Note #4 – Inheritance / Polymorphism

Dr. Lily Chang

Inheritance is an important and powerful feature for reusing software. Suppose you need to define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so as to avoid redundancy and make the system easy to comprehend and easy to maintain?

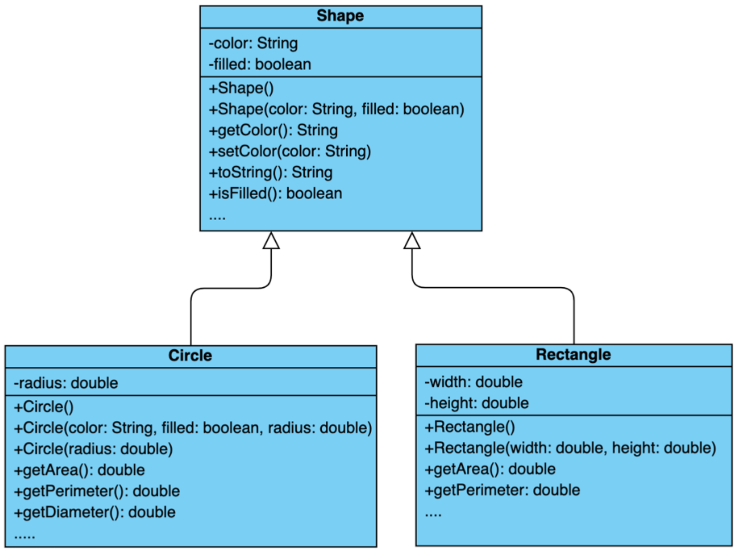
Business world is dynamic and challenging. Software help the businesses make important decisions and stay competitive. Often times, software must change to meet new requirements and business needs. What would be the best approach to minimize the impact on the running software and incorporate new functionalities into the software?

- **Superclass (parent class) and Subclass (child class) in Java**

A Java class is used to model objects with the same properties (data) and behaviors (operations.) Different classes may have some common properties and behaviors, which can be generalized in a "superclass" that can be shared by the "subclasses". A subclass is a specialized class that extends the superclass, which is a generalized class. Subclasses inherit the properties and behaviors from the superclass and define additional properties and behaviors. Structuring the Java code with the inheritance relationship removes the duplicate code in the subclasses. A subclass can also be considered as an "extension" of the superclass. When a software change is necessary, defining a subclass to extend or modify the behavior will minimize the impact on the existing code.

Consider the geometric objects. Suppose you want to design the classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors. For example, they can be drawn in a certain color and be filled or unfilled. In this case, a general **Shape** class can be defined to model the geometric objects sharing the properties **color** and **filled** and their appropriate getter and setter methods. Shape class also contains the **toString()** method, which returns a textual representation of the object, and it is an overriding method implemented in most Java classes.

Since a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. Therefore, it makes sense to define the **Circle** class that extends the **Shape** class. Likewise, **Rectangle** can also be defined as a special type of geometric objects. The following figure shows the relationship among these classes. A triangular arrow pointing to the generalized class is used to denote the generalization (inheritance) relationship between the two classes involved.



In Java's terminology, if class **C1** extends another class **C2**, C1 is the **subclass**, and **C2** is the **superclass**. A superclass is also referred to as a *parent class* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*.

A subclass inherits the instance variables and accessible methods from its superclass and may also add new data fields and methods. In the example above, **Circle** and **Rectangle** are subclasses of **Shape class**, and **Shape class** is the superclass for **Circle** and **Rectangle**. A class defines an abstract data type (ADT.) A type defined by a subclass is called a **subtype**, and a type defined by its superclass is called a **supertype**. In other words, **Circle** is a subtype of **Shape**, and **Shape** is a supertype for **Circle**.

The subclass and its superclass are said to form a ***is-a*** relationship. A **Circle** object is a special type of general **Shape**. The Circle class inherits all accessible data fields and methods from the Shape class. In addition, it has a new data field, **radius** and its associated getter and setter methods. The Circle class also contains additional methods for returning the area, perimeter and diameter of the circle object.

The **Rectangle** class inherits all accessible data fields and methods from the **Shape** class. In addition, it has the data fields **width** and **height** and their associated getter and setter methods. It also contains the methods for returning the area and perimeter of the rectangle. Note that you may have used the terms width and length to describe the sides of a rectangle in geometry. The common terms used in computer science are width and height, where width refers to the horizontal length, and height to the vertical length.

In Java, the keyword **extends** is used to tell the compiler that the subclass extends the superclass. For example, the **Circle** class above extends the **Shape** class with the following syntax and inherits the data fields and all the public methods defined in the Shape class.

```
public class Circle extends Shape
                 ↑               ↑
              subclass       superclass
```

- **The "super" keyword**

In Java, every class has one and only one direct superclass (single inheritance.) A subclass inherits data and operations from the superclass, but NOT constructors. A subclass can override a superclass's methods when a change of the behavior is necessary. Classes that don't explicitly inherit from another class inherit from Object class. You can use a subclass wherever a superclass class is expected. That is, assigning a reference of subtype to a variable of supertype.

There are some rules a subclass must follow. The first line in a subclass's constructor must invoke a superclass's constructor via "super" (superclass). If a subclass overrides a superclass's method and want to run the superclass's version, use the **super** key word. For example, **super()**, or **super(arguments).** If a superclass wants to give the subclasses direct access to its data, they must be declared as **protected.**

The statement **super()** at the first line in a subclass's constructor invokes the default (no-argument) constructor of its superclass, and the statement **super(arguments)** invokes the superclass's constructor that matches the **arguments**. The statement **super()** or **super(arguments)** must be the first statement of a subclass's constructor. Invoking a superclass's constructors with the constructor names will cause a syntax error. Using "super" is the only way to explicitly invoke a superclass's constructor. For example,

```
public Circle(double radius, String color, boolean filled) {
    super(color, filled); //invoke the superclass's constructor with 2 arguments
    this.radius = radius;
}
```

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts **super()** as the first statement in the constructor. For example:

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(parameters) {
  // some statements
}
```

Equivalent

```
public ClassName(parameters) {
  super();
  // some statements
}
```

In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks. If the superclass is derived from another class, the superclass constructor invokes its superclass's constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is called **constructor chaining**. If possible, you should provide a default (no-argument) constructor for every class to make the class easy to extend and to avoid errors.

The keyword **super** can also be used to reference a method other than the constructor in the superclass. The syntax is: **super.method(arguments);** note that the following syntax will give you an error: **super.super.method();**

- **Overriding Methods**

A subclass inherits methods from a superclass. Sometimes, it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding. **toString()** and **equals()** methods are good examples of overriding the behaviors of the superclass. Because, in most cases, there are additional data fields in the subclasses, how you determine 2 objects of the subclasses are equal is different. For example, the toString() method in the Circle class.

```
public String toString() {
    return super.toString() + "\nradius is " + radius;
}
```

The overriding method must have the same signature as the overridden method and same or compatible return type. Compatible means that the overriding method's return type is a subtype of the overridden method's return type.

An instance method can be overridden only if it is accessible. Thus, a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax SuperClassName.staticMethodName().

More examples below.

```java
public class Box {
   private int x, y; //upper-left hand corner of the Box
   private int width, height;

   public Box( int startX, int startY, int w, int h) {
      x = startX;
      y = startY;
      width = w;
      height = h;
   }

   public void show() {
      // Some code that draws the box
   }

   public void hide() {
      ...
   }

   public void resize ( int newW, int newH) {
      hide();
      width = newW;
      height = newH;
      show();
   }

   public void move ( int deltaX, int deltaY ) {
      hide();
      x = x + deltaX;
      y = y + deltaY;
      show();
   }

   public int area() {
      return width * height;
   }
}
```

parent class

```java
public class ColoredBox extends Box {
   private Color color;
   public ColoredBox (int startX, int startY, int w,
                      int h, Color c ) {
      super( startX, startY, w, h );
      color = c;
   }

   public void show() {
       .....
       super.show();   // if you want to call it
       .....
   }

   public Color getColor() {
      return color;
   }
}
```

child class

```java
public class Bicycle {
   private int cadence;
   private int gear;
   private int speed;
   public Bicycle(int startCadence, int startSpeed,
                  int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }
   public void setCadence(int newValue) {
      cadence = newValue;
   }
   public void setGear(int newValue) {
      gear = newValue;
   }
   public void applyBrake(int decrement) {
      speed -= decrement;
   }
   public void speedUp(int increment) {
      speed += increment;
   }
}
```

parent class

```java
public class MountainBike extends Bicycle {
// the MountainBike subclass adds one field
   private int seatHeight;
   // the MountainBike subclass has one constructor
   public MountainBike(int startHeight,
                       int startCadence,
                       int startSpeed,
                       int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
   }
    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
       seatHeight = newValue;
    }
}
```

child class

- **Overriding vs. Overloading**

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass. For example, in TestOverriding class below, the method **p(double i)** in class **A** overrides the same method defined in class **B**. In the TestOverloading class however, the class **A** has two overloaded methods: **p(double i)** and **p(int i)**. The method **p(double i)** is inherited from **B**.

```java
public class TestOverloading {
    public static void main(String args[]) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    public void p(int i) {
        System.out.println(i); //overload the method in B
    }
}
```

```java
public class TestOverriding {
    public static void main(String args[]) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    public void p(double i) {
        System.out.println(i); //override the method in B
    }
}
```

When you run the **TestOverriding** class, both **a.p(10)** and **a.p(10.0)** invoke the **p(double i)** method defined in class **A** to display **10.0**. When you run the **TestOverloading** class, **a.p(10)** invokes the **p(int i)** method defined in class **A** to display **10** and **a.p(10.0)** invokes the **p(double i)** method defined in class **B** to display **20.0**.

Note that, overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class, or in different classes related by inheritance. Overridden methods have the same signature; overloaded methods have the same name but different parameter lists. To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place **@Override** before the overriding method in the subclass. This annotation denotes that the annotated method is required to override a method in its superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error. For example, if toString() is mistyped as tostring, a compile error is reported. If the @Override annotation isn't used, the compiler won't report an error. Using the @Override annotation avoids mistakes.

- **Polymorphism**

Three important features of object-oriented programming are encapsulation, inheritance, and polymorphism. The inheritance relationship enables a subclass to inherit data and operations from its superclass and define additional data and operations specific to the subclass. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle object is a Shape object, but not every Shape object is a circle object. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type. Consider the code below, what is the output?

```java
public class PolymorphismDemo {
    public static void main(String[] args) {
        displayObject(new Circle(1, "red", false));
        displayObject(new Rectangle(1, 1, "black", true));
    }

    public static void displayObject(Shape object) {
        System.out.println(object.toString() + " color is "
                         + object.getColor());
    }

}
```

Console output:

```
Circle color is red
Rectangle color is black
```

An object of a subclass can be used wherever its superclass object is used. This is commonly known as **polymorphism** (from a Greek word meaning "many forms"). In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object. The supertype is a "generalization" of the subtypes. Below is another example.

```java
public class Box {
   private int x, y; //upper-left hand corner of the Box
   private int width, height;

   public Box( int startX, int startY, int w, int h) {
      x = startX;
      y = startY;
      width = w;
      height = h;
   }

   public void show() {
      // Some code that draws the box
   }

   public void hide() {
      ...
   }

   public void resize ( int newW, int newH) {
      hide();
      width = newW;
      height = newH;
      show();
   }

   public void move ( int deltaX, int deltaY ) {
      hide();
      x = x + deltaX;
      y = y + deltaY;
      show();
   }

   public int area() {
      return width * height;
   }
}
```

parent class

```java
public class ColoredBox extends Box {
   private Color color;
   public ColoredBox (int startX, int startY, int w,
                     int h, Color c ) {
      super( startX, startY, w, h );
      color = c;
   }

   public void show() {
       .....
       super.show();   // if you want to call it
       .....
   }

   public Color getColor() {
      return color;
   }
}
```

child class

```java
Box b = new Box( 50, 40, 20, 30 );
ColoredBox cb = new ColoredBox( 80, 60, 10, 20, Color.red );
cb = b; //invalid; assignment expected a ColoredBox
b = cb; //valid; assignment expected a Box, got a child is OK
```

In the above code segment, cb is an instance of ColoredBox class, and b is an instance of the supertype Box class. If you assign the reference of b to cb, you will get a compile error since cb is an instance of the subclass ColoredBox, which is more specific and includes additional data fields that the Box class doesn't have. On the other hand, although b is an instance of Box class, you can assign the reference of cb to b since the ColoredBox is a subtype of Box. The problem is,

1. Given cb.move(3, 4); which show() method is called when move is executed?
2. Given b.show(); which show() method is called, Box's show or ColoredBox's show?

As another example, which **toString()** method, Shape class's or Object class's, is invoked by **obj** based on the following code?

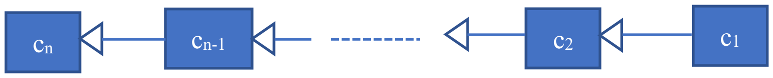
```java
Object obj = new Shape("red", false);
System.out.println(obj.toString());
```

A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime. When a program is running and a method is activated, the Java Runtime System **checks the data type of the actual object** and uses the method from that type (rather than the method from the type of the reference variable.

A variable must be declared a type. The type that declares a variable is called the variable's **declared type**. Here, obj's declared type is Object. A variable of a reference type can hold a null value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype. The **actual type** of the variable is the actual class for the object referenced by the variable. Here, obj's actual type is Shape, because obj references an object created using new Shape("red", false). As for which toString() method is invoked by obj is determined by obj's actual type. This is known as **dynamic binding**. The "power" of polymorphism is extensibility. Write code that can handle changes in the future without being modified!

Matching a method signature and binding a method implementation are two separate issues. The declared type of the reference variable decides which method to match at compile time. The compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time. A method may be implemented in several classes along the inheritance chain. The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable.

Dynamic binding works as follows: Suppose that an object obj is an instance of classes C_1. There is an inheritance chain C_1, C_2, . . . , C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, . . . , and C_{n-1} is a subclass of C_n, as shown below. That is, C_n is the most general class, and C_1 is the most specific class. In Java's inheritance chain, C_n is the Object class. If object obj is an instance of C_1 and invokes a method p, the JVM searches for the implementation of the method p in the order of C_1, C_2, . . . , C_{n-1}, and C_n, until the p method is found. Once an implementation is found, the search stops and **the first-found implementation is invoked**.



- **LSP (Liskov Substitution Principle)**

1. A child class (subclass) can be used wherever a parent class (superclass) is expected; child class must be completely substitutable for their parent class
2. For every overriding method in a child class: require no more, promise no less; make sure a child class just extend without replacing the functionality of the superclass
3. Use the notion of "is-a" and LSP to determine if your inheritance is good - don't overdo inheritance!

- **Casting Objects and the instanceof Operator**

One object reference can be typecast into another object reference. This is called casting object. For example, the statement **Object obj = new Student(),** known as **implicit casting**, is valid because an instance of Student is an instance of Object, which is a superclass of all Java classes. Suppose you want to assign the object reference **obj** to a variable of the Student type using the following statement.

```
Student student = obj;
```

In this case, a compile error occurs. Why does the statement **Object obj = new Student()** work, but **Student student = obj** doesn't? The reason is that a **Student** object is always an instance of **Object**, but an **Object** is not necessarily an instance of **Student**. Even though you can see that **obj** is really a **Student** object, the compiler is not clever enough to know it. To tell the compiler **obj** is a **Student** object, use **explicit casting**. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows.

```
Student student = (Student) obj;
```

To help understand casting, you may also consider the analogy of fruit, apple, and orange, with the **Fruit** class as the superclass for **Apple** and **Orange**. An apple is a fruit, so you can always safely assign an instance of **Apple** to a variable for **Fruit**. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of **Fruit** to a variable of **Apple**.

It is always possible to cast an instance of a subclass to a variable of a superclass (known as **upcasting**) because an instance of a subclass is always an instance of its superclass. When casting an instance of a superclass to a variable of its subclass (known as **downcasting**), explicit casting must be used to confirm your intention to the compiler with the **(SubclassName)** cast notation. For the casting to be successful, you must make sure the object to be cast is an instance of the subclass. If the superclass object is not an instance of the subclass, a runtime ***ClassCastException*** occurs. For example, if an object is not an instance of **Student**, it cannot be cast into a variable of **Student**. It is a good practice, therefore, to ensure the object is an instance of another object before attempting a casting. This can be accomplished by using the **instanceof** operator.

```java
public void someMethod(Object obj) {
    if (obj instanceof Circle) {
        Circle circle = (Circle) obj;
        System.out.println("circle radius: " + circle.getRadius());
    }

}
```

You may be wondering why casting is necessary. The variable obj is declared Object. The **declared type decides which method to match at compile time**. Using obj.getRadius() will cause a compile error, because the Object class does not have the getRadius() method. The compiler cannot find a match for obj.getRadius(). Therefore, it is necessary to cast obj into the Circle type to tell the compiler that obj is also an instance of Circle. Why not declare obj as a Circle type in the first place? To enable **generic programming**, it is a good practice to declare a variable with a supertype that can accept an object of any subtype. As another example, the overriding equals() method.

```java
public boolean equals(Object obj) {
    if (obj instanceof Student) {
        Student s = (Student) obj;
        return s.name.equals(name) &&
                     s.startDate.equals(startDate);
    }
    return false;
}
```

The object member access operator dot (.) precedes the casting operator. Use parentheses to ensure that casting is done before the dot operator.

- **The Protected Data and Methods**

A protected member of a class can be directly accessed from a subclass. So far you have used the private and public keywords to specify whether data fields and methods can be accessed from outside of the class. Private members can be accessed only from inside of the class, and public members can be accessed from any other classes. Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow non-subclasses in different packages to access these data fields and methods. To accomplish this, you can use the **protected** keyword. This way you can access protected data fields or methods in a superclass from its subclasses.

A subclass may override a protected method defined in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

The "final" keyword. You may occasionally want to prevent classes from being extended. In such cases, use the final modifier to indicate a class is final and cannot be a parent class. For example,

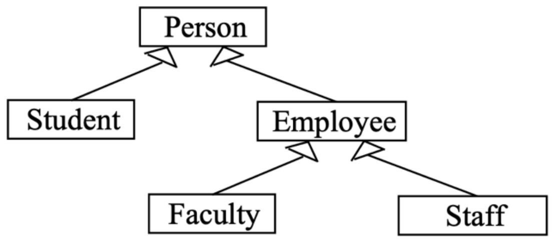
```
public final class Student { } //cannot be extended

public class International extends Student { } //will get a compile error
```

- **Example 1 - the Person, Student, Employee, Faculty, and Staff classes**

Design a class named Person and its two subclasses named Student and Employee. Make Faculty and Staff subclasses of Employee.

- A person has a name
- A student has a standing: 1: freshman, 2: sophomore, 3: junior, or 4: senior.
- An employee has a date hired
- A faculty member has a rank
- A staff member has a title

(a) Draw the UML diagram for the classes.
(b) Implement the default constructor and parameterized constructor.
(c) Implement the overriding **toString()** method in each class to display the class name, and the data fields specific to the classes.
(d) Write a test program that creates a Person, Student, Employee, Faculty, and Staff, and invokes their toString() methods. (Polymorphism)



Example 2 – overloading vs overriding

(1) Will there be a compile error for the following code segment?
(2) If the program runs, what is the output?
(3) Is class B overriding or overloading `m()` method?

```
public class Example1 {
    public static void main(String[] args) {
        B b = new B();
        b.m(5);
        System.out.println("i is " + b.i);
    }
}

class A {
    int i;

    public void m(int i) {
        this.i = i;
    }
}

class B extends A {
    public void m(String s) {
    }
}
```

Example 3 – find errors

```
public class Example2 {
    public static void main(String[] args) {
        m(new GraduateStudent()); //(a) causes a compile error?
        m(new Student()); //(b) causes a compile error?
        m(new Person());  //(c) causes a compile error?
        m(new Object());  //(d) causes a compile error?
    }
    public static void m(Student x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student { }
class Student extends Person {
    @Override
    public String toString() {
        return "Student";
    }
}
class Person {
    @Override
    public String toString() {
        return "Person";
    }
}
```

Example 4 – what is the output of the following code?

```
public class Example3 {
    public static void main(String[] args) {
        new Person().printPerson();
        new Student().printPerson();
    }
}

class Student extends Person {
    private String getInfo() {
        return "Student";
    }
}

class Person {
    private String getInfo() {
        return "Person";
    }

    public void printPerson() {
        System.out.println(getInfo());
    }
}
```

Example 5 – what is the output of the following code?

```
public class Example4_1 { // Program 1:
    public static void main(String[] args) {
        Object a1 = new A();
        Object a2 = new A();
        System.out.println(a1.equals(a2)); // true or false?
    }
}

class A {
    int x;
    public boolean equals(Object a) {
        return this.x == ((A)a).x;
    }
}

public class Example4_2 { // Program 2:
    public static void main(String[] args) {
        Object a1 = new A();
        Object a2 = new A();
        System.out.println(a1.equals(a2)); // true or false?
    }
}

class A {
    int x;
    public boolean equals(A a) {
        return this.x == a.x;
} }
```