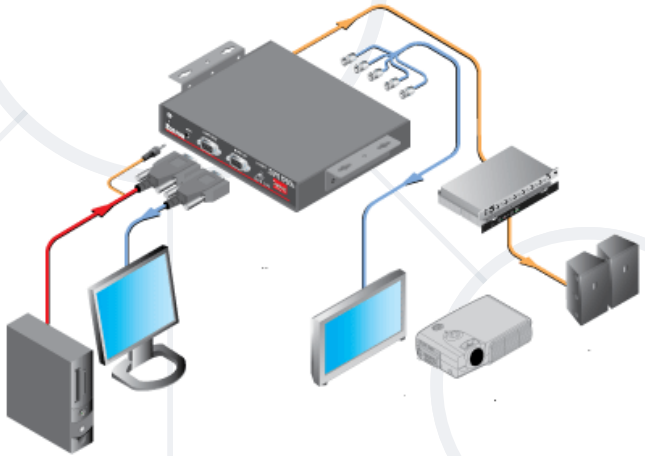
Interfaces and Abstraction

Interfaces vs Abstract Classes
Abstraction vs Encapsulation



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

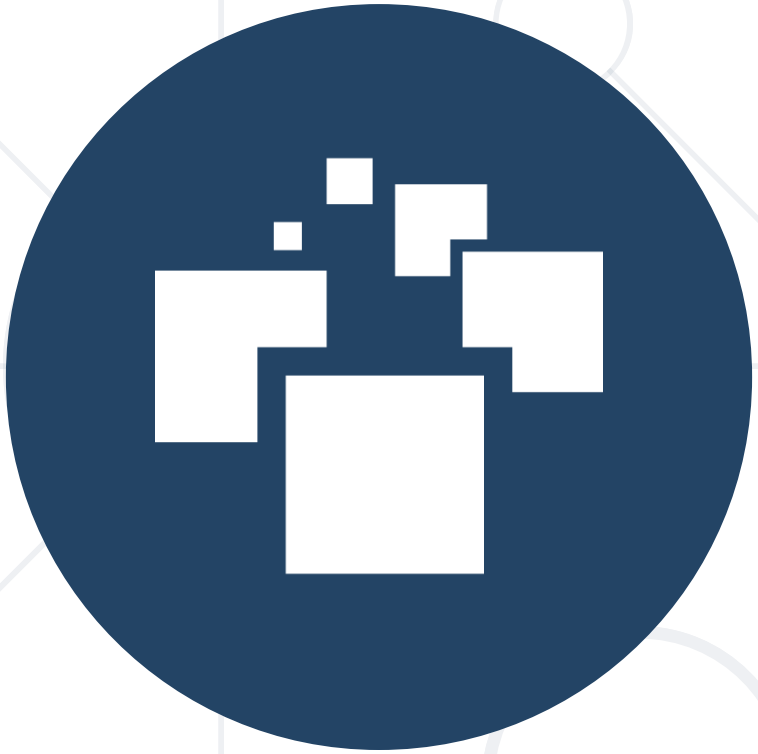
Have a Question?

Software University

sli.do

#csharp-advanced

Table of Contents

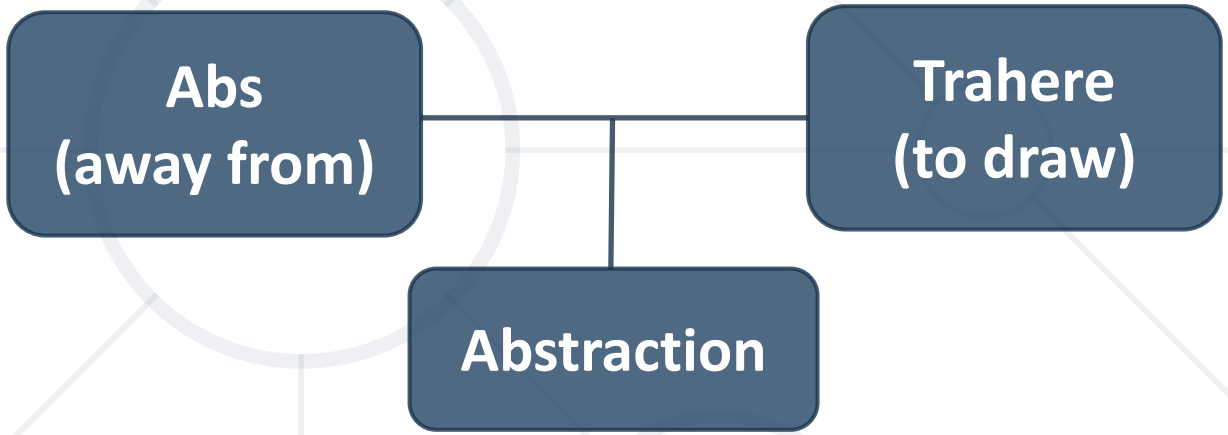


Achieving Abstraction

Abstraction

What is Abstraction?

- From the Latin



- **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 what?

- **… 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();
}
```

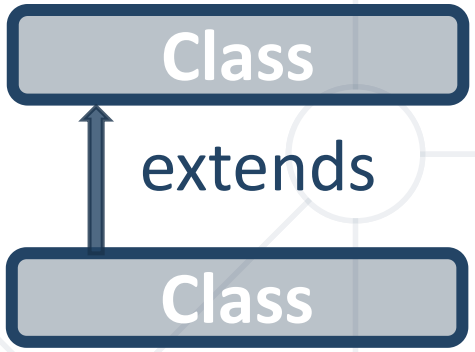
```
class Document : IPrintable {
  public void Print()
  { Console.WriteLine("Hello"); }
```

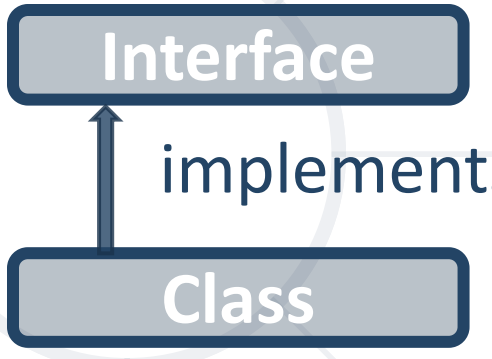
Interface (2)

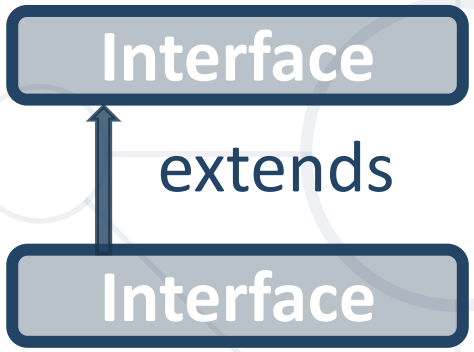
- Contains signatures of **methods** (in C# 8.0 interfaces could have a default implementation), **properties**, **events** or **indexers**
- Can **inherit one** or **more** base interfaces
- When a base type list contains a base class and interfaces, the **base class** must come **first** in the list
- 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

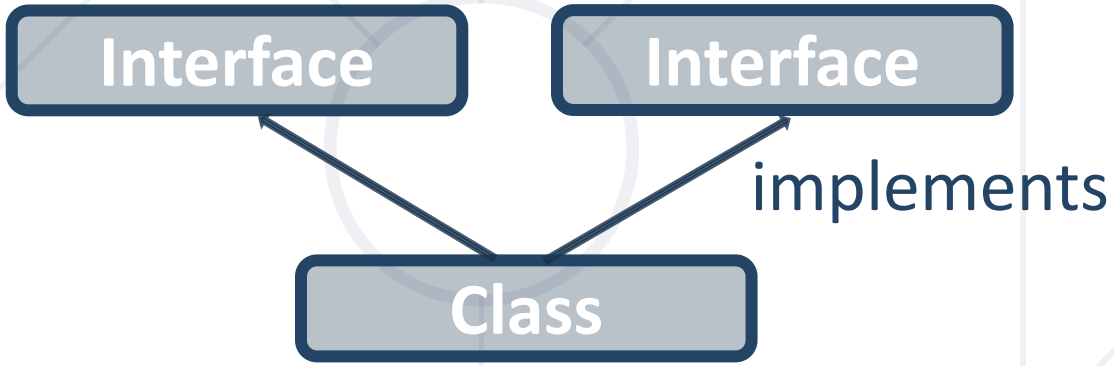
- Relationship between **classes** and **interfaces**

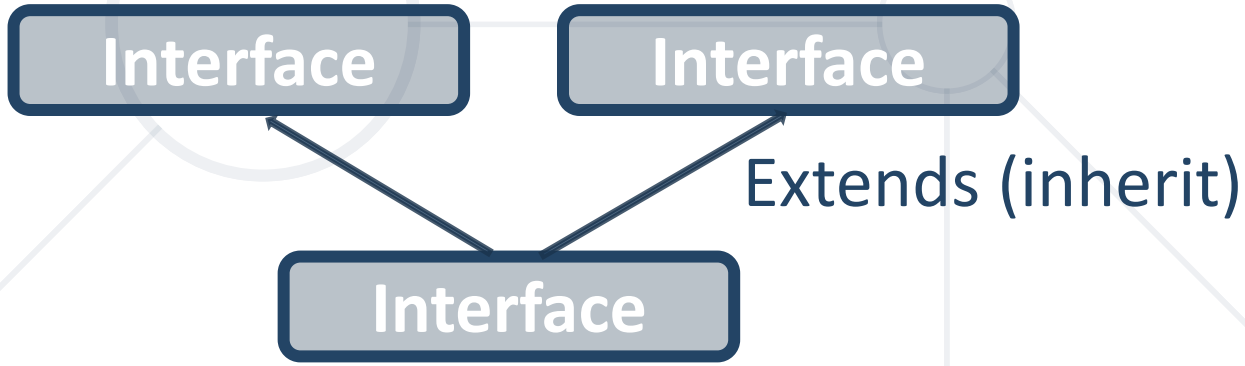






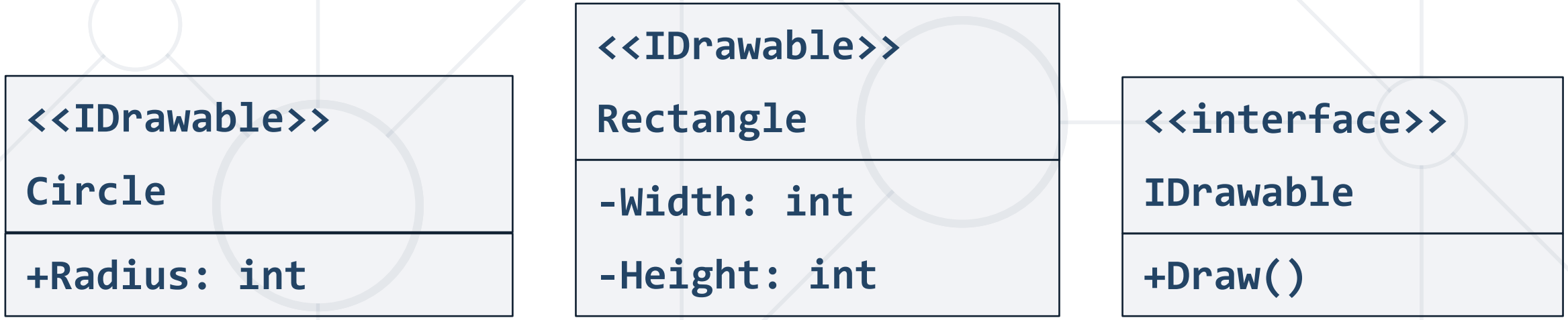
- Multiple implementation and inheritance





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 "*"



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

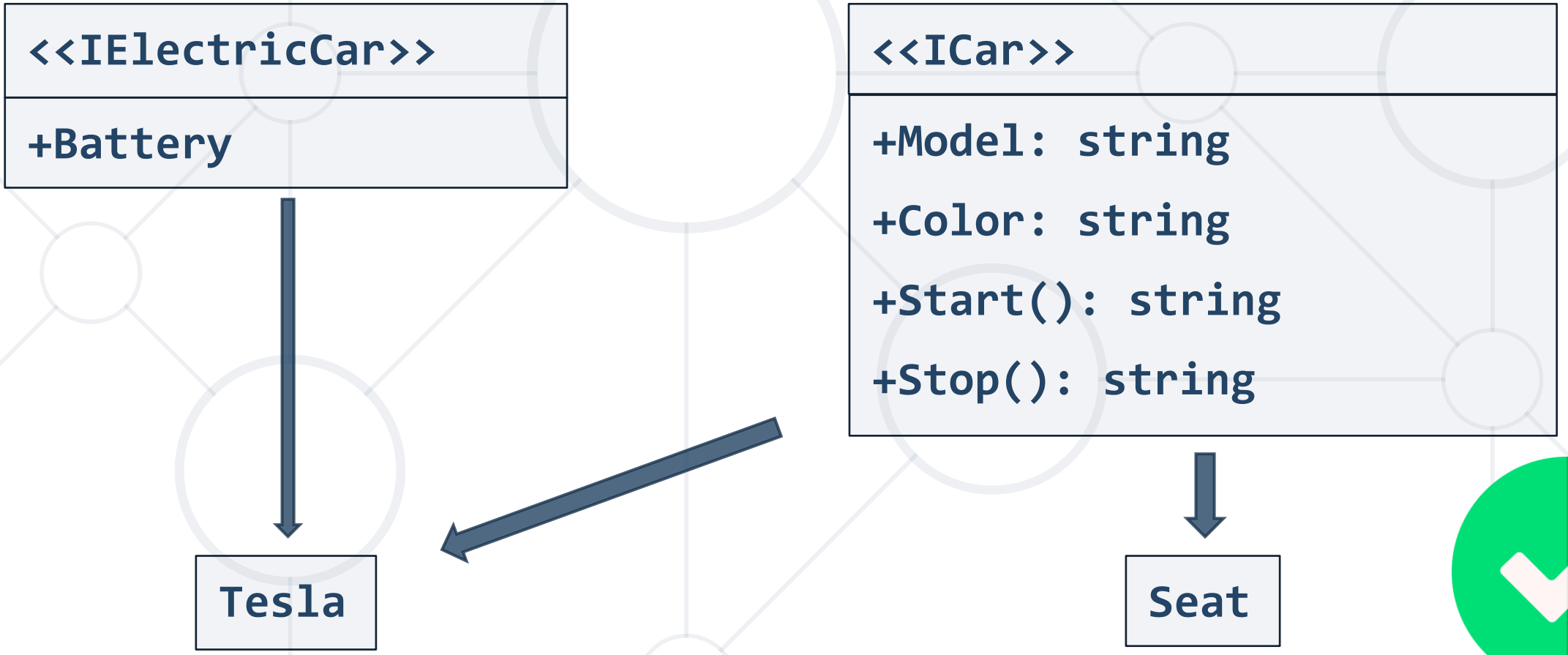
- Interface
  - A class may **implement several interfaces**
  - **Cannot have access modifiers**, everything is assumed as public
  - **Cannot provide any code**, just the signature
- Abstract Class (AC)
  - May **inherit only one abstract** class
  - Can **provide implementation** and/or just the **signature**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

Interface vs Abstract Class

- 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

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

public interface IElectricCar {
    int Batteries { get; }
}
```

Solution: Cars

```
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

```
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

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

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