C# supports two forms of string literals: ***regular string literals*** and ***verbatim string literals***.

A regular string literal consists of zero or more characters enclosed in double quotes, as in `"hello"`, and may include both simple escape sequences (such as `\t` for the tab character), and hexadecimal and Unicode escape sequences.

A verbatim string literal consists of an `@` character followed by a double-quote character, zero or more characters, and a closing double-quote character. A simple example is `@"hello"`.

```
string c = "hello \t world";                // hello      world
string d = @"hello \t world";               // hello \t world
```

The following pre-processing directives are available:

- `#define` and `#undef`, which are used to define and undefine, respectively, conditional compilation symbols (Declaration directives).
- `#if`, `#elif`, `#else`, and `#endif`, which are used to conditionally skip sections of source code (Conditional compilation directives).
- `#line`, which is used to control line numbers emitted for errors and warnings (Line directives).
- `#error` and `#warning`, which are used to issue errors and warnings, respectively (Diagnostic directives).
- `#region` and `#endregion`, which are used to explicitly mark sections of source code (Region directives).
- `#pragma`, which is used to specify optional contextual information to the compiler (Pragma directives).

Declaration directives

The declaration directives are used to define or undefine conditional compilation symbols.

Any `#define` and `#undef` directives in a source file must occur before the first *token* (Tokens) in the source file; otherwise a compile-time error occurs. In intuitive terms, `#define` and `#undef` directives must precede any "real code" in the source file.

The example:

```
#define Enterprise

#if Professional || Enterprise
    #define Advanced
#endif

namespace Megacorp.Data
{
    #if Advanced
    class PivotTable {...}
    #endif
}
```

is valid because the `#define` directives precede the first token (the `namespace` keyword) in the source file.

Declared accessibility

The ***declared accessibility*** of a member can be one of the following:

- Public, which is selected by including a `public` modifier in the member declaration. The intuitive meaning of `public` is "access not limited".
- Protected, which is selected by including a `protected` modifier in the member declaration. The intuitive meaning of `protected` is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive meaning of `internal` is "access limited to this program".
- Protected internal (meaning protected or internal), which is selected by including both a `protected` and an `internal` modifier in the member declaration. The intuitive meaning of `protected internal` is "access limited to this program or types derived from the containing class".
- Private, which is selected by including a `private` modifier in the member declaration. The intuitive meaning of `private` is "access limited to the containing type".

Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does

not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have `public` declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared in compilation units or namespaces can have `public` or `internal` declared accessibility and default to `internal` declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to `private` declared accessibility. (Note that a type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Struct members can have `public`, `internal`, or `private` declared accessibility and default to `private` declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have `protected` or `protected internal` declared accessibility. (Note that a type declared as a member of a struct can have `public`, `internal`, or `private` declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Interface members implicitly have `public` declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have `public` declared accessibility. No access modifiers are allowed on enumeration member declarations.

Protected access for instance members

When a `protected` instance member is accessed outside the program text of the class in which it is declared, and when a `protected internal` instance member is accessed outside the program text of the program in which it is declared, the access must take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place through an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

Let `B` be a base class that declares a protected instance member `M`, and let `D` be a class that derives from `B`. Within the *class_body* of `D`, access to `M` can take one of the following forms:

- An unqualified *type_name* or *primary_expression* of the form `M`.
- A *primary_expression* of the form `E.M`, provided the type of `E` is `T` or a class derived from `T`, where `T` is the class type `D`, or a class type constructed from `D`
- A *primary_expression* of the form `base.M`.

In addition to these forms of access, a derived class can access a protected instance constructor of a base class in a *constructor_initializer*(Constructor initializers).

In the example

C#Copy

```
public class A
{
    protected int x;

    static void F(A a, B b) {
        a.x = 1;        // Ok
        b.x = 1;        // Ok
    }
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;        // Error, must access through instance of B
        b.x = 1;        // Ok
    }
}
```

Accessibility constraints

Several constructs in the C# language require a type to be ***at least as accessible as*** a member or another type. A type `T` is said to be at least as accessible as a member or type `M` if the accessibility domain of `T` is a superset of the accessibility domain of `M`. In

other words, `T` is at least as accessible as `M` if `T` is accessible in all contexts in which `M` is accessible.

The following accessibility constraints exist:

- The direct base class of a class type must be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
- The type of a constant must be at least as accessible as the constant itself.
- The type of a field must be at least as accessible as the field itself.
- The return type and parameter types of a method must be at least as accessible as the method itself.
- The type of a property must be at least as accessible as the property itself.
- The type of an event must be at least as accessible as the event itself.
- The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- The return type and parameter types of an operator must be at least as accessible as the operator itself.
- The parameter types of an instance constructor must be at least as accessible as the instance constructor itself.

In the example

C#Copy

```
class A {...}

public class B: A {...}
```

the `B` class results in a compile-time error because `A` is not at least as accessible as `B`.

Likewise, in the example

C#Copy

```
class A {...}

public class B
{
    A F() {...}
```

```
    internal A G() {...}

    public A H() {...}
}
```

the `H` method in `B` results in a compile-time error because the return type `A` is not at least as accessible as the method.

Integral types

C# supports nine integral types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`. The integral types have the following sizes and ranges of values:

- The `sbyte` type represents signed 8-bit integers with values between -128 and 127.
- The `byte` type represents unsigned 8-bit integers with values between 0 and 255.
- The `short` type represents signed 16-bit integers with values between -32768 and 32767.
- The `ushort` type represents unsigned 16-bit integers with values between 0 and 65535.
- The `int` type represents signed 32-bit integers with values between -2147483648 and 2147483647.
- The `uint` type represents unsigned 32-bit integers with values between 0 and 4294967295.
- The `long` type represents signed 64-bit integers with values between -9223372036854775808 and 9223372036854775807.
- The `ulong` type represents unsigned 64-bit integers with values between 0 and 18446744073709551615.
- The `char` type represents unsigned 16-bit integers with values between 0 and 65535. The set of possible values for the `char` type corresponds to the Unicode character set. Although `char` has the same representation as `ushort`, not all operations permitted on one type are permitted on the other.

The decimal type

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations. The `decimal` type can represent values ranging from `1.0 * 10^-28` to approximately `7.9 * 10^28` with 28-29 significant digits.

Class types

Certain predefined class types have special meaning in the C# language, as described in the table below.

Class type	Description
`System.Object`	The ultimate base class of all other types. See The object type.
`System.String`	The string type of the C# language. See The string type.
`System.ValueType`	The base class of all value types. See The System.ValueType typ
`System.Enum`	The base class of all enum types. See Enums.
`System.Array`	The base class of all array types. See Arrays.
`System.Delegate`	The base class of all delegate types. See Delegates.
`System.Exception`	The base class of all exception types. See Exceptions.

Delegate types

A delegate is a data structure that refers to one or more methods. For instance methods, it also refers to their corresponding object instances.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static

and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance on which to invoke the method.

Static variables

A field declared with the `static` modifier is called a ***static variable***. A static variable comes into existence before execution of the static constructor (Static constructors) for its containing type, and ceases to exist when the associated application domain ceases to exist.

The initial value of a static variable is the default value (Default values) of the variable's type.

Reference parameters

A parameter declared with a `ref` modifier is a ***reference parameter***.

A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member or anonymous function invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

Within an instance method or instance accessor of a struct type, the `this` keyword behaves exactly as a reference parameter of the struct type (This access).

Implicit numeric conversions

The implicit numeric conversions are:

- From `sbyte` to `short`, `int`, `long`, `float`, `double`, or `decimal`.
- From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `short` to `int`, `long`, `float`, `double`, or `decimal`.
- From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.

- From `int` to `long`, `float`, `double`, or `decimal`.
- From `uint` to `long`, `ulong`, `float`, `double`, or `decimal`.
- From `long` to `float`, `double`, or `decimal`.
- From `ulong` to `float`, `double`, or `decimal`.
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `float` to `double`.

Conversions from `int`, `uint`, `long`, or `ulong` to `float` and from `long` or `ulong` to `double` may cause a loss of precision, but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.

Operator overloading

The ***overloadable unary operators*** are:

C#Copy

```
+   -   !   ~   ++   --   true   false
```

Although `true` and `false` are not used explicitly in expressions (and therefore are not included in the precedence table in Operator precedence and associativity), they are considered operators because they are invoked in several expression contexts: boolean expressions (Boolean expressions) and expressions involving the conditional (Conditional operator), and conditional logical operators (Conditional logical operators).

The ***overloadable binary operators*** are:

C#Copy

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the `=`, `&&`, `||`, `??`, `?:`, `=>`, `checked`, `unchecked`, `new`, `typeof`, `default`, `as`, and `is` operators.

When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded. For example, an overload of operator `*` is also an overload of operator `*=`. This is described further in Compound assignment. Note that the

assignment operator itself (`=`) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.+

Cast operations, such as `(T)x`, are overloaded by providing user-defined conversions (User-defined conversions).

Implicitly typed local variable declaration

In the context of a local variable declaration, the identifier var acts as a contextual keyword (Keywords).When the *local_variable_type* is specified as `var` and no type named `var` is in scope, the declaration is an ***implicitly typed local variable declaration***, whose type is inferred from the type of the associated initializer expression. Implicitly typed local variable declarations are subject to the following restrictions:

- The *local_variable_declaration* cannot include multiple *local_variable_declarator*s.
- The *local_variable_declarator* must include a *local_variable_initializer*.
- The *local_variable_initializer* must be an *expression*.
- The initializer *expression* must have a compile-time type.
- The initializer *expression* cannot refer to the declared variable itself

The following are examples of incorrect implicitly typed local variable declarations:

C#Copy

```
var x;               // Error, no initializer to infer type from
var y = {1, 2, 3};   // Error, array initializer not permitted
var z = null;        // Error, null does not have a type
var u = x => x + 1;  // Error, anonymous functions do not have a type
var v = v++;         // Error, initializer cannot refer to variable itself
```

Switches

C#Copy

```
switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
```

```
    return;
}
```

The governing type of a `switch` statement may be the type `string`. For example:

C#Copy

```
void DoCommand(string command) {
    switch (command.ToLower()) {
    case "run":
        DoRun();
        break;
    case "save":
        DoSave();
        break;
    case "quit":
        DoQuit();
        break;
    default:
        InvalidCommand(command);
        break;
    }
}
```

Like the string equality operators (String equality operators), the `switch` statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a `case` label constant.

Extern aliases

An *extern_alias_directive* introduces an identifier that serves as an alias for a namespace. The specification of the aliased namespace is external to the source code of the program and applies also to nested namespaces of the aliased namespace.

An *extern_alias_directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.

The following program declares and uses two extern aliases, `X` and `Y`, each of which represent the root of a distinct namespace hierarchy:

C#Copy

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

The program declares the existence of the extern aliases `X` and `Y`, but the actual definitions of the aliases are external to the program. The identically named `N.B` classes can now be referenced as `X.N.B` and `Y.N.B`, or, using the namespace alias qualifier, `X::N.B` and `Y::N.B`. An error occurs if a program declares an extern alias for which no external definition is provided.

Using alias directives

Within member declarations in a compilation unit or namespace body that contains a *using_alias_directive*, the identifier introduced by the *using_alias_directive* can be used to reference the given namespace or type. For example:

C#Copy

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;

    class B: A {}
}
```

Above, within member declarations in the `N3` namespace, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:

C#Copy

```
namespace N3
{
    using R = N1.N2;

    class B: R.A {}
}
```

Abstract classes

The `abstract` modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An abstract class differs from a non-abstract class in the following ways:

- An abstract class cannot be instantiated directly, and it is a compile-time error to use the `new` operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be `null` or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract members.
- An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract members, thereby overriding those abstract members. In the example

C#Copy

```
abstract class A
{
    public abstract void F();
}

abstract class B: A
{
    public void G() {}
```

```
}

class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

Sealed classes

The `sealed` modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

Static classes

The `static` modifier is used to mark the class being declared as a ***static class***. A static class cannot be instantiated, cannot be used as a type and can contain only static members. Only a static class can contain declarations of extension methods (Extension methods).

Generic Base Classes

For a constructed class type, if a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type_parameter* in the base class declaration, the corresponding *type_argument* of the constructed type. Given the generic class declarations

C#Copy

```
class B<U,V> {...}

class G<T>: B<string,T[]> {...}
```

the base class of the constructed type `G<int>` would be `B<string,int[]>`.

The direct base class of a class type must be at least as accessible as the class type itself (Accessibility domains). For example, it is a compile-time error for a `public` class to derive from a `private` or `internal` class.

Partial methods

Partial methods are useful for allowing one part of a type declaration to customize the behavior of another part, e.g., one that is generated by a tool. Consider the following partial class declaration:

C#Copy

```
partial class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    partial void OnNameChanging(string newName);

    partial void OnNameChanged();
}
```

If this class is compiled without any other parts, the defining partial method declarations and their invocations will be removed, and the resulting combined class declaration will be equivalent to the following:

C#Copy

```
class Customer
{
    string name;
```

```
    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Assume that another part is given, however, which provides implementing declarations of the partial methods:

C#Copy

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Then the resulting combined class declaration will be equivalent to the following:

C#Copy

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    void OnNameChanging(string newName)
    {
```

```
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Static and instance members

The following example illustrates the rules for accessing static and instance members:

C#Copy

```
class Test
{
    int x;
    static int y;

    void F() {
        x = 1;            // Ok, same as this.x = 1
        y = 1;            // Ok, same as Test.y = 1
    }

    static void G() {
        x = 1;            // Error, cannot access this.x
        y = 1;            // Ok, same as Test.y = 1
    }

    static void Main() {
        Test t = new Test();
        t.x = 1;          // Ok
        t.y = 1;          // Error, cannot access static member through instance
        Test.x = 1;       // Error, cannot access instance member through type
        Test.y = 1;       // Ok
    }
}
```

Nested types

A type declared within a class or struct declaration is called a ***nested type***. A type that is declared within a compilation unit or namespace is called a ***non-nested type***.

In the example
C#Copy

```
using System;

class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

class `B` is a nested type because it is declared within class `A`, and class `A` is a non-nested type because it is declared within a compilation unit.

Fully qualified name

The fully qualified name (Fully qualified names) for a nested type is `S.N` where `S` is the fully qualified name of the type in which type `N` is declared.

this access

A nested type and its containing type do not have a special relationship with regard to *this_access* (This access). Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the `this` for the instance of the containing type as a constructor argument for the nested type. The following example
C#Copy

```
using System;

class C
{
    int i = 123;

    public void F() {
        Nested n = new Nested(this);
        n.G();
    }

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}
```

shows this technique. An instance of `C` creates an instance of `Nested` and passes its own `this` to `Nested`'s constructor in order to provide subsequent access to `C`'s instance members.

Readonly fields

When a *field_declaration* includes a `readonly` modifier, the fields introduced by the declaration are ***readonly fields***. Direct assignments to readonly fields can only occur as part of that declaration or in an instance constructor or static constructor in the same class. (A readonly field can be assigned to multiple times in these contexts.) Specifically, direct assignments to a `readonly` field are permitted only in the following contexts:

- In the *variable_declarator* that introduces the field (by including a *variable_initializer* in the declaration).
- For an instance field, in the instance constructors of the class that contains the field declaration; for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a `readonly` field as an `out` or `ref` parameter.

Attempting to assign to a `readonly` field or pass it as an `out` or `ref` parameter in any other context is a compile-time error.

Using static readonly fields for constants

A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile-time. In the example

C#Copy

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

the `Black`, `White`, `Red`, `Green`, and `Blue` members cannot be declared as `const` members because their values cannot be computed at compile-time. However, declaring them `static readonly` instead has much the same effect.

Field initialization

The initial value of a field, whether it be a static field or an instance field, is the default value (Default values) of the field's type. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never "uninitialized". The example

C#Copy

```
using System;

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

produces the output

Copy

```
b = False, i = 0
```

because `b` and `i` are both automatically initialized to default values.

Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class.

A *method_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

- The declaration includes a valid combination of access modifiers (Access modifiers).
- The declaration does not include the same modifier multiple times.
- The declaration includes at most one of the following modifiers: `static`, `virtual`, and `override`.
- The declaration includes at most one of the following modifiers: `new` and `override`.
- If the declaration includes the `abstract` modifier, then the declaration does not include any of the following modifiers: `static`, `virtual`, `sealed` or `extern`.
- If the declaration includes the `private` modifier, then the declaration does not include any of the following modifiers: `virtual`, `override`, or `abstract`.
- If the declaration includes the `sealed` modifier, then the declaration also includes the `override` modifier.
- If the declaration includes the `partial` modifier, then it does not include any of the following modifiers: `new`, `public`, `protected`, `internal`, `private`, `virtual`, `sealed`, `override`, `abstract`, or `extern`.

A method that has the `async` modifier is an async function and follows the rules described in Async functions.

Parameters

A *fixed_parameter* with a *default_argument* is known as an ***optional parameter***, whereas a *fixed_parameter* without a *default_argument* is a ***required parameter***. A required parameter may not appear after an optional parameter in a *formal_parameter_list*.

A `ref` or `out` parameter cannot have a *default_argument*. The *expression* in a *default_argument* must be one of the following:

- a *constant_expression*
- an expression of the form `new S()` where `S` is a value type
- an expression of the form `default(S)` where `S` is a value type

The *expression* must be implicitly convertible by an identity or nullable conversion to the type of the parameter.

Reference parameters

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

The example

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }
```

```
    static void Main() {
        int i = 1, j = 2;
```

```
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

produces the output

```
i = 2, j = 1
```

In a method that takes reference parameters it is possible for multiple names to represent the same storage location. In the example

C#Copy

```
class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}
```

the invocation of `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`.

Output parameters

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location.

Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a *variable_reference* (Precise rules for determining definite assignment) of the same type as the formal parameter

Output parameters are typically used in methods that produce multiple return values. For example:

C#Copy

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

The example produces the output:

Copy

```
c:\Windows\System\
hello.txt
```

Note that the `dir` and `name` variables can be unassigned before they are passed to `SplitPath`, and that they are considered definitely assigned following the call.

Parameter arrays

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter (Value parameters) of the same type.

The example

C#Copy

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
    }

    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

produces the output

Copy

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
```

```
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F` automatically creates a four-element `int[]` with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `F` creates a zero-element `int[]` and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

C#Copy

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

Static and instance methods

When a method declaration includes a `static` modifier, that method is said to be a static method. When no `static` modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as `this` (This access).

Virtual methods

When an instance method declaration includes a `virtual` modifier, that method is said to be a virtual method. When no `virtual` modifier is present, the method is said to be a non-virtual method.

The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as ***overriding*** that method (Override methods).

In a virtual method invocation, the ***run-time type*** of the instance for which that invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the ***compile-time type*** of the instance is the determining factor. In precise terms, when a method named `N` is invoked with an argument list `A` on an instance with a compile-time type `C` and a run-time type `R` (where `R` is either `C` or a class derived from `C`), the invocation is processed as follows:

- First, overload resolution is applied to `C`, `N`, and `A`, to select a specific method `M` from the set of methods declared in and inherited by `C`. This is described in Method invocations.
- Then, if `M` is a non-virtual method, `M` is invoked.
- Otherwise, `M` is a virtual method, and the most derived implementation of `M` with respect to `R` is invoked.

The following example illustrates the differences between virtual and non-virtual methods:

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }

    public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A
{
    new public void F() { Console.WriteLine("B.F"); }

    public override void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
```

```
        b.G();
    }
}
```

In the example, `A` introduces a non-virtual method `F` and a virtual method `G`. The class `B` introduces a new non-virtual method `F`, thus hiding the inherited `F`, and also overrides the inherited method `G`. The example produces the output:

Copy

```
A.F
B.F
B.G
B.G
```

Notice that the statement `a.G()` invokes `B.G`, not `A.G`. This is because the run-time type of the instance (which is `B`), not the compile-time type of the instance (which is `A`), determines the actual method implementation to invoke.

Override methods

When an instance method declaration includes an `override` modifier, the method is said to be an ***override method***. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of that method.

The following example demonstrates how the overriding rules work for generic classes:

C#Copy

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
    public virtual void H(C<T> x) {...}
}

class D: C<string>
```

```
{
    public override string F() {...}            // Ok
    public override C<string> G() {...}         // Ok
    public override void H(C<T> x) {...}        // Error, should be C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}                 // Ok
    public override C<U> G() {...}              // Ok
    public override void H(C<T> x) {...}        // Error, should be C<U>
}
```

Sealed methods

When an instance method declaration includes a `sealed` modifier, that method is said to be a ***sealed method***. If an instance method declaration includes the `sealed` modifier, it must also include the `override` modifier. Use of the `sealed` modifier prevents a derived class from further overriding the method.

In the example

C#Copy

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }

    public virtual void G() {
        Console.WriteLine("A.G");
    }
}

class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
```

```
    }

    override public void G() {
        Console.WriteLine("B.G");
    }
}

class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

the class `B` provides two override methods: an `F` method that has the `sealed` modifier and a `G` method that does not. `B`'s use of the sealed `modifier` prevents `C` from further overriding `F`.

Abstract methods

When an instance method declaration includes an `abstract` modifier, that method is said to be an ***abstract method***. Although an abstract method is implicitly also a virtual method, it cannot have the modifier `virtual`.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method. Instead, non-abstract derived classes are required to provide their own implementation by overriding that method. Because an abstract method provides no actual implementation, the *method_body* of an abstract method simply consists of a semicolon.

Abstract method declarations are only permitted in abstract classes (Abstract classes).

In the example

C#Copy

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}
```

```
public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

the `Shape` class defines the abstract notion of a geometrical shape object that can paint itself. The `Paint` method is abstract because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are non-abstract, they are required to override the `Paint` method and provide an actual implementation.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. In the example

C#Copy

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
```

```
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

class `A` declares a virtual method, class `B` overrides this method with an abstract method, and class `C` overrides the abstract method to provide its own implementation.

External methods

When a method declaration includes an `extern` modifier, that method is said to be an ***external method***. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method_body* of an external method simply consists of a semicolon. An external method may not be generic.

The `extern` modifier is typically used in conjunction with a `DllImport` attribute (Interoperation with COM and Win32 components), allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

When an external method includes a `DllImport` attribute, the method declaration must also include a `static` modifier. This example demonstrates the use of the `extern` modifier and the `DllImport` attribute:

C#Copy

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);
```

```
    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

Extension methods

When the first parameter of a method includes the `this` modifier, that method is said to be an ***extension method***. Extension methods can only be declared in non-generic, non-nested static classes. The first parameter of an extension method can have no modifiers other than `this`, and the parameter type cannot be a pointer type.

The following is an example of a static class that declares two extension methods:

C#Copy

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method can be invoked using instance method invocation syntax (Extension method invocations), using the receiver expression as the first argument.

The following program uses the extension methods declared above:

C#Copy

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

The `Slice` method is available on the `string[]`, and the `ToInt32` method is available on `string`, because they have been declared as extension methods. The meaning of the program is the same as the following, using ordinary static method calls:

C#Copy

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

The four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

Accessors

Based on the presence or absence of the `get` and `set` accessors, a property is classified as follows:

- A property that includes both a `get` accessor and a `set` accessor is said to be a ***read-write*** property.
- A property that has only a `get` accessor is said to be a ***read-only*** property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a `set` accessor is said to be a ***write-only*** property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression.

In the example

C#Copy

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}
```

the `Button` control declares a public `Caption` property. The `get` accessor of the `Caption` property returns the string stored in the private `caption` field. The `set` accessor checks if the new value is different from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The `get` accessor simply returns a value stored in a private field, and the `set` accessor modifies that private field and then performs any additional actions required to fully update the state of the object.

Given the `Button` class above, the following is an example of use of the `Caption` property:

```
Button okButton = new Button();
okButton.Caption = "OK";            // Invokes set accessor
string s = okButton.Caption;        // Invokes get accessor
```

Here, the `set` accessor is invoked by assigning a value to the property, and the `get` accessor is invoked by referencing the property in an expression.

Events

An ***event*** is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying ***event handlers***.

The following example shows how event handlers are attached to instances of the `Button` class:

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
```

```
    }

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to the `Click` events.

Destructors

A ***destructor*** is a member that implements the actions required to destruct an instance of a class.

The output of the example

C#Copy

```
using System;

class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}

class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}

class Test
```

```
{
   static void Main() {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
   }
}
```

is

```
B's destructor
A's destructor
```

since destructors in an inheritance chain are called in order, from most derived to least derived.

Class and struct differences

Structs differ from classes in several important ways:

- Structs are value types (Value semantics).
- All struct types implicitly inherit from the class `System.ValueType` (Inheritance).
- Assignment to a variable of a struct type creates a copy of the value being assigned (Assignment).
- The default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null` (Default values).
- Boxing and unboxing operations are used to convert between a struct type and `object` (Boxing and unboxing).
- The meaning of `this` is different for structs (This access).
- Instance field declarations for a struct are not permitted to include variable initializers (Field initializers).
- A struct is not permitted to declare a parameterless instance constructor (Constructors).
- A struct is not permitted to declare a destructor (Destructors).

Value semantics

Structs are value types (Value types) and are said to have value semantics. Classes, on the other hand, are reference types (Reference types) and are said to have reference semantics.

A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object. When a struct `B` contains an instance field of type `A` and `A` is a struct type, it is a compile-time error for `A` to depend on `B` or a type constructed from `B`.

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

is an error because `Node` contains an instance field of its own type.

Given the declaration

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

the code fragment

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

outputs the value `10`. The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the assignment to `a.x`. **Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.**

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a `ref` or `out` parameter.

Structs initialize to the default values of their data members. Referring to the `Point` struct declared above, the example

```
Point[] a = new Point[100];
```

initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero.

Structs should be designed to consider the default initialization state a valid state. In the example

```
using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
```

```
    }
}
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the struct must be prepared to handle this state.

Field initializers

As described in Default values, the default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to `null`. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

The example

C#Copy

```
struct Point
{
    public int x = 1;  // Error, initializer not permitted
    public int y = 1;  // Error, initializer not permitted
}
```

is in error because the instance field declarations include variable initializers.

Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null (Default constructors). A struct can declare instance constructors having parameters. For example

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Given the above declaration, the statements

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

both create a `Point` with `x` and `y` initialized to zero.

Array

An array initializer consists of a sequence of variable initializers, enclosed by "`{`" and "`}`" tokens and separated by "`,`" tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

C#Copy

```
int[] a = {0, 2, 4, 6, 8};
```

it is simply shorthand for an equivalent array creation expression:

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example:

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
int[,] b = new int[5, 2];
```

and then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

Interface

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or structs that implement the interface.

The ***base interfaces*** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. In the example

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

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`.

In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interfaces members cannot be declared with the modifiers `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

The example

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
```

```
}
```

declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to hide the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in Hiding through inheritance.

In the example

```
interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);               // Invokes IInteger.Add
        n.Add(1.0);             // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);   // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);    // Only IDouble.Add is a candidate
    }
}
```

the invocation `n.Add(1)` selects `IInteger.Add` by applying the overload resolution rules of Overload resolution. Similarly the invocation `n.Add(1.0)` selects `IDouble.Add`.

Interface implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct directly implements an interface, the interface identifier is included in the base class list of the class or struct. For example:

C#Copy

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

A class or struct that directly implements an interface also directly implements all of the interface's base interfaces implicitly. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list. For example:

C#Copy

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

interface ITextBox: IControl
{
    void SetText(string text);
}
```

```
class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

Here, class `TextBox` implements both `IControl` and `ITextBox`.

Enums

An ***enum type*** is a distinct value type (Value types) that declares a set of named constants.

The example
C#Copy

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

declares an enum type named `Color` with members `Red`, `Green`, and `Blue`.

Each enum type has a corresponding integral type called the ***underlying type*** of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. Note that `char` cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

The example
C#Copy

```
enum Color: long
{
```

```
    Red,
    Green,
    Blue
}
```

declares an enum with an underlying type of `long`. A developer might choose to use an underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not in the range of `int`, or to preserve this option for the future.

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. The example

C#Copy

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

results in a compile-time error because the constant values `-1`, `-2`, and `-3` are not in the range of the underlying integral type `uint`.

Multiple enum members may share the same associated value. The example

C#Copy

```
enum Color
{
    Red,
    Green,
    Blue,

    Max = Blue
}
```

shows an enum in which two enum members -- `Blue` and `Max` -- have the same associated value.

The example

C#Copy

```
using System;

enum Color
{
    Red,
    Green = 10,
    Blue
}

class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }

    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);

            case Color.Green:
                return String.Format("Green = {0}", (int) c);

            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);

            default:
                return "Invalid color";
        }
    }
}
```

prints out the enum member names and their associated values. The output is:

Copy

```
Red = 0
Green = 10
Blue = 11
```

Delegates

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}
    static bool G(string s) {...}
}
```

The method `X.F` is compatible with the delegate type `Predicate<int>` and the method `X.G` is compatible with the delegate type `Predicate<string>` .

```
delegate int D1(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}

class B
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

The methods `A.M1` and `B.M1` are compatible with both the delegate types `D1` and `D2` , since they have the same return type and parameter list; however, these delegate types are two different types, so they are not interchangeable. The methods `B.M2` , `B.M3` , and `B.M4` are incompatible with the delegate types `D1` and `D2` , since they have different return types or parameter lists.

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);        // static method
        C t = new C();
        D cd2 = new D(t.M2);        // instance method
        D cd3 = new D(cd2);        // another delegate
    }
}
```

Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.

The following example shows how to instantiate, combine, remove, and invoke delegates:

C#Copy

```
using System;

delegate void D(int x);

class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }
```

```
    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);                // call M1

        D cd2 = new D(C.M2);
        cd2(-2);                // call M2

        D cd3 = cd1 + cd2;
        cd3(10);                // call M1 then M2

        cd3 += cd1;
        cd3(20);                // call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);                // call M1, M2, M1, then M3

        cd3 -= cd1;             // remove last M1
        cd3(40);                // call M1, M2, then M3

        cd3 -= cd4;
        cd3(50);                // call M1 then M2

        cd3 -= cd2;
        cd3(60);                // call M1

        cd3 -= cd2;             // impossible removal is benign
        cd3(60);                // call M1

        cd3 -= cd1;             // invocation list is empty so cd3 is null

        cd3(70);                // System.NullReferenceException thrown

        cd3 -= cd1;             // impossible removal is benign
    }
```

```
}
```

As shown in the statement `cd3 += cd1;`, a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. In an invocation list such as this, when that delegate is removed, the last occurrence in the invocation list is the one actually removed.

Immediately prior to the execution of the final statement, `cd3 -= cd1;`, the delegate `cd3` refers to an empty invocation list. Attempting to remove a delegate from an empty list (or to remove a non-existent delegate from a non-empty list) is not an error.

The output produced is:

Copy

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

Exceptions

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application level error conditions. The exception mechanism in C# is quite similar to that of C++, with a few important differences:

- In C#, all exceptions must be represented by an instance of a class type derived from `System.Exception`. In C++, any value of any type can be used to represent an exception.

How exceptions are handled

Exceptions are handled by a `try` statement (The try statement).

When an exception occurs, the system searches for the nearest `catch` clause that can handle the exception, as determined by the run-time type of the exception. First, the current method is searched for a lexically enclosing `try` statement, and the associated catch clauses of the try statement are considered in order. If that fails, the method that called the current method is searched for a lexically enclosing `try` statement that encloses the point of the call to the current method. This search continues until a `catch` clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the run-time type of the exception being thrown. A `catch` clause that doesn't name an exception class can handle any exception.

Once a matching catch clause is found, the system prepares to transfer control to the first statement of the catch clause. Before execution of the catch clause begins, the system first executes, in order, any `finally` clauses that were associated with try statements more nested that than the one that caught the exception.

If no matching catch clause is found, one of two things occurs:

- If the search for a matching catch clause reaches a static constructor (Static constructors) or static field initializer, then a `System.TypeInitializationException` is thrown at the point that triggered the invocation of the static constructor. The inner exception of the `System.TypeInitializationException` contains the exception that was originally thrown.
- If the search for matching catch clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

Exceptions that occur during destructor execution are worth special mention. If an exception occurs during destructor execution, and that exception is not caught, then the execution of that destructor is terminated and the destructor of the base class (if any) is called. If there is no base class (as in the case of the `object` type) or if there is no base class destructor, then the exception is discarded.

Attributes

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the *method_modifiers* `public`, `protected`, `internal`, and `private`.+

C# enables programmers to invent new kinds of declarative information, called ***attributes***. Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation.

Attributes are defined through the declaration of attribute classes (Attribute classes), which may have positional and named parameters (Positional and named parameters). Attributes are attached to entities in a C# program using attribute specifications (Attribute specification), and can be retrieved at run-time as attribute instances (Attribute instances).

Attribute classes

A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an ***attribute class***. The declaration of an attribute class defines a new kind of ***attribute*** that can be placed on a declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.

Attribute usage

The attribute `AttributeUsage` (The AttributeUsage attribute) is used to describe how an attribute class can be used.

`AttributeUsage` has a positional parameter (Positional and named parameters) that enables an attribute class to specify the kinds of declarations on which it can be used. The example

C#Copy

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

defines an attribute class named `SimpleAttribute` that can be placed on *class_declaration*s and *interface_declaration*s only. The example

C#Copy

```
[Simple] class Class1 {...}

[Simple] interface Interface1 {...}
```

shows several uses of the `Simple` attribute. Although this attribute is defined with the name `SimpleAttribute`, when this attribute is used, the `Attribute` suffix may be omitted, resulting in the short name `Simple`. Thus, the example above is semantically equivalent to the following:

C#Copy

```
[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` has a named parameter (Positional and named parameters) called `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is true, then that attribute

class is a ***multi-use attribute class***, and can be specified more than once on an entity. If `AllowMultiple` for an attribute class is false or it is unspecified, then that attribute class is a ***single-use attribute class***, and can be specified at most once on an entity.

The example

C#Copy

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;

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

    public string Name {
        get { return name; }
    }
}
```

defines a multi-use attribute class named `AuthorAttribute`. The example

C#Copy

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

shows a class declaration with two uses of the `Author` attribute.

`AttributeUsage` has another named parameter called `Inherited`, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If `Inherited` for an attribute class is true, then that attribute is inherited. If `Inherited` for an attribute class is false then that attribute is not inherited. If it is unspecified, its default value is true.

An attribute class `X` not having an `AttributeUsage` attribute attached to it, as in

C#Copy

```
using System;

class X: Attribute {...}
```

is equivalent to the following:

C#Copy

```
using System;

[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

Positional and named parameters

Attribute classes can have ***positional parameters*** and ***named parameters***. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class.

The example

C#Copy

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {        // Positional parameter
        ...
    }

    public string Topic {                     // Named parameter
        get {...}
```

```
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter, `url`, and one named parameter, `Topic`. Although it is non-static and public, the property `Url` does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

C#Copy

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the ***attribute parameter types***, which are:

- One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.
- The type `object`.
- The type `System.Type`.

- An enum type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility (Attribute specification).
- Single-dimensional arrays of the above types.
- A constructor argument or public field which does not have one of these types, cannot be used as a positional or named parameter in an attribute specification.

It is a compile-time error to use a single-use attribute class more than once on the same entity. The example

C#Copy

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`.

Conditional methods

A method decorated with the `Conditional` attribute is a conditional method. The `Conditional` attribute indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either included or omitted depending on

whether this symbol is defined at the point of the call. If the symbol is defined, the call is included; otherwise, the call (including evaluation of the receiver and parameters of the call) is omitted.

```
#define DEBUG

using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`.

Conditional attribute classes

An attribute class (Attribute classes) decorated with one or more `Conditional` attributes is a ***conditional attribute class***. A conditional attribute class is thus associated with the conditional compilation symbols declared in its `Conditional` attributes. This example:

C#Copy

```
using System;
using System.Diagnostics;
```

```
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

declares `TestAttribute` as a conditional attribute class associated with the conditional compilations symbols `ALPHA` and `BETA`.

The Obsolete attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main() {
        A a = new A();          // Warning
        a.F();
    }
}
```

the class `A` is decorated with the `Obsolete` attribute. Each use of `A` in `Main` results in a warning that includes the specified message, "This class is obsolete; use class B instead."

Unsafe code

In the example

C#Copy

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

the `unsafe` modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus, it is possible to declare the `Left` and `Right` fields to be of a pointer type. The example above could also be written

C#Copy

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts.

Some examples of pointer types are given in the table below:

Example	Description
`byte*`	Pointer to `byte`
`char*`	Pointer to `char`
`int**`	Pointer to pointer to `int`
`int*[]`	Single-dimensional array of pointers to `int`

Example	Description
`void*`	Pointer to unknown type

Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the `*` is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example

C#Copy

```
int* pi, pj;    // NOT as int *pi, *pj;
```

The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

In the example

C#Copy

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
```

```
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

the `->` operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written:

C#Copy

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

In the example

C#Copy

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

a pointer element access is used to initialize the character buffer in a `for` loop. Because the operation `P[E]` is precisely equivalent to `*(P + E)`, the example could equally well have been written:

C#Copy

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

The pointer element access operator does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is undefined. This is the same as C and C++.

The address-of operator

In the example

C#Copy

```
using System;

class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

`i` is considered definitely assigned following the `&i` operation used to initialize `p`.

Pointer arithmetic

Given two expressions, `P` and `Q`, of a pointer type `T*`, the expression `P - Q` computes the difference between the addresses given by `P` and `Q` and then divides that difference by `sizeof(T)`. The type of the result is always `long`. In effect, `P - Q` is computed as `((long)(P) - (long)(Q)) / sizeof(T)`.

For example:

C#Copy

```
using System;

class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

which produces the output:

Copy

```
p - q = -14
q - p = 14
```

Pointer comparison

In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `=>` operators (Relational and type-testing operators) can be applied to values of all pointer types.

The sizeof operator

Expression	Result
`sizeof(sbyte)`	`1`
`sizeof(byte)`	`1`
`sizeof(short)`	`2`
`sizeof(ushort)`	`2`
`sizeof(int)`	`4`
`sizeof(uint)`	`4`
`sizeof(long)`	`8`
`sizeof(ulong)`	`8`
`sizeof(char)`	`2`
`sizeof(float)`	`4`
`sizeof(double)`	`8`
`sizeof(bool)`	`1`

The fixed statement

In an unsafe context, the *embedded_statement* (Statements) production permits an additional construct, the `fixed` statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

The example

C#Copy

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case it would have been an error to use the regular `&` operator since the variables are all classified as moveable variables.

The fourth `fixed` statement in the example above produces a similar result to the third.

This example of the `fixed` statement uses `string`:

C#Copy

```
class Test
{
    static string name = "xx";

    unsafe static void F(char* p) {
```

```
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```

Stack allocation

In an unsafe context, a local variable declaration (Local variable declarations) may include a stack allocation initializer which allocates memory from the call stack.

The content of the newly allocated memory is undefined.

In the example

C#Copy

```
using System;

class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }
```

```
    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns.

Dynamic memory allocation

Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system. For example, the `Memory` class below illustrates how the heap functions of an underlying operating system might be accessed from C#:

C#Copy

```
using System;
using System.Runtime.InteropServices;

public unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    static int ph = GetProcessHeap();

    // Private instance constructor to prevent instantiation.
    private Memory() {}

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size) {
        void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
```

```
    public static void Copy(void* src, void* dst, int count) {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd) {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd) {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
        }
    }

    // Frees a memory block.
    public static void Free(void* block) {
        if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
    }

    // Re-allocates a memory block. If the reallocation request is for a
    // larger size, the additional region of memory is automatically
    // initialized to zero.
    public static void* ReAlloc(void* block, int size) {
        void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Returns the size of a memory block.
    public static int SizeOf(void* block) {
        int result = HeapSize(ph, 0, block);
        if (result == -1) throw new InvalidOperationException();
        return result;
    }

    // Heap API flags
    const int HEAP_ZERO_MEMORY = 0x00000008;

    // Heap API functions
    [DllImport("kernel32")]
    static extern int GetProcessHeap();

    [DllImport("kernel32")]
    static extern void* HeapAlloc(int hHeap, int flags, int size);

    [DllImport("kernel32")]
    static extern bool HeapFree(int hHeap, int flags, void* block);
```

```
    [DllImport("kernel32")]
    static extern void* HeapReAlloc(int hHeap, int flags, void* block, int size);

    [DllImport("kernel32")]
    static extern int HeapSize(int hHeap, int flags, void* block);
}
```

An example that uses the `Memory` class is given below:

C#Copy

```
class Test
{
    static void Main() {
        unsafe {
            byte* buffer = (byte*)Memory.Alloc(256);
            try {
                for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
                byte[] array = new byte[256];
                fixed (byte* p = array) Memory.Copy(buffer, p, 256);
            }
            finally {
                Memory.Free(buffer);
            }
            for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
        }
    }
}
```

The example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with values increasing from 0 to 255. It then allocates a 256 element byte array and uses `Memory.Copy` to copy the contents of the memory block into the byte array. Finally, the memory block is freed using `Memory.Free` and the contents of the byte array are output on the console.