Encapsulation

Benefits of Encapsulation

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Hiding Implementation

Encapsulation

- Process of wrapping code and data together into a **single unit**
- Flexibility and extensibility of the code
- Allows **validation** and **data binding**
- Structural changes remain **local**
- Reduces **complexity**

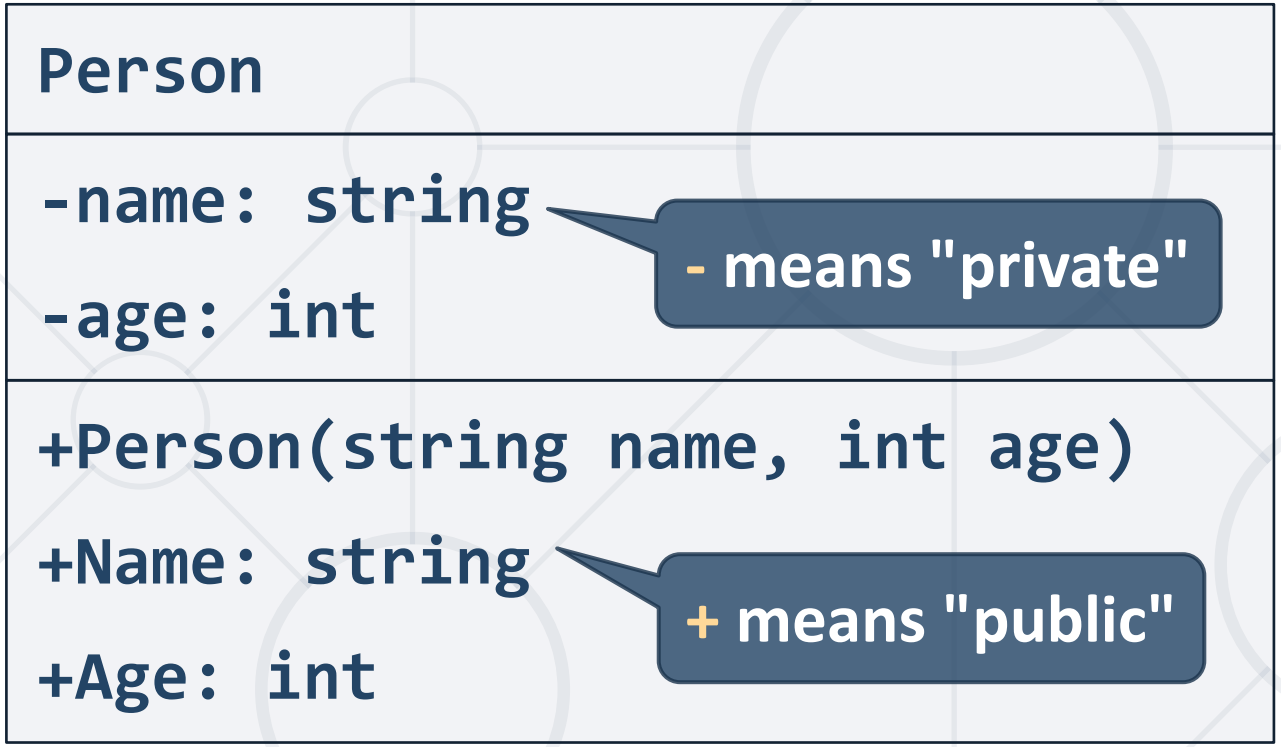
```
public class Student
{
   private string studentName;

   public string Name
   {
      get { return studentName; }
      set { studentName = value; }
   }
}
```

accessible only by public methods of class

accessors to get and set value

Encapsulation – Example

- Fields should be **private**



- Properties should be **public**

Visibility of Class Members

Private Access Modifier

- It's the main way to perform encapsulation and **hide data** from the outside world

```
private string name;
Person (string name)
{
  this.name = name;
}
```

- The **default field** and **method** modifier is **private**
- **Avoid** declaring **private classes** and **interfaces**
  - accessible only within the declared class itself

Public Access Modifier (1)

- The most **permissive** access level
- There are **no restrictions** on accessing public members

```
public class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
}
```

Public Access Modifier (2)

- To access class directly from a namespace use the **using** keyword to include the namespace

```
namespace Mathematical
{
   public class Basic
   {
      public double PI = 3.14;
   }
}
```

```
using System;
using Mathematical;

namespace Distinct
{
   public class Program
   {
      Console.WriteLine(Basic.Pi);
   }
}
```

Internal Access Modifier

- `internal` is the **default class** access modifier

```
class Person
{
    internal string Name { get; set; }
    internal int Age { get; set; }
}
```

- **Accessible** to any other class **in the** same **project**

```
Team rm = new Team("Real");
rm.Name = "Real Madrid";
```

Problem: Sort Persons by Name and Age

- Sort persons by name and age
- Create a class **Person**, which should have **public properties** with **private setters** for:

Person
+FirstName:string
+LastName:string
+Age:int
+ToString():string

Solution: Sort Persons by Name and Age (1)

```
public class Person
{
  // TODO: Add a constructor
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public int Age { get; private set; }
  public override string ToString()
  {
    return $"{FirstName} {LastName} is {Age} years old.";
  }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3163#0

Solution: Sort Persons by Name and Age (2)

```
var lines = int.Parse(Console.ReadLine());
var people = new List<Person>();
for (int i = 0; i < lines; i++)
{
    var cmdArgs = Console.ReadLine().Split();
    // Create variables for constructor parameters
    // Initialize a Person
    // Add it to the list
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3163#0

Solution: Sort Persons by Name and Age (3)

```
//continued from previous slide
var sorted = people.OrderBy(p => p.FirstName)
   .ThenBy(p => p.Age).ToList();

Console.WriteLine(string.Join(Environment.NewLine, sorted));
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3163#0

Problem: Salary Increase

- Expand `Person` with `Salary`
- Add getter for `Salary`
- Add a method, which updates `Salary` with a given percent
- Persons younger than 30 get half of the normal increase

Person
+FirstName: string
+Age: int
+Salary: decimal
+IncreaseSalary(decimal): void
+ToString(): string

Solution: Salary Increase

```
public decimal Salary { get; private set; }
public void IncreaseSalary(decimal percentage)
{
  if (this.Age > 30)
    this.Salary += this.Salary * percentage / 100;
  else
    this.Salary += this.Salary * percentage / 200;
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3163#1

Validation in Getters or Setters

Validation (1)

- Setters are a good place for simple **data validation**

```
public decimal Salary
{
   get { return this.salary }
   set {
      if (value < 460)
         throw new ArgumentException("...");
      this.salary = value;
         }
}
```

Throw exceptions

- Callers of your methods should take care of **handling** exceptions

Validation (2)

- Constructors use **private setters** with validation logic

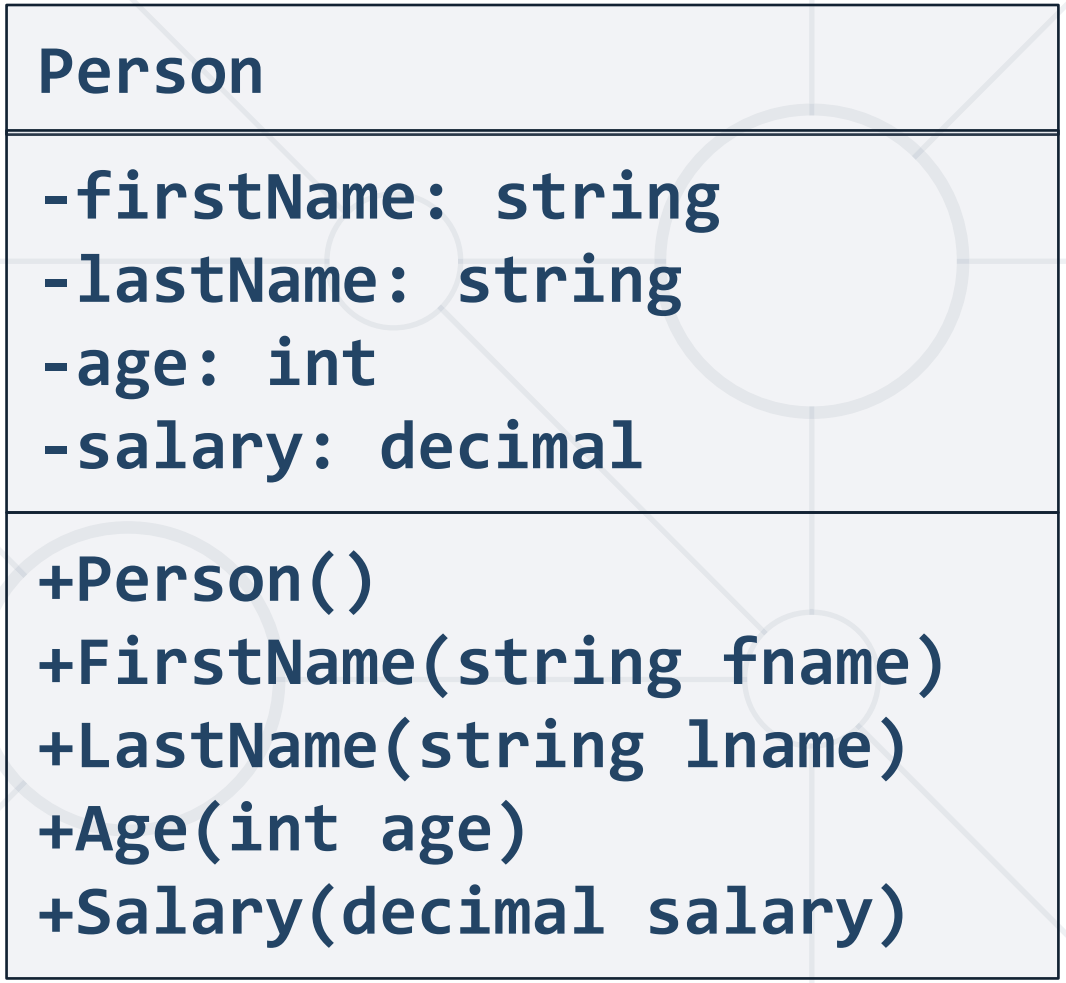
```
public Person(string firstName, string lastName, int age, decimal salary)
{
  this.FirstName = firstName;
  this.LastName = lastName;
  this.Age = age;
  this.Salary = salary;
}
```

Validation happens inside the setter

- Guarantee **valid state** of the object after its creation

Problem: Validate Data

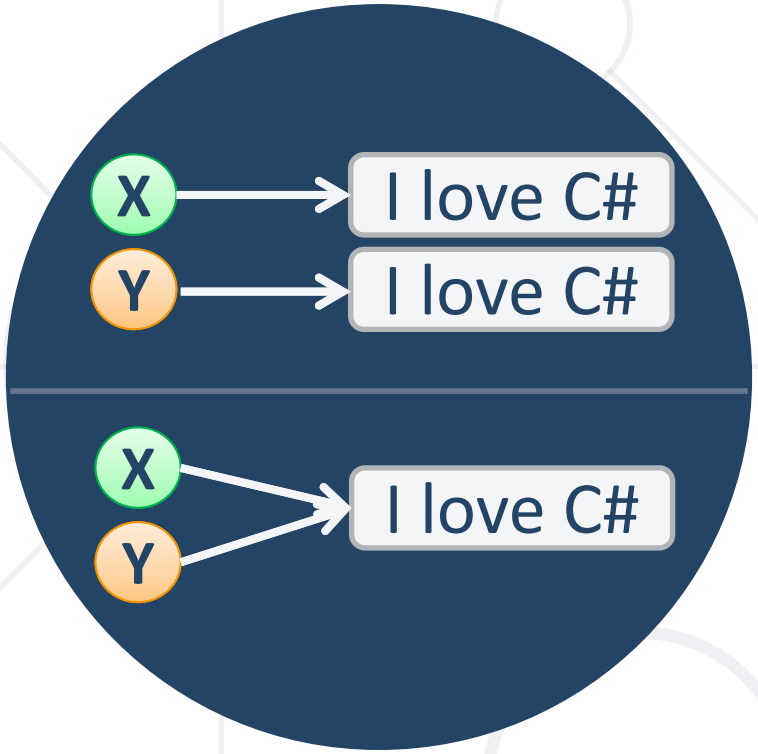
- Expand **Person** with validation for every field
- Names must be at least 3 symbols
- Age cannot be zero or negative
- Salary cannot be less than 460



Solution: Validate Data

```
public int Age
{
  get => this.age;
  private set
  {
    if (age < 1)
      throw new ArgumentException("...");
    this.age = value;
  }
}
// TODO: Add validation for the rest
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3163#2



Changeable and Unchangeable

Mutable vs Immutable Objects

- **Mutable Objects**
  - Mutable == changeable
  - Use the same memory location
  - `StringBuilder`
  - `List`
- **Immutable Objects**
  - Immutable == unchangeable
  - Create new memory every time they're modified
  - `string`
  - `int`

Mutable Fields

- **Private mutable** fields are still **not encapsulated**

```
class Team
{
  private List<Person> players;
  public List<Person> Players { get { return this.players; } }
}
```

- In this case you can **access** the field methods through the **getter**

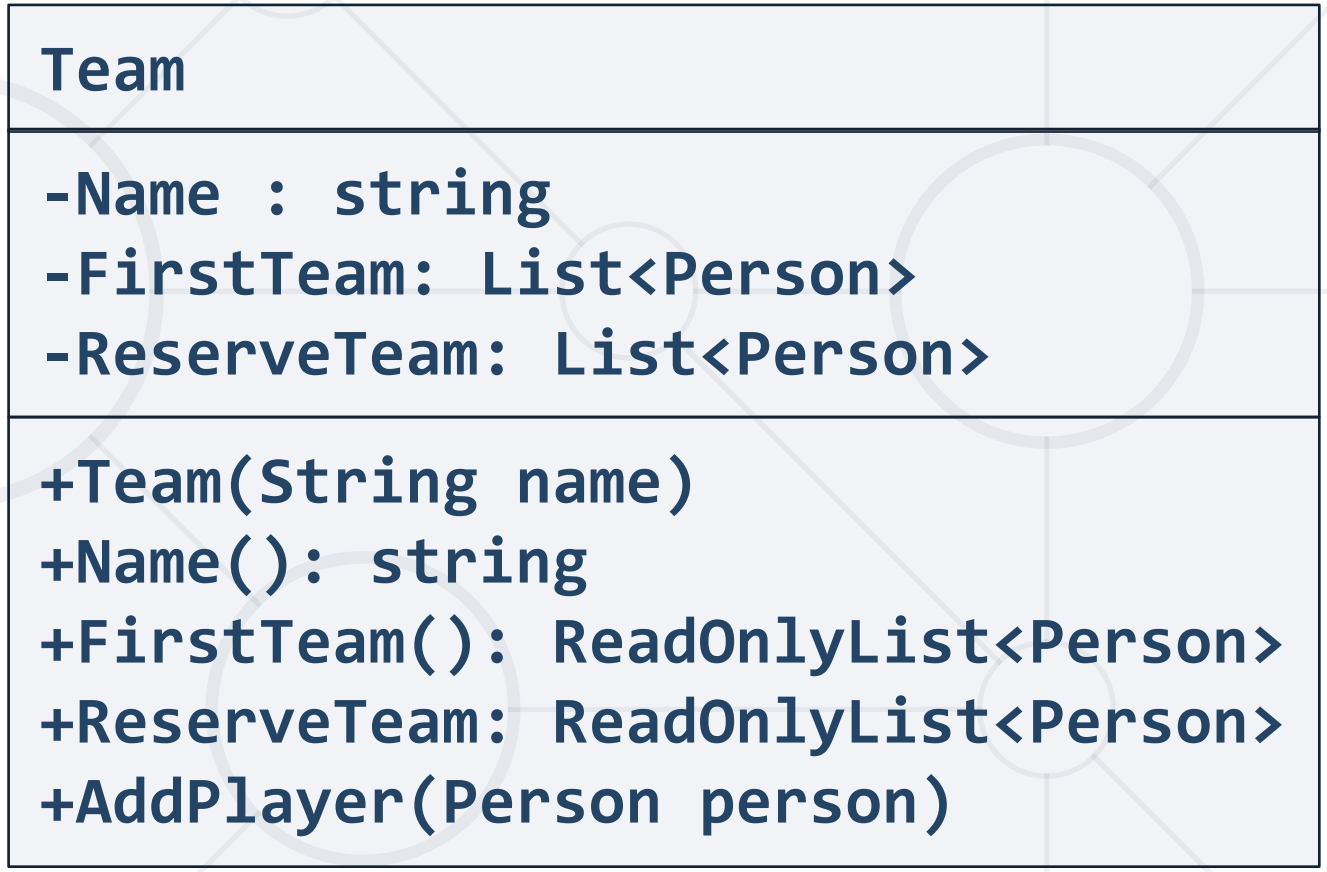
Encapsulate Mutable Fields

- You can use `IReadOnlyCollection` to encapsulate collections

```
public class Team
{
  private List<Person> players;
  public IReadOnlyCollection<Person> Players
  {
    get { return this.players.AsReadOnly(); }
  }
  public void AddPlayer(Person player)
    => this.players.Add(player);
}
```

Problem: Team

- Team have two squads
  - First team & Reserve team
- Read persons from console and add them to team
- If they are younger than 40, they go to first squad
- Print both squad sizes



Solution: Team (1)

```
private string name;
private List<Person> firstTeam;
private List<Person> reserveTeam;

public Team(string name)
{
    this.name = name;
    this.firstTeam = new List<Person>();
    this.reserveTeam = new List<Person>();
}
// continues on the next slide
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3163#3

Solution: Team(2)

```
public IReadOnlyCollection<Person> FirstTeam
{
  get { return this.firstTeam.AsReadOnly(); }
}
// TODO: Implement reserve team getter
public void AddPlayer(Person player)
{
  if (player.Age < 40)
    firstTeam.Add(player);
  else
    reserveTeam.Add(player);
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3163#3

Summary

- Encapsulation:
  - Hides **implementation**
  - Reduces **complexity**
  - Ensures that structural changes remain local
- **Immutable** and **Mutable** objects

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg