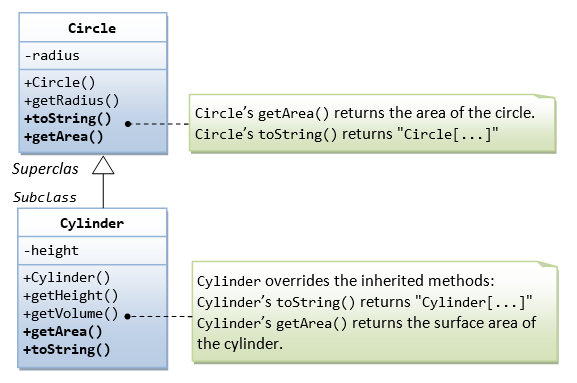
**Polymorphism**

The word "polymorphism" means "many forms". It comes from Greek word "poly" (means many) and "morphos" (means form). For examples, in chemistry, carbon exhibits polymorphism because it can be found in more than one form: graphite and diamond. But, each of the form has it own distinct properties (and price).

**4.1  Substitutability**

A subclass possesses all the attributes and operations of its superclass (because a subclass inherited all attributes and operations from its superclass). This means that a subclass object can do whatever its superclass can do. As a result, we can *substitute* a subclass instance when a superclass instance is expected, and everything shall work fine. This is called *substitutability*.



In our earlier example of Circle and Cylinder: Cylinder is a subclass of Circle. We can say that Cylinder "is-a" Circle (actually, it "is-more-than-a" Circle). Subclass-superclass exhibits a so called "is-a" relationship.

##### Circle.java

// The superclass Circle

public class Circle {

// private instance variable

private double radius;

// Constructor

public Circle(double radius) {

this.radius = radius;

}

// Getter

public double getRadius() {

return this.radius;

}

// Return the area of this circle

public double getArea() {

return radius \* radius \* Math.PI;

}

// Describe itself

public String toString() {

return "Circle[radius=" + radius + "]";

}

}

##### Cylinder.java

// The subclass Cylinder

public class **Cylinder extends Circle** {

// private instance variable

private double height;

// Constructor

public Cylinder(double height, double radius) {

super(radius);

this.height = height;

}

// Getter

public double getHeight() {

return this.height;

}

// Return the volume of this cylinder

public double getVolumne() {

return super.getArea() \* height;

}

// Override the inherited method to return the surface area

@Override

public double getArea() {

return 2.0 \* Math.PI \* getRadius() \* height;

}

// Override the inherited method to describe itself

@Override

public String toString() {

return "Cylinder[height=" + height + "," + super.toString() + "]";

}

}

Via substitutability, we can create an instance of Cylinder, and assign it to a Circle (its superclass) reference, as follows:

// Substitute a subclass instance to a superclass reference

Circle c1 = new Cylinder(1.1, 2.2);

You can invoke all the methods defined in the Circle class for the reference c1, (which is actually holding a Cylinder object), e.g.

// Invoke superclass Circle's methods

c1.getRadius();

This is because a subclass instance possesses all the properties of its superclass.

However, you CANNOT invoke methods defined in the Cylinder class for the reference c1, e.g.

// CANNOT invoke method in Cylinder as it is a Circle reference!

c1.getHeight(); // compilation error

c1.getVolume(); // compilation error

This is because c1 is a reference to the Circle class, which does not know about methods defined in the subclass Cylinder.

c1 is a reference to the Circle class, but holds an object of its subclass Cylinder. The reference c1, however, retains its internal identity. In our example, the subclass Cylinder overrides methods getArea() and toString(). c1.getArea() or c1.toString() invokes the overridden version defined in the subclass Cylinder, instead of the version defined in Circle. This is because c1 is in fact holding a Cylinder object internally.

c1.toString(); // Run the overridden version!

c1.getArea(); // Run the overridden version!

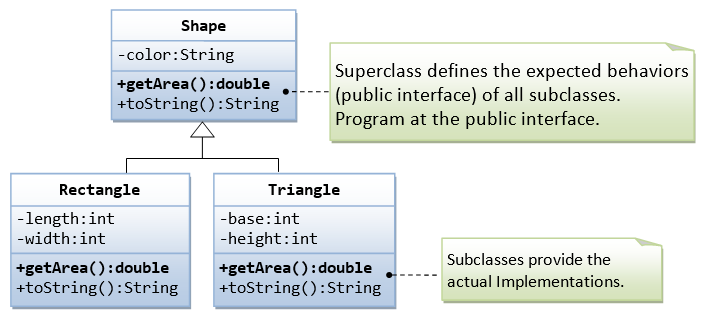
##### Summary

1. A subclass instance can be assigned (substituted) to a superclass' reference.
2. Once substituted, we can invoke methods defined in the superclass; we cannot invoke methods defined in the subclass.
3. However, if the subclass overrides inherited methods from the superclass, the subclass (overridden) versions will be invoked.

#### 4.2  Polymorphism EG. 1： Shape and its Subclasses

Polymorphism is very powerful in OOP to *separate the interface and implementation* so as to allow the programmer to *program at the interface* in the design of a *complex system*.

Consider the following example. Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on. We should design a superclass called Shape, which defines the public interfaces (or behaviors) of all the shapes. For example, we would like all the shapes to have a method called getArea(), which returns the area of that particular shape. The Shape class can be written as follow.



##### The Superclass Shape.java

/\*

\* Superclass Shape maintain the common properties of all shapes

\*/

public class **Shape** {

// Private member variable

private String color;

// Constructor

public Shape (String color) {

this.color = color;

}

@Override

public String toString() {

return "Shape[color=" + color + "]";

}

// All shapes must have a method called getArea().

public double getArea() {

// We have a problem here!

// We need to return some value to compile the program.

System.err.println("Shape unknown! Cannot compute area!");

return 0;

}

}

Take note that we have a problem writing the getArea() method in the Shape class, because the area cannot be computed unless the actual shape is known. We shall print an error message for the time being. In the later section, I shall show you how to resolve this problem.

We can then derive subclasses, such as Triangle and Rectangle, from the superclass Shape.

##### The Subclass Rectangle.java

/\*

\* The Rectangle class, subclass of Shape

\*/

public class **Rectangle extends Shape** {

// Private member variables

private int length;

private int width;

// Constructor

public Rectangle(String color, int length, int width) {

super(color);

this.length = length;

this.width = width;

}

@Override

public String toString() {

return "Rectangle[length=" + length + ",width=" + width + "," + super.toString() + "]";

}

// Override the inherited getArea() to provide the proper implementation

@Override

public double getArea() {

return length\*width;

}

}

##### The Subclass Triangle.java

/\*

\* The Triangle class, subclass of Shape

\*/

public class **Triangle extends Shape** {

// Private member variables

private int base;

private int height;

// Constructor

public Triangle(String color, int base, int height) {

super(color);

this.base = base;

this.height = height;

}

@Override

public String toString() {

return "Triangle[base=" + base + ",height=" + height + "," + super.toString() + "]";

}

// Override the inherited getArea() to provide the proper implementation

@Override

public double getArea() {

return 0.5\*base\*height;

}

}

The subclasses override the getArea() method inherited from the superclass, and provide the proper implementations for getArea().

##### A Test Driver (TestShape.java)

In our application, we could create references of Shape, and assigned them instances of subclasses, as follows:

/\*

\* A test driver for Shape and its subclasses

\*/

public class **TestShape** {

public static void main(String[] args) {

Shape s1 = new Rectangle("red", 4, 5); // Upcast

System.out.println(s1); // Run Rectangle's toString()

System.out.println("Area is " + s1.getArea()); // Run Rectangle's getArea()

Shape s2 = new Triangle("blue", 4, 5); // Upcast

System.out.println(s2); // Run Triangle's toString()

System.out.println("Area is " + s2.getArea()); // Run Triangle's getArea()

}

}

The expected outputs are:

Rectangle[length=4,width=5,Shape[color=red]]

Area is 20.0

Triangle[base=4,height=5,Shape[color=blue]]

Area is 10.0

The beauty of this code is that all the references are from the superclass (i.e., programming at the interface level). You could instantiate different subclass instance, and the code still works. You could extend your program easily by adding in more subclasses, such as Circle, Square, etc, with ease.

Nonetheless, the above definition of Shape class poses a problem, if someone instantiate a Shape object and invoke the getArea() from the Shape object, the program breaks.

public class **TestShape** {

public static void main(String[] args) {

// Constructing a Shape instance poses problem!

Shape s3 = new Shape("green");

System.out.println(s3);

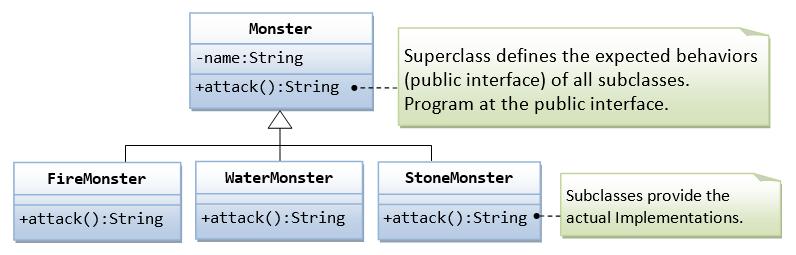
System.out.println("Area is " + s3.getArea()); // Invalid output

}

}

This is because the Shape class is meant to provide a common interface to all its subclasses, which are supposed to provide the actual implementation. We do not want anyone to instantiate a Shape instance. This problem can be resolved by using the so-called abstract class.

#### 4.3  Polymorphism EG. 2： Monster and its Subclasses



Polymorphism is a powerful mechanism in OOP to separate the interface and implementation so as to allow the programmer to program at the interface in the design of a complex system. For example, in our game app, we have many types of monsters that can attack. We shall design a superclass called Monster and define the method attack() in the superclass. The subclasses shall then provides their actual implementation. In the main program, we declare instances of superclass, substituted with actual subclass; and invoke method defined in the superclass.

##### Superclass Monster.java

/\*

\* The superclass Monster defines the expected common behaviors for its subclasses.

\*/

public class Monster {

// private instance variable

private String name;

// Constructor

public Monster(String name) {

this.name = name;

}

// Define common behavior for all its subclasses

public String attack() {

return "!^\_&^$@+%$\* I don't know how to attack!";

// We have a problem here!

// We need to return a String; else, compilation error!

}

}

##### Subclass FireMonster.java

public class FireMonster extends Monster {

// Constructor

public FireMonster(String name) {

super(name);

}

// Subclass provides actual implementation

@Override public String attack() {

return "Attack with fire!";

}

}

##### Subclass WaterMonster.java

public class WaterMonster extends Monster {

// Constructor

public WaterMonster(String name) {

super(name);

}

// Subclass provides actual implementation

@Override public String attack() {

return "Attack with water!";

}

}

##### Subclass StoneMonster.java

public class StoneMonster extends Monster {

// Constructor

public StoneMonster(String name) {

super(name);

}

// Subclass provides actual implementation

@Override public String attack() {

return "Attack with stones!";

}

}

##### A Test Driver TestMonster.java

public class **TestMonster** {

public static void main(String[] args) {

// Program at the "interface" defined in the superclass.

// Declare instances of the superclass, substituted by subclasses.

Monster m1 = new FireMonster("r2u2"); // upcast

Monster m2 = new WaterMonster("u2r2"); // upcast

Monster m3 = new StoneMonster("r2r2"); // upcast

// Invoke the actual implementation

System.out.println(m1.attack()); // Run FireMonster's attack()

System.out.println(m2.attack()); // Run WaterMonster's attack()

System.out.println(m3.attack()); // Run StoneMonster's attack()

// m1 dies, generate a new instance and re-assign to m1.

m1 = new StoneMonster("a2b2"); // upcast

System.out.println(m1.attack()); // Run StoneMonster's attack()

// We have a problem here!!!

Monster m4 = new Monster("u2u2");

System.out.println(m4.attack()); // garbage!!!

}

}

#### 4.4  Upcasting & Downcasting

##### Upcasting a Subclass Instance to a Superclass Reference

Substituting a subclass instance for its superclass is called "upcasting". This is because, in a UML class diagram, subclass is often drawn below its superclass. Upcasting is always safe because a subclass instance possesses all the properties of its superclass and can do whatever its superclass can do. The compiler checks for valid upcasting and issues error "incompatible types" otherwise. For example,

Circle c1 = new Cylinder(1.1, 2.2); // Compiler checks to ensure that R-value is a subclass of L-value.

Circle c2 = new String(); // Compilation error: incompatible types

##### Downcasting a Substituted Reference to Its Original Class

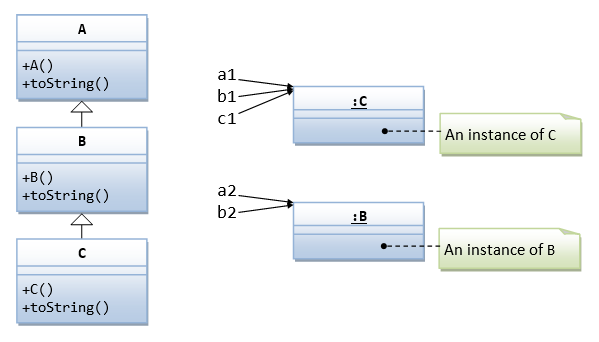
You can revert a substituted instance back to a subclass reference. This is called "downcasting". For example,

Circle c1 = new Cylinder(1.1, 2.2); // upcast is safe

Cylinder cy1 = (Cylinder) c1; // downcast needs the casting operator

Downcasting requires explicit type casting operator in the form of prefix operator (new-type). Downcasting is not always safe, and throws a runtime ClassCastException if the instance to be downcasted does not belong to the correct subclass. A subclass object can be substituted for its superclass, but the reverse is not true.

##### Another Example on Upcasting and Downcasting



public class **A** {

public A() { // Constructor

System.out.println("Constructed A");

}

public String toString() {

return "This is A";

}

}

public class **B extends A** {

public B() { // Constructor

super();

System.out.println("Constructed B");

}

@Override

public String toString() {

return "This is B";

}

}

public class **C extends B** {

public C() { // Constructor

super();

System.out.println("Constructed C");

}

@Override

public String toString() {

return "This is C";

}

}

The following program tests the upcasting an downcasting (refer to the above instance diagram):

public class **TestCasting** {

public static void main(String[] args) {

A a1 = new C(); // upcast

System.out.println(a1); // run C's toString()

B b1 = (B)a1; // downcast okay

C c1 = (C)b1; // downcast okay

A a2 = new B(); // upcast

System.out.println(a2); // run B's toString()

B b2 = (B)a2; // downcast okay

C c2 = (C)a2; // compilation okay, but runtime error ClassCastException

}

}

##### Casting Operator

Compiler may not be able to detect error in explicit cast, which will be detected only at runtime. For example,

Circle c1 = new Circle(5);

Point p1 = new Point();

c1 = p1; // compilation error: incompatible types (Point is not a subclass of Circle)

c1 = (Circle)p1; // runtime error: java.lang.ClassCastException: Point cannot be casted to Circle

#### 4.5  The "instanceof" Operator

Java provides a binary operator called instanceof which returns true if an object is an instance of a particular class. The syntax is as follows:

anObject **instanceof** aClass

Circle c1 = new Circle();

System.out.println(c1 instanceof Circle); // true

if (c1 instanceof Circle) { ...... }

An instance of subclass is also an instance of its superclass. For example,

Circle c1 = new Circle(1.1);

Cylinder cy1 = new Cylinder(2.2, 3.3);

System.out.println(c1 instanceof Circle); // true

System.out.println(c1 instanceof Cylinder); // false

System.out.println(cy1 instanceof Cylinder); // true

System.out.println(cy1 instanceof Circle); // true

Circle c2 = new Cylinder(4.4, 5.5);

System.out.println(c2 instanceof Circle); // true

System.out.println(c2 instanceof Cylinder); // true

#### 4.6  Summary of Polymorphism

1. A subclass instance processes all the attributes operations of its superclass. When a superclass instance is expected, it can be substituted by a subclass instance. In other words, a reference to a class may hold an instance of that class or an instance of one of its subclasses - it is called substitutability.
2. If a subclass instance is assign to a superclass reference, you can invoke the methods defined in the superclass only. You cannot invoke methods defined in the subclass.
3. However, the substituted instance retains its own identity in terms of overridden methods and hiding variables. If the subclass overrides methods in the superclass, the subclass's version will be executed, instead of the superclass's version.

Assignment 1

