Removing Corruption by Only Making Valid State Transitions

Zoran Horvat

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com

Keep Object's State Correct

An object's internal state is correct

No need to defend from incorrect state
Who would have guessed?

Mutable Classes and Defensive Code

When a class exposes mutators...

... then some mutations may cause bugs...

... and then we have to include defensive code...

```
obj = new Student();
obj.Mutate();
obj.MutateAgain();
…
```

```
student.Name = “Someone Else”;

student.AddGrade(Grade.B);
```

- **The simplest mutator:**
  Property setter
- **More complex mutators:**
  A method which changes state
- **Mutations must be defended**

```
public class Student
{
  public string NameInitial =>
    this.Name.Substring(0,
        char.IsHighSurrogate(this.Name[0]) ? 2 : 1);
}
```

`NameInitial` property getter depends on `Name`

Name **must not be null**

- Otherwise, `Name.Substring()` call would fail with `NullReferenceException`

```
public class Student
{
  public string NameInitial =>
    this.Name.Substring(0,
        char.IsHighSurrogate(this.Name[0]) ? 2 : 1);
}
```

`NameInitial` property getter depends on `Name`

Name **string must contain at least one character**

- Otherwise, `Name[0]` indexer call would fail with `IndexOutOfRangeException`

```
public class Student
{
  public string NameInitial =>
    this.Name.Substring(0,
        char.IsHighSurrogate(this.Name[0]) ? 2 : 1);
}
```

`NameInitial` property getter depends on `Name`

If `Name` begins with high surrogate, then at least one character follows

- Otherwise, `Name.Substring(0, 2)` call would fail with `ArgumentOutOfRangeException`

```
public class Student
{
  public string NameInitial =>
    this.Name.Substring(0,
        char.IsHighSurrogate(this.Name[0]) ? 2 : 1);
}
```

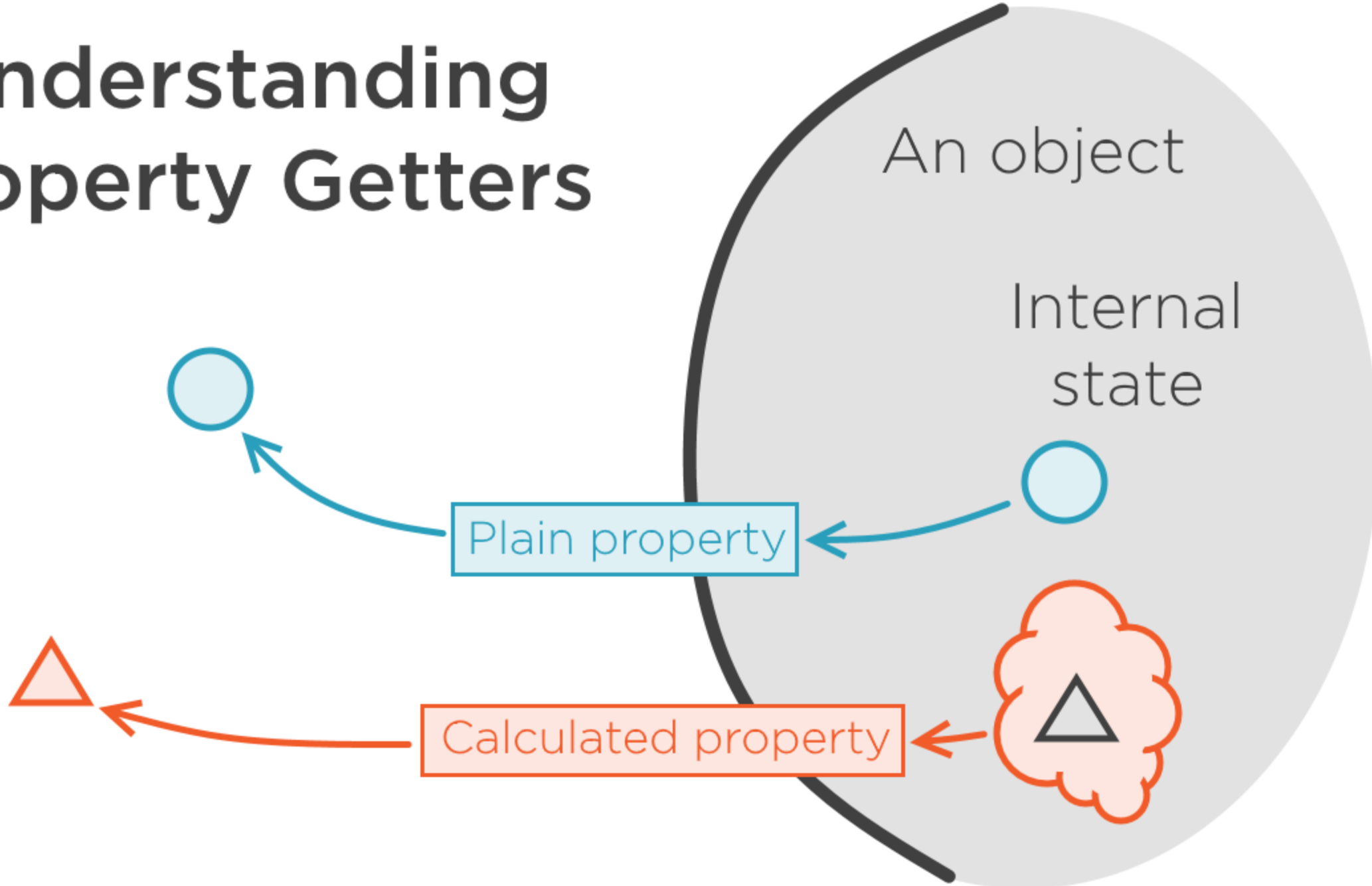
`NameInitial` property getter depends on `Name`

There is no defensive code in the property getter

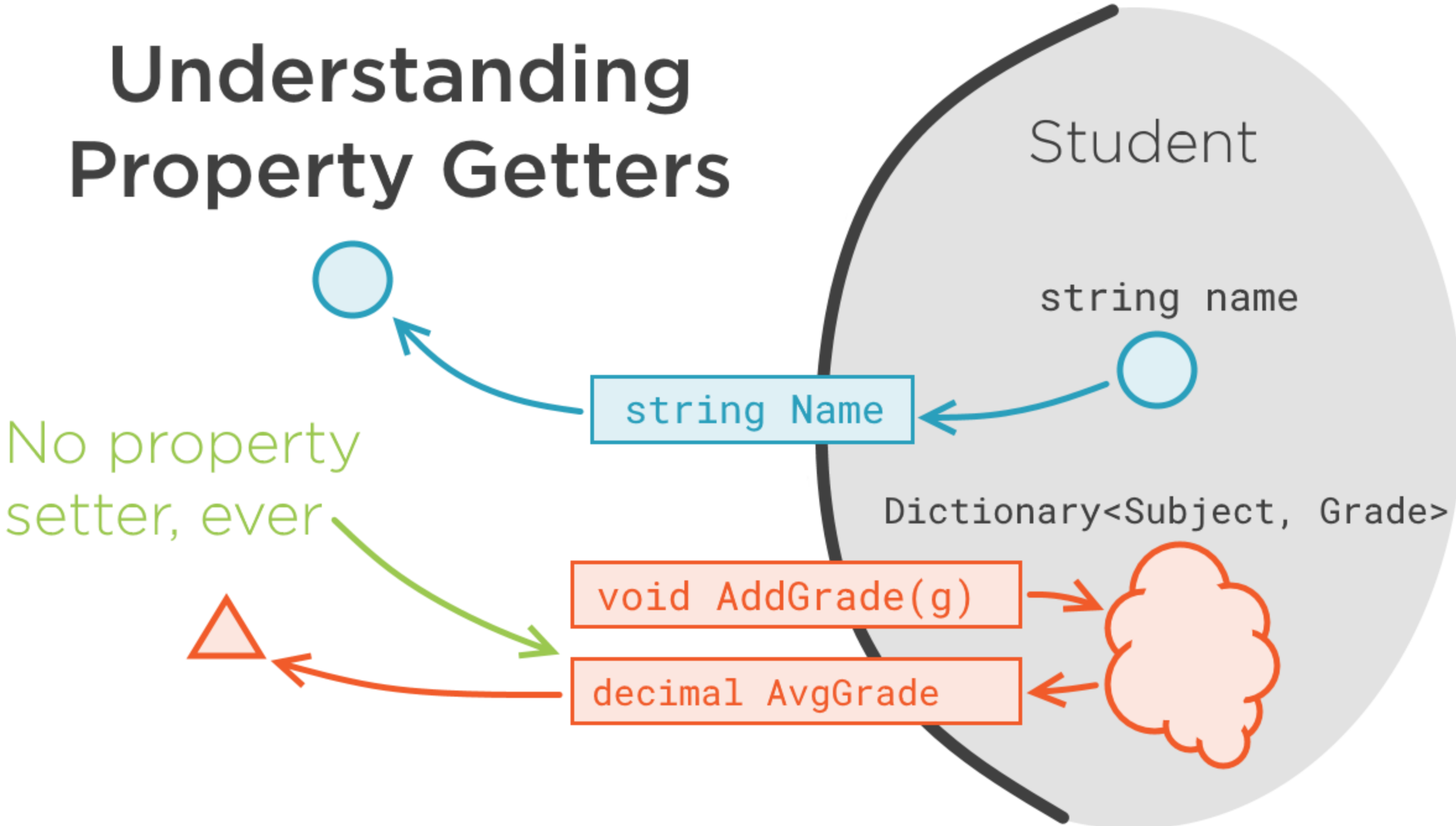
- It depends on already established correctness of the `Name` string
- Preconditions had to be verified in the constructor

Adding `Name` setter might allow incorrect state again

Understanding Property Getters



Understanding Property Getters



```
class Student
{
  public decimal AvgGrade { get; }

  public string Name { get; }
}
```

- **AvgGrade is the result of the calculation process**
- **Name property gives out raw data**

  Invites the caller to implement the operation

  That breaks encapsulation

  This class should apply Name-related business logic

```
class Student
{
  public decimal AvgGrade { get; }

  public string Name { get; set; }
}
```

- **Raw data properties allow setters**

  Setters must be defended

  Object with no setters requires no defend past the constructor

```
class ImmutableStudent
{
  public decimal AvgGrade { get; }

  public string Name { get; }

  public ImmutableStudent
    WithName(string name);
}
```

- **We can make the class immutable**

- **Create new** `Student` **when name has to be changed**

Pitfalls of Defense

```
public void AddGrade(Subject subject, Grade grade)
{
    if (!Enum.IsDefined(typeof(Grade), grade))
        throw new ArgumentException();

    if (subject == null ||
        !this.IsEnlistedFor(subject))
        throw new ArgumentException();

    if (this.Grades.ContainsKey(subject) &&
        this.Grades[subject] != Grade.F)
        throw new ArgumentException();

    this.Grades[subject] = grade;
}
```

Fail

Defensive code implies that the method ***doesn't*** work

There are too many reasons for this piece of code to ***not*** execute

Pitfalls of Defense

```
public void AddGrade(Subject subject, Grade grade)
{
    if (!Enum.IsDefined(typeof(Grade), grade))
        throw new ArgumentException();

    if (subject == null ||
        !this.IsEnlistedFor(subject))
        throw new ArgumentException();

    if (this.Grades.ContainsKey(subject) &&
        this.Grades[subject] != Grade.F)
        throw new ArgumentException();

    this.Grades[subject] = grade;
}
```

Better idea:

Come up with the design in which these cases cannot happen

Constructors as Partial Functions

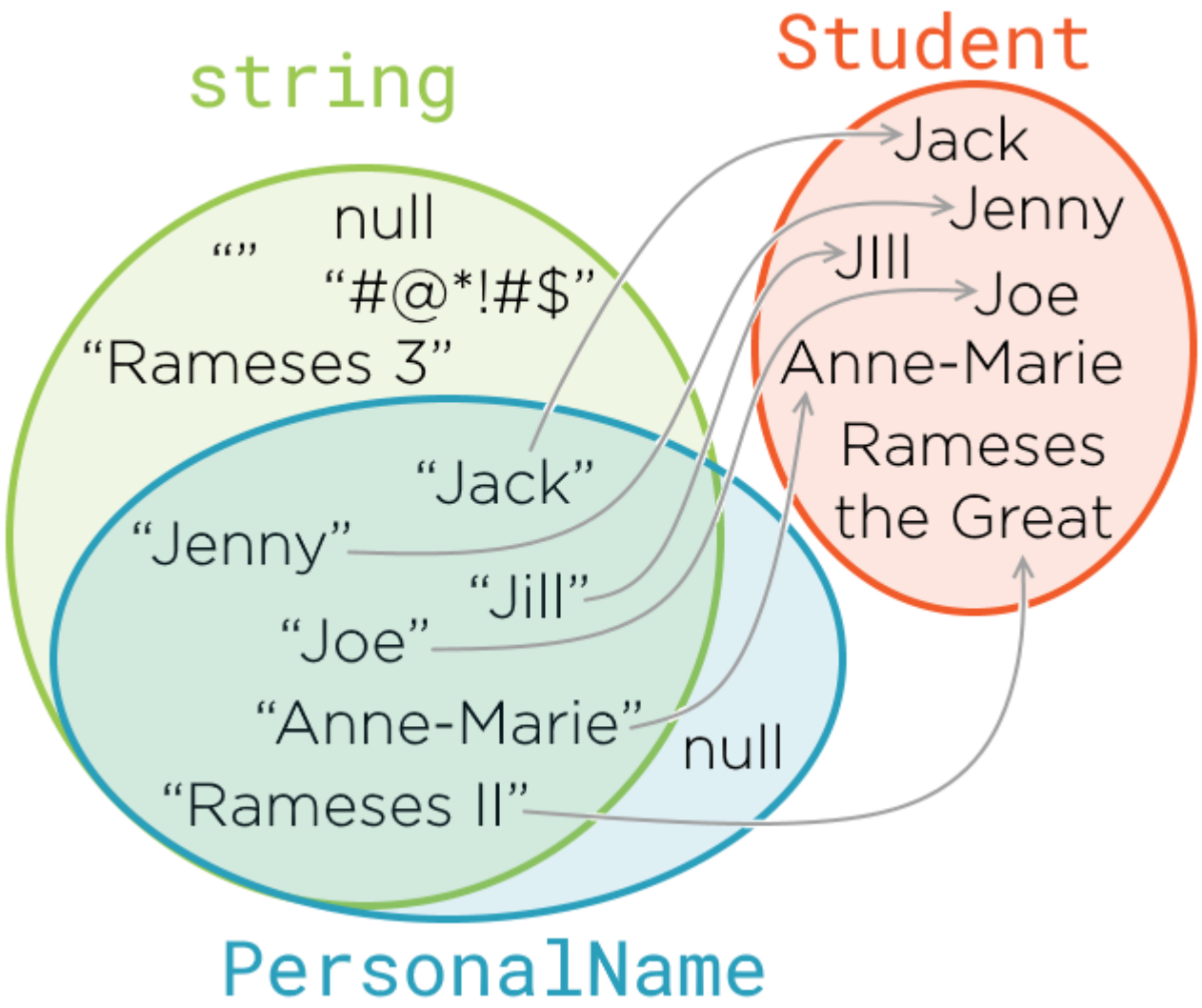
`student.AddGrade(subject, grade)`

`grade` not defined	✗ fail
`student` not enlisted for `subject`	✗ fail
`student` has prior grade on `subject`	✗ fail
otherwise...	✓ go!

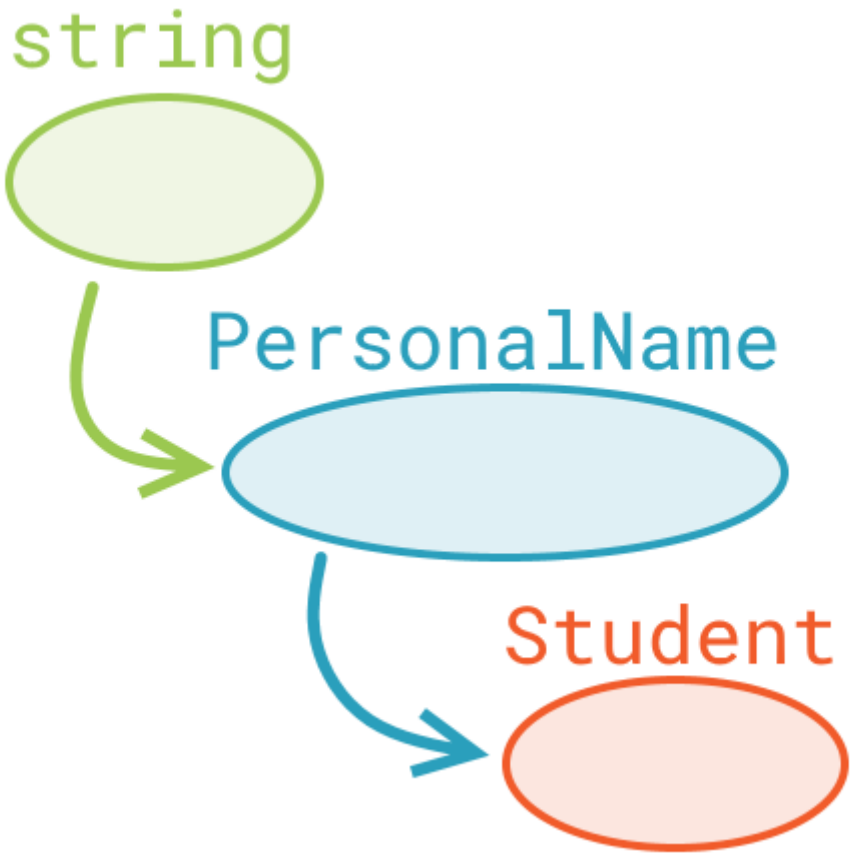
$\texttt{Constructor}(\texttt{arg}_1, \texttt{arg}_2, \ldots, \texttt{arg}_N) \rightarrow \texttt{object}$

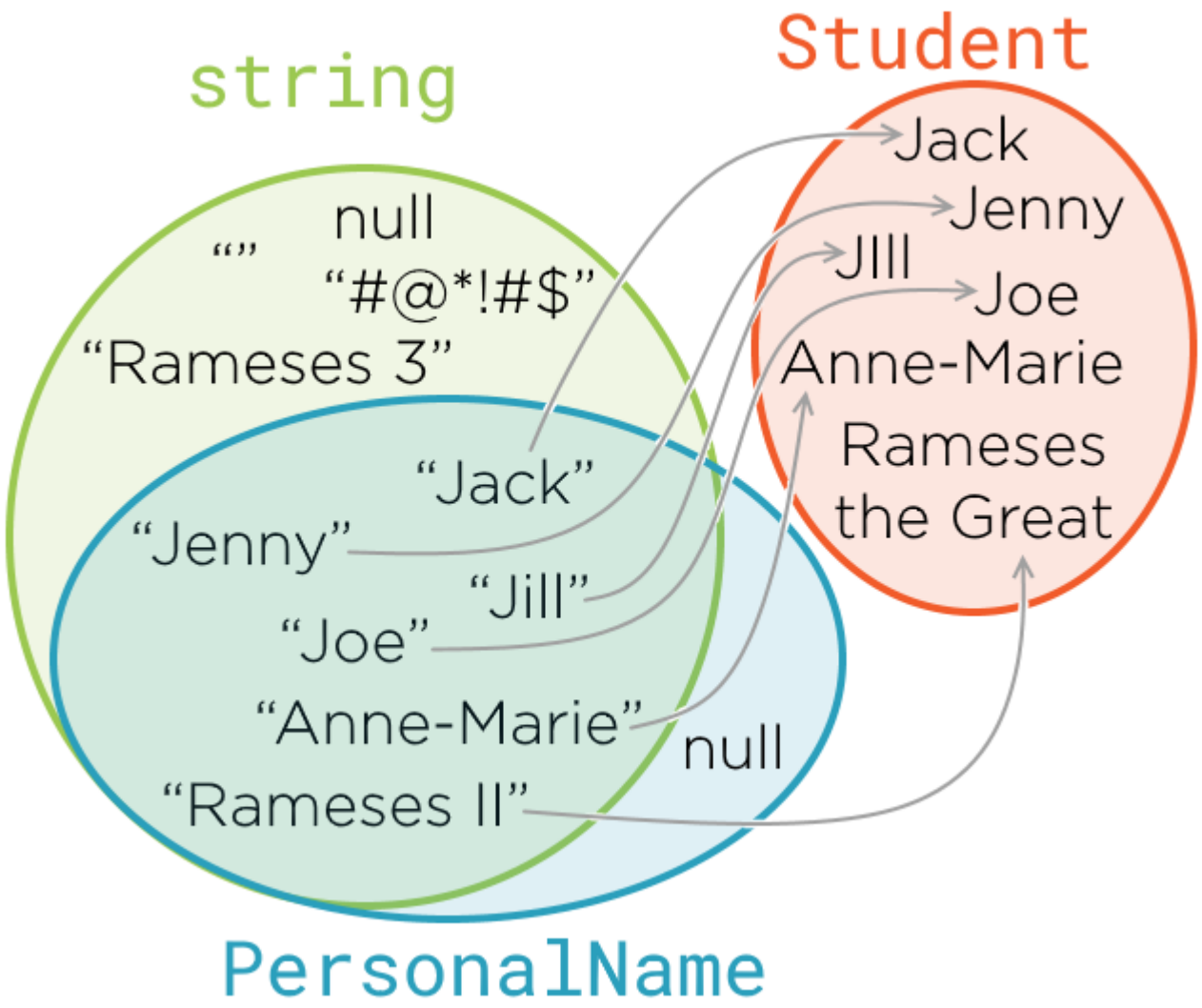
Constructors as Partial Functions

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



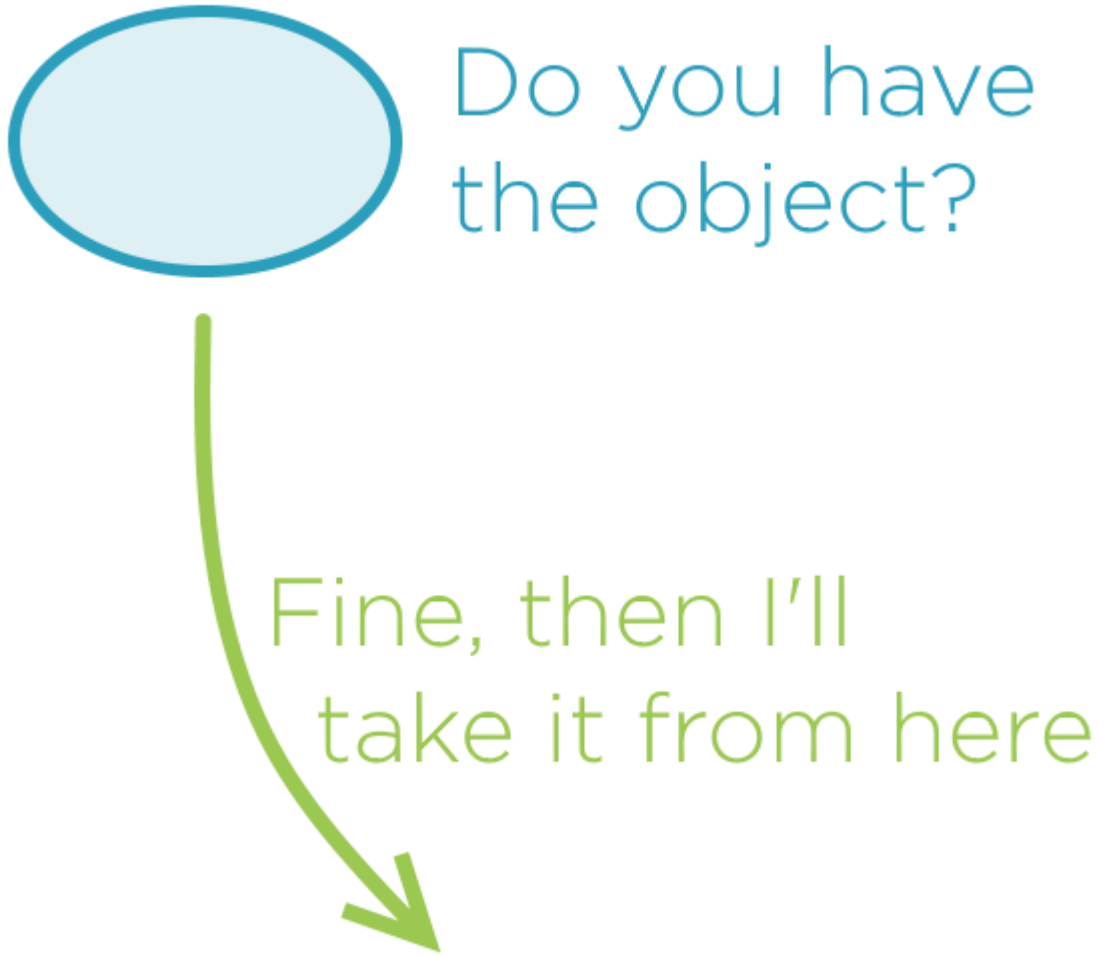
Constructors as Partial Functions





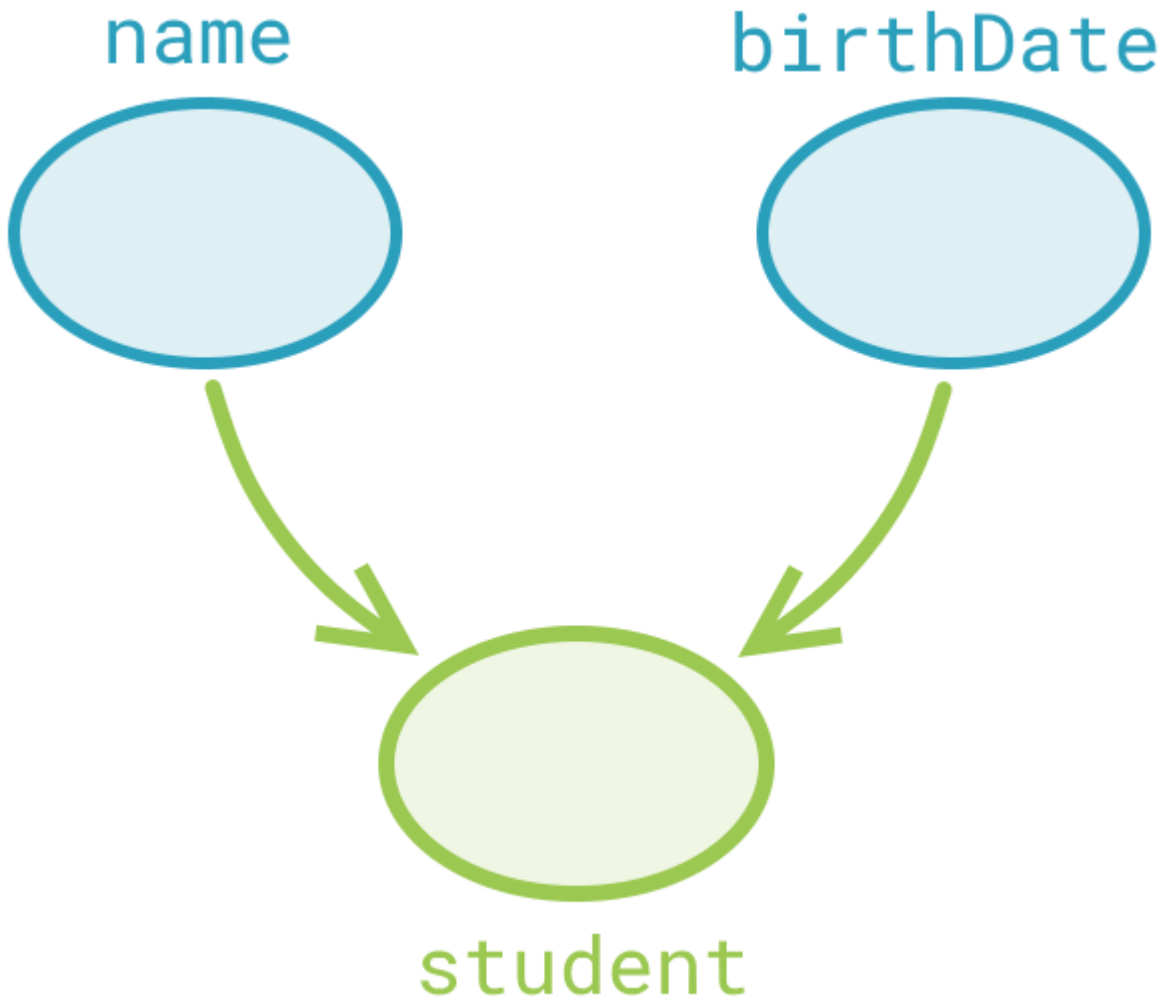
The Object Rule

IF YOU HAVE AN OBJECT, THEN IT'S FINE.



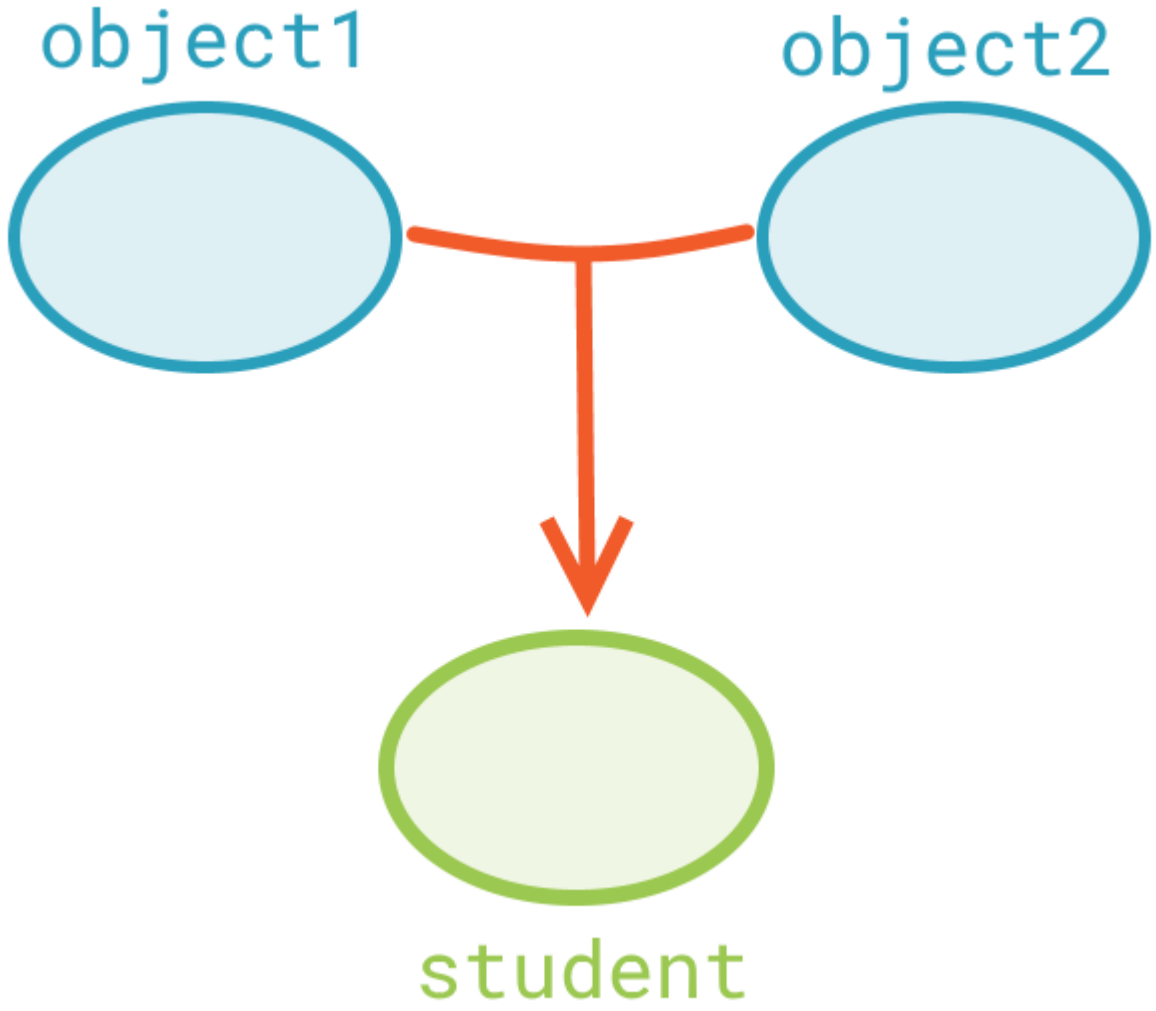
The Object Rule

If you have an object, then it's fine.

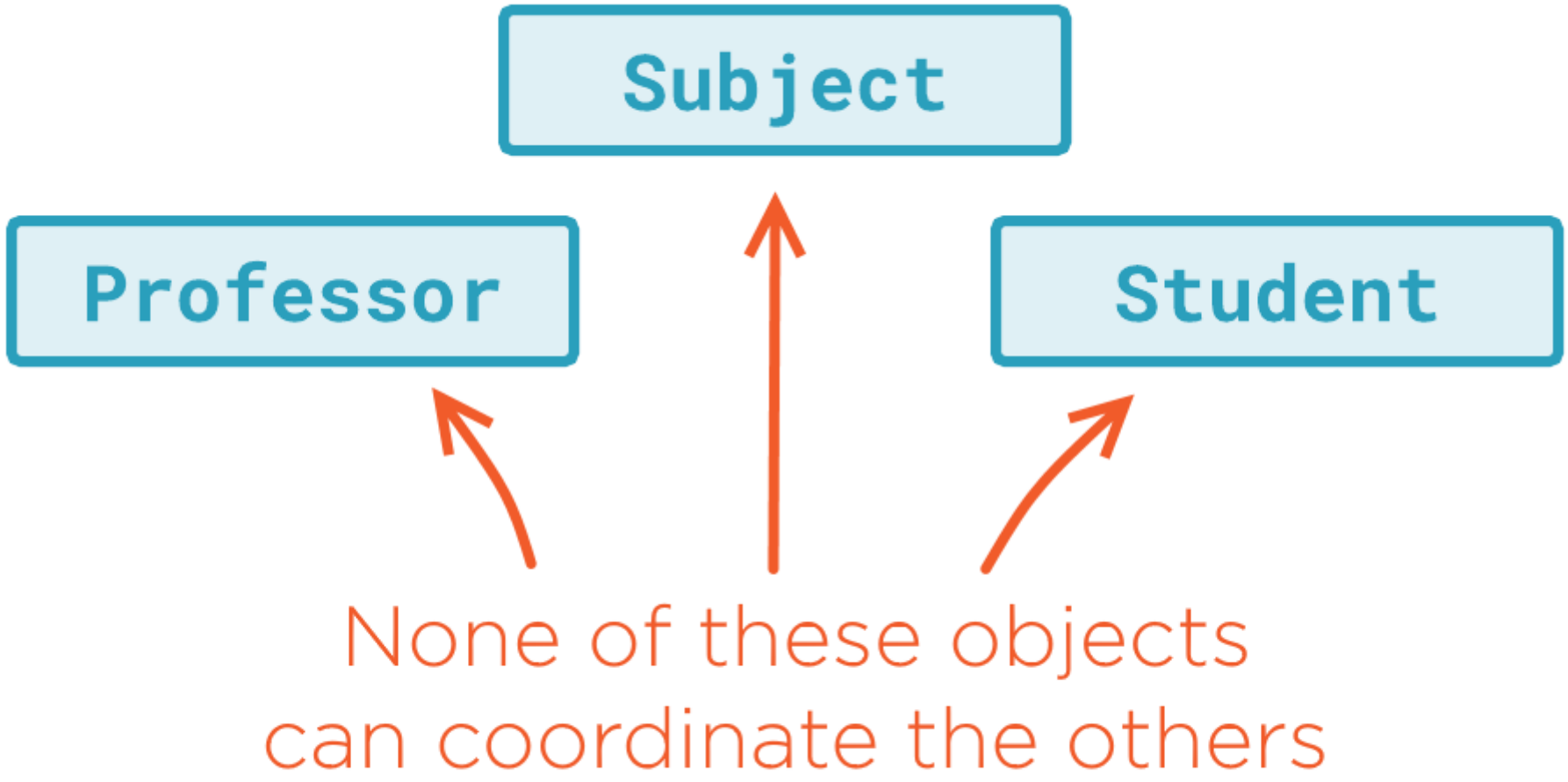


The Object Rule

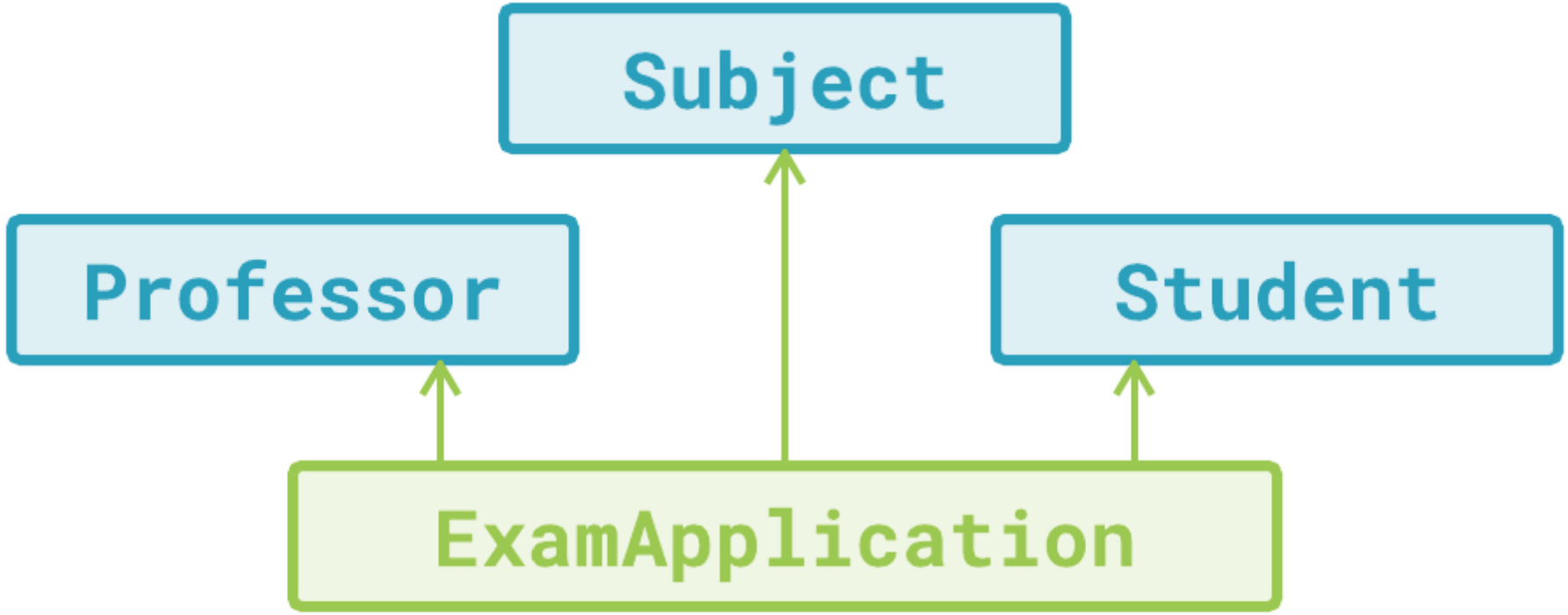
If you have an object, then it's fine.



Complex Construction Validation

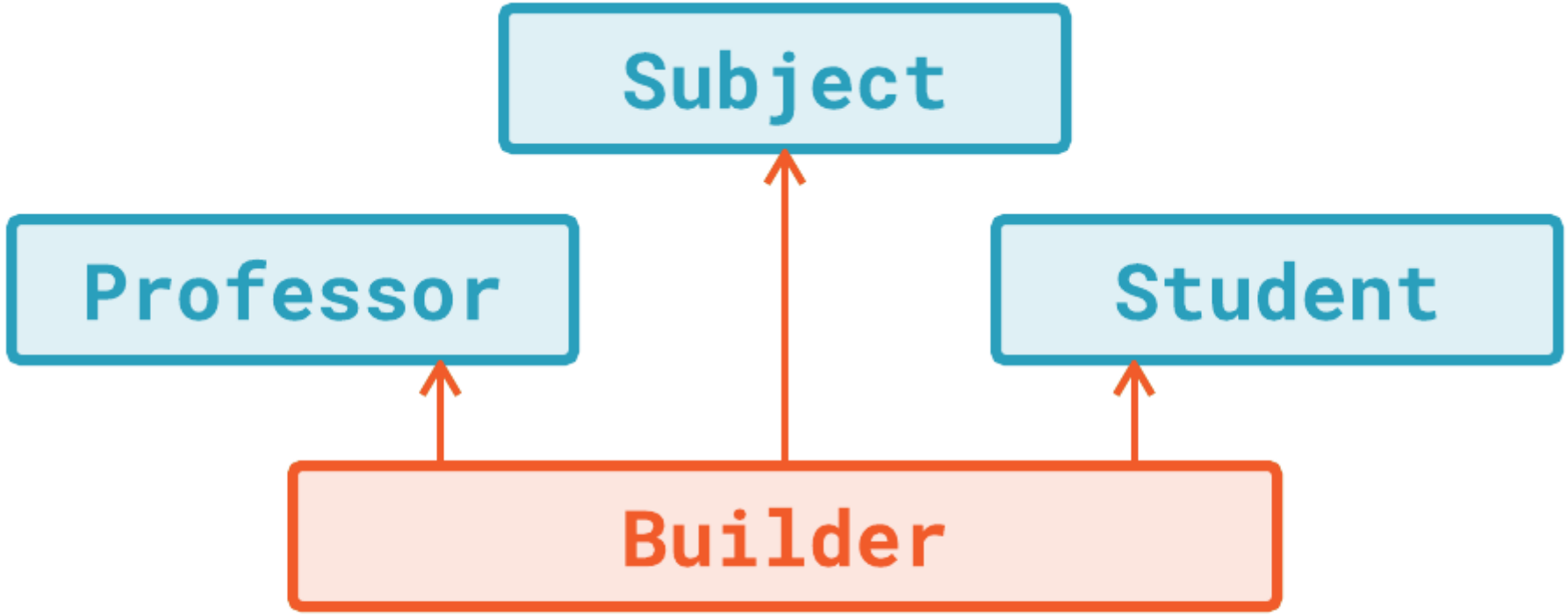


Complex Construction Validation



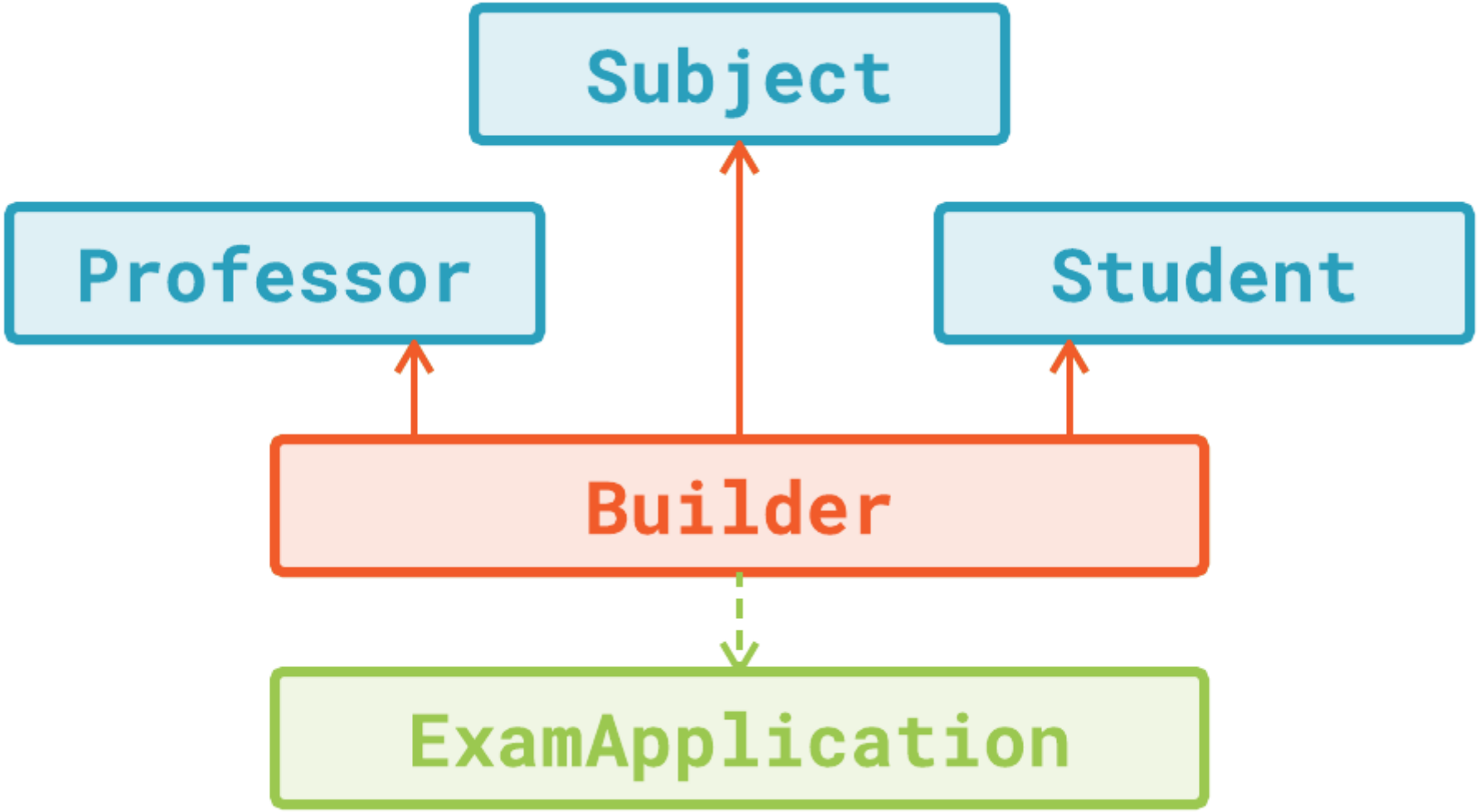
Constructor fails
on validation error...

Complex Construction Validation



Builder validates objects during its lifetime

Complex Construction Validation



Complex Construction Validation

Lightweight Solution

Use constructor as a factory function

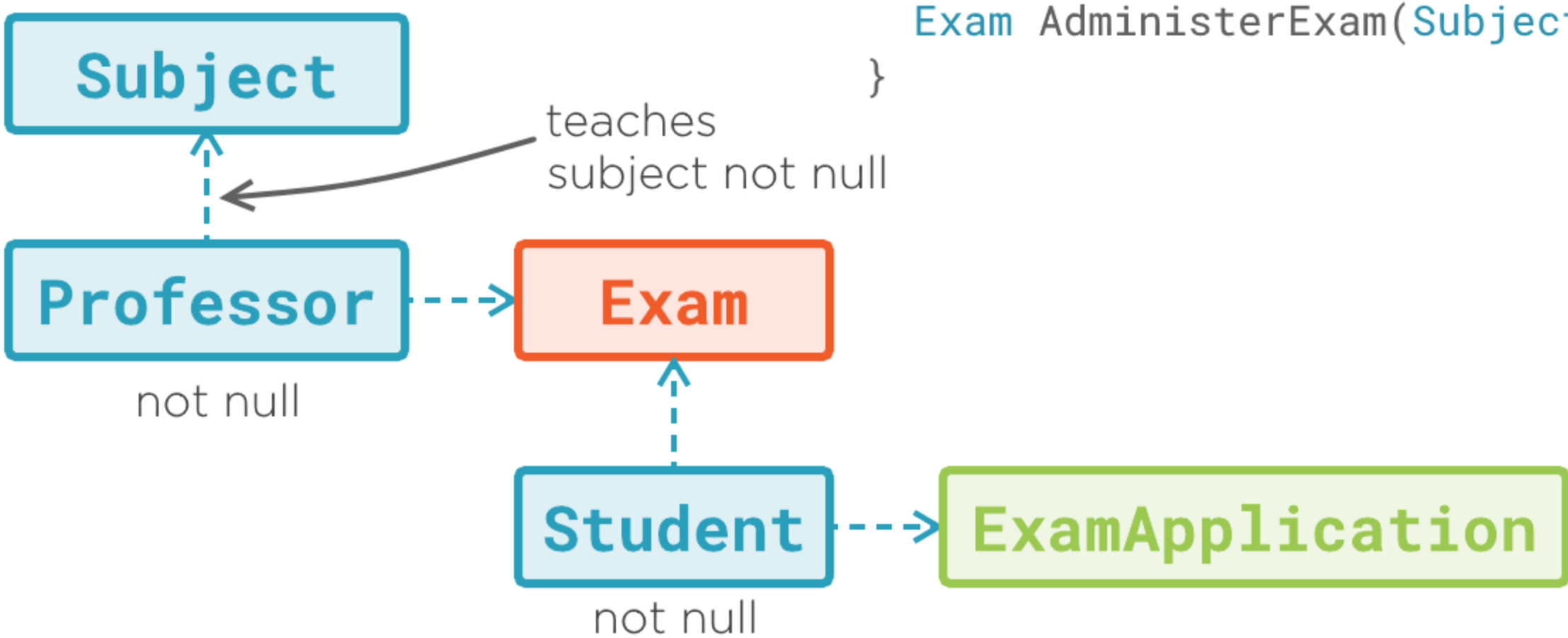
Middle Solution

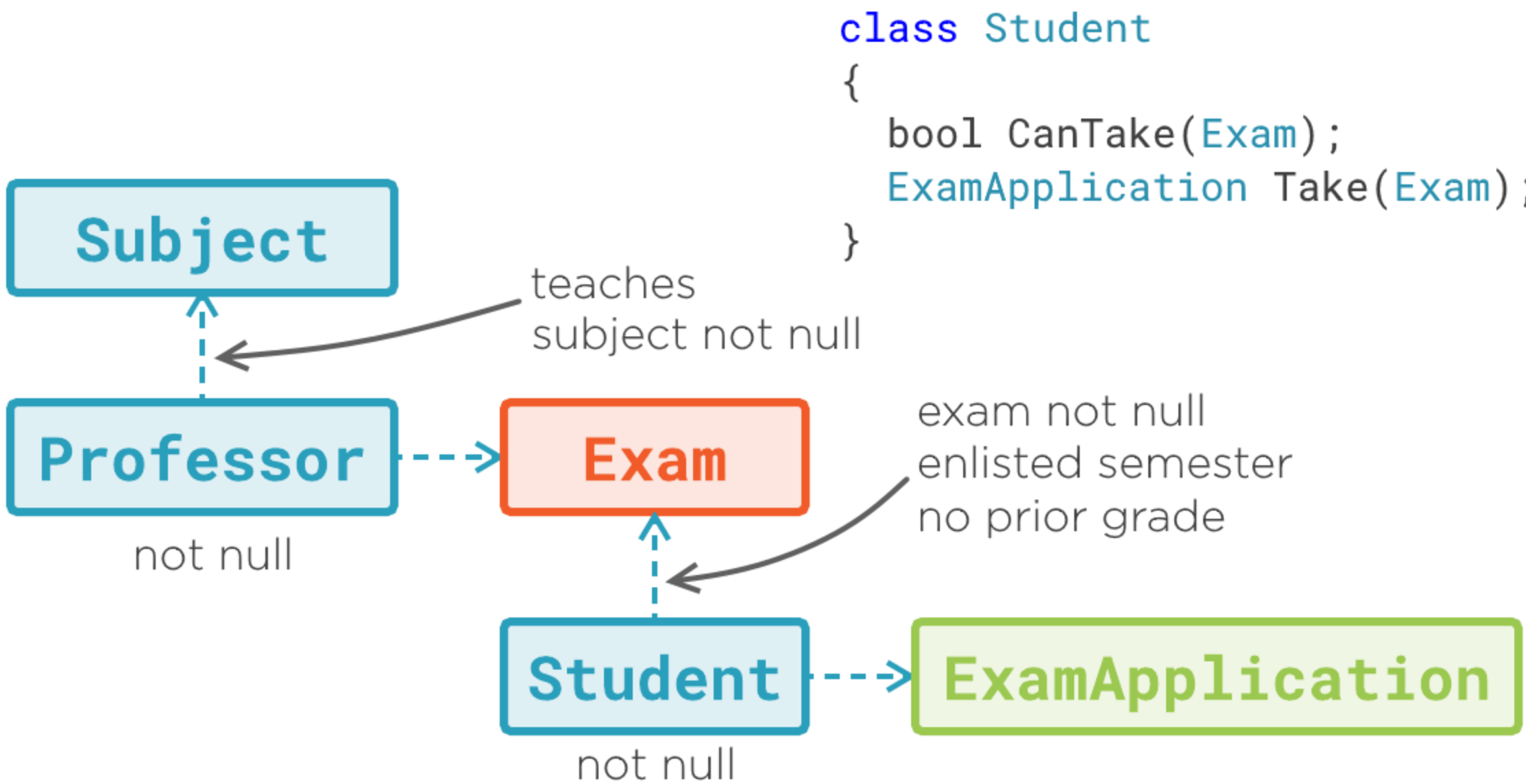
Series of small factory functions

Heavyweight Solution

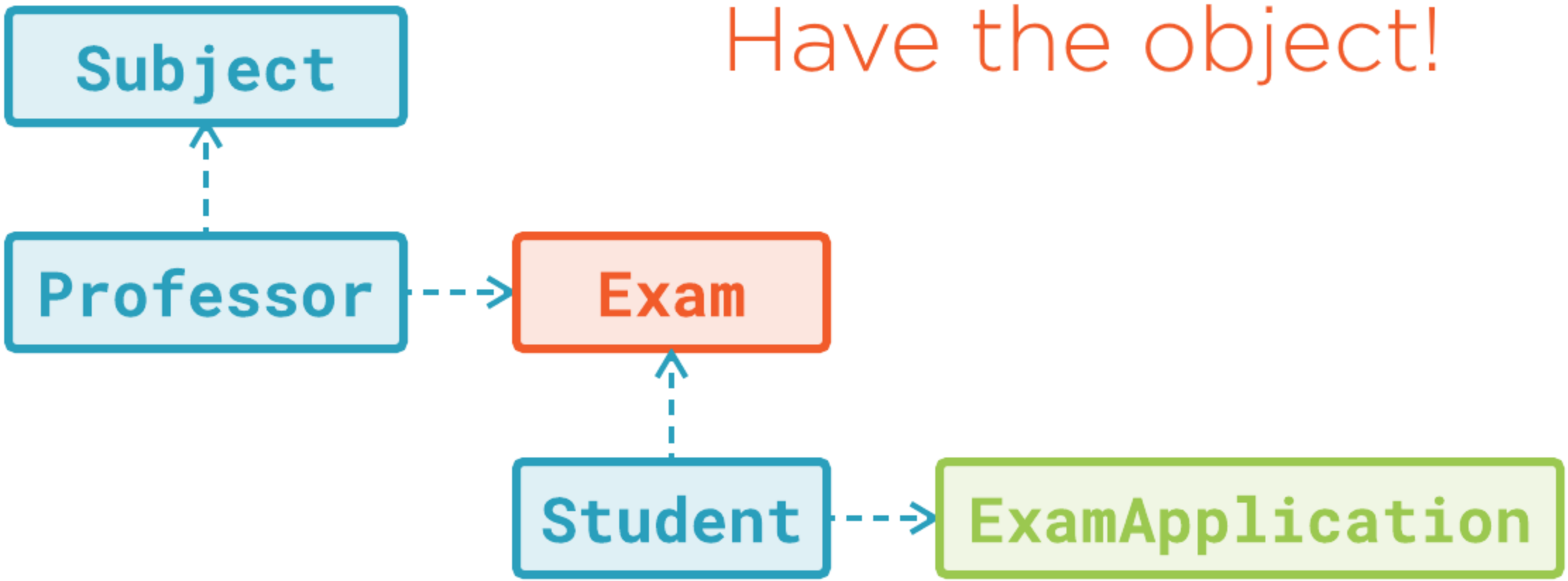
Use builder for complex validation

```
class Professor
{
  bool IsTeaching(Subject);
  Exam AdministerExam(Subject);
}
```




class Student
{
bool CanTake(Exam);
ExamApplication Take(Exam);
}
Subject
teaches
subject not null
Professor
not null
Exam
exam not null
enlisted semester
no prior grade
Student
not null
ExamApplication

The Best Defense:
Have the object!



Persisting Rich Domain Model

Domain model ⇒ Construction validation

Persistence ⇒ Plain construction

Persisting Rich Domain Model

Persist Domain Model

... and make it ORM-friendly

Persist Separate Model

... and keep the Domain Model persistence-ignorant

Persisting Domain Model

Default constructor

Property setters

No setter validation

Database ID field

Include parameterless constructor in all model classes

Constructor can be private (ORM will access it via reflection)

Default constructor

Property setters

No setter validation

Database ID field

Persisting Domain Model

Setters on all properties defining persistable state

Setters can be private
(ORM will access them via reflection)

Persisting Domain Model

Default constructor

Property setters

No setter validation

Database ID field

Property setters must be dumb

ORM is not aware of any setter rules

Object materialization may fail
if setters can throw

Persisting Domain Model

Default constructor

Property setters

No setter validation

Database ID field

Include database identity
in every model class

New model

Plain getters/setters

Two-way mapping

Separate Persistence Model

Define new model in the Infrastructure layer

No construction rules

No validation

New model

Plain getters/setters

Two-way mapping

Separate Persistence Model

Plain property getters and setters

Setters with no validation

Not same as the Domain Model

Designed to support
fast and easy persistence

New model

Plain getters/setters

Two-way mapping

Separate Persistence Model

There must be a mapping between Domain Model and Persistence Model

Map new domain object before persisting it

Map persisted object to domain object before using it

How to Decide...

Simple Model

Make the Domain Model persistable

That saves a lot of work

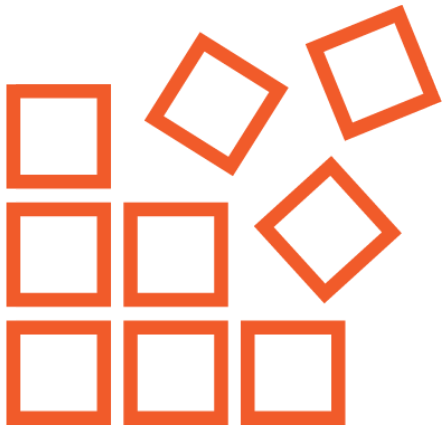
Complex Model

Invest in separating persistence from domain

Hard to add persistence to an already complex Domain Model

That would add unwanted complexity

Persistence tradeoffs are not welcome in complex domains



Development of entire separate Persistence Model

Development of entire complex Domain Model

Proof of concept

- No need to make concessions to persistence in the Domain Model

Summary

State mutations

- Changes happen after construction
- Transition leads to new consistent state

Consequences

- No defensive code
- Short implementation
- No alternative execution branch
- No alternatives for the caller, either

One method to enforce consistency

- Defend in all state changing methods
- Better solutions are yet to come

Summary

Defending from invalid arguments

- Define domain of every function
- State and arguments combined must belong to the domain
- Otherwise, method cannot execute

Restricting argument domains

- Define custom argument types
- Remove reasons to fail

Next module

Avoiding Primitive Types