Chapter3. Inheritance

Introduction

Inheritance is a mechanism that facilitates **code reuse** and the **management of its evolution**. Through inheritance, objects of a class have access to the data and methods of the **parent class** and can extend them. Subclasses can **redefine** (see polymorphism section) the inherited variables and methods. For variables, they simply need to be redeclared with the same name but a different type. Methods are **redefined** (see polymorphism section) with the same name, the same types, and the same number of arguments; otherwise, it is considered **overloading** (see polymorphism section). Successive inheritance of classes allows for the definition of a class hierarchy consisting **of super classes** and **subclasses**. A class that inherits from another is a subclass, and the one it inherits from is a superclass. A class can have multiple subclasses.

Principle of Inheritance

The main idea of inheritance is to organize classes hierarchically. The inheritance relationship is unidirectional, and if class B inherits from class A, we say that B is a subclass of A. This concept of a subclass means that class B is a special case of class A, and therefore, objects instantiating class B also instantiate class A.

Let's take the example of the classes Square, Rectangle, and Circle. Figure 1 provides a hierarchical organization of these classes such that Square inherits from Rectangle, which in turn inherits, along with Circle, from a class Shape.

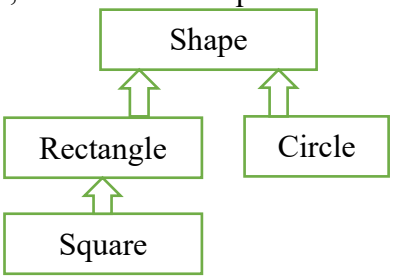

Figure 1: Example of inheritance relationships

1. Subclasses and Inheritance

1.1. Definitions:

Defining a new class from another ⇒ Derivation

- ✓ New class: derived class
- ✓ Original class: superclass or base class.

- A class **B** that inherits from a class **A** gains the **attributes** and **methods** of class **A** without having to redefine them.
- **B** is a subclass of **A**; alternatively, **A** is the superclass of **B** or the base class.
- It is also said that **B** is a derived class of class **A**, or that class **B** extends class **A**.
- A class can have only one superclass; multiple inheritance is not allowed in Java. However, a class can have multiple subclasses (as shown in Figure 2).
- The **Object** class is the parent class of all classes in Java. All variables and methods contained in the **Object** class are accessible from any class, as through successive inheritance, all classes inherit from the **Object** class.

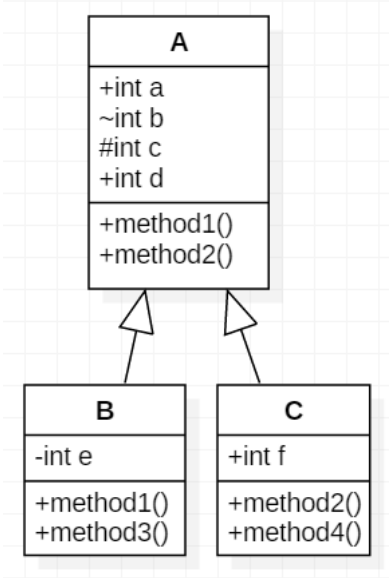

Figure 2: The principle of inheritance: class **B,C** inherits from class **A**. The # symbol indicates a protected attribute. The + symbol indicates a public attribute. The ~ symbol indicates a friendly attribute(no modifier). The - symbol indicates a private attribute.

- Any instance of B is also an instance of A.
- Any instance of B contains all the members of A plus the members defined in B.
- At the class level, all static members of A are also static members of B (and B additionally has its own static members defined in B).
- Methods from A can be redefined in B.

1.2 Key Concepts of Inheritance

- ✓ **Base Class (Parent Class):** This is the class whose properties and methods are inherited by other classes. It is the "generic" class that provides common attributes and behaviors.
- ✓ **Derived Class (Child Class):** This is the class that inherits from the base class. It can use the attributes and methods of the parent class and can also define its own unique features or override inherited behaviors.
- ✓ **Overriding**: The child class can modify or redefine methods inherited from the parent class. This is known as method overriding. This allows the child class to provide a specific implementation of a method that is already defined in the parent class.
- ✓ **Accessing Parent Class Methods:** The derived class can access methods from the parent class using the **super()** function in many programming languages (such as java). This is particularly useful when the child class overrides a method but still wants to use the functionality from the parent class. The concept of inheritance refers to a relationship between different classes that allows a new class to be defined based on existing classes.

2. Single and Multilevel Inheritance

a. In single inheritance: a child class inherits from only one parent class. This is the simplest form of inheritance. It is referred to as single inheritance when a child class has only one parent class.

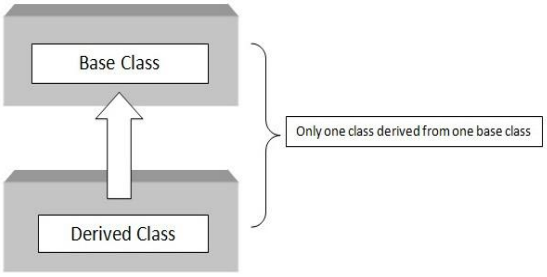

Figure 2: single inheritance

As shown in the figure 2, single inheritance only one class can be derived from the base class

b.multilevel Inheritance: is a type of inheritance in object-oriented programming where a class derives from another class, which in turn is derived from another class, creating a chain of inheritance.

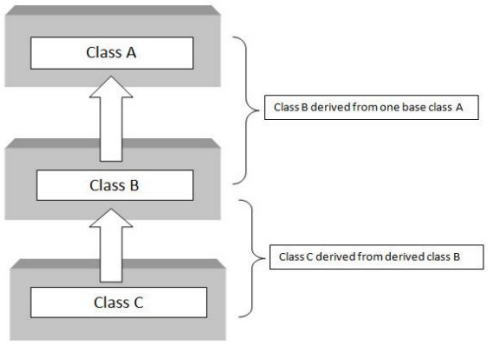

Figure 4: Multilevel Inheritance

As shown in above block diagram shown by figure 4, class C has class B and class A as parent classes

c. Multiple Inheritance: In multiple inheritance, a child class inherits from more than one parent class. This allows the child class to inherit features from multiple sources. Figure 5 illustrates the multiple inheritance

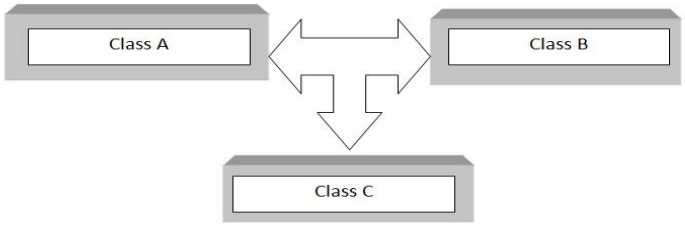

Figure 5: Multiple Inheritance

3 Hierarchical Inheritance: is a type of inheritance where multiple subclasses inherit properties and behaviors (methods) from a single base class. In other words, one parent class is extended by multiple child classes. This type of inheritance allows for the reuse of code from the base class by different subclasses.

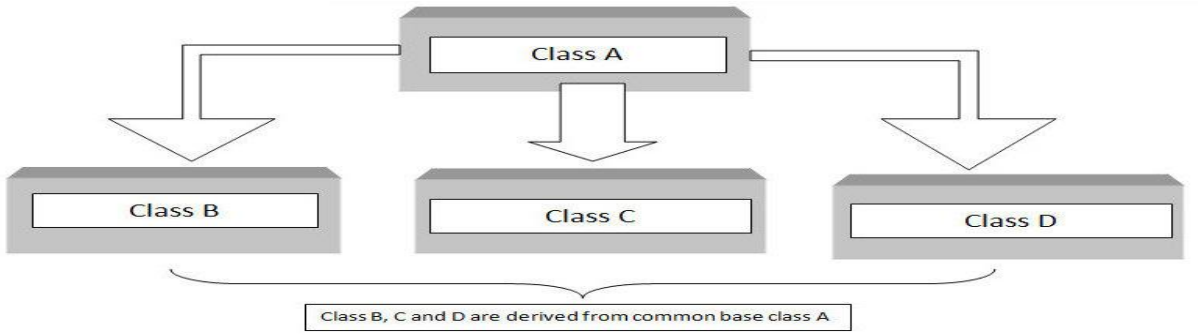

Figure 6: Hierarchical Inheritance

As shown in figure 6 hierarchical inheritance all the derived classes have common base class

4. Polymorphism

Polymorphism refers to an object's ability to be an instance of multiple classes. The term describes the capacity of a method to exhibit different behaviors based on its context, such as the types of its parameters or the type of the current object.

For example, consider creating a function to compute the maximum of two numbers. It would be convenient to use the same function name regardless of the parameter types:

```
int Max(int a, int b) { ... }
float Max(float a, float b) { ... }
double Max(double a, double b) { ... }

k = Max(u, v);
```

The compiler identifies the types of variables **u** and **v** and invokes the corresponding **Max** function. While each **Max** function handles different data types, they conceptually perform the same calculation. This is an example of **polymorphism**.

Polymorphism extends to inheritance. For instance, in a simulation game displaying various vehicle types, it would be ideal to have a single **display()** function. Each object (e.g., a car, plane, or bicycle) uses the same function name but behaves differently. A **Car** object might display a Peugeot 205, while a **Plane** object displays an Airbus A320. Creating unique functions like **displayCar()** or **displayPlane()** would be cumbersome. With polymorphism in inheritance, a single function name can exist in a hierarchy but behave differently depending on the instantiated object.

5. Inheritance and Polymorphism in Java
a. Single Inheritance (extends)
a.1. Syntax:

class Child extends Parent { ... }

The extends keyword specifies that a class inherits from another. If no parent class is specified, the compiler assumes **Object** as the parent.

Example 1:

```
// Base class
class Graphic {
private int x, y;
Graphic(int x, int y) { this.x = x; this.y = y; }
void display() { System.out.println("The object's center is at: " + x + " and " + y); }
double surface() { return 0; }
}

// Derived class 1
class Circle extends Graphic {
private double radius = 1;
void display() { System.out.println("This is a circle with radius " + radius);
super.display();  }
double surface() { return (radius * radius * 3.14); }
}

// Derived class 2
class Rectangle extends Graphic {
private int width, length;
Rectangle(int x, int y, int width, int length) {
super(x, y);
this.width = width;
this.length = length;}
double surface() { return (width * length); }
}

// Derived class 3
class Square extends Graphic {
private double side;
Square(double side, int x, int y) {super(x, y);this.side = side;}
double surface() { return (side * side); }
}
```

Interpretation:
The Circle class inherits attributes and methods from Graphic, overrides surface and display, and adds the radius attribute.
The Rectangle class overrides the surface method and adds the width and length attributes.
The Square class adds the side attribute and overrides the surface method.

Notes:

The concept of inheritance allows the derived class to:

- ✓ A **derived class** inherits attributes and methods from the **base class**.
- ✓ It can add its own attributes and methods.
- ✓ Methods inherited from the parent class can be overridden or overloaded.
- ✓ An object can be viewed as an instance of its class and all its ancestor classes.
- ✓ All Java classes implicitly extend the **Objec**t class.
- ✓ Java allows direct inheritance from only one base class (no multiple inheritance). Multiple inheritance can be simulated using interfaces.
- ✓ To invoke a method from a parent class, simply prefix the method with **super**. To call the constructor of the parent class, write **super(parameters)** with the appropriate parameters.
- ✓ The link between a child class and a parent class is managed by the language: an evolution in the management rules of the parent class leads to an automatic modification of the child class as soon as the latter is recompiled.
- ✓ In **Java**, it is **mandatory** in a constructor of a child class to **explicitly** or **implicitly** call the constructor of the **parent class**.

b. Encapsulation in Inheritance
i. Member Protection (protected)
i.1. Access to Inherited Properties:
Variables and methods defined as **public** remain accessible through inheritance.
Variables marked as **private** are inherited but cannot be **directly accessed** (only through inherited methods). **The protected** modifier allows access to inherited members within derived classes but not outside of them.

Example 2: Derived and base classes in the same package:

```
package Shapes;
class Graphic {
  private int x, y;
  public String color;
  void display() { System.out.println("The object's center is at (" + x + ", " + y + ")"); }
  double area() { return 0; }
}

package Shapes;
class Circle extends Graphic {
  double radius;
  void setCenter(int x1, int y1, double r) {   // x = x1; // Invalid because x is private
    radius = r;
  }
  public double surface () { return (radius * radius * 3.14); }
  public void displayCircle() {  display(); // Valid, display is friendly (no modifier)
    System.out.println("Radius: " + radius + " Color: " + color); }
}
```

i.1.1. Interpretation:

In the first case, the derived class and the base class are located in the **same package**. Therefore, the Cercle class was able to access the public members (attributes and methods) of the base class (e.g., the couleur attribute). Similarly, it was able to access the "friendly" members, which are those without a visibility modifier (e.g., the **display method**). However, it could not access the private members (x and y).

Example 3: The derived class and the base class are not in the **same package**

```
package Forms;
class Graphic {
   private int x, y;
   public String color;
   protected String name;
   void display() {   System.out.println("The center of the object = (" + x + ", " + y + ")");  }
   double surface() {  return 0;   }
} // end of the Graphic class

package FormsCirc;
class Circle extends Graphic {
   double radius;
   void modify_centre(int x1, int y1, double r) {  x = x1; y = y1;
      // ERROR: x and y are declared private in the base class
      radius = r;
   }
   public double surface() {   return (radius * 2 * 3.14);   }
   public void display() {    display(); // ERROR: the visibility modifier of this member is
friendly
      Centre = 15;  // CORRECT: the visibility modifier of this member is protected
      System.out.println("The radius = " + radius + " The color = " + color);  }
}
```

i.1.2. Interpretation:

In the second case, the derived class and the base class are not in the **same package**. Therefore, the Circle class was able to access the public members (attributes and methods) of the base class (e.g., the color attribute). However, it could not access the "friendly" members, which are those without a visibility modifier (e.g., the **display** method), or the private members (x and y).

The protected attribute name of type String is accessible by the derived class Circle, as it follows the rules of encapsulation. (See the chapter on variable access for details on encapsulation).

Key Points:

We have already seen that there are three types of visibility modifiers:
public (**public**), private (**private**), and package or **friendly** (**no modifier**).
Additionally, there is a fourth type of access modifier called protected (keyword **protected**).

ii. Class Constructors (this() and super())

The reserved keyword **super** allows invoking a field (attribute or method) of the **superclass** of the current object being worked on. This is particularly useful when such a field is otherwise hidden by a field of the same name in the current class (either because the field in question is a redefined method or a masked attribute).

- ✓ When a field of the current object is invoked within a method, the name of the object is usually not explicitly specified. However, this can be done using the keyword **this**.When **super** is used, it replaces the implicit **this**.
- ✓ Using **super** effectively replaces the class of the current object with its superclass.
- ✓ It is forbidden to "climb multiple levels" by writing something like **super.super.etc.**
- ✓ The syntax **super(parameters)** is also used in the first line of a constructor to invoke the constructor of the superclass with the corresponding parameters.
- ✓

For more about the **this** keyword, see the section on **accessing the this instance in Chapter 2: "Encapsulation."**

ii.1. Construction and Initialization of Derived Objects

In **Java**, the constructor of a derived class must handle the complete construction of the object. If a constructor in a derived class calls a constructor in the **base class**, this must be the **very first statement** in the constructor. The base class constructor is invoked using the keyword **super**.

```
class Graphic {
   private int x, y;
   public Graphic(int x1, int y1) {
      this.x = x1;
      this.y = y1;
   }
   // other methods
}

class Circle extends Graphic {
   private double radius;
   Cercle(int a, int b, double r) {
      super(a, b);  // calls the constructor of the base class
      radius = r;   }
   // other methods
}
```

Notes:

In general, both a **base class** and a **derived class** each have at least one constructor. The constructor of the derived class completes the construction of the derived object by first invoking the constructor of the **base class**.
However, there are special cases, which are:

Case 1: The **base class** has no constructor In this case, if the derived class wants to define its own constructor, it can optionally call the clause "**super()**" in the first line of the constructor to invoke the **default constructor** of the **base class**.

Case 2: The **derived class** does not have a constructor. In this case, the default constructor of the derived class must initialize the attributes of the derived class and call:

- The default constructor of the base class if it does not have any constructor.
- The constructor without arguments if it has at least one constructor.
- If there is no constructor without arguments and there are other constructors with arguments, the compiler will generate errors.

Example 5 (Case 1)

```
class A {
  // no constructor
}
class B extends A {
  public B(….) {
    super(); // calling default constructor of A
    ........
  }
}
B b = new B(…); // Calling implicit default constructor of A
```

Example 6 (Case 2)

```
class A {
  public A() {….} // constructor1
  public A(int n) {…..} // constructor2
}


class B extends A {
  // …..no constructor
}
B b = new B(); // Calling constructor1 of A
```

Example 7 (Case 2)

```
class A {
  public A(int n) {…..} // constructor1
}
class B extends A {
  // …..no constructor
}
B b = new B();
/* Compilation error occurs because A does not have a constructor
without arguments and has another constructor with arguments. */
```

Example 8 (Case 1 and 2)

```
class A {
  // no constructors
}
class B extends A {
  // no constructors
}
B b = new B();
// Calls the default constructor of B which in turn calls the default constructor of A.
```

iii. The 'Object' class

Every **class** implicitly derives from the **Object** class, which is a member of the **java.lang** package. In other words, the following two definitions are equivalent:

class A { ... }
class A extends Object { ... }

Object is the base class of **Java**. All objects that we create subsequently inherit the methods of the **Object** class. Hence, every class is a subtype of **Object**. As a result, any method declared with an **Object** type parameter can accept an instance of any class (but not a value of a primitive type). All array types are also subtypes of **Object** (as well as the **Cloneable** and **java.io.Serializable** interfaces). Therefore, any array can be assigned to a variable of type **Object**. Here are some of the methods defined by the **Object class**:

```
package java.lang;

public class Object {
  public String toString() { ... }
  public final Class getClass() { ... }
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
  protected Object clone() throws CloneNotSupportedException { ... }
  ...
}
```

The **toString** method returns a string representation of the object. This method is called when an object is passed as an argument to the **print** or **println** methods or in a string concatenation expression. It is often useful to **override** this method (as shown in the polymorphism section) to obtain a more meaningful representation.

A more readable string representation. For example, the **Point** class could redefine it as follows:

```
class Point {
  // ...
  public String toString() {
    return "(" + x + ", " + y + ")"; }
}
```

The **getClass method** returns a reference to an instance of the Class class that represents the object's class. This instance allows retrieving various information about this class (its name, its members, its superclass, etc.), for example:

```
void printClassName() {
  System.out.println("The class of " + this + " is " + this.getClass().getName());}
```

The **equals** method is used to test the equality of two objects. The method defined in the **Object** class is as discriminative as possible, meaning that **x.equals(y)** returns **true** if and only if x and y refer to the same object, i.e., if x == y. For example, if the Point class does not override equals, the value of the expression: **new Point().equals(new Point())**

Which compares two different instances, returns **false**. Therefore, it is useful to **override equals**. In the case of the **Point** class, a point is equal to an object **o** if **o** is a point and if the corresponding fields are equal. The expression **o** instanceof **Point**, of type boolean, is used to test if the type of the object designated by **o** is a subtype of **Point**:

```
package geometrie;

class Point {
  // ...
  public boolean equals(Object o) {
    return o instanceof Point &&
          this.x == ((Point)o).x &&
          this.y == ((Point)o).y;  }
}
```

It would be tempting to define a method that only compares instances of **Point** between themselves:

```
class Point {
  // ...
  boolean equals(Point p) { ... }
}
```

However, this would not be an **override** of the **equals** method of the **Object** class, because it does not have the same signature. In the following situation, the equality of the two points would thus be tested using **the equals** method from **Object** and not the one from **Point**:

```
Object o = new Point(), p = new Point(1, 2);
boolean b = o.equals(p);
```

The **clone method** allows **duplicating** an object. This important operation will be examined later. The purpose of this method is to **create** and **return** an exact copy of the considered **object**. We want **x.clone() != x and x.clone().getClass() == x.getClass().**

iv. Implicit and Explicit Type Casting (Cast)

Type **casting** is necessary when there is a risk of data loss, such as when assigning a **long integer** (64 bits) to an **int integer (32 bits)**, or a **double floating-point** number to a **float**. In these cases, the type is forced, signaling to the compiler that we are aware of the potential risk of losing information. It is also necessary when we want to take the integer part of a floating-point number. The following assignments are **implicitly** allowed (without writing the casting instruction: that is, without using parentheses, as seen in examples in this section) between integers and/or real numbers; for instance, we can assign a **byte** to a **short**, or a **float** to a **double**. Here the **implicit** conversions allowed in Java.

byte---→ short---→ int---→long---→float---→ double

Examples iv.1:

When the two operands of an operator are not of the same type, the one with the lower type is converted to the type of the other. For the variable l4 below, i3 (which is 10) is multiplied by the real value 2.5f, which is converted into a long integer. For l5, i3 is converted into a floating-point number, and the result in floating-point is converted into a long integer. The results are different.

```
int i3 = 10;
long l4 = i3 * (long) 2.5f; // result: 20
long l5 = (long) (i3 * 2.5f); // result: 25
```

Note:

In arithmetic expressions, integers of type **byte** or **short** are considered as **int** (automatically converted to **int**). For example, if b1 is a byte, then b1 + 1 is considered an int. The assignment b1 = b1 + 1 is illegal without a cast (from int to byte), whereas b1 = 5 is accepted.

Example iv.2

```
// Cast.java forced casts (explicit conversions)
class Cast {
   public static void main(String[] args) {
      byte b1 = 15;
      int i1 = 100;
      long l1 = 2000;
      float f1 = 2.5f; // f for float; by default type is double
      double d1 = 2.5;  // or 2.5d

      // b1 = b1 + 1;    // cast required: b1 + 1 is automatically converted to int
      b1 = (byte) (b1 + 1);  // OK with cast
      // byte b2 = i1;       // cast required: from int -> byte
      byte b2 = (byte) i1;    // OK with cast
      System.out.println("b2 : " + b2);
```

```
        long l2 = i1;              // OK without cast: from int -> long
        System.out.println("l2 : " + l2);

        // int i2 = l1;             // cast required: from long -> int
        int i2 = (int) l1;         // OK with cast
        System.out.println("i2 : " + i2);

        float f2 = l1;             // OK without cast: from long -> float
        System.out.println("f2 : " + f2);

        // long l3 = f1;    // cast required: from float -> long
        long l3 = (long) f1;       // OK with cast
        System.out.println("l3 : " + l3);

        double d2 = f1;            // OK without cast: from float -> double
        System.out.println("d2 : " + d2);

        // float f3 = d1;    // cast required: from double -> float
        float f3 = (float) d1;    // OK with cast
        System.out.println("f3 : " + f3);

        int i3 = 10;
        long l4 = i3 * (long) 2.5f;        // l4 is 20
        long l5 = (long) (i3 * 2.5f);      // l5 is 25
        System.out.println("\nl4 : " + l4 + "\nl5 : " + l5);

        // In an arithmetic expression, byte, short are considered as int
    }
}

byte b3 = 10;
//short s4 = 2*b3;               // error: b3 is considered an int
short s4 = (short) (2 * b3);   // cast required: from int -> short
System.out.println("s4 : " + s4);

int i4 = 10;
long l6 = 1000;
//int i5 = i4 + l6;    // error: the result is of type long
int i5 = (int) (i4 + l6);     // cast required: from long -> int

System.out.println("i5 : " + i5);
} // main
} // class Cast
```

v. Limitation of Inheritance (final)

v.1. The final Modifier

Case 1: The **final** modifier placed before a class **prevents** its inheritance.

Example v.1:

```
final class A {
   // Methods and attributes
}
class B extends A              // Error because class A is declared final
{
   //....
}
```

Case 2: The **final** modifier placed before a **method prevents** its **overriding** (see the Polymorphism course).

Example v.2:

```
class A {
   final public void f(int x) {…}
   //…Other methods and attributes
}

class B {
   //Other methods and attributes
   public void f(int x) {…} // Error because method f is declared final. No overriding allowed.
}
```

Case 3: For a **field**, **final** indicates that it is a **constant**, instance if it does not simultaneously have the static modifier, and class-level if the field is **final static**. A final field can only be **assigned once**.

Example v.3:

```
class C1 {
   final int i = 5; // i is a constant
}
```

Note:

Classes and methods can be declared **final** for security reasons to ensure that their behavior cannot be modified by a subclass.

Advice on Inheritance
When creating a "parent" class, you should consider the following points:

- **Access control for instance variables:** Instance variables are often private, but you should carefully decide between **protected** and **private** access modifiers. The goal is to ensure that the class maintains encapsulation while still allowing access by subclasses when needed.

- **Preventing method overriding:** To prevent a method from being overridden (overloading), declare it with the **final** modifier. When creating a subclass, for each inherited method that is not final, you should consider the following cases:

✓ The inherited method is suitable for the subclass: Do not override it.
✓ The inherited method is partially suitable due to the specialization added by the subclass: You should override or overload the method. Typically, an override will first call the inherited method (via super) to ensure the evolution of the code.
✓ The inherited method is not suitable: You need to override or overload the method without calling the inherited method during the override.

For more details on overriding and overloading, refer to the polymorphism course. Polymorphism

c. Polymorphism

Inheritance allows one class to interact with another by requesting a service that can be executed in many different ways, depending on whether the service, always named the same, is redefined in various subclasses, all inheriting from the message receiver. This is the foundation of polymorphism. It allows the first class to treat all of its interacting classes as one, without altering its behavior when a new subclass is added or when a subclass modifies its response to the messages from the first class. This leads to simplicity in design, reduced code repetition, and seamless evolution: **polymorphism** is at the core of **Object-Oriented Programming**.

Polymorphism in Java is the ability of subclasses to have their own behavior while also sharing some functionality with other subclasses.

i. Method Overloading
Method overloading within the same class involves defining methods with the same name but different signatures (by changing the **number of parameters**, **parameter types**, **both**, or the **return type**). A derived class can also overload a method from a parent class. Of course, the new methods will only be usable by the derived class or its descendants, but not by the parent classes.

Example 1:
```
public class Account {
   public double balance;
   public void calculate() { balance += 300; }
   public void display() { System.out.println("balance = :" + balance); }
}
```

```
public class NormalAccount extends Account {
   public void calculate(double rate) { // Overloading the 'calculate' method by adding the 'rate' parameter
      super.calculate(); // Calling the overridden method
       balance *= rate;   }

   public void display() {
      System.out.println("balance = :" + balance); }
}

public class Test {
   public static void main(String[] args) {
      Account acc1 = new Account(); // Compte c1
      NormalAccount acc2 = new NormalAccount(); // Compte_normal c2

      acc1.calculate(); // c1.calcul()
      acc2.calculate(1.07); // c2.calcul(1.07)

      acc1.display(); // c1.afficher()
      acc2.display(); // c2.afficher()
   }
}
public class Test {
   public static void main(String[] args) {
      Compte c1 = new Compte();
      Compte_normal c2 = new Compte_normal();

      c1.calcul();
      c2.calcul(1.07);

      c1.afficher();
      c2.afficher();
   }
}
```

ii. Method Overriding

You can **override** a method from the **base class** in a **derived class**, using the same signature as the base class method. The access modifier of the method in a derived class can be the same as that of the base class, or less restrictive, but it cannot be more **restrictive**.

- ✓ Any method declared as public in the base class must also be declared as public in the derived class.
- ✓ You cannot omit the access modifier in the derived class or specify it as private or protected.
- ✓ A package-private method can be overridden as a public method.
- ✓ A public method cannot become private in a subclass

Imperative:
We must respect the method signature when overriding it!

- ✓ Parameters: (number, type, order)
- ✓ Return type
- ✓ Accessibility modifier: You can broaden its access.
- ✓ The method from the superclass is accessible, but it must be prefixed with the keyword **super**.

Example 2

```
public class Account {
   public double balance;
   public void calculate() {  balance += 300; }
   public void display() { System.out.println("balance = :" + balance);}

public class NormalAccount extends Account {
   public void calculate() {
      super.calculate(); // Call to the overridden method
      balance *= 1.07; // Update the balance
   }

   public void display() {
      System.out.println("balance = :" + balance); }
}

public class Test {
   public static void main(String[] args) {
      // TODO Auto-generated method stub
      Account c1 = new Account();
      NormalAccount c2 = new NormalAccount();
      c1.calculate();
      c2.calculate();
      c1.display();
      c2.display();  }
}
```

ii.1. Constraints on Method Overriding

When overriding a method in a subclass, certain rules and constraints must be followed to ensure that the method is correctly redefined:
Method Signature: The method signature in the subclass must be exactly the same as the method in the superclass. This means that:
The method name must be identical.
The method parameters (number, type, and order) must be the same.
Return Type: The return type of the method in the subclass should be the same or a subtype (covariant return type) of the return type in the superclass method.
Access Modifier: The access level of the overriding method cannot be more restrictive than that of the overridden method.

If the superclass method is public, the subclass method must also be public or less restrictive.

A protected or private method in the superclass cannot be overridden as public in the subclass.
Exceptions: The overriding method cannot throw more checked exceptions than the method it overrides. It can only throw the same exceptions or fewer exceptions.
Super Keyword: The overridden method from the superclass can still be accessed in the subclass using the super keyword.
Final Methods: If a method in the superclass is declared as final, it cannot be overridden in the subclass.
Static Methods: Static methods **cannot be overridden**. If a method is static in the superclass, the subclass can declare a method with the same name, but it is not technically overriding—this is called "**method hiding**."

These rules ensure that method overriding respects the object-oriented principles of inheritance and polymorphism while maintaining the integrity of the class structure.

Access rights: The overriding method should not reduce the access rights of a method.
the overriding method should not decrease the access level of the inherited method. If the original method is public, the overridden method must also be public. You cannot make an overridden public method private or protected.

Notes:

The redeclaration must be strictly identical to that of the superclass: **same name, same parameters, and same return type.**
Thanks to the keyword **super**, the overridden method in the subclass can reuse code written in the superclass method, which is otherwise no longer visible.
If the method signature changes, it is no longer an override but an overload.

Example 3: Overriding.

```
// Super-class
class Graphics {
   private double x, y;
   public String color;
   public void display() {
   System.out.println("« The center of the object = (" + x + " , " + y + " ) ");  }

   double surface() { return 0; }
} // End of Graphics class

// Derived class
class Circle extends Graphics {
   double radius;
   public double surface() { return (radius * 2 * 3.14); // Overriding "surface"    }

   public void display() { // Overriding "display" method
      super.display(); // Calling display from the super-class
      System.out.println("The radius = " + radius + " The color = " + color); }
}

// Main class to test
public class Main {
```

```
    public static void main(String[] args) {
        Circle c = new Circle();
        c.radius = 5;
        c.color = "Red";
        c.display(); // It will call the overridden display method
        System.out.println("Area of the circle = " + c.surface()); // Calls the overridden area method }
}
```

This code defines a superclass **Graphics** and a derived class **Circle**. The **Circle** class overrides both the **surface** and **display** methods from the **Graphics** class. In the main method, we create an instance of the **Circle** class, set its properties, and display its attributes and surface.

Example 4: Method Overloading

```
class A {
    public void f(int n) { }
}

class B extends A {
    public void f(float n) { }
}

public class Main {
    public static void main(String[] args) {
        A a;
        B b;
        int n;
        float x;

        a.f(n);  // Calls f(int) from A
        // a.f(x);  // Error, no f(float) in A

        b.f(n);  // Calls f(int) from A
        b.f(x);  // Calls f(float) from B
    }
}
```

Example 5: Using Both Overloading and Overriding

```
class A {
    public void f(int n) {}
    public void f(float n) {}
}

class B extends A {
    public void f(int n) {}  // Overriding f(int) from A
    public void f(double n) {}  // Overloading f from A and B
}

public class Main {
    public static void main(String[] args) {
```

```
        A a;
        B b;
        int n;
        float x;
        double y;

        a.f(n);  // Calls f(int) from A
        a.f(x);  // Calls f(float) from A
        // a.f(y);  // Error, no f(double) in A

        b.f(n);  // Calls f(int) from B, because f(int) is overridden in B
        b.f(x);  // Calls f(float) from A
        b.f(y);  // Calls f(double) from B    }
}
```

Remark:
If a method in a derived class has the same signature as a method in the base class:
Both methods must have the same return type.
The access rights of the method in the derived class must not be more restrictive than in the base class.
A static method cannot be overridden.

Example 6: Polymorphism
For polymorphism, we will reuse the classes from Example 2 and add an array to store Account objects.

```
public class test {
   public static void main(String[] args) {
      // TODO Auto-generated method stub
      Account c1 = new Account();
      NormalAccount c2 = new NormalAccount();
      Account c3 = new Account();
      NormalAccount c4 = new NormalAccount();
      Account t[] = new Account[4];

      // A table that will store objects of different types
      // where each object can have behavior different from others
      // while executing their corresponding methods
      t[0] = c1; t[1] = c1; t[2] = c1; t[3] = c1;
      for (int i = 0; i < 4; i++) {
         t[i].calculate();
         t[i].display();
      }
   }
}// Each object executes its own method
```

d. Abstract Classes (Usage and Importance)
The concept of an abstract class lies between that of a class and that of an interface (refer to the next section). It is a class that cannot be directly instantiated because some of its methods are not implemented. An abstract class can therefore contain variables, implemented methods, and method signatures to be implemented. An abstract class can partially or fully implement interfaces and can inherit from a class or an abstract class.

The **abstract** keyword is used before the class keyword to declare an **abstract class**, as well as for the declaration of **method** signatures to be implemented.

a. Interest

- ✓ Place in an abstract class all the functionalities we want to be available when creating descendants.
- ✓ An abstract class serves as the base class for inheritance.

b. What exactly is it?

- ✓ An abstract class does not allow object instantiation.
- ✓ An abstract class can contain methods and fields (which can be inherited), as well as one or more abstract methods.
- ✓ An abstract method is a method whose signature and return type are provided in the class, but nothing else (no method body, i.e., definition).

c. Syntax
abstract class A { // etc.}

d. Particularities

- ✓ A class having an **abstract method** is, by **default, abstract**. So no need to declare it "abstract" at this stage.
- ✓ Abstract classes must be declared "**public,**" otherwise inheritance is not possible!
- ✓ A class derived from an abstract class is not required to override all abstract methods. It may not define any of them. If that's the case, it remains abstract.
- ✓ A class derived from a non-abstract class can be declared abstract.

Remarks:
It is said that the class or method is abstract.

- ✓ An abstract method only has its signature (its prototype), that is to say its return type followed by its name, followed by the list of parameters in parentheses, followed by a semicolon.
- ✓ An abstract method cannot be declared static, private, or final.
- ✓ As soon as a class contains an abstract method, it must also be declared abstract.
- ✓ An abstract class cannot be instantiated. It must be extended and all the abstract methods it contains must be defined to be used.

Here is a small example of an abstract class:

```
public abstract class Person {
   public abstract String getName();
```

```
   public String toString() {  return "My name is " + getNom();   } }
```

Example 7: Abstract Class

Imagine we want to assign two variables, origin_x and origin_y, to any object representing a shape. Since an interface cannot contain variables, we need to turn Form into an abstract class as follows:

```
public abstract class Form {
   private double origin_x;
   private double origin_y;

   public Form() {
      this.origin_x = 0;
      this.origin_y = 0;
   }

   public double getOriginX() { return this.origin_x; }
   public double getOriginY() { return this.origin_y; }
   public void setOriginX(double x) { this.origin_x = x; }
   public void setOriginY(double y) { this.origin_y = y; }

   public abstract double surface();
   public abstract void display();
}
```

Moreover, we need to restore the inheritance of the Rectangle and Circle classes from Form:

```
public class Rectangle extends Form {}
public class Circle extends Form {}
```

e.Interfaces (Usage and Importance)

An interface is a concept in object-oriented programming that is independent of inheritance. It provides a way to define a contract for what methods a class must implement, without specifying how those methods are implemented. Unlike a class, an interface cannot contain any implementation code; it only contains method signatures. An interface is typically used to represent a common behavior that can be shared across multiple classes. The interface is simply a special kind of abstract class.

Definition of Interfaces: An **interface** defines a set of methods that a class must implement. These methods in the interface do not have any body (no implementation), only the signature (method name, return type, parameters).

```
public interface Animal {
   void makeSound();
   void eat();
}
```

In this example, the Animal **interface** specifies that any class implementing it must provide implementations for makeSound() and eat() methods.

Implementing an Interface: A class can implement an interface by providing concrete implementations for all the methods declared in that interface. A class can implement **one or more interfaces.**

```
public class Dog implements Animal {
   public void makeSound() { System.out.println("Bark");  }
   public void eat() { System.out.println("Dog is eating"); }
}
```
The class Dog implements the Animal interface, meaning it provides implementations for both makeSound() and eat().

Multiple Interface Inheritance: Unlike classes, Java allows a class to implement multiple interfaces. **This is one way to achieve multiple inheritance in Java**, as Java does not support multiple inheritance of classes.

```
public interface Swimmer { void swim();}

public class Fish implements Animal, Swimmer {
   public void makeSound() {System.out.println("Blub blub");    }
   public void eat() {System.out.println("Fish is eating"); }
   public void swim() {System.out.println("Fish is swimming"); }
}
```
In this example, Fish implements both Animal and Swimmer interfaces, meaning it must provide implementations for makeSound(), eat(), and swim().

Interface Inheritance: Interfaces **can inherit** from other **interfaces.** This means one interface can extend another, and the class that implements the child interface will have to implement the methods of both interfaces.

```
public interface Mammal extends Animal { void nurse();}

public class Human implements Mammal {
   public void makeSound() { System.out.println("Hello"); }
   public void eat() { System.out.println("Human is eating"); }
   public void nurse() {System.out.println("Human is nursing"); }
}
```
Default Methods: From Java 8 onwards, interfaces can contain **default methods** with a body. This allows you to provide a default implementation for methods, so classes that implement the interface do not have to provide an implementation for that method unless they want to override it.
```
public interface Animal {
void makeSound();
default void sleep() { System.out.println("Animal is sleeping"); }
}
```

No Constructors: An interface cannot have constructors, as it cannot be instantiated on its own. It is only implemented by a class.

Key Points to Remember about Interfaces:

- ✓ An interface cannot contain implementation code for its methods (unless using default methods).
- ✓ A class can implement multiple interfaces, allowing it to inherit multiple behaviors.
- ✓ Interfaces provide a way to define a contract of behaviors for different classes without dictating how the behaviors should be implemented.
- ✓ An interface can be extended by another interface.

Example of Interface Usage:

```
public interface Vehicle {
   void start();
   void stop();
}

public class Car implements Vehicle {
   public void start() {System.out.println("Car is starting"); }
   public void stop() {System.out.println("Car is stopping"); }
}

public class TestVehicle {
   public static void main(String[] args) {
      Vehicle myCar = new Car();
      myCar.start(); // Output: Car is starting
      myCar.stop();  // Output: Car is stopping }
}
```

In this example, Vehicle is an interface, and Car is a class that implements the interface, providing its own implementations for the methods start() and stop().

Chapter end