4. Inheritance, polymorphism and Abstract class

A class denotes a set of objects that encapsulates static characteristics (variables) and dynamic characteristics (methods), serving as a model for creating objects. It acts as a template, defining the structure of objects by specifying variables and governing their actions through methods. In the context of oriented object programming, a class can have a superclass which leads to create a hierarchical relationship where it inherits attributes and behaviors from another class.

4.1. Inheritance

Inheritance is an important object-oriented programming concept in which one class gets the properties and behavior of another class. It establishes a IS-A relationship between two classes. This relationship is common known as a parent-child relationship. Inheritance is a mechanism wherein one class (called **child-class**) inherits the attributes and methods of another class (called **super-class**) with the gain that this class has the ability to update the behavior of her super class.

In Java, classes are organized into a hierarchy, where a superclass can pass on its attributes and methods to a subclass. Java supports single inheritance and the multiple inheritance (inheritance from more than one class) is not supported in the traditional sense. In java, a class can extend only one superclass. This design choice was made to avoid the complications and ambiguities associated with multiple inheritance. However, Java supports the multiple inheritance through interfaces. An interface in Java is a collection of abstract methods, and a class can implement multiple interfaces. By implementing multiple interfaces, a class can inherit the abstract methods declared in each interface.

4.1.1. Syntax

The syntax for inheritance in Java can be made by using ***extends*** keyword in order to indicate that a class is inheriting from another class.

```
class ChildClass extends ParentClass {
    // Fields and methods specific to the child class
}
```

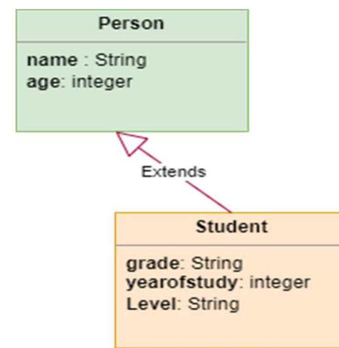
Example

Let's consider a "Person" class that serves as a representation of real-life humans.

```
class person {
  public String name;
  public int Age; }
```

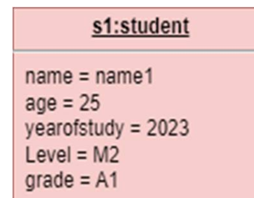
Now, we want to create another class which named ***student*** that represents a student in such university. We know that a student is a person, implying that this new class can inherit all characteristics of person class

```
class student extends person {
    public String grade;
    public int yearOfStudy;
    public String Level; }
```



An objects of student class is fulfilled as shown below:

```
class Main{
  public static void main(String[] args) {
    student s1 = new student();
    s1.name = "name1";
    s1.age = 25;
    s1.grade = "A1";
    s1.yearofstudy = 2023;
    s1.Level = "M2" ;}
```



4.2. Inheritance and constructor

If one class (subclass) extends another class (superclass), the ***super*** keyword is used as reference variable when a class (subclass) extends another class (superclass). It plays an important role in situations concerning inheritance. Here's a brief overview of how the ***super*** keyword is used:

4.2.1. Invoking Superclass Constructors: to ensure an appropriate initialization of the object with both own attributes and inherited attributes, the ***super*** keyword is used to invoke the constructor of the superclass from the constructor of the subclass.

4.2.2. Calling Superclass Methods: to allow the execution of the superclass method version before or after performing additional actions in the subclass, the **super** keyword used for this purpose while calling methods from the superclass.

Let's extend our previous example to demonstrate the use of constructors and inheritance in Java. We extend the person class with a constructor that have two parameters and invoking it in student class to have a proper initialization of any object of student class.

```
class person {
  public String name;
  public int Age;
  person(String name, int Age){
          this.name = name; this.age = age}}

class student extends person {
public String grade;
public int yearOfStudy;
public String Level; }
student(string name, int Age, String grade, int yearofstudy, String Level){
super(name,age); // invoking the constructor of superclass
this.grade = grade ; this.yearofstudy = yearofstudy; this.Level=Level }}
```

NB: The keyword ***'this'*** is employed to indicate the current instance, it acts as a reference variable that is used to refer to the current instance of the object within which it appears. It is often used to distinguish instance variables from local variables when they use the same name, similarly it is used to invoke current object's method.

4.3. Polymorphism

Polymorphism is an important concept of object-oriented programming. It just means the ability to take more than one form by an object. Polymorphism allows us to create flexible code. We can achieve polymorphism in Java using: ***overloading and overriding*** techniques. The example below explains why we need for polymorphism model, its shows a generic class "Shape" and the its child's classes "Circle and rectangle"

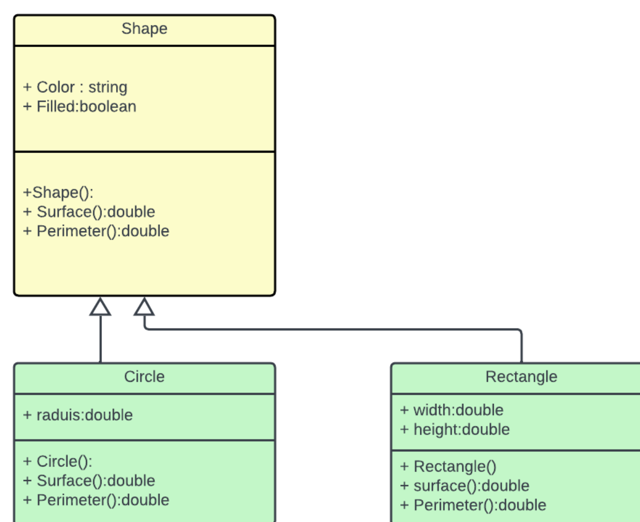


Figure 6 inheritance and polymorphism

4.3.1. Method overloading:

Method overloading which was introduced in previous chapter, its permits to create more than method with the same name and different parameter lists within class. We explain how compiler determines which method to invoke based on the arguments passed during a method call.

4.3.2. Method Overriding:

Another crucial approach for achieving polymorphism is method overriding, wherein a subclass redefines the implementation of a method that already exists in its superclass. This permits the modification of a method in a subclass while maintaining its existence in the superclass. In general, any subclass can override any method from a superclass unless the method is explicitly marked with the ***final*** or ***static*** keywords. It is essential to note that the overriding method must retain the same name and parameter list as the overridden method.

In previous example the inherited methods surface() and perimeter() for both circle and rectangle which representing the calculations for the surface area and perimeter, respectively, have not the same implementation, each class had the own version of these methods. So, in order to suit the specific formulas for both circles Rectangles. Similarly, these classes can override these methods with formulas appropriate for any object. This approach allows for a specialized implementation of geometric calculations tailored to the characteristics of each shape while following to the common definition in the Shape superclass.

```
Class shap{  …
      Public surface() { retrun …}
      Public perimeter() { retrun …}}
Class Circle extends shape{ …
      Public surface() { retrun Math.PI * Math.pow(radius, 2);;}
      Public perimeter() { retrun 2 * Math.PI * radius;}}
Class rectangle extends shape { …
      Public surface() { retrun width * height}
      Public perimeter() { retrun (width + height)*2 }}
```

4.4. Abstract Class and method

An abstract class in Java serves as a model for other classes that cannot be instantiated, it is destined to be a super class. Generally, it is designed to provide a common structure for its subclasses. An abstract class can hold both abstract methods (methods without any code) and concrete methods (methods with code), and can have attributes (fields or member variables) just like regular classes.

The main purpose of abstraction is to define a common interface and behavior for a group of related classes. It allows developers to focus on the relevant aspects of a system without paying attention to unnecessary complexities. This promotes code reusability, as the common code is centralized in the abstract class, and it establishes a clear hierarchy among related classes.

Abstraction is a major key factor in achieving polymorphism (previous section). Through abstract classes and interfaces, different objects can be treated consistently based on their common abstractions.

An abstract method is a method that haven't any code and is defined using the abstract keyword (like abstract class), it is designed for implementation in subclasses which their implementations must be provided by any class that extends the abstract class.

In the previous example the surface and perimeter methods of the shape class must be abstract methods because we cannot calculate the surface and perimeter of unknown form (must be left blank) and their existence is vital because all geometric from derived from shape class have these characteristics and can be calculated using the appropriate formulas for each kind of shape.

Example:

```
abstract class Shape { // abstract subclass (Circle)
   abstract double surface();
   void display() { System.out.println("This is a shape.");}}

class Circle extends Shape {
    // Implementation of abstract method for calculating area
    @Override
    double surface() { return Math.PI * Math.pow(radius, 2); } }
class Rectangle extends Shape {
    // Implementation of abstract method for calculating area
    @Override
    double surface() { return length * width; }}
```

Note: You cannot instantiate an object of the shape class using ***new shape()***. This will result a compilation error because an abstract class cannot be directly instantiated.

4.5. Interface

Interfaces are an essential part of object-oriented paradigm, they serve several purposes particularly in realizing abstraction (see previous section). Straightforwardly, an interface is a collection of abstract methods. It allows to define a set of abstract methods that must be all implemented by the class that implements the interface. An interface can also include constants (public, static, final fields) and default methods (methods with a default implementation) note that the data variables are not allowed in an interface.

Syntax: Writing an interface is similar to writing a class; the only difference is replacing the ***"class"*** keyword with the ***"interface"*** keyword.

```
public interface Name_Of_Interface {
 //constant, final, static fields
 // abstract method declarations   }
```

Example:

Writing an interface that contains a constant and two abstract methods

```
public interface Example_Of_Interface {
     int MY_CONST = 42; // Implicitly public, static, and final
     void myMethod1();
     void myMethod2(); }
```

Now, writing a class that implements this interface. This class must implement all abstract methods of the interface. To do this, we use the ***"implements"*** keyword like below:

```
public class Example_Of_class implements Example_Of_Interface  {
     public void myMethod1() { System.out.println("method1")};
     public void myMethod2() { System.out.println("method1")};  }
```

4.6. Overload (method surcharging)

In Java, the term "surcharge" is usually used to denote to method overloading (see chapter 3). Method overloading allows a class to have multiple methods with the same name but with different signature (parameter lists). The different versions of the method are called "overloaded."

4.7. Modifiers

Modifiers are keywords that you used to control the access of variables, methods and other program elements of classes (seen in chapter 3):

Here is some modifiers that can used in access control:

- **public**: Accessible from anywhere. There is no restriction on access.
- **protected**: Accessible within the same package and by subclasses, even if they are in a different package.
- **default** (no modifier): Accessible within the same package only. This is also known as package-private.
- **private**: Accessible only within the same class.
- **static :** Used to specify that a is a variable class and belongs to the class itself, or method is a method class
- **final:** in Java, the final modifier is used for variable, method or class, it act like a constant in variable which means that you cannot changing the value of this variables, for method, the final modifier is used to indicate that this method cannot be override and finally this modifier is used with class to specify that the class cannot be inherited. In summary, this modifier is used to apply restrictions on the use of a class, method, or variable.