Writing Purely Functional Code in C#

UNDERSTANDING THE NEED FOR DISCRIMINATED UNIONS

Zoran Horvat
CEO AT CODING HELMET
@zoranh75 http://csharpmentor.com

Functional Concepts in C#

Func delegates

```
delegate int AddTo(int x);

class BusinessModel
{
  void Recalculate(AddTo adder) { ... }
}
```

Functional Concepts in C#

Func delegates

```
delegate int AddTo(int x);

class BusinessModel
{
  void Recalculate(AddTo adder) { ... }
}

class BusinessModel
{
  void Recalculate(Func<int, int> adder) { ... }
}
```

Functional Concepts in C#

Func delegates

Lambdas

```
class BusinessModel
{
  public void Recalculate(Func<int, int> adder);
}
```

Functional Concepts in C#

Func delegates

Lambdas

```
class BusinessModel
{
  public void Recalculate(Func<int, int> adder);
}

class Worker
{
  int Adder(int x) { return x + 2; }

  void DoStuff()
  {
    ...
    model.Recalculate(this.Adder);
  }
}
```

Functional Concepts in C#

Func delegates

Lambdas

```
class BusinessModel
{
  public void Recalculate(Func<int, int> adder);
}

class Worker
{


  void DoStuff()
  {
    ...
    model.Recalculate(x => x + 2);
  }
}
```

Functional Concepts in C#

Func delegates

Lambdas

LINQ

```
List<Car> favoriteCars = new List<Car>();
foreach (Vehicle v in city.RegisteredVehicles)
{
  Car aCar = v as Car;
  if (aCar != null && aCar.Color == Colors.Red)
  {
    favoriteCars.Add(aCar);
  }
}
return favoriteCars;
```

Functional Concepts in C#

Func delegates

Lambdas

LINQ

```
List<Car> favoriteCars = new List<Car>();
foreach (Vehicle v in city.RegisteredVehicles)
{
  Car aCar = v as Car;
  if (aCar != null && aCar.Color == Colors.Red)
  {
    favoriteCars.Add(aCar);
  }
}
return favoriteCars;

Car goodOne = city.RegisteredVehicles
  .OfType<Car>()
  .Where(car => car.Color == Colors.Red)
  .First();
```

Functional Concepts in C#

Func delegates

Lambdas

LINQ

Extension methods

```
static class CarObsessionBehavior
{
    public Car GetBestCar(this City from)
    {
        ...
    }
}
```

Functional Concepts in C#

Func delegates

Lambdas

LINQ

Extension methods

Anonymous types

```
var x = new
{
    Vehicle = redCar,
    Owner = "Joe"
};
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals

```
var tuple = (redCar, "Joe");
(Car vehicle, string owner) = (redCar, "Joe");
var (vehicle, owner) = (redCar, "Joe");

Tuple<Car, string> tuple =
  Tuple.Create(redCar, "Joe");
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals

```
var tuple = (redCar, "Joe");
(Car vehicle, string owner) = (redCar, "Joe");
var (vehicle, owner) = (redCar, "Joe");

Tuple<Car, string> tuple =
  Tuple.Create(redCar, "Joe");
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching

```
if (vehicle is Car car && car.DoorsCount == 5) ...

vehicle is Car car && car.DoorsCount == 5
  ? car.Color
  : Colors.Grey;

switch (vehicle)
{
  case Car car:
    return car.DoorsCount;
  case Truck truck:
    return truck.Cabin.Doors.Count();
  case Boat _: return 1;
  default: return 0;
}
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference

```
city.Vehicles
  .Select<Vehicle, Person>(v => v.OwnedBy)
  .Select<Person, Color>(p => p.FavoriteColor)
  .Select<Color, double>(c => c.DistanceFrom(red))
  .Average();
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference

```
city.Vehicles
   .Select<Vehicle, Person>(v => v.OwnedBy)
   .Select<Person, Color>(p => p.FavoriteColor)
   .Select<Color, double>(c => c.DistanceFrom(red))
   .Average();
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference

```
city.Vehicles
  .Select(v => v.OwnedBy)
  .Select(p => p.FavoriteColor)
  .Select(c => c.DistanceFrom(red))
  .Average();
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference

```
class Treadmill
{
  public TRes CountMeIn<TSub, TMsr, TRes>
    (TSub subject, TMsr bodyMeasure) { ... }
}

treadmill.CountMeIn<_, _, DateTime>(name, weight);
```

won't compile

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference

```
class Treadmill
{
  public TRes CountMeIn<TSub, TMsr, TRes>
    (TSub subject, TMsr bodyMeasure) { ... }
}

treadmill.CountMeIn<_, _, DateTime>(name, weight);

treadmill.CountMeIn<string, float, DateTime>
  (name, weight);
```

compiles

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference

```
class Treadmill
{
  public TRes CountMeIn<TSub, TMsr, TRes>
    (TSub subject, TMsr bodyMeasure) { ... }
}

treadmill.CountMeIn<_, _, DateTime>(name, weight);

treadmill.CountMeIn<string, float, DateTime>
  (name, weight);

DateTime when = treadmill.CountMeIn(name, weight);
```

won't compile

Functional Concepts in C#

- Func delegates
- Lambdas
- LINQ
- Extension methods
- Anonymous types
- Tuple literals
- Pattern matching
- Type inference

```
class Treadmill
{
  public TRes CountMeIn<TSub, TMsr, TRes>
    (TSub subject, TMsr bodyMeasure) { ... }
}

treadmill.CountMeIn<_, _, DateTime>(name, weight);

treadmill.CountMeIn<string, float, DateTime>
  (name, weight);

DateTime when = treadmill.CountMeIn(name, weight);

DateTime when =
  treadmill.CountMeIn<string, float, DateTime>
    (name, weight);
```

compiles

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference

```
class Treadmill
{
  public TRes CountMeIn<TSub, TMsr, TRes>
    (TSub subject, TMsr bodyMeasure) { ... }
}

treadmill.CountMeIn<_, _, DateTime>(name, weight);

treadmill.CountMeIn<string, float, DateTime>
  (name, weight);

DateTime when = treadmill.CountMeIn(name, weight);

DateTime when =
  treadmill.CountMeIn<string, float, DateTime>
    (name, weight);
```

Functional Concepts in C#

Func delegates
Lambdas
LINQ
Extension methods
Anonymous types
Tuple literals
Pattern matching
Type inference
Generics

```
public T FindFittest<T>(IEnumerable<T> seq)
  where T : IComparable<T> =>
  seq.Aggregate((best, cur) =>
    cur.CompareTo(best) > 0 ? cur : best);

public T FindBeautiest<T>(IEnumerable<T> seq)
  where T : IEquatable<Color> =>
  seq.First(x => x.Equals(Colors.Red));
```

Writing Purely Functional Code

Function *evaluates*,
it does not *execute*

Result depends on *arguments*,
not on *state*

Values do *not change*, etc.

Why in C#?

Compare to F#

Writing Purely Functional Code

Previous course:

Making Your C# Code More Functional

Transforming an object model
to accommodate functional elements

Writing Purely Functional Code

Previous course:

Making Your C# Code More Functional

Transforming an object model
to accommodate functional elements

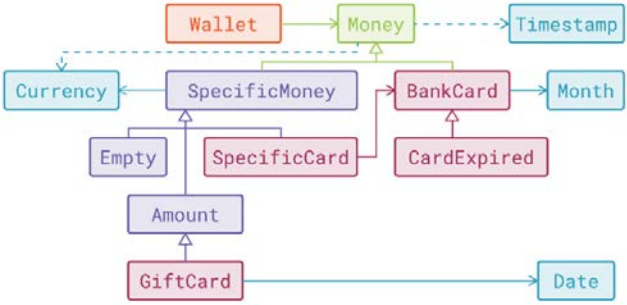
This course:

Writing Purely Functional Code in C#

Design a functional model from scratch

Making Your C# More Functional

OO falling short



Making Your C# More Functional

OO falling short

Object filters

```
this.Content
  .On(Timestamp.Now)
  .Of(expense.Currency)
  .Take(expense.Value)
  .ToList();
```

Making Your C# More Functional

OO falling short

Object filters

FP theory

Pure functions,

Referential transparency,

Memoization, etc.

Making Your C# More Functional

OO falling short

Object filters

FP theory

Pattern matching

```
switch (money)
{
  case Amount amount
    when amount.Currency == expense.Currency:
    return amount.Value >= expense.Value;
  case GiftCard gift
    when gift.ValidBefore.CompareTo(now) >= 0 &&
         gift.Currency == expense.Currency:
    return gift.Value >= expense.Value;
  case BankCard card
    when card.ValidBefore.CompareTo(now) >= 0:
    return true;
  default:
    return false;
}
```

Making Your C# More Functional

OO falling short

Object filters

FP theory

Pattern matching

```
switch (money)
{
  case Amount amount
    when amount.Currency == expense.Currency:
    return amount.Value >= expense.Value;

  case GiftCard gift
    when gift.ValidBefore.CompareTo(now) >= 0 &&
         gift.Currency == expense.Currency:
    return gift.Value >= expense.Value;

  case BankCard card
    when card.ValidBefore.CompareTo(now) >= 0:
    return true;

  default:
    return false;
}
```

Making Your C# More Functional

- OO falling short
- Object filters
- FP theory
- Pattern matching
- Function composition

$$(g \circ f)(x) = g(f(x))$$

$$z = p.f(x).g();$$

$$z = x.f().g();$$

Making Your C# More Functional

OO falling short

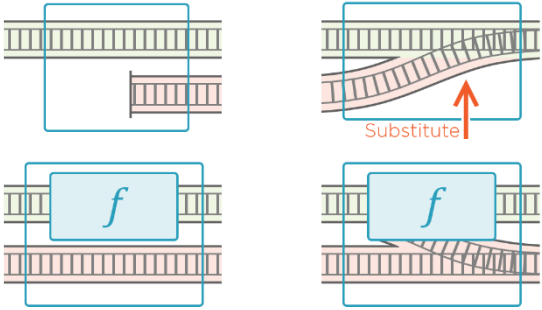
Object filters

FP theory

Pattern matching

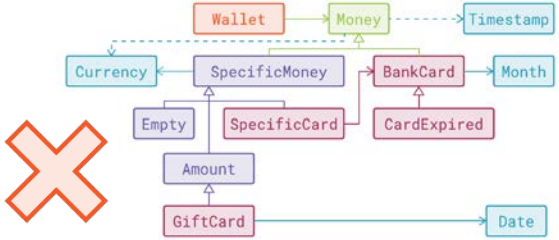
Function composition

Railway-oriented programming



What Follows in This Course

Types vs. classes



No class hierarchies

No member functions

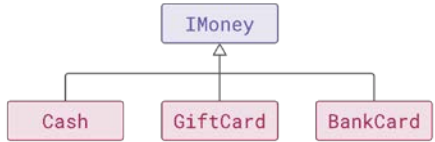
No polymorphism

What Follows in This Course

Types vs. classes

Discriminated unions

```
type Money =
  Cash | GiftCard | BankCard;
```



What Follows in This Course

Types vs. classes

Discriminated unions

Functional functions

What is the function in FP?

Function vs. object method

Function composition

Higher-order functions

Partial function application, etc.

Applying functions to C#

What Follows in This Course

Types vs. classes

Discriminated unions

Functional functions

Value -typed semantic

Custom value types

Tuples

Record types

Upcoming support in C#

What Follows in This Course

Types vs. classes

Discriminated unions

Functional functions

Value -typed semantic

Control flow

Pattern matching

The only method of controlling execution flow in functional programming

Looping implemented via LINQ

What Follows in This Course

Types vs. classes

Discriminated unions

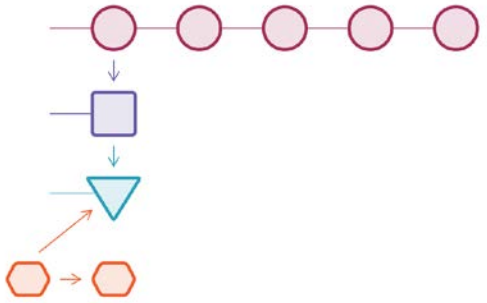
Functional functions

Value -typed semantic

Control flow

Sequences

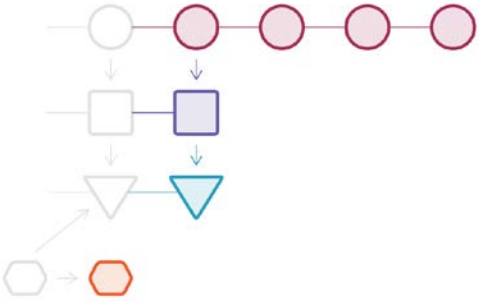
Functional sequence processing with LINQ



What Follows in This Course

Types vs. classes
Discriminated unions
Functional functions
Value -typed semantic
Control flow
Sequences

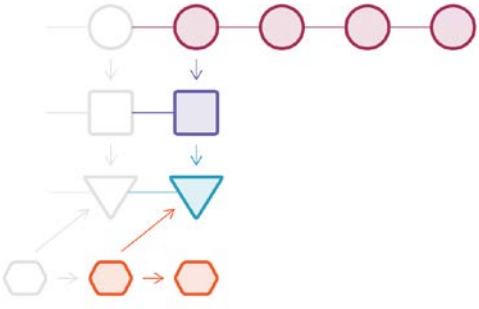
Functional sequence processing with LINQ



What Follows in This Course

Types vs. classes
Discriminated unions
Functional functions
Value -typed semantic
Control flow
Sequences

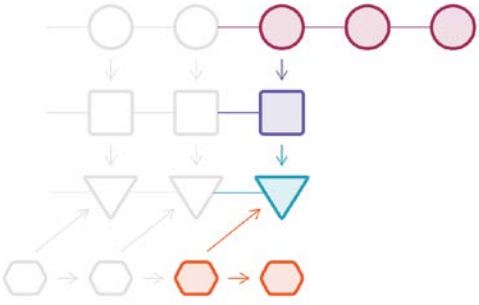
Functional sequence processing with LINQ



What Follows in This Course

Types vs. classes
Discriminated unions
Functional functions
Value -typed semantic
Control flow
Sequences

Functional sequence processing with LINQ



What Follows in This Course

Types vs. classes

Discriminated unions

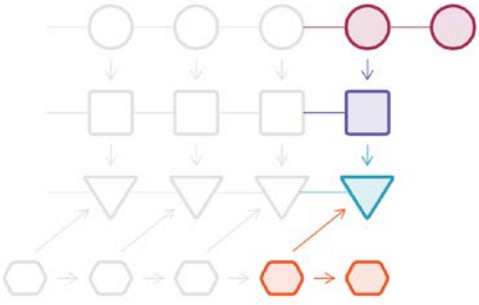
Functional functions

Value -typed semantic

Control flow

Sequences

Functional sequence processing with LINQ



What Follows in This Course

Types vs. classes

Discriminated unions

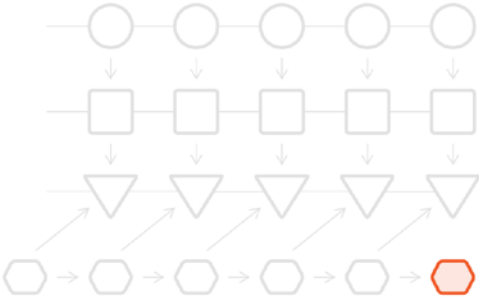
Functional functions

Value -typed semantic

Control flow

Sequences

Functional sequence processing with LINQ



What Follows in This Course

Types vs. classes

Discriminated unions

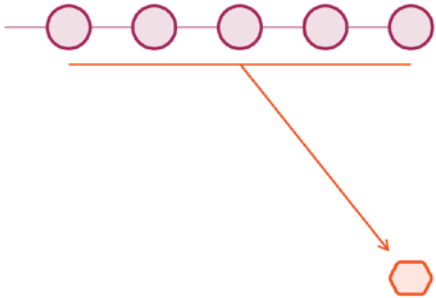
Functional functions

Value -typed semantic

Control flow

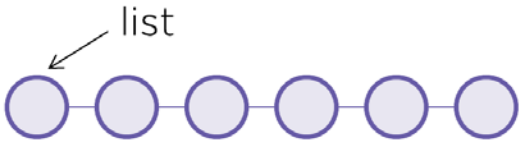
Sequences

Functional sequence processing with LINQ



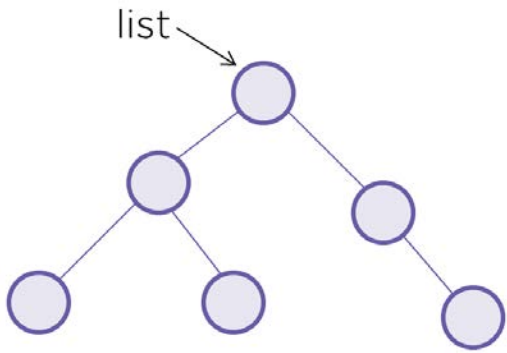
What Follows in This Course

- Types vs. classes
- Discriminated unions
- Functional functions
- Value -typed semantic
- Control flow
- Sequences
- Immutable collections



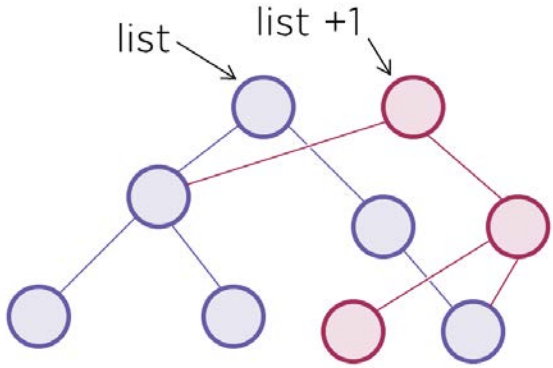
What Follows in This Course

Types vs. classes
Discriminated unions
Functional functions
Value -typed semantic
Control flow
Sequences
Immutable collections



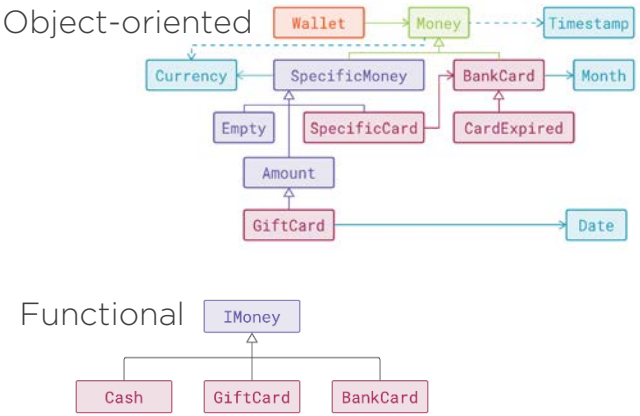
What Follows in This Course

- Types vs. classes
- Discriminated unions
- Functional functions
- Value -typed semantic
- Control flow
- Sequences
- Immutable collections

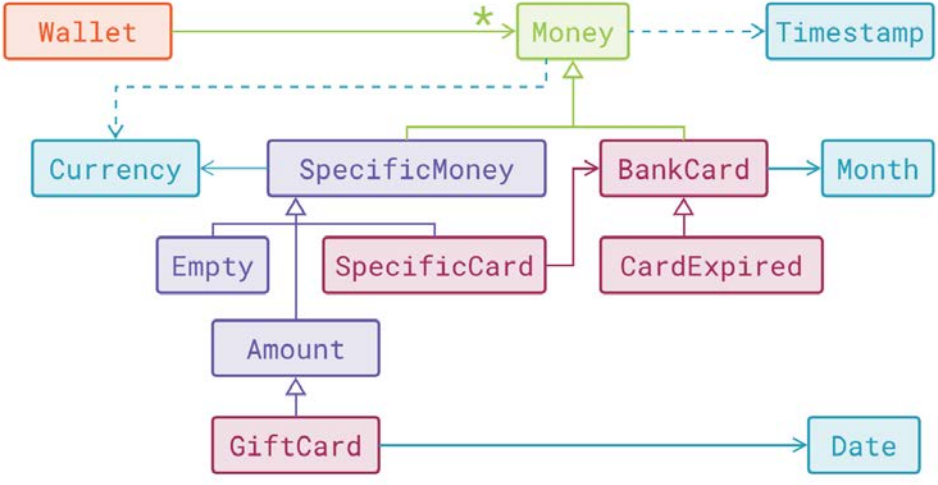


What Follows in This Course

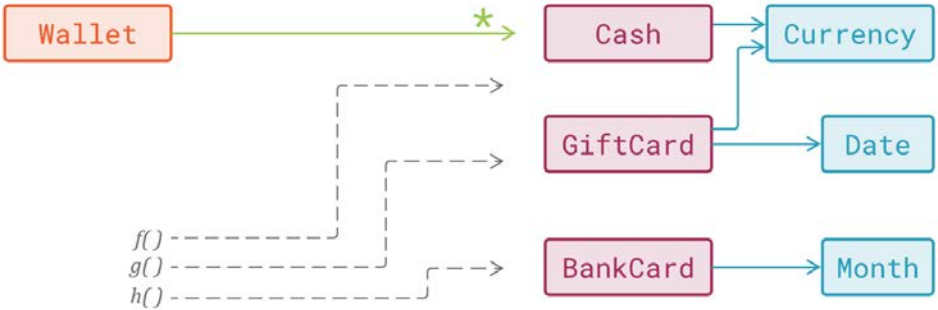
Types vs. classes
Discriminated unions
Functional functions
Value -typed semantic
Control flow
Sequences
Immutable collections



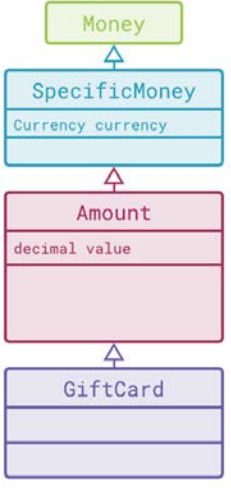
Example Domain



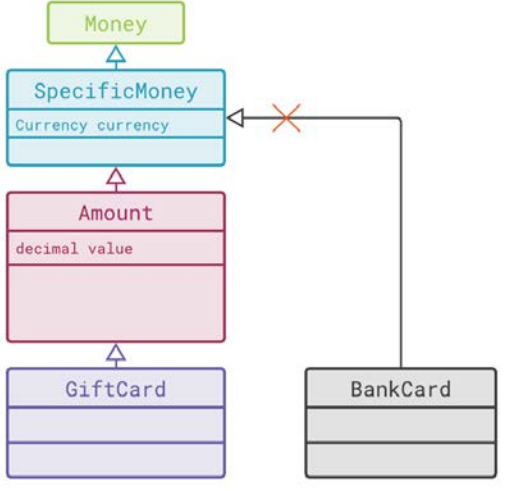
Example Domain



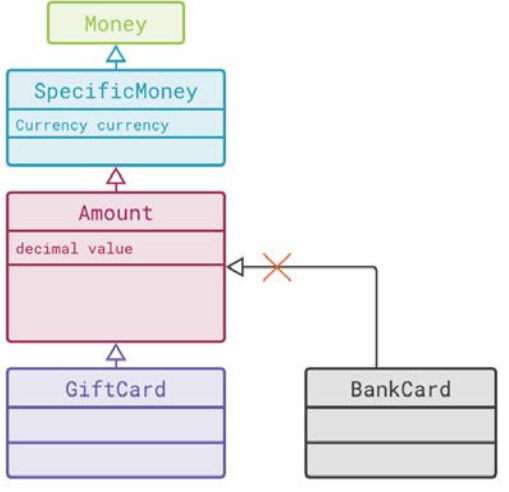
Understanding Class Inheritance



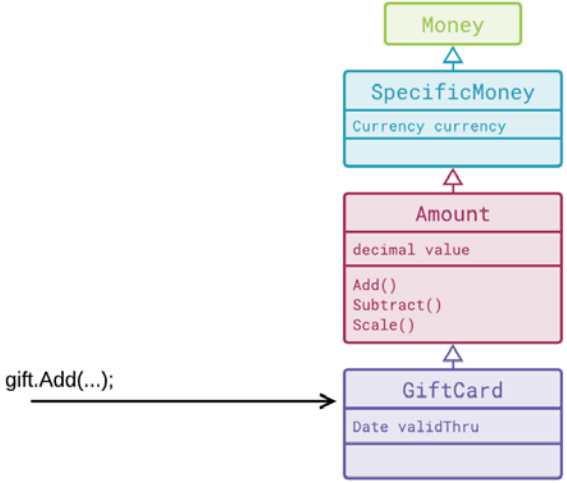
Understanding Class Inheritance



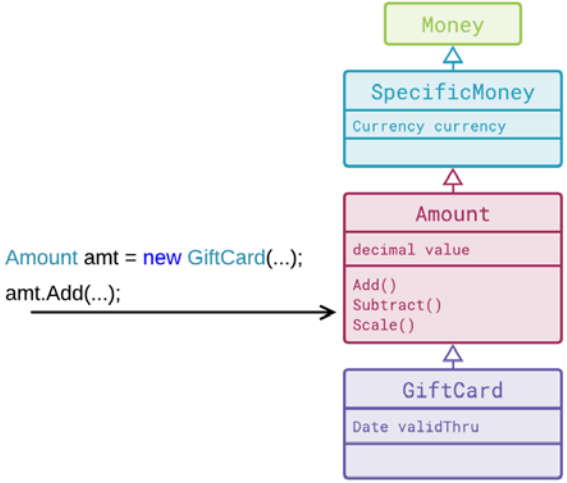
Understanding Class Inheritance



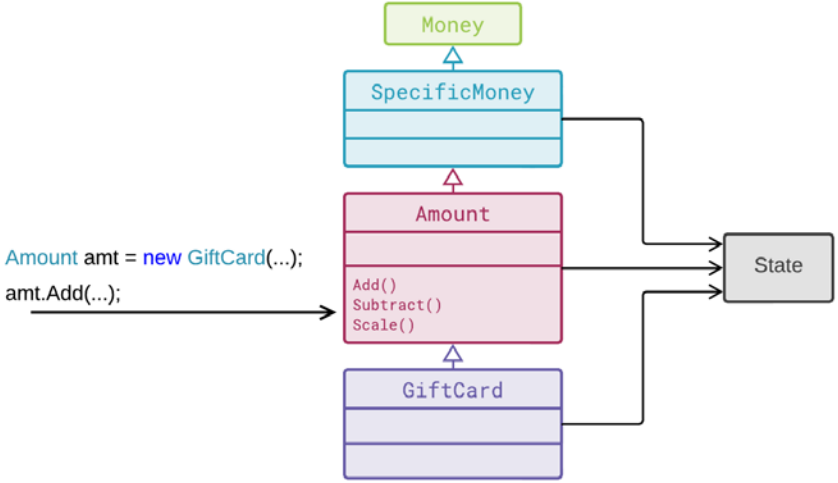
Understanding Class Inheritance



Understanding Class Inheritance



Understanding Class Inheritance



Summary

Discriminated unions

- Constraining a set of types
- Function defined on all of them
- Completeness verified by compiler

Interface inheritance in C#

- Still no discriminated unions in C#

In the rest of the course

- Applying functional programming concepts directly to C#
- Leverage latest C# syntax
- Start off with lambdas

Next module:
Functions as objects, big time