C# Programming

Notes by JACPro

C# 1.x Notes

These notes were written when directly studying from the C# documentation page "The History of C#". I read through all documentation pages for the listed "major features of C# 1.0" and all relevant documentation linked by each of them.

This was done in preparation for reading Jon Skeet's *C# In Depth*, as he teaches the reader about C#'s many features by exploring each released version of C# in order. Skeet builds off an assumed knowledge of the features released as part of C# 1.x.

Table of Contents

C# 1.x Notes 2

Classes and Structs 5

Classes 5

Inheritance 5

Abstract Classes 6

Sealed Classes 6

Structs 6

Instantiating Structs 7

Interfaces 9

Interfaces 9

Interface Properties 12

Delegates 13

Delegates 13

Events 14

Events 14

Properties 16

Properties 16

Auto-Implementation 17

Read-Only Properties 18

Computed Properties 19

Cached Evaluated Properties 19

Attaching attributes to auto-implemented properties 21

Implementing INotifyPropertyChanged 21

Operators and Expressions 23

Operators and Expressions 23

Operator Associativity 26

Operand Evaluation 26

Statements 27

Statements 27

Types of Statements 28

Declaration Statements 29

Expression Statements 29
The empty statement 29
Embedded Statements 30
Nested statement blocks 31
Unreachable statements 31
Attributes 32
Attributes 32
Declaring Attributes 32
Properties of Attributes 33
Attribute Parameters 33
Attribute Targets 34
Expression-Bodied Members 36
Expression-Bodied Members 36

Classes and Structs

Classes

Classes are like blueprints for declaring objects with certain rules, attributes, and capabilities. They define the kinds of data and functionality their objects will have.

Classes can be declared as follows:

```
public class Customer
{
	//fields
	//properties
	//methods
	//events
}

Customer object2;
```

The above code declares type and namespace but will be null unless an object is assigned.

You can create and assign an object as follows:

```
Customer object2 = new Customer();
```

Classes are reference types:

```
Customer object3 = object2;
```

`object2` and `object3` refer to the same object, so changes to one will be reflected in the other.

Inheritance

You can only inherit from a single class.

```
public class Manager : Employee
{
	//fields, properties, methods, and events inherited from Employee
```

```
	//new fields, properties, methods, and events for Manager
}
```

Abstract Classes

The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

Abstract classes cannot be instantiated.

```
public abstract class A
{
	//Class members here
}
```

Sealed Classes

Sealed classes cannot be used as base classes because they prevent derivation.

```
public sealed class D
{
	//Class members here
}
```

A method, index, property, or event can be declared as sealed in a derived class when overwriting a virtual member of the base class; this prevents the virtual aspect from being derived further.

```
public class D : C
{
	public sealed override void DoWork() { }
}
```

Structs

A structure type is a value type that can encapsulate data and related functionality. Use the "struct" keyword to define a structure type.

```
public struct Coords
{
	public Coords (double x, double y)
	{
		X = x;
```

```
            Y = y;
      }

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

      public override string ToString() =>$“({X}, {Y})”;
}
```

Typically, you use structure types to design small data-centric types that provide little or no behaviour.

You can also declare readonly structs using the readonly keyword.

```
public readonly struct Coords
{
      //can’t declare a parameterless constructor
      public Coords (double x, double y)
      {
            //struct constructor must initialize all instance fields
            X = x;
            Y = y;
      }

      //can’t initialize instance fields or properties at declaration
      //but can initialize static or const field or static property at
      declaration

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

      public override string ToString() =>$“({X}, {Y})”;
}
```

Structs can’t inherit from other classes or structs and they can’t be inherited from. However, they can implement interfaces.

Instantiating Structs

Typically, structs are instantiated using the new keyword; in addition to explicitly declared constructors, structs have an implicit parameterless constructor, which produces the default value of the type.

If all instance fields are accessible, structs can also be instantiated without the “new” operator:

```
public struct Coords
{
	public double x;
	public double y;
}

public static void Main()
{
	Coords p;
	p.x = 3;
	p.y = 4;
}
```

Interfaces

Interfaces

Interfaces contain definitions for a group of related functionalities that a non-abstract class or struct must implement. Interfaces may define static methods, which must have implementation. Since C# 8.0, interfaces can define default implementations for members. Interfaces can't declare instance data (e.g. fields, auto-implemented properties, and property-like events).

While C# doesn't support multiple inheritance, it does allow you to implement an unrestricted number of interfaces, meaning a class can use behaviour from many different places. Additionally, structs are unable to inherit from classes or other structs, but they can implement interfaces, allowing structs to include behaviour from other sources.

Interfaces can be defined using the interface keyword, as follows:

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

By convention, interface names should start with a capital I.

Any methods declared by an interface must be implemented by any class or struct that implements that interface, with a signature matching the one specified by the interface. For example, a class that implements IEquatable must always have a method called Equals, which takes a parameter of type T and returns a Boolean value.

The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`; it must be implemented by the class or struct that implements the interface.

Interfaces can contain:

- Instance methods
- Properties
- Events
- Indexers
- Static Constructors
- Fields

- Constants
- Operators

Interfaces can't contain:

- Instance fields
- Instance constructors
- Finalizers

By default, interface members are public.

To implement an interface member, the corresponding member of the class that implements it must be public, non-static, and have an identical name and signature to the interface member.

When implementing an interface, a class or struct must provide an implementation for every member which is declared but not implemented by the interface. However, if a base class implements an interface, any class derived from that base class will inherit the base class' implementations for all interface members.

You can implement interfaces as follows:

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

	//Implementation of the Equals method defined by IEquatable<T>
	public bool Equals (Car car)
	{
		return (this.Make, this.Model, this.Year)
		== (car.Make, car.Model, car.Year);
	}
}
```

If implementing two interfaces that both contain members with the same signature, both interfaces will use the same implementation for those members.

```
public interface IControl
{
	void Paint();
}

public interface ISurface
```

```
{
      void Paint();
}

public class SampleClass : IControl, ISurface
{
      public void Paint()
      {
            Console.Writeline("Paint method in SampleClass");
      }
}
```

In the above code, calls to the paint method of `SampleClass`, `IControl` and `ISurface` will all call the same method. In order to implement different methods of the same name each interface, you can use explicit interface implementation.

```
public class SampleClass : IControl, ISurface
{
      void IControl.Paint()
      {
            Console.Writeline("Paint method for IControl");
      }

      void ISurface.Paint()
      {
            Console.Writeline("Paint method for ISurface");
      }
}
```

Class indexers and properties can define additional accessors for properties and indexers defined by an interface. For example, if an interface defines a property with only a get accessor, the class implementing the interface can go on to declare both get and set accessors. However, if the property uses explicit implementation (see above), the accessors must match.

Interfaces may also inherit from an unrestricted number of interfaces; derived interfaces will inherit all members from the base interfaces, including members inherited by those base interfaces. A class inheriting a derived interface can be implicitly converted to any of its base interfaces.

A class may implement an interface multiple times by implementing a base interface and an interface derived from it; however, a class may only provide one implementation of any given interface.

If an interface is inherited because the base class implements it, the implementation of the interface from the base class will be provided. However, any virtual interface members can be overwritten with new implementations if required.

When interfaces declare default implementations for methods, classes implementing those interfaces will inherit those implementations; implementations defined in interfaces are virtual, so the implementing class can override them with a new implementation.

Interface Properties

Unlike fields, properties can be declared in an interface.

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

Interface properties don't usually have a body; unlike in classes and structs, declaring property accessors without a body won't auto-implement the property.

Delegates

Delegates

A delegate is a type that represents references to methods with a particular parameter list and return type. When instantiating a delegate, its instance can be associated with any method with a compatible signature and return type. The method can then be called through the delegate instance.

Delegates are used to pass methods to other methods as arguments. Event handlers are methods invoked through delegates; you create a custom method and a class can call that method when a certain event occurs.

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

Any accessible class or struct with a method matching the delegate type can be assigned to the delegate. Both static and instance methods can be used.

Delegates make it possible to programmatically change method calls and plug new code into existing classes.

Delegates are ideal for callback methods; for example, a method that compares two objects could be passed by reference as an argument to a sorting algorithm. Since the comparison code is in a separate procedure, the sorting algorithm could be written in a more general way (i.e. pass different comparison methods to the sorting algorithm so that the same algorithm can be used for sorting various different types of data).

Events

Events

Events are a **late binding mechanism** (i.e. the method/function being called is looked up at runtime so the method itself can be passed as an argument) built upon the language support for delegates.

Events are used by objects to broadcast to components listening for this event that something has happened.

Many graphical systems use events to report user interaction such as mouse movements or key presses, though events are also used in other scenarios.

You can define events to be raised by your classes. Note that there may not be any objects registered for any given event; code must be written so that events are not raised unless listeners are configured.

Also note that subscribing to events creates a coupling between the object of the event and the subscribed object that's listening (**event source** and **event sink**). You **must** ensure that event sinks unsubscribe from the event source when no longer interested in events from that source.

To define an event, use the `event` keyword:

```
public event EventHandler<FileListArgs> Progress;
```

The type of the event must be a **delegate type**. There are certain **conventions** to follow when declaring an event:

- Event delegate type should have a void return.
- Event declarations should be a **verb** or verb phrase.
    - Use past tense verbs to report something that has happened.
    - Use present tense verbs to report something that is about to happen.

Using present tense often indicates that the event class is customisable in some way. A common scenario is to support cancellation; for example, a `Closing` event may include an argument that would indicate whether the close operation should continue. Another reason is to enable callers to an event to modify the object's behaviour by updating the properties of

the event argument. You could raise an event to indicate the next action an algorithm proposes to take; the event handler may then mandate a different action by modifying the properties of the event argument.

To raise an event, call the event handlers using the delegate invocation syntax:

```
Progress?.Invoke(this, new FileListArgs(file));
```

The `?.` operator makes it easy to ensure that you do not attempt to raise the event when there are no subscribers to that event.

You can subscribe to an event using the **+=** operator:

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

You can unsubscribe using the `-=` operator:

```
fileLister.Progress -= onProgress;
```

It's important to declare a local variable for the expression that represents the event handlers; this ensures that the unsubscribe action removes the event handler.

Properties

Properties

Properties behave like fields when accessed; however properties are implemented with accessors that allow additional statements to be executed when they are accessed or assigned. Put simply, properties will appear as fields to developers accessing them, however, unlike fields, you may implement them with additional functionality such as validation, custom accessibility, or lazy evaluation.

The syntax for an **auto-implemented property** is as follows:

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

You can initialize a property to a value other than its default type by assigning a value after the closing brace for the property. For example:

```
public string FirstName { get; set; } = string.Empty;
```

Instead of using an auto property, you may define the fields used for storage by the property manually:

```
public string FirstName
{
      get { return firstName; }
      set { firstName = value; }
}
private string firstName;
```

When a property implementation is a single expression, you can use **expression-bodied members** for the getter or setter:

```
public string FirstName
{
      get => firstName;
      set => firstName = value;
}
private string firstName;
```

The `set` accessor always has a single parameter named `value`. The `get` accessor must return a value that is convertible to the type of the property (e.g. in the string FirstName property, a string must be returned).

You can write code in the `set` accessor to validate property values; for example, you could ensure a string property is not null or white space:

```
public string FirstName
{
      get => firstName;
      set
      {
            if (string.IsNullOrWhiteSpace(value))
            {
                  throw new ArgumentException (“First name must
                        not be blank”);
            }
            firstName = value;
      }
}
private string firstName;
```

You can simplify the preceding example by using the thrown expression as part of the property setter validation:

```
public string FirstName
{
      get => firstName;
      set => firstName = (!string.IsNullOrWhiteSpace(value))?value : throw
            new ArgumentException (“First name must not be blank”);
}
```

Auto-Implementation

Automatic properties are used to make private variables visible outside the class by using properties with get; set; methods to expose private fields.

- Fields are normal member variables of a class.
- Properties enable a class to publicly expose methods for getting and setting private fields. Because they provide a definition but no implementation, they can be used by interfaces (unlike fields).

```
public int SomeProperty { get; set;}
```

Automatic properties are used when no additional logic is required in the property accessors (i.e. using only standard get; set; accessors).

Automatic properties are shorthand for the following non-automatic property code:

```
private int _someField;
public int SomeProperty
{
      get { return _someField; }
      set { _someField = value; }
}
```

The above code shows how you might implement a property that was declared in an interface (or how to declare and implement a property in a class).

Uses of properties include:

- Validating data before allowing a change.
- Transparently exposing data in a class where that data is actually retrieved from some other source (e.g. a database).
- They can take an action when data is changed, such as raising an event or changing the value of other fields.

Read-Only Properties

You may assign different accessibility levels to get and set accessors, rather than leaving them as public; for example, you could have the set accessor as private so that only other methods in the same class are able to change its value.

```
public string FirstName { get; private set; }
```

While the value in the example above can still be viewed from anywhere, it can now only be changed from within the class the contains the property.

You can place any access modifier on the accessors of a property (i.e. `private`, `protected`, `internal`, `protected internal`). However, access modifiers on accessors must be more restrictive than that on the property itself (e.g. you can't declare a private property with public accessors).

You may also restrict modifications to a property so that a value can only be set in an initializer or constructor:

```
public Person (string firstName) => this.FirstName = firstName;
public string FirstName { get; }
```

The above feature is primarily used for initializing collections that are exposed as read-only properties:

public class Measurements

```
{
      public ICollection<DataPoint> points { get; } = new
            List<DataPoint>();
}
```

Computed Properties

Properties may return computed values; they are not restricted to only returning the value of a member field:

```
public class Person
{
      public string FirstName { get; set; }
      public string LastName { get; set; }
      public string FullName { get {return $"{FirstName} {LastName}"; } }
}
```

The above example uses the $ character for string interpolations; this allows for interpolation expressions (such as `{FirstName}`) and so provides a more readable and convenient syntax for writing formatted strings.

You could instead use an expression-bodied member to achieve the same effect in a more succinct way. Expression-bodied members use the lambda expression syntax and define methods that contain a single expression only.

```
public class Person
{
      public string FirstName { get; set; }
      public string LastName { get; set; }
      public string FullName => $"{FirstName} {LastName}";
}
```

Cached Evaluated Properties

You can store a computed property to create a cached evaluated property. For example, the FullName property in the above examples could be cached after being evaluated, meaning the string formatting will only occur the first time the property is accessed:

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

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

	private string fullName;
	public string FullName
	{
		get
		{
			if (fullName == null)
			{
				fullName = $"{FirstName} {LastName}";
			}
			return fullName;
		}
	}
}
```

In practice, the above code would be impractical; while the fullName field may be accurate the first time it's accessed, it won't change if the first or last name is changed. For example, if the full name is "John Smith" when the FullName getter is first called, the correct name will be returned. However, if the first name is then changed to "Tom", the fullName field won't be updated to reflect this, meaning all future calls to the FullName getter will still return "John Smith".

This can be amended by having the set accessors for FirstName and LastName reset the value of the fullName field to null (meaning it will be evaluated again the next time it's called):

```
public class Person
{
	private string firstName;
	public string FirstName
	{
		get => firstName;
		set
		{
			firstName = value;
			fullName = null;
		}
	}




	private string lastName;
	public string LastName
	{
```

```
            get => lastName;
            set
            {
                  lastName = value;
                  fullName = null;
            }
      }

      private string fullName;
      public string FullName
      {
            get
            {
                  if (fullName == null)
                  {
                        fullName = $"{FirstName} {LastName}";
                  }
                  return fullName;
            }
      }
}
```

In this version of the code, the value of fullName is only calculated when it's needed. Note that while these changes have made the code cleaner and more efficient, developers that use this class do not need to know the details about its implementation and so they will use the Person class in the same way regardless of which of the implementations listed above is used. This is the primary reason for using properties to expose data members of an object.

Attaching attributes to auto-implemented properties

In auto-implemented properties, you can still attach field attributes to the compiler generated backing field by using `NonSerializedAttribute`; this can only be attached to fields, not properties:

```
[field:NonSerialized]
public int Id { get; set; }
```

Implementing INotifyPropertyChanged

You may need to write code in a property accessor to support the `INotifyPropertyChanged` interface, which is used to notify data binding clients when a

property's value changes. When such a change occurs, the `INotifyPropertyChanged.PropertyChanged` event will be raised, meaning the data binding libraries will then be able to appropriately update display elements based on that change. This could be implemented for the FirstName property of the example Person class in the following way:

```
public class Person : INotifyPropertyChanged
{
      public string FirstName
      {
            get => firstName;
            set
            {
                  if (value != firstName)
                  {
                        PropertyChanged?.Invoke(this, new
                            PropertyChangedEventArgs(nameof(firstName)));
                  }
                  firstName = value;
            }
      }
      private string firstName;

      public event PropertyChangedEventHandler PropertyChanged'
}
```

NOTE: The ?. operator is the null conditional operator; this checks for and handles NullReferenceExceptions before evaluation the right side of the operator. In the case of our example, a `NullReferenceException` would be thrown if the PropertyChanged event is raised while there are no subscribers; however, the ?. stops the event from ever being raised under these circumstances.

NOTE: The `nameof` operator is used above to convert the property name symbol to a string representation. It's common to use `nameof` to avoid errors where a property name may be mistyped or changed.

Operators and Expressions

Operators and Expressions

C# provides many operators, most of which are supported by the built-in types (bool, int string etc.), which allow you to perform basic operations with the values of those types. The supported operators include:

- Arithmetic operators – Perform arithmetic operations with numeric operands.
- Comparison operators – Compare numeric operands.
- Boolean logical operators – Perform logical operations with bool operands.
- Bitwise and shift operators – Perform bitwise or shift operations with operands that are one of the integral types.
- Equality operators – Check if the given operands are equal or not.

These operators can typically be overloaded.

NOTE: Operator overloading is where a type provides a custom implementation of an operation in the case that one or both operands are of that type. With this, you can define custom structures to represent data in a useful way (e.g. to represent functions).

As in maths, you can change the order in which operators in an expression are performed by using parentheses. This allows you to manipulate operator precedence.

Other kinds of expressions that C# provides include:

- Interpolated string expressions – these provide a convenient syntax for creating formatted strings:

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.Pi * r *
            r:F3}.";
```

- Lambda expressions – these allow you to create anonymous functions:

```
int[] numbers = {2, 3, 4, 5};
var maxSquare = numbers.Max(x => x * x);
Console.WriteLine(maxSquare); //calculates largest square from numbers in
array
```

- Query expressions – these allow you to use query capabilities (similar to SQL) directly in C#:

```
var scores = new[] {90, 97, 78, 68, 85};
IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select sore;
Console.WriteLine(string.Join(" ", highScoresQuery);
//Output: 97, 90, 85
```

Operator Precedence

In expressions with multiple operators, operators will be evaluated in order of precedence. In mathematical operations, the BODMAS precedence order applies, meaning that in a sum with both addition and multiplication, the multiplication will be performed first:

```
var a = 2 + 2 * 2; //a = 6
```

As with BODMAS, you can change the order of precedence using brackets:

```
var a = (2+2) * 2; //a = 8
```

The following table lists the C# operators in order of precedence from highest to lowest:

Operators	Category or name
x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Primary
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unary
x..y	Range
switch	`switch` expression
with	`with` expression
x * y, x / y, x % y	Multiplicative
x + y, x – y	Additive
x << y, x >> y	Shift
x < y, x > y, x <= y, x >= y, is, as	Relational and type-testing
x == y, x != y	Equality
`x & y`	Boolean logical AND or bitwise logical AND
`x ^ y`	Boolean logical XOR or bitwise logical XOR
`x | y`	Boolean logical OR or bitwise logical OR
x && y	Conditional AND
x || y	Conditional OR
x ?? y	Null-coalescing operator
c ? t : f	Conditional operator
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x ??= y, =>	Assignment and lambda declaration

Operator Associativity

When two operators in an expression have the same precedence, associativity determines the order in which they will be performed.

- Left-associative operators are evaluated from left to right. Excluding assignment operators and null-coalescing operators, all binary operators are left-associative: i.e. `a + b + c` will be calculated as `(a + b) + c`

- Right-associative operators are evaluated from right to left. Assignment operators, null-coalescing operators and the ?: conditional operator are right-associative: i.e. `x = y = z` will be calculated as `x = (y = z)`

Operand Evaluation

Operands in an expression are evaluated from left to right, ignoring operator precedence and associativity. For example:

Expression	Order of Evaluation
a + b	a, b, +
a + b * c	a, b, c, *, +
a / b + c * d	a, b, /, c, d, *, +
a / (b + c) * d	a, b, c, +, /, d, *

While all operands are usually evaluated, some are only evaluated conditionally; certain operators may cause only certain operands to be evaluated. These operands are: the logical `AND` and `OR`, null-coalescing operators `??` and `??=`, null-conditional operators `?.` and `?[]`, and conditional operator `?:`.

Statements

Statements

Statements are the actions that a program takes. They encompass many things in C# code, but common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to other code blocks given certain conditions.

The order in which these statements are executed is known as the flow of control/flow of execution. The flow of control may vary each time a program is run, as different user inputs can cause different conditions to be met, causing branches to lead to other code blocks.

Statements can consist of single lines of code that end in a semi-colon or a series of single-line statements in a block. Statement blocks are enclosed in {} brackets and can contain nested statement blocks.

The following code demonstrates various types of statements and their names:

```
public class SimpleStatements
{
	static void Main()
	{
		int counter; //declaration statement
		counter = 1; //assignment statement

		//declaration statements with initializers
		int[] radii = { 15, 32, 108, 74, 9 };
		const double pi = 3.14159;

		//foreach statement block
		foreach (int radius in radii)
		{
			double circumference = pi * (2 * radius);

			//expression statement (method invocation)
```

```
                System.Console.WriteLine("Radius of circle #{0} is {1}.
                      Circumference = {2:N2}", counter, radius,
                      circumference);

                //expression statement (postfix increment)
                counter++;

            } //end of foreach statement block
        } // end of Main method body
} // end of SimpleStatements class
```

Types of Statements

- Declaration statements – Introduce a new variable or constant. You may also assign a value to the declared variable; for constants, assigning a value is mandatory.
- Expression statements – See section below.
- Selection statements – Allow you to branch to different sections of code by testing of certain conditions are met (e.g. `if`, `else`, `switch`, `case`).
- Iteration statements – Allow you to loop through collections and perform actions on each item until a specified condition is met.
- Jump statements – Transfer program control over to a different section of code (e.g. `break`, `continue`, `default`, `goto`, `return`, `yield`).
- Exception-handling statements – These help prevent errors and allow for meaningful recovery from exception conditions at runtime (e.g. `throw`, `try-catch`, `try-finally`, `try-catch-finally`).
- Checked and unchecked – Allow you to specify whether numerical operations should be allowed to cause an overflow when the result is stored in a variable which is too small.
- Await statement – Methods marked with the `async` modifier allow the use of the `await` operator in the method. When control reaches an `await` expression in an `async` method, control is temporarily returned to the caller; progress is suspended until the awaited task is completed, then execution in the method can resume.
- Yield return statement – Iterators perform custom iteration over collections (e.g. lists, arrays). An iterator can use the `yield return` statement to facilitate returning each item in the collection in turn. When a `yield return` statement is reached, the current element is returned and the location in the collection is stored. When the iterator is next called, it will continue from the stored location.
- Fixed statement – Prevents the garbage collector from moving the storage location of a movable variable.

- Lock statement – Enables access to blocks of code to be limited to one thread at a time.
- Labelled statement – A statement can be given a label, after which you can use the `goto` keyword to jump to it.
- The empty statement – This consists of a single semicolon only and does nothing. It can be used in places where providing a statement is mandatory but no action is necessary.

Declaration Statements

The following code demonstrates how to use a declaration statement to declare a variable with or without an initializer, and how to declare a constant with the necessary initializer:

```
//variable declaration statements
double area;
double radius = 2;

//constant declaration statement
const double pi = 3.14159;
```

Expression Statements

The following code demonstrates examples of expression statements, such as assignment, object creation and method invocation:

```
//expression statement (assignment)
area = pi * (radius * radius);

//expression statement (method invocation)
System.Console.WriteLine();

//expression statement (new object creation)
System.Collections.Generic.List<string> strings = new
System.Collections.Generic.List<string>();
```

The empty statement

The following code demonstrates two examples of how the empty statement might be used:

```
void ProcessMessages()
{
      while (ProcessMessage())
```

```
      {
            ; //statement required here
      }
}

void F()
{
      //…
      if (done) goto exit;
//…
exit:
      ; //statement required here
}
```

Embedded Statements

Some statements are always followed by an embedded statement (e.g. `do`, `while`, `for` and `foreach`). An embedded statement may be a single statement or a block of statements enclosed in {} brackets (though single statements may also be enclosed in {} brackets for readability purposes). The following are examples of embedded statements:

```
//Recommended styling – code in {} blocks is easier to read
foreach(string s in
System.IO.Directory.GetDirectories(System.Environment.CurrentDirectory))
{
      System.Console.WriteLine(s);
}

//Not recommended; harder to read without {} brackets
foreach(string s in
System.IO.Directory.GetDirectories(System.Environment.CurrentDirectory))
      System.Console.WriteLine(s);
```

Embedded statements that are declaration statements or labelled statements must be enclosed in {} brackets:

```
if (pointB == true)
      int radius = 5; //this will throw error CS1023
```

Issues like those demonstrated above can be easily fixed by enclosing the embedded statement in {} brackets in a block:

```
if (b == true)
{
      //no error will be thrown
```

```
      System.DateTime d = System.DateTime.Now;
      System.Console.WriteLine(d.ToLongDateString());
}
```

Nested statement blocks

Statement blocks can be nested, for example:

```
foreach (string s in
System.IO.Directory.GetDirectories(System.Environment.CurrentDirectory))
{
      if (S.StartsWith(“CSharp”))
      {
            if (S.EndsWith(“TempFolder”))
            {
                  return s;
            }
      }
}
return “Not found.”;
```

Unreachable statements

If the compiler concludes that flow of control can never reach a given statement, it will trigger a warning with code CS0162. For example:

```
const int val = 5;
if (val < 4)
{
      //this will trigger a CS0162 warning
      System.Console.WriteLine(“This statement is unreachable.”);
}
```

Attributes

Attributes

Attributes provide a powerful method of associating metadata with code. After associating an attribute with a program entity, you can query that attribute at runtime using reflection. This means that at runtime, you can gather information about the behaviour of various elements (e.g. classes, methods, structures, enumerators, assemblies) by querying their attributes.

Declaring Attributes

You can create custom attributes to specify any additional required information as follows:

```
[System.AttributeUsage(System.AttributeTargets.Class |
System.AttributeTargets.Struct)]
public class AuthorAttribute : System.Attribute
{
      private string name;
      public double version;

      public AuthorAttribute(string name)
      {
            this.name = name;
            version = 1.0;
      }
}
```

Once declared, you can use a custom attribute in the following way:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
{
      //Some code
}
```

To allow multiple copies of a single attribute on any given element, you can pass AttributeUsage the AllowMultiple parameter:

```
[System.AttributeUsage(System.AttributeTargets.Class |
      System.AttributeTargets.Struct, AllowMultiple = True)]
public class AuthorAttribute : SystemAttribute
```

You can apply multiple attributes of a given type to a single class as follows:

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
      //some code
}
```

It is standard convention to name all attributes with the suffix "Attribute". However, you do not need to specify the Attribute suffix when using attributes in code (e.g. AuthorAtrribute is used in code with the name "Author").

You can apply one or more attributes to entire assemblies, modules, or single elements like classes and properties. These attributes can accept arguments in the same way as methods and properties. A program using attributes can examine the metadata they contain or the metadata in other programs using reflection.

Properties of Attributes

- They add metadata to your program. Metadata is information about the types defined in a program. All .NET assemblies contain metadata that describes the types and type members defined in that assembly. You can also add custom attributes to specify any additional required information.
- You can apply one or more attributes to entire assemblies, modules, or single elements like classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- A program can examine its own metadata or the metadata in other programs using reflection.

Attribute Parameters

Attributes may have positional or named positional or named parameters. Named parameters are optional and can be specified in any order. For example:

```
[Dll((Import("user32.dll", SetLastError=false, ExactSpelling=false)]
```

is equivalent to writing:

```
[Dll((Import("user32.dll", ExactSpelling=false, SetLastError=false)]
```

Positional parameters must be placed in the correct order to correlate to the order of parameters declared in the attribute constructor. They must also precede named parameters. For instance, in the example written above, the parameter with value "`user32.dll`" is the first parameter listed in that attribute's constructor; hence it must be specified first when using the attribute.

Named/optional parameters correspond to either properties or fields of the attribute. Attributes should have associated documentation information about default parameter values.

Attribute Targets

The target of an attribute is the entity which the attribute applies to (e.g. an attribute may apply to a class, a method, or an entire assembly). By default, attributes apply to the element that directly follows them; however, you can explicitly state what kinds of element an attribute may be applied to using the `target` parameter:

```
[target : attribute-list]
```

The following list comprises all possible kinds of target:

- `assembly` – entire assembly
- `module` – current assembly module
- `field` – a field in a class or struct
- `event` – an event
- `method` – entire method or `get` and `set` property accessors
- `param` – method parameters or property `set` accessor parameters
- `property` – a property
- `return` – return value of a method/property indexer/`get` property accessor
- `type` – struct, class, interface, enum or delegate

The field target value would be specified to attach an attribute to the backing field of an auto-implemented property.

The following code demonstrates how to apply attributes to assemblies and modules:

```
using System;
using System.Reflection;
[assembly : AssemblyTitleAttribute("Production assembly 4")]
[module : CLSCompliant(true)]
```

The next extract demonstrates how you can specify whether to apply an attribute to a method, parameter, or return value:

```
//default – applies to method
[ValidatedContract]
int Method1() { return 0; }

//attribute applies to method
[method : ValidatedContract]
int Method2() { return 0; }

//attribute applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

//apply attribute to method's return value
[return : ValidatedContract]
int Method4() { return 0; }
```

Common Attribute Uses

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol.
- Describing how to marshal method parameters when interoperating with native code.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the `DllImportAttribute` class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Expression-Bodied Members

Expression-Bodied Members

Expression bodies are used to provide a member's implementation in a concise, readable form. Expression body definitions can be used whenever the logic for a supported member (e.g. method or property) consists of a single expression.

The general syntax is as follows:

```
//"expression" is a valid expression
member => expression;
```

Support was introduced in C# 6.0 and expanded in C# 7.0. The following type members can use expression bodies:

- Methods (C# 6.0)
- Read-Only Properties (C# 6.0)
- Properties (C# 7.0)
- Constructors (C# 7.0)
- Finalizers (C# 7.0)
- Indexers (C# 7.0)

Methods

An expression bodied method consists of a single expression that returns a value whose type matches the method's return type (or performs some operation, in the case of void methods).

In the following example, expression bodies are used to override the default `ToString()` method and declare a basic method `DisplayName()` that simply prints to the console:

```
using System;

public class Person
{
	public Person (string firstName, string lastName)
	{
		fName = firstName;
		lName = lastName;
	}

	private string fname;
	private string lname;

	public override ToString() => $"{fName} {lName}".Trim();
	public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
	static void Main()
	{
		Person p = new Person("Bill", "Gates");
		Console.WriteLine(p);
		p.DisplayName();
	}
}
```

Read-Only Properties

You can implement a read-only property in an expression body using the following syntax:

```
PropertyType PropertyName => expression;
```

The following example shows how a read-only property may be implemented as an expression body in context:

```
public class Location
{
	private string locationName;

	public Location(string name);
	{
		locationName = name;
	}

	public string Name => locationName;
}
```

Properties

You can define the `get` and `set` accessors of a property by using separate expression bodies for each:

```
public class Location
{
      private string locationName;

      public Location(string name) => Name = name;

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

Constructors

An expression body for a constructor typically consists of a single assignment expression or method call that handles its arguments or initializes an instance state.

In the following example, the constructor contains the singular string parameter “name” and the expression body assigns its value to the Name property:

```
public class Location
{
      private string locationName;

      public Location(string name) => Name = name;

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

Finalizers

An expression body for a finalizer typically contains cleanup statements (e.g. to release unmanaged resources). In the following example, the expression body is used to indicate that the finalizer was called:

```
using System;
```

```
public class Destroyer
{
      public override string ToString() => GetType.Name();

      ~Destroyer() => Console.WriteLine($”The {ToString()}
                        destructor is executing.”);
}
```

Indexers

As with properties, the indexer `set` and `get` accessors can use expression bodies, providing the `get` accessor comprises a single expression that returns a value and the `set` accessor performs a simple assignment.

The following example defines a class “Sports”, which has a string array listing the names of several sports. The indexer `get` and `set` accessors use expression bodies:

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

public class Sports
{
      private string[] types = { "Baseball", "Basketball", "Football",
                                 "Hockey", "Soccer", "Tennis",
                                 "Volleyball" };

      public string this[int i]
      {
            get => types[i];
            set => types[i] = value;
      }
}
```