



**CENTRO UNIVERSITARIO UAEM
ATLACOMULCO**

MATERIA

**PARADIGMAS DE LA
PROGRAMACIÓN**

DOCENTE

JULIO ALBERTO DE LA TEJA LÓPEZ

ALUMNO

ALEXIS VALENCIA MARTÍNEZ

CARRERA

**LICENCIATURA EN INGENIERÍA EN
SISTEMAS COMPUTACIONALES**

ICO-27

FECHA DE ENTREGA

12/SEPTIEMBRE/2023

Ejercicio 1: Clases abstractas e interfaces de las figuras Geométricas

- CÓDIGO

```
Public class Main {
    Public interface Calculable {
        Double calcularArea();
        Double calcularPerimetro();
    }

    Public abstract static class FiguraGeometrica implements Calculable {
        Protected String nombre;

        Public FiguraGeometrica(String nombre) {
            This.nombre = nombre;
        }
    }

    Public static class Circulo extends FiguraGeometrica {
        Private double radio;

        Public Circulo(double radio) {
            Super("Circulo");
            This.radio = radio;
        }

        @Override
        Public double calcularArea() {
            Return Math.PI * Math.pow(radio, 2);
        }

        @Override
        Public double calcularPerimetro() {
            Return 2 * Math.PI * radio;
        }
    }

    Public static class Rectangulo extends FiguraGeometrica {
        Private double base;
        Private double altura;

        Public Rectangulo(double base, double altura) {
            Super("Rectangulo");
            This.base = base;
            This.altura = altura;
        }

        @Override
        Public double calcularArea() {
            Return base * altura;
        }
    }
}
```

```

    }

    @Override
    Public double calcularPerimetro() {
        Return 2 * (base + altura);
    }
}

Public static class Triangulo extends FiguraGeometrica {
    Private double base;
    Private double altura;

    Public Triangulo(double base, double altura) throws Exception {
        Super("Triangulo");
        If (base <= 0 || altura <= 0) {
            Throw new Exception("No se puede crear un triángulo con base o altura menor o igual a cero");
        }
        This.base = base;
        This.altura = altura;
    }

    @Override
    Public double calcularArea() {
        Return (base * altura) / 2;
    }

    @Override
    Public double calcularPerimetro() {
        If (base <= 0 || altura <= 0) {
            System.out.println("Error: No se puede calcular el perímetro con base o altura menor o igual a cero");
            Return -1;
        }
        Return base + altura + Math.sqrt(Math.pow(base, 2) + Math.pow(altura, 2));
    }
}

Public static void main(String[] args) {
    Try {
        FiguraGeometrica[] figuras = new FiguraGeometrica[3];
        Figuras[0] = new Circulo(5);
        Figuras[1] = new Rectangulo(4, 2);
        Figuras[2] = new Triangulo(3, 6);

        For (FiguraGeometrica figura : figuras) {
            System.out.println("La figura es un " + figura.nombre + ", su área es " + figura.calcularArea() + " y su
perímetro es " + figura.calcularPerimetro());
        }
    } catch (Exception e) {

```

```
System.out.println("Error: " + e.getMessage());
```

```
}
}
}
```

- SALIDA

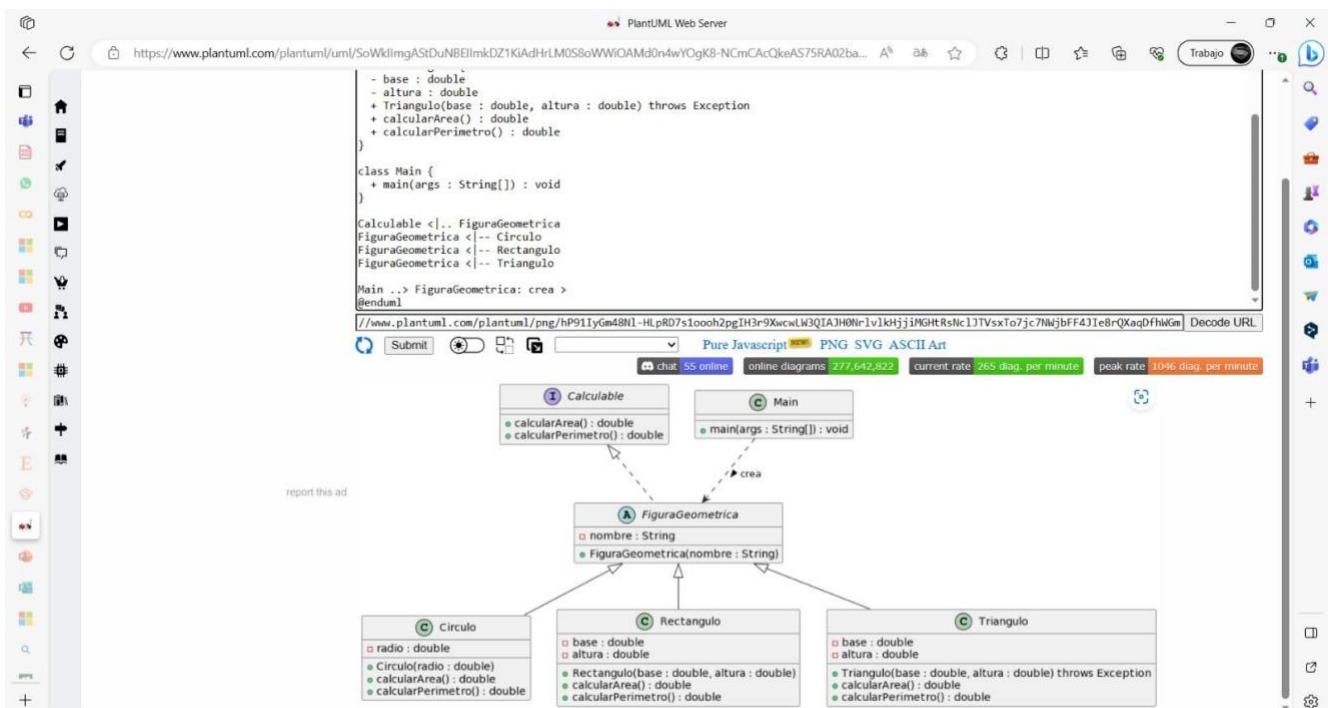
The screenshot shows a web-based IDE with a file explorer on the left, a code editor in the center, and a console on the right. The code editor displays the following Java code:

```
1 public class Main {
2     public interface Calculable {
3         double calcularArea();
4         double calcularPerimetro();
5     }
6
7     public abstract static class FiguraGeometrica implements
8         Calculable {
9         protected String nombre;
10
11         public FiguraGeometrica(String nombre) {
12             this.nombre = nombre;
13         }
14     }
15
16     public static class Circulo extends FiguraGeometrica {
17         private double radio;
18
19         public Circulo(double radio) {
20             super("Circulo");
21             this.radio = radio;
22         }
23
24         @Override
25         public double calcularArea() {
26             return Math.PI * Math.pow(radio, 2);
27         }
28     }
29
30     @Override
31     public static class Triangulo extends FiguraGeometrica {
32         private double base;
33         private double altura;
34
35         public Triangulo(double base, double altura) {
36             super("Triangulo");
37             this.base = base;
38             this.altura = altura;
39         }
40
41         @Override
42         public double calcularArea() {
43             return 0.5 * base * altura;
44         }
45
46         @Override
47         public double calcularPerimetro() {
48             // Calcula el perímetro de un triángulo
49             // usando la fórmula de Herón
50             double s = (base + altura + Math.sqrt(base * base + altura * altura)) / 2;
51             return Math.sqrt(s * (s - base) * (s - altura) * (s - Math.sqrt(base * base + altura * altura)));
52         }
53     }
54 }
55
56 public static void main(String[] args) {
57     if (args.length == 0) {
58         System.out.println("Uso: java Main <nombre> <radio o base y altura>");
59         return;
60     }
61     String nombre = args[0];
62     if (args.length == 2) {
63         double radio = Double.parseDouble(args[1]);
64         Circulo c = new Circulo(radio);
65         System.out.println("La figura es un Circulo, su área es " + c.calcularArea() + " y su perímetro es " + c.calcularPerimetro());
66     } else if (args.length == 4) {
67         double base = Double.parseDouble(args[1]);
68         double altura = Double.parseDouble(args[2]);
69         Triangulo t = new Triangulo(base, altura);
70         System.out.println("La figura es un Triangulo, su área es " + t.calcularArea() + " y su perímetro es " + t.calcularPerimetro());
71     } else {
72         System.out.println("Error: número incorrecto de argumentos");
73     }
74 }
```

The console on the right shows the output of the program:

```
> sh -c javac -classpath ./target/dependency/* -d . $(find . -type f -name '*.java')
> java -classpath ./target/dependency/* Main
La figura es un Circulo, su área es 78.53981633974483 y su perímetro es 31.41592653589793
La figura es un Rectangulo, su área es 8.0 y su perímetro es 12.0
La figura es un Triangulo, su área es 9.0 y su perímetro es 15.70820393249937
>
```

- DIAGRAMA UML



Ejercicio 2: Clases abstractas e interfaces en personajes en un Videojuego

- CÓDIGO

// Definimos la clase abstracta "Personaje"

```
Abstract class Personaje {
```

```
    String nombre;
```

```
    Int nivel;
```

```
    Public Personaje(String nombre, int nivel) {
```

```
        If (nivel < 1) {
```

```
            Throw new IllegalArgumentException("El nivel no puede ser menor que 1");
```

```
        }
```

```
        This.nombre = nombre;
```

```
        This.nivel = nivel;
```

```
    }
```

```
    Public void atacar() {
```

```
        If (nivel < 5) {
```

```
            Throw new RuntimeException("El nivel debe ser al menos 5 para realizar un ataque");
```

```
        }
```

```
    }
```

```
}
```

// Definimos la interfaz "HabilidadesMágicas"

```
Interface HabilidadesMágicas {
```

```
    Void usarHabilidadEspecial();
```

```
}
```

// Definimos la interfaz "HabilidadesFísicas"

```
Interface HabilidadesFísicas {
```

```
    Void gritar();
```

```
}
```

```
// Creamos una clase "Jugador" que extiende de "Personaje" e implementa "HabilidadesMágicas"
```

```
Class Jugador extends Personaje implements HabilidadesMágicas {
```

```
    String clase;
```

```
    Public Jugador(String nombre, int nivel, String clase) {
```

```
        Super(nombre, nivel);
```

```
        This.clase = clase;
```

```
    }
```

```
@Override
```

```
Public void atacar() {
```

```
    Super.atacar();
```

```
    System.out.println(nombre + " ataca con su espada!");
```

```
}
```

```
@Override
```

```
Public void usarHabilidadEspecial() {
```

```
    System.out.println(nombre + " usa su habilidad especial!");
```

```
}
```

```
}
```

```
// Creamos una clase "Enemigo" que extiende de "Personaje" e implementa "HabilidadesFísicas"
```

```
Class Enemigo extends Personaje implements HabilidadesFísicas {
```

```
    String tipo;
```

```
    Public Enemigo(String nombre, int nivel, String tipo) {
```

```
        Super(nombre, nivel);
```

```
        This.tipo = tipo;
```

```
}
```

```
@Override
```

```
Public void atacar() {
```

```
    Super.atacar();
```

```
    System.out.println(nombre + " ataca con su garra!");
```

```
}
```

```
@Override
```

```
Public void gritar() {
```

```
    System.out.println(nombre + " grita!");
```

```
}
```

```
}
```

```
// Creamos una clase "Paladín" que extiende de "Personaje" e implementa ambas interfaces
```

```
Class Paladín extends Personaje implements HabilidadesMágicas, HabilidadesFísicas {
```

```
    String clase;
```

```
    Public Paladín(String nombre, int nivel, String clase) {
```

```
        Super(nombre, nivel);
```

```
        This.clase = clase;
```

```
}
```

```
@Override
```

```
Public void atacar() {
```

```
    Super.atacar();
```

```
    System.out.println(nombre + " ataca con su espada y magia!");
```

```
}
```

```
@Override
```

```
Public void usarHabilidadEspecial() {  
    System.out.println(nombre + " usa su habilidad especial mágica!");  
}
```

```
@Override
```

```
Public void gritar() {  
    System.out.println(nombre + " grita con fuerza!");  
}
```

```
}
```

```
Public class Main {
```

```
Public static void main(String[] args) {
```

```
Try {
```

```
    Personaje personaje1 = new Jugador("Juan", 1, "Guerrero");
```

```
    Personaje1.atacar();
```

```
    ((Jugador) personaje1).usarHabilidadEspecial();
```

```
    Personaje personaje2 = new Enemigo("Esqueleto", 1, "Esqueleto");
```

```
    Personaje2.atacar();
```

```
    ((Enemigo) personaje2).gritar();
```

```
    Personaje personaje3 = new Paladín("Pedro", 1, "Paladín");
```

```
    Personaje3.atacar();
```

```
    ((Paladín) personaje3).usarHabilidadEspecial();
```

```
    ((Paladín) personaje3).gritar();
```

```
} catch (RuntimeException e) { // Modificado aquí
```

```
    System.out.println("Excepción capturada: El nivel debe ser al menos 5 para realizar un ataque");
```

```
}
```



```
}  
  
}
```

- SALIDA

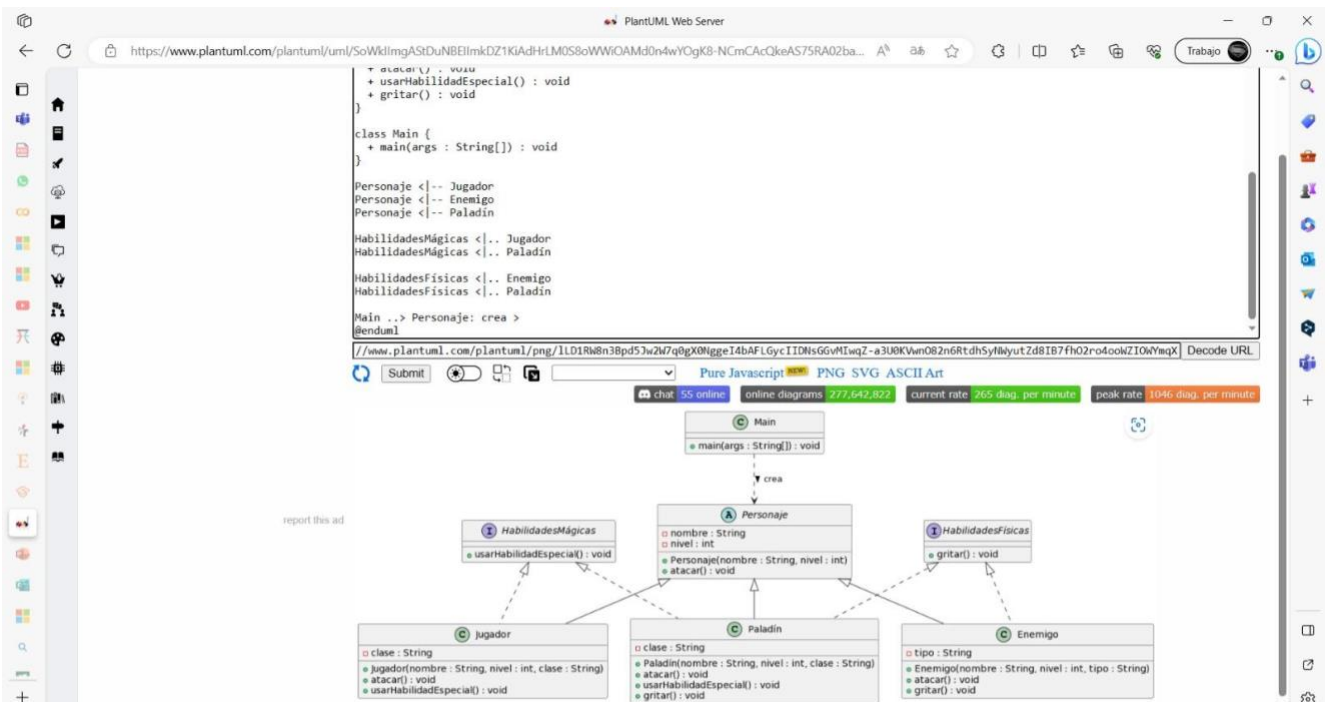
The screenshot shows a Replit IDE window titled "Main.java (5) - Replit". The code editor displays the following Java code:

```
10     this.nombre = nombre;  
11     this.nivel = nivel;  
12 }  
13  
14 public void atacar() {  
15     if (nivel < 5) {  
16         throw new RuntimeException("El nivel debe ser al menos  
17         5 para realizar un ataque");  
18     }  
19 }  
20  
21 // Definimos la interfaz "HabilidadesMágicas"  
22 interface HabilidadesMágicas {  
23     void usarHabilidadEspecial();  
24 }  
25  
26 // Definimos la interfaz "HabilidadesFísicas"  
27 interface HabilidadesFísicas {  
28     void gritar();  
29 }  
30  
31 // Creamos una clase "Jugador" que extiende de "Personaje" e  
32 // implementa "HabilidadesMágicas"  
33 class Jugador extends Personaje implements HabilidadesMágicas {  
34     String clase;  
35     public Jugador(String nombre, int nivel, String clase) {  
36         super(nombre, nivel);  
37         this.clase = clase;  
38     }  
39 }  
40  
41 class Main {  
42     public static void main(String[] args) {  
43         Jugador jugador = new Jugador("Jugador", 10, "Jugador");  
44         jugador.atacar();  
45     }  
46 }
```

The console output shows the following commands and results:

```
> sh -c javac -classpath ./target/dependency/* -d . $(find . -name '*.java')  
> java -classpath ./target/dependency/* Main  
Excepción capturada: El nivel debe ser al menos 5 para realizar un ataque
```

- DIAGRAMA UML



Ejercicio 3: Clases abstractas e interfaces en una Paletería

- CÓDIGO

Class Main {

Public static void main(String[] args) {

// Crear objetos de las clases PaletaAgua y PaletaCrema

PaletaAgua paletaAgua = new PaletaAgua("Fresa", 10.0, true);

PaletaCrema paletaCrema = new PaletaCrema("Chocolate", 15.0, true);

// Guardar los objetos en un arreglo de tipo Paleta

Paleta[] paletas = {paletaAgua, paletaCrema};

// Recorrer el arreglo y llamar a los métodos de cada objeto

For (Paleta paleta : paletas) {

Paleta.mostrarInformacion();

// Usar un casting apropiado según el tipo de paleta

If (paleta instanceof PaletaAgua) {

((PaletaAgua) paleta).mostrarBaseAgua();

} else if (paleta instanceof PaletaCrema) {

((PaletaCrema) paleta).mostrarTexturaCremosa();

}

}

}

// Definir la clase abstracta Paleta como una clase interna

Static abstract class Paleta {

Public String sabor;

Public double precio;

Public Paleta(String sabor, double precio) {

This.sabor = sabor;

```
    This.precio = precio;
```

```
}
```

```
Public void mostrarInformacion() {
```

```
    System.out.println("Sabor: " + this.sabor);
```

```
    System.out.println("Precio: " + this.precio);
```

```
}
```

```
}
```

```
// Definir la interfaz Caracteristica como una clase interna
```

```
Interface Caracteristica {
```

```
    // Declarar los métodos abstractos que deben implementar las clases que la usen
```

```
    Void mostrarBaseAgua();
```

```
    Void mostrarTexturaCremosa();
```

```
}
```

```
// Definir la clase PaletaAgua como una clase interna que hereda de la clase abstracta Paleta e implementa la interfaz Caracteristica
```

```
Static class PaletaAgua extends Paleta implements Caracteristica {
```

```
    Public boolean baseAgua;
```

```
    Public PaletaAgua(String sabor, double precio, boolean baseAgua) {
```

```
        Super(sabor, precio);
```

```
        This.baseAgua = baseAgua;
```

```
}
```

```
// Sobrescribir el método mostrarBaseAgua() según la característica de la paleta de agua
```

```
Public void mostrarBaseAgua() {
```

```
    System.out.println("Base de agua: " + (this.baseAgua ? "Sí" : "No"));
```

```
}
```

```
// Sobrescribir el método mostrarTexturaCremosa() según la característica de la paleta de agua

Public void mostrarTexturaCremosa() {

    System.out.println("Textura cremosa: No");

}

}
```

// Definir la clase PaletaCrema como una clase interna que hereda de la clase abstracta Paleta e implementa la interfaz Caracteristica

```
Static class PaletaCrema extends Paleta implements Caracteristica {

    Public boolean cremosa;

    Public PaletaCrema(String sabor, double precio, boolean cremosa) {

        Super(sabor, precio);

        This.cremosa = cremosa;

    }

}
```

```
// Sobrescribir el método mostrarBaseAgua() según la característica de la paleta de crema

Public void mostrarBaseAgua() {

    System.out.println("Base de agua: No");

}
```

```
// Sobrescribir el método mostrarTexturaCremosa() según la característica de la paleta de crema

Public void mostrarTexturaCremosa() {

    System.out.println("Textura cremosa: " + (this.cremosa ? "Sí" : "No"));

}

}

}
```

- SALIDA

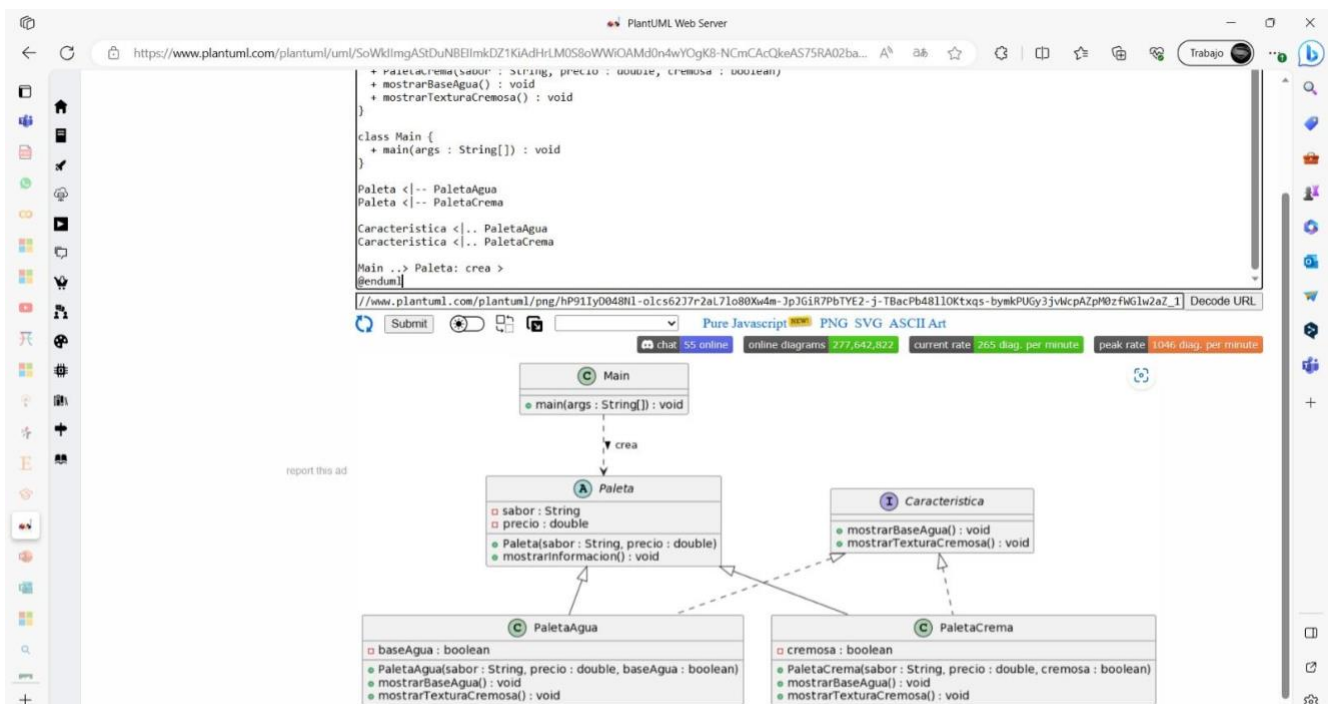
The screenshot shows a Replit IDE window titled 'Main.java - Main.java (6) - Replit'. The code in 'Main.java' defines a class 'Main' with a 'main' method. It creates two objects: 'PaletaAgua' (Fresa, 10.0, true) and 'PaletaCrema' (Chocolate, 15.0, false). It then iterates over an array of these objects, calling 'mostrarInformacion()' for 'PaletaAgua' and 'mostrarTexturaCremosa()' for 'PaletaCrema'. The code also defines an abstract class 'Paleta' with attributes 'sabor' and 'precio', and methods 'mostrarBaseAgua()' and 'mostrarTexturaCremosa()'. The output in the console shows the results of these method calls:

```

> sh -c javac -classpath ./target/dependency/* -d . $(find . -type f -name '*.java')
> java -classpath ./target/dependency/* Main
Sabor: Fresa
Precio: 10.0
Base de agua: Sí
Sabor: Chocolate
Precio: 15.0
Textura cremosa: Sí

```

- DIAGRAMA UML



The reason I applied this concept is to take advantage of the advantages of inheritance and polymorphism in Java. By using an abstract class, I can define common attributes and methods of palettes and avoid code repetition. By using an interface, I can declare the methods that the classes that implement it must have and ensure the consistency of their behavior