New Mansoura University
Faculty of Computer Science and Engineering

Object-Oriented Programming

(CSE015)

Lecture Notes

Dr. Mohamed Handosa

Spring 2025

Contents

Chapter 1

Classes and Objects

The main difference between Object-Oriented Programming (OOP) and structured programming is that structured programming represents a program that uses a set of modules or functions, while OOP represents a program using a set of objects and their interactions.

OOP allows the programmer to represent real-world scenarios using objects. An object is any entity that has a *state* and a *behavior*. The **state** of an object is the set of attributes that represent the characteristics of that object. For example, a television has size, color, model,c. The **behavior** of an object is the set of operations that the object can perform. For example, a television can show picture, change channels, tune for a channel, etc. In OOP, objects can interact to achieve the program goals.

1.1 Basic Concepts

In OOP, a **class** is a template (or blueprint) for objects, and an **object** is an **instance** of a class. The following figure illustrates the difference between class and objects:

In real life, almost any object has a *state* and a *behavior*. The **state** of an object is defined by the values of its attributes, while the **behavior** is defined by the operations it can perform. A **class** is represented by a set of attributes and operations. For example, the attributes of a car include brand and color, while its operations include drive and brake.

When objects are created (or **instantiated**) from a class, each object will have all the attributes and operations of that class. However, each object will have its own state (i.e. attribute values). For example, we can create three objects from the `Car` class. Each `Car` object will have a `brand` attribute. However, the value of that attribute can differ from one `Car` object to another (e.g., Audi, Nissan, Volvo).

In C#, attributes are represented using variables (called fields), while operations are represented using methods.

1.2 Creating Classes

To create a class, use the `class` keyword. The following example creates a class named `Car` with an attribute named `color`:

```
class Car
{
    public string color;
}
```

When a variable is declared directly in a class, it is often referred to as a field (or attribute).

> It is a good practice to start with an uppercase first letter when naming classes. In addition, it is common for the name of the C# file and the class to match, as it organizes our code.

1.3 Creating Objects

An object is created from a class. To create an object, specify the class name, followed by the object name, and use the keyword `new`. The following example creates an object named `x` of type `Car`.

```
class Program
{
    static void Main()
    {
        Car x = new Car();
    }
}
```

You can create multiple objects of one class. The following example created two objects of type `Car`.

```
class Program
{
    static void Main()
    {
        Car x = new Car();
        Car y = new Car();
    }
}
```

1.4 Reference-Type Variables

There are two types of variables, which are value-type and reference-type variables. A **value-type** variable stores a value in its memory

location while a **reference-type** variable stores a reference in its memory location. For example:

```
1  static void Main()
2  {
3      int x = 5;
4      double y = 7.2;
5      int[] a = new int[10];
6      Car c = new Car();
7  }
```

In the above code, line 3 creates a variable named `x` of type `int` and assigns a value of 5 to it. Similarly, line 4 creates a variable named `y` of type `double` and assigns a value of 7.2 to it. Both `x` and `y` are value-type variables because they store actual values.

On the other hand, line 5 creates an array using the keyword `new` and then assigns its address to a variable named `a`. Therefore, the variable `a` is a reference-type variable because it holds a reference to the array (not the array itself). Similarly, line 6 creates an object using the keyword `new` and then assigns its address to a variable named `c`. Therefore, the variable `c` is a reference-type variable because it holds a reference to the object (not the object itself).

1.5 Memory Layout

Before a program can start execution, the operating system must first load it into memory. A program in execution is called a **process**. The layout of a process in memory consists of four sections:

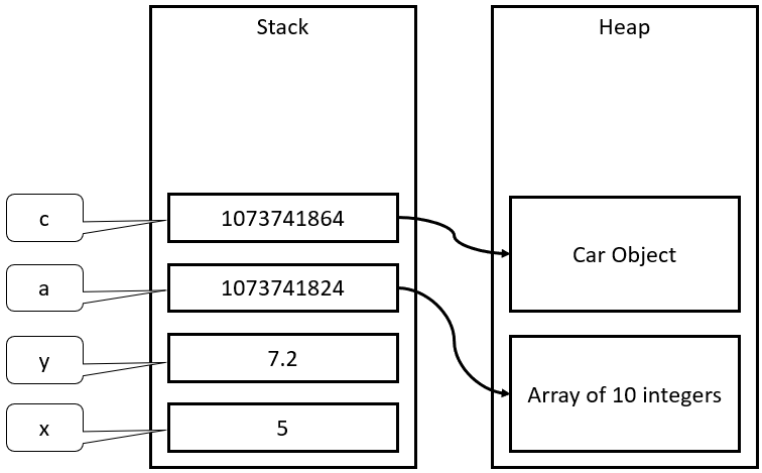
- **Text section:** stores instructions.
- **Data section:** stores global variables.
- **Stack:** stores automatic (i.e. local) variables.
- **Heap:** stores dynamically allocated data (e.g. arrays, objects).

For example, consider the following program:

```
1  static void Main()
2  {
3      int x = 5;
4      double y = 7.2;
5      int[] a = new int[10];
6      Car c = new Car();
7  }
```

In the above code, the variables `x`, `y`, `a`, and `c` are all local variables; and therefore, they are stored in the stack section. On the

other hand, the array itself (line 5) and the object itself (line 6) are dynamically allocated data; and therefore they are stored in the heap section.



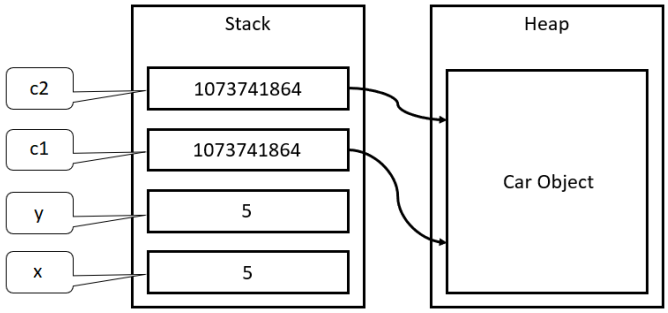
As shown in the above figure, the variables `x`, `y`, `a`, and `c` are stored in the stack section because they are all local variables while the array itself and the object itself are stored in the heap section. Note that the variables `x` and `y` are value-type variables storing the values 5 and 7.2 while the variables `a` and `c` are reference-type variables, where `a` stores a reference to the array (i.e. the address of the array in the heap section) and `c` stores a reference to the object (i.e. the address of the object in the heap section).

1.6 Reference variables and Assignment

In an assignment statement, the assignment operator `=` stores the value of the right-hand side into the variable on the left-hand side. For example,

```
1 static void Main()
2 {
3     int x = 5;
4     int y = x;
5     Car c1 = new Car();
6     Car c2 = c1;
7 }
```

In line 4, the assignment operator copies the contents of the variable `x` into the variable `y`. Therefore, the value of `y` becomes 5 as well. Similarly, in line 6, the assignment operator copies the contents of the variable `c1` into the variable `c2`. However, remember that the contents of `c1` is not the `Car` object but rather its address in the heap section. Therefore, `c2` becomes another reference to the same `Car` object. In other words, there is one `Car` object in the heap and there are two reference-type variables, `c1` and `c2`, in the stack. Both `c1` and `c2` are holding the same reference, which is the address of the `Car` object.


Stack
Heap
c2
1073741864
c1
1073741864
y
5
x
5
Car Object

Chapter 2

Class Members

Fields and methods inside classes are referred to as **class members**. The following example creates a `Player` class with two class members: one field and one method.

```
1  class Player
2  {
3      public string name; // Field
4
5      public void Run() // Method
6      {
7          System.Console.WriteLine("{0} is running.", name);
8      }
9  }
```

2.1 Fields

Variables inside a class are called fields, and you can access them by creating an object of the class, and by using the dot syntax `.`. The following example creates one object of the `Player` class, with the name `p1`:

```
1 static void Main()
2 {
3     Player p1 = new Player();
4     p1.name = "Salah";
5     System.Console.WriteLine(p1.name);
6 }
```

In the above code, line 4 assigns the value `Salah` to the `name` field of the `p1` object. Afterward, line 5 prints the value of the `name` field. The output should be:

Output

```
Salah
```

You can create any number of objects from a given class and each object will have its own copy of the fields. The values of the fields of a given object represent its state. For example, the following code creates two objects of the `Player` class:

```
1  static void Main()
2  {
3      Player p1 = new Player();
4      Player p2 = new Player();
5      p1.name = "Salah";
6      p2.name = "Marmoush";
7      System.Console.WriteLine(p1.name);
8      System.Console.WriteLine(p2.name);
9  }
```

In the above code, lines 3 and 4 create two new objects of type `Player`. Afterward, line 5 assigns the value `Salah` to the `name` field of the object named `p1` and line 6 assigns the value `Marmoush` to the `name` field of the object named `p2`. Finally, lines 7 and 8 print the value of the `name` field for each of the two objects. The output should be:

Output

```
Salah
Marmoush
```

2.2 Methods

Object methods define how an object of a class behaves. Just like with fields, you can access methods with the dot syntax. For example:

```
1  using System;
2  namespace Example
3  {
4      class Program
5      {
6          static void Main()
7          {
8              Player p1 = new Player();
9              p1.name = "Salah";
10             p1.Run();
11         }
12     }
13
14     class Player
15     {
16         public string name;
17
18         public void Run()
19         {
20             Console.WriteLine("{0} is running.", name);
21         }
22     }
23 }
```

In the above code, line 10 calls the `Run` method of the `p1` object. The output should be:

Output

```
Salah is running.
```

2.3 Static Methods

A static method belongs to the class rather than an object created from that class. A static member can be declared using the keyword `static` as in the following example:

```
1  namespace Example
2  {
3      class Program
4      {
5          static void Main()
6          {
7              double sum = Calculator.Add(5, 6);
8              System.Console.WriteLine(sum);
9          }
10     }
11     class Calculator
12     {
13         public static double Add(double x, double y)
14         {
15             return x + y;
16         }
17     }
18 }
```

In the above code, lines 15-18 define a static method named `Add`. In line 8, the `Main` method calls the `Add` method. Note that calling a static method does not require creating an object. A static method is called directly by using the class name itself.

2.4 Static Fields

A static field belongs to the class rather than an object created from that class. Static fields serve as global variables that are always in memory (in the data section) while the program is running and can be accessed from anywhere in the code at anytime. For example:

```
1  using System;
2
3  namespace Example
4  {
5      class Program
6      {
7          static void Main()
8          {
9              Reset();
10             Increment();
11             Console.WriteLine(Data.counter);
12         }
13
14         static void Reset() { Data.counter = 0; }
15
16         static void Increment() { Data.counter++; }
17     }
18
19     class Data
20     {
21         public static int counter;
22     }
23 }
```

In the above code, line 21 declares a static field named `counter`. Note that this field can be accessed from anywhere in the code. The `Reset` method can set its value to 0, the `Increment` method can add one to its value, and the `Main` method can print its value. Line 9 calls the `Reset` method that sets the `counter` value to 0. Line 10 calls the `Increment` method that changes the `counter` value to 1. Finally, line 11 prints the `counter` value to the screen. The output should be:

Output

```
1
```

2.5 Static Classes

A static class cannot be instantiated (i.e. cannot have objects). A static class contains only static members, which can be accessed by using the class name itself.

A static class can be used as a convenient container for a set of methods. For example, in the .NET Class Library, the static `System.Math` class contains methods that perform mathematical operations, without any requirement to create an object of the `Math` class. That is, you call the members of the class by specifying the class name and the method name, as shown in the following example.

```
1  class Program
2  {
3      static void Main()
4      {
5          int x = 5;
6          double y = System.Math.Pow(x, 2);
7          System.Console.WriteLine(y);
8      }
9  }
```

In the above code, line 7 calls the static method named `Pow` directly by using the static class named `Math` to calculate the value of `x` raised to 2. The output is 25 as shown below.

Output

```
25
```

The following example shows a static class with two methods that convert temperature from Fahrenheit to Celsius and vice versa:

```
1  static class TemperatureConverter
2  {
3      public static double ToCelsius(double fahrenheit)
4      {
5          double celsius = (fahrenheit - 32) * 5 / 9;
6          return celsius;
7      }
8
```

```
9       public static double ToFahrenheit(double celsius)
10      {
11          double fahrenheit = (celsius * 9 / 5) + 32;
12          return fahrenheit;
13      }
14  }
15
16  class Program
17  {
18      static void Main()
19      {
20          double x = TemperatureConverter.ToFahrenheit(100);
21          System.Console.WriteLine(x);
22          double y = TemperatureConverter.ToCelsius(x);
23          System.Console.WriteLine(y);
24      }
25  }
```

In the above code, line 1 defines the `TemperatureConverter` class as a static class using the `static` keyword. This means that you cannot create objects from that class. Note that all the members of the `TemperatureConverter` class must be static as well. The `Main` calls the static methods directly by using the class name and the output should be as follows.

Output

```
212
100
```

In summary, a static class cannot be instantiated and can have only static members. A non-static class can contain both static and non-static members. A static member is callable on a class even when no instance of the class has been created. The static member is always accessed by the class name, not the instance name. Only one copy of a static member exists, regardless of how many instances of the class are created. On the other hand, non-static members can be accessed only after creating an object and each object has its own copy of those non-static members.

2.6 Access Modifiers

By now, you are quite familiar with the `public` keyword that appears in many of the previous examples:

```
1 class Product
2 {
3     public float price;
4 }
```

The `public` keyword is an access modifier. An **access modifier** sets the access level or visibility of a class member. C# has the following access modifiers:

Modifier	Description
`public`	The member is accessible for all classes
`private`	The member is only accessible within the same class
`protected`	The member is accessible within the same class, or in a class that is inherited from that class.
`internal`	The member is only accessible within its own assembly, but not from another assembly.

2.6.1 Public Modifier

A `public` class member is accessible to all classes. For example:

```
1  namespace Market
2  {
3      class Product
4      {
5          public float price;
6      }
7
8      class Program
9      {
10         static void Main()
11         {
12             Product p = new Product();
13             p.price = 20;
14         }
15     }
16 }
```

In the above code, line 5 defines a `public` field named `price`. Line 13 can access the `public` field although it belongs to a method in a different class.

2.6.2 Private Modifier

A `private` class member is accessible only to its class. For example:

```
1  namespace Market
2  {
3      class Product
4      {
5          private float price;
6      }
7
8      class Program
9      {
10         static void Main()
11         {
12             Product p = new Product();
13             p.price = 20; // Error
14         }
15     }
16 }
```

In the above code, line 5 defines a `private` field named `price`. line 13 cannot access the `price` field because it belongs to a method

in a different class. Trying to compile this program will result in an error with the following message:

Error Message

'Product.price' is inaccessible due to its protection level

2.6.3 Default Access Modifier

All members of a class are `private` by default. For example:

```
1  class Product
2  {
3      string name; // private
4      float price; // private
5
6      void PrintData() // private
7      {
8          System.Console.WriteLine("{0}\t{1}", name, price);
9      }
10 }
```

In the above code, we did not specify any access modifiers. Therefore, the two fields and the method are all `private` members of the class `Product`. This means that other classes cannot access the two fields nor call the method. A `private` method can be called only from within its class (i.e. by another member of its class).

2.6.4 Accessing Private Members

A class can provide `public` members to allow for indirect access to its `private` members, as in the following example:

```
namespace Market
{
    class Product
    {
        private float price;
        public void SetPrice(float priceValue)
        {
            if(priceValue >= 0) price = priceValue;
        }

        public float GetPrice()
        {
            return price;
        }
    }

    class Program
    {
        static void Main()
        {
            Product p = new Product();
            p.SetPrice(20);
            Console.WriteLine(p.GetPrice());
        }
    }
}
```

In the above code, line 5 defines the `price` field to be `private`. This prevents other classes from accessing the `price` field. Yet, both `SetPrice` and `GetPrice` methods can access the `price` field because they belong to the same class.

The only way to access the `price` field from other classes is through the two `public` methods, `SetPrice` and `GetPrice`. Line 22 sets the value of the `price` field indirectly by calling `SetPrice` and line 23 gets the value of the `price` field by calling `GetPrice`.

In line 8, the `SetPrice` assigns the value of the `priceValue` parameter to the `price` field only if it is non-negative. This way we can protect the `price` field from having negative values.

Encapsulation (or data-hiding) allows for protecting data fields from having invalid values. It is a protective shield that prevents the internal data and functionality of an object from being accessed by the code outside this shield. Encapsulation can be achieved by declaring all the fields in the class as `private` and make those fields accessible only through the `public` methods of that class. This ensures controlled access to the fields.

Chapter 3

Constructors & Destructors

A constructor is a special method that is called when an object is created. It can be used to set the initial values for fields. For example:

```
1  class Product
2  {
3      public string name;
4      public float price;
5
6      public Product()
7      {
8          name = "Milk"
9          price = 10;
10     }
11 }
```

In the above code, lines 6-10 define a constructor that initializes the values of the fields `name` and `price` to `Milk` and `10`.

> Note that the constructor name must match the class name, and it cannot have a return type (like `void` or `int`).

The constructor is called when the object is created. For example, the following code creates an object of type `Product` named `p1`.

```
1  static void Main()
2  {
3      Product p1;
4      p1 = new Product();
5      System.Console.WriteLine(p1.name);
6      System.Console.WriteLine(p1.price);
7  }
```

In the above code, line 3 creates a variable named `p1` of type `Product`. The variable `p1` is a **reference-type** variable that is stored in the stack section and has an initial value of `null` (i.e. nothing). Line 4 creates a new object of type `Product` via the keyword `new` followed by a call to the constructor `Product()`. After creating the object in the heap section and executing the constructor, the address of the newly created object is assigned to the variable `p1` using the assignment operator `=`. This way `p1` holds a reference

to the object. Lines 5 and 6 use `p1` to access the fields of the object and print their values.

> All classes have constructors by default. If you do not create a class constructor yourself, C# creates one for you. The default constructor is parameter-less (i.e. it has no parameters).

3.1 Constructor Parameters

Constructors can also take parameters, which is used to initialize fields. The following example adds a `string` parameter named `productName` and a `float` parameter named `productPrice` to the constructor.

```
1  class Product
2  {
3      public string name;
4      public float price;
5
6      public Product(string productName, float productPrice)
7      {
8          name = productName;
9          price = productPrice;
10     }
11 }
```

Line 8 sets `name` to `productName` and line 9 sets `price` to `productPrice`. When we call the constructor, we pass two parameters to the constructor (`Eggs` and `40`), which will set the value of `name` to `Eggs` and the value of `price` to `40`.

```
1 static void Main()
2 {
3     Product p1 = new Product("Eggs", 40);
4     System.Console.WriteLine(p1.name);
5     System.Console.WriteLine(p1.price);
6 }
```

In the above code, line 3 creates an object of type `Product` and assigns its reference to the variable `p1`. Afterward, lines 4 and 5 use `p1` to access the fields of the object and print their values to the screen. The output should be as follows:

Output

```
Eggs
40
```

3.2 The this Reference

When we create an object, we store a reference to it in a reference-type variable. This way, we can use that variable to access object members.

The following example defines a class named `Product` and creates two objects from that class:

```
1  namespace Market
2  {
3      class Program
4      {
5          static void Main()
6          {
7              Product p1 = new Product(10);
8              Product p2 = new Product(40);
9              System.Console.WriteLine(p1.price);
10             System.Console.WriteLine(p2.price);
11         }
12     }
13
14     class Product
15     {
16         public float price;
17         public Product(float price)
18         {
19             price = price;    // Bug
20         }
21     }
22 }
```

Note the bug at line 19, where the constructor assigns the parameter to itself rather than to the field. Therefore, the output will be:

Output

```
0
0
```

The reason behind that bug is that the constructor parameter has the same name as as the field name (i.e. `price`). Within the constructor scope, the `price` parameter is a local variable that hides (or shadows) the `price` field. Therefore, the constructor cannot access the `price` field by simply stating its name.

One solution to this problem is to use the keyword `this`, which acts as a reference to the current object. This way, we can refer to the `price` field from within the constructor even if there is a local variable with the same name. Using the keyword `this`, we can fix the bug by modifying the code to be as follows:

```
17          public Product(float price)
18          {
19              this.price = price;     // Bug fix
20          }
```

In the above code, `this.price` refers to the field while `price` refers to the local variable (i.e. the parameter).

Note that we accessed the `price` field from the `Main` method using `p1.price` for the first object (line 11) and `p2.price` for the

second object (line 12). However, for any `Product` object, we can access its `price` field from within its constructor (or from within any other member method of the `Product` class) using `this.price`.

> Using the keyword `this` is *optional* if there is no local variable with the same name as the class member we are trying to access.

3.3 Overloaded Constructors

Just like other methods, constructors can be overloaded. The following example defines a class named `Product` with two fields and three overloaded constructors. One constructor has no parameters (lines 6-10), another has one parameter (lines 12-16), and the last constructor takes has two parameters (lines 18-22).

```
1  class Product
2  {
3      public string name;
4      public float price;
5
6      public Product()
7      {
8          name = "Milk";
9          price = 10;
10     }
```

```
11
12      public Product(string name)
13      {
14          this.name = name;
15          price = 10;
16      }
17
18      public Product(string name, float price)
19      {
20          this.name = name;
21          this.price = price;
22      }
23  }
```

A class with overloaded constructors allows for creating objects of that class using any of those constructors. For example, the following code creates three different objects, each using a different constructor.

```
1   class Program
2   {
3       static void Main()
4       {
5           Product p1 = new Product();
6           Product p2 = new Product("Cheese");
7           Product p3 = new Product("Eggs", 40);
8           Console.WriteLine("{0}\t{1}", p1.name, p1.price);
9           Console.WriteLine("{0}\t{1}", p2.name, p2.price);
10          Console.WriteLine("{0}\t{1}", p3.name, p3.price);
11      }
12  }
```

Line 5 creates a `Product` object using the parameter-less constructor. Line 6 creates a `Product` object using the single-parameter constructor. Line 7 creates a `Product` object using the two-parameters constructor. The output should be as follows:

Output

```
Milk        10
Cheese      10
Eggs        20
```

3.4 Constructor Initializers

Constructor initializers allow one constructor to call another constructor. A constructor can call another using the `this` keyword as in the following example:

```
1  class Product
2  {
3      public string name;
4      public float price;
5
6      public Product() : this("Milk", 10)
7      {
8
9      }
10
```

```
11      public Product(string name, float price)
12      {
13          this.name = name;
14          this.price = price;
15      }
16  }
```

In line 6, the parameterless `Product()` constructor calls the two-parameters `Product(string name, float price)` constructor using the keyword `this` followed by the arguments `("Milk", 10)`.

Constructor initializers help avoiding duplicate code, where one constructor can call another instead of having multiple copies of the same code in more than one constructor. The following example defines a class named `Product` with three constructors and creates three objects, each with a different constructor:

```
1   using System;
2   namespace Market
3   {
4       class Program
5       {
6           static void Main()
7           {
8               Console.WriteLine("Create an object using the
                ↪  2-parameter constructor.");
9               Product p1 = new Product("Eggs", 40);
10
11              Console.WriteLine("Create an object using the
                ↪  1-parameter constructor.");
```

```
12              Product p2 = new Product("Cheese");
13              Console.WriteLine();
14
15              Console.WriteLine("Create an object using the
                ↪  0-parameter constructor.");
16              Product p3 = new Product();
17              Console.WriteLine();
18          }
19      }
20
21      class Product
22      {
23          public string name;
24          public float price;
25
26          public Product() : this("Milk")
27          {
28              Console.WriteLine("0-parameters constructor.");
29          }
30
31          public Product(string name) : this(name, 10)
32          {
33              Console.WriteLine("1-parameter constructor.");
34          }
35
36          public Product(string name, float price)
37          {
38              this.name = name; this.price = price;
39              Console.WriteLine("2-parameters constructor.");
40          }
41      }
42  }
```

In the above code, using the parameterless constructor calls the one-parameter constructor and using the one-parameter constructor calls the two-parameters constructor. The output should be as follows:

Output

```
Create an object using the 2-parameter constructor.
2-parameters constructor.

Create an object using the 1-parameter constructor.
2-parameters constructor.
1-parameter constructor.

Create an object using the 0-parameter constructor.
2-parameters constructor.
1-parameter constructor.
0-parameters constructor.
```

3.5 Garbage Collection

A local variable is temporary. It is created in the stack section when defined and is removed automatically when it goes out of scope. Therefore, local variables are also called temporary variables or automatic variables. For example,

```
1  class Program
2  {
3      static void Main()
4      {
5          int x;
6          x = 5;
7
8          if(x > 3)
9          {
10             Player p;
11             p = new Player();
12         }
13     }
14 }
```

In the above code, line 5 defines a variable named `x`. When line 5 executes, the variable `x` is created in the stack and remain there until its scope ends at line 13, where it is removed automatically from the stack. Similarly, when line 10 executes, the variable `p` is created in the stack. When line 11 executes, a `Player` object is created in the heap by using the `new` operator and its address is assigned to `p`. Later, the variable `p` is removed from the stack when its scope ends at line 12. However, the `Player` object remains in the heap.

Unlike automatic variables that are created in the stack, objects and arrays are not removed automatically from the heap. Each time you create a new object, the required memory is allocated from the heap. However, memory is not infinite. Therefore, objects that are no longer

in use needs to be removed to free up some memory. The C++ language provides a `delete` operator that allows a programmer to manually remove an object from the heap. Unlike C++, C# do not provide a `delete` operator and uses a garbage collector instead.

Garbage collection relieves the programmer from performing manual memory cleanup. The .NET's garbage collector performs a collection from time to time in order to free some memory. When the garbage collector performs a collection, it checks for objects in the heap that are no longer being used by the application and destroys them to reclaim their memory. In C#, you can call the garbage collector manually and force it to start memory cleanup using the `GC` class as follows,

```
1 System.GC.Collect(); // Not recommended
```

Garbage collection may take a significant proportion of total processing time and can affect the program performance. Therefore, calling the garbage collector is not recommended. It is better to let it decide the best time to perform a collection.

3.6 Destructors

A **destructor** is used to perform any necessary final clean-up when a class instance (i.e. object) is being removed from memory.

A **destructor** is a special method that has the same name as its class but starts with a tilde [~]. Destructors cannot be overloaded and cannot be called. A class can have only one destructor. A destructor does not have an access modifier and does not take parameters. For example,

```
class Car
{
    ~Car()  // destructor
    {
        // cleanup statements...
    }
}
```

The programmer has no control over when the destructor is called. If the garbage collector considers an object eligible for destruction, it calls its destructor (if any) and reclaims the memory used by that object. If your object allocates resources, such as windows, files, and network connections, you should use a destructor to free those resources before the object is destroyed.

Chapter 4

Objects, Methods, & Arrays

You have been using value-type variables, such as int or double, as parameters to methods. However, it is both correct and common to use a reference type as a parameter. Doing so allows an object or an array to be passed to a method. Similarly, you have been using value-types to create arrays. For example,

```
1 int[] x = new int[10];
```

However, it is both correct and common to create reference-type arrays. Doing so allows for creating an array of objects (to be specific, an array of references to objects).

4.1 Passing References to Methods

References can be passed to methods. For example,

```
namespace School
{
    class Student
    {
        public string name;
        public float score;
    }

    class Program
    {
        static void Main()
        {
            Student x = new Student();
            x.name = "Ziad";
            x.score = 90;
            float y = 5;
            System.Console.WriteLine(x.score);
            AddBonus(x, y);
            System.Console.WriteLine(x.score);
        }

        static void AddBonus(Student s, float bonus)
        {
            s.score += bonus;
        }
    }
}
```

In the above code, line 17 prints the value of the `score` field (i.e. 90) to the screen. Line 18 calls the `AddBonus` method that takes two parameters. It passes the two arguments (`x` and `y`) to it. The arguments are passed by value. Therefore, the value of `x` is copied to `s` and the value of `y` is copied to `bonus`.

Line 24 uses the reference-type variable `s` (which is a copy of `x`) to access the same `Student` object and increment the current value of its `score` field (i.e. `90`) by the value stored in the *bonus* field (i.e. `5`). Line 19 prints the updated value of the `score` field (i.e. `95`) to screen. The output should be as follows.

Output

```
90
95
```

As you may already know, pass-by-value creates a copy of the passed argument. In the above example, the first argument is a reference to an object. Creating a copy of a reference does not copy the object itself. The `x` variable in the `Main` method and the `s` variable in the `AddBonus` method both refer to the same `Student` object. Therefore, using the reference-type variable `x` in lines 17 and 19, we were able to access the `Student` object and using the reference-type variable `s` in line 24, we were able to access the same `Student` object and update the value of its `score` field.

4.2 Passing References by Reference

Passing a reference-type variable by reference allows a method to change the object it refers to. For example,

```
using System;

namespace School
{
    class Student
    {
        public string name;
        public float score;

        public Student(string name, float score)
        {
            this.name = name;
            this.score = score;
        }

        public void PrintInfo()
        {
            Console.WriteLine("{0}:\t{1}", name, score);
        }
    }

    class Program
    {
        static void Main()
        {
            Student x = new Student("Ziad", 90);
            Student y = new Student("Malik", 95);
```

```
28              Console.WriteLine("Before swap:");
29              x.PrintInfo();
30              y.PrintInfo();
31              Swap(ref x, ref y);
32              Console.WriteLine("After swap:");
33              x.PrintInfo();
34              y.PrintInfo();
35          }
36
37          static void Swap(ref Student s1, ref Student s2)
38          {
39              Student temp = s1; s1 = s2; s2 = temp;
40          }
41      }
42  }
```

In the above code, line 31 passes the two reference-type variables `x` and `y` by reference to the `Swap` method. Therefore, we were able to modify `x` and `y` by modifying `s1` and `s2`. The output is:

Output

```
Before swap:
Ziad:                   90
Malik:                  95
After swap:
Malik:                  95
Ziad:                   90
```

4.3 Variable Number of Parameters

You can create a method that takes an arbitrary number of arguments using the `params` modifier. For example,

```
1  namespace Params
2  {
3      class Program
4      {
5          static void Main()
6          {
7              double x = GetAverage(5, 4);
8              double y = GetAverage(17, 4, 6, 45, 89);
9              System.Console.WriteLine("x={0}, y={1}", x, y);
10         }
11
12         static double GetAverage(params double[] values)
13         {
14             if (values.Length == 0) return 0;
15             else
16             {
17                 double sum = 0;
18                 for (int i = 0; i < values.Length; i++)
19                 {
20                     sum += values[i];
21                 }
22                 return sum / values.Length;
23             }
24         }
25     }
26 }
```

In the above code, line 12 uses the `params` modifier to declare an array parameter that will be able to receive zero or more arguments. The number of elements in the array will be equal to the number of arguments passed to the method. Line 7 passes two parameters to the `GetAverage` method while line 8 passes five parameters. The `GetAverage` method then accesses the array to obtain the arguments.

4.4 Returning Objects

A method can return an object of a given class. For example,

```
1  namespace Shapes
2  {
3      class Rectangle
4      {
5          public int width;
6          public int height;
7
8          public Rectangle() : this(0, 0)
9          {
10         }
11
12         public Rectangle(int width, int height)
13         {
14             this.width = width;
15             this.height = height;
16         }
```

```
17          public void Draw()
18          {
19              for (int y = 0; y < height; y++)
20              {
21                  for (float x = 0; x < width; x++)
22                  {
23                      System.Console.Write('*');
24                  }
25                  System.Console.WriteLine();
26              }
27          }
28
29          public Rectangle Scale(int factor)
30          {
31              int newWidth = width * factor;
32              int newHeight = height * factor;
33              return new Rectangle(newWidth, newHeight);
34          }
35      }
36
37      class Program
38      {
39          static void Main()
40          {
41              Rectangle rect1 = new Rectangle(4, 2);
42              rect1.Draw();
43              Rectangle rect2 = rect1.Scale(2);
44              System.Console.WriteLine();
45              rect2.Draw();
46              System.Console.ReadKey();
47          }
48      }
49  }
```

In the above code, the `Scale` method (lines 29-34) creates a new `Rectangle` that is larger than the original `Rectangle` object by a specified factor. Afterward, it returns the new `Rectangle` object. The output should be:

Output

```
****
****

********
********
********
********
```

4.5 Arrays of Objects

You can declare arrays that hold elements of any type. An array of a value-type (e.g. int) holds the actual values. In contrast, an array of a reference-type holds a set of references. An array of objects holds references to objects rather than the objects themselves.

In order to create an array of objects, you need to create an array of references before creating the actual objects and assigning their references to the elements of the array. For example,

```
using System;

namespace School
{
    class Student
    {
        public string name;
        public float score;

        public Student(string name, float score)
        {
            this.name = name;
            this.score = score;
        }
    }

    class Program
    {
        static void Main()
        {
            Student[] students = new Student[3];

            students[0] = new Student("Ziad", 90);
            students[1] = new Student("Malik", 95);
            students[2] = new Student("Jodie", 97);

            System.Console.WriteLine(students[0].name);
            System.Console.WriteLine(students[1].name);
            System.Console.WriteLine(students[2].name);
        }
    }
}
```

In the above code, line 21 creates a reference-type variable named `students` in the stack before creating an array of type `Student` with three elements in the heap and assigning its reference to the variable named `students`.

So far, no objects were created and the value of each array elements is `null`. Lines 23 creates a `Student` object and assigns its reference to the first element in the array (at index 0). Similarly, lines 24 and 25 create two other objects and assign their addresses to the second and the third elements of the array. Now, each element in the array holds a reference to a `Student` object.

Finally, lines 27-29 obtain the references of the objects by indexing the array before using each reference to access the corresponding object and print the value of its `name` field. The output should be:

Output

```
Ziad
Malik
Jodie
```

Chapter 5

Properties and Indexers

Encapsulation makes sure that "sensitive" data is hidden within the class and cannot be accessed by other classes. To achieve this, we must declare fields as `private` members and provide `public` methods to access and update the value of those `private` fields. For example:

```
1  class Product
2  {
3      private float price;
4
5      public void SetPrice(float priceValue)
6      {
7          if(priceValue >= 0) price = priceValue;
8          else Console.WriteLine("Invalid price value.");
9      }
```

```
10      public float GetPrice()
11      {
12          return price;
13      }
14  }
```

5.1 Defining Properties

A property is a class member that provides a flexible mechanism to access a private field. It has two accessors: a `get` and a `set`. We can rewrite the previous example using properties as follows:

```
1   class Product
2   {
3       private float price;
4       public float Price
5       {
6           set
7           {
8               if(value >= 0) price = value;
9               else Console.WriteLine("Invalid price value.");
10          }
11          get
12          {
13              return price;
14          }
15      }
16  }
```

In the above code, lines 4-15 define a `public` property of type `float` named `Price` with two accessors, a `get` and a `set`. The `Price` property is associated with the `price` field. The `get` accessor returns the value of the `price` field. The `set` accessor assigns a value to the `price` field. The `value` keyword represents the value we assign to the property.

5.2 Using Properties

Properties can be used as if they are public data members, but they are actually special methods called *accessors*. For example, we can create a `Product` object and use its property as follows:

```
1  class Program
2  {
3      static void Main()
4      {
5          Product p = new Product();
6          p.Price = 10;
7          System.Console.WriteLine(p.Price);
8      }
9  }
```

In the above code, we use the public property named `Price` to access and update the private field named `price`.

5.3 Automatic Properties

C# also provides a way to use short-hand (automatic properties), where you do not have to define the field for the property, and you only have to write get; and set; inside the property. For example:

```
1  class Product
2  {
3      private string name;
4
5      public string Name
6      {
7          get{return name;}
8          set{name = value;}
9      }
10 }
```

The following example will produce the same result as the example above. The only difference is that there is less code:

```
1  class Product
2  {
3      public string Name
4      {
5          get;
6          set;
7      }
8  }
```

5.4 Read-Only Properties

A property can be made read-only by providing only the `get` accessor without providing the `set` accessor. For example:

```
namespace Market
{
    class Product
    {
        private float price;
        public float Price
        {
            get { return price; }
        }

        public Product(float price)
        {
            this.price = price;
        }
    }

    class Program
    {
        static void Main()
        {
            Product p = new Product(10);
            System.Console.WriteLine(p.Price);
            p.Price = 9 // Error
        }
    }
}
```

In the above code, lines 6-9 define a read-only property called `Price`. Line 22 reads the value of the `price` field through the `Price` property using the `get` accessor and prints it to the screen. However, line 23 cannot assign a value to the `Price` property because it does not have a `set` accessor. The output will be the following error message:

Output

```
Property or indexer 'Product.Price' cannot be
assigned to -- it is read only
```

Automatic properties can be made read-only as well by making the `set` accessor `private`. For example:

```
1  using System;
2
3  namespace Market
4  {
5      class Product
6      {
7          // A read-only automatic property
8          public string Name { get; private set; }
9
10         public Product(string name)
11         {
12             Name = name;
13         }
14     }
15
```

```
16      class Program
17      {
18          static void Main()
19          {
20              Product p = new Product("Milk");
21              Console.WriteLine(p.Name);
22              p.Name = "Eggs" // Error
23          }
24      }
25  }
```

In the above code, line 12 can set the value of the `Name` property. Line 21 can print the value of the `Name` property to the screen. However, line 22 cannot assign a value to the `Name` property because it is read-only.

5.5 Write-Only Properties

A property can be made write-only by providing the `set` accessor and not providing the `get` accessor. An auto-implemented property can be made write-only by making its `get` accessor `private`. For example:

```
1  public string Name { private get; set; }
```

5.6 Defining Indexers

An **indexer** allows an object to be indexed just like arrays. Indexers resemble properties except that their accessors take parameters.

The following example defines a class named `Histogram` with an indexer that takes one parameter of type `int`. A histogram is a graphical representation that shows the frequencies of a a set of values.

```
1  using System;
2
3  namespace Statistics
4  {
5      class Histogram
6      {
7          // A field
8          private int[] frequencies;
9
10         // A read-only automatic property
11         public int MinIndex { get; private set; }
12
13         // A read-only automatic property
14         public int MaxIndex { get; private set; }
15
16         // A constructor
17         public Histogram(int minIndex, int maxIndex)
18         {
19             MinIndex = minIndex;
20             MaxIndex = maxIndex;
21             frequencies = new int[MaxIndex - MinIndex + 1];
22         }
```

```
23
24          // An indexer
25          public int this[int index]
26          {
27              get
28              {
29                  if (index >= MinIndex && index <= MaxIndex)
30                  {
31                      return frequencies[index - MinIndex];
32                  }
33                  else
34                  {
35                      return 0; // Invalid index
36                  }
37              }
38
39              set
40              {
41                  if (index >= MinIndex && index <= MaxIndex)
42                  {
43                      frequencies[index - MinIndex] = value;
44                  }
45              }
46          }
47      }
48  }
```

In the above code, line 8 defines a `private` array of type `int` named `frequencies`. Line 21 initializes the array with a size that is enough to hold all values between the `minIndex` and the `maxIndex`. For example, if the `minIndex` is `5` and the `maxIndex` is `10`,

the size of the array should be `6`, where the element at index `0` corresponds to `5` and the element at index `5` corresponds to `10`.

Lines 25-46 define a `public` indexer to access the elements of the `frequencies` array. In line 25, the indexer takes one parameter of type `int` named `index` and returns a value of type `int`. The indexer has two accessors `get` and `set`.

Lines 27-37 define the `get` accessor. It returns an element from the `frequencies` array based on the value of the `index`. For example, if the `MinIndex` is `5` and the value of `index` is `7`, the indexer will return the value located at index `2` in the `frequencies` array.

Lines 39-45 define the `set` accessor of the indexer. It sets the value of an element in the `frequencies` array based on the value of `index`. For example, if the `MinIndex` is `5` and the passed value of the `index` parameter is `8`, the indexer will set the value located at index `3` in the `frequencies` array.

5.7 Using Indexers

An indexer allows an object to be indexed like an array. The following example shows how to access the elements of a `Histogram` object using its indexer.

```
namespace Statistics
{
    class Histogram
    {
        private int[] frequencies;

        public int MinIndex { get; private set; }
        public int MaxIndex { get; private set; }

        public Histogram(int minIndex, int maxIndex)
        {
            MinIndex = minIndex;
            MaxIndex = maxIndex;
            frequencies = new int[MaxIndex - MinIndex + 1];
        }

        public int this[int index]
        {
            get
            {
                if (index >= MinIndex && index <= MaxIndex)
                    return frequencies[index - MinIndex];
                else
                    return 0; // Invalid index
            }

            set
            {
                if (index >= MinIndex && index <= MaxIndex)
                    frequencies[index - MinIndex] = value;
            }
        }
```

```
33          public void Draw()
34          {
35              for (int i = MinIndex; i <= MaxIndex; i++)
36              {
37                  Console.Write("{0}\t", i);
38
39                  int frequency = this[i];
40
41                  for (int j = 0; j < frequency; j++)
42                      Console.Write("*");
43
44                  Console.WriteLine();
45              }
46          }
47      }
48
49      class Program
50      {
51          static void Main()
52          {
53              Histogram histogram = new Histogram(-2, 1);
54              int[] values = { 0, -2, -2, -2, -1, 1, 0, 1 };
55              foreach (int x in values) histogram[x]++;
56
57              Console.WriteLine("Frequency of (-2) = {0}",
                ↪  histogram[-2]);
58
59              Console.WriteLine();
60
61              histogram.Draw();
62          }
63      }
64  }
```

In the above code, lines 33-46 define a method named `Draw`. It calls the indexer using the keyword `this` in line 39 to get the frequency associated with the value `i`. The `Draw` method uses the `this` keyword to call the indexer because they both belong to the same class (i.e. the `Histogram` class).

Line 55 uses the indexer by writing the object name followed by the index enclosed within square brackets. This way, it can access the elements inside that object as if the `histogram` object was an array.

Line 57 prints the frequency of the value `-2` on the screen. It calls the indexer of the `histogram` object and passes the value `-2` to it as the index. Note that it is okay to pass a negative value to the indexer. After all, the `histogram` object is not an actual array, it just acts like one. The value passed to the indexer is just a parameter not an actual index and thus it can have any value or any type (i.e. it does not have to be of type `int`). Finally, line 61 draws the histogram to the screen. The output of the above program should be

Output

```
Frequency of (-2) = 3


-2   ***
-1   *
0    **
1    **
```

5.8 Overloading Indexers

Like methods, indexers can also be overloaded. In C#, we can have multiple indexers within the same class but they must have different number of parameters and/or different parameter types. For example:

```
1  namespace Indexers
2  {
3      class ValuePrintWidth
4      {
5          public int DefaultCharWidth { get; private set; }
6          public ValuePrintWidth(int defaultCharacterWidth)
7          {
8              DefaultCharWidth = defaultCharacterWidth;
9          }
10
11         // First indexer
12         public int this[string s, int characterWidth]
13         {
14             get { return s.Length * characterWidth; }
15         }
16
17         // Second indexer
18         public int this[double x, int characterWidth]
19         {
20             get
21             {
22                 string s = Convert.ToString(x);
23                 return this[s, characterWidth];
24             }
25         }
```

```
26          // Third indexer
27          public int this[string s]
28          {
29              get
30              {
31                  return this[s, DefaultCharWidth];
32              }
33          }
34
35          // Fourth indexer
36          public int this[double x]
37          {
38              get
39              {
40                  string s = Convert.ToString(x);
41                  return this[s];
42              }
43          }
44      }
45
46      class Program
47      {
48          static void Main()
49          {
50              ValuePrintWidth x = new ValuePrintWidth(10);
51
52              Console.WriteLine(x["Ziad"]);
53              Console.WriteLine(x[16.649]);
54              Console.WriteLine(x["Ziad", 20]);
55              Console.WriteLine(x[16.649, 20]);
56          }
57      }
58  }
```

In the above code, note that the `ValuePrintWidth` class does not have an underlying array defined within the class. However, it has four overloaded indexers. The first and the second indexers are both two dimensional indexers while the third and the fourth indexers are both one-dimensional indexers.

The first indexer is defined in lines 12-15. It takes two parameters of type `string` and `int` and returns the width of the passed string when printed to the screen given the width of a single character.

The second indexer is defined in lines 18-25. It takes two parameters of type `double` and `int` and returns the width of the `double` value when printed to the screen given the width of a single character. Note that it calls the first indexer in line 23.

The third indexer is defined in lines 27-33. It takes a single parameter of type `string` and returns its width when printed to the screen based on the default width of a single character. Note that it calls the first indexer in line 31.

The fourth indexer is defined in lines 36-43. It takes a single parameter of type `double` and returns the width of the `double` value when printed to the screen based on the default width of a single character. Note that it calls the third indexer in line 41.

Lines 52-55 call the four indexers and the output should be as follows:

Output

```
40
60
80
120
```

Indexers can have one or more dimensions. There is no requirement that an indexer actually operate on an array. The role of the indexer is to provide a functionality that appears "array-like" to the user of the indexer. Use an indexer for any purpose for which an array-like syntax is beneficial.

Chapter 6

Operator Overloading

You can redefine or overload most of the built-in operators available in C#. Thus a programmer can use operators with user-defined types as well. Overloaded operators are methods with special names (the keyword operator followed by the symbol for the operator being defined). similar to any other method, an overloaded operator has a return type and a parameter list. Operators include:

- Unary operators: `+`, `-`, `!`, `~`, `++`, `--`
- binary operators: `+`, `-`, `*`, `/`, `%`
- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`

6.1 Overloading Unary Operators

Unary operators (+, -, !, ~, ++, --) take one operand and can be overloaded. For example:

```
using System;

namespace Batteries
{
    class Battery
    {
        public string Name { get; private set; }

        public int Energy { get; private set; }

        public Battery(string name, int energy)
        {
            Name = name;
            Energy = energy;
        }

        public static Battery operator --(Battery b)
        {
            return new Battery(b.Name, b.Energy - 1);
        }

        public void PrintInfo()
        {
            Console.WriteLine("{0}: {1}", Name, Energy);
        }
    }

```

```
28      class Program
29      {
30          static void Main()
31          {
32              Battery x = new Battery("Energizer", 100);
33              x.PrintInfo();
34              x--;
35              x.PrintInfo();
36              Console.ReadKey();
37          }
38      }
39  }
```

In the above code, lines 17-20 overload the decrement operator (`--`) using a method named `operator --` that takes a single operand of type `Battery` and returns a new `Battery` object with an energy that is less by `1`. Line 32 defines a `Battery` object named `x` with an energy of `100`. Line 34 decrements `x` by `1`. The output should be:

Output

```
Energizer:  100
Energizer:  99
```

Note that the operator overloading method must be defined as a `static` method.

6.2 Overloading Binary Operators

Binary operators (`+`, `-`, `*`, `/`, `%`) take two operands and can be overloaded. For example:

```
using System;
namespace Batteries
{
    class Battery
    {
        public string Name { get; private set; }
        public int Energy { get; private set; }

        public Battery(string name, int energy)
        {
            Name = name;
            Energy = energy;
        }

        public static Battery operator +(Battery b1,
        ↪ Battery b2)
        {
            string name = b1.Name + " & " + b2.Name;
            int energy = b1.Energy + b2.Energy;
            return new Battery(name, energy);
        }

        public void PrintInfo()
        {
            Console.WriteLine("{0}: {1}", Name, Energy);
        }
    }
```

```
27
28      class Program
29      {
30          static void Main()
31          {
32              Battery x = new Battery("Eveready", 30);
33              Battery y = new Battery("Energizer", 60);
34              Battery z = x + y;
35              x.PrintInfo();
36              y.PrintInfo();
37              z.PrintInfo();
38              Console.ReadKey();
39          }
40      }
41  }
```

In the above code, lines 15-20 overload the addition operator `+` using a method named `operator +` that takes two operands of type `Battery` and returns a new `Battery` object. Line 32 defines a `Battery` object named `x` with an energy of `30` and line 33 defines a `Battery` object named `x` with an energy of `60`. Line 34 defines a `Battery` object named `z` that is equal to the sum of `x` and `y`. The output should be:

Output

```
Eveready:  30
Energizer:  60
Eveready & Energizer:  90
```

6.3 Overloading Comparison Operators

Comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) take two operands and can be overloaded. For example:

```
1  using System;
2
3  namespace Batteries
4  {
5      class Battery
6      {
7          public string Name { get; private set; }
8          public int Energy { get; private set; }
9
10         public Battery(string name, int energy)
11         {
12             Name = name;
13             Energy = energy;
14         }
15
16         public static bool operator >(Battery b1, Battery
           ↪ b2)
17         {
18             return b1.Energy > b2.Energy;
19         }
20
21         public static bool operator <(Battery b1, Battery
           ↪ b2)
22         {
23             return b1.Energy < b2.Energy;
24         }
25
```

```
26          public void PrintInfo()
27          {
28              Console.WriteLine("{0}: {1}", Name, Energy);
29          }
30      }
31
32      class Program
33      {
34          static void Main()
35          {
36              Battery x = new Battery("Eveready", 30);
37              Battery y = new Battery("Energizer", 60);
38              Battery z = null;
39
40              if(x > y) z = x;
41              else z = y;
42
43              x.PrintInfo();
44              y.PrintInfo();
45              z.PrintInfo();
46              Console.ReadKey();
47          }
48      }
49  }
```

In the above code, lines 16-19 overload the greater than operator `>` using a method named `operator >` that takes two operands of type `Battery` and returns a Boolean. Similarly, lines 21-24 overload the less than operator `<` using a method named `operator <` that takes two operands of type `Battery` and returns a Boolean.

Line 36 defines a `Battery` object named `x` with an energy of `30` and line 37 defines a `Battery` object named `x` with an energy of `60`. Lines 40-41 assign the `Battery` object with maximum energy to `z`. The output should be:

Output

```
Eveready:  30
Energizer:  60
Energizer:  60
```

Note that the conditional operators `>` and `<` must be overloaded in pair (i.e. you cannot overload one and not overload the other). Similarly, the conditional operators `==` and `!=` must be overloaded in pair.

6.4 Non-Overloadable Operators

The conditional logical operators (`&&`, `||`) cannot be overloaded directly. The assignment operators (`+=`, `-=`, `*=`, `/=`, `%=`) cannot be overloaded. Other operators that cannot be overloaded include `=`, `.`, `?:`, `->`, `new`, `is`, `sizeof`, `typeof`.

Chapter 7

Inheritance

Inheritance is one of the core concepts of OOP languages. It is a mechanism that allows for deriving a class from another class resulting in a hierarchy of classes that share a set of attributes and methods.

The "inheritance" concept introduces the following two terms:

- **Base Class (parent):** the class being inherited from.
- **Derived Class (child):** the class that inherits from another.

Inheritance allows one class to inherit class members (e.g. fields and methods) from another class.

7.1 Base Class

The base class is also known as the parent class. It is a regular class as in the following example:

```
class Person
{
    public string name;
    public int maxSpeed;

    public void Run(int speed)
    {
        if(speed > maxSpeed) speed = maxSpeed;
        Console.WriteLine("{0} is running at a speed of
        ↪  {1}", name, speed);
    }
}
```

A regular class becomes a base class (or a parent class), only when another class inherits from it.

7.2 Derived Class

In C#, to inherit from a class, use the `:` symbol. In the following example, the `Player` class inherits from the `Person` class:

```
class Player : Person
{
    public int maxForce;

    public void Shoot(int force)
    {
        if(force > maxForce) force = maxForce;
        Console.WriteLine("{0} is shooting with a force of
        ↪ {1}", name, force);
    }
}
```

In the above code, the `Player` class is a derived (or child) class while the `Person` class is a base (or parent) class. Now, any `Player` object is also a `Person` object. The following example creates a `Player` object:

```
class Program
{
    static void Main()
    {
        Player p = new Player();
        p.name = "Salah";
        p.maxSpeed = 100;
        p.maxForce = 80;
        p.Run(110);
        p.Shoot(90);
    }
}
```

In the above code, note that the `Player` object has five members. This is because the `Player` class defines two members (`maxForce` and `Shoot`) and inherits three members (`name`, `maxSpeed`, and `Run`) from the `Person` class.

7.3 Code Reusability

Inheritance allows you to reuse fields and methods of an existing class when you create a new class. This saves effort and makes your code maintainable. For example, consider the following two classes:

```
1  class CheckingAccount
2  {
3      public string id;
4      public float balance;
5      public float monthlyFees
6  }
7
8  class SavingAccount
9  {
10     public string id;
11     public float balance;
12     public float interestRate;
13 }
```

In the above code, the fields `id` and `balance` exist in both classes. We can rewrite the above code using inheritance as follows:

```
class Account
{
    public string id;
    public float balance;
}

class CheckingAccount : Account
{
    public float monthlyFees
}

class SavingAccount : Account
{
    public float interestRate;
}
```

In the above code, we created a parent class named `Account` to hold the common members of the two classes named `CheckingAccount` and `SavingAccount`. Afterward, we made each class inherit from the `Account` class before adding its additional members. This way, we have a single copy of the common members in once place rather than having multiple copies of the code.

Note than in a real banking software system, the common members between the `CheckingAccount` and `SavingAccount` are way more than the two fields provided in the example. Using inheritance can help avoid duplicate code. This saves the time and effort required for development besides making the code easier to read and and maintain.

7.4 Abstract Classes

Abstract classes cannot be instantiated (i.e. we cannot create instance objects from them), but they can be inherited. For example:

```
1  abstract class Account
2  {
3      public string id;
4      public float balance;
5  }
6
7  class CheckingAccount : Account
8  {
9      public float monthlyFees
10 }
11
12 class SavingAccount : Account
13 {
14     public float interestRate;
15 }
```

In the above code, we do not want to create objects from the `Account` class. The only purpose of the `Account` class is to serve as a parent class for the `CheckingAccount` and `SavingAccount` classes.

In line 1, we use the `abstract` keyword to prevent a programmer from creating an `Account` object by mistake. Trying to create an `Account` object will result in an error.

7.5 Sealed Classes

If you don't want other classes to inherit from a class, use the `sealed` keyword. For example:

```
1  abstract class Account
2  {
3      public string id;
4      public float balance;
5  }
6
7  sealed class CheckingAccount : Account
8  {
9      public float monthlyFees
10 }
11
12 sealed class SavingAccount : Account
13 {
14     public float monthlyFees
15 }
```

In the above code, line 7 defines the `CheckingAccount` class as a sealed class using the keyword `sealed`. Consequently, the programmer cannot create a class that inherits from the `CheckingAccount` class. Similarly, line 12 defines the `SavingAccount` class as a sealed class using the keyword `sealed`. This means that the programmer cannot create a class that inherits from the `SavingAccount` class.

7.6 Constructor Calling Order

When an object is instantiated from a child class the constructor of the parent class is called before the constructor of the child class. For example:

```
1  using System;
2  namespace Bank
3  {
4      class Account
5      {
6          public string id;
7          public float balance;
8
9          public Account()
10         {
11             Console.WriteLine("Account object created");
12         }
13     }
14
15     class CheckingAccount : Account
16     {
17         public float monthlyFees;
18
19
20         public CheckingAccount()
21         {
22             Console.WriteLine("CheckingAccount object
               ↪  created");
23         }
24     }
```

```
25
26      class Program
27      {
28          static void Main()
29          {
30              CheckingAccount x = new CheckingAccount();
31              Console.ReadKey();
32          }
33      }
34  }
```

In the above code, lines 9-12 define a default constructor for the `Account` class. In line 15, the `CheckingAccount` class inherits from the `Account` class. Lines 20-23 define a default constructor for the `CheckingAccount` class. Line 30 creates a `CheckingAccount` object by calling the default constructor named `CheckingAccount()`.

Creating a `CheckingAccount` object requires an `Account` object to be created first to act as a base for building the `CheckingAccount` object. Therefore, the `Account()` constructor is called before the `CheckingAccount()` constructor. The output should be:

Output

```
Account object created
CheckingAccount object created
```

7.7 Constructor Initializers

It is possible for a constructor in a child class to call another constructor in its parent class. For example:

```
1  class Account
2  {
3      public string id;
4      public float balance;
5
6      public Account(string id, int balance)
7      {
8          this.id = id;
9          this.balance = balance;
10     }
11 }
12
13 class CheckingAccount : Account
14 {
15     public float monthlyFees;
16
17     public CheckingAccount(string id, int balance, float
       ↪  monthlyFees) : base(id, balance)
18     {
19         this.monthlyFees = monthlyFees;
20     }
21 }
```

In line 17, the `CheckingAccount` constructor calls the `Account` constructor using the keyword `base`.

Note that we use the keyword `this` to call another constructor in the same class but we use the keyword `base` to call another constructor in the "base" (or parent) class.

7.8 Inheritance Tree

A class can inherit from a class that inherits from another class and so on. For example,

```
1 class A { public int x; }
2
3 class B : A { public int y; }
4
5 class C : B { public int z; }
```

In the above code, an object of type `A` will have a single member named `x`. An object of type `B` will have a two members named `x` and `y`. An object of type `C` will have three members named `x`, `y`, and `z`.

7.9 Access Modifiers

The `protected` access modifier is used to specify that the member can only be accessed by code in the same class or in a derived class. For example,

```
namespace Bank
{
    class Account
    {
        private int id;
        protected string name;
        public float balance;

        public void SetId(int id)
        {
            this.id = id;
        }
    }

    class SavingAccount : Account
    {
        public SavingAccount(int id, string name)
        {
            this.id = id; // Not allowed
            this.SetId(id); // Ok
            this.name = name;
            this.balance = 0;
        }
    }

```

```
26      class Program
27      {
28          static void Main()
29          {
30              Account x = new Account();
31              x.id = 1; // Not allowed
32              x.SetId(1);
33              x.name = "Ahmed"; // Not allowed
34              x.balance = 1000;
35          }
36      }
37  }
```

In the above code, the `SavingAccount` class is a child class of the `Account` class. Therefore, it inherits all the members in its parent. However, in line 19, the child class cannot access the inherited `id` field directly because `id` is a `private` member of its parent. In line 20, the child class accesses the `id` indirectly by using the inherited `SetId` method, which is a `public` member of its parent. In line 21, the child class can access the inherited `name` field because the `name` field is a `protected` member of its parent. In line 22, the child class can of course access the inherited `balance` field because it is a `public` member of its parent.

In line 31, the `Program` class cannot access the `id` field directly because `id` is a `private` member of the `Account` class. In line 32, the `Program` class accesses the `id` indirectly by using the `SetId` method, which is a `public` member of the `Account` class. In line

33, the `Program` class cannot access the `name` field because the `name` field is a `protected` member of the `Account` class. In line 34, the `Program` class can of course access the `balance` field because it is a `public` member of the `Account` class.

Consider a program that has a class named `A` along with other classes. Assume that `A` has private, protected, and public members. The following table shows the ability of the different classes in the program to access the members of `A`.

Class Members	`A` Itself	Children of `A`	Other Classes
Private members of `A`	Yes	No	No
Protected members of `A`	Yes	Yes	No
Public members of `A`	Yes	Yes	Yes

Chapter 8

Polymorphism & Interfaces

Polymorphism is the ability of objects of different types to provide a unique interface for different implementations of methods. The behavior of an object to respond to a call to its method members is determined based on object type at run time.

An interface is a blueprint of a class. It cannot have method body and cannot be instantiated. It is used to achieve multiple inheritance which can't be achieved by class.

This lecture starts with covering a set of topics related to inheritance before proceeding to discuss polymorphism and interfaces.

8.1 Hiding Inherited Members

The following example shows a class named `Animal` that has a method named `Speak` and a `Cat` class that inherits from the `Animal` class:

```
namespace Farm
{
    class Animal
    {
        public void Speak()
        {
            System.Console.WriteLine("Animal speaks");
        }
    }

    class Cat : Animal
    {
    }

    class Program
    {
        static void Main()
        {
            Cat c = new Cat();
            c.Speak();
        }
    }
}
```

In the above program, the `Cat` class inherits the `Speak` method from the `Animal` class. Therefore, the output of the program should be:

Output

```
Animal speaks
```

It is possible for a derived class to define a member that has the same name as a member in its base class. When this happens, the member in the base class is hidden within the derived class. For example:

```
1  namespace Farm
2  {
3      class Animal
4      {
5          public int Age { get; protected set; }
6
7          public void Speak()
8          {
9              System.Console.WriteLine("Animal speaks");
10         }
11     }
12
13     class Cat : Animal
14     {
15         public void Speak()
16         {
17             System.Console.WriteLine("Meow Meow");
18         }
19     }
```

```
20
21      class Program
22      {
23          static void Main()
24          {
25              Cat c = new Cat();
26              c.Speak();
27              System.Console.ReadKey();
28          }
29      }
30  }
```

In the above code, lines 15-18 define a method named `Speak` within the `Cat` class. This method hides the `Speak` method inherited by the `Cat` class from the `Animal` class. Line 26 calls the `Speak` method of a `Cat` object named `c`. The output will be:

Output

```
Meow Meow
```

8.2 Base Class References

A base class reference can be used to refer to an object of a derived class. For example:

```
using System;

namespace Farm
{
    class Animal
    {
        public void Speak()
        {
            Console.WriteLine("Animal speaks");
        }
    }

    class Cat : Animal
    {
        public void Speak()
        {
            Console.WriteLine("Meow Meow");
        }

        public void Jump() { /* Jump code */ }
    }

    class Program
    {
        static void Main()
        {
            Animal x = new Cat();
            x.Speak();
            Console.ReadKey();
        }
    }
}
```

In the above code, line 25 creates a reference-type variable of type `Animal` named `x`. Afterward, it creates an object of type `Cat` before assigning its address to `x`. There is no problem with referring to a `Cat` object as an `Animal` object because the class `Cat` is derived from the class `Animal` and inherits all of its members. Therefore, we can safely use the reference `x` to call any member that is defined in the `Animal` class because it will for sure be found in the `Cat` object.

Line 28 calls the `Speak` method of the object. Although the object is actually a `Cat`, the `x` reference is of type `Animal` and therefore it sees only the `Animal` part of the `Cat` object (i.e. it can only call the members defined within the `Animal` class). Therefore, the call for `Speak` in line 28 will actually call the `Speak` method defined within the `Animal` class. The output of the above program will be:

Output

```
Animal speaks
```

8.3 Polymorphism

Polymorphism is often referred to as the third pillar of OOP after encapsulation and inheritance. Inheritance allows a child class to inherit members from a parent class.

Polymorphism allows a child class to introduce a different implementation of an inherited method.

The following program contains a `Cat` class and a `Dog` class that are both derived from an `Animal` class. The `Animal` class has a `Speak` method. The `Cat` and the `Dog` classes introduce their own implementation of the `Speak` method using hiding as shown below:

```
1  using System;
2
3  namespace Farm
4  {
5      // Parent class
6      class Animal
7      {
8          public void Speak()
9          {
10             Console.WriteLine("Animal speaks");
11         }
12     }
13
14     // Child class
15     class Cat : Animal
16     {
17         public void Speak() // Hiding inherited member
18         {
19             Console.WriteLine("Meow Meow");
20         }
21     }
22
```

```
23      // Child class
24      class Dog : Animal
25      {
26          public void Speak() // Hiding inherited member
27          {
28              Console.WriteLine("Woof Woof");
29          }
30      }
31
32      class Program
33      {
34          static void Main()
35          {
36              Cat cat = new Cat();
37              Dog dog = new Dog();
38              PlaySound(cat);
39              PlaySound(dog);
40              Console.ReadKey();
41          }
42
43          static void PlaySound(Animal x)
44          {
45              x.Speak();
46          }
47      }
48  }
```

In the above code, lines 43-46 define a method named `PlaySound` that takes an `Animal` object and calls its `Speak` method. Line 38 calls the `PlaySound` method passing it a `Cat` object named `cat` as an `Animal` object. Line 39 calls the same method passing

it a `Dog` object named `dog` as an `Animal` object. The output of the above program will be:

Output

```
Animal speaks
Animal speaks
```

It would be great, if we can call the `Speak` method of an `Animal` object but instead of getting the "Animal speaks" message, we get a "Meow Meow" message if the `Animal` object is actually a `Cat` object and get a "Woof Woof" message if the `Animal` object is actually a `Dog` object. This is exactly what we can achieve using **polymorphism**. The following example, uses polymorphism to allow each `Animal` object to act in its own way, when the `Speak` method is called:

```
1  using System;
2
3  namespace Farm
4  {
5      class Animal
6      {
7          // A virtual method can be overridden
8          // in child classes (i.e. Cat and Dog)
9          public virtual void Speak()
10         {
11             Console.WriteLine("Animal speaks");
12         }
13     }
```

```
14      class Cat : Animal
15      {
16          public override void Speak() // Overriding Speak
17          {
18              Console.WriteLine("Meow Meow");
19          }
20      }
21
22      class Dog : Animal
23      {
24          public override void Speak() // Overriding Speak
25          {
26              Console.WriteLine("Woof Woof");
27          }
28      }
29
30      class Program
31      {
32          static void Main()
33          {
34              Cat cat = new Cat();
35              Dog dog = new Dog();
36              PlaySound(cat);
37              PlaySound(dog);
38              Console.ReadKey();
39          }
40
41          static void PlaySound(Animal x)
42          {
43              x.Speak();
44          }
45      }
46  }
```

In the above code, line 9 uses the `virtual` keyword to define the `Speak` method of the `Animal` class as a method that can be overridden if needed by any class that is derived from the `Animal` class.

Line 16 uses the `override` keyword to define the `Speak` method of the `Cat` class as a method that takes the place of the `Speak` method inherited from the `Animal` class. Similarly, line 24 uses the `override` keyword to define the `Speak` method of the `Dog` class as a method that takes the place of the `Speak` method inherited from the `Animal` class. The output of the above program will be:

Output

```
Meow Meow
Woof Woof
```

Polymorphism is a Greek word, meaning "one name many forms". "Poly" means many and "morph" means forms. Polymorphism allows different objects from the same class (e.g. objects from the `Animal` class) to have multiple implementations of the same method (e.g. two `Animal` objects can have different implementations of the `Speak` method because one `Animal` object is actually a `Cat` while the other is actually a `Dog`).

8.4 Interfaces

An **interface** is a blueprint of a class. It cannot be instantiated and its members cannot have implementation. For example,

```
using System;

namespace Farm
{
    interface IAnimal
    {
        void Speak();
    }

    class Cat : IAnimal
    {
        // Implement the Speak method
        public void Speak()
        {
            Console.WriteLine("Meow Meow");
        }
    }

    class Dog : IAnimal
    {
        // Implement the Speak method
        public void Speak()
        {
            Console.WriteLine("Woof Woof");
        }
    }

```

```
28      class Program
29      {
30          static void Main()
31          {
32              Cat cat = new Cat();
33              Dog dog = new Dog();
34              PlaySound(cat);
35              PlaySound(dog);
36              Console.ReadKey();
37          }
38
39          static void PlaySound(IAnimal x)
40          {
41              x.Speak();
42          }
43      }
44  }
```

In the above code, lines 5-8 define an `interface` named `IAnimal` that has a single method named `Speak`. Line 7 defines the `Speak` method but without implementation and without an access modifier.

Line 10 defines a class named `Cat` that implements the `IAnimal` interface and line 19 defines a class named `Dog` that implements the `Animal` interface as well. Both the `Cat` and the `Dog` classes must implement the `Speak` method defined by the `IAnimal` interface.

A method is defined within an interface without implementation nor an access modifier. All members of an interface are `public`. A class that implements an interface, must provide the implementation of all the members declared inside that interface.

8.5 Multiple Inheritance

Multiple inheritance is a feature of some OOP languages (e.g. C++) in which a class can inherit from more than one parent class. C# does not support multiple inheritance (i.e. a derived class cannot have more than one parent). However, besides inheriting from one parent, a class can implement multiple interfaces. For example:

```
1  using System;
2  namespace School
3  {
4      interface IAthlete
5      {
6          void Run();
7      }
8
9      interface IStudent
10     {
11         void TakeExam();
12     }
13
```

```
14      class Person
15      {
16          public string Name { get; set; }
17
18          public Person(string name)
19          {
20              Name = name;
21              Console.WriteLine("Person created.");
22          }
23      }
24      class AthleteStudent : Person, IAthlete, IStudent
25      {
26          public AthleteStudent(string name) : base(name)
27          {
28              Console.WriteLine("AthleteStudent created.");
29          }
30
31          public void Run()
32          {
33              Console.WriteLine("{0} is running.", Name);
34          }
35
36          public void TakeExam()
37          {
38              Console.WriteLine("{0} is taking exam.", Name);
39          }
40      }
41
42      class Program
43      {
44          static void Main()
45          {
46              AthleteStudent x = new AthleteStudent("Ziad");
47              Person p = x;
```

```
48              IAthlete a = x;
49              IStudent s = x;
50
51              Console.WriteLine(p.Name);
52              a.Run();
53              s.TakeExam();
54          }
55      }
56  }
```

In the above code, lines 4-7 define an interface named `IAthlete` with a `Run` method. Lines 9-12 define an interface named `IStudent` with a `TakeExam` method. Lines 14-23 define a class named `Person` with a `Name` read-only property and a constructor that takes one parameter to set the `Name` value.

Line 24 defines a class named `AthleteStudent` that inherits from the `Person` class and implements the `IAthlete` and the `IStudent` interfaces. Lines 26-29 define a constructor that calls the constructor of the base class to set the value of the `Name` property. Lines 31-34 implement the `Run` method defined in the `IAthlete` interface. Lines 36-39 implement the `TakeExam` method defined in the `IStudent` interface.

Line 46 defines an object of the `AthleteStudent` class and assigns its address to a variable of type `AthleteStudent` named `x`. Line 47 creates a variable of type `Person` named `p` and assigns the value of `x` to it. Now, the variable `p` refers to the same object but

sees that object as a `Person` (i.e. it sees the `Name` property but not the `Run` nor the `TakeExam` methods).

Line 48 creates a variable named `a` of type `IAthlete` and assign the value of `x` to it. Now, the variable `a` refers to the same object but sees that object as an `IAthlete` (i.e. it sees the `Run` method but not the `Name` property nor the `TakeExam` method).

Line 49 creates a variable named `s` of type `IStudent` and assign the value of `x` to it. Now, the variable `s` refers to the same object but sees that object as an `IStudent` (i.e. it sees the `TakeExam` method but not the `Name` property nor the `Run` method).

Line 51 prints the Name of the object using the `p` reference-type variable. Line 52 calls the `Run` method using the `a` reference-type variable. Line 53 calls the `TakeExam` using the `s` reference-type variable. The output of the above program should be:

Output

```
Person object created.
AthleteStudent object created.
Ziad
Ziad is running.
Ziad is taking an exam.
```

For a class `A` to inherit from another class `B` and at the same time implement multiple interfaces `X`, `Y`, `Z`, ..., the definition of the class `A` should use the `:` operator followed by a comma-separated list that must start with the name of the `B` class followed by the names of the interfaces that `A` implement in any order. For example:

```
1 class A : B, X, Y, Z { } // Ok
2 class A : B, Y, Z, X { } // Ok
3 class A : X, Y, Z, B { } // Error (B must come first)
```

Chapter 9

Enumerations & Structures

9.1 Enumeration

An enumeration type (or "enum") is a data type consisting of a set of named values, called "enumerators". For example:

```
enum Color
{
    Red,
    Green,
    Yellow
}
```

A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value. For example:

```
1  Color c = Color.Red;
```

The following example, defines a `Person` class and uses enumeration to create a type that holds the person's gender.

```
1  using System;
2
3  namespace Profiles
4  {
5
6      enum Gender
7      {
8          Male,
9          Female
10     }
11
12     class Person
13     {
14         public string name;
15         public Gender gender;
16
17         public void PrintInfo()
18         {
19             Console.WriteLine("{0}\t{1}", name, gender);
20         }
21     }
```

```
22
23      class Program
24      {
25          static void Main()
26          {
27              Person p = new Person();
28              p.name = "Sahar";
29              p.gender = Gender.Female;
30              p.PrintInfo();
31          }
32      }
33  }
```

In the above code, lines 6-10 define an enumeration type `Gender`. Line 15 defines a member field named `gender` of type `Gender`. Line 29 assigns the value `Gender.Female` to the `gender` field of the `Person` object using the variable `p`. The output will be:

Output

```
Sahar       Female
```

9.2 Structures

A **structure** is a value type that can encapsulate data and related functionality. You use the `struct` keyword to define a structure type. For example:

```
using System;
namespace Drawing
{
    public struct Point
    {
        public Point(double x, double y)
        {
            X = x;
            Y = y;
        }

        public double X { get; set; }
        public double Y { get; set; }

        public void Print()
        {
            Console.WriteLine("({0}, {1})", X, Y);
        }
    }

    class Program
    {
        static void Main()
        {
            Point p1 = new Point(5, 12);
            Point p2 = p1;
            p2.X = 70;
            p1.Print();
            p2.Print();
        }
    }
}
```

In the above code, lines 4-19 define a structure named `Point` using the keyword `struct`. Line 25 creates a `Point` instance and assigns it to a variable named `p1` of type `Point`. Line 26 defines a variable named `p2` of type `Point` and assigns the value of `p1` to it. The assignment operator makes a copy of the structure instance named `p1`.

Line 27 modifies the `Point` instance named `p2` by assigning a value to its `X` property. Lines 28 and 29 call the `Print` method of both `p1` and `p2`, respectively. The output should be:

Output

```
(5, 12)
(70, 12)
```

Note that modifying the structure instance named `p2` did not affect the structure instance named `p1`. This is because line 26 created a copy of the structure instance `p1` and assigned that copy to `p2`.

A class-type variable contains a reference to an instance of the class (i.e. class-type variables are reference-type variables). In contrast, a structure-type variable contains the structure instance itself (i.e. structure-type variables are value-type variables). Therefore, the assignment operator `=` copies the reference of object for class-type variables but copies the structure instance itself for structure-type variables.

9.3 Structures versus Classes

A structure is almost similar to a class because both are user-defined data types and both hold a set of members (e.g. fields, method). However, they are not the same. The following table summarized the differences between a class and a structure.

Class	**Structure**
Reference-type	Value-type
Stored on the heap memory	Stored on stack memory
Can contain a parameter-less constructor	Cannot contain a parameter-less constructor
Can contain a destructor	Cannot contain a destructor
Can inherit from another class	Cannot inherit from another structure or another class
Can contain protected members	Cannot have protected members
Can contain virtual or abstract methods	Cannot contain virtual or abstract methods
Two class-type variables can contain a reference to the same object and any operation on one variable can affect the other.	Each structure-type variable contains its own structure instance and any operation on one variable can not effect another.

Chapter 10

Delegates & Events

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur.

10.1 Delegates

A delegate can hold a reference to a method. Delegates allow for passing methods as arguments to other methods. For example:

```
using System;

namespace Example
{
    delegate int MyDelegate(int x, int y);

    class Program
    {
        static int Add(int a, int b)
        {
            return a + b;
        }

        static void Execute(MyDelegate x)
        {
            Console.WriteLine(x(5, 6));
        }

        static void Main()
        {
            MyDelegate d = new MyDelegate(Add);
            Console.WriteLine(d(7, 8));
            Execute(d);
        }
    }
}
```

In the above code, line 5 defines a delegate named `MyDelegate` with two parameters of type `int` and an `int` return type.

Lines 9-12 define a method named `Add` that has the same signature and return type as the delegate defined in line 5. Lines 14-17 define a method named `Execute` that takes a delegate instance as a parameter and calls that delegate instance at line 16.

Line 21 defines a new instance of the `MyDelegate` delegate and passes the `Add` method to the constructor. Now, `d` holds a reference to the `Add` method. Line 22 calls the delegate instance named `d`, which actually calls the `Add` method.

Line 23 calls the `Execute` method and passes the delegate instance `d` to it as an argument. The `Execute` method in line 14 gets a delegate instance as a parameter.

Line 16 calls that delegate instance, which means calling the method whose reference is held by this delegate instance (i.e. the `Add` method). The output should be:

Output

```
15
11
```

10.2 Multicast Delegates

A delegate can hold more than one method reference. This is called multicast delegate because when that delegate is called, all the methods whose references are held by that delegate are called in order.

Multiple methods can be assigned to the delegate using `+=` operator and removed using `-=` operator. For example:

```
1  using System;
2
3  namespace Example
4  {
5      delegate int MyDelegate(int x, int y);
6
7      class Program
8      {
9          static int Add(int a, int b)
10         {
11             Console.WriteLine("Add called");
12
13             return a + b;
14         }
15
16         static int Subtract(int a, int b)
17         {
18             Console.WriteLine("Subtract called");
19
20             return a - b;
21         }
22
```

```
23          static void Main()
24          {
25              MyDelegate d = new MyDelegate(Add);
26              d += new MyDelegate(Subtract);
27              Console.WriteLine("Result = {0}", d(12, 8));
28
29              Console.WriteLine("---------------");
30              Console.WriteLine("Remove Subtract");
31              Console.WriteLine("---------------");
32
33              d -= new MyDelegate(Subtract);
34              Console.WriteLine("Result = {0}", d(12, 8));
35          }
36      }
37  }
```

In the above code, line 5 defines a delegate named `MyDelegate` with two parameters of type `int` and a return type of `int`.

Line 25 creates a variable named `d` of type `MyDelegate` and assigns a method named `Add` to it. Line 26 assign another method named `Subtract` to `d`. Now, `d` holds two references to the two methods `Add` and `Subtract`.

Line 27 calls the delegate `d`, which results in calling the two methods `Add` and `Subtract` in order. Note that both methods return values. If a multicast delegate returns a value then it returns the value from the last assigned method (i.e. the `Subtract` method).

Line 33 removes the `Subtract` method from `d`. Afterward, line 34 calls `d` again. This time only the `Add` method will be called. The output of the above program should be:

Output

```
Add called
Subtract called
Result = 4
---------------
Remove Subtract
---------------
Add called
Result = 20
```

If a multicast delegate returns a value then it returns the value from the last assigned target method.

10.3 Events

Events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. An **event handler** is a method that is called whenever the event is raised. For example:

```
using System;

namespace PowerManagement
{
    enum State
    {
        On,
        Off
    }

    delegate void Handler(State oldState, State newState);

    class PowerSwitch
    {
        public event Handler StateChanged;

        public State CurrentState { get; private set; }

        public PowerSwitch()
        {
            CurrentState = State.Off;
        }

        public void TurnOn()
        {
            if (CurrentState != State.On)
            {
                CurrentState = State.On;
                if (StateChanged != null)
                    StateChanged(State.Off, State.On);
            }
        }
```

```
33          public void TurnOff()
34          {
35              if (CurrentState != State.Off)
36              {
37                  CurrentState = State.Off;
38                  if (StateChanged != null)
39                      StateChanged(State.On, State.Off);
40              }
41          }
42      }
43
44      class Program
45      {
46          static void Main()
47          {
48              PowerSwitch p = new PowerSwitch();
49              p.StateChanged += new Handler(OnStateChanged);
50              p.TurnOn();
51          }
52
53          static void OnStateChanged(State oldState, State
            ↪ newState)
54          {
55              Console.WriteLine("Power state changed from {0}
                ↪ to {1}", oldState, newState);
56          }
57      }
58  }
```

In the above code, lines 5-9 define an enumeration named `State`. Line 11 defines a delegate named `Handler` with two parameters of type `State` and a `void` return type.

Line 15 defines an event named `StateChanged` of type `Handler`. The `StateChanged` event can be assigned one or more delegate instances of type `Handler`, where each delegate instance refers to a method that has the same signature and return type as the `Handler` delegate.

Lines 30 calls (or raises) the `StateChanged` event, which should result in calling the event handlers (i.e. the methods) assigned to it if any. Raising an event that does not have at least one event handler (i.e. its value is `null`), will result in a run-time exception. Therefore, line 29 makes sure that the value of the `StateChanged` event is not `null` before raising it. Similarly, line 39 raises the `StateChanged` event if its value is not `null` and passes two arguments of type `State` to it according to the method signature specified by the `Handler` delegate.

Line 48 defines a `PowerSwitch` object named `p`. Line 49 creates an instance of the `Handler` delegate and assigns it to the `StateChanged` event of the object `p`. Note that the assignment took place using the `+=` operator because events do not allow assignment using the `=` operator as it is the case with delegates. Line 50 calls the `TurnOn` method, which raises the `StateChanged` event in line 30. This results in calling the `OnStateChanged` method defined in lines 53-56. The `OnStateChanged` method is referred to as the **event handler** because it is called whenever the event is raised. The output will be:

Output

```
Power state changed from Off to On
```

Chapter 11

Files & Streams

A computer program can take input data, process it, and provide output to the user. Data is stored in memory while the program is running. Once the program terminates, this data is lost (erased from memory). Files allow a program to store data permanently (e.g. on a hard disk drive).

11.1 Files

The `File` class from the `System.IO` namespace, allows us to work with files. The `File` class has many useful methods for creating and getting information about files. The following table lists some of them.

Method	Description
`AppendText()`	Appends text at the end of an existing file
`Copy()`	Copies a file
`Create()`	Creates or overwrites a file
`Delete()`	Deletes a file
`Exists()`	Tests whether the file exists
`ReadAllText()`	Reads the contents of a file
`Replace()`	Replaces the contents of a file with the contents of another file
`WriteAllText()`	Creates a new file and writes the contents to it. If the file already exists, it will be overwritten.

The following example uses the `WriteAllText()` method to write some content to a file named `MyFile.txt` on your desktop.

```
1  using System;
2  using System.IO;
3
4  namespace Files
5  {
6      class Program
7      {
8          static void Main()
9          {
10             string text = "Hello world!";
11
12             string path = Environment.GetFolderPath(
13                 Environment.SpecialFolder.Desktop);
14
```

```
15              File.WriteAllText(path + "\\MyFile.txt", text);
16          }
17      }
18  }
```

In the above code, line 12 gets the path to the desktop folder on your computer. Line 15 creates (or overwrites) a file named `MyFile.txt` on your desktop and stores the contents of the string `text` in it.

The following example uses the `ReadAllText()` method to read the contents of `MyFile.txt` before printing it to the screen.

```
1   using System;
2   using System.IO;
3   namespace Files
4   {
5       class Program
6       {
7           static void Main()
8           {
9               string path = Environment.GetFolderPath(
10                  Environment.SpecialFolder.Desktop);
11              string text = File.ReadAllText(path +
                ↪  "\\MyFile.txt");
12              Console.WriteLine(text);
13          }
14      }
15  }
```

In the above code, line 9 gets the path to the desktop folder on your computer. Line 11 reads the contents of a file named `MyFile.txt` on your desktop into a string named `text`. Finally, line 12 prints the value of the `text` variable to the screen.

11.2 Streams

In C# file operations, normally streams are used to read and write to files. A **stream** is an additional layer created between an application and a file. Streams are normally used when reading data from large files. By using streams, the data from large files in broken down into small chunks and sent to the stream. These chunks of data can then be read from the application.

The reason for breaking it down into small chunks is because of the performance impact of reading a big file in one shot. If you were to read the data from say, a 100 MB file at one shot, your application could just hang and become unstable. The best approach is then to use streams to break the file down into manageable chunks.

So when a write operation is carried out on file, the data to be written, is first written to the stream. From the stream, the data is then written to the file. The same goes for the read operation. In the read operation, data is first transferred from the file to the stream. The data is then read from the application via the stream.

11.2.1 Stream Writer

The stream writer is used to write data to a file using streams. The data from the application is first written into the stream. After that the stream writes the data to the file.

The following example uses the `StreamWriter` class to write some content to a file named `MyFile.txt` on your desktop.

```
1  using System;
2  using System.IO;
3
4  namespace Files
5  {
6      class Program
7      {
8          static void Main()
9          {
10             string text = "Hello world!";
11
12             string path = Environment.GetFolderPath(
13                 Environment.SpecialFolder.Desktop);
14
15             StreamWriter writer = new StreamWriter(path +
               ↪  "\\MyFile.txt");
16             writer.Write(text);
17             writer.Close();
18         }
19     }
20 }
```

In the above code, line 12 gets the path to the desktop folder on your computer. Line 15 creates a `StreamWriter` object named `writer` by calling the constructor and passing the file name to it. Line 16 writes the contents of the `text` variable to the `writer` stream (not the file). Finally, line 17 closes the stream. It's normally a good practice to close file handlers when the file is no longer required for writing purposes.

11.2.2 Stream Reader

he stream reader is used to read data from a file using streams. The data from the file is first read into the stream. Thereafter the application reads the data from the stream.

The following example uses the `StreamReader` class to read the contents of `MyFile.txt` before printing it to the screen.

```
1  using System;
2  using System.IO;
3  namespace Files
4  {
5      class Program
6      {
7          static void Main()
8          {
9              string path = Environment.GetFolderPath(
10                 Environment.SpecialFolder.Desktop);
```

```
11
12              StreamReader reader = new StreamReader(path +
                ↪  "\\MyFile.txt");
13              string text = reader.ReadToEnd();
14              reader.Close();
15
16              Console.WriteLine(text);
17          }
18      }
19  }
```

In the above code, line 9 gets the path to the desktop folder on your computer. Line 12 creates a `StreamReader` object named `reader` by calling the constructor and passing the file name to it. Line 13 uses the `ReadToEnd()` method of the `reader` object to read the contents of the file into the `text` variable. Line 14 closes the stream. It's normally a good practice to close file handlers when the file is no longer required for reading purposes. Finally, line 16 prints the contents of `text` to the screen.

Chapter 12

Exception Handling

An **exception** is a problem that arises during the execution of a program, such as an attempt to divide by zero. C# exception handling is built upon three keywords: `try`, `catch`, and `finally`.

- The `try` block allows you to define a block of code to be tested for errors while it is being executed.
- The `catch` block allows you to define a block of code to be executed, if an error occurs in the `try` block.
- The `finally` block is used to execute a block of statements, whether an exception is thrown or not thrown. (e.g. if you open a file, it must be closed whether an exception is raised or not).

The `try` and `catch` blocks come in pairs. The `finally` block is optional. The general syntax takes the following form:

```
try
{
    // Block of code to try
}
catch (Exception e)
{
    // Block of code to handle errors
}
finally // Optional
{
    // Block of code to execute anyways
}
```

12.1 Exceptions

Consider the following example:

```
1 int[] myNumbers = {1, 2, 3};
2 Console.WriteLine(myNumbers[10]); // error!
```

In the above code, line 2 will generate an error, because `myNumbers[10]` does not exist. The error message will be:

Exception

```
System.IndexOutOfRangeException: 'Index was outside
the bounds of the array.'
```

12.2 The `try` and `catch` Blocks

We can use `try...catch` to catch the error and execute some code to handle it, as in the following example:

```
1  try
2  {
3      int[] myNumbers = {1, 2, 3};
4      Console.WriteLine(myNumbers[10]);
5  }
6  catch (Exception e)
7  {
8    Console.WriteLine(e.Message);
9  }
```

In the above code, line 6 catches the exception and stores its information in the object named `e`. line 8 uses the `Message` property of the object `e` to obtain the message that describes the exception.

The output will be:

Output

```
Index was outside the bounds of the array.
```

You can omit the `(Exception e)` part from line 6 but you will not have access to any information about the exception that has occurred. In this case, you can output your own error message, as in the following example:

```
1  try
2  {
3      int[] myNumbers = {1, 2, 3};
4
5      Console.WriteLine(myNumbers[10]);
6  }
7  catch
8  {
9      Console.WriteLine("Something went wrong!");
10 }
```

12.3 The `finally` Block

The `finally` block lets you execute code, after `try...catch`, regardless of the result. The following example shows a proper way to write some content to a file named `MyFile.txt` on your desktop.

```
using System;
using System.IO;
namespace Files
{
    class Program
    {
        static void Main()
        {
            string text = "Hello world!";

            string path = Environment.GetFolderPath(
                Environment.SpecialFolder.Desktop);

            StreamWriter writer = null;

            try
            {
                writer = new StreamWriter(path +
                ↪  "\\MyFile.txt");
                writer.Write(text);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (writer != null) writer.Close();
            }
        }
    }
}
```

In the above code, line 14 defines a `StreamWriter` reference-type variable named `writer` and sets its value to `null` (i.e. nothing).

Line 18 tries to create a `StreamWriter` object and assign its address to the `writer` variable. If line 18 fails due to an exception, the value of the `writer` variable will remain as `null` and the execution will immediately jump to the `catch` block.

Line 19 tries to write the contents of the `text` string to the stream. If line 19 fails due to an exception, the execution will immediately jump to the `catch` block.

Line 23 will be executed only if an exception occurs in the `try` block. Line 27 will execute anyways whether or not an exception occurs in the `try` block.

Note that line 27 checks the value of the `writer` variable before calling the `Close()` method. Remember that if line 18 fails the value of the `writer` variable will remain `null`. If `writer` is `null`, there is no need to call the `Close()` method for an object that does not exist. In fact, trying to do so will throw a `NullReferenceException`.

12.4 Throwing Exceptions

You can throw an exception in your code using the keyword `throw`. This can be useful to indicate that some error has occurred and needs some attention. For example:

```
1  namespace SIS
2  {
3      class Student { }
4
5      class Program
6      {
7          static void Main()
8          {
9              Student student = null;
10             Save(student);
11         }
12
13         static void Save(Student s)
14         {
15             if (s == null)
16             {
17                 throw new ArgumentNullException();
18             }
19             else
20             {
21                 // Save student data to database or file
22             }
23         }
24     }
25 }
```

Line 17 throws an exception of type `ArgumentNullException` to draw attention whenever the value of the passed parameter `s` is `null`. The `ArgumentNullException` class is derived from the `Exception` class. C# provides several classes that are derived from the `Exception` class to represent different types of exceptions.

12.5 Catching Multiple Exceptions

If we want to handle the exceptions separately and make a particular decision for each one, we should have specific information about the exceptions that may occur. In this case, the best way is to use an individual catch block for each exception. For example:

```
1  using System;
2
3  namespace Calculator
4  {
5      class Program
6      {
7          static void Main()
8          {
9              Divid("Hi", "Hello");
10             Divid("5", "0");
11             Divid("99999999999999999999999", "9");
12             Divid("99", "9");
13         }
14
```

```
15          static void Divid(string s1, string s2)
16          {
17              try
18              {
19                  int x = Convert.ToInt32(s1);
20                  int y = Convert.ToInt32(s2);
21                  Console.WriteLine("Result = {0}", x / y);
22              }
23              catch (FormatException)
24              {
25                  Console.WriteLine("Format Exception!");
26              }
27              catch (DivideByZeroException)
28              {
29                  Console.WriteLine("Divide By Zero
                    ↪  Exception!");
30              }
31              catch (OverflowException)
32              {
33                  Console.WriteLine("Overflow Exception!");
34              }
35              catch(Exception)
36              {
37                  Console.WriteLine("Unknown exception!");
38              }
39          }
40      }
41  }
```

In the above code, lines 15-40 define a method named `Divid` that contains a `try` block with multiple `catch` blocks. Each `catch` block is responsible for catching a specific type of exception if it occurs.

The `FormatException` class is a child of the `Exception` class. It is thrown by the `ToInt32` method if the passed string cannot be converted to an `int`. The `DivideByZeroException` class is a child of the `Exception` class. It is thrown by a division operation if the denominator is equal to zero. The `OverflowException` is a child of the `Exception` class. It is thrown if the value is too large to be stored in the specified type.

Executing line 9 results in a `FormatException`. Executing line 10 results in a `DivideByZeroException`. Executing line 11 results in a `OverflowException`. Executing line 12 does not result in any exceptions. The output of the above program should be:

Output

```
Format Exception!
Divide By Zero Exception!
Overflow Exception!
Result = 11
```

In a try statement with multiple catch blocks, if an exception occurs, the catch blocks try to catch it in order. For example, in the above program if a `DivideByZeroException` occurred, the first catch block (lines 23-26) will not execute because it can only catch a `FormatException`. The second catch block (lines 27-30) will catch the `DivideByZeroException` and therefore the subsequent catch blocks will not execute.

All exception classes in C# are derived from the `Exception` class. Therefore, a catch block that catches an exception of type `Exception` will catch any exception and will not allow it to pass to next catch blocks if any.

12.6 Defining Exception Classes

A program can throw a predefined exception (e.g. an instance of the `Exception` class). In addition, a program can create its own exception classes by deriving from Exception. For example:

```
1  using System;
2  namespace Calculator
3  {
4      class InvalidScoreException : Exception
5      {
6          public InvalidScoreException()
7              : base("Invalid score.")
8          {
9          }
10
11         public InvalidScoreException(string message)
12             : base(message)
13         {
14         }
15     }
```

```
16
17      class Student
18      {
19          public string Name { get; private set; }
20          public float Score { get; private set; }
21
22          public Student(string name, float score)
23          {
24              if(score < 0 || score > 100)
25              {
26                  throw new InvalidScoreException();
27              }
28
29              Name = name;
30              Score = score;
31          }
32      }
33
34      class Program
35      {
36          static void Main()
37          {
38              try
39              {
40                  Student s = new Student("Ahmed", -20);
41              }
42              catch(InvalidScoreException e)
43              {
44                  Console.WriteLine(e.Message);
45              }
46          }
47      }
48  }
```

In the above code, a class named `InvalidScoreException` that is derived from the `Exception` class is defined in lines 4-15. Lines 6-9 define a default constructor (i.e. parameterless constructor) that calls the base constructor and passes the exception message to it. Lines 11-14 define a one-parameter constructor that takes a message and passes it to the base constructor.

Line 26 throws an exception of type `InvalidScoreException` if the score is less that zero or greater than 100. Lines 38-45 define a `try..catch` statement. The catch block defined in lines 42-45 catch an exception only if its type is `InvalidScoreException` and displays its message to the screen. The output will be:

Output

```
Invalid score.
```

Chapter 13

Graphical User Interfaces

C# has all the features of any powerful, modern language. In C#, the most rapid and convenient way to create your Graphical User Interface (GUI) is to do so visually, using the **Windows Forms** Designer and **Toolbox**. Windows Forms controls are reusable components that encapsulate user interface functionality and are used in client side Windows based applications.

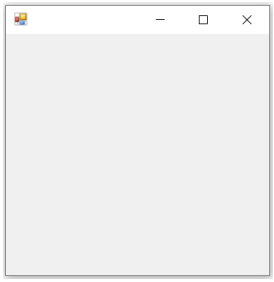
A control is a component on a form used to display information or accept user input. The `Control` class provides the base functionality for all controls that are displayed on a form.

13.1 Windows Forms

The `Form` class represents a window or dialog box that makes up an application's user interface. The `Form` class is available under the `System.Windows.Forms` namespace. The following example creates a simple windows application.

```
1  using System.Windows.Forms;
2
3  namespace WindowsApp
4  {
5      class Program
6      {
7          static void Main()
8          {
9              Form form = new Form();
10             Application.Run(form);
11         }
12     }
13 }
```

In the above code, line 9 creates a `Form` object named `form`. Line 10 uses the `Run` method of the `Application` to run the application using the `form` object as the main form to display on startup. The output should be:



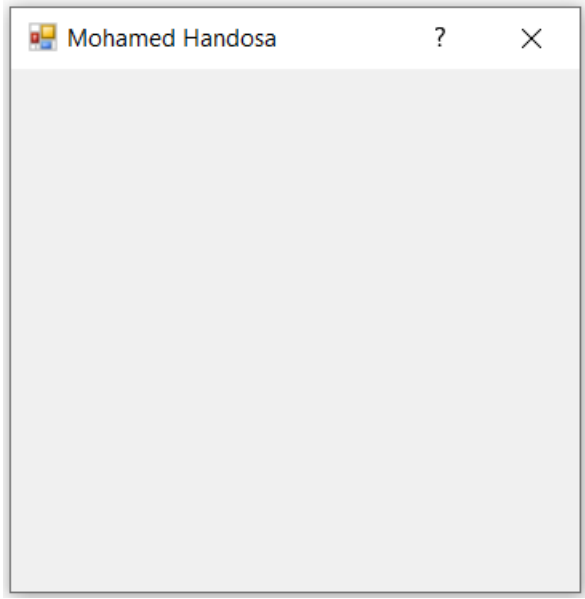
A form has many elements such as the **control menu**, the **minimize button**, the **maximize button**, the **close button**, and the **title bar**. All of those elements and more can be controlled using the members of the `Form` class. For example, the following properties allow for controlling some aspects of a form object:

Property	Description
ControlBox	Gets or sets a value indicating whether a control box is displayed in the caption bar of the form.
HelpButton	Gets or sets a value indicating whether a Help button should be displayed in the caption box of the form.
MaximizeBox	Gets or sets a value indicating whether the Maximize button is displayed in the caption bar of the form.
MinimizeBox	Gets or sets a value indicating whether the Minimize button is displayed in the caption bar of the form.
Text	Gets or sets the text associated with this control.

The following example modifies the caption bar of the form using some of its properties:

```
1  using System.Windows.Forms;
2
3  namespace WindowsApp
4  {
5      class Program
6      {
7          static void Main()
8          {
9              Form form = new Form();
10             form.MinimizeBox = false;
11             form.MaximizeBox = false;
12             form.HelpButton = true;
13             form.ControlBox = true;
14             form.Text = "Mohamed Handosa";
15             Application.Run(form);
16         }
17     }
18 }
```

In the above code, line 10 sets the `MinimizeBox` property to `false`. Line 11 sets the `MaximizeBox` property to `false`. Line 12 sets the `HelpButton` property to `true`. Line 13 sets the `ControlBox` property to `true`. Line 14 sets the `Text` property to "Mohamed Handosa". The output will be:



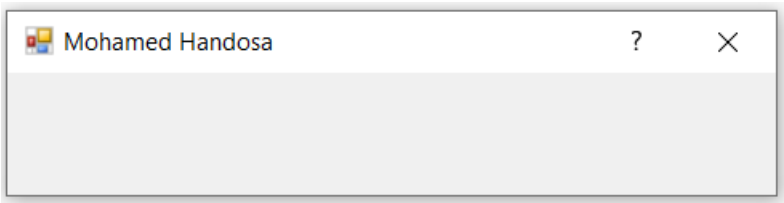
The `HelpButton` of a form will not display unless the `MinimizeBox` and the `MaximizeBox` are both set to `false`.

Customizing a form can get complicated, especially when we start to add controls and manipulate the members of each control. Therefore, it is a common practice to make use of inheritance.

Instead of creating a `Form` object and then customizing it, it is better to create a custom `Form` class first and then instantiate it directly. We can do this by inheriting the `Form` class as in the following example:

```
using System.Drawing;
using System.Windows.Forms;

namespace WindowsApp
{
    class Program
    {
        static void Main()
        {
            MyForm form = new MyForm();
            Application.Run(form);
        }
    }

    class MyForm : Form
    {
        public MyForm()
        {
            MinimizeBox = true;
            MaximizeBox = false;
            HelpButton = true;
            ControlBox = true;
            Text = "Mohamed Handosa";

            Width = 400;
            Height = 100;

            Point location = new Point(200, 350);
            DesktopLocation = location;
        }
    }
}
```

In the above code, lines 15-31 create a class named `MyForm` that inherits from the `Form` class. The class sets the values for the required properties in its constructor. This way we have a class that is already customized and ready to be directly instantiated. Line 10 creates an instance of the `MyForm` class and line 11 runs the application using the `form` object as a startup form. The output will be:



13.2 Labels

The `Label` class represents a standard Windows label. It inherits the `Control` class and can be added to a `Form` object. The following example creates a form with two labels:

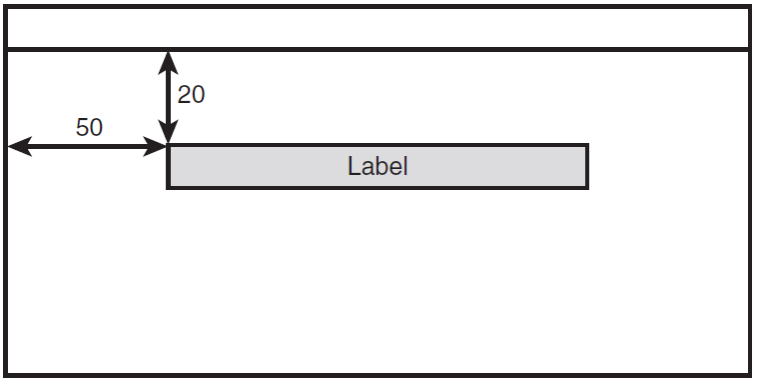
```
using System;
using System.Windows.Forms;

namespace WindowsApp
{
    class Program
    {
        static void Main()
        {
            MyForm form = new MyForm();
            Application.Run(form);
        }
    }

    class MyForm : Form
    {
        private Label myLabel;
        private Label myDateLabel;

        public MyForm()
        {
            // Caption bar properties
            MinimizeBox = true;
            MaximizeBox = false;
            HelpButton = true;
            ControlBox = true;
            Text = "Mohamed Handosa";

            // Create label objects
            myLabel = new Label();
            myDateLabel = new Label();

```

```
33              // Add the labels to the form
34              Controls.Add(myLabel);
35              Controls.Add(myDateLabel);
36
37              // Set myLabel text
38              myLabel.Text = "The time now is:";
39
40              // Set myLabel size to fit text
41              myLabel.AutoSize = true;
42
43              // Set myLabel position
44              myLabel.Top = 20;
45              myLabel.Left = 50;
46
47              // Get the current time
48              DateTime now = DateTime.Now;
49
50              // Convert time to a string and assign it
51              myDateLabel.Text = now.ToString();
52
53              // Set myDateLabel size to fit the date string
54              myDateLabel.AutoSize = true;
55
56              // Set myDateLabel position
57              myDateLabel.Top = 20;
58              myDateLabel.Left = myLabel.Right + 10;
59
60              // Set form size to fit the two labels
61              Width = myDateLabel.Right + 50;
62              Height = myLabel.Bottom + 100;
63          }
64      }
65  }
```

In the above code, line 17 and 18 define two class members of type `Label`. Later, in the constructor lines 30 and 31 initialize those two members to refer to two newly created `Label` objects.

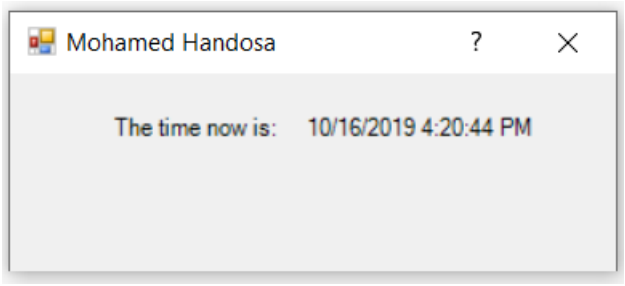
Lines 34 and 35 add the `Label` objects to the form by adding them to the `Controls` property of the form. The `Controls` property is a collection that stores the controls that the form will display. The `Controls` property has an `Add` method, which allows for adding as many `Control` objects as needed. Note that a `Label` object is also a `Control` object because the `Label` class inherits the `Control` class. Therefore, a `Label` object can be added to the `Controls` property of the form.

Lines 44 and 45 set the position of the `myLabel` object on the form using the two properties `Top` and `Left` as illustrated in the following figure:



Line 48 gets the current date and time using the static property `Now` of the `DateTime` structure and assign it to the `DateTime` variable named `now`. Line 51 calls the `ToString()` method of the `now` object to get the date and time in string format and then assign it to the `Text` property of the `myDateLabel` object.

The output will be something like the following:



13.3 Buttons

The `Button` class represents a Windows button control. It inherits the `Control` class and can be added to a `Form` object. The following example creates a form with a label and a button. When the button is clicked, it sets the text of the label to the current time.

```
1  using System;
2  using System.Windows.Forms;
3
4  namespace WindowsApp
5  {
6      class Program
7      {
8          static void Main()
9          {
10             MyForm form = new MyForm();
11             Application.Run(form);
12         }
13     }
14
15     class MyForm : Form
16     {
17         // A field of type Button
18         private Button myButton;
19
20         // A field of type Label
21         private Label myDateLabel;
22
23         // Constructor
24         public MyForm()
25         {
26             // Caption bar properties
27             MinimizeBox = false;
28             MaximizeBox = false;
29             HelpButton = true;
30             ControlBox = true;
31             Text = "Mohamed Handosa";
32
```

```
33              // Create a label
34              myDateLabel = new Label();
35
36              // Add the label to the form
37              Controls.Add(myDateLabel);
38
39              // Create a button
40              myButton = new Button();
41
42              // Add the button to the form
43              Controls.Add(myButton);
44
45              // Set myDateLabel size
46              myDateLabel.Width = 150;
47              myDateLabel.Height = 50;
48
49              // Set myDateLabel position
50              myDateLabel.Top = 20;
51              myDateLabel.Left = 50;
52
53              // Set myButton size
54              myButton.Width = 150;
55              myButton.Height = 50;
56
57              // Set myButton location
58              myButton.Top = 100;
59              myButton.Left = 50;
60
61              // Set myButton caption
62              myButton.Text = "Update Time";
63
64              // Set myButton click event
65              myButton.Click += MyButton_Click;
66
```

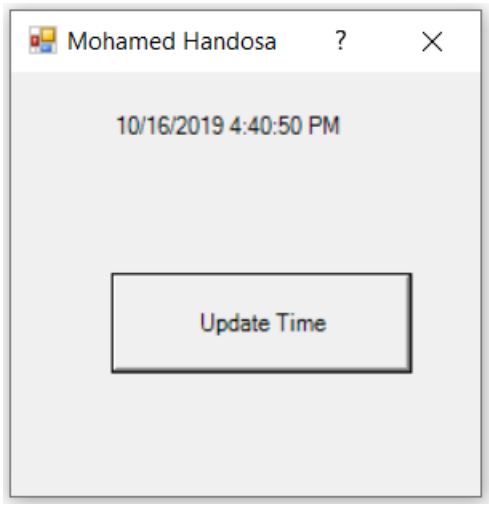
```
67              // Set form size to fit the two labels
68              Width = myDateLabel.Right + 50;
69              Height = myButton.Bottom + 100;
70          }
71
72          // A method that will be called when the Click
            ↪  event occurs
73          private void MyButton_Click(object sender,
            ↪  EventArgs e)
74          {
75              // Get the current time
76              DateTime now = DateTime.Now;
77
78              // Convert time to a string and assign it to
                ↪  myDateLabel text
79              myDateLabel.Text = now.ToString();
80          }
81      }
82  }
```

In the above code, line 18 defines a field of type `Button`. Line 40 creates a new `Button` object and assign its address to the `myButton` field. Line 43 adds the `myButton` object to the `Controls` property of the form. Lines 53-62 modify some properties of the `myButton` object.

Line 65 adds the `MyButton_Click()` method to the `Click` event of the `myButton` object using the `+=` operator. This means that whenever the user clicks the `myButton` object, that method will be called and executed. Note that the method must have a specific signa-

ture to be added to the `Click` event. The return type of the method must be `void` and the method must have exactly two parameters: an `object` parameter and an `EventArgs` parameter.

The Output will be something like the following:



13.4 Textboxes

The `TextBox` class represents a Windows text box control. It inherits the `Control` class and can be added to a `Form` object. For example:

```
using System;
using System.Windows.Forms;
namespace WindowsApp
{
    class Program
    {
        static void Main()
        {
            MyForm form = new MyForm();
            Application.Run(form);
        }
    }

    class MyForm : Form
    {
        // First name label
        private Label firstNameLabel;

        // Last name label
        private Label lastNameLabel;

        // Full name label
        private Label fullNameLabel;

        // First name text box
        private TextBox firstNameTextBox;

        // Last name text box
        private TextBox lastNameTextBox;

        // Exit button
        private Button exitButton;
```

```
33
34          public MyForm()
35          {
36              Text = "Mohamed Handosa";
37
38              // Create the labels
39              firstNameLabel = new Label();
40              lastNameLabel = new Label();
41              fullNameLabel = new Label();
42
43              // Add the labels to the form
44              Controls.Add(firstNameLabel);
45              Controls.Add(lastNameLabel);
46              Controls.Add(fullNameLabel);
47
48              // Create the text boxes
49              firstNameTextBox = new TextBox();
50              lastNameTextBox = new TextBox();
51
52              // Add the text boxes to the form
53              Controls.Add(firstNameTextBox);
54              Controls.Add(lastNameTextBox);
55
56              // Create the button
57              exitButton = new Button();
58
59              // Add the button to the form
60              Controls.Add(exitButton);
61
62              // Set firstNameLabel properties
63              firstNameLabel.Top = 20;
64              firstNameLabel.Left = 50;
65              firstNameLabel.Width = 100;
66              firstNameLabel.Height = 20;
```

```
67              firstNameLabel.Text = "First name: ";
68
69              // Set lastNameLabel properties
70              lastNameLabel.Top = 80;
71              lastNameLabel.Left = 50;
72              lastNameLabel.Width = 100;
73              lastNameLabel.Height = 20;
74              lastNameLabel.Text = "Last name: ";
75
76              // Set fullNameLabel properties
77              fullNameLabel.Top = 140;
78              fullNameLabel.Left = 50;
79              fullNameLabel.Width = 200;
80              fullNameLabel.Height = 20;
81
82              // Set firstNameTextBox properties
83              firstNameTextBox.Top = 20;
84              firstNameTextBox.Left = 160;
85              firstNameTextBox.Width = 100;
86              firstNameTextBox.Height = 20;
87
88              // Set lastNameTextBox properties
89              lastNameTextBox.Top = 80;
90              lastNameTextBox.Left = 160;
91              lastNameTextBox.Width = 100;
92              lastNameTextBox.Height = 20;
93
94              // Set exitButton properties
95              exitButton.Top = 200;
96              exitButton.Left = 50;
97              exitButton.Width = 100;
98              exitButton.Height = 20;
99              exitButton.Text = "Exit";
100
```

```
101             // Set form size to fit the controls
102             Width = fullNameLabel.Right + 50;
103             Height = exitButton.Bottom + 100;
104
105             // Set the text changed event for
                ↪  firstNameTextBox
106             firstNameTextBox.TextChanged +=
                ↪  FirstNameTextBox_TextChanged;
107
108             // Set the text changed event for
                ↪  lastNameTextBox
109             lastNameTextBox.TextChanged +=
                ↪  LastNameTextBox_TextChanged;
110
111             // Set the click event for the exitButton
112             exitButton.Click += ExitButton_Click;
113         }
114
115         // A method that will be called when the user
            ↪  changes firstNameTextBox
116         private void FirstNameTextBox_TextChanged(object
            ↪  sender, EventArgs e)
117         {
118             UpdateFullName();
119         }
120
121         // A method that will be called when the user
            ↪  changes lastNameTextBox
122         private void LastNameTextBox_TextChanged(object
            ↪  sender, EventArgs e)
123         {
124             UpdateFullName();
125         }
126
```

```
127         // A method that will be called when the user
            ↪ Clicks the exitButton
128         private void ExitButton_Click(object sender,
            ↪ EventArgs e)
129         {
130             // Exit the application
131             Application.Exit();
132         }
133
134         private void UpdateFullName()
135         {
136             fullNameLabel.Text = firstNameTextBox.Text + "
                ↪ " + lastNameTextBox.Text;
137         }
138     }
139 }
```

In the above code, lines 16-32 create several fields, which are three `Label` fields, two `TextBox` fields, and one `Button` field. Line 36 sets the `Text` property of the form. Lines 39-41 initialize the three `Label` fields. Lines 44-46 add the three `Label` objects to the controls of the form. Lines 49 and 50 initialize the two `TextBox` fields. Lines 53 and 54 add the two `TextBox` objects to the controls of the form. Line 57 initializes the `exitButton` field and line 60 adds it to the controls of the form.

Lines 62-99 set the properties of the six `Control` objects (i.e. the three `Label` objects, the two `TextBox` objects,and the `Button` object). Lines 102 and 103 set the width and height of the form.

Line 106 adds the `FirstNameTextBox_TextChanged()` method (lines 116-119) to the `TextChanged` event of the `firstNameTextBox` object using the `+=` operator. This means that whenever the user changes the text of the `firstNameTextBox` object, that method will be called and executed.

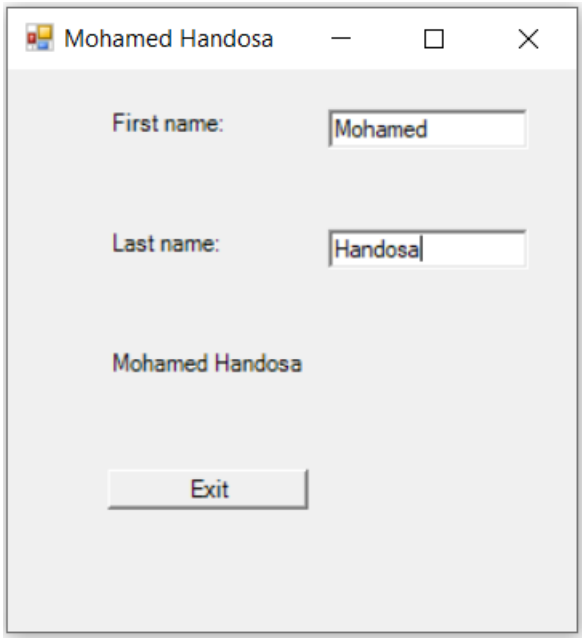
Line 109 adds the `LastNameTextBox_TextChanged()` method (lines 122-125) to the `TextChanged` event of the `lastNameTextBox` object using the `+=` operator. This means that whenever the user changes the text of the `lastNameTextBox` object, that method will be called and executed.

Note that a method must have a specific signature to be added to the `TextChanged` event. The return type of the method must be `void` and the method must have exactly two parameters: an `object` parameter and an `EventArgs` parameter.

The two `_TextChanged()` methods call the `UpdateFullName()` method, which in turn updates the `Text` of the `fullNameLabel` to be the concatenation of the `Text` of the `fistNameTextBox` and the `Text` of the `lastNameTextBox`.

Line 112 adds the `ExitButton_Click()` method to the `Click` event of the `exitButton` object using the `+=` operator. This means that whenever the user clicks the `exitButton` object, that method will be called and executed.

Lines 128-132 define the `ExitButton_Click()` method that calls the `Exit()` static method of the `Application` class to terminate the application. The output will be something like the following:



The Microsoft Visual Studio IDE provides a **Windows Forms** designer that allows a developer to simply drag and drop controls from a **Toolbox** to the form and modify their properties from the **Properties** window. While the developer is designing the form visually, the designer generates the corresponding C# code automatically behind the scenes.