3. Basic Concepts

3.1 Application Startup

An assembly that has an ***entry point*** is called an ***application***. When an application runs, a new ***application domain*** is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain.

An application domain enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects are not directly shared between application domains. For instance, each application domain has its own copy of static variables for these types, and a static constructor for a type runs at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for creating and destroying application domains.

Application startup occurs when the execution environment calls a designated method, which is the application's entry point. This entry point method is always named `Main`, and it can have one of the following signatures.

```
static void Main() {...}
static void Main(string[] args) {...}
static int Main() {...}
static int Main(string[] args) {...}
```

As shown, the entry point may optionally return an `int` value. This return value is used in application termination (§3.2).

The entry point may optionally have one formal parameter. The parameter may have any name, but the type of the parameter must be `string[]`. If the formal parameter is present, the execution environment creates and passes a `string[]` argument containing the command-line arguments specified when the application started. The `string[]` argument is never `null`, but it may have a length of zero if no command-line arguments were specified.

Because C# supports method overloading, a class or struct can contain multiple definitions of some method, provided each has a different signature. However, within a single program, no class or struct can contain more than one method called `Main` whose definition qualifies it to be used as an application entry point. Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter or their only parameter is other than type `string[]`.

An application can consist of multiple classes or structs. It is possible for more than one of these classes or structs to contain a method called `Main` whose definition qualifies it to be used as an application entry point. In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these `Main` methods as the entry point.

In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility (§3.5.1) of a method is determined by the access modifiers (§10.2.3) specified in its declaration, and similarly the declared accessibility of a type is determined by the access modifiers specified in its declaration. For a given method of a given type to be callable, both the type and the member must be accessible. However, the application entry point is a special case. Specifically, the execution environment can access the application's entry point regardless of its declared accessibility and regardless of the declared accessibility of its enclosing type declarations.

In all other respects, entry point methods behave like those that are not entry points.

3.2 Application Termination

Application termination returns control to the execution environment.

If the return type of the application's ***entry point*** method is `int`, the value returned serves as the application's ***termination status code***. The purpose of this code is to communicate success or failure to the execution environment.

If the return type of the entry point method is `void`, reaching the right brace (`}`) that terminates that method or executing a `return` statement that has no expression results in a termination status code of `0`.

Prior to an application's termination, destructors for all of its objects that have not yet been garbage collected are called, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example).

3.3 Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces (§9), which can contain type declarations and nested namespace declarations. Type declarations (§9.5) define classes (§10), structs (§11), interfaces (§13), enums (§14), and delegates (§15). The kinds of members permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants (§10.3), fields (§10.4), methods (§10.5), properties (§10.6), events (§10.7), indexers (§10.8), operators (§10.9), instance constructors (§10.10), static constructors (§10.11), destructors (§10.12), and nested types (§10.2.6).

A declaration defines a name in the ***declaration space*** to which the declaration belongs. Except for overloaded members (§3.6), it is a compile-time error to have two or more declarations that introduce members with the same name in a declaration space. It is never possible for a declaration space to contain different kinds of members with the same name. For example, a declaration space can never contain a field and a method by the same name.

There are several different types of declaration spaces, as described in the following.

- Within all source files of a program, *namespace-member-declaration*s with no enclosing *namespace-declaration* are members of a single combined declaration space called the ***global declaration space***.
- Within all source files of a program, *namespace-member-declaration*s within *namespace-declaration*s that have the same fully qualified namespace name are members of a single combined declaration space.
- Each class, struct, or interface declaration creates a new declaration space. Names are introduced into this declaration space through *class-member-declaration*s, *struct-member-declaration*s, or *interface-member-declaration*s. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct member declaration cannot introduce a member by the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators. For example, a class, struct, or interface may contain multiple method declarations with the same name, provided these method declarations differ in their signature (§3.6). Note that base classes do not contribute to the declaration space of a class, and base interfaces do not contribute to the declaration space of an interface. Thus, a derived class or interface is allowed to declare a member with the same name as an inherited member. Such a member is said to ***hide*** the inherited member.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through *enum-member-declarations*.

- Each *block* or *switch-block*, as well as a *for*, *foreach* and *using* statement, creates a different declaration space for local variables and constants called the ***local variable declaration space***. Names are introduced into this declaration space through *local-variable-declarations* and *local-constant-declarations*. If a block is the body of an instance constructor, method, or operator declaration, or it is a get or set accessor for an indexer declaration, then the parameters declared in such a declaration are members of the block's ***local variable declaration space***. The local variable declaration space of a block includes any nested blocks. Thus, within a nested block it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other block.
- Each *block* or *switch-block* creates a separate declaration space for labels. Names are introduced into this declaration space through *labeled-statement*s, and the names are referenced through *goto-statement*s. The ***label declaration space*** of a block includes any nested blocks. Thus, within a nested block it is not possible to declare a label with the same name as a label in an enclosing block.

The textual order in which names are declared is generally of no significance. In particular, textual order is not significant for the declaration and use of namespaces, constants, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types. Declaration order is significant in the following ways.

- Declaration order for field declarations and local variable declarations determines the order in which their initializers (if any) are executed.
- Local variables must be defined before they are used (§3.7).
- Declaration order for enum member declarations (§14.3) is significant when *constant-expression* values are omitted.

The declaration space of a namespace is "open ended," and two namespace declarations with the same fully qualified name contribute to the same declaration space.

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

These two namespace declarations contribute to the same declaration space, in this case declaring two classes with the fully qualified names `Megacorp.Data.Customer` and `Megacorp.Data.Order`. Because the two declarations contribute to the same declaration space, it would cause a compile-time error if each contained a declaration of a class with the same name.

As specified, the declaration space of a block includes any nested blocks. Thus, in the following example, the `F` and `G` methods result in a compile-time error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. However, the `H` and `I` methods are valid because the two `i`'s are declared in separate non-nested blocks.

```
class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }

    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }

    void H() {
        if (true) {
            int i = 0;
        }
        if (true) {
            int i = 1;
        }
    }

    void I() {
        for (int i = 0; i < 10; i++)
            H();
        for (int i = 0; i < 10; i++)
            H();
    }
}
```

3.4 Members

Namespaces and types have ***members***. The members of an entity are generally available by using a qualified name that starts with a reference to the entity, followed by a "`.`" token, followed by the name of the member.

Members of a type are either declared in the type or ***inherited*** from the base class of the type. When a type inherits from a base class, all members of the base class (except instance constructors, destructors, and static constructors) become members of the derived type. The declared accessibility of a base class member does not control whether the member is inherited—inheritance extends to any member that is not an instance constructor, static constructor, or destructor. However, an inherited member may not be accessible in a derived type either because of its declared accessibility (§3.5.1) or because it is hidden by a declaration in the type itself (§3.7.1.2).

3.4.1 Namespace Members

Namespaces and types that have no enclosing namespace are members of the ***global namespace***. This corresponds directly to the names declared in the global declaration space.

Namespaces and types declared within a namespace are members of that namespace. This corresponds directly to the names declared in the declaration space of the namespace.

Namespaces have no access restrictions. It is not possible to declare private, protected, or internal namespaces, and namespace names are always publicly accessible.

3.4.2 Struct Members

The members of a struct are the members declared in the struct and the members inherited from the struct's direct base class `System.ValueType` and the indirect base class `object`.

The members of a simple type correspond directly to the members of the struct type aliased by the simple type.

- The members of `sbyte` are the members of the `System.SByte` struct.
- The members of `byte` are the members of the `System.Byte` struct.
- The members of `short` are the members of the `System.Int16` struct.
- The members of `ushort` are the members of the `System.UInt16` struct.
- The members of `int` are the members of the `System.Int32` struct.
- The members of `uint` are the members of the `System.UInt32` struct.
- The members of `long` are the members of the `System.Int64` struct.
- The members of `ulong` are the members of the `System.UInt64` struct.
- The members of `char` are the members of the `System.Char` struct.
- The members of `float` are the members of the `System.Single` struct.
- The members of `double` are the members of the `System.Double` struct.
- The members of `decimal` are the members of the `System.Decimal` struct.
- The members of `bool` are the members of the `System.Boolean` struct.

3.4.3 Enumeration Members

The members of an enumeration are the constants declared in the enumeration and the members inherited from the enumeration's direct base class `System.Enum` and the indirect base classes `System.ValueType` and `object`.

3.4.4 Class Members

The members of a class are the members declared in the class and the members inherited from the base class (except for the class `object`, which has no base class). The members inherited from the base class include the constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not the instance constructors, destructors, and static constructors of the base class. Base class members are inherited without regard to their accessibility.

A class declaration may contain declarations of constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types.

The members of `object` and `string` correspond directly to the members of the class types they alias.

- The members of `object` are the members of the `System.Object` class.
- The members of `string` are the members of the `System.String` class.

3.4.5 Interface Members

The members of an interface are the members declared in the interface and in all base interfaces of the interface. The members in class `object` are not, strictly speaking, members of any interface (§13.2). However, the members in class `object` are available via member lookup in any interface type (§7.3) and the members inherited from the class `object`.

3.4.6 Array Members

The members of an array are the members inherited from the class `System.Array`.

3.4.7 Delegate Members

The members of a delegate are the members inherited from the class `System.Delegate`.

3.5 Member Access

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility (§3.5.1) of the member combined with the accessibility of the immediately containing type, if any.

When access to a particular member is allowed, the member is said to be ***accessible***. Conversely, when access to a particular member is disallowed, the member is said to be

inaccessible. Access to a member is permitted when the textual location in which the access takes place is included in the accessibility domain (§3.5.2) of the member.

3.5.1 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, but 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, but 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.

3.5.2 Accessibility Domains

The ***accessibility domain*** of a member consists of the (possibly disjoint) sections of program text in which access to the member is permitted. For purposes of defining the accessibility domain of a member, a member is said to be "top level" if it is not declared within a type, and a member is said to be "nested" if it is declared within another type. Furthermore, the program text of a program is defined as all program text contained in all source files of the program, and the program text of a type is defined as all program text contained between the opening and closing `{` and `}` tokens in the *class-body*, *struct-body*, *interface-body*, or *enum-body* of the type (including, possibly, types that are nested within the type).

The accessibility domain of a predefined type (such as `object`, `int`, or `double`) is unlimited.

The accessibility domain of a top-level type `T` that is declared in a program `P` is defined as follows.

- If the declared accessibility of `T` is `public`, the accessibility domain of `T` is the program text of `P` and any program that references `P`.
- If the declared accessibility of `T` is `internal`, the accessibility domain of `T` is the program text of `P`.

From these definitions it follows that the accessibility domain of a top-level type is always at least the program text of the program in which that type is declared.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that `M` itself may possibly be a type).

- If the declared accessibility of `M` is `public`, the accessibility domain of `M` is the accessibility domain of `T`.
- If the declared accessibility of `M` is `protected internal`, let `D` be the union of the program text of `P` and the program text of any type derived from `T`, which is declared outside `P`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.

- If the declared accessibility of M is protected, let D be the union of the program text of T and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- If the declared accessibility of M is internal, the accessibility domain of M is the intersection of the accessibility domain of T with the program text of P.
- If the declared accessibility of M is private, the accessibility domain of M is the program text of T.

From these definitions it follows that the accessibility domain of a nested member is always at least the program text of the type in which the member is declared. Furthermore, it follows that the accessibility domain of a member is never more inclusive than the accessibility domain of the type in which the member is declared.

In intuitive terms, when a type or member M is accessed, the following steps are evaluated to ensure that the access is permitted.

- First, if M is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if M is public, the access is permitted.
- Otherwise, if M is protected internal, the access is permitted if it occurs within the program in which M is declared or if it occurs within a class derived from the class in which M is declared and takes place through the derived class type (§3.5.3).
- Otherwise, if M is protected, the access is permitted if it occurs within the class in which M is declared or if it occurs within a class derived from the class in which M is declared and takes place through the derived class type (§3.5.3).
- Otherwise, if M is internal, the access is permitted if it occurs within the program in which M is declared.
- Otherwise, if M is private, the access is permitted if it occurs within the type in which M is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

In the following example

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}
```

```
internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

the classes and members have the following accessibility domains.

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the program text of the containing program.
- The accessibility domain of `A.Z` is the program text of `A`.
- The accessibility domain of `B.Z` and `B.D` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.C.Z` is the program text of `B.C`.
- The accessibility domain of `B.D.X` and `B.D.Y` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.D.Z` is the program text of `B.D`.

As the example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have `public` declared accessibility, all but `A.X` have accessibility domains that are constrained by a containing type.

As described in §3.4, all members of a base class (except for instance constructors, destructors, and static constructors) are inherited by derived types. This includes even private members of a base class. However, the accessibility domain of a private member includes only the program text of the type in which the member is declared. In the following example, the `B` class inherits the private member `x` from the `A` class.

```
class A
{
    int x;
```

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

class B: A
{
    static void F(B b) {
        b.x = 1;    // Error, x not accessible
    }
}
```

Because the member is private, it is only accessible within the *class-body* of `A`. Thus, the access to `b.x` succeeds in the `A.F` method but fails in the `B.F` method.

3.5.3 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 is required to take place *through* an instance of the derived class type in which the access occurs. In other words, 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 `D` or a class derived 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* (§10.10.1).

In the following example, within `A`, it is possible to access `x` through instances of both `A` and `B` because in either case the access takes place *through* an instance of `A` or a class derived from `A`.

```
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
    }
}
```

However, within `B`, it is not possible to access `x` through an instance of `A` because `A` does not derive from `B`.

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

```
class A {...}

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

Likewise, in the following example, 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.

```
class A {...}

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

    internal A G() {...}

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

3.6 Signatures and Overloading

Methods, instance constructors, indexers, and operators are characterized by their ***signatures***.

- The signature of a method consists of the name of the method and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of a method specifically does not include the return type; furthermore, it does not include the `params` modifier that may be specified for the rightmost parameter.
- The signature of an instance constructor consists of the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of an instance constructor specifically does not include the `params` modifier that may be specified for the rightmost parameter.
- The signature of an indexer consists of the type of each of its formal parameters, considered in the order left to right. The signature of an indexer specifically does not include the element type, nor does it include the `params` modifier that may be specified for the right-most parameter.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.

Signatures are the enabling mechanism for ***overloading*** of members in classes, structs, and interfaces.

- Overloading of methods permits a class, struct, or interface to declare multiple methods with the same name, provided their signatures are unique within that class, struct, or interface.
- Overloading of instance constructors permits a class or struct to declare multiple instance constructors, provided their signatures are unique within that class or struct.
- Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their signatures are unique within that class, struct, or interface.
- Overloading of operators permits a class or struct to declare multiple operators with the same name, provided their signatures are unique within that class or struct.

The following example shows a set of overloaded method declarations along with their signatures.

```
interface ITest
{
    void F();                      // F()
    void F(int x);                 // F(int)
    void F(ref int x);             // F(ref int)
    void F(int x, int y);          // F(int, int)
    int F(string s);               // F(string)
    int F(int x);                  // F(int)      error
    void F(string[] a);            // F(string[])
    void F(params string[] a);     // F(string[]) error
}
```

Note that any `ref` and `out` parameter modifiers (§10.5.1) are part of a signature. Thus, `F(int)` and `F(ref int)` are unique signatures. Also, note that the return type and the `params` modifier are not part of a signature, so it is not possible to overload solely based on the return type or on the inclusion or exclusion of the `params` modifier. As such, the declarations of the methods `F(int)` and `F(params string[])` identified in the previous example result in a compile-time error.

3.7 Scopes

The ***scope*** of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be ***nested***, and an inner scope may redeclare the meaning of a name from an outer scope (this does not, however, remove the restriction imposed by §3.3 that within a nested block it is not possible to declare a local variable with the same name as a local variable in an enclosing block).

The name from the outer scope is then said to be hidden in the region of program text covered by the inner scope, and access to the outer name is only possible by qualifying the name.

- The scope of a namespace member declared by a *namespace-member-declaration* (§9.4) with no enclosing *namespace-declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-declaration* whose fully qualified name is `N` is the *namespace-body* of every *namespace-declaration* whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of a name defined or imported by a *using-directive* (§9.3) extends over the *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *using-directive* occurs. A *using-directive* may make zero or more namespace or type names 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, a *using-directive* is not transitive but rather affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a member declared by a *class-member-declaration* (§10.2) is the *class-body* in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes that are included in the accessibility domain (§3.5.2) of the member.
- The scope of a member declared by a *struct-member-declaration* (§11.2) is the *struct-body* in which the declaration occurs.
- The scope of a member declared by an *enum-member-declaration* (§14.3) is the *enum-body* in which the declaration occurs.
- The scope of a parameter declared in a *method-declaration* (§10.5) is the *method-body* of that *method-declaration*.
- The scope of a parameter declared in an *indexer-declaration* (§10.8) is the *accessor-declarations* of that *indexer-declaration*.
- The scope of a parameter declared in an *operator-declaration* (§10.9) is the *block* of that *operator-declaration*.
- The scope of a parameter declared in a *constructor-declaration* (§10.10) is the *constructor-initializer* and *block* of that *constructor-declaration*.
- The scope of a label declared in a *labeled-statement* (§8.4) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local-variable-declaration* (§8.5.1) is the block in which the declaration occurs.

- The scope of a local variable declared in a *switch-block* of a `switch` statement (§8.7.2) is the *switch-block*.
- The scope of a local variable declared in a *for-initializer* of a `for` statement (§8.8.3) is the *for-initializer*, the *for-condition*, the *for-iterator*, and the contained *statement* of the `for` statement.
- The scope of a local constant declared in a *local-constant-declaration* (§8.5.2) is the block in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant-declarator*.

Within the scope of a namespace, class, struct, or enumeration member, it is possible to refer to the member in a textual position that precedes the declaration of the member. In the following example, it is valid for `F` to refer to `i` before it is declared.

```
class A
{
    void F() {
        i = 1;
    }
    int i = 0;
}
```

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local-variable-declarator* of the local variable. For example, in the `F` method, the first assignment to `i` specifically does not refer to the field declared in the outer scope.

```
class A
{
    int i = 0;

    void F() {
        i = 1;                  // Error, use precedes declaration
        int i;
        i = 2;
    }

    void G() {
        int j = (j = 1);        // Valid
    }

    void H() {
        int a = 1, b = ++a;     // Valid
    }
}
```

Rather, it refers to the local variable and results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, using `j` in the initializer for the declaration of `j` is valid because its use does not precede the *local-variable-declarator*. In the

H method, a subsequent *local-variable-declarator* correctly refers to a local variable declared in an earlier *local-variable-declarator* within the same *local-variable-declaration*.

The scoping rules for local variables are designed to guarantee that the meaning of a name used in an expression context is always the same within a block. If the scope of a local variable were to extend only from its declaration to the end of the block, then in the previous example the first assignment would assign to the instance variable and the second assignment would assign to the local variable, possibly leading to compile-time errors if the statements of the block were later rearranged.

The meaning of a name within a block may differ based on the context in which the name is used. In the following example, the name A is used in an expression context to refer to the local variable A and in a type context to refer to the class A.

```
using System;

class A {}

class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;                          // expression context
        Type t = typeof(A);                    // type context
        Console.WriteLine(s);                  // writes "hello, world"
        Console.WriteLine(t);                  // writes "A"
    }
}
```


3.7.1 Name Hiding

The scope of an entity typically encompasses more program text than the declaration space of the entity. In particular, the scope of an entity may include declarations that introduce new declaration spaces containing entities of the same name. Such declarations cause the original entity to become ***hidden***. Conversely, an entity is said to be ***visible*** when it is not hidden.

Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The characteristics of the two types of hiding are described in the following sections.

3.7.1.1 *Hiding through Nesting*

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter and local variable declarations.

In the example

```
class A
{
    int i = 0;

    void F() {
        int i = 1;
    }

    void G() {
        i = 1;
    }
}
```

within the `F` method, the instance variable `i` is hidden by the local variable `i`, but within the `G` method, `i` still refers to the instance variable.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. In the example

```
class Outer
{
    static void F(int i) {}

    static void F(string s) {}

    class Inner
    {
        void G() {
            F(1);              // Invokes Outer.Inner.F
            F("Hello");        // Error
        }
        static void F(long l) {}
    }
}
```

the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` results in a compile-time error.

3.7.1.2 *Hiding through Inheritance*

Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from base classes. This type of name hiding takes one of the following forms.

- A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.
- A method introduced in a class or struct hides all nonmethod base class members with the same name and all base class methods with the same signature (method name and parameter count, modifiers, and types).
- An indexer introduced in a class or struct hides all base class indexers with the same signature (parameter count and types).

The rules governing operator declarations (§10.9) make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus, operators never hide one another.

Contrary to hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a warning to be reported. In the following example, the declaration of `F` in `Derived` causes a warning to be reported.

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    public void F() {}     // Warning, hiding an inherited name
}
```

Hiding an inherited name is specifically not an error because that would preclude separate evolution of base classes. For example, the previous situation might have come about because a later version of `Base` introduced an `F` method that was not present in an earlier version of the class. Had the previous situation been an error, then *any* change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by hiding an inherited name can be eliminated by using the `new` modifier.

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}
```

The `new` modifier indicates that the `F` in `Derived` is "new" and that it is indeed intended to hide the inherited member.

A declaration of a new member hides an inherited member only within the scope of the new member. In the following example

```
class Base
{
    public static void F() {}
}
```

```
class Derived: Base
{
    new private static void F() {}    // Hides Base.F in Derived only
}

class MoreDerived: Derived
{
    static void G() { F(); }          // Invokes Base.F
}
```

the declaration of `F` in `Derived` hides the `F` that was inherited from `Base`, but because the new `F` in `Derived` has private access, its scope does not extend to `MoreDerived`. Thus, the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`.

3.8 Namespace and Type Names

Several contexts in a C# program require a *namespace-name* or a *type-name* to be specified. Either form of name is written as one or more identifiers separated by "`.`" tokens.

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier
    namespace-or-type-name `.` *identifier*

A *type-name* is a *namespace-or-type-name* that refers to a type. Following resolution as described shortly, the *namespace-or-type-name* of a *type-name* must refer to a type; otherwise, a compile-time error occurs.

A *namespace-name* is a *namespace-or-type-name* that refers to a namespace. Following resolution as described shortly, the *namespace-or-type-name* of a *namespace-name* must refer to a namespace; otherwise, a compile-time error occurs.

The meaning of a *namespace-or-type-name* is determined as follows.

- If the *namespace-or-type-name* consists of a single identifier, then the following happens.
  - If the *namespace-or-type-name* appears within the body of a class or struct declaration, then starting with that class or struct declaration and continuing with each enclosing class or struct declaration (if any), if a member with the given name exists, is accessible, and denotes a type, then the *namespace-or-type-name* refers to that member. Note that nontype members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) are ignored when determining the meaning of a *namespace-or-type-name*.

  - Otherwise, starting with the namespace in which the *namespace-or-type-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located.
    - If the namespace contains a namespace member with the given name, then the *namespace-or-type-name* refers to that member and, depending on the member, is classified as a namespace or a type.
    - Otherwise, if the namespace has a corresponding namespace declaration enclosing the location where the *namespace-or-type-name* occurs, then the following occurs.
      - If the namespace declaration contains a *using-alias-directive* that associates the given name with an imported namespace or type, then the *namespace-or-type-name* refers to that namespace or type.
      - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type with the given name, then the *namespace-or-type-name* refers to that type.
      - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type with the given name, then the *namespace-or-type-name* is ambiguous and an error occurs.
  - Otherwise, the *namespace-or-type-name* is undefined and a compile-time error occurs.
- Otherwise, the *namespace-or-type-name* is of the form `N.I`, where `N` is a *namespace-or-type-name* consisting of all identifiers but the rightmost one, and `I` is the rightmost identifier. `N` is first resolved as a *namespace-or-type-name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` is resolved as follows.
  - If `N` is a namespace and `I` is the name of an accessible member of that namespace, then `N.I` refers to that member and, depending on the member, is classified as a namespace or a type.
  - If `N` is a class or struct type and `I` is the name of an accessible type in `N`, then `N.I` refers to that type.
  - Otherwise, `N.I` is an *invalid namespace-or-type-name*, and a compile-time error occurs.

3.8.1 Fully Qualified Names

Every namespace and type has a ***fully qualified name***, which uniquely identifies the namespace or type amongst all others. The fully qualified name of a namespace or type `N` is determined as follows.

- If `N` is a member of the global namespace, its fully qualified name is `N`.
- Otherwise, its fully qualified name is `S.N`, where `S` is the fully qualified name of the namespace or type in which `N` is declared.

In other words, the fully qualified name of `N` is the complete hierarchical path of identifiers that lead to `N`, starting from the global namespace. Because every member of a namespace or type must have a unique name, it follows that the fully qualified name of a namespace or type is always unique.

The following example shows several namespace and type declarations along with their associated fully qualified names.

```
class A {}              // A

namespace X             // X
{
    class B             // X.B
    {
        class C {}      // X.B.C
    }

    namespace Y         // X.Y
    {
        class D {}      // X.Y.D
    }
}

namespace X.Y           // X.Y
{
    class E {}          // X.Y.E
}
```

3.9 Automatic Memory Management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a ***garbage collector***. The memory management life cycle of an object is as follows.

1. When the object is created, memory is allocated for it, the constructor is run, and the object is considered live.
2. If the object, or any part of it, cannot be accessed by any possible continuation of execution, other than the running of destructors, the object is considered no longer in use, and it becomes eligible for destruction. The C# compiler and the garbage collector may choose to analyze code to determine which references to an object may be used in the

future. For instance, if a local variable that is in scope is the only existing reference to an object, but that local variable is never referred to in any possible continuation of execution from the current execution point in the procedure, the garbage collector may (but is not required to) treat the object as no longer in use.

3. Once the object is eligible for destruction, at some unspecified later time, the destructor (§10.12) (if any) for the object is run. Unless overridden by explicit calls, the destructor for the object is run once only.
4. Once the destructor for an object is run, if that object (or any part of it) cannot be accessed by any possible continuation of execution, including the running of destructors, the object is considered inaccessible and the object becomes eligible for collection.
5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

The garbage collector maintains information about object usage and uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or inaccessible.

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not require that destructors be run or that objects be collected as soon as they are eligible or that destructors be run in any particular order or on any particular thread.

The behavior of the garbage collector can be controlled, to some degree, via static methods on the class `System.GC`. This class can be used to request a collection to occur, destructors to be run (or not run), and so forth.

Because the garbage collector is allowed wide latitude in deciding when to collect objects and run destructors, a conforming implementation may produce output that differs from that shown by the following code. The program creates an instance of class `A` and an instance of class `B`.

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}

class B
{
    object Ref;
```

```
    public B(object o) {
        Ref = o;
    }
    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}

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

These objects become eligible for garbage collection when the variable `b` is assigned the value `null` because after this time it is impossible for any user-written code to access them. The output could be either the following

```
Destruct instance of A
Destruct instance of B
```

or the following

```
Destruct instance of B
Destruct instance of A
```

because the language imposes no constraints on the order in which objects are garbage collected.

In subtle cases, the distinction between "eligible for destruction" and "eligible for collection" can be important. For example, in the following code

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }

    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}
class B
{
    public A Ref;
```

```
    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}
```

if the garbage collector chooses to run the destructor of `A` before the destructor of `B`, then the output of this program might be as follows.

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```

Note that although the instance of `A` was not in use and `A`'s destructor was run, it is still possible for methods of `A` (in this case, `F`) to be called from another destructor. Also, note that running of a destructor may cause an object to become usable from the mainline program again. In this case, the running of `B`'s destructor caused an instance of `A` that was previously not in use to become accessible from the live reference `Test.RefA`. After the call to `WaitForPendingFinalizers`, the instance of `B` is eligible for collection, but the instance of `A` is not because of the reference `Test.RefA`.

To avoid confusion and unexpected behavior, it is generally a good idea for destructors to only perform cleanup on data stored in their object's own fields and not to perform any actions on referenced objects or static fields.

3.10 Execution Order

Execution of a C# program proceeds such that the side effects of each executing thread are preserved at critical execution points. A ***side effect*** is defined as a read or write of a volatile field, a write to a nonvolatile variable, a write to an external resource, and the throwing of an exception. The critical execution points at which the order of these side effects must be preserved are references to volatile fields (§10.4.3), `lock` statements (§8.12), and thread creation and termination. The execution environment is free to change the order of execution of a C# program, subject to the following constraints.

- Data dependence is preserved within a thread of execution. That is, the value of each variable is computed as if all statements in the thread were executed in original program order.
- Initialization ordering rules are preserved (§10.4.4 and §10.4.5).
- The ordering of side effects is preserved with respect to volatile reads and writes (§10.4.3). Additionally, the execution environment need not evaluate part of an expression if it can deduce that that expression's value is not used and that no needed side effects are produced (including any caused by calling a method or accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an exception thrown by another thread), it is not guaranteed that the observable side effects are visible in the original program order.