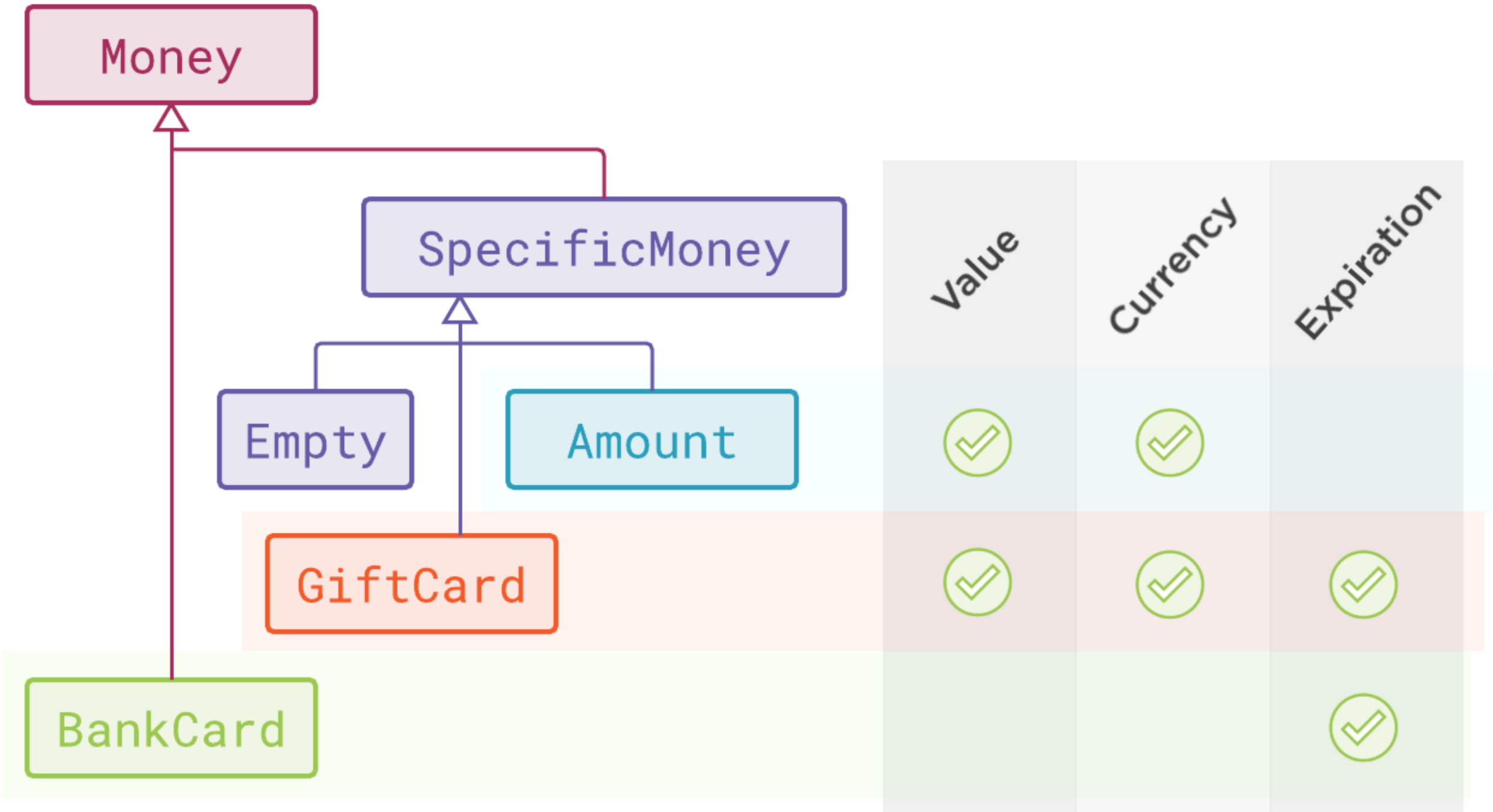
Pattern Matching with C# 7

Zoran Horvat
CEO AT CODING HELMET

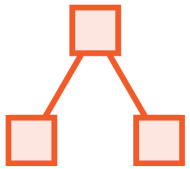
@zoranh75 http://csharpmentor.com


Money
SpecificMoney
Empty
Amount
GiftCard
BankCard
Value
Currency
Expiration

```
public class Wallet
{
  private (Amount paid, Money remaining) Pay(Money money, Amount amount) { }
}
```

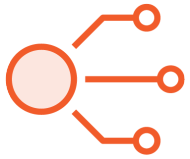
Money type	Condition	Returns
`Amount`	Mismatched currency	`(0, money)`
`Amount`	Nothing remains	`(amount, 0)`
`Amount`	Some money remains	`(amount, money – amount)`
`GiftCard`	Mismatched currency	`(0, money)`
`GiftCard`	Card expired	`(0, 0)`
`GiftCard`	Nothing remains	`(amount, 0)`
`GiftCard`	Some money remains	`(amount, money – amount)`
`BankCard`	Card expired	`(0, 0)`
`BankCard`	Card valid	`(amount, money)`

Pattern Matching in C#



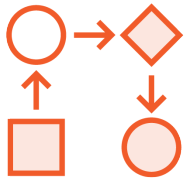
`if` instruction

```
if (money is BankCard card && card.ValidBefore...) ...
```



`switch` instruction

```
case BankCard card when card.ValidBefore...
```



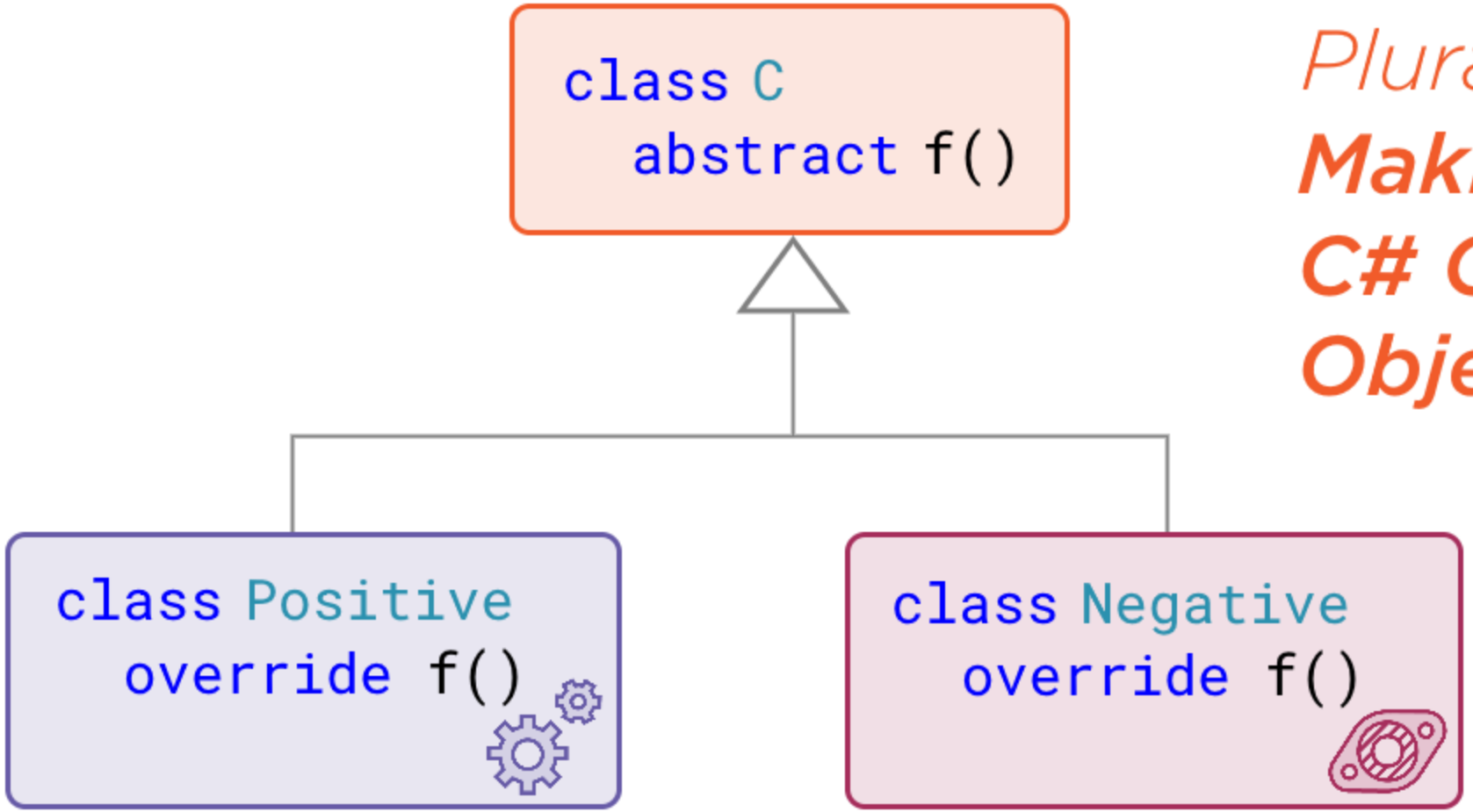
Ternary operator

```
money is BankCard card ? resultA : resultB;
```

Inappropriate Branching

```
if (condition)
  positive
else
  negative
```

Inappropriate Branching



Pluralsight course:
Making Your C# Code More Object-oriented

Inappropriate Branching

```
class C
  f()
    if (condition)
      positive();
    else
      negative();
```

Pluralsight course:
Making Your C# Code More Object-oriented

Allow branching if it depends on public method's input

Branching vs. Guarding

Branched execution

```
if (condition)
  full-operation-A;
else
  full-operation-B;
```

Guarded execution

lightweight exit

```
if (not-applicable) return fast;
full-operation;
return heavy-result;
```

Guard Clauses

If-throw pattern

```
if (arg < 0) throw new ArgumentException();
full-operation
```

If-return pattern in non-void methods

```
if (arg < 0) return 0;
full-operation
```

If-return pattern in void methods

```
if (arg < 0) return;
full-operation
```

Understanding the Ternary Operator

```
var x = condition ? A : B
```

ternary operator ?:

```
var x;
if (condition)
  x = A;
else
  x = B;
```

Understanding the Ternary Operator

```
var x = condition ? A : B
```

Are these two really equally important?

Or do they only look like they are?

```
var x;
if (condition)
  x = A;
else
  x = B;
```

Understanding the Ternary Operator

```
var x = condition ? A : B
```

```
var x;
if (condition)
  x = lightweightA;
else
  x = heavyweightB;
```

Understanding the Ternary Operator

```
var x = condition ? A : B
```

```
if (not-applicable)
  return lightweightA;
return heavyweightB;
```

Understanding the Ternary Operator

```
var x = not-applicable ? lightweightA : heavyweightB;
```

applicable in assignment

```
if (not-applicable)
  return lightweightA;
return heavyweightB;
```

Understanding the Ternary Operator

```
var x = not-applicable ? lightweightA : heavyweightB;
```

applicable in assignment

Assignment from switch still not possible (in C# 7):

```
var x =
  switch (money)
  {
    case not-applicable => lightweightA;
    default => heavyweightB;
  }
```

Assignment Instruction

`target = value` – assigns `value` to `target`

Assignment Instruction

l-value

Anything that appears on the left

- Variable
- Object's field
- Settable property

r-value

Anything that appears on the right

- All the l-values
- Function call
- Function name

```
Func<int, int> f = obj.MyImpl;
```

- `switch` instruction
  (still not in C# 7)

Ternary Operator with Patterns

```
var x = not-applicable ? lightweightA : heavyweightB;
```

Boolean conditions

```
var x =
  conditionA   ? resultA
  : conditionB ? resultB
  : conditionC ? resultC
  ...
  : defaultResult;
```

And how about using **patterns** in ternary operator?

Ternary Operator with Patterns

Ternary operator with Boolean conditions

```
public virtual Amount Subtract(Amount other) =>
  other == null ? throw new ArgumentNullException(nameof(other))
  : other.Currency != this.Currency
      ? throw new ArgumentException("Mismatched currency.")
  : other.Value > this.Value
      ? throw new ArgumentException("Insufficient funds.")
  : new Amount(this.Currency, this.Value - other.Value);
```

Ternary operator with pattern matching expressions

```
public Money PayableAt(Money money, Amount expense, Timestamp time) =>
    money is GiftCard gift && gift.ValidBefore.CompareTo(time) < 0
        ? Amount.Zero(expense.Currency)
    : money is BankCard card && card.ValidBefore.CompareTo(time) < 0
        ? Amount.Zero(expense.Currency)
    : money;
```

Ternary Operator with Patterns

```
var x =
  patternA   ? resultA
  : patternB ? resultB
  : patternC ? resultC
  ...
  : defaultResult;
```

We can list patterns in an impossible order

Ternary operator does not produce compile-time warnings

The **`switch`** instruction causes compile-time error when one case masks the other

To Match or Not to Match

Choose either functional design
or polymorphic execution

Apply pattern matching to implement a function
for all affected types without using inheritance

With class hierarchies, pattern matching
will probably make your code messier

Summary

Pattern matching in C# 7

- Type matching expressions
- Capturing strongly typed variables
- Functional-style pattern matching

New pattern matching syntax

- `if` instruction with type matching
- `switch` instruction with type patterns
- Ternary operator with type matching

Branching is back in the game!

Summary

When to match type patterns

- When data are separate from behavior
- Bad idea when mixing objects and functional design
- Mixing paradigms can lead to implicit execution paths
- Don't force the caller implement features (in any coding style)

In object-oriented code

- Rely on virtual functions and polymorphic execution at run time

Next module:
Metaprogramming with Extension Methods