Interfaces and Abstraction

Interfaces vs Abstract Classes
Abstraction vs Encapsulation

SoftUni Foundation

SoftUni Team

Technical Trainers

Software University

http://softuni.bg

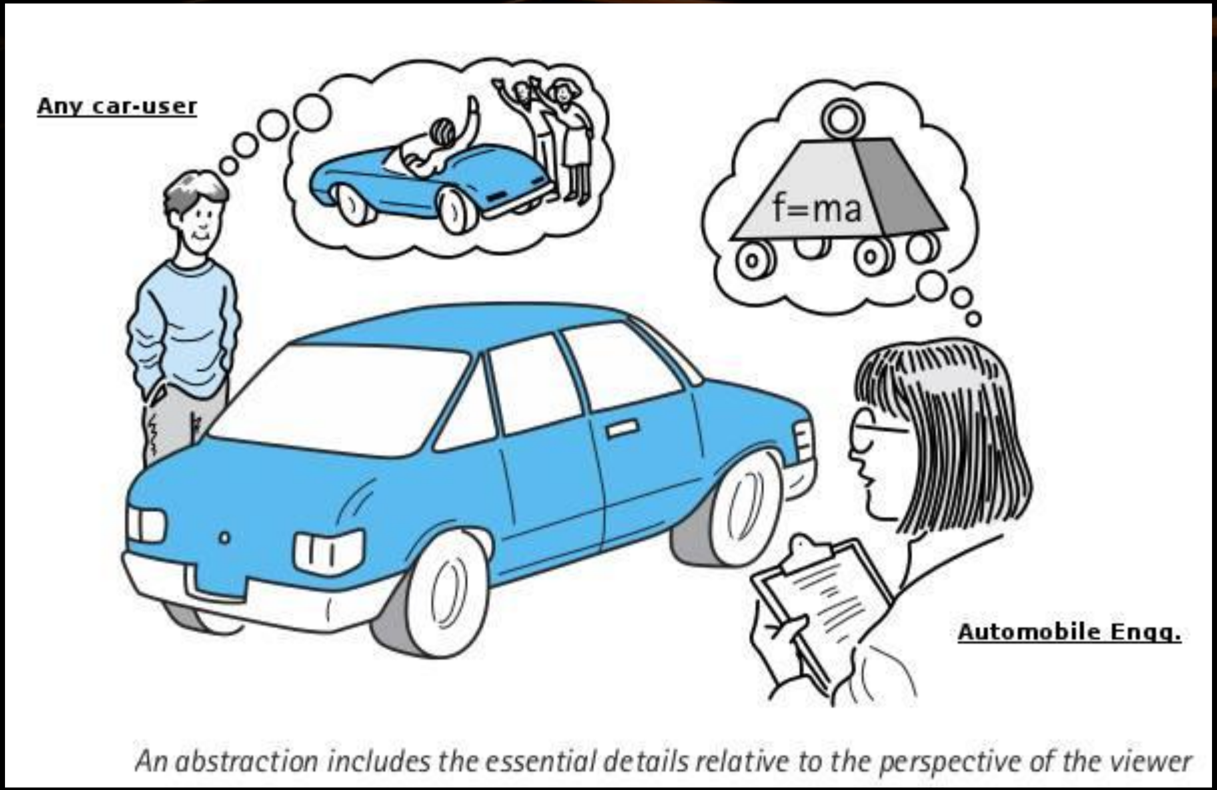
Table of Contents

SoftUni Foundation

Questions

sli.do

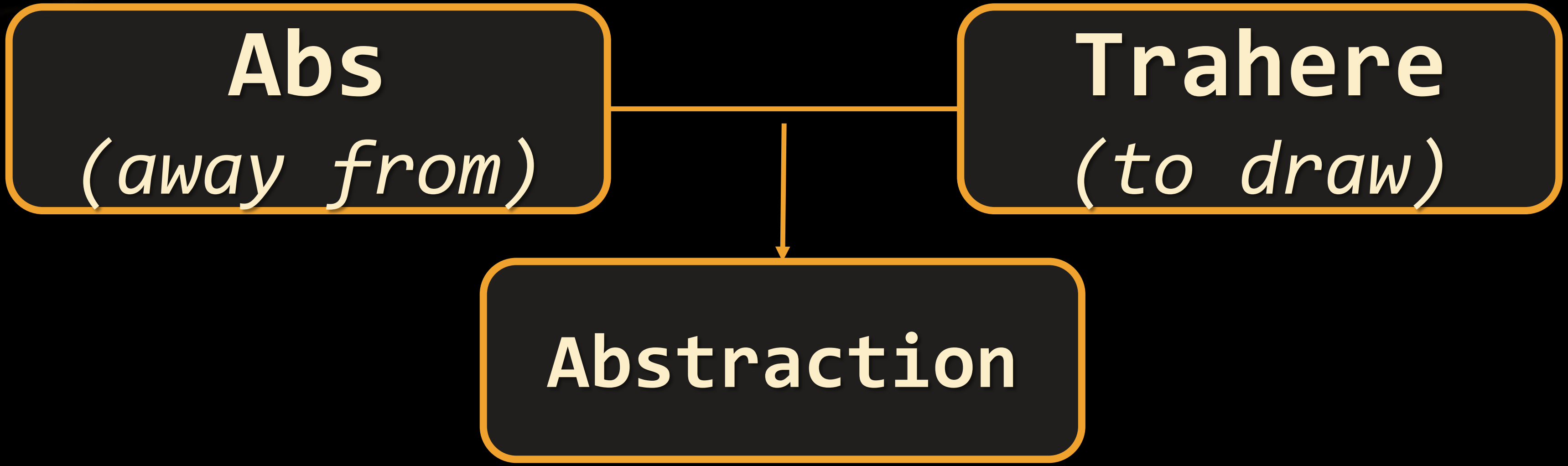
#JavaFundamentals


Any car-user
f=ma
Automobile Engg.
An abstraction includes the essential details relative to the perspective of the viewer

Abstraction

What is Abstraction?

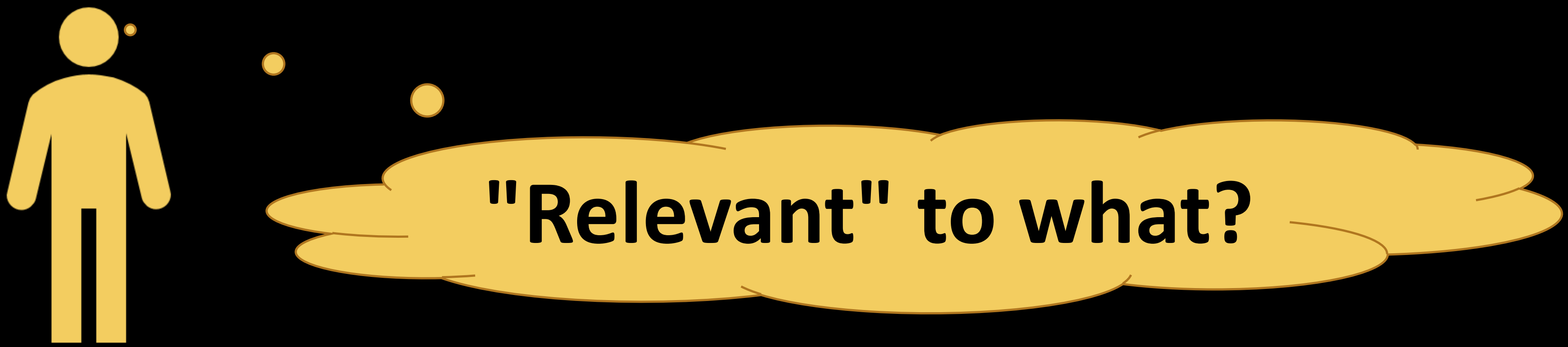
- From Latin



Process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.

Abstraction in OOP

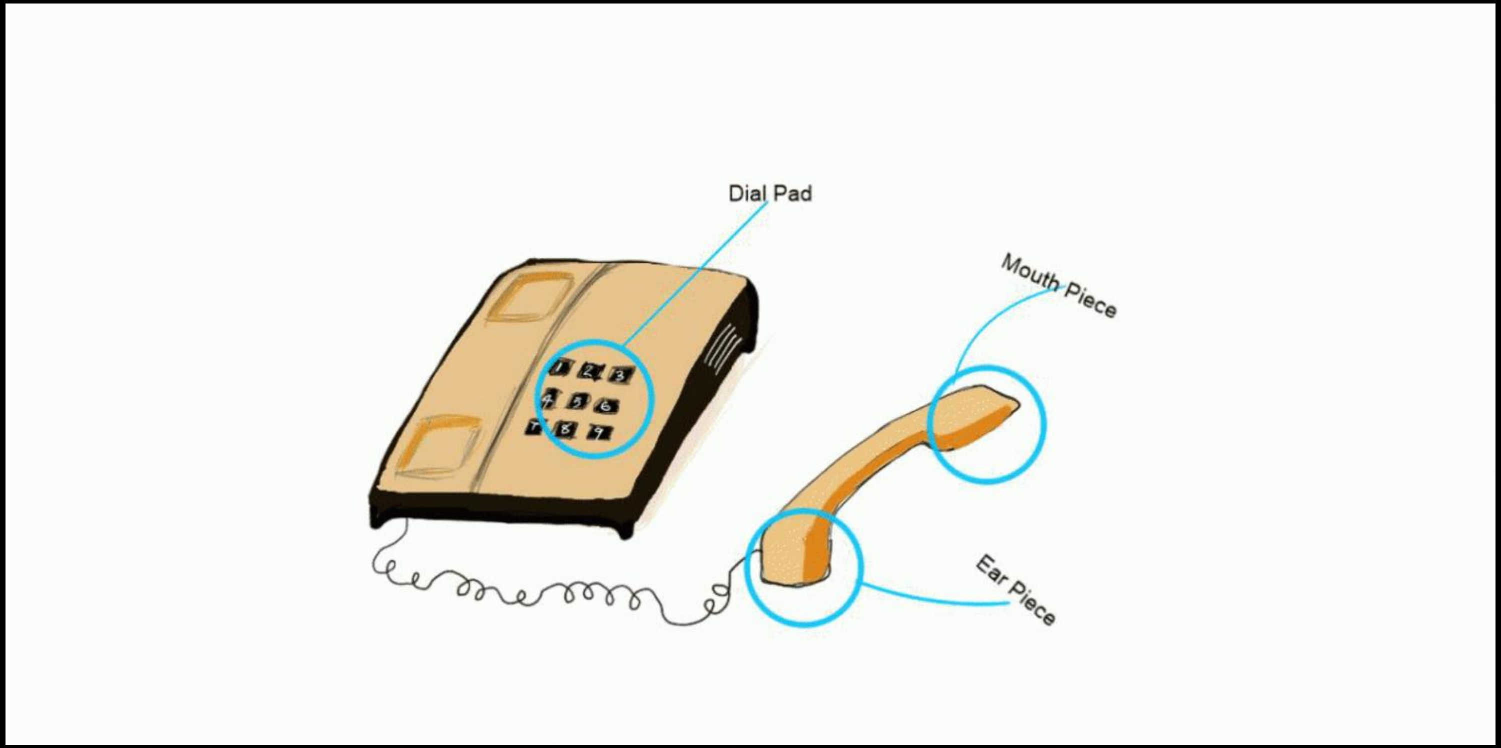
- **Abstraction** means ignoring **irrelevant** features, properties, or functions and emphasizing the **relevant ones ...**



- **... relevant to the project** we develop
- Abstraction helps managing complexity

Abstraction Example

- **Abstraction** lets you focus on **what the object does** instead of **how it does it.**



How do we achieve abstraction?

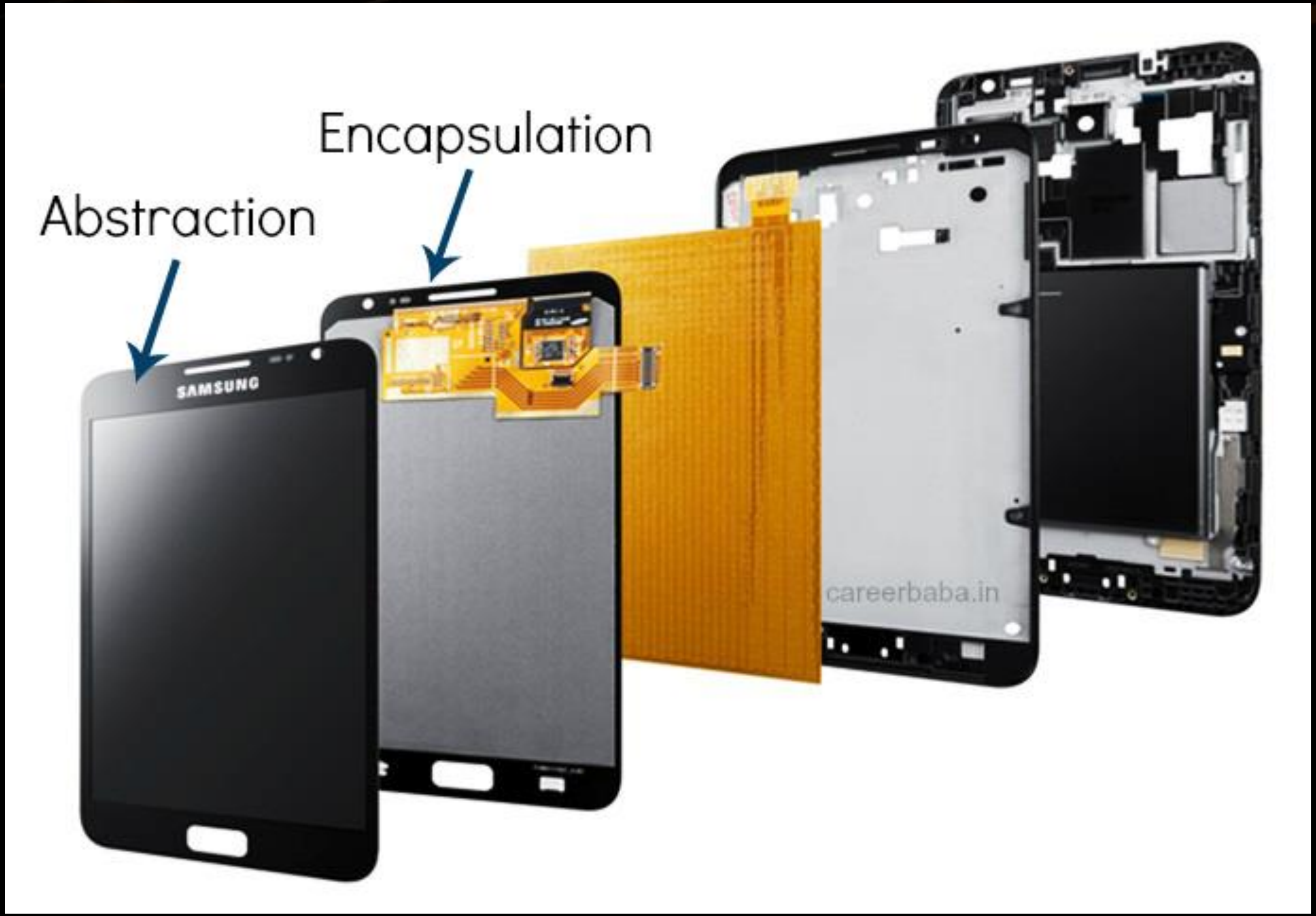
- There are two ways to achieve abstraction in Java
  - Interfaces (**100% abstraction**)
  - Abstract class (**0% - 100% abstraction**)

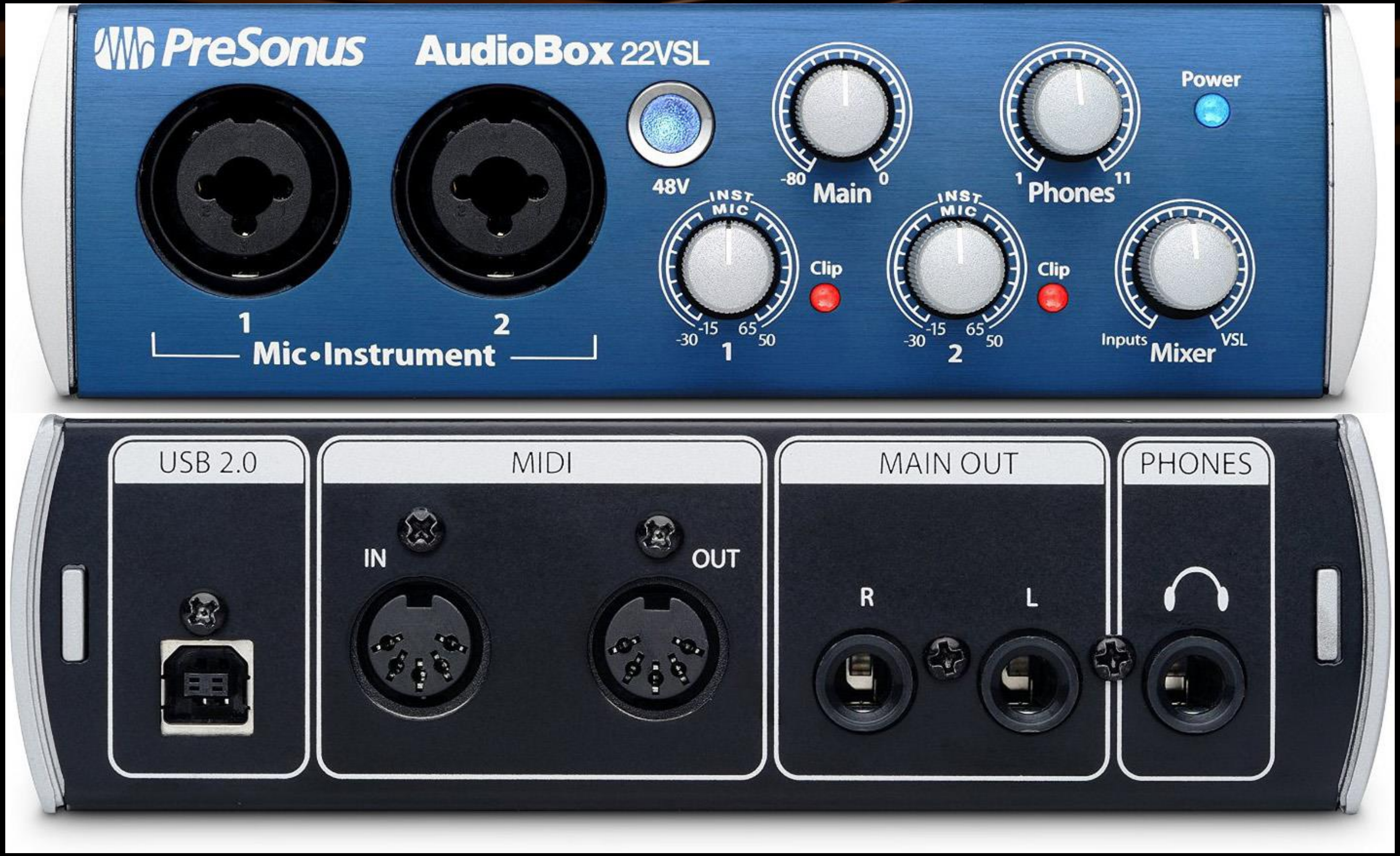
```
public interface Animal {}
public abstract class Mammal {}
public class Person extends Mammal implements Animal {}
```

Abstraction vs Encapsulation

- **Abstraction**
  - Achieved with **interfaces** and **abstract classes**
  - Hiding the implementation details and **showing only functionality** to the user.
- **Encapsulation**
  - Achieved with access modifiers (**private, public…**)
  - Hiding the code and data in a single unit to **protect the data from the outside world**

Abstraction vs Encapsulation (2)




PreSonus
AudioBox 22VSL
48V
Main
Phones
Power
Clip
Clip
Mixer
Inputs
VSL
1
2
Mic•Instrument
USB 2.0
MIDI
IN
OUT
MAIN OUT
R
L
PHONES

Interface

Interface

- Internal addition by compiler

```
public interface Printable {
   int MIN = 5;
   void print();
}
```

Keyword

public or default modifier

Name

compiler

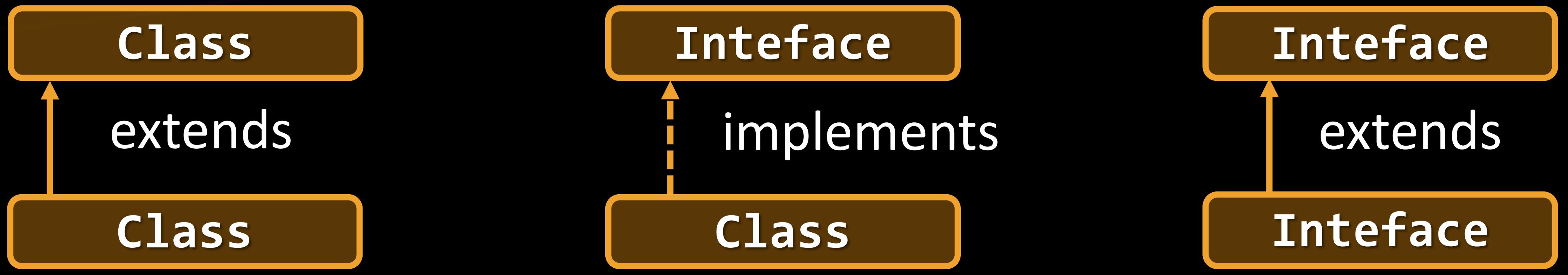
```
interface Printable {
   public static final int MIN = 5;
   public abstract void print();
}
```

adds public static final before fields

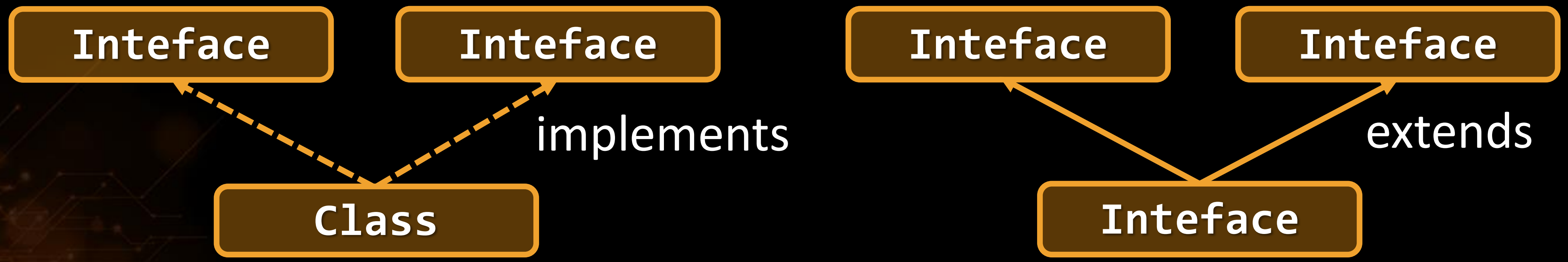
public abstract before methods

implements vs extends

- Relationship between classes and interfaces



- Multiple inheritance



Interface Example

- Implementation of `print()`is provided in class Document

```
public interface Printable {
  void print();
}
```

```
class Document implements Printable {
public void print() { System.out.println("Hello"); }

public static void main(String args[]){
Printable doc = new Document();
doc.print();   }
}
```

Polymorphism

Problem: Shapes Drawing

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

```
<<interface>>
Drawable
-------------
+draw()
```

```
<<Drawable>>
Circle
-------------
-radius: Integer
```

```
<<Drawable>>
Rectangle
-------------
-width: Integer
-height: Integer
```

Solution: Shapes Drawing

```
public interface Drawable {
  void draw();
}
```

```
public class Rectangle implements Drawable {
  public void draw() { /*draw a rectangle*/}
} //TODO:fields and constructor
```

```
public class Circle implements Drawable {
  public void draw() { /*draw a circle*/}
} //TODO:fields and constructor
```

Solution: Shapes Drawing - Rectangle Draw

```
public void draw(){
  drawLine(this.width, '*', '*');
  for (int i = 1; i < this.height - 1; ++i)
    drawLine(this.width, '*', ' ');
  drawLine(this.width, '*', '*');
}
```

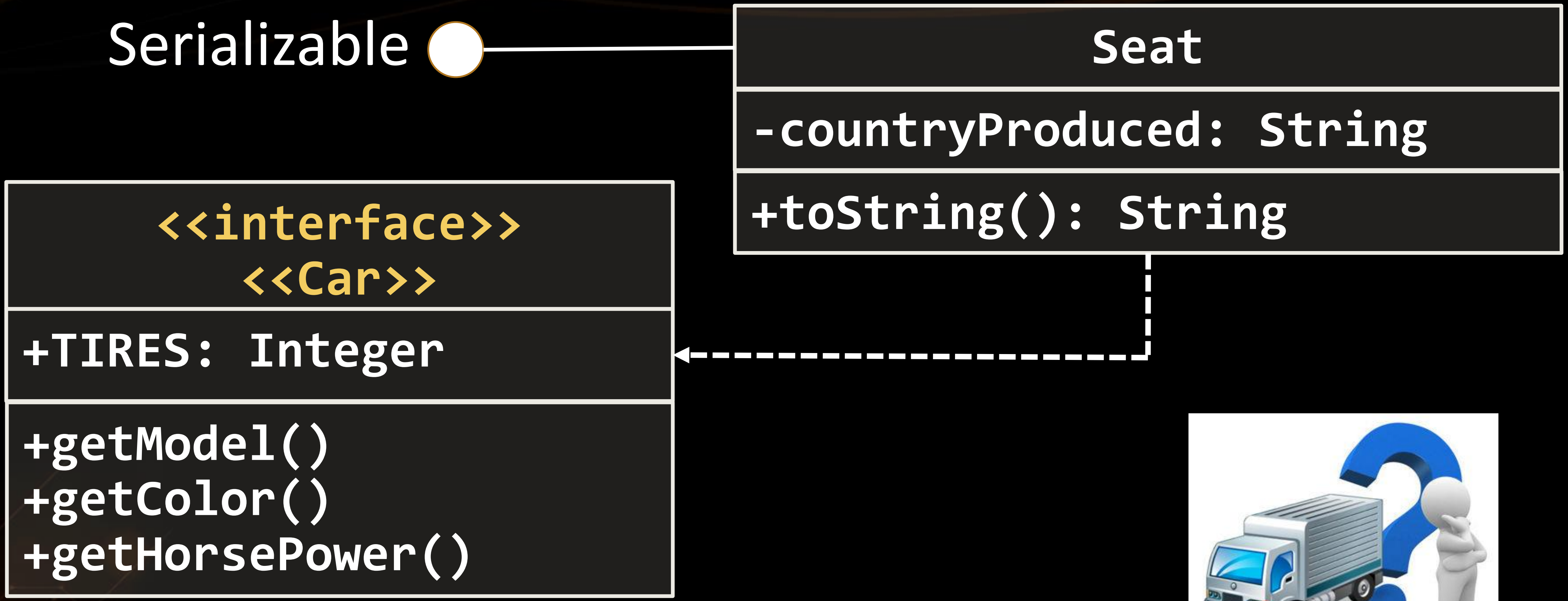
Solution: Shapes Drawing - Rectangle Draw

```
private void drawLine(int width, char end, char mid){
  System.out.print(end);
  for (int i = 1; i < width - 1; ++i)
    System.out.print(mid);
  System.out.println(end);
}
```

Solution: Shapes Drawing - Circle Draw

```
double r_in = this.radius - 0.4;
double r_out = this.radius + 0.4;
  for (double y = this.radius; y >= -this.radius; --y) {
    for (double x = -this.radius; x < r_out; x += 0.5) {
      double value = x * x + y * y;
      if (value >= r_in * r_in && value <= r_out * r_out) {
        System.out.print("*");
      } else
        System.out.print(" "); }
  System.out.println(); }
```

Problem: Car Shop



Solution: Car Shop

```java
public interface Car {
    int TIRES = 4;
    String getModel();
    String getColor();
    int getHorsePower();
}
```

Solution: Car Shop(2)

```java
public class Seat implements Car, Serializable {
  //TODO: Add fields, constructor and private methods
  @Override
  public String getModel() { return this.model; }
  @Override
  public String getColor() { return this.color; }
  @Override
  public int getHorsePower() { return this.horsePower; }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/498#0

Interfaces

Live Exercises in Class (Lab)

Extend Interface

- Interface can **extend another interface**

```
public interface Showable  {
  void show();
}
```

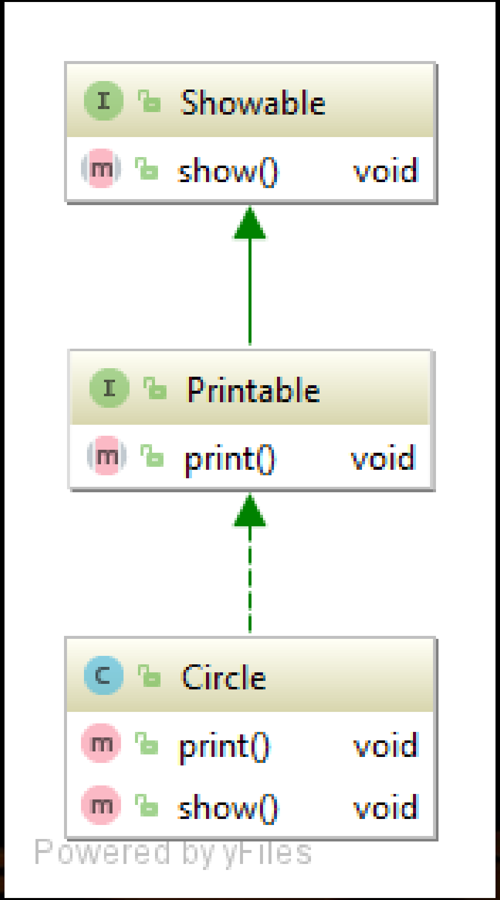
```
public interface Printable extends Showable {
  void print();
}
```

Extend Interface (2)

- Class which implements **child interface MUST** provide implementation for **parent interface** too

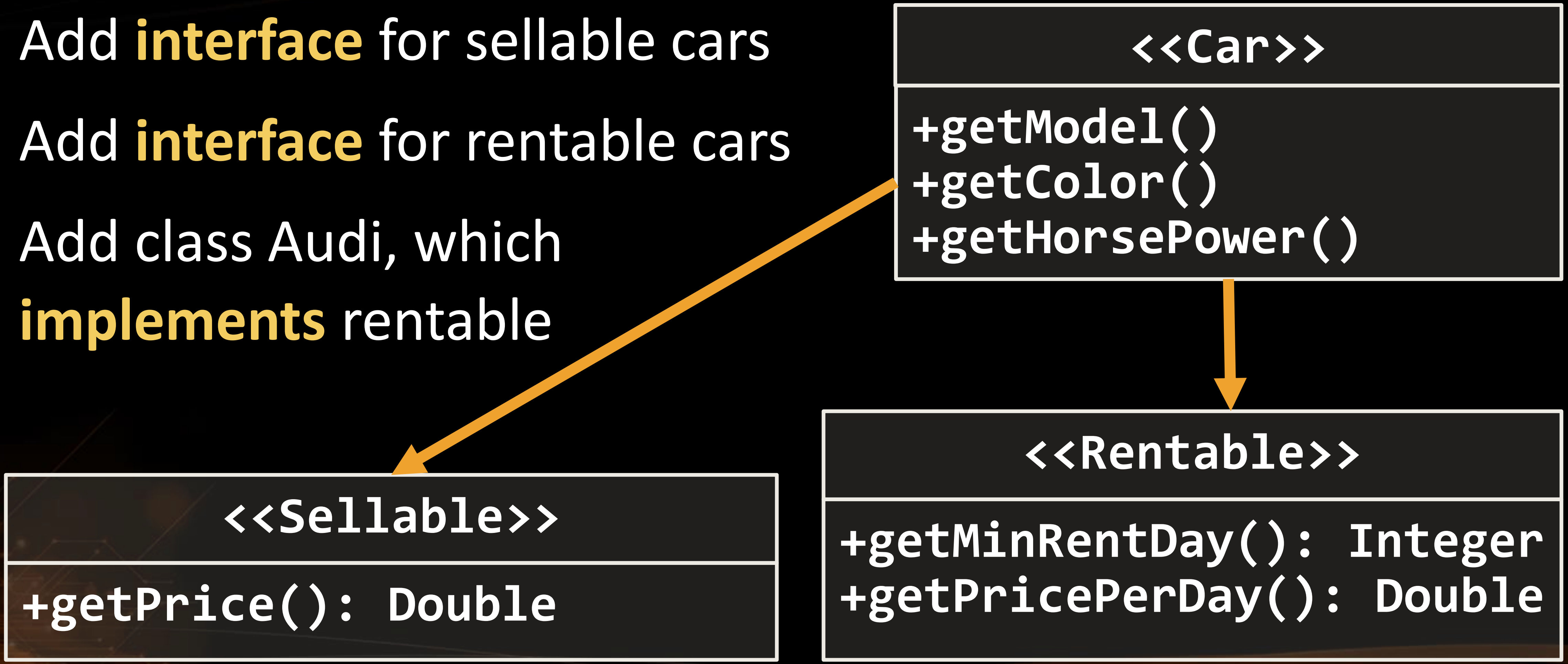
```
class Circle implements Printable {
public void print() {
  System.out.println("Hello");
}

public void show() {
  System.out.println("Welcome");
}
```



Problem: Car Shop Extended

- Refactor your first problem code
- Add **interface** for sellable cars
- Add **interface** for rentable cars
- Add class Audi, which **implements** rentable



Solution: Car Shop Extended

```
public interface Sellable extends Car {
    Double getPrice();
}
```

```
public interface Rentable extends Car{
    Integer getMinRentDay();
    Double getPricePerDay();
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/498#0

Solution: Car Shop Extended(2)

```
public class Audi implements Rentable {
  public String getModel() { return this.model; }
  public String getColor() { return this.color; }
  public int getHorsePower() { return this.horsePower; }
  public Integer getMinRentDay() {
    return this.minDaysForRent; }
  public Double getPricePerDay() {
    return this.pricePerDay; }
}
```

Default Method

- Since Java 8 we can have **method body** in the **interface**

```
public interface Drawable {
  void draw();
  default void msg() {
    System.out.println("default method:)
  }
}
```

- If you need to Override default method **think about your design**

Default Method (2)

- Implementation is **not needed** for **default methods**

```
class TestInterfaceDefault {
  public static void main(String args[]) {
    Drawable d=new Rectangle();
    d.draw();  //drawing rectangle
    d.msg();  //default method
  }
}
```

Static Method

- Since Java 8, we can have **static method** in **interface**

```
public interface Drawable {
  void draw();
  static int cube(int x) { return x*x*x; }
}
```

```
public static void main(String args[]){
  Drawable d = new Rectangle();
  d.draw();
  System.out.println(Drawable.cube(3));}   //27
```

Problem: Say Hello

- Design a project, which has:
  - **Interface** for Person
  - Three implementation for different nationalities
  - Override where needed

```
<<interface>>
<<Person>>
---------------------
+getName(): String
+sayHi()
```

```
<<Person>>
European
---------------------
-name: String
```

```
<<Person>>
Bulgarian
---------------------
-name: String
---------------------
+sayHi(): String
```

```
<<Person>>
Chinese
---------------------
-name: String
---------------------
+sayHi(): String
```

Solution: Say Hello

```
public interface Person {
  String getName();
  default void sayHello() { System.out.println("Hello");}
}
```

```
public class European implements Person{
  private String name;
  public European(String name) { this.name = name; }
public String getName() { return this.name; }
}
```

Solution: Say Hello(2)

```
public class Bulgarian implements Person {
  private String name;
  public Bulgarian(String name) {
    this.name = name;
  }
  public String getName() { return this.name; }
  public void sayHello() {System.out.println("Здравей");}
}
//TODO: Do the same for Chinese
```

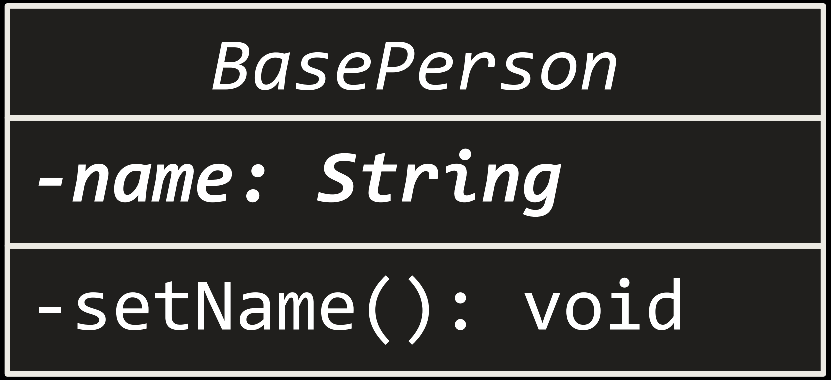
Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/498#2

Interface vs Abstract Class

- Abstract Class
  - Doesn't support **multiple inheritance.**
  - Can have **abstract and non-abstract** methods.
  - Can have **final, non-final, static and non-static** variables.
- Interface
  - Supports **multiple inheritance.**
  - Can have only **abstract, default and static** methods.
  - Can have only **static and final variables.**

Problem: Say Hello Extended

- Refactor the code from the last problem
- Add BasePerson **abstract class**
  - Move in it all **code duplication** from European, Bulgarian, Chinese



```
BasePerson
-name: String
-setName(): void
```

Solution: Say Hello Extended

```
public abstract class BasePerson implements Person{
  private String name;
  protected BasePerson(String name) {
    this.setName(name);
  }
  private void setName(String name) {
    this.name = name;
  }
  @Override
  public String getName() {
    return this.name;
} }
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/498#3

Interfaces and Abstract Class

Live Exercises in Class (Lab)

Summary

- Abstraction
- Interface
  - Implements vs Extends
  - Default and Static methods
- Interface vs Abstract Class

Interfaces And Abstraction

Questions?

https://softuni.bg/courses/

Trainings @ Software University (SoftUni)

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

Software University

SoftUni Foundation

FORUM

License