Note #4 – Polymorphism

- **Polymorphism**

The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism. The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. 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 is a geometric object, but not every geometric object is a circle. 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 {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new Circle(1, "red", false));
    displayObject(new Rectangle(1, 1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(GeometricObject object) {
    System.out.println("Created on " + object.getDateCreated() +
      ". Color is " + object.getColor());
  }
}
```

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.

```java
public class Box {
protected int x, y; //upper-left hand corner of the Box
   protected 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() {
      show();
   }

   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

```
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
```

Suppose cb and b refer to a ColoredBox, then some questions that arise:
1. Given cb.move(3, 4); what show() is called when move is executed?
2. Given b.show(); what show() is called, Box's show or ColoredBox's show?

Another example, which **toString()** method is invoked by **o** based on the following code?

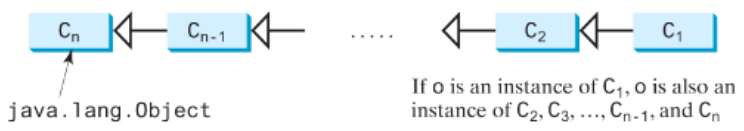
```
Object o = new GeometricObject();
System.out.println(o.toString());
```

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, o'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, o's actual type is GeometricObject, because o references an object created using new GeometricObject(). Which toString() method is invoked by o is determined by o'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 o is an instance of classes C1, C2, . . . , Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, . . . , and Cn-1 is a subclass of Cn, as shown below. That is, Cn is the most general class, and C1 is the most specific class. In Java, Cn is the Object class. If o invokes a method p, the JVM searches for the implementation of the method p in C1, C2, . . . , Cn-1, and Cn, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.


C_n
C_{n-1}
.....
C_2
C_1
java.lang.Object
If o is an instance of C_1, o is also an instance of C_2, C_3, ..., C_{n-1}, and C_n

- **LSP (Liskov Substitution Principle)**

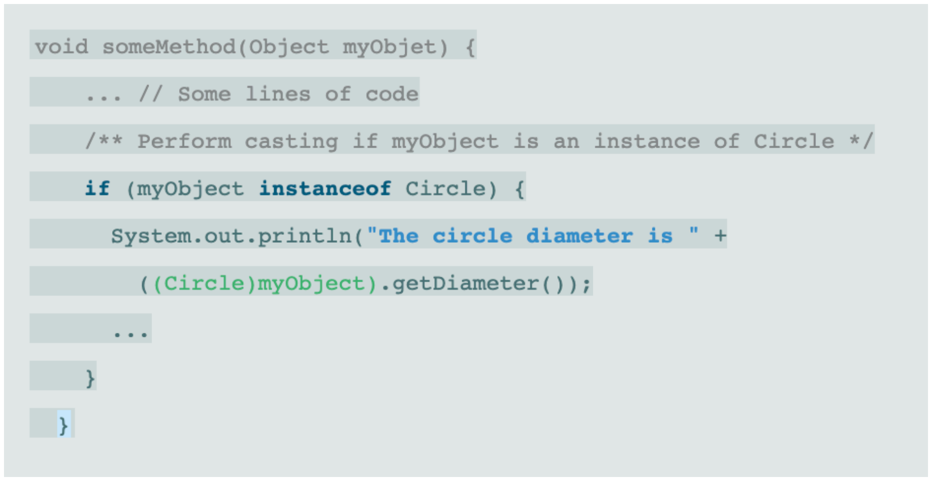
1. A child can be used wherever a parent is expected; child class must be completely substitutable for their parent class
2. For every overriding method in a child class: require no more, promise no less; make sure a child class just extend without replacing the functionality of old classes
3. Use the 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 o = new Student(),** known as **implicit casting**, is valid because an instance of Student is an instance of Object. Suppose you want to assign the object reference o to a variable of the Student type using the following statement: **Student s = o;** in this case a compile error would occur. Why does the statement **Object o = new Student()** work, but **Student s = o** 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 **o** is really a **Student** object, the compiler is not clever enough to know it. To tell the compiler **o** 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 s = (Student) o;**

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.



```
void someMethod(Object myObjet) {
    ... // Some lines of code
    /** Perform casting if myObject is an instance of Circle */
    if (myObject instanceof Circle) {
      System.out.println("The circle diameter is " +
        ((Circle)myObject).getDiameter());
      ...
    }
  }
```

You may be wondering why casting is necessary. The variable myObject is declared Object. The **declared type decides which method to match at compile time**. Using myObject.getDiameter() would cause a compile error, because the Object class does not have the getDiameter method. The compiler cannot find a match for myObject.getDiameter(). Therefore, it is necessary to cast myObject into the Circle type to tell the compiler that myObject is also an instance of Circle. Why not declare myObject 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 m = (Student) obj;
        return m.name.equals(name) &&
                     m.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.