Programming 2

Inheritance & Polymorphism

Motivation – Lame Shape Application

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangle[100];
        Circle[] theCircles=new Circle[100];
        Triangle[] theTriangles=new Triangle[100];

        public void addShape(Rectangle r){}
        public void addShape(Triangle t){}
        public void addShape(Circle c){}

        public void draw(){
                for (Rectangle r : theRects)
                        r.draw();
                for (Circle c : theCircles)
                        c.draw();
                for (Triangle t : theTriangles)
                        t.draw();
        }

        /* lots more, e.g. UI-stuff */
}
```

this “graphics-suite” can handle Rectangles, Circles, Triangles

Motivation – Lame Shape Application

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangle[100]
        Circle[] theCircles=new Circle[100];
        Triangle[] theTriangles=new Triangle[10

        public void addShape(Rectangle r){}
        public void addShape(Triangle t){}
        public void addShape(Circle c){}

        public void draw(){
                for (Rectangle r : theRects)
                        r.draw();
                for (Circle c : theCircles)
                        c.draw();
                for (Triangle t : theTriangles)
                        t.draw();
        }

        /* lots more, e.g. UI-stuff */
}
```

three list implementations, very much alike

Motivation – Lame Shape Application

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangl
        Circle[] theCircles=new Circle[10
        Triangle[] theTriangles=new Trian

        public void addShape(Rectangle r)
        public void addShape(Triangle t){
        public void addShape(Circle c){}

        public void draw(){
                for (Rectangle r : theRects)
                        r.draw();
                for (Circle c : theCircles)
                        c.draw();
                for (Triangle t : theTriangles)
                        t.draw();
        }

        /* lots more, e.g. UI-stuff */
}
```

first the Rectangles, then the Circles, then the Triangles.
we do not support different layers!

Motivation – Lame Shape Application

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangle[100];
        Circle[] theCircles=new Circle[100];
        Triangle[] theTriangles=new Triangle

        public void addShape(Rectangle r){}
        public void addShape(Triangle t){}
        public void addShape(Circle c){}

        public void draw(){
                for (Rectangle r : theRects)
                        r.draw();
                for (Circle c : theCircles)
                        c.draw();
                for (Triangle t : theTriangles)
                        t.draw();
        }

        /* lots more, e.g. UI-stuff */
}
```

three times pretty much the same code: call draw() on all instances

Motivation – Lame Shape Application

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangle[100];
        Circle[] theCircles=new Circle[100];
        Triangle[] theTriangles=new Triangle[100];

        public vo
        public vo
        public vo

        public vo
                fo

                fo

                for (Triangle t : theTriangles)
                        t.draw();
        }

        /* lots more, e.g. UI-stuff */
}
```

What changes would be necessary, if we wanted to include more Shapes, e.g. Polygons, Lines, Stars,… ?

Motivation – Lame Shape Application

another array

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangle[100];
        Circle[] theCircles=new Circle[100];
        Triangle[] theTriangles=new Triangle[100];
        Polygon[] thePolys=new Polygon[100];

        public void addShape(Rectangle r){}
        public void addShape(Triangle t){}
        public void addShape(Circle c){}
        public void addShape(Polygon p){}

        public void draw(){
                for (Rectangle r : theRects)
                        r.draw();
                for (Circle c : theCircles)
                        c.draw();
                for (Triangle t : theTriangles)
                        t.draw();
                for (Polygon p : thePolys)
                        p.draw();
        }
}
```

Motivation – Lame Shape Application

```
public class LameShapeApplication {

       Rectangle[] theRects=new Rectangle[100];
       Circle[] theCircles=new Circle[100];
       Triangle[] theTriangles=new Triangle[10
       Polygon[] thePolys=new Polygon[100];

       public void addShape(Rectangle r){}
       public void addShape(Triangle t){}
       public void addShape(Circle c){}
       public void addShape(Polygon p){}

       public void draw(){
              for (Rectangle r : theRects)
                     r.draw();
              for (Circle c : theCircles)
                     c.draw();
              for (Triangle t : theTriangles)
                     t.draw();
              for (Polygon p : thePolys)
                     p.draw();
       }
}
```

another addShape-version

Motivation – Lame Shape Application

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangle[100];
        Circle[] theCircles=new Circle[100];
        Triangle[] theTriangles=new Triangle[100];
        Polygon[] thePolys=new Polygon[100];

        public void addShape(Rectangle r){}
        public void addShape(Triangle t){}
        public void addShape(Circle c){}
        public void addShape(Polygon p){}

        public void draw(){
                for (Rectangle r : theRects)
                        r.draw();
                for (Circle c : theCircles)
                        c.draw();
                for (Triangle t : theTriangles)
                        t.draw();
                for (Polygon p : thePolys)
                        p.draw();
        }
}
```

more of the same : polygons are drawn on top of the rest!

Motivation – Lame Shape Application

```
public class LameShapeApplication {

        Rectangle[] theRects=new Rectangle[100];
        Circle[] theCircles=new Circle[100];
        Triangle[] theTriangles=new Triangle[100];
        Polygon[] thePolys=new Polygon[100];

        public voi
        public voi
        public voi
        public voi

        public voi
                for

                for (Circle c : theCircles)
                        c.draw();
                for (Triangle t : theTriangles)
                        t.draw();
                for (Polygon p : thePolys)
                        p.draw();
        }
}
```

now, we have drawing and list logic implemented four times, plus we still do NOT support layers

Shape Classes

Rectangle

```
- Position
- rotationAngle
- width
- height
- lineStyle
- lineColor
- lineWidth
- fillColor
```

```
+ setPosition(Position):void
+ getPosition(): Position
+ setWidth(double):void
+ getWidth(): double
+ setHeight(double):void
+ getHeight(): double
…
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()
```

Circle

```
- Position
- rotationAngle
- center
- radius
- lineStyle
- lineColor
- lineWidth
- fillColor
```

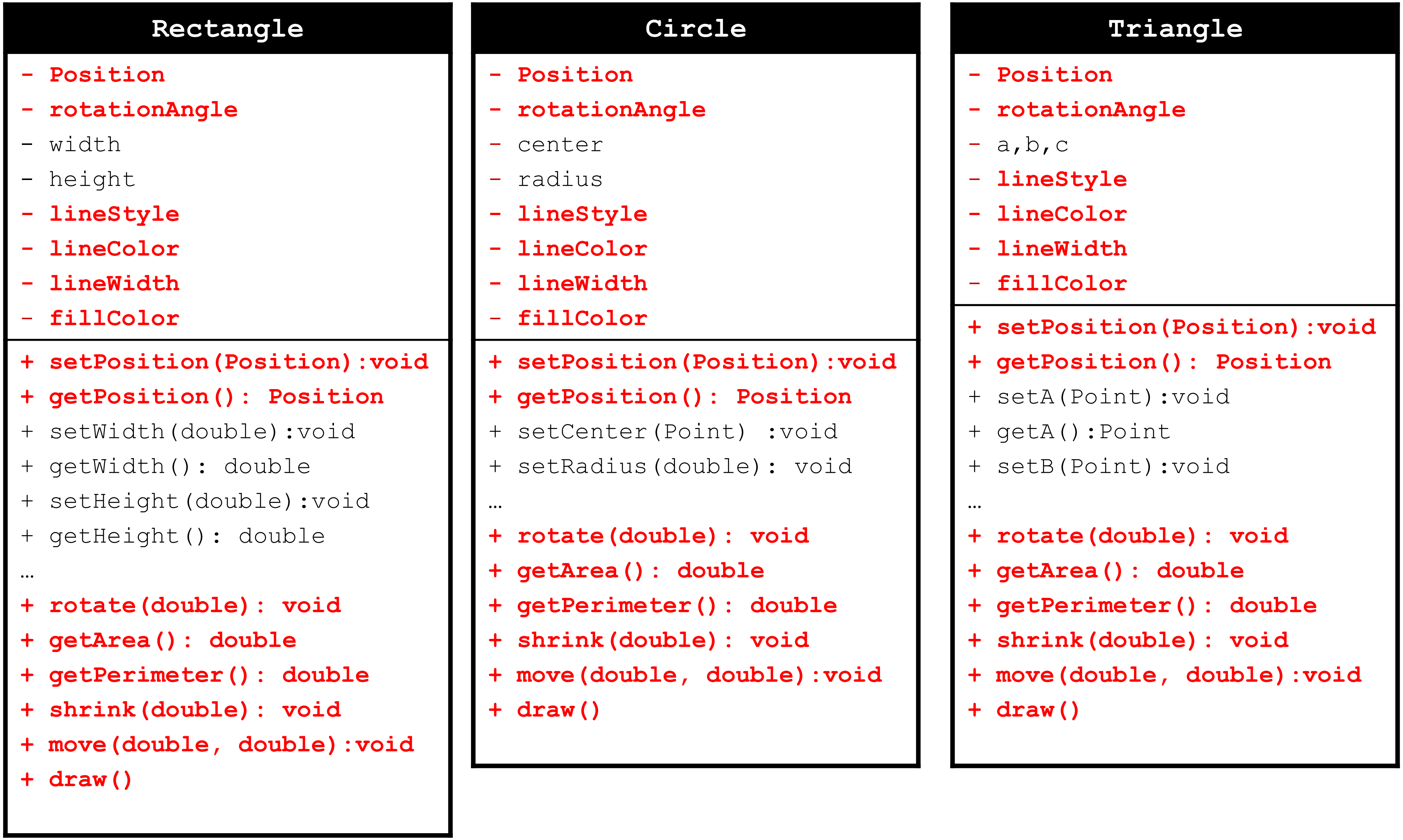
```
+ setPosition(Position):void
+ getPosition(): Position
+ setCenter(Point) :void
+ setRadius(double): void
…
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()
```

Triangle

```
- Position
- rotationAngle
- a,b,c
- lineStyle
- lineColor
- lineWidth
- fillColor
```

```
+ setPosition(Position):void
+ getPosition(): Position
+ setA(Point):void
+ getA():Point
+ setB(Point):void
…
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()
```

Shape Classes – common members

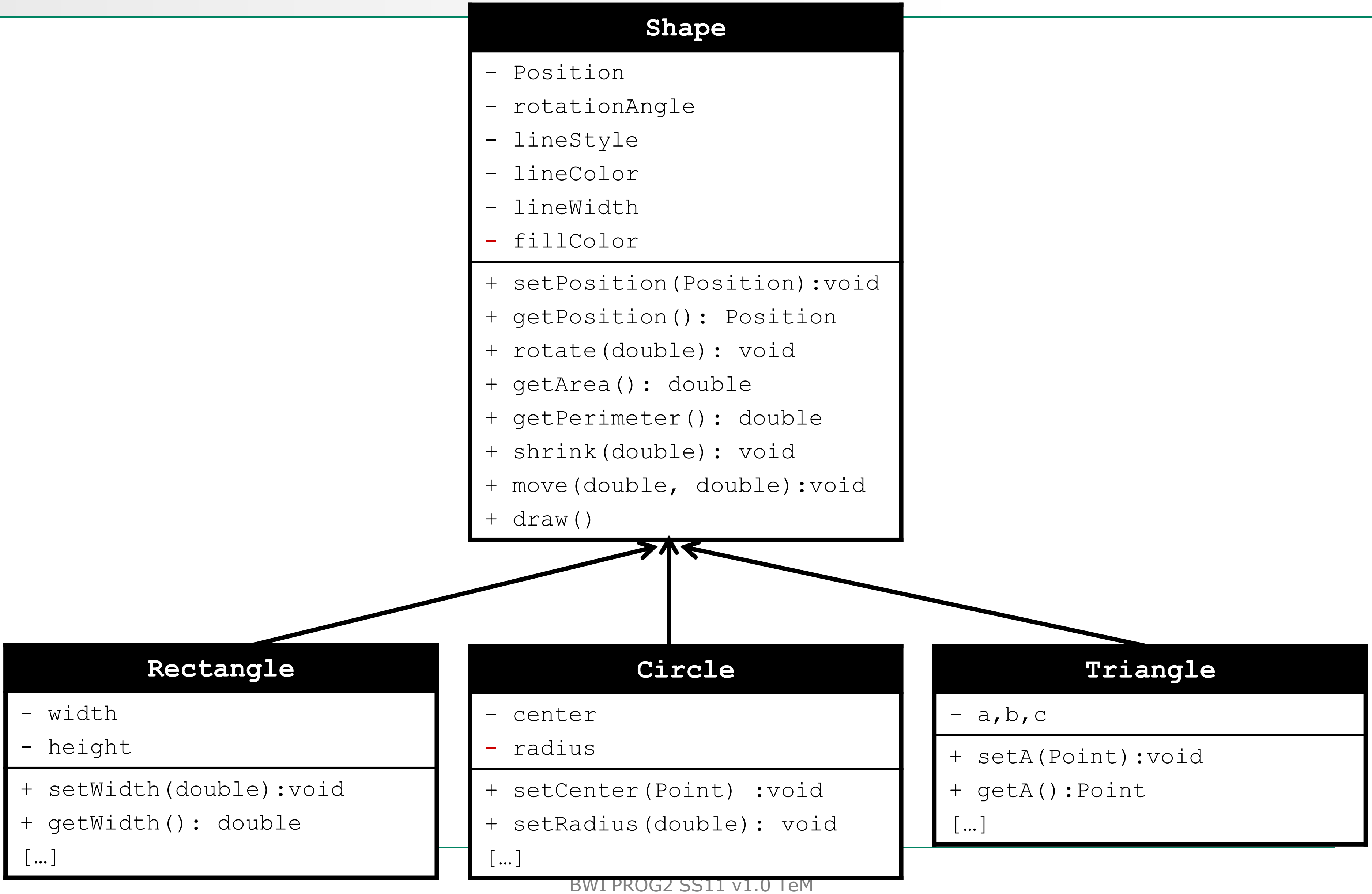


```
Rectangle
- Position
- rotationAngle
- width
- height
- lineStyle
- lineColor
- lineWidth
- fillColor
+ setPosition(Position):void
+ getPosition(): Position
+ setWidth(double):void
+ getWidth(): double
+ setHeight(double):void
+ getHeight(): double
…
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()
```

```
Circle
- Position
- rotationAngle
- center
- radius
- lineStyle
- lineColor
- lineWidth
- fillColor
+ setPosition(Position):void
+ getPosition(): Position
+ setCenter(Point) :void
+ setRadius(double): void
…
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()
```

```
Triangle
- Position
- rotationAngle
- a,b,c
- lineStyle
- lineColor
- lineWidth
- fillColor
+ setPosition(Position):void
+ getPosition(): Position
+ setA(Point):void
+ getA():Point
+ setB(Point):void
…
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()
```

Encapsulate commons in a class



Shape

- Position
- rotationAngle
- lineStyle
- lineColor
- lineWidth
- fillColor

+ setPosition(Position):void
+ getPosition(): Position
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()

Rectangle

- width
- height

+ setWidth(double):void
+ getWidth(): double

[…]

Circle

- center
- radius

+ setCenter(Point) :void
+ setRadius(double): void

[…]

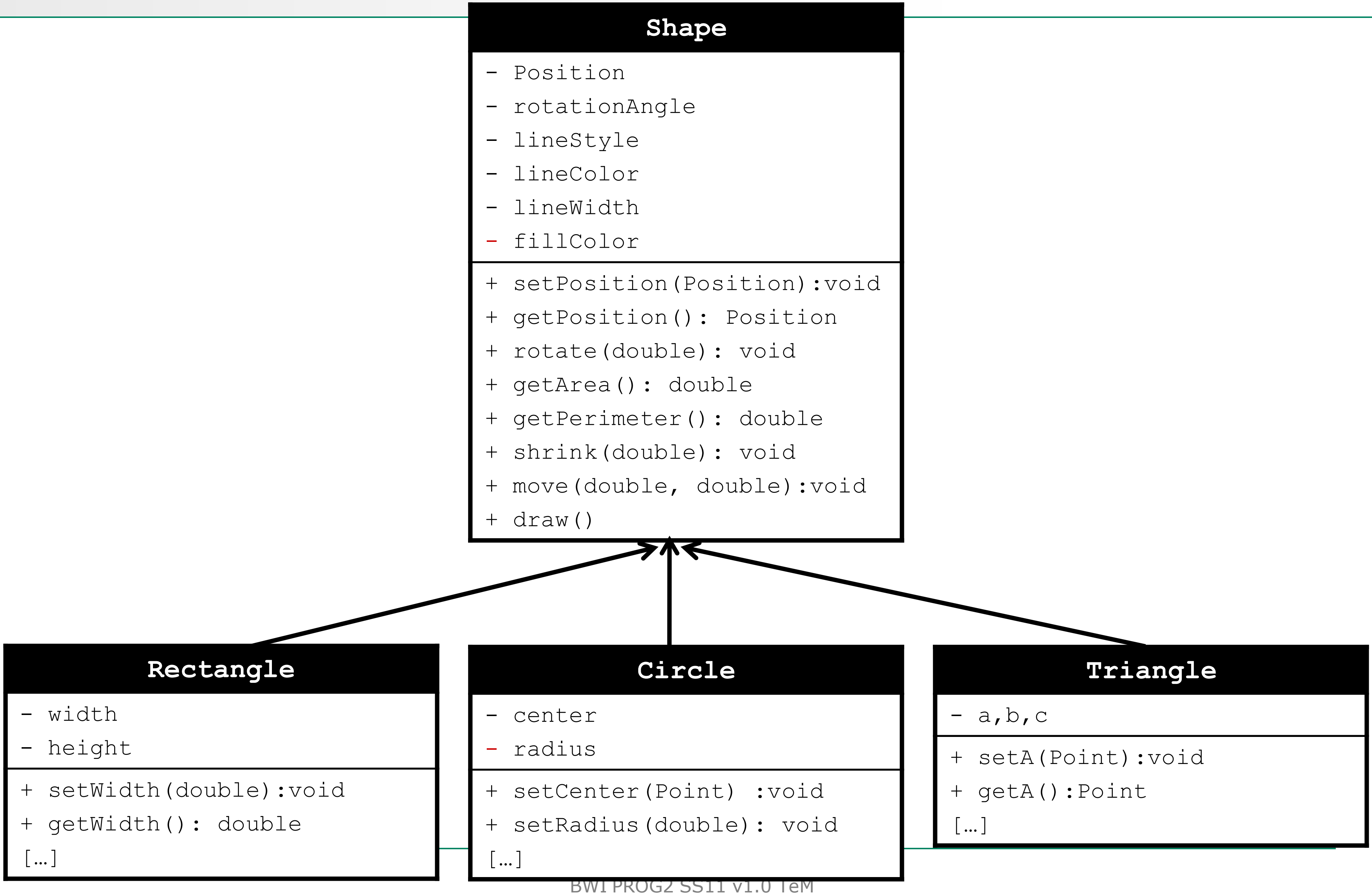
Triangle

- a,b,c

+ setA(Point):void
+ getA():Point

[…]

Encapsulate commons in a class



Shape

- Position
- rotationAngle
- lineStyle
- lineColor
- lineWidth
- fillColor

+ setPosition(Position):void
+ getPosition(): Position
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()

Rectangle

- width
- height

+ setWidth(double):void
+ getWidth(): double

[…]

Circle

- center
- radius

+ setCenter(Point) :void
+ setRadius(double): void

[…]

Triangle

- a,b,c

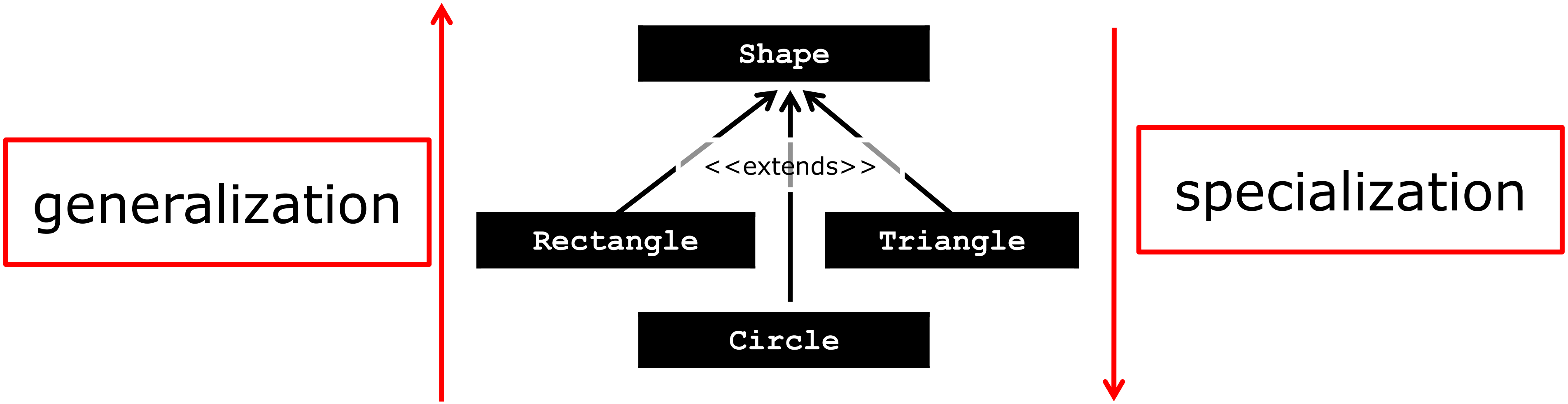
+ setA(Point):void
+ getA():Point

[…]

Inheritance

- Inheritance is the mechanism of creating classes based on existing classes
- Shape encapsulates the common attributes and behavior of Rectangle, Triangle, Circle
- Rectangle, Triangle, Circle extend the attributes and behavior of Shape
- Shape is the base class (superclass)
- Rectangle, Triangle, Circle are subclasses of Shape

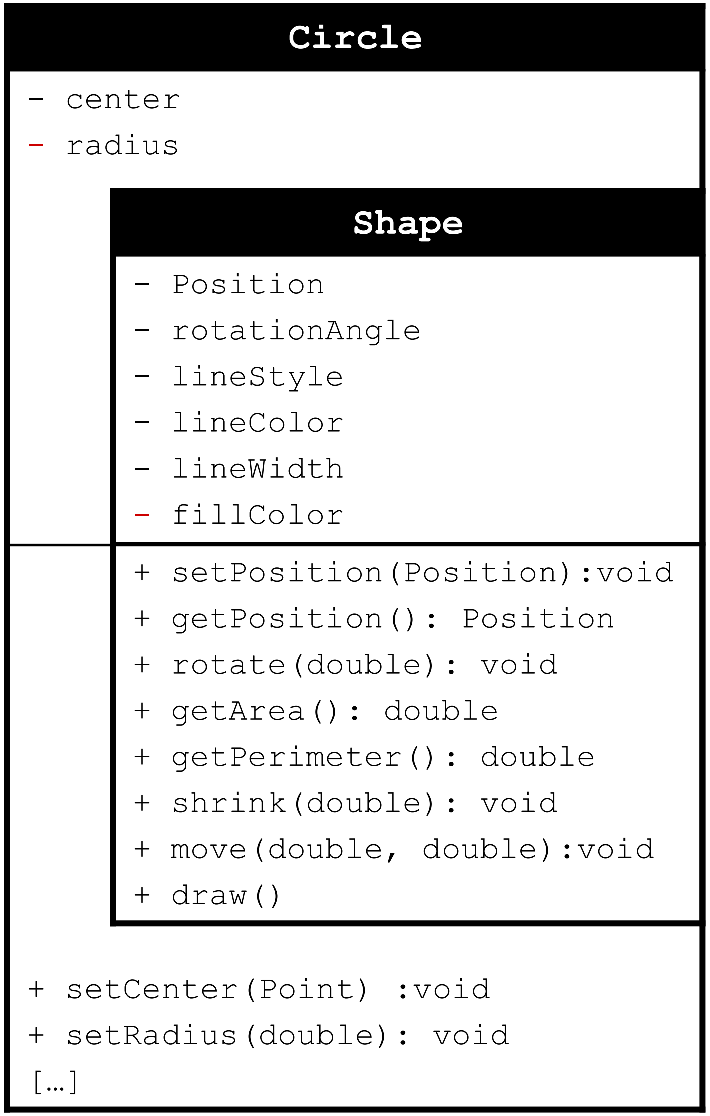
Inheritance Tree



- Rectangle, Circle, Trianlge *IS-A* Shape
- Rectangle, Circle, Trianlge *extend* Shape
- Rectangle, Circle, Trianlge are *subclasses* of Shape
- Shape is the *superclass* of Rectangle, Circle, Trianlge

Circle IS-A Shape

- Circle has everything Shape has, plus some more
- Circle extends Shape
- at heart, Circle is still (also) Shape
- Circle can act as Shape



Circle

- center
- radius

Shape

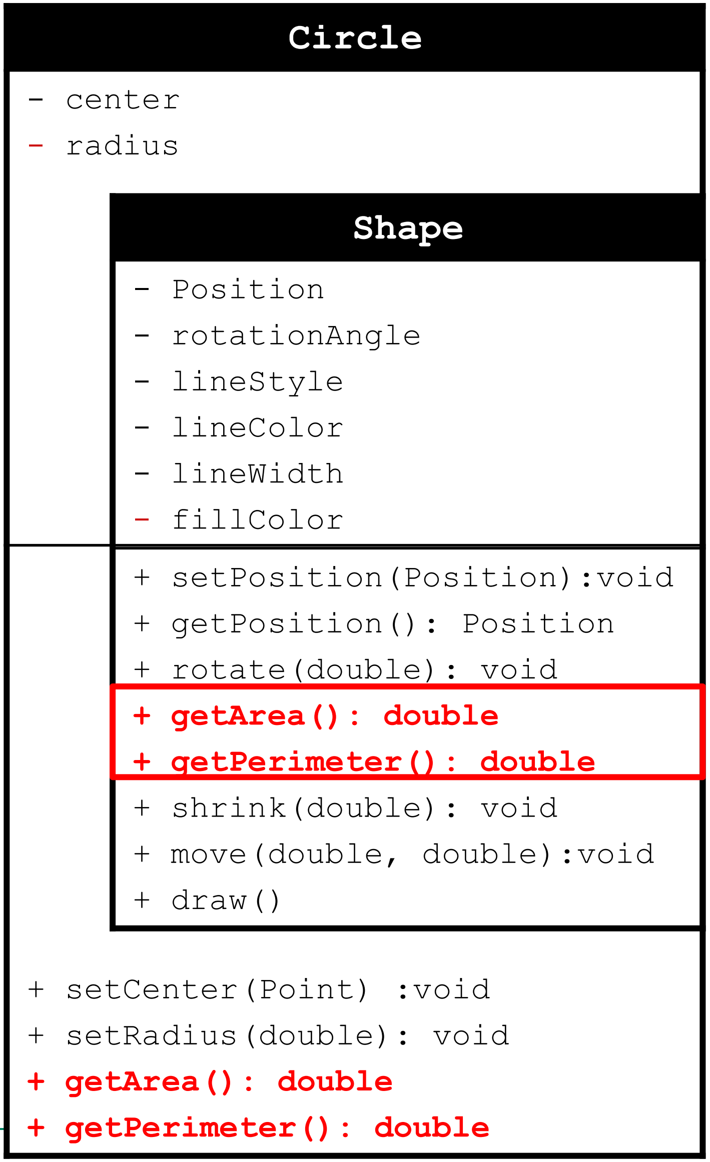
- Position
- rotationAngle
- lineStyle
- lineColor
- lineWidth
- fillColor

+ setPosition(Position):void
+ getPosition(): Position
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()

+ setCenter(Point) :void
+ setRadius(double): void
[…]

Circle redefines Shape behavior

- some methods might need to be reimplemented in Circle
- Circle implements subclass-specific behavior
- superclass interface-contract is obeyed



Circle

- center
- radius

Shape

- Position
- rotationAngle
- lineStyle
- lineColor
- lineWidth
- fillColor

+ setPosition(Position):void
+ getPosition(): Position
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()

+ setCenter(Point) :void
+ setRadius(double): void
+ getArea(): double
+ getPerimeter(): double

Polymorphism

- Polymorphism is the mechanism that
  - a subclass instance can act as a superclass instance
  - a subclass can re-implement a superclass interface with subclass specific behavior
- Circle, Rectangle, Triangle cannot change the getArea-signature (the interface)
- Circle, Rectangle, Triangle can redefine the calculation of the area (the implementation of the interface)

Shape in Java

```
public class Shape {
        private Position position;
        private double rotationAngle;
        private Style lineStyle;
        private Color lineColor;
        private int lineWidth;
        private Color fillColor;

        public Shape() {/**/}
        public Position getPosition() {/**/}
        public void setPosition(Position position) {/**/}
        public void rotate(double angle) {/**/}
        public double getArea() {/**/}
        public double getPerimeter() {/**/}
        public void shrink(double factor) {/**/}
        public void move(double x, double y) {/**/}
        public void draw() {/**/}
}
```

Shape in Java

```java
public class Shape {
/**/
       public Shape() {
              position=new Position();
              rotationAngle=0;
              lineStyle=new Style();
              lineColor=new Color();
              lineWidth=1;
              fillColor=new Color();
       }
/**/
}
```

default position
no rotation
default style, color,
etc..

Shape in Java

```
public class Shape {
/**/
        public void rotate(double angle) {
                rotationAngle+=angle;
                rotationAngle%=360;
        }

        public double getArea() {
                return 0;
        }
        public double getPerimeter() {
                return 0;
        }
        public void move(double x, double y) {
                position.move(x,y);
        }
/**/
}
```

keep in [0,360)

play it safe, we do not know how to calculate area, perimeter of a generic shape

position has move()

Extending Shape in Java

```
public class Circle extends Shape {

        private Point center;
        private double radius;
        public void setRadius(double radius) {
                this.radius= ((radius<0)?-1:1)*radius;
        }


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

        public void move(double x, double y){/
        public void draw(){/**/}
        /**/

}
```

Circle is a subclass of Shape

additional properties+methods

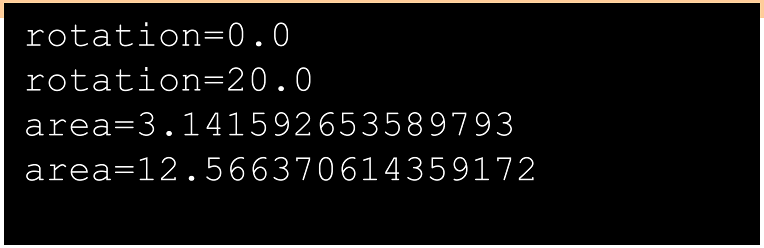
redefine behavior by overriding inherited methods

Circle Application

```
public class CirlceApp {

        public static void main(String[] args) {
                Circle c=new Circle();
                c.setRadius(1);
                TextIO.putln("rotation="+c.getRotationAngle());
                c.rotate(20);
                TextIO.putln("rotation="+c.getRotationAngle());

                TextIO.putln("area="+c.getArea());
                c.setRadius(2);
                TextIO.putln("area="+c.getArea());
        }

}
```

already defined in Shape

Circle-version is called



```
rotation=0.0
rotation=20.0
area=3.141592653589793
area=12.566370614359172
```

Circle acts like a special Shape

```
public class CirlceApp {

        public static void main(String[] args) {
                Shape c=new Circle(1);

                TextIO.putln("rotation="+c.getRotationAngle());
                c.rotate(20);
                TextIO.putln("rotation="+c.getRotationAngle());

                TextIO.putln("area="+c.getArea());
        }

}
```

treat the Circle as a Shape

Circle-version is called

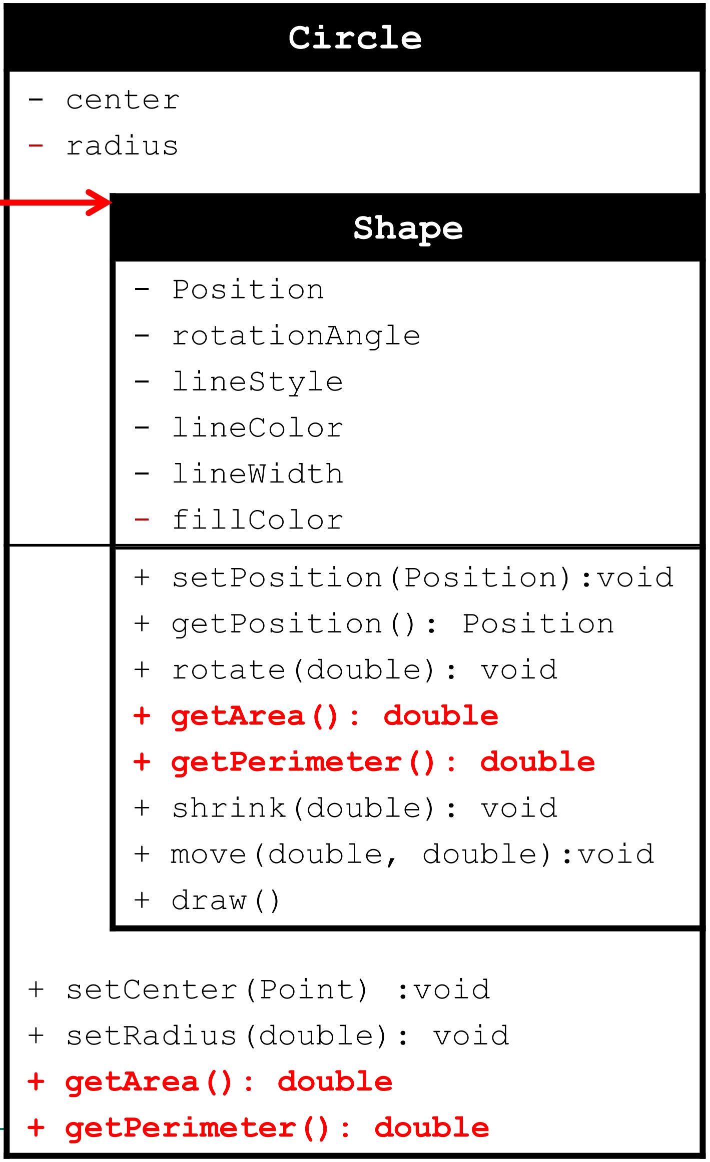
```
rotation=0.0
rotation=20.0
area=3.141592653589793
```

Polymorphism revisited

- subclass instances can act as superclass instances

```
Shape c=new Circle(1);
```

- Circle IS-A Shape
- Circle has everything that is expected of a Shape – it can act as a Shape



Circle

- center
- radius

Shape

- Position
- rotationAngle
- lineStyle
- lineColor
- lineWidth
- fillColor

+ setPosition(Position):void
+ getPosition(): Position
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()

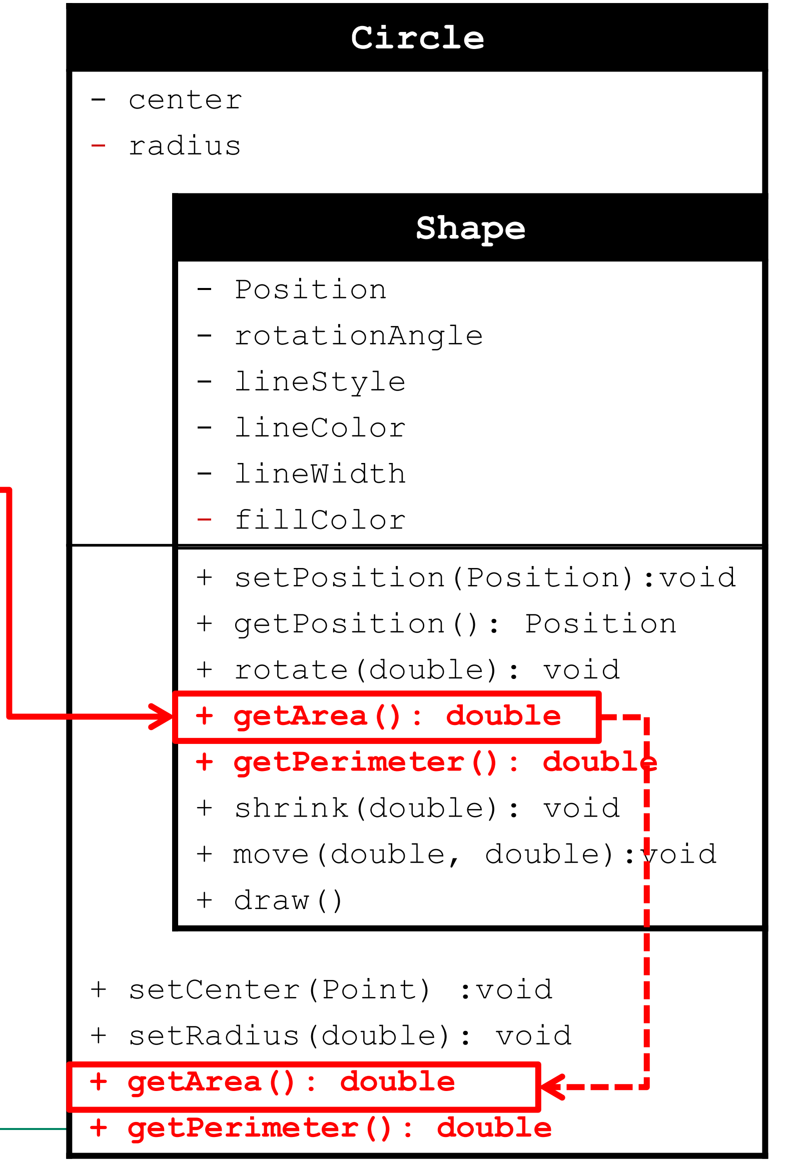
+ setCenter(Point) :void
+ setRadius(double): void
+ getArea(): double
+ getPerimeter(): double

Polymorphism revisited

- call to a Shape method
- overidden in Circle

```
Shape c=new Circle(1);
c.getArea();
```

- most specific version of method is called at runtime



Circle

- center
- radius

Shape

- Position
- rotationAngle
- lineStyle
- lineColor
- lineWidth
- fillColor

+ setPosition(Position):void
+ getPosition(): Position
+ rotate(double): void
+ getArea(): double
+ getPerimeter(): double
+ shrink(double): void
+ move(double, double):void
+ draw()

+ setCenter(Point) :void
+ setRadius(double): void
+ getArea(): double
+ getPerimeter(): double

Polymorphism

- A subclass instance can be stored in a superclass reference
- It is a reference to the superclass-aspect of the instance
- calling a polymorphic method using a superclass reference executes the most specific implementation of the method

Cool Shape Application

```java
public class CoolShapeApplication {

        Shape[] theShapes = new Shape[100];

        public void addshape(Shape s){/**/}

        public void draw(){
                for (Shape s : theShapes)
                        s.draw();
        }
        /* lots more, e.g. UI-stuff */
}
```

one array to hold all different kinds of shapes

Cool Shape Application

```
public class CoolShapeApplication {

        Shape[] theShapes = new Shape[100];

        public void addshape(Shape s){/**/}

        public void draw(){
                for (Shape s : theShapes)
                        s.draw();
        }
        /* lots more, e.g. UI-stuff */
}
```

list logic implemented once – works for all kinds of shapes

Cool Shape Application

```
public class CoolShapeApplication {

        Shape[] theShapes = new Shape[100];

        public void addshape(Shape s){/**/}

        public void draw(){
                for (Shape s : theShapes)
                        s.draw();
        }
        /* lots more, e.g. UI-stuff */
}
```

drawing logic implemented once – for all kinds of shapes.
plus: we finally support layers

Cool Shape Application

```
public class CoolShapeApplication {

        Shape[] theShapes = new Shape[100];

        public void addshape(Shape s){/**/}

        public voi
                fo

        }
        /* lots mo
}
```

What changes would be necessary, if we wanted to include more Shapes, e.g. Polygons, Lines, Stars,… ?

Cool Shape Application

```java
public class CoolShapeApplication {

        Shape[] theShapes = new Shape[100];

        public void addshape(Shape s){/**/}

        public void draw(){
                for (Shape s : theShapes)
                        s.draw();
        }
        /* lots more, e.g. UI-stuff */
}
```

none!
this code works for
ALL FUTURE SHAPES
(that obey the contract)

Cool Shape Application

```
public class CoolShapeApplication {

        Shape[] theShapes = new Shape[100];

        public void addshape(Shape s){/**/}

        public void draw(){
                for (Shape s : theShapes)
                        s.draw();
        }
        /* lots more, e.g. UI-stuff *
}
```

after defining a new Shape subtype, only the code that creates its instances must be aware of the new type

Super-Constructor

```
public class Circle extends Shape {
/**/

        public Circle(){
                super();
                center=new Point();
                radius=1;
        }

        public Circle(double radius){
                this();
                setRadius(radius);
        }
/**/
}
```

call the super constructor to create a default shape and add Circle-specific default values

Super-Constructor

```java
public class Circle extends Shape {
/**/

        public Circle(){
                super();
                center=new Point();
                radius=1;
        }

        public Circle(double radius){
                this();
                setRadius(radius);
        }
/**/
}
```

call to super constructor must be first statement

Super-Constructor

```
public class Circle extends Shape {
/**/

        public Circle(){
                super();
                center=new Point();
                radius=1;
        }

        public Circle(double radius){
                this();
                setRadius(radius);
        }
/**/
}
```

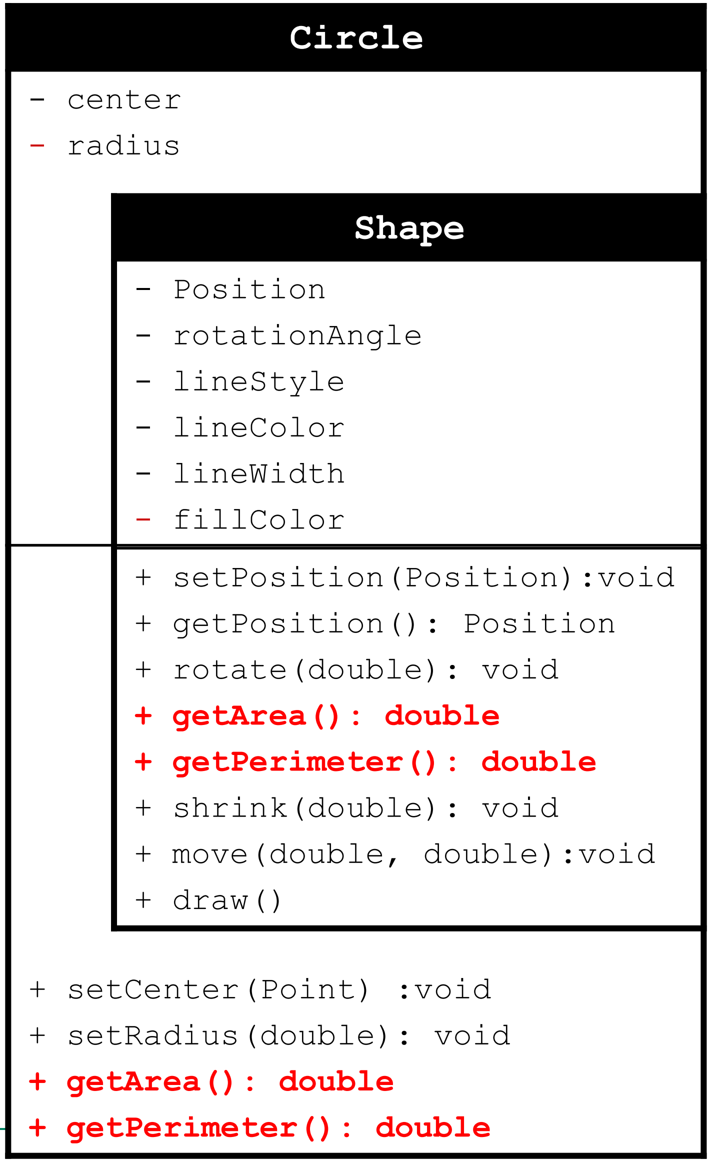
call an overloaded constructor, then set values

Polymorphism revisited

- cannot call a Circle method using a Shape reference

```
Shape c=new Circle(1);
c.setRadius(2);
```

- setRadius is not part of Shape
- Circle lost part of its identity – it is treated as a Shape instance



Circle

- center
- radius

Shape

- Position
- rotationAngle
- lineStyle
- lineColor
- lineWidth
- fillColor

+ setPosition(Position):void
+ getPosition(): Position
+ rotate(double): void
+ **getArea(): double**
+ **getPerimeter(): double**
+ shrink(double): void
+ move(double, double):void
+ draw()

+ setCenter(Point) :void
+ setRadius(double): void
+ **getArea(): double**
+ **getPerimeter(): double**

Late Binding

- An invoked method must be part of the reference-class
- This is checked at compile-time
- If it is not part (even though we are pretty sure that the object has the method) compilation fails
- compiler cannot know which type is stored in a reference at runtime – it could be any (future) subclass
- the check is safe, because any subclass is guaranteed to have all methods of the superclass (interface-contract!)

Late Binding

If the method is part of the reference-definition, compilation proceeds

- WHICH version of a polymorphic method is executed, is decided at runtime
- this is decided based on the actual type of the instance
- the most specific implementation is then executed
- this process is called *Late Binding*

Type casting

- With the cast operator, a reference can be converted

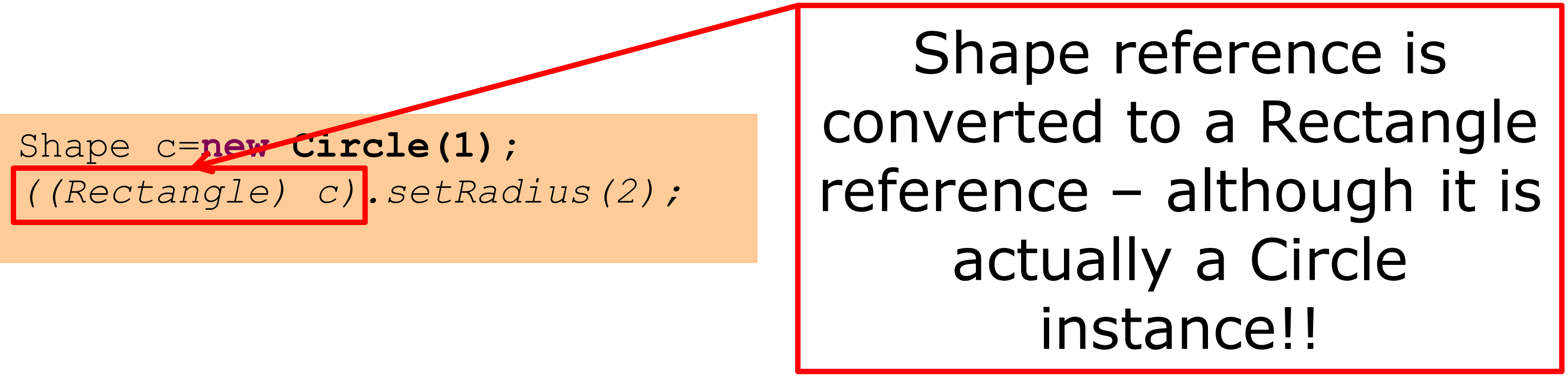
```
Shape c=new Circle(1);
((Shape) c).setRadius(2);
```

Shape reference is converted to a Circle reference

- a reference can be converted to a subtype-reference : this is called “down-casting”
- do NOT cast unless you are at least a 100% positive it works

Type casting

- This is why you should NOT cast



```
Shape c=new Circle(1);
((Rectangle) c).setRadius(2);
```

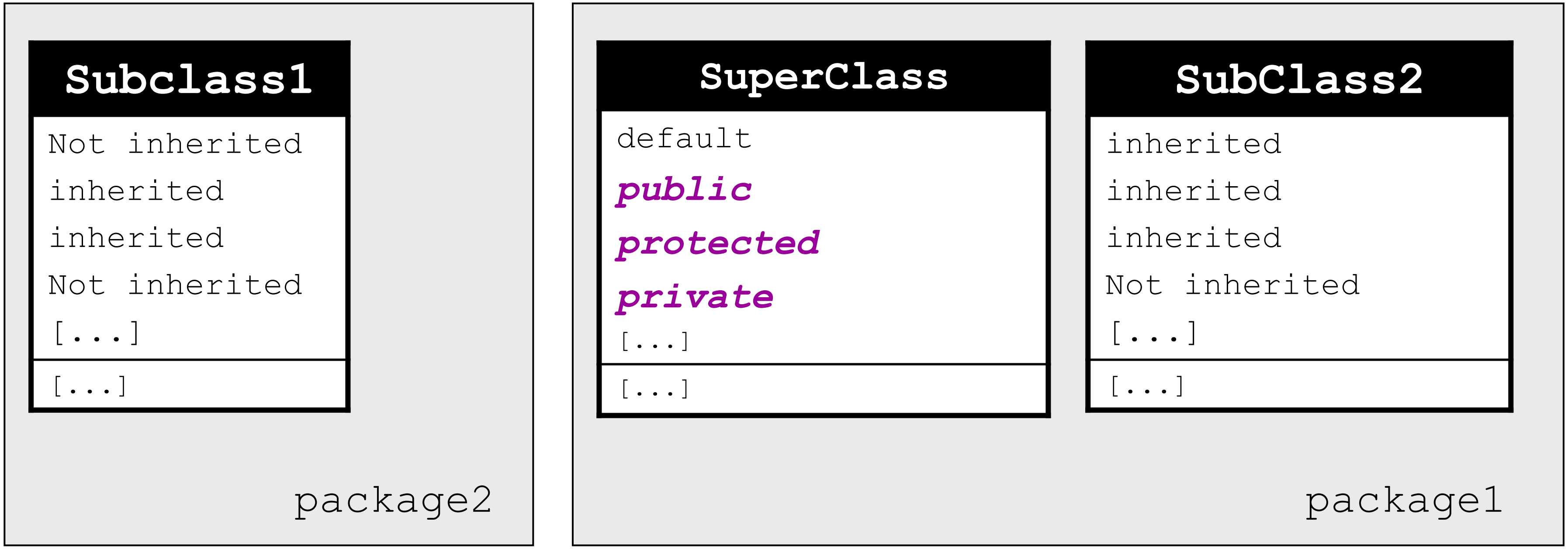
Shape reference is converted to a Rectangle reference – although it is actually a Circle instance!!

- compiler cannot know what c is at runtime
- cast COULD be possible, since we COULD HAVE stored a Rectangle in the Shape reference

Access levels revisited

- any member (attributes, methods, constructors,…) can be assigned one of the following access levels
  - **`public`**:
    any code can access
  - default (no access modifier):
    any code in the same package can access
  - **`protected`**:
    any subclass can access, even in different packages
  - **`private`**:
    only the class itself can access

Packages



Subclass1

Not inherited
inherited
inherited
Not inherited
[...]

[...]

package2

SuperClass

default
public
protected
private
[...]

[...]

SubClass2

inherited
inherited
inherited
Not inherited
[...]

[...]

package1

only ***public*** and ***protected*** members are inherited

- ***private*** members in the baseclass are not inherited within a package
- members without access modifier are inherited

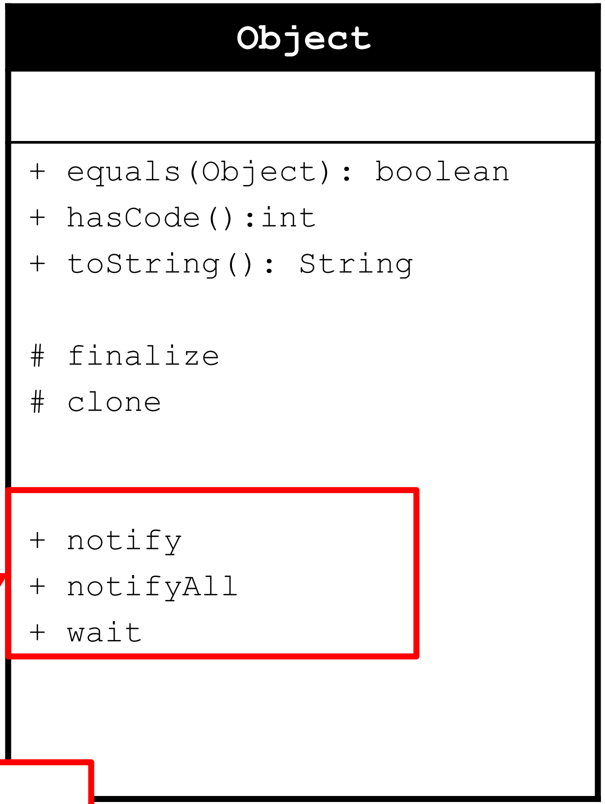
Programming 2

Class Object

class Object

- Every Java class is implicitly derived from the base class Object

- Object has a number of methods that all our classes “get for free”



Object

+ equals(Object): boolean
+ hasCode():int
+ toString(): String

# finalize
# clone

+ notify
+ notifyAll
+ wait

not covered here, important for concurrency (threads)

Object methods

- `Object.toString():String`
  - returns a String representation of the object
  - default is: `<type>@<hashcode>`

    e.g.: `Circle@c17164`
  - this is the reason why everything can be an argument to putln(): putln calls toString on the argument and displays the returned String

Object methods

- `Object.toString():String`
  - Always override toString()
  - When practical, it should return *all* the interesting information contained in the object
  - Provide access to all the information contained in the value returned by toString() – otherwise client code is forced to parse that String
  - call the superclass toString() with `super.toString()`, if necessary

Object methods

- `Object.equals(Object):boolean`
  - indicates whether some other object is "equal" to this one
  - defines a null-consistent equivalence relation (symmetric, reflexive, transitive)
  - by default, every instance is equals only to itself
  - override only if equality other than object equality is needed
  - obey the contract, if you override equals – other code (Collections) depend on it

Object methods

- `hashCode():int`
  - returns a hash code value for the object
  - equal objects have same hash code
  - unequal objects need not have different hash code
  - should be overridden when equals is overridden

Object methods

- `Object.finalize():void`
  - called when the garbage collector eventually destroys the object
  - overriding should be avoided for performance (and other) reasons
- `Object.clone():Object`
  - creates and returns a copy of the object
  - many technical complications when overridden and/or used