Language tools

Property

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they're public data members, but they're special methods called *accessors*. This feature enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. An init property accessor is used to assign a new value only during object construction. These accessors can have different access levels. For more information, see Restricting Accessor Accessibility.
- The value keyword is used to define the value being assigned by the set or init accessor.
- Properties can be *read-write* (they have both a get and a set accessor), *read-only* (they have a get accessor but no set accessor), or *write-only* (they have a set accessor, but no get accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as auto-implemented properties.

```
 public class TimePeriod
{
    private double _seconds;
    public double Hours
    {
        get { return _seconds / 3600; }
        private set
        {
            if (value < 0 || value > 24)
                throw new
ArgumentOutOfRangeException(nameof(value),
                      "The valid range is between 0
and 24.");
            _seconds = value * 3600;
```

Expression body definitions

Property accessors often consist of single-line statements that just assign or return the result of an expression. You can implement these properties as expression-bodied members. Expression body definitions consist of the => symbol followed by the expression to assign to or retrieve from the property.

Read-only properties can implement the get accessor as an expression-bodied member. In this case, neither the get accessor keyword nor the return keyword is used. The following example implements the read-only Name property as an expression-bodied member.

```
public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}
```

Both the get and the set accessor can be implemented as expression-bodied members. **In this case, the get and set keywords must be present**. The following example illustrates the use of expression body definitions for both accessors. The return keyword isn't used with the get accessor.

```
public string Name
    {
        get => _name;
        set => _name =
value;
    }
```

Auto-implemented properties

In some cases, property get and set accessors just assign a value to or retrieve a value from a backing field without including any extra logic. By using auto-implemented properties, you can simplify your code while having the C# compiler transparently provide the backing field for you. If a property has both a get and a set (or a get and an init) accessor, both must be auto-implemented.

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

Beginning with C# 11, you can add the required member to force client code to initialize any property or field:

```
public required decimal Price { get; set; }
```

Delegates

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.

- Delegates are used to pass methods as arguments to other methods.
- Event handlers are nothing more than methods that are invoked through delegates.

```
public delegate int PerformCalculation(int x, int y);
```

- Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate.

- The method can be either static or an instance method. This flexibility means you can programmatically change method calls, or plug new code into existing classes.
- Delegates are ideal for defining callback methods (A callback is a function that will be called when a process is done executing a specific task).
- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

The following example declares a delegate named Callback that can encapsulate a method that takes a string as an argument and returns void:

```
      public delegate void Callback(string message);
```

- A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with a lambda expression.
- The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. For example:

```
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
// Instantiate the delegate.
Callback handler = DelegateMethod;
// Call the delegate.
handler("Hello World");
```

- Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the Del type as a parameter:

```
public static void MethodWithCallback(int param1, int param2, Callback callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}

MethodWithCallback(1, 2, handler);
```

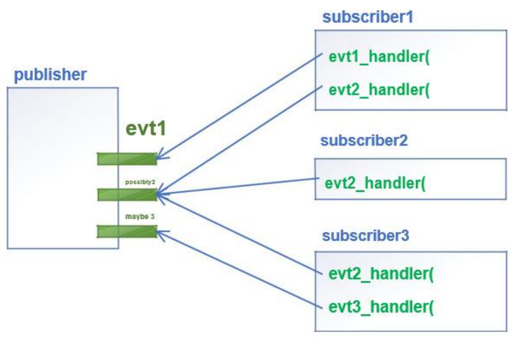
- A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback allMethodsDelegate = d1 + d2;
callMethodsDelegate += d3;
```

Events

Events in .NET are based on the delegate model. The delegate model follows the observer design pattern, which enables a subscriber to register with and receive notifications from a provider. An event sender pushes a notification that an event has happened, and an event receiver receives that notification and defines a response to it.



- An event is a message sent by an object to signal the occurrence of an action. The action can be caused by user interaction, such as a button click, or it can result from some other program logic, such as changing a property's value.
- The object that raises the event is called the *event sender*.
- The event sender doesn't know which object or method will receive (handle) the events it raises.
- The event is typically a member of the event sender; for example, the Click event is a member of the Button class, and the PropertyChanged event is a member of the class that implements the INotifyPropertyChanged interface.
- Typically, to raise an event, you add a method that is marked as protected and virtual (in C#)
- Name this method On*EventName*; for example, OnDataReceived.
- The method should take one parameter that specifies an event data object, which is an object of type EventArgs or a derived type.

The following example shows how to declare an event named ThresholdReached. The event is associated with the EventHandler delegate and raised in a method named OnThresholdReached.

```
class Counter
{
    public event EventHandler ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        ThresholdReached?.Invoke(this, e);
    }
}
```

.NET provides the EventHandler and EventHandler<TEventArgs> delegates to support most event scenarios. Use the EventHandler delegate for all events that don't include event data. Use the EventHandler<TEventArgs> delegate for events that include data about the event. These delegates have no return type value and take two parameters (an object for the source of the event and an object for event data).

For scenarios where the EventHandler and EventHandler<TEventArgs> delegates don't work, you can define a delegate. Scenarios that require you to define a delegate are rare, such as when you must work with code that doesn't recognize generics. You mark a delegate with the C# delegate and Visual Basic Delegate keyword in the declaration. The following example shows how to declare a delegate named ThresholdReachedEventHandler:

```
public delegate void ThresholdReachedEventHandler(object sender, ThresholdReachedEventArgs e);
```

Event data

- Data that is associated with an event can be provided through an event data class. .NET provides many event data classes that you can use in your applications. For example, the SerialDataReceivedEventArgs class is the event data class for the SerialPort.DataReceived event.
- .NET follows a naming pattern of ending all event data classes with EventArgs.
- The EventArgs class is the base type for all event data classes.
- EventArgs is also the class you use when an event doesn't have any data associated with it.
- When you create an event that is only meant to notify other classes that something happened and doesn't need to pass any data, include the EventArgs class as the second parameter in the delegate.
- You can pass the EventArgs.Empty value when no data is provided. The EventHandler delegate includes the EventArgs class as a parameter.
- When you want to create a customized event data class, create a class that derives from EventArgs

```
public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

Event handlers

To respond to an event, you define an event handler method in the event receiver. This method must match the signature of the delegate for the event you're handling. In the event handler, you perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button. To receive notifications when the event occurs, your event handler method must subscribe to the event.

The following example shows an event handler method named c_ThresholdReached that matches the signature for the EventHandler delegate.

```
class ProgramTwo
{
    static void Main()
    {
        var c = new Counter();
        c.ThresholdReached += c_ThresholdReached;
    }
    static void c_ThresholdReached(object sender, EventArgs e)
    {
        Console.WriteLine("The threshold was reached.");
    }
}
```

How are events different compared to delegates:

- An event is a delegate object defined using the event keyword.
- Events can only be defined as a member of a class (we cannot define events as local variables, whereas delegates we can).
- The = operator cannot be used just the += and the -= ones. This is important, as you can’t overwrite the list of registered event handlers by accident.
- Events can be fired only inside of the owner class (publisher).

Attributes

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.
- They are the equivalent of the Java annotations.
- Attributes can be used for many purposes. E.g. the Serializable attribute can indicate that a class can be serialized into a stream

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets ([]) above the declaration of the entity to which it applies.

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

- C# provides several predefined attributes that offer information to the compiler or runtime.
- Examples include `[Serializable]`, `[Obsolete]`, `[Conditional]`, and `[DllImport]`.
- You can define your own custom attributes by creating a class that inherits from the `System.Attribute` class.
- The convention is to name custom attributes with the suffix "Attribute."

Value and reference types

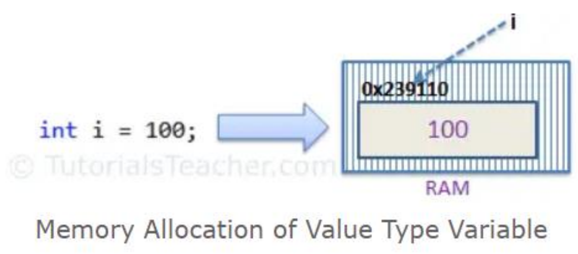
In .NET every type is either a value or reference type. This defines how it behaves in several ways.

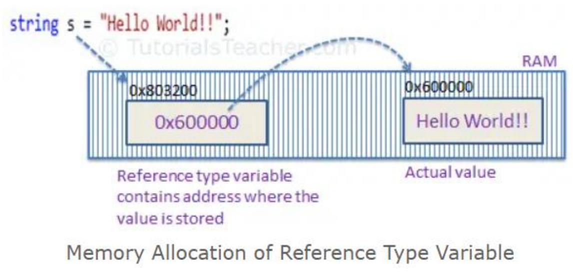
The most imporant properties:

- **Value type:** The variable contains the actual value (not a pointer to the value). This category includes basic simple types like int, char, decimal, double, float, bool, enum, etc., as well as **struct**s. When part of a complex type, they are stored in memory in an ’inline’ way. When used as local variables, memory for them is allocated on the stack. When passed as function parameters or return types, they behave just as for example int in C++: a copy is passed. They have a significant performance advantage compared to reference types: allocation is fast. Caveats: they can’t derive from another type and no type can derive from them. However, they can implement interfaces (more on this later).

- **Reference type:** they are made up of two parts: a pointer/reference which is null by default and the refered to object, which has memory allocated for it on the heap. As such, it is disposed of by the garbage collector and allocated using the **new** keyword. Reference types include all .NET classes from the Base Class Library (for example string, File), as well as arrays([]) and every type we define using the **class** keyword. Interfaces are also reference types (more on this later).

To sum up, whenever we define a class, it will behave exactly as classes in Java. When we use the simple types (int, char, ..), an enum or a struct, then these behave as they do in C/C++ (their composites behave the same as long as we don't use references).





string s = "Hello World!!";
© TutorialsTeacher.com
RAM
0x803200
0x600000
0x600000
Hello World!!
Reference type variable contains address where the value is stored
Actual value
Memory Allocation of Reference Type Variable

The following keywords are used to declare reference types:

- class
- interface
- delegate
- record

C# also provides the following built-in reference types:

- dynamic
- object
- string

Built-in reference types (C# reference)

The object type

The object type is an alias for System.Object in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from System.Object. You can assign values of any type to variables of type object.

The string type

The string type represents a sequence of zero or more Unicode characters. string is an alias for System.String in .NET.
Although string is a reference type, the equality operators == and != are defined to compare the values of string objects, not references. Value based equality makes testing for string equality more intuitive. For example:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b)); // Output: Determines whether the specified Object instances are the same instance.
// Output: True, False
```

The previous example displays "True" and then "False" because the content of the strings is equivalent, but a and b don't refer to the same string instance.

Strings are *immutable*--the contents of a string object can't be changed after the object is created. For example, when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to b. The memory that had been allocated for b (when it contained the string "h") is then eligible for garbage collection.

Code examples:

Property

```
class Person
{
    private string name;
    private int yearOfBirth;

    // Declare a Name property of type string:
    public string Name
    {
        get { return name; }
        set
        {
           if (value == null)
                  throw new ArgumentNullException("The Name property can not be null.”);
           name = value;
        }
    }

    public int YearOfBirth
    {
        get
        {
           return yearOfBirth;
        }
        set
        {
            if (yearOfBirth < 1800 && yearOfBirth > 5000)
                  throw new ArgumentException(„Invalid yearOfBirth value.”);
           yearOfBirth = value;
        }
    }

    // Calculated, read-only.
    public int Age
    {
        get { return DateTime.Now.Year-Age; }
    }
// We need to be careful and use properties consistently everywhere!
    // We use the property to set the value in order to have the validation in place
    public Person(string myName, int yearOfBirth)
    {
        Name = name;
        YearOfBirth = yearOfBirth;
    }
 }

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person(„Steve”, 1980);
        p1.YearOfBirth = 1995; // calls set
        Console.WriteLine("Name: {0}, kor: Age:{1}", p1.Name, p1.Age); // calls get

        p1.Age = 20; // Compiler error, set not define
        p1.YearOfBirth = 1000; // Runtime exception
        ...
    }
}
```

Delegate:

```
class Complex
{
    public double Re, Im;
//  Constructor
    public Complex(double re, double im)
    {
        this.Re = re;
        this.Im = im;
    }
 }

delegate bool FirstIsSmallerDelegate(object a, object b);

class Sorter
{
    // Sorting the list by the help of a delegate parameter.
    // We would like to use it for any kind of type. We don't have the code for the
    // Complex class, we can not make the Complex class implement the
    // IComparable interface. Solution: passing the comparing function as a parameter.
    public static void HyperSort(ArrayList list,
        FirstIsSmallerDelegate firstIsSmaller)
    {
        for (...)
        {
           ...
                if (firstIsSmaller(list[j], list[j - 1]))
                  ...
            }
        }
    }

}

class Program
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();
        list.Add(new Complex(1, 2)); // Insert one element, sorting must work for it :).
        // ...

        // Method reference for a static method
        Sorter.HyperSort(list, new FirstIsSmallerDelegate(FirstIsSmaller_Complex));
        // More simple syntax
        Sorter.HyperSort(list, FirstIsSmaller_Complex);

        // Method reference for a method of an object (a bit forced)
        // this could also be static
        Comparers comps = new Comparers();
        Sorter.HyperSort(list, new FirstIsSmallerDelegate(comps.FirstIsSmaller_Complex));

        // The delegate is a type. It could be a local or a member variable as well.
            FirstIsSmallerDelegate fis1 = new FirstIsSmallerDelegate(FirstIsSmaller_Complex);
        bool isFIS = fis1(new Complex(1, 1), new Complex(2, 2));

        // Every delegate is derived from MultiCastDelegate. It can hold references for
        // multiple methods. The += operators can be used to add new references.
        // In the following example the FirstIsSmaller_Complex will be called twice.
        // It does not make sense here but this feature will be important for events.
        fis1 += FirstIsSmaller_Complex;
        fis1(new Complex(1, 1), new Complex(2, 2));
    }
    public static bool FirstIsSmaller_Complex(object a, object b)
    {
Complex ca = (Complex)a;
        Complex cb = (Complex)b;
        return Math.Sqrt(...);
    }
}


class Comparers
{
    public bool FirstIsSmaller_Complex(object a, object b)
    {
        // just like in the Program class
        --||--
    }

    public bool FirstIsSmaller_Person(object a, object b)
    {
        ...
    }
}
```

Event:

```
public delegate void LogHandler(string msg);

class Logger
{
    // A class can have an event as well
    public event LogHandler Log;
    // Other events may come here

    public void WriteLine(string msg)
    {
        if (Log != null)
            Log(msg);
    }
}

class FileLogListener
{
    // FileStream tag.
    public void WriteToFile(string msg)
    {
        // ...
    }
}

class App
{
    Logger log = new Logger();
    FileLogListener fileLogListener = new FileLogListener();

    public App()
    {
        log.Log += new LogHandler(WriteToConsole);
        log.Log += new LogHandler(fileLogListener.WriteToFile);
    }

    public void Process()
    {
        log.WriteLine("Process begin...");
        //...
        log.WriteLine("Process end...");
    }

    public void Cleanup()
    {
        log.Log -= new LogHandler(writeConsole);
    }

    void WriteToConsole(string msg)
    {
        Console.WriteLine(msg);
    }
}

class Program
{
    static void Main(string[] args)
    {
        App app = new App();
        app.Process();
    }
}
```

```
Attribute:
[Serializable] // The class is serializable
class User
{
    string name;

    [NonSerialized] // This field should not be serialized
    string password;

    // Security requeirements
    [PrincipalPermission(SecurityAction.Demand, Role = "Admin")]
    public static void DeleteUser(int userId)
    {
    }

    // ...
}

class Program
{
    static void Main(string[] args)
    {
        User user = new User();
        // setting the parameters ...

        //  Serialization into a filestream
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream1 = new FileStream("Dump.dat", FileMode.Create);
        formatter.Serialize(stream1, user);
        stream1.Close();

        //  Deserialization from a filestream
        FileStream stream2 = new FileStream("Dump.dat", FileMode.Open);
        User u = (User)formatter.Deserialize(stream2);
        stream2.Close();
    }
}
```

Value and Reference types:

```
class Person
{
    public string Name { get; set; }
}

Person person1 = new Person { Name = "Alice" };
Person person2 = person1; // Both person1 and person2 now refer
to the same object

person2.Name = "Bob"; // Changes the object's Name property
through person2
Console.WriteLine(person1.Name); // Output: Bob, since both
variables refer to the same object
```

```
class Person
{
    public string Name { get; set; }
}

Person person1 = new Person { Name = "Alice" };
Person person2 = person1; // Both person1 and person2 now refer
to the same object

Person person3 = new Person { Name = "Alice" };
Person person4 = new Person { Name = "Alice" };

bool isEqual1 = person1 == person2; // true, since both
variables refer to the same object
bool isEqual2 = person1 == person3; // false, even though the
properties are the same, they refer to different objects
bool isEqual3 = person3 == person4; // false, since they refer
to two different objects with different memory addresses
```