8 Types

Article • 04/07/2023

8.1 General

The types of the C# language are divided into two main categories: ***reference types*** and ***value types***. Both value types and reference types may be ***generic types***, which take one or more ***type parameters***. Type parameters can designate both value types and reference types.

```ANTLR
type
    : reference_type
    | value_type
    | type_parameter
    | pointer_type     // unsafe code support
    ;
```

pointer_type (§23.3) is available only in unsafe code (§23).

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store ***references*** to their data, the latter being known as ***objects***. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

> *Note*: When a variable is a ref or out parameter, it does not have its own storage but references the storage of another variable. In this case, the ref or out variable is effectively an alias for another variable and not a distinct variable. *end note*

C#'s type system is unified such that *a value of any type can be treated as an object*. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing boxing and unboxing operations (§8.3.13).

For convenience, throughout this specification, some library type names are written without using their full name qualification. Refer to §C.5 for more information.

8.2 Reference types

8.2.1 General

A reference type is a class type, an interface type, an array type, a delegate type, or the `dynamic` type.

ANTLR

```
reference_type
    : class_type
    | interface_type
    | array_type
    | delegate_type
    | 'dynamic'
    ;

class_type
    : type_name
    | 'object'
    | 'string'
    ;

interface_type
    : type_name
    ;

array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : value_type
    | class_type
    | interface_type
    | delegate_type
    | 'dynamic'
    | type_parameter
    | pointer_type      // unsafe code support
    ;

rank_specifier
    : '[' ','* ']'
    ;

delegate_type
    : type_name
    ;
```

pointer_type is available only in unsafe code (§23.3).

A reference type value is a reference to an ***instance*** of the type, the latter known as an object. The special value `null` is compatible with all reference types and indicates the absence of an instance.

8.2.2 Class types

A class type defines a data structure that contains ***data members*** (constants and fields), ***function members*** (methods, properties, events, indexers, operators, instance constructors, finalizers, and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using *object_creation_expression*s (§12.8.16.2).

Class types are described in §15.

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 §8.2.3.
`System.String`	The string type of the C# language. See §8.2.5.
`System.ValueType`	The base class of all value types. See §8.3.2.
`System.Enum`	The base class of all `enum` types. See §19.5.
`System.Array`	The base class of all array types. See §17.2.2.
`System.Delegate`	The base class of all `delegate` types. See §20.1.
`System.Exception`	The base class of all exception types. See §21.3.

8.2.3 The object type

The `object` class type is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the `object` class type.

The keyword `object` is simply an alias for the predefined class `System.Object`.

8.2.4 The dynamic type

The `dynamic` type, like `object`, can reference any object. When operations are applied to expressions of type `dynamic`, their resolution is deferred until the program is run. Thus, if the operation cannot legitimately be applied to the referenced object, no error is given during compilation. Instead, an exception will be thrown when resolution of the operation fails at run-time.

The `dynamic` type is further described in §8.7, and dynamic binding in §12.3.1.

8.2.5 The string type

The `string` type is a sealed class type that inherits directly from `object`. Instances of the `string` class represent Unicode character strings.

Values of the `string` type can be written as string literals (§6.4.5.6).

The keyword `string` is simply an alias for the predefined class `System.String`.

8.2.6 Interface types

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

Interface types are described in §18.

8.2.7 Array types

An array is a data structure that contains zero or more variables, which are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

Array types are described in §17.

8.2.8 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.

> *Note*: 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. *end note*

Delegate types are described in §20.

8.3 Value types

8.3.1 General

A value type is either a struct type or an enumeration type. C# provides a set of predefined struct types called the ***simple types***. The simple types are identified through keywords.

ANTLR

```
value_type
    : non_nullable_value_type
    | nullable_value_type
    ;

non_nullable_value_type
    : struct_type
    | enum_type
    ;

struct_type
    : type_name
    | simple_type
    | tuple_type
    ;

simple_type
    : numeric_type
    | 'bool'
    ;

numeric_type
    : integral_type
    | floating_point_type
    | 'decimal'
    ;

integral_type
    : 'sbyte'
    | 'byte'
    | 'short'
    | 'ushort'
    | 'int'
    | 'uint'
    | 'long'
    | 'ulong'
    | 'char'
    ;

floating_point_type
    : 'float'
    | 'double'
    ;

tuple_type
    : '(' tuple_type_element (',' tuple_type_element)+ ')'
    ;

tuple_type_element
    : type identifier?
    ;

enum_type
    : type_name
    ;

nullable_value_type
    : non_nullable_value_type '?'
    ;
```

Unlike a variable of a reference type, a variable of a value type can contain the value `null` only if the value type is a nullable value type (§8.3.12). For every non-nullable value type there is a corresponding nullable value type denoting the same set of values plus the value `null`.

Assignment to a variable of a value type creates a *copy* of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

8.3.2 The System.ValueType type

All value types implicitly inherit from the `class` `System.ValueType`, which, in turn, inherits from class `object`. It is not possible for any type to derive from a value type, and value types are thus implicitly sealed (§15.2.2.3).

Note that `System.ValueType` is not itself a *value_type*. Rather, it is a *class_type* from which all *value_type*s are automatically derived.

8.3.3 Default constructors

All value types implicitly declare a public parameterless instance constructor called the ***default constructor***. The default constructor returns a zero-initialized instance known as the ***default value*** for the value type:

- For all *simple_types*, the default value is the value produced by a bit pattern of all zeros:
  - For `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`, the default value is `0`.
  - For `char`, the default value is `'\x0000'`.
  - For `float`, the default value is `0.0f`.
  - For `double`, the default value is `0.0d`.
  - For `decimal`, the default value is `0m` (that is, value zero with scale 0).
  - For `bool`, the default value is `false`.
  - For an *enum_type* `E`, the default value is `0`, converted to the type `E`.
- For a *struct_type*, the default value is the value produced by setting all value type fields to their default value and all reference type fields to `null`.
- For a *nullable_value_type* the default value is an instance for which the `HasValue` property is false. The default value is also known as the ***null value*** of the nullable value type. Attempting to read the `Value` property of such a value causes an exception of type `System.InvalidOperationException` to be thrown (§8.3.12).

Like any other instance constructor, the default constructor of a value type is invoked using the `new` operator.

> *Note*: For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. For value types, the default value expression (§12.8.20) produces the same result as using the default constructor. *end note*

> *Example*: In the code below, variables `i`, `j` and `k` are all initialized to zero.
>
> C#
>
> ```
> class A
> {
>     void F()
>     {
>         int i = 0;
>         int j = new int();
>         int k = default(int);
>     }
> }
> ```

> *end example*

Because every value type implicitly has a public parameterless instance constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is however permitted to declare parameterized instance constructors (§16.4.9).

8.3.4 Struct types

A struct type is a value type that can declare constants, fields, methods, properties, events, indexers, operators, instance constructors, static constructors, and nested types. The declaration of struct types is described in §16.

8.3.5 Simple types

C# provides a set of predefined `struct` types called the simple types. The simple types are identified through keywords, but these keywords are simply aliases for predefined `struct` types in the `System` namespace, as described in the table below.

Keyword	Aliased type
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

Because a simple type aliases a struct type, every simple type has members.

> *Example*: `int` has the members declared in `System.Int32` and the members inherited from `System.Object`, and the following statements are permitted:
>
> C#
> ```
> int i = int.MaxValue;      // System.Int32.MaxValue constant
> string s = i.ToString();   // System.Int32.ToString() instance
> method
> string t = 123.ToString(); // System.Int32.ToString() instance
> method
> ```
>
> *end example*

> *Note*: The simple types differ from other struct types in that they permit certain additional operations:
>
> - Most simple types permit values to be created by writing *literals* (§6.4.5), although C# makes no provision for literals of struct types in general. *Example*: `123` is a literal of type `int` and `'a'` is a literal of type `char`. *end example*
> - When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile-time. Such an expression is known as a *constant_expression* (§12.23). Expressions involving operators defined by other struct types are not considered to be constant expressions
> - Through `const` declarations, it is possible to declare constants of the simple types (§15.4). It is not possible to have constants of other struct types, but a similar effect is provided by static readonly fields.
> - Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined conversion operator (§10.5.3).
>
> *end note*.

8.3.6 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 from `-128` to `127`, inclusive.
- The `byte` type represents unsigned 8-bit integers with values from `0` to `255`, inclusive.
- The `short` type represents signed 16-bit integers with values from `-32768` to `32767`, inclusive.
- The `ushort` type represents unsigned 16-bit integers with values from `0` to `65535`, inclusive.
- The `int` type represents signed 32-bit integers with values from `-2147483648` to `2147483647`, inclusive.
- The `uint` type represents unsigned 32-bit integers with values from `0` to `4294967295`, inclusive.
- The `long` type represents signed 64-bit integers with values from `-9223372036854775808` to `9223372036854775807`, inclusive.
- The `ulong` type represents unsigned 64-bit integers with values from `0` to `18446744073709551615`, inclusive.
- The `char` type represents unsigned 16-bit integers with values from `0` to `65535`, inclusive. The set of possible values for the `char` type corresponds to the Unicode character set.

  > *Note*: Although `char` has the same representation as `ushort`, not all operations permitted on one type are permitted on the other. *end note*

All signed integral types are represented using two's complement format.

The *integral_type* unary and binary operators always operate with signed 32-bit precision, unsigned 32-bit precision, signed 64-bit precision, or unsigned 64-bit precision, as detailed in §12.4.7.

The `char` type is classified as an integral type, but it differs from the other integral types in two ways:

- There are no predefined implicit conversions from other types to the `char` type. In particular, even though the `byte` and `ushort` types have ranges of values that are fully representable using the `char` type, implicit conversions from sbyte, byte, or `ushort` to `char` do not exist.
- Constants of the `char` type shall be written as *character_literal*s or as *integer_literal*s in combination with a cast to type char.

> *Example*: `(char)10` is the same as `'\x000A'`. *end example*

The `checked` and `unchecked` operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions (§12.8.19). In a `checked` context, an overflow produces a compile-time error or causes a `System.OverflowException` to be thrown. In an `unchecked` context, overflows are ignored and any high-order bits that do not fit in the destination type are discarded.

8.3.7 Floating-point types

C# supports two floating-point types: `float` and `double`. The `float` and `double` types are represented using the 32-bit single-precision and 64-bit double-precision IEC 60559 formats, which provide the following sets of values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two (§12.10.3).
- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero.

> *Example*: `1.0 / 0.0` yields positive infinity, and `–1.0 / 0.0` yields negative infinity. *end example*

- The ***Not-a-Number*** value, often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.
- The finite set of non-zero values of the form $s \times m \times 2^e$, where s is 1 or −1, and m and e are determined by the particular floating-point type: For `float`, $0 < m < 2^{24}$ and $-149 \le e \le 104$, and for `double`, $0 < m < 2^{53}$ and $-1075 \le e \le 970$. Denormalized floating-point numbers are considered valid non-zero values. C# neither requires nor forbids that a conforming implementation support denormalized floating-point numbers.

The `float` type can represent values ranging from approximately 1.5×10^{-45} to 3.4×10^{38} with a precision of 7 digits.

The `double` type can represent values ranging from approximately 5.0×10^{-324} to 1.7×10^{308} with a precision of 15-16 digits.

If either operand of a binary operator is a floating-point type then standard numeric promotions are applied, as detailed in §12.4.7, and the operation is performed with `float` or `double` precision.

The floating-point operators, including the assignment operators, never produce exceptions. Instead, in exceptional situations, floating-point operations produce zero, infinity, or NaN, as described below:

- The result of a floating-point operation is rounded to the nearest representable value in the destination format.
- If the magnitude of the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the magnitude of the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating-point operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

Floating-point operations may be performed with higher precision than the result type of the operation. To force a value of a floating-point type to the exact precision of its type, an explicit cast (§12.9.7) can be used.

> *Example*: Some hardware architectures support an "extended" or "long double" floating-point type with greater range and precision than the `double` type, and implicitly perform all floating-point operations using this higher precision type. Only at excessive cost in performance can such hardware architectures be made to perform floating-point operations with *less* precision, and rather than require an implementation to forfeit both performance and precision, C# allows a higher precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `double` range, but the subsequent division brings the temporary result back into the `double` range, the fact that the expression is evaluated in a higher range format can cause a finite result to be produced instead of an infinity. *end example*

8.3.8 The Decimal type

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations. The `decimal` type can represent values including those in the range at least -7.9×10^{-28} to 7.9×10^{28}, with at least 28-digit precision.

The finite set of values of type `decimal` are of the form $(–1)^v \times c \times 10^{-e}$, where the sign v is 0 or 1, the coefficient c is given by $0 \le c < Cmax$, and the scale e is such that $Emin \le e$

≤ *Emax*, where *Cmax* is at least 1 × 10^{28}, *Emin* ≤ 0, and *Emax* ≥ 28. The `decimal` type does not necessarily support signed zeros, infinities, or NaN's.

A `decimal` is represented as an integer scaled by a power of ten. For `decimal`s with an absolute value less than `1.0m`, the value is exact to at least the 28th decimal place. For `decimal`s with an absolute value greater than or equal to `1.0m`, the value is exact to at least 28 digits. Contrary to the `float` and `double` data types, decimal fractional numbers such as `0.1` can be represented exactly in the decimal representation. In the `float` and `double` representations, such numbers often have non-terminating binary expansions, making those representations more prone to round-off errors.

If either operand of a binary operator is of `decimal` type then standard numeric promotions are applied, as detailed in §12.4.7, and the operation is performed with `double` precision.

The result of an operation on values of type `decimal` is that which would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value, and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as "banker's rounding"). That is, results are exact to at least the 28th decimal place. Note that rounding may produce a zero value from a non-zero value.

If a `decimal` arithmetic operation produces a result whose magnitude is too large for the `decimal` format, a `System.OverflowException` is thrown.

The `decimal` type has greater precision but may have a smaller range than the floating-point types. Thus, conversions from the floating-point types to `decimal` might produce overflow exceptions, and conversions from `decimal` to the floating-point types might cause loss of precision or overflow exceptions. For these reasons, no implicit conversions exist between the floating-point types and `decimal`, and without explicit casts, a compile-time error occurs when floating-point and `decimal` operands are directly mixed in the same expression.

8.3.9 The Bool type

The `bool` type represents Boolean logical quantities. The possible values of type `bool` are `true` and `false`. The representation of `false` is described in §8.3.3. Although the representation of `true` is unspecified, it shall be different from that of `false`.

No standard conversions exist between `bool` and other value types. In particular, the `bool` type is distinct and separate from the integral types, a `bool` value cannot be used in place of an integral value, and vice versa.

> *Note*: In the C and C++ languages, a zero integral or floating-point value, or a null pointer can be converted to the Boolean value `false`, and a non-zero integral or floating-point value, or a non-null pointer can be converted to the Boolean value `true`. In C#, such conversions are accomplished by explicitly comparing an integral or floating-point value to zero, or by explicitly comparing an object reference to `null`. *end note*

8.3.10 Enumeration types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which shall be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. The set of values of the enumeration type is the same as the set of values of the underlying type. Values of the enumeration type are not restricted to the values of the named constants. Enumeration types are defined through enumeration declarations (§19.2).

8.3.11 Tuple types

A tuple type represents an ordered, fixed-length sequence of values with optional names and individual types. The number of elements in a tuple type is referred to as its ***arity***. A tuple type is written `(T1 I1, ..., Tn In)` with n ≥ 2, where the identifiers `I1...In` are optional ***tuple element names***.

This syntax is shorthand for a type constructed with the types `T1...Tn` from `System.ValueTuple<...>`, which shall be a set of generic struct types capable of directly expressing tuple types of any arity between two and seven inclusive. There does not need to exist a `System.ValueTuple<...>` declaration that directly matches the arity of any tuple type with a corresponding number of type parameters. Instead, tuples with an arity greater than seven are represented with a generic struct type `System.ValueTuple<T1, ..., T7, TRest>` that in addition to tuple elements has a `Rest` field containing a nested value of the remaining elements, using another `System.ValueTuple<...>` type. Such nesting may be observable in various ways, e.g. via the presence of a `Rest` field. Where only a single additional field is required, the generic struct type `System.ValueTuple<T1>` is used; this type is not considered a tuple type in itself. Where more than seven additional fields are required, `System.ValueTuple<T1, ..., T7, TRest>` is used recursively.

Element names within a tuple type shall be distinct. A tuple element name of the form `ItemX`, where `X` is any sequence of non-`0`-initiated decimal digits that could represent the position of a tuple element, is only permitted at the position denoted by `X`.

The optional element names are not represented in the `ValueTuple<...>` types, and are not stored in the runtime representation of a tuple value. There is an identity conversion between all tuple types with the same arity and identity-convertible sequences of element types, as well as to and from the corresponding constructed `ValueTuple<...>` type.

The `new` operator §12.8.16.2 cannot be applied with the tuple type syntax `new (T1, ..., Tn)`. Tuple values can be created from tuple expressions (§12.8.6), or by applying the `new` operator directly to a type constructed from `ValueTuple<...>`.

Tuple elements are public fields with the names `Item1`, `Item2`, etc., and can be accessed via a member access on a tuple value (§12.8.7. Additionally, if the tuple type has a name for a given element, that name can be used to access the element in question.

> *Note*: Even when large tuples are represented with nested `System.ValueTuple<...>` values, each tuple element can still be accessed directly with the `Item...` name corresponding to its position. *end note*

> *Example*: Given the following examples:
>
> C#
>
> ```csharp
> (int, string) pair1 = (1, "One");
> (int, string word) pair2 = (2, "Two");
> (int number, string word) pair3 = (3, "Three");
> (int Item1, string Item2) pair4 = (4, "Four");
> // Error: "Item" names do not match their position
> (int Item2, string Item123) pair5 = (5, "Five");
> (int, string) pair6 = new ValueTuple<int, string>(6, "Six");
> ValueTuple<int, string> pair7 = (7, "Seven");
> Console.WriteLine($"{pair2.Item1}, {pair2.Item2}, {pair2.word}");
> ```
>
> The tuple types for `pair1`, `pair2`, and `pair3` are all valid, with names for no, some or all of the tuple type elements.

> The tuple type for `pair4` is valid because the names `Item1` and `Item2` match their positions, whereas the tuple type for `pair5` is disallowed, because the names `Item2` and `Item123` do not.
>
> The declarations for `pair6` and `pair7` demonstrate that tuple types are interchangeable with constructed types of the form `ValueTuple<...>`, and that the `new` operator is allowed with the latter syntax.
>
> The last line shows that tuple elements can be accessed by the `Item` name corresponding to their position, as well as by the corresponding tuple element name, if present in the type. *end example*

8.3.12 Nullable value types

A nullable value type can represent all values of its underlying type plus an additional null value. A nullable value type is written `T?`, where `T` is the underlying type. This syntax is shorthand for `System.Nullable<T>`, and the two forms can be used interchangeably.

Conversely, a ***non-nullable value type*** is any value type other than `System.Nullable<T>` and its shorthand `T?` (for any `T`), plus any type parameter that is constrained to be a non-nullable value type (that is, any type parameter with a value type constraint (§15.2.5)). The `System.Nullable<T>` type specifies the value type constraint for `T`, which means that the underlying type of a nullable value type can be any non-nullable value type. The underlying type of a nullable value type cannot be a nullable value type or a reference type. For example, `int??` and `string?` are invalid types.

An instance of a nullable value type `T?` has two public read-only properties:

- A `HasValue` property of type `bool`
- A `Value` property of type `T`

An instance for which `HasValue` is `true` is said to be non-null. A non-null instance contains a known value and `Value` returns that value.

An instance for which `HasValue` is `false` is said to be null. A null instance has an undefined value. Attempting to read the `Value` of a null instance causes a `System.InvalidOperationException` to be thrown. The process of accessing the Value property of a nullable instance is referred to as ***unwrapping***.

In addition to the default constructor, every nullable value type `T?` has a public constructor with a single parameter of type `T`. Given a value `x` of type `T`, a constructor invocation of the form

```C#
new T?(x)
```

creates a non-null instance of `T?` for which the `Value` property is `x`. The process of creating a non-null instance of a nullable value type for a given value is referred to as ***wrapping***.

Implicit conversions are available from the `null` literal to `T?` (§10.2.7) and from `T` to `T?` (§10.2.6).

The nullable type `T?` implements no interfaces (§18). In particular, this means it does not implement any interface that the underlying type `T` does.

8.3.13 Boxing and unboxing

The concept of boxing and unboxing provide a bridge between *value_type*s and *reference_type*s by permitting any value of a *value_type* to be converted to and from type `object`. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an `object`.

Boxing is described in more detail in §10.2.9 and unboxing is described in §10.3.7.

8.4 Constructed types

8.4.1 General

A generic type declaration, by itself, denotes an ***unbound generic type*** that is used as a "blueprint" to form many different types, by way of applying ***type arguments***. The type arguments are written within angle brackets (`<` and `>`) immediately following the name of the generic type. A type that includes at least one type argument is called a ***constructed type***. A constructed type can be used in most places in the language in which a type name can appear. An unbound generic type can only be used within a *typeof_expression* (§12.8.17).

Constructed types can also be used in expressions as simple names (§12.8.4) or when accessing a member (§12.8.7).

When a *namespace_or_type_name* is evaluated, only generic types with the correct number of type parameters are considered. Thus, it is possible to use the same identifier to identify different types, as long as the types have different numbers of type parameters. This is useful when mixing generic and non-generic classes in the same program.

> *Example*:
>
> C#
>
> ```
> namespace Widgets
> {
>     class Queue {...}
>     class Queue<TElement> {...}
> }
>
> namespace MyApplication
> {
>     using Widgets;
>
>     class X
>     {
>         Queue q1;      // Non-generic Widgets.Queue
>         Queue<int> q2; // Generic Widgets.Queue
>     }
> }
> ```
>
> *end example*

The detailed rules for name lookup in the *namespace_or_type_name* productions is described in §7.8. The resolution of ambiguities in these productions is described in §6.2.5. A *type_name* might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic `class` declaration, and the instance type of the containing declaration is implicitly used for name lookup (§15.3.9.7).

> *Example*:
>
> C#
>
> ```
> class Outer<T>
> {
>     public class Inner {...}
>
>     public Inner i; // Type of i is Outer<T>.Inner
> }
> ```

> *end example*

A non-enum constructed type shall not be used as an *unmanaged_type* (§8.8).

8.4.2 Type arguments

Each argument in a type argument list is simply a *type*.

```
ANTLR

type_argument_list
    : '<' type_arguments '>'
    ;

type_arguments
    : type_argument (',' type_argument)*
    ;

type_argument
    : type
    ;
```

Each type argument shall satisfy any constraints on the corresponding type parameter (§15.2.5).

8.4.3 Open and closed types

All types can be classified as either ***open types*** or ***closed types***. An open type is a type that involves type parameters. More specifically:

- A type parameter defines an open type.
- An array type is an open type if and only if its element type is an open type.
- A constructed type is an open type if and only if one or more of its type arguments is an open type. A constructed nested type is an open type if and only if one or more of its type arguments or the type arguments of its containing type(s) is an open type.

A closed type is a type that is not an open type.

At run-time, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic type is bound to a particular run-time type. The run-time processing of all statements and expressions always occurs with closed types, and open types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Since an open type does not exist at run-time, there are no static variables associated with an open type. Two closed constructed types are the same type if they are constructed from the same unbound generic type, and their corresponding type arguments are the same type.

8.4.4 Bound and unbound types

The term ***unbound type*** refers to a non-generic type or an unbound generic type. The term ***bound type*** refers to a non-generic type or a constructed type.

An unbound type refers to the entity declared by a type declaration. An unbound generic type is not itself a type, and cannot be used as the type of a variable, argument or return value, or as a base type. The only construct in which an unbound generic type can be referenced is the `typeof` expression (§12.8.17).

8.4.5 Satisfying constraints

Whenever a constructed type or generic method is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or method (§15.2.5). For each `where` clause, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows:

- If the constraint is a `class` type, an interface type, or a type parameter, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it shall be the case that type `A` is convertible to type `C` by one of the following:
    - An identity conversion (§10.2.2)
    - An implicit reference conversion (§10.2.8)
    - A boxing conversion (§10.2.9), provided that type `A` is a non-nullable value type.
    - An implicit reference, boxing or type parameter conversion from a type parameter `A` to `C`.
- If the constraint is the reference type constraint (`class`), the type `A` shall satisfy one of the following:
    - `A` is an interface type, class type, delegate type, array type or the dynamic type.

    > *Note*: `System.ValueType` and `System.Enum` are reference types that satisfy this constraint. *end note*

    - `A` is a type parameter that is known to be a reference type (§8.2).

- If the constraint is the value type constraint (`struct`), the type `A` shall satisfy one of the following:
  - `A` is a `struct` type or `enum` type, but not a nullable value type.

    > *Note*: `System.ValueType` and `System.Enum` are reference types that do not satisfy this constraint. *end note*

  - `A` is a type parameter having the value type constraint (§15.2.5).
- If the constraint is the constructor constraint `new()`, the type `A` shall not be `abstract` and shall have a public parameterless constructor. This is satisfied if one of the following is true:
  - `A` is a value type, since all value types have a public default constructor (§8.3.3).
  - `A` is a type parameter having the constructor constraint (§15.2.5).
  - `A` is a type parameter having the value type constraint (§15.2.5).
  - `A` is a `class` that is not abstract and contains an explicitly declared public constructor with no parameters.
  - `A` is not `abstract` and has a default constructor (§15.11.5).

A compile-time error occurs if one or more of a type parameter's constraints are not satisfied by the given type arguments.

Since type parameters are not inherited, constraints are never inherited either.

> *Example*: In the following, `D` needs to specify the constraint on its type parameter `T` so that `T` satisfies the constraint imposed by the base `class` `B<T>`. In contrast, `class` `E` need not specify a constraint, because `List<T>` implements `IEnumerable` for any `T`.
>
> C#
>
> ```csharp
> class B<T> where T: IEnumerable {...}
> class D<T> : B<T> where T: IEnumerable {...}
> class E<T> : B<List<T>> {...}
> ```
>
> *end example*

8.5 Type parameters

A type parameter is an identifier designating a value type or reference type that the parameter is bound to at run-time.

ANTLR

```
type_parameter
    : identifier
    ;
```

Since a type parameter can be instantiated with many different type arguments, type parameters have slightly different operations and restrictions than other types.

> *Note*: These include:
>
> - A type parameter cannot be used directly to declare a base class (§15.2.4.2) or interface (§18.2.4).
> - The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter. They are detailed in §12.5.
> - The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in §10.2.12 and §10.3.9.
> - The literal `null` cannot be converted to a type given by a type parameter, except if the type parameter is known to be a reference type (§10.2.12). However, a default expression (§12.8.20) can be used instead. In addition, a value with a type given by a type parameter *can* be compared with null using `==` and `!=` (§12.12.7) unless the type parameter has the value type constraint.
> - A `new` expression (§12.8.16.2) can only be used with a type parameter if the type parameter is constrained by a *constructor_constraint* or the value type constraint (§15.2.5).
> - A type parameter cannot be used anywhere within an attribute.
> - A type parameter cannot be used in a member access (§12.8.7) or type name (§7.8) to identify a static member or a nested type.
> - A type parameter cannot be used as an *unmanaged_type* (§8.8).
>
> *end note*

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a closed constructed type §8.4.3. The run-time execution of all statements and expressions involving type parameters uses the type that was supplied as the type argument for that parameter.

8.6 Expression tree types

Expression trees permit lambda expressions to be represented as data structures instead of executable code. Expression trees are values of ***expression tree types*** of the form `System.Linq.Expressions.Expression<TDelegate>`, where `TDelegate` is any delegate type. For the remainder of this specification these types will be referred to using the shorthand `Expression<TDelegate>`.

If a conversion exists from a lambda expression to a delegate type `D`, a conversion also exists to the expression tree type `Expression<TDelegate>`. Whereas the conversion of a lambda expression to a delegate type generates a delegate that references executable code for the lambda expression, conversion to an expression tree type creates an expression tree representation of the lambda expression. More details of this conversion are provided in §10.7.3.

> *Example*: The following program represents a lambda expression both as executable code and as an expression tree. Because a conversion exists to `Func<int,int>`, a conversion also exists to `Expression<Func<int,int>>`:
>
> C#
>
> ```
> Func<int,int> del = x => x + 1;              // Code
> Expression<Func<int,int>> exp = x => x + 1; // Data
> ```
>
> Following these assignments, the delegate `del` references a method that returns `x + 1`, and the expression tree exp references a data structure that describes the expression `x => x + 1`.
>
> *end example*

`Expression<TDelegate>` provides an instance method `Compile` which produces a delegate of type `TDelegate`:

C#

```
Func<int,int> del2 = exp.Compile();
```

Invoking this delegate causes the code represented by the expression tree to be executed. Thus, given the definitions above, `del` and `del2` are equivalent, and the following two statements will have the same effect:

C#

```
int i1 = del(1);
int i2 = del2(1);
```

After executing this code, `i1` and `i2` will both have the value `2`.

The API surface provided by `Expression<TDelegate>` is implementation-specific beyond the requirement for a `Compile` method described above.

> *Note*: While the details of the API provided for expression trees are implementation-specific, it is expected that an implementation will:
>
> - Enable code to inspect and respond to the structure of an expression tree created as the result of a conversion from a lambda expression
> - Enable expression trees to be created programatically within user code
>
> *end note*

8.7 The dynamic type

The type `dynamic` uses dynamic binding, as described in detail in §12.3.2, as opposed to static binding which is used by all other types.

`dynamic` is considered identical to `object` except in the following respects:

- Operations on expressions of type `dynamic` can be dynamically bound (§12.3.3).
- Type inference (§12.6.3) will prefer `dynamic` over `object` if both are candidates.
- `dynamic` cannot be used as
  - the type in an *object_creation_expression* (§12.8.16.2)
  - a *predefined_type* in a *member_access* (§12.8.7.1)
  - the operand of the `typeof` operator
  - an attribute argument
  - a constraint
  - an extension method type
  - any part of a type argument within *struct_interfaces* (§16.2.5) or *interface_type_list* (§15.2.4.1).

Because of this equivalence, the following holds:

- There is an implicit identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing `dynamic` with `object`.

- Implicit and explicit conversions to and from `object` also apply to and from `dynamic`.
- Signatures that are the same when replacing `dynamic` with `object` are considered the same signature.
- The type `dynamic` is indistinguishable from `object` at run-time.
- An expression of the type `dynamic` is referred to as a ***dynamic expression***.

8.8 Unmanaged types

ANTLR

```
unmanaged_type
    : value_type
    | pointer_type     // unsafe code support
    ;
```

An *unmanaged_type* is any type that isn’t a *reference_type*, a *type_parameter*, or a constructed type, and contains no instance fields whose type is not an *unmanaged_type*. In other words, an *unmanaged_type* is one of the following:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.
- Any *enum_type*.
- Any user-defined *struct_type* that is not a constructed type and contains instance fields of *unmanaged_type*s only.