

# Inheritance

## Class Hierarchies



**SoftUni Team**  
**Technical Trainers**



**Software University**  
<https://about.softuni.bg/>

# Table of Contents

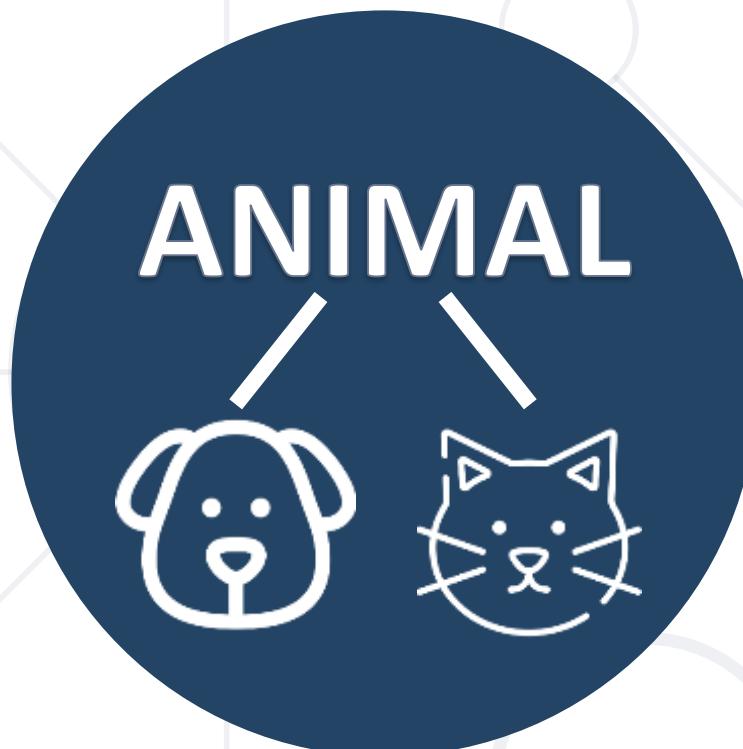
1. Inheritance
2. Class Hierarchies
  - Inheritance in C#
3. Accessing Base Class Members
4. Reusing Classes
5. Type of Class Reuse

Have a Question?



sli.do

#csharp-advanced



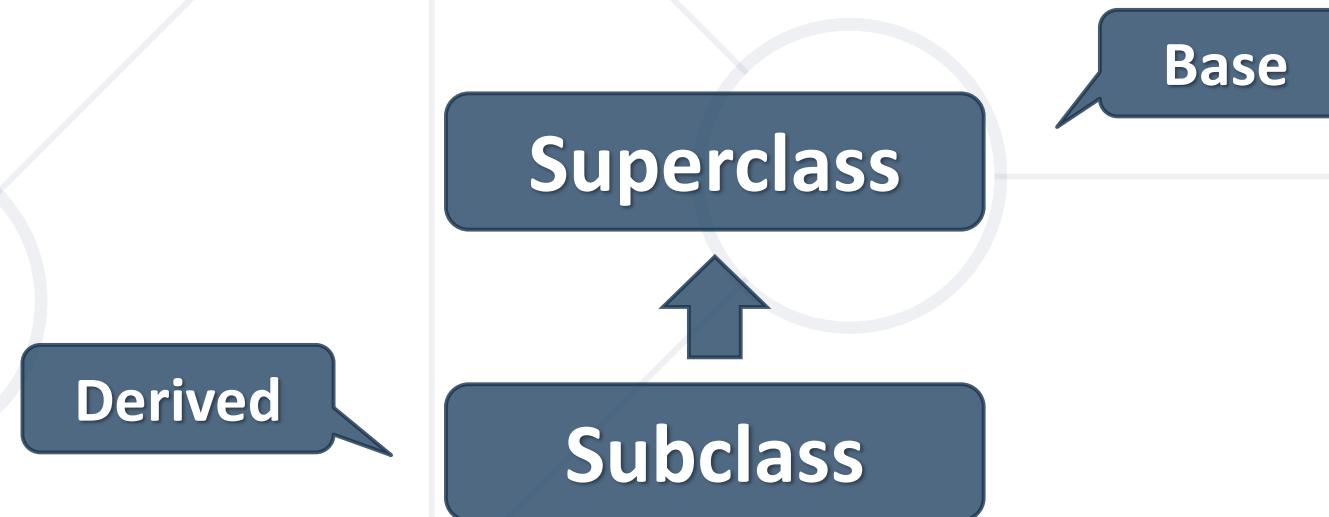
**ANIMAL**

**Inheritance**

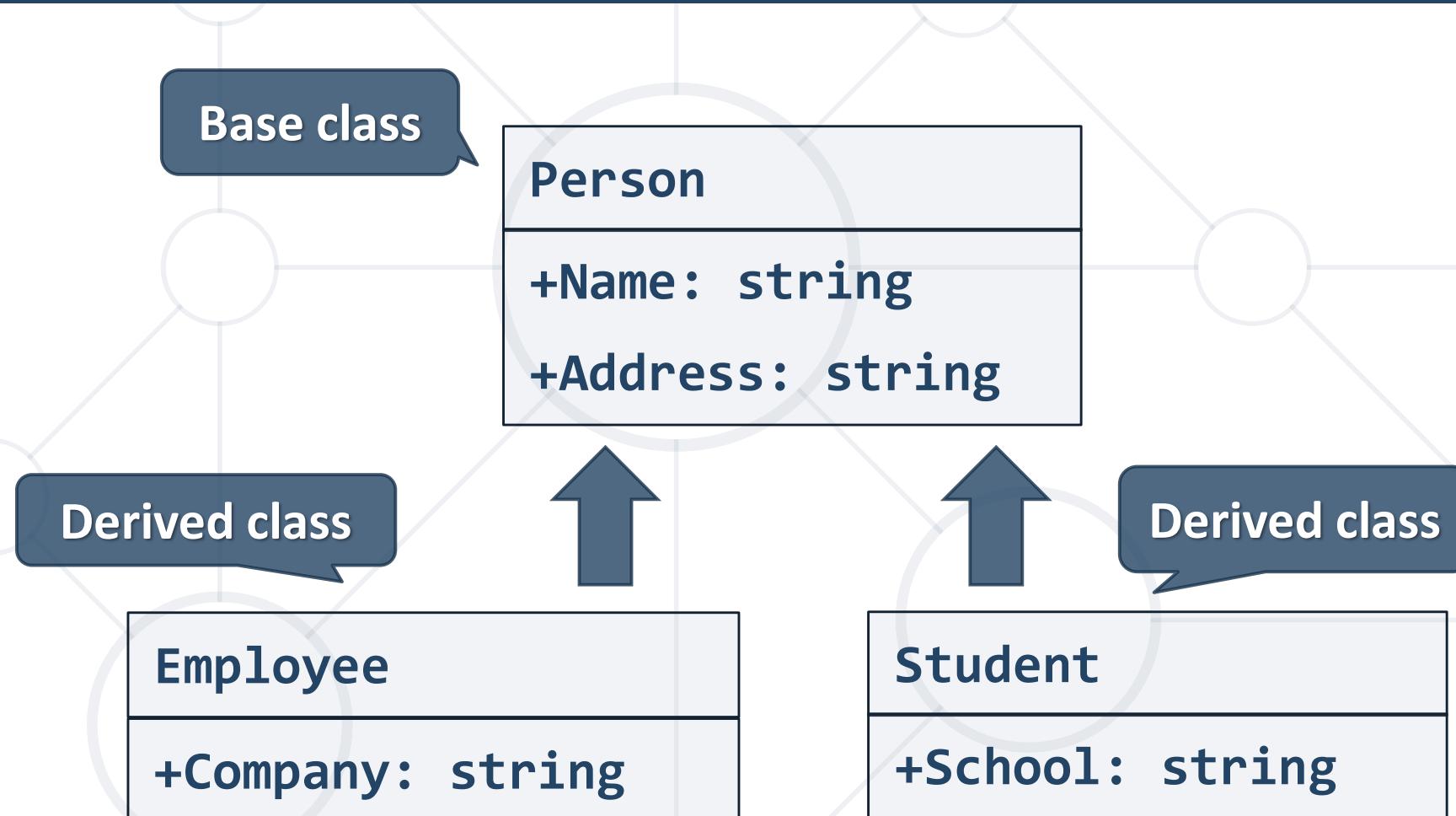
**Extending Classes**

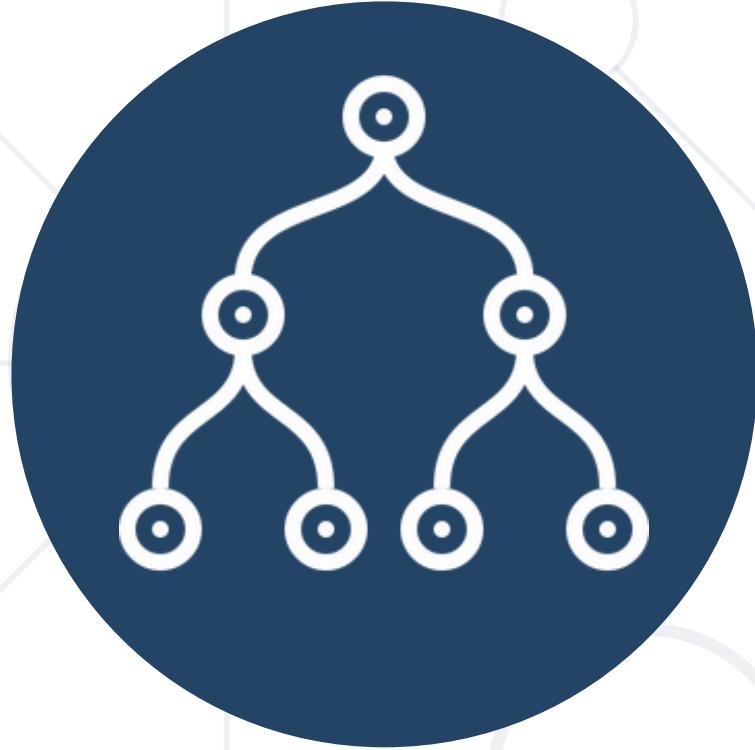
# Inheritance

- **Superclass** - Parent class, Base Class
  - The class giving its **members** to its **child class**
- **Subclass** - Child class, **Derived class**
  - The class taking members from its base class



# Inheritance – Example

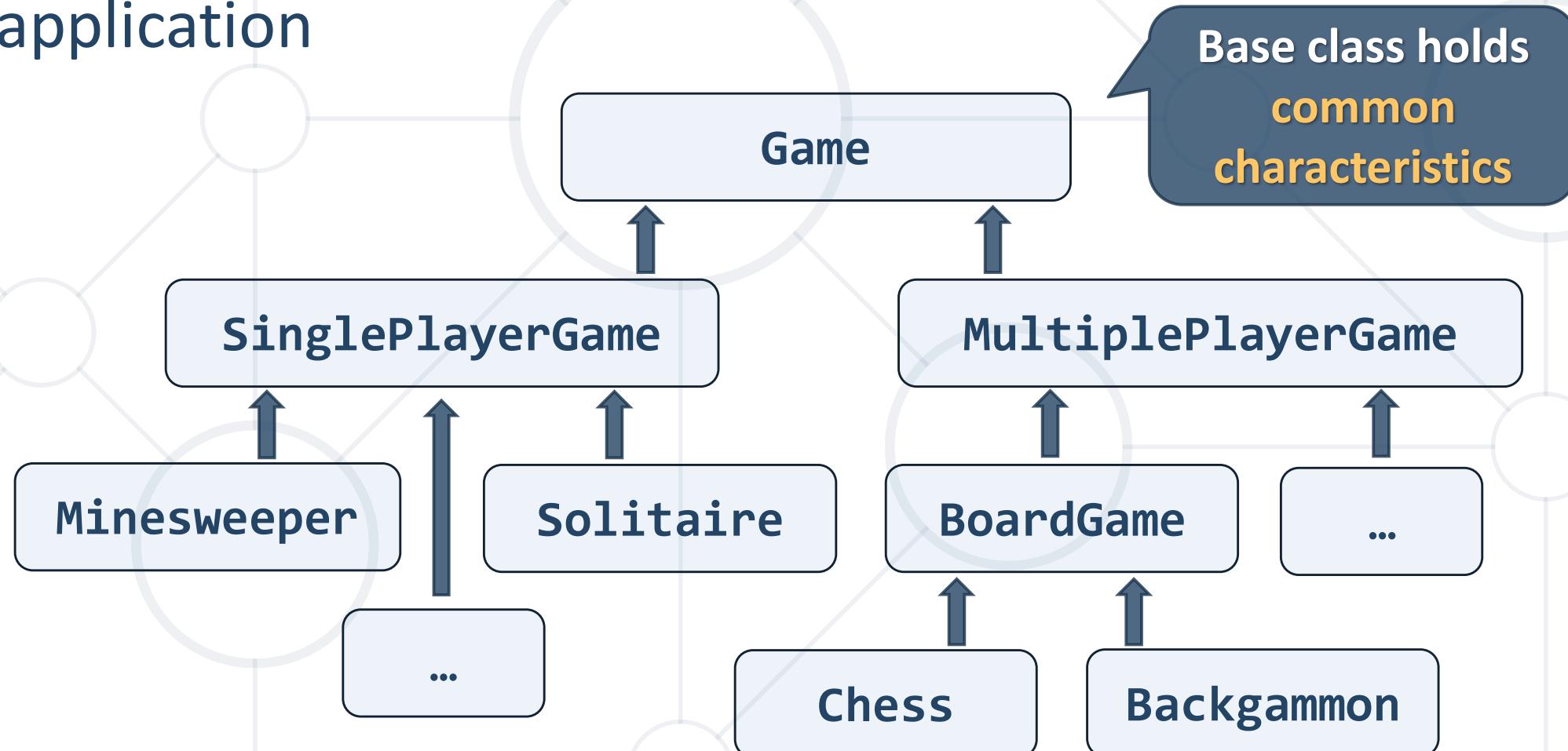




# Class Hierarchies

# Class Hierarchies

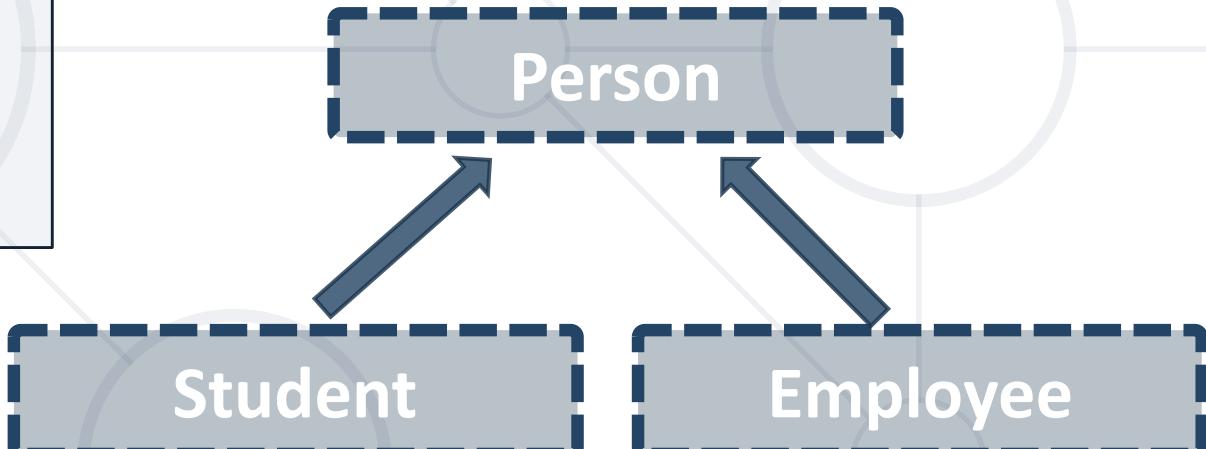
- **Inheritance** leads to **hierarchies** of classes and/or interfaces in an application



# Inheritance in C#

- In C# inheritance is defined by the `:` operator

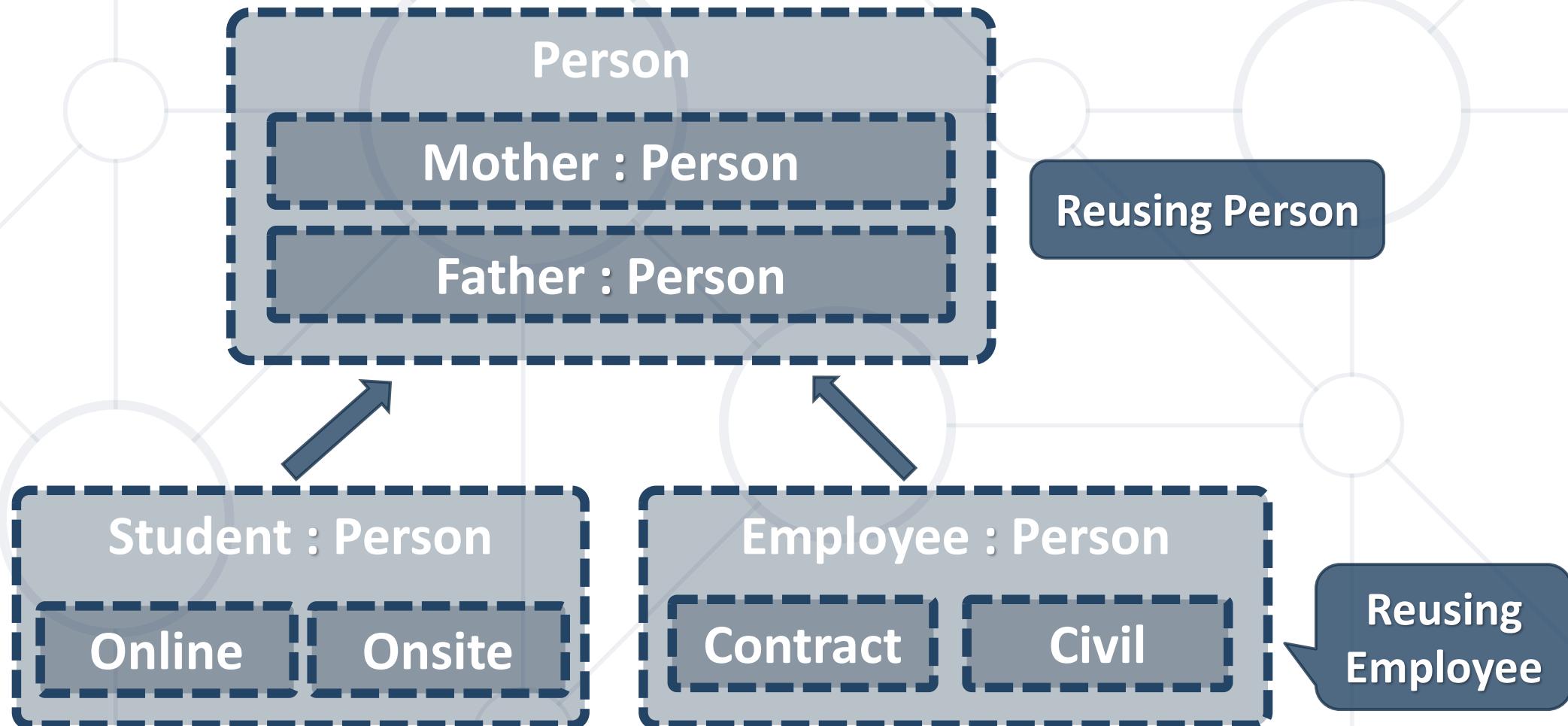
```
class Person { ... }  
class Student : Person { ... }  
class Employee : Person { ... }
```



Student : Person

# Inheritance - Derived Class

- Derived classes **take all members** from base classes



# Using Inherited Members

- You can access inherited members as usual

```
class Person { public void Sleep() { ... } }

class Student : Person { ... }

class Employee : Person { ... }
```

```
Student student = new Student();

student.Sleep();

Employee employee = new Employee();

employee.Sleep();
```

# Reusing Constructors

- Constructors are **not inherited**
- They can be **reused** by the child classes

```
class Student : Person {  
    private School school;  
    public Student(string name, School school)  
        :base(name) {this.school = school;}  
}
```

# Thinking about Inheritance - Extends

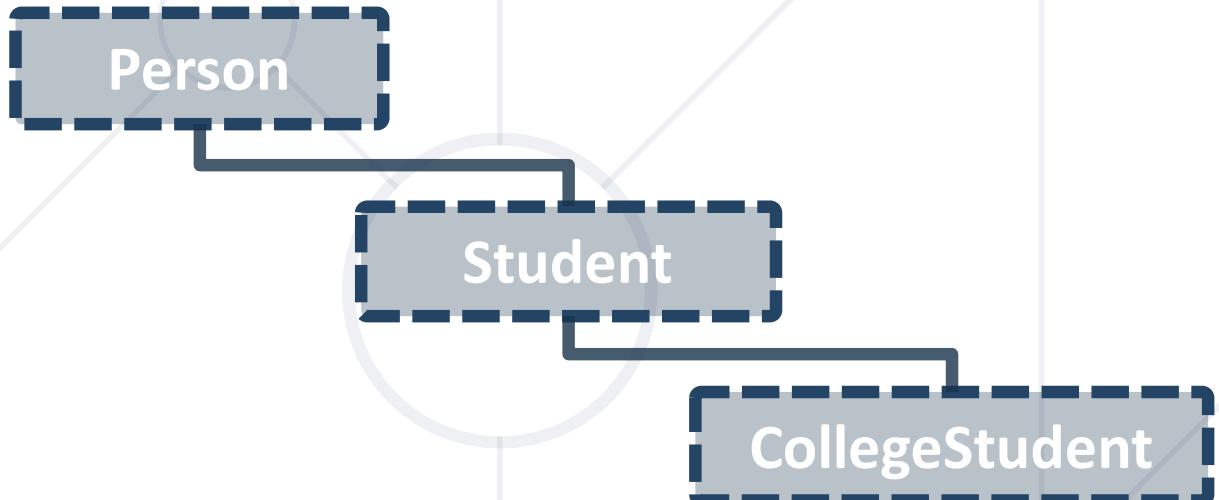
- Derived class instance **contains** instance of its base class



# Transitive Relation

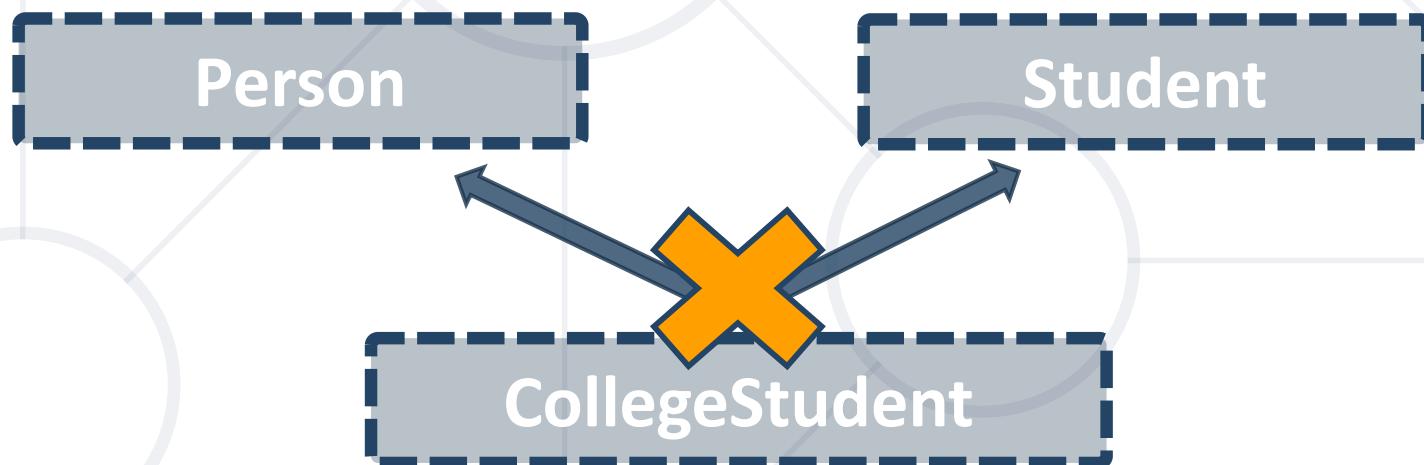
- Inheritance has a **transitive relation**

```
class Person { ... }  
class Student : Person { ... }  
class CollegeStudent : Student { ... }
```



# Multiple Inheritance

- In C# there is **no multiple** inheritance
- Only **multiple interfaces** can be implemented





# Accessing Base Class Members

# Access to Base Class Members

- Use the **base** keyword

```
class Person { ... }
class Employee : Person
{
    public void Dismiss(string reasons)
    {
        Console.WriteLine($"{base.name} got fired because of {reasons}");
    }
}
```

# Problem: Single Inheritance

Animal  
+Eat():void



Dog  
+Bark():void



```
Dog dog = new Dog();  
dog.Eat();  
dog.Bark();
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1499#0>

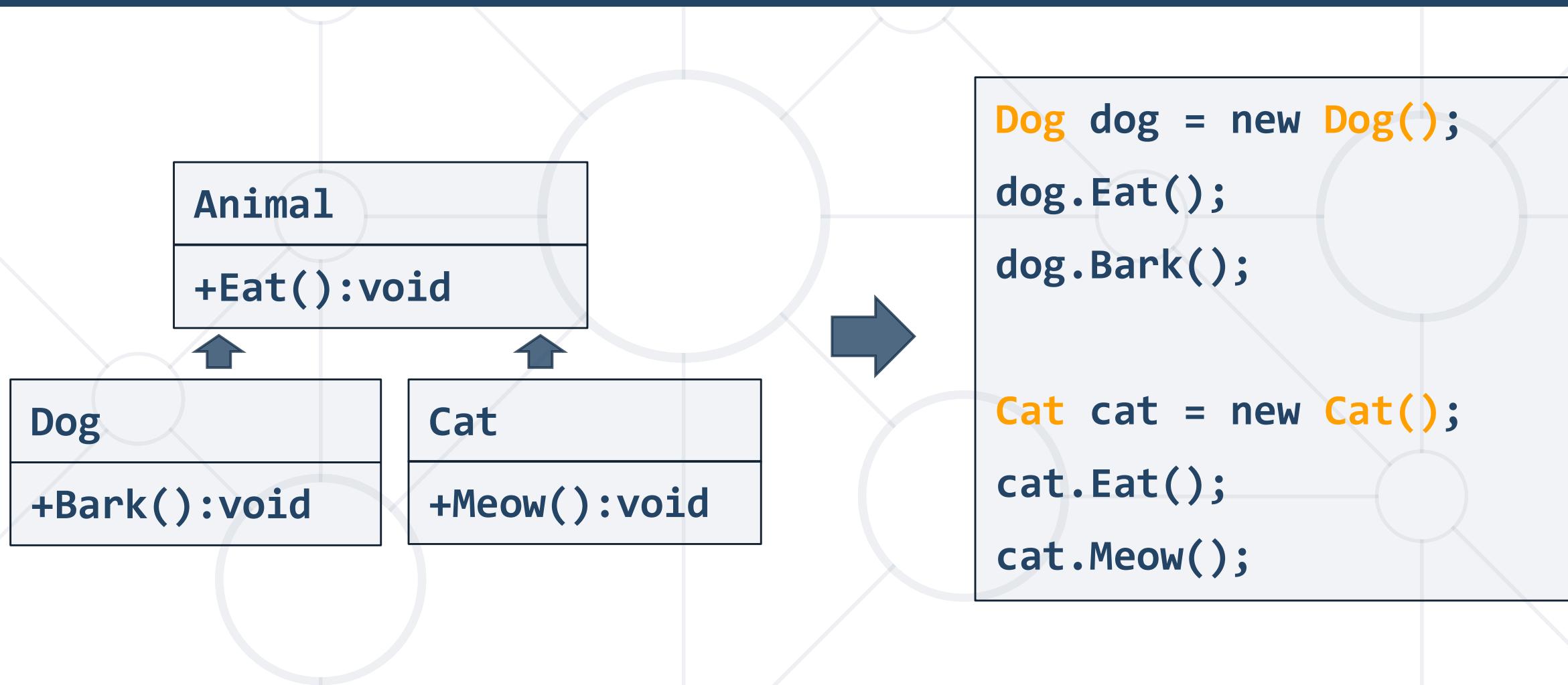
# Problem: Transitive Inheritance



```
Puppy puppy = new Puppy();
puppy.Eat();
puppy.Bark();
puppy.Weep();
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1499#1>

# Problem: Hierarchical Inheritance



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1499#2>



# Reusing Classes

Reusing Code at Class Level

# Inheritance and Access Modifiers

- Derived classes **can access all public and protected members**
- **Internal** members **are accessed in the same assembly**
- **Private** fields are **inherited**, but not visible in subclasses

```
class Person {  
    private string id;  
    string name;  
    protected string address;  
    public void Sleep(); }
```

# Shadowing Variables

- Derived classes **can hide** superclass variables

```
class Person { protected int weight; }
```

```
class Patient : Person
{
    protected float weight;
    public void Method()
    {
        double weight = 0.5d;
    }
}
```

Hides **int weight**

Hides **float weight**

# Shadowing Variables - Access

- Use **base** and **this** to specify member access

Base class member

```
class Patient : Person
{
    protected float weight;
    public void Method()
    {
        double weight = 0.5d;
        this.weight = 0.6f;
        base.weight = 1;
    }
}
```

Local variable

Instance member

# Virtual Methods

- Virtual - defines a method that **can be overriden**

```
public class Animal
{
    public virtual void Eat() { ... }

}

public class Dog : Animal
{
    public override void Eat() {}

}
```

# Sealed Modifier

- The **sealed** modifier prevents other classes from **inheriting** from it
- You can use the **sealed** modifier on a **method** or a **property** in a **base** class:
- It enables you to **allow classes** to **derive** from your class
- **Prevents** the **overriding** of specific **virtual methods** and properties



# Types of Class Reuse

Extension (Inheritance) and Composition

# Extension (Inheritance) (IS-A relation)

- **Duplicate code** is error prone
- **Reuse classes through extension**
- Sometimes the only way

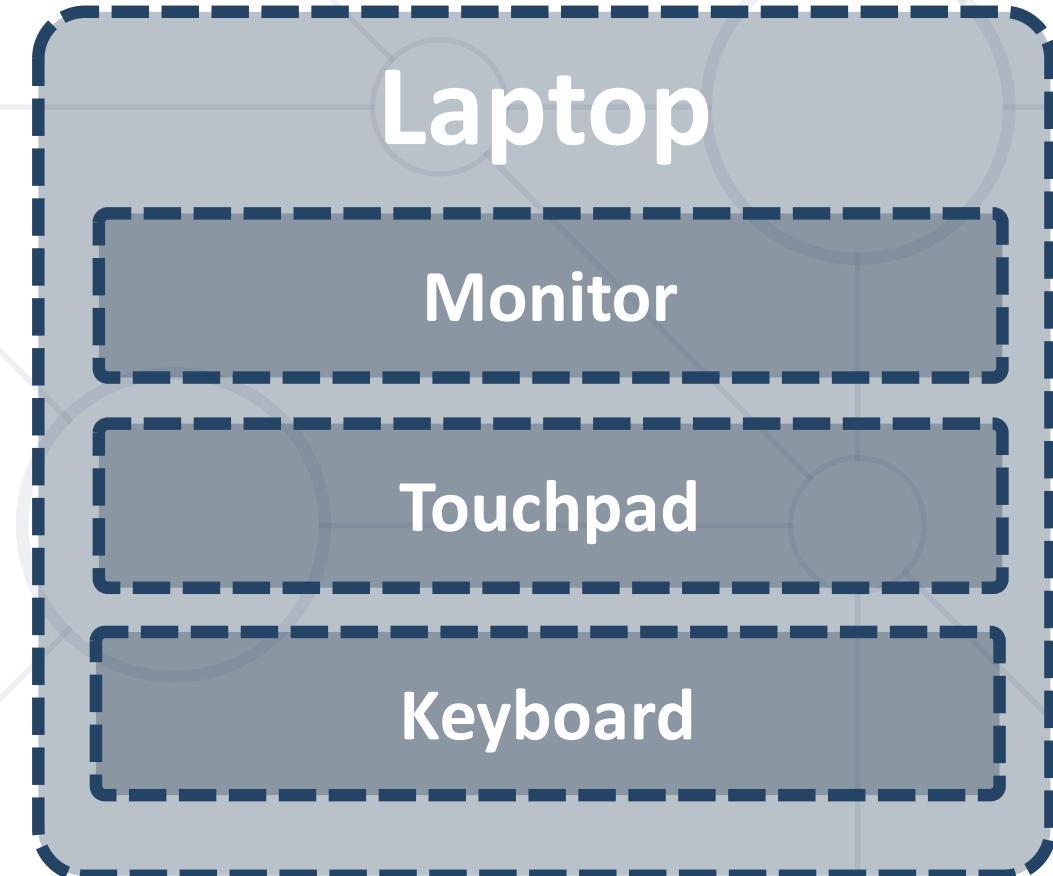


# Composition (HAS-A relation)

- Using classes to **define** classes

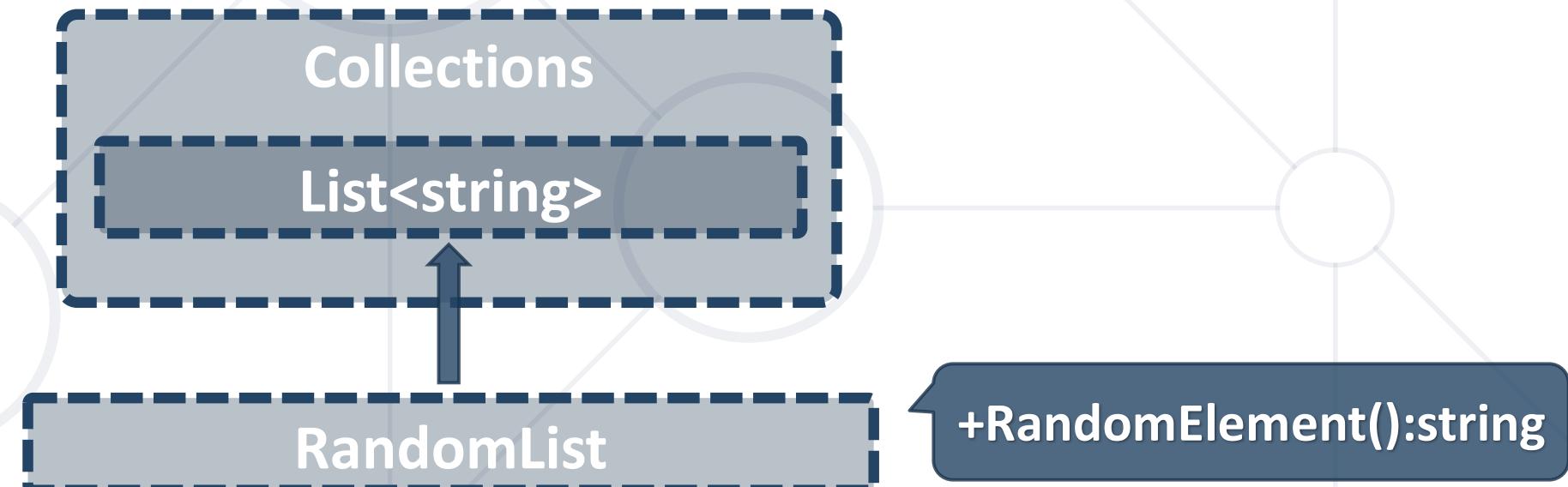
```
class Laptop {  
    Monitor monitor;  
    Touchpad touchpad;  
    Keyboard keyboard;  
    ...  
}
```

**Reusing  
classes**



# Problem: Random List

- Create a list that has
  - All functionality of a `List<string>`
  - Method that returns and removes a random element



# Solution: Random List

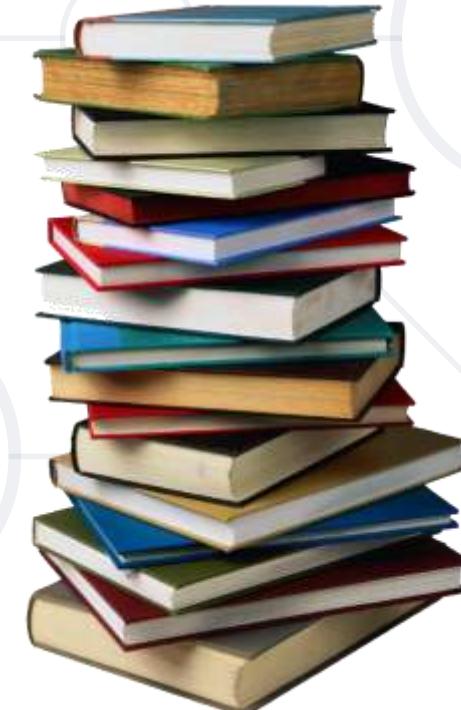
```
public class RandomList : List<string> {  
    private Random rnd; // TODO: Add constructor  
    public string RemoveRandomElement() {  
        int index = rnd.Next(0, this.Count);  
        string str = this[index];  
        this.RemoveAt(index);  
        return str;  
    }  
}
```

# Problem: Stack of Strings

- Create a simple **StackOfStrings** class which **inherits** the `Stack<string>`

**StackOfStrings**

`+IsEmpty(): Boolean`  
`+AddRange(): void`



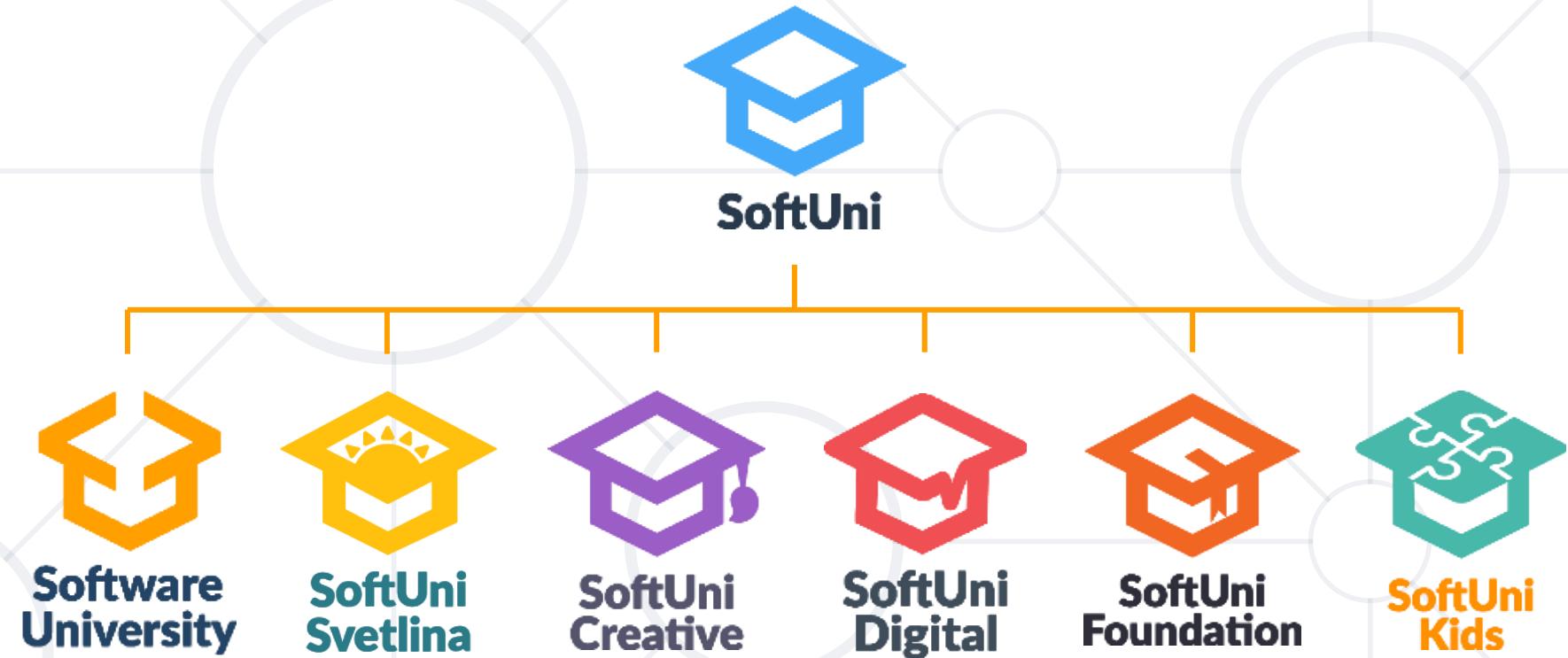
# Solution: Stack of Strings

```
public class StackOfStrings : Stack<string> {  
    public bool IsEmpty() {  
        return this.Count == 0;  
    }  
    public void AddRange(IEnumerable<string> collection) {  
        foreach (var element in collection)  
            this.Push(element);  
    }  
}
```

- Inheritance is a powerful tool for **code reuse**
- **Subclass inherits** members from **Superclass** and can **override** methods
- Look for classes with the **same role**
- Look for **IS-A** and **IS-A-SUBSTITUTE**
- Consider **Composition** and **Delegation**



# Questions?



# SoftUni Diamond Partners

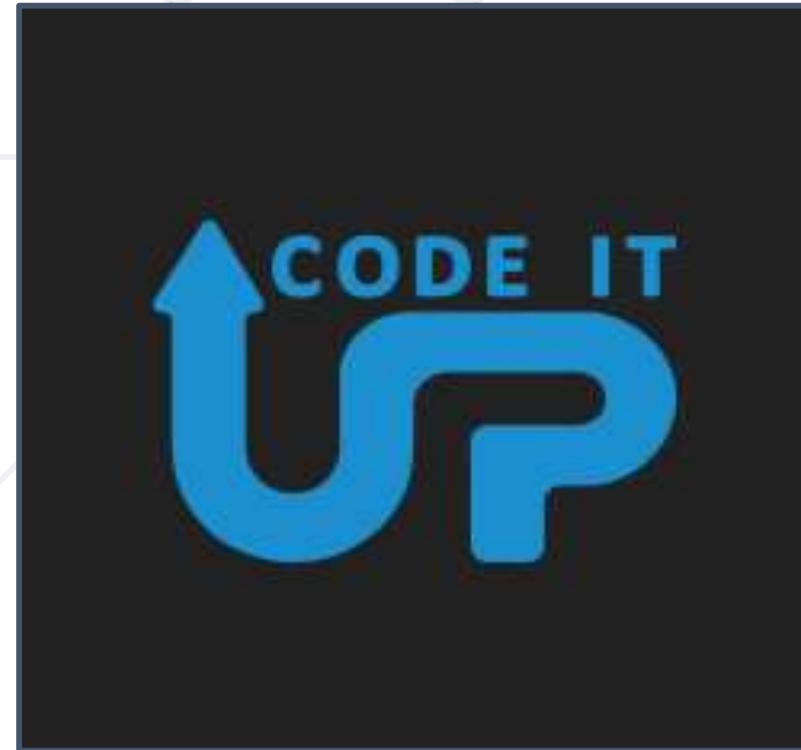


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Encapsulation

## Benefits of Encapsulation



SoftUni Team  
Technical Trainers



SoftUni



Software University  
<https://softuni.bg>

# Table of Contents

- Encapsulation
- Access Modifiers
- State Validation
- Mutable and Immutable Objects



sli.do

# #csharp-advanced



**Encapsulation**  
Hiding Implementation

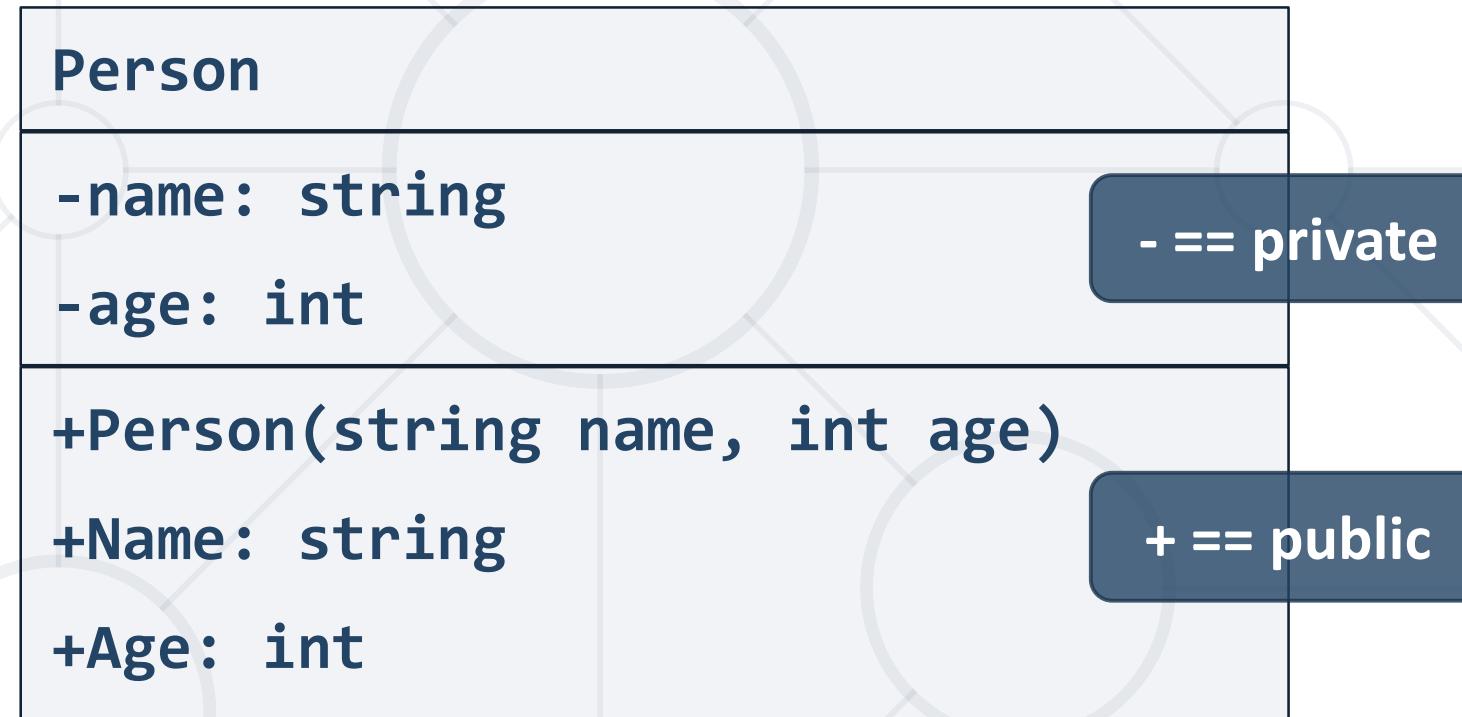
# Encapsulation

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



# Encapsulation – Example

- Fields should be **private**



- Properties should be **public**



# Keyword This

- Reference to the **current object**
- Refers to the **current instance** of the class
- Can be passed as a **parameter to other methods**
- Can be **returned** from method
- Can invoke **current class methods**





# Visibility of Class Members

Access Modifiers

# 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

- 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; }  
}
```

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

# 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 People by Name and Age

- Create a read-only class **Person**
- Read and sort people by first name and age

**Person**

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



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1497#0>

# Solution: Sort People 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.org/Contests/Practice/Index/1497#0>

# Solution: Sort People 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.org/Contests/Practice/Index/1497#0>

# Solution: Sort People 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.org/Contests/Practice/Index/1497#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

```
class Person {  
    public string FirstName { get; set; }  
    public int Age { get; set; }  
    public decimal Salary { get; set; }  
    public void IncreaseSalary(decimal percent) {  
        if (Age < 30) {  
            Salary += Salary * percent / 2;  
        } else {  
            Salary += Salary * percent;  
        }  
    }  
    public string ToString() {  
        return $"{FirstName} {Age} {Salary}";  
    }  
}
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1497#1>

# 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.org/Contests/Practice/Index/1497#1>



**Validation**

# Validation (1)

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

```
public decimal Salary {  
    get { return this.salary; }  
    set {  
        if (value < 650)  
            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 650

## Person

```
-firstName: string  
-lastName: string  
-age: int  
-salary: decimal
```

```
+Person()  
+FirstName  
+LastName  
+Age  
+Salary
```

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1497#2>

# 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.org/Contests/Practice/Index/1497#2>

# Mutable vs Immutable Objects

- Mutable Objects
  - Mutable == changeable
  - Use the same memory location
  - **StringBuilder**
  - **List**
- Immutable Objects
  - Immutable == unchangeable (read-only)
  - Create new memory every time they're modified
  - **string**
  - **Tuples**



# 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

# Immutable 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); // mutable now
}
```

# 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

## Team

```
-name : string
-firstTeam: List<Person>
-reserveTeam: List<Person>

+Team(string name)
+Name: string
+FirstTeam: ReadOnlyList<Person>
+ReserveTeam: ReadOnlyList<Person>
+AddPlayer(Person person)
```

# 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/1497/Encapsulation-Lab>

# Solution: Team (2)

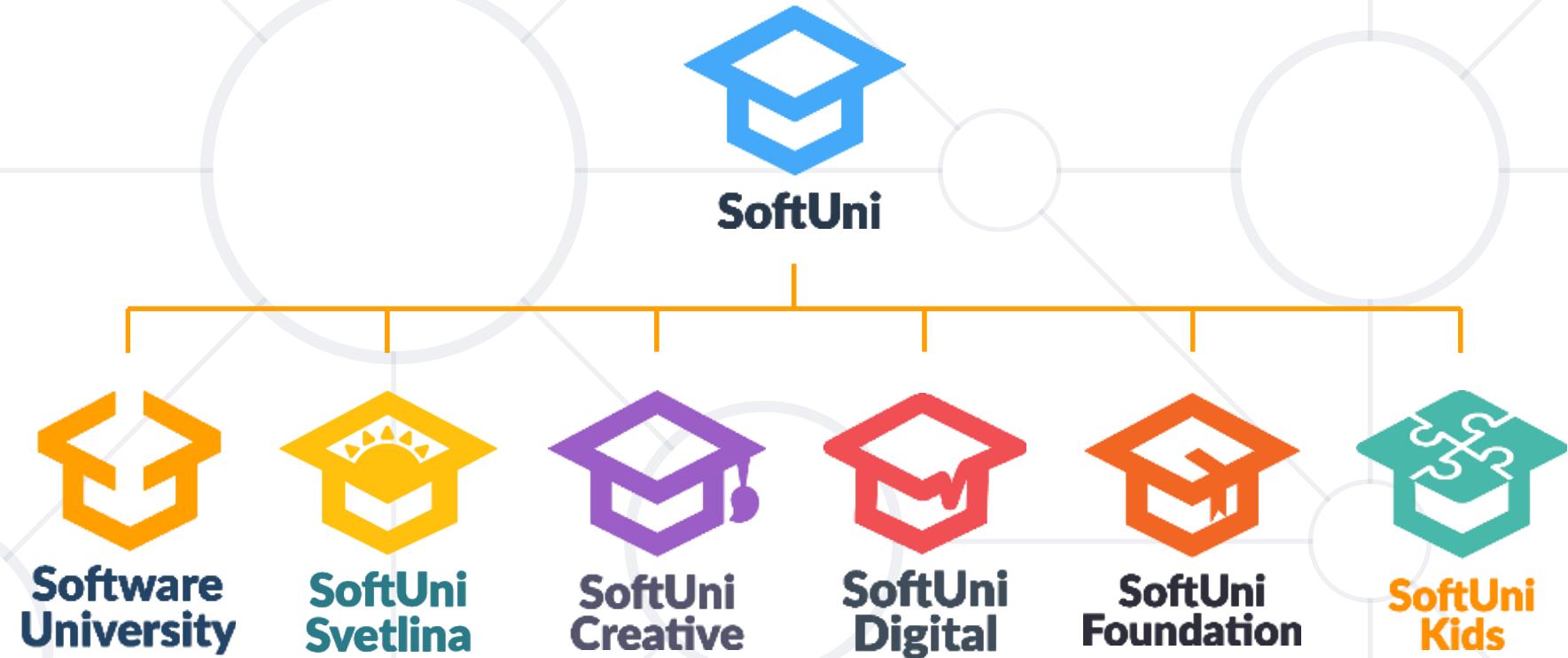
```
public IReadOnlyCollection<Person> FirstTeam {  
    get { return this.firstTeam.AsReadOnly(); }  
}  
// TODO: Implement reserve team getter  
public void AddPlayer(Person player) {  
    if (player.Age < 40)  
        firstTeam.Add(player);  
    else  
        reserveTeam.Add(player); }
```

Check your solution here: <https://judge.softuni.bg/Contests/1497/Encapsulation-Lab>

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



# Questions?



# SoftUni Diamond Partners

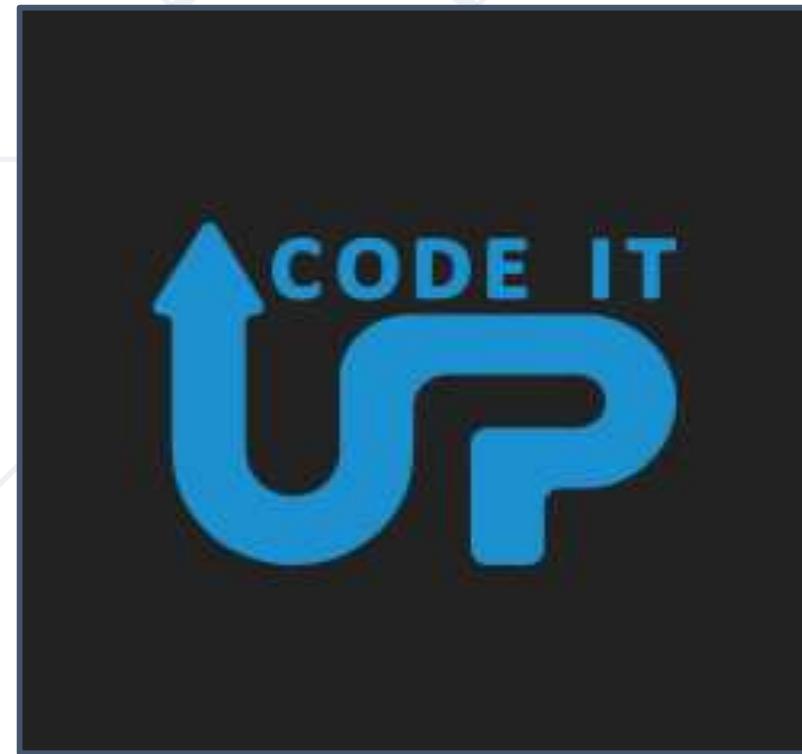


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

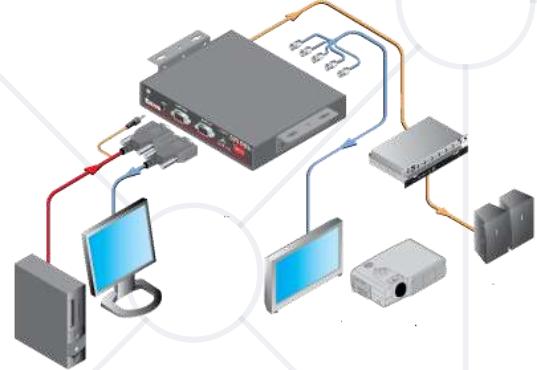


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



# Interfaces and Abstraction

Interfaces vs Abstract Classes  
Abstraction vs Encapsulation



SoftUni Team

Technical Trainers

 Software University



SoftUni



Software University  
<https://softuni.bg>

# Table of Contents

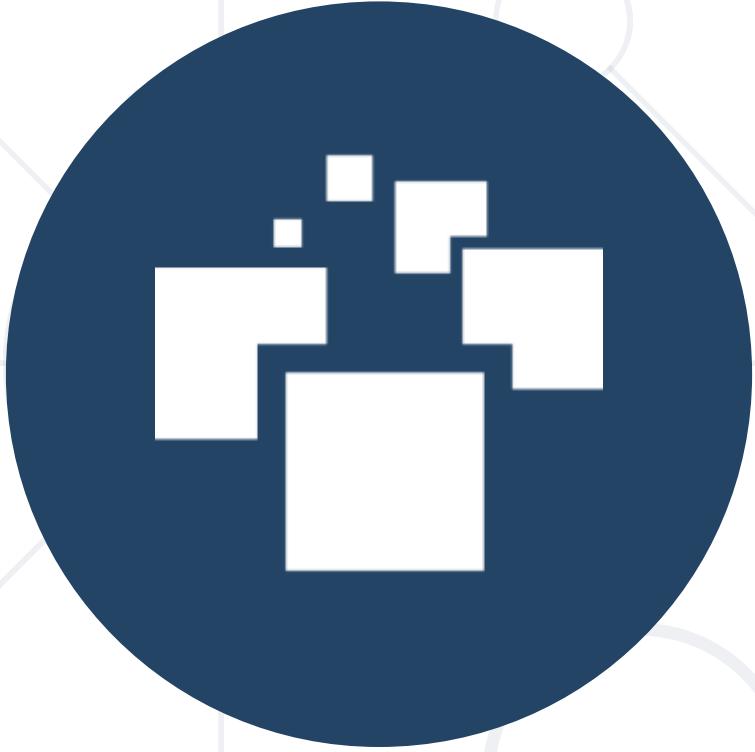
- Abstraction
- Interfaces
- Abstract Classes
- Interfaces vs Abstract Classes

Have a Question?



sli.do

**#csharp-advanced**



# Achieving Abstraction

Abstraction

# What is Abstraction?

- From the Latin

Abs  
(away from)

Trahere  
(to draw)

Abstraction

- Preserving information, relevant in a given context, and forgetting information that is irrelevant in that context



# Abstraction in OOP

- **Abstraction** means ignoring **irrelevant** features, properties, or functions and emphasizing the **ones** ...



- ... **relevant** to the **context** of the **project** we develop
- Abstraction helps **managing** complexity
- Abstraction lets you focus on **what the object does** instead of **how it does it**

# How Do We Achieve Abstraction?

- There are two ways to achieve abstraction
  - Interfaces
  - Abstract class

```
public interface IAnimal {}  
  
public abstract class Mammal {}  
  
public class Person : Mammal, IAnimal {}
```

# Abstraction vs Encapsulation

## ■ Abstraction

- Process of **hiding the implementation details** and showing only functionality to the user
- Achieved with **interfaces** and **abstract classes**

## ■ Encapsulation

- Used to **hide the code** and **data** inside a **single unit** to **protect the data from the outside world**
- Achieved with **access modifiers** (private, protected, public ... )





# Working with Interfaces

Interfaces

# Interface (1)

- Internal addition by compiler

```
public interface IPrintable {  
    void Print();  
}
```

Keyword

Name (starts with  
I per convention)

compiler

```
public interface IPrintable {  
    public abstract void Print();  
}
```



# Interface Example

- The implementation of **Print()** is provided in class **Document**

```
public interface IPrintable {  
    void Print();  
}
```

```
class Document : IPrintable {  
    public void Print()  
    { Console.WriteLine("Hello"); }}
```

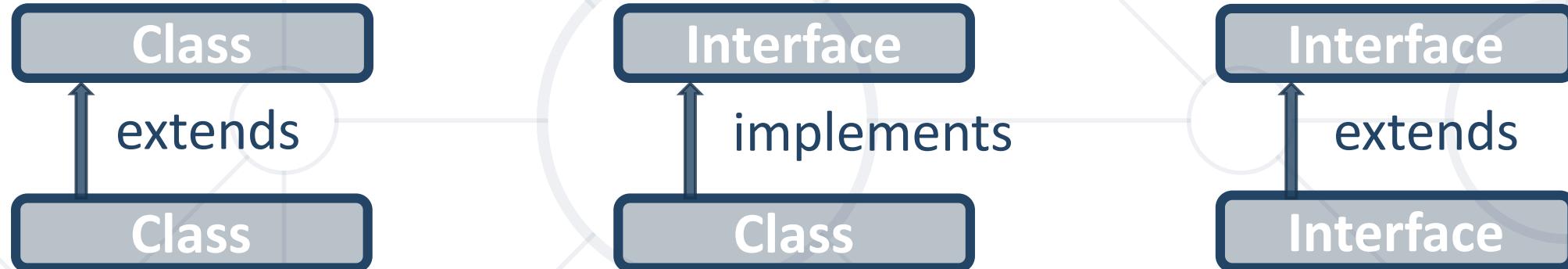


# Interface (2)

- Contains signatures of **methods** (in C# 8.0 interfaces could have a default implementation), **properties**, **events** or **indexers**
- Can **inherit one or more** base interfaces
- When a base type list contains a base class and interfaces, the **base class** must come **first** in the list
- A class that **implements** an interface can explicitly implement **members** of that **interface**
  - An explicitly implemented member **cannot** be accessed through a class instance, but only through the interface

# Multiple Implementation

- Relationship between **classes** and **interfaces**



- Multiple implementation and inheritance



# Problem: Shapes

- Build a project that contains an **interface** for **drawable objects**
- Implements two type of shapes: **Circle** and **Rectangle**
- Both classes have to print on the console their shape with "\*"

`<<IDrawable>>`

`Circle`

`+Radius: int`

`<<IDrawable>>`

`Rectangle`

`-Width: int`

`-Height: int`

`<<interface>>`

`IDrawable`

`+Draw()`

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1501#0>

# Solution: Shapes

```
public interface IDrawable {  
    void Draw();  
}
```

```
public class Rectangle : IDrawable {  
    // TODO: Add fields and a constructor  
    public void Draw() { // TODO: implement } }
```

```
public class Circle : IDrawable {  
    // TODO: Add fields and a constructor  
    public void Draw() { // TODO: implement } }
```

# Solution: Shapes – Rectangle Draw

```
public void Draw() {  
    DrawLine(this.width, '*', '*');  
    for (int i = 1; i < this.height - 1; ++i)  
        DrawLine(this.width, '*', ' ');  
    DrawLine(this.width, '*', '*'); }  
  
private void DrawLine(int width, char end, char mid) {  
    Console.Write(end);  
    for (int i = 1; i < width - 1; ++i)  
        Console.Write(mid);  
    Console.WriteLine(end); }
```

# Solution: Shapes – Circle Draw

```
double rIn = this.radius - 0.4;  
double rOut = this.radius + 0.4;  
  
for (double y = this.radius; y >= -this.radius; --y) {  
    for (double x = -this.radius; x < rOut; x += 0.5) {  
        double value = x * x + y * y;  
        if (value >= rIn * rIn && value <= rOut * rOut)  
            Console.Write("*");  
        else  
            Console.Write(" ");  
    }  
    Console.WriteLine();  
}
```



# Abstract Classes and Methods

Abstract Classes

# Abstract Class

- Cannot be instantiated
- May contain **abstract methods** and **accessors**
- Must provide **implementation** for all **inherited** interface members
- Implementing an interface might map the interface methods onto **abstract** methods



# Abstract Methods

- An **abstract method** is implicitly a **virtual** method
- Abstract method declarations are only permitted in **abstract classes**
- An abstract method declaration provides no actual implementation:

```
public abstract void Build();
```





# Interfaces vs Abstract Classes

# Interface vs Abstract Class (1)

- Interface
  - A class may **implement several interfaces**
  - **Cannot have access modifiers**, everything is assumed as public
  - **Cannot provide any code**, just the signature
- Abstract Class (AC)
  - May **inherit only one abstract** class
  - Can **provide implementation** and/or just the **signature** that have to be overridden
  - **Can contain access modifiers** for the fields, functions, properties



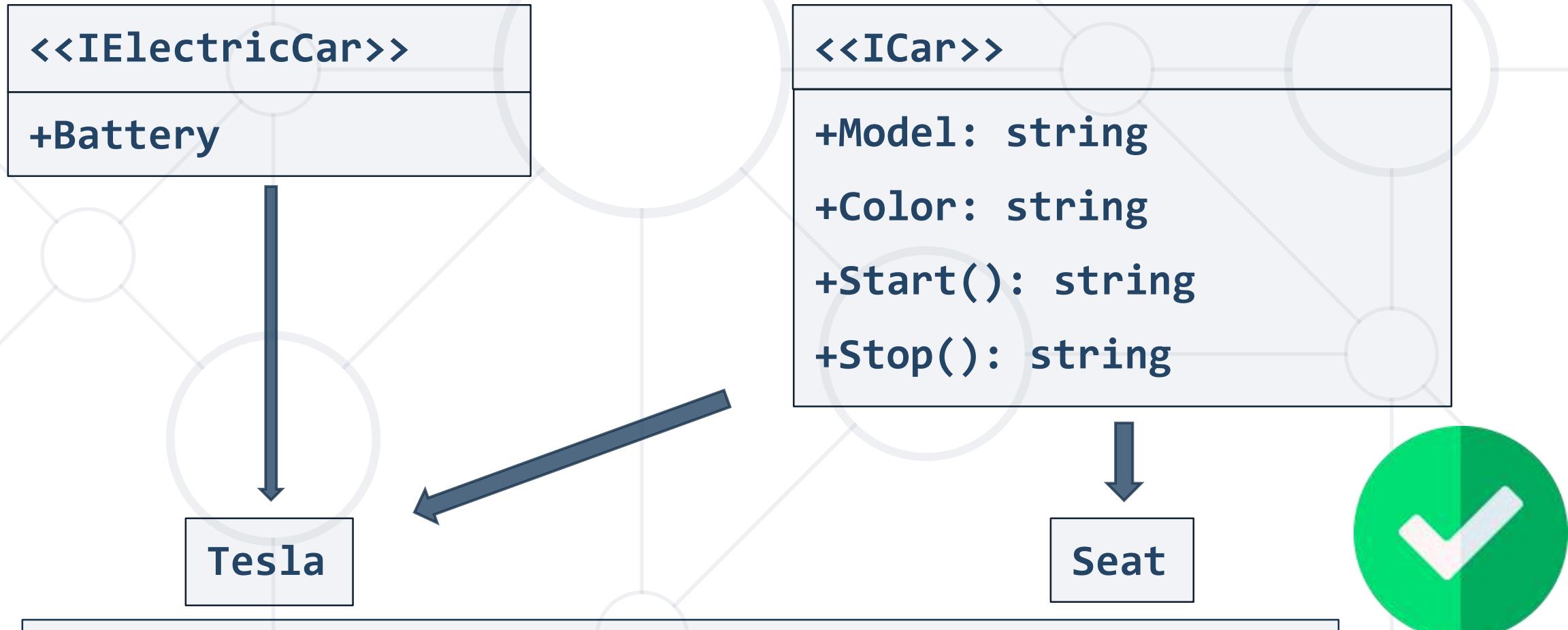
# Interface vs Abstract Class (2)

- Interface
  - Fields and constants **can't be defined**
  - If we add a **new method** **we have to track down all the implementations** of the interface and **define implementation** for the new method
- Abstract Class
  - Fields and constants **can be defined**
  - If we add a **new method** **we** have the option of **providing default implementation** and therefore all the existing code might work properly



# Problem: Cars

- Build a hierarchy of interfaces and classes



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1501#1>

# Problem: Cars

- Build a hierarchy of interfaces and classes
  - Create an interface called **IElectricCar**
    - It should have a property **Battery**
  - Create an interface called **ICar**
    - It should have properties: **Model: String, Color: String**
    - It should also have methods: **Start(): String, Stop(): String**
  - Create class **Tesla**, which implements **IElectricalCar** and **ICar**
  - Create class **Seat**, which implements **ICar**

# Solution: Cars (1)

```
public interface ICar {  
    string Model { get; }  
    string Color { get; }  
    string Start();  
    string Stop();  
}  
  
public interface IElectricCar {  
    int Batteries { get; }  
}
```

# Solution: Cars (2)

```
public class Tesla : ICar, IElectricCar {  
    public string Model { get; private set; }  
    public string Color { get; private set; }  
    public int Batteries { get; private set; }  
    public Tesla (string model, string color, int batteries)  
    { // TODO: Add Logic here }  
    public string Start()  
    { // TODO: Add Logic here }  
    public string Stop()  
    { // TODO: Add Logic here }  
}
```

# Solution: Cars (3)

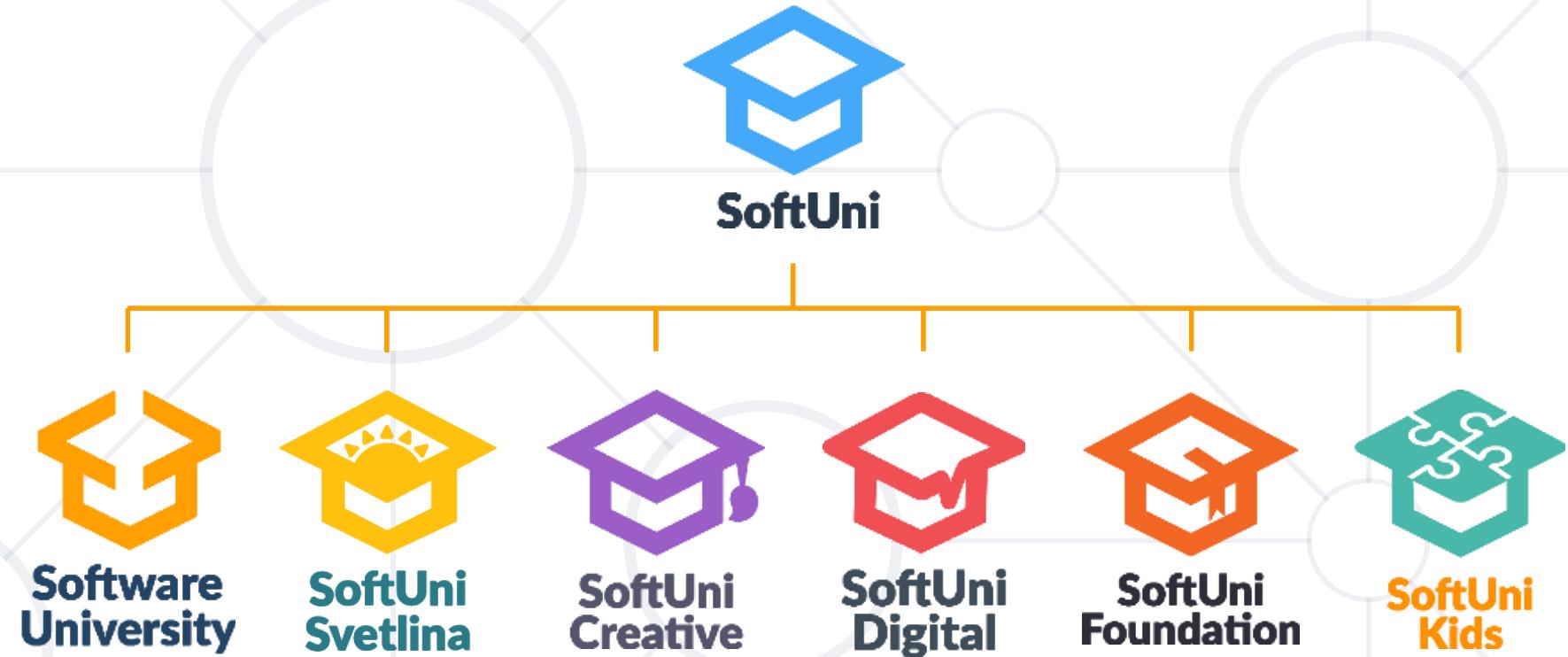
```
public class Seat : ICar {  
    public string Model { get; private set; }  
    public string Color { get; private set; }  
    public Tesla(string model, string color)  
    { // TODO: Add Logic here }  
    public string Start()  
    { // TODO: Add Logic here }  
    public string Stop()  
    { // TODO: Add Logic here }  
}
```

# Summary

- Abstraction
- How do we achieve abstraction
- Interfaces
- Abstract classes



# Questions?



# SoftUni Diamond Partners



SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Polymorphism

## Polymorphism, Override and Overload Methods



**SoftUni Team**  
**Technical Trainers**



**Software University**  
<https://softuni.bg>

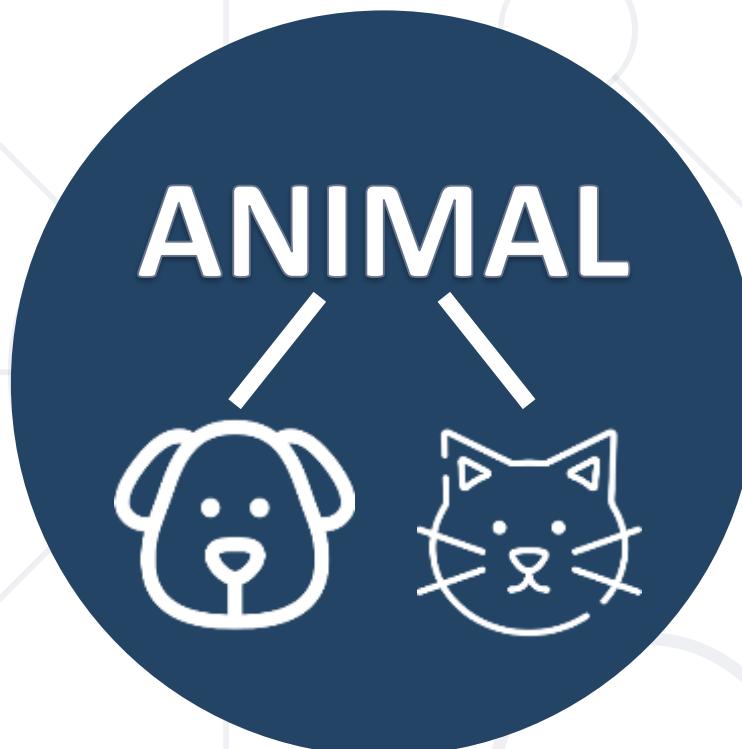
# Table of Contents

- Polymorphism
- The **is** Keyword
- The **as** Keyword
- Compile-time Polymorphism
  - Overload Methods
- Runtime Polymorphism
  - Override Methods



sli.do

# #csharp-advanced



**ANIMAL**



**Polymorphism**

# What is Polimorphism?

- From the Greek

Polys  
(many)

Morphe  
(shape/forms)

Polymorphos

- This is something similar to a word having several different meanings depending on the context
- Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance



# Polymorphism in OOP

- Ability of an **object** to take on **many forms**

```
public interface IAnimal {}  
public abstract class Mammal {}  
public class Person : Mammal, IAnimal {}
```

Person **IS-A** Person

Person **IS-AN** Object

Person **IS-AN** Animal

Person **IS-A** Mammal



# Variable Type and Data Type

- **Variables Type** is the compile-time type of the variable
- **Data Type** is the actual runtime type of the variable
- If you need an **object method** you need to **cast it or override it**

```
public class Person : Mammal, IAnimal {}  
object objPerson = new Person();  
IAnimal person = new Person();  
Mammal mammal = new Person();  
Person person = new Person();
```

Variable Type

Data Type

# Keyword – is

- Runtime check if an **object** is an **instance** of a specific **class**

```
public class Person : Mammal, IAnimal {}  
  
IAnimal person = new Person();  
  
Mammal personOne = new Person();  
  
Person personTwo = new Person();  
  
if (person is Person) {  
    ((Person)person).getSalary();  
}
```

Check object type of person

Cast to object  
type and use its  
methods

# is Type Pattern

- Type pattern - tests whether an expression can be converted to a specified type and casts it to a variable of that type

```
public class Person : Mammal, IAnimal {}  
Mammal personOne = new Person();  
Person personTwo = new Person();  
if (personOne is Person person)  
{  
    person.GetSalary();  
}
```

Checks if object is of type  
person and casts it

Uses its  
methods

# is Constant Pattern

- When performing pattern matching with the constant pattern, **is** tests whether an expression equals a specified constant
- Checking for **null** can be performed using the constant pattern

```
int i = 0;
int min = 0, max = 10;
while(true)
{
    Console.WriteLine($"i is {i}");
    i++;
    if(i is max or min) break;
}
```

# is var Pattern

- A pattern match with the **var pattern** always succeeds

```
Enumerable.Range(0, 100).Where(  
    x => x % 10 is var r && r >= 1 && r <= 3)
```

- The value of **expr** is always assigned to a local variable named **varname**
- **varname** is a variable of the same type as **expr**
- Note that if **expr** is null, the **is** expression still is true and assigns null to **varname**

# Keyword – is

Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else", **slap yourself.**

From *Effective C++*, by Scott Meyers

# Keyword – as

- You can use the **as** operator to perform certain types of conversions between compatible reference types

```
public class Person : Mammal, IAnimal {}  
  
IAnimal person = new Person();  
Mammal personOne = new Person();  
Person personTwo;  
personTwo = personOne as Person;  
if (personTwo != null)  
{  
    // Do something specific for Person  
}
```

Convert Mammal to Person

Check if conversion is  
successful

# Types of Polymorphism

- Runtime

```
public class Shape {}  
public class Circle : Shape {}  
public static void Main()  
{  
    Shape shape = new Circle();  
    shape.Draw();  
}
```

- Compile-time

```
public static void Main()  
{  
    int Sum(int a, int b, int c)  
    double Sum(double a, double b)  
}
```



# Compile-time Polymorphism

- Also known as **Static Polymorphism**

```
public static void Main()
{
    static int MyMethod(int a, int b) {}
    static double MyMethod(double a, double b) { ... }
}
```

Method  
overloading

- Argument lists could differ in:

- Number of parameters
- Data type of parameters
- Order of parameters

# Problem: MathOperation

## MathOperation

```
+Add(int, int): int  
+Add(double, double, double): double  
+Add(decimal, decimal, decimal): decimal
```



```
MathOperations mo = new MathOperations();  
Console.WriteLine(mo.Add(2, 3));  
Console.WriteLine(mo.Add(2.2, 3.3, 5.5));  
Console.WriteLine(mo.Add(2.2m, 3.3m, 4.4m));
```

# Solution: MathOperation

```
public int Add(int a, int b)
{
    return a + b;
}
public double Add(double a, double b, double c)
{
    return a + b + c;
}
public decimal Add(decimal a, decimal b, decimal c)
{
    return a + b + c;
}
```

# Rules for Overloading a Method

- Name should be the same
- **Signature** must be different
  - **Number** of arguments
  - **Type** of arguments
  - **Order** of arguments
- Return type is not a part of its signature
- Overloading can take place in the **same class** or in its **sub-classes**
- Constructors can be **overloaded**

# Runtime Polymorphism (1)

- Has two distinct aspects:
- At run time, objects of a **derived class** may be treated as objects of **a base class** in places, such as method parameters and collections or arrays
  - When this occurs, the **object's declared type** is no longer identical to **its run-time type**

# Runtime Polymorphism(2)

- Base classes may define and implement **virtual methods**
  - Derived classes can **override**
  - They provide **their own definition and implementation**
- At run-time, the CLR looks up the run-time type of the object and invokes that override of the virtual method

# Runtime Polymorphism (1)

- Also known as **Dynamic Polymorphism**

```
public class Rectangle {  
    public virtual double Area() {  
        return this.a * this.b;  
    }  
}  
  
public class Square : Rectangle {  
    public override double Area() {  
        return this.a * this.a;  
    }  
}
```

Method  
overriding

# Runtime Polymorphism (2)

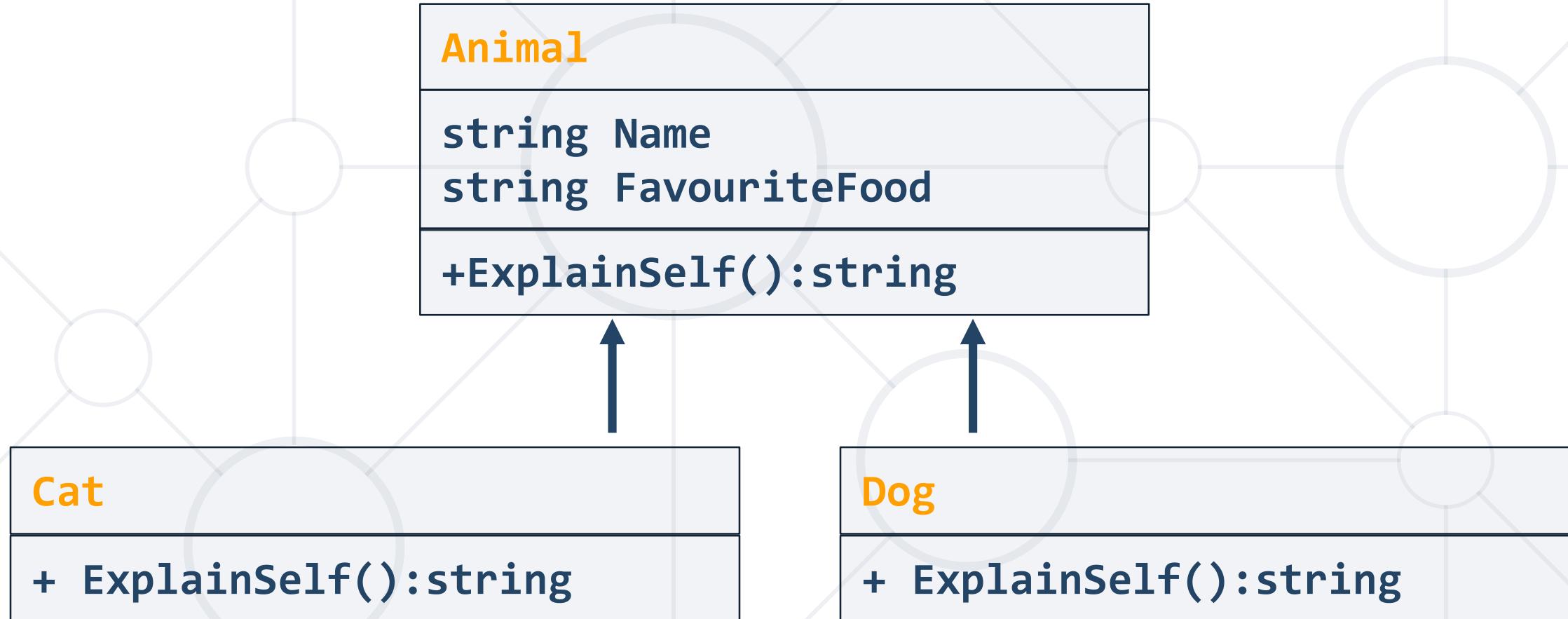
- Usage of **override** method

```
public static void Main()
{
    Rectangle rect = new Rectangle(3.0, 4.0);
    Rectangle square = new Square(4.0);

    Console.WriteLine(rect.Area()); // 12.0
    Console.WriteLine(square.Area()); // 16.0
}
```

Method  
overriding

# Problem: Animals



Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1503#1>

# Solution: Animals (1)

```
public abstract class Animal {  
    // Create Constructor  
    public string Name { get; private set; }  
    public string FavouriteFood { get; private set; }  
    public virtual string ExplainSelf() {  
        return string.Format(  
            "I am {0} and my favourite food is {1}",  
            this.Name,  
            this.FavouriteFood);  
    }  
}
```

# Solution: Animals (2)

```
public class Dog : Animal
{
    public Dog(string name, string favouriteFood)
        : base(name, favouriteFood) { }
    public override string ExplainSelf()
    {
        return base.ExplainSelf() +
            Environment.NewLine +
            "BARK";
    }
}
```

# Solution: Animals (3)

```
public class Cat : Animal
{
    public Cat(string name, string favouriteFood)
        : base(name, favouriteFood) { }
    public override string ExplainSelf()
    {
        return base.ExplainSelf() +
            Environment.NewLine +
            "MEOW";
    }
}
```

# Rules for Overriding Method

- **Overriding** must take place in any sub-classes
- The overriding method and the base must have the **same return type** and the **same signature**
- Base method must have the **virtual** keyword
- Overriding method must have the **abstract** or **override** keyword
- **Private** and **static** methods **cannot** be overridden
- **Virtual** members can use **base keyword** to call the **base class**

# Virtual Members

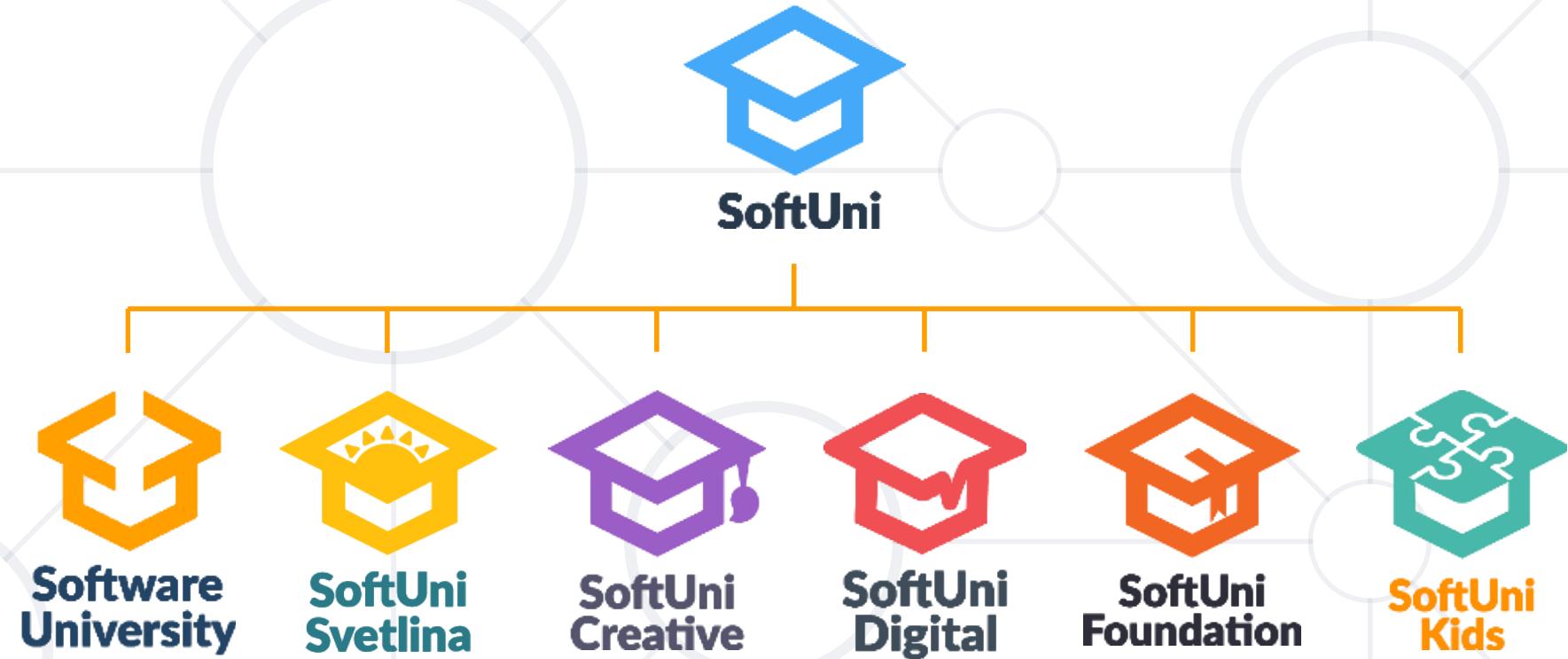
- Virtual members **remain virtual indefinitely**
- A derived class can stop virtual inheritance by declaring an override as **sealed**
  - Sealed methods can be replaced by derived classes by using the **new** keyword
- The **override** modifier extends the base class virtual method
  - The **new** modifier hides an accessible base class method

# Summary

- Polymorphism - **Definition and Types**
- **is** Keyword
- **as** Keyword
- Overload Methods
- Override Methods



# Questions?



# SoftUni Diamond Partners

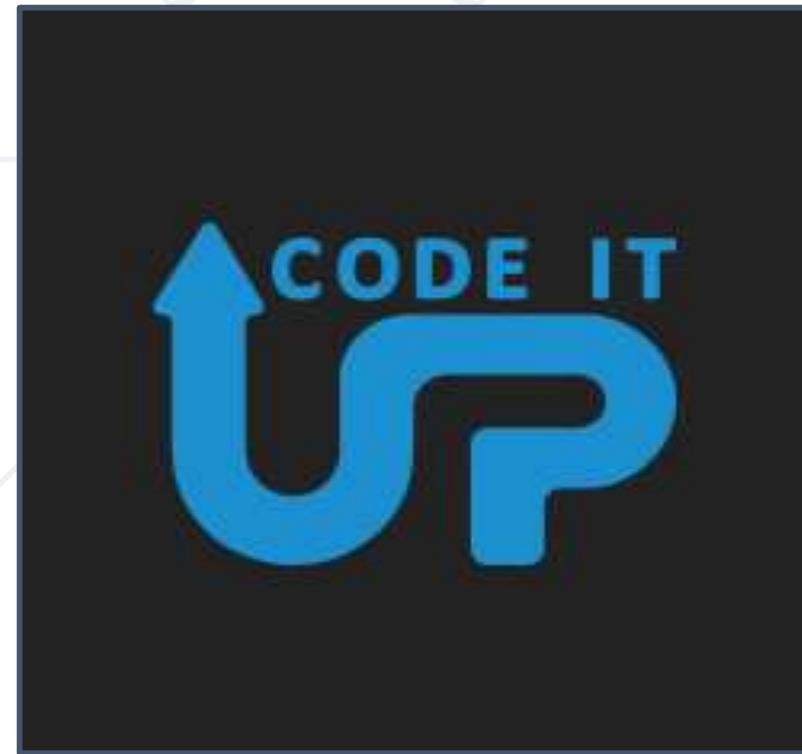


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

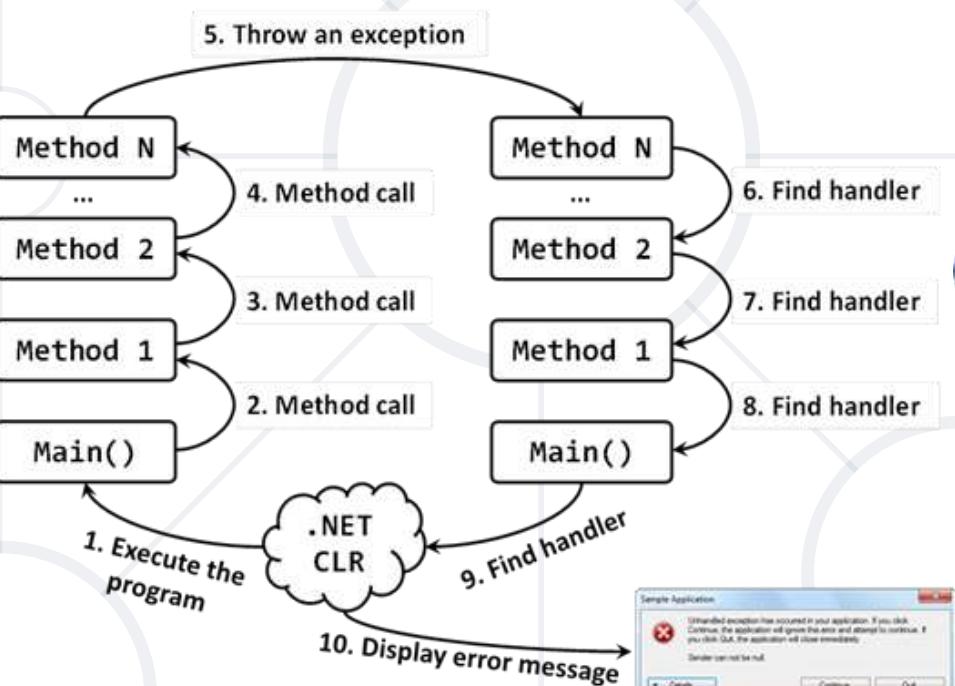


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



# Exception Handling

## Handling Errors During the Program Execution



SoftUni Team  
Technical Trainers



# Table of Contents

## 1. What are Exceptions?

- The **System.Exception** Class
- Types of Exceptions and their Hierarchy

## 2. Handling Exceptions

- **try-catch-finally**

## 3. Raising (Throwing) Exceptions

- **throw new Exception(...)**

## 4. Exceptions: Best Practices



Have a Question?



sli.do

**#csharp-advanced**



# The Paradigm of Exceptions in OOP

## What Are Exceptions?

# What Are Exceptions?

- **Exceptions** handle errors and problems at runtime
- **Throw** an exception to signal about a problem

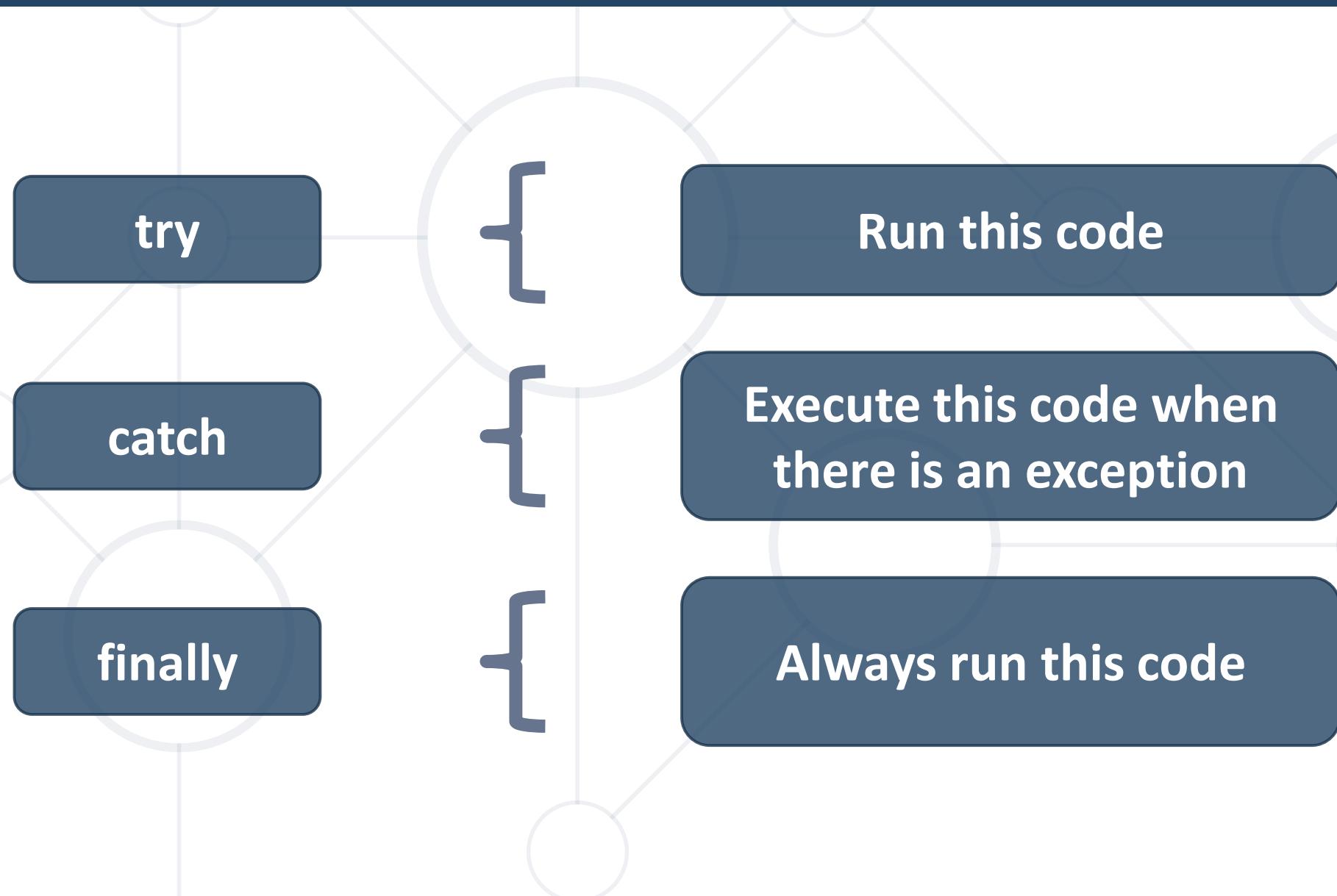
```
if (size < 0)  
    throw new Exception("Size cannot be negative!");
```

- **Catch** an exception to handle the problem

```
try {  
    size = int.Parse(text);  
} catch (Exception ex) {  
    Console.WriteLine("Invalid size!");  
}
```



# How Do Exceptions Work?

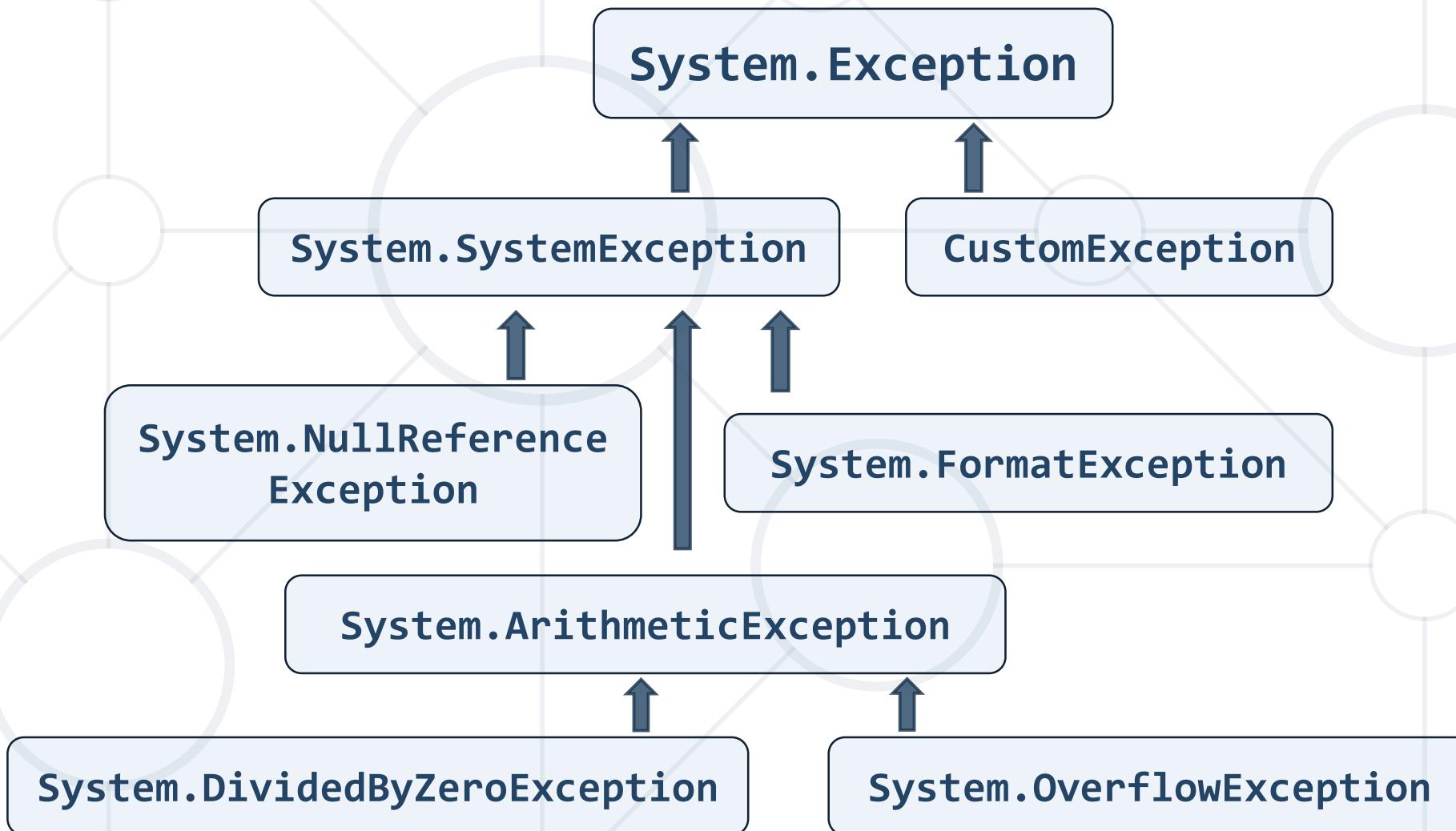


# The System.Exception Class

- Exceptions in C# are **objects**
- The **System.Exception** class is a base for all exceptions in CLR
  - Holds information about the **error**
  - **Message** – a text description of the exception
  - **StackTrace** – the snapshot of the stack at the moment of exception throwing
  - **InnerException** – exception that caused the current exception (if any)



# Exception Hierarchy in .NET





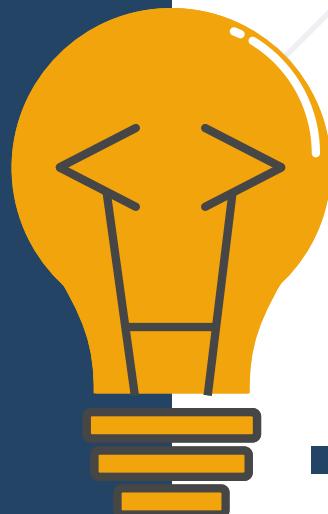
# Catching and Processing Errors

## Handling Exceptions

# Handling Exceptions

- In C# exceptions can be handled by the try-catch construction

```
try {  
    // Do some work that can raise an exception  
}  
catch (SomeException) {  
    // Handle the caught exception  
}
```



- **catch** blocks can be used multiple times to process different exception types

# Multiple Catch Blocks – Example

```
string s = Console.ReadLine();
try {
    int.Parse(s);
    Console.WriteLine(
        "You entered a valid Int32 number {0}.", s);
}
catch (FormatException) {
    Console.WriteLine("Invalid integer number!");
}
catch (OverflowException) {
    Console.WriteLine(
        "The number is too big to fit in Int32!");
}
```



# Handling Exceptions

- When catching an exception of a particular class, all its inheritors (child exceptions) are caught too, e.g.

```
try {  
    // Do some work that can cause an exception  
}  
catch (ArithmetricException ae) {  
    // Handle the caught arithmetic exception  
}
```

- Handles **ArithmetricException** and its descendants  
**DivideByZeroException** and **OverflowException**

# Find the Mistake!

```
string str = Console.ReadLine();
try {
    Int32.Parse(str);
}
catch (Exception) {
    Console.WriteLine("Cannot parse the number!");
}
catch (FormatException) {
    Console.WriteLine("Invalid integer number!");
}
catch (OverflowException) {
    Console.WriteLine("The number is too big to fit in Int32!");
}
```

Should be last

Unreachable code

Unreachable code

# Handling All Exceptions

- For **handling all exceptions** (disregarding the exception type, even unmanaged) use the construction:

```
try
{
    // Do some work that can raise any exception
}
catch
{
    // Handle the caught exception
}
```

# The Try-finally Statement

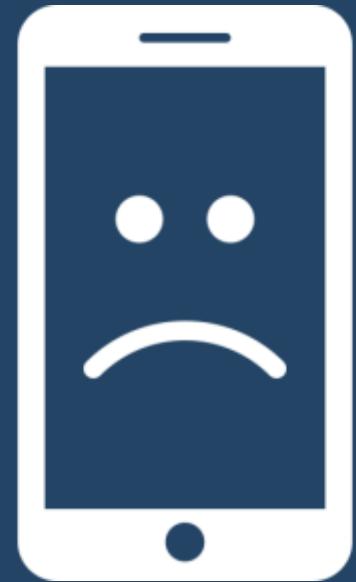
- The **try-finally** statement ensures the **finally** block is **always executed** (with or without exception):

```
try {  
    // Do some work that can cause an exception  
}  
finally {  
    // This block will always execute  
}
```

- Used for execution of **cleanup code**, e. g. releasing resources

# Try-finally – Example

```
static void TestTryFinally() {
    Console.WriteLine("Code executed before try-finally.");
    try {
        string str = Console.ReadLine();
        int.Parse(str);
        Console.WriteLine("Parsing was successful.");
        return; // Exit from the current method
    } catch (FormatException) {
        Console.WriteLine("Parsing failed!");
    } finally {
        Console.WriteLine("This cleanup code is always executed.");
    }
    Console.WriteLine("This code is after the try-finally block.");
}
```



# Throwing Exceptions

## Using the "throw" Keyword

# Using Throw Keyword

- Throwing an exception with an error message:

```
throw new ArgumentException("Invalid amount!");
```

- Exceptions can accept **message + another exception** (cause):

```
try {  
    ...  
}  
catch (SQLException sqlEx) {  
    throw new InvalidOperationException("Cannot save invoice.",  
sqlEx); }
```

- This is called "**chaining**" exceptions

# Throwing Exceptions

- Exceptions are thrown (raised) by the **throw** keyword
- Notify the calling code in case of an error or problem
- When an exception is thrown:
  - The program execution stops
  - The exception travels over the stack
    - Until a matching **catch** block is reached to handle it
- Unhandled exceptions display an error message

# Re-Throwing Exceptions

- Caught exceptions can be **re-thrown** again:

```
try {
    Int32.Parse(str);
}
catch (FormatException fe) {
    Console.WriteLine("Parse failed!");
    throw fe; // Re-throw the caught exception
}
```

```
catch (FormatException) {
    throw; // Re-throws the last caught exception
}
```

# Throwing Exceptions – Example

```
public static double Sqrt(double value) {
    if (value < 0)
        throw new System.ArgumentOutOfRangeException("value",
            "Sqrt for negative numbers is undefined!");
    return Math.Sqrt(value);
}
static void Main() {
    try {
        Sqrt(-1);
    }
    catch (ArgumentOutOfRangeException ex) {
        Console.Error.WriteLine("Error: " + ex.Message);
        throw;
    }
}
```



# Best Practices

## Best Practices for Exception Handling

# Using Catch Block

- **Catch** blocks should:
  - Begin with the exceptions lowest in the hierarchy
  - Continue with the more general exceptions
  - Otherwise, a compilation error will occur
- Each **catch** block should handle only these exceptions which it expects
  - If a method is not competent to handle an exception, it should leave it unhandled
  - Handling all exceptions disregarding their type is a popular **bad practice** (anti-pattern)!

# Choosing the Exception Type

- When an invalid parameter value is passed to a method:
  - **ArgumentException**, **ArgumentNullException**,  
**ArgumentOutOfRangeException**
- When requested operation is not supported
  - **NotSupportedException**
- When a method is still not implemented
  - **NotImplementedException**
- If no suitable standard exception class is available
  - Create own exception class (inherit **Exception**)

# Exceptions – Best Practices (1)

- When raising an exception, always pass to the constructor a **good explanation message**
- When throwing an exception always pass a good description of the problem
  - The exception message should explain what causes the problem and how to solve it
    - Good: "*Size should be integer in range [1...15]*"
    - Good: "*Invalid state. First call Initialize()*"
    - Bad: "*Unexpected error*"
    - Bad: "*Invalid argument*"

# Exceptions – Best Practices (2)

- Exceptions can decrease the application performance
  - Throw exceptions only in situations which are really exceptional and should be handled
  - Do not throw exceptions in the normal program control flow
  - The .NET runtime could throw exceptions at any time with no way to predict them
    - E. g. **System.OutOfMemoryException**

# Creating Custom Exceptions

- Custom exceptions inherit an exception class (e. g. **System.Exception**)

```
public class PrinterException : Exception
{
    public PrinterException(string msg)
        : base(msg) { ... }
}
```

- Thrown just like any other exception

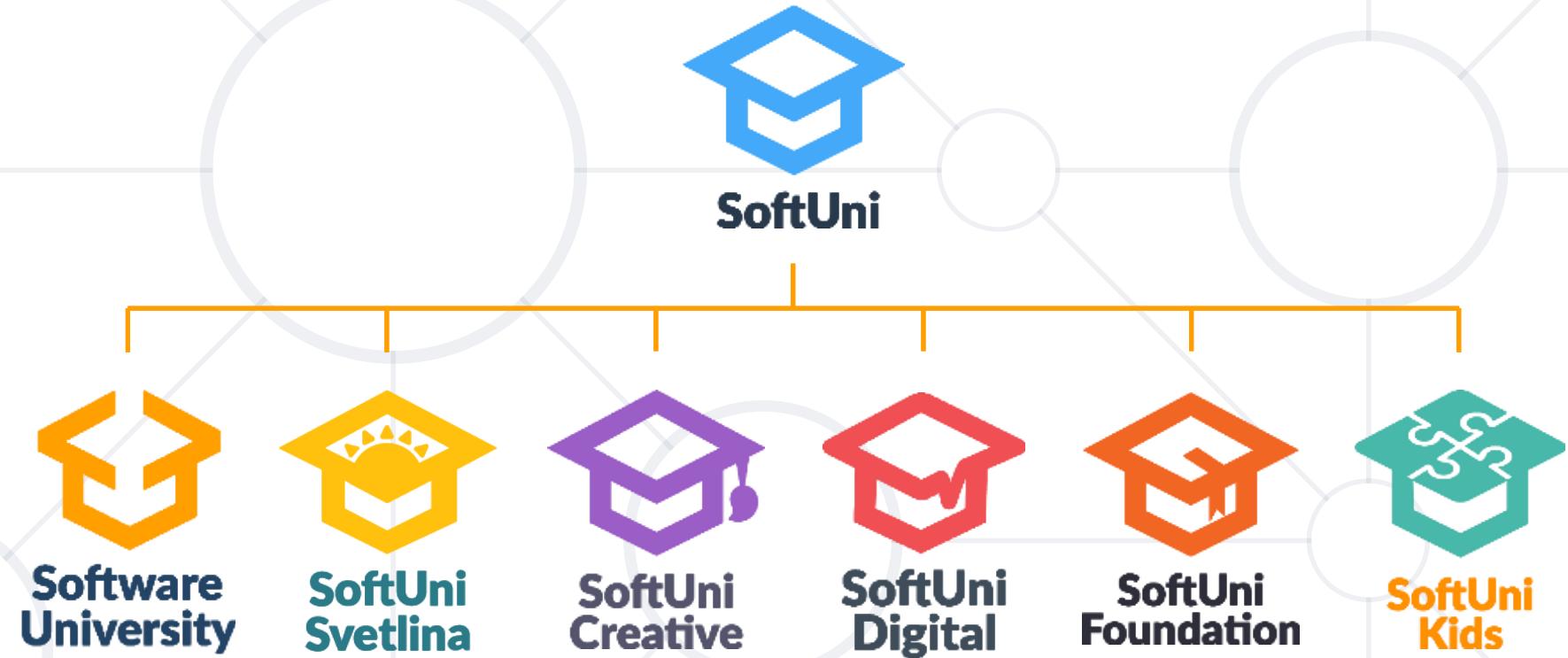
```
throw new PrinterException("Printer is out of paper!");
```

- Exceptions provide a **flexible** error handling mechanism

```
throw new Exception("Invalid size!");
```
- **Try-catch** allows exceptions to be handled

```
try { ... } catch (Exception ex) { ... }
```
- Unhandled exceptions cause error messages
- **Try-finally** ensures a given code block is always executed

# Questions?



# SoftUni Diamond Partners

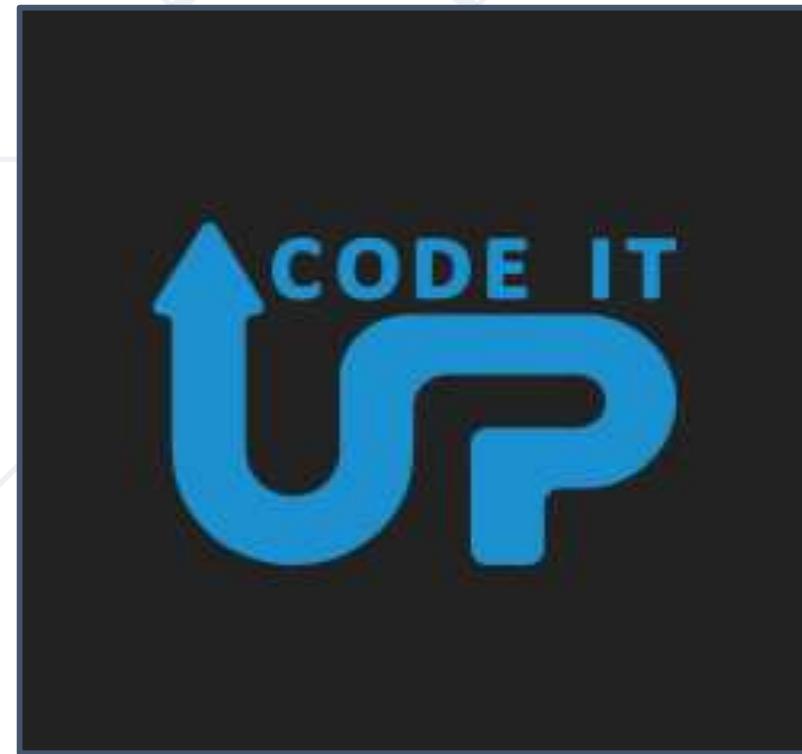


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Debugging

## Building Rock-Solid Software



SoftUni Team  
Technical Trainers



SoftUni



Software University  
<https://softuni.bg>

# Table of Contents

- Introduction to Debugging
- Visual Studio Debugger
- Breakpoints
- Data Inspection
- Threads and Stacks
- Finding a Defect

Have a Question?



sli.do

**#csharp-advanced**



# Introduction to Debugging

# What is Debugging?

- The process of locating and fixing or bypassing **bugs** (errors) in computer program code
- To **debug** a program:
  - Start with a **problem**
  - Isolate the **source** of the problem
  - **Fix** it
- **Debugging tools** (called **debuggers**) help identify coding errors at various development stages

# Debugging vs. Testing

## ■ Testing

- A means of initial detection of errors
- The process of verifying and validating that a software or application is bug free

## ■ Debugging

- A means of diagnosing and correcting the root causes of errors that have already been detected
- The process of identifying, analyzing and fixing a bug in the software



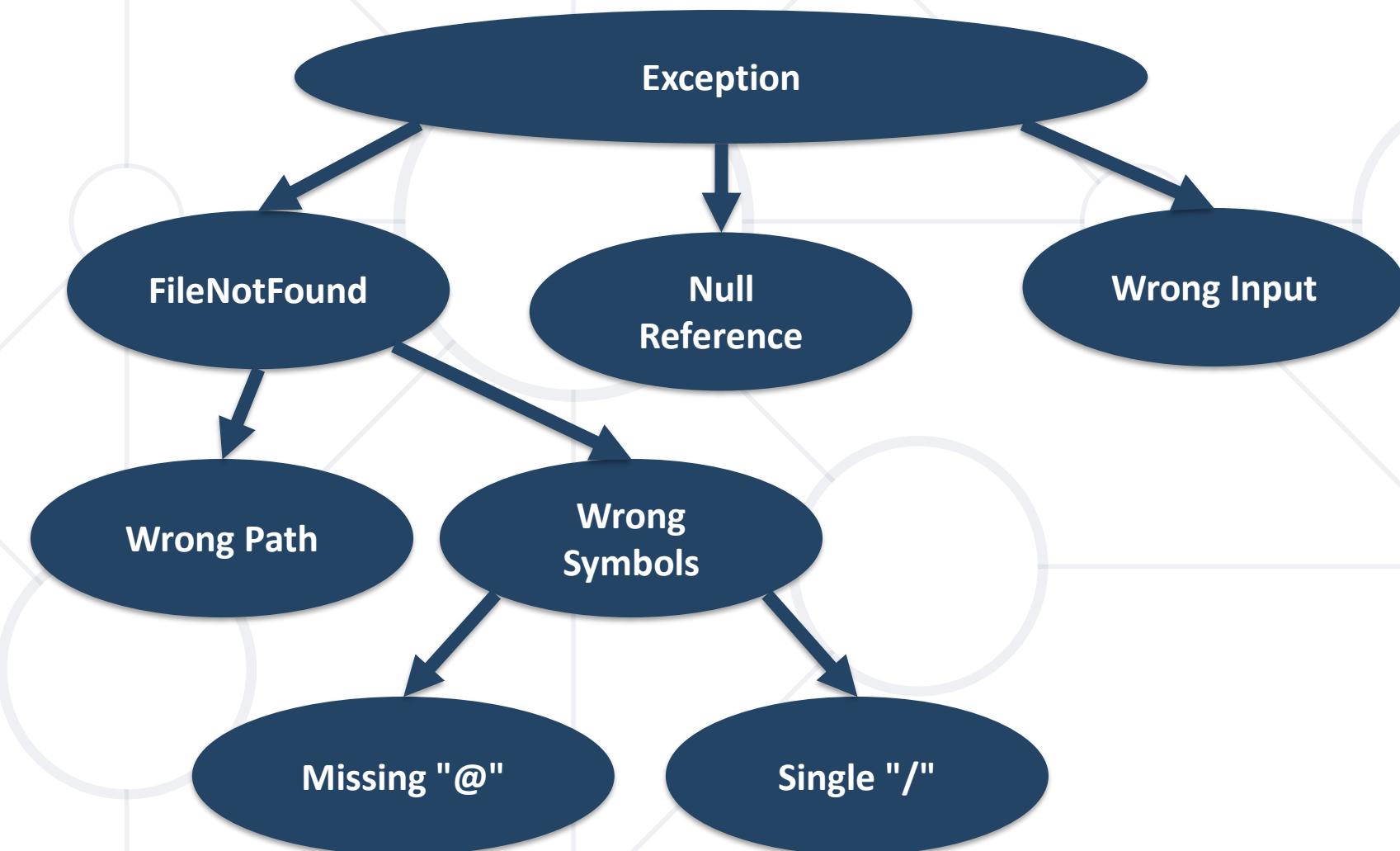
# Importance of Debugging

- \$60 Billion per year in economic **losses** due to software **defects**
  - E.g. the **Cluster spacecraft failure** was caused by a bug
- Perfect code is an **illusion**
  - There are factors that are out of our control
- **Legacy** code
  - You should be able to debug code that is written years ago
- **Deeper understanding** of system as a whole

# Debugging Philosophy

- Debugging can be viewed as one big **decision tree**
  - Individual nodes represent **theories**
  - Leaf nodes represent possible **root causes**
  - Traversal of tree boils down to process state **inspection**
  - Minimizing time to resolution is **key**
    - Careful traversal of the decision tree
    - Pattern recognition
    - Visualization and ease of use helps minimize time to resolution

# Example Debugging – Decision Tree





# Visual Studio Debugger

# Visual Studio Debugger

- Visual Studio IDE gives us a lot of **tools to debug** your application
  - Adding **breakpoints**
  - Visualize the **program flow**
  - Control the **flow of execution**
  - **Data tips**
  - **Watch variables**
  - Debugging **multithreaded programs**
  - And many more...

# Debugging a Solution

- Debug menu, Start Debugging item
  - **F5** is a shortcut
- Easier access to the source code and symbols since its loaded in the solution
- Certain differences exist in comparison to debugging an already running process
  - Hosting for an **ASP.NET application**
    - Visual Studio uses a replacement of the real IIS

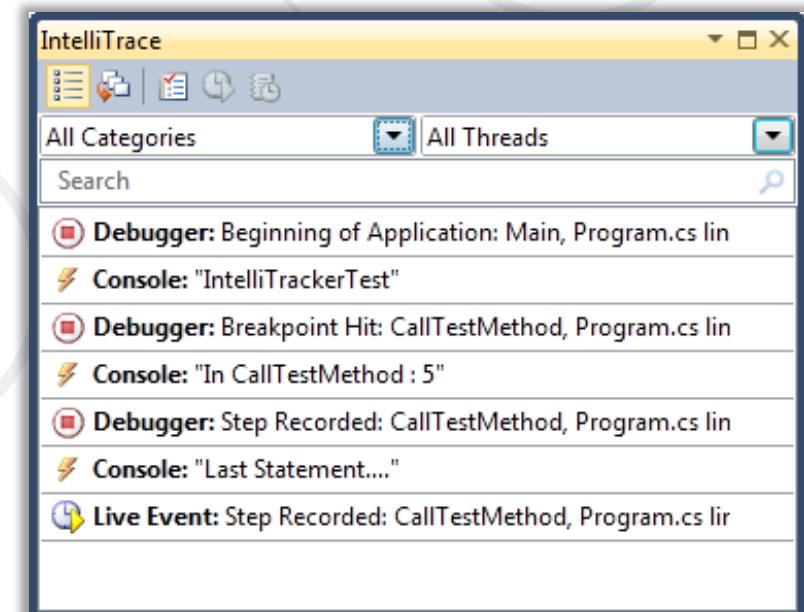
# Debug Windows

- Debug Windows are the means to introspect on the state of a process
- Opens a new window with the selected information in it
- Window categories
  - **Data inspection**
  - **Threading**
- Accessible from menu
  - **Debug -> Windows**

# Debugging Toolbar

- Convenient shortcut to common debugging tasks
  - Step into
  - Step over
  - Continue
  - Break
  - Breakpoints
- Customizable to fit your needs
  - Add / Remove buttons

- IntelliTrace operates in the background, records what you are doing during debugging
- You can easily get a past state of your application from IntelliTrace
- You can navigate through your code and see what's happened
- To navigate, just click any of the events that you want to explore





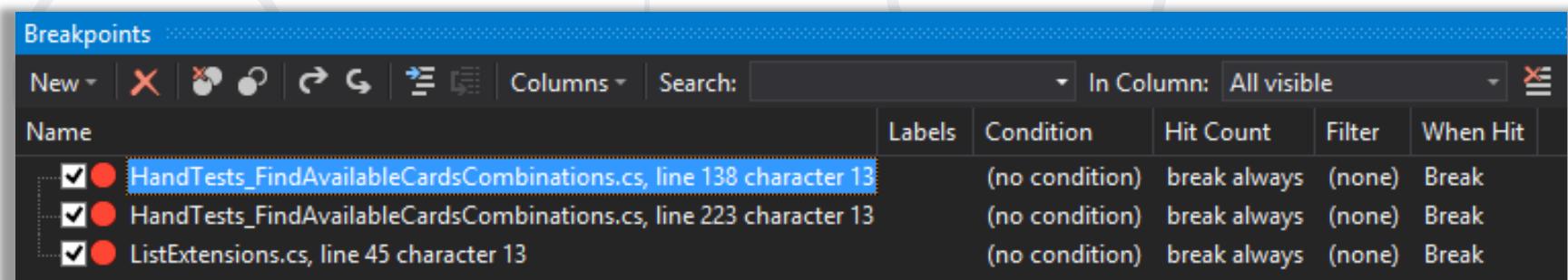
# Breakpoints

# Breakpoints

- Ability to stop execution based on certain criteria is key when debugging
  - When a **function is hit**
  - When **data changes**
  - When a specific **thread** hits a **function**
  - Much more...
- Visual Studio's debugger has a huge feature set when it comes to breakpoints

# Managing Breakpoints

- Managed in the breakpoint window
- Adding breakpoints
- Removing or **disabling** breakpoints
- **Labeling** or **grouping** breakpoints
- Export/import breakpoints

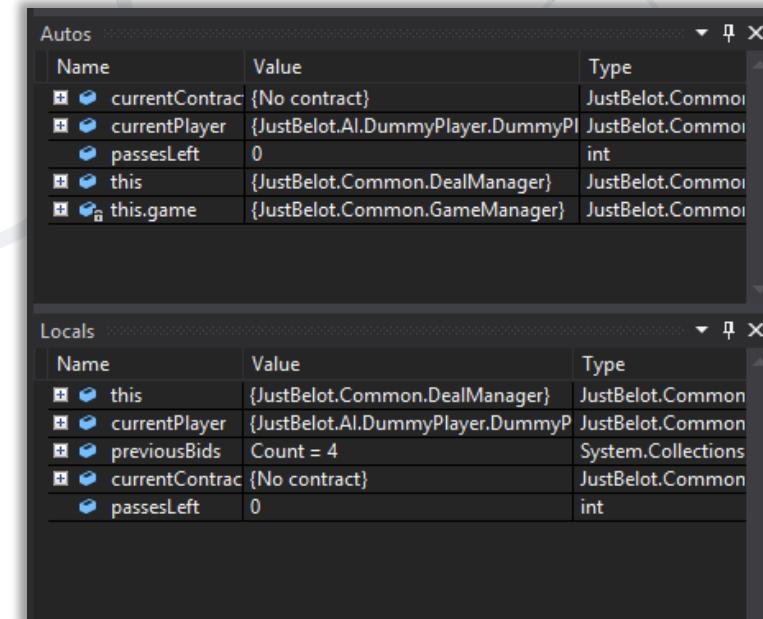




# Data Inspection

# Visual Studio Data Inspection

- Visual Studio offers great data inspection features
  - Watch windows
  - Autos and Locals
  - Memory and Registers
  - Data Tips
  - Immediate window



# Watch Window

- Allows you to inspect various states of your application
- Several different kinds of "**predefined**" watch windows
  - **Autos**
  - **Locals**
- "**Custom**" watch windows also possible
  - Contains only variables that you choose to add
  - Right click on the variable and select "Add to Watch"

# Autos and Locals

- Locals watch window contains the local variables for the specific stack frame
  - **Debug -> Windows -> Locals**
  - Displays: name of the variable, value and type
  - Allows drill down into objects by clicking on the + sign in the tree control
- **Autos** lets the debugger decide which variables to show in the window
  - Loosely based on the current and previous statement

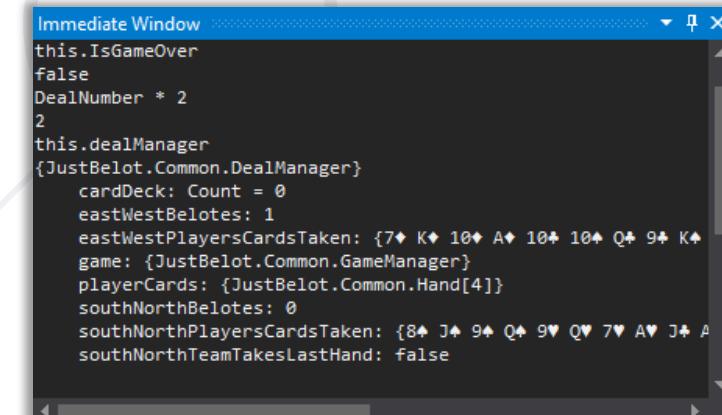
# Memory and Registers

- **Memory** window can be used to inspect process wide memory
  - Address field can be a raw pointer or an expression
  - Drag and drop a variable from the source window
  - Number of columns displayed can be configured
  - Data format can be configured
- **Registers** window can be used to inspect processor registers

- Provides information about variables
  - Variables must be within scope of current execution
- Place mouse pointer over any variable
  - Variables can be expanded by using the + sign
- Pinning the data tip causes it to always stay open
- Comments can be added to data tips
- Data tips support drag and drop
- Importing and exporting data tips

# Immediate Window

- Useful when debugging due to the **expansive expressions** that can be **executed**
  - To output the value of a variable {**name of variable**}
  - To set values, use {**name of variable**}={**value**}
  - To call a method, use {**name of variable**}.  
**<method>(arguments)**
- Similar to regular code
- Supports **IntelliSense**



```
Immediate Window
this.IsGameOver
false
DealNumber * 2
2
this.dealManager
{JustBelot.Common.DealManager}
    cardDeck: Count = 0
    eastWestBelotes: 1
    eastWestPlayersCardsTaken: {7♦ K♦ 10♦ A♦ 10♣ 10♦ Q♦ 9♦ K♦}
    game: {JustBelot.Common.GameManager}
    playerCards: {JustBelot.Common.Hand[4]}
    southNorthBelotes: 0
    southNorthPlayersCardsTaken: {8♦ J♦ 9♦ Q♦ 9♥ Q♥ 7♥ A♥ J♦ A}
    southNorthTeamTakesLastHand: false
```



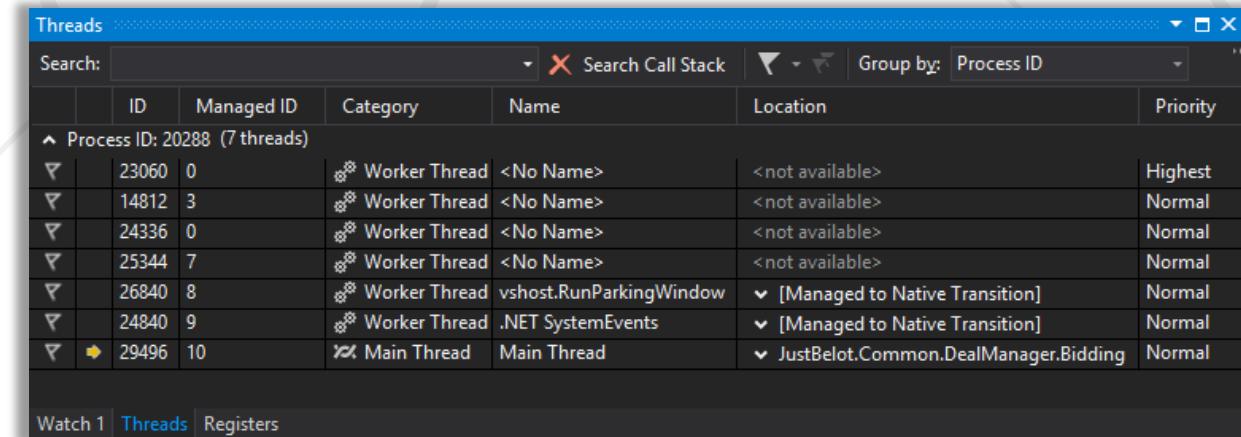
# Threads and Stacks

# Threads

- Fundamental units of code execution
- Commonly, programs use **more** than **one thread**
  - In .NET, there is always more than one thread
- Each thread has a memory area associated with it known as a **stack**
  - Stores **local variables**
  - Stores frame **specific information**
- Memory area employs last in first out semantics

# Threads Window

- Contains an overview of thread activity in the process
- Includes basic information in a per thread basis
  - Thread ID's
  - Category
  - Name
  - Location
  - Priority



	ID	Managed ID	Category	Name	Location	Priority
▲ Process ID: 20288 (7 threads)						
▼	23060	0	Worker Thread	<No Name>	<not available>	Highest
▼	14812	3	Worker Thread	<No Name>	<not available>	Normal
▼	24336	0	Worker Thread	<No Name>	<not available>	Normal
▼	25344	7	Worker Thread	<No Name>	<not available>	Normal
▼	26840	8	Worker Thread	vshost.RunParkingWindow	▼ [Managed to Native Transition]	Normal
▼	24840	9	Worker Thread	.NET SystemEvents	▼ [Managed to Native Transition]	Normal
▼	29496	10	Main Thread	Main Thread	▼ JustBelot.Common.DealManager.Bidding	Normal

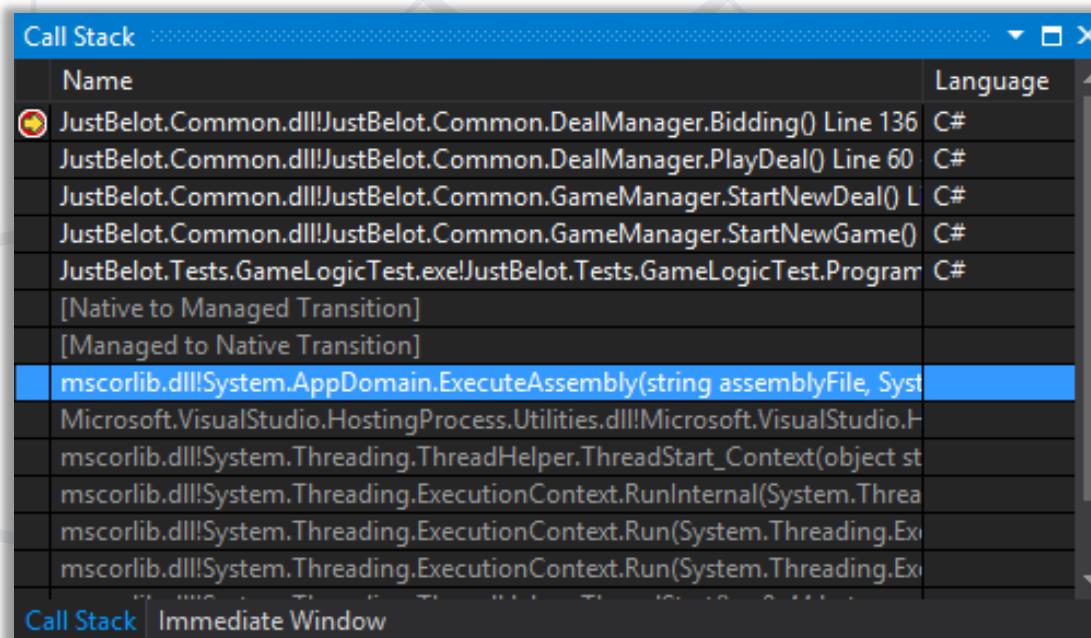
# Breakpoint Filters

- Allows you to exert even more control of when a breakpoint hits
- Examples of customization
  - Machine **name**
  - Process **ID**
  - Process **name**
  - Thread **ID**
  - Thread **name**
- Multiple can be combined using &, ||, !



# Call Stacks

- Visual Studio shows the elements of a call stack
  - Local variables
  - Method frames





# Finding a Defect

# Tips for Finding Defects (1)

- 
- Use all available data
  - Refine the test cases
  - Check unit tests
  - Use available tools
  - Reproduce the error in several different ways
  - Generate more data to generate more hypotheses
  - Use the results of negative tests
  - Brainstorm for possible hypotheses

# Tips for Finding Defects (2)

- 
- Narrow the suspicious region of the code
  - Be suspicious of classes and routines that have had defects before
  - Check code that's changed recently
  - Expand the suspicious region of the code
  - Integrate incrementally
  - Check for common defects
  - Talk to someone else about the problem
  - Take a break from the problem

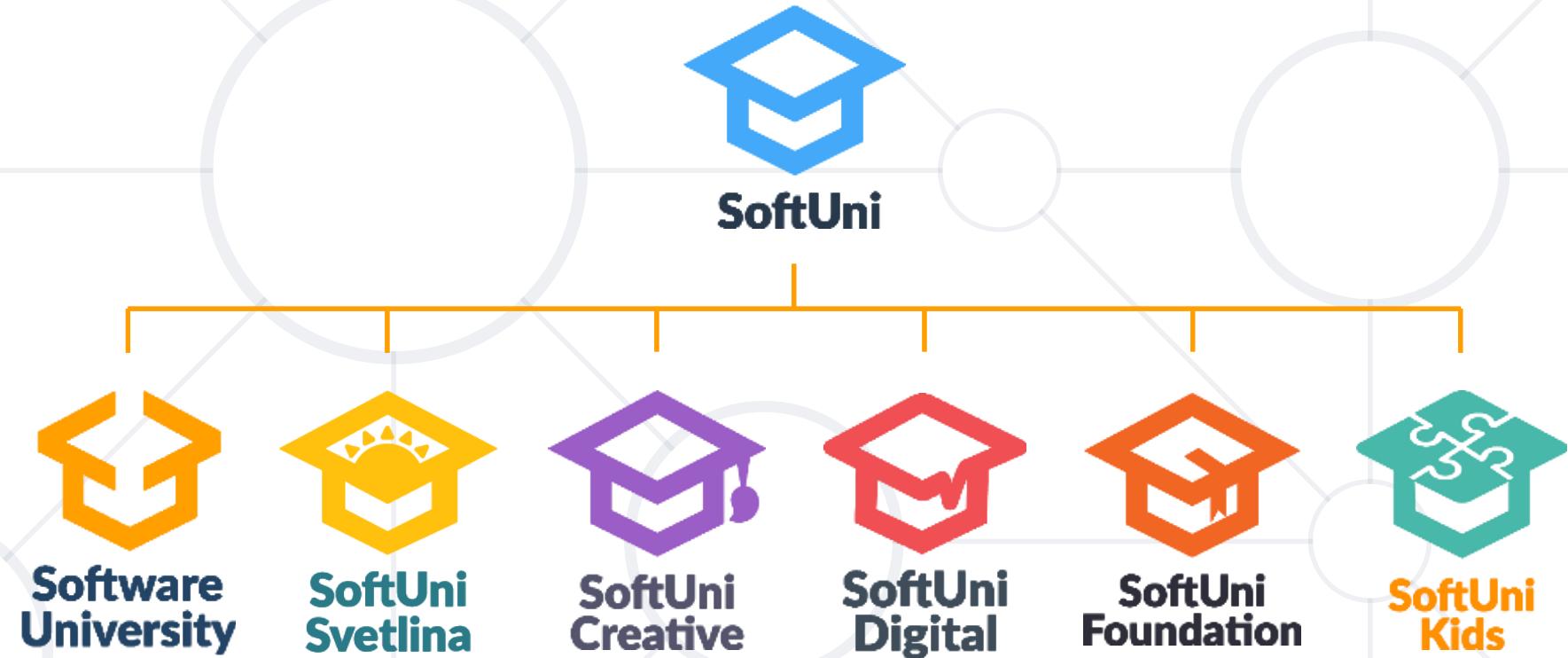
# Fixing a Defect

- 
- Understand the problem before you fix it
  - Understand the program, not just the problem
  - Confirm the defect diagnosis
  - Relax
  - Save the original source code
  - Fix the problem, not the symptom
  - Make one change at a time
  - Add a unit test that expose the defect
  - Look for similar defects

- Introduction to **Debugging**
- Visual Studio Debugger
- **Breakpoints**
- Data Inspection
  - **Locals, Autos, Watch**
- Finding a **Defect**



# Questions?



# SoftUni Diamond Partners



SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# SOLID Principles

## Design Principles and Approaches

- S Single Responsibility
- O Open/Closed
- L Liskov substitution
- I Interface Segregation
- D Dependency Inversion

SoftUni Team

Technical Trainers



Software University  
<https://softuni.bg>

# Table of Contents

1. Single Responsibility
2. Open/Closed
3. Liskov Substitution
4. Interface Segregation
5. Dependency Inversion



sli.do

# #csharp-advanced

# Why Clean Code Matters?

- How **clean code** (or its absence) **affects** our software?

"...So if you want to go **fast**,  
if you want to get done **quickly**,  
if you want your code to be **easy to write**,  
**make it easy to read.**"

- Robert C. Martin



**Single Responsibility**

# What is Single Responsibility?

- Every class should be responsible **for only a single part of the functionality** and that responsibility should be entirely **encapsulated** by the class.

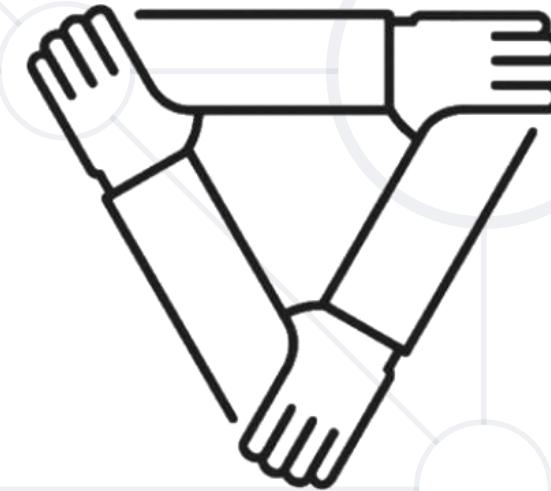
"There should never be more than one reason for a class to change."

- Robert C. "Uncle Bob" Martin



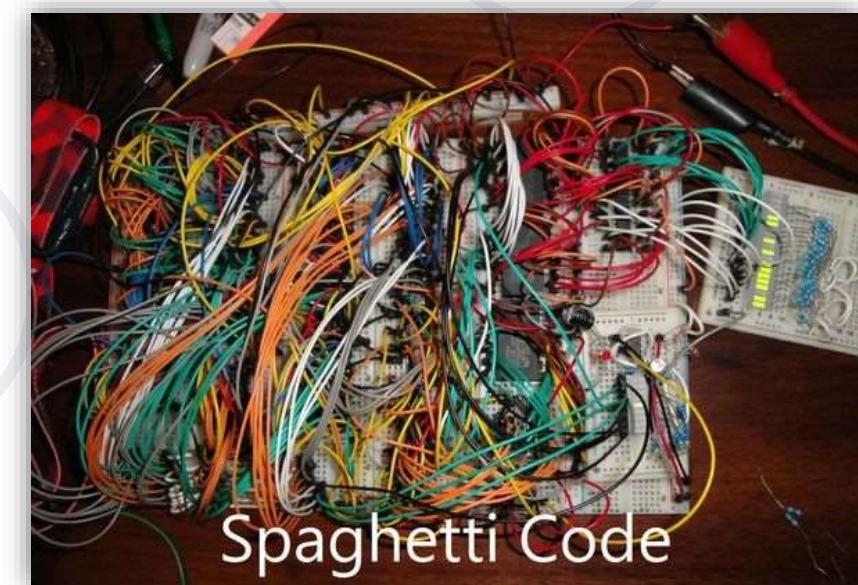
# Strong Cohesion

- **Cohesion** refers to the grouping of **functionally related processes** into a particular module.
- Aim for **strong cohesion**
  - Each **task** maps a **single** code unit
  - A method should do **one operation**
  - A class should represent **one entity**



# Loose Coupling

- **Coupling** - the degree of dependence between modules
  - How closely connected two modules are
  - The strength of the relationship between modules
- Aim for **loose** coupling
  - Supports **readability** and **maintainability**
  - Often a sign of good system **design**

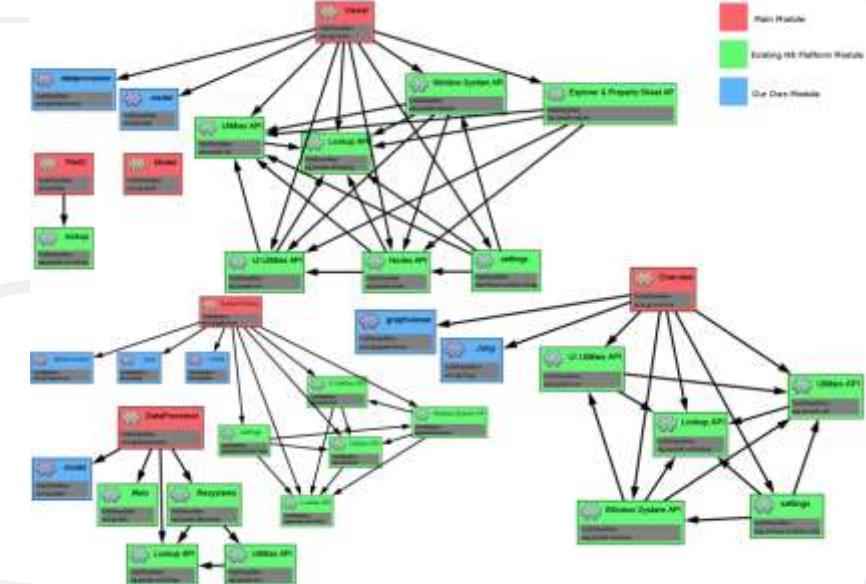


# Dependencies and Coupling

- Depend directly on other modules

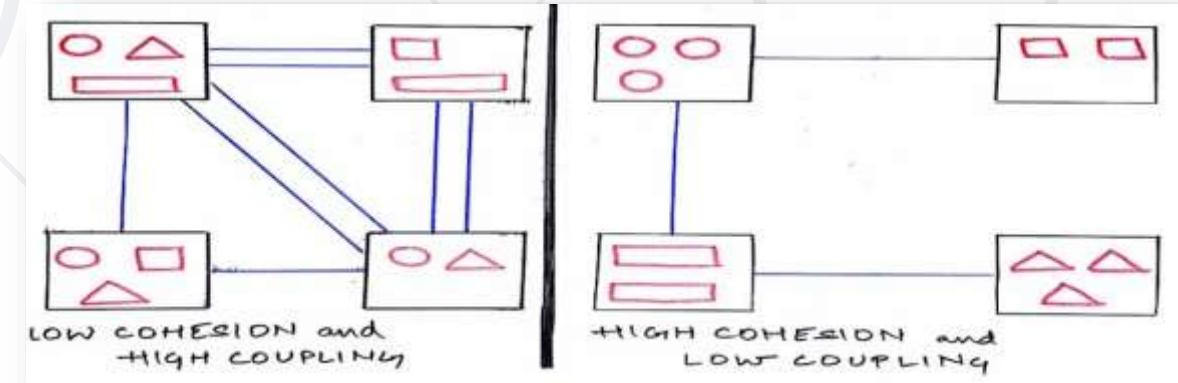


- Depend on abstractions



# Cohesion and Coupling – Approaches

- Small number of instance variables inside a class
- Each method of a class should manipulate one or more of those variables
- Two modules should exchange as little information as possible
- Use abstraction
- Creating an easily reusable subsystem

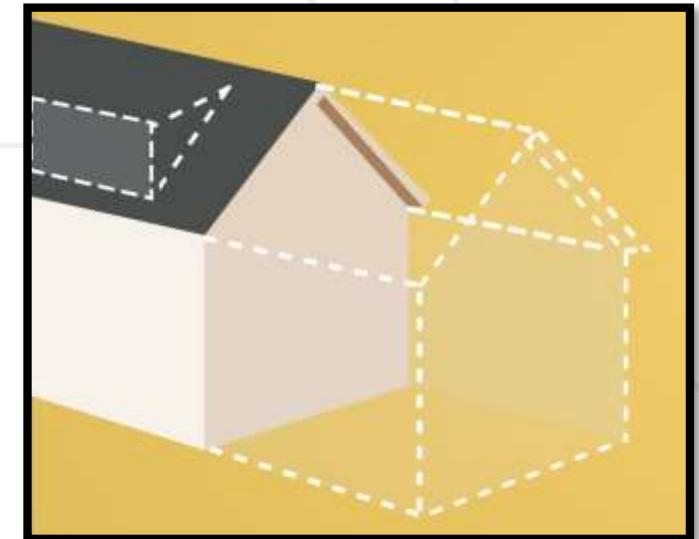




**Open/Closed**

# What is the Open/Closed Principle?

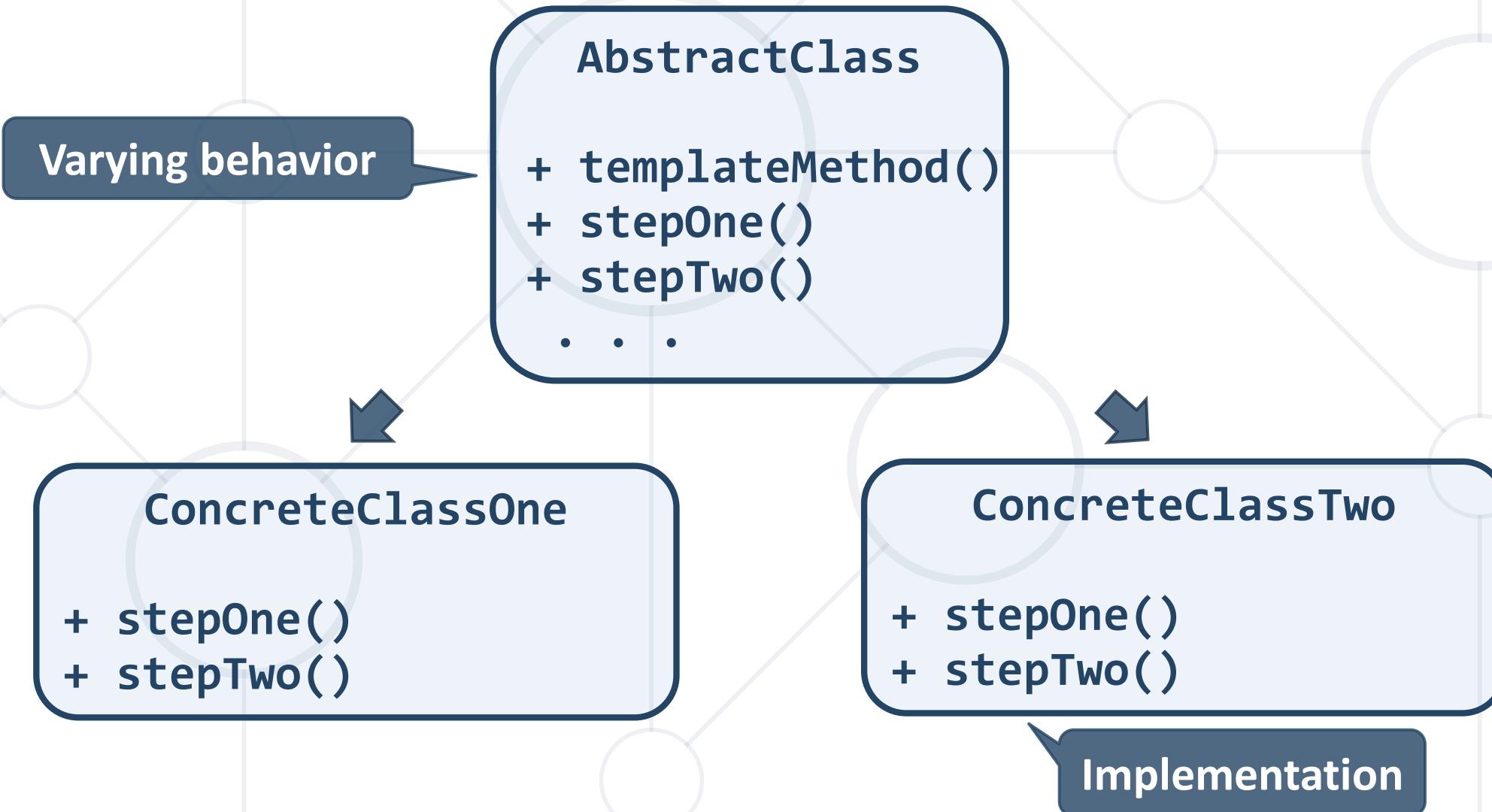
- Software entities like **classes, modules** and **functions** should be **open for extension**, but **closed for modifications**
- **Extensibility**
  - Adding a new behavior **doesn't require** changes over existing source code
- **Reusability**
  - subsystems are **suitable for reusing** in other projects - modularity



- Parameters
  - Control behavior specifics via a **parameter** or a **delegate**
- Rely on abstraction, **not implementation**
  - Inheritance / Template Method Pattern
- Strategy Pattern
  - Plug in model (insert a new implementation of the interface)

- By experience - know the problem domain and if a **change** is very **likely to recur**
- New domain problem - implement the **most simple way**
  - Changes once - **modify**, second time - **refactor**
- TANSTAAFL - There Ain't No Such Thing As A Free Lunch
  - OCP adds **complexity** to design
  - No design can be **closed against all changes**
- Need to **retest (recheck functionality)** after changes

# Template Method Pattern (1)



# Template Method Pattern (2)

```
public abstract class CrossCompiler
{
    public void CrossCompile()
    {
        // some functionality...
        this.CollectSource();
        // some functionality...
        this.CompileToTarget();
        // some functionality...
    }

    protected abstract void CollectSource();
    protected abstract void CompileToTarget();
}
```

Template method

# Template Method Pattern (3)

```
public class IPhoneCompiler : CrossCompiler
{
    protected override void CollectSource()
    protected override void CompileToTarget()
    { // iPhone specific compilation }
}
```

```
public class AndroidCompiler : CrossCompiler
{
    protected override void CollectSource()
    protected override void CompileToTarget()
    { // Android specific compilation }
}
```



# Liskov Substitution

# LSP – Substitutability

- Derived types must be completely **substitutable** for their base types
- Derived classes
  - only **extend** functionalities of the base class
  - must **not** remove **base** class **behavior**

Student **IS-SUBSTITUTED-FOR** Person



# Design Smell – Violations

- Type Checking
- Overridden methods say "*I am not implemented*"
- Base class depends on its subtypes



VIOLATION

- Tell Don't Ask
  - If you need to check what is the object - move the behavior **inside the object**
- New Base Class - if **two classes** share a common behavior, but are not substitutable, create a third, from which **both derive**
- There **shouldn't** be any **virtual methods** in constructors



# Interface Segregation

# What is Interface Segregation?

- Segregate interfaces
  - Prefer **small, cohesive** (lean and focused) interfaces
  - Divide "**fat**" interfaces into "**role**" interfaces



**"Clients** should not be forced to depend on methods they do not use."

- Agile Principles, Patterns and Practices in C#

# Fat Interfaces

- Classes whose interfaces are not cohesive have "fat" interfaces

```
public interface IWorker
{
    void Work();
    void Sleep();
}
```

```
public class Robot : IWorker
{
    void Work() { ... }
    void Sleep()
    { throw new NotImplementedException(); }
}
```

# Design Smells – Violations

- Not implemented methods
- A Client references a class, but only uses a small portion of it

"Abstraction is elimination  
of the irrelevant and  
amplification of the essential."  
- Robert C. Martin

- What does the client **see** and **use**?
- The "**fat**" interfaces implement a **number of small** interfaces with just what you need
- All **public members** of a class divided in **separate classes**
  - again, could be thought of as an interface
- Let the **client define interfaces** - "**role**" interfaces

# Cohesive Interfaces

- Small and Cohesive "Role" Interfaces

```
public interface IWorker
{
    void Work();
}

public interface ISleeper
{
    void Sleep();
}

public class Robot : IWorker
{
    void Work() { // Do some work... }
}
```

# Adapter Pattern

- Problem that the **Adapter pattern** solves
  - **Reusing** classes that do not have an **interface** that a client requires
  - Making classes with **incompatible** interfaces work together
  - Providing **an alternative** interface for a class

# Adapter Pattern (1)

- Convert the **incompatible** interface of a class Adaptee into another interface - Target, that clients require

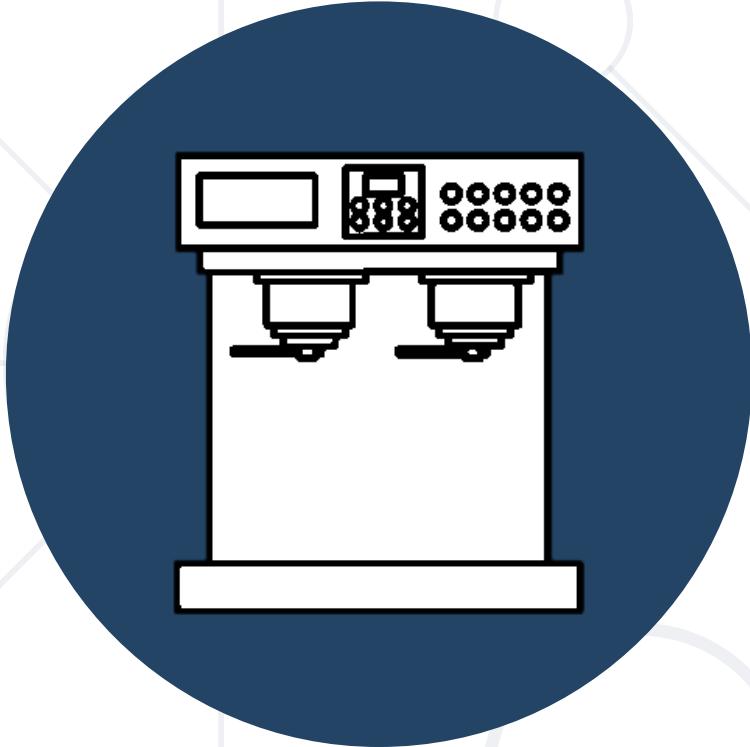
```
class Adaptee
{
    public void SpecificRequest()
    {
        Console.Write
            ("Called SpecificRequest()");
    }
}
```

```
interface ITarget
{
    void Request();
}
```

# Adapter Pattern (2)

- Define a separate class - Adapter, that does the job

```
class Adapter : ITarget
{
    private Adaptee adaptee = new Adaptee();
    public void Request()
    {
        // Possibly do some other work
        adaptee.SpecificRequest();
    }
}
```



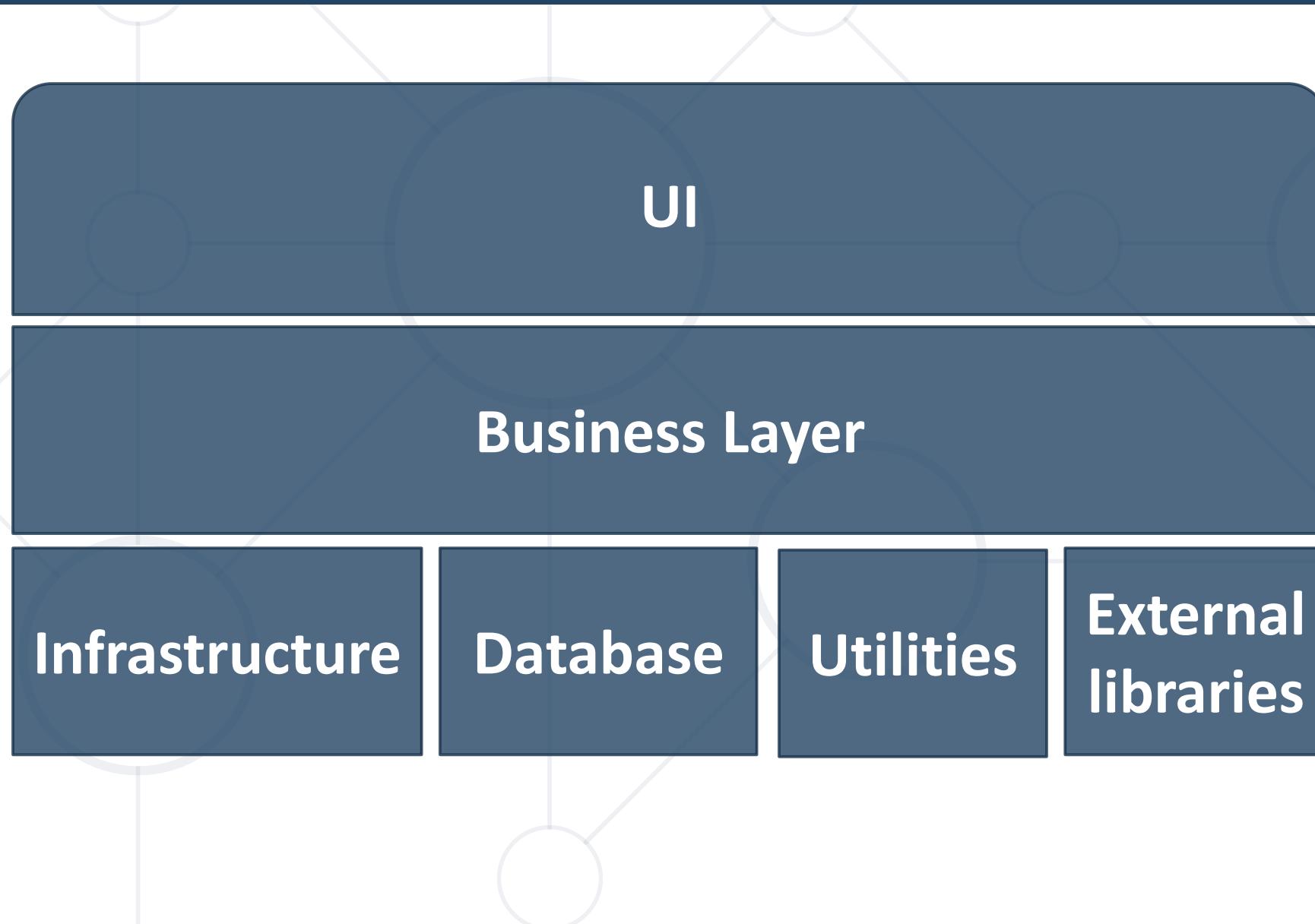
# Dependency Inversion

# Dependency Examples

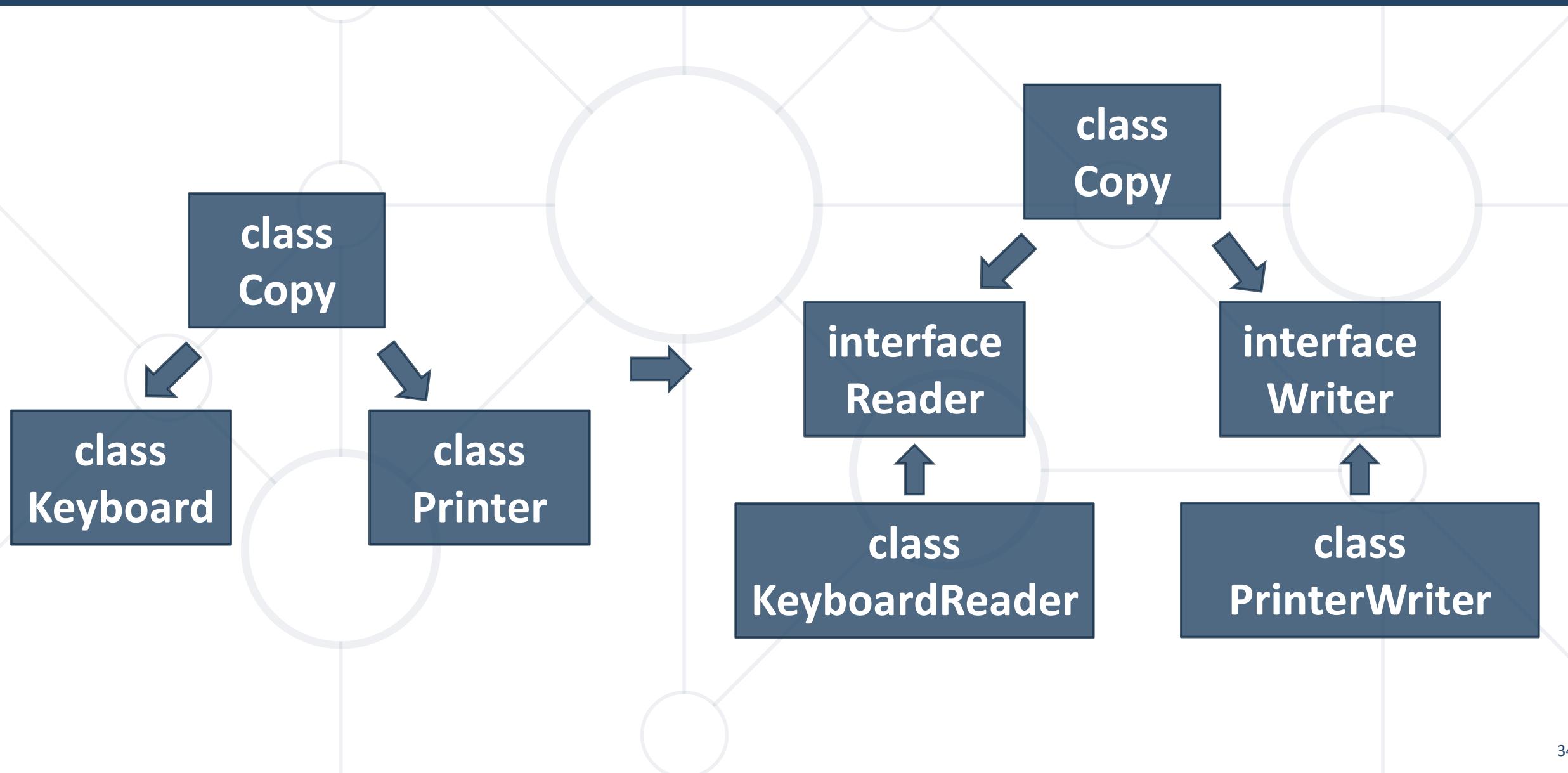
- A **dependency** is any external component / system:
  - Framework
  - 3<sup>rd</sup> party library
  - Database
  - File system
  - Email
  - Web service
  - System resource (e.g. clock)
- Configuration
- The **new** keyword
- Static method
- Global function
- Random generator
- Console



# Dependencies in Traditional Programming



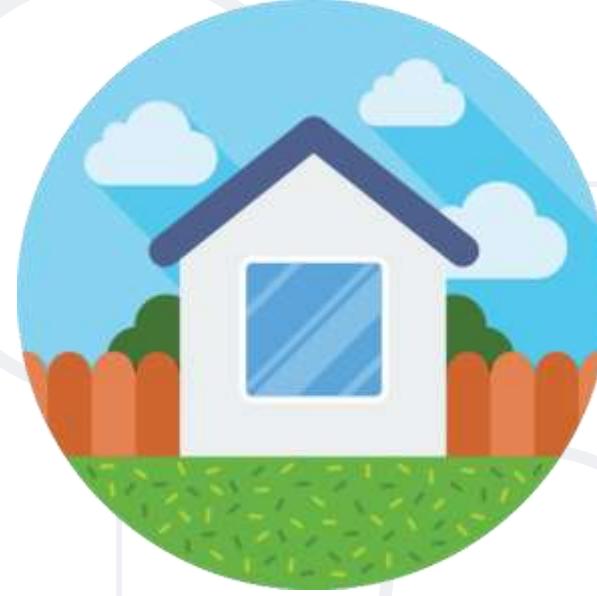
# Depend On Abstractions



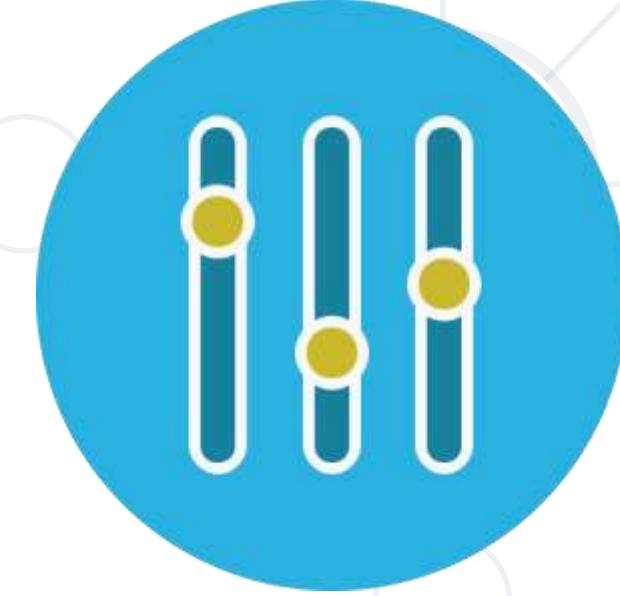
# Types of Dependency Inversion



**Constructor  
injection**



**Property  
injection**



**Parameter  
injection**

# Constructor Inversion – Pros and Cons

## ■ Pros

- Class' requirements are self-documenting
- We don't have to worry about state validation

## ■ Cons

- Too many parameters
- Sometimes, the functionality doesn't need all of the dependencies



# Constructor Inversion – Example

```
class Copy
{
    private IReader reader;
    private IWriter writer;
    public Copy(IReader reader, IWriter writer)
    {
        this.reader = reader;
        this.writer = writer;
    }
    // Read/Write data through the reader/writer
}
var copy = new Copy(new ConsoleReader(),
                    new FileWriter("out.txt"));
```

# Property Inversion – Pros and Cons

- Pros

- Functionality can be changed at any time
- That makes the code very flexible

- Cons

- State can be invalid
- Less intuitive to use



# Property Inversion – Example

```
class Copy
{
    public IReader Reader { get; set; }
    public IWriter Writer { get; set; }
    public void CopyAllChars()
    {
        // Read/Write data through the reader/writer
    }
}
Copy copy = new Copy();
copy.Reader = new ConsoleReader();
copy.Writer = new FileWriter("output.txt");
copy.CopyAllChars();
```

# Parameter Inversion – Pros and Cons

- Pros

- Changes are only localized to the method

- Cons

- Too many parameters
  - Breaks the method signature

# Parameter Inversion – Example

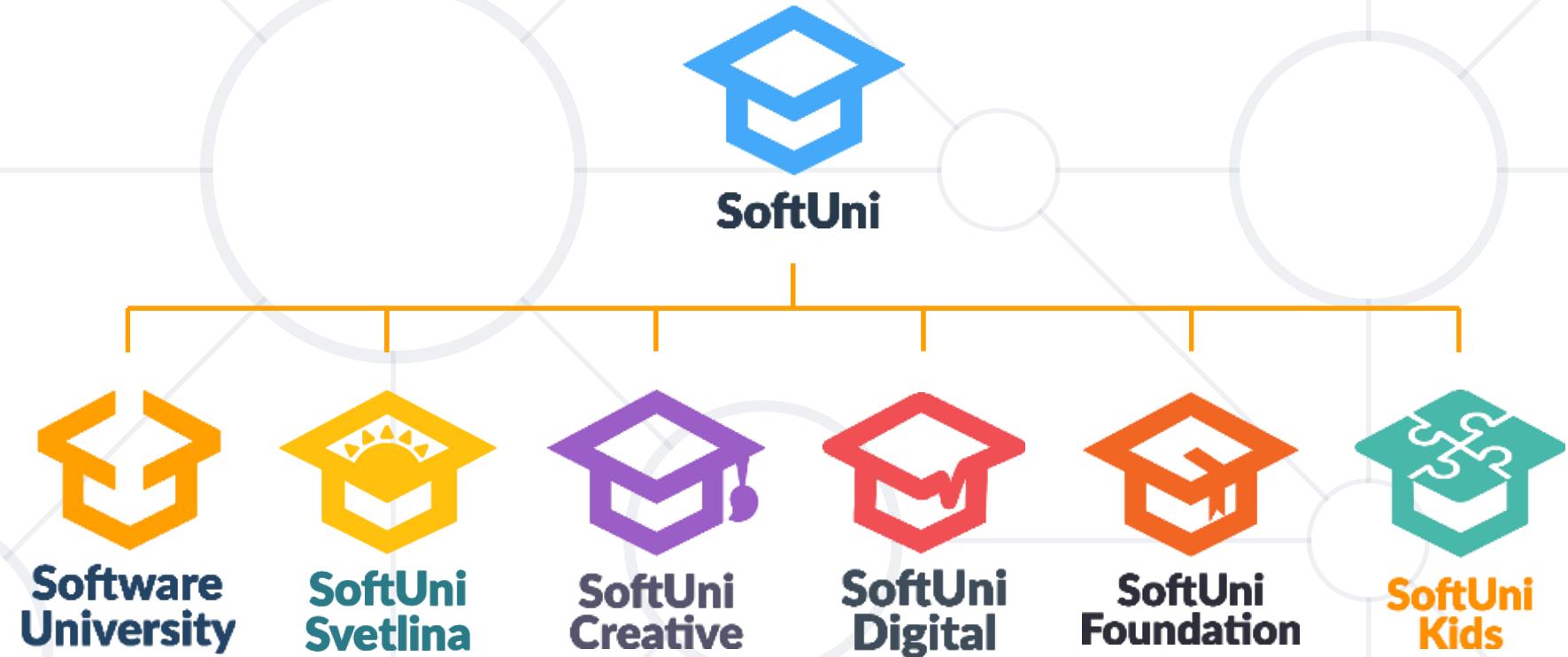
```
class Copy
{
    public void CopyAllChars(IReader reader, IWriter writer)
    {
        // Read/Write data through the Reader/Writer
    }
}
Copy copy = new Copy();
var reader = new ConsoleReader();
var writer = new FileWriter("output.txt");
copy.CopyAllChars(reader, writer);
```

- Classic DIP Violations:
  - Using the **new** keyword
  - Using **static** methods / properties
- How to fix code, that violates the DIP:
  - **Extract interfaces** + use **constructor injection**
  - Set up an Inversion of Control (**IoC**) container

- **SOLID** principle make software more:
  - Understandable
  - Flexible
  - Maintainable



# Questions?



# SoftUni Diamond Partners

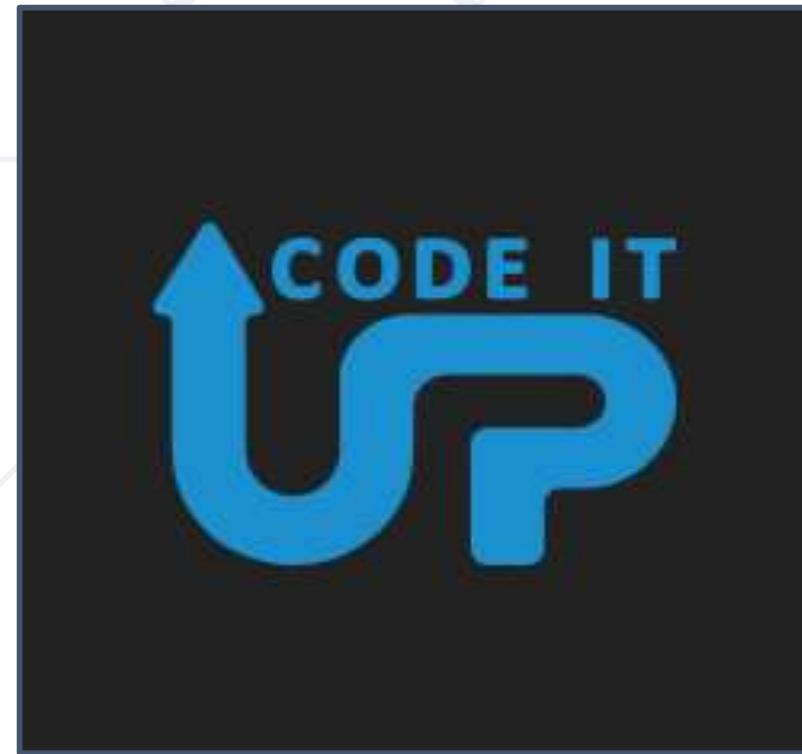


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



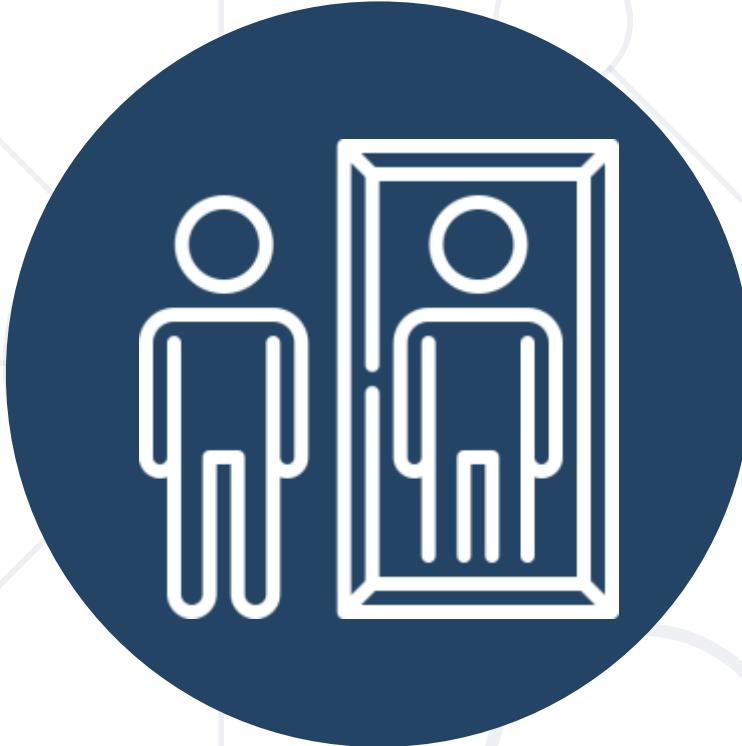
# Table of Contents

- Reflection - What? Why? Where? When?
- Reflection API
  - Type Class
  - Reflecting Fields
  - Reflecting Constructors
  - Reflecting Methods
- Attributes
  - Applying Attributes to Code Elements
  - Built-in Attributes
  - Defining Attributes



sli.do

# #csharp-advanced



**What? Why? Where? When?**

**Reflection**

# What is Metaprogramming?

- 
- **Programming technique**, in which computer programs have the ability to treat other **programs as their data**
  - Programs can be designed to:
    - **Read**
    - **Generate**
    - **Analyze**
    - **Transform**
    - **Modify itself while running**

# What is Reflection?

- The ability of a programming language to be its **own metalanguage**
- Programs can examine information about **themselves**



# When to Use Reflection?

- Whenever we want:
  - Code to become more **extendible** (e.g. plugins)
  - To **reduce code** length significantly (e.g. mapping)
  - Dynamic object **initialization** (e.g. IoC containers)
  - **Assembly** information at run time (e.g. ASP.NET Core)
  - Examine other **programs** (e.g. unit testing)



# When Not to Use Reflection?

- If it is **possible** to **perform** an operation **without** using **reflection**, then it is preferable to **avoid using it**
- Cons from using Reflection
  - **Performance** overhead
  - **Security** restrictions
  - Exposure of **internals**



# Reflecting Class and Members

Reflection API

# Type Class

- Primary way to access **metadata**
- Obtained at **compile time**, if you know its **name**:

```
Type myType = typeof(ClassName);
```

- Can be obtained at **runtime**, if the name is **unknown**:

```
Type myType = Type.GetType("Namespace.ClassName");
```

- Get the type of an instance

```
obj.GetType();
```

You need fully qualified  
class name as string

# Class Name

- Obtain Class name
  - Fully qualified class name - Type.FullName

```
string fullName = typeOf(SomeClass).FullName;
```

- Class name without the namespace - Type.Name

```
string simpleName = typeOf(SomeClass).Name;
```

# Base Class and Interfaces

- Obtain **base type**

```
Type baseType = testClass.BaseType;
```

- Obtain **interfaces**

```
Type[] interfaces = testClass.GetInterfaces();
```

- All the **interfaces that the class implements** are returned
  - Even interfaces from **base classes**

# Creating New Instances Dynamically

- **Activator.CreateInstance**

- Creates an instance of a type by invoking the constructor that **matches** the specified **arguments**

```
var sbType = Type.GetType("System.Text.StringBuilder");
StringBuilder sbInstance =
    (StringBuilder) Activator.CreateInstance(sbType);
StringBuilder sbInstCapacity = (StringBuilder)Activator
    .CreateInstance(sbType, new object[] { 10 });
```

- Obtain public fields

```
FieldInfo field = type.GetField("name");
FieldInfo[] publicFields = type.GetFields();
```

- Obtain all fields

```
FieldInfo[] allFields = type.GetFields(
    BindingFlags.Static |
    BindingFlags.Instance |
    BindingFlags.Public |
    BindingFlags.NonPublic);
```

# Binding Flags

- The **BindingFlags** enum specifies what kinds of types we are looking up

```
FieldInfo[] allFields =  
    type.GetFields(BindingFlags.NonPublic);
```

- Can be combined with bitwise OR (**|** operator):

```
FieldInfo[] allFields = type.GetFields(  
    BindingFlags.Public |  
    BindingFlags.NonPublic);
```

Returns both public  
and nonpublic fields

# Field Type and Name

- Get **public** field **name** and **type**

```
FieldInfo field = type.GetField("fieldName");
string fieldName = field.Name;
Type fieldType = field.FieldType;
```

- Use **BindingFlags** to specify access modifiers, if the field is not public, otherwise **GetField** returns **null**

# Changing a Field's State

```
Type testType = typeof(Test);  
Test testInstance =  
    (Test) Activator.CreateInstance(testType);  
FieldInfo field = testType.GetField("testInt");  
  
field.SetValue(testInstance, 5);  
int fieldValue =  
    (int)field.GetValue(testInstance);
```

Changes the  
object's state

# Access Modifiers

- Each modifier is **a flag bit** that is either set or cleared
- Check **access modifier** of a **member** of the class

```
field.IsPrivate      // private
field.IsPublic       // public
field.IsNonPublic    // everything but public
field.IsFamily        // protected
field.IsAssembly     // internal
```

# Reflect Constructors

- Obtain **constructors**

```
ConstructorInfo[] publicCtors =  
    type.GetConstructors();
```

- Obtain **all non static constructors**

```
ConstructorInfo[] allNonStaticCtors =  
    type.GetConstructors(  
        BindingFlags.Instance |  
        BindingFlags.Public |  
        BindingFlags.NonPublic);
```

# Reflect Constructors(2)

- Obtain a certain constructor

```
ConstructorInfo constructor =  
    type.GetConstructor(new Type[] parametersType);
```

- Get constructor parameters

```
Type[] parameterTypes =  
    constructor.GetParameters();
```

- Instantiating objects using a specific constructor

```
StringBuilder builder =  
    (StringBuilder)constructor.Invoke(  
        new object[] { "gosh", 5 });
```

Supply positional parameters  
in an object array

# Reflect Methods

- Obtain all **public** methods

```
MethodInfo[] publicMethods = sbType.GetMethods();
```

- Obtain a **certain** method

```
MethodInfo appendMethod =
    sbType.GetMethod("Append");
MethodInfo overloadMethod = sbType.GetMethod(
    "Append", new [] { typeof(array) });
```

# Method Invoke

- Obtain method **parameters** and **return type**

```
ParameterInfo[] appendParameters =  
    appendMethod.GetParameters();  
Type returnType = appendMethod.ReturnType;
```

- **Invoke** methods

```
appendMethod.Invoke(builder, new object[] { "hi!" });
```

Target object  
instance

Parameters for  
the method



# Data about Data

## Attributes

# Attributes

- Data holding class
- Describes parts of your code
- Applied to:
  - Classes, Fields, Methods, etc.



```
[Obsolete]
public void DeprecatedMethod
{
    Console.WriteLine("Deprecated!");
}
```

# Attributes Usage

- Generate **compiler messages or errors**

[Obsolete]

```
public enum Coin // Enum 'Coin' is obsolete
```

- Tools, which rely on attributes:
  - **Code generation tools**
  - **Documentation generation tools**
  - **Testing Frameworks**
- Runtime - **ORM, Serialization** etc.

# Applying Attributes – Example

- Attribute's name is surrounded by **square brackets: []**
  - Placed before their target declaration

```
[Flags] // System.FlagsAttribute
public enum FileAccess
{
    Read = 1,
    Write = 2,
    ReadWrite = Read | Write
}
```

- **[Flags]** attribute indicates that the enum type can be treated like a set of bit flags, stored as a single integer

# Attributes with Parameters

- Attributes can accept **parameters** for their constructors and public properties

```
[DllImport("user32.dll", EntryPoint="MessageBox")]
public static extern int ShowMessageBox(int hWnd,
    string text, string caption, int type);

...
ShowMessageBox(0, "Some text", "Some caption", 0);
```

- The **[DllImport]** attribute refers to:
  - System.Runtime.InteropServices.DllImportAttribute**
  - "**user32.dll**" is passed to the constructor
  - "**MessageBox**" value is assigned to **EntryPoint**

# Custom Attributes Requirements

- Must inherit the `System.Attribute` class
- Their names must end with "`Attribute`"
- Possible targets must be defined via `[AttributeUsage]`
- Can define constructors with parameters
- Can define public fields and properties

# Problem: Create Attribute

- Create an attribute **Author** with a **string** element called **name** that:
  - Can be used over **classes and methods**
  - Allow multiple attributes of same type

```
[Author("Victor")]
public class StartUp
{
    [Author("Georg")]
    static void Main(string[] args)
    { ... }
}
```

Check your solution here: <https://judge.softuni.bg/Contests/1520/Reflection-and-Attributes-Lab>

# Solution: Create Attribute

```
[AttributeUsage(AttributeTargets.Class |
    AttributeTargets.Method,
    AllowMultiple = true)]
public class AuthorAttribute : Attribute
{
    public AuthorAttribute(string name)
    {
        this.Name = name;
    }

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

# Problem: Coding Tracker

- Create a class **Tracker** with a method:
  - **void PrintMethodsByAuthor()**
- Print to the console authors for all methods
  - Use **SoftUni attribute** and **reflection**

Check your solution here: <https://judge.softuni.bg/Contests/1520/Reflection-and-Attributes-Lab>

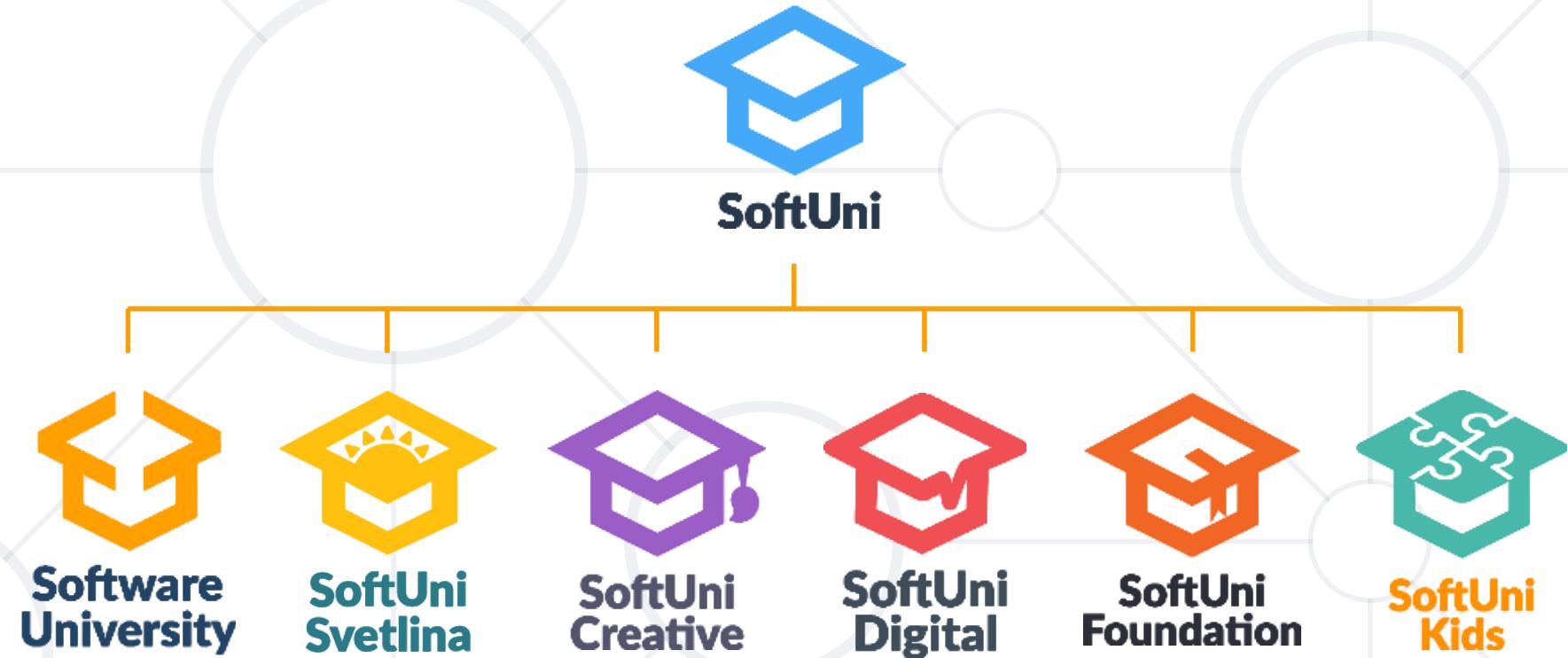
# Solution: Coding Tracker

```
var type = typeof(StartUp);
var methods =
    type.GetMethods(
        BindingFlags.Instance | BindingFlags.Public |
        BindingFlags.Static);
foreach (var method in methods) {
    if(method.CustomAttributes
        .Any(n => n.AttributeType == typeof(AuthorAttribute))){
        var attributes = method.GetCustomAttributes(false);
        foreach(AuthorAttribute attr in attributes){
            Console.WriteLine("{0} iw written by {1}",
                method.Name, attr.Name);
    }
    // Add the missing brackets
```

- **Reflection:**
  - Allows us to get **information about types**
  - Allows us to dynamically **call methods**,  
**get/set** values, etc.
- **Attributes** allow adding metadata in classes / types / etc.
  - Built-in attributes
  - Custom attributes
  - Can be accessed at runtime



# Questions?



# SoftUni Diamond Partners

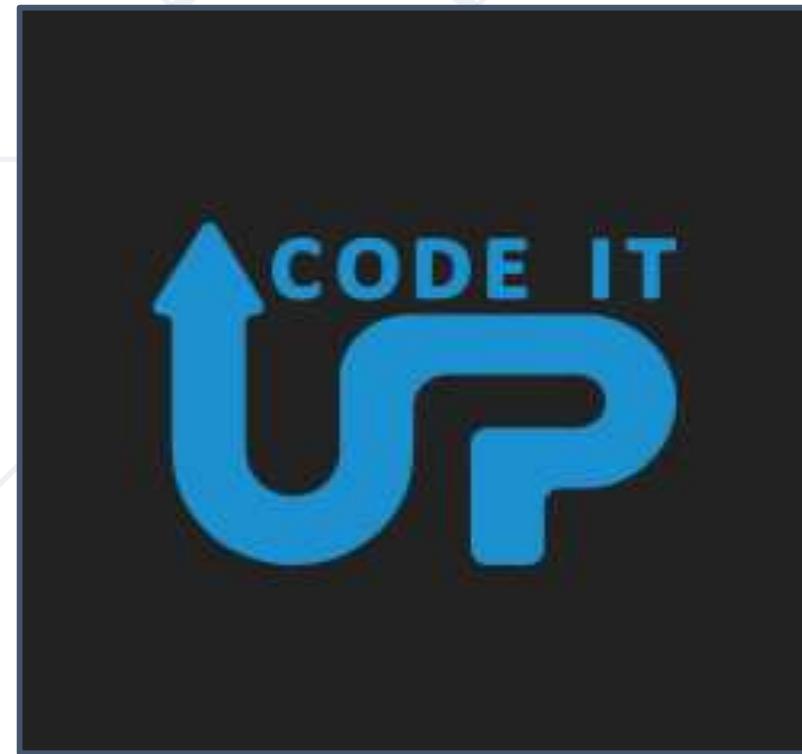


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

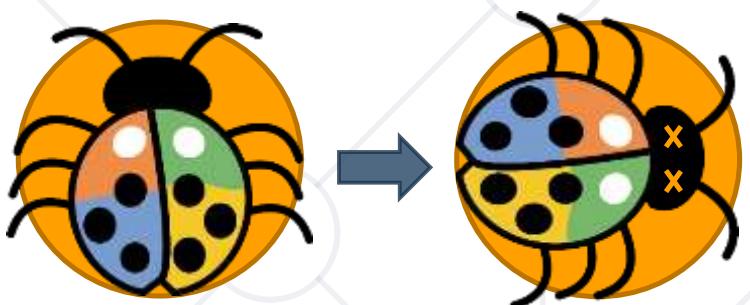


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



# Unit Testing

## Building Rock-Solid Software



**SoftUni Team**  
**Technical Trainers**



**Software University**  
<https://softuni.bg>

# Table of Contents

- Seven Testing Principles
- What is Unit Testing?
- Unit Testing Frameworks
- NUnit
  - NUnit Setup
  - Test Classes and Test Methods
  - 3A-s Pattern
- Good Practices



sli.do

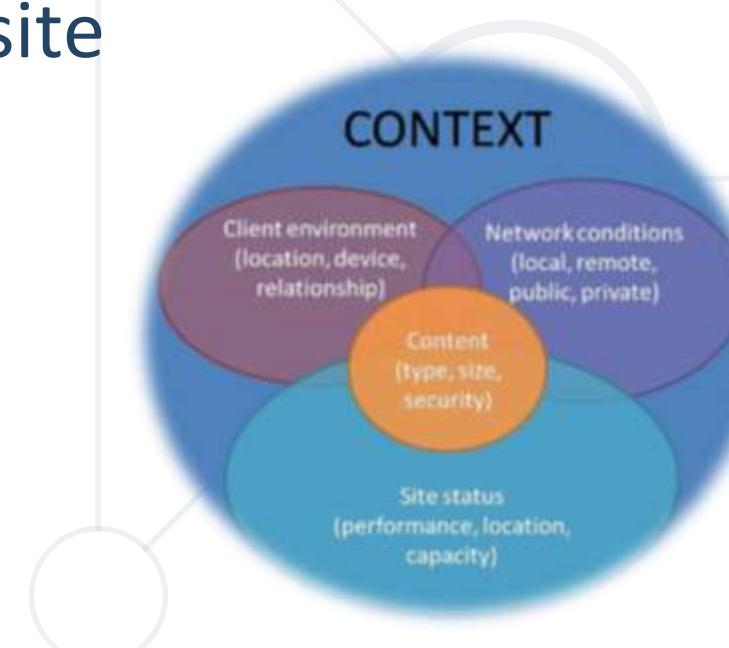
# #csharp-advanced



# Seven Testing Principles

# Seven Testing Principles (1)

- Testing is context dependent
  - Testing is done differently in **different contexts**
- Example:
  - Safety-critical software is tested **differently** from an e-commerce site



# Seven Testing Principles (2)

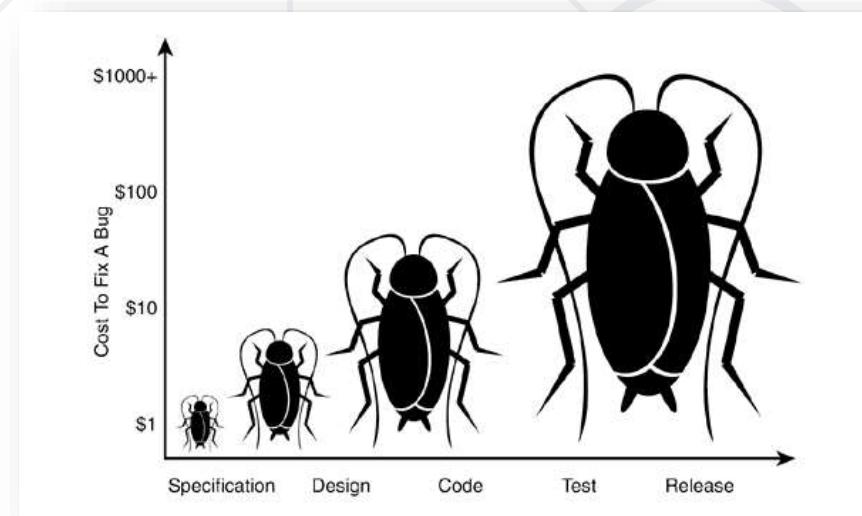
- Exhaustive testing is **impossible**
  - All combinations of inputs and preconditions are usually almost **infinite number**
- Testing everything is not feasible
  - Except for trivial cases
  - Risk analysis and priorities should be used to focus testing efforts
- E.g.: [Big list of naughty strings](#)

A QA Tester walks into a bar:

He orders a beer.  
He orders 3 beers.  
He orders 2976412836 beers.  
He orders 0 beers.  
He orders -1 beer.  
He orders q beers.  
He orders nothing.  
Él ordena una cerveza.  
He orders a deer.  
He tries to leave without paying.  
He starts ordering a beer, then throws himself through the window half way through.  
He orders a beer, gets his receipt, then tries to go back.

# Seven Testing Principles (3)

- Early testing is **always preferred**
  - Testing activities shall be started as early as possible
    - And shall be focused on defined objectives
  - The later a bug is found – the more it costs!



# Seven Testing Principles (4)

- Defect clustering
  - Testing effort shall be focused **proportionally**
    - To the expected and later observed defect density of modules
  - A **small number** of modules usually contains **most of the defects** discovered (80/20 principle)
    - Responsible for most of the operational failures

# Seven Testing Principles (5)

- Pesticide paradox
  - Same tests repeated **over and over again** tend to **lose their effectiveness**
  - Previously **undetected** defects remain **undiscovered**
  - New and modified test cases should be developed

# Seven Testing Principles (6)

- Testing shows presence of defects
  - Testing can **show that defects are present**
  - Cannot prove that there are no defects
  - Appropriate testing **reduces** the probability for defects

# Seven Testing Principles (7)

- Absence-of-errors fallacy
  - **Finding** and **fixing** defects itself does not help in these cases:
    - The system built is unusable
    - Does not fulfill the users' needs and expectations



# Software Used to Test Software

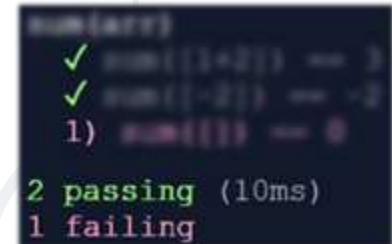
What is Unit Testing?

- **Unit test** == a piece of code that **tests specific functionality** in certain software component (unit)

```
int Sum(int[] arr)
{
    int sum = arr[0];
    for (int i=1; i<arr
        .Length; i++)
        sum += arr[i];
    return sum;
}
```

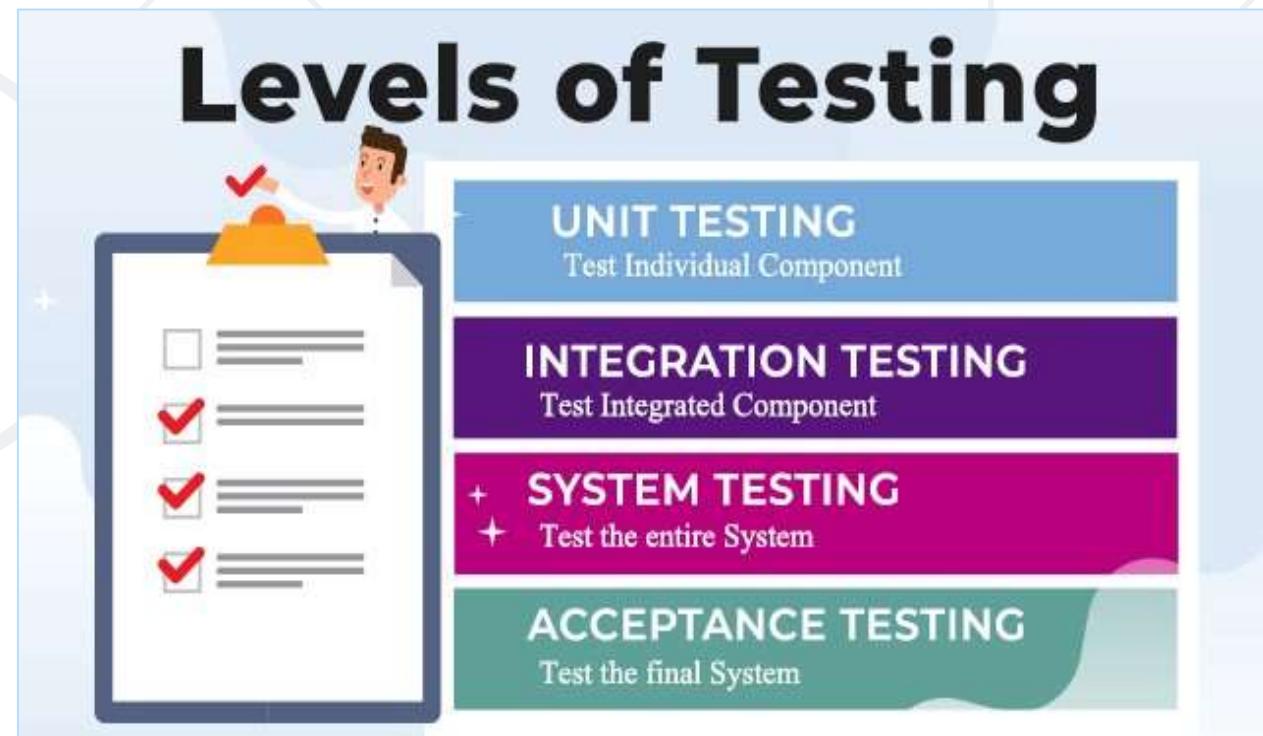
```
void Test_SumTwoNumbers() {
    if (Sum(new int[]{1, 2}) != 3)
        throw new Exception("1+2 != 3");
}

void Test_SumEmptyArray() {
    if (Sum(new int[]{ }) != 0)
        throw new Exception("sum [] != 0");
}
```



# Test Levels

- **Unit tests**
  - Test a **single component** (mocking the dependencies)
  - NUnit, JUnit, PyUnit, Mocha
- **Integration tests**
  - Test an **interaction** between components, e. g. **API tests**
- **System tests / acceptance tests / end-to-end tests**
  - Test the **entire system**, e. g. Selenium, Appium, Cypress, Playwright





# Unit Testing Frameworks

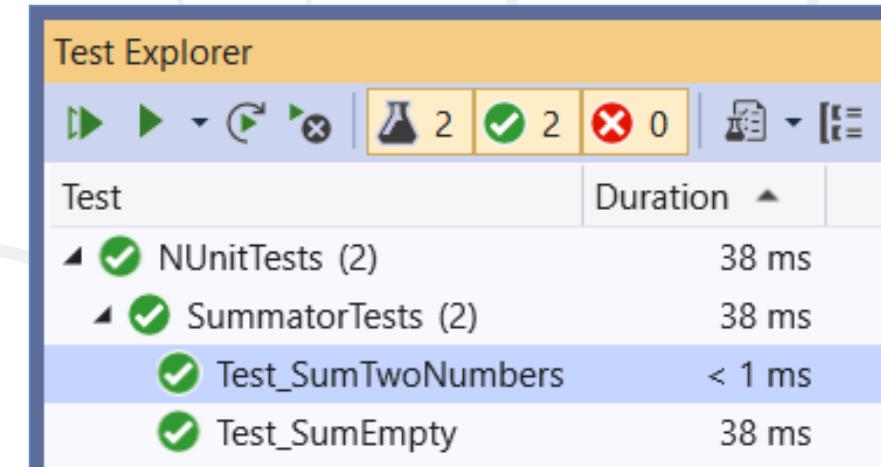
- **Testing frameworks** provide **foundation for test automation**
  - Consists of **libraries**, code **modules** and **tools** for test automation
  - **Structure the tests** into hierarchical or other form
  - **Implement** test cases, **execute the tests** and **generate reports**
  - **Assert** the execution results and exit conditions
  - Perform initialization at **startup** and cleanup at **shut down**
- **Examples** of testing frameworks:
  - NUnit, xUnit, MSTest (C#), JUnit (Java), Mocha (JS), PyUnit (Python)

# Testing Framework – Example

- Testing frameworks simplify automated testing and reporting
  - Example: **NUnit** testing framework for C#

```
using NUnit.Framework;

public class SummatorTests
{
    [Test]
    public void Test_SumTwoNumbers() {
        var sum = Sum(new int[] { 1, 2 });
        Assert.AreEqual(3, sum);
    }
}
```



- **Unit testing framework == automated testing framework == testing framework == test framework**
  - Many names for similar concepts → why?
  - Testing frameworks like **JUnit** and **NUnit** were initially designed for **unit testing**, but nothing limits them to wider use
  - With additional libraries, NUnit and JUnit are used for:
    - **Integration testing, API testing, Web service testing**
    - **End-to-end testing, Web UI testing, mobile testing, etc.**



# NUnit: First Steps

Setup and First Test

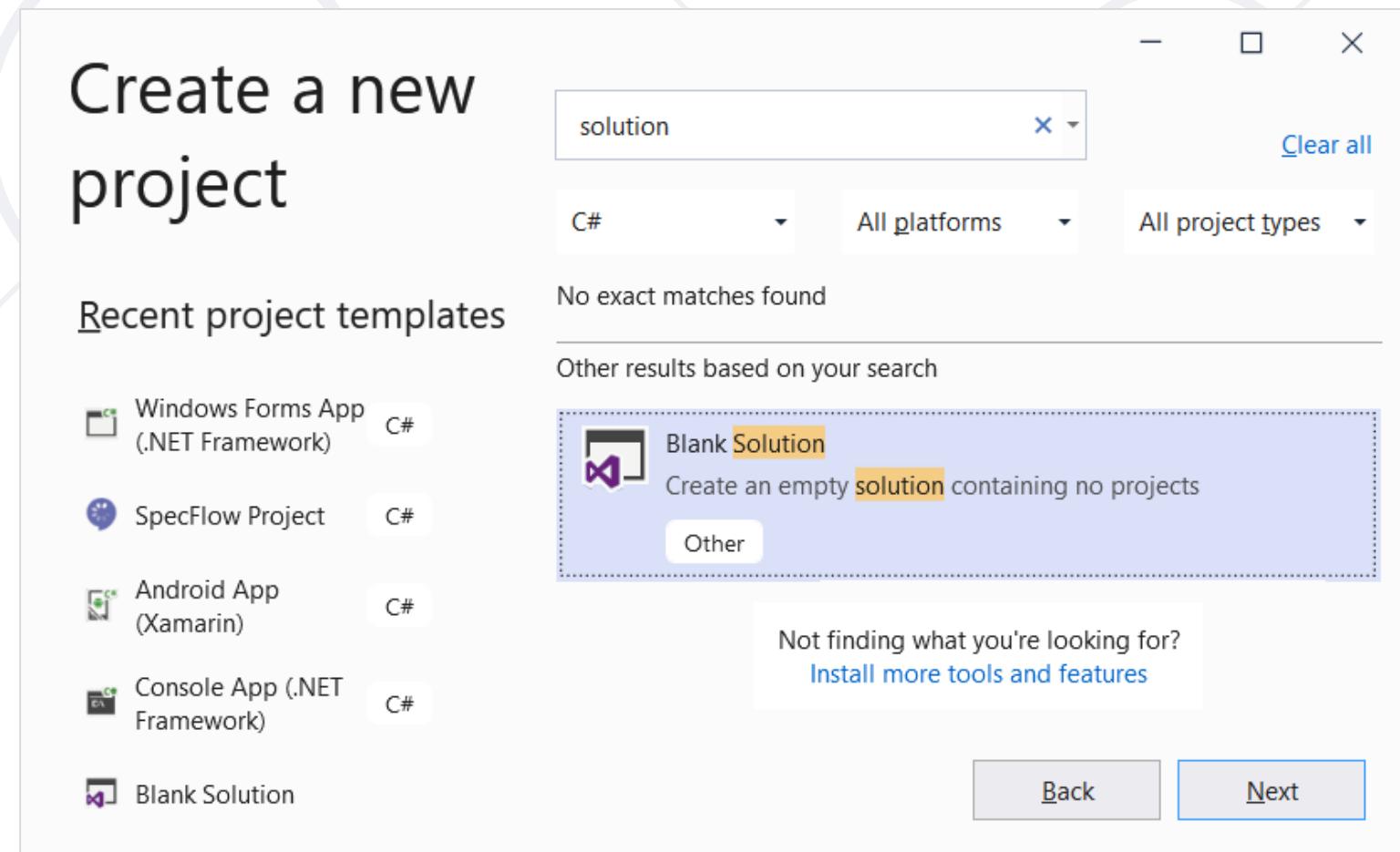
# NUnit: Overview

- **NUnit** == popular C# testing framework
  - Supports test suites, test cases, before & after code, startup & cleanup code, timeouts, expected errors, ...
  - Like **JUnit** (for Java)
  - Free, open-source
  - Powerful and mature
  - Wide community
  - Built-in support in Visual Studio
  - Official site: [nunit.org](http://nunit.org)



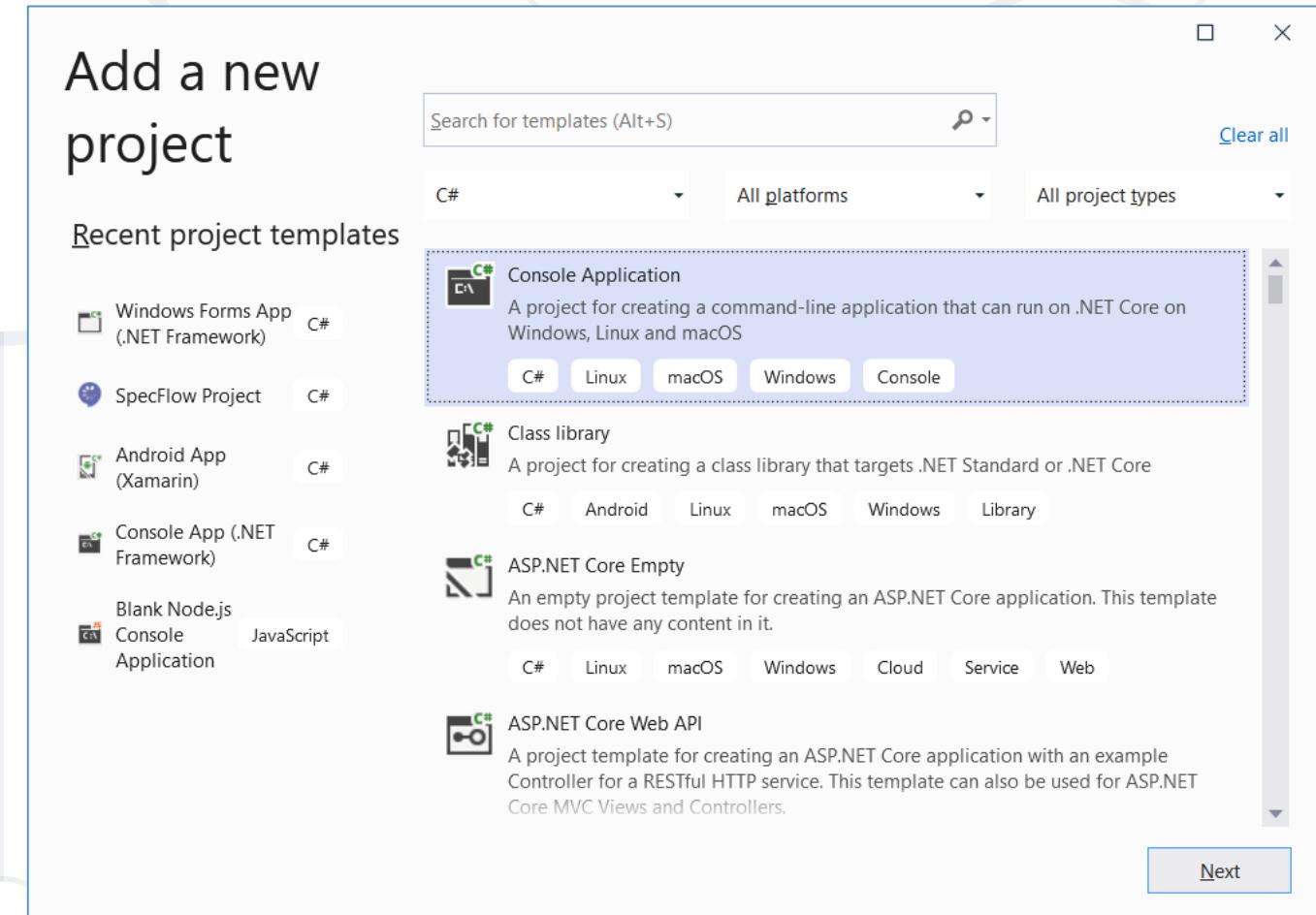
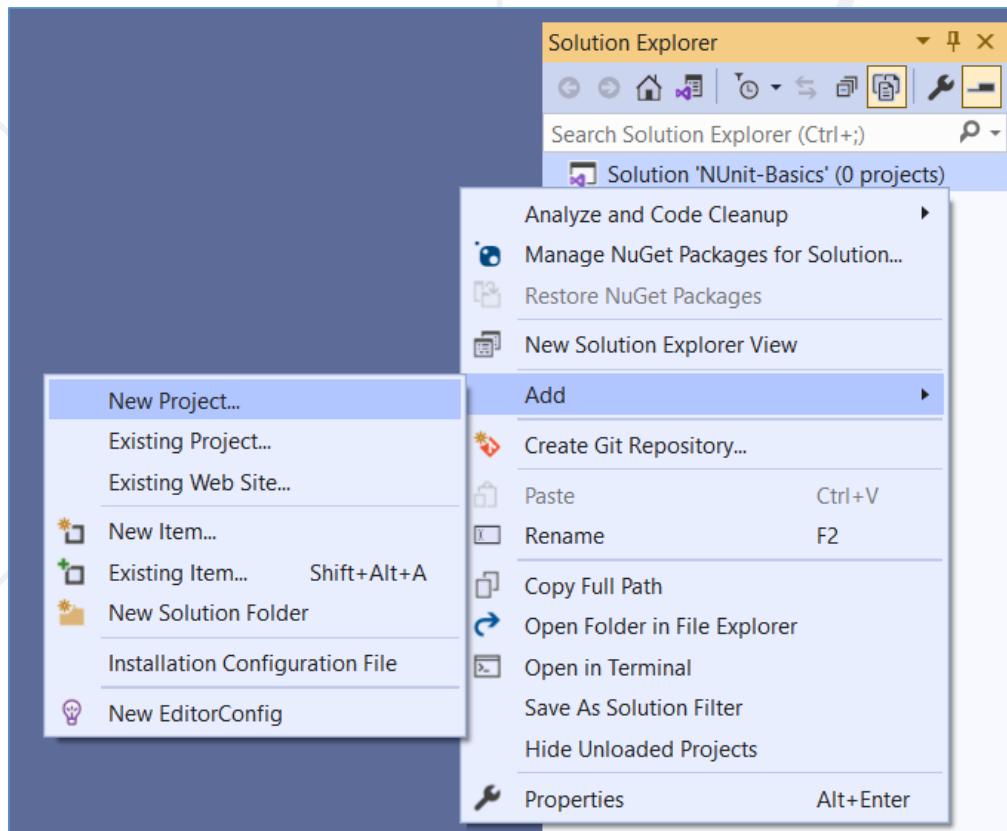
# Creating a Blank Solution

- Create a **blank solution** in Visual Studio
  - It will hold the **project for testing**
  - And the **unit test project (tests)**

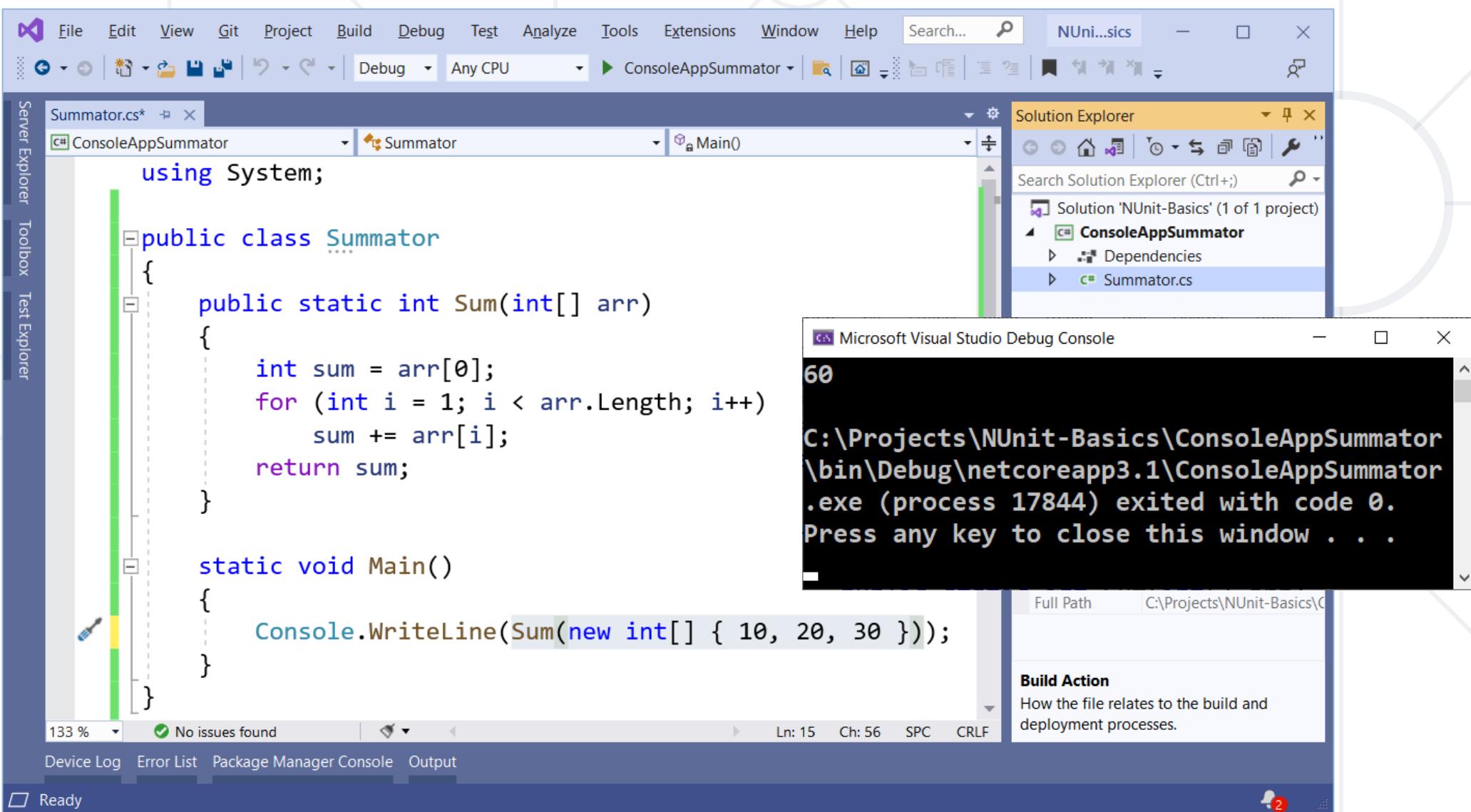


# Creating a Project for Testing

- Create a **console-based app**, to hold the **code for testing**



# Creating a Project for Testing (2)



```
using System;

public class Summator
{
    public static int Sum(int[] arr)
    {
        int sum = arr[0];
        for (int i = 1; i < arr.Length; i++)
            sum += arr[i];
        return sum;
    }

    static void Main()
    {
        Console.WriteLine(Sum(new int[] { 10, 20, 30 }));
    }
}
```

Microsoft Visual Studio Debug Console

60

C:\Projects\NUnit-Basics\ConsoleAppSummator\bin\Debug\netcoreapp3.1\ConsoleAppSummator.exe (process 17844) exited with code 0.

Press any key to close this window . . .

Full Path C:\Projects\NUnit-Basics\ConsoleAppSummator\ConsoleAppSummator.cs

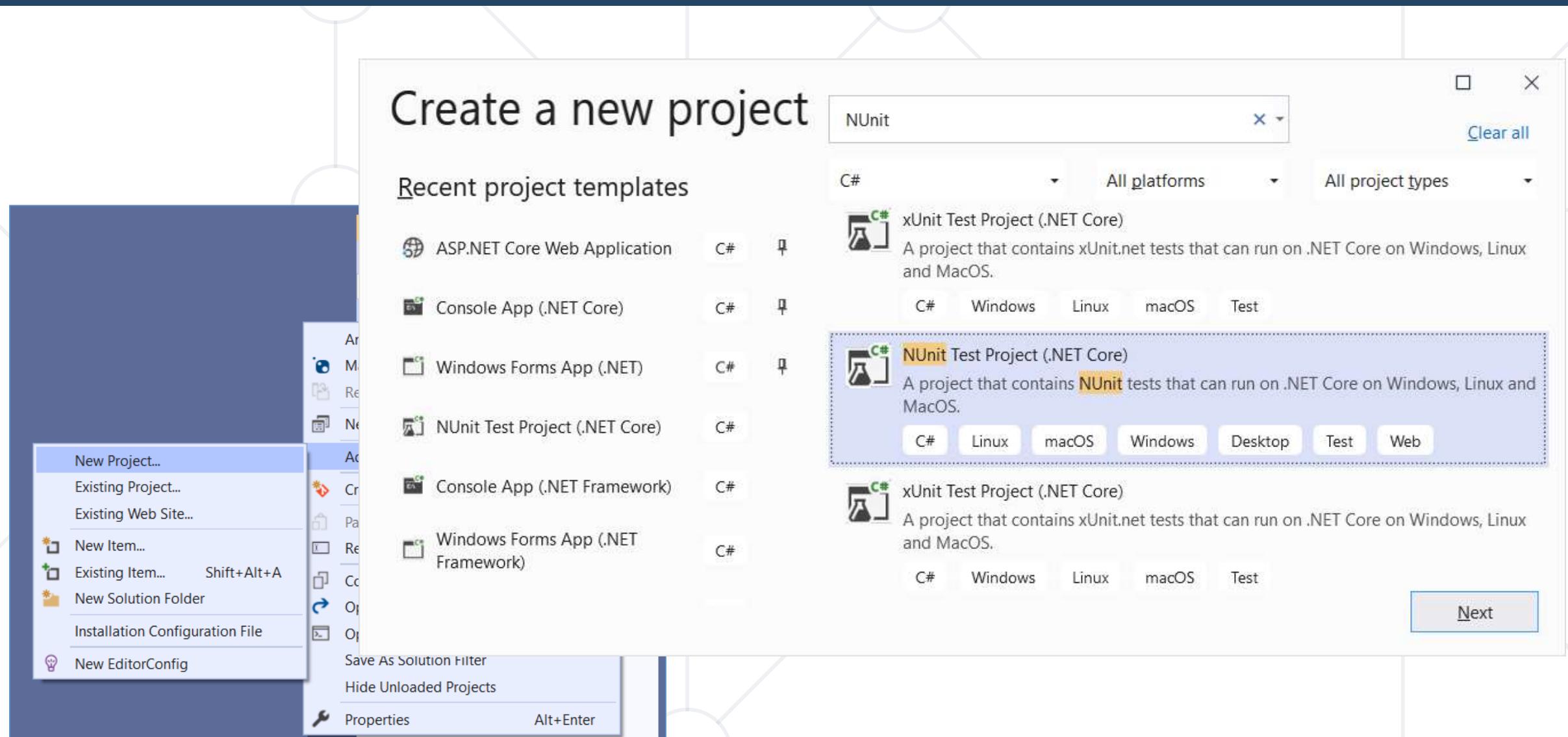
Build Action

How the file relates to the build and deployment processes.

Device Log Error List Package Manager Console Output

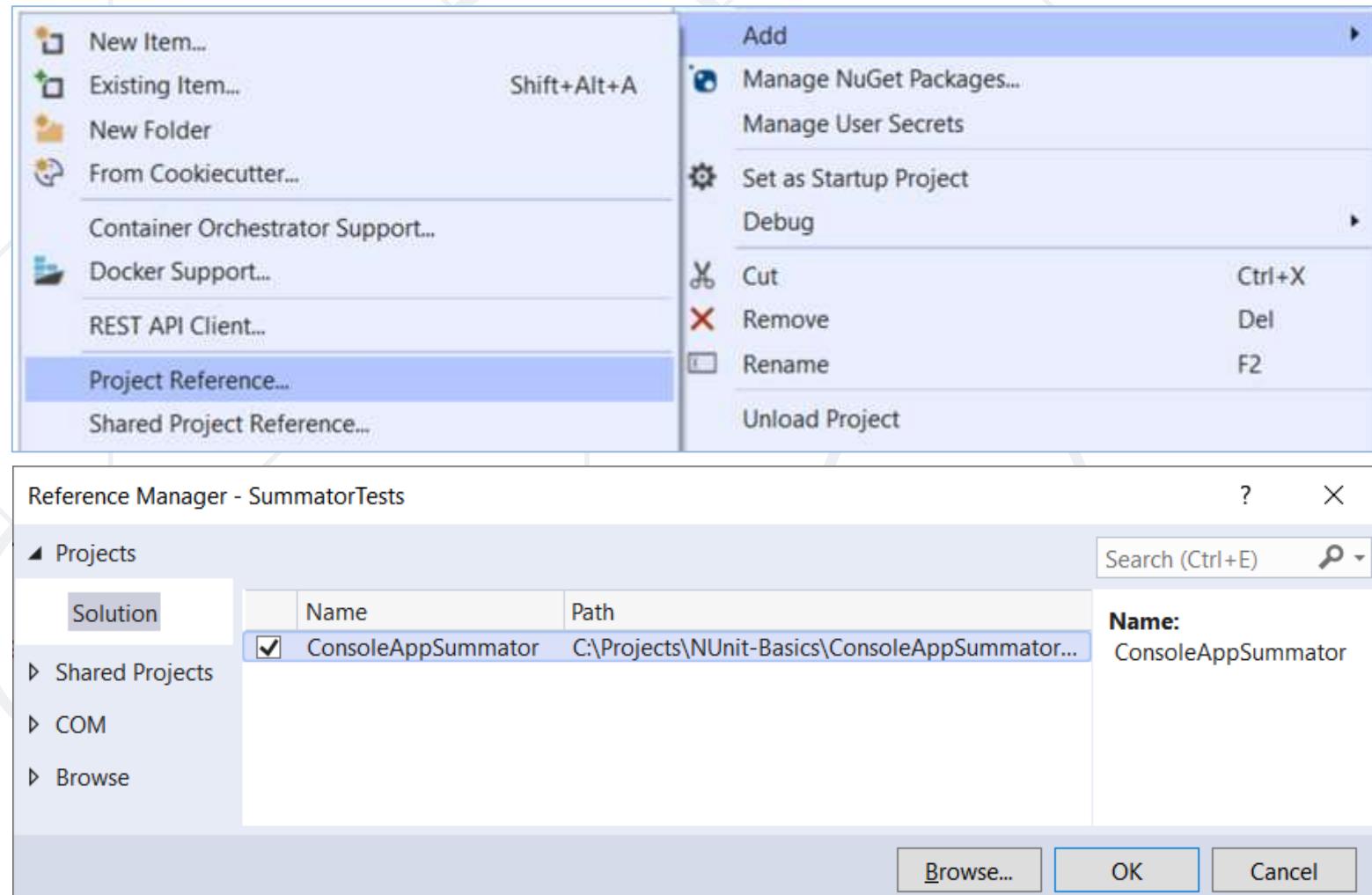
Ready

# Creating an NUnit Project



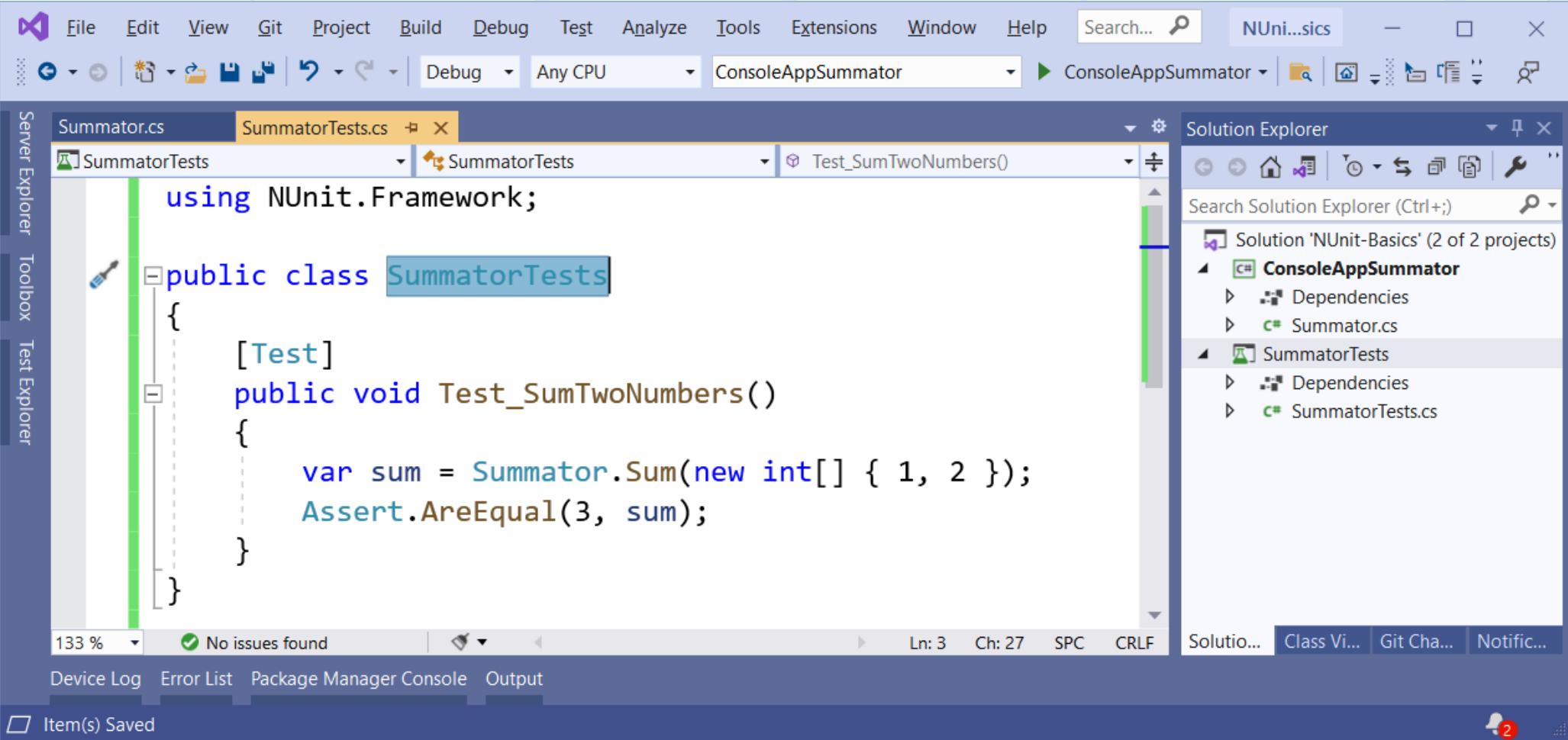
# Adding Project Reference

- Add Project Reference to the target project for testing:



# Writing the First Test

## ■ Writing the first **NUnit** test method:



The screenshot shows the Microsoft Visual Studio IDE interface. The title bar says "NUni...sics" and the solution name is "ConsoleAppSummator". The current file being edited is "SummatorTests.cs". The code in the editor is:

```
using NUnit.Framework;

public class SummatorTests
{
    [Test]
    public void Test_SumTwoNumbers()
    {
        var sum = Summator.Sum(new int[] { 1, 2 });
        Assert.AreEqual(3, sum);
    }
}
```

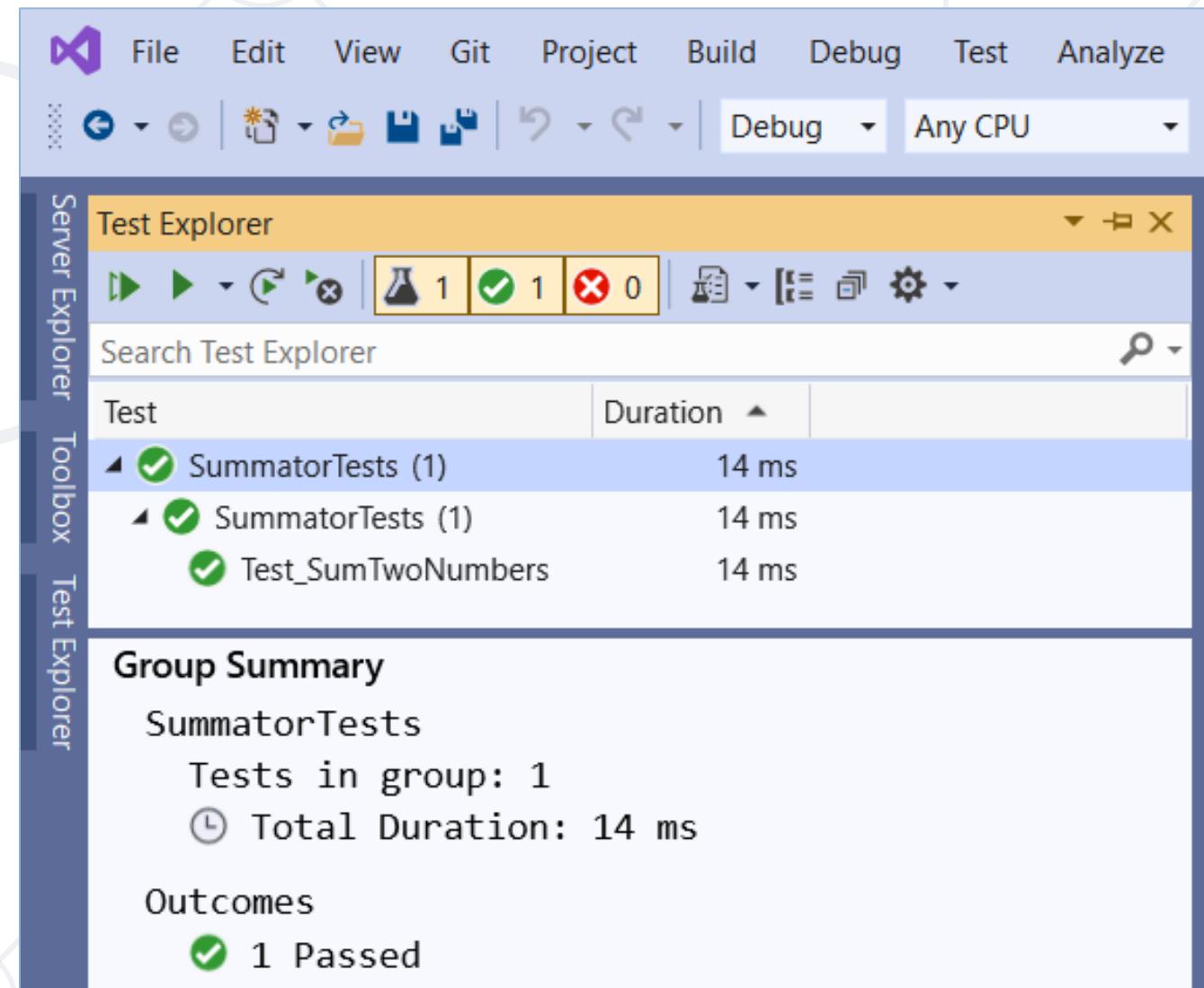
The "Solution Explorer" window on the right shows the project structure:

- Solution 'NUnit-Basics' (2 of 2 projects)
  - ConsoleAppSummator
    - Dependencies
    - Summator.cs
  - SummatorTests
    - Dependencies
    - SummatorTests.cs

The status bar at the bottom shows "133 %", "No issues found", and other development details.

# Running the Tests

- The **[Test Explorer]** tool in Visual Studio
  - Show the **[Test Explorer]**:
    - **[Ctrl + E] + T**
    - Visualizes the **hierarchy** of tests
    - **Executes** tests
    - **Reports** results



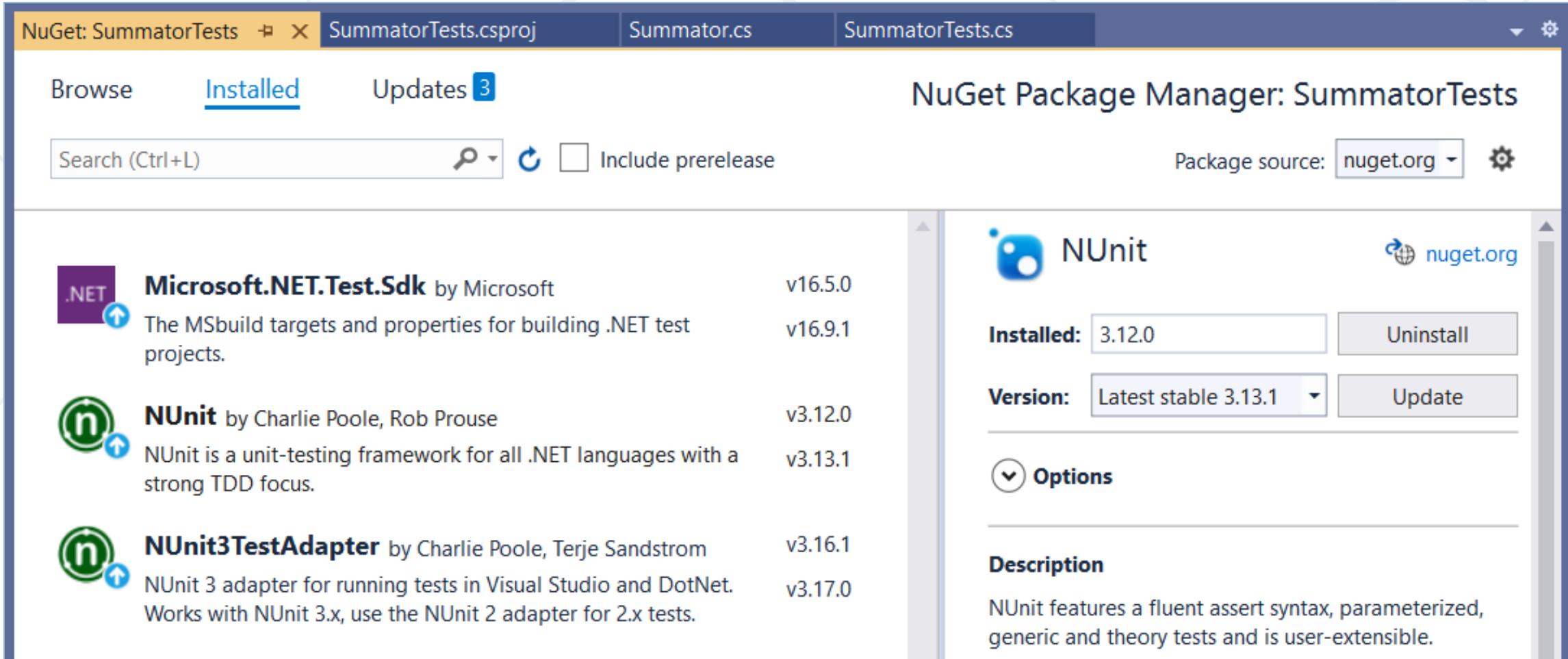


# NUnit: Basics

Test Classes and Test Methods

# NUnit: NuGet Packages

- **NuGet packages**, required to run NUnit tests in Visual Studio



The screenshot shows the NuGet Package Manager interface within Visual Studio. The title bar indicates the project is "SummatorTests.csproj". The "Installed" tab is selected, showing three packages:

Package	Version	Description
Microsoft.NET.Test.Sdk	v16.5.0 v16.9.1	The MSbuild targets and properties for building .NET test projects.
NUnit	v3.12.0 v3.13.1	NUnit is a unit-testing framework for all .NET languages with a strong TDD focus.
NUnit3TestAdapter	v3.16.1 v3.17.0	NUnit 3 adapter for running tests in Visual Studio and DotNet. Works with NUnit 3.x, use the NUnit 2 adapter for 2.x tests.

On the right side, the details for the NUnit package are expanded, showing it is installed at version 3.12.0, with options to "Uninstall" or "Update" to the latest stable version (3.13.1). The description notes that NUnit features a fluent assert syntax, parameterized, generic and theory tests and is user-extensible.

# Test Classes and Test Methods

- Test classes hold test methods:

```
using NUnit.Framework;  
  
[TestFixture]  
public class SummatorTests  
{  
    [Test]  
    public void Test_SumTwoNumbers() {  
        var sum = Sum(new int[] { 1, 2 });  
        Assert.AreEqual(3, sum);  
    }  
}
```

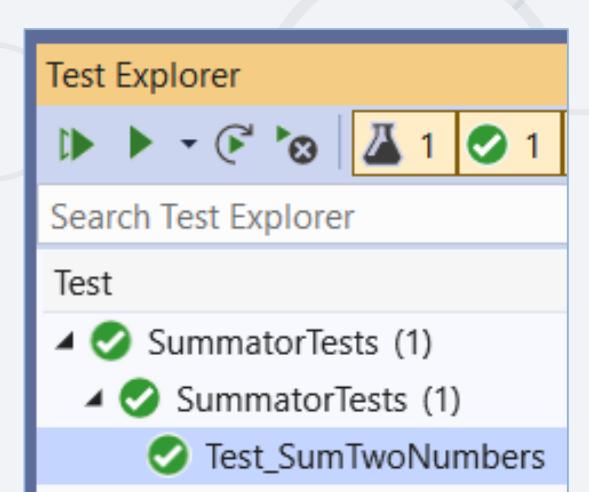
Import NUnit

Optional notation

Test class

Test method

Assertion



# Initialization and Cleanup Methods

```
private Summator summator;  
  
[SetUp] // or [OneTimeSetUp]  
public void TestInitialize()  
{  
    this.summator = new Summator();  
}  
  
[TearDown] // or [OneTimeTearDown]  
public void TestCleanup()  
{  
    // ...  
}
```

Executes before  
each test

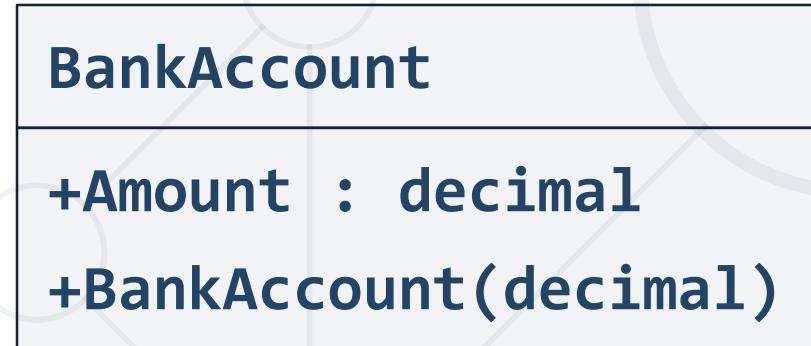
Executes after  
each test

# Problem: NUnit Test

- Create console application project
- Add BankAccount class
- Create NUnit Project
- Test the BankAccount class

# Solution: NUnit Test (1)

- Create a console **application**
  - Add BankAccount class for us to test



- Create a new **NUnit Test Project**
  - Name it like the project you are testing, but with "**.Tests**" suffix
  - Open **Test Explorer** (Ctrl + E, T or Test->Windows->TestExplorer)

# Solution: NUnit Test (2)

- Write your first test

```
[TestFixture]  
public class BankAcountTests  
{  
    [Test] Test Method  
    public void AccountInitializeWithPositiveValue() {  
        BankAccount account = new BankAccount(2000m);  
        Assert.That(account.Amount, Is.EqualTo(2000m));  
    }  
}
```

Attribute that marks a class that contains tests

Test Method

Assert class comes with NUnit

# The "AAA" Testing Pattern

- Automated tests usually follow the **"AAA" pattern**
  - **Arrange**: prepare the **input** data and entrance conditions
  - **Act**: invoke the **action** for testing
  - **Assert**: check the **output** and exit conditions

```
[Test]
public void Test_SumNumbers()
{
    // Arrange
    var nums = new int[] {3, 5};

    // Act
    var sum = Sum(nums);

    // Assert
    Assert.AreEqual(8, sum);
}
```



# How to Write Good Tests

## Unit Testing Best Practices

# Naming the Test Methods

- Test names should answer the question "*what's inside?*"
  - Should use **business domain terminology**
  - Should be **descriptive** and **readable**

```
IncrementNumber() {}  
Test1() {}  
TestTransfer() {}
```



```
Test_DepositAddsMoneyToBalance() {}  
Test_DepositNegativeShouldNotAddMoney() {}  
Test_TransferSubtractsFromSourceAddsToDestAccount() {}
```



# Automated Tests: Good Practices

- Test cases must be **repeatable**
  - Tests should behave the same if you run them many times
  - The expected results must be **consistent** and easily verified
- Test cases should **have no dependencies**
  - The order of test execution should never be important
  - Input data and entrance conditions should be set in the test
  - Test cases may depend on the test initialization only: **[SetUp]**
  - Tests should **cleanup** properly any resources used

# Automated Tests: Good Practices (2)

- Single scenario per test case, not multiple

```
[Test]
public void Test_Collection_RemoveAt()
{
    var names = new Collection<string>("Peter", "Maria", "Steve", "Mia");
    var removed = names.RemoveAt(1);
    Assert.That(removed, Is.EqualTo("Maria"));
    Assert.That(names.ToString(), Is.EqualTo("[Peter, Steve, Mia]"));
}

[Test]
public void Test_Collections_RemoveAtStart()
{
    var names = new Collection<string>("Peter", "Maria", "Steve", "Mia");
    var removed = names.RemoveAt(0);
    Assert.That(removed, Is.EqualTo("Peter"));
    Assert.That(names.ToString(), Is.EqualTo("[Maria, Steve, Mia]"));
}

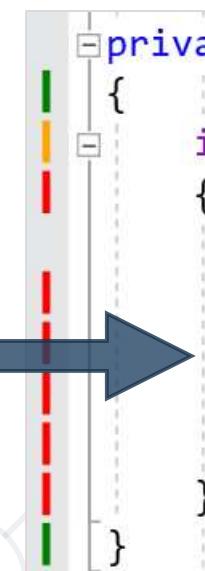
[Test]
public void Test_Collections_RemoveAtEnd()
{
    var names = new Collection<string>("Peter", "Maria", "Steve", "Mia");
    var removed = names.RemoveAt(3);
    Assert.That(removed, Is.EqualTo("Mia"));
    Assert.That(names.ToString(), Is.EqualTo("[Peter, Maria, Steve]"));
}

[Test]
public void Test_Collections_RemoveAtMiddle()
{
    var names = new Collection<string>("Peter", "Maria", "Steve", "Mia");
    var removed = names.RemoveAt(2);
    Assert.That(removed, Is.EqualTo("Steve"));
    Assert.That(names.ToString(), Is.EqualTo("[Peter, Maria, Mia]"));
}
```

# Testing Private Methods

- **Private methods** should be tested indirectly
  - By testing the **public methods** with certain inputs and entrance conditions, that will invoke the target private methods
  - Check the **code coverage** to ensure all code is tested!
- Example:

```
public void Add(T item)
{
    this.EnsureCapacity();
    this.items[this.Count] = item;
    this.Count++;
}
```

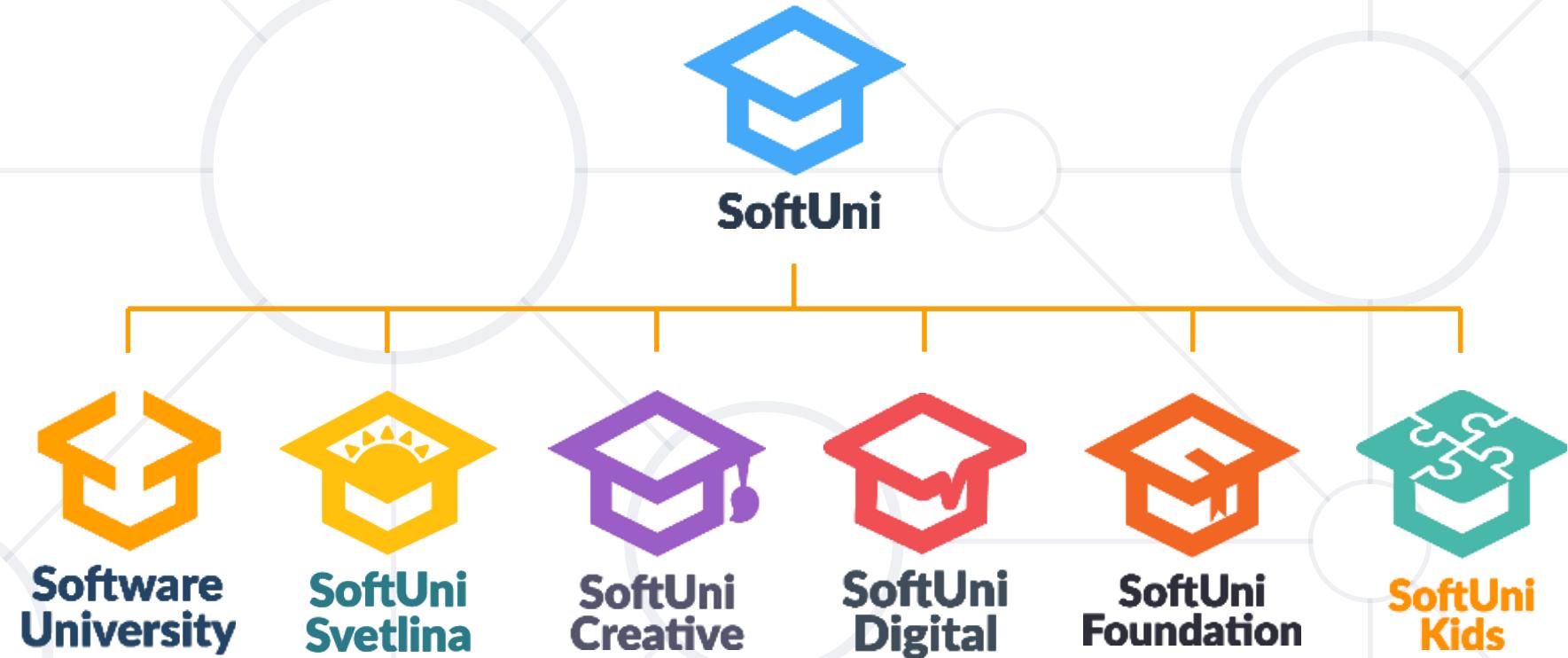


```
private void EnsureCapacity()
{
    if (this.Count == this.Capacity)
    {
        // Grow the capacity 2 times and move the items
        T[] oldItems = this.items;
        this.items = new T[2 * oldItems.Length];
        for (int i = 0; i < this.Count; i++)
            this.items[i] = oldItems[i];
    }
}
```

- **Unit testing** == automated testing of single component (unit)
- **Testing framework** == foundation for writing tests
- **NUnit** == automated testing framework for C#
- The **AAA pattern**: Arrange, Act, Assert



# Questions?



# SoftUni Diamond Partners

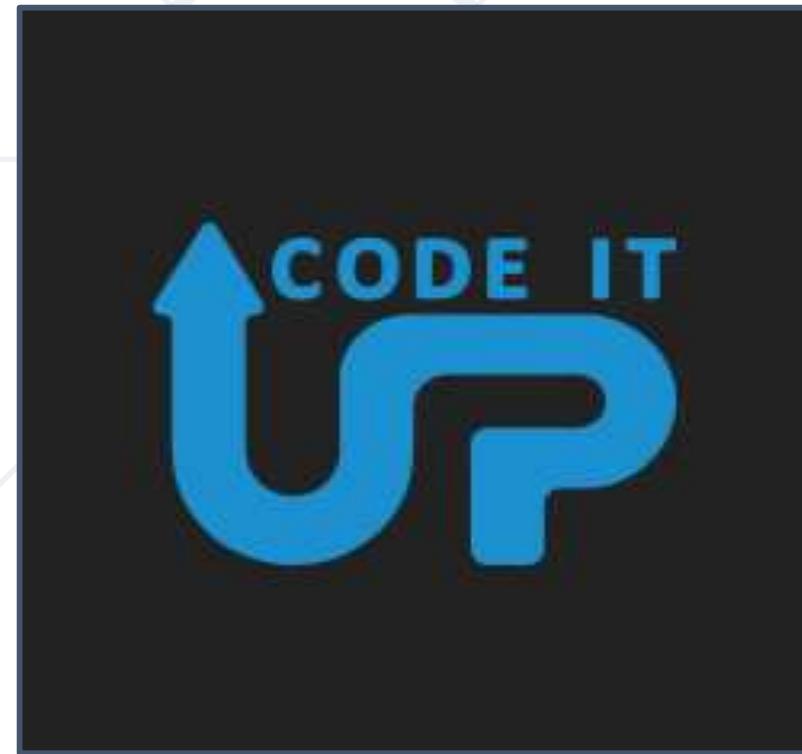


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University

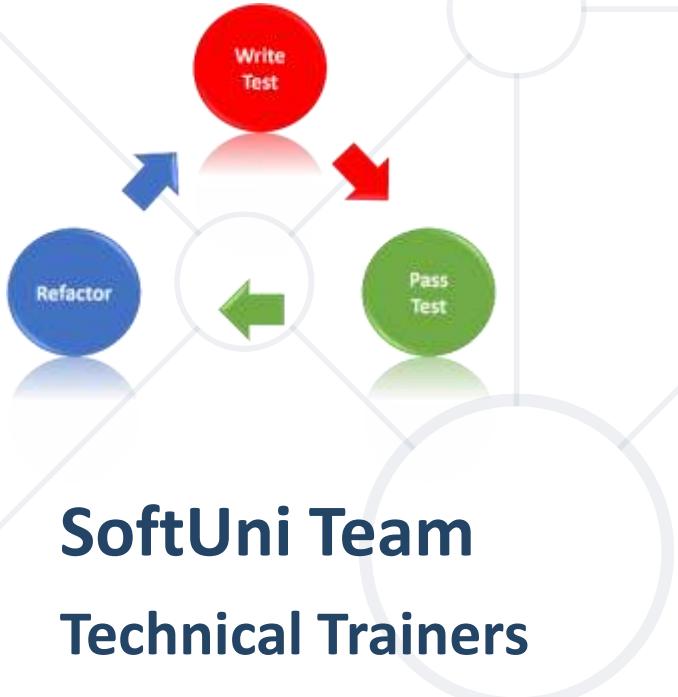


- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>



# Mocking and Test-Driven Development

Learn the "Test First" Approach to Coding



SoftUni Team

Technical Trainers



Software University

<https://softuni.bg>

# Table of Contents

- Isolating Behaviors
- Mocking
- Test-Driven Development
  - Reasons to use TDD
  - Myths and Misconceptions

Have a Question?



sli.do

**#csharp-advanced**



# Isolating Behaviors

Dependencies

# Coupling and Testing (1)

- Consider testing the following code:
  - We want to test a **single behavior**

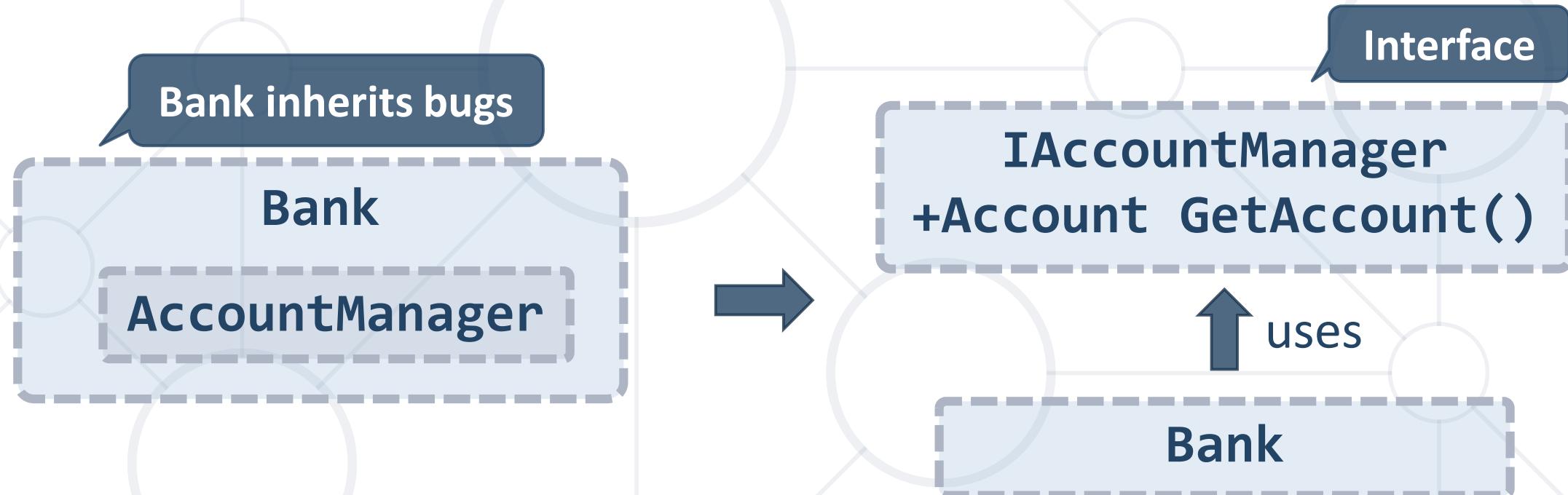
```
public class Bank {  
    private AccountManager accountManager;  
  
    public Bank()  
    {  
        this.AccountManager = new AccountManager();  
    }  
  
    public AccountInfo GetInfo(string id) { ... }  
}
```

Concrete  
Implementation

Bank depends on  
AccountManager

# Coupling and Testing (2)

- Need to find solution to **decouple classes**



# Dependency Injection

- Decouples classes and makes code testable

```
public interface IAccountManager
{
    Account Account { get; }
}

public class Bank
{
    private IAccountManager accountManager;
    public Bank(IAccountManager accountManager)
    {
        this.accountManager = accountManager;
    }
}
```

Independent from Implementation

Injecting dependencies

# Fake Implementations

- Not readable, cumbersome and has too much boilerplate

```
[Test]  
public void TestRequiresFakeImplementationOfBigInterface() {  
    // Arrange  
    Database db = new BankDatabase()  
    {  
        // Too many methods...  
    }  
  
    AccountManager manager = new AccountManager(db);  
    // Act...  
    // Assert...  
}
```

*Not suitable for  
big interfaces*

# Problem: Test GetGreeting

- Refactor **GetGreeting** to get the date time from outside
- Refactor **GetGreeting** to get the write location from outside
- Write tests to cover the **GetGreeting** method

```
public class GreetingProvider
{
    public string GetGreeting()
    {
        if (DateTime.Now.Hour < 12)
            Console.WriteLine("Good morning!");
        ...
    }
}
```



# Mocking

- Mock objects **simulate behavior** of real objects
  - The object supplies non-deterministic results - **Time**
  - It has states that are difficult to create or reproduce - **Network**
  - It is slow - **Database**
  - It does not yet exist or may change behavior
  - It would have to include information and methods exclusively for testing purposes (and not for its actual task)

- Moq provides us with an easy way of **creating mock objects**
  - Simple to use
  - Strongly typed
  - Minimalistic

```
Mock<.IContainer> mockContainer = new Mock<.IContainer>();  
Mock<ICustomerView> mockView = new Mock<ICustomerView>();
```

# Mocking Example

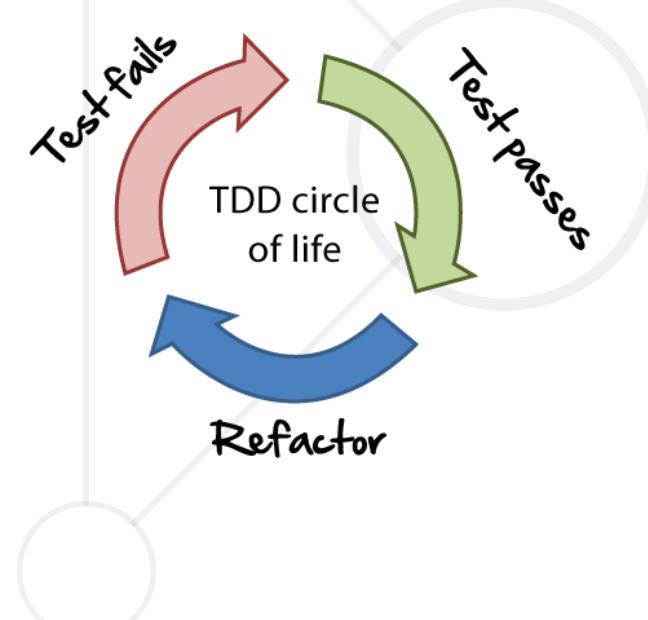
```
Mock<ITarget> fakeTarget = new Mock<ITarget>();  
  
fakeTarget  
.Setup(p => p.TakeAttack(It.IsAny<int>()))  
.Callback(() => hero.Weapon.DurabilityPoints -= 1);  
  
fakeTarget  
.Setup(p => p.Health)  
.Returns(0);
```



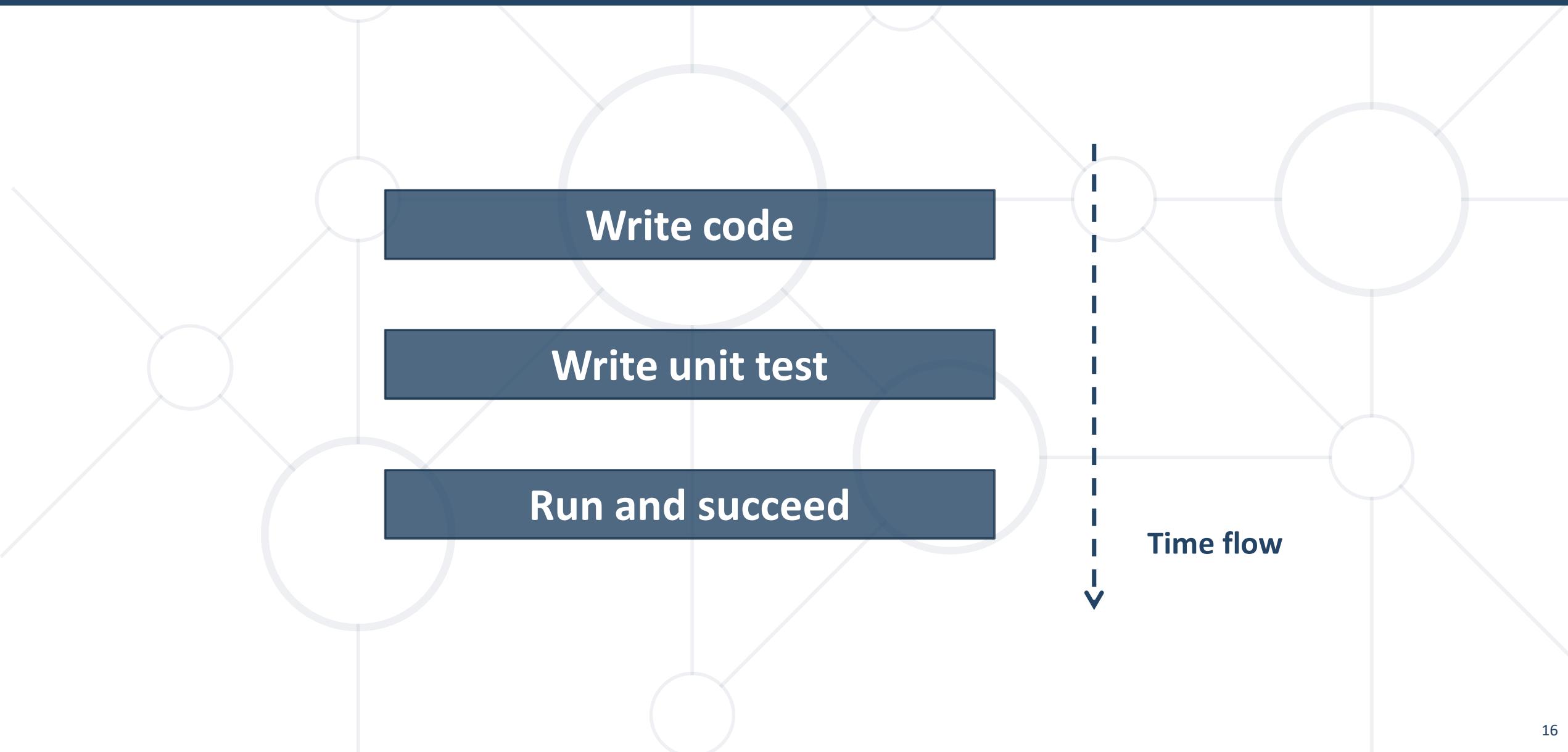
**Test-Driven Development**

# Unit Testing Approaches

- "Code First" (code and test) approach
  - Classical approach
- "Test First" approach
  - Test-driven development (TDD)



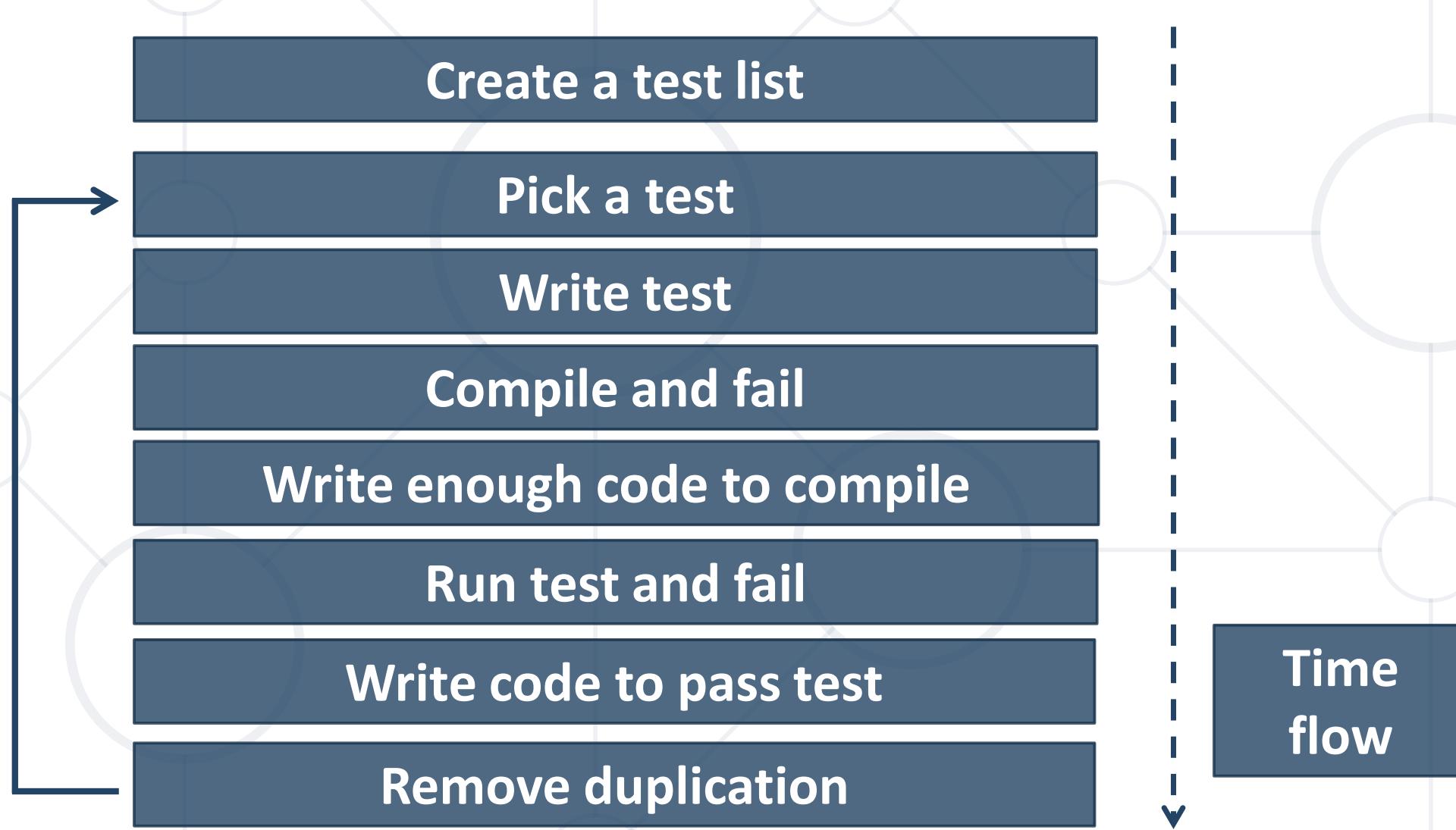
# The Code and Test Approach



# The Test-Driven Development Approach



# Test-Driven Development (TDD)



# Why TDD?

- **TDD** helps find design issues early
  - Avoids reworking
  - Writing code to satisfy a test is a focused activity
    - **Less** chance of **error**
- **Tests** will be more **comprehensive** than if they are written after the code



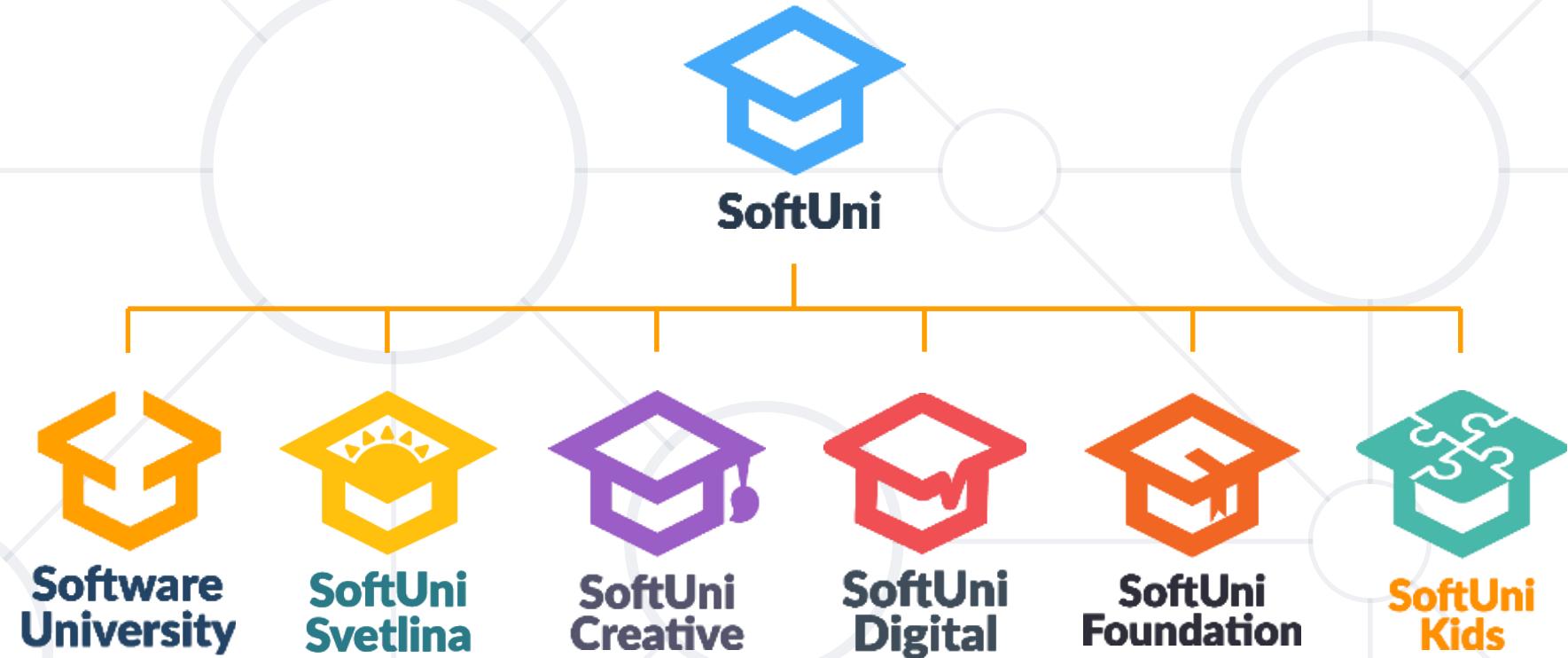
# Myths and Misconceptions

- You create a 100% regression test suite
- The unit tests form 100% of your design specification
- You only need to unit test
- TDD is sufficient for testing
- TDD doesn't scale (partially true)
  - Your test suite takes too long to run
  - Not all developers know how to test
  - Everyone might not be taking a TDD approach

- Code and Test
  - Write code, then test it
- Test-Driven Development
  - Write tests first
- Reasons to use TDD
  - Prevent some application design flaws
  - Manage complexity more easily



# Questions?



# SoftUni Diamond Partners

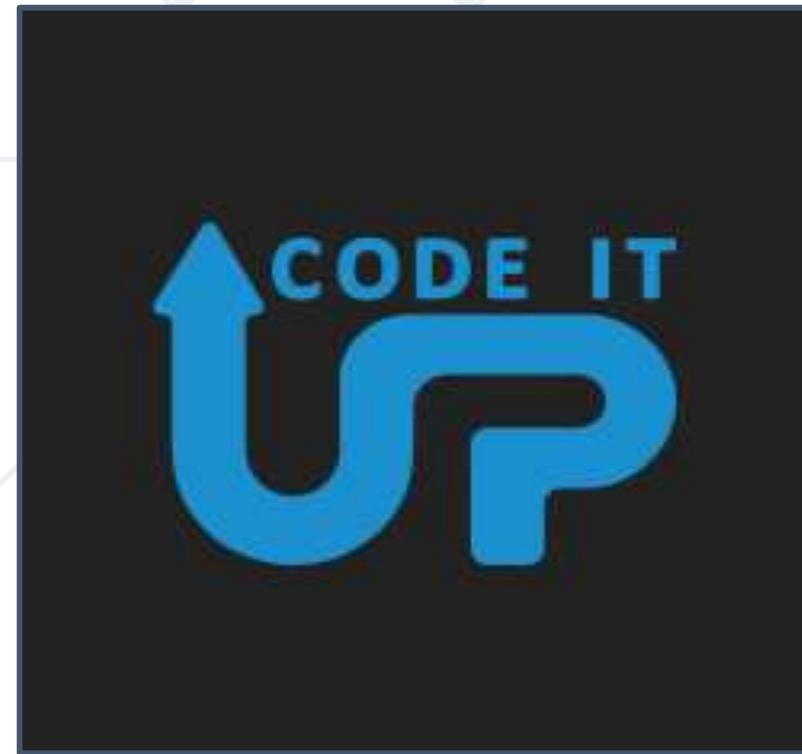


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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



# Design Patterns



SoftUni Team

Technical Trainers



SoftUni



Software University  
<https://softuni.bg>

# Table of Contents

- Definition of Design Patterns
- Benefits and Drawbacks
- Types of Design Patterns
  - Creational
  - Structural
  - Behavioral

Have a Question?



sli.do

**#csharp-advanced**



# **Definition, Solutions and Elements**

**Design Patterns**

# What Are Design Patterns?

- General and reusable solutions to common problems in software design
- A template for solving given problems
- Add additional layers of abstraction in order to reach flexibility



# What Do Design Patterns Solve?

- Patterns solve **software structural problems** like:
  - Abstraction
  - Encapsulation
  - Separation of concerns
  - Coupling and cohesion
  - Separation of interface and implementation



# Elements of a Design Pattern

- Pattern name - Increases **vocabulary** of designers
- Problem - **Intent**, context and when to apply
- Solution - **Abstract** code
- Consequences - **Results** and trade-offs





# Benefits and Drawbacks

Why Design Patterns?

- Names form a common vocabulary
- Enable large-scale **reuse** of software architectures
- Help improve developer **communication**
- Can **speed-up** the development

# Drawbacks

- Do not lead to a direct code reuse
- Deceptively simple
- Developers may suffer from **pattern overload** and **overdesign**
- Validated by **experience** and discussion, not by automated testing
- Should be used only if **understood well**



# Types of Design Patterns

# Main Types

- Creational patterns
  - Deal with **initialization and configuration** of classes and objects
- Structural patterns
  - Describe ways to **assemble** objects to implement **new functionality**
  - **Composition** of classes and objects
- Behavioral patterns
  - Deal with dynamic **interactions** among societies of classes
  - Distribute **responsibility**



# Creational Patterns

# Purposes

- Deal with **object creation** mechanisms
- Trying to create objects in a **manner suitable to the situation**
- Two main ideas
  - **Encapsulating** knowledge about which classes the system uses
  - **Hiding** how instances of these classes are created



# List of Creational Patterns

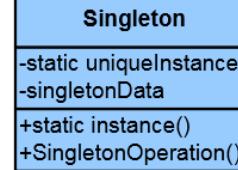
- Singleton
- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Fluent Interface
- Object Pool
- Lazy Initialization

## Singleton

Type: Creational

### What it is:

Ensure a class only has one instance and provide a global point of access to it.

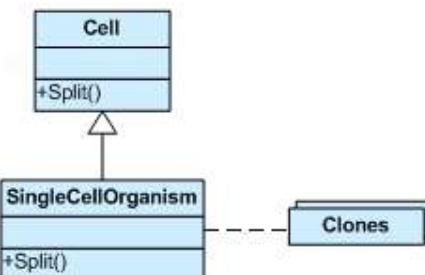
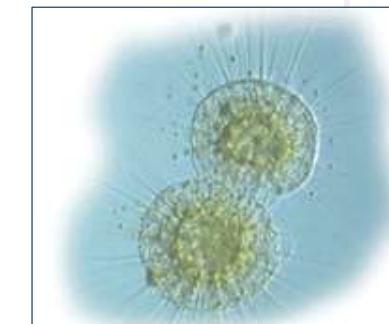
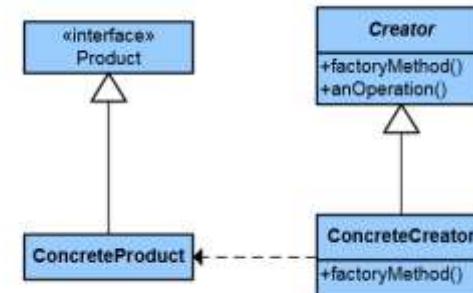


## Factory Method

Type: Creational

### What it is:

Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

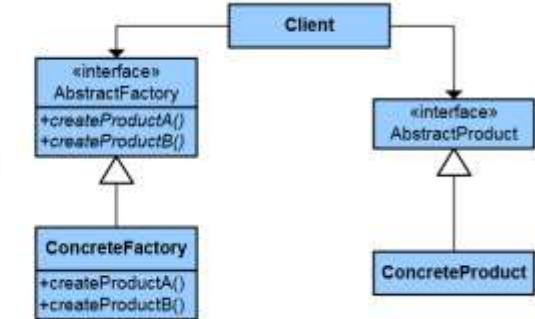


## Abstract Factory

Type: Creational

### What it is:

Provides an interface for creating families of related or dependent objects without specifying their concrete class.

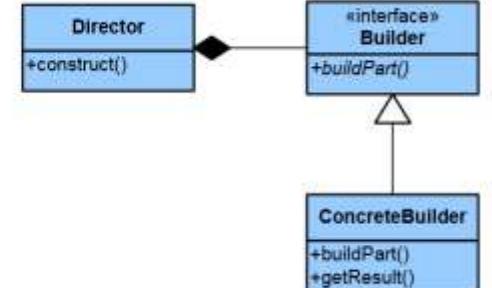


## Builder

Type: Creational

### What it is:

Separate the construction of a complex object from its representation so that the same construction process can create different representations.



# Singleton Pattern

- The most often used creational design pattern
- A Singleton class is supposed to have **only one instance**
- It is **not a global variable**
- Possible problems
  - Lazy loading
  - Thread-safe

## Singleton

Type: Creational

### What it is:

Ensure a class only has one instance and provide a global point of access to it.

## Singleton

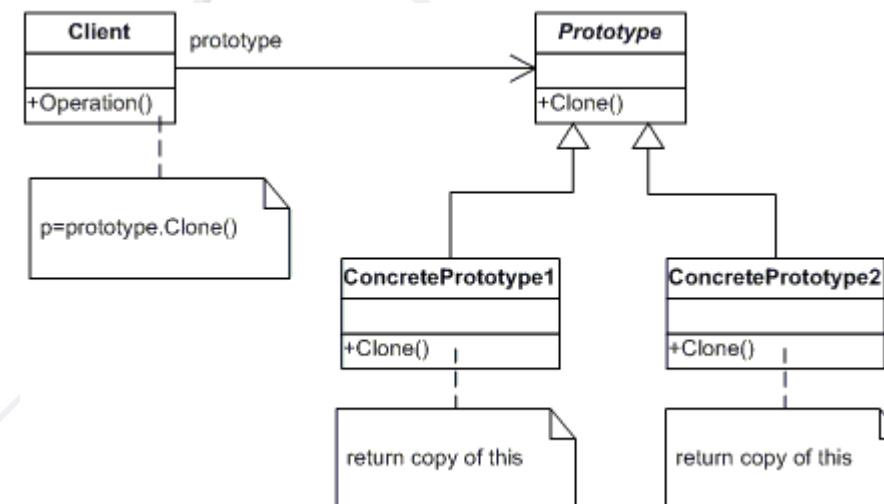
```
-static uniqueInstance  
-singletonData  
+static instance()  
+SingletonOperation()
```

# Double-Check Singleton Example

```
public sealed class Singleton {  
    private static Singleton instance;  
    private static readonly object padlock = new object();  
    private Singleton() { }  
    public static Singleton Instance {  
        get {  
            if (instance == null) {  
                lock (padlock) {  
                    if (instance == null)  
                        instance = new Singleton(); } }  
            return instance; } } }
```

# Prototype Pattern

- Factory for **cloning** new instances from a prototype
  - Create new objects by copying this prototype
  - Instead of using the "new" keyword
- **ICloneable** interface acts as Prototype



# The Prototype Abstract Class

```
abstract class Prototype {  
    private string _id;  
  
    public Prototype(string id) {  
        this._id = id; }  
  
    public string Id => this._id;  
  
    public abstract Prototype Clone();  
}
```

# A Concrete Prototype Class

```
class ConcretePrototype : Prototype
{
    public ConcretePrototype(string id) : base(id) { }

    public override Prototype Clone()
        => return (Prototype)this.MemberwiseClone();
}
```



# Structural Patterns

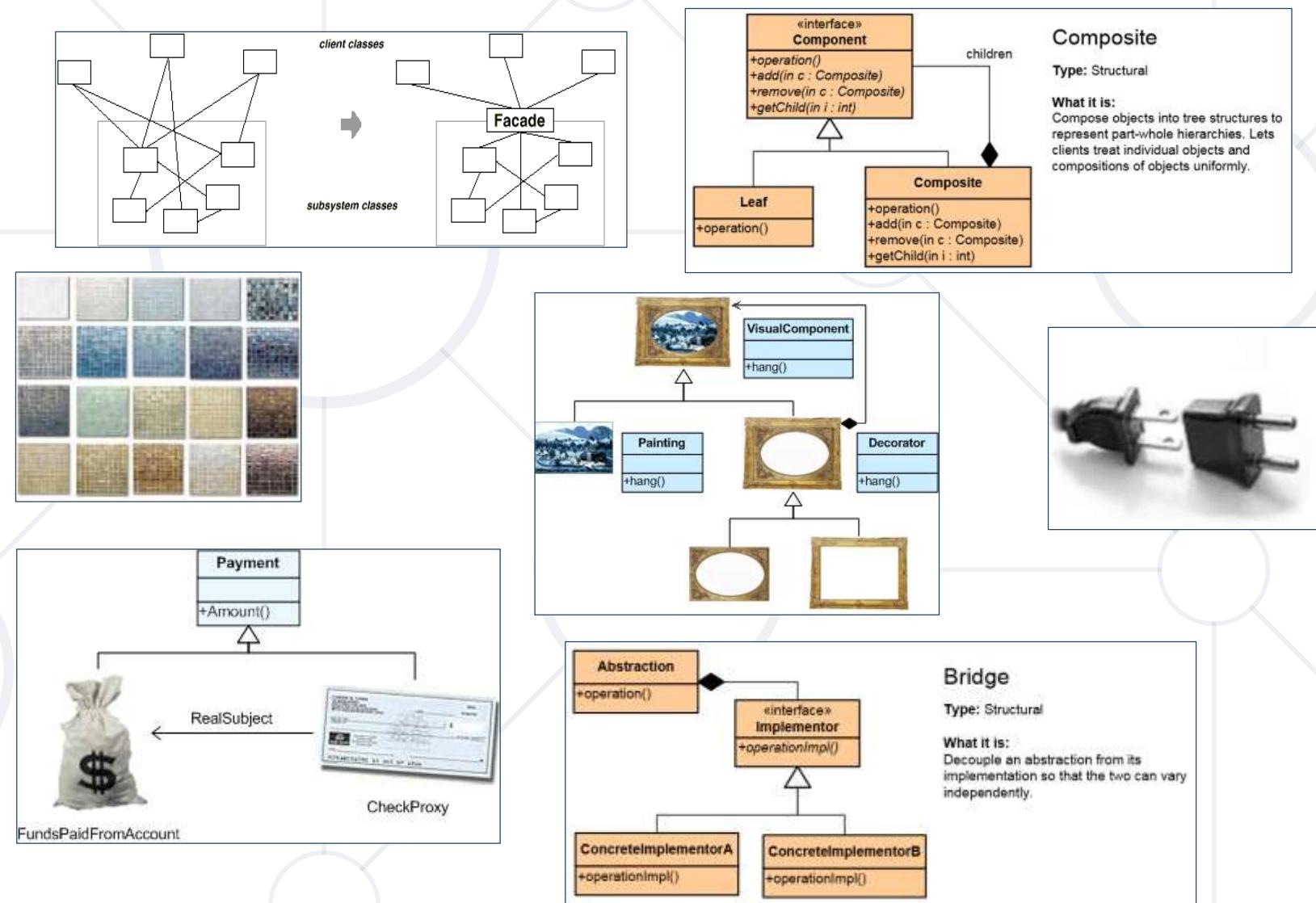
# Purposes

- Describe ways to assemble **objects** to implement a **new functionality**
- Ease the design by identifying a simple way to realize **relationship** between entities
- All about Class and Object composition
  - **Inheritance** to compose interfaces
  - Ways to compose objects to obtain **new functionality**



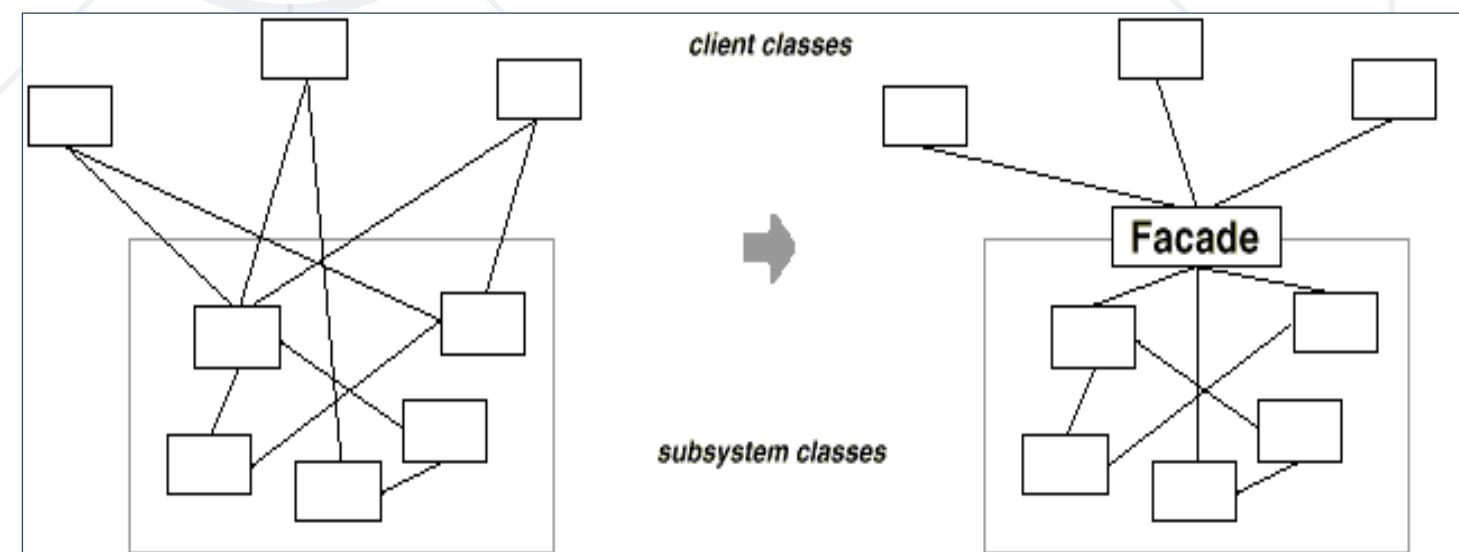
# List of Structural Patterns

- Facade
- Composite
- Flyweight
- Proxy
- Decorator
- Adapter
- Bridge



# Facade Pattern

- Provides a **unified interface** to a set of interfaces in a subsystem
- Defines a **higher-level interface** that makes the subsystem easier to use



# The Facade Class (1)

```
class Facade
{
    private SubSystemOne _one;
    private SubSystemTwo _two;

    public Facade()
    {
        _one = new SubSystemOne();
        _two = new SubSystemTwo();
    }
}
```

# The Facade Class (2)

```
public void MethodA() {  
    Console.WriteLine("\nMethodA() ----- ");  
    _one.MethodOne();  
    _two.MethodTwo(); }  
  
public void MethodB() {  
    Console.WriteLine("\nMethodB() ----- ");  
    _two.MethodTwo(); }  
}
```

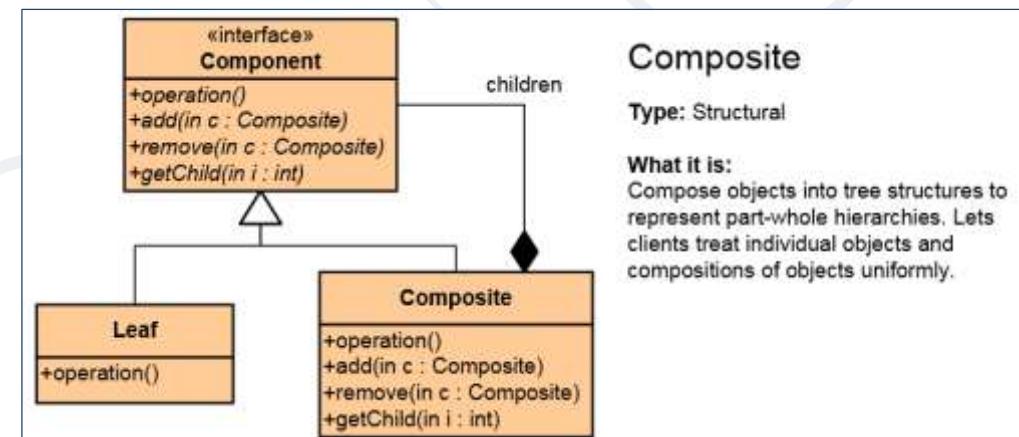
# Subsystem Classes

```
class SubSystemOne
{
    public void MethodOne()
        => Console.WriteLine(" SubSystemOne Method");
}
```

```
class SubSystemTwo
{
    public void MethodTwo()
        => Console.WriteLine(" SubSystemTwo Method");
}
```

# Composite Pattern

- Allows to **combine** different types of objects in tree structures
- Gives the possibility to treat the **same object(s)**
- Used when
  - You have different objects that you want to **treat the same way**
  - You want to present **hierarchy** of objects



# The Component Abstract Class

```
abstract class Component {  
    protected string name;  
  
    public Component(string name) {  
        this.name = name; }  
  
    public abstract void Add(Component c);  
    public abstract void Remove(Component c);  
    public abstract void Display(int depth);  
}
```

# The Composite Class (1)

```
class Composite : Component {  
    private List<Component> _children = new List<Component>();  
  
    public Composite(string name) : base(name) { }  
  
    public override void Add(Component component)  
    => _children.Add(component);  
  
    public override void Remove(Component component)  
    => _children.Remove(component);
```

# The Composite Class (2)

```
public override void Display(int depth)
{
    Console.WriteLine(new String('-', depth) + name);

    foreach (Component component in _children)
    {
        component.Display(depth + 2);
    }
}
```

# The Leaf Class

```
class Leaf : Component {  
    public Leaf(string name) : base(name) { }  
  
    public override void Add(Component c)  
        => Console.WriteLine("Cannot add to a leaf");  
    public override void Remove(Component c)  
        => Console.WriteLine("Cannot remove from a leaf");  
    public override void Display(int depth)  
        => Console.WriteLine(new String('-', depth) + name);  
}
```



# Behavioral Patterns

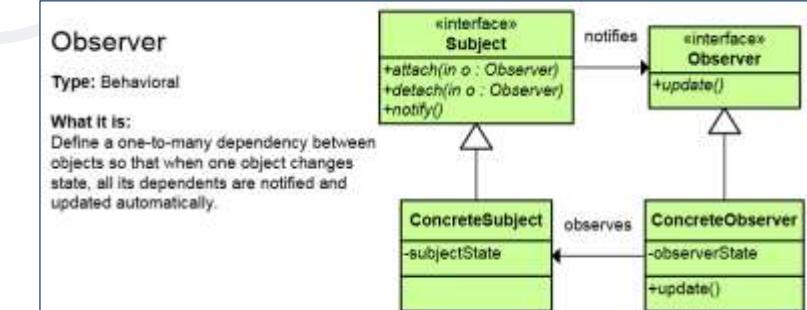
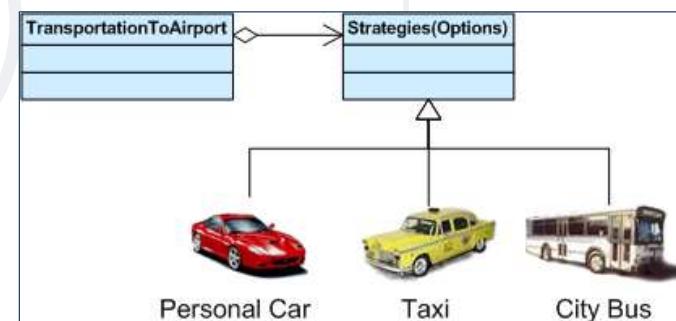
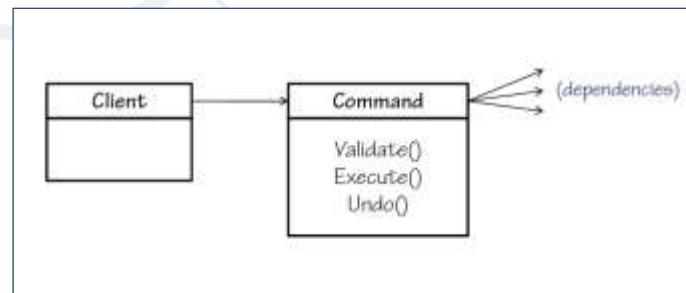
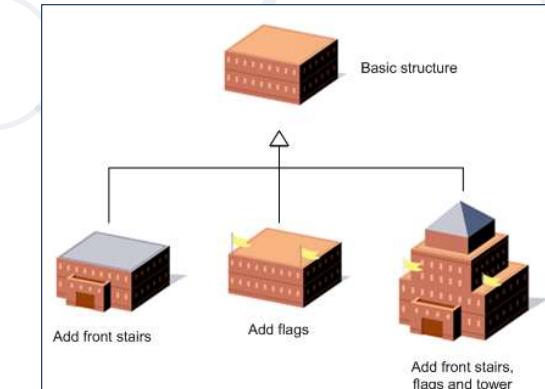
# Purposes

- Concerned with **interaction** between objects
  - Either with the **assignment of responsibilities** between objects
  - Or **encapsulating behavior** in an object and delegating requests to it
- Increases **flexibility** in carrying out cross-classes communication



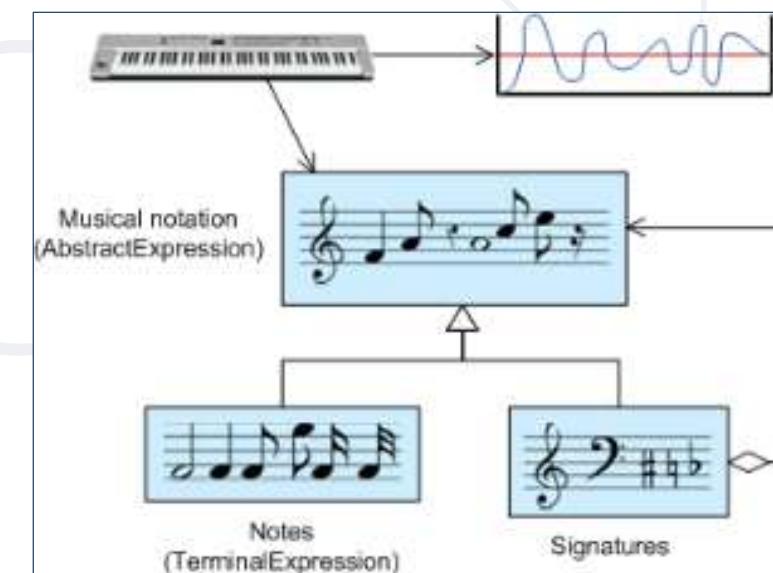
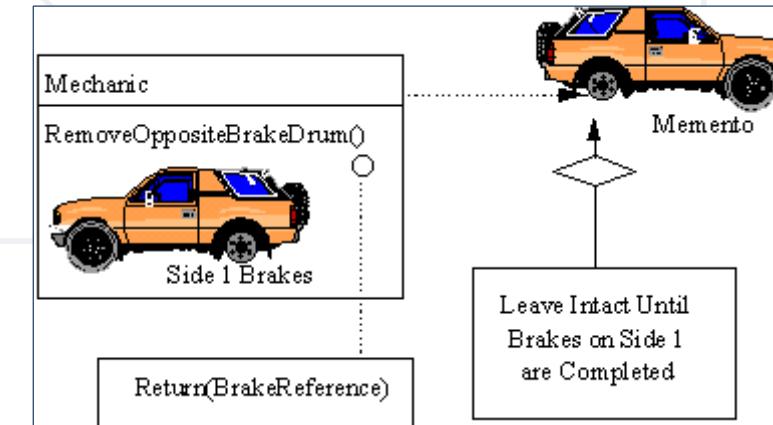
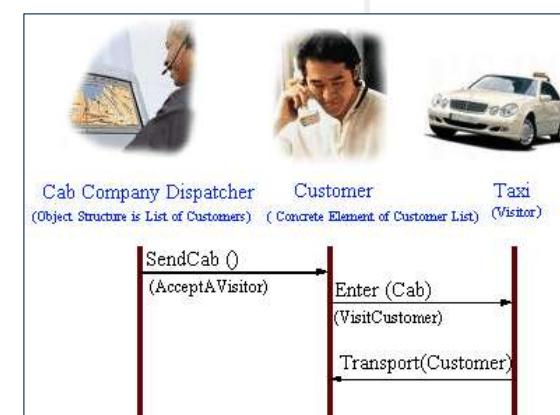
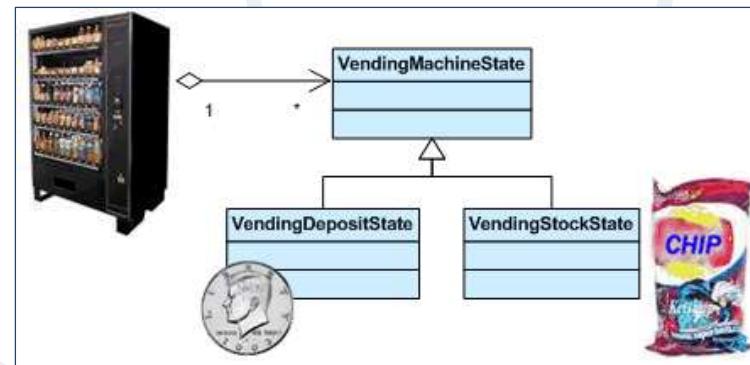
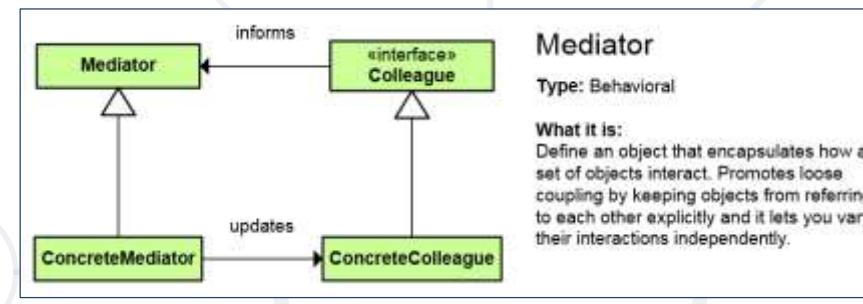
# List of Behavioral Patterns (1)

- Chain of Responsibility
- Iterator
- Command
- Template Method
- Strategy
- Observer



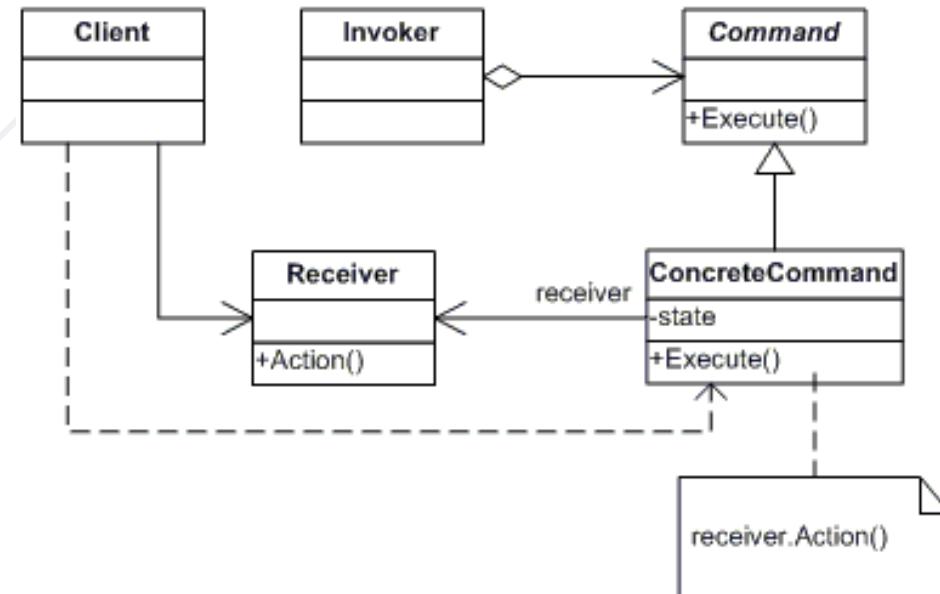
# List of Behavioral Patterns (2)

- Mediator
- Memento
- State
- Interpreter
- Visitor



# Command Pattern

- An object **encapsulates** all the information needed to call a method at a later time
- Lets you **parameterize** clients with different requests, queue or log requests, and support undoable operations



# The Command Abstract Class

```
abstract class Command
{
    protected Receiver receiver;

    public Command(Receiver receiver) {
        this.receiver = receiver; }

    public abstract void Execute();
}
```

# Concrete Command Class

```
class ConcreteCommand : Command
{
    public ConcreteCommand(Receiver receiver)
        : base(receiver) { }

    public override void Execute()
        => receiver.Action();
}
```

# The Receiver Class

```
class Receiver
{
    public void Action()
    {
        Console.WriteLine("Called Receiver.Action()");
    }
}
```

# The Invoker Class

```
class Invoker
{
    private Command _command;

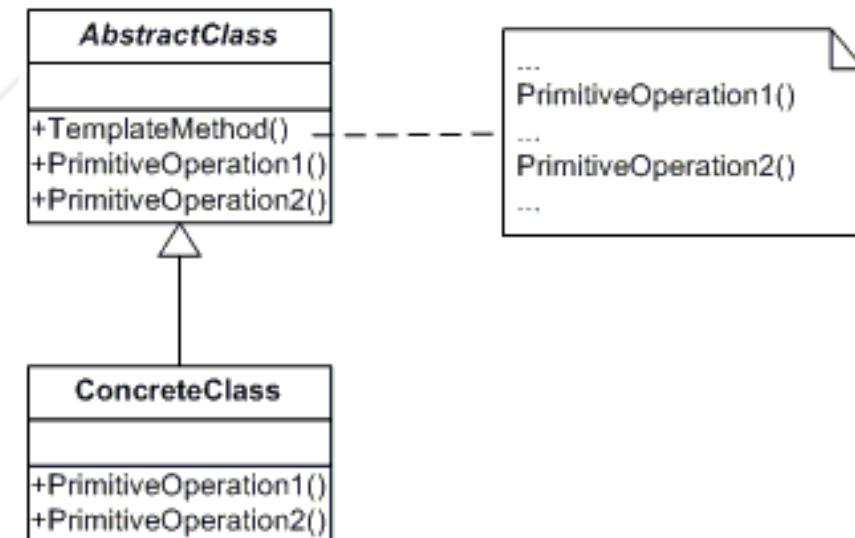
    public void SetCommand(Command command)
        => this._command = command;

    public void ExecuteCommand()
        => _command.Execute();

}
```

# Template Method Pattern

- Define the **skeleton** of an algorithm in a method, leaving some implementation to its subclasses
- Allows the subclasses to **redefine** the implementation of some of the **parts** of the algorithm, but not its structure



# The Abstract Class

```
abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMethod() {
        PrimitiveOperation1();
        PrimitiveOperation2();
        Console.WriteLine("");
    }
}
```

# A Concrete Class

```
class ConcreteClassA : AbstractClass
{
    public override void PrimitiveOperation1()
        => Console.WriteLine("ConcreteClassA.
            PrimitiveOperation1()");

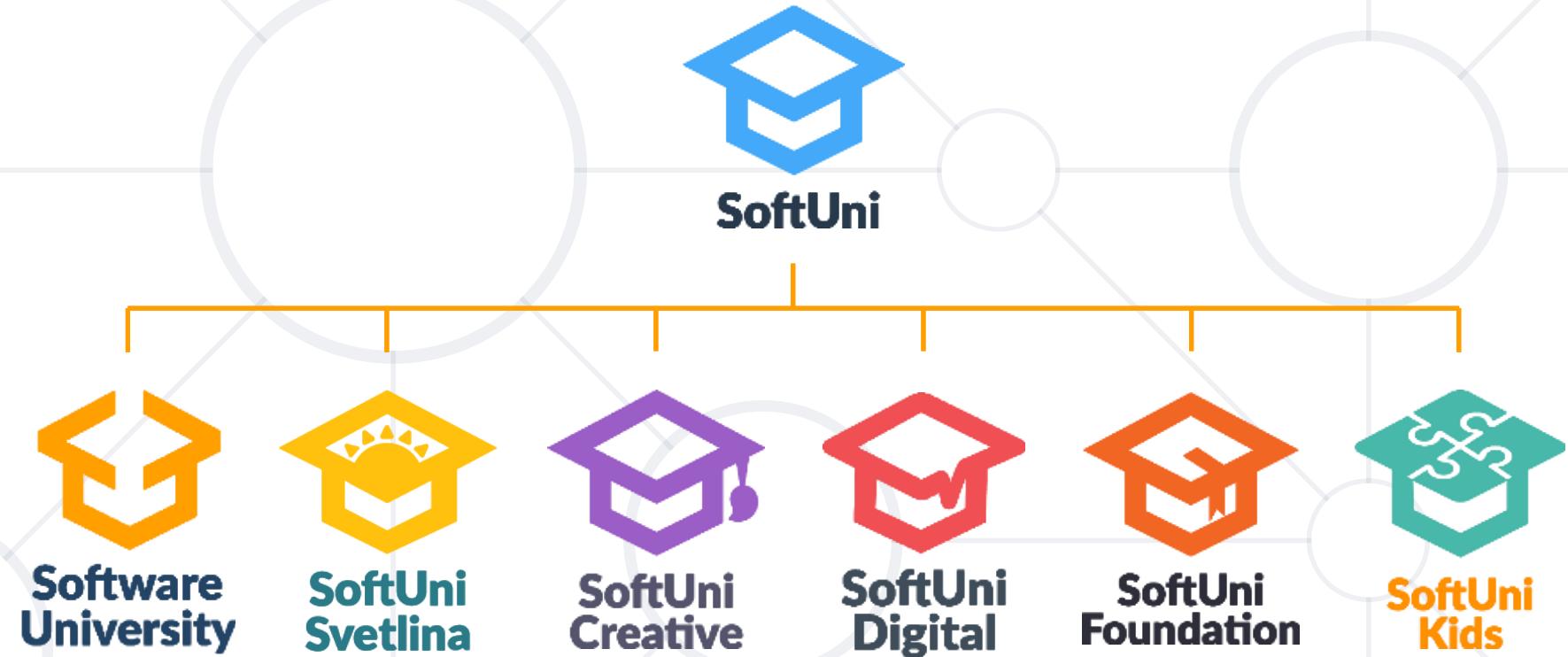
    public override void PrimitiveOperation2()
        => Console.WriteLine("ConcreteClassA
            .PrimitiveOperation2()");

}
```

- Design Patterns
  - Provide solution to common problems
  - Add additional layers of abstraction
- Three main types of Design Patterns
  - Creational
  - Structural
  - Behavioral



# Questions?



# SoftUni Diamond Partners

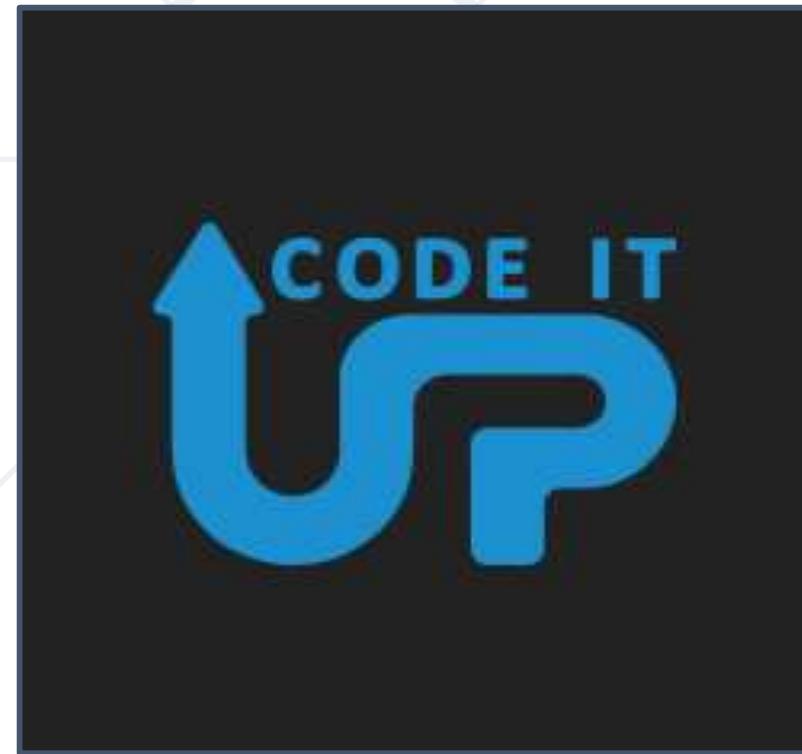


SCHWARZ



Bosch.IO

# Educational Partners



# Trainings @ Software University (SoftUni)



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University Forums
  - [forum.softuni.bg](http://forum.softuni.bg)



Software  
University



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

