14-Inheritance

Author: Vincent Lau

Learning Objectives

- Understand the concept of inheritance
- List the benefits of inheritance
- Understand what can be done in a subclass
- Create a hierarchy of classes using an abstract class
- Understand and use vertical constructor chaining
- Explain when to use the *protected, final* and *super* keyword
- State the differences between abstract classes and interfaces

Introduction

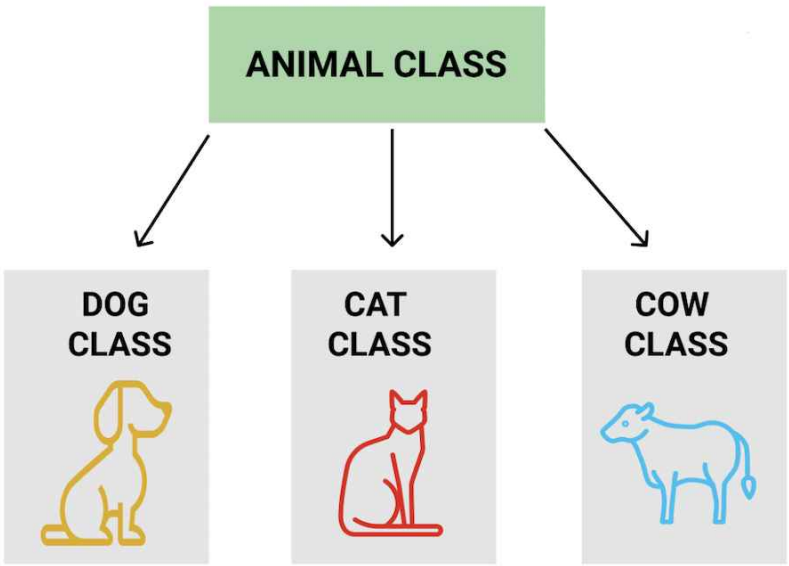
Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). In Java, you can create a new class that extends an existing class to reuse its features and customize or extend its functionality. This new class becomes a subclass of the original class, and it inherits all the non-private members (fields and methods) of the superclass.

Definitions

- A class that inherits another class is called a ***subclass*** (also a *derived class*, *extended class,* or *child class*).
- The class from which the subclass is derived is called a ***superclass*** (also a *base class* or a *parent class*).
- A subclass **inherits all *public, protected, and same-package members*** (including fields, methods, and nested classes) from its superclass. **Constructors are not members**, so they are not inherited by subclasses, but the constructor of the **direct superclass** can be invoked from the subclass.

Concepts & Syntax

- Different kinds of objects often have a certain amount in common with each other. For example, Dog, Cat, and Cow all share the characteristics of Animal (name, age and weight). However, each of them also defines additional features that make them different. **For example, Dog Class may have a barking() method, where Cat does not, since not all Animal knows barking.**
- Object-oriented programming allows classes to *inhereit* commonly used state and behavior from other classes.
- In Java, each class is allowed to have **ONE direct superclass (parent class)**, and each superclass can have an **unlimited number of *subclasses*.**



```java
class Animal {
    String name;
    int age;
    int weight;

    void sound() {
        System.out.println("Animal makes a sound");
    }

    void getName() {
        return this.name;
    }
    // other getters, setters ...
}

class Dog extends Animal {

    void barking() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {

}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.sound(); // Output: Animal makes a sound
        animal.getName();

        Dog dog = new Dog();
        dog.barking(); // Output: Dog barks
        dog.getName(); // Dog object has getName() method as well

        Cat cat = new Cat();
        cat.getName();
    }
}
```

- "extends Animal", this gives Dog all the same fields and methods as Animal, yet allows its code to focus exclusively on the features that make it unique.
- Animal is a normal class in this example. Please try to understand the "Inheritance" concept, and think of the relationship between Animal, Cat and Dog.

Fields & Methods Inheritance

- A subclass inherits all of the ***public*** **and** ***protected*** **instance variables of its parent**, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the ***package-private*** members of its parent **(But instance variable in parent or child should always be private)**.
- The inherited fields and methods can be used directly by subclasses.
- You can declare new fields or methods in the subclass that are not in the superclass.

```
1  public class Vehicle {
2      private String licensePlate;
3      private int capacity;
4
5      public Vehicle(String licensePlate, int capacity) {
6          this.licensePlate = licensePlate;
7          this.capacity = capacity;
8      }
9
10     public int getNumOfPassengers() {
11         return capacity - 1; // minus driver
12     }
13
14     public int getCapacity() {
15         return capacity;
16     }
17
18     public String getLicensePlate() {
19         return licensePlate;
20     }
21
22     public void setLicensePlate(String licensePlate) {
23         this.licensePlate = licensePlate;
24     }
25
26 }
27
28 public class Taxi extends Vehicle {
29     // cannot inherits private fields from superclass
30     private BigDecimal minimalCharge;
31
32     public Taxi(String licensePlate, int capacity, BigDecimal minimalCharge) {
33         super(licensePlate, capacity); // calling superclass's constructor
34         this.minimalCharge = minimalCharge;
35     }
```

```
36
37      // Inherits all public methods from superclass
38      ...
39      // additional method
40      public void setMinimalCharge(BigDecimal minimalCharge) {
41          this.minimalCharge = minimalCharge;
42      }
43  }
```

Constructors Inheritance

- Classes have constructors, but they **can't be instantiated**.
- The purpose of constructors in a class is merely for subclasses to invoke and initialize common attributes.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super` .

Accessing Superclass Members

- An overridden method in the superclass can be accessed from the subclass through the use of the keyword *super*.

```
1  public class Mother {
2      public void doSomething() {
3          System.out.println("[Mother] do something in Mother...");
4      }
5  }
6
7  public class Child extends Mother {
8      @Override
9      public void doSomething() {
10         // Mother mother = new Mother();
11         super.doSomething();
12         System.out.println("[Child] do something in Child...");
13     }
14
15     public static void main(String[] args) {
16         Child child = new Child();
17         child.doSomething();
18     }
19 }
20
21 /*
```

```
Output:
[Mother] do something in Mother...
[Child] do something in Child...
*/
```

Subclass Constructors

- In the subclass constructor, invocation of a superclass constructor must be the first line (For non no-args constructor.
- When the subclass's no-args constructor is invoked, the superclass's no-args constructor will still be called **even when the subclass does not specify the call**. If the superclass does not have a no-args constructor, a compile-time error will be thrown.

```java
public class Superclass {
    private int someValue;

    // No-args constructor in the superclass
    public Superclass() {
        System.out.println("[Superclass] No-args constructor called");
    }

    // Constructor with args in the superclass
    public Superclass(int someValue) {
        this.someValue = someValue;
        System.out.println("[Superclass] Constructor with args called");
    }
}

public class Subclass extends Superclass {
    private int someOtherValue;

    // No-args constructor in the subclass
    public Subclass() {
        // Does not specify superclass's no-args constructor here
        // But Superclass() constructor will still be invoked implicitly
        System.out.println("[Subclass] No-args constructor called");
    }

    // Constructor with args in the subclass
    public Subclass(int someValue, int someOtherValue) {
        super(someValue); // Superclass's constructor with args will be called
        this.someOtherValue = someOtherValue;
        System.out.println("[Subclass] Constructor with args called");
    }
```

```
    public static void main(String[] args) {
        Subclass s1 = new Subclass();
        Subclass s2 = new Subclass(99, 100);
    }
}

/*
Output:
[Superclass] No-args constructor called
[Subclass] No-args constructor called
[Superclass] Constructor with args called
[Subclass] Constructor with args called
*/
```

Multilevel Inheritance

```
class A {
    void methodA() {
        System.out.println("Method from class A");
    }
}

class B extends A {
    void methodB() {
        System.out.println("Method from class B");
    }
}

class C extends B {
    void methodC() {
        System.out.println("Method from class C");
    }
}

public class Main {
    public static void main(String[] args) {
        C c = new C();
        c.methodA(); // Output: Method from class A
        c.methodB(); // Output: Method from class B
        c.methodC(); // Output: Method from class C
    }
}
```

In this example, class `C` extends class `B`, which in turn extends class `A`. This forms a chain of inheritance, and class `C` can access the methods from both class `A` and class `B`.

Name Clashing - Field/ Method

- You can declare a field in the subclass with the same name as the one in the superclass, thus *`hiding`* it. **This design doesn't make sense, as the subclass should not have the same attribute as superclass, once you decide to inherit the parent class.**
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *`overriding`* it.
- You can write a new ***static* method** in the subclass that has the same signature as the one in the superclass, thus *`hiding`* it.(ParentClss.staticMethodName() & ChildClass.staticMethodName()).

```
1  public abstract class Animal {
2      protected String name;
3      protected int weight;
4      protected int height;
5      public String someProperty;
6
7      public Animal(String name, int weight, int height) {
8          this.name = name;
9          this.weight = weight;
10         this.height = height;
11         this.someProperty = "[Animal] someProperty from Animal...";
12     }
13
14     // abstract method must be implemented by subclasses
15     public abstract void makeSound();
16
17     // default method is overridden optionally by subclasses
18     public void sleep() {
19         System.out.println("[Animal] Zzzzzzzzzzzzzzz");
20     }
21
22     // final method cannot be overridden by subclasses
23     public final void doNothing() {
24         System.out.println("[Animal] doNothing");
25     }
26
27     public static void someStaticMethod() {
28         System.out.println("[Animal] Some static method from Animal...");
29     }
```

```
30
31      // getters and setters for the remaining fields
32  }
33
34  public class Cat extends Animal{
35      public String someProperty;
36
37      public Cat(String name, int weight, int height) {
38          super(name, weight, height);
39          this.someProperty = "[Cat] someProperty ...";
40      }
41
42      @Override
43      public void makeSound() {
44          System.out.println("[Cat] Meowwwwwwww");
45      }
46
47      @Override
48      public void sleep() {
49          System.out.println("[Cat] Meowwwwwwww... Zzzzzzzzzzzz");
50      }
51
52      public String getSomeProperty() {
53          return someProperty;
54      }
55  }
56
57  public class Dog extends Animal {
58      public Dog(String name, int weight, int height) {
59          super(name, weight, height);
60      }
61
62      @Override
63      public void makeSound() {
64          System.out.println("[Dog] Woof Woof !!");
65      }
66
67      @Override
68      public void sleep() {
69          System.out.println("[Dog] Woof woof... Zzzzzzzzzzzz");
70      }
71
72      // Static method with the same name as superclass --> thus hiding
    superclass method
73      public static void someStaticMethod() {
74          System.out.println("[Dog] Some static method from Dog...");
75      }
```

```
}

public class AnimalInheritanceDemo {
    public static void main(String[] args) {
        // Animal animal = new Animal();   // Compilation error
        Cat cat = new Cat("Garfield", 10, 10);
        Dog dog = new Dog("Goofy", 30, 30);

        cat.makeSound();
        cat.sleep();

        dog.makeSound();
        dog.sleep();

        System.out.println("someProperty : " + cat.someProperty);
        System.out.println("getSomeProperty : " + cat.getSomeProperty());

        Animal.someStaticMethod();
        Dog.someStaticMethod();
    }
}

/*
Output:

[Cat] Meowwwwwww
[Cat] Meowwwwwww... Zzzzzzzzzzzz
[Dog] Woof Woof !!
[Dog] Woof woof... Zzzzzzzzzzzz
someProperty : [Cat] someProperty ...
getSomeProperty : [Cat] someProperty ...
[Animal] Some static method from Animal...
[Dog] Some static method from Dog...

*/
```

Private Members in a Superclass

- A subclass does not inherit the `private` attributes of its parent class. However, **if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass**.
- A nested class has access to all the private members of its enclosing class-both fields and methods. Therefore, the public or protected nested class inherited by subclass has indirect access to all of the private members of the superclass.

Rules of Overriding

- **Overriding a superclass's method** from a subclass requires the instance method in the subclass to have **the same method signature (method name & parameters) + return type** as the instance method in the superclass.
- An overriding method can return a subtype of the type returned in the superclass (i.e. being more specific in terms of the return type). This subtype is called a `covariant type`.
- When overriding a method, it is **strongly recommended to use the *@Override* annotation** so that the **compiler can check whether the same method really exists in the superclass**. It will generate an error if not.
- The access specifier for an overriding method in the subclass **can allow more, but not less, access** than the overridden method in the superclass. For example, a protected instance method in the superclass can be made public, but not private, in the subclass. For example, if the
- In short, an overriding method in a subclass may have a **more specific return type** and **more open access level**.

```
1  public class ClassD {
2      public CharSequence getString() { // Interface CharSequence, -> Data Type
   IV
3          return "returnSomething from ClassD";
4      }
5  }
6
7  public class ClassE extends ClassD {
8      @Override
9      public String getString() { // private is NOT allowed in this Override
   method
10         // String is a child class for CharSequence Interface
11         return "returnSomething from ClassE";
12     }
13 }
```

Rules of Hiding

Here's an example to illustrate **Hiding Static Methods**:

```
1  class Parent {
2      public static void display() {
3          System.out.println("Superclass");
```

```
    }
}

class Child extends Parent {

    public static void display() {
        System.out.println("Subclass");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent.display();  // Output: Superclass
        Child.display();   // Output: Subclass

        Parent parent = new Child(); // Chapter 17: Polymorphism
        parent.display();  // Output: Superclass
    }
}
```

Here's an example to illustrate Hiding Fields:

- If a subclass declares a field that has the **same name as a field in the superclass, it hides the superclass's field, even if their types are different.**
- Within the subclass, the hidden field in the superclass can not be referenced directly.

Here's an example to illustrate hiding fields in Java:

- Hiding fields is **strongly discouraged**.

```
class Superclass {
    public int number = 5;
    public int number2 = 5;
}

class Subclass extends Superclass {
    public int number = 10;
}

public class Main {
    public static void main(String[] args) {
        Subclass obj = new Subclass();
        System.out.println(obj.number);  // Output: 10
        System.out.println(((Superclass) obj).number);  // Output: 5
        System.out.println(obj.number2); // 5
    }
```

```
17 }
```

Final Class & Method

In Java, the `final` keyword can be used to restrict the inheritance of a class and to prevent a method from being overridden by its subclasses. Let's explore both use cases:

Final Class

- When a class is declared as `final`, it means that this class cannot be subclassed (extended) by other classes.
- This is useful when you want to create a class that should not have any further specialized subclasses or when you want to prevent any modifications to the behavior of the class through inheritance.
- Any attempt to extend a `final` class will result in a compilation error.

Example of a final class:

```
1 final class FinalClass {
2     // Class members and methods
3 }
4
5 // The following class extension would result in a compilation error
6 // class Subclass extends FinalClass {
7 // ...
8 // }
```

Final Method

- When a method is declared as `final`, it means that this method cannot be overridden by its subclasses.
- This is useful when you want to ensure that the behavior of a specific method in the superclass remains consistent across all subclasses and cannot be changed.
- Once a method is marked as `final`, any attempt to override it in a subclass will result in a compilation error.

Example of a final method:

```
1 class Parent {
2     final void finalMethod() {
```

```
        // Method implementation
    }
}

class Child extends Parent {
    // Attempt to override the final method would result in a compilation error
    // @Override
    // void finalMethod() { }
}
```

In the example, the `FinalClass` is declared as `final`, so it cannot be subclassed. Any attempt to create a subclass of `FinalClass` (like `Subclass`) will result in a compilation error.

Similarly, the `finalMethod()` in the `Parent` class is marked as `final`, so it cannot be overridden by its subclasses (like `Child`). Attempting to override the `finalMethod()` in the `Child` class will also result in a compilation error.

Using the `final` keyword judiciously can help you enforce design decisions and ensure the consistency and immutability of certain aspects of your classes and methods. It allows you to communicate your intent clearly to other developers and helps in maintaining the integrity of your code.

Abstract Classes & Methods

- An *abstract class* is a class that is declared ***abstract***.
- An abstract class **may or may not have abstract methods**.
- When a class contains **one or more abstract methods, it must be declared as an abstract class** using the `abstract` keyword.
- An **abstract class CANNOT** be **instantiated**, but they can be **subclassed**.
- An **abstract method** is a method declared in an abstract class that **has no implementation** (no method body). It is intended to be overridden (implemented) by concrete subclasses.
- It serves as a blueprint for other classes and is meant to be subclassed by concrete (non-abstract) classes. Abstract classes provide a way to **define common behavior and characteristics that can be shared among multiple subclasses**.

```
// A class with abstract methods must be declared an abstract class.
public abstract class Animal {
    abstract void doSomething(int x, int y);
```

```java
}

public class Dog extends Animal {
    @Override
    void doSomething(int x, int y) {
        System.out.printf("x is %s, y is %s\n", x, y);
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.doSomething(1, 2);
    }
}

// Output: x is 1, y is 2
```

To conclude, the key points about abstract classes:

1. **Cannot be Instantiated**: Abstract classes cannot be instantiated using the `new` keyword, meaning you cannot create objects directly from an abstract class.
2. **May Contain Abstract Methods**: Abstract classes can have abstract methods (methods without a body) that must be implemented by concrete subclasses.
3. **May Contain Concrete Methods**: Abstract classes can also have concrete (implemented) methods that provide default behavior that can be inherited by the subclasses.
4. **May Contain Instance Variables**: Abstract classes can have instance variables and constructors.
5. **Subclassing (Inheritance)**: Concrete subclasses of an abstract class must implement all the abstract methods inherited from the abstract superclass. Once all abstract methods are implemented, the subclass becomes concrete and can be instantiated.

Shape Example

```java
// Abstract class representing a basic shape
abstract class Shape {
    // Abstract method to calculate the area
    abstract double area();

    // Concrete method providing default behavior
    void display() {
        System.out.println("This is a shape.");
    }
}
```

```
11
12 // Concrete subclass Circle
13 class Circle extends Shape {
14     private double radius;
15
16     Circle(double radius) {
17         this.radius = radius;
18     }
19
20     @Override
21     double area() {
22         return Math.PI * radius * radius;
23     }
24 }
25
26 // Concrete subclass Rectangle
27 class Rectangle extends Shape {
28     private double length;
29     private double width;
30
31     Rectangle(double length, double width) {
32         this.length = length;
33         this.width = width;
34     }
35
36     @Override
37     double area() {
38         return length * width;
39     }
40 }
41
42 public class Main {
43     public static void main(String[] args) {
44         Circle circle = new Circle(5.0);
45         System.out.println("Circle Area: " + circle.area());
46         circle.display();
47
48         Rectangle rectangle = new Rectangle(4.0, 6.0);
49         System.out.println("Rectangle Area: " + rectangle.area());
50         rectangle.display();
51     }
52 }
53
```

Benefits of Inheritance

- **Reusability** - Using inheritance allows you to derive new classes from existing classes, so that you can reuse fields and methods of the existing classes without having to write them again.
- **Polymorphism** - Objects of a child class can be passed around and treated as if they were objects of the parent class (Covered in other chapters)

Questions

- What is the purpose of inheritance?
- What are the benefits of inheritance?
- What can be done in a subclass?
- What makes a class an abstract class?
- What is vertical constructor chaining? What is the associated keyword being used in the constructor?
- When do we use the *protected, final* and *super* keywords?