### Conversions

In this chapter you look at conversions in C# and how they allow for flexibility when using types.

Conversions do one thing and one thing alone — allow an expression of one type to be treated as another type. Conversions can take one of two forms:

* + **Implicit.** These are conversions that can occur automatically as required within the code.
  + **Explicit.** These conversions require a cast to be called.

All conversions in C# must be static and must either take the type that the conversion is defined on or return that type.

int x = 01234;

long y = x; // this is an implicit conversion, from int to long

int z = (int) y; // this is an explicit conversion, from long to int

In the preceding example, there is a conversion from int to long. This is an implicit conversion, and expressions of the type int can be treated as though they have the type long. However, the reverse, a conversion from long to int, is an explicit conversion, and an explicit cast is needed for this to work.

##### Implicit Conversions

The following conversions are all considered implicit:

* + Identity conversions
  + Implicit numeric conversions
  + Implicit enumeration conversions
  + Implicit reference conversions
  + Boxing conversions
* Implicit type parameter conversions
* Implicit constant expression conversions
* User-defined implicit conversions

There are many situations where an implicit conversion can occur. For example, in:

* Assignments
* Function member invocations
* Cast expressions

Identity Conversions

An identity conversion involves a conversion from one type to the same type. Very little is useful about this. It serves as nothing more than a way of making sure that errors aren’t generated when trying to convert one type to the same type.

Implicit Numeric Conversions

The following are implicit numeric conversions:

* From sbyte to decimal, double, float, int, long, and short
* From long to double, decimal, or float
* From ulong to double, decimal, or float
* From char to double, decimal, float, ushort, int, uint, long, or ulong
* From float to double
* From byte to decimal, double, short, ushort, int, uint, long, ulong, or float
* From short to double, decimal, int, long, or float
* From ushort to double, decimal, int, uint, long, ulong, or float
* From int to double, decimal, long, or float
* From uint to double, decimal, long, ulong, or float

Conversions from int, uint, long or ulong to float and from long or ulong to double quite often cause a loss of precision in the resulting value. This should be borne in mind if you’re carrying out high- precision technical work. However, such conversions will never cause a loss of magnitude of the value (a number that has a magnitude that is 103 will still retain the same magnitude).

*No other implicit numeric conversions cause any loss of precision in the resulting value.*

It’s important to bear in mind that no implicit conversion to the char type is possible, and other integral values won’t automatically convert to this type (if you think about it, it wouldn’t make sense if they did, since character strings would make no sense as any other type).

Implicit Enumeration Conversions

Implicit enumeration conversions simply allow the decimal integer literal 0 to be converted to any enum

type without causing an error. The enum types are:

* + byte
  + sbyte
  + short
  + ushort
  + int
  + uint
  + long
  + ulong

Implicit Reference Conversions

The following are implicit reference conversions:

* + From any reference type to object
  + 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 any array type to System.Array
  + From any delegate type to System.Delegate
  + From any array type to any interface implemented by System.Array
  + From any delegate type to System.ICloneable
  + From the null type to any reference type
  + From an array type S with an element type SE to an array type T with an element type TE, pro- vided all of the following are true:
    - S and T differ only in element type.
    - An implicit reference conversion exists from SE to TE.
  + From a one-dimensional array type S[] to System.Collections.Generic.IList<S> and base interfaces of this interface
  + From a one-dimensional array type S[] to System.Collections.Generic.IList<T> and base interfaces of this interface (if there is an implicit reference conversion from S to T)

If the type parameter is known to be a reference type, the following implicit references exist:

* From the null type to T
* 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

Boxing Conversions

A boxing conversion allows any value type to be implicitly converted as follows:

* To the type object
* To System.ValueType
* To any interface type implemented by the value type

It also allows any enum type to be implicitly converted to System.Enum. Boxing a value of a value type consists of:

* Allocating an object instance
* Copying the value type value into that instance A few additional notes:
* An enum can be boxed to the type System.Enum, because it is the direct base class for all enums.
* A struct or enum can be boxed to the type System.ValueType, because that is the direct base class for all structs and a base class for all enums.

For any type parameter T that is not a reference type, the following are all considered to be boxing con- versions:

* From T to its effective base class C, from T to any base class C, and from T to any interface imple- mented by C
* From T to an interface type I in T’s interface set and from T to any base interface of I

Implicit Type Parameter Conversions

For a type parameter T that is not known to be a reference type, there will be an implicit conversion from

T to a type parameter U, provided that the type parameter T depends on U.

At runtime, if T is a value type and U is a reference type, the conversion will be carried out as though it is a boxing conversion.

At runtime, if both T and U are value types, T and U are necessarily the same type, and no conversion will be carried out on either of the types.

At runtime, if T is a reference type, U will also be a reference type, and the conversion is carried out as either an implicit reference conversion or an identity conversion.

Implicit Constant Expression Conversions

An implicit conversion expression allows for the following conversions to be carried out:

* + Any constant expression of the type int can be converted to byte, sbyte, short, ushort, uint, or ulong as long as the value of the constant expression is within the range of the result- ing type.
  + Any constant expression of the type long can be converted to the type ulong, as long as the value of the constant expression is not negative.

User Defined Implicit Conversions

A user-defined implicit conversion consists of:

* + An optional standard implicit conversion, followed by
  + The execution of a user-defined implicit conversion operator, followed by
  + Another optional standard implicit conversion

##### Explicit Conversions

Explicit conversions are classed as follows:

* + All implicit conversions
  + Explicit numeric conversions
  + Explicit enumeration conversions
  + Explicit reference conversions
  + Unboxing conversions
  + Explicit type parameter conversions
  + User-defined explicit conversions

Explicit Numeric Conversions

Explicit numeric conversions are conversions from one numeric type to another where an implicit con- version does not 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 explicit conversions cover all implicit and explicit numerical conversions, it is always possible to convert from one numeric type to anther using a cast expression (covered in greater detail in Chapter 9).*

Using explicit numeric conversions can sometimes cause a loss of information; bear this in mind if high precision is important. It is also possible for explicit numeric conversions to throw an exception.

Explicit numeric conversions are processed depending on the type of conversion being carried out.

Integral Type to Integral Type

This conversion depends on the overflow-checking context in which the conversion takes place, which we will now look at.

* When carried out in a checked context, the conversion will be successful if the value of the source operand falls within the range of the destination type. A System.OverflowException is thrown if the value of the source operand falls outside the range of the destination type.
* When carried out in an unchecked context, the conversion will always be successful. The fol- lowing processes will be carried out:
  + If the source type is larger than the destination type, the source is truncated by discarding significant bits.
  + If the source is smaller, the source value is sign-extended if the source type is signed (sim- ply put, this means that the + or – is added) or zero-extended if it is unsigned.
  + If the source type is identical to the destination type, they are treated as equivalent.

Decimal to Integral Type

In conversions that go from decimal to an integral type, the source type is always rounded — toward zero — to the nearest integral value. This integer becomes the result of the conversion. There is signifi- cant loss of precision here.

If the resulting integral value falls outside of the range of the destination type, the conversion results in a

System.OverflowException being thrown.

Float/Double to Int Type

Conversion from float to int and double to int depends on the overflow-checking context in which the conversion takes place.

* + - In a checked context, the value is rounded — toward zero — to the nearest negative integral value. If this resulting integral value falls within the range of the destination type, the value is the result of the conversion. If it falls outside, a System.OverflowException is thrown.
    - In an unchecked context, the conversion will always be successful. The value is rounded — toward zero — to the nearest integral value. If this value falls within the range of the destination type, this becomes the value of the conversion; otherwise, the result of the conversion is an unspecified value.

Double to Float

In conversions from double to float, the double value is rounded to the nearest float value.

*Be aware that this rounding may cause a value that is initially nonzero to be rounded to a zero value.*

Double values that are too big to be represented as a float will result in a positive infinity or negative infinity value.

If the double value is NaN, the result of this conversion will also be NaN.

Float/Double to Decimal

In conversions from float or double to decimal, the source values will be converted to decimal and then subsequently rounded to the nearest number. This rounding might cause a nonzero number to be rounded to zero, which will result in a significant loss of precision.

If the source number is too large to be represented as decimal or if the value is either NaN or infinity, a

System.OverflowException will be thrown.

Decimal to Float/Double

In conversions that involve a conversion from decimal to float or double, the value is rounded to the nearest float or double value as required by the code.

If the value being converted does not fall within the range of the destination type, a

System.OverflowException is thrown.

Explicit Enumeration Conversions

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

Explicit Reference Conversions

Explicit reference conversions are:

* From object to any reference type
* From any class type S to any class type T, as long as S is a base class of T
* From any class type S to any interface type T, as long as S is not sealed and provided S does not implement T
* From any interface type S to any class type T, as long as T is not sealed or provided T

implements S

* From any interface type S to any interface type T, as long as S is not derived from T
* From System.Array and the interfaces it implements, to any array type
* From System.Delegate and the interfaces it implements, to any delegate type
* From a one-dimensional array type S[] to System.Collections.Generic.IList<T> and its base interfaces, as long as there is an explicit reference conversion from S to T
* From System.Collections.Generic.IList<T> and its base interfaces to a one-dimensional array type S[], as long as there is an implicit or explicit reference conversion from S[] to System.Collections.Generic.IList<T>
* From an array type S with an element type SE to an array type T with an element type TE, as long as all of the following are true:
  + S and T differ only in element type.
  + An explicit reference conversion exists from SE to TE.

For a type-parameter T which is a reference type, the following explicit reference conversions are allowable:

* 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, as long as there isn’t already an implicit reference conversion from T to I
* From a type parameter U to T, as long as T depends on U

Explicit reference conversions are carried out between reference types that require runtime checks to ensure they are correct.

For an explicit reference conversion to be successful during runtime, the value of the source operand must be null, or the runtime type of the object referenced by the source operand has to be a type that can be converted to the destination type by an implicit reference conversion.

If an explicit reference conversion is unsuccessful, a System.InvalidCastException is thrown.

Unboxing Conversions

An unboxing conversion allows:

* + An explicit conversion from type object to System.ValueType to a value type
  + From an interface type to any value type that implements the interface type
  + From the type System.Enum to any enumeration type

An unboxing operation is a two-step process and proceeds as follows:

* + A check is carried out to make sure that the object instance is a boxed value of a given value or enumeration type.
  + The value is copied from the instance.

Explicit Type Parameter Conversions

For a type parameter T that is not known to be a reference type, the following explicit conversions are allowed:

* + From T to any interface type I, provided there is not already an implicit conversion from T to I
  + From a type parameter U to T, provided that T depends on U

User-Defined Explicit Conversions

User-defined explicit conversions are made up of:

* + An optional explicit conversion, followed by
  + The execution of a user-defined implicit or explicit conversion operator, followed by
  + Another optional standard explicit conversion

##### Standard Conversions

The standard conversions, explained in the following sections, are predefined and can occur as part of a user-defined conversion.

Standard Implicit Conversions

The following conversions are all standard implicit conversions:

* + Identity conversions
  + Implicit numeric conversions
  + Implicit reference conversions
  + Boxing conversions
* Implicit type parameter conversions
* Implicit constant expression conversions
* Implicit nullable conversions

Standard Explicit Conversions

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

User-Defined Conversions

C# allows for predefined implicit and explicit conversions to be augmented by user-defined conversions. This is carried out by declaring conversion operators in class and struct types.

*It is not possible to redefine a conversion already defined as implicit or explicit.*

User-Defined Implicit Conversions

User-defined implicit conversions from type S to type T are carried out as follows:

* Find the types S0 and T0 that result from deleting the trailing ? modifiers from S and T.
* Find the set of types, D, from which user-defined conversion operators will be considered. This set consists of S0, which is a class or struct, the base classes of S0 if S0 is a class, and T0 if T0 is a class or struct.
* Discover the set of applicable conversion operators, U. This is made up of the user-defined and, if S and T are both nullable, lifted implicit conversion operators declared by the classes or structs in D that convert from a type encompassing S to a type encompassed by T.

*If* U *is empty, there is no conversion, and a compile-time error occurs.*

* Work out the most specific source type, SX, of the operators in U. If any of the operators in U con- vert from S, SX is S; otherwise, SX is the most encompassed type in the combined set of source types of the operators in U.
* Work out the most specific target type, TX, from the operators in U. If any of the operators in U convert to T, TX is T; otherwise, TX is the most encompassed type in the combined set of target types of the operators in U.
* Work out the most specific conversion operator. If U contains exactly one user-defined conver- sion operator that converts SX to TX, this is the most specific; otherwise, if U contains one lifted conversion operator that converts from SX to TX, this is the most specific conversion operator. If the conversion is ambiguous, a compile-time error occurs.
  + Finally, the conversions are applied as follows:
  + If S is not SX, a standard implicit conversion from S to SX is carried out.
  + The most specific conversion is invoked and converted from SX to TX.
  + If TX is not T, a standard implicit conversion from TX to T is carried out.

##### Anonymous Method Conversions

An implicit conversion exists from an anonymous method expression to any compatible delegate type. If D is a delegate type and A is an anonymous method expression, D is compatible with A if the following conditions are true:

* + - If A does not contain an anonymous method signature, D can have zero or more parameters of any type as long as the out parameter of D is modified.
    - If A has an anonymous method signature, D will have the same number of parameters. Each parameter of A will be compatible with the corresponding parameter on D (this occurs when they are both of the same type and when the presence or absence of the out or ref modifiers on A match those of D).

##### Method Group Conversions

Method group conversions are implicit conversion methods that transform them to a compatible dele- gate. This is similar to the implicit anonymous group method.

If D is a delegate type and E is an expression classified as a method group, D will be compatible with E if (and only if) E contains at least one method that is applicable in its normal form to any argument list having types and modifiers matching the parameter types and modifiers of D.

The compile-time application of this conversion of E to D is identical to the compile-time processing of the delegate creating expression D(E).

##### Null Type Conversions

An implicit conversion is allowed from the null type to any nullable type. This conversion will produce a null value of the given nullable type.

Nullable Conversions

Before we look at nullable conversions, allow us to introduce two terms:

* + - **Wrapping.** This is a process of packaging a value of type T in an instance of type T?. A value x

of type T is wrapped to type T? by evaluating a new expression: T?(x).

* + - **Unwrapping.** This is the process of returning a value of type T contained in an instance of type T?. This is done by evaluating the expression x.Value. Unwrapping null instances will cause a System.InvalidOperationException to be thrown.

Nullable conversions allow for predefined conversions that work on non-nullable values types. Each predefined conversion converts from a nullable value type S to a non-nullable value T.

For every predefined implicit or explicit conversion that converts from a non-nullable value type S to a non-nullable value T, the following must exist:

* There must be either an implicit or explicit nullable conversion from S? to T?.
* There must be an implicit or explicit nullable conversion from S to T?.
* There must be an explicit nullable conversion from S? to T.

In the preceding, a nullable conversion can be either an implicit or explicit conversion.

##### Summary

This chapter looked in detail at both implicit and explicit conversions in C#. As a standalone chapter, the content here might seem complex, which is why it’s recommended that you read this chapter as part of a bigger reading plan and read the referenced chapters too.

In Chapter 9, you look at expressions in C#.