# Pattern Matching Demo

## Start

Start with the TollCalculator.cs file from the Build 2019 demo. We did pretty much everything we could with patterns back then. But now there are several new improvements to patterns in C# 9.0.

## Type Patterns

Let’s look at the “CalculateToll” method. In places where we just wanted to match the type we still had to give a discard instead of an identifier:

public static decimal CalculateToll(object vehicle) =>

vehicle switch

{

Car c => c.Passengers switch

{

0 => 2.00m + 0.50m,

1 => 2.0m,

2 => 2.0m - 0.50m,

\_ => 2.00m - 1.0m,

},

Taxi { Fares: 0 } => 3.50m + 1.00m,

Taxi { Fares: 1 } => 3.50m,

Taxi { Fares: 2 } => 3.50m - 0.50m,

Taxi \_ => 3.50m - 1.00m,

Bus(var c, var r) when ((double)r / c) < 0.50 => 5.00m + 2.00m,

Bus(var c, var r) when ((double)r / c) > 0.90 => 5.00m - 1.00m,

Bus \_ => 5.00m,

DeliveryTruck t when t.GrossWeightClass > 5000 => 10.00m + 5.00m,

DeliveryTruck t when t.GrossWeightClass < 3000 => 10.00m - 2.00m,

DeliveryTruck \_ => 10.00m,

\_ => throw new ArgumentException(

message: "Not a known vehicle type", paramName: nameof(vehicle))

};

That’s no longer necessary. We can remove the discards from the type tests for “Taxi”, “Bus” and “DeliveryTruck” like so:

Taxi { Fares: 0 } => 3.50m + 1.00m,

Taxi { Fares: 1 } => 3.50m,

Taxi { Fares: 2 } => 3.50m - 0.50m,

Taxi => 3.50m - 1.00m,

Bus(var c, var r) when ((double)r / c) < 0.50 => 5.00m + 2.00m,

Bus(var c, var r) when ((double)r / c) > 0.90 => 5.00m - 1.00m,

Bus => 5.00m,

DeliveryTruck t when t.GrossWeightClass > 5000 => 10.00m + 5.00m,

DeliveryTruck t when t.GrossWeightClass < 3000 => 10.00m - 2.00m,

DeliveryTruck => 10.00m,

## Relational Patterns

Find the “GetTimeBand” method in the same file:

private static TimeBand GetTimeBand(DateTime timeOfToll)

{

int hour = timeOfToll.Hour;

if (hour < 6)

return TimeBand.Overnight;

else if (hour < 10)

return TimeBand.MorningRush;

else if (hour < 16)

return TimeBand.Daytime;

else if (hour < 20)

return TimeBand.EveningRush;

else

return TimeBand.Overnight;

}

In the past, there wasn’t really anything we could have done with pattern matching to improve this code. Now, with relational patterns, we can!

Place the editor caret on “if” and use the “Convert to ‘switch’ expression” IDE refactoring:

private static TimeBand GetTimeBand(DateTime timeOfToll)

{

int hour = timeOfToll.Hour;

return hour switch

{

< 6 => TimeBand.Overnight,

< 10 => TimeBand.MorningRush,

< 16 => TimeBand.Daytime,

< 20 => TimeBand.EveningRush,

\_ => TimeBand.Overnight

};

}

*Optional*: Of course, now we don’t need the local variable “hour” and we can inline it, showing the “fluentness” of switch expressions. And now we can turn the method body into an expression.

## Logical Patterns

Eureka! Relational patterns can also be used in the “CalculateToll” method. Let’s see if we can make the “DeliveryTruck” patterns a bit more succinct.

First, we need to introduce a nested switch expression for the “DeliveryTruck” cases that switches on “t.GrossWeightClass”. We’ll start by writing the following:

DeliveryTruck t => t.GrossWeightClass switch

{

},

Now, we can use relational patterns for each case like so (by cutting and pasting them):

DeliveryTruck t => t.GrossWeightClass switch

{

< 3000 => 10.00m - 2.00m,

> 5000 => 10.00m + 5.00m,

\_ => 10.00m

},

It’s unfortunate that the final case has to go at the end because it’s really *between­* the other two cases? Can we move it between them? Well, that introduces an error because the final case is now unreachable, but we can use an “and” pattern to remove the discard and make it work!

DeliveryTruck t => t.GrossWeightClass switch

{

< 3000 => 10.00m - 2.00m,

>= 3000 and <= 5000 => 10.00m,

> 5000 => 10.00m + 5.00m

},

There are both and, or and not patterns, and they are easily distinguished from the and’s, or’s and not’s of normal expressions by being spelled out as words.

A common case for not-patterns is to say that something is “not null”.

At the end of the big switch expression we have this case:

\_ => throw new ArgumentException(

message: "Not a known vehicle type", paramName: nameof(vehicle))

That really covers both nulls and unknown kinds of vehicles. Let’s split this out. Copy the line and edit both:

not null => throw new ArgumentException(

message: $"Not a known vehicle type: {vehicle}", paramName: nameof(vehicle))

null => throw new ArgumentNullException(nameof(vehicle)),

This lets us easily throw different kinds of exceptions, and in the non-null case we can safely use the object in the output.