

Polymorphism

Abstract Classes, Abstract Methods, Override Methods



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

sli.do

#java-advanced

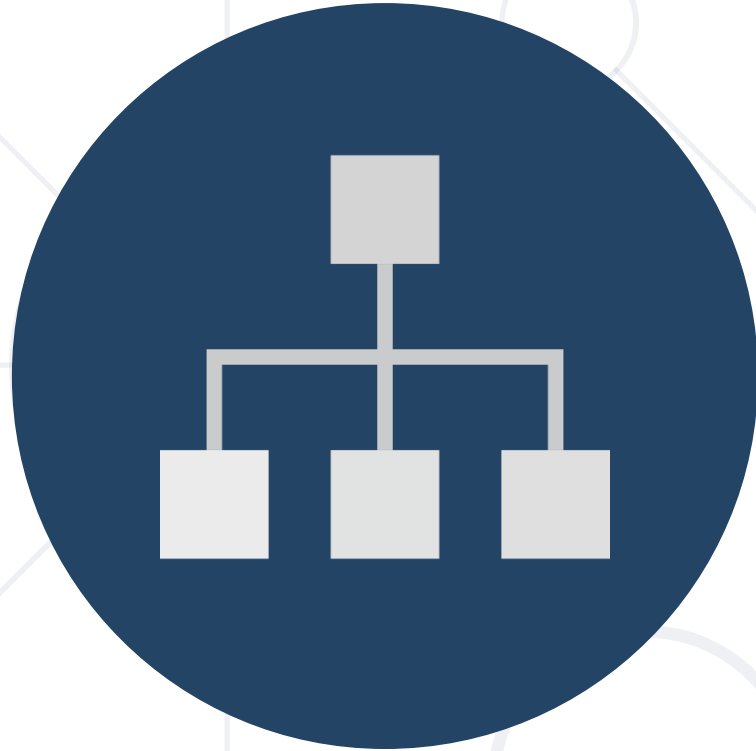
1. Polymorphism

- What is Polymorphism?
- Types of Polymorphism
- **Override** Methods
- **Overload** Methods

2. Abstract Classes

- Abstract Methods

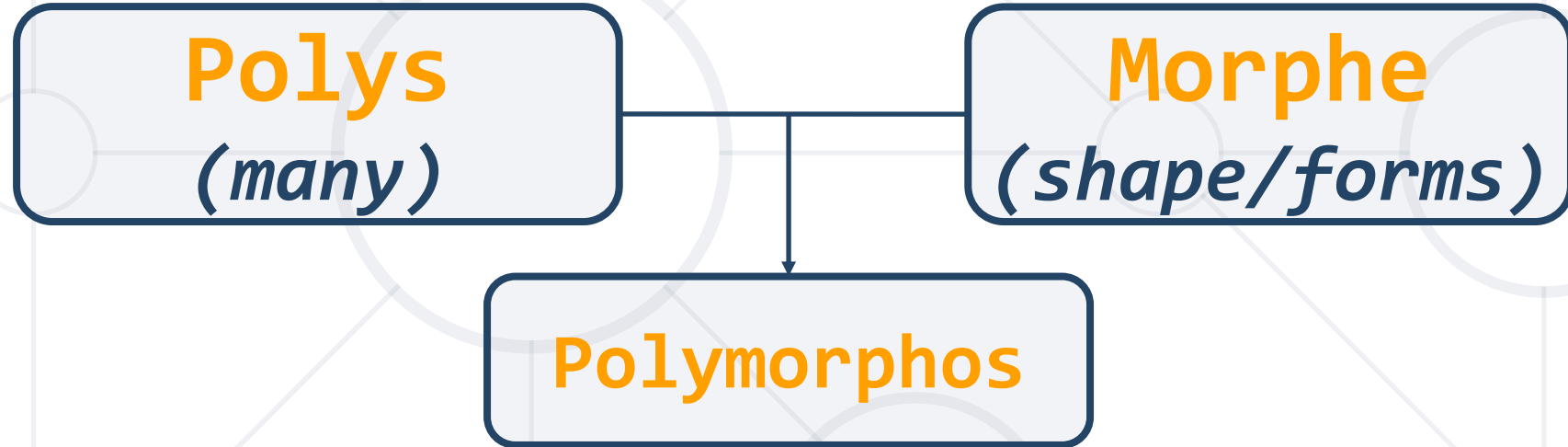




Polymorphism

What is Polymorphism?

- From the Greek



- Such as a word having several different meanings based on the context
- Often referred to as the third pillar of OOP, after encapsulation and inheritance



- The ability of an **object** to take on **many forms**

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

Person **IS-A** Person

Person **IS-A** Mammal

Person **IS-AN** Animal

Person **IS-AN** Object

Reference Type and Object Type

```
public class Person extends Mammal implements Animal {}  
Animal person = new Person();  
Mammal personOne = new Person();  
Person personTwo = new Person();
```

Reference Type

Object Type

- **Variables** are saved in a **reference** type
- You can use only **reference methods**
- If you need an **object method** you need to **cast it or override it**

Keyword – Instanceof

- Check if an **object** is an **instance** of a specific **class**

```
Mammal george = new Person();  
Person peter = new Person();
```

```
if (george instanceof Person) {  
    ((Person) george).getSalary();  
}
```

Check object
type of person

```
if (peter.getClass() == Person.class) {  
    ((Person) peter).getSalary();  
}
```

Cast to object type and use its methods

Types of Polymorphism

- **Runtime** polymorphism

```
public class Shape {}  
public class Circle extends Shape {}  
public static void main(String[] args) {  
    Shape shape = new Circle();  
}
```

Method
overriding

- **Compile time** polymorphism

```
int sum(int a, int b, int c){}  
double sum(Double a, Double b){}
```

Method
overloading

- Also known as **Static Polymorphism**

```
static int myMethod(int a, int b) {}  
static Double myMethod(Double a, Double b) {}
```

Method overloading

- Argument lists could **differ** in
 - Number of parameters
 - The data type of parameters
 - The sequence of Data type parameters

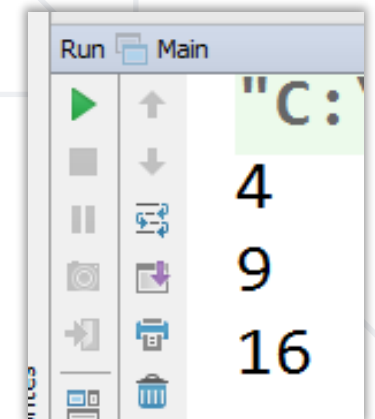
Problem: MathOperation

- Create a class **MathOperation**, which should have method **add()**
- Must be invoked with **two, three or four integers**

MathOperation

```
+add(int a, int b): int  
+add(int a, int b, int c): int  
+add(int a, int b, int c, int d): int
```

```
MathOperation mathOperation = new MathOperation();  
  
System.out.println(mathOperation.add(a: 2, b: 2));  
System.out.println(mathOperation.add(a: 3, b: 3, c: 3));  
System.out.println(mathOperation.add(a: 4, b: 4, c: 4, d: 4));
```



Solution: MathOperation

```
public class MathOperation {  
    public int add(int a, int b) {  
        return a + b;  
    }  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
    public int add(int a, int b, int c, int d) {  
        return a + b + c + d;  
    }  
}
```

Rules for Overloading Method

- **Overloading** can take place in the **same class** or its **subclass**
- **Constructors** in Java can be **overloaded**
- Overloaded methods must have a **different argument list**
- The overloaded method should always be part of the same class (can also take place in a subclass), with the **same name** but **different parameters**
- They may have the **same** or **different return types**

- Using of **override** method

```
public static void main(String[] args) {  
    Rectangle rect = new Rectangle(3.0, 4.0);  
    Rectangle square = new Square(4.0);
```

Method overriding

```
    System.out.println(rect.area());  
    System.out.println(square.area());  
}
```

- Also known as **Dynamic Polymorphism**

```
public class Rectangle {  
    public Double area() {  
        return this.a * this.b;  
    }  
}
```

```
public class Square extends Rectangle {  
    @Override  
    public Double area() {  
        return this.a * this.a;  
    }  
}
```

Method overriding

Rules for Overriding Method

- **Overriding** can take place in **sub-class**
- The **argument list** must be the **same** as that of the **parent method**
- The overriding method must have the **same return type**
- **Access modifier** cannot be more **restrictive**
- **Private**, **static**, and **final** methods can **NOT** be overridden
- The overriding method **must not** throw new or broader **checked exceptions**

The image features a central dark blue circle containing a blue gear-like shape with the text "Abstract Class" inside. This central element is surrounded by a network of light gray circles and lines, suggesting a complex system or architecture. The background is white with a subtle grid of light gray lines.

Abstract
Class

Abstract Classes

Abstract Classes

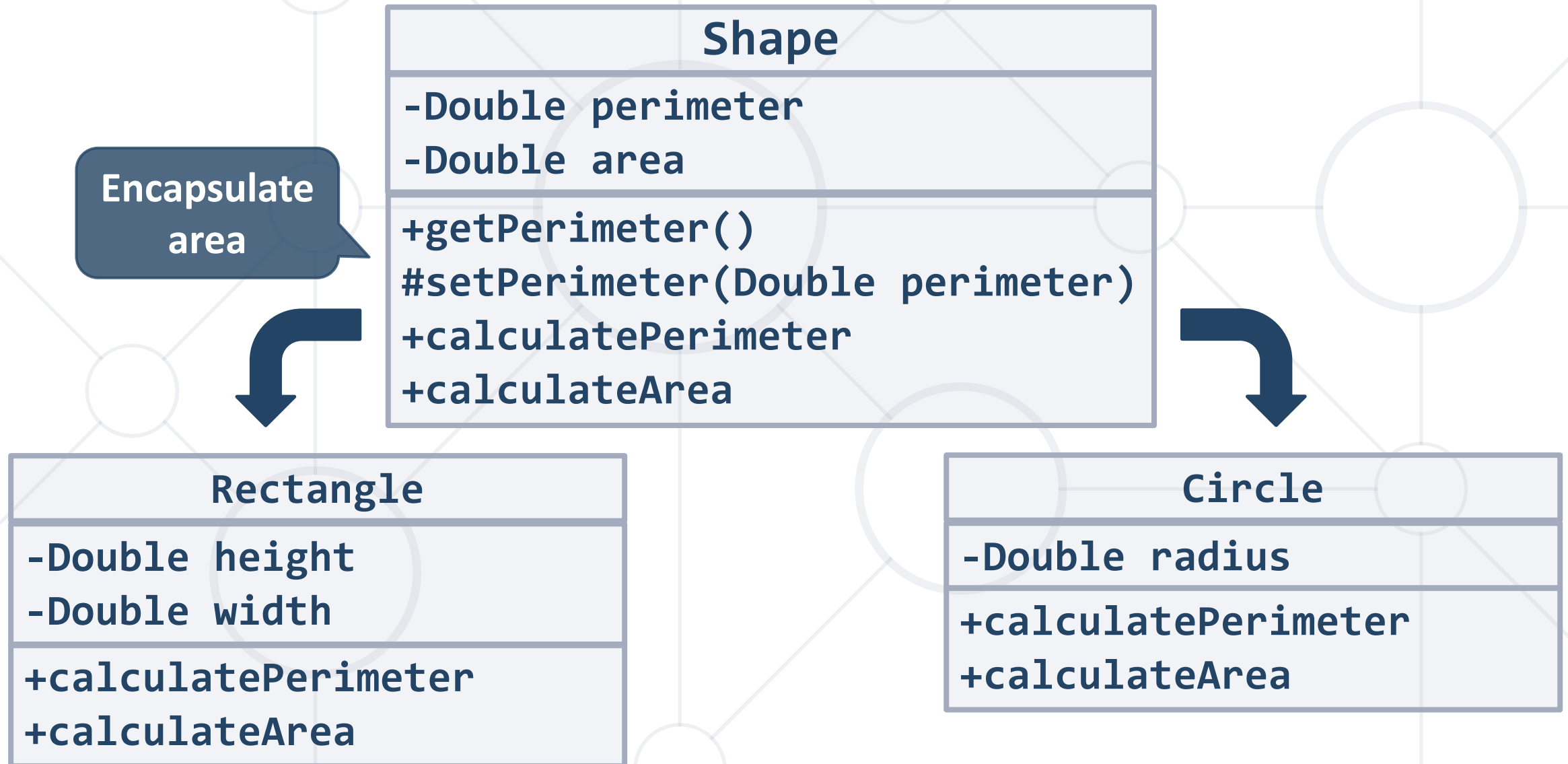
- An abstract class **can NOT be instantiated**

```
public abstract class Shape {}  
public class Circle extends Shape {}  
Shape shape = new Shape(); // Compile time error  
Shape circle = new Circle(); // polymorphism
```



- An **abstract** class may or may not include **abstract methods**
- If it has at least one abstract method, it must be declared **abstract**
- To use an **abstract class**, you need to **inherit it**

Problem: Shapes



```
public abstract class Shape {  
    private Double perimeter;  
    private Double area;  
    protected void setPerimeter(Double perimeter) {  
        this.perimeter = perimeter;  
    }  
    public Double getPerimeter() { return this.perimeter; }  
    protected void setArea(Double area) {this.area = area; }  
    public Double getArea() { return this.area; }  
    protected abstract void calculatePerimeter();  
    protected abstract void calculateArea();  
}
```

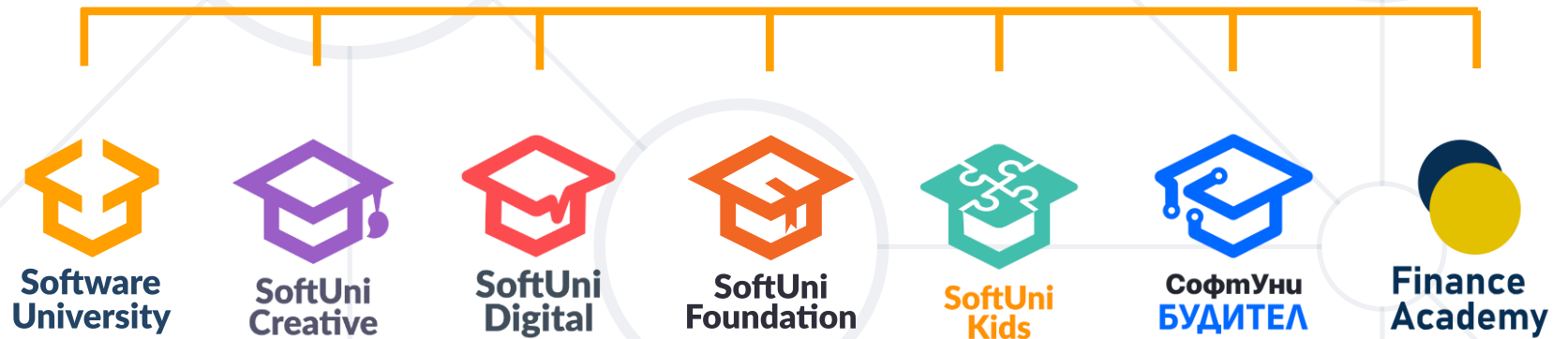
```
public class Rectangle extends Shape {  
    // TODO: Add fields  
    public Rectangle(Double height, Double width) {  
        this.setHeight(height); this.setWidth(width);  
        this.calculatePerimeter(); this.calculateArea(); }  
    // TODO: Add getters and setters  
    @Override  
    protected void calculatePerimeter() {  
        setPerimeter(this.height * 2 + this.width * 2); }  
    @Override  
    protected void calculateArea() {  
        setArea(this.height * this.width); } }  
}
```

```
public class Circle extends Shape {  
    private Double radius;  
    public Circle (Double radius) {  
        this.setRadius(radius);  
        this.calculatePerimeter();  
        this.calculateArea();  
    }  
    public final Double getRadius() {  
        return radius;  
    }  
  
    // TODO: Finish encapsulation  
    // TODO: Override calculate Area and Perimeter  
}
```

- Polymorphism - Definition and Types
- Override Methods
- Overload Methods
- Abstraction
 - Classes
 - Methods



Questions?



SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity



- 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>

