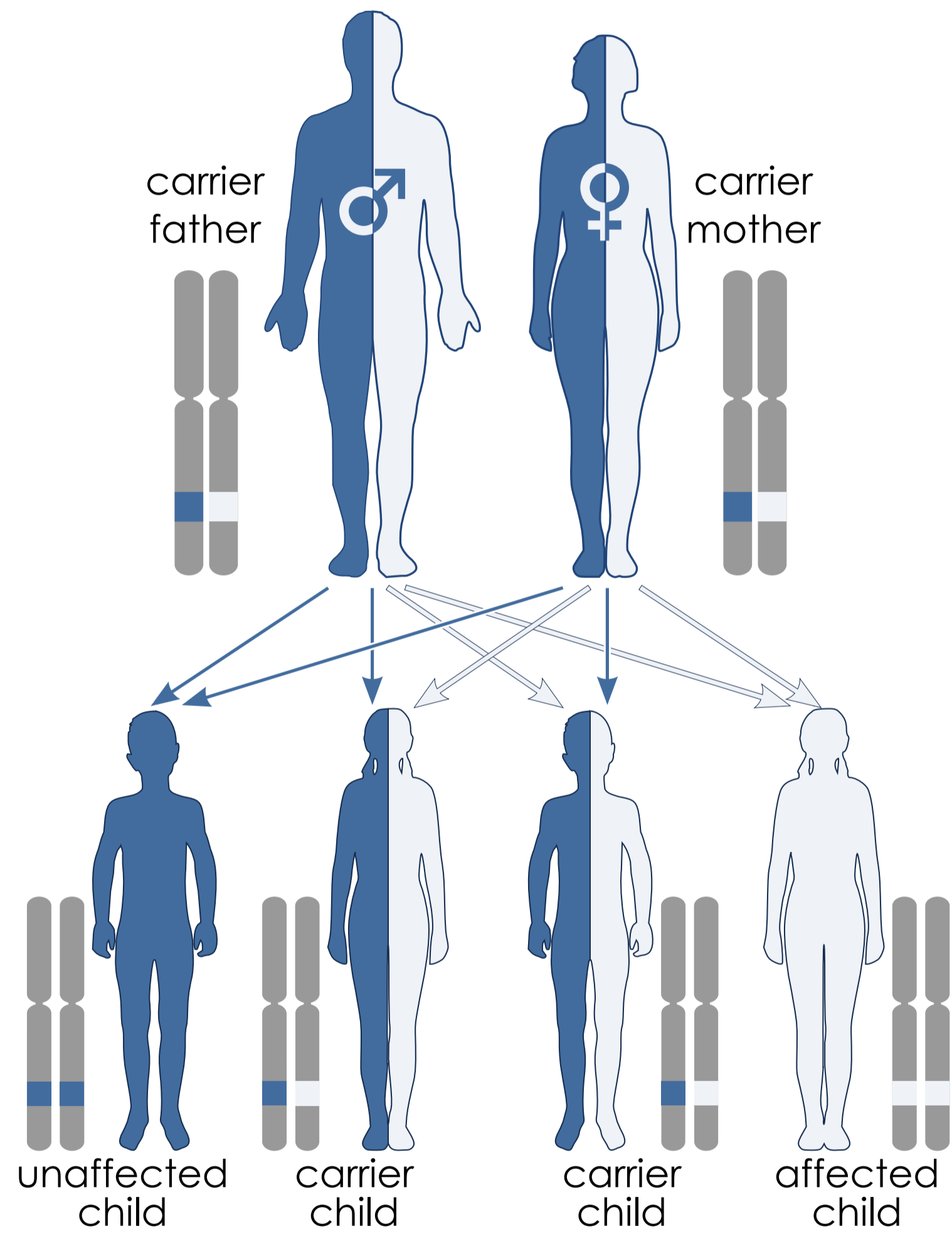


Overview




Inheritance

Guess who's inheriting the money





Autosomal recessive inheritance

-  *Unaffected*
-  *Affected*
-  *Carrier*

Duplicate code

```
public class Rectangle{  
    // Center coordinate  
    private double x; ①  
    private double y;  
  
    public void move  
        (double dx, double dy){ ②  
        x += dx;  
        y += dy;  
    }  
    private double width, height; ③ ...  
}
```

```
public class Circle {  
    // Center coordinate  
    private double x; ①  
    private double y;  
  
    public void move  
        (double dx, double dy){ ②  
        x += dx;  
        y += dy;  
    }  
    private double radius; ③ ...  
}
```

Idea: Centralize common code

- Create a parent class `Shape` containing common code portions.
- Relate both `Rectangle` and `Circle` to `Shape`.

Common and specific properties

Common Rectangle and Circle attributes

```
double x;  
double y;
```

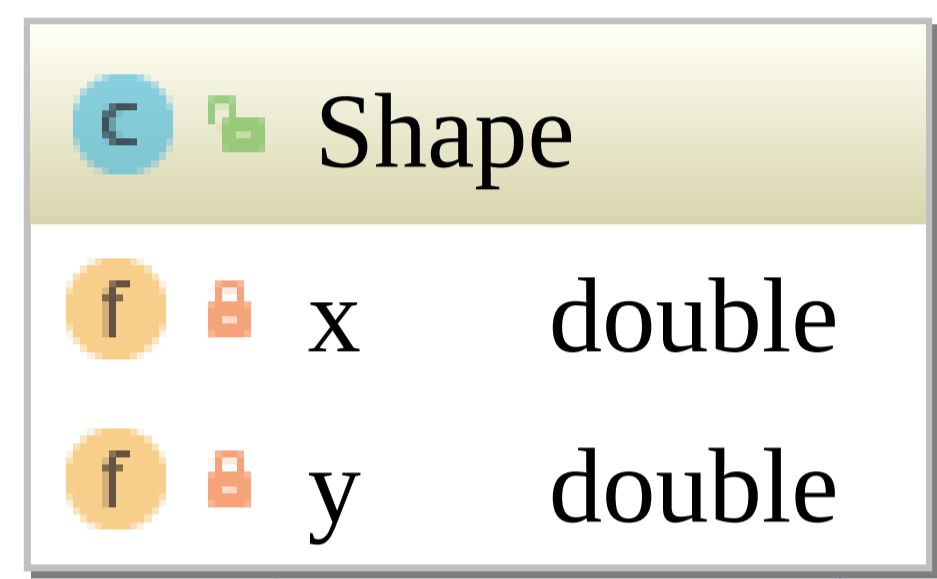
Rectangle attributes Circle attributes

```
double width;  
double height;
```

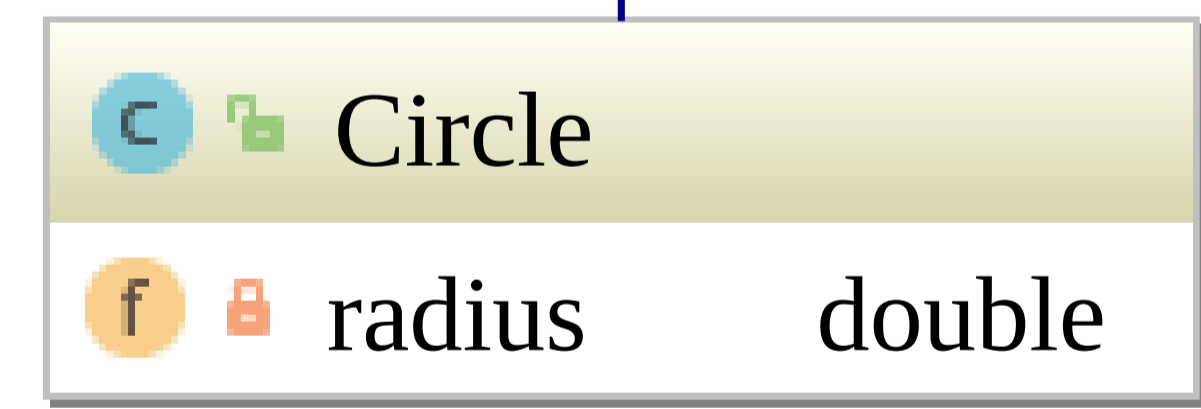
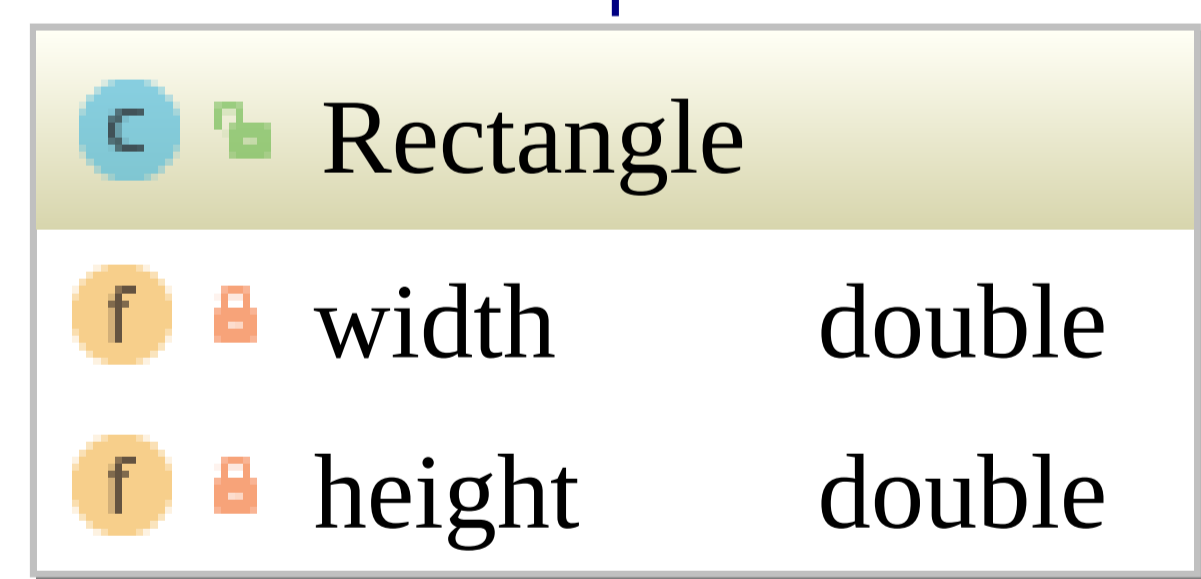
```
double radius;
```

Basic shape inheritance

Superclass



Derived classes



Inheritance

- Derived classes inherit state and behaviour.
- Refinement, specialization.
- “is-A” relationship:
 - A rectangle **is a** shape.
 - A circle **is a** shape.

Implementing Shape hierarchy

```
public class Shape {  
    private double x, y;  
}
```

```
public class Rectangle extends Shape  
    private double width;  
    private double height;  
}
```

```
public class Circle extends Shape {  
    private double radius;  
}
```

Creating instances

```
final double x = 2, y = 3;  
final Shape shape = new Shape(x, y);  
  
final double width = 2, height = 5;  
final Rectangle r = new Rectangle(x, y, width, height);  
  
final double radius = 2.5;  
final Circle circle = new Circle(x, y, radius);
```

Shape constructor

```
/**  
 * Creating a shape located at center coordinate.  
 * @param x The center's x component.  
 * @param y The center's y component.  
 */  
public Shape(double x,double y) {  
    this.x = x;  
    this.y = y;  
}
```

Creating Rectangle instances

```
final Rectangle r =  
    new Rectangle(x, y ①,  
                 width, height ②);
```

- ① Center coordinate components “belonging” to superclass Shape.
 - ② width and height “belonging” to class Rectangle.
-

Solution: Nested constructor call. Coming soon ...

Rectangle constructor

```
/**
 * Creating a rectangle at (x|y) of given width and height.
 * @param x Center's x component.
 * @param y Center's y component.
 * @param width Rectangle's width.
 * @param height Rectangle's height.
 */
public Rectangle(double x, double y,
                 double width, double height) {
    super(x, y) ❶;
    this.width = width; this.height = height ❷;
}
```

Inheritance

→ Overriding equals () and hashCode ()

Shape.equals()

```
public abstract class Shape {  
    ...  
    @Override ❶ public boolean equals(final Object o) {  
        if (o instanceof Shape s ❷) {  
            return x == s.x && y == s.y; ❸  
        } else {  
            return false; ❹  
        } ...  
    }  
}
```

Rectangle.equals()

```
public class Rectangle extends Shape {  
    ...  
    @Override public boolean equals(final Object o) {  
        if (o instanceof Rectangle r) {  
            return super.equals(o) 1 &&  
                width == r.width && height == r.height 2;  
        } else {  
            return false;  
        } ...  
    }  
}
```


Followup exercises

168. Let me pass, please!

169. Why is `==` correctly comparing enum instances?

Inheritance

↳ Overriding toString()

Printing a Shape's info

Code

```
package inherit;  
  
public class Run {  
  
    public static void main(String[] args) {  
        final Shape shape =  
            new Shape(2.0, 3.0); // Center coordinates  
        System.out.println(shape);  
    }  
}
```

Output

```
inherit.Shape@37d31475
```

Desired:

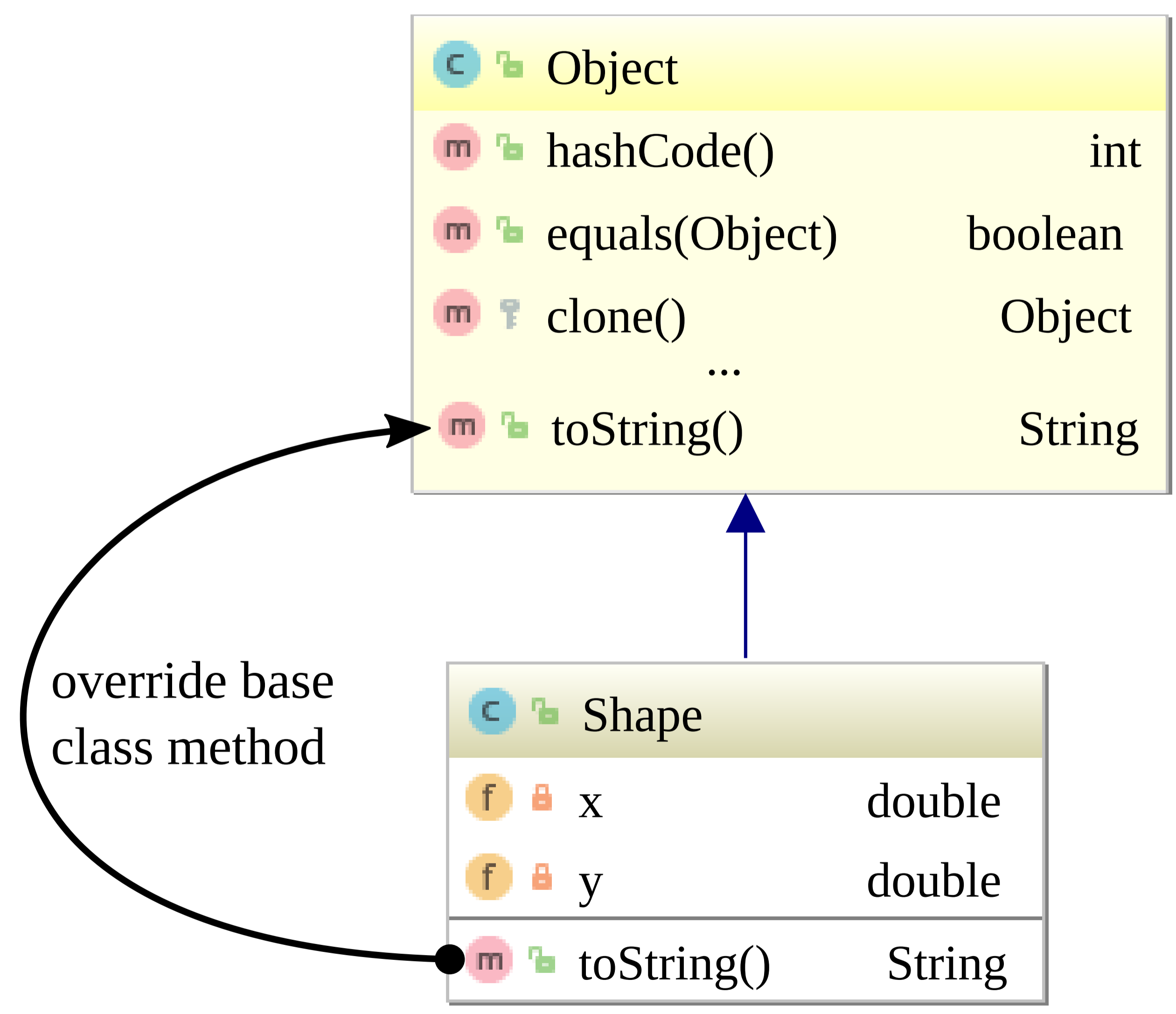
```
(2.0|3.0)
```

Overwriting toString()

```
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
ww > src > main > java > inherit > Sh 01/10/01 Grow [Run] [Debug] [Test] [Find]
Shape.java x
3 public class Shape {
4
5     @Override
6     public String toString() {
7         return super.toString();
8     }
9
10    /**
11     * Creating a shape located at center coordinate.
12     * @param x The center's x component.
13     * @param y The center's y component
```

Tool provided default implementation, most likely to be changed

Shape extending Object



Logging Rectangle instances

Code

```
// Center coordinates, width and height  
final Rectangle r = new Rectangle(2.0, 3.0, 3.0, 4.0);  
System.out.println(r);
```

Output

Desired:

```
(2.0|3.0)
```

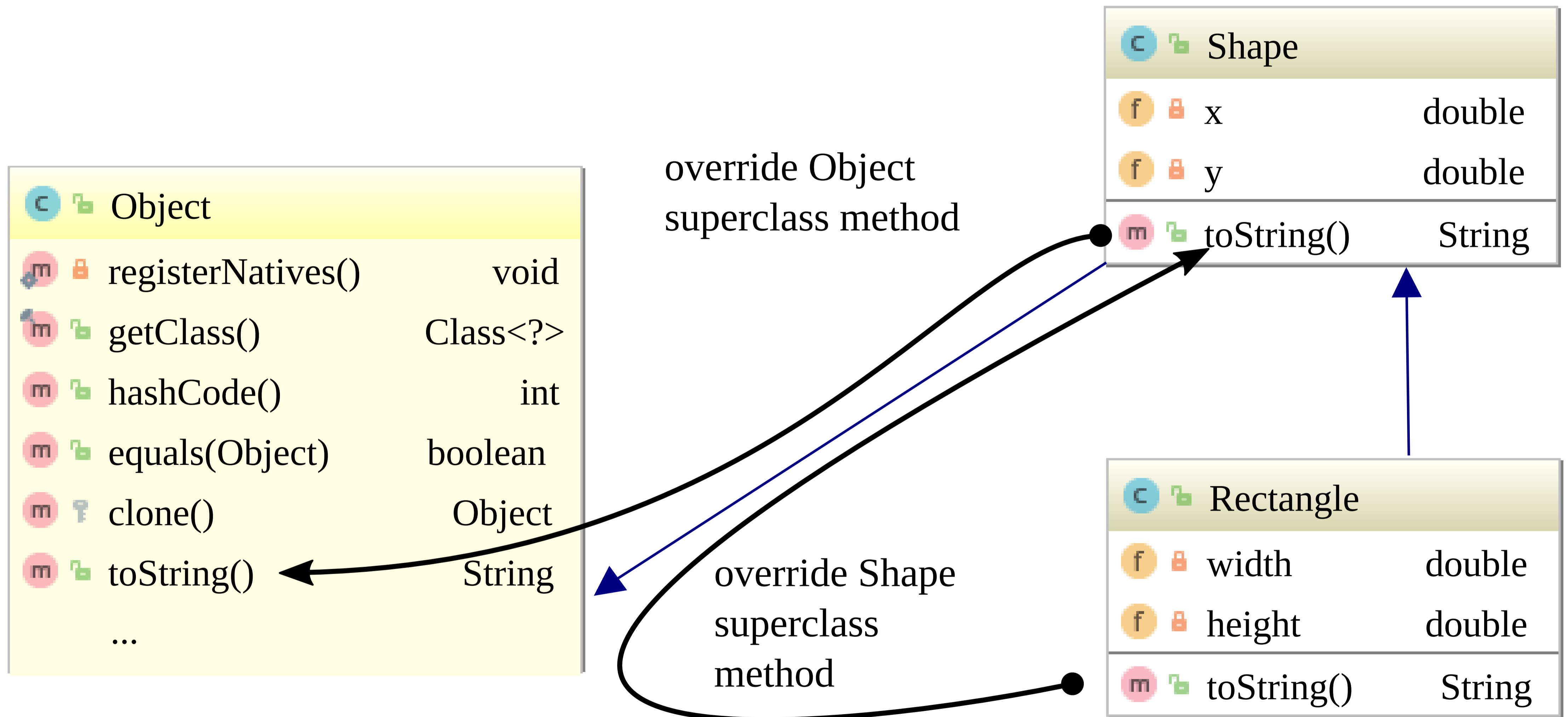
```
Rectangle at (2.0|3.0), width= 3.0, height=4.0
```

Override toString() in class Rectangle.

```
public class Rectangle extends Shape {  
    @Override public String toString() {  
        return "Rectangle at " + super.toString() ① +  
            ", width= " + width + ", height=" + height; ②  
    } ...  
}
```

- ① The `super` keyword allows for calling the superclass `Shape`'s `toString()` method inserting the (2.0|3.0) center coordinate.
- ② Append class `Rectangle`'s “own” `width` and `height` attribute values.

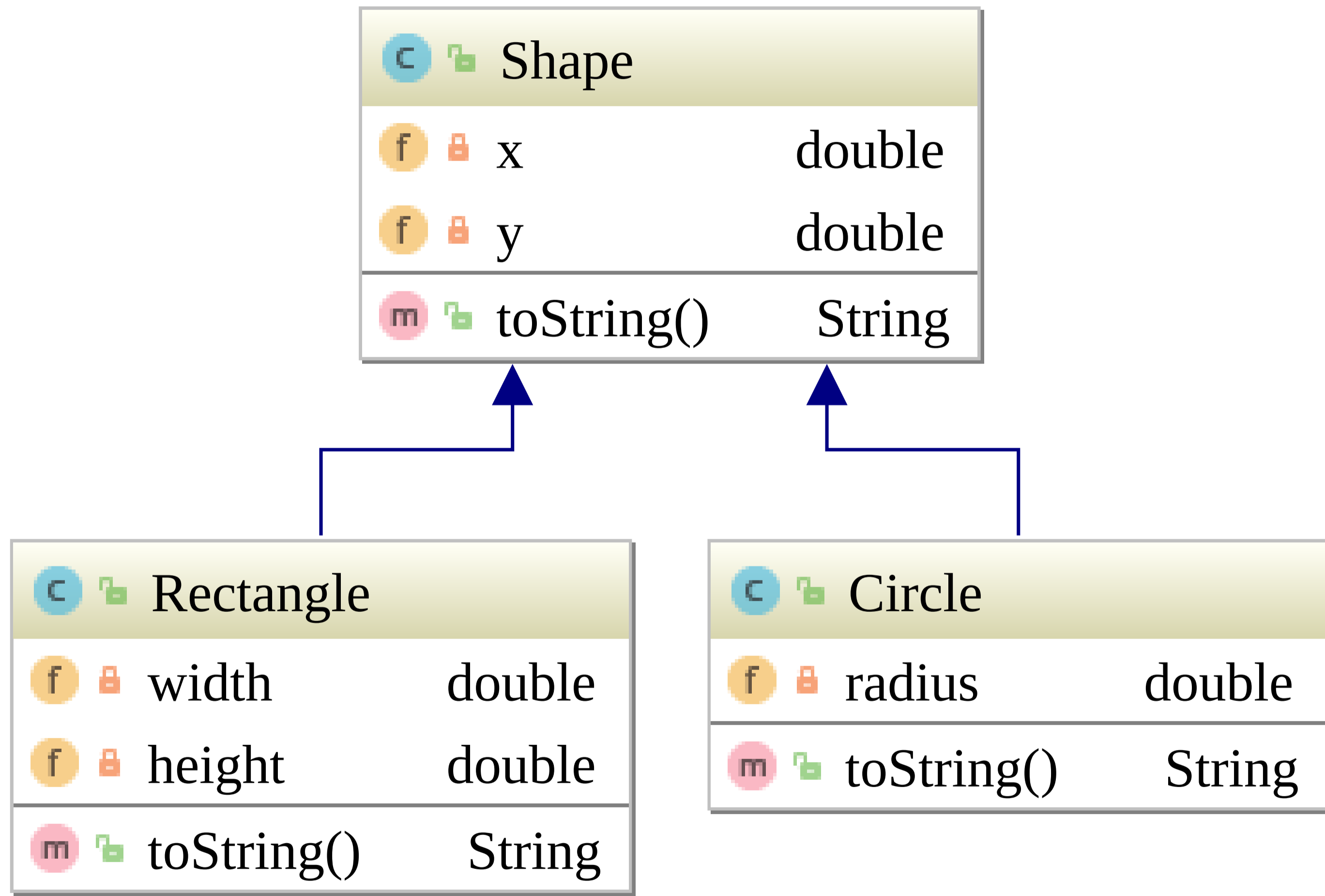
Rectangle extending Shape



Implementing Circle.toString()

```
public class Circle extends Shape {
    /**
     * Creating a circle of given center and radius
     * @param x Center's x component.
     * @param y Center's y component.
     * @param radius The circle's radius.
     */
    public Circle(double x, double y, double radius) {
        super(x, y);
        this.radius = radius;
    }
    @Override public String toString() {
        return "Circle at " + super.toString() + ", radius= " + radius;
    }
    private double radius;
}
```

Shape and toString()



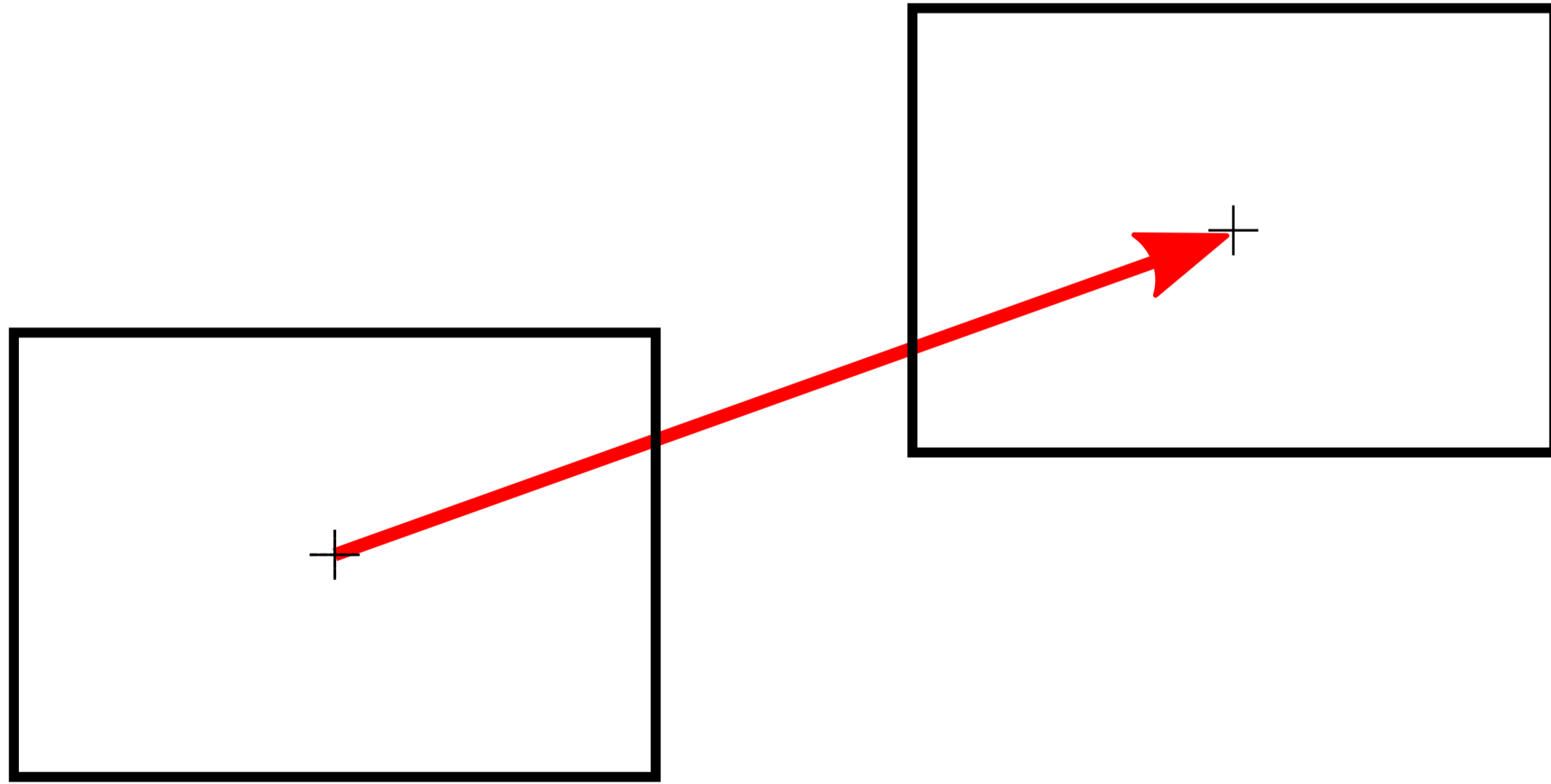
170. `String` vs. `StringBuffer`

171. Alternate implementation of opposite directions

Inheritance

↳ `final` methods

Moving Shape instances



Implementing Shape movements

```
public class Shape {  
    /**  
     * Move by a given translation vector  
     * @param xTrans Translation's x component  
     * @param yTrans Translation's y component  
     */  
    public void move(final int xTrans, final int yTrans) {  
        x += xTrans;  
        y += yTrans;  
    } ...  
}
```

Fools are everywhere!

```
public class Rectangle extends Shape {  
    @Override public void move(int xTrans, int yTrans) {  
        // I'm so dumb!  
        ...  
    }  
}
```

Solution: **final** prevents overriding

```
public abstract class Shape {  
... public final void move(final int xTrans, final int yTrans) {  
    x += xTrans;  
    y += yTrans;  
}...
```

```
public class Rectangle extends Shape {  
    // Syntax error: 'move(int, int)' cannot override  
    // 'move(int, int)' in 'inherit.Shape'; overridden method is final  
    @Override public void move(int xTrans, int yTrans) {...
```


Inheritance

↳ Abstract methods

Calculating a shape's area

```
public class Rectangle extends Shape
/**
 * Calculate the area.
 * @return The rectangle's area
 */
public double getArea() {
    return width * height;
} ...
```

```
public class Circle extends Shape
/**
 * Calculate the area.
 * @return The circle's area
 */
public double getArea() {
    return PI * radius * radius;
} ...
```

Desired: Polymorphic `getArea()` call

```
final Shape[] shapes ① = {  
    new Circle(1, 1, 2.) ②,  
    new Rectangle(1, -1, 2., 3.) ②};  
  
for (final Shape s : shapes) {  
    System.out.println(s.toString() + ": area = " + s.getArea()); ③  
}
```

```
Circle at (1.0|1.0), radius= 2.0: area = 12.566370614359172  
Rectangle at (1.0|-1.0), width= 2.0, height=3.0: area = 6.0
```

Problems:

- No meaningful `getArea ()` method in class `Shape` possible.
- Meaningful implementations exist both in subclass `Rectangle` and `Circle`.

Solution: Abstract method `getArea ()` in superclass `Shape`.

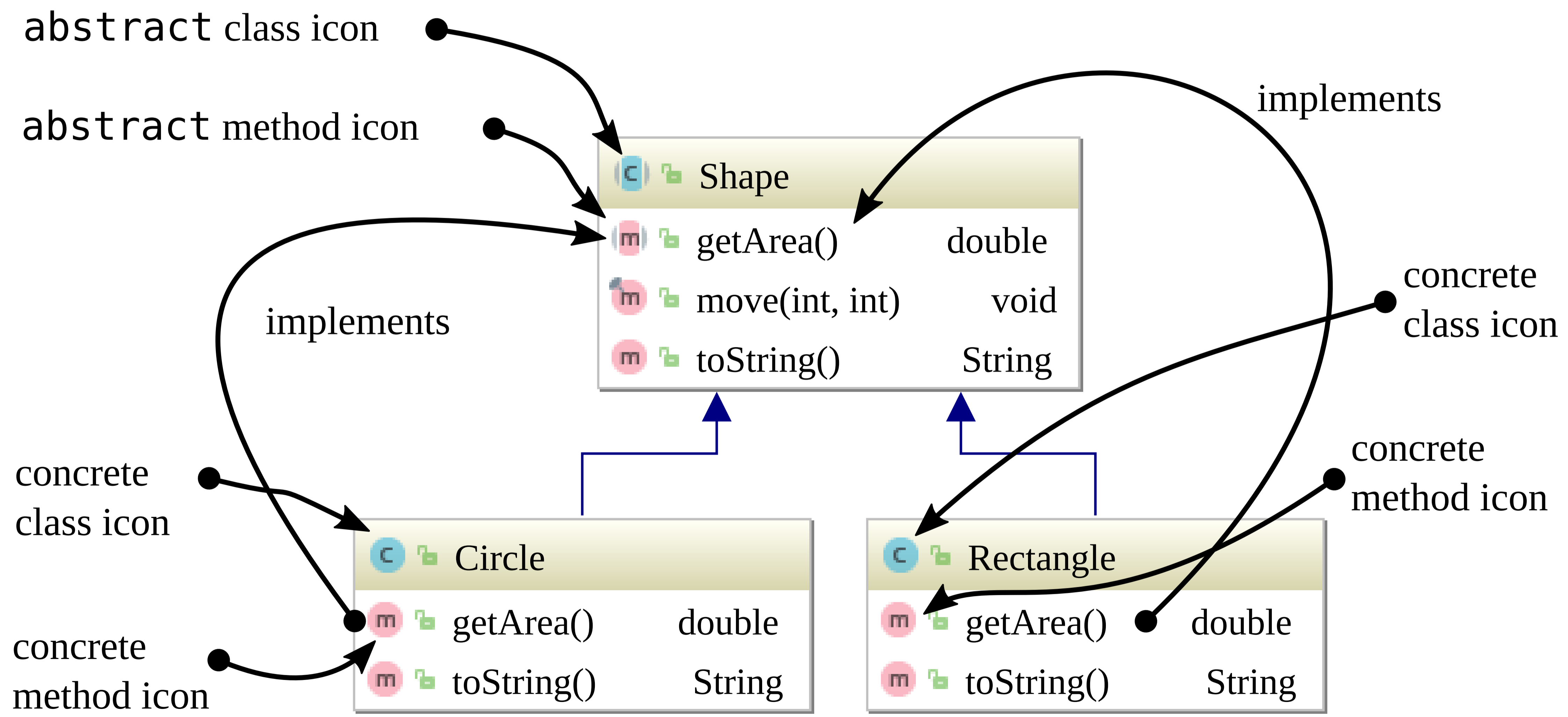
abstract method `getArea()`

```
abstract ❶ public class Shape {  
    /**  
     * Calculate the shape's area.  
     * @return The shape's area  
     */  
    abstract ❷ public double getArea() ❸; ...
```

```
public class Rectangle extends Shape {  
    @Override ❶  
    public double getArea() {  
        return width * height;  
    } ...
```

```
public class Circle ... {  
    @Override ❶  
    public double getArea() {  
        return Math.PI *  
            radius * radius;  
    } ...
```

abstract method getArea()



What's a "shape" anyway?



No instances of **abstract** classes.

```
final Shape s =  
    new Shape(1., 2.); // 'Shape' is abstract; cannot be instantiated
```


Mandatory `getArea()` implementation.

```
// Error: Class 'Circle' must either be declared abstract or  
// implement abstract method 'getArea()' in 'Shape'  
public class Circle extends Shape {  
  
    public Circle(double x, double y, double radius) {  
        super(x, y);  
        this.radius = radius;  
    }  
    private double radius;  
}
```

Facts about `abstract` fields, methods and classes.

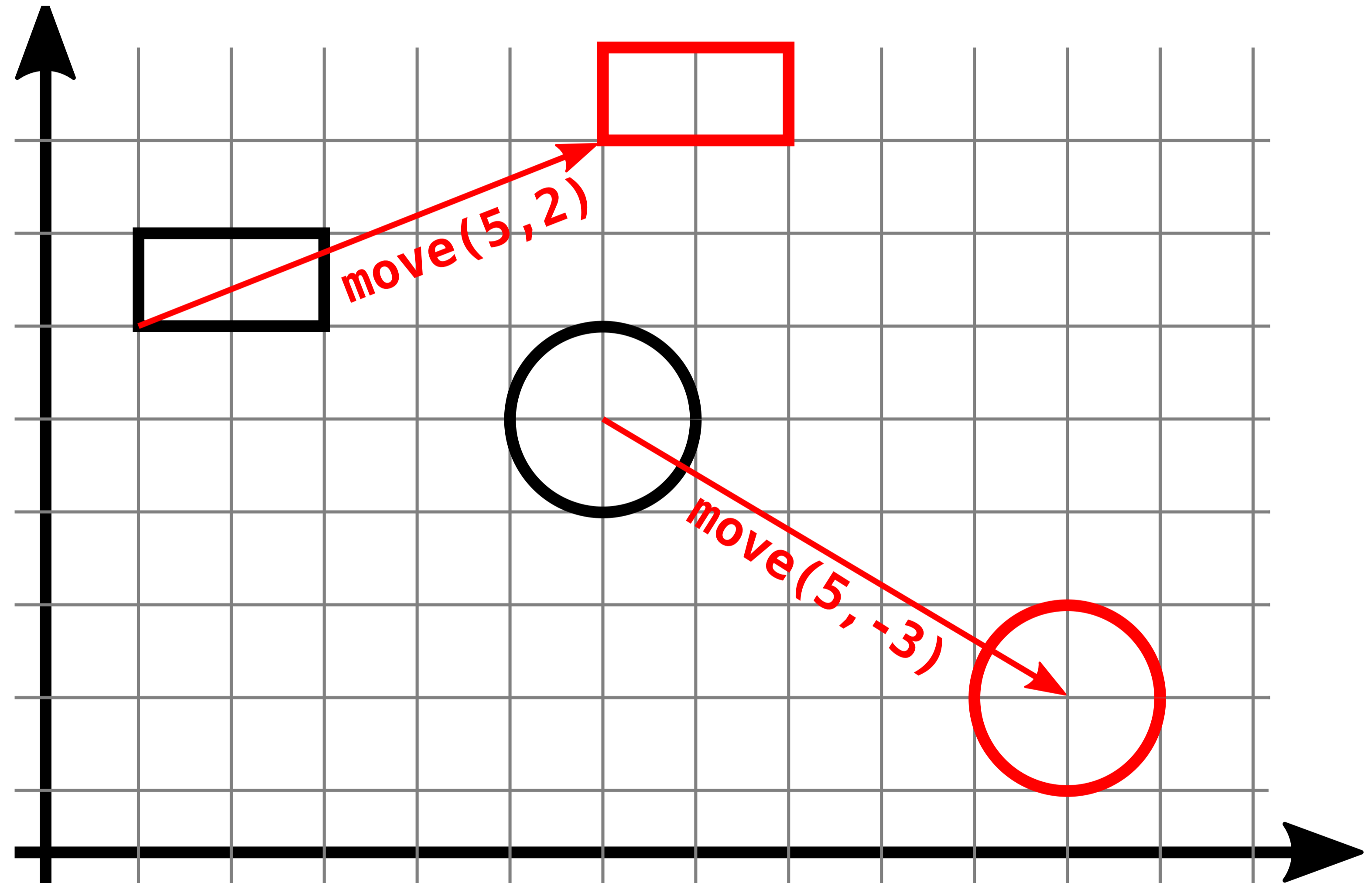
- A class containing an `abstract` method must itself be declared `abstract`.
- `abstract` classes are allowed to host non-`abstract` methods.
- A class may be declared `abstract` irrespective of purely containing non-`abstract` methods.

Inheritance

- ⇒ Abstract methods

- ⇒ Geometry classes reconsidered

Moving shapes



Followup exercises

172. Defining a Shape class hierarchy

173. Scaling shapes

174. Providing toString() methods

Inheritance

⇒ protected access

protected access

```
package model;
public abstract class Shape ❶ {

    final protected long creationTime ❷ = System.nanoTime();
    ...
}
-----
package model.sub;
public class Rectangle ❸ extends Shape {
    static final Logger log = LogManager.getLogger(Rectangle.class);

    @Override public double getArea() {
        log.info("Rectangle creation time:" + creationTime ❹);
        return width * height;
    } ...
}
```

175. `protected` vs. “package private”

176. `protected` access involving different instances

Inheritance

↳ `final` classes

final classes

```
public final class Shape { ... }
```

```
-----
```

```
public class Rectangle  
    extends Shape { // Error: final class cannot be extended
```

```
...
```

final classes rationale

- Design decision.
- Slight performance gain.

Note

Prominent Example: `java.lang.String`.

Inheritance

→ The `instanceof` operator

“Defeating” polymorphism

```
public static void main(String[] args) {
    final Shape[] shapes = {
        new Circle(1, 1, 2.),
        new Rectangle(1, -1, 2., 3.)};
    print(shapes);
}
static void print(final Shape[] shapes) {
    for (final Shape s : shapes) {
        if (s instanceof Rectangle) {
            System.out.println("Type Rectangle");
        } else if (s instanceof Circle) {
            System.out.println("Type Circle");
        }
    }
}
```

```
Type Circle
Type Rectangle
```

Defining equals (. . .): Expectations

```
Rectangle r1 = new Rectangle(1, 2, 5, 4),
           r2 = new Rectangle(1, 2, 1, 7),
           r3 = new Rectangle(1, 2, 5, 4);

Circle c = new Circle(-2, 3, 5);

System.out.print(r1.equals("Hi")); // false: Differing classes Rectangle and String.
System.out.print(r1.equals(r2));  // false: Differing width and height.
System.out.print(r3.equals(r1));  // true:  Two rectangles having identical
                                   //      (x|y), width and height.
System.out.print(r1.equals(c));    // false: Differing classes Rectangle and Circle.
System.out.print(c.equals(c));    // true: Object equal to itself.
```

Defining equals (. . .) of Shape instances

Two Shape instances shall be considered equal if:

- Both instances are of common type i.e. either Rectangle or Circle.
- Their center coordinates match within a threshold of 10^{-15} .
- width and height or radius match within a threshold of 10^{-15} .

Comparing center coordinates

```
public abstract class Shape {  
    private double x, y;  
  
    protected boolean equalCenter(final Shape o) {  
        return Math.abs(o.x - x) + Math.abs(o.y - y) < 1.E-15;  
    }  
    ...  
}
```


Implementing `Rectangle.equals()`

```
public class Rectangle extends Shape {
    @Override public boolean equals(Object o) {
        if (o instanceof Rectangle r) {

            final Rectangle oRectangle = (Rectangle) o; // Cast is «legal»
            return super.equalCenter(r) &&
                Math.abs(r.width- width) +
                    Math.abs(r.height- height) < 1.E-15;
        }
        return false;
    }
    ...
}
```

For `o == null` the expression `o instanceof Rectangle` evaluates to false.

Implementing `Circle.equals()`

```
public class Circle extends Shape {  
    @Override public boolean equals(final Object o) {  
        if (o instanceof Circle c){  
            return super.equalCenter(c) &&  
                Math.abs(c.radius - radius) < 1.E-15;  
        }  
        return false;  
    } ...  
}
```

Testing equality of Shape objects

```
final Rectangle
```

```
    r1 = new Rectangle(2, 3, 1,4),
```

```
    r2 = new Rectangle(2, 3, 2,8),
```

```
    r3 = new Rectangle(2, 3, 1,4);
```

```
final Circle c = new Circle(2,3, 7);
```

```
System.out.println("r1.equals(r2): " + r1.equals(r2)
```

```
System.out.println("r1.equals(r3): " + r1.equals(r3)
```

```
System.out.println("c.equals(r1): " + c.equals(r1))
```

```
r1.equals(r2): false  
r1.equals(r3): true  
c.equals(r1): false
```

Inheritance

↳ The `@Override` annotation.

Overriding Object.toString()

```
public class Shape {  
    double x, y;  
    ...  
    @Override // Promise: Subsequent method overrides Object.toString();  
    public String toString() {  
        return "(" + x + "|" + y + ")";  
    }  
}
```

@Override: Easy compile time error detection

```
public class Shape {  
    double x, y;  
    ...  
    @Override // Error: method does not override a method from a supertype  
    public String toString(int value) {  
        return "(" + x + "|" + y + ")";  
    }  
}
```

Explanation: The given method does not override `Object.toString()`.

Inheritance

⇒ Runtime polymorphism

Meta info of an instance

Metainfo

User
attributes

I'm a Rectangle
x y width height

I'm a Circle
x y radius

Getting instance meta info

Code

```
final Object[] instances = {
    new String("Test"),
    new Date()
};
printMetaInfo(instances);
-----
static void printMetaInfo(Object[] objects) {
    for (Object o: objects) {
        System.out.println(o.getClass().getName());
    }
}
```

Result

```
java.lang.String
java.util.Date
```

Implementing `getArea()`

```
public abstract class Shape {  
    ...  
    abstract public double getArea();  
}
```

```
public class Rectangle extends Shape {  
    ...  
    private int width, height;  
    @Override public double getArea() {  
        return width * height; }  
}
```

```
public class Circle extends Shape {  
    ...  
    private int radius;  
    @Override public double getArea() {  
        return Math.PI * radius * radius  
    }  
}
```

Polymorphic `getArea()` call

Code

```
final Shape[] shapes = {
    new Rectangle(0, 0, 2, 3),
    new Circle(0, 0, 1)};
}
printAreas(shapes);
-----
static void printAreas(Shape[] shapes) {
    for (Shape shape : shapes) {
        System.out.println(shape.getArea());
    }
}
```

Result

```
Area:6.0
Area:3.141592653589793
```

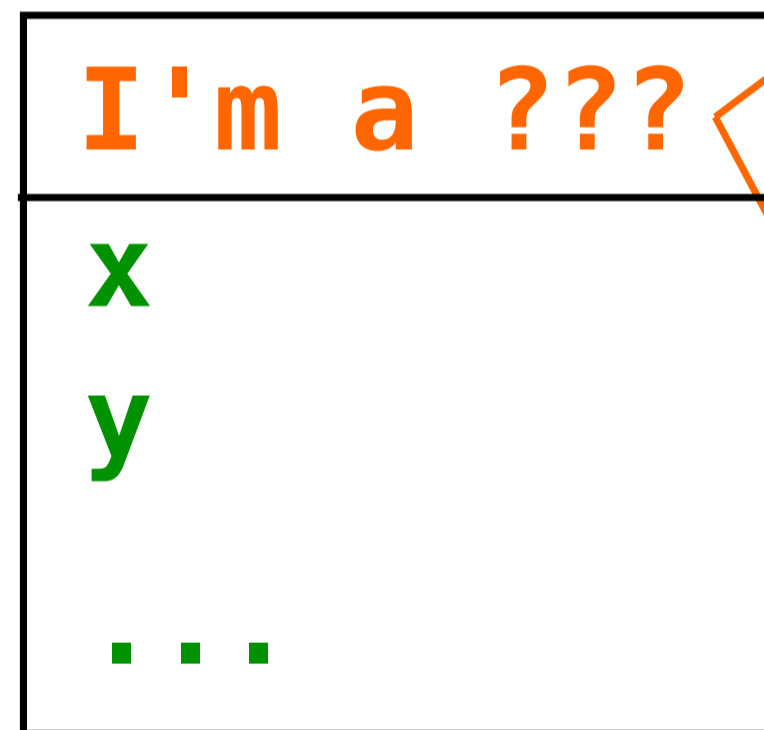
Runtime polymorphic dispatch

Shape is a
Rectangle:

Execute **Rectangle::getArea()**

Shape is a
Circle:

Execute **Circle::getArea()**



Conventional `getArea()` implementation

```
public abstract class Shape {  
    ...  
    public final double getArea(){  
        if (this instanceof Rectangle rectangle) {  
            return rectangle.width * rectangle.height;  
        } else if (this instanceof Circle circle) {  
            return circle.radius * circle.radius * Math.PI;  
        } else {  
            return 0; // Unreachable  
        }  
    }  
}
```

```
public class Rectangle extends Shape {  
    ...  
    protected int width, height;  
}
```

```
public class Circle extends Shape {  
    ...  
    protected int radius;  
}
```