time.
   - **Tooltips** While your program is halted, you can hover your cursor over variables to display variable contents in tooltips.
   - **Stepping into methods** Stepping into methods allows you to trace your program's execution line by line from one method to the next. You may observe variable values and program logic at each point in the program.
   - **Stepping over methods** Stepping over calling instructions helps you to save time by allowing you to skip over sections of code that already work correctly.
   - **Call Stack window** While your program is halted at a breakpoint, the Call Stack window allows you to observe the prior sequence of the methods that were called. This map of prior instructions can help to identify faulty logic that led to an error.

- **Locals window** The Locals window lists all variables and corresponding values of a method while the program is halted there. This quick but helpful summary is automatically provided for you.
- **Watch window** The Watch window allows you to monitor a custom list of variables while stepping through your code.
- **Try-catch blocks** Try-catch blocks allow you to gracefully handle run-time errors. These structures are especially useful for preventing program crashes when accessing resources such as files or database content.
- **Logging data** Writing log data to a file can help you monitor program changes when hundreds, thousands, or even millions of instructions must execute to create error conditions.

7. **D.** Try-catch blocks allow programs to gracefully handle errors when referencing external resources such as files or database content.
8. `c` = 4; `d` = 5; `sum` = 9

Chapter 2

1. Variable declaration occurs when stating the data type and variable name. Variable initialization occurs when assigning a value to the variable for the first time.
2. It is not possible to assign a fraction value to a float variable without clarifying that the data type is a float. This can be fixed by adding the numeric literal `f` to the end of the number.
3.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            const double YEAR_IN_DAYS = 224.7d;
            Console.Write("Duration of Venus's year in days: ");
            Console.WriteLine(YEAR_IN_DAYS);
            Console.ReadLine();
        }
    }
}
```

4. Change the instruction from

```
int island = MAUI;
```

to

```
int island = OAHU;
```

5.

```
using System;

namespace Starter {
    class Program {
        public static void Main() {
            const int NUM_ISLANDS = 3;
            const int HAWAII = 0; const int MAUI = 1; const int OAHU = 2;
            // Part A:  Assigning size in the declaration.
            string[] name = new string[NUM_ISLANDS];
            name[HAWAII]  = "Hawaii"; // 1st element at index '0'
            name[MAUI]    = "Maui";   // 2nd element at index '1'
            name[OAHU]    = "Oahu";   // 3rd element at index '2'

            // Part B:  Assigning size later in the program.
            int[] population;
            population = new int[NUM_ISLANDS];
            population[HAWAII] = 187200;
            population[MAUI]   = 145000;
            population[OAHU]   = 955000;

            string[] highestLocation = new string[NUM_ISLANDS];
            highestLocation[HAWAII]  = "Mauna Kea Volcano";
            highestLocation[MAUI]    = "East Maui Volcano";
            highestLocation[OAHU]    = "Waianae Mountains";

            // Part C:  Initializing the array and size in the declaration.
            float[] squareMiles = new float[] { 4028.0f, 727.2f, 596.7f };

            int island = MAUI;
            Console.WriteLine("* Statistics for " + name[island] + " *");
            Console.WriteLine("Population: " + population[island]);
            Console.WriteLine("Square miles: " + squareMiles[island]);
            Console.WriteLine("Location of highest elevation: " +
                               highestLocation[island]);
            Console.ReadLine();
        }
    }
}
```

6. The last instruction before *Console.ReadLine()* becomes

```
Console.WriteLine("Thursday's forecast is for a low temperature of "
                + forecast[THU, LOW] + " and a high temperature of "
                + forecast[THU, HIGH] + ".");
```

7.

```
using System;

namespace Starter {
    class Program {
        enum Semester { Fall, Winter, Spring }
        public static void Main() {
            int springValue = (int)Semester.Spring;
            Console.WriteLine("Spring value: " + springValue);
            Semester springSemester = (Semester)springValue;
            Console.WriteLine("The enumerator associated with 'springValue' = " +
                               springSemester.ToString());

            string[] semesters = Enum.GetNames(typeof(Semester));
            Console.Write("Enumerators include: ");
            Console.Write(semesters[0] + ", " +
                          semesters[1] + ", " +
                          semesters[2]);
            Console.ReadLine();
        }
    }
}
```

8.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            const int MON = 0, TUE = 1, WED = 2, THU = 3, FRI = 4;
            const int LOW = 0, HIGH = 1, RAIN = 2;

            const int NUM_DAYS = 5, NUM_WEATHER_STATS = 3;   // Dimension sizes.

            // Declare and initialize array.
            int[,] forecast    = new int[NUM_DAYS, NUM_WEATHER_STATS];

            // Assign values to each element in array.
            forecast[MON, LOW]=70; forecast[MON, HIGH]=79; forecast[MON, RAIN]=30;
            forecast[TUE, LOW]=68; forecast[TUE, HIGH]=82; forecast[TUE, RAIN]=75;
            forecast[WED, LOW]=69; forecast[WED, HIGH]=80; forecast[WED, RAIN]=98;
            forecast[THU, LOW]=71; forecast[THU, HIGH]=81; forecast[THU, RAIN]=80;
            forecast[FRI, LOW]=67; forecast[FRI, HIGH]=78; forecast[FRI, RAIN]=34;

            // Show Friday's forecast.
            Console.WriteLine("Friday's forecast is for a low temperature of "
                            + forecast[FRI, LOW]  + " and a high temperature of "
                            + forecast[FRI, HIGH] + ".  The chance of rain is "
                            + forecast[FRI, RAIN] + "%.");

            Console.ReadLine();
        }
    }
}
```

Chapter 3

1.

```
using System;
using System.Security.Cryptography;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {

            float vInitial = 10.0f;
            float a        = 0.5f;
            float t        = 20.0f;
            float vFinal   = vInitial + a * t;
            Console.WriteLine("Final velocity = " + vFinal + " m/s");
            Console.ReadLine();
        }
    }
}
```

2. 2

3. 46

4. The two techniques recommended are

- Generating one *Random* object with the default *Random* constructor. This object can then be used over time to generate many random numbers.
- Generating a random number by passing a random seed to the *Random* constructor.

5.

```
using System;
using System.Security.Cryptography;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Create byte array.
            const int TOTAL_BYTES =  4;
            const int MIN         =  1;
            const int MAX         = 22;
            byte[]    byteArray   = new byte[TOTAL_BYTES];

            // Generate Cryptography class object.
               RNGCryptoServiceProvider crypto
                              = new RNGCryptoServiceProvider();

            // Generate and display first integer.
            crypto.GetBytes(byteArray);
            int seed          = BitConverter.ToInt32(byteArray, 0);
```

```
                Random random     = new Random(seed);
                int randomInteger = random.Next(MIN, MAX);
                Console.WriteLine("Random integer between 1 and 21: "
                                  + randomInteger);

                // Generate and display second integer.
                crypto.GetBytes(byteArray);
                seed              = BitConverter.ToInt32(byteArray, 0);
                random            = new Random(seed);
                randomInteger     = random.Next(MIN, MAX);
                Console.WriteLine("Random integer between 1 and 21: "
                                  + randomInteger);
                Console.ReadLine();
            }
        }
    }
```

Chapter 4

1.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static float GetSurfaceArea(float radius) {
            const float PI = 3.14159f;
            return  4.0f * PI * (float) Math.Pow(radius, 2);
        }

        static void Main() {
            const float RADIUS = 9.0f;
            float surfaceArea  = GetSurfaceArea(RADIUS);
            Console.WriteLine(surfaceArea);
            Console.ReadLine();
        }
    }
}
```

2. **A.** True; **B.** False; **C.** True; **D.** False; **E.** False; **F.** False

3. Local declarations have limited accessibility, which limits unwanted data tampering. Local declarations are easier to read since the declarations are located where they are used. The limited existence of a local variable makes it easier to debug than a class-level variable.

4.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Call overload with no fax.
            DisplayContact("Banana Republic Women", "735 State Street",
                           "Santa Barbara", "CA", "93101");
            Console.WriteLine();

            // Call overload with fax.
            DisplayContact("Legal Aid Society", "1223 West Sixth Street",
                           "Cleveland", "OH", "44113", "216-586-3220");
            Console.WriteLine();

            // Call overload with fax and phone.
            DisplayContact("Caveman Zipline", "1135 Hwy West",
                           "Sullivan", "MO", "63080",
                           "573-468-4000", "573-468-9477");

            Console.ReadLine();
        }

        // Overload with no fax.
        static void DisplayContact(string orgName, string street,  string city,
                                   string state, string zip) {
            Console.WriteLine(orgName);
            Console.WriteLine(street);
            Console.WriteLine(city + ", " + state);
            Console.WriteLine(zip);
        }

        // Overload with fax.
        static void DisplayContact(string orgName, string street, string city,
                                   string state, string zip, string fax) {
            // Use DisplayContact() overload with no fax.
            DisplayContact(orgName, street, city, state, zip);
            Console.WriteLine("fax: " + fax);                  // Show fax.
        }

        // Overload with fax and phone.
        static void DisplayContact(string orgName, string street, string city,
                                   string state, string zip, string fax,
                                   string phone) {
            // Use DisplayContact() overload with fax.
            DisplayContact(orgName, street, city, state, zip, fax);
            Console.WriteLine("phone: " + phone);              // Show phone.
        }
    }
}
```

5. The scope of the *gravity* variable is restricted to *ShowLocalizedValues()*. Therefore, the variable value does not exist outside this method.

6. **A.** value; **B.** reference

7.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            float house  = 200000.0f;
            float salary = 75000.0f;
            float annualFoodCost = 10000.0f;

            // Explicitly pass variables by reference
            AdjustForInflation(ref house, ref salary, ref annualFoodCost);
            Console.WriteLine("** Values After Inflation Adjustment **");
            Console.WriteLine("Home:   " + house);
            Console.WriteLine("Income: " + salary);
            Console.WriteLine("Annual Food Cost: " + annualFoodCost);
            Console.ReadLine();
        }

        // Receive arguments explicitly by reference.
        static void AdjustForInflation(ref float home, ref float income,
                                       ref float annualFoodCost) {
            const float RATE = 1.03f;
            home   *= RATE;
            income *= RATE;
            annualFoodCost *= RATE;
        }
    }
}
```

Chapter 5

1. **A.** True; **B.** True; **C.** False; **D.** False

2.

```
using System;
namespace Starter {
    class Program {
        // Define constants to store tornado classifications.
        const int INACTIVE = -1;
        const int F0 = 0, F1 = 1, F2 = 2, F3 = 3, F4 = 4;
        // Show warning based on tornado strength.
        public static void ShowWarning(int strength) {
            if (strength == F4) {
                Console.WriteLine("Devastating damage warning." );
            }
            else if (strength == F3) {
                Console.WriteLine("F3+: Severe damage warning.");
            }
            else if (strength == F2) {
                Console.WriteLine("F2: Significant damage warning.");
            }
            else if (strength == F1) {
                Console.WriteLine("F1: Moderate damage warning.");
            }
            else {
                Console.WriteLine("Inactive: No damage expected.");
            }
        }

        public static void Main() {
            int strength = INACTIVE;
            ShowWarning(strength);  // Call ShowWarning() method.
            strength     = F3;
            ShowWarning(strength);  // Call ShowWarning() method.
            strength     = F4;
            ShowWarning(strength);  // Call ShowWarning() method.
            Console.ReadLine();
        }
    }
}
```

3.

```
using System;
namespace Starter {
    class Program {
        public static void Main() {
            double celsius    = 19.3;
            double fahrenheit = (celsius * 9 / 5) + 32;

            Console.WriteLine("Celsius: " + celsius);
            Console.WriteLine("Fahrenheit: "
                            + fahrenheit.ToString("N2"));

            if (fahrenheit <= 32)
                Console.WriteLine("It is cold.");
            else if(fahrenheit > 32 && fahrenheit < 65)
                Console.WriteLine("It is chilly out.");
            else if(fahrenheit >=65 && fahrenheit < 80)
                Console.WriteLine("This feels good.");
            else
                Console.WriteLine("It is hot out.");
            Console.ReadLine();
        }
    }
}
```

4.

```
using System;

namespace Starter {
    class Program {
        const int F1 = 1, F2 = 2, F3 = 3, F5 = 5;

        static void ShowData(int minimumSpeed) {
            Console.WriteLine("Minimum wind speed: " + minimumSpeed );
            Console.WriteLine();
        }

        static void ShowFrequency(int tornadoRating) {
            int minimumSpeed = 0;           // Miles per hour.

            switch (tornadoRating) {        // Start switch.
                case F1:                    // Check for matching value.
                    minimumSpeed = 73;      // Execute this block if match.
                    ShowData(minimumSpeed);
                    break;                  // Exit switch.
                case F2:                    // Check for matching value.
                    minimumSpeed = 112;     // Execute this block if match.
                    ShowData(minimumSpeed);
                    break;                  // Exit switch.
                case F3:
                    minimumSpeed = 207;
                    ShowData(minimumSpeed);
                    break;
                default:                    // Default code if no case match.
                    Console.WriteLine("Information not available.");
                    Console.WriteLine();
                    break;                  // Exit switch.
            }                               // End switch.
        }

        public static void Main()  {
            Console.WriteLine("* Statistics for F2 tornado*");
            ShowFrequency(F2);
            Console.WriteLine("* Statistics for F3 tornado*");
            ShowFrequency(F3);
            Console.WriteLine("* Statistics for F5 tornado*");
            ShowFrequency(F5);
            Console.ReadLine();
        }
    }
}
```

5.

```
using System;

namespace Starter {
    class Program {
        public static void Main()  {
            Console.WriteLine("Input 'a', 'b', or anything else.");
            string input = Console.ReadLine();
            switch (input) {
                case "a":
                    Console.WriteLine("You pressed a.");
                    break;
                case "b":
                    Console.WriteLine("You pressed b.");
                    break;
                default:
                    Console.WriteLine("You did not press a or b.");
                    break;
            }
            Console.ReadLine();
        }
    }
}
```

6.

```
using System;

namespace Starter {
    class Program {
        public static void Main()  {
            decimal        balance = 10000m;
            const decimal INTEREST = 1.07m;
            const int    NUM_YEARS = 25;

            for (int i = 0; i < NUM_YEARS; i++) {
                balance *= INTEREST;
                Console.Write("Balance at end of year " + (i + 1) + ":  ");
                Console.WriteLine(balance.ToString("N2"));
            }
            Console.ReadLine();
        }
    }
}
```

7.

```
using System;

namespace Starter {
    class Program {
        public static void Main()  {
            decimal        balance = 10000m;
            const decimal INTEREST = 1.07m;
            const int    NUM_YEARS = 25;
                  int      counter = 1;
            while(counter <= NUM_YEARS) {
                balance *= INTEREST;
                Console.Write("Balance at end of year " + (counter++) + ":  ");
                Console.WriteLine(balance.ToString("N2"));

            }
            Console.ReadLine();
        }
    }
}
```

8.

```
using System;

namespace Starter {
    class Program {
        public static void Main()  {
            decimal        balance = 10000m;
            const decimal INTEREST = 1.07m;
            const int    NUM_YEARS = 25;
                  int      counter = 1;
            do {
                balance *= INTEREST;
                Console.Write("Balance at end of year " + (counter) + ":  ");
                Console.WriteLine(balance.ToString("N2"));
            } while (counter++ < NUM_YEARS);
            Console.ReadLine();
        }
    }
}
```

9.

```
using System;

namespace Starter {
    class Program {
        public static void Main()  {
            float[] floatArray = { 13.3f, 14.2f, 5.2f };
            for (int i = 0; i < floatArray.Length; i++) {
                Console.WriteLine(floatArray[i]);
            }
            Console.ReadLine();
        }
    }
}
```

Chapter 6

1.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            string fullName = "Pat McGee";
            char lastCharacter = fullName[fullName.Length - 1];
            Console.WriteLine("Last Character of "
                             + fullName + " is "
                             + lastCharacter + ".");
            Console.ReadLine();
        }
    }
}
```

2.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            string    fullName = "Jeffrey steinberg";
            int       position = fullName.IndexOf(" ") + 1;
            char      lastChar = fullName[position ];
            string lastInitial = lastChar.ToString().ToUpper();
            fullName = fullName.Remove(position, 1);
            fullName = fullName.Insert(position, lastInitial);
            Console.WriteLine(fullName);
            Console.ReadLine();
        }
    }
}
```

3. `^[a-zA-Z]+_*-*.*(@retrofitness.com)$`

4. `^(Cable|DSL)$`

5. `^([1-9]|[1-9]{2,2}|10[0-9]|110)$`

6. `.*[g|G][o|O].*[g|G][o|O].*`

7. `^\$[1-9][0-9]{0,}\.[0-9]{2,2}`

8.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            do {
                Console.Write("Input percentage earned: ");
                string input = Console.ReadLine();
                float percentage;
                bool result  = float.TryParse(input, out percentage);
                if (result) {
                    Console.WriteLine("You entered "
                                    + percentage.ToString("N2")
                                    + "%");
                    break;
                }
                else
                    Console.WriteLine(
                            "This value is incorrect. Please try again.");
            } while (true);
            Console.ReadLine();
        }
    }
}
```

Chapter 7

1.

```
using System;
namespace Starter {
    class Program {
        public static void Main() {
            DateTime dt = DateTime.Now;
            Console.Write(dt.ToString("dddd") + " ");
            Console.Write(dt.ToString("MMMM" + " "));
            Console.Write(dt.ToString("dd") + ", ");
            Console.WriteLine(dt.ToString("t"));
            Console.ReadLine();
        }
    }
}
```

2.

```
using System;
namespace Starter {
    class Program {
        public static void Main() {
            const int DAY=1, MTH=1, YRS = 1980;
            DateTime dt2000 = new DateTime(YRS, MTH, DAY);
            DateTime dtNow  = DateTime.Now;
            TimeSpan     ts = dtNow.Subtract(dt2000);
            Console.WriteLine(ts.TotalSeconds.ToString("N0"));
            Console.ReadLine();
        }
    }
}
```

3.

```
using System;
namespace Starter {
    class Program {
        public static void Main() {
            const int  YEAR = 2013, DAY=1, MTH=1, CYCLE = 4;
            int    leapYear = 0;
            double dayCount = 365;
            for(int year=YEAR; year>YEAR - CYCLE; year-- ) {
                DateTime next  = new DateTime(year + 1, MTH, DAY);
                DateTime start = new DateTime(year, MTH, DAY);
                TimeSpan ts = next.Subtract(start);
                if (ts.TotalDays > dayCount) {
                    leapYear = year;
                    dayCount = ts.TotalDays;
                }
            }
            Console.WriteLine("The leap year was: " + leapYear);
            Console.WriteLine("The day count for that year was " + dayCount);
            Console.ReadLine();
        }
    }
}
```

Chapter 8

1. Objects are implementations of classes.
Default constructors have no parameters.
Classes by default are public.
Methods by default are private.

2.

```
using System;
namespace Starter {
    class Program{
        public static void Main() {
            // Object creation.
            Location empireStateBldg = new Location(350, " 5th Avenue");
            Location republicPlaza   = new Location();
            Location addressC = new Location(1, "Main Street", "San Jose");

            // Call public method to store data in object.
            republicPlaza.SetLocation(9, "Raffles Place");

            // Use object to access a public method for display.
            empireStateBldg.DisplayLocation();
            republicPlaza.DisplayLocation();
            addressC.DisplayLocation();
            Console.WriteLine(addressC.GetCity());
            Console.ReadLine();
        }
    }

    // Class declaration.
    class Location {
        // Private data members.
        private int    streetNumber;
        private string streetName;
        private string city;

        // Default constructor.
        public Location() {
        }

        // Overloaded constructor.
        public Location(int streetNumber, string streetName) {
            this.streetName   = streetName;
            this.streetNumber = streetNumber;
        }
        public Location(int streetNumber, string streetName,
                        string city) {
            this.streetName   = streetName;
            this.streetNumber = streetNumber;
            this.city         = city;
        }

        // Publicly accessible methods.
        public void SetLocation(int streetNumber, string streetName) {
            this.streetName   = streetName;
            this.streetNumber = streetNumber;
        }
```

```
            public string GetCity() {
                return city;
            }
            public void DisplayLocation() {
                Console.Write(streetNumber + " ");
                Console.WriteLine(streetName);
            }
        }
    }
```

3.

```
using System;
namespace Starter {
    class Program {
        public static void Main() {
            DateTime birthday = new DateTime(1511, 6, 18);
            // Declare and initialize a Person object.
            Person     person = new Person("Bartolomeo", "Ammannati",
                                           birthday);
            Console.Write(person.FullName + " " );
            Console.WriteLine(birthday.ToString("yyyy/MM/dd"));
            Console.WriteLine("Age: " + person.Age);
            Console.ReadLine(); // user must press 'Exit' to quit.
        }
    }

    class Person {
        // This property is available to the object for reads but not for writes.
        public DateTime Birthday  { get; private set; }

        // These properties are not readable or writable outside the class.
        private string FirstName  { get; set; }
        private string LastName   { get; set; }

        // This property has read-only access and is publicly accessible.
        public  string FullName   { get {return FirstName + " " + LastName; }}
        public int Age {
            get {
                // Generates yyyy.mmdd
                string now = DateTime.Now.ToString("yyyy.MMdd");
                // Generates 1511.0618 for Bartolomeo Ammannati
                string dob = this.Birthday.ToString("yyyy.MMdd");
                // Calculate age.
                int age = (int)(Convert.ToSingle(now) - Convert.ToSingle(dob));
                return age;
            }
        }

        public Person(string firstName, string lastName, DateTime birthday) {
            FirstName = firstName;
            LastName  = lastName;
            Birthday  = birthday;
        }
    }
}
```

4. *facilityMonthlyCost* and *deptMonthlyCost* are fully qualified because their class reference is prefixed with the namespace in the declaration:

```
Facility.Cost facilityMonthlyCost = new Facility.Cost(MONTHLY_RENT);
Team.Cost     deptMonthlyCost     = new Team.Cost(MONTHLY_WAGES, SUPPLIES);
```

The declaration for *dept* is not fully qualified:

```
Department  dept               = new Department("Sales", RENT_RATIO);
```

Declarations for *facilityMonthlyCost* and *deptMonthlyCost* need to be fully qualified to avoid ambiguous references to more than one *Cost* class in two separate namespaces.

5.

```
using Acme.Facility;  // Reference the Facility namespace
using Acme.Team;      // Reference the Team namespace
```

6. Monument/Castle1.cs:

```
using System;

namespace ConsoleApplication1.Monument {
    public partial class Castle {
        int    Year { get; set; }  // Private properties
        string Name { get; set; }
        string Architect { get; set; }

        // Constructor
        public Castle(string name, int constructed) {
            Year = constructed;
            Name = name;
        }
        public Castle(string name, int constructed, string architect) {
            Year = constructed;
            Name = name;
            Architect = architect;
        }
    }
}
```

Monument/Castle2.cs:

```
using System;

namespace ConsoleApplication1.Monument {
    public partial class Castle {
        public void DiplayDetail() {    // Output
            Console.WriteLine("*** " + Name + " Castle ***");
            Console.WriteLine("Build Date: " + Year);
            Console.WriteLine("Architect:  " + Architect);
            Console.WriteLine();
        }
    }
}
```

Program.cs:

```
using System;
using ConsoleApplication1.Monument;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            Castle castleA = new Castle("Ballymoon", 1300, "Roger Bigod");
            Castle castleB = new Castle("Leighlinbridge", 1547,
                                        "Edward Bellingham");

            castleA.DiplayDetail();
            castleB.DiplayDetail();
            Console.ReadLine();
        }
    }
}
```

Chapter 9

1. **A.** False; **B.** True; **C.** False; **D.** False; **E.** True; **F.** False; **G.** False; **H.** True

2.

```
using System;
namespace Starter {
    class Program {
        public static void Main() {
            const string TAX_CODE = "TAX-FREE GIVING";
            Charity unicef    = new Charity("Unicef", "Child Advocacy");
            Charity unitedWay = new Charity("United Way", TAX_CODE,
                                            "Community Support");

            Console.ReadLine();
        }
    }

    public class Organization {
       // These properties can't be accessed outside the base class.
       private string Name    { get; set; }
       private string TaxCode { get; set; }
       private string Sector  { get; set; }
```

```
        // Base class constructor.
        public Organization(string name, string taxCode, string sector) {
            Name    = name;
            TaxCode = taxCode;
            Sector  = sector;
            Console.WriteLine("* Base constructor *");
        }
        // This protected method can only be accessed from child classes.
        protected void ShowOrganizationInfo() {
            Console.WriteLine("Organization Name: " + Name);
            Console.WriteLine("Sector:   " + Sector);
            Console.WriteLine("Tax code: " + TaxCode);
            Console.WriteLine();
        }
    }
    public class Charity : Organization {
        const string TAXATION_CODE = "Unassigned";
        // Calls base constructor with 2 child constructor parameters
        // and 1 constant.
        public Charity(string name, string sector)
                      : base(name, TAXATION_CODE, sector) {
            Console.WriteLine("* Child Constructor A *");
            ShowOrganizationInfo();
        }
        // Calls base constructor with three child constructor parameters.
        public Charity(string name, string taxCode, string sector)
                      : base(name, taxCode, sector) {
            Console.WriteLine("* Child Constructor B *");
            ShowOrganizationInfo();
        }
    }
}
```

3. The override is possible because *DisplayPromotions()* is virtual.

```
using System;

namespace Starter {
    class Program {
        static void Main() {
            Fashion fashionDepartment = new Fashion();
            fashionDepartment.DisplayPromotions();
            Console.ReadLine();
        }
    }

    // Stores sale event data.
    class Promotion {
        public string      Name  { get; private set; }
        public DateTime    Start { get; private set; }
        public DateTime    End   { get; private set; }
        public Promotion(DateTime start, DateTime end, string name) {
            Start = start; End = end; Name = name;
        }
    }

    // Abstract class that stores and displays promotional information.
    abstract class Department {
        public abstract string  DepartmentName { get; protected set; }
        protected Promotion[]   sales;
        protected abstract void AssignPromotions();

        public virtual void DisplayPromotions() {
            Console.WriteLine("{0} Department Promotions: ",
                              DepartmentName);

        }
    }

    // Implementing class that sets department name and assigns promotions.
    class Fashion : Department {
        public override string DepartmentName { get; protected set; }
        public Fashion() {
            DepartmentName = "Fashion";
            AssignPromotions();
        }
        protected override void AssignPromotions() {
            DateTime  start     = new DateTime(2014, 9, 1);  // Sept.  1
            DateTime  end       = new DateTime(2014, 9, 15); // Sept. 15
            Promotion promotion = new Promotion(start, end,
                                                "Fall Sale");
            sales = new Promotion[] { promotion };
        }
        public override void DisplayPromotions() {
            base.DisplayPromotions();
            foreach(Promotion sale in sales) {
                Console.WriteLine("Name:  " + sale.Name);
                Console.Write("Start: " + sale.Start.ToString("m"));
                Console.WriteLine(", "  + sale.Start.ToString("yyyy"));
                Console.Write("End:   " + sale.End.ToString("m"));
                Console.WriteLine(", "  + sale.End.ToString("yyyy"));
            }
        }
    }
}
```

4.

```
using System;

namespace Starter {
    class Program {
        static void Main() {
```

```
        Womens fashionDepartment = new Womens();
        fashionDepartment.DisplayPromotions();
        Console.ReadLine();
    }
}

// Stores sale event data.
class Promotion {
    public string      Name  { get; private set; }
    public DateTime    Start { get; private set; }
    public DateTime    End   { get; private set; }

    public Promotion(DateTime start, DateTime end, string name) {
        Start = start; End = end; Name = name;
    }
}

// Abstract class that stores and displays promotional information.
abstract class Department {
    public abstract string  DepartmentName { get; protected set; }
    protected Promotion[]   sales;
    protected abstract void AssignPromotions();

    public virtual void DisplayPromotions() {
        Console.WriteLine("{0} Department Promotions: ",
                          DepartmentName);
        foreach(Promotion sale in sales) {
            Console.WriteLine("Name:  " + sale.Name);
            Console.WriteLine("Start: " + sale.Start.ToString("m"));
            Console.WriteLine("End:   " + sale.End.ToString("m"));
        }
    }
}

// Implementing class that sets department name and assigns promotions.
class Fashion : Department {
    public override string DepartmentName { get; protected set; }
    public Fashion() {
        DepartmentName = "Fashion";
        AssignPromotions();
    }
    protected override void AssignPromotions() {
        DateTime  start     = new DateTime(2014, 9, 1);  // Sept.  1
        DateTime  end       = new DateTime(2014, 9, 15); // Sept. 15
        Promotion promotion = new Promotion(start, end,
                                            "Fall Sale");
        sales = new Promotion[] { promotion };
    }
}
class Womens : Fashion {
   public override void DisplayPromotions() {
        Console.WriteLine("*** Women's ***");
```

```
                base.DisplayPromotions();
            }
        }
    }
```

5.

```
using System;
namespace Starter {
    public class Program {
        public static void Main() {
            const int NUM_MAMMALS    = 2;
            // Declare array of Mammal Objects
            Mammal[] mammals         = new Mammal[NUM_MAMMALS];

            // Initialize array with different mammal types
            // using polymorphism.
            mammals[0]               = new Lion();
            mammals[1]               = new Bear();

            foreach (Mammal mammalGroup in mammals) {
                mammalGroup.DisplayMammalType();
            }
            Console.ReadLine();
        }
    }
    public abstract class Mammal {         // Base class
        public abstract void DisplayMammalType();
    }
    public class Lion : Mammal {           // Child class
        public override void DisplayMammalType() {
            Console.WriteLine("I am a Lion!");
        }
    }
    public class Bear : Mammal {           // Child class
        public override void DisplayMammalType() {
            Console.WriteLine("I am a Bear!");
        }
    }
}
```

6. The error message "'Starter.RegionalSalesRep': cannot derive from sealed type 'Starter.SalesRep'" appears because it is not possible to derive a new class from a sealed class.

Chapter 10

1. A. True; **B.** False; **C.** True

2. A. Dictionary; **B.** ArrayList; **C.** KeyValuePair; **D.** Hashtable; **E.** List

3.

```
using System;
using System.Collections.Generic;

namespace Starter {
    public class Program {
        public static void Main( ) {
            // Build a List of Account family objects with polymorphism.
            List<Account> accounts  = new List<Account>();
            accounts.Add(new Checking());
            accounts.Add(new Savings());
            accounts.Add(new JointSavings());

            // Display the Account family object values.
            for(int i=0; i<accounts.Count; i++)
                accounts[i].DisplayDetail();
            Console.ReadLine();
        }
    }
    // Create the Account base class.
    public abstract class Account {
        public abstract void DisplayDetail();
    }
    // Create a child of the Account class.
    public class Checking : Account {
        public override void DisplayDetail() {
            Console.WriteLine("This is a Checking object.");
        }
    }
    // Create a child of the Account class.
    public class Savings : Account {
        public override void DisplayDetail() {
            Console.WriteLine("This is a Savings object.");
        }
    }
    // Create a JointSavings class.
    public class JointSavings : Account {
        public override void DisplayDetail() {
            Console.WriteLine("This is a JointSavings object.");
        }
    }
}
```

4.

```
using System;
using System.Collections;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
          ArrayList errands   = new ArrayList();    // Declare and initialize.
            errands.Add(new GroceryItem("Apples", 6));// Add objects.
            errands.Add(new GroceryItem("Carrots", 3));
            errands.Add(new BillableItem("Hydro", 145.33m));
            string[] toDo       = { "Go to tailors.", "Put air in tire." };
            errands.InsertRange(2, toDo);           // Add string array starting
                                                    // at third position.
            GroceryItem bananas = new GroceryItem("Bananas", 5);
            errands.Insert(1, bananas);             // Add item at 2nd position.
            ShowErrands(errands);                   // Show all items.
            errands.RemoveAt(0);                    // Remove first item.
            errands.Remove(bananas);                // Remove item by value.
            ShowErrands(errands);                   // Show all items.
            Console.ReadLine();
        }
        static void ShowErrands(ArrayList errands) {
            Console.WriteLine("* ToDo List *");
            foreach (Object obj in errands) {
                // If object is 'GroceryItem' item cast it and call its
                // Display() method.
                if (obj.GetType() == typeof(GroceryItem)) {
                    GroceryItem groceryItem = (GroceryItem)obj;
                    groceryItem.Display();
                }
                else if (obj.GetType() == typeof(BillableItem)) {
                    BillableItem billableItem = (BillableItem)obj;
                    billableItem.Display();
                }
                else
                    Console.WriteLine(obj.ToString());
            }
            Console.WriteLine();
        }
    }
    class BillableItem {
        public string  Description { get; set; }
        public decimal Amount      { get; set; }
        public BillableItem(string description, decimal amount) {
            Description = description;
            Amount      = amount;
        }
        public void Display() {
            Console.WriteLine("Pay " + Description + " bill:  "
                            + Amount.ToString("C"));
        }
    }
    class GroceryItem {
        public string  Description { get; set; }
        public int     Quantity    { get; set; }
        public GroceryItem(string description, int quantity) {
            Description = description;
            Quantity    = quantity;
        }
        public void Display() {
            Console.WriteLine(Description + ":  " + Quantity);
        }
    }
}
```

5.

```
using System;
using System.Collections.Generic;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            KeyValuePair<int, string> kvp
            = new KeyValuePair<int, string>(1, "Apple");
            Console.WriteLine("They key is:  " + kvp.Key);
            Console.WriteLine("The value is: " + kvp.Value);
            Console.ReadLine();
        }
    }
}
```

6.

```
using System;
using System.Collections.Generic;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            Dictionary<string, string> books
                                       = new Dictionary<string, string>();
            books.Add("978-0071809375",
                      "JavaScript, Fourth Edition: A Beginner's Guide");
            books.Add("978-0071817912", "jQuery: A Beginner's Guide");
            foreach (KeyValuePair<string, string> book in books)
                Console.WriteLine("ISBN: " + book.Key + "  Title: "
                                           + book.Value);
            Console.ReadLine();
        }
    }
}
```

7.

```
using System;
using System.Collections;

namespace Starter {
    class Program {
        public static void Main( ) {
            Hashtable documents = new Hashtable();
            documents.Add("Invoice: 208", 44.33m);
            documents.Add("Invoice: 344", 34.22m);
            documents.Add("YRC", "Packing List - Jan. 28");

            foreach (DictionaryEntry license in documents) {
                Console.Write("Key: " + license.Key);
                Console.WriteLine("   Value: " + license.Value);
            }
            Console.ReadLine();
        }
    }
}
```

Chapter 11

1.

```
using System;

namespace Starter {
    public class Program {
        delegate string FormatFloat(float input);
        public static void Main() {
            FormatFloat formatFloat = new FormatFloat(RoundNumber);
            Console.WriteLine(formatFloat(3.1415f));
            Console.ReadLine();
        }
        public static string RoundNumber(float input) {
            return input.ToString("N2");
        }
    }
}
```

2.

```
using System;
namespace Starter {
    public class Program {
        delegate string FormatFloat(float input);
        public static void Main() {
            FormatFloat formatFloat = (input) => input.ToString("N2");
            Console.WriteLine(formatFloat(3.1415f));
            Console.ReadLine();
        }
    }
}
```

3.

```
using System;
namespace Starter {
    public class Program {
        public static void Main() {
           Func<float, string> formatFloat = (input) => input.ToString("N2");
            Console.WriteLine(formatFloat(3.1415f));
            Console.ReadLine();
        }
    }
}
```

4. Input before:

```
StartOutputCallback - Starting output!
I am a subscriber.
EndOutputCallback   - Ending output!

I am not a subscriber.
```

Input after:

```
StartOutputCallback - Starting output!
I am a subscriber.
EndOutputCallback   - Ending output!

I am not a subscriber.
EndOutputCallback   - Ending output!
```

5.

```
using System;

namespace Starter {
    public class Program {
        public static void Main() {
            Input input      = new Input();
            input.UserInput += InputCallback;
            input.GetUserInput();
            Console.ReadLine();
        }
        public static void InputCallback(string message) {
            Console.WriteLine(message);
        }
    }
    public class Input {
        public delegate void InputNotification(string input);
        public event         InputNotification UserInput;

        public void GetUserInput() {
            while (true) {
                Console.WriteLine(
                "Type any characters or 'q' to quit and press enter. ");
                string input = Console.ReadLine();
                if (UserInput != null) {
                    if (input.Trim() != "q")
                        UserInput("You typed: " + input);
                    else {
                        UserInput("You typed 'q' to quit.");
                        break;
                    }
                }
            }
        }
    }
}
```

Chapter 12

1. All interface members are public, so these members can be accessed by a consumer.
2.

```
using System;
using System.Collections.Generic;

namespace Starter {
    class Program {
        public static void Main( ) {
            Savings account = new Savings(123, 150.00m);
            account.ShowBalance();
            account.DeductMonthlyCharge();
            account.ShowBalance();
            Console.ReadLine();
        }
    }
```

```
        interface IAccount {
            decimal Balance   { get; set; }
            int     AccountID { get; }
            void    DeductMonthlyCharge();
            void    ShowBalance();
        }
        class Savings : IAccount {
            public decimal Balance   { get; set; }
            public int     AccountID { get; private set; }

            public Savings(int accountID, decimal balance) {
                AccountID = accountID;
                Balance   = balance;
            }
            public void DeductMonthlyCharge() {
                const decimal MONTHLY_CHARGE = 15.0m;
                Balance -= MONTHLY_CHARGE;
            }
            public void ShowBalance() {
                Console.WriteLine("Balance for account #"
                                  + AccountID + " = "
                                  + Balance.ToString("C"));
            }
        }
    }
```

3. Replace

```
Employee  employee   =  new Employee("Treele", "Grumbus");
```

with

```
IPerson   employee   =  new Employee("Treele", "Grumbus");
```

4.

```
using System;

namespace Starter {
    class Program {
        public static void Main() {
            Staff staff = new Staff(1, "Paolo", "Morselli",
                                   "pmorselli@acme.com",
                                    new DateTime(1968, 1, 6));
            staff.DisplayEmployeeData();
            Console.ReadLine();
        }
    }
    interface IPerson {              // IPerson is at the top of the hierarchy.
        string FirstName { get; set; }
        string LastName  { get; set; }
        DateTime Birthdate { get; }
    }
    interface IEmployee : IPerson {  // IEmployee extends IPerson.
        int EmployeeID { get; set; }
        string Email   { get; }
        void DisplayEmployeeData();
    }
    class Staff : IEmployee {        // This class implements IEmployee.
        public string  FirstName  { get; set; }
        public string  LastName   { get; set; }
        public int     EmployeeID { get; set; }
        public string  Email      { get; private set; }
        public DateTime Birthdate { get; private set; }

        public Staff(int id, string firstName, string lastName,
                     string email, DateTime birthdate) {
            FirstName  = firstName;
            LastName   = lastName;
            EmployeeID = id;
            Email      = email;
            Birthdate  = birthdate;
        }
        public void DisplayEmployeeData() {
           Console.WriteLine(EmployeeID + ": " + FirstName + " " + LastName);
            Console.WriteLine("Email: " + Email);
            Console.WriteLine("Birthdate: " + Birthdate.ToString("D"));
        }
    }
}
```

5.

```
using System;
using System.Collections.Generic;

namespace Starter {
    class Program {
        public static void Main( ) {
            Inspection<string> homeInspection = new Inspection<string>();
            homeInspection.Items.Add(
                          "No visible leaks from bathroom faucets.");
            homeInspection.Items.Add(
                          "No visible leaks from kitchen faucets.");
            homeInspection.Items.Add("No visible leaks from shower heads.");

            Console.WriteLine("Sample Date: " +
                              homeInspection.Time.ToString("f"));
            foreach (string sample in homeInspection.Items)
```

```
                Console.WriteLine(sample);
            Console.ReadLine();
        }
    }
    interface IInspection<T> {
        DateTime Time   { get; set; }
        List<T>  Items  { get; set; }
    }
    class Inspection<T> : IInspection<T> {
        public DateTime Time   { get; set;}
        public List<T>  Items  { get; set; }

        public Inspection() {
            Time   = DateTime.Now;
            Items  = new List<T>();
        }
    }
}
```

6. Change the *CompareTo()* method to either

```
public int CompareTo(Product product) {
    return product.Name.CompareTo(Name);  // Sort by Name property.
}
```

or

```
public int CompareTo(Product product) {
    return -Name.CompareTo(product.Name);  // Sort by Name property.
}
```

7.

```
public int CompareTo(Product product) {
    return Price.CompareTo(product.Price);  // Sort by Price property.
}
```

Chapter 13

1.

vendor	supplier_email
GFS	jane@gfs.com
Sysco	frank@sysco.com

2. vendor:
   - datatype:varchar
   - length:25
   - nullable:false
   - supplier_email:
   - datatype:varchar
   - length:30
   - nullable:false

Chapter 14

1. The Microsoft Entity Framework is called an object-relational mapping framework because it generates classes that map to database objects.
2. **A.** The *DbContext* class stores information about the database connection and references *DbSet* objects that are database entity collections.
   B. An entity is a class that is generated to represent a database table.
   C. A *DbSet* class instance defines a property that represents the collection of objects of an entity.
3. **A.** False; **B.** True; **C.** True; **D.** True
4.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Declare the DbContext.
            FoodStoreEntities    context = new FoodStoreEntities();

            // Get all Store objects in the DbSet named Stores.
            var stores = context.Stores;

            // Display details for each Store object in the Stores DbSet.
            foreach (Store productObject in stores) {
                Console.Write(productObject.branch  + " - ");
                Console.WriteLine(productObject.region);
            }
```

```
                Console.ReadLine();
            }
        }
    }
```

5. **A.** Lazy; **B.** Immediate; **C.** Lazy: **D.** Immediate; **E.** Lazy; **F.** Immediate; **G.** Immediate
6. Lazy loading allows you to build a complex query in separate components before running it. It only executes when data is needed.
7. Immediate loading can issue requests for data that is needed quickly. This can help to reduce load times for large queries.

Chapter 15

1.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = from i in context.Invoices               // Select columns
            select new { i.invoiceNum, i.branch };   // from Invoice table.

foreach (var invoice in query)                       // Show rows.
    Console.WriteLine(invoice.invoiceNum + " - " + invoice.branch);
Console.ReadLine();
```

2.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = from i in context.Invoices               // Select all columns
            select i;                                // from Invoice table.

foreach (var invoice in query)                       // Show rows.
    Console.WriteLine(invoice.invoiceNum + " - " + invoice.branch);
Console.ReadLine();
```

3. The compiler is not able to convert objects stored in the anonymous type to *Product* objects.
4. Yes, the execution is deferred. *IQueryable* queries do not force immediate data loading.
5.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = from p in context.Products
            where p.mfg == "Duncan Hines"            // Apply filter.
            select new { p.productID, p.name, p.price, p.mfg };
foreach (var item in query) {                        // Show rows.
    decimal price   = (decimal)item.price;
    string priceStr = price.ToString("C");
    Console.WriteLine(item.productID + " - " + item.name + " - "
                    + priceStr + " - " + item.mfg);
}
Console.ReadLine();
```

6.

```
FoodStoreEntities context = new FoodStoreEntities();
var query = ( from p in context.Employees
              where p.employee_id == 9001
              select new { p.employee_id, p.first_name, p.last_name })
                                                      .FirstOrDefault();
Console.WriteLine(query.employee_id + " " + query.first_name + " "
                + query.last_name);
Console.ReadLine();
```

7.

```
FoodStoreEntities context = new FoodStoreEntities();
var query =   from s in context.Stores
              where s.region == "BC"
              select new { Branch = s.branch, Region = s.region};
foreach (var item in query)
    Console.WriteLine(item.Branch + " " + item.Region);
Console.ReadLine();
```

8.

```
string[] stores
       = new string[]{ "Mission", "Vancouver", "Seattle" }; // Define array.
FoodStoreEntities context = new FoodStoreEntities();
var query =   from  s in context.Stores
              where stores.Contains(s.branch)               // Apply filter.
              select new {s.branch, s.region, s.building_name};
foreach (var item in query)                                 // Show data.
    Console.WriteLine(item.building_name + ", " + item.branch + ", " +
                      item.region);
Console.ReadLine();
```

9.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = from s in context.Stores
            orderby s.branch  descending // Order (descending) by Branch.
            select new { Branch = s.branch, Region = s.region };
foreach (var store in query)             // Show results.
    Console.WriteLine(store.Branch   + " - " + store.Region);
Console.ReadLine();
```

10.

```
FoodStoreEntities context = new FoodStoreEntities();
var query = from s in context.Stores
            orderby s.region, s.branch   // Order results
            select new { Branch = s.branch, Region = s.region };
foreach (var store in query)             // Show results
    Console.WriteLine(store.Branch   + " - " + store.Region);
Console.ReadLine();
```

11.

```
// Define string array.
string[] manufacturers = new string[]
        { "Florida Orange", "California Orange", "California Gold" };

// Only select values from the array that contain "Orange".
var mfgQuery = from m in manufacturers
               where m.Contains("Orange") select m;

FoodStoreEntities context = new FoodStoreEntities();

// Select from Manufacturer where names exist in the filtered array set.
var query = from m in context.Manufacturers
            where mfgQuery.Contains(m.mfg)
            select new { m.mfg, m.mfgDiscount};
            foreach (var item in query)
                Console.WriteLine(item.mfg + " " + item.mfgDiscount);
Console.ReadLine();
```

12.

```
FoodStoreEntities context = new FoodStoreEntities();
int              mfgCount = context.Manufacturers.Count();
Console.WriteLine("Manufacturer counter = " + mfgCount);
Console.ReadLine();
```

13.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var vendorCounts = from  s in context.Stores
                   group s by s.region into summary  // Set group column.
                   select new {                      // Define columns.
                   Region     = summary.Key,         // Set group column.
                   StoreCount = summary.Count() };   // Get count per group.
foreach (var item in vendorCounts)                   // Show results
    Console.WriteLine(item.Region + ": " + item.StoreCount);
Console.ReadLine();
```

14.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
try {
    Manufacturer mfg    = new Manufacturer();          // Create Manufacturer.
    mfg.mfg             = "Ichiban";                   // Initialize object.
    mfg.mfgDiscount     = 10;
    context.Manufacturers.Add(mfg);                    // Add Manufacturer.
    context.SaveChanges();                             // Commit changes.
}
catch (Exception e) {
    Console.WriteLine(e.Message  // Inform user if object is not added.
  + " Manufacturer objects with duplicate mfg values are not permitted.");
}
var manufacturers = context.Manufacturers.ToList();  // Show Manufacturers.
foreach (Manufacturer item in manufacturers)
    Console.WriteLine(item.mfg + " - " + item.mfgDiscount);
Console.ReadLine();
```

15.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
Manufacturer mfg = (from   p in context.Manufacturers
                    where  p.mfg == "Ichiban"
                    select p).FirstOrDefault();
try {
    context.Manufacturers.Remove(mfg);                 // Delete Manufacturer.
    context.SaveChanges();                             // Commit changes.
}
catch (Exception e) {
    Console.WriteLine(e.Message  // Inform user if object is not deleted.
  + " Unable to remove item if it does not exist.");
}
var manufacturers = context.Manufacturers.ToList();  // Show Manufacturers.
foreach (Manufacturer item in manufacturers)
    Console.WriteLine(item.mfg + " - " + item.mfgDiscount);
Console.ReadLine();
```

Chapter 16

1. **A.** Deferred execution; **B.** Immediate loading

2. **A.** The name of the inferred parameter is *p*. **B.** The parameter is a *Product* object.

3.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = from e in context.Employees              // Select employees.
            select new { First = e.first_name, Last = e.last_name };
```

```
foreach (var item in query)                          // Display employees.
    Console.WriteLine(item.First + " " + item.Last);
Console.ReadLine();
```

4.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query =  context.Employees                       // Select employees.
                    .Select( e=>new {First=e.first_name, Last=e.last_name});
foreach (var item in query)                          // Display employees.
    Console.WriteLine(item.First + " " + item.Last);
Console.ReadLine();
```

5.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var          query = context.Products                // Query Products.
            .Where(p=>p.name != "Cake Mix" && p.name != "Cookie Dough");
foreach (var item in query) {                        // Show Product data.
    decimal price = (decimal)item.price;             // Enables formatting.
    Console.WriteLine(item.productID + " " + item.name + " Manufacturer-"
                    + item.mfg + " Vendor-" + item.vendor + " " +
                      price.ToString("C"));
}
Console.ReadLine();
```

6.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = context.Stores                           // Get filtered stores.
    .Select(s => new { Branch=s.branch, Region=s.region })
    .Where(s=>s.Region != "BC");
foreach(var item in query)                           // Show Store data.
    Console.WriteLine(item.Branch + ", " + item.Region);
Console.ReadLine();
```

7.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
// Must add a reference to the "System.Collections.Generic" namespace.
List<string> branchList = new List<string>();        // Initialize List.
             branchList.Add("Vancouver");            // Add to List.
             branchList.Add("Seattle");
var  query = context.Invoices
    .Where(inv=>branchList.Contains(inv.branch))     // Filter on List.
    .Select(  i => new { InvoiceNum = i.invoiceNum, Branch = i.branch });
foreach (var item in query)                          // Show query detail.
    Console.WriteLine(item.InvoiceNum + " " + item.Branch);
Console.ReadLine();
```

8.

```
string[] manufacturers                              // Define string array.
         = new string[] {"Duncan Hines", "Pillsbury"};
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var          query = context.Products                // Query Products.
            .Where(p=> manufacturers.Contains(p.mfg));
foreach (var item in query) {                        // Show Product data.
    decimal price = (decimal)item.price;             // Enables formatting.
    Console.WriteLine(item.productID + " " + item.name + " Manufacturer-"
                    + item.mfg + " Vendor-" + item.vendor + " " +
                      price.ToString("C"));
}
Console.ReadLine();
```

9.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = context.Employees                        // Query filtered data.
                   .Where(e => e.first_name.StartsWith("J"));
foreach (var item in query)                          // Show Employee data.
    Console.WriteLine(item.first_name + " " + item.last_name);
Console.ReadLine();
```

10.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var products = context.Products                      // Query products.
              .OrderByDescending(p => p.name)
              .ThenByDescending(p => p.vendor);
foreach (var p in products) {                        // Show detail.
    decimal price = (decimal)p.price;
    Console.WriteLine(p.productID + " - " + p.name + " - "
                    + p.vendor + " - " + price.ToString("C"));
}
Console.ReadLine();
```

11.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = context.Buildings                        // Query unique names.
                   .Select( b => new { b.building_name }).Distinct();
foreach (var item in query)                          // Show names.
    Console.WriteLine(item.building_name);
Console.ReadLine();
```

12.

```
string[] manufacturers = // Declare and initialize string array.
new string[] { "Florida Orange", "California Orange", "California Gold" };
var mfgQuery = manufacturers.Where(m => m.Contains("Orange"));
```

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = context.Manufacturers                    // Query Manufacturers.
                  .Select(m => new { m.mfg, m.mfgDiscount })
                  .Where(m => mfgQuery.Contains(m.mfg));
foreach (var item in query)                          // Show data.
    Console.WriteLine("Name: " + item.mfg +
                      "  Discount: " + item.mfgDiscount);
Console.ReadLine();
```

13. The *Union()* extension method can only join queries that each consist of the same number of columns with the same names.

14.

```
FoodStoreEntities context = new FoodStoreEntities();
int count = context.Employees.Count();
Console.WriteLine("The employee count is: " + count);
Console.ReadLine();
```

15.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = context.Stores
    .GroupBy(s => new { s.region },                  // Set grouping column.
                      ( key, s ) =>                  // Define key.
                 new { Region = key.region,          // Assign query column.
                        Count  = s.Count() });
foreach (var row in query)                           // Show results.
    Console.WriteLine("Region:" + row.Region + " Count:" + row.Count);
Console.ReadLine();
```

Chapter 17

1. **A.** *Store* is the parent entity. *PurchaseOrder* is the child entity. **B.** The column *branch* is the primary key for the *Store* entity, **C.** The column *po_num is the primary key for the PurchaseOrder entity,* **D.** Many; **E.** 0 or 1; **F.** *branch*
2. Yes
3. **A.**

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = from  s in context.Stores                // Query Stores.
            from  p in context.PurchaseOrders        // and PurchaseOrders.
            where s.branch == p.branch               // Join using the keys.
            select new { s.branch, s.region, p.po_num };
foreach (var row in query)                           // Display results.
    Console.WriteLine(row.branch + " - " + row.region + " - " + row.po_num);
Console.ReadLine();
```

B.

```
FoodStoreEntities context = new FoodStoreEntities();
var query =  context.Stores.Join(                 // Outer DbSet.
                 context.PurchaseOrders,          // Inner DbSet.
                 s => s.branch,                   // Join columns.
                 p => p.branch,
                 (s, p) => new { s, p })          // Entity references.
    // 'sp' is the combined Stores and Invoices DbSet from Join().
    .Select(sp => new { Branch = sp.s.branch,     // Define query columns.
                        Region = sp.s.region,
                        PO     = sp.p.po_num });
foreach (var row in query)                        // Display results.
    Console.WriteLine(row.Branch + " - " + row.Region + " - " + row.PO);
Console.ReadLine();
```

C.

```
FoodStoreEntities context = new FoodStoreEntities();   // Reference database.
var query  = context.PurchaseOrders       // Invoice has 0..1 matching Store.
            .Where(p => p.branch != null)// Ensure join column isn't null.
            .Select(result => new         // Define query columns.
           { result.branch, result.Store.region, result.po_num });
foreach (var item in query)               // Show results.
   Console.WriteLine(item.branch + " - " + item.region + " - " + item.po_num);
Console.ReadLine();
```

4.

```
FoodStoreEntities context = new FoodStoreEntities();
var query =  context.Invoices.Join(      // Outer DbSet.
                 context.Stores,         // Inner DbSet.
                 i => i.branch,          // Join columns.
                 s => s.branch,
                 (i, s) => new { i, s })            // Entity references.
    // 'si' is the combined Stores and Invoices DbSet from Join().
    .Where( si => si.i.branch == "Vancouver")       // Apply filter.
    .Select(si => new { Branch     = si.s.branch,   // Define query columns.
                        Region     = si.s.region,
                        InvoiceNum = si.i.invoiceNum });
foreach (var item in query)
    Console.WriteLine("Invoice "   + item.InvoiceNum.ToString() + ": "
                   + item.Branch + ", " + item.Region);
Console.ReadLine();
```

5.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
var query = from m in context.Manufacturers          // Left table rows.
            from p in context.Products               // Right table rows.
                     .Where(prod => prod.mfg == m.mfg) // Join columns.
                     .DefaultIfEmpty()  // Return null Product values if
                                        // no match found.
```

```
    select new {
        Branch     = m.mfg,
        Region     = m.mfgDiscount,
        // Store invoiceNum value if it exists otherwise assign 0.
        ProductNum =  (p.productID != null)? p.productID : 0 };
foreach (var item in query)             // Show results.
    Console.WriteLine(item.Branch + " - "
                    + item.Region + " - " + item.ProductNum);
Console.ReadLine();
```

6. When using navigation properties for outer joins, the *DbSet* used in the equal join query must be for the entity that has only one matching instance in a neighboring entity. A full outer join, however, requires a union of outer joins from each entity in the relationship. In a one-to-many relationship, one of the entities relates to many instances of the other, so a navigation property cannot be used for this outer join.

7.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.

// Outer join on Stores.
var allStores   = from s in context.Stores                   // Outer table.
                  from b in context.Buildings                // Inner table.
                  .Where(bldg => bldg.unit_num == s.unit_num // Join.
                     && bldg.building_name == s.building_name)
                  .DefaultIfEmpty() // Set nulls for inner table if no match.
    select new {
        Branch     = s.branch,
        Region     = s.region,
        // Set building name and unit if they exist otherwise set null or 0.
        Building   =  (b.building_name != null)? b.building_name : "null",
        UnitNum    =  (b.unit_num != null)? b.unit_num: 0 };

// Outer join on Buildings.
var allInvoices = from b in context.Buildings                   // Outer table.
                  from s in context.Stores                      // Inner table.
                  .Where(store => store.unit_num == b.unit_num  // Join.
                     && store.building_name == b.building_name)
                  .DefaultIfEmpty() // Set nulls for inner table if no match.
    select new {
        // Set branch and region if values exist. Otherwise assign null.
        Branch     = (s.branch != null)? s.branch : "null",
        Region     = (s.region != null)? s.region : "null",
        Building   =  b.building_name,
        UnitNum    =  b.unit_num };

// Full outer join.
var fullJoin = allStores.Union(allInvoices);
foreach (var item in fullJoin) // Show results.
    Console.WriteLine(item.Branch + " - " + item.Region + " - "
                    + item.Building + " - " + item.UnitNum);
Console.ReadLine();
```

8.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
Product product = ( from   p in context.Products     // Get Product object.
                    where  p.productID == 2
                    select p).FirstOrDefault();
PurchaseOrder order
               = ( from   po in context.PurchaseOrders // Get PurchaseOrder.
                   where  po.po_num == 102
                   select po).FirstOrDefault();
try {
    order.Products.Add(product);                     // Insert in link table.
    context.SaveChanges();                           // Save to database.
}
catch (Exception e) {
    Console.WriteLine("Exception: " + e.Message
            + " A duplicate entry may exist "
            + "or invalid primary-foreign key relationships may exist.");
}

var query = from po in context.PurchaseOrders // Show bridged data where
            where po.po_num == 102 select po; // po_num is 102.
foreach (var po in query)
    foreach (var p in po.Products)
        Console.WriteLine("po_num: " + po.po_num + "  " +
                          "productID: " + p.productID);
Console.ReadLine();
```

9.

```
FoodStoreEntities context = new FoodStoreEntities(); // Reference database.
// Get parent objects.
PurchaseOrder order = (from   po in context.PurchaseOrders
                       where  po.po_num == 102
                       select po).FirstOrDefault();
Product product = (from   p in context.Products
                   where  p.productID == 2
                   select p).FirstOrDefault();

if (order.Products.Contains(product)) {              // Check for nulls.
    order.Products.Remove(product);                  // Delete  object.
    context.SaveChanges();
}
var invoices = from po in context.PurchaseOrders
               where po.po_num == 102 select po;
foreach (var po in invoices)
    foreach (var p in po.Products)
        Console.WriteLine("Purchase Order: " + po.po_num  + "  "
                         +"productID: " + p.productID);
Console.ReadLine();
```

Chapter 18

1. Defining column names, and column types optionally, is required first to define the row structure before any rows can be stored.

2.

```
using System;
using System.Data;
using Starter.DataAccessLayer;

namespace Starter {
    class Program {
        static void ShowOutput(DataTable dt) {
            if (dt == null)
                return;

            foreach (DataRow row in dt.Rows) {
                foreach (DataColumn col in dt.Columns) {
                    Console.Write(row[col].ToString() + ", ");
                }
                Console.WriteLine();
            }
            Console.WriteLine();
        }

        public static void Main() {
            DataTable medicalDiscoveries = new DataTable();
            medicalDiscoveries.Columns.Add("Year", typeof(int));
            medicalDiscoveries.Columns.Add("Detail", typeof(string));
            medicalDiscoveries.Rows.Add(1895, "X-Ray");
            medicalDiscoveries.Rows.Add(1921, "Isolation of insulin");
            medicalDiscoveries.Rows.Add(1928, "Penicillin");
            ShowOutput(medicalDiscoveries);
            Console.ReadLine();
        }
    }
}
```

3. The *System.Configuration* assembly must be referenced. The namespace reference is needed so you can access the *ConfigurationManager* class that reads the connection string from the App.config file.

4. **A.** "The SQL is either invalid or your connection failed. Please check your App.config reference just in case: Object reference not set to an instance of an object."

   B. Change the connection string *name* reference to the following:

```
<connectionStrings>
<add name="NewConnection" connectionString="…" />
</connectionStrings>
```

5. Change the instruction in *Main()* from

```
DataTable dt = RunSQL("SELECT * FROM Product");
```

to

```
DataTable dt = RunSQL("SELECT * FROM Employee");
```

6.

```
static void ShowOutput(DataTable dt) {
    if (dt == null) {
        Console.WriteLine("Empty dataset: Check your SQL.");
        return;
    }
    foreach (DataRow row in dt.Rows) {
        Console.Write(row["productID"] + ", ");
        Console.Write(row["name"] + ", ");
        Console.Write(row["mfg"] + ", ");
        Console.Write(row["vendor"] + ", ");
        decimal price;
        decimal.TryParse(row["price"].ToString(), out price);
        Console.WriteLine(price.ToString("C") );
        Console.WriteLine();
    }
}
```

7.

	SqlDataAdapter	**SqlDataReader**
Function	Retrieves data to the application in one large data set.	Retrieves data to the application row by row.
Advantages	Offers forward and backward navigation of the entire result set. Automates the opening, use, and closure of the database connection. Reduces errors by automating connectivity management for live data connections and for .NET controls. Is efficient for smaller to medium-sized queries.	Has low memory requirement. Enables simultaneous database server and C# application processing for faster performance. Is efficient for large queries.
Disadvantages	Requires more memory.	Offers forward-only navigation of the result set. Requires manual handling for opening, using, and closing the database connection. Is more prone to errors. Offers little to no performance gain for small queries.

8. Add the following method to the *DBCommands* class:

```
public static DataTable SpFindProduct(int productID) {
    string[] parameterNames  =   { "@productID" };
    string[] parameterValues =   { productID.ToString() };
    return DBEngine.ExecProcedure( "spFindProduct",
                                   parameterNames,
                                   parameterValues);
}
```

   Then add the following instruction to *Main()*:

```
ShowOutput(DBCommands.SpFindProduct(1));
```

9. Changes to ReaderCommands.cs and Program.cs are required.

   ReaderCommands.cs:

```
using System;

using System.Data;
using System.Data.SqlClient;

namespace Starter.DataAccessLayer {
    class ReaderCommands {
        // Execute procedure with no parameters.
        public SqlDataReader SpGetAllProducts() {
            ReaderEngine readerEngine = new ReaderEngine();
            return
            readerEngine.ExecParameterlessProcedure("spGetAllProducts");
        }
        // Execute procedure with parameters.
        public SqlDataReader SpProductDetail(string name,
                                             string vendor) {
            ReaderEngine readerEngine = new ReaderEngine();
            const string NAME         =   "spProductDetail";
            string[] parameterNames   = { "@name", "@vendor" };
            string[] parameterValues  = { name, vendor };
            return readerEngine.ExecProcedure(NAME,
                                              parameterNames,
                                              parameterValues);
        }

        // *** NEW ***
        public SqlDataReader SpFindProduct(int productID) {
            ReaderEngine readerEngine = new ReaderEngine();
            const string NAME         =   "spFindProduct";
            string[] parameterNames   = { "@productID" };
            string[] parameterValues  = { productID.ToString() };
            return readerEngine.ExecProcedure(NAME,
                                              parameterNames,
                                              parameterValues);
        }
        // Run SQL only.
        public SqlDataReader RunSQL(string sql) {
            ReaderEngine readerEngine = new ReaderEngine();
            return readerEngine.ExecSQL(sql);
        }
    }
}
```

Program.cs:

```
using System;
using Starter.DataAccessLayer;
using System.Data.SqlClient;

namespace Starter {
    class Program {
        public static void Main() {
            ReaderCommands reader;

            // Execute a stored procedure with no parameters.
            reader = new ReaderCommands();
            ShowOutput(reader.SpGetAllProducts());

            // Execute a stored procedure with parameters.
            reader = new ReaderCommands();
            ShowOutput(reader.SpProductDetail("Orange Juice", "GFS"));

            // Execute SQL.
            reader = new ReaderCommands();
            ShowOutput(reader.RunSQL("SELECT * FROM Product"));

            // *** NEW ***
            reader = new ReaderCommands();
            ShowOutput(reader.SpFindProduct(1));

            Console.ReadLine();
        }

        // *** Modified ***
        static void ShowOutput(SqlDataReader reader) {
            // Retrieve and display rows from database one-by-one.
            while (reader!= null && reader.Read()) {
                // Reference database columns by name.
                int    id      = (int)reader["productID"];
                string productName  = (string) reader["name"];
                decimal price       = (decimal)reader["price"];
                Console.WriteLine(id + ", " + productName + ", "
                                + price.ToString("C"));
            }
            // Remember to close the reader so you can use it later.
            if(reader != null)
                reader.Close();
            Console.WriteLine();
        }
    }
}
```

Chapter 19

1. Change the statement that initializes the file from

```
using (StreamWriter sw = new StreamWriter(path, !APPEND)) { }
```

to

```
using (StreamWriter sw = new StreamWriter(path, APPEND)) { }
```

2.

```
using System;
using System.IO;

namespace Starter {
    class Program {
        public static void Main() {
            const string FILE_PATH = "../../names.txt";
            WriteToFile(FILE_PATH); // File path and name is two directories
            Console.ReadLine();     // above the 'bin' folder.
        }
        static void WriteToFile(string path) {
            try {
                // Create writer to add or append to a file.
                using (StreamReader sr  = new StreamReader(path)) {
                    while (!sr.EndOfStream) {
                        string contents = sr.ReadLine();
                        Console.WriteLine(contents);
                    }
                }
            }
            catch (Exception e) {   // Show error detail if any.
                Console.WriteLine("Read error:  " + e.Message);
            }
        }
    }
}
```

3. The error occurs because the *ReadInt32()* method only accesses a fraction of decimal data where the file pointer is positioned. Since the data is read sequentially, only reading a portion of a data block will offset the remaining read instructions. This offset causes the error.

4.

```
static void ReadBinaryData() {
    byte[] buffer = new byte[BUFFER_SIZE];
    try{
        using (FileStream fileStream
                          = new FileStream(FILE_PATH,FileMode.Open)) {
            const int OFFSET = 8; // Skip first two integers (8 bytes).
            fileStream.Seek(OFFSET, SeekOrigin.Begin); // Start at 3rd int.
            fileStream.Read(buffer, 0, buffer.Length); // Fill the buffer.
            ShowBufferContents(buffer);

            const int NEW_BUFFER_SIZE = 8;             // Size: 2 integers.
            const int NEW_OFFSET      = 4;             // Size: 1 integer.
            buffer = new byte[NEW_BUFFER_SIZE];
            fileStream.Seek(NEW_OFFSET, SeekOrigin.Current ); // 1 int after
                                                              // current
                                                              // spot.
            fileStream.Read(buffer, 0, buffer.Length);
            ShowBufferContents(buffer);
        }
    }
    catch (Exception e) {
        Console.WriteLine("Read error:  " + e.Message);
    }
}
```

Chapter 20

1. A.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\us_states.xml";
            XElement root         = XElement.Load(FILEPATH);

            // Query syntax.
            IEnumerable<XElement> querySyntax =
            from    e in root.Elements("state")
            select  e;
            Display(querySyntax, "* Query Syntax *");
            Console.ReadLine();
        }

        static void Display(IEnumerable<XElement> elements, string title) {
            Console.WriteLine(title);

            // Show the value of the 'name' element.
            foreach (XElement e in elements) {
                Console.Write(e.Element("capital").Element("city").Value);
                Console.WriteLine(", "     + e.Element("capital")
                               .Element("population").Value);
            }
            Console.WriteLine();
        }
    }
}
```

B.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\us_states.xml";
            XElement root         = XElement.Load(FILEPATH);

            // Method-based syntax.
            IEnumerable<XElement> methodBasedSyntax =
            root.Elements("state");
            Display(methodBasedSyntax, "* Method Based Syntax *");
            Console.ReadLine();
        }

        static void Display(IEnumerable<XElement> elements, string title) {
            Console.WriteLine(title);

            // Show the value of the 'name' element.
            foreach (XElement e in elements) {
                Console.Write(e.Element("capital").Element("city").Value);
                Console.WriteLine(", "     + e.Element("capital")
                               .Element("population").Value);
            }
            Console.WriteLine();
        }
    }
}
```

2. A.

```
using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\employees.xml";
            XElement root         = XElement.Load(FILEPATH);
            Console.WriteLine("Employee count: " +
                              root.Elements("employee").Count());
            Console.ReadLine();
        }
    }
}
```

B.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\employees.xml";
            XElement root         = XElement.Load(FILEPATH);
            IEnumerable<XElement> employees = root.Elements("employee");

            foreach (XElement employee in employees)
                Console.WriteLine(employee.Element("firstName").Value + " "
                                + employee.Element("lastName").Value);
            Console.ReadLine();
        }
    }
}
```

C.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
```

```
namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\employees.xml";
            XElement root         = XElement.Load(FILEPATH);
            IEnumerable<XElement> employees = root.Elements("employee")
                                              .Elements("certifications")
                                              .Elements("certification")
                                              .Where(e=>e.Value=="CA")
                                              .Ancestors("employee");
            foreach (XElement employee in employees) {
                Console.WriteLine(employee.Element("firstName").Value + " "
                                 + employee.Element("lastName").Value);
            }
            Console.ReadLine();
        }
    }
}
```

D.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\employees.xml";
            XElement root         = XElement.Load(FILEPATH);
            IEnumerable<XElement> employees = root.Elements("employee");
            foreach (XElement employee in employees) {
                Console.Write(employee.Element("firstName").Value + " "
                            + employee.Element("lastName").Value + "   ");
                IEnumerable<XElement> certifications =
                                      employee.Elements("certifications")
                                              .Elements("certification");
                foreach (XElement certification in certifications) {
                    Console.Write(certification.Value + " " );
                }
                Console.WriteLine();
            }
            Console.ReadLine();
        }
    }
}
```

E.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\employees.xml";
            XElement root         = XElement.Load(FILEPATH);
            IEnumerable<XElement> employees = root.Elements("employee");

            XElement newEmployee =
                new XElement("employee", new XAttribute("department",
                                                        "operations"),
                                         new XAttribute("id", 33),
                                         new XElement("firstName", "Don"),
                                         new XElement("lastName", "Martin"),
                                         new XElement("certifications",
                                         new XElement("certification","BA")));
            root.Add(newEmployee);
            root.Save(FILEPATH);

            foreach (XElement employee in employees) {
                Console.Write(employee.Element("firstName").Value + " "
                            + employee.Element("lastName").Value + "   ");
                IEnumerable<XElement> certifications
                                     = employee.Elements("certifications")
                                               .Elements("certification");
                foreach (XElement certification in certifications) {
                    Console.Write(certification.Value + " " );
                }
                Console.WriteLine();
            }
            Console.ReadLine();
        }
    }
}
```

F.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
                // Load the file.
                const string FILEPATH = "..\\..\\employees.xml";
                XElement root          = XElement.Load(FILEPATH);
                var employee           = root.Elements("employee")
               .Where(e => e.Attribute("id").Value == "26")
                             .FirstOrDefault();
                if (employee != null) {      // Delete element.
                    employee.Remove();
                    root.Save(FILEPATH);     // Save changes.
                }
                Console.Write(root);         // Show XML.
                Console.ReadLine();
            }
        }
    }
```

3.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            XElement cat = new XElement("cat","lion");

            // Load the file.
            const string FILEPATH = "..\\..\\animals.xml";
            XElement root         = XElement.Load(FILEPATH);
            XElement leopard      = root.Elements("felines").Elements("cat")
                                  .Where(e => e.Value == "leopard")
                                  .FirstOrDefault();
            if (leopard != null) {      // Add element.
                leopard.AddAfterSelf(cat);
                root.Save(FILEPATH);
            }
            Console.Write(root);        // Show XML.
            Console.ReadLine();
        }
    }
}
```

4.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;
```

```
namespace ConsoleApplication1 {
    class Program {
        static void Main() {
            // Load the file.
            const string FILEPATH = "..\\..\\animals.xml";
            XElement root         = XElement.Load(FILEPATH);
            XElement lion         = root.Elements("felines").
Elements("cat")
                                        .Where(e => e.Value == "lion")
                                        .FirstOrDefault();
            if (lion != null) {      // Update element.
                lion.Value = "cave lion";
                root.Save(FILEPATH);
            }
            Console.Write(root);     // Show XML.
            Console.ReadLine();
        }
    }
}
```

5.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Xml.Linq;
using System.Xml.Serialization;

namespace ConsoleApplication1 {
    public class Weather {
        [XmlAttribute("day")]
        public string Day  { get; set; }
        [XmlElement("high")]
        public int    High { get; set; }
        [XmlElement("low")]
        public int    Low  { get; set; }

        public Weather() { }
        public Weather(string day, int high, int low) {
            Day  = day;
            High = high;
            Low  = low;
        }
    }
    public class WeatherReport {
        [XmlElement("weather")]
        public List<Weather> report { get; set; }
        public WeatherReport() {
            report = new List<Weather>();
        }
    }

    class Program {
        static void Main() {
            WeatherReport weather      = new WeatherReport();
            weather.report.Add(new Weather("Monday", 85, 68));
            weather.report.Add(new Weather("Tuesday", 88, 71));

            // Load the file.
            const string FILEPATH      = "..\\..\\weather.xml";
            StringWriter stringWriter  = new StringWriter();
            XmlSerializer  serializer
            = new XmlSerializer(typeof(WeatherReport));
            XmlSerializerNamespaces ns = new XmlSerializerNamespaces();
            ns.Add("","");                          // Remove xml namespace.

            serializer.Serialize(stringWriter, weather, ns);

            string xml = stringWriter.ToString();   // Convert to string.
            XElement element = XElement.Parse(xml); // Convert to XML.
            element.Save(FILEPATH);                 // Save file.
            Console.WriteLine(element);             // Show XML.
            Console.ReadLine();
        }
    }
}
```

Chapter 21

1. **A.** Object; **B.** Array
2. **A**, **B**, and **C**:

```
using System;
using Newtonsoft.Json.Linq;

namespace Starter {
    class Program {
        public static void Main() {
            dynamic cafe      = new JObject();
            dynamic beverages = new JArray();

            // Create and add beverages to the beverages array.
            dynamic coffee = CreateBeverage("Coffee",1.95m,2.55m,2.95m);
            beverages.Add(coffee);
            dynamic frapp  = CreateBeverage("Frappucino",2.95m,3.65m,4.15m);
            beverages.Add(frapp);
            cafe.Drinks  = beverages; // Assign beverages array to parent.
            Console.WriteLine(cafe);  // Show the resulting JSON.
            DisplayMenu(cafe);        // Show the data with formatting.
            Console.ReadLine();
        }

        static JObject CreateBeverage(string name, decimal small,
                                      decimal medium, decimal large) {
            dynamic beverage = new JObject();
            beverage.Name    = name;          // Create and assign name.

            dynamic price    = new JObject(); // Create and assign prices.
            price.Tall       = small;
            price.Grande     = medium;
            price.Venti      = large;
            beverage.Price   = price;
            return beverage;
        }

        static void DisplayMenu(dynamic cafe) {
            for(int i=0; i<cafe.Drinks.Count; i++) {
                dynamic drink = new JObject();
                drink = cafe.Drinks[i];
                Console.WriteLine();
                Console.WriteLine(drink.Name + " Prices");
                Console.Write("Tall: "
                            + drink.Price.Tall.ToString("C") + " ");
                Console.Write("Grande: "
                            + drink.Price.Grande.ToString("C") + " ");
                Console.WriteLine("Venti: "
                            + drink.Price.Venti.ToString("C"));
            }
        }
    }
}
```

D.

```
using System;
using Newtonsoft.Json.Linq;
using System.Collections.Generic;
using Newtonsoft.Json;

namespace Starter {
    public class PriceGroup {
        public decimal Tall    { get; set; }
        public decimal Grande  { get; set; }
        public decimal Venti   { get; set; }
        public PriceGroup()    { }
    }
    public class Beverage {
        public string     Name { get; set; }
```

```
            public PriceGroup Price;
            public Beverage() {
                Price = new PriceGroup();
            }
        }
        public class Menu {
            public List<Beverage> Drinks {get;set;}
            public Menu() {
                Drinks = new List<Beverage>();
            }
        }
        class Program {
            public static void Main() {
                dynamic cafe      = new JObject();
                dynamic beverages = new JArray();

                // Create and add beverages to the beverages array.
                dynamic coffee = CreateBeverage("Coffee",1.95m,2.55m,2.95m);
                beverages.Add(coffee);
                dynamic frapp  = CreateBeverage("Frappucino",2.95m,3.65m,4.15m);
                beverages.Add(frapp);
                cafe.Drinks    = beverages; // Assign array to parent.
                ConvertToCSharp(cafe);
                Console.ReadLine();
            }
            static void ConvertToCSharp(dynamic beverages) {
                string json = JsonConvert.SerializeObject(beverages);
                Menu drinks = JsonConvert.DeserializeObject<Menu>(json);
                foreach (Beverage beverage in drinks.Drinks) {
                    Console.WriteLine(beverage.Name + " ");
                    Console.Write("Tall: " + beverage.Price.Tall + " ");
                    Console.Write("Grande: " + beverage.Price.Grande + " ");
                    Console.WriteLine("Venti: " + beverage.Price.Venti + " ");
                    Console.WriteLine();
                }
            }

            static JObject CreateBeverage(string name, decimal small,
                                          decimal medium, decimal large) {
                dynamic beverage = new JObject();
                beverage.Name    = name;          // Create and assign name.

                dynamic price    = new JObject(); // Create and assign prices.
                price.Tall       = small;
                price.Grande     = medium;
                price.Venti      = large;
                beverage.Price   = price;
                return beverage;
            }
        }
    }
```

3.

```
using System;
using Newtonsoft.Json.Linq;

namespace Starter {
    class Program {
        public static void Main() {
            // Create parent JSON object.
            dynamic coffee = new JObject();
            dynamic prices = new JArray();  // Create array for prices.

            dynamic tall   = new JObject(); // Add first price to array.
            tall.Price     = 1.95m;
            prices.Add(tall);

            dynamic grande = new JObject(); // Add second price to array.
            grande.Price   = 2.95m;
            prices.Add(grande);

            dynamic venti  = new JObject(); // Add third price to array.
            venti.Price    = 3.95m;
            prices.Add(venti);

            coffee.Prices = prices;     // Append array to coffee object.
            Console.WriteLine(coffee);  // Show the JSON.
            Console.ReadLine();
        }
    }
}
```

Chapter 22

1. A. True; **B.** True; **C.** True

2.

```
using System;
using System.Reflection;
using System.Text.RegularExpressions;

namespace ConsoleApplication1 {
    static class Program {
        public static void Main() {
            ShowValidationStatus(new Person("John", "Dillinger")); // Valid.
            ShowValidationStatus(new Person("John", "32333"));     // Invalid.
            Console.ReadLine();
        }

        static void ShowValidationStatus(Person name) {
            bool valid = Validation.IsValid(name);
            Console.Write(name.First + " " + name.Last);
            if(valid)
                Console.WriteLine(" is valid.");
            else
                Console.WriteLine(" is not valid.");
        }
    }

    // Define custom attribute with optional AttributeUsage flags.
    [AttributeUsage(AttributeTargets.Property, // Target structure.
                    AllowMultiple = false,     // 1 instance allowed.
                    Inherited     = true)]     // Can't be inherited.
    public class MyRegex : Attribute {
        public string Pattern { get; private set;}
        public MyRegex(string pattern) {
            Pattern = pattern;
        }
    }

    public class Person {
        [MyRegex("^[a-zA-Z]+$")]           // Apply attribute to only permit
        public string First { get; set; } // upper or lower case letters.
        [MyRegex("^[a-zA-Z]+$")]
        public string Last { get; set; }

        public Person(string firstName, string lastName) {
            First = firstName;
            Last  = lastName;
        }
    }

    public class Validation {
        static bool CheckRegex(Object obj, PropertyInfo property) {

            // Select attributes & value where MyRegex attribute found.
            var attributes = property.GetCustomAttributes(typeof(MyRegex));

            // Iterate through 'MyRegex' attributes.
            foreach(Attribute attribute in attributes) {

                // Convert value with 'MyRegex' attribute to a string.
                var  propertyValue = property.GetValue(obj, null);
                string  personName = propertyValue.ToString();

                // Get pattern from MyRegex attribute.
                MyRegex MyRegex = (MyRegex)attribute;
                string    pattern   = MyRegex.Pattern;

                // Ensure property value conforms to the pattern.
                if(!Regex.IsMatch(personName, pattern))
                    return false;
            }
```

```
                return true;
            }

            public static bool IsValid(object obj) {
                var classType  = obj.GetType();
                var properties = classType.GetProperties();  // Get properties.

                // Search properties and apply regex where 'MyRegex' applies.
                foreach (var property in properties) {
                    if (!CheckRegex(obj, property))
                        return false;
                }
                return true;
            }
        }
    }
```

3.

```
using System;
using System.Reflection;
using System.Text.RegularExpressions;

namespace ConsoleApplication1 {
    static class Program {
        public static void Main() {
            ShowValidationStatus(new Product(10.0f)); // Valid instance.
            ShowValidationStatus(new Product(12.0f)); // Invalid instance.
            ShowValidationStatus(new Product(-4.5f)); // Invalid instance.
            Console.ReadLine();
        }

        static void ShowValidationStatus(Product name) {
            bool valid = Validation.IsValid(name);
            Console.Write(name.Price);
            if(valid)
                Console.WriteLine(" is valid.");
            else
                Console.WriteLine(" is not valid.");
        }
    }

    // Define custom attribute with optional AttributeUsage flags.
    [AttributeUsage(AttributeTargets.Property, // Target structure.
                    AllowMultiple = false,     // 1 instance allowed.
                    Inherited     = true)]     // Can't be inherited.
    public class FloatMaximum : Attribute {
        public float Maximum { get; private set;}
        public FloatMaximum(float maximum) {

            Maximum = maximum;
        }
    }
    public class Product {
        [FloatMaximum(11.50f)]                 // 0.0f <= property <= 11.50f
        public float Price { get; set; }
        public Product(float price) {
            Price = price;
        }
    }

    public class Validation {
        static bool IsLessThanMax(Object obj, PropertyInfo property) {

            // Select attributes & value where FloatMaximum attribute found.
            var attributes = property
                           .GetCustomAttributes(typeof(FloatMaximum));

            // Iterate through 'FloatMaximum' attributes.
            foreach(Attribute attribute in attributes) {

               // Convert property with 'FloatMaximum' attribute to a string.
                var   propertyValue = property.GetValue(obj, null);
                string  strProperty = propertyValue.ToString();
                float   propertyVal;
                if (!float.TryParse(strProperty, out propertyVal))
                    return false; // Return false when not a float.

                // Get maximum from FloatMaximum attribute.
                FloatMaximum floatMaximum = (FloatMaximum)attribute;
                float        attributeMax = floatMaximum.Maximum;

                // Ensure value is greater than 0 and less than maximum.
                if(propertyVal >= 0 && propertyVal <= attributeMax)
                    return true;
                return false;
            }
            return true;
        }

        public static bool IsValid(object obj) {
            var classType  = obj.GetType();
            var properties = classType.GetProperties();  // Get properties.

            // Search properties and use regex where 'FloatMaximum' applies.
            foreach (var property in properties) {
                if (IsLessThanMax(obj, property))
                    return true;
            }
            return false;
        }
    }
}
```

Chapter 23

1. **A.** True; **B.** False; **C.** False; **D.** True
2.

```
using System;

namespace ConsoleApplication1 {
    class Program {
        static void Main() {

            Budget retail = new Budget(88000m, 46700m); // Create budget.
            retail.ShowBudget("Retail Budget:");

            Budget online = new Budget(43000m, 2300m);  // Create budget.
            online.ShowBudget("Online Budget:");

            if (retail >= online)
                Console.WriteLine("Retail has an equal or higher profit.");
            else
                Console.WriteLine("Online has an equal or higher profit.");
            Console.WriteLine();

            Console.ReadLine();
        }
    }

    class Budget {
        public decimal Revenue  { get; set; }
        public decimal Expenses { get; set; }

        public Budget() { }
        public Budget(decimal revenue, decimal expenses) {
            Revenue  = revenue;
            Expenses = expenses;
        }
        // 'true' operator ensures Budget properties are not zero.
        public static bool operator true(Budget budget) {
            if (budget.Revenue == 0 && budget.Expenses == 0)
                return false;
            return true;
        }
        // 'false' operator only required if 'true' operator is present.
        public static bool operator false(Budget budget) {
            return false;
        }
        static decimal Compare(Budget a, Budget b) {
            decimal profitA = a.Revenue - a.Expenses;
            decimal profitB = b.Revenue - b.Expenses;
            return  profitA - profitB;
        }
        public static bool operator >=(Budget a, Budget b) {
            if(Compare(a,b) >= 0)
                return true;
            return false;
        }
        public static bool operator <=(Budget a, Budget b) {
            if(Compare(a,b) <= 0)
                return true;
            return false;
        }
        // '+' overload adds Budget objects.
        public static Budget operator +(Budget a, Budget b) {
            Budget sum   = new Budget();
            sum.Revenue  = a.Revenue  + b.Revenue;
            sum.Expenses = a.Expenses + b.Expenses;
            return sum;
        }
        // Show Budget properties.
        public void ShowBudget(string title) {
            Console.WriteLine(title);
            Console.WriteLine("Revenue:  " + Revenue.ToString("C"));
            Console.WriteLine("Expenses: " + Expenses.ToString("C"));
            Console.WriteLine();
        }
    }
}
```