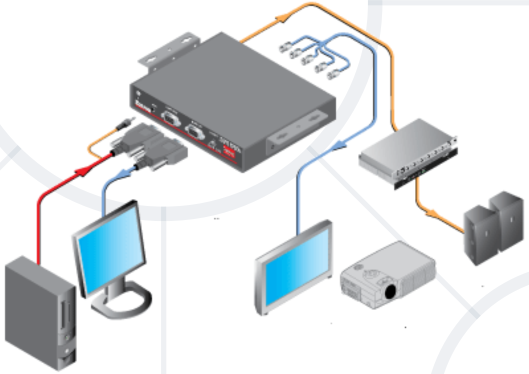
Abstract Classes and Interfaces

Abstraction vs Encapsulation

Interfaces vs Abstract Classes



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

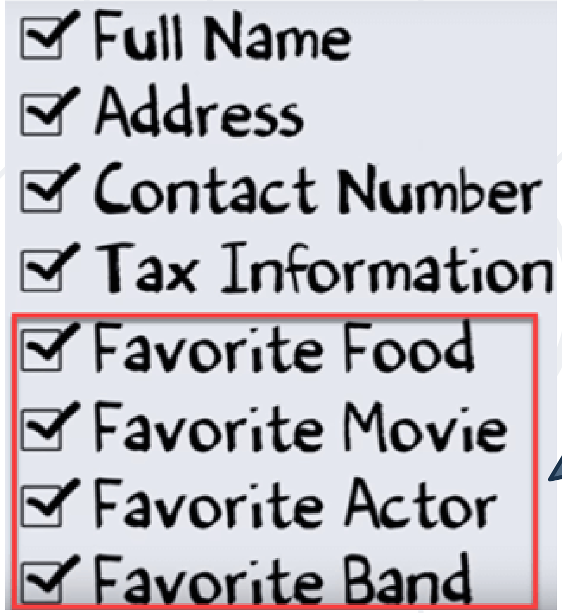
Table of Contents

1. Abstraction
2. Interfaces
3. Abstract Classes
4. Interfaces vs Abstract Classes

Achieving Abstraction

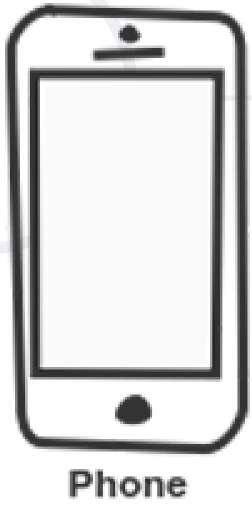
Abstraction in OOP

- **Abstraction** "shows" only **essential attributes** and "hides“ unnecessary information
- It helps **managing** complexity
- Abstraction lets you focus on **what the object does** instead of **how it does it**



☑ Full Name
☑ Address
☑ Contact Number
☑ Tax Information
☑ Favorite Food
☑ Favorite Movie
☑ Favorite Actor
☑ Favorite Band

We do not need this information for a banking application



You can use it, but you don't need to know what's going on under the hood

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

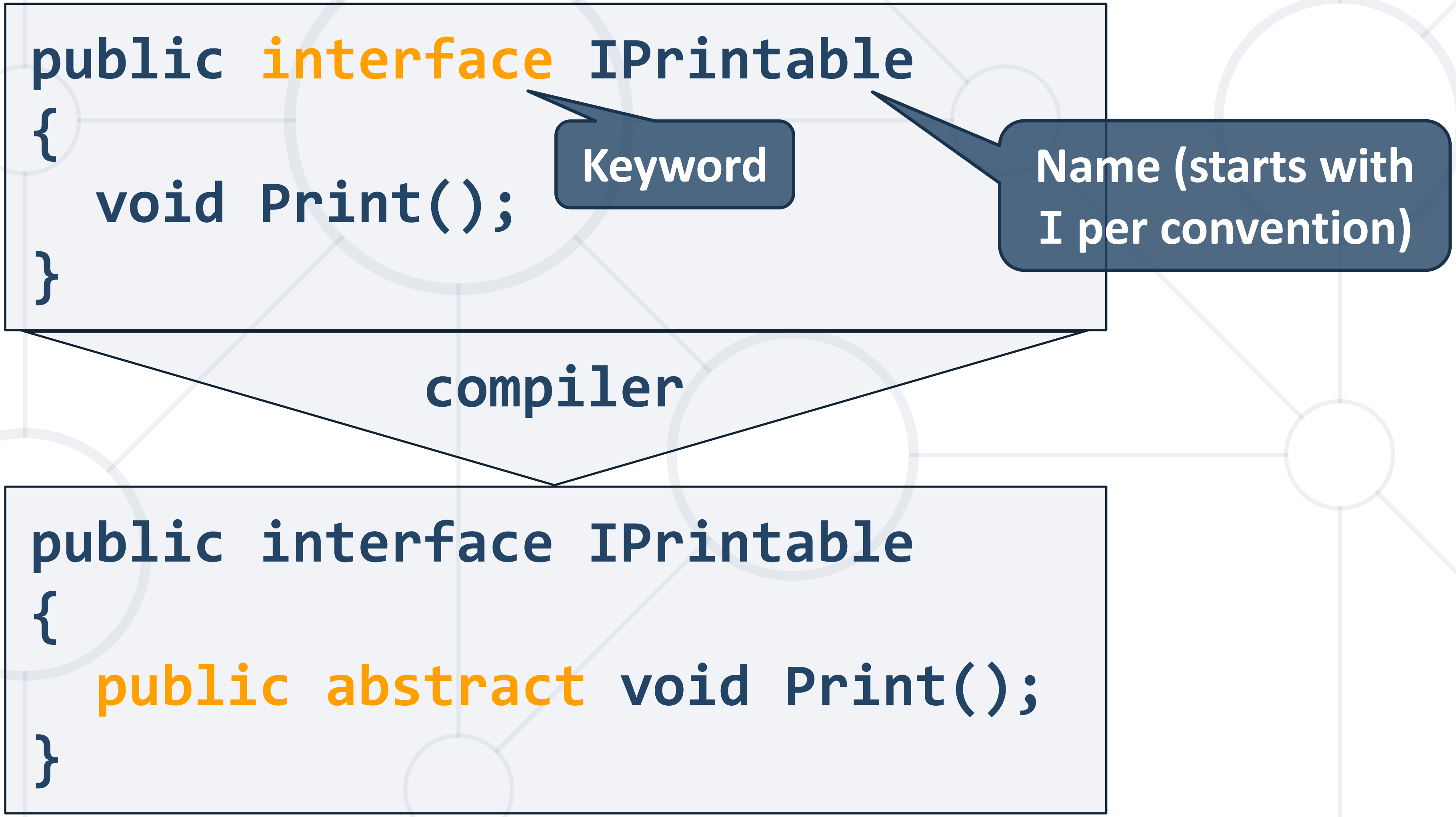
- Abstraction
  - Process of **hiding the implementation details** and showing only functionality to the user
  - Achieved with **interfaces** and **abstract classes**

- 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

Interface in Reality

- Internal addition by compiler



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

Keyword

Name (starts with I per convention)

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

Interface Example

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

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

Only the signatures

Classes must come first

One or more interfaces

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

Explicit Interface (1)

- A class that **implements** an interface can **explicitly** implement **members** of that **interface**

```
public interface IFile
{
  void ReadFile();
}
```

```
public interface IBinaryFile
{
  void ReadFile();
}
```

```
class FileInfo : IFile, IBinaryFile
{
  void IFile.ReadFile()
  {
    Console.WriteLine("Reading File");
  }
}
```

Explicitly implemented member

Explicit Interface (2)

- An explicitly implemented member **cannot** be accessed through a class instance, but only **through an instance of the interface**

```
public interface IBinaryFile
{
  void ReadFile();
  void OpenBinaryFile();
}
```

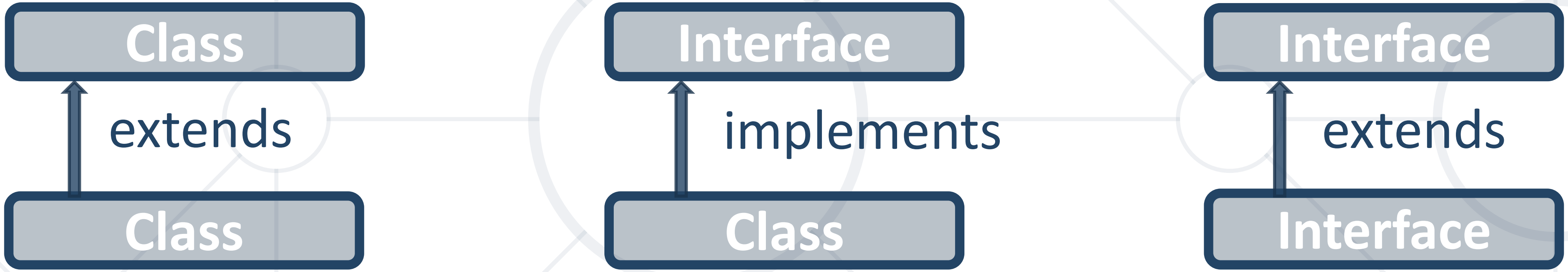
```
class FileInfo : IFile, IBinaryFile
{
  void IFile.ReadFile() {…}
  void OpenBinaryFile() {…}
}
```

```
public static void Main()
{
  IBinaryFile file = new FileInfo();
  file.OpenBinaryFile();
}
```

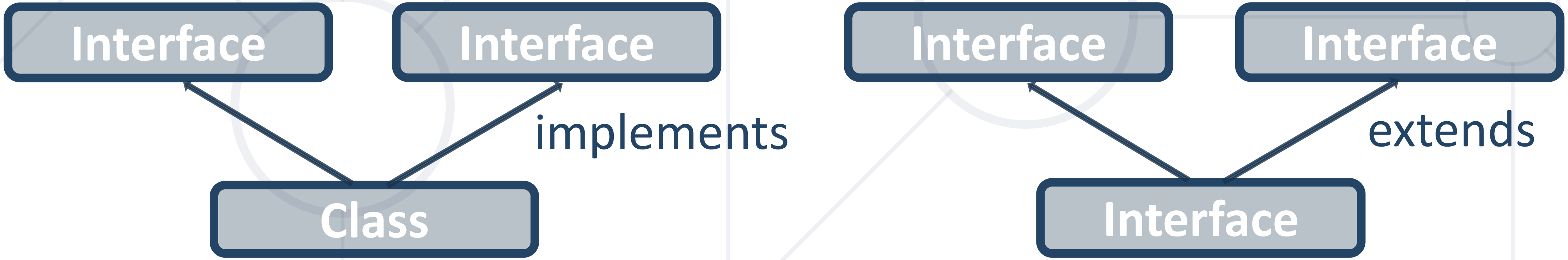
Accessed through instance

Multiple Inheritance

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

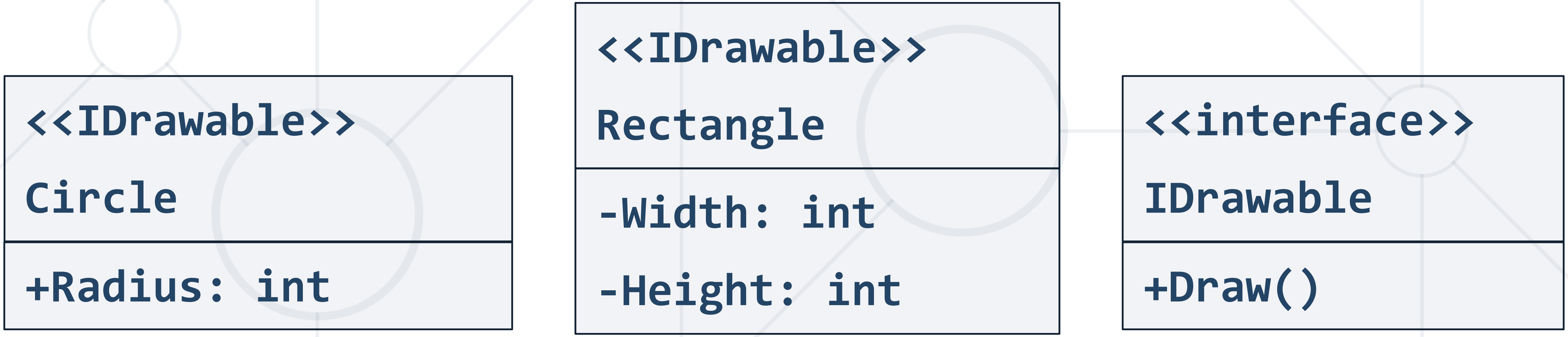


- Multiple inheritance



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()

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();
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3165#0

Abstract Classes and Methods

Abstract Class

Software University

- We can use an **abstract class** as a **base class** and all derived classes must implement abstract members

```
abstract class Shape
{
  public abstract int GetArea();
}
```

Named method

```
class Square : Shape
{
  int side;
  public Square(int n) => side = n;
  public override int GetArea() => side * side;
}
```

Child class(es) fills out the implementation

Abstract Class

- Abstract classes **may** contain **abstract methods and accessors**

```
abstract class BaseClass
{
  protected int x = 100;
  public abstract void AbstractMethod();
  public abstract int X { get; }
}
```

```
class DerivedClass : BaseClass
{
  public override void AbstractMethod() { x++; }
  public override int X   // overriding property
  { get { return x + 10; } }
}
```

Abstract Class

- Must provide **implementation** for all **inherited** interface members
- Implementing an interface might map the interface methods onto **abstract** methods

```
interface IService
{
  int Add();
}
```

```
abstract class ServiceBase : IService
{
  public abstract int Add();
}
```

```
public static void Main()
{
  ServiceBase service = new ServiceBase();
}
```

Abstract class **cannot** be instantiated

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:

```
abstract class ServiceBase : IService
{
  public abstract int Add();
}
```

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 **contain access modifiers** for the fields, functions, properties
  - Can **provide implementation** and/or just the **signature** that have to be overridden

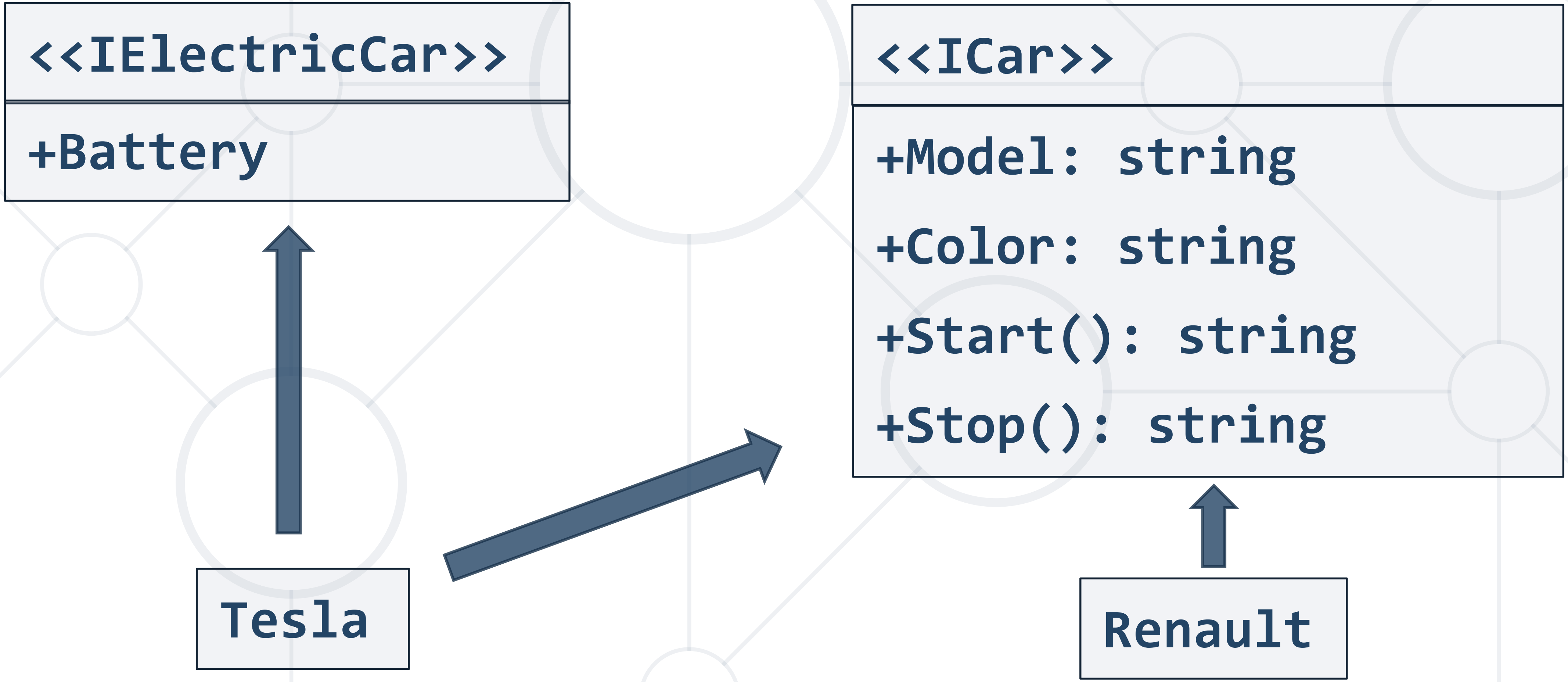
Interface vs Abstract Class (2)

- 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 (1)

- Build a hierarchy of interfaces and classes



Problem: Cars (2)

- 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 `Renault`, 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 }
}
```

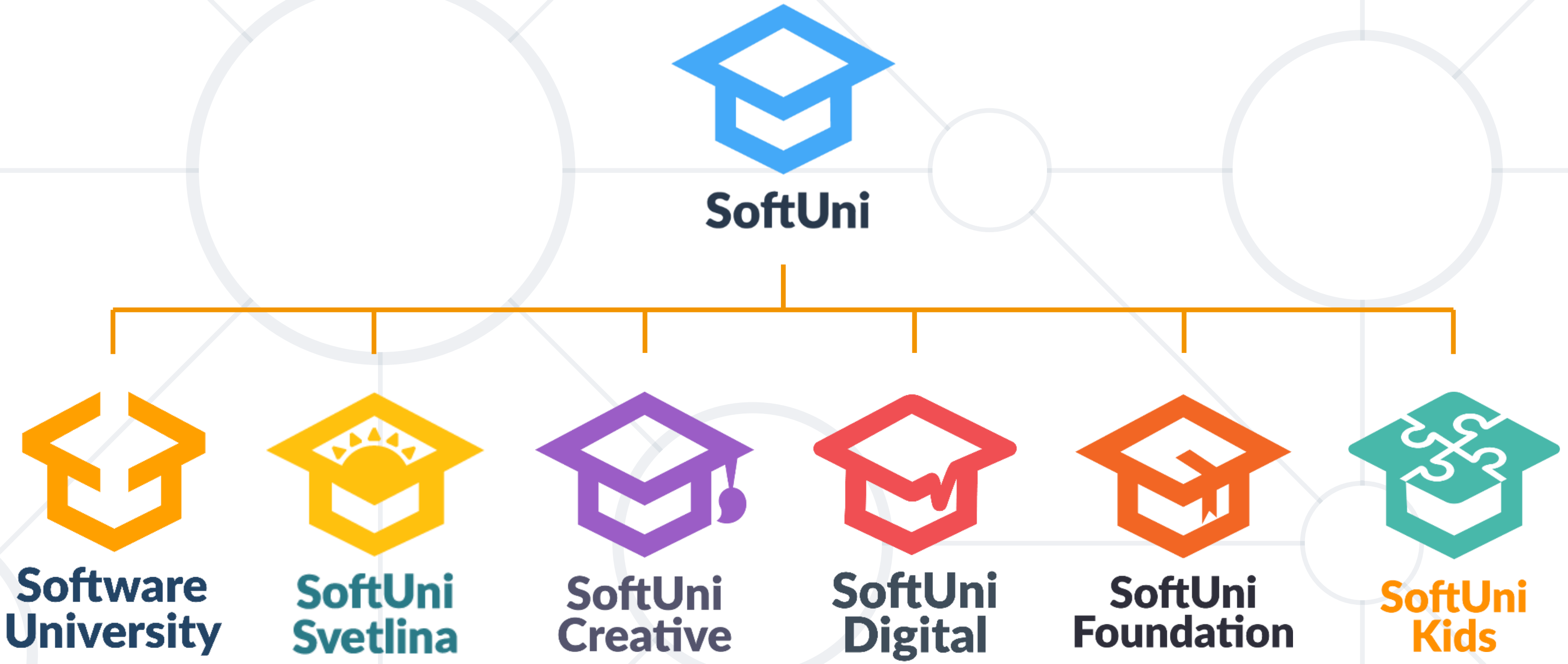
Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/3165#1

Summary

Software University

- **Abstraction** – Abstraction "shows" only essential attributes and "hides" unnecessary information
- How do we achieve abstraction – by interfaces or abstract class
- **Interfaces** – Holds only the signatures of methods and properties
- **Abstract classes** – base class and all derived classes must implement abstract members

Questions?



SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License