**CHAPTER 11: INHERITANCE AND POLYMORPHISM**

* 1. **INTRODUCTION**

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, Inheritance is the best way to design these classes so as to avoid redundancy and make the system easy to comprehend and easy to maintain.

* 1. **SUPERCLASSES AND SUBCLASSES**

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 behaviors, 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.

In Java terminology, a class C1 extended from another class C2 is called a subclass, and C2 is called a 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 accessible data fields and methods from its superclass and may also add new data fields and methods.

* 1. **USING THE super KEYWORD**

The keyword **super** refers to the superclass and can be used to invoke the superclass’s methods and constructors.

A subclass inherits accessible data fields and methods from its superclass.

The keyword **super** refers to the superclass of the class in which super appears.

It can be used in two ways:

1. To call a superclass constructor
2. To call a superclass method
   * 1. **CALLING SUPERCLASS 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(arguments);**

The statement **super()** invokes the no-arg constructor of its superclass, and the **statement super(arguments)** invokes the superclass constructor that matches the **arguments**.

* + 1. **CONSTRUCTOR CHAINING**

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(){

// statements

}

public ClassName(){

super();

// statements

}

Equivalent

public ClassName(parameters){

// statements

}

public ClassName(parameters){

super();

// statements

}

Equivalent

In any case, constructing an instance of a class invokes the constructors of all the super classes 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.**

* + 1. **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(arguments);**

You could rewrite the printCircle() method in the Circle class as follows:

public void printCircle() {

System.out.println("The circle is created " +

super.getDateCreated() + " and the radius is " + radius);

}

* 1. **OVERRIDING METHODS**

To override a method, the method must be defined in the subclass using the same signature as in its superclass.

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.**

* 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**.
  1. **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.

To override a method, the method must be defined in the subclass using the same signature and the same or compatible return type.

Example showing the difference between Overriding and Overloading.

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 {

// This method overrides the method in B

public void p**(double i)** {

System.out.println(i);

}

}

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 {

// This method overloades the method in B

public void p**(int i)** {

System.out.println(i);

}

}

(a)

(b)

When you run the TestOverriding class in (a), 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 in (b), 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 the following:

* Overridden methods are in different classes related by inheritance; overloaded meth ods 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.

* 1. **THE Object CLASS AND IT’S toString() METHOD**

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{

…

}

Equivalent

The signature of the **toString()** method is:

**public String toString()**

Example:

**public String toString() {**

**return "created on " + dateCreated + "\ncolor: " + color + " and filled: " + filled;**

**}**

* 1. **POLYMORPHISM**

Polymorphism means that a variable of a supertype can refer to a subtype object.

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.

An object of a subclass can be used wherever its superclass object is used. This is commonly known as **polymorphism**.

* 1. **DYNAMIC BINDING**

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();**

**Object o = new GeometricObject();**

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 at runtime.

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**.

* 1. **CASTING OBJECTS AND THE instanceof OPERATOR**

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

In the preceding section, the statement

**m(new Student());**

Assigns the object **new Student()** to a parameter of the Object type. This statement is equivalent to

**Object o = new Student();**

**m(o);**

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

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.

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.

Consider the following code:

**void someMethod(Object myObject) {**

**... // 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());**

**...**

**}**

**}**

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.

NOTE: **instanceof** is a Java keyword. Every letter in a Java keyword is in lowercase.

* 1. **THE Object’s equal METHOD**

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

Another method defined in the **Object** class that is often used is the **equals** method. 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.

You can override the **equals** method in the **Circle** class to compare whether two circles are equal based on their radius as follows:

**@Override**

**public boolean equals(Object o) {**

**if (o instanceof Circle)**

**return radius == ((Circle)o).radius;**

**else**

**return false;**

**}**

**Note**: The **==** comparison operator is used for comparing two primitive-data-type values or for determining whether two objects have the same references.

* 1. **THE ArrayList CLASS**

An **ArrayList** object can be used to store a list of objects.

You can create an array to store objects. However, once the array is created, its size is fixed. Java provides the **ArrayList** class, which can be used to store an unlimited number of objects.

Here are some methods in **ArrayList** .

|  |  |
| --- | --- |
| java.util.ArrayList<E> | |
| +ArrayList() | Creates an empty list. |
| +add(e: E): void | Appends a new element e at the end of this list. |
| +add(index: int, e: E): void | Adds a new element e at the specified index in this list. |
| +clear(): void | Removes all elements from this list. |
| +contains(o: Object): Boolean | Returns true if this list contains the element o. |
| +get(index: int): E | Returns the element from this list at the specified index. |
| +indexOf(o: Object): int | Returns the index of the first matching element in this list. |
| +isEmpty(): Boolean | Returns true if this list contains no elements. |
| +lastIndexOf(o: Object): int | Returns the index of the last matching element in this list. |
| +remove(o: Object): Boolean | Removes the first element CDT from this list. Returns true if an element is removed. |
| +size(): int | Returns the number of elements in this list. |
| +remove(index: int): E | Removes the element at the specified index. Returns the removed element. |
| +set(index: int, e: E): E | Sets the element at the specified index. |

An **ArrayList** stores an unlimited number of objects.

**ArrayList** is known as a generic class with a generic type E. You can specify a concrete type to replace E when creating an **ArrayList**.

Here are the Differences and Similarities between Arrays and **ArrayList.**

|  |  |  |
| --- | --- | --- |
| Operation | Array | ArrayList |
| Creating an array/ArrayList | String[] a = new String[10] | ArrayList list = new ArrayList<>(); |
| Accessing an element | a[index] | list.get(index); |
| Updating an element | a[index] = "London"; | list.set(index, "London"); |
| Returning size | a.length | list.size(); |
| Adding a new element |  | list.add("London"); |
| Inserting a new element |  | list.add(index, "London"); |
| Removing an element |  | list.remove(index); |
| Removing an element |  | list.remove(Object); |
| Removing all elements |  | list.clear(); |

* 1. **USEFUL METHODS FOR LISTS**

Java provides the methods for creating a list from an array, for sorting a list, and for finding maximum and minimum element in a list, and for shuffling a list.

Often you need to create an array list from an array of objects or vice versa. You can write the code using a loop to accomplish this, but an easy way is to use the methods in the Java API. Here is an example to create an array list from an array:

String[] array = {"red", "green", "blue"};

ArrayList list = new ArrayList<>(Arrays.asList(array));

The static method **asList** in the Arrays class returns a list that is passed to the **ArrayList** constructor for creating an **ArrayList**. Conversely, you can use the following code to create an array of objects from an array list:

**String[] array1 = new String[list.size()];**

**list.toArray(array1);**

You can use the static **max** and **min** in the **java.util.Collections** class to return the maximum and minimal element in a list.

* 1. **THE protected DATA AND METHODS**

A protected member of a class can be accessed from a subclass.

Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow nonsubclasses 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.

The modifiers private, protected, and public are known as visibility or accessibility modifiers because they specify how classes and class members are accessed.

The visibility of these modifiers increases in this order:

**Visibility increases**

**private, default (no modifier), protected, public**

Data and Methods Visibility Table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Modifier on Members in a Class | Accessed from the Same Class | Accessed from the Same Package | | Accessed from a Subclass in a Different Package | | Accessed from a Different Package |
| Public | ✓ | ✓ | ✓ | | ✓ | |
| Protected | ✓ | ✓ | ✓ | | – | |
| Default (no modifier) | ✓ | ✓ | – | | – | |
| Private | ✓ | – | – | | – | |

**Note:** A subclass may override a protected method defined in its superclass and change its visibility to public.

* 1. **PREVENTING EXTENDING AND OVERRIDING**

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 a class is final and cannot be a parent class. The **Math** class is a final class. The **String**, **StringBuilder**, and **StringBuffer** classes, and all wrapper classes for primitive data types 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**

**}**

**}**

**THE END!**