# Systeemontwikkelingsmethoden
# Abstract Classes & Inheritance

Hans Philippi

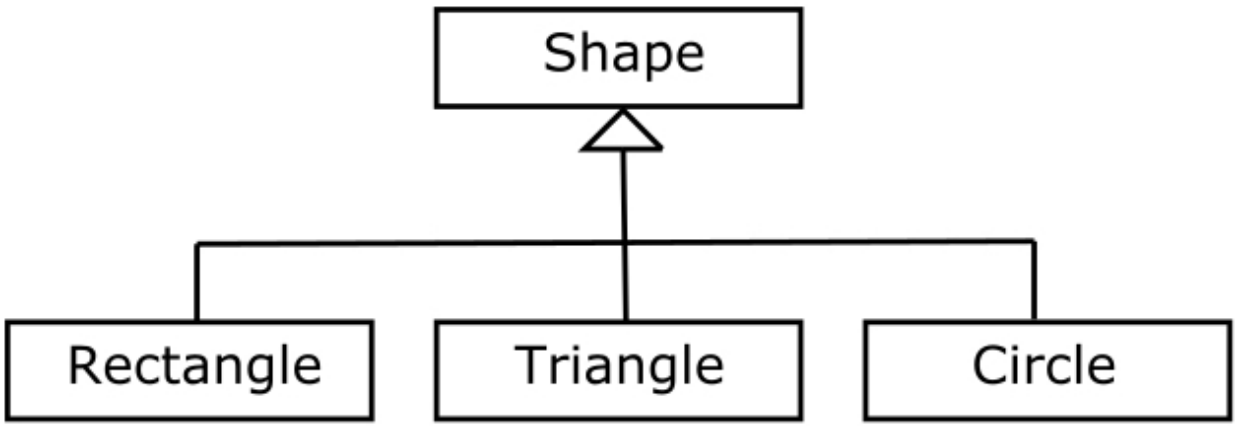
September 11, 2019

# Abstract Classes & Inheritance

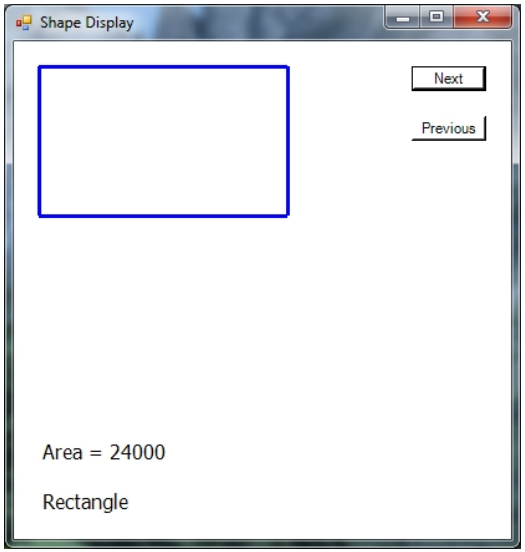
- Abstract class: an OO programming concept (Java, C#)
- Abstract classes support principles of good programming in several ways
- Abstract classes play an important role in *design patterns*

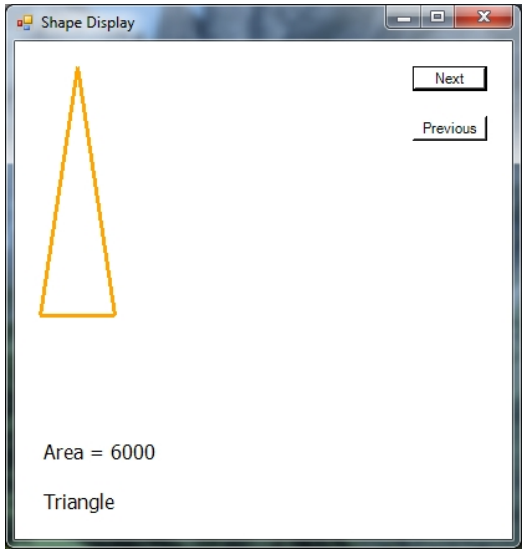
# Abstract Classes

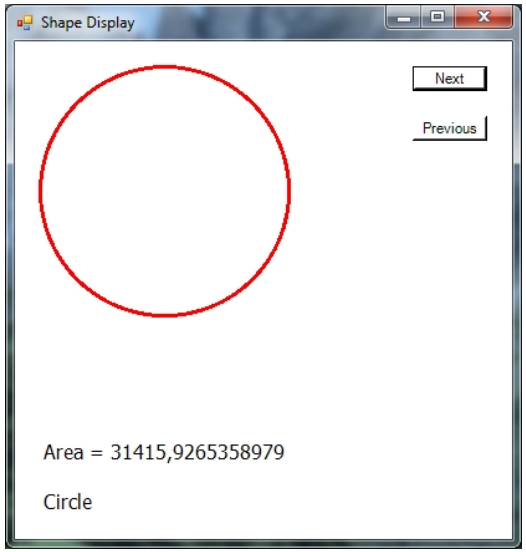
- Abstract classes are related to the concept of generalization (ISA hierarchy) in Entity-Relationship modeling



# Our running example: shapes

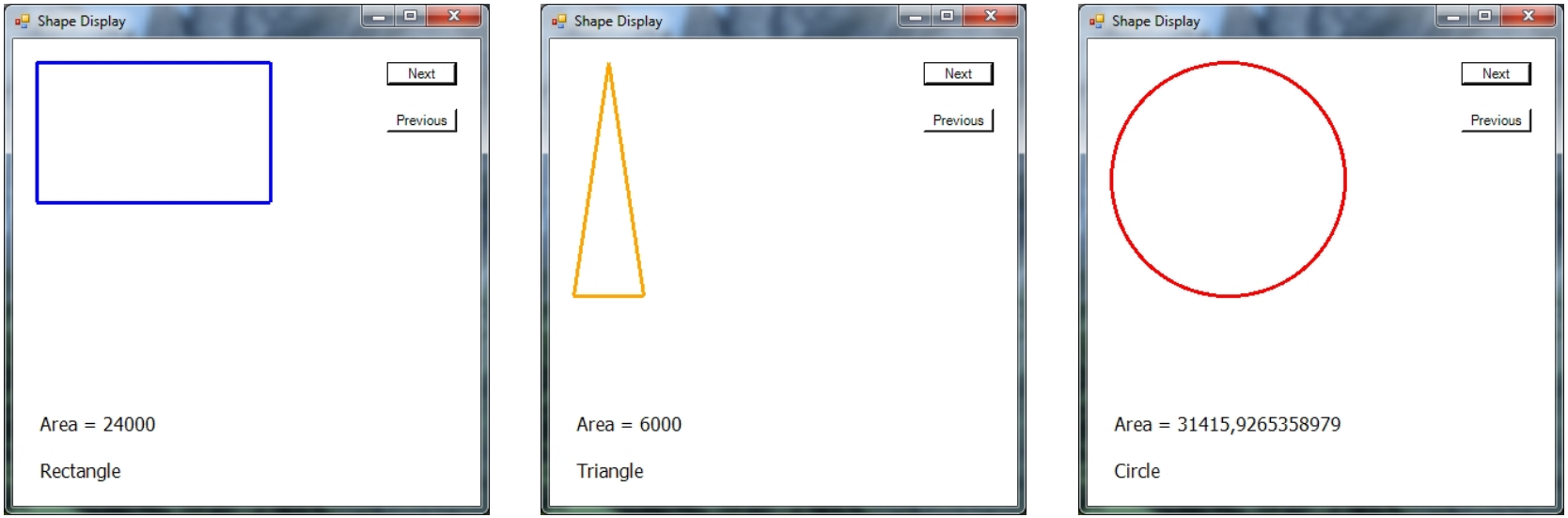



- Shapes are either rectangles, equal sided triangles or circles
- We have a window to scroll through a collection of shapes

# Our running example: shapes




- When scrolling through the collection, we want each shape to be drawn
- Furthermore, we want to see the area and a description of the shape

# Abstract class: the common properties/methods of shapes

```
abstract class Shape
{
    public String Description { get; set; }
    public abstract double Area();
    public virtual void Draw(...)
        { ... }
}
```

- Each shape has a description, simply a string
- Each shape has an area ...
- Each shape can be drawn ...

# Abstract class: the common properties/methods of shapes

```
abstract class Shape
{
    public String Description { get; set; }
    public abstract double Area();
    public virtual void Draw(...)
        { ... }
}
```

- The public properties and methods are often called the *interface* of a class
- Be aware that we do **not** mean the keyword *interface* from Java / C# here

# The code for displaying a shape

```
shape.Draw(...);
DrawString(shape.Area());
DrawString(shape.Description);
```

- The way to deal with shapes in general is defined by `Draw(...)` , `Area()` and `Description`
- New kinds of shapes can be added without changing this code
- ... as long as these three methods/properties are well defined

# Inheritance & subclasses

```
public String Description { get; set; }
```

- Each shape has a `Description`. When we create a concrete subclass, for instance `Rectangle`, we *inherit* this property from the superclass `Shape`.

```
class Rectangle :  Shape
{ ... }
```

- The class specification for `Rectangle` does not contain a property `Description`
- But if we have a `Rectangle` object `r`, we can refer to `r.Description`

# Inheritance, subclasses, abstract method and overriding

```
    public abstract double Area();
```

- Each shape has an area, but the calculation depends on the specific kind of shape. So the calculation should be specified for every concrete shape.

```
class Rectangle :  Shape
{
    public int Width { get; }
    public int Height { get; }
    ...
    public override double Area()
    {
        return this.Width * this.Height;
    }
}
```

# Inheritance, subclasses, abstract method and overriding

```
    public abstract double Area();
```

- Each shape has an area, but the calculation depends on the specific kind of shape. So the calculation should be specified for each concrete shape.

```
class Circle :  Shape
{
    public int Radius { get;}
    ...
    public override double Area()
    {
        return Math.PI * Radius * Radius;
    }
}
```

# Virtual method and overriding

```
abstract class Shape
{
    ...
    protected Color edgeColor;
    protected Pen pen;
    public virtual void Draw(...)
        { pen = new Pen(this.edgeColor, 3); }
    ...
}
```

- Each shape will be drawn by the same pen, defined in the abstract class Shape. Further details will be different for every concrete shape.

# Virtual method and overriding

- Each shape will be drawn by the same pen, defined in the abstract class Shape. Further details will be different for every concrete shape.

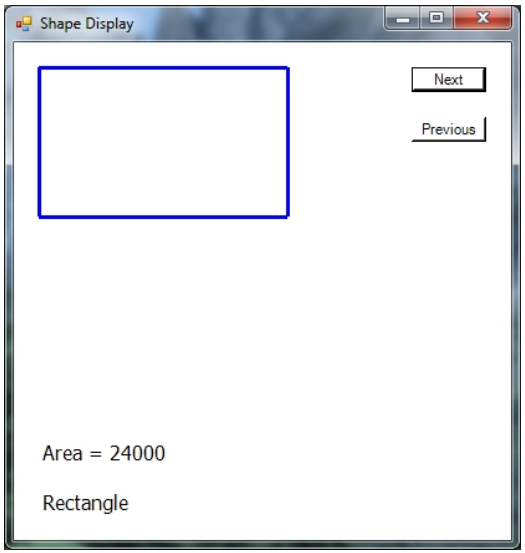
```
class Triangle :  Shape
{      ...
    public override void Draw(...)
    {
        base.Draw(...);
        // the pen is defined
        // as in the abstract superclass
        ...
        // code for drawing a triangle
    }
    ...
```

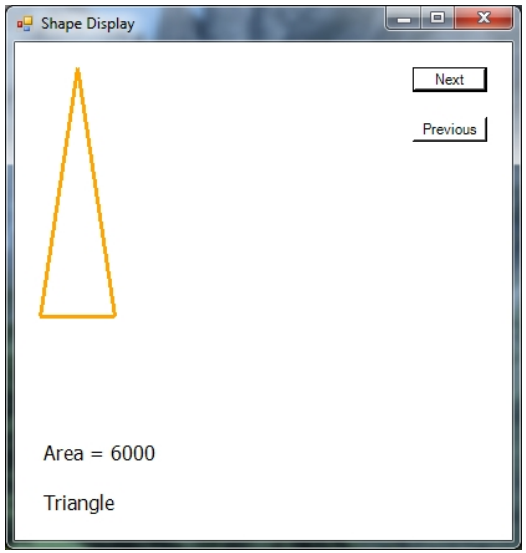
# Abstract Classes and principles of good programming

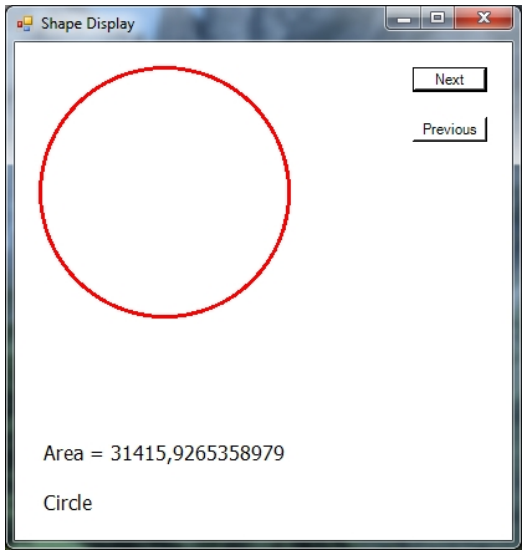
Abstract classes provide a way to apply several principles of good programming:

- *"Avoid replication of code"*
- *"Program to an interface, not to an implementation"*
- *"Find what varies, and encapsulate it"*

# Principles of good programming



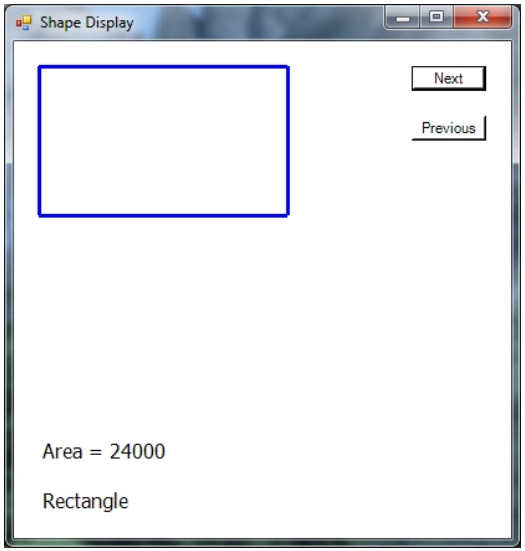

- *"Avoid replication of code"*
- There are common aspects in dealing with the different kinds of shapes, for instance the property Description
- These common aspects should not lead to code duplication
- Code duplication leads to horrible issues when maintaining and adapting code

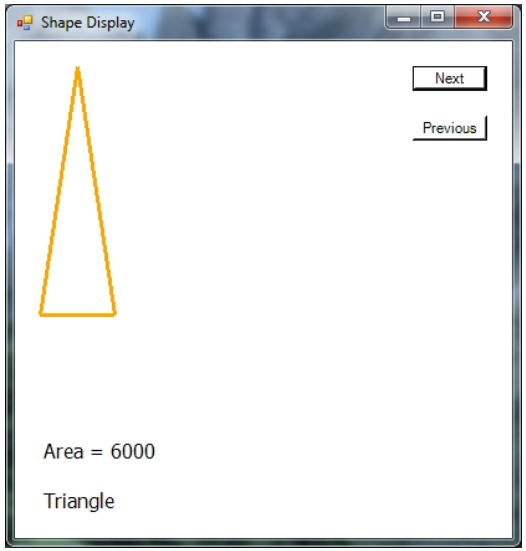
# Abstract Classes and principles of good programming

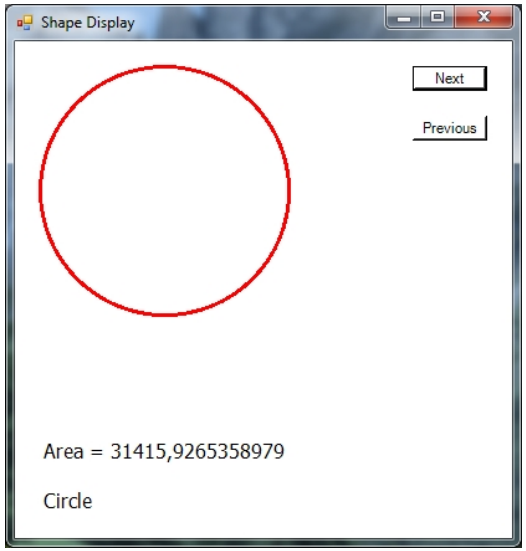
*"Program to an interface, not to an implementation"*

- The properties and behaviour of objects should be described clearly, at the right level of abstraction
- This public set of properties and behaviour defines the *interface* of an abstract class
- The naughty details of implementing this behaviour should be hidden from the user of the object/class: *encapsulation*

# Principles of good programming

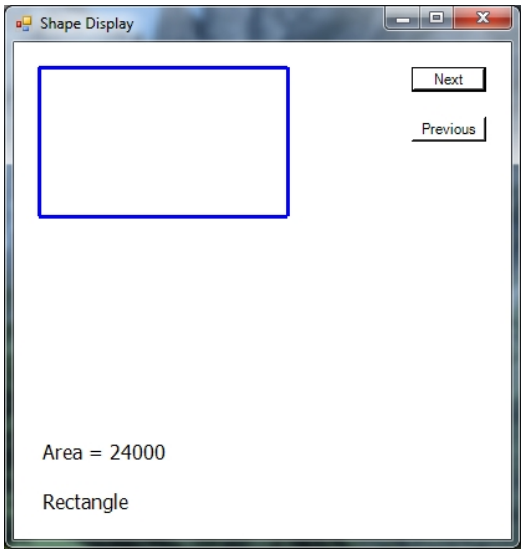


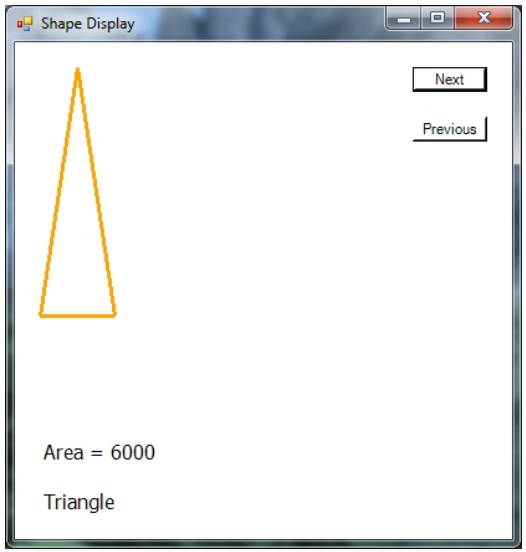



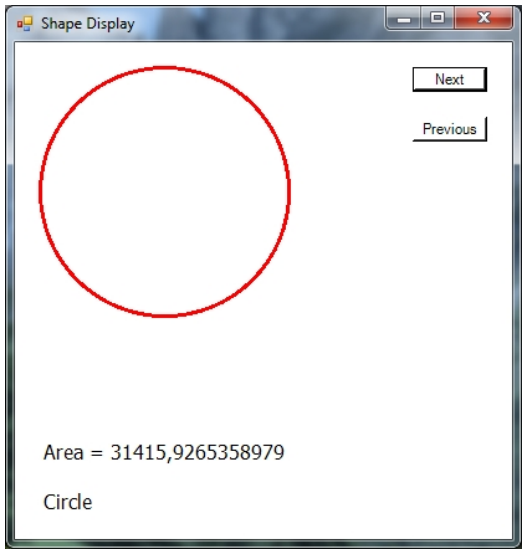




- *"Program to an interface, not to an implementation"*
- Take care that implementation details of the different kinds of shapes are hidden at the levels where they are not relevant
- This supports maintainability and extensibility of your software

# Principles of good programming




- *"Find what varies, and encapsulate it"*
- This principle enables you to handle different shapes by using *only* the common properties
- *Encapsulation:* the naughty implementation details of the different kinds of shapes are hidden at the levels where they are not relevant

# Abstract Classes: some remarks

- You cannot make concrete *instances* of abstract classes. Always use a concrete subclass.
- This piece of code is correct:

```
Shape shape1 = new Circle(160, Color.Green);
Shape shape2 = new Rectangle(80, 60, Color.Red);
Circle shape3 = new Circle(120, Color.Yellow);
```

- You see that methods `Draw` and `Area`, as defined in `Shape`, are applied to different kinds of objects: rectangles, circles and triangles. This phenomenon is called *polymorphism*.

# Abstract Classes: some remarks

- The notion of the keyword *interface* [1] in C# is in some ways similar to the notion of *abstract class*.
- Difference 1: an *interface* is an empty shell; all methods in an *interface* are fully abstract.
- Difference 2: a class may have no more than one (abstract) superclass. However a class may implement more than one *interfaces*.

[1] Warning: we switched to the specific C#-meaning of the word

# Abstract Classes: exercise for this afternoon

- A full description of the exercise can be found on GitHub
- The full code of our example is also available on GitHub
- Exercise: when displaying shapes, also show the circumference
- When you are finished, demonstrate your program to the teaching assistant

# Abstract Classes: epilog

- We have met with some (new) notions:
- Abstract Class, Inheritance, Subclass, Superclass, Encapsulation
- We have seen three rules of thumb regarding good design
- We will study *design patterns* to support making great software
- Abstract classes are needed to understand and implement design patterns
- In fact, design patterns turn out to be applications of our three rules