Dismissing Defensive Code by Avoiding Primitive Types

Zoran Horvat

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com

Coding with Enumerations

Practical Effect	Also known as…
Assignment must be guarded	No type safety
Change affects users	No polymorphism
Guarding requires alternative implementation	Design flaws
Enumeration produces no objects	Violation of object-oriented concepts

Coding with Enumerations

Enumeration

```
enum Grade
```

```
if (Enum.IsDefined(
    typeof(Grade), value) ...
```

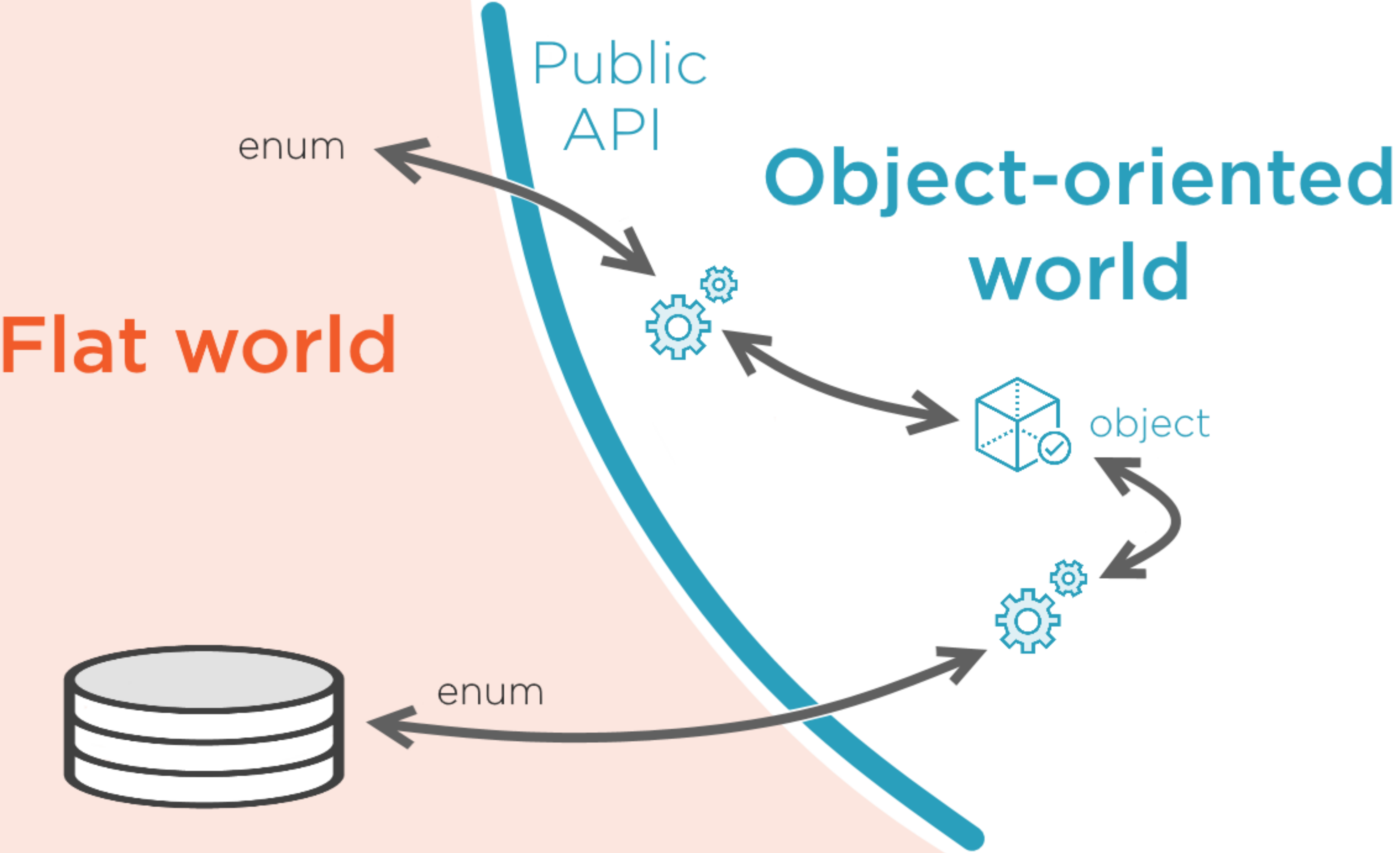
Class

```
class Grade
{
  private Grade() { }

  public static Grade A { get; }
  public static Grade B { get; }
  public static Grade C { get; }
  public static Grade D { get; }
  public static Grade F { get; }

  public string Label { get; }
  public bool IsPassing { get; }
  ...
}
```

```
if (grade != null) ...
```



Flat world

Public API

Object-oriented world

enum

object

enum

The Stringification Trap

Turning objects into strings = Stringification

To send over network, save to database, etc.

Then, why strings inside domain objects?

The Stringification Trap

```
public abstract class Student
{
  public string Name { get; }

  public Student(string name)
  {
    if (string.IsNullOrEmpty(name))
      throw new ArgumentException();
    if (char.IsHighSurrogate(name[name.Length - 1]))
      throw new ArgumentException();
    this.Name = name;
  }
  ...
}
```

Student’s name is plain string

And it incurs all sorts of complications

The Stringification Trap

```
public abstract class Student
{
  public string Name { get; }

  public Student(string name)
  {
    if (string.IsNullOrEmpty(name))
      throw new ArgumentException();
    if (char.IsHighSurrogate(name[name.Length - 1]))
      throw new ArgumentException();
    this.Name = name;
  }
  ...
}
```

Customer:

Student has a name

Name is never empty

Programmer:

`Student` class contains a `string`

which is not `NullOrEmtpy`...

Pitfalls of Primitive Types

Verbose code

Hard to make it right

Validations all around

Encapsulate primitive types in a class

Do not accept primitive type arguments

Example: Money as Decimal

`System.Decimal`

-217.81 -99.95 -4.6 2.11 3.14 9,840.19

a + b = c amount1 + amount2 = sum

No common addition defined for money

$5 + $3 = $8 ✓

$5 + 3€ = ? ✗

Example: Money as Decimal

```
if (currency1 == currency2)
{
  sum = amount1 + amount2;
  currency = currency1;
}
else
{
  ???
}
```

Example: Money as Decimal

```
if (currency1 == currency2)
{
  sum = amount1 + amount2;
  currency = currency1;
}
else
{
  ???
}
```

```
void Deposit(decimal amount)
{
  if (amount > 0)
  {
    Accept(amount);
  }
  else
  {
    FailOrSomething();
  }
}
```

Command-Query Segregation Principle (CQRS)

Often demonstrated on large scale

But equally applicable to medium-scale projects

The principle:

Keep commands separate from queries.

Command-Query Segregation Principle (CQRS)

Command

Modifies the system state

Applies all the rules

Performs all the validations

Heavy
Complex
Expensive

Query

Keeps the system state intact
(only asks what the state is)

Loading lots of data

Expects to get the results fast
(e.g. populating a grid)

No rule checking
(that was already done, wasn’t it?)

Separating the Models

Domain Model

“The Model”

Supports commands

View Model

Looks similar to the Domain Model

Contains no validation

Only properties, like in DTOs

View Model vs. DTO

Data Transfer Object (DTO)	View Model
Enables transfer over a flat channel	Meant to be rendered on the view
Public property getters for serialization	Public property getters to populate the view
Parameterless constructor	Parameterless constructor
Public property setters for deserialization	Public property setters for fast materialization

Summary

What's wrong with primitive types?

- Enumerations,
- Strings,
- Numeric data, etc.

Summary

Enumerations

- Useful information encoded as int
- No syntactical hint about meaning
- Operations are not close to data
- Lots of defensive code in the consumer

Summary

Strings

- Associated with general operations
- Not useful in any concrete domain
- *Stringification:* Turning domain-related data into plain strings
- Magnet for bugs and defensive code

Summary

Numeric data

- Not enough domain-related information
- Lots of defense at consuming end

Summary

Wrapping primitive types into a class

- Place all defense close to raw data
- Expose domain-related behavior

No more defensive code in the consumer

- Improves overall application stability

And a word about persistence…

- Heavily encapsulated objects can still be persisted
- No need to make concessions to the persistence layer

Next module

Function Domains vs. Domain Rules