POLYMORPHISM

The word *polymorphism* means *many forms* – it is derived from the Greek *polumorphos* = *poly* (many) + *morphous* (forms). In Biology it refers to an organism that changes form throughout its life. For example, a butterfly starts life as an egg, then a caterpillar, then a pupa, before emerging in its adult form.

In the world of human engineered objects, polymorphism refers to an action or mechanism that can have different behaviors or implementations depending on the object.

*Example*

The brake pedal on a vehicle is a polymorphous mechanism. As a driver, you engage the brake by stomping the pedal with your foot – an action that is almost universal in vehicles. The actual braking mechanism, however, can vary greatly in different vehicles – drum brakes, disc brakes, anti-lock brakes, air brakes, etc.

*Example*

The clickable end of a pen or pencil is a polymorphous mechanism. On a pen, it acts like a toggle – click once to extend the pen point, click again to retract it. On a pencil, click once to extend the lead, click again and it extends further, again and it extends even further, and so on.
In software, **polymorphism** refers to a method that has different implementations in different subclasses of a class hierarchy. This allows you to create a subclass and morph a method inherited from the superclass so that it’s behavior is more appropriate to the subclass.

---

**Example**

Class **Figure** class is meant to be a superclass for geometric figures. Every figure has an area, so the class has a method **getArea** to calculate and return the area. This general figure has no dimensions, so the area cannot be calculated; thus, **getArea** simply returns zero.

```java
public class Figure {
    public double getArea() {
        // Return the figure's area.
        return 0.0;
    }
}
```

A rectangle is _a_ figure with length and width whose area = length × width. Class **Rectangle** models this by extending **Figure**, adding fields **length** and **width** to hold its dimensions. A constructor initializes these dimensions for each newly built **Rectangle** object. Method **getArea**, which is automatically inherited from **Figure**, is morphed to calculate the area of a rectangle.

```java
public class Rectangle extends Figure {
    public double length, width;
    public Rectangle(double length, double width) {
        // Build a rectangle with given dimensions.
        this.length = length;
        this.width = width;
    }
    public double getArea() {
        // Morph to calculate area of a rectangle.
        return length * width;
    }
}
```
A circle IS_A figure with a radius $r$ whose area = $\pi r^2$. Class `Circle` models this by extending `Figure`, adding field `radius` and morphing method `getArea` so that it calculates the area of a circle.

```java
public class Circle extends Figure {
    public double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

Here’s the class diagram for the figure hierarchy:

```
Figure

getArea(): double

Rectangle

length: double
width: double
Rectangle(length: double, width: double)
getArea(): double

Circle

radius: double
Circle(radius: double)
getArea(): double
```
A reference variable with the superclass as its declared data type can refer to an object of any class in the hierarchy; but not vice-versa.

| Examples |
|-----------------|------------------|
| Figure a = new Figure( ); | OK |
| Figure b = new Rectangle( 3, 4 ); | OK |
| Figure c = new Circle( 10 ); | OK |
| Rectangle d = new Figure( ); | Error |

The class written out in the declaration of the reference variable is its static data type; the class of the object to which the reference variable refers is its dynamic data type.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference variable b given below has static data type of Figure and dynamic data type of Rectangle.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.getArea( )</td>
</tr>
<tr>
<td>b.toString( )</td>
</tr>
<tr>
<td>b.getPerimeter( )</td>
</tr>
</tbody>
</table>
The dynamic data type is used by the JVM to insure that the correct method is called during the program run.

**Example**

Line 1 When executed, the JVM initializes `r` to refer to a newly built `Rectangle` object.

Line 2 When executed, `r.getArea()` calls the method in class `Rectangle`, which is the dynamic data type of `r`.

Line 3 When executed, the JVM initializes `c` to refer to a newly built `Circle` object.

Line 4 When executed, `c.getArea()` calls the method `Circle.getArea`.

```
1 Figure r = new Rectangle( 3, 4 );
2 System.out.println( r.getArea( ) );
3 Figure c = new Circle( 10 );
4 System.out.println( c.getArea( ) );
```

Engineering polymorphism refers to a mechanism that has different implementations depending on the object. The brake pedal in an automobile is used to stop it regardless of whether the underlying system uses disc brakes, drum brakes, air brakes, etc.

In software engineering, you employ polymorphism by giving a reference variable the superclass as its static data type. This allows:

1. The reference variable to point to an object of any class in the hierarchy.
2. The compiler, during compile time, to pass as valid any call that the object makes to a method declared in the superclass.
3. The JVM, during runtime, to call the correct implementation of a morphed method based on the dynamic data type.
**Example**

These concepts are illustrated in application `FigureAreas` shown below.

**During compile time**

Line 7 The compiler gives reference variable `f` the static data type of `Figure`.

Lines 10–18 The compiler allows `f` to point to either a newly built `Rectangle` or `Circle` object. At compile time, it does not know which figure the user will select during the run.

Line 19 The compiler allows the call `f.getArea()` because `getArea` is a method in any `Figure` object.

**During runtime**

Lines 10–18 Depending on the user’s choice, the JVM builds either a `Rectangle` or `Circle` object and places its reference into `f`.

Line 19 The JVM calls the correct implementation of `getArea` based on the dynamic data type of `f`.

```java
import static javax.swing.JOptionPane.*;

public class FigureAreas {
    public static void main( String[] args ) {
        Figure f = null;
        String prompt = "Rectangle or Circle?";
        String whatFigure = showInputDialog( prompt );
        switch ( whatFigure ) {
            case "Rectangle":
                f = new Rectangle( 3, 4 );
                break;
            case "Circle":
                f = new Circle( 10 );
                break;
            default:
                showMessageDialog( null, "Area = " + f.getArea( ) );
        }
    }
}
```
A subclass can be the superclass of another.

Example

A square IS_A rectangle whose 4 sides have the same length. The area of a square is computed the same way as that of a rectangle.

Class **Square** below models a square by extending the class **Rectangle**. Everything it needs is inherited. To build a square, its constructor takes one argument – the length of the square’s side – and calls the constructor of its superclass **Rectangle** (at line 8) to initialize the length and width to the same value.

```java
public class Square extends Rectangle {
    // 'length', 'width' are inherited

    public Square( double side )
        // Build a square with given side.
        {
            super( side, side ); // call Rectangle's constructor
        }
}
```

Here’s the complete class diagram for the figure hierarchy:

![Class Diagram](image-url)
Exercises

For each of the statements shown below, circle what’s wrong and explain. Some of the statements may be correct.

1. Figure a = new Figure( );
2. Figure b = new Rectangle( 12, 24 );
3. Figure c = new Circle( 10 );
4. Figure d = new Square( 5 );
5. Rectangle e = new Figure( 12, 24 );
6. Rectangle f = new Figure( );
7. Rectangle g = new Square( 5 );
8. Square h = new Rectangle( 5 );
9. Square i = new Rectangle( 5, 5 );

Enter the classes Figure, Circle, Rectangle and Square into jGRASP, saving each to its own Java file. Compile them. Complete each of the exercises below using jGRASP’s “Interactions” window.

10. Declare a reference variable whose static data type is Figure. Initialize it to refer to a 20×30 Rectangle object. Print its area.
11. Declare a reference variable whose static data type is Figure. Initialize it to refer to a Circle object of radius 25. Print its area.
12. Declare a reference variable whose static data type is Figure. Initialize it to refer to a Square object with a side length of 16. Print its area.
Enter the classes **Figure, Circle, Rectangle** and **Square** into jGRASP, saving each to its own Java file. Enter class **FigureAreas** into jGRASP and save it. Make sure all five Java files are in the same folder on your disk.

13. Modify **FigureAreas** so that it allows the user to choose “Square” and, if he or she does so, creates a **Square** object with side length of 5. For example:

   ![Image of GUI showing selection of Square and area calculation]

14. Modify your solution the previous problem so that it allows the user to enter, within a single input dialog, a string indicating the type of figure followed by the figure’s dimensions. The data must be delimited by spaces. Create the appropriate object initialized with the input dimensions. For example:

   ![Image of GUI showing input of figure and dimensions and area calculation]

15. Add a **Triangle** class that extends the **Figure** class. A triangle has three sides so code the constructor to take three arguments with which to initialize the **Triangle** object’s three instance variables. You can compute the area of a triangle using Hero’s formula. If you don’t know the formula, Google it.

   Modify your solution to the previous problem so that it adds “Triangle” to the figures that the user can build and calculate the area.