characteristics. C# pattern matching provides more concise syntax for testing expressions and taking action when an expression matches. The "is expression" supports pattern matching to test an expression and conditionally declare a new variable to the result of that expression. The "switch expression" enables you to perform actions based on the first matching pattern for an expression. These two expressions support a rich vocabulary of *patterns*.

This article provides an overview of scenarios where you can use pattern matching. These techniques can improve the readability and correctness of your code. For a full discussion of all the patterns you can apply, see the article on patterns in the language reference.

Null checks

One of the most common scenarios for pattern matching is to ensure values aren't `null`. You can test and convert a nullable value type to its underlying type while testing for `null` using the following example:

C#

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

The preceding code is a *declaration pattern* to test the type of the variable, and assign it to a new variable. The language rules make this technique safer than many others. The variable `number` is only accessible and assigned in the true portion of the `if` clause. If you try to access it elsewhere, either in the `else` clause, or after the `if` block, the compiler issues an error. Secondly, because you're not using the `==` operator, this pattern works

when a type overloads the `==` operator. That makes it an ideal way to check null reference values, adding the `not` pattern:

C#

```
string? message = "This is not the null string";
```

```
if (message is not null)
{
    Console.WriteLine(message);
}
```

The preceding example used a *constant pattern* to compare the variable to `null`. The `not` is a *logical pattern* that matches when the negated pattern doesn't match.

Type tests

Another common use for pattern matching is to test a variable to see if it matches a given type. For example, the following code tests if a variable is non-null and implements the System.Collections.Generic.IList<T> interface. If it does, it uses the ICollection<T>.Count property on that list to find the middle index. The declaration pattern doesn't match a `null` value, regardless of the compile-time type of the variable. The code below guards against `null`, in addition to guarding against a type that doesn't implement `IList`.

C#

```
public static T MidPoint<T>(IEnumerable<T> sequence)
{
    if (sequence is IList<T> list)
    {
        return list[list.Count / 2];
    }
    else if (sequence is null)
    {
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be
null.");
    }
    else
    {
        int halfLength = sequence.Count() / 2 - 1;
        if (halfLength < 0) halfLength = 0;
        return sequence.Skip(halfLength).First();



    }
}
```

The same tests can be applied in a `switch` expression to test a variable against multiple different types. You can use that information to create better algorithms based on the specific run-time type.

Compare discrete values

You can also test a variable to find a match on specific values. The following code shows one example where you test a value against all possible values declared in an enumeration:

C#

```
public State PerformOperation(Operation command) =>
   command switch
   {
       Operation.SystemTest => RunDiagnostics(),
       Operation.Start => StartSystem(),
       Operation.Stop => StopSystem(),
       Operation.Reset => ResetToReady(),
       _ => throw new ArgumentException("Invalid enum value for command",
nameof(command)),
   };
```

The previous example demonstrates a method dispatch based on the value of an enumeration. The final `_` case is a *discard pattern* that matches all values. It handles any error conditions where the value doesn't match one of the defined `enum` values. If you omit that switch arm, the compiler warns that you haven't handled all possible input values. At run time, the `switch` expression throws an exception if the object being examined doesn't match any of the switch arms. You could use numeric constants instead of a set of enum values. You can also use this similar technique for constant string values that represent the commands:

C#

```
public State PerformOperation(string command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),

       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
   };
```

The preceding example shows the same algorithm, but uses string values instead of an

enum. You would use this scenario if your application responds to text commands instead of a regular data format. Starting with C# 11, you can also use a `Span<char>` or a `ReadOnlySpan<char>` to test for constant string values, as shown in the following sample:

C#

```
public State PerformOperation(ReadOnlySpan<char> command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
   };
```

In all these examples, the *discard pattern* ensures that you handle every input. The compiler helps you by making sure every possible input value is handled.

Relational patterns

You can use *relational patterns* to test how a value compares to constants. For example, the following code returns the state of water based on the temperature in Fahrenheit:

C#

```
string WaterState(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        (> 32) and (< 212) => "liquid",
        < 32 => "solid",
        > 212 => "gas",
        32 => "solid/liquid transition",
        212 => "liquid / gas transition",
    };
```

The preceding code also demonstrates the conjunctive `and` *logical pattern* to check that both relational patterns match. You can also use a disjunctive `or` pattern to check that either pattern matches. The two relational patterns are surrounded by parentheses, which you can use around any pattern for clarity. The final two switch arms handle the cases for the melting point and the boiling point. Without those two arms, the compiler warns you that `or` logic doesn't cover every possible input.

that your logic doesn't cover every possible input.

The preceding code also demonstrates another important feature the compiler provides for pattern matching expressions: The compiler warns you if you don't handle every input value. The compiler also issues a warning if a switch arm is already handled by a previous switch arm. That gives you freedom to refactor and reorder switch expressions. Another way to write the same expression could be:

C#

```
string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        < 212 => "liquid",
        212 => "liquid / gas transition",
        _ => "gas",
};
```

The key lesson in this, and any other refactoring or reordering, is that the compiler validates that you've covered all inputs.

Multiple inputs

All the patterns you've seen so far have been checking one input. You can write patterns that examine multiple properties of an object. Consider the following `Order` record:

C#

```
public record Order(int Items, decimal Cost);
```

The preceding positional record type declares two members at explicit positions. Appearing first is the `Items`, then the order's `Cost`. For more information, see Records.

The following code examines the number of items and the value of an order to calculate a discounted price:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
```

```
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate
discount on null order"),
        var someObject => 0m,
    };
```

The first two arms examine two properties of the `Order`. The third examines only the cost. The next checks against `null`, and the final matches any other value. If the `Order` type defines a suitable Deconstruct method, you can omit the property names from the pattern and use deconstruction to examine properties:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        ( > 10,  > 1000.00m) => 0.10m,
        ( > 5, > 50.00m) => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate
discount on null order"),
        var someObject => 0m,
    };
```

The preceding code demonstrates the *positional pattern* where the properties are deconstructed for the expression.

List patterns

You can check elements in a list or an array using a *list pattern*. A list pattern provides a means to apply a pattern to any element of a sequence. In addition, you can apply the *discard pattern* (`_`) to match any element, or apply a *slice pattern* to match zero or more elements.

List patterns are a valuable tool when data doesn't follow a regular structure. You can use pattern matching to test the shape and values of the data instead of transforming it into a set of objects.

Consider the following excerpt from a text file containing bank transactions:

Output

```
04-01-2020, DEPOSIT,    Initial deposit,            2250.00
04-15-2020, DEPOSIT,    Refund,                      125.65
04-18-2020, DEPOSIT,    Paycheck,                    825.65
04-22-2020, WITHDRAWAL, Debit,           Groceries,  255.73
05-01-2020, WITHDRAWAL, #1102,           Rent, apt, 2100.00
05-02-2020, INTEREST,                                 0.65
05-07-2020, WITHDRAWAL, Debit,           Movies,      12.57
04-15-2020, FEE,                                      5.55
```

It's a CSV format, but some of the rows have more columns than others. Even worse for processing, one column in the `WITHDRAWAL` type has user-generated text and can contain a comma in the text. A *list pattern* that includes the *discard* pattern, *constant* pattern and *var* pattern to capture the value processes data in this format:

C#

```
decimal balance = 0m;
foreach (string[] transaction in ReadRecords())
{
    balance += transaction switch
    {
        [_, "DEPOSIT", _, var amount]     => decimal.Parse(amount),
        [_, "WITHDRAWAL", .., var amount] => -decimal.Parse(amount),
        [_, "INTEREST", var amount]       => decimal.Parse(amount),
        [_, "FEE", var fee]               => -decimal.Parse(fee),
        _                                 => throw new
InvalidOperationException($"Record {string.Join(", ", transaction)} is not in
the expected format!"),
    };
    Console.WriteLine($"Record: {string.Join(", ", transaction)}, New balance:
{balance:C}");
}
```

The preceding example takes a string array, where each element is one field in the row. The `switch` expression keys on the second field, which determines the kind of transaction, and the number of remaining columns. Each row ensures the data is in the correct format. The discard pattern (`_`) skips the first field, with the date of the transaction. The second field matches the type of transaction. Remaining element matches skip to the field with the amount. The final match uses the *var* pattern to capture the string representation of the amount. The expression calculates the amount to add or subtract from the balance.

List patterns enable you to match on the shape of a sequence of data elements. You use the *discard* and *slice* patterns to match the location of elements. You use other patterns to

match characteristics about individual elements.

This article provided a tour of the kinds of code you can write with pattern matching in C#. The following articles show more examples of using patterns in scenarios, and the full vocabulary of patterns available to use.

See also

- Use pattern matching to avoid 'is' check followed by a cast (style rules IDE0020 and IDE0038)
- Exploration: Use pattern matching to build your class behavior for better code
- Tutorial: Use pattern matching to build type-driven and data-driven algorithms
- Reference: Pattern matching