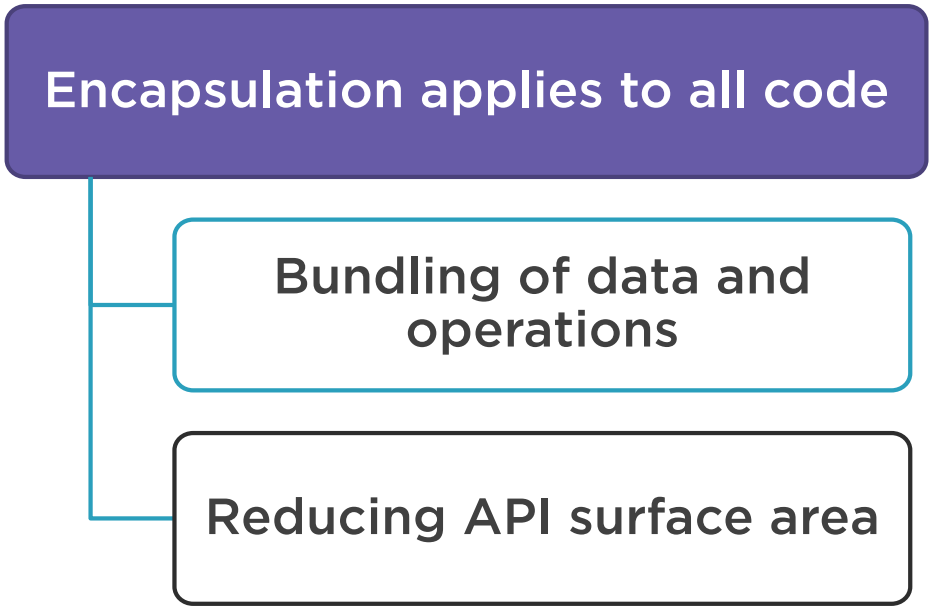
Working with Many-to-one Relationships

Vladimir Khorikov

@vkhorikov www.enterprisecraftsmanship.com

DbContext Encapsulation



Recap: DbContext Encapsulation

```
var optionsBuilder = new DbContextOptionsBuilder<SchoolContext>();
optionsBuilder
    .UseSqlServer(connectionString)
    .UseLoggerFactory(loggerFactory)
    .EnableSensitiveDataLogging();

using (var context = new SchoolContext(optionsBuilder.Options))
{
    /* ... */
}
```

- ✗ API surface is too wide
- ✗ Too much room for mistake

Recap: DbContext Encapsulation

```
string connectionString = GetConnectionString();

using (var context = new SchoolContext(connectionString, true))
{
    /* ... */
}
```

- ✓ Exposes the absolute minimum of configuration options
- ✓ No room for inconsistencies

Recap: DbContext Encapsulation

Overconfiguration issue applies to any infrastructure with rich configuration capabilities

Keep configuration surface as small as possible

Recap: Getting Rid of Public Setters

```
public class Student
{
    public long Id { get; private set; }
    public string Name { get; private set; }
    public string Email { get; private set; }
    public Course FavoriteCourse { get; private set; }
}
```

- ✓ Made property setters private
- ✓ Start with as few modification APIs as possible

Recap: Getting Rid of Public Setters

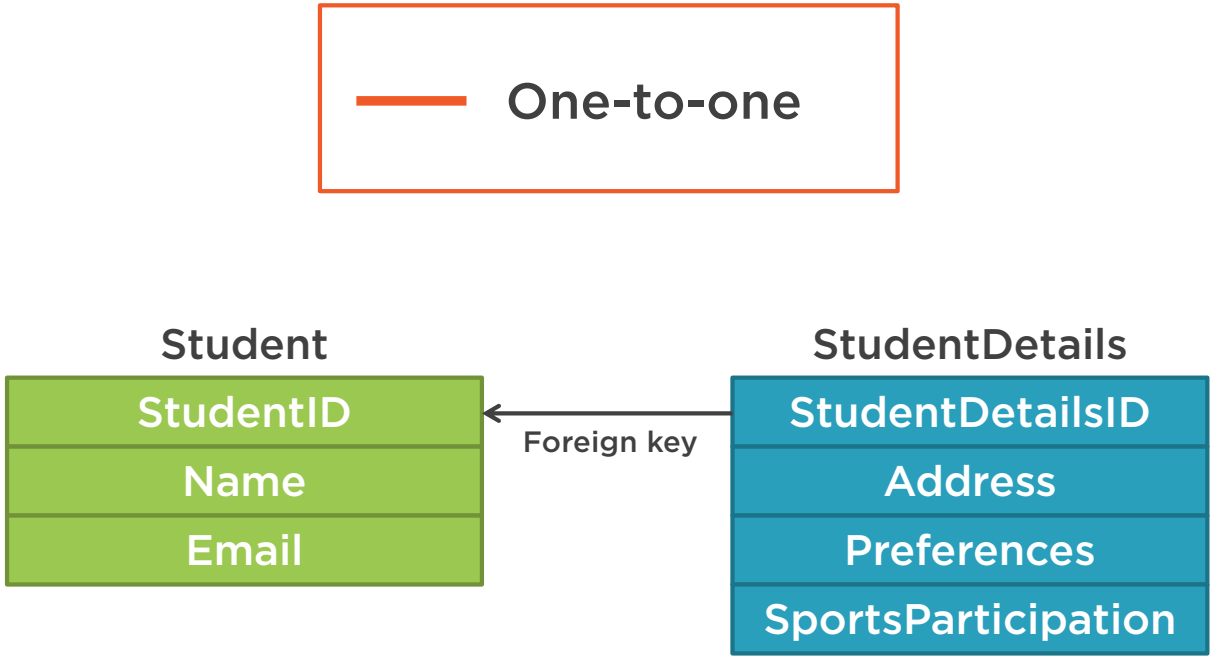
```
public class Student
{
    public long Id { get; private set; }
    public string Name { get; private set; }
    public string Email { get; private set; }
    public long FavoriteCourseId { get; private set; }

    public Student(string name, string email, long favoriteCourseId)
    {
        Name = name;
        Email = email;
        FavoriteCourseId = favoriteCourseId;
    }
}
```

- ✓ No invalid state
- ✓ Public members must maintain all invariants

Types of Relationships

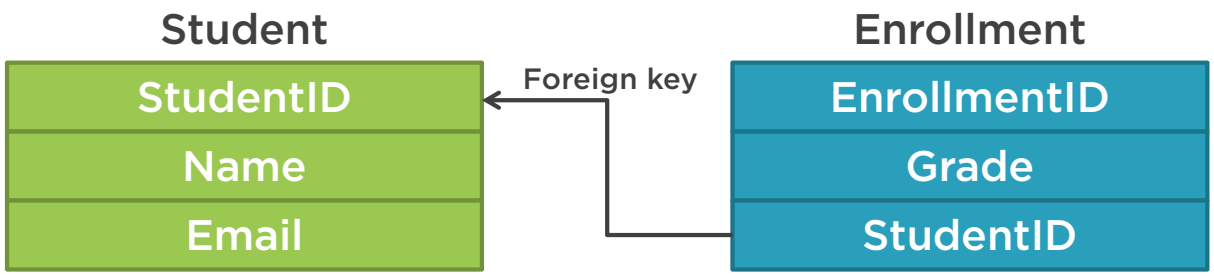
Types of Relationships




```
student.Details
```

```
studentDetails.Student
```

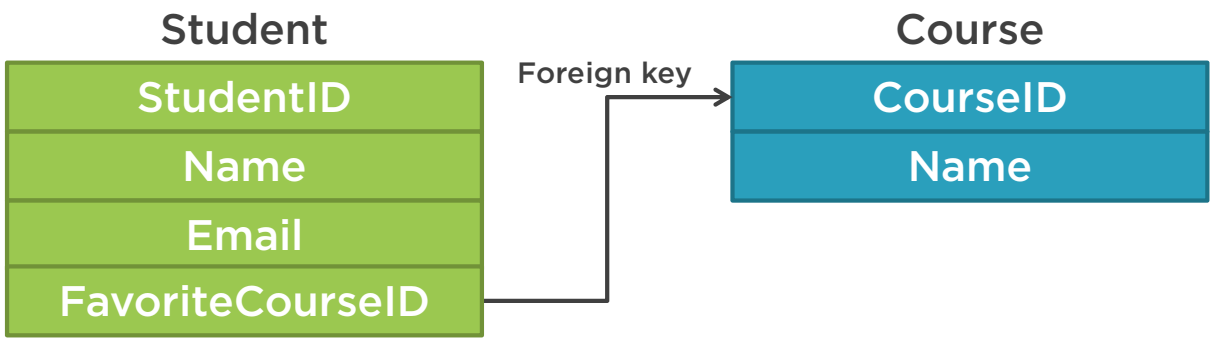
Types of Relationships




```
student.Enrollments
```

```
enrollment.Student
```

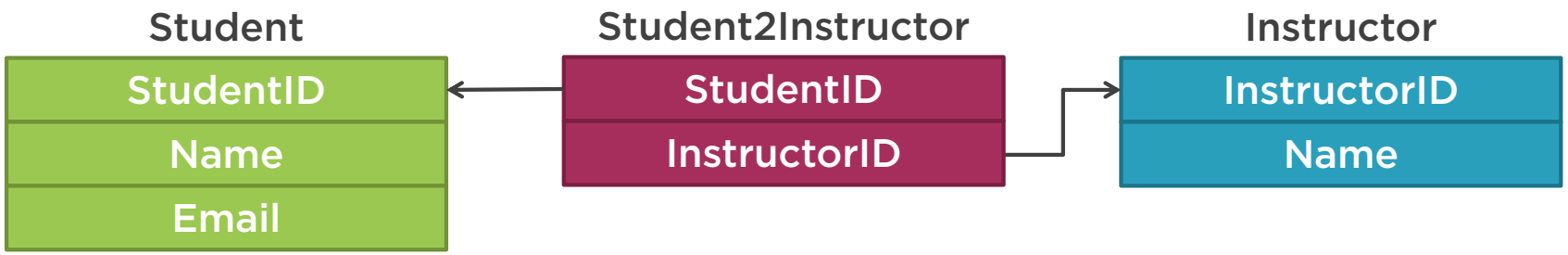
Types of Relationships




```
student.FavoriteCourse
```

```
course.Students
```

Types of Relationships



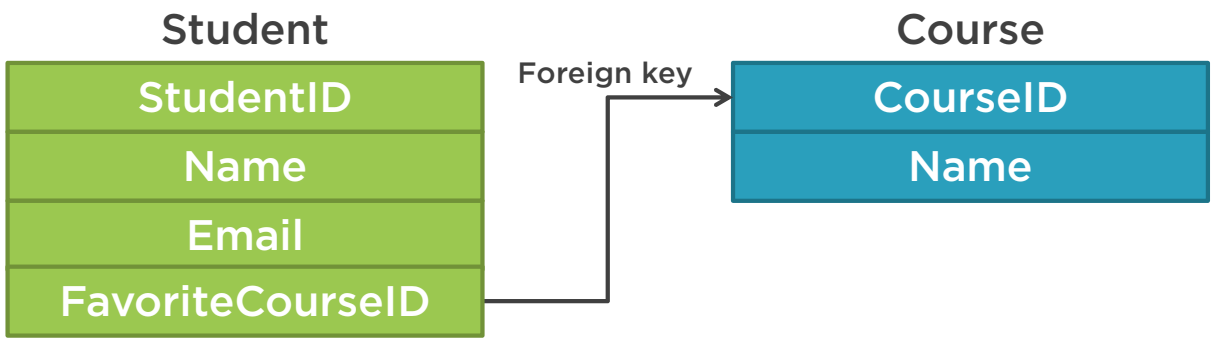

```
student.Instructors
```

```
instructor.Students
```

Types of Relationships

The fewer relationships in the domain model, the better

Types of Relationships




```
student.FavoriteCourse
```

~~course.Students~~

Reduce the number of relationships

IDs vs. Navigation Properties

```
public class Student
{
    /* ... */
    public long FavoriteCourseId { }
}
```

```
public class Student
{
    /* ... */
    public Course FavoriteCourse { }
}
```

Prefer navigation properties over IDs

IDs vs. Navigation Properties

How would the domain model look with no need for persistence?

No need for an ID

Use a direct reference

IDs vs. Navigation Properties

Separation of domain and database concerns = **Persistence ignorance**

Not always possible to achieve full persistence ignorance

IDs vs. Navigation Properties

Use navigation properties in place of IDs

IDs vs. Navigation Properties

Surrogate IDs

Generated by the database

Natural IDs

IDs vs. Navigation Properties

```
public class Student
{
    public long Id { }
    public string Name { }
    public string Email { }
    public Course FavoriteCourse { }
}
```

Surrogate ID

Natural ID

```
public class Course
{
    public long Id { }
    public string Name { }
}
```

Surrogate ID

IDs vs. Navigation Properties

Surrogate IDs

Generated by the database

Natural IDs

Part of domain model

Surrogate IDs violate the principle of Separation of Concerns

IDs vs. Navigation Properties

Entity's own ID doesn't violate SoC

```
public class Student
{
    public long Id { }
    public string Name { }
    public string Email { }
    public long FavoriteCourseId { }
}
```

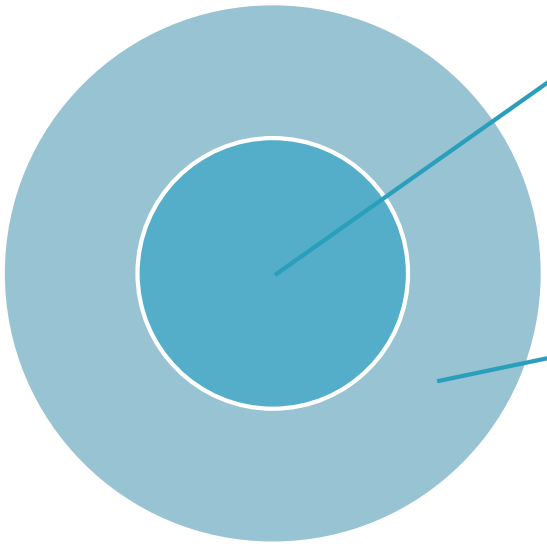
Represents entity's identity

Can deal with own identity

Shouldn't deal with identities of others

IDs vs. Navigation Properties

Persistence Ignorance only applies to the domain model



Entities
Aggregates
Value Objects
Domain Events
Pure Domain Services

Domain model is persistence-ignorant

Repositories
Impure Domain Services
Application Services

Application services are not persistence-ignorant

IDs vs. Navigation Properties

```
// Controller (application service)
public void EnrollStudent(long studentId, long courseId)
{
    Student student = _dbContext.Students.Find(studentId);
    Course course = _dbContext.Courses.Find(courseId);

    student.EnrollIn(course);

    _dbContext.SaveChanges();
}
```

Prepares data

Delegates work to domain model

Persists the results

Use IDs outside the domain model

Don’t use IDs inside the domain model

Recap: Refactoring to Navigation Properties

- Refactored from an ID to a navigation property
- Use navigation properties instead of IDs
  - Only applies to IDs of related entities
  - Only applies to the domain layer

Summary

Encapsulated the DbContext

- Expose as low of a configuration surface as possible
- Only allow to configure things that change depending on the environment

Make all property setters in the domain model private by default

Types of relationships

Keep as few relationships as possible in the domain model

Use navigation properties instead of IDs in the domain model

In the Next Module

Working with Lazy Loading