CSY2030
Systems Design & Development

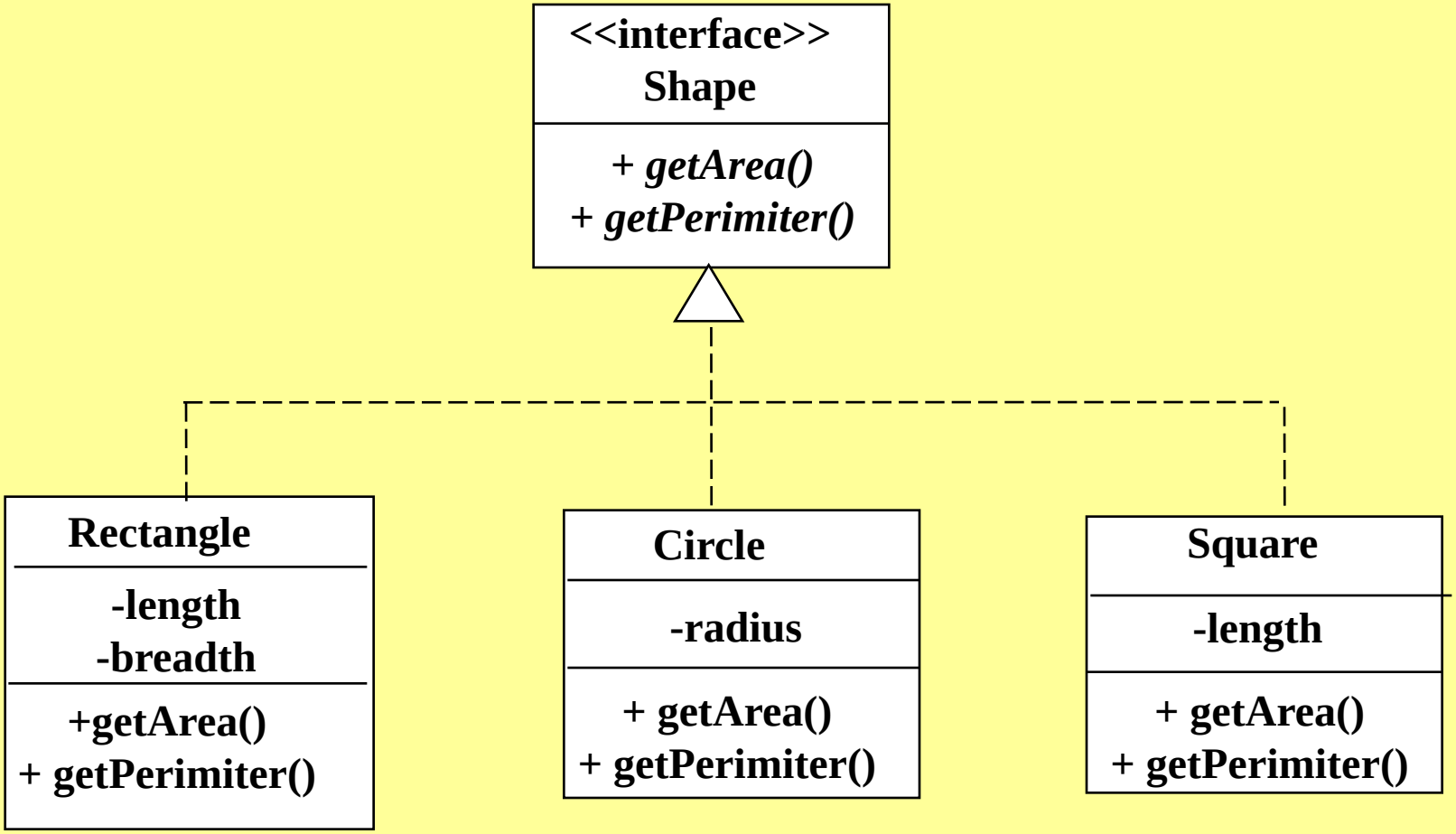
Interface and Abstract Classes

Interface and Abstract Classes

- An *Interface* class is a class with no attributes and a set of operations which have no implementation
  - It is up to the subclasses to provide the implementation
  - Represented by class name preceded by <<**interface**>> and connections are dashed lines
- *Abstract* classes are similar to interface classes but they may have implementations defined for some of their operations and they may also have attributes
  - Represented by class name preceded by <<**abstract**>> and connections are dashed lines

Interface Class

Example



Interface Classes

- Interfaces allow you to declare a base type without defining any behaviour
- They only contain method headers not the implementation
- Any class that implements the interface must provide the methods required by the interface

Interface Classes

- This is useful when multiple classes have the same method but the implementation is different for each one
  - For example, all shapes have a *getArea()* method but it's different for each shape
    - For rectangles it's *length*breadth*
    - For circles its *pi*radius*radius*
    - For squares it's *length*length*
    - etc

Interface Class

- Interfaces can only contain method headers
- Interfaces cannot define instance variables (but can define contstants)
- Interfaces do not contain a method body

```
public interface Shape {
    public double getArea();
    public double
    getPerimeter();
}
```

Note the semicolon instead of the opening brace {

Interface Class

- To use an interface use the **implements** keyword like you would *extends* for superclasses

```
public class Rectangle implements Shape {
```

- Once you implement an interface, you must supply the methods (with the same arguments) described in the interface
- If you implement an interface and do not provide the methods from the interface the program will not compile

Interface Class

```
public class Rectangle implements Shape {

    private double width;
    private double length;

    public Rectangle(double width, double length) {
        this.width = width;
        this.length = length;
    }

    public double getWidth() {
        return width;
    }

    public double getLength() {
        return length;
    }

}
```

This will not compile. By Implementing the *Shape* Interface, you are saying the class will provide the methods *getArea()* and *getPerimeter()*

Interface Class

```
public class Rectangle implements Shape {
    private double width;
    private double length;

    public Rectangle(double width, double length) {
        this.width = width;
        this.length = length;
    }


    public double getWidth() {
        return width;
    }

    public double getLength() {
        return length;
    }

    public double getArea() {
        return width * length;
    }

    public double getPerimeter() {
        return (width*2) + (length*2);
    }
}
```

This will compile now that the methods have been added

Interface Class

- Interfaces can be used as types for variables and arguments

```
public static void showArea(Shape shape) {
    System.out.println("The area is " + shape.getArea());
}
```

- Because the interface says anything that implements the *Shape* interface must provide a *getArea()* method, anything passed into the method is guaranteed to have a *getArea()* method

Interface Class

- We have already defined the *Rectangle* class that implements the *Shape* interface
  - Lets now define a *Triangle* class and a *Circle* class that also implement the *Shape* interface
  - This will allow us to see how we can have different implementations of the same methods from the interface

Interface Class

```
public class Triangle implements Shape {
	private double side1;
	private double side2;
	private double side3;

	public Triangle(double side1, double side2, double side3) {
		this.side1 = side1;
		this.side2 = side2;
		this.side3 = side3;
	}

	public double getArea() {
		double p = getPerimeter()/2;
		return Math.sqrt(p * (p - side1) * (p - side2) * (p - side2));
	}

	public double getPerimeter() {
		return side1 + side2 + side3;
	}
}
```

Interface Class

```
public class Circle implements Shape {

    private double radius;

    public Circle(double radius) {
       this.radius = radius;
    }

    public double getArea() {
        return Math.PI * (radius*radius);
    }

    public double getPerimeter() {
         return 2 * Math.PI * radius;
    }
}
```

Interface Class

```
public class Program {
    public static void main(String[] args) {

        Rectangle rectangle1 = new Rectangle(24, 2);
        Circle circle1 = new Circle(5.2);
        Triangle triangle1 = new Triangle(2, 3, 6);

        showArea(rectangle1);
        showArea(circle1);
        showArea(triangle1);
    }


    public static void showArea(Shape shape) {
        System.out.println("The area is " + shape.getArea());
    }
}
```

All three types (Rectangle, Circle and Triangle) implement the shape interface so can be passed into a method that requires a shape instance as a argument

Because anything that implements the shape interface must provide a getArea() method getArea() can be called on anything that implements the shape interface

Interface Class

- A class that implements an interface can have any constructor
- The constructor is not declared in the Interface
- This avoids problems that occur with inheritance where the super constructor must be called

Interface Class

- When an argument has the type *Shape*, it can be any class that implements it.
- Java provides a way of getting the class name of any shape:
  - *object.getClass().getSimpleName()*
- If *object* is an instance of *Rectangle* it will return *Rectangle*, if it's an instance of *Circle* it will return *Circle*, etc..

Interface Class

```
public class Program {
   public static void main(String[] args) {

         Rectangle rectangle1 = new Rectangle(24, 2);
         Circle circle1 = new Circle(5.2);
         Triangle triangle1 =  new Triangle(2, 3, 6);

         showArea(rectangle1);
         showArea(circle1);
         showArea(triangle1);
         }

         public static void showArea(Shape shape) {
                  System.out.println("The shape is a " +
                           shape.getClass().getSimpleName() +
                  " and has an area of " + shape.getArea()
                  + " and a perimeter of " + shape.getPerimeter());
         }
 }
```

Output

```
The shape is a Rectangle and has an area of 48.0 and a perimeter of 52.0
The shape is a Circle and has an area of 84.94535306801 and a perimeter of 32.672533385
The shape is a Triangle and has an area of 10.96884240152 and a perimeter of 11.0
```

The *instanceof* Operator

- When objects are passed into methods, because they can be a subclass they may not have the same type as was used in the argument

```
public static void showArea(Shape shape) {
    System.out.println("The shape is a " +
        shape.getClass().getSimpleName() +
        " and has an area of " + shape.getArea()
        + " and a perimeter of " + shape.getPerimeter());
}
```

- The *shape* variable could store an instance of *Rectangle*, *Circle* or *Triangle*

The *instanceof* Operator

- The *instanceof* operator is a boolean operator like “==” or >
- On one side it takes an object variable and on the other it takes a class name

The *instanceof* Operator

```
public static void main(String[] args) {

        Rectangle rectangle1= new Rectangle(24, 2);
        Circle circle1 = new Circle(5.2);
        Triangle triangle1 = new Triangle(2, 3, 6);

        showArea(rectangle1);
        showArea(circle1);
        showArea(triangle1);
        }

        public static void showArea(Shape shape) {
                if (shape instanceof Circle) {
                        System.out.println("Shape is a circle");
                }
                else {
                        System.out.println("Shape is not a circle");
                }
        }
 }
```

Output

```
Shape is not a circle
Shape is a circle
Shape is not a circle
```

Interfaces – Continued

- When using *extends* on classes you may only extend one class at a time
- With interfaces, you can extend as many as you want
- For example you can define interfaces for properties that shapes may have

Interfaces – Continued

```
public interface BorderColour {
    public String getBorderColour();
}

public interface BackgroundColour {
    public String getBackgroundColour();
}
```

Interfaces – Continued

- To define a *Circle* that has a border and a background colour you could extend the circle class and implement the *BackgroundColour* and *BorderColour* interfaces:

```
public class BorderedBackgroundCircle extends Circle implements BackgroundColour,
BorderColour {

        private String borderColour;
        private String backgroundColour;

        public BorderedBackgroundCircle(double radius, String borderColour,
                        String backgroundColour) {
                        super(radius);
                this.borderColour = borderColour;
                this.backgroundColour = backgroundColour;
        }

        public String getBorderColour() {
                return this.borderColour;
        }

        public String getBackgroundColour() {
                return this.backgroundColour;
       }
}
```

Interfaces – Continued

- This isn't possible with classes and extends

```
class CircleWithBackgroundColour extends Circle {
    public String getBackgroundColour() {
    }
}

class CircleWithBorderColour extends Circle {
    public String getBorderColour() {
}
}
```

Interfaces

- In this example we can make a shape with a background colour or border colour, however, we would need to create a class *CircleWithBackgroundAndBorder* that extended circle directly to have both properties
- However, with interfaces we can have a class that:
  - Has a border
  - Has a background
  - Has a background and a border

Interfaces

- This is somewhat possible with inheritance:

```
class CircleWithBackgroundColour extends Circle {
    public String getBackgroundColour() {}
}

class CircleWithBorderColour extends
    CircleWithBorderColour {
    public String getBorderColour() {}
```

- This allows either:
  - A circle with a background colour
  - A circle with a background colour and a border colour
- But it does not allow a circle with just a border colour

Interfaces

- Because a class can implement multiple interfaces, interfaces are a lot more flexible than extending classes
- Some programmers say you should almost never use *extends* and instead only use *implements*
- Doing this is very difficult but does result in more robust and flexible applications.

Abstract Classes

- Abstract classes can be thought of as a cross between a normal class and an interface
- Abstract classes can provide:
  - Field methods
  - Abstract methods
- A “non-abstract” class is known as a “Concrete class”
  - concrete class has a complete definition and no “abstract” methods

Abstract Classes

- Abstract methods, like methods in interfaces do not have an implementation and must be provided by any subclasses

```
public abstract class ColouredShape {
    private String colour;

    public String getColour() {
        return colour;
    }

    public ColouredShape(String colour) {
        this.colour = colour;
    }

    public abstract double getArea();
    public abstract double getPerimeter();
}
```

Classes with abstract methods must be declared abstract

Abstract methods are similar to methods in interfaces but are marked “abstract” and do not have a body

Abstract Classes

- Abstract Classes cannot be initiated

```
public static void main(String[] args) {
    ColouredShape shape1 = new ColouredShape("Blue");
}
```

- Will error because if you tried to do the following there is no code for *getArea()*

```
ColouredShape shape1 = new ColouredShape("Blue");
double area = shape1.getArea();
```

Abstract Classes

- To use an abstract class you have to extend it and provide implementations for the abstract methods

```
public class ColouredCircle extends ColouredShape {
    private double radius;

   public ColouredCircle(String      radius) {
       colour, double super(colour);
       this.radius = radius;
    }
    public double getArea() {
        return Math.PI *
        (radius*radius);
    }

    public double getPerimeter()
    {
        return 2 * Math.PI *
        radius;
```

Abstract Classes

- You can now use the *ColouredCircle* class like any other class

```
ColouredShape shape1 = new ColouredCircle("Blue", 2.4);
double area = shape1.getArea();
```

Summary

- An *Interface* class is a class with no attributes and a set of operations which have no implementation
  - It is up to the subclasses to provide the implementation
- *Abstract* classes are similar to interface classes but they may have implementations defined for some of their operations and they may also have attributes
- Use the *instanceof* operator to test whether an object is an instance of a specified type