

Pattern Matching



Filip Ekberg

Principal Consultant & CEO

@fekberg | fekberg.com

Pattern Matching in C#

First introduced in C# 7.0

Improved with each new version
of C# to get more patterns



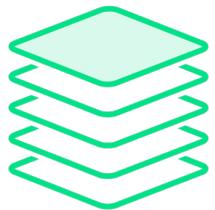
**Expect newer versions of C#
to introduce additional
patterns**



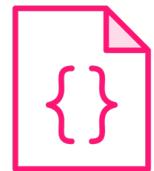
What Is Pattern Matching?



A way to write code that determines what an object is



Is the type a specific sub-class?
order is ProcessedOrder



Does it contain any specific values?

order is ProcessedOrder { IsReadyForShipment: true }



Does a value conform to a given range?

order is CancelledOrder { Total: <100 }



Before Pattern Matching



Before Pattern Matching

```
if(order.GetType() == typeof(ProcessedOrder))  
{  
}  
}
```



Before Pattern Matching

```
if(order.GetType() == typeof(ProcessedOrder))  
{  
    var processedOrder = order as ProcessedOrder;  
    if(processedOrder.IsReadyForShipment)  
    {  
    }  
}
```



Pattern Matching

```
if(order is ProcessedOrder { IsReadyForShipment: true })  
{  
}
```



Pattern matching can make
the code easier to read and
understand



Pattern Matching: Type Pattern



Pattern Matching: Type Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)  
{  
}  
}
```



Pattern Matching: Type Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)  
{  
}
```



Checks if the **ShippingProvider** is **set to an instance** that **matches the type** on the right hand side



Pattern Matching: Type Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)
```



```
if(order.ShippingProvider.GetType() == typeof(SwedishPostalServiceShippingProvider))  
{  
    var provider = order.ShippingProvider as SwedishPostalServiceShippingProvider;  
}
```



The goal is to make this easier
by using pattern matching



Pattern Matching: Type Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)  
{  
}  
}
```



Pattern Matching: Type Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)
{
    var provider = order.ShippingProvider as SwedishPostalServiceShippingProvider;

    if(provider.DeliverNextDay)
    {

    }
}
```



Pattern Matching

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)
```



Pattern Matching

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)
```



Define the **pattern(s)** on the **right hand side** of the **is operator**



Patterns can be easy and complex!



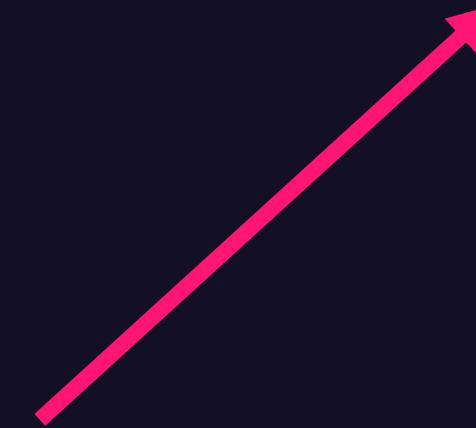
Pattern Matching: Declaration Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider )  
{  
}  
}
```



Pattern Matching: Declaration Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider provider)  
{  
}  
}
```



Capture the shipping **provider** as the **matched type** and make it **available** in the **if-statement**



Pattern Matching: Declaration Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider provider)
{
    var nextDayDelivery = provider.DeliverNextDay;
}
```



Property that **only exist** on the **SwedishPostalServiceShippingProvider**



**A combination of patterns
can produce a very complex
yet readable evaluation**



Patterns



- Type pattern
- Declaration pattern
- Constant pattern
- Relational pattern
- Logical pattern
- Property pattern
- Positional pattern
- Var pattern
- Discard pattern
- Parenthesized pattern



How to Use Patterns in C#



How to Use Patterns in C#

```
// Using the is operator  
if(GetOrder() is ProcessedOrder) {}
```



How to Use Patterns in C#

```
// Using the is operator  
if(GetOrder() is ProcessedOrder) {}
```

```
// Using a switch statement  
switch(GetOrder())  
{  
    case ProcessedOrder order: break;  
}
```



How to Use Patterns in C#

```
// Using the is operator  
if(GetOrder() is ProcessedOrder) {}
```

```
// Using a switch statement  
switch(GetOrder())  
{  
    case ProcessedOrder order when order.Total > 100: break;  
}
```



How to Use Patterns in C#

```
// Using the is operator  
if(GetOrder() is ProcessedOrder) {}
```

```
// Using a switch statement  
switch(GetOrder())  
{  
    case ProcessedOrder order when order.Total > 100: break;  
}
```

```
// Using a switch expression  
var result = GetOrder() switch {  
    ProcessedOrder => ""  
}
```



How to Use Patterns in C#

```
// Using the is operator
if(GetOrder() is ProcessedOrder) {}
```

```
// Using a switch statement
switch(GetOrder())
{
    case ProcessedOrder order when order.Total > 100: break;
}
```

```
// Using a switch expression
var result = GetOrder() switch {
    ProcessedOrder => ""
```



You will often combine
more than one pattern



Pattern Matching: Property Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)  
{  
}
```



Pattern Matching: Property Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider { FreightCost: >100 })  
{  
}  
}
```



Pattern Matching: Property Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider { FreightCost: >100 })  
{  
}  
}
```



This property pattern is a **nested (recursive) pattern**



Pattern Matching: Property Pattern

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider { FreightCost: >100 })  
{  
}  
}
```



This is a **relational pattern**



Pattern matching is powerful!

Use it wisely.



Introducing the Switch Expression

A combination of switch and expression bodied member

Always returns a value



Creating a Switch Expression



Creating a Switch Expression

```
decimal freightCost
```



Creating a Switch Expression

```
decimal freightCost = order.ShippingProvider switch {}
```



Creating a Switch Expression

```
decimal freightCost = order.ShippingProvider switch {}
```



```
switch (order.ShippingProvider)  
{  
}
```



Creating a Switch Expression

```
decimal freightCost = order.ShippingProvider switch  
{  
};
```



Creating a Switch Expression

```
decimal freightCost = order.ShippingProvider switch  
{  
    ←───────────────── Add all your patterns here  
};
```



Creating a Switch Expression

```
decimal freightCost = order.ShippingProvider switch  
{  
    SwedishPostalServiceShippingProvider  
};
```



Adds a type pattern



Creating a Switch Expression

```
decimal freightCost = order.ShippingProvider switch  
{  
    SwedishPostalServiceShippingProvider => 50m  
};
```



What to return when there is a match



Creating a Switch Expression

```
decimal freightCost = order.ShippingProvider switch
{
    SwedishPostalServiceShippingProvider => 50m
};
```



All expressions must return
types that are compatible
with each other



Multiple Patterns in a Switch Expression

```
decimal freightCost = order.ShippingProvider switch
{
    SwedishPostalServiceShippingProvider { NextDay: true } => 100m,
    SwedishPostalServiceShippingProvider => 50m,
    ShippingProvider => 200m
};
```



Multiple Patterns in a Switch Expression

```
decimal freightCost = order.ShippingProvider switch
{
    SwedishPostalServiceShippingProvider { NextDay: true } => 100m,
    SwedishPostalServiceShippingProvider => 50m,
    ShippingProvider => 200m
};
```

↑

Recursive pattern



Match all possible cases or add a default case



Discard Pattern

```
decimal freightCost = order.ShippingProvider switch
{
    SwedishPostalServiceShippingProvider { NextDay: true } => 100m,
    SwedishPostalServiceShippingProvider => 50m,
    _ => 200m
};
```



Discard Pattern

```
decimal freightCost = order.ShippingProvider switch
{
    SwedishPostalServiceShippingProvider { NextDay: true } => 100m,
    SwedishPostalServiceShippingProvider => 50m,
    _ => 200m
};
```



Discard Pattern

```
decimal freightCost = order.ShippingProvider switch
{
    SwedishPostalServiceShippingProvider { NextDay: true } => 100m,
    SwedishPostalServiceShippingProvider => 50m,
    _ => 200m
}; ↑
```

**Discard pattern will match
everything else including null**



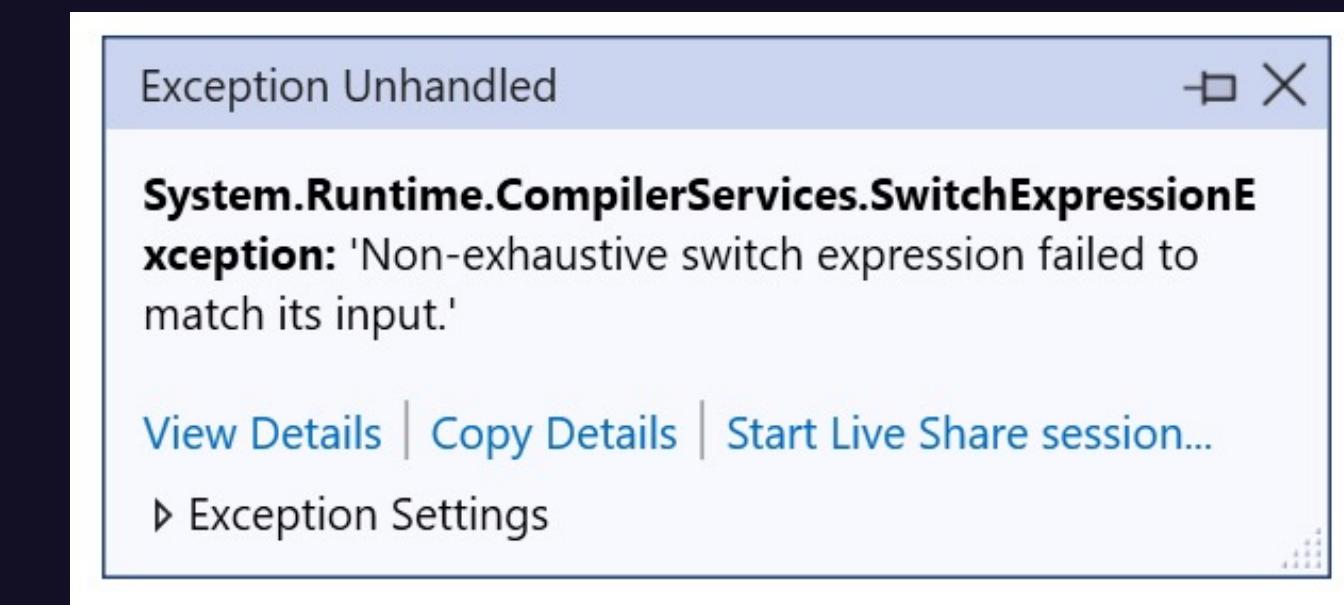
No Match and No Default Case

```
order.ShippingProvider = null;  
  
decimal freightCost = order.ShippingProvider switch  
{  
    SwedishPostalServiceShippingProvider => 50m  
};
```



No Match and No Default Case

```
order.ShippingProvider = null;  
  
decimal freightCost = order.ShippingProvider switch  
{  
    SwedishPostalServiceShippingProvider => 50m  
};
```



Order of Patterns

The order matters!

Most granular patterns at the top
of the switch expression



Unreachable Pattern

```
decimal freightCost = order.ShippingProvider switch
{
    _ => 100m,
    SwedishPostalServiceShippingProvider => 50m
};
```



```
class WarehouseManagementSystem.Domain.SwedishPostalServiceShippingProvider
```

CS8510: The pattern is unreachable. It has already been handled by a previous arm of the switch expression or it is impossible to match.



**There are still many
patterns to explore!**



Pattern Matching: Type Pattern

```
ShippingProvider provider = order switch
{
};
```



Pattern Matching: Type Pattern

```
ShippingProvider provider = order switch
{
    PriorityOrder    => new GlobalExpressShippingProvider(),
    LowPriorityOrder => new BoatShippingProvider(),
    -                  => new()
};
```



Pattern Matching: Type Pattern

```
object instance = GetInstance();
```



Pattern Matching: Type Pattern

```
object instance = GetInstance();
```

```
var result = instance switch  
{
```

```
};
```



Pattern Matching: Type Pattern

```
object instance = GetInstance();  
  
var result = instance switch  
{  
    string      => "This is a string",  
    int         => "This is an integer",  
    null        => "This is null",  
    _           => "This is everything else that is not null",  
};
```



Pattern Matching with Generics

```
decimal CalculateFor<T>(T instance) where T : ShippingProvider  
{  
}  
}  
}
```



Pattern Matching with Generics

```
decimal CalculateFor<T>(T instance) where T : ShippingProvider  
{  
    return instance switch  
    {  
        SwedishPostalServiceShippingProvider => 100m,  
    };  
}
```



Pattern Matching with Generics

```
decimal CalculateFor<T>(T instance) where T : ShippingProvider  
{  
    return instance switch  
    {  
        SwedishPostalServiceShippingProvider => 100m,  
        string      => 0m,  
        int         => 0m,  
        null        => 0m,  
        _           => 0m,  
    } ;  
}
```



Pattern Matching with Generics

```
decimal CalculateFor<T>(T instance) where T : ShippingProvider  
{  
    return instance switch  
    {  
        SwedishPostalServiceShippingProvider => 100m,  
        string      => 0m, ← The generic constraint is  
        int         => 0m, ← ignored and this is valid  
        null        => 0m,  
        -           => 0m,  
    } ;  
}
```



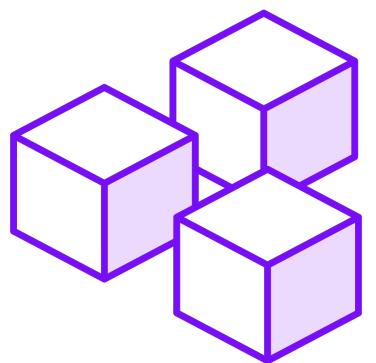
The compiler will let you
know if the pattern is not
applicable to the given type



Powerful and Useful Pattern



**What sub-class is the type?
Is the order cancelled or shipped?**



**Is the instance set to a value?
Checked during runtime!**



**Capture in a local variable as the actual type!
Used together with the declaration pattern**



Pattern Matching: Positional Pattern



Pattern Matching: Positional Pattern

```
if(order is (100, true)) { ... }
```



Pattern Matching: Positional Pattern

```
if(order is (100, true)) { ... }
```



Pattern Matching: Positional Pattern

```
if(order is (100, true)) { ... }
```



Can contain **any patterns**
These are constant patterns



Positional Pattern with Deconstruct

```
if(order is (100, true)) { ... }
```

```
public void Deconstruct(out decimal total, out bool ready)  
{ ... }
```



Positional Pattern with Deconstruct

```
total    ready  
↓      ↓  
if(order is (100, true)) { ... }
```

```
public void Deconstruct(out decimal total, out bool ready)  
{ ... }
```



**Do you have to provide
constants for all
deconstructed values?**

No!



Positional Pattern with a Discard

Discard the value and allow **any value**

```
if(order is (_, true)) { ... }
```

```
public void Deconstruct(out decimal total, out bool ready)  
{ ... }
```



Positional Pattern with a Discard

What is this?



```
if(order is (_, true)) { ... }
```

```
public void Deconstruct(out decimal total, out bool ready)  
{ ... }
```



Positional Pattern with a Discard

```
if(order is (_, ready: true)) { ... }
```

```
public void Deconstruct(out decimal total, out bool ready)  
{ ... }
```



Can you use the property pattern instead?

That may be even more readable in less complex situations



**What is deconstructed may
be the result of a complex
computation**



You can use the positional
pattern to match tuple
values



Constant Patterns



Numbers

```
{ total: 100.50m }
```



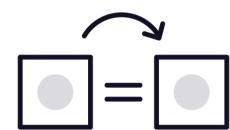
Characters or string literals

```
{ name: "Filip Ekberg" }
```



Enum

```
{ status: OrderStatus.Shipped }
```



Constant fields

```
{ name: DefaultShippingProviderName }
```



Null

```
{ name: null }
```



You don't have to use the
property pattern together
with the type pattern



An Alternative Null Check



An Alternative Null Check

```
if(order is { })  
{  
}  
}
```



An Alternative Null Check

Empty property pattern!
This will verify that it is not null



```
if(order is { })  
{  
}
```



Properties of Properties

```
if(order is { })  
{  
}  
}
```



Properties of Properties

```
if(order is { ShippingProvider.FreightCost: >100 })  
{  
}  
}
```



Properties of Properties

```
if(order is { ShippingProvider.FreightCost: >100 })  
{  
}
```



Will first ensure that ShippingProvider is not null!



```
var cost = order.ShippingProvider switch
{
    SwedishPostalServiceShippingProvider
    { DeliverNextDay: true } provider => provider.Freight + 50m
}
```

Combination of Patterns

Type pattern

Property pattern

Constant pattern

Declaration pattern

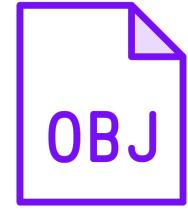


Multiple Property Patterns

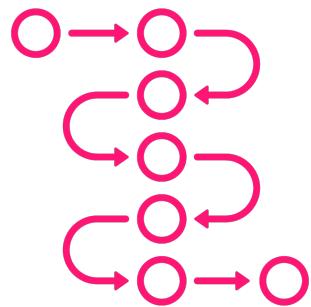
```
if (order is {
    ShippingProvider: SwedishPostalServiceShippingProvider,
    Total: >100
})
{ }
```



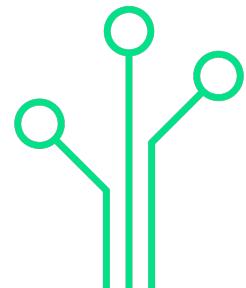
Logical Patterns



Negated not
order **is** not null



Conjunctive and
order **is** (total: >50 **and** <100, _)



Disjunctive or
order **is** CancelledOrder **or** ShippedOrder



Negate Any Pattern

```
if(order is not CancelledOrder) { ... }
```

```
if(order is not { Total: <100 }) { ... }
```

```
if(order is not (_, true)) { ... }
```



Negate Any Pattern

```
if(order is not CancelledOrder) { ... }
```

```
if(order is not { Total: <100 }) { ... }
```

```
if(order is not (_, false)) { ... }
```

Don't negate the Boolean



Negating a Property Pattern

```
if(order is not { ShippingProvider.FreightCost: >100 })  
{ ... }
```



Negating a Property Pattern

```
if(order is not { ShippingProvider.FreightCost: >100 })  
{ ... }
```



Negating a Property Pattern

```
if(order is not { ShippingProvider.FreightCost: >100 })  
{ ... }
```



First evaluates this entire pattern to check null and the property values.

Then negates that!



Relational Pattern



Relational Pattern

```
if(order is { Total: <100 }) { ... }
```



A **constant value** matched
against the related **property**
or **position**



Define an Allowed Range with Patterns

```
if(order is { Total: >50 and <=100 }) { ... }
```



How do we exclude shipped and cancelled orders?



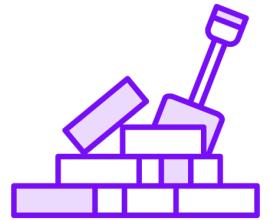
Try to compare this to an approach that does not use pattern matching



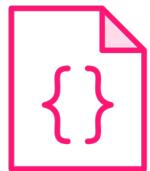
Pattern Matching



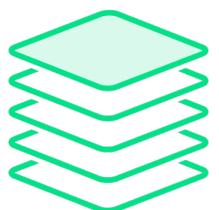
What a type is and what it isn't



How can this be deconstructed?



Does it have a property with a given value?

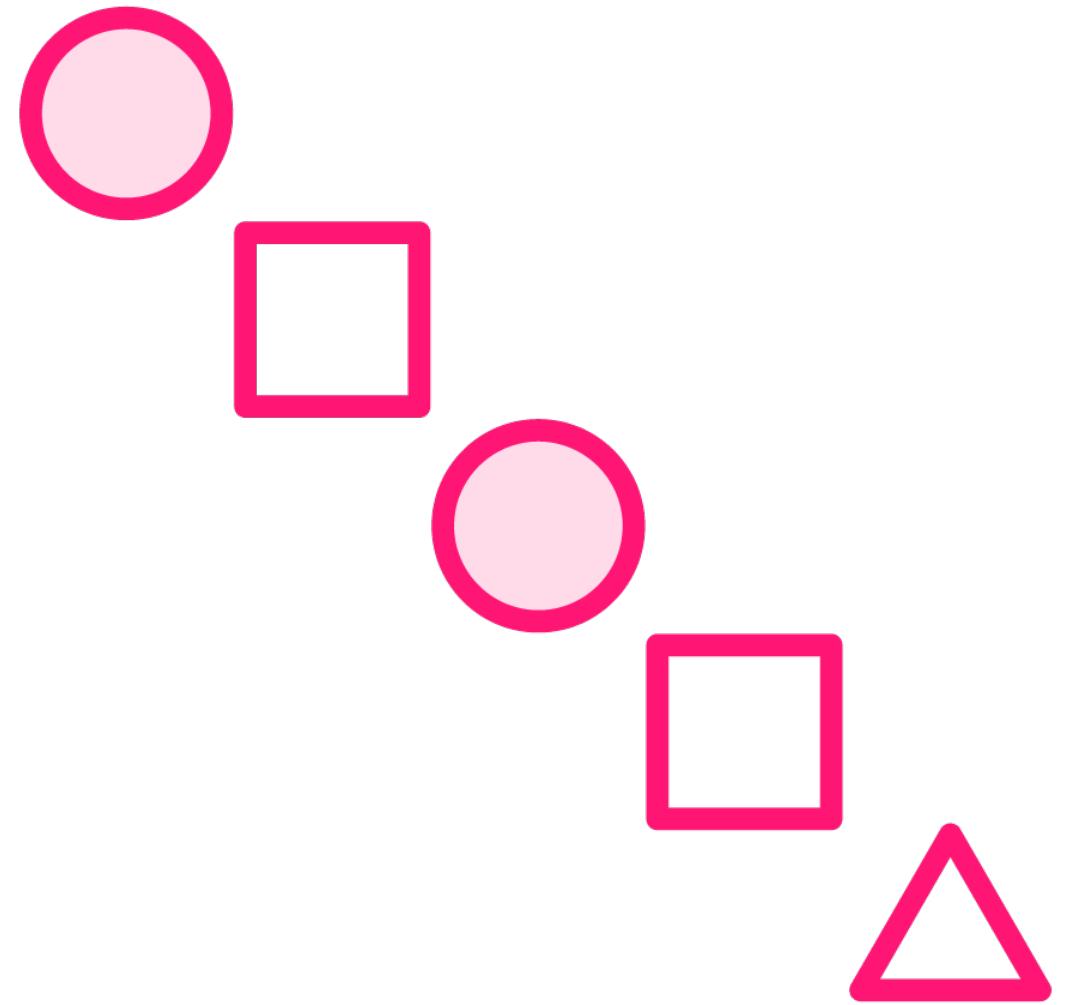


Which sub-class has been used?



**Pattern matching helps to
express the intent in
a clear way**





Type
Declaration
Constant
Relational
Logical
Property
Positional
Var
Discard
Parenthesized



Pattern: Type

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider)  
{  
}  
}
```



Pattern: Declaration

```
if(order.ShippingProvider is SwedishPostalServiceShippingProvider provider)  
{  
}  
}
```



Capture a local variable
as the actual type



Pattern: Constant

```
if(order is { Total: 100 })  
{  
  
}  
  
if(provider is { Name: "Swedish Postal Service" })  
{  
  
}
```



Pattern: Constant

```
if(order is { Total: 100 })  
{  
}  
  
if(provider is { Name: "Swedish Postal Service" })  
{  
}
```

Constants



Pattern: Relational

```
if(order is { Total: >100 })  
{  
}  
}
```



Pattern: Logical



Pattern: Logical

```
// Negated not  
if(order is not CancelledOrder) { }
```



Pattern: Logical

// Negated not

```
if(order is not CancelledOrder) { }
```

// Disjunctive or

```
if(order is not (CancelledOrder or ShippedOrder)) { }
```

// Conjunctive and

```
if(order is { Total: >100 and <1000 }) { }
```



Pattern: Logical

```
// Negated not  
if(order is not CancelledOrder) { }  
  
// Disjunctive or  
if(order is not (CancelledOrder or ShippedOrder)) { }  
  
// Conjunctive and  
if(order is { Total: >100 and <1000 }) { }
```



Pattern: Negated Not

```
if(order is not null) { }
```



Pattern: Property

```
if(order is { ShippingProvider: SwedishPostalService { FreightCost: 50m } })  
{ }
```



Pattern: Positional

```
if(order is (>100, true)) { }
```

```
if((total, ready) is (>100, true)) { }
```



Pattern: Var

```
var result = GetOrder() switch
{
    var match => ""
};
```



Pattern: Discard

```
var result = GetOrder() switch
{
    (_, true) => "",
    _ => ""
};
```



Pattern: Parenthesized

```
var result = GetOrder() switch
{
    Priority or ((total: >100, _) and not (Cancelled or Shipped)) => "",
    _ => ""
};
```



Case Guards

```
var result = order switch
{
    PriorityOrder when HasAvailability() => "",
    _ => ""
};
```



A powerful language feature that is constantly improved



Up Next:

Record Types

