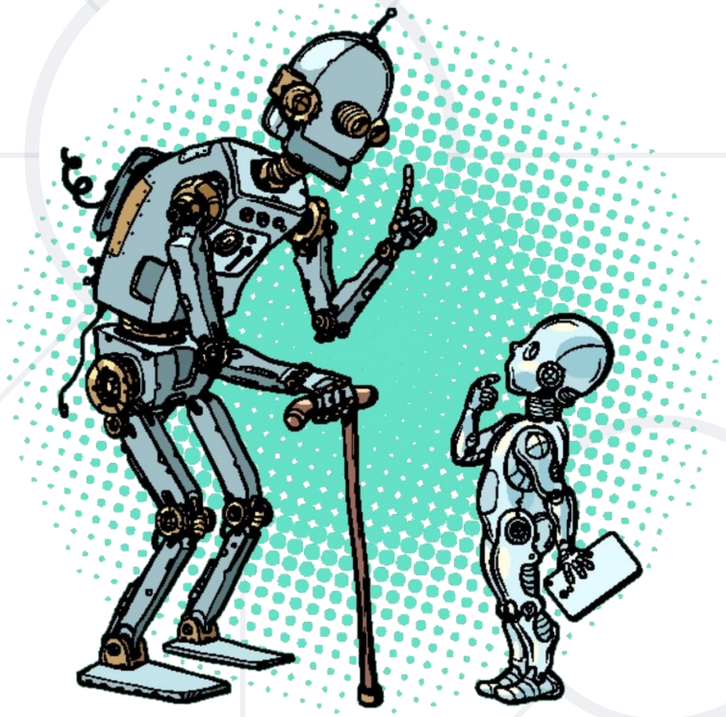
Inheritance

Class Hierarchies



SoftUni Team

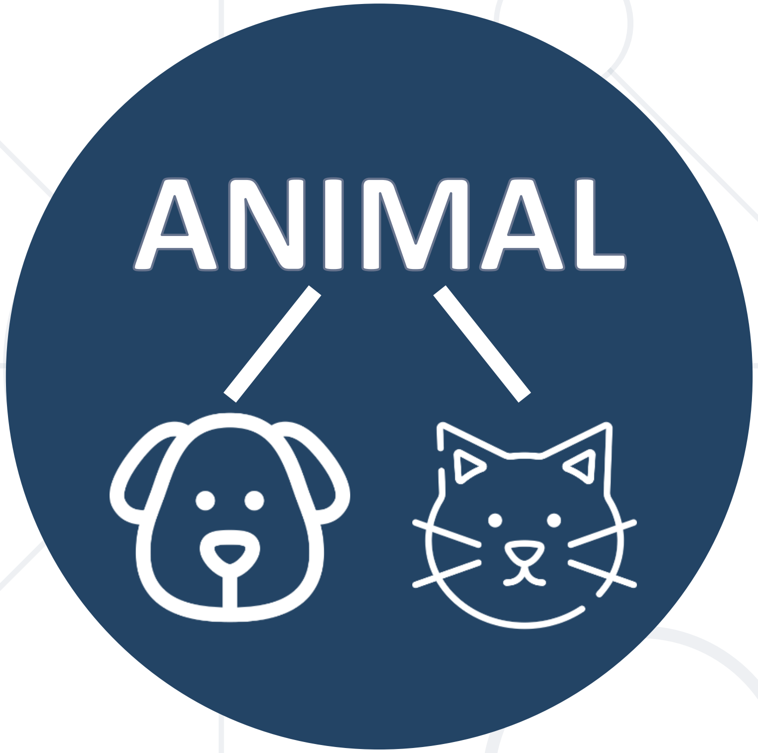
Technical Trainers

Software University

Software University

https://about.softuni.bg/

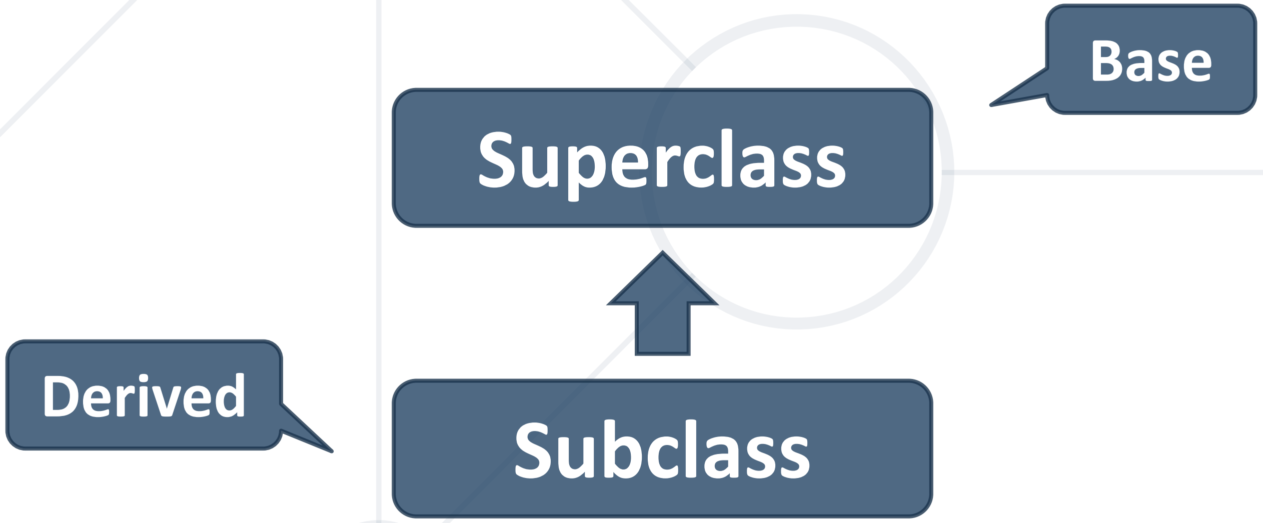
Table of Contents



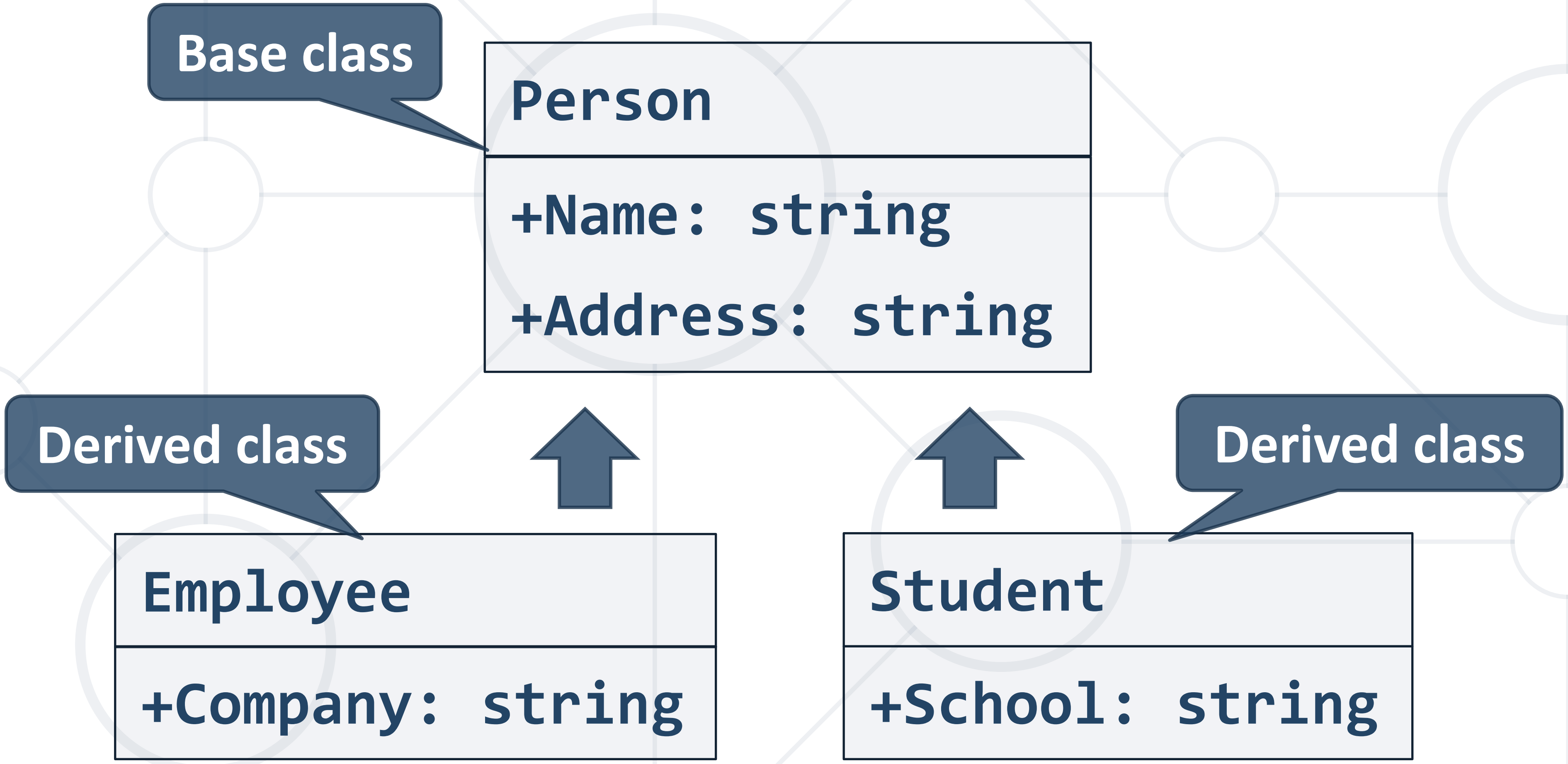
Extending Classes

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



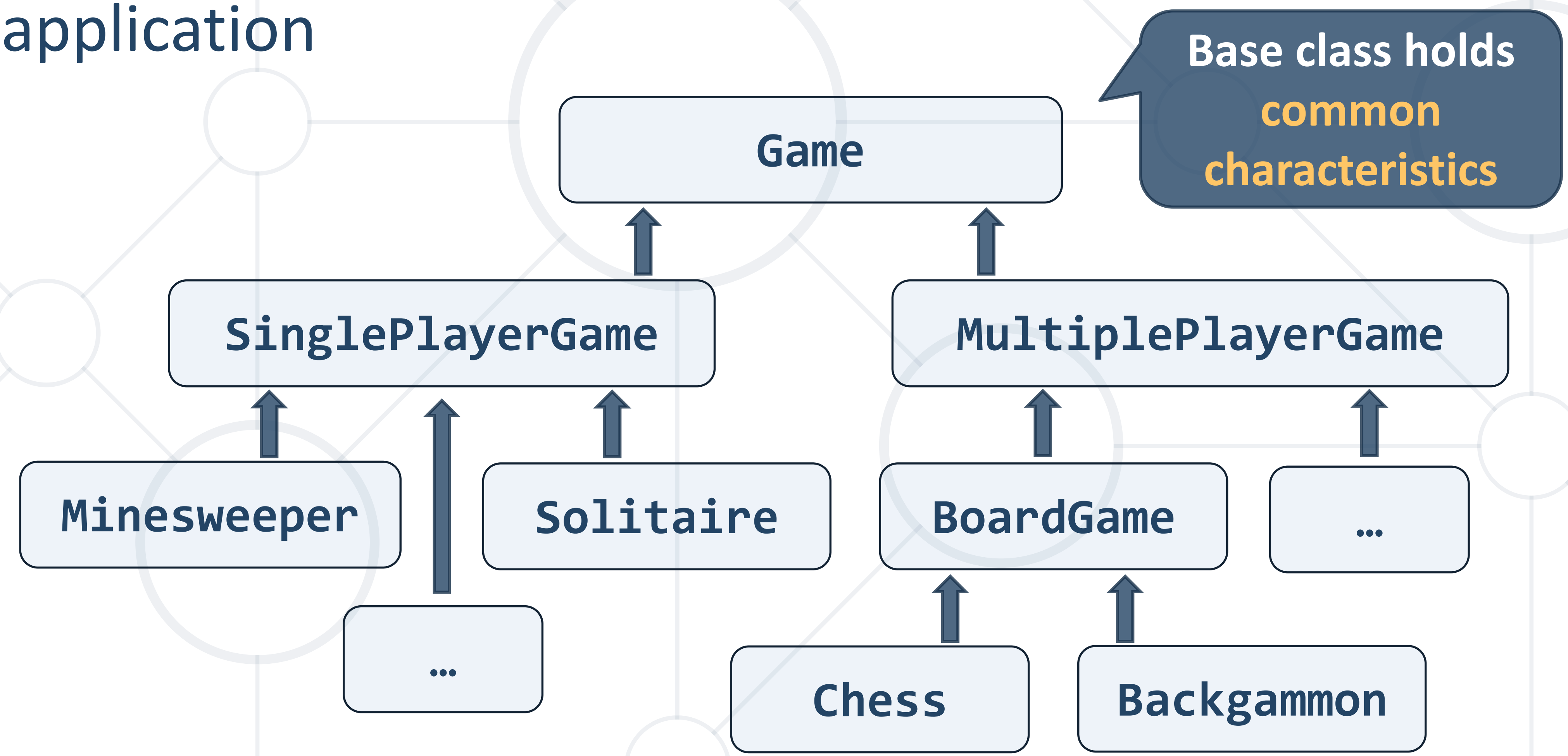
Inheritance – Example



Inheritance Leads to Hierarchies

Class Hierarchies

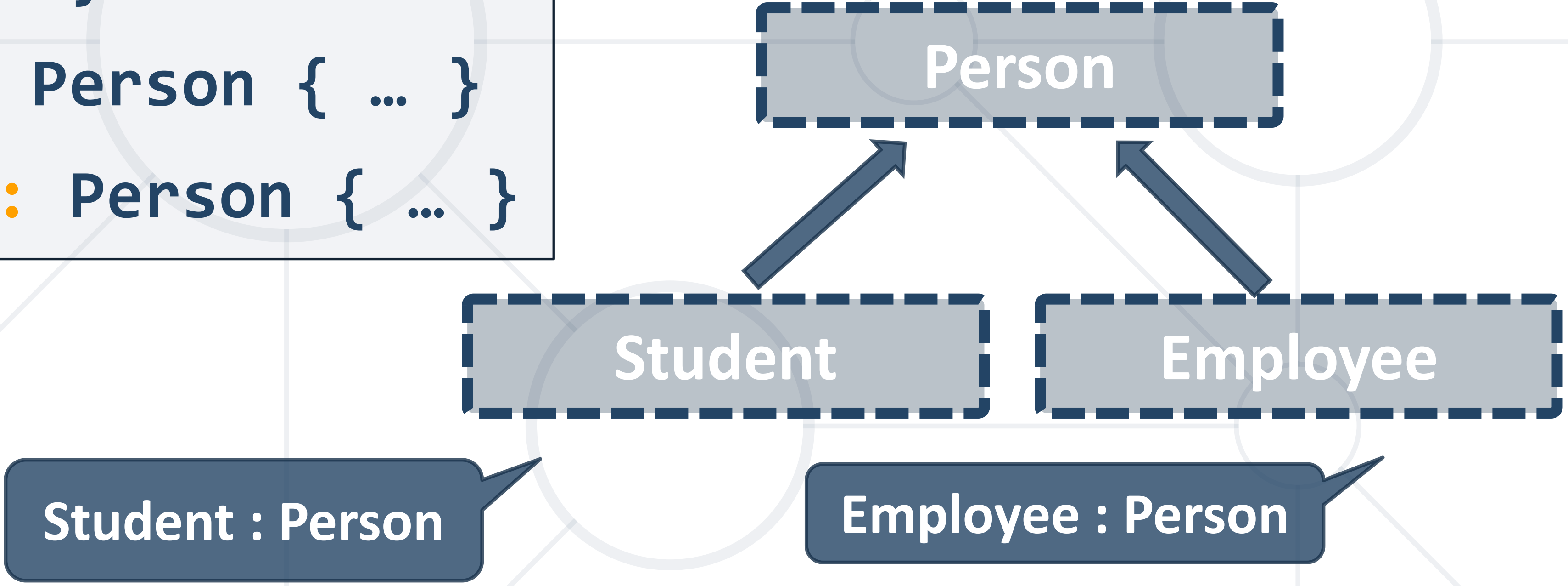
- **Inheritance** leads to **hierarchies** of classes and/or interfaces in an application



Inheritance in C#

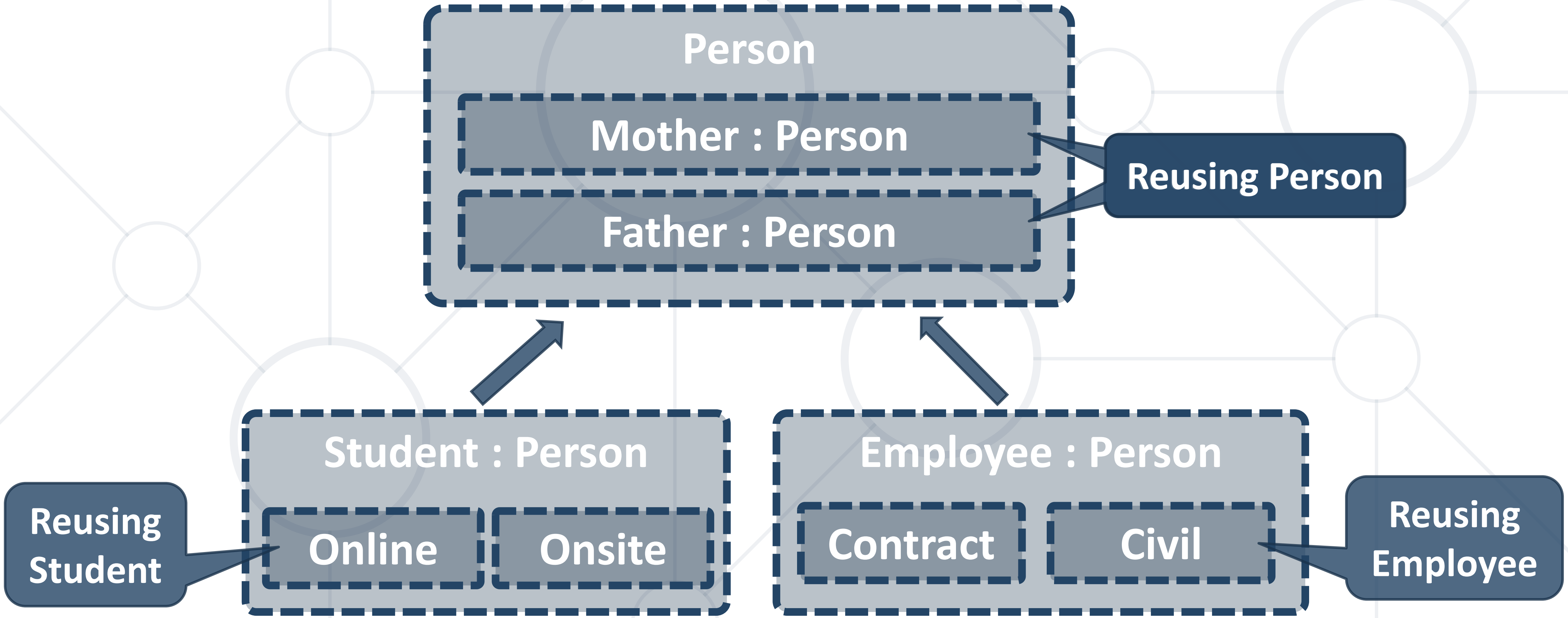
- In C# inheritance is defined by the : operator

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



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

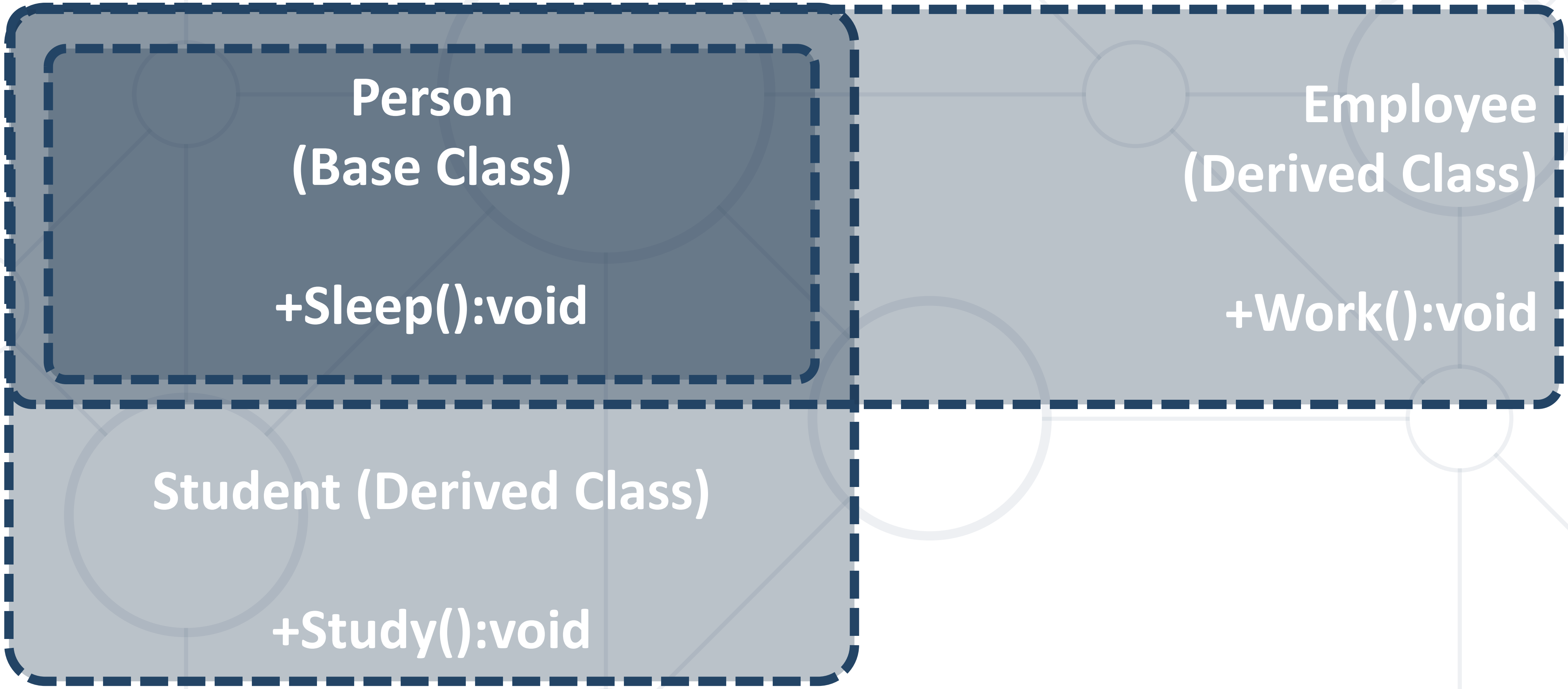
- Constructors are **not inherited**
- They can be **reused** by the child classes

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

Call the base (parent) constructor

Thinking about Inheritance – Extends

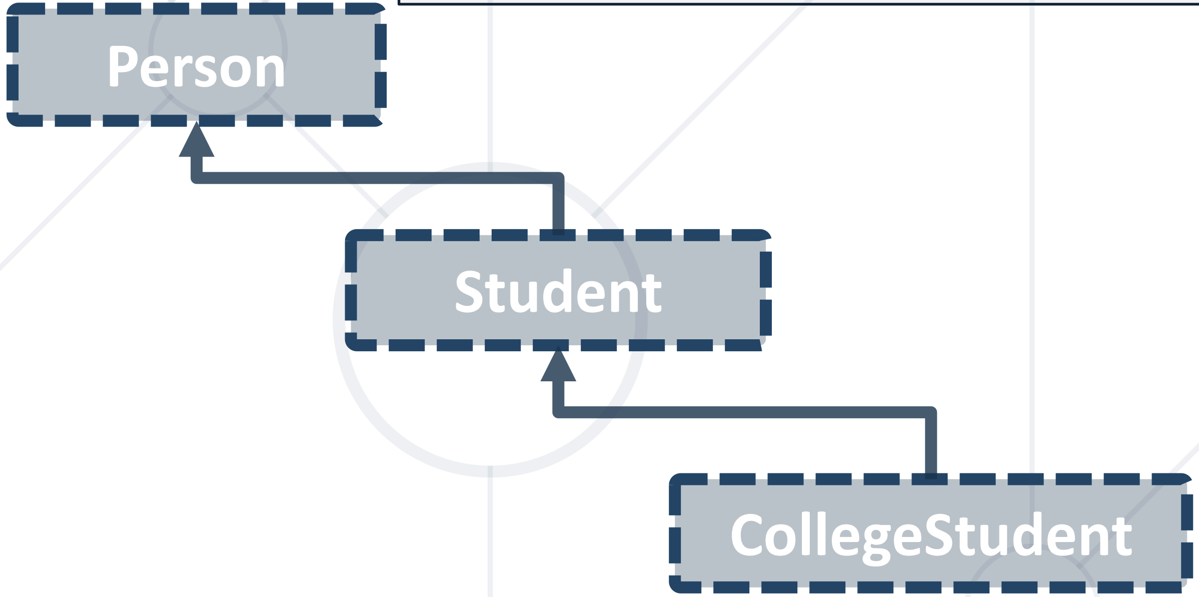
- Derived class instance **contains** instance of its base class



Transitive Relation

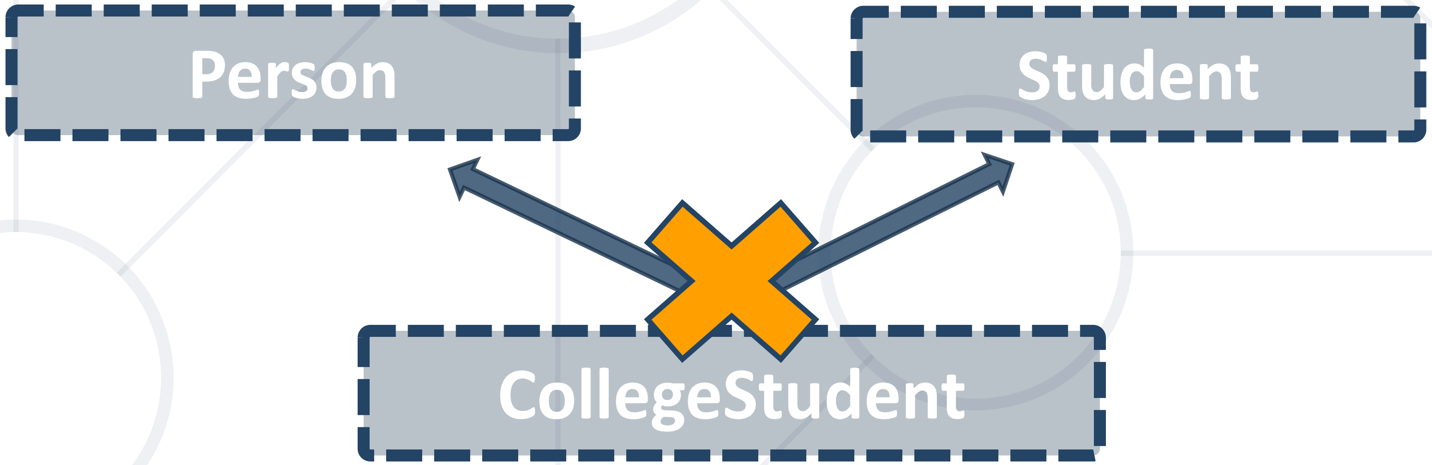
- Inheritance has a **transitive relation**

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



Multiple Inheritance

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



The Base Keyword

Access to Base Class Members

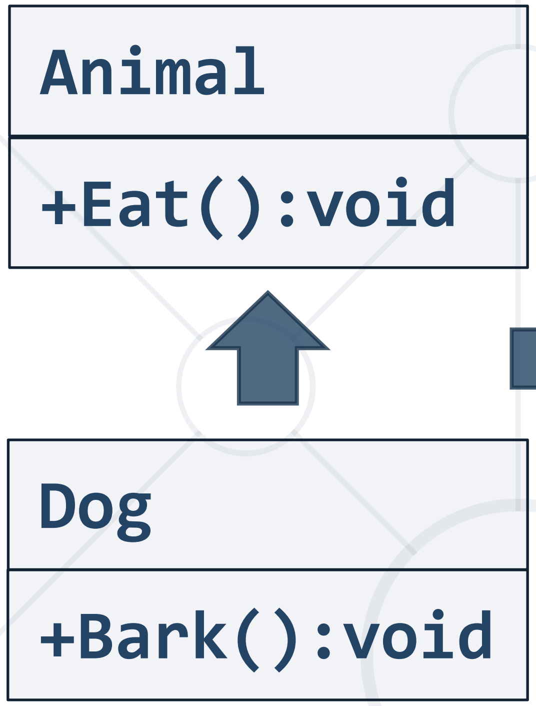
- Use the **base** keyword

```
class Person { … }

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

Problem: Dog Inherits Animal

- Create two classes: **Animal** and **Dog**:



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

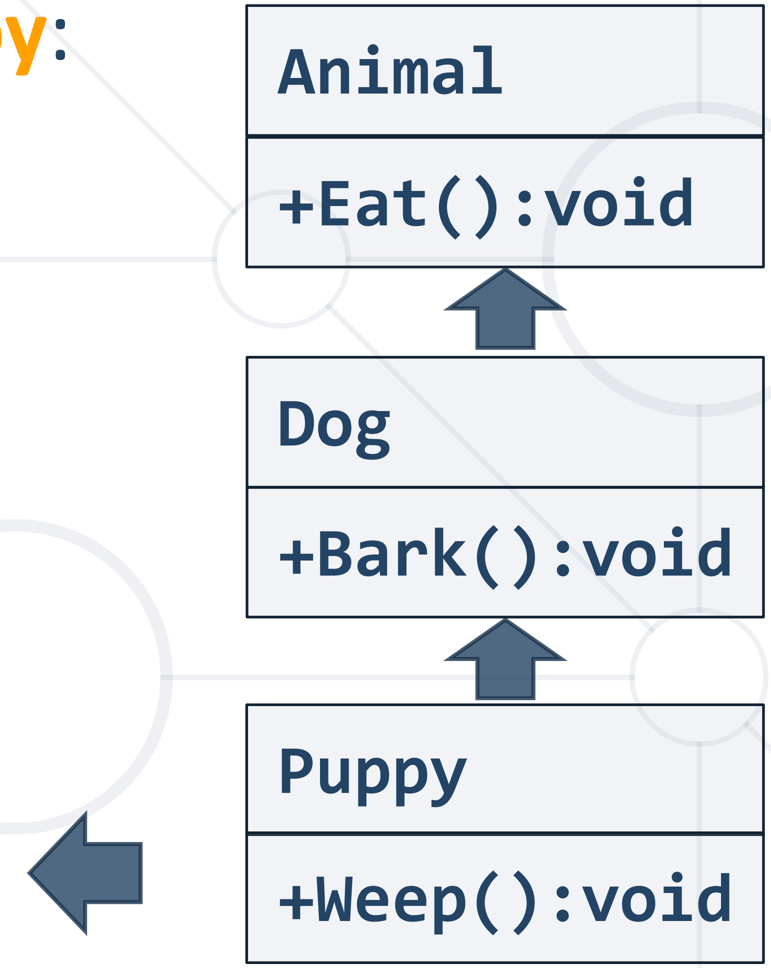
- **Animal** with method **Eat()** that prints: **"eating…"**
- **Dog** with method **Bark()** that prints: **"barking…"**
- **Dog** should inherit from **Animal**

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3164#0

Problem: Inheritance Chain

- Create classes: **Animal**, **Dog** and **Puppy**:
- **Dog** should inherit from **Animal**
- **Puppy** should inherit from **Dog**

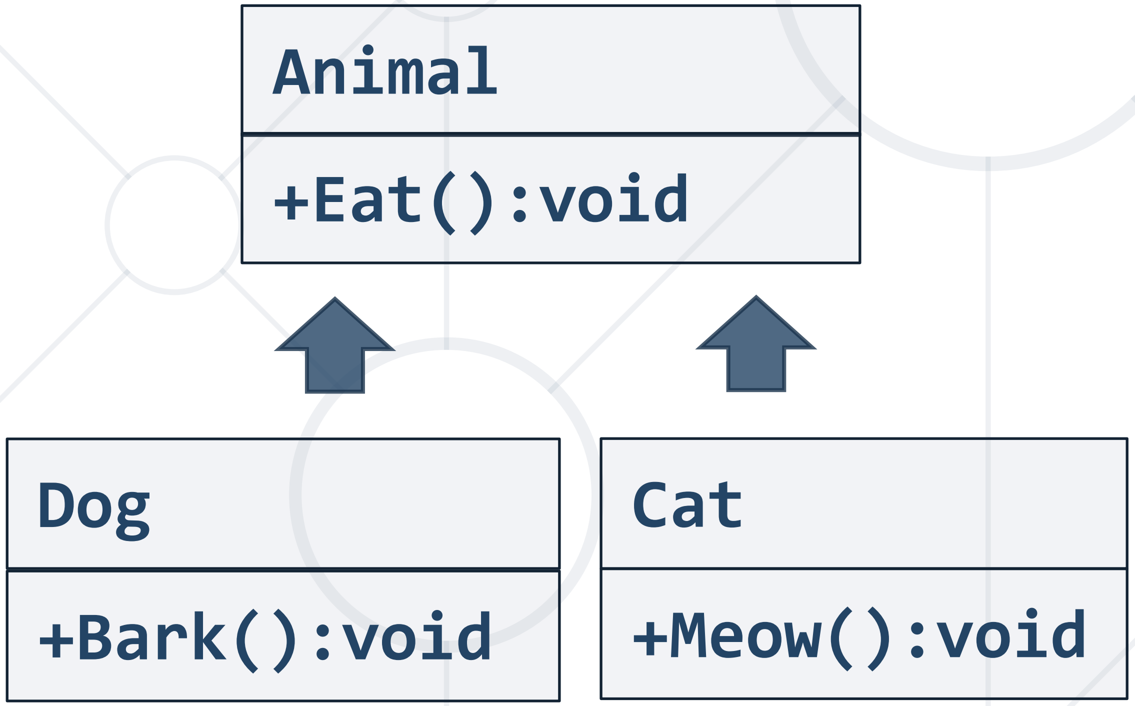
```
Puppy puppy = new Puppy();
puppy.Eat();
puppy.Bark();
puppy.Weep();
```



Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3164#1

Problem: Inheritance Hierarchy

- Create three classes named **Animal**, **Dog** and **Cat**:
- **Dog** and **Cat** should inherit from **Animal**



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

Cat cat = new Cat();
cat.Eat();
cat.Meow();
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3164#2

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 **not inherited** 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

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

Local variable

Instance member

Base class 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 (1)

- The **sealed** modifier prevents other classes from **inheriting** from it

```
class Dinosaur
{
  public void Eat() {…}
}
```

```
sealed class TRex : Dinosaur
{
  public void Eat() {…}
}
```

```
class EvolvedTRex : TRex
{
}
```

Sealed Modifier (2)

- 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

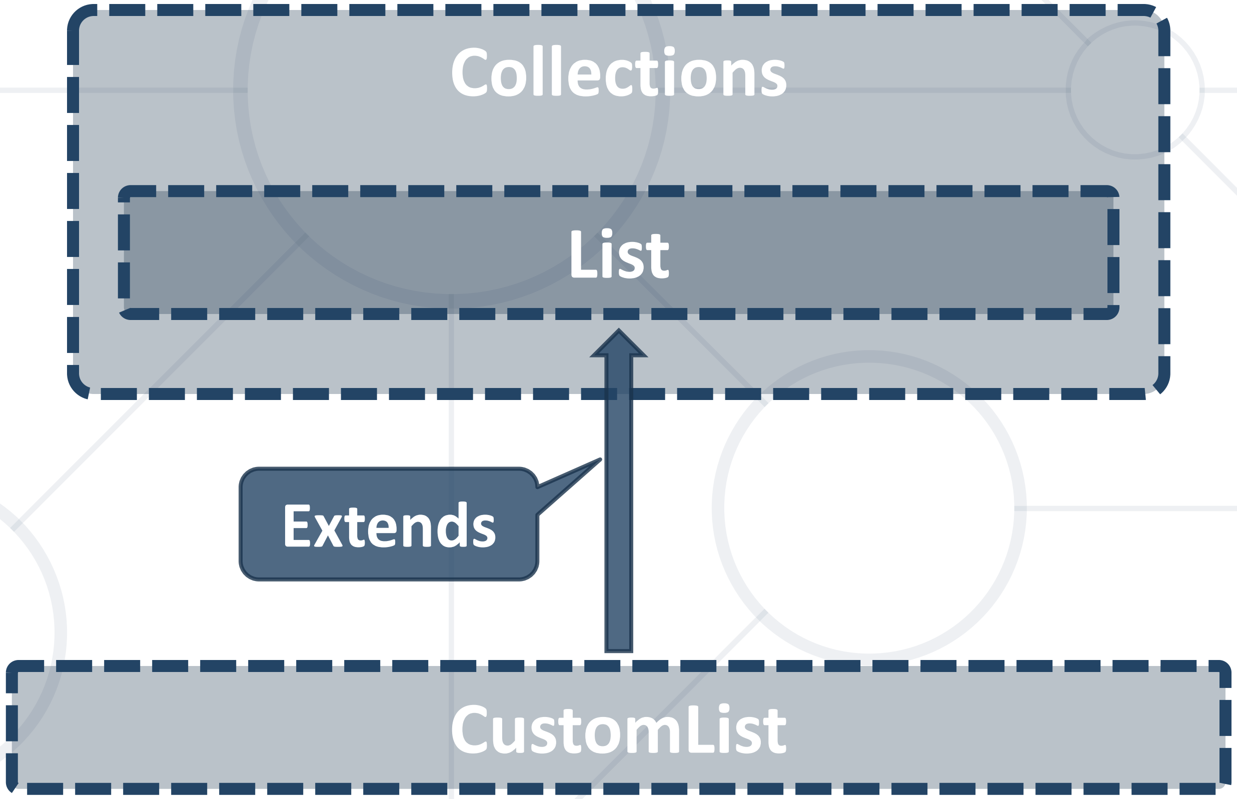
```
class Bird
{
  public virtual void Fly() {}
}
```

```
class Waimanu : Bird
{
  public sealed override void Fly() {}
}
```

```
class Penguin : Waimanu
{
  public void Walk() {}
}
```

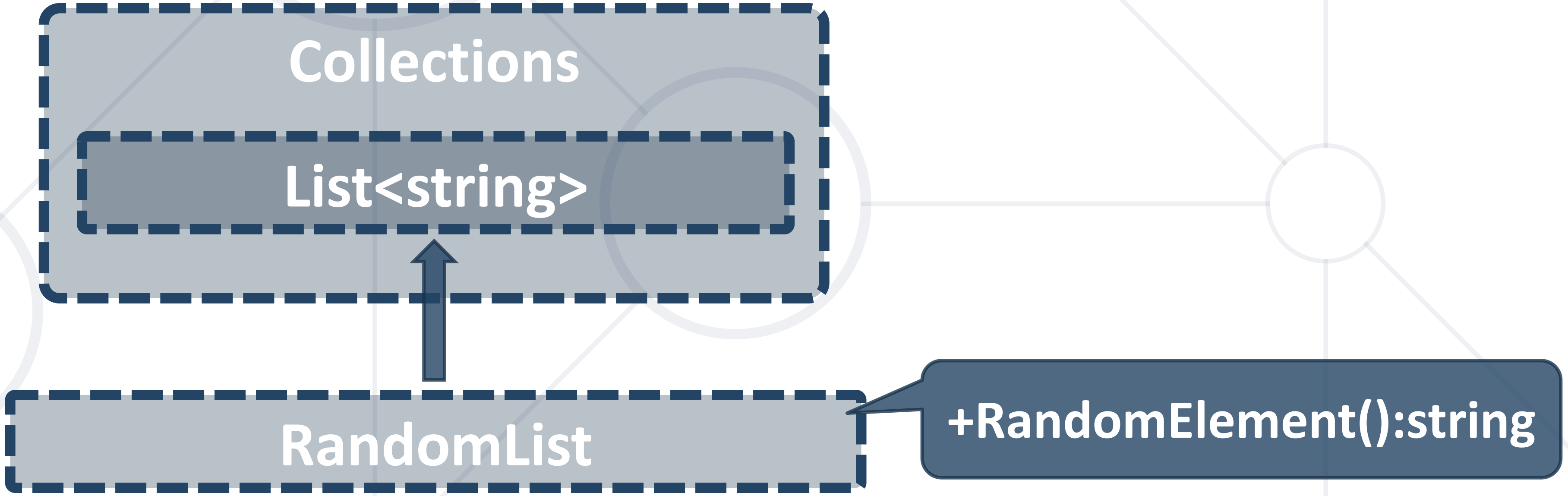
Inheritance Benefits – Extension

- We can **extend a class** that we **can't otherwise change**



Problem: Random List

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



Solution: Random List

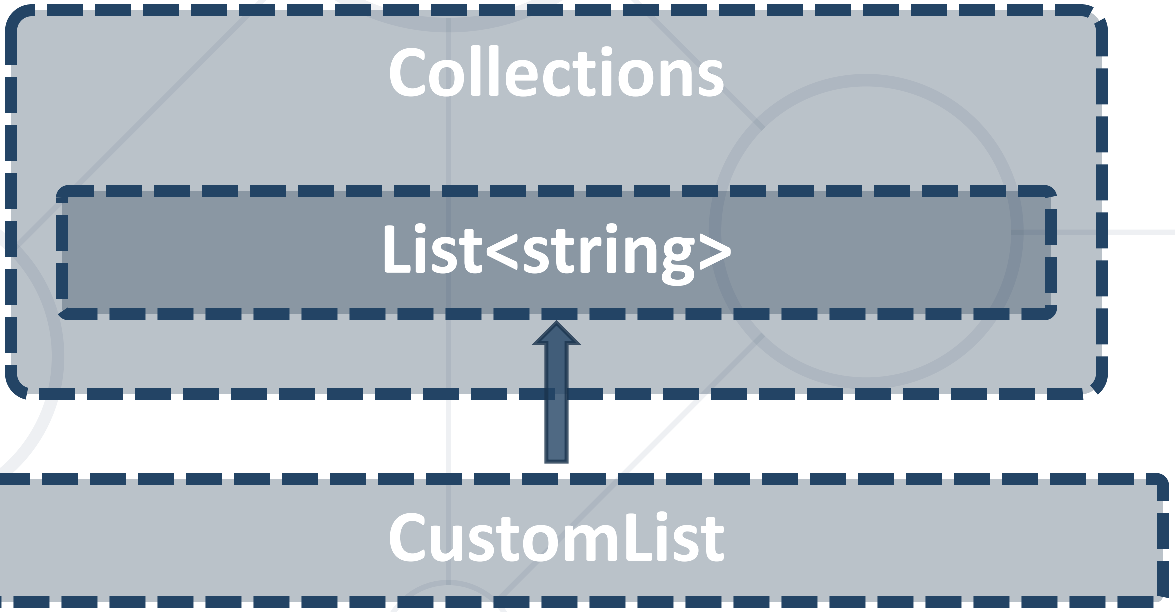
```
public class RandomList : List<string>
{
  private Random rnd; // TODO: Add constructor
  public string RemoveRandomElement()
  {
    int index = rnd.Next(0, this.Count);
    string str = this[index];
    this.RemoveAt(index);
    return str;
  }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3164#3

Extension, Composition, Delegation

Extension

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

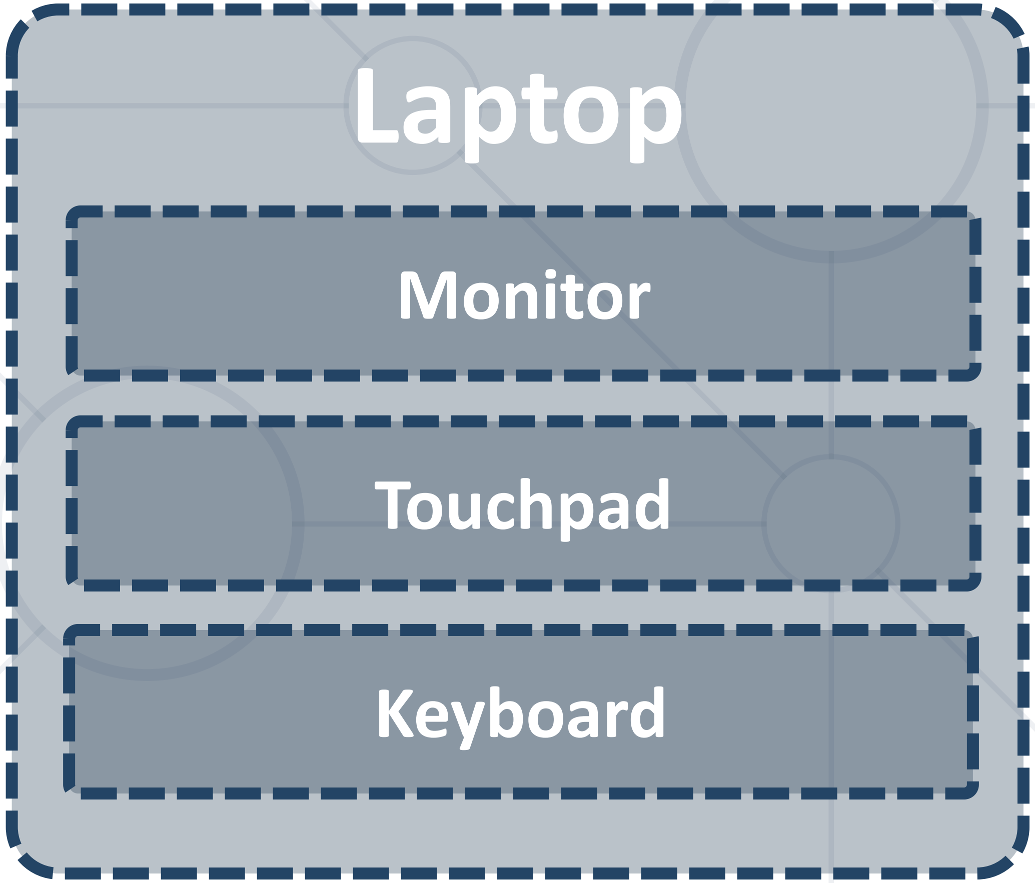


Composition

- Using classes to **define** class fields and properties

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

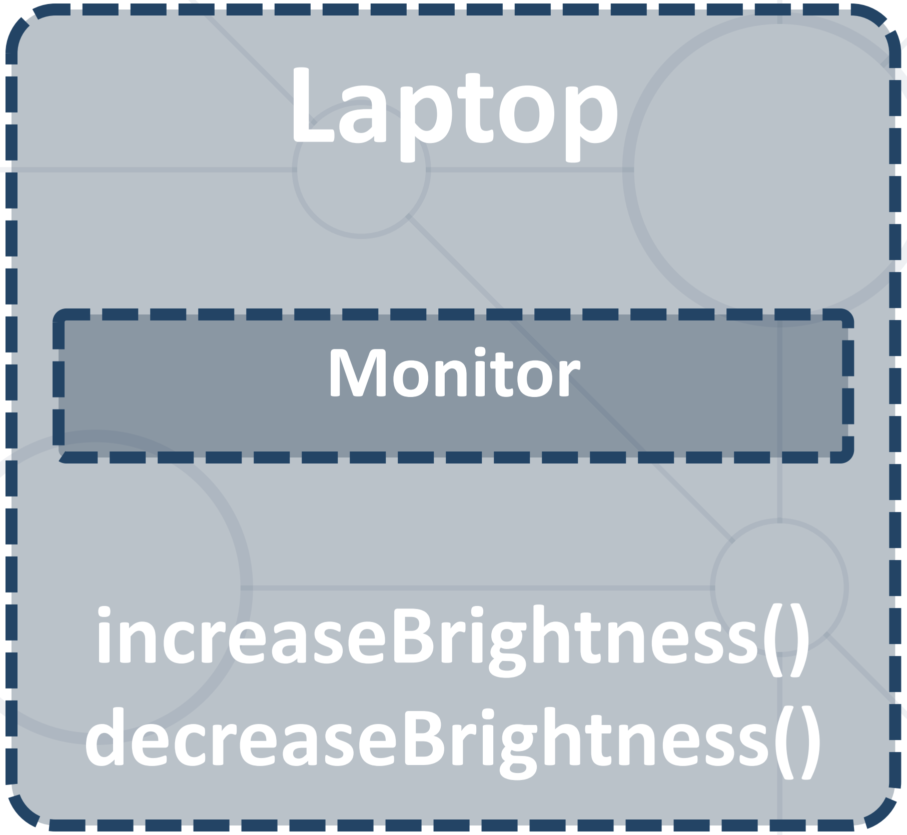
Reusing classes



Delegation

```
class Laptop
{
  Monitor monitor;
  void IncrBrightness() =>
    monitor.Brighten();

  void DecrBrightness() =>
    monitor.Dim();
}
```

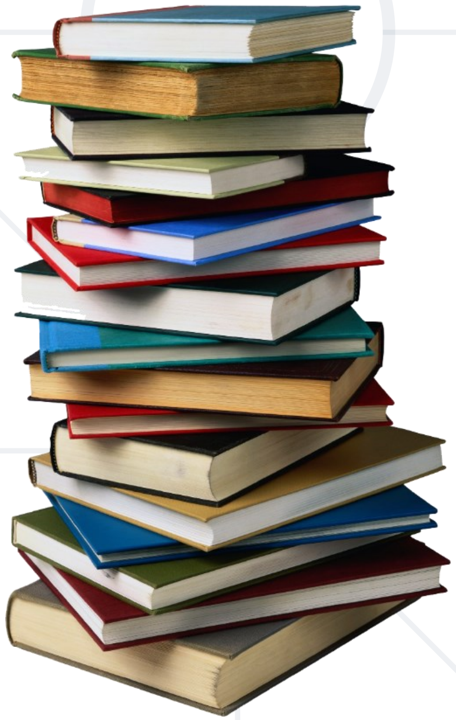


Problem: Stack of Strings

- Create a **`StackOfStrings`** class which **inherits** the **`Stack<string>`** and adds the following methods:

`StackOfStrings`

`+IsEmpty(): boolean`
`+AddRange(elements): void`



Solution: Stack of Strings

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

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3164#4

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;
  }
}
```

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!");
```

Problem: Exception Trace

- **Read** all lines from a **file** and **sum** the **numbers**
- Use **class `MyFileReader`**
- If the file **path** is null or empty **`throw new ArgumentException`** with message `"Invalid Path or File Name."`
- If any value in the file **cannot be parsed `throw new ArgumentException`** with message `"Error: On the line {line number} of the file the value was not in the correct format."`
- If everything is **successful**, **print**: `"The sum of all correct numbers is: {numbers sum}"`

Solution: Exception Trace (1)

```
public class MyFileReader {

  private string path;
  public MyFileReader(string path)
  {
    this.Path = path;
  }

  public string Path
  {
    get { return path; }
    set {
      if (string.IsNullOrEmpty(value)) {
        throw new ArgumentException("Invalid Path or File Name."); }
      path = value;
    }
  }
```

Solution: Exception Trace (2)

```
  public void ReadAndSum() {
    string[] inputFromFile = File.ReadAllLines(this.Path);

    List<int> numbers = new List<int>();
    int countRow = 0;

    foreach (var value in inputFromFile) {
      countRow++;
      try { numbers.Add(int.Parse(value)); }

      catch (Exception) {
        throw new ArgumentException($"Error: On the line {countRow}
          of the file the value was not in the correct format."); }
    }
  Console.WriteLine($"The sum of all correct numbers is: {numbers.Sum()}");
  }
}
```

Solution: Exception Trace (3)

```
static void Main() {
  try {
    MyFileReader reader1 = new MyFileReader(@"C:\temp\numbers.txt");
    reader1.ReadAndSum();
  }
  catch (Exception ex) {
    Console.Error.WriteLine("Error: " + ex.Message);
  }
  try {
    MyFileReader reader2 = new MyFileReader(@"");
    reader2.ReadAndSum();
  }
  catch (Exception ex) {
    Console.Error.WriteLine("Error: " + ex.Message);
  }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3164#5

Summary

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

Software University

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License