>>>>>>>>>>>Inheritance and Polymorphism<<<<<<<<<

Brought To You By

{My Life} Programming School

Learn Java By Reading First, Then Watch Someone Tackling Exercises Programmatically By Visiting The Link.
https://www.mylifeprogrammingschool.com

11.1 Introduction

{Key Point} Object-oriented programming allows you to define new classes from existing classes. This is called inheritance. *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? The answer is to use inheritance.

11.2 Superclasses and Subclasses

{Key Point} Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses). You use a class to model objects of the same type. Different classes may have some common properties and behaviours, which can be generalized in a class that can be shared by other classes. You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class. See the following example

```
// General Class                    // Specialized Class
class A {                           class B extends A{
        property 1
        property 2
        …
        property n

        A() {                               B() {

        }                                   }

        A(arguments){                       B(arguments){


        }                                   }

        method1(){}
        method1(){}
        …
        methodm(){}
}                                   }
```

Class B inherit all the properties and methods of class A.

11.3 Using the super Keyword

{Key Point} The keyword **super** refers to the superclass and can be used to invoke the superclass's methods and constructors. A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited by a subclass. They can only be invoked from the constructors of the subclasses using the keyword **super**. The syntax to call a superclass's constructor is: **super**(), or **super**(parameters); The statement **super()** invokes the no-arg constructor of its superclass, and the statement **super(arguments)** invokes the superclass constructor that matches the **arguments**. The statement **super()** or **super(arguments)** must be the first statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor.

```
class A {

        A() {

        }

        A(arguments){


        }
}
```

```
class B extends A{

        B() {
                //Invokes A() implicitly
        }

        B(arguments){
                // Invokes A(arguments)
                super(arguments);
        }
}
```

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

        }
```

```
        public ClassName() {
                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 parent-class constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is called *constructor chaining*.

11.4 Calling Superclass Methods

The keyword **super** can also be used to reference a method other than the constructor in the superclass. The syntax is:

```
super.method(parameters);
```

```
public class Company{

	public Company(){

	}

	void displayCompanyInfo(){
		// Some Code
	}
}

public class PtyCompany extends Company{

	public PtyCompany(){
		super(); // Implicitly called, so it can be removed.

		// Invokes the one on this class.
		displayCompanyInfor();

		// Invokes the one on the super class.
		super.displayCompanyInfor();
	}

	void displayCompanyInfo(){
		// Some Code
	}
}
```

11.5 Overriding Methods

To override a method, the method must be defined in the subclass using the same signature and the same return type as in its 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. The method doSomething() is overridden in the example below.

```
public class MyClass{
	public void doSomething(){
		…
	}
}

public class AnothoerClass extends MyClass{

	@override
	public void doSomething(){
		…
	}
}
```

11.6 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.

```
public class Test {
	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 {
	// This method overrides the method in B
	public void p(double i) {
		System.out.println(i);
	}
}

public class Test {
	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 {
        // This method overloads the method in B
        public void p(int i) {
                System.out.println(i);
        }
}
```

11.7 The Object Class and Its toString() Method

{Key Point} Every class in Java is descended from the **java.lang.Object** class. If no inheritance is specified when a class is defined, the superclass of the class is **Object** by default.

```
public class ClassName {        public class ClassName extends Object{

        ...                             …
}                               }
```

The above two ways of defining a class are equivalent. This section introduces the **toString** method in the **Object** class.The signature of the **toString()** method is:

Invoking **toString()** on an object returns a string that describes the object. By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal. For example, consider the following code for the **Loan** class:

```
        public String toString()

Loan loan = new Loan();
System.out.println(loan.toString());
```

The output for this code displays something like **Loan@15037e5**. This message is not very helpful or informative. Usually you should override the **toString** method so that it returns a descriptive string representation of the object.

11.8 Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object. The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism. You have already learned the first two. This section introduces polymorphism.
*First, let us define two useful terms: subtype and supertype. A class defines a type. A type defined by a subclass is called a subtype, and a type defined by its superclass is called a supertype. Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.*

```
class GeometricObject{          class Circle extends GeometricObject{
        double getArea{}
}                               }

class Rectangle extends GeometricObject{

}
```

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. For example

```
class Test{
        public static void main(String[] args){
                Circle circle = new Circle();
                System.out.print(computeArea(circle));

                Rectangle rectangle = new Rectangle();
                System.out.print(computeArea(rectangle));

        }
        public static double computeArea(GeometricObject o){
                return o.getArea();
        }
}
```

The method **computeArea** method takes a parameter of the **GeometricObject** type. You can invoke **computeArea** by passing any instance of **GeometricObject**. 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.

11.9 Dynamic Binding

{Key Point} A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime. A method can be defined in a superclass and overridden in its subclass. For example, the **toString()** method is defined in the **Object** class and overridden in **GeometricObject**. Consider the following code:

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

Which **toString()** method is invoked by **o**? To answer this question, we first introduce two terms: declared type and actual type. 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*.

Dynamic binding works as follows: Suppose 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**. 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.

11.10 Casting Objects and the instanceof Operator

One object reference can be typecast into another object reference. This is called casting object.

The statement

```
computeArea(new Circle());
```

*assigns the object **circle** to a parameter of the GeometricObject type.*

This statement is equivalent to

```
Object o = new Circle(); // Implicit casting
computeArea (o);
```

*The statement **Object o = new Circle()**, known as implicit casting, is legal because an instance of **Circle** is an instance of **Object**.*

*Suppose you want to assign the object reference **o** to a variable of the **a Circle** type using the following statement:*

```
Circle b = o;
```

*In this case a compile error would occur. Why does the statement **Object o = new Circle()** work but **Circle b = o** doesn't? The reason is that a **Circle** object is always an instance of **Object**, but an **Object** is not necessarily an instance of **Circle**. Even though you can see that **o** is really a **Circle** object, the compiler is not clever enough to know it. To tell the compiler that **o** is a **Circle** 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:*

```
Circle b = (Circle)o; // Explicit casting
```

*For the casting to be successful, you must make sure that 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. It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the **instanceof** operator. Consider the following code:*

```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
        System.out.println("The circle area is " +
        ((Circle)myObject).getArea());
        ...
}
```

11.11 The Object's equals Method

Like the **toString()** method, the **equals(Object)** method is another useful method defined in the **Object** class. Its signature is

```
public boolean equals(Object o)
```

This method tests whether two objects are equal. The syntax for invoking it is:

```
object1.equals(object2);
```

The default implementation of the **equals** method in the **Object** class is:

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

This implementation checks whether two reference variables point to the same object using the **==** operator. You should override this method in your custom class to test whether two distinct objects have the same content. The **equals** method is overridden in many classes in the Java API, such as **java.lang.String** and **java.util.Date**, to compare whether the contents of two objects are equal.

11.12 The protected Data and Methods

{Key Point} A protected member of a class can be 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 nonsubclasses 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. The modifiers **private**, **protected**, and **public** are known as *visibility* or *accessibility modifiers* because they specify how classes and class members are accessed.

```
package 1
public class C1 {

        public int x;
        protected int y;
        int z;
        private int u;

        protected void m() {
        }
}

package 2

public class C3 extends C1 {
        can access x;
        can access y;
        can access z;
        cannot access u;

        can invoke m();
}

package 1
public class C2 {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        can access o.z;
        cannot access o.u;
        can invoke o.m();
}
```

11.13 Preventing Extending and Overriding

{Key Point} Neither a final class nor a final method can be extended. A final data field is a constant. You may occasionally want to prevent classes from being extended. In such cases, use the **final** modifier to indicate that a class is final and cannot be a parent class. The **Math** class is a final class. The **String**, **StringBuilder**, and **StringBuffer** classes are also final classes. For example, the following class **A** is final and cannot be extended:

```
public final class A {
        // Data fields, constructors, and methods omitted
}
```

You also can define a method to be final; a final method cannot be overridden by its subclasses. For example, the following method **m** is final and cannot be overridden:

```
public class Test {
        // Data fields, constructors, and methods omitted
        public final void m() {
        // Do something
        }
}
```

{Note} The modifiers **public**, **protected**, **private**, **static**, **abstract**, and **final** are used on classes and class members (data and methods), except that the **final** modifier can also be used on local variables in a method. A **final** local variable is a constant inside a method.

====================End Of Chapter Eleven=================

Now You Can Checkout Chapter 11 Exercises And Their Corresponding Possible Solution On https://www.mylifeprogrammingschool.com/java