10 Conversions

Article • 03/09/2023

10.1 General

A ***conversion*** causes an expression to be converted to, or treated as being of, a particular type; in the former case a conversion may involve a change in representation. Conversions can be ***implicit*** or ***explicit***, and this determines whether an explicit cast is required.

> *Example*: For instance, the conversion from type `int` to type `long` is implicit, so expressions of type `int` can implicitly be treated as type `long`. The opposite conversion, from type `long` to type `int`, is explicit and so an explicit cast is required.
>
> C#
>
> ```
> int a = 123;
> long b = a;      // implicit conversion from int to long
> int c = (int) b; // explicit conversion from long to int
> ```
>
> *end example*

Some conversions are defined by the language. Programs may also define their own conversions (§10.5).

Some conversions in the language are defined from expressions to types, others from types to types. A conversion from a type applies to all expressions that have that type.

> *Example*:
>
> C#
>
> ```
> enum Color { Red, Blue, Green }
>
> // The expression 0 converts implicitly to enum types
> Color c0 = 0;
>
> // Other int expressions need explicit conversion
> Color c1 = (Color)1;
>
> // Conversion from null expression (no type) to string
> string x = null;
> ```

```
// Conversion from lambda expression to delegate type
Func<int, int> square = x => x * x;
```

end example

10.2 Implicit conversions

10.2.1 General

The following conversions are classified as implicit conversions:

- Identity conversions
- Implicit numeric conversions
- Implicit enumeration conversions
- Implicit interpolated string conversions
- Implicit reference conversions
- Boxing conversions
- Implicit dynamic conversions
- Implicit type parameter conversions
- Implicit constant expression conversions
- User-defined implicit conversions
- Anonymous function conversions
- Method group conversions
- Null literal conversions
- Implicit nullable conversions
- Implicit tuple conversions
- Lifted user-defined implicit conversions
- Default literal conversions
- Implicit throw conversion

Implicit conversions can occur in a variety of situations, including function member invocations (§12.6.6), cast expressions (§12.9.7), and assignments (§12.21).

The pre-defined implicit conversions always succeed and never cause exceptions to be thrown.

> *Note*: Properly designed user-defined implicit conversions should exhibit these characteristics as well. *end note*

For the purposes of conversion, the types `object` and `dynamic` are considered equivalent.

However, dynamic conversions (§10.2.10 and §10.3.8) apply only to expressions of type `dynamic` (§8.2.4).

10.2.2 Identity conversion

An identity conversion converts from any type to the same type. One reason this conversion exists is so that a type T or an expression of type T can be said to be convertible to T itself.

In some cases there is an identity conversion between types that are not exactly the same, but are considered equivalent. Such identity conversions exist:

- between `object` and `dynamic`.
- between tuple types with the same arity, when an identity conversion exists between each pair of corresponding element types.
- between types constructed from the same generic type where there exists an identity conversion between each corresponding type argument.

In most cases, an identity conversion has no effect at runtime. However, since floating point operations may be performed at higher precision than prescribed by their type (§8.3.7), assignment of their results may result in a loss of precision, and explicit casts are guaranteed to reduce precision to what is prescribed by the type (§12.9.7).

10.2.3 Implicit numeric conversions

The implicit numeric conversions are:

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

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

There are no predefined implicit conversions to the `char` type, so values of the other integral types do not automatically convert to the `char` type.

10.2.4 Implicit enumeration conversions

An implicit enumeration conversion permits a *constant_expression* (§12.23) with any integer type and the value zero to be converted to any *enum_type* and to any *nullable_value_type* whose underlying type is an *enum_type*. In the latter case the conversion is evaluated by converting to the underlying *enum_type* and wrapping the result (§8.3.12).

10.2.5 Implicit interpolated string conversions

An implicit interpolated string conversion permits an *interpolated_string_expression* (§12.8.3) to be converted to `System.IFormattable` or `System.FormattableString` (which implements `System.IFormattable`). When this conversion is applied, a string value is not composed from the interpolated string. Instead an instance of `System.FormattableString` is created, as further described in §12.8.3.

10.2.6 Implicit nullable conversions

The implicit nullable conversions are those nullable conversions (§10.6.1) derived from implicit predefined conversions.

10.2.7 Null literal conversions

An implicit conversion exists from the `null` literal to any reference type or nullable value type. This conversion produces a null reference if the target type is a reference type, or the null value (§8.3.12) of the given nullable value type.

10.2.8 Implicit reference conversions

The implicit reference conversions are:

- From any *reference_type* to `object` and `dynamic`.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is derived from `T`.

- From any *class_type* `S` to any *interface_type* `T`, provided `S` implements `T`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is derived from `T`.
- From an *array_type* `S` with an element type S_i to an *array_type* `T` with an element type T_i, provided all of the following are true:
  - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
  - An implicit reference conversion exists from S_i to T_i.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>`, `System.Collections.Generic.IReadOnlyList<T>`, and their base interfaces, provided that there is an implicit identity or reference conversion from `S` to `T`.
- From any *array_type* to `System.Array` and the interfaces it implements.
- From any *delegate_type* to `System.Delegate` and the interfaces it implements.
- From the null literal (§6.4.5.7) to any reference-type.
- From any *reference_type* to a *reference_type* `T` if it has an implicit identity or reference conversion to a *reference_type* T_0 and T_0 has an identity conversion to `T`.
- From any *reference_type* to an interface or delegate type `T` if it has an implicit identity or reference conversion to an interface or delegate type T_0 and T_0 is variance-convertible (§18.2.3.3) to `T`.
- Implicit conversions involving type parameters that are known to be reference types. See §10.2.12 for more details on implicit conversions involving type parameters.

The implicit reference conversions are those conversions between *reference_type*s that can be proven to always succeed, and therefore require no checks at run-time.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted.

> *Note*: In other words, while a reference conversion can change the type of the reference, it never changes the type or value of the object being referred to. *end note*

10.2.9 Boxing conversions

A boxing conversion permits a *value_type* to be implicitly converted to a *reference_type*. The following boxing conversions exist:

- From any *value_type* to the type `object`.

- From any *value_type* to the type `System.ValueType`.
- From any *enum_type* to the type `System.Enum`.
- From any *non_nullable_value_type* to any *interface_type* implemented by the *non_nullable_value_type*.
- From any *non_nullable_value_type* to any *interface_type* `I` such that there is a boxing conversion from the *non_nullable_value_type* to another *interface_type* I_0, and I_0 has an identity conversion to `I`.
- From any *non_nullable_value_type* to any *interface_type* `I` such that there is a boxing conversion from the *non_nullable_value_type* to another *interface_type* I_0, and I_0 is variance-convertible (§18.2.3.3) to `I`.
- From any *nullable_value_type* to any *reference_type* where there is a boxing conversion from the underlying type of the *nullable_value_type* to the *reference_type*.
- From a type parameter that is not known to be a reference type to any type such that the conversion is permitted by §10.2.12.

Boxing a value of a *non-nullable-value-type* consists of allocating an object instance and copying the value into that instance.

Boxing a value of a *nullable_value_type* produces a null reference if it is the null value (`HasValue` is false), or the result of unwrapping and boxing the underlying value otherwise.

> *Note*: The process of boxing may be imagined in terms of the existence of a boxing class for every value type. For example, consider a `struct S` implementing an interface `I`, with a boxing class called `S_Boxing`.

```
C#

interface I
{
    void M();
}

struct S : I
{
    public void M() { ... }
}

sealed class S_Boxing : I
{
    S value;

    public S_Boxing(S value)
    {
```

```
        this.value = value;
    }

    public void M()
    {
        value.M();
    }
}
```

Boxing a value `v` of type `S` now consists of executing the expression `new S_Boxing(v)` and returning the resulting instance as a value of the target type of the conversion. Thus, the statements

C#

```
S s = new S();
object box = s;
```

can be thought of as similar to:

C#

```
S s = new S();
object box = new S_Boxing(s);
```

The imagined boxing type described above does not actually exist. Instead, a boxed value of type `S` has the runtime type `S`, and a runtime type check using the `is` operator with a value type as the right operand tests whether the left operand is a boxed version of the right operand. For example,

C#

```
int i = 123;
object box = i;
if (box is int)
{
    Console.Write("Box contains an int");
}
```

will output the following:

Console

```
Box contains an int
```

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a *reference_type* to type `object`, in which the value continues to reference the same instance and simply is regarded as the less derived type `object`. For example, the following

C#

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

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

class A
{
    void M()
    {
        Point p = new Point(10, 10);
        object box = p;
        p.x = 20;
        Console.Write(((Point)box).x);
    }
}
```

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20 would be output because `p` and `box` would reference the same instance.

The analogy of a boxing class should not be used as more than a helpful tool for picturing how boxing works conceptually. There are numerous subtle differences between the behavior described by this specification and the behavior that would result from boxing being implemented in precisely this manner.

end note

10.2.10 Implicit dynamic conversions

An implicit dynamic conversion exists from an expression of type dynamic to any type `T`. The conversion is dynamically bound §12.3.3, which means that an implicit

conversion will be sought at run-time from the run-time type of the expression to `T`. If no conversion is found, a run-time exception is thrown.

This implicit conversion seemingly violates the advice in the beginning of §10.2 that an implicit conversion should never cause an exception. However, it is not the conversion itself, but the *finding* of the conversion that causes the exception. The risk of run-time exceptions is inherent in the use of dynamic binding. If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

> *Example*: The following illustrates implicit dynamic conversions:
>
> ```
> C#
> ```
>
> ```
> object o = "object";
> dynamic d = "dynamic";
> string s1 = o;          // Fails at compile-time – no conversion
> exists
> string s2 = d;          // Compiles and succeeds at run-time
> int i = d;              // Compiles but fails at run-time – no con-
> version exists
> ```
>
> The assignments to `s2` and `i` both employ implicit dynamic conversions, where the binding of the operations is suspended until run-time. At run-time, implicit conversions are sought from the run-time type of `d` (`string`) to the target type. A conversion is found to `string` but not to `int`.
>
> *end example*

10.2.11 Implicit constant expression conversions

An implicit constant expression conversion permits the following conversions:

- A *constant_expression* (§12.23) of type `int` can be converted to type `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the *constant_expression* is within the range of the destination type.
- A *constant_expression* of type `long` can be converted to type `ulong`, provided the value of the *constant_expression* is not negative.

10.2.12 Implicit conversions involving type parameters

For a *type_parameter* `T` that is known to be a reference type (§15.2.5), the following implicit reference conversions (§10.2.8) exist:

- From `T` to its effective base class `C`, from `T` to any base class of `C`, and from `T` to any interface implemented by `C`.
- From `T` to an *interface_type* `I` in `T`'s effective interface set and from `T` to any base interface of `I`.
- From `T` to a type parameter `U` provided that `T` depends on `U` (§15.2.5).

> *Note*: Since `T` is known to be a reference type, within the scope of `T`, the run-time type of `U` will always be a reference type, even if `U` is not known to be a reference type at compile-time. *end note*

- From the null literal (§6.4.5.7) to T.

For a *type_parameter* `T` that is *not* known to be a reference type §15.2.5, the following conversions involving `T` are considered to be boxing conversions (§10.2.9) at compile-time. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. At run-time, if `T` is a reference type, the conversion is executed as an implicit reference conversion or identity conversion.

- From `T` to its effective base class `C`, from `T` to any base class of `C`, and from `T` to any interface implemented by `C`.

> *Note*: `C` will be one of the types `System.Object`, `System.ValueType`, or `System.Enum` (otherwise `T` would be known to be a reference type). *end note*

- From `T` to an *interface_type* `I` in `T`'s effective interface set and from `T` to any base interface of `I`.

For a *type_parameter* `T` that is *not* known to be a reference type, there is an implicit conversion from `T` to a type parameter `U` provided `T` depends on `U`. At run-time, if `T` is a value type and `U` is a reference type, the conversion is executed as a boxing conversion. At run-time, if both `T` and `U` are value types, then `T` and `U` are necessarily the same type and no conversion is performed. At run-time, if `T` is a reference type, then `U` is necessarily also a reference type and the conversion is executed as an implicit reference conversion or identity conversion (§15.2.5).

The following further implicit conversions exist for a given type parameter `T`:

- From `T` to a reference type `S` if it has an implicit conversion to a reference type S_0 and S_0 has an identity conversion to `S`. At run-time, the conversion is executed the same way as the conversion to S_0.

- From `T` to an interface type `I` if it has an implicit conversion to an interface type I_0, and I_0 is variance-convertible to `I` (§18.2.3.3). At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

In all cases, the rules ensure that a conversion is executed as a boxing conversion if and only if at run-time the conversion is from a value type to a reference type.

10.2.13 Implicit tuple conversions

An implicit conversion exists from a tuple expression `E` to a tuple type `T` if `E` has the same arity as `T` and an implicit conversion exists from each element in `E` to the corresponding element type in `T`. The conversion is performed by creating an instance of `T`'s corresponding `System.ValueTuple<...>` type, and initializing each of its fields in order from left to right by evaluating the corresponding tuple element expression of `E`, converting it to the corresponding element type of `T` using the implicit conversion found, and initializing the field with the result.

If an element name in the tuple expression does not match a corresponding element name in the tuple type, a warning shall be issued.

> *Example*:
>
> C#
>
> ```
> (int, string) t1 = (1, "One");
> (byte, string) t2 = (2, null);
> (int, string) t3 = (null, null);        // Error: No conversion
> (int i, string s) t4 = (i: 4, "Four");
> (int i, string) t5 = (x: 5, s: "Five"); // Warning: Names are
> ignored
> ```
>
> The declarations of `t1`, `t2`, `t4` and `t5` are all valid, since implicit conversions exist from the element expressions to the corresponding element types. The declaration of `t3` is invalid, because there is no conversion from `null` to `int`. The declaration of `t5` causes a warning because the element names in the tuple expression differs from those in the tuple type.
>
> *end example*

10.2.14 User-defined implicit conversions

A user-defined implicit conversion consists of an optional standard implicit conversion, followed by execution of a user-defined implicit conversion operator, followed by another optional standard implicit conversion. The exact rules for evaluating user-defined implicit conversions are described in §10.5.4.

10.2.15 Anonymous function conversions and method group conversions

Anonymous functions and method groups do not have types in and of themselves, but they may be implicitly converted to delegate types. Additionally, some lambda expressions may be implicitly converted to expression tree types. Anonymous function conversions are described in more detail in §10.7 and method group conversions in §10.8.

10.2.16 Default literal conversions

An implicit conversion exists from a *default_literal* (§12.8.20) to any type. This conversion produces the default value (§9.3) of the inferred type.

10.2.17 Implicit throw conversions

While throw expressions do not have a type, they may be implicitly converted to any type.

10.3 Explicit conversions

10.3.1 General

The following conversions are classified as explicit conversions:

- All implicit conversions
- Explicit numeric conversions
- Explicit enumeration conversions
- Explicit nullable conversions
- Explicit tuple conversions
- Explicit reference conversions
- Explicit interface conversions
- Unboxing conversions
- Explicit type parameter conversions
- Explicit dynamic conversions

- User-defined explicit conversions

Explicit conversions can occur in cast expressions (§12.9.7).

The set of explicit conversions includes all implicit conversions.

> *Note*: This, for example, allows an explicit cast to be used when an implicit conversion to the same type exists, in order to force the selection of a particular method overload. *end note*

The explicit conversions that are not implicit conversions are conversions that cannot be proven always to succeed, conversions that are known possibly to lose information, and conversions across domains of types sufficiently different to merit explicit notation.

10.3.2 Explicit numeric conversions

The explicit numeric conversions are the conversions from a *numeric_type* to another *numeric_type* for which an implicit numeric conversion (§10.2.3) does not already exist:

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

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any *numeric_type* to any other *numeric_type* using a cast expression (§12.9.7).

The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An explicit numeric conversion is processed as follows:

- For a conversion from an integral type to another integral type, the processing depends on the overflow checking context (§12.8.19) in which the conversion takes place:
  - In a `checked` context, the conversion succeeds if the value of the source operand is within the range of the destination type, but throws a `System.OverflowException` if the value of the source operand is outside the range of the destination type.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
    - If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
    - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type
- For a conversion from `decimal` to an integral type, the source value is rounded towards zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, a `System.OverflowException` is thrown.
- For a conversion from `float` or `double` to an integral type, the processing depends on the overflow-checking context (§12.8.19) in which the conversion takes place:
  - In a checked context, the conversion proceeds as follows:
    - If the value of the operand is NaN or infinite, a `System.OverflowException` is thrown.
    - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.
    - Otherwise, a `System.OverflowException` is thrown.
  - In an unchecked context, the conversion always succeeds, and proceeds as follows.
    - If the value of the operand is NaN or infinite, the result of the conversion is an unspecified value of the destination type.
    - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.
    - Otherwise, the result of the conversion is an unspecified value of the destination type.
- For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small to represent as a `float`, the result becomes zero with the same sign as the value. If the magnitude of the

`double` value is too large to represent as a `float`, the result becomes infinity with the same sign as the value. If the `double` value is NaN, the result is also NaN.

- For a conversion from `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number if required (§8.3.8).
  - If the source value is too small to represent as a `decimal`, the result becomes zero, preserving the sign of the original value if `decimal` supports signed zero values.
  - If the source value's magnitude is too large to represent as a `decimal`, or that value is infinity, the result is infinity preserving the sign of the original value, if the decimal representation supports infinities; otherwise a System.OverflowException is thrown.
  - If the source value is NaN, the result is NaN if the decimal representation supports NaNs; otherwise a System.OverflowException is thrown.
- For a conversion from `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest `double` or `float` value. If the source value's magnitude is too large to represent in the target type, or that value is infinity, the result is infinity preserving the sign of the original value. If the source value is NaN, the result is NaN. While this conversion may lose precision, it never causes an exception to be thrown.

> *Note*: The `decimal` type is not required to support infinities or NaN values but may do so; its range may be smaller than the range of `float` and `double`, but is not guaranteed to be. For `decimal` representations without infinities or NaN values, and with a range smaller than `float`, the result of a conversion from `decimal` to either `float` or `double` will never be infinity or NaN. *end note*

10.3.3 Explicit enumeration conversions

The explicit enumeration conversions are:

- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal` to any *enum_type*.
- From any *enum_type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal`.
- From any *enum_type* to any other *enum_type*.

An explicit enumeration conversion between two types is processed by treating any participating *enum_type* as the underlying type of that *enum_type*, and then performing an implicit or explicit numeric conversion between the resulting types.

> *Example*: Given an *enum_type* `E` with and underlying type of `int`, a conversion from `E` to `byte` is processed as an explicit numeric conversion (§10.3.2) from `int` to `byte`, and a conversion from `byte` to `E` is processed as an implicit numeric conversion (§10.2.3) from `byte` to `int`. *end example*

10.3.4 Explicit nullable conversions

The explicit nullable conversions are those nullable conversions (§10.6.1) derived from explicit and implicit predefined conversions.

10.3.5 Explicit reference conversions

The explicit reference conversions are:

- From object and dynamic to any other *reference_type*.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is a base class of `T`.
- From any *class_type* `S` to any *interface_type* `T`, provided `S` is not sealed and provided `S` does not implement `T`.
- From any *interface_type* `S` to any *class_type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is not derived from `T`.
- From an *array_type* `S` with an element type S_i to an *array_type* `T` with an element type T_i, provided all of the following are true:
  - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
  - An explicit reference conversion exists from S_i to T_i.
- From `System.Array` and the interfaces it implements, to any *array_type*.
- From a single-dimensional *array_type* `S[]` to `System.Collections.Generic.IList<T>`, `System.Collections.Generic.IReadOnlyList<T>`, and its base interfaces, provided that there is an identity conversion or explicit reference conversion from `S` to `T`.
- From `System.Collections.Generic.IList<S>`, `System.Collections.Generic.IReadOnlyList<S>`, and their base interfaces to a single-dimensional array type `T[]`, provided that there is an identity conversion or explicit reference conversion from `S` to T.
- From `System.Delegate` and the interfaces it implements to any *delegate_type*.

- From a reference type `S` to a reference type `T` if it has an explicit reference conversion from `S` to a reference type T_0 and T_0 and there is an identity conversion from T_0 to `T`.
- From a reference type `S` to an interface or delegate type `T` if it there is an explicit reference conversion from `S` to an interface or delegate type T_0 and either T_0 is variance-convertible to `T` or `T` is variance-convertible to T_0 §18.2.3.3.
- From $D<S_1...S_v>$ to $D<T_1...T_v>$ where $D<X_1...X_v>$ is a generic delegate type, $D<S_1...S_v>$ is not compatible with or identical to $D<T_1...T_v>$, and for each type parameter X_i of `D` the following holds:
  - If X_i is invariant, then S_i is identical to T_i.
  - If X_i is covariant, then there is an identity conversion, implicit reference conversion or explicit reference conversion from S_i to T_i.
  - If X_i is contravariant, then S_i and T_i are either identical or both reference types.
- Explicit conversions involving type parameters that are known to be reference types. For more details on explicit conversions involving type parameters, see §10.3.9.

The explicit reference conversions are those conversions between *reference_type*s that require run-time checks to ensure they are correct.

For an explicit reference conversion to succeed at run-time, the value of the source operand shall be `null`, or the type of the object referenced by the source operand shall be a type that can be converted to the destination type by an implicit reference conversion (§10.2.8). If an explicit reference conversion fails, a `System.InvalidCastException` is thrown.

> *Note*: Reference conversions, implicit or explicit, never change the value of the reference itself (§8.2.1), only its type; neither does it change the type or value of the object being referenced. *end note*

10.3.6 Explicit tuple conversions

An explicit conversion exists from a tuple expression `E` to a tuple type `T` if `E` has the same arity as `T` and an implicit or explicit conversion exists from each element in `E` to the corresponding element type in `T`. The conversion is performed by creating an instance of `T`'s corresponding `System.ValueTuple<...>` type, and initializing each of its fields in order from left to right by evaluating the corresponding tuple element expression of `E`, converting it to the corresponding element type of `T` using the explicit conversion found, and initializing the field with the result.

10.3.7 Unboxing conversions

An unboxing conversion permits a *reference_type* to be explicitly converted to a *value_type*. The following unboxing conversions exist:

- From the type `object` to any *value_type*.
- From the type `System.ValueType` to any *value_type*.
- From the type `System.Enum` to any *enum_type*.
- From any *interface_type* to any *non-nullable_value_type* that implements the *interface_type*.
- From any *interface_type* `I` to any *non_nullable_value_type* where there is an unboxing conversion from an *interface_type* I_0 to the *non_nullable_value-type* and an identity conversion from `I` to I_0.
- From any *interface_type* `I` to any *non_nullable_value_type* where there is an unboxing conversion from an *interface_type* I_0 to the *non_nullable_value_type* and either either I_0 is variance_convertible to `I` or `I` is variance-convertible to I_0 (§18.2.3.3).
- From any *reference_type* to any *nullable_value_type* where there is an unboxing conversion from *reference_type* to the underlying *non_nullable_value_type* of the *nullable_value_type*.
- From a type parameter which is not known to be a value type to any type such that the conversion is permitted by §10.3.9.

An unboxing operation to a *non_nullable_value_type* consists of first checking that the object instance is a boxed value of the given *non_nullable_value_type*, and then copying the value out of the instance.

Unboxing to a *nullable_value_type* produces the null value of the *nullable_value_type* if the source operand is `null`, or the wrapped result of unboxing the object instance to the underlying type of the *nullable_value_type* otherwise.

> *Note*: Referring to the imaginary boxing class described in §10.2.9, an unboxing conversion of an object box to a *value_type* `S` consists of executing the expression `((S_Boxing)box).value`. Thus, the statements
>
> ```
> C#
> ```
>
> ```
> object box = new S();
> S s = (S)box;
> ```
>
> conceptually correspond to

```
C#
```

```
object box = new S_Boxing(new S());
S s = ((S_Boxing)box).value;
```

end note

For an unboxing conversion to a given *non_nullable_value_type* to succeed at run-time, the value of the source operand shall be a reference to a boxed value of that *non_nullable_value_type*. If the source operand is `null` a `System.NullReferenceException` is thrown. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

For an unboxing conversion to a given *nullable_value_type* to succeed at run-time, the value of the source operand shall be either null or a reference to a boxed value of the underlying *non_nullable_value_type* of the *nullable_value_type*. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

10.3.8 Explicit dynamic conversions

An explicit dynamic conversion exists from an expression of type `dynamic` to any type `T`. The conversion is dynamically bound (§12.3.3), which means that an explicit conversion will be sought at run-time from the run-time type of the expression to `T`. If no conversion is found, a run-time exception is thrown.

If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

Example: Assume the following class is defined:

```
C#
```

```
class C
{
    int i;

    public C(int i)
    {
        this.i = i;
    }

    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
```

```
    }
}
```

The following illustrates explicit dynamic conversions:

C#

```
object o = "1";
dynamic d = "2";
var c1 = (C)o; // Compiles, but explicit reference conversion
fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

The best conversion of `o` to `C` is found at compile-time to be an explicit reference conversion. This fails at run-time, because `"1"` is not in fact a `C`. The conversion of `d` to `C` however, as an explicit dynamic conversion, is suspended to run-time, where a user defined conversion from the run-time type of `d` (`string`) to `C` is found, and succeeds.

end example

10.3.9 Explicit conversions involving type parameters

For a *type_parameter* `T` that is known to be a reference type (§15.2.5), the following explicit reference conversions (§10.3.5) exist:

- From the effective base class `C` of `T` to `T` and from any base class of `C` to `T`.
- From any *interface_type* to `T`.
- From `T` to any *interface_type* `I` provided there isn't already an implicit reference conversion from `T` to `I`.
- From a *type_parameter* `U` to `T` provided that `T` depends on `U` (§15.2.5).

> *Note*: Since `T` is known to be a reference type, within the scope of `T`, the run-time type of U will always be a reference type, even if `U` is not known to be a reference type at compile-time. *end note*

For a *type_parameter* `T` that is *not* known to be a reference type (§15.2.5), the following conversions involving `T` are considered to be unboxing conversions (§10.3.7) at compile-time. At run-time, if `T` is a value type, the conversion is executed as an unboxing conversion. At run-time, if `T` is a reference type, the conversion is executed as an explicit reference conversion or identity conversion.

- From the effective base class `C` of `T` to `T` and from any base class of `C` to `T`.

  > *Note*: C will be one of the types `System.Object`, `System.ValueType`, or `System.Enum` (otherwise `T` would be known to be a reference type). *end note*

- From any *interface_type* to `T`.

For a *type_parameter* `T` that is *not* known to be a reference type (§15.2.5), the following explicit conversions exist:

- From `T` to any *interface_type* `I` provided there is not already an implicit conversion from `T` to `I`. This conversion consists of an implicit boxing conversion (§10.2.9) from `T` to `object` followed by an explicit reference conversion from `object` to `I`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion followed by an explicit reference conversion. At run-time, if `T` is a reference type, the conversion is executed as an explicit reference conversion.
- From a type parameter `U` to `T` provided that `T` depends on `U` (§15.2.5). At run-time, if `T` is a value type and `U` is a reference type, the conversion is executed as an unboxing conversion. At run-time, if both `T` and `U` are value types, then `T` and `U` are necessarily the same type and no conversion is performed. At run-time, if `T` is a reference type, then `U` is necessarily also a reference type and the conversion is executed as an explicit reference conversion or identity conversion.

In all cases, the rules ensure that a conversion is executed as an unboxing conversion if and only if at run-time the conversion is from a reference type to a value type.

The above rules do not permit a direct explicit conversion from an unconstrained type parameter to a non-interface type, which might be surprising. The reason for this rule is to prevent confusion and make the semantics of such conversions clear.

> *Example*: Consider the following declaration:

```C#
class X<T>
{
    public static long F(T t)
    {
        return (long)t;         // Error
    }
}
```

If the direct explicit conversion of `t` to `long` were permitted, one might easily expect that `X<int>.F(7)` would return `7L`. However, it would not, because the standard numeric conversions are only considered when the types are known to be numeric at binding-time. In order to make the semantics clear, the above example must instead be written:

```
C#
```

```
class X<T>
{
    public static long F(T t)
    {
        return (long)(object)t;         // Ok, but will only work
when T is long
    }
}
```

This code will now compile but executing `X<int>.F(7)` would then throw an exception at run-time, since a boxed `int` cannot be converted directly to a `long`.

end example

10.3.10 User-defined explicit conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion. The exact rules for evaluating user-defined explicit conversions are described in §10.5.5.

10.4 Standard conversions

10.4.1 General

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

10.4.2 Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions (§10.2.2)
- Implicit numeric conversions (§10.2.3)

- Implicit nullable conversions (§10.2.6)
- Null literal conversions (§10.2.7)
- Implicit reference conversions (§10.2.8)
- Boxing conversions (§10.2.9)
- Implicit constant expression conversions (§10.2.11)
- Implicit conversions involving type parameters (§10.2.12)

The standard implicit conversions specifically exclude user-defined implicit conversions.

10.4.3 Standard explicit conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists.

> *Note*: In other words, if a standard implicit conversion exists from a type `A` to a type `B`, then a standard explicit conversion exists from type `A` to type `B` and from type `B` to type `A`. *end note*

10.5 User-defined conversions

10.5.1 General

C# allows the pre-defined implicit and explicit conversions to be augmented by user-defined conversions. User-defined conversions are introduced by declaring conversion operators (§15.10.4) in class and struct types.

10.5.2 Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion.

For a given source type `S` and target type `T`, if `S` or `T` are nullable value types, let S_0 and T_0 refer to their underlying types, otherwise S_0 and T_0 are equal to `S` and `T` respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- S_0 and T_0 are different types.
- Either S_0 or T_0 is the class or struct type in which the operator declaration takes place.
- Neither S_0 nor T_0 is an *interface_type*.

- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

The restrictions that apply to user-defined conversions are specified in §15.10.4.

10.5.3 Evaluation of user-defined conversions

A user-defined conversion converts a ***source expression***, which may have a **source type**, to another type, called the ***target type***. Evaluation of a user-defined conversion centers on finding the ***most-specific*** user-defined conversion operator for the source expression and target type. This determination is broken into several steps:

- Finding the set of classes and structs from which user-defined conversion operators will be considered. This set consists of the source type and its base classes, if the source type exists, along with the target type and its base classes. For this purpose it is assumed that only classes and structs can declare user-defined operators, and that non-class types have no base classes. Also, if either the source or target type is a nullable-value-type, their underlying type is used instead.
- From that set of types, determining which user-defined and lifted conversion operators are applicable. For a conversion operator to be applicable, it shall be possible to perform a standard conversion (§10.4) from the source expression to the operand type of the operator, and it shall be possible to perform a standard conversion from the result type of the operator to the target type.
- From the set of applicable user-defined operators, determining which operator is unambiguously the most-specific. In general terms, the most-specific operator is the operator whose operand type is "closest" to the source expression and whose result type is "closest" to the target type. User-defined conversion operators are preferred over lifted conversion operators. The exact rules for establishing the most-specific user-defined conversion operator are defined in the following subclauses.

Once a most-specific user-defined conversion operator has been identified, the actual execution of the user-defined conversion involves up to three steps:

- First, if required, performing a standard conversion from the source expression to the operand type of the user-defined or lifted conversion operator.
- Next, invoking the user-defined or lifted conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined or lifted conversion operator. In other words, a conversion from type `S` to type `T` will never first execute a user-defined conversion from `S` to `X` and then execute a user-defined conversion from `X` to `T`.

- Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following subclauses. The definitions make use of the following terms:
- If a standard implicit conversion (§10.4.2) exists from a type `A` to a type `B`, and if neither `A` nor `B` are *interface_type* `s`, then `A` is said to be ***encompassed by*** `B`, and `B` is said to ***encompass*** `A`.
- If a standard implicit conversion (§10.4.2) exists from an expression `E` to a type `B`, and if neither `B` nor the type of `E` (if it has one) are *interface_type* `s`, then `E` is said to be *encompassed by* `B`, and `B` is said to *encompass* `E`.
- The ***most-encompassing type*** in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most-encompassing type. In more intuitive terms, the most-encompassing type is the "largest" type in the set—the one type to which each of the other types can be implicitly converted.
- The ***most-encompassed type*** in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most-encompassed type. In more intuitive terms, the most-encompassed type is the "smallest" type in the set—the one type that can be implicitly converted to each of the other types.

10.5.4 User-defined implicit conversions

A user-defined implicit conversion from an expression `E` to a type `T` is processed as follows:

- Determine the types `S`, S_0 and T_0.
  - If `E` has a type, let `S` be that type.
  - If `S` or `T` are nullable value types, let S_i and T_i be their underlying types, otherwise let S_i and T_i be `S` and `T`, respectively.
  - If S_i or T_i are type parameters, let S_0 and T_0 be their effective base classes, otherwise let S_0 and T_0 be S_x and T_i, respectively.

- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of S_0 (if S_0 exists and is a class or struct), the base classes of S_0 (if S_0 exists and is a class), and T_0 (if T_0 is a class or struct). A type

is added to the set `D` only if an identity conversion to another type already included in the set doesn't exist.

- Find the set of applicable user-defined and lifted conversion operators, `U`. This set consists of the user-defined and lifted implicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing `E` to a type encompassed by `T`. If `U` is empty, the conversion is undefined and a compile-time error occurs.
  - If `S` exists and any of the operators in `U` convert from `S`, then S_x is `S`.
  - Otherwise, S_x is the most-encompassed type in the combined set of source types of the operators in `U`. If exactly one most-encompassed type cannot be found, then the conversion is ambiguous and a compile-time error occurs.

- Find the most-specific target type, T_x, of the operators in `U`:
  - If any of the operators in `U` convert to `T`, then T_x is `T`.
  - Otherwise, T_x is the most-encompassing type in the combined set of target types of the operators in `U`. If exactly one most-encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.

- Find the most-specific conversion operator:
  - If `U` contains exactly one user-defined conversion operator that converts from S_x to T_x, then this is the most-specific conversion operator.
  - Otherwise, if `U` contains exactly one lifted conversion operator that converts from S_x to T_x, then this is the most-specific conversion operator.
  - Otherwise, the conversion is ambiguous and a compile-time error occurs.

- Finally, apply the conversion:
  - If E does not already have the type S_x, then a standard implicit conversion from `E` to S_x is performed.
  - The most-specific conversion operator is invoked to convert from S_x to T_x.
  - If T_x is not `T`, then a standard implicit conversion from T_x to `T` is performed.

A user-defined implicit conversion from a type `S` to a type `T` exists if a user-defined implicit conversion exists from a variable of type `S` to `T`.

10.5.5 User-defined explicit conversions

A user-defined explicit conversion from an expression `E` to a type `T` is processed as follows:

- Determine the types `S`, S_0 and T_0.

    - If `E` has a type, let `S` be that type.
    - If `S` or `T` are nullable value types, let S_i and T_i be their underlying types, otherwise let S_i and T_i be `S` and `T`, respectively.
    - If S_i or T_i are type parameters, let S_0 and T_0 be their effective base classes, otherwise let S_0 and T_0 be S_i and T_i, respectively.
- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of S_0 (if S_0 exists and is a class or struct), the base classes of S_0 (if S_0 exists and is a class), T_0 (if T_0 is a class or struct), and the base classes of T_0 (if T_0 is a class). `A` type is added to the set `D` only if an identity conversion to another type already included in the set doesn't exist.
- Find the set of applicable user-defined and lifted conversion operators, `U`. This set consists of the user-defined and lifted implicit or explicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing `E` or encompassed by `S` (if it exists) to a type encompassing or encompassed by `T`. If `U` is empty, the conversion is undefined and a compile-time error occurs.
- Find the most-specific source type, S_x, of the operators in `U`:
    - If S exists and any of the operators in `U` convert from `S`, then S_x is `S`.
    - Otherwise, if any of the operators in `U` convert from types that encompass `E`, then S_x is the most-encompassed type in the combined set of source types of those operators. If no most-encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
    - Otherwise, S_x is the most-encompassing type in the combined set of source types of the operators in `U`. If exactly one most-encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most-specific target type, T_x, of the operators in `U`:
    - If any of the operators in `U` convert to `T`, then T_x is `T`.
    - Otherwise, if any of the operators in `U` convert to types that are encompassed by `T`, then T_x is the most-encompassing type in the combined set of target types of those operators. If exactly one most-encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
    - Otherwise, T_x is the most-encompassed type in the combined set of target types of the operators in `U`. If no most-encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most-specific conversion operator:
    - If U contains exactly one user-defined conversion operator that converts from S_x to T_x, then this is the most-specific conversion operator.
    - Otherwise, if `U` contains exactly one lifted conversion operator that converts from S_x to T_x, then this is the most-specific conversion operator.

    - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
    - If `E` does not already have the type S_x, then a standard explicit conversion from E to S_x is performed.
    - The most-specific user-defined conversion operator is invoked to convert from S_x to T_x.
    - If T_x is not `T`, then a standard explicit conversion from T_x to `T` is performed.

A user-defined explicit conversion from a type `S` to a type `T` exists if a user-defined explicit conversion exists from a variable of type `S` to `T`.

10.6 Conversions involving nullable types

10.6.1 Nullable Conversions

Nullable conversions permit predefined conversions that operate on non-nullable value types to also be used with nullable forms of those types. For each of the predefined implicit or explicit conversions that convert from a non-nullable value type `S` to a non-nullable value type `T` (§10.2.2, §10.2.3, §10.2.4, §10.2.11, §10.3.2 and §10.3.3), the following nullable conversions exist:

- An implicit or explicit conversion from `S?` to `T?`
- An implicit or explicit conversion from `S` to `T?`
- An explicit conversion from `S?` to `T`.

A nullable conversion is itself classified as an implicit or explicit conversion.

Certain nullable conversions are classified as standard conversions and can occur as part of a user-defined conversion. Specifically, all implicit nullable conversions are classified as standard implicit conversions (§10.4.2), and those explicit nullable conversions that satisfy the requirements of §10.4.3 are classified as standard explicit conversions.

Evaluation of a nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
    - If the source value is null (`HasValue` property is `false`), the result is the null value of type `T?`.
    - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying conversion from `S` to `T`, followed by a wrapping from `T` to `T?`.

- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T` followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S?` to `T`, the conversion is evaluated as an unwrapping from `S?` to `S` followed by the underlying conversion from `S` to `T`.

10.6.2 Lifted conversions

Given a user-defined conversion operator that converts from a non-nullable value type `S` to a non-nullable value type `T`, a ***lifted conversion operator*** exists that converts from `S?` to `T?`. This lifted conversion operator performs an unwrapping from `S?` to `S` followed by the user-defined conversion from `S` to `T` followed by a wrapping from `T` to `T?`, except that a null valued `S?` converts directly to a null valued `T?`. A lifted conversion operator has the same implicit or explicit classification as its underlying user-defined conversion operator.

10.7 Anonymous function conversions

10.7.1 General

An *anonymous_method_expression* or *lambda_expression* is classified as an anonymous function (§12.19). The expression does not have a type, but can be implicitly converted to a compatible delegate type. Some lambda expressions may also be implicitly converted to a compatible expression tree type.

Specifically, an anonymous function `F` is compatible with a delegate type `D` provided:

- If `F` contains an *anonymous_function_signature*, then `D` and `F` have the same number of parameters.
- If `F` does not contain an *anonymous_function_signature*, then `D` may have zero or more parameters of any type, as long as no parameter of `D` has the out parameter modifier.
- If `F` has an explicitly typed parameter list, each parameter in `D` has the same type and modifiers as the corresponding parameter in `F`.
- If `F` has an implicitly typed parameter list, `D` has no ref or out parameters.
- If the body of `F` is an expression, and *either* `D` has a void return type *or* `F` is async and `D` has a `«TaskType»` return type (§15.15.1), then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (w.r.t §12) that would be permitted as a *statement_expression* (§13.7).

- If the body of `F` is a block, and *either* `D` has a void return type *or* `F` is async and `D` has a `«TaskType»` return type , then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid block (w.r.t §13.3) in which no `return` statement specifies an expression.
- If the body of `F` is an expression, and *either* `F` is non-async and `D` has a non-`void` return type `T`, *or* `F` is async and `D` has a `«TaskType»<T>` return type (§15.15.1), then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (w.r.t §12) that is implicitly convertible to `T`.
- If the body of `F` is a block, and *either* `F` is non-async and `D` has a non-void return type `T`, *or* `F` is async and `D` has a `«TaskType»<T>` return type, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (w.r.t §13.3) with a non-reachable end point in which each return statement specifies an expression that is implicitly convertible to `T`.

Example: The following examples illustrate these rules:

C#

```
delegate void D(int x);
D d1 = delegate { };                        // Ok
D d2 = delegate() { };                      // Error, signature
mismatch
D d3 = delegate(long x) { };                // Error, signature
mismatch
D d4 = delegate(int x) { };                 // Ok
D d5 = delegate(int x) { return; };         // Ok
D d6 = delegate(int x) { return x; };       // Error, return type
mismatch

delegate void E(out int x);
E e1 = delegate { };                        // Error, E has an
out parameter
E e2 = delegate(out int x) { x = 1; };      // Ok
E e3 = delegate(ref int x) { x = 1; };      // Error, signature
mismatch

delegate int P(params int[] a);
P p1 = delegate { };                        // Error, end of
block reachable
P p2 = delegate { return; };                // Error, return type
mismatch
P p3 = delegate { return 1; };              // Ok
P p4 = delegate { return "Hello"; };        // Error, return type
mismatch
P p5 = delegate(int[] a)                    // Ok
```

```
{
    return a[0];
};
P p6 = delegate(params int[] a)              // Error, params mod-
ifier
{
    return a[0];
};
P p7 = delegate(int[] a)                     // Error, return type
mismatch
{
    if (a.Length > 0) return a[0];
    return "Hello";
};

delegate object Q(params int[] a);
Q q1 = delegate(int[] a)                    // Ok
{
    if (a.Length > 0) return a[0];
    return "Hello";
};
```

end example

Example: The examples that follow use a generic delegate type `Func<A,R>` that represents a function that takes an argument of type `A` and returns a value of type `R`:

C#

```
delegate R Func<A,R>(A arg);
```

In the assignments

C#

```
Func<int,int> f1 = x => x + 1; // Ok
Func<int,double> f2 = x => x + 1; // Ok
Func<double,int> f3 = x => x + 1; // Error
Func<int, Task<int>> f4 = async x => x + 1; // Ok
```

the parameter and return types of each anonymous function are determined from the type of the variable to which the anonymous function is assigned.

The first assignment successfully converts the anonymous function to the delegate type `Func<int,int>` because, when `x` is given type `int`, `x + 1` is a valid expression that is implicitly convertible to type `int`.

> Likewise, the second assignment successfully converts the anonymous function to the delegate type Func<int,double> because the result of `x + 1` (of type `int`) is implicitly convertible to type `double`.
>
> However, the third assignment is a compile-time error because, when `x` is given type `double`, the result of `x + 1` (of type `double`) is not implicitly convertible to type `int`.
>
> The fourth assignment successfully converts the anonymous async function to the delegate type `Func<int, Task<int>>` because the result of `x + 1` (of type `int`) is implicitly convertible to the effective return type `int` of the async lambda, which has a return type `Task<int>`.
>
> *end example*

A lambda expression `F` is compatible with an expression tree type `Expression<D>` if `F` is compatible with the delegate type `D`. This does not apply to anonymous methods, only lambda expressions.

Anonymous functions may influence overload resolution, and participate in type inference. See §12.6 for further details.

10.7.2 Evaluation of anonymous function conversions to delegate types

Conversion of an anonymous function to a delegate type produces a delegate instance that references the anonymous function and the (possibly empty) set of captured outer variables that are active at the time of the evaluation. When the delegate is invoked, the body of the anonymous function is executed. The code in the body is executed using the set of captured outer variables referenced by the delegate. A *delegate_creation_expression* (§12.8.16.6) can be used as an alternate syntax for converting an anonymous method to a delegate type.

The invocation list of a delegate produced from an anonymous function contains a single entry. The exact target object and target method of the delegate are unspecified. In particular, it is unspecified whether the target object of the delegate is `null`, the `this` value of the enclosing function member, or some other object.

Conversions of semantically identical anonymous functions with the same (possibly empty) set of captured outer variable instances to the same delegate types are permitted (but not required) to return the same delegate instance. The term

semantically identical is used here to mean that execution of the anonymous functions will, in all cases, produce the same effects given the same arguments. This rule permits code such as the following to be optimized.

C#

```
delegate double Function(double x);

class Test
{
    static double[] Apply(double[] a, Function f)
    {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++)
        {
            result[i] = f(a[i]);
        }
        return result;
    }

    static void F(double[] a, double[] b)
    {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```

Since the two anonymous function delegates have the same (empty) set of captured outer variables, and since the anonymous functions are semantically identical, the compiler is permitted to have the delegates refer to the same target method. Indeed, the compiler is permitted to return the very same delegate instance from both anonymous function expressions.

10.7.3 Evaluation of lambda expression conversions to expression tree types

Conversion of a lambda expression to an expression tree type produces an expression tree (§8.6). More precisely, evaluation of the lambda expression conversion produces an object structure that represents the structure of the lambda expression itself.

Not every lambda expression can be converted to expression tree types. The conversion to a compatible delegate type always *exists*, but it may fail at compile-time for implementation-specific reasons.

> *Note*: Common reasons for a lambda expression to fail to convert to an expression tree type include:
>
> - It has a block body
> - It has the `async` modifier
> - It contains an assignment operator
> - It contains an `out` or `ref` parameter
> - It contains a dynamically bound expression
>
> *end note*

10.8 Method group conversions

An implicit conversion exists from a method group (§12.2) to a compatible delegate type (§20.4). If `D` is a delegate type, and `E` is an expression that is classified as a method group, then `D` is compatible with `E` if and only if `E` contains at least one method that is applicable in its normal form (§12.6.4.2) to any argument list (§12.6.2) having types and modifiers matching the parameter types and modifiers of `D`, as described in the following.

The compile-time application of the conversion from a method group `E` to a delegate type `D` is described in the following.

- A single method `M` is selected corresponding to a method invocation (§12.8.9.2) of the form `E(A)`, with the following modifications:
  - The argument list `A` is a list of expressions, each classified as a variable and with the type and modifier (`in`, `out`, or `ref`) of the corresponding parameter in the *formal_parameter_list* of `D` — excepting parameters of type `dynamic`, where the corresponding expression has the type `object` instead of `dynamic`.
  - The candidate methods considered are only those methods that are applicable in their normal form and do not omit any optional parameters (§12.6.4.2). Thus, candidate methods are ignored if they are applicable only in their expanded form, or if one or more of their optional parameters do not have a corresponding parameter in `D`.
- A conversion is considered to exist if the algorithm of §12.8.9.2 produces a single best method `M` which is compatible (§20.4) with `D`.
- If the selected method `M` is an instance method, the instance expression associated with `E` determines the target object of the delegate.
- If the selected method `M` is an extension method which is denoted by means of a member access on an instance expression, that instance expression determines the

target object of the delegate.

- The result of the conversion is a value of type `D`, namely a delegate that refers to the selected method and target object.

Example: The following demonstrates method group conversions:

```C#
delegate string D1(object o);
delegate object D2(string s);
delegate object D3();
delegate string D4(object o, params object[] a);
delegate string D5(int i);
class Test
{
    static string F(object o) {...}

    static void G()
    {
        D1 d1 = F;          // Ok
        D2 d2 = F;          // Ok
        D3 d3 = F;          // Error – not applicable
        D4 d4 = F;          // Error – not applicable in normal
form
        D5 d5 = F;          // Error – applicable but not compati-
ble
    }
}
```

The assignment to `d1` implicitly converts the method group `F` to a value of type `D1`.

The assignment to `d2` shows how it is possible to create a delegate to a method that has less derived (contravariant) parameter types and a more derived (covariant) return type.

The assignment to `d3` shows how no conversion exists if the method is not applicable.

The assignment to `d4` shows how the method must be applicable in its normal form.

The assignment to `d5` shows how parameter and return types of the delegate and method are allowed to differ only for reference types.

end example

As with all other implicit and explicit conversions, the cast operator can be used to explicitly perform a particular conversion.

> *Example*: Thus, the example
>
> ```C#
> object obj = new EventHandler(myDialog.OkClick);
> ```
>
> could instead be written
>
> ```C#
> object obj = (EventHandler)myDialog.OkClick;
> ```
>
> *end example*

A method group conversion can refer to a generic method, either by explicitly specifying type arguments within `E`, or via type inference (§12.6.3). If type inference is used, the parameter types of the delegate are used as argument types in the inference process. The return type of the delegate is not used for inference. Whether the type arguments are specified or inferred, they are part of the method group conversion process; these are the type arguments used to invoke the target method when the resulting delegate is invoked.

> *Example*:
>
> ```C#
> delegate int D(string s, int i);
> delegate int E();
>
> class X
> {
>     public static T F<T>(string s, T t) {...}
>     public static T G<T>() {...}
>
>     static void Main()
>     {
>         D d1 = F<int>;        // Ok, type argument given explicit-
> ly
>         D d2 = F;             // Ok, int inferred as type argument
>         E e1 = G<int>;        // Ok, type argument given explicit-
> ly
>         E e2 = G;             // Error, cannot infer from return
> type
> ```

```
    }
}
```

end example

Method groups may influence overload resolution, and participate in type inference. See §12.6 for further details.

The run-time evaluation of a method group conversion proceeds as follows:

- If the method selected at compile-time is an instance method, or it is an extension method which is accessed as an instance method, the target object of the delegate is determined from the instance expression associated with `E`:
  - The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the instance expression is of a *reference_type*, the value computed by the instance expression becomes the target object. If the selected method is an instance method and the target object is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - If the instance expression is of a *value_type*, a boxing operation (§10.2.9) is performed to convert the value to an object, and this object becomes the target object.
- Otherwise, the selected method is part of a static method call, and the target object of the delegate is `null`.
- A delegate instance of delegate type `D` is obtained with a reference to the method that was determined at compile-time and a reference to the target object computed above, as follows:
- The conversion is permitted (but not required) to use an existing delegate instance that already contains these references.
- If an existing instance was not reused, a new one is created (§20.5). If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown. Otherwise the instance is initialized with the given references.