Chapter 3 : Inheritance

This presentation explores inheritance and polymorphism in Java, mechanisms that

- facilitate code reuse and evolution management.
- Inheritance allows objects of a class to access and extend data and methods of a parent class.
- Subclasses can redefine inherited variables and methods, creating a class hierarchy of superclasses and subclasses.

We'll also cover polymorphism, which enables objects to be instances of multiple classes, Java.

AK **by Ali KHALFI**

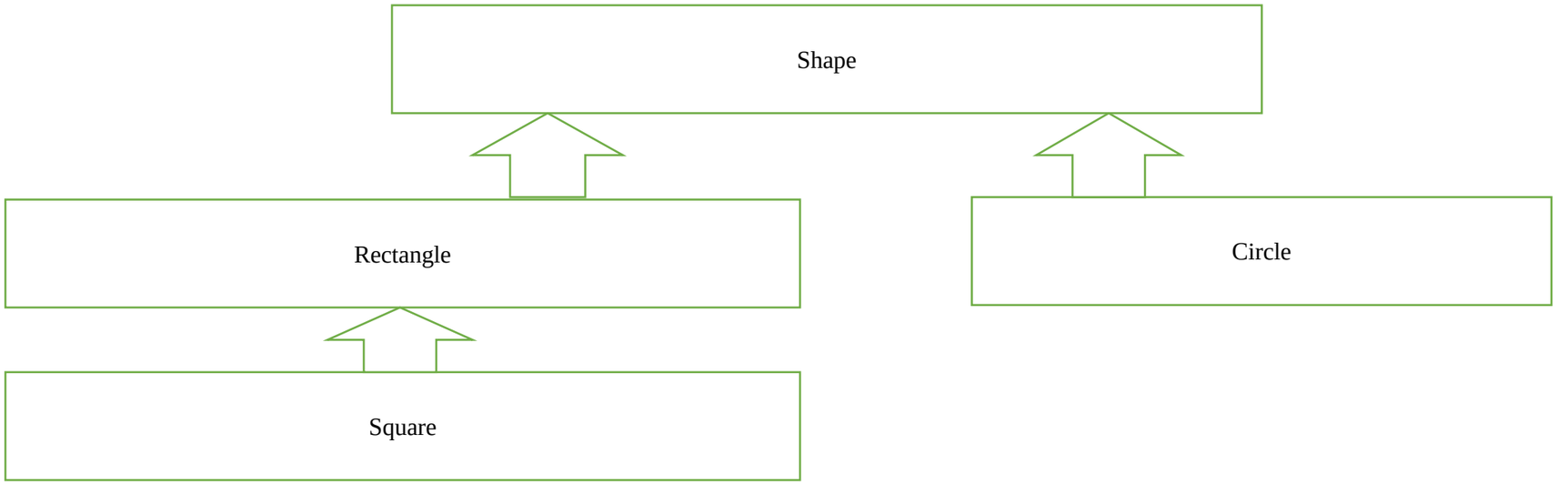
Principle of Inheritance

Hierarchical Organization

- Inheritance organizes classes hierarchically class A.
- The inheritance relationship is unidirectional
- if class B inherits from class A, we say that B is a subclass of A

Example: Shapes

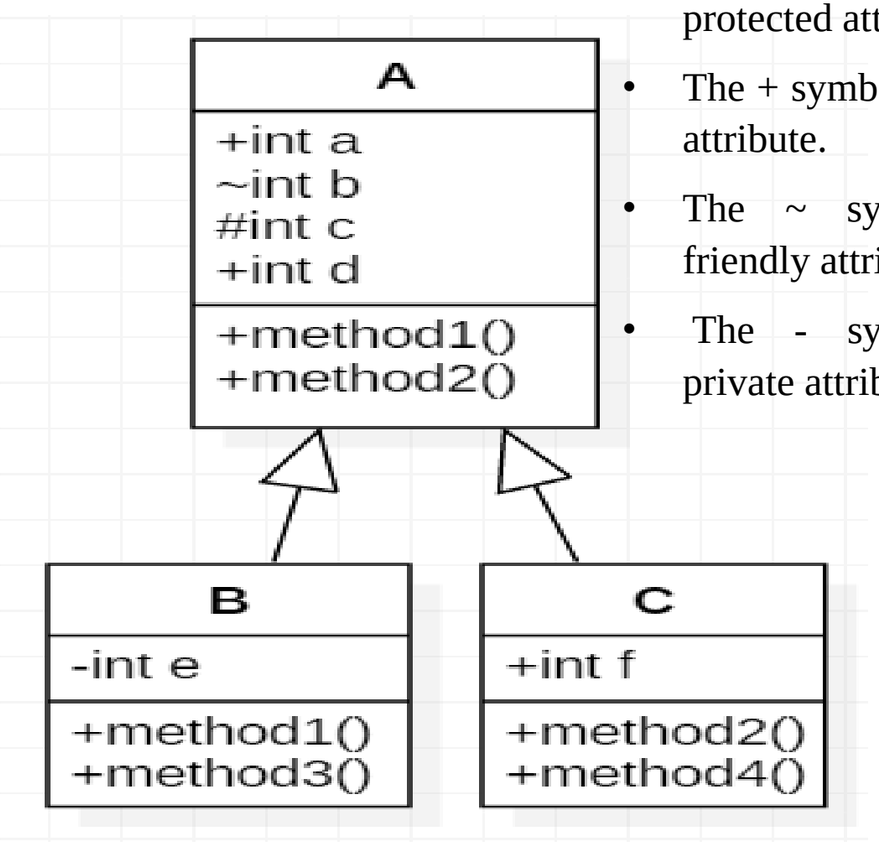
Square inherits from Rectangle, which inherits from Shape.



1.Subclasses and Inheritance

1.1. Definitions

- Defining a **new class** from **another** is derivation.
- The new class is a **derived class**, and the original is a **superclass** or **base class**.
- A class B inheriting from A gains A's **attributes** and **methods** without **redefinition**.
- A class can have only one superclass; multiple inheritance is **not allowed** in Java.
- The **Object** class is the **parent** class of all classes in Java



- 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.Subclasses and Inheritance

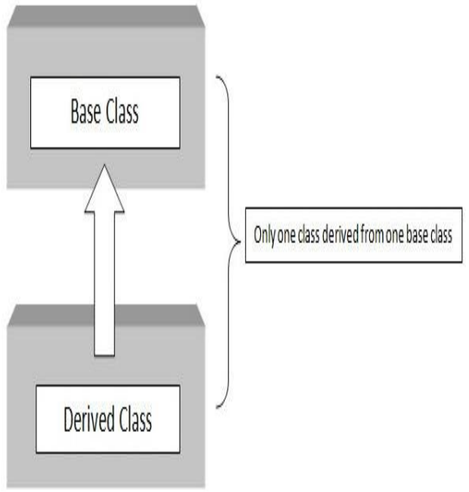
1.2 Key Concepts of Inheritance

- ✓ **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 **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.
- ✓ **Overloading:** The child class can **modify the signature of methods inherited (without affecting the name)** from the parent class. This is known as method overloading . 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. This is particularly useful when the child class overrides a method but still wants to use the functionality from the parent class.

2.Single and Multilevel Inheritance

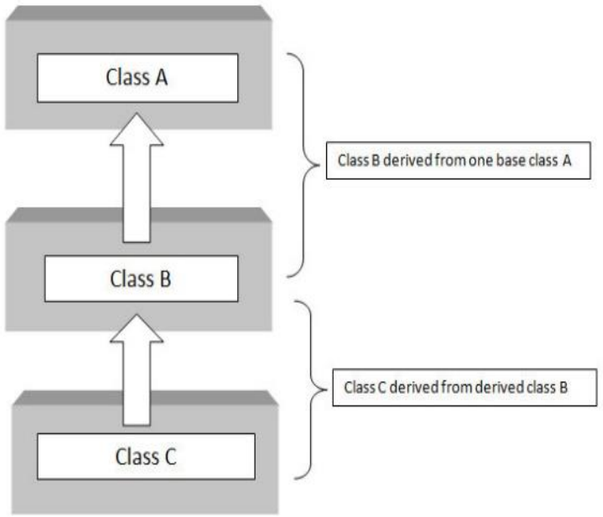
A-Single Inheritance

A child class inherits from only one parent class.



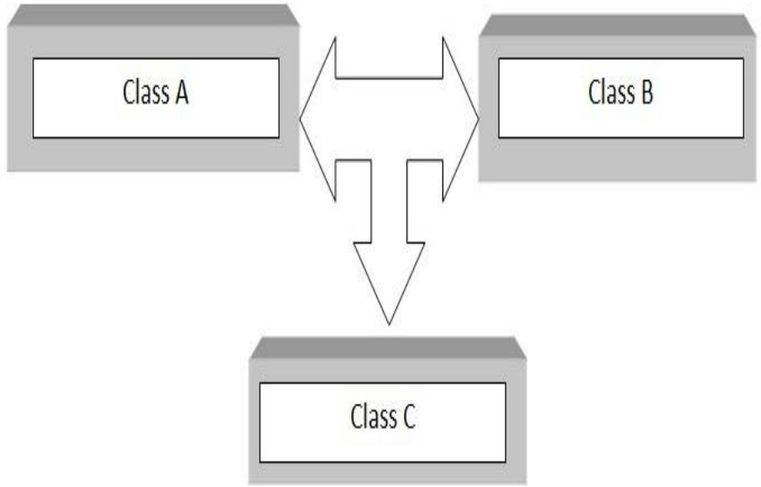
B-Multilevel Inheritance

A class derives from another class, which in turn is derived from another.
Class C has class B and class A as parent classes.



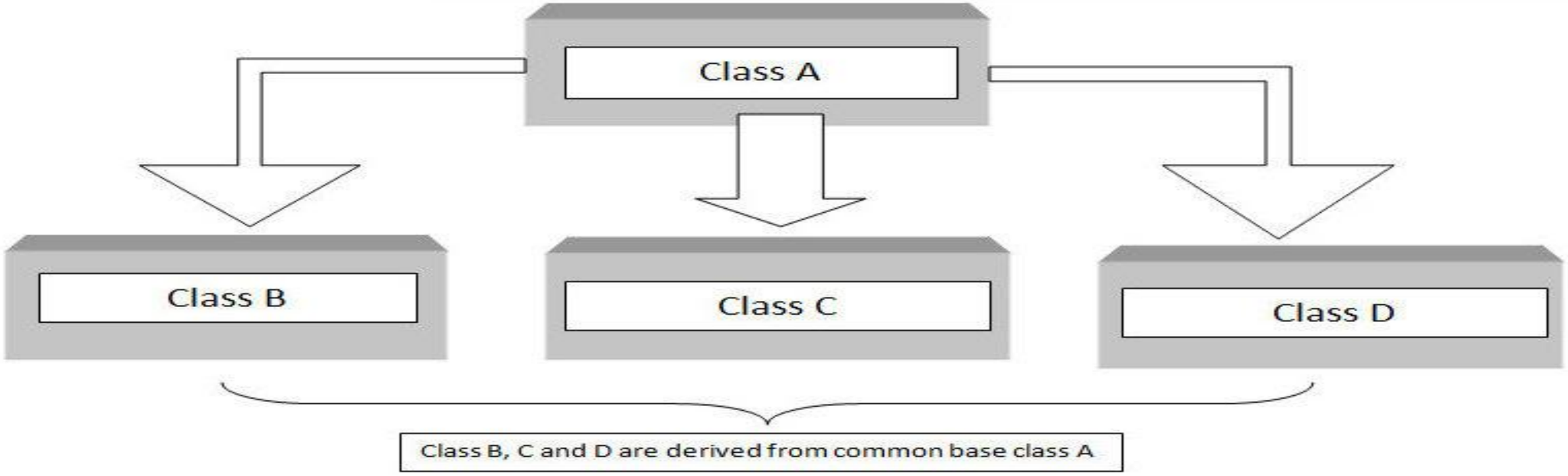
C-Multiple Inheritance

a child class inherits from more than one parent class. This allows the child class to inherit features from multiple sources



3.Hierarchical Inheritance

Multiple subclasses inherit properties and behaviors from a single base class. One parent class is extended by multiple child classes, allowing code reuse from the base class.



4.Polymorphism

Polymorphism refers to an object's ability to be an instance of multiple classes. It describes a method's capacity to exhibit different behaviors based on its context, such as parameter types.

Example :

```
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.**

5.Inheritance and Polymorphism in Java

a. Single Inheritance (extends)
a.1. Syntax:

class Child extends Parent { ... }

Example

class student extends Person { ... }

5.Inheritance and Polymorphism in Java

Example 1:

```
// Base class
class Graphic {
private int x, 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 width, int length) {this.width = width;this.length = length;}
double surface() { return (width * length); }}
```

Interpretation:

The Circle class **inherits attributes and methods** from **Graphic**, **overrides surface** and **display**, and **adds** the radius attribute.

5.Inheritance and Polymorphism in Java

1. A derived class **inherits** attributes and methods from **the base class**.
2. It can add its own **attributes** and **methods**.
3. **Methods inherited** from the parent class can **be overridden** or **overloaded**.
4. All Java classes implicitly extend the **Object** class.
5. Java allows direct inheritance from only **one base class** (no multiple inheritance).
6. Multiple inheritance can be simulated using **interfaces**.
7. 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.

5.Inheritance and Polymorphism in Java

b. Encapsulation in Inheritance :

5-1-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**

- The **protected** modifier **allows access** to inherited members within derived classes .

5.Inheritance and Polymorphism in Java

5.2.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**.

5.Inheritance and Polymorphism in Java

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.

5.Inheritance and Polymorphism in Java

(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
```

(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
```

(Case 21)

```
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. */
```

(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.
```

5.Inheritance and Polymorphism in Java

5.3.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 { ... }
```

5.Inheritance and Polymorphism in Java

5.4.Implicit and Explicit Type Casting (Cast):

The following assignments are implicitly allowed (without writing the casting instruction) without using parentheses, 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

inversely the explicit conversion is required

5.Inheritance and Polymorphism in Java

5.5.The final Modifier

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

```
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

```
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 :

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

c. Polymorphism

c.1.Method Overloading

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.

c. Polymorphism

c.2.Method Overriding

- 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.
- A public method cannot become private in a subclass

c. Polymorphism

c.3.Using Both Overloading and Overriding

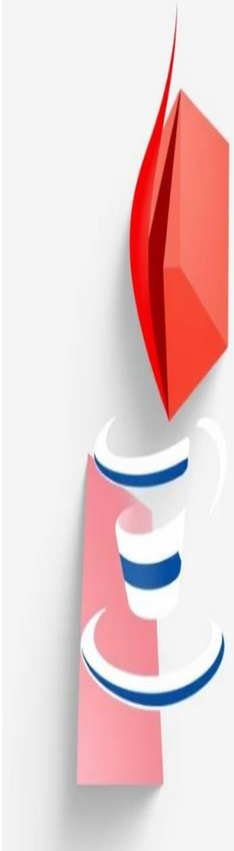
```
class A {
    public void f(int n) {}
    public void f(float n) {}
}

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
}
```

c. Polymorphism

Overloading vs. Overriding

Feature	Overloading	Overriding
Definition	Same name, **different parameters**	Same name **and signature** in subclass
Polymorphism	**Compile-time** (static binding)	**Runtime** (dynamic binding)
Inheritance	Not required	Requires **inheritance** (subclass)
Return Type	Can vary	Must be **same or covariant**
Example	**print(int), print(String)**	**Parent.show() vs Child.show()**



c. Polymorphism

Example :

```
public class test {
    public static void main(String[] args) {
        Graph g1 = new Graph();
        Rectangle r1 = new Rectangle();
        Circle c1 = new Circle();
       Graph t[] = new Graph[3];

        // 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] = g1; t[1] = r1; t[2] = c1;
        for (int i = 0; i < 3; i++) {
            system.out.println(t[i].surface());
            t[i].display(); }  // Each object executes its own method
        }    }
```

d. Abstract Classes

Functionalities

Place in an abstract class all the functionalities we want to be available when creating descendants.

Methods and Fields

An abstract class can contain methods and fields (which can be inherited), as well as one or more abstract methods.

Base Class

An abstract class serves as the base class for inheritance.

Instantiation

An abstract class does not allow object instantiation.

d. Abstract Classes

c. Syntax

abstract class A { // etc.}

Notes

- **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.
- **A class derived from a non-abstract class can be declared abstract.**

d. Abstract Classes

Notes

- 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.
- A class derived from a non-abstract class can 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.
- An abstract method cannot be declared static, private, or final.A
- n 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. A
- An abstract method cannot be declared static, private, or final.

d. Abstract Classes

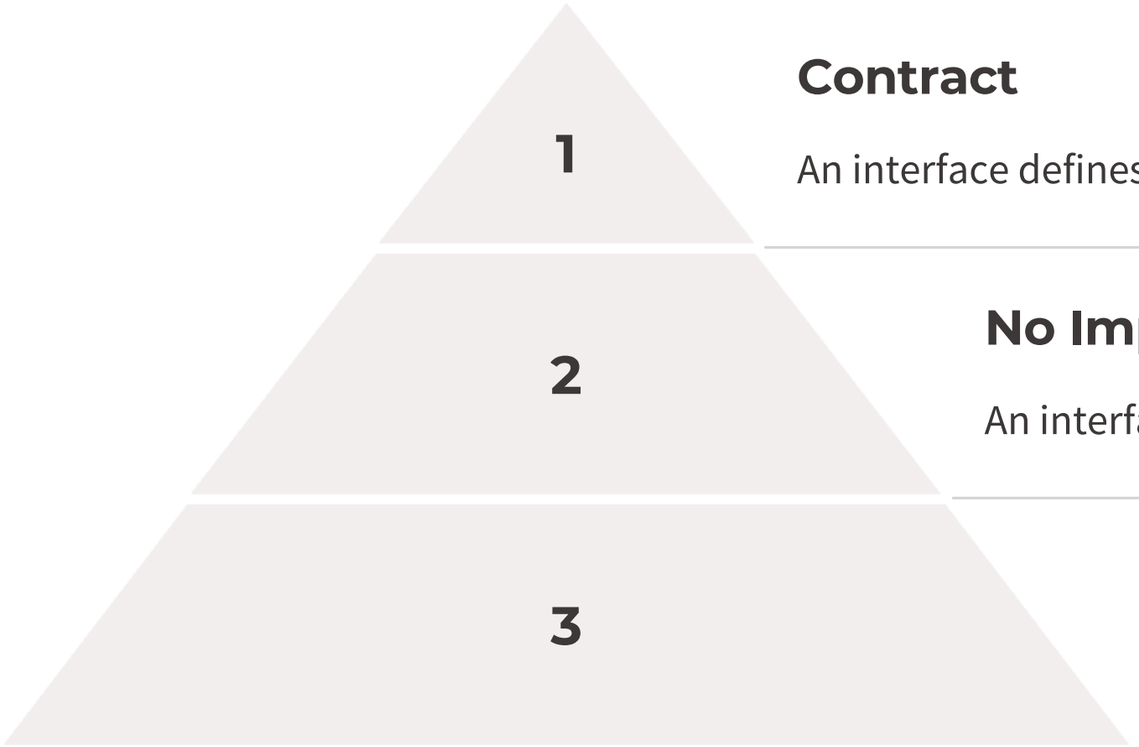
Example : Abstract Class

```
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



Contract

An interface defines a contract for what methods a class must implement.

No Implementation

An interface cannot contain any implementation code.

Multiple Interfaces

A class can implement multiple interfaces.

- 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.

e.Interfaces

Example

```
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**.

```
public class Dog implements Animal {
    public void makeSound() { System.out.println("Bark");  }
    public void eat() { System.out.println("Dog is eating"); }
}
```

e.Interfaces

Example

```
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().

e.Interfaces

- **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

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

e.Interfaces

Key Points to Remember about Interfaces:

- An interface cannot contain implementation code for its methods (unless using default methods).

```
public interface Animal {void makeSound();
default void sleep() { System.out.println("Animal is sleeping"); }}
```

- 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.

e.Interfaces : Example

```
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().

The chapter end

Questions ….?