Fundamentals of Java

Polymorphism, abstract classes, interfaces

Polymorphism

Poly = many

Morphism = forms

In Java, an object can have many types.

```
PositionedRectangle pr =
        new PositionedRectangle(4, 4, 10, 7);
```

`pr` is a `PositionedRectangle`. AND,

`pr` is also a `Rectangle`.

Polymorphism

That means, it is possible to store `pr` into a variable of type `Rectangle`:

```
Rectangle r = pr;
```

The dynamic type of `r` is `PositionedRectangle`, but the static type is `Rectangle`. Only statically known methods can be invoked on `r`:

```
int area = r.getArea(); // OK
r.move(1, 1); // won't compile
```

Polymorphism

The positioned rectangle `pr` can also be passed as an argument to a parameter of type `Rectangle`:

```
someMethod(pr);
void someMethod(Rectangle r) {
    System.out.println(r.getArea());
}
```

This method does not statically know that `r` might sometimes be a `PositionedRectangle`, and it doesn't need to know.

This method will work on *any* type of rectangle.

That is good! The method is reusable.

Polymorphism

```
List<Rectangle> rects = new ArrayList<Rectangle>();
rects.add(new Rectangle(4, 4));
rects.add(new Rectangle(10, 5));
rects.add(new PositionedRectangle(1, 1, 5, 4));

for (Rectangle r : rects)
    r.show();
```

Although `rects` contains an assortment of different types of rectangles, polymorphism allows them all to be considered of type `Rectangle`.

Thus, we can loop over all rectangles and show them.

We do not need an if/else to handle the different types.

Classes vs Types

`Rectangle` is a class. *AND*...

`Rectangle` is a type.

Technically,

- The class defines how a rectangle works. i.e. the code.
- The type describes the interface only. i.e. what fields and methods a rectangle has, but ***not*** how the methods work.

Classes vs Types

Thus,

- when we speak of a class, we are referring to an object's behaviour/code/how it works.
- when we speak of a type, we are referring to an object's interface. What methods does this object have? How can we use this object?

```
  (type)                  (class)
Rectangle r       = new Rectangle(...);
```

what methods does r have?

how does r work inside?

Classes vs Types

Thus,

- when we speak of a class, we are referring to an object's behaviour/code/how it works.
- when we speak of a type, we are referring to an object's interface. What methods does this object have? How can we use this object?

```
  (type)                    (class)
Rectangle r   = new PositionedRectangle
(...);
```

what methods does r have?

how does r work inside?

Subclassing vs Subtyping

PositionedRectangle is a subclass of Rectangle. *AND...*

PositionedRectangle is a subtype of Rectangle.

- As a subclass, class `PositionedRectangle` inherits "code" from class `Rectangle`.
- As a subtype, type `PositionedRectangle` describes objects that can also be described by type `Rectangle`. i.e. Every object of type `PositionedRectangle` is also of type `Rectangle`.

In Java, subclassing implies subtyping.

This is not true in all languages!

Abstract classes

```
public abstract class Shape {}
public class Circle extends Shape {}
public class Triangle extends Shape {}
```

- Class `Shape` does not describe a concrete thing and should not be instantiated directly.
- Class `Shape` exists only to provide shared code to be inherited by concrete classes of shapes like `Circle` and `Triangle`.
- Use keyword `abstract` to declare that a class is not concrete.

Abstract methods

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

- An abstract method describes type information but not class information (i.e. there's no code).
- All Shapes have a getArea() method, but we don't know how it works.
- Each concrete subclass *must* override this method.

Abstract methods

```
public class Circle extends Shape {
    private double radius;
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
public class Triangle extends Shape {
    private double base, height;
    public double getArea() {
        return base / 2.0 * height;
    }
}
```

Any subclass that fails to provide code for getArea() would also have to be declared abstract (i.e. if a class has missing code, it cannot be instantiated)

Abstract methods

```
List<Shape> shapes = new ArrayList<Shape>();
shapes.add(new Circle(...
shapes.add(new Triangle(...
shapes.add(new Circle(...
for (Shape shape : shapes) {
    System.out.println(shape.getArea());
}
```

Loop over the assortment of shapes, and print out each shape's area.

Using polymorphism, all circles and triangles etc. are all treated uniformly as `Shape`s.

Interfaces

An interface is a class in which:

- All methods are `public` and `abstract`
- All fields are `public`, `static` and `final` (i.e. constants).

Effectively, an interface contains purely type information and no code.

Interfaces

Use the `interface` keyword instead of `class`.

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

Methods are public,static by default.
Fields are public,static,final by default.

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

Implementing an interface

```
public class Circle implements Shape {
    private double radius;
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
public class Triangle implements Shape {
    private double base, height;
    public double getArea() {
        return base / 2.0 * height;
    }
}
```

A class *implements* rather than *extends* an interface.

No subclassing is involved - only subtyping.

Interfaces vs Classes

A class **cannot** extend multiple superclasses.

```
    class A extends B, C, D
```

error

An class **can** implement multiple interfaces.

```
class A implements B, C, D
```

OK