Type-testing operators and cast expressions - `is`, `as`, `typeof` and casts

Article • 04/08/2023

These operators and expressions perform type checking or type conversion. The `is` operator checks if the run-time type of an expression is compatible with a given type. The `as` operator explicitly converts an expression to a given type if its run-time type is compatible with that type. Cast expressions perform an explicit conversion to a target type. The `typeof` operator obtains the System.Type instance for a type.

is operator

The `is` operator checks if the run-time type of an expression result is compatible with a given type. The `is` operator also tests an expression result against a pattern.

The expression with the type-testing `is` operator has the following form

C#

```
E is T
```

where `E` is an expression that returns a value and `T` is the name of a type or a type parameter. `E` can't be an anonymous method or a lambda expression.

The `is` operator returns `true` when an expression result is non-null and any of the following conditions are true:

- The run-time type of an expression result is `T`.
- The run-time type of an expression result derives from type `T`, implements interface `T`, or another implicit reference conversion exists from it to `T`.
- The run-time type of an expression result is a nullable value type with the underlying type `T` and the Nullable<T>.HasValue is `true`.
- A boxing or unboxing conversion exists from the run-time type of an expression result to type `T`.

The `is` operator doesn't consider user-defined conversions.

The following example demonstrates that the `is` operator returns `true` if the run-time type of an expression result derives from a given type, that is, there exists a reference conversion between types:

C#

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

The next example shows that the `is` operator takes into account boxing and unboxing conversions but doesn't consider numeric conversions:

C#

```
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

For information about C# conversions, see the Conversions chapter of the C# language specification.

Type testing with pattern matching

The `is` operator also tests an expression result against a pattern. The following example shows how to use a declaration pattern to check the run-time type of an expression:

C#

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b);  // output 30
}
```

For information about the supported patterns, see Patterns.

as operator

The `as` operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion isn't possible, the `as` operator returns `null`. Unlike a cast expression, the `as` operator never throws an exception.

The expression of the form

```C#
E as T
```

where `E` is an expression that returns a value and `T` is the name of a type or a type parameter, produces the same result as

```C#
E is T ? (T)(E) : (T)null
```

except that `E` is only evaluated once.

The `as` operator considers only reference, nullable, boxing, and unboxing conversions. You can't use the `as` operator to perform a user-defined conversion. To do that, use a cast expression.

The following example demonstrates the usage of the `as` operator:

```C#
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]);
// output: 40
}
```

ⓘ Note

As the preceding example shows, you need to compare the result of the `as` expression with `null` to check if the conversion is successful. You can use the **is operator** both to test if the conversion succeeds and, if it succeeds, assign its result to a new variable.

Cast expression

A cast expression of the form `(T)E` performs an explicit conversion of the result of expression `E` to type `T`. If no explicit conversion exists from the type of `E` to type `T`, a compile-time error occurs. At run time, an explicit conversion might not succeed and a cast expression might throw an exception.

The following example demonstrates explicit numeric and reference conversions:

C#

```csharp
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count);  // output: 3
Console.WriteLine(list[1]);  // output: 20
```

For information about supported explicit conversions, see the Explicit conversions section of the C# language specification. For information about how to define a custom explicit or implicit type conversion, see User-defined conversion operators.

Other usages of ()

You also use parentheses to call a method or invoke a delegate.

Other use of parentheses is to adjust the order in which to evaluate operations in an expression. For more information, see C# operators.

typeof operator

The `typeof` operator obtains the System.Type instance for a type. The argument to the `typeof` operator must be the name of a type or a type parameter, as the following example shows:

```
C#

void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

The argument mustn't be a type that requires metadata annotations. Examples include the following types:

- `dynamic`
- `string?` (or any nullable reference type)

These types aren't directly represented in metadata. The types include attributes that describe the underlying type. In both cases, you can use the underlying type. Instead of `dynamic`, you can use `object`. Instead of `string?`, you can use `string`.

You can also use the `typeof` operator with unbound generic types. The name of an unbound generic type must contain the appropriate number of commas, which is one less than the number of type parameters. The following example shows the usage of the `typeof` operator with an unbound generic type:

```
C#

Console.WriteLine(typeof(Dictionary<,>));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

An expression can't be an argument of the `typeof` operator. To get the System.Type instance for the run-time type of an expression result, use the Object.GetType method.

Type testing with the `typeof` operator

Use the `typeof` operator to check if the run-time type of the expression result exactly matches a given type. The following example demonstrates the difference between type checking done with the `typeof` operator and the is operator:

C#

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output:
False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // out-
put: True
    }
}
```

Operator overloadability

The `is`, `as`, and `typeof` operators can't be overloaded.

A user-defined type can't overload the `()` operator, but can define custom type conversions that can be performed by a cast expression. For more information, see User-defined conversion operators.

C# language specification

For more information, see the following sections of the C# language specification:

- The is operator
- The as operator
- Cast expressions
- The typeof operator

See also

- C# reference
- C# operators and expressions
- How to safely cast by using pattern matching and the is and as operators
- Generics in .NET