Pattern matching - the `is` and `switch` expressions, and operators `and`, `or` and `not` in patterns

Article • 01/31/2023

You use the is expression, the switch statement and the switch expression to match an input expression against any number of characteristics. C# supports multiple patterns, including declaration, type, constant, relational, property, list, var, and discard. Patterns can be combined using Boolean logic keywords `and`, `or`, and `not`.

The following C# expressions and statements support pattern matching:

- is expression
- switch statement
- switch expression

In those constructs, you can match an input expression against any of the following patterns:

- Declaration pattern: to check the run-time type of an expression and, if a match succeeds, assign an expression result to a declared variable.
- Type pattern: to check the run-time type of an expression. Introduced in C# 9.0.
- Constant pattern: to test if an expression result equals a specified constant.
- Relational patterns: to compare an expression result with a specified constant. Introduced in C# 9.0.
- Logical patterns: to test if an expression matches a logical combination of patterns. Introduced in C# 9.0.
- Property pattern: to test if an expression's properties or fields match nested patterns.
- Positional pattern: to deconstruct an expression result and test if the resulting values match nested patterns.
- var pattern: to match any expression and assign its result to a declared variable.
- Discard pattern: to match any expression.
- List patterns: to test if sequence elements match corresponding nested patterns. Introduced in C# 11.

Logical, property, positional, and list patterns are *recursive* patterns. That is, they can contain *nested* patterns.

For the example of how to use those patterns to build a data-driven algorithm, see Tutorial: Use pattern matching to build type-driven and data-driven algorithms.

Declaration and type patterns

You use declaration and type patterns to check if the run-time type of an expression is compatible with a given type. With a declaration pattern, you can also declare a new local variable. When a declaration pattern matches an expression, that variable is assigned a converted expression result, as the following example shows:

C#

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower());  // output: hello, world!
}
```

A *declaration pattern* with type `T` matches an expression 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 following example demonstrates two cases when this condition is true:

  C#

  ```
  var numbers = new int[] { 10, 20, 30 };
  Console.WriteLine(GetSourceLabel(numbers));  // output: 1

  var letters = new List<char> { 'a', 'b', 'c', 'd' };
  Console.WriteLine(GetSourceLabel(letters));  // output: 2

  static int GetSourceLabel<T>(IEnumerable<T> source) => source
  switch
  {
      Array array => 1,
      ICollection<T> collection => 2,
      _ => 3,
  };
  ```

  In the preceding example, at the first call to the `GetSourceLabel` method, the first pattern matches an argument value because the argument's run-time type `int[]`

derives from the Array type. At the second call to the `GetSourceLabel` method, the argument's run-time type List<T> doesn't derive from the Array type but implements the ICollection<T> interface.

- The run-time type of an expression result is a nullable value type with the underlying type `T`.

- A boxing or unboxing conversion exists from the run-time type of an expression result to type `T`.

The following example demonstrates the last two conditions:

C#

```
int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b);  // output: 30
}
```

If you want to check only the type of an expression, you can use a discard `_` in place of a variable's name, as the following example shows:

C#

```
public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => ve-
hicle switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
    };
}
```

Beginning with C# 9.0, for that purpose you can use a *type pattern*, as the following example shows:

C#

```
public static decimal CalculateToll(this Vehicle vehicle) => vehicle
switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
};
```

Like a declaration pattern, a type pattern matches an expression when an expression result is non-null and its run-time type satisfies any of the conditions listed above.

To check for non-null, you can use a negated `null` constant pattern, as the following example shows:

```C#
if (input is not null)
{
    // ...
}
```

For more information, see the Declaration pattern and Type pattern sections of the feature proposal notes.

Constant pattern

You use a *constant pattern* to test if an expression result equals a specified constant, as the following example shows:

```C#
public static decimal GetGroupTicketPrice(int visitorCount) => visi-
torCount switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visi-
tors: {visitorCount}", nameof(visitorCount)),
};
```

In a constant pattern, you can use any constant expression, such as:

- an integer or floating-point numerical literal
- a char
- a string literal.
- a Boolean value `true` or `false`
- an enum value
- the name of a declared const field or local
- `null`

The expression must be a type that is convertible to the constant type, with one exception: An expression whose type is `Span<char>` or `ReadOnlySpan<char>` can be matched against constant strings in C# 11 and later versions.

Use a constant pattern to check for `null`, as the following example shows:

C#

```
if (input is null)
{
    return;
}
```

The compiler guarantees that no user-overloaded equality operator `==` is invoked when expression `x is null` is evaluated.

Beginning with C# 9.0, you can use a negated `null` constant pattern to check for non-null, as the following example shows:

C#

```
if (input is not null)
{
    // ...
}
```

For more information, see the Constant pattern section of the feature proposal note.

Relational patterns

Beginning with C# 9.0, you use a *relational pattern* to compare an expression result with a constant, as the following example shows:

C#

```
Console.WriteLine(Classify(13));  // output: Too high
Console.WriteLine(Classify(double.NaN));  // output: Unknown
Console.WriteLine(Classify(2.4));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};
```

In a relational pattern, you can use any of the relational operators `<`, `>`, `<=`, or `>=`. The right-hand part of a relational pattern must be a constant expression. The constant expression can be of an integer, floating-point, char, or enum type.

To check if an expression result is in a certain range, match it against a conjunctive and pattern, as the following example shows:

```C#
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14)));  // output: spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19)));  // output: summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17)));  // output: winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};
```

If an expression result is `null` or fails to convert to the type of a constant by a nullable or unboxing conversion, a relational pattern doesn't match an expression.

For more information, see the Relational patterns section of the feature proposal note.

Logical patterns

Beginning with C# 9.0, you use the `not`, `and`, and `or` pattern combinators to create the following *logical patterns*:

- *Negation* `not` pattern that matches an expression when the negated pattern doesn't match the expression. The following example shows how you can negate a constant `null` pattern to check if an expression is non-null:

C#

```
if (input is not null)
{
    // ...
}
```

- *Conjunctive* `and` pattern that matches an expression when both patterns match the expression. The following example shows how you can combine relational patterns to check if a value is in a certain range:

C#

```
Console.WriteLine(Classify(13));  // output: High
Console.WriteLine(Classify(-100));  // output: Too low
Console.WriteLine(Classify(5.7));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -40.0 => "Too low",
    >= -40.0 and < 0 => "Low",
    >= 0 and < 10.0 => "Acceptable",
    >= 10.0 and < 20.0 => "High",
    >= 20.0 => "Too high",
    double.NaN => "Unknown",
};
```

- *Disjunctive* `or` pattern that matches an expression when either pattern matches the expression, as the following example shows:

C#

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19)));
// output: winter
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9)));
// output: autumn
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11)));
// output: spring

static string GetCalendarSeason(DateTime date) => date.Month
switch
{
    3 or 4 or 5 => "spring",
    6 or 7 or 8 => "summer",
    9 or 10 or 11 => "autumn",
```

```
    12 or 1 or 2 => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date),
$"Date with unexpected month: {date.Month}."),
};
```

As the preceding example shows, you can repeatedly use the pattern combinators in a pattern.

Precedence and order of checking

The pattern combinators are ordered from the highest precedence to the lowest as follows:

- `not`
- `and`
- `or`

When a logical pattern is a pattern of an `is` expression, the precedence of logical pattern combinators is **higher** than the precedence of logical operators (both bitwise logical and Boolean logical operators). Otherwise, the precedence of logical pattern combinators is **lower** than the precedence of logical and conditional logical operators. For the complete list of C# operators ordered by precedence level, see the Operator precedence section of the C# operators article.

To explicitly specify the precedence, use parentheses, as the following example shows:

C#

```csharp
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A'
and <= 'Z');
```

> ⓘ **Note**
>
> The order in which patterns are checked is undefined. At run time, the right-hand nested patterns of `or` and `and` patterns can be checked first.

For more information, see the Pattern combinators section of the feature proposal note.

Property pattern

You use a *property pattern* to match an expression's properties or fields against nested patterns, as the following example shows:

C#

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020,
Month: 5, Day: 19 or 20 or 21 };
```

A property pattern matches an expression when an expression result is non-null and every nested pattern matches the corresponding property or field of the expression result.

You can also add a run-time type check and a variable declaration to a property pattern, as the following example shows:

C#

```
Console.WriteLine(TakeFive("Hello, world!"));  // output: Hello
Console.WriteLine(TakeFive("Hi!"));  // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7'
}));  // output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' }));  // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new string(symbols.-
Take(5).ToArray()),
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};
```

A property pattern is a recursive pattern. That is, you can use any pattern as a nested pattern. Use a property pattern to match parts of data against nested patterns, as the following example shows:

C#

```
public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start: { Y: 0 } } or { End: { Y: 0 } };
```

The preceding example uses two features available in C# 9.0 and later: `or` pattern combinator and record types.

Beginning with C# 10, you can reference nested properties or fields within a property pattern. This capability is known as an *extended property pattern*. For example, you can refactor the method from the preceding example into the following equivalent code:

C#

```
static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start.Y: 0 } or { End.Y: 0 };
```

For more information, see the Property pattern section of the feature proposal note and the Extended property patterns feature proposal note.

> **Tip**
>
> You can use the **Simplify property pattern (IDE0170)** style rule to improve code readability by suggesting places to use extended property patterns.

Positional pattern

You use a *positional pattern* to deconstruct an expression result and match the resulting values against the corresponding nested patterns, as the following example shows:

C#

```
public readonly struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) => (x, y) = (X, Y);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};
```

At the preceding example, the type of an expression contains the Deconstruct method, which is used to deconstruct an expression result. You can also match expressions of tuple types against positional patterns. In that way, you can match multiple inputs against various patterns, as the following example shows:

C#

```
static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime visitDate)
    => (groupSize, visitDate.DayOfWeek) switch
    {
        (<= 0, _) => throw new ArgumentException("Group size must be positive."),
        (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
        (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
        (>= 10, DayOfWeek.Monday) => 30.0m,
        (>= 5 and < 10, _) => 12.0m,
        (>= 10, _) => 15.0m,
        _ => 0.0m,
    };
```

The preceding example uses relational and logical patterns, which are available in C# 9.0 and later.

You can use the names of tuple elements and `Deconstruct` parameters in a positional pattern, as the following example shows:

C#

```
var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}");  // output: Sum of [1 2 3] is 6
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}
```

You can also extend a positional pattern in any of the following ways:

- Add a run-time type check and a variable declaration, as the following example shows:

C#

```
public record Point2D(int X, int Y);
public record Point3D(int X, int Y, int Z);

static string PrintIfAllCoordinatesArePositive(object point) =>
point switch
{
    Point2D (> 0, > 0) p => p.ToString(),
    Point3D (> 0, > 0, > 0) p => p.ToString(),
    _ => string.Empty,
};
```

The preceding example uses positional records that implicitly provide the `Deconstruct` method.

- Use a property pattern within a positional pattern, as the following example shows:

C#

```
public record WeightedPoint(int X, int Y)
{
    public double Weight { get; set; }
}

static bool IsInDomain(WeightedPoint point) => point is (>= 0, >=
0) { Weight: >= 0.0 };
```

- Combine two preceding usages, as the following example shows:

C#

```
if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ..
}
```

A positional pattern is a recursive pattern. That is, you can use any pattern as a nested pattern.

For more information, see the Positional pattern section of the feature proposal note.

`var` pattern

You use a `var` *pattern* to match any expression, including `null`, and assign its result to a new local variable, as the following example shows:

C#

```
static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
            .Range(start: 0, count: 5)
            .Select(s => rand.Next(minValue: -10, maxValue: 11))
            .ToArray();
}
```

A `var` pattern is useful when you need a temporary variable within a Boolean expression to hold the result of intermediate calculations. You can also use a `var` pattern when you need to perform more checks in `when` case guards of a `switch` expression or statement, as the following example shows:

C#

```
public record Point(int X, int Y);

static Point Transform(Point point) => point switch
{
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2)));  // output: Point
{ X = -1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2)));  // output: Point
{ X = 5, Y = -2 }
}
```

In the preceding example, pattern `var (x, y)` is equivalent to a positional pattern `(var x, var y)`.

In a `var` pattern, the type of a declared variable is the compile-time type of the expression that is matched against the pattern.

For more information, see the Var pattern section of the feature proposal note.

Discard pattern

You use a *discard pattern* `_` to match any expression, including `null`, as the following example shows:

```C#
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday));  // out-
put: 5.0
Console.WriteLine(GetDiscountInPercent(null));  // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10));  // output:
0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOf-
Week switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
    _ => 0.0m,
};
```

In the preceding example, a discard pattern is used to handle `null` and any integer value that doesn't have the corresponding member of the DayOfWeek enumeration. That guarantees that a `switch` expression in the example handles all possible input values. If you don't use a discard pattern in a `switch` expression and none of the expression's patterns matches an input, the runtime throws an exception. The compiler generates a warning if a `switch` expression doesn't handle all possible input values.

A discard pattern can't be a pattern in an `is` expression or a `switch` statement. In those cases, to match any expression, use a var pattern with a discard: `var _`. A discard pattern can be a pattern in a `switch` expression.

For more information, see the Discard pattern section of the feature proposal note.

Parenthesized pattern

Beginning with C# 9.0, you can put parentheses around any pattern. Typically, you do that to emphasize or change the precedence in logical patterns, as the following example shows:

C#

```
if (input is not (float or double))
{
    return;
}
```

List patterns

Beginning with C# 11, you can match an array or a list against a *sequence* of patterns, as the following example shows:

C#

```
int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers is [1, 2, 3]);  // True
Console.WriteLine(numbers is [1, 2, 4]);  // False
Console.WriteLine(numbers is [1, 2, 3, 4]);  // False
Console.WriteLine(numbers is [0 or 1, <= 2, >= 3]);  // True
```

As the preceding example shows, a list pattern is matched when each nested pattern is matched by the corresponding element of an input sequence. You can use any pattern within a list pattern. To match any element, use the discard pattern or, if you also want to capture the element, the var pattern, as the following example shows:

C#

```
List<int> numbers = new() { 1, 2, 3 };

if (numbers is [var first, _, _])
{
    Console.WriteLine($"The first element of a three-item list is
{first}.");
}
// Output:
// The first element of a three-item list is 1.
```

The preceding examples match a whole input sequence against a list pattern. To match elements only at the start or/and the end of an input sequence, use the *slice pattern* `..`, as the following example shows:

C#

```
Console.WriteLine(new[] { 1, 2, 3, 4, 5 } is [> 0, > 0, ..]);  //
True
Console.WriteLine(new[] { 1, 1 } is [_, _, ..]);  // True
Console.WriteLine(new[] { 0, 1, 2, 3, 4 } is [> 0, > 0, ..]);  //
False
Console.WriteLine(new[] { 1 } is [1, 2, ..]);  // False

Console.WriteLine(new[] { 1, 2, 3, 4 } is [.., > 0, > 0]);  // True
Console.WriteLine(new[] { 2, 4 } is [.., > 0, 2, 4]);  // False
Console.WriteLine(new[] { 2, 4 } is [.., 2, 4]);  // True

Console.WriteLine(new[] { 1, 2, 3, 4 } is [>= 0, .., 2 or 4]);  //
True
Console.WriteLine(new[] { 1, 0, 0, 1 } is [1, 0, .., 0, 1]);  // True
Console.WriteLine(new[] { 1, 0, 1 } is [1, 0, .., 0, 1]);  // False
```

A slice pattern matches zero or more elements. You can use at most one slice pattern in a list pattern. The slice pattern can only appear in a list pattern.

You can also nest a subpattern within a slice pattern, as the following example shows:

C#

```
void MatchMessage(string message)
{
    var result = message is ['a' or 'A', .. var s, 'a' or 'A']
        ? $"Message {message} matches; inner part is {s}."
        : $"Message {message} doesn't match.";
    Console.WriteLine(result);
}

MatchMessage("aBBA");  // output: Message aBBA matches; inner part is
BB.
MatchMessage("apron");  // output: Message apron doesn't match.

void Validate(int[] numbers)
{
    var result = numbers is [< 0, .. { Length: 2 or 4 }, > 0] ?
"valid" : "not valid";
    Console.WriteLine(result);
}

Validate(new[] { -1, 0, 1 });  // output: not valid
Validate(new[] { -1, 0, 0, 1 });  // output: valid
```

For more information, see the List patterns feature proposal note.

C# language specification

For more information, see the Patterns and pattern matching section of the C# language specification.

For information about features added in C# 8 and later, see the following feature proposal notes:

- C# 8 - Recursive pattern matching
- C# 9 - Pattern-matching updates
- C# 10 - Extended property patterns
- C# 11 - List patterns
- C# 11 - Pattern match Span<char> on string literal

See also

- C# reference
- C# operators and expressions
- Pattern matching overview
- Tutorial: Use pattern matching to build type-driven and data-driven algorithms