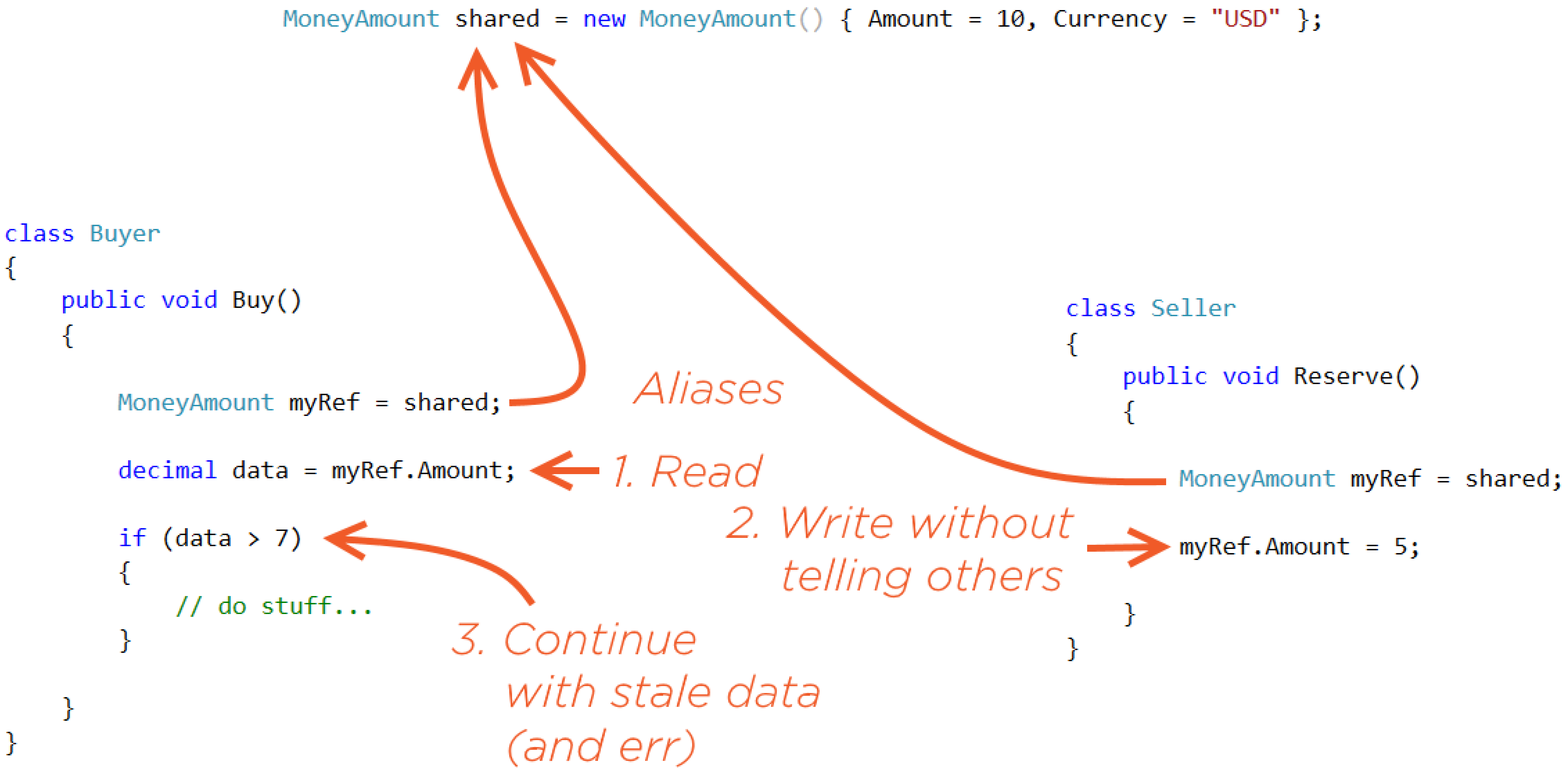
Using Immutable Objects When Possible

Zoran Horvat

OWNER AT CODING HELMET CONSULTANCY

@zoranh75 codinghelmet.com

Aliasing Bugs Explained



```
MoneyAmount shared = new MoneyAmount() { Amount = 10, Currency = "USD" };
```

```
class Buyer
{
    public void Buy()
    {

        MoneyAmount myRef = shared;

        decimal data = myRef.Amount;

        if (data > 7)
        {
            // do stuff...
        }

    }
}
```

```
class Seller
{
    public void Reserve()
    {

        MoneyAmount myRef = shared;

        myRef.Amount = 5;

    }
}
```

Aliases

1. Read

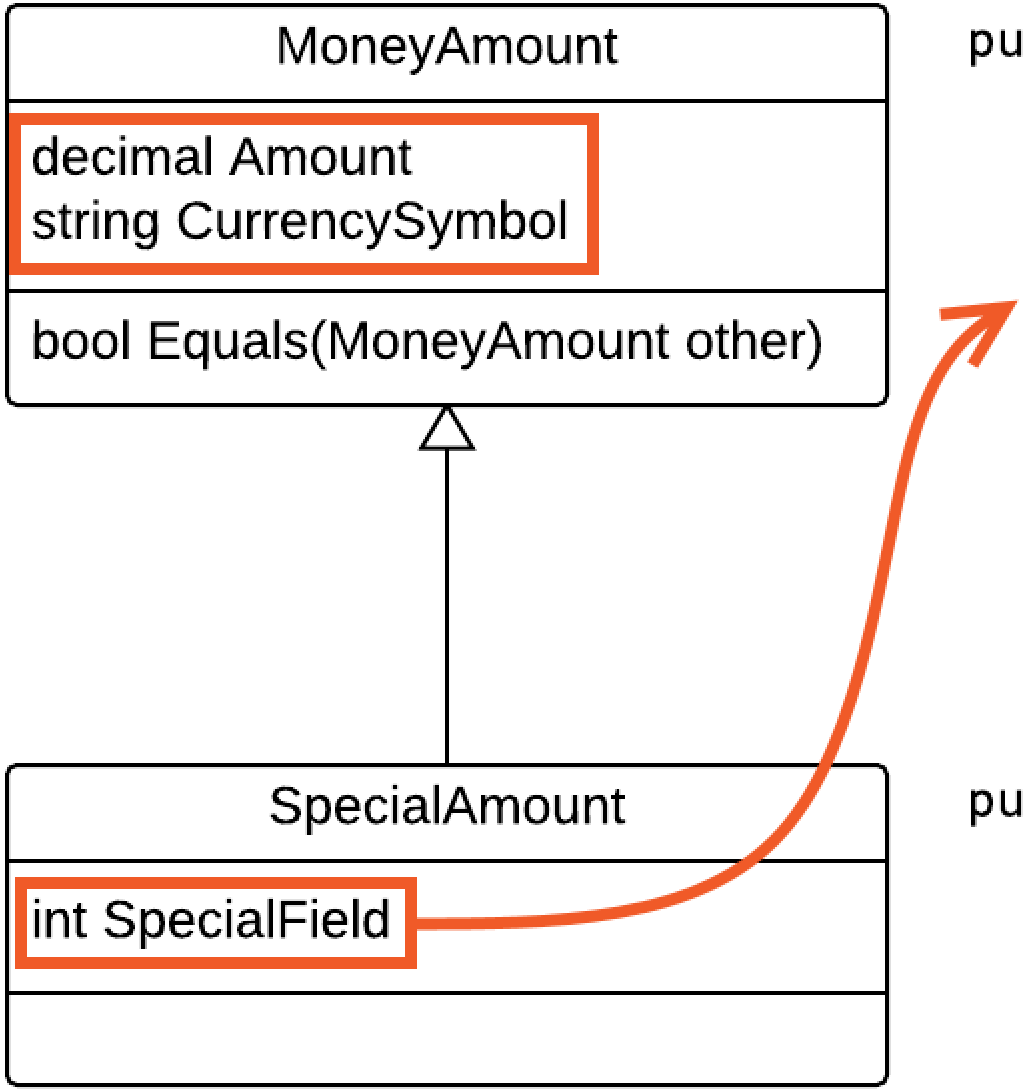
2. Write without telling others

3. Continue with stale data (and err)

```
class Seller
{
  // ...
  void Reserve(MoneyAmount cost)
  {
    if (IsHappyHour)
      cost.Amount *= .5M;
    // do the rest...
  }
  // ...
}


class Buyer
{
  // ...
  void Buy(MoneyAmount cost)
  {
    this.Seller.Reserve(cost);
    // do the rest...
  }
}
```

- **Refrain from modifying shared objects**
- **Object received as an argument is shared with the caller**

  `Reserve()` method can assume that `cost` object is an alias
- **True problem: Having an alias doesn’t mean we have a bug!**
- **Only sometimes, some `Buyer` will not work well with some `Seller`**
- **Avoid the possibility of aliasing bugs by not modifying shared objects**



MoneyAmount

decimal Amount
string CurrencySymbol

bool Equals(MoneyAmount other)

SpecialAmount

int SpecialField

```
public bool Equals(MoneyAmount other) =>
  other != null &&
  this.Amount == other.Amount &&
  this.CurrencySymbol == other.CurrencySymbol
```

Missing SpecialField test!

```
public bool Equals(MoneyAmount other) =>
  other != null &&
  this.Amount == other.Amount &&
  this.CurrencySymbol == other.CurrencySymbol &&
  this.SpecialField == other.SpecialField
```

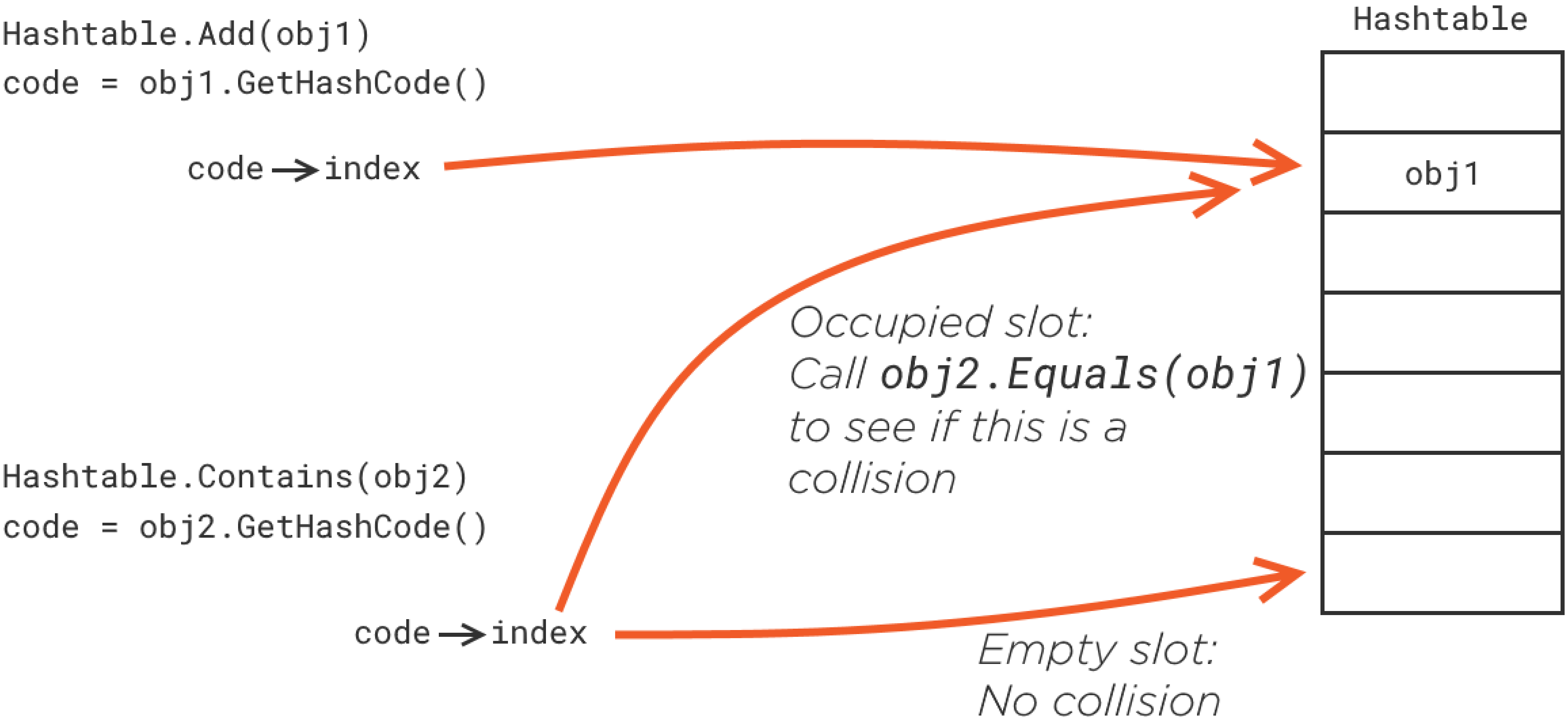
```
Base a = new Base();

Derived b = new Derived();


bool eq1 = a.Equals(b); // True

bool eq2 = b.Equals(a); // False
```

- **We may compare objects of base and derived class**

- **Base** `Equals` **may return** `True`

- **Derived** `Equals` **would return** `False`
  Derived Equals bases decision on additional fields

- **Equivalence relation is symmetric:**
  a == b if and only if b == a



```
Hashtable.Add(obj1)
code = obj1.GetHashCode()
```

code → index

Hashtable

obj1

Occupied slot:
Call ***obj2.Equals(obj1)*** *to see if this is a collision*

```
Hashtable.Contains(obj2)
code = obj2.GetHashCode()
```

code → index

Empty slot:
No collision

Value Object vs. Entity

Entity requires mutation over its lifetime

Value object remains unchanged after instantiation

Majority of objects we create can be treated as values

```
sealed class MoneyAmount : IEquatable<MoneyAmount>
{
  public decimal Amount { get; }
  public string CurrencySymbol { get; }

  public MoneyAmount(decimal amount,
                     string currencySymbol) { ... }

  public MoneyAmount Scale(decimal factor) { ... }

  public static MoneyAmount operator *
      (MoneyAmount amount, decimal factor) { ... }

  public override bool Equals(object obj) { ... }

  public bool Equals(MoneyAmount other) { ... }

  public override int GetHashCode() { ... }

  public static bool operator ==
      (MoneyAmount a, MoneyAmount b) { ... }

  public static bool operator !=
      (MoneyAmount a, MoneyAmount b) { ... }

}
```

- **Remove property setters**
- **Introduce factory method**
  Constructor is just fine
- **Add operations closed under the value type**
  Don’t force consumers do that
  Operation returns new instance of the same type
  That makes the class safe to use
  Next step: value-typed semantic
- **Implement full value-typed semantics**
  Declare the class sealed
  Override Equals() method
  Implement IEquatable<T>
  Override GetHashCode()
  Overload == and != operators

```csharp
class MoneyAmount
{
  public decimal Amount { get; set; }
  public string CurrencySymbol { get; set; }
}
```

- **Mutable class is truly simple**
  But also susceptible to bugs at the calling end

```csharp
class MoneyAmount
{
  public decimal Amount { get; }
  public string CurrencySymbol { get; }

  public MoneyAmount(decimal amount,
                     string currencySymbol) { ... }

  public MoneyAmount Scale(decimal factor) { ... }

}
```

- **It takes a dozen of lines of code to make a class just immutable**
  That is still far away from true value type

```
sealed class MoneyAmount : IEquatable<MoneyAmount>
{
  public decimal Amount { get; }
  public string CurrencySymbol { get; }

  public MoneyAmount(decimal amount,
                     string currencySymbol) { ... }

  public MoneyAmount Scale(decimal factor) { ... }

  public override bool Equals(object obj) { ... }

  public bool Equals(MoneyAmount other) { ... }

  public override int GetHashCode() { ... }

  public static bool operator ==
      (MoneyAmount a, MoneyAmount b) { ... }

  public static bool operator !=
      (MoneyAmount a, MoneyAmount b) { ... }

}
```

- **It takes a lot more code to make true value type**
  - Supports equality comparison
  - Supports hashtables

- **C# doesn’t help with building value types**
  - Write value types when necessary

- **Immutable classes are just enough in many cases**
  - Full value types help avoid repeated equality testing code
  - They also help avoid bugs

Summary

Leave the class mutable or make it immutable?

- It is easy to introduce immutability
- Immutability makes it impossible to create aliasing bugs

From immutable class to value class

- Value objects used just like integers or strings
- Greatly simplifies code maintenance
- Greatly improves application stability
- Unfortunately, it inflates simple classes

Next module –
Living Without Null References