Unit 5

4.0 Types, Objects and Namespaces

(Text Book Chapter 03)

4.0 Types, Objects and Namespaces

.Net is thoroughly object oriented. Not only does .NET allow you to use objects, it demands it. Almost every ingredient you'll need to use to create a web application is, on some level, really a kind of object.

4.1 C# Types

C# language contains three basic types. They are **Value Types, Reference Types** and **Boxing** and **Unboxing.** The following hierarchy shows these types.

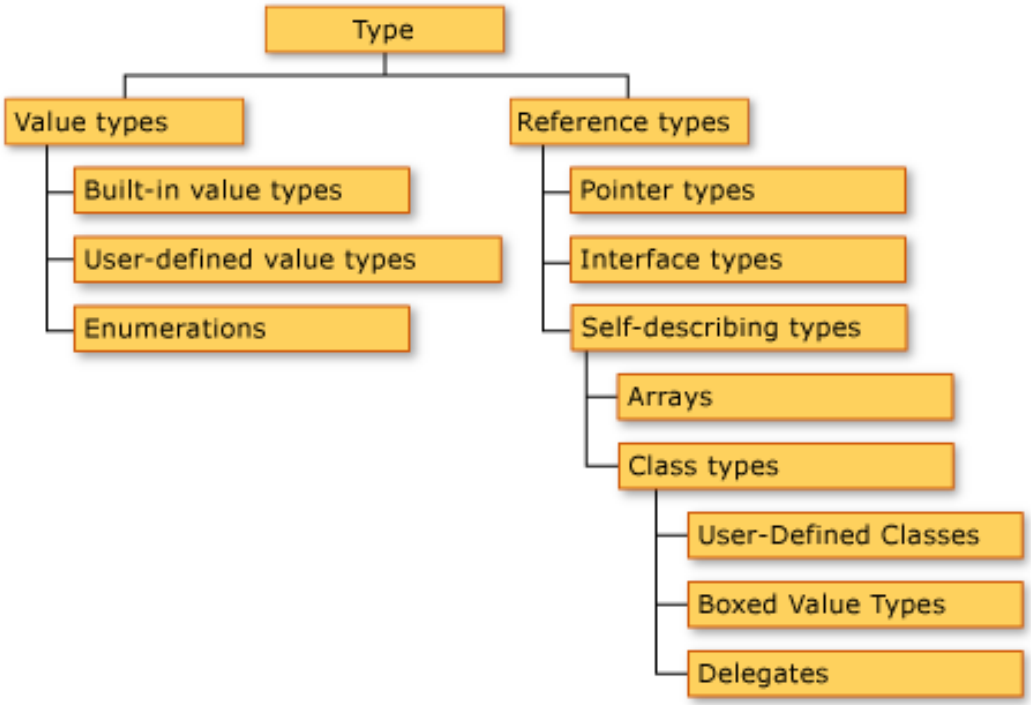

Figure 4.1

4.1.1 Types -> Value Types -> Built-In Value types

Built-In value types are also called Primitive Types. The built-in value types Numeric types and bool type.

- **Numeric types**
  - Integral types
  - Floating-point types
  - decimal
- **bool**

4.0 Types, Objects and Namespaces

4.1.1.1 Numeric Types

Integral types their physical sizes and ranges are shown in Figure 4.2

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Figure 4.2

Floating-Point types their physical sizes and ranges are shown in Figure 4.3

Type	Approximate range	Precision
float	±1.5 × 10−45 to ±3.4 × 1038	7 digits
double	±5.0 × 10−324 to ±1.7 × 10308	15-16 digits

Figure 4.3

decimal keyword denotes a 128-bit data type. Compared to floating-point types, the decimal type has a greater precision and a smaller range, which makes it suitable for financial and monetary calculations. The approximate range and precision for the decimal type are shown in the Figure 4.4 table.

Type	Approximate Range	Precision	.NET Framework type
decimal	±1.0 × 10−28 to ±7.9 × 1028	28-29 significant digits	System.Decimal

Figure 4.4

➢4.0 Types, Objects and Namespaces

4.1.1.2 bool Type

You can assign a Boolean value to a bool variable, for example:

```
bool alphabetic = (c > 64 && c < 123);
```

4.1.1.3 Conversions

In C++, a value of type bool can be converted to a value of type int; in other words, false is equivalent to zero and true is equivalent to nonzero values. In C#, there is no conversion between the bool type and other types. For example, the following if statement is invalid in C#, while it is legal in C++:

```
int x = 123;
if (x)   // Invalid in C#
{
  printf("The value of x is nonzero.");
}
```

Example 4.1-1 – C++ Example

```
int x = 123;
if (x!= 0)  //The C# way
{
  Console.Write("The value of x is nonzero.")
}
```

Example 4.1-2 – C# Example

4.1.2 Types -> Value Types -> User Defined types

User defined types are struct types and enum types.

4.1.2.1 User Defined Types

- struct types
- enum types

struct type

A struct type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory. The following example shows a simple struct declaration:

```
public struct Book
{
   public decimal price;
   public string title;
   public string author;
}
```

structs can also contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types, although if several such members are required, you should consider making your type a class instead.

structs can implement an interface but they cannot inherit from another struct. For that reason, struct members cannot be declared as protected.

enum types

The enum keyword is used to declare an enumeration, a distinct type consisting of a set of named constants called the enumerator list. Every enumeration type has an underlying type, which can be any integral type except char. This declaration takes the following form::

[attributes] [modifiers] enum identifier [:base-type] {enumerator-list} [;]

where:

attributes (Optional)	: Additional declarative information.
modifiers (Optional)	: The allowed modifiers are new and the four access modifiers.
identifier	: The enum name.
base-type (Optional)	: The underlying type that specifies the storage allocated for each enumerator. It can be one of the integral types except char. The default is int.
enumerator-list	: The enumerators' identifiers separated by commas, optionally including a value assignment.

The default underlying type of the enumeration elements is int. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example:

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

In this enumeration, Sat is 0, Sun is 1, Mon is 2, and so forth. Enumerators can have initializers to override the default values. For example:

```
enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

In this enumeration, the sequence of elements is forced to start from 1 instead of 0. See the example next page.

➢4.0 Types, Objects and Namespaces

```
using System;
public class EnumTest
{
  enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
  public static void Main()
  {
    int x = (int) Days.Sun;
    int y = (int) Days.Fri;
    Console.WriteLine("Sun = {0}", x);
    Console.WriteLine("Fri = {0}", y);
  }
}
```

Example 4.2 – Enum usage

The above example outputs the following.

Sun = 2
Fri = 7
Notice that if you remove the initializer from Sat=1, the result will be:

Sun = 1
Fri = 6

4.1.3 Types -> Reference Types
Variables of reference types, referred to as objects, store references to the actual data. This section introduces the following keywords used to declare reference types:

4.1.5 Types -> Reference Types -> Pointer Types
In an unsafe context, a type may be a pointer type as well as a value type or a reference type. A pointer type declaration takes one of the following forms:

type* identifier;
void* identifier; //allowed but not recommended.

➢4.0 Types, Objects and Namespaces

Any of the following types may be a pointer type:
-sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, or bool.
-Any enum type.
-Any pointer type.
-Any user-defined struct type that contains fields of unmanaged types only.

Pointer types do not inherit from object and no conversions exist between pointer types and object. Also, boxing and unboxing do not support pointers. However, you can convert between different pointer types and between pointer types and integral types.

When you declare multiple pointers in the same declaration, the * is written along with the underlying type only, not as a prefix to each pointer name. For example:

```
int* p1, p2, p3;          // Ok
int *p1, *p2, *p3;        // Invalid in C#
```

A pointer cannot point to a reference or to a struct that contains references because it is possible for an object reference to be garbage collected even if a pointer is pointing to it. The GC does not keep track of whether an object is being pointed to by any pointer types. The value of the pointer variable of type myType* is the address of a variable of type myType. Next sections discuss about pointer types.

4.1.5 Types -> Reference Types -> interface Types

Interfaces are defined by using the interface keyword, as shown in Example 4.3. Interfaces describe a group of related functionalities that can belong to any class or struct. Interfaces can consist of methods, properties, events, indexers, or any combination of those four member types. An interface cannot contain fields. Interfaces members are automatically public.

```
interface IEquatable<T>
{
   bool Equals(T obj);
}
```

Example 4.3

➢4.0 Types, Objects and Namespaces

Interfaces describe a group of related functionalities that can belong to any class or struct. Interfaces can consist of methods, properties, events, indexers, or any combination of those four member types. An interface cannot contain fields. Interfaces members are automatically public. When a class or struct is said to inherit an interface, it means that the class or struct provides an implementation for all of the members defined by the interface. The interface itself provides no functionality that a class or struct can inherit in the way that base class functionality can be inherited. However, if a base class implements an interface, the derived class inherits that implementation.

Classes and structs can inherit from interfaces in a manner similar to how classes can inherit a base class or struct. A class or struct also can inherit more than one interface. When a class or struct inherits an interface, it inherits only the method names and signatures, because the interface itself contains no implementations. For example, to implement an interface member, the corresponding member on the class must be public, non-static, and have the same name and signature as the interface member.

Properties and indexers on a class can define extra accessors for a property or indexer defined on an interface. For example, an interface may declare a property with a get accessor, but the class implementing the interface can declare the same property with both a get and set method. However, if the property or indexer uses explicit implementation, the accessors must match. Interfaces and interface members are abstract; interfaces do not provide a default implementation. For more information, see Abstract and Sealed Classes and Class Members.

```
public class Car : IEquatable<Car>
{
   public string Make {get; set;}
   public string Model { get; set; }
   public string Year { get; set; }

   // Implementation of
       IEquatable<T> interface
   public bool Equals(Car car) {
      if (this.Make == car.Make &&
         this.Model == car.Model &&
         this.Year == car.Year)  {
         return true;
      }
      else return false;
   }
}
```

Example 4.4

➢4.0 Types, Objects and Namespaces

4.1.6 Types -> Reference Types -> Self-Describing Types
Self describing types are further classified into array and class types.

4.1.7 Types -> Reference Types -> Self-Describing Types-> Arrays
In Chapter 1 (pages 36+) an introduction to C# arrays were given.

4.1.8 Types -> Reference Types -> Self-Describing Types-> User-Defined
Classes Classes are declared using the keyword class, as shown in example 4.5.

Classes create the template to instantiate objects in the computer memory. Objects of classes interact via the properties, methods and events defined within each class. Object of a class is created using the following paradigm:

```
class TestClass
{
   // Methods, properties, fields,
   // events, delegates
   // and nested classes go here.
}
```

Example 4.5

<Class Name> identifier = new <Class Name>();

For example, One object can send messages to methods of another object. One object can fire an event of another etc.

Unlike C++, only single inheritance is allowed in C#. In other words, a class can inherit implementation from one base class only. However, a class can implement more than one interface. The following table shows examples of class inheritance and interface implementation:

Inheritance	Example
None	class ClassA { }
Single	class DerivedClass: BaseClass { }
None, implements two interfaces	class ImplClass: IFace1, IFace2 { }
Single, implements one interface	class ImplDerivedClass: BaseClass, IFace1 { }

Figure 4.5

➢4.0 Types, Objects and Namespaces

The access levels **protected** and **private** are only allowed on nested classes.

You can also declare **generic classes** that have type parameters; see Generic Classes for more information.

A class can contain declarations of the following members:

Constructors
Destructors
Constants
Fields
Methods
Properties
Indexers
Operators
Events
Delegates
Classes
Interfaces
Structs

The example 4.6 demonstrates declaring class fields, constructors, and methods. It also demonstrates object instantiation and printing instance data. In this example, two classes are declared, the Child class, which contains two private fields (name and age) and two public methods. The second class, TestClass, is used to contain Main. The result of the above example is shown below.

```
Child #1: Craig, 11 years old.
Child #2: Sally, 10 years old.
Child #3: N/A, 0 years old.
```

Example 4.6

```
class Child
{
  private int age;
  private string name;

  // Default constructor:
  public Child()
  { name = "N/A";   }

  // Constructor:
  public Child(string name, int age)
  {
    this.name = name;
    this.age = age;
  }

  // Printing method:
  public void PrintChild()
  { Console.WriteLine("{0}, {1} years old.", name, age); }
}

class StringTest
{
  static void Main()
  {
    // Create objects by using the new operator:
    Child child1 = new Child("Craig", 11);
    Child child2 = new Child("Sally", 10);

    // Create an object using the default constructor:
    Child child3 = new Child();

    // Display results:
    Console.Write("Child #1: ");
    child1.PrintChild();
    Console.Write("Child #2: ");
    child2.PrintChild();
    Console.Write("Child #3: ");
    child3.PrintChild();
  }
}
```

➢4.0 Types, Objects and Namespaces

Notice, in example 3.5, the private fields (name and age) can only be accessed through the public methods of the Child class. For example, you cannot print the child's name, from the Main method, using a statement like this:

```
Console.Write(child1.name);   // Error
```

Accessing private members of Child from Main would only be possible if Main were a member of the class.

Types declared inside a class without an Access Modifier default to private, so the data members in this example would still be private if the keyword were removed.

Finally, notice that for the object created using the default constructor (child3), the age field was initialized to zero by default.

4.2 Static Classes and Static Class Members

Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class. Static class members can be used to separate data and behavior that is independent of any object identity: the data and functions do not change regardless of what happens to the object. Static classes can be used when there is no data or behavior in the class that depends on object identity.

4.2.1 Static Classes

A class can be declared static, indicating that it contains only static members. It is not possible to create instances of a static class using the new keyword. Static classes are loaded automatically by the .NET Framework common language runtime (CLR) when the program or namespace containing the class is loaded.

Use a static class to contain methods that are not associated with a particular object. For example, it is a common requirement to create a set of methods that do not act on instance data and are not associated to a specific object in your code. You could use a static class to hold those methods. The main features of a static class are:

1. They only contain static members
2. They cannot contain Instance Constructors
3. They cannot be instantiated.
4. They are sealed

Creating a static class is therefore much the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added. The compiler will guarantee that instances of this class cannot be created. Static classes are sealed and therefore cannot be inherited. Static classes cannot contain a constructor, although it is still possible to declare a static constructor to assign initial values or set up some static state.

4.0 Types, Objects and Namespaces

A static class example is the C#'s Math class. The Math class contains all static methods. It does not maintain any state; only a set of functions that 'gives' you certain set of values.

In order to create you own static class, just think of what you plan to do. For example say you want to have class to convert temperatures from Fahrenheit to Celsius and vise versa. For this all you need is to have two functions in a class. So we can write the class as static as shown in example 4.7-1

```
public static class TemperatureConverter
{
   public static double CelsiusToFahrenheit(string temperatureCelsius)
   {
      // Convert argument to double for calculations.
      double celsius = System.Double.Parse(temperatureCelsius);

      // Convert Celsius to Fahrenheit.
      double fahrenheit = (celsius * 9 / 5) + 32;

      return fahrenheit;
   }

   public static double FahrenheitToCelsius(
                              string temperatureFahrenheit)
   {
      // Convert argument to double for calculations.
      double fahrenheit = System.Double.Parse(temperatureFahrenheit);

      // Convert Fahrenheit to Celsius.
      double celsius = (fahrenheit - 32) * 5 / 9;

      return celsius;
   }
}
```

Example 4.7-1

Example 4.7-2 shows how the TemperatureConverter is used.

Notice that there is no need to create an instance of the class TemperatureConverter to use the methods of the class. Just by using the class paradigm ClassName.MethodName will suffice.

```
class TestTemperatureConverter
{
   static void Main()
   {
      System.Console.WriteLine("Please select the convertor direction");
      System.Console.WriteLine("1. From Celsius to Fahrenheit.");
      System.Console.WriteLine("2. From Fahrenheit to Celsius.");
      System.Console.Write(":");

      string selection = System.Console.ReadLine();
      double F, C = 0;

      switch (selection)
      {
         case "1":
            System.Console.Write("Please enter the Celsius temperature: ");
            F = TemperatureConverter.CelsiusToFahrenheit(System.Console.ReadLine());
            System.Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
            break;

         case "2":
            System.Console.Write("Please enter the Fahrenheit temperature: ");
            C = TemperatureConverter.FahrenheitToCelsius(System.Console.ReadLine());
            System.Console.WriteLine("Temperature in Celsius: {0:F2}", C);
            break;

         default:
            System.Console.WriteLine("Please select a convertor.");
            break;
      }
   }
}
```

Example 4.7-2

4.2.2Static Members

A static method, field, property, or event is callable on a class even when no instance of the class has been created. If any instances of the class are created, they cannot be used to access the static member. Only one copy of static fields and events exists, and static methods and properties can only access static fields and static events. Static members are often used to represent data or calculations that do not change in response to object state; for instance, a math library might contain static methods for calculating sine and cosine.

Static class members are declared using the static keyword before the return type of the member, for example, static members are initialized before the static member is accessed for the first time, and before the static constructor, if any is called. To access a static class member, use the name of the class instead of a variable name to specify the location of the member. For example:

Automobile.Drive();
int i = Automobile.NumberOfWheels;

```
public class Automobile
{
   public static int NumberOfWheels = 4;
   public static int SizeOfGasTank
   {
      get
      {
         return 15;
      }
   }
   public static void Drive() { }
   public static event EventType RunOutOfGas;

   //other non-static fields and properties...
}
```

Example 4.8

4.2.3 Using Properties

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a get accessor and/or a set accessor. The code block for the get accessor is executed when the property is read; the code block for the set accessor is executed when the property is assigned a new value. A property without a set accessor is considered read-only. A property without a get accessor is considered write-only. A property that has both accessors is read-write.

```
public class Date
{
 private int month = 7;
 public int Month
 {
  get
  {
   return month;
  }
  set
  {
   if ((value > 0) && (value < 13))
   { month = value; }
  }
 }
}
public class Program
{
 static void Main(string[] args)
 {
  Date date = new Date();
  date.Month = 12;
  System.Console.WriteLine("Month : " +
                  date.Month);
  System.Console.ReadLine();
 }
}
```

Example 4.9

Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a ref (C# Reference) or out (C# Reference) parameter.

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a get-accessor and/or a set accessor. See Example 4.9

➢4.0 Types, Objects and Namespaces

4.2.4 Auto-Implemented Properties

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects . When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field can only be accessed through the property's get and set accessors.

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
  // Auto-Impl Properties for trivial get and set
  public double TotalPurchases { get; set; }
  public string Name { get; set; }
  public int CustomerID { get; set; }

  // Constructor
  public Customer(double purchases, string name, int ID)
  {
    TotalPurchases = purchases;
    Name = name;
    CustomerID = ID;
  }
  // Methods
  public string GetContactInfo() {return "ContactInfo";}
  public string GetTransactionHistory() {return "History";}

  // .. Additional methods, events, etc.
}
```

Example 4.10-1

```
class Program
{
  static void Main()
  {
    // Intialize a new object.
    Customer cust1 = new Customer
                        ( 4987.63, "Northwind",90108 );

    //Modify a property
    cust1.TotalPurchases += 499.99;
  }
}
```

Example 4.10-2

4.2.5 Methods and Constructors

4.2.5.1 Methods

Methods are declared in a class or struct by specifying the access level such as public or private, optional modifiers such as abstract or sealed, the return value, the name of the method, and any method parameters. These parts together are the signature of the method. A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters. The class in example 4.11-1 contains three methods.

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas. The methods of the Motorcycle class can therefore be called as in the example 4.11-2

```
abstract class Motorcycle
{
   // Anyone can call this.
   public void StartEngine() {/* Method statements here */ }

   // Only derived classes can call this.
   protected void AddGas(int gallons) { /* Method statements here */ }

   // Derived classes can override the base class implementation.
   public virtual int Drive(int miles, int speed) {
    /* Method statements here */ return 1; }

   // Derived classes must implement this.
   public abstract double GetTopSpeed();
}
```

Example 4.11-1

```
class TestMotorcycle : Motorcycle
{

   public override double GetTopSpeed()
   {
      return 108.4;
   }

   static void Main()
   {

      TestMotorcycle moto = new TestMotorcycle();

      moto.StartEngine();
      moto.AddGas(15);
      moto.Drive(5, 20);
      double speed = moto.GetTopSpeed();
      Console.WriteLine("My top speed is {0}", speed);
   }
}
```

Example 4.11-2

4.2.5.2 Constructors

Whenever a class or struct is created, its constructor is called. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.

If you do not provide a constructor for your object, C# will create one by default that instantiates the object and sets member variables to the default values. Static classes and structs can also have constructors.

4.2.5.3 Using Constructors

Constructors are class methods that are executed when an object of a given type is created. Constructors have the same name as the class, and usually initialize the data members of the new object.

In the example 4.12, a class named **Taxi** is defined by using a simple constructor. This class is then instantiated with the new operator. The Taxi constructor is invoked by the new operator immediately after memory is allocated for the new object.

```
public class Taxi
{
   public bool isInitialized;
   public Taxi()
   {
      isInitialized = true;
   }
}

class TestTaxi
{
   static void Main()
   {
      Taxi t = new Taxi();
      Console.WriteLine(t.isInitialized);
   }
}
```

Example 4.12

4.2.6 Passing Parameters

In C#, parameters can be passed either by value or by reference. Passing parameters by reference allows function members (methods, properties, indexers, operators, and constructors) to change the value of the parameters and have that change persist. To pass a parameter by reference, use the ref or out keyword. For simplicity, only the ref keyword is used in the examples of this topic. For information on the difference between ref and out, see ref, out, and Passing Arrays Using ref and out.

4.2.6.1 Passing Value-Type Parameters

A value-type variable contains its data directly as opposed to a reference-type variable, which contains a reference to its data. Therefore, passing a value-type variable to a method means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no affect on the original data stored in the variable. If you want the called method to change the value of the parameter, you have to pass it by reference, using the ref or out keyword. For simplicity, the following examples use **ref**.

```
// PassingParams1.cs
using System;
class PassingValByVal
{
  static void SquareIt(int x)
  // The parameter x is passed by value.
  // Changes to x will not affect the original value of myInt.
  {
     x *= x;
     Console.WriteLine("The value inside the method: {0}", x);
  }
  public static void Main()
  {
     int myInt = 5;
     Console.WriteLine("The value before calling the method: {0}",  myInt);
     SquareIt(myInt);   // Passing myInt by value.
     Console.WriteLine("The value after calling the method: {0}", myInt);
  }
}
```

```
Output

The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 5
```

Example 4.13

4.2.6.2 Passing Reference-Type Parameters

A variable of a reference type does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data pointed to by the reference, such as the value of a class member. However, you cannot change the value of the reference itself; that is, you cannot use the same reference to allocate memory for a new class and have it persist outside the block. To do that, pass the parameter using the ref (or out) keyword. For simplicity, the following examples use **ref**.

The following example demonstrates passing a reference-type parameter, myArray, by value, to a method, Change. Because the parameter is a reference to myArray, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, myArray.

Example 4.14

```
// PassingParams2.cs
using System;
class PassingValByRef
{
   static void SquareIt(ref int x)
   // The parameter x is passed by reference.
   // Changes to x will affect the original value of myInt.     {
      x *= x;
      Console.WriteLine("The value inside the method: {0}", x);
   }
   public static void Main()     {
      int myInt = 5;
      Console.WriteLine("The value before calling the method: {0}",  myInt);
      SquareIt(ref myInt);   // Passing myInt by reference.
      Console.WriteLine("The value after calling the method: {0}",  myInt);
   }
}
```

```
Output
Inside Main, before calling the method, the first element is: 1
Inside the method, the first element is: -3
Inside Main, after calling the method, the first element is: 888
```

4.3 Namespaces

The namespace keyword is used to declare a scope. This namespace scope lets you organize code and gives you a way to create globally unique types.

Example 4.15 is a simple example. Within a namespace, you can declare one or more of the following types:

- another namespace
- class
- interface
- struct
- enum
- delegate

Whether or not you explicitly declare a namespace in a C# source file, the compiler adds a default namespace. This unnamed namespace, sometimes called the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access and this is not modifiable..

```
namespace SampleNamespace
{
   class SampleClass { }

   interface SampleInterface { }

   struct SampleStruct { }

   enum SampleEnum { a, b }

   delegate void SampleDelegate(int i);

   namespace SampleNamespace.Nested
   {
      class SampleClass2 { }
   }
}
```

Example 4.15

➢4.0 Types, Objects and Namespaces

4.3.1 Using Namespaces to control scope

The namespace keyword is used to declare a scope. The ability to create scopes within your project helps Organize code and provides a way to create globally-unique types. In the following example, a class entitled **SampleClass** is defined in two namespaces, one nested inside the other. The **dot(.)** Operator is used to differentiate which method gets called.

```
namespace SampleNamespace
{
  class SampleClass
  {
    public void SampleMethod()
    { System.Console.WriteLine("SampleMethod inside SampleNamespace"); }
  }

  // Create a nested namespace, and define another class.
  namespace NestedNamespace
  {
    class SampleClass
    {
      public void SampleMethod()
      { System.Console.WriteLine("SampleMethod inside NestedNamespace"); }
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      // Displays "SampleMethod inside SampleNamespace."
      SampleClass outer = new SampleClass();
      outer.SampleMethod();

      // Displays "SampleMethod inside SampleNamespace."
      SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
      outer2.SampleMethod();

      // Displays "SampleMethod inside NestedNamespace."
      NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
      inner.SampleMethod();
    }
  }
}
```

Example 4.16

4.4 Advanced Concepts : Relationships between classes

4.4.1 Composition and Aggregation

Composition gives the 'Part-Of' relationship. For example Composition is shown on a UML diagram in Figure 4.6 as a filled diamond.

Figure 4.6

If we were going to model a car, it would make sense to say that an engine is part-of a car. Within composition, the lifetime of the part (Engine) is managed by the whole (Car), in other words, when Car is destroyed, Engine is destroyed along with it. So how do we express this in C#?

```
public class Engine
{
  . . .
}
public class Car
{
   Engine e = new Engine();
   .......
}
```

Example 4.17

As you can see in the example code above, Car manages the lifetime of Engine.

4.0 Types, Objects and Namespaces

Aggregation is form of **Composition**. aggregation gives us a 'has-a' relationship. Within aggregation, the lifetime of the part is not managed by the whole. To make this clearer, we need an example. Aggregation would make sense in this situation, as a Customer 'has-a' Address. It wouldn't make sense to say that an Address is 'part-of' the Customer, because it isn't. Consider it this way, if the customer ceases to exist, does the address? I would argue that it does not cease to exists. Aggregation is shown on a UML diagram as an unfilled diamond as shown in the Figure 4.7 below.

Figure 4.7

So how do we express the concept of aggregation in C#? Well, it's a little different to composition. Consider the following code.

```
public class Address
{
 . . .
}

public class Person
{
   private Address address;

   public Person(Address address)
   {
      this.address = address;
   }
   . . .
}
```

Person would then be used as follows:

```
Address address = new Address();
Person person = new Person(address);

or

Person person = new Person( new Address() );
```

As you can see, Person does not manage the lifetime of Address. If Person is destroyed, the Address still exists. This scenario does map quite nicely to the real world.

Example 4.18

4.4.2 Inheritance

Classes can inherit from another class. Inheritance creates the **is-a** relationship. This is accomplished by putting a colon after the class name when declaring the class, and naming the class to inherit from—the base class—after the colon, as follows:

```
public class Animal
{
   public Animal() { }
}

public class Cat : Animal
{
   public Cat() { }
}
```

Example 4.19

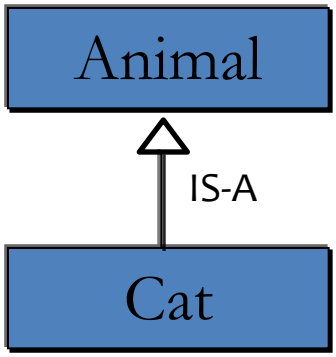

Figure 4.8

The new class—the derived class—then gains all the non-private data and behavior of the base class in addition to any other data or behaviors it defines for itself. The new class then has two effective types: the type of the new class and the type of the class it inherits.

In the example above, class Cat is effectively both Cat and Animal. When you access a Cat object, you can use the cast operation to convert it to an Animal object. The Cat object is not changed by the cast, but your view of the Cat object becomes restricted to Animal's data and behaviors. After casting a Cat to an Animal, that Animal can be cast back to a Cat. Not all instances of Animal can be cast to Cat—just those that are actually instances of Cat. If you access class Cat as a Cat type, you get both the class Animal and class Cat data and behaviors.

4.4.3 Object Casting

For objects (reference types) an explicit cast is required if you need to convert from a base type to a derived type:

4.0 Types, Objects and Namespaces

For example, let's create Giraffe class.

```
public class Giraffe: Animal
{
   public Giraffe() { }
}
```

Example 4.20

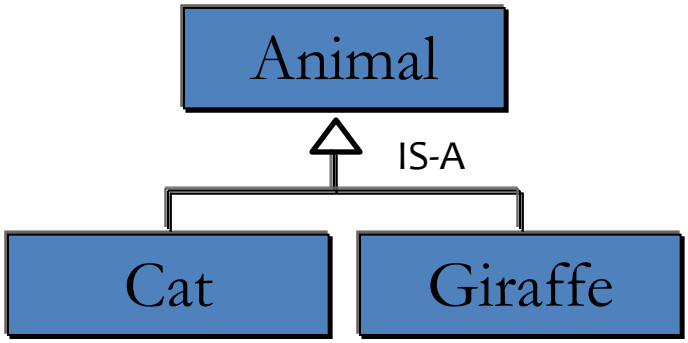

Figure 4.9

In example 4.21 below creates an instance of the Giraffe class. Then an Implicit conversion to Animal is made. Since the Giraffe is–a Animal this conversion is legal. But then in the latter part of the example, an Animal object is converted to a Giraffe object. This cannot be done directly as all Animals are not Giraffes. Therefore we must do a explicit conversion (Typically using type casting).

```
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back to derived type. Note: This will
//compile but throw an exception at run time if the right-side object is not in fact a
//Giraffe.
Giraffe g2 = (Giraffe) a;
```

Example 4.21

4.5 Partial classes

It is possible to split the definition of a class or a struct, or an interface over two or more source files. Each source file contains a section of the class definition, and all parts are combined when the application is compiled.

There are several situations when splitting a class definition is desirable: When working on large projects, spreading a class over separate files allows multiple programmers to work on it simultaneously.

When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when creating Windows Forms, Web Service wrapper code, and so on. You can create code that uses these classes without having to edit the file created by Visual Studio.

To split a class definition, use the partial keyword modifier, as shown in examples 4.22-1 and 4.22-2

```
//This part of the class is typed in Employee1.cs
public partial class Employee
{
   public void DoWork()
   {
   }
}
```

Example 4.22-1

```
//This part of the class is typed in Employee2.cs
public partial class Employee
{
   public void GoToLunch()
   {
   }
}
```

Example 4.22-2

4.6 Generics

What Are Generics?

Generics allow you to define type-safe classes without compromising type safety, performance, or productivity. You implement the server only once as A generic server, while at the same time you can declare and use it with any type. To do that, use the < and > brackets, enclosing a generic type parameter. For example, here is how you define and use a generic stack: (See example 4.23)

On the surface C# generics look syntactically similar to C++ templates, but there are important differences in the way they are implemented and supported by the compiler. This has significant implications on the manner in which you use generics.

The example 4.23 shows a very simple case of a Generic class called Stack. The statement Stack<T> means that an instance of the Stack class can be created for any type. This helps to avoid writing redundant code for all Stack types.

```
public class Stack<T>
{
  T[] m_Items;
  public void Push(T item)
  {...}
  public T Pop()
  {...}
}

public static void Main(String [] args)
{
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
}
```

Example 4.23