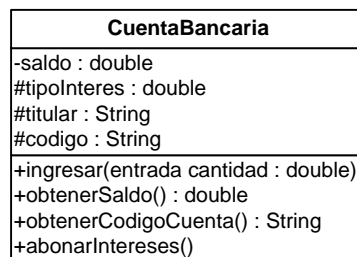


Tema 4.

Anexo I. Estudio de la implementación de la herencia, clases abstractas y polimorfismo en Java.

Para ver como se implementa la herencia, clases abstractas y el polimorfismo vamos a utilizar el dominio de información de cuentas bancarias empleado en el Tema 2. En nuestra práctica del Tema 2, nuestro programa manipulaba una clase denominada **CuentaBancaria**. La clase **CuentaBancaria** se muestra en el siguiente diagrama UML.

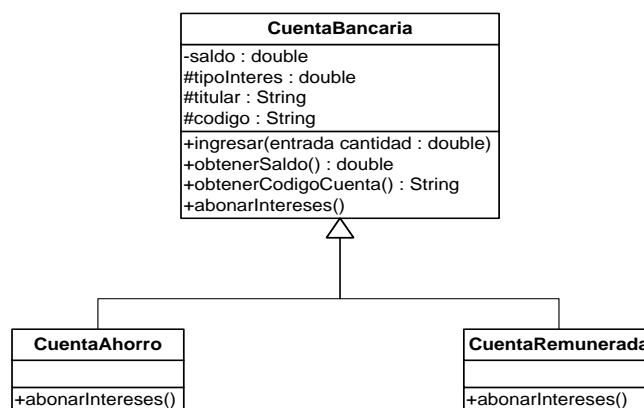


Supongamos que necesitamos manejar dos tipos de cuentas bancarias: **CuentaAhorro** y **CuentaRemunerada**. Las características de cada cuenta son las siguientes:

- **Cuenta Remunerada**
 - En este tipo de cuenta **se abonan intereses siempre** independientemente del saldo
- **CuentaAhorro**
 - En este tipo de cuenta **se abonan intereses sólo sí el saldo es superior a 3000** euros.

Como se puede observar el método **abonarIntereses()** tiene un comportamiento que está en función del tipo de cuenta. Es decir es diferente para cada subclase. Este método debe ser **sobrescrito** en cada subclase a fin de dar la implementación adecuada.

Analizando estas clases obtenemos el siguiente diagrama de clases UML



1. Implementación de la Generalización/especialización en Java

La definición de la clase base **CuentaBancaria.java** es la siguiente:

```
public abstract class CuentaBancaria {  
  
    private double saldo;  
  
    protected String titular;  
  
    protected double tipoInteres;  
  
    protected String codigo;  
  
    //Constructor  
  
    public CuentaBancaria(double saldo, String titular, String codigo, double  
tipoInteres) {  
  
        this.saldo = saldo;  
  
        this.titular = titular;  
  
        this.codigo = codigo;  
  
        this.tipoInteres = tipoInteres;  
  
    }  
  
    //Método para ingresar dinero en la cuenta  
  
    public void ingresar(double cantidad){  
  
        saldo+=cantidad;  
  
    }  
  
    //Método para obtener el saldo  
  
    public double obtenerSaldo(){return saldo;}  
  
    //Método para obtener el código de identificación  
  
    public String obtenerCodigoCuenta(){return codigo;}  
  
    //Método abstracto que necesita ser sobrescrito por cada subclase  
  
    abstract void abonarIntereses();  
  
}
```

Observe con atención el siguiente método de la clase **CuentaBancaria**

```
abstract void abonarIntereses();
```

El método se define como abstracto (**abstract**), esto indica que no tiene ninguna implementación (su cuerpo está vacío), luego la responsabilidad de la implementación corresponde a cada subclase.

Importante: Debido a que **CuentaBancaria** define un método abstracto debe declararse como abstracta (abstract class)

1.1.Intento de instancia de una clase abstracta

Si intentamos realizar una instancia de la clase *CuentaBancaria* el compilador nos avisara que no se puede instanciar una clase abstracta.

```
public class ProgramaPrincipal {  
  
    public static void main(String[] args) {  
  
        //El intento de una instancia de una clase abstracta dará error  
  
        CuentaBancaria c001=new CuentaBancaria(100,"Gustavo","C00023",3.2);  
  
    }  
  
}
```

1.2.Definición de una subclase en Java.

Sintaxis general:

```
public class <NombreSubclase> extends <NombreClaseBase>{  
  
    //Cuerpo de la clase  
  
}
```

La palabra reservada **extends** indica que la clase *<NombreSubclase>* hereda de la clase denominada *<NombreClaseBase>*.

Por defecto toda clase en Java es una subclase de la clase base denominada **Object**.

Subclase *CuentaAhorro.java*

```
public class CuentaAhorro extends CuentaBancaria {  
  
    //Constructor  
  
    public CuentaAhorro(double saldo, String titular, String codigo, double tipoInteres)  
    {  
  
        super(saldo, titular, codigo, tipoInteres);  
  
    }  
  
    void abonarIntereses() {  
  
        //Abonar intereses sólo si el saldo es superior a 3000 euros  
  
        if (this.obtenerSaldo()>=3000.0){  
  
            double cant=this.obtenerSaldo()*(tipoInteres/100);  
  
            this.ingresar(cant);  
  
        }  
  
    }  
  
}
```

Definición Subclase **CuentaRemunerada.java**

```
public class CuentaRemunerada extends CuentaBancaria{

    //Constructor

    public CuentaRemunerada(double saldo, String titular, String codigo, double
tipoInteres) {

        super(saldo, titular, codigo, tipoInteres);

    }

    public void abonarIntereses(){

        //Abonar intereses siempre

        double cant=this.obtenerSaldo()*(tipoInteres/100);

        this.ingresar(cant);

    }

}
```

Prueba de la subclase **CuentaAhorro**

```
public class ProgramaPrincipal {

    public static void main(String[] args) {

        CuentaAhorro c001=new CuentaAhorro(1000,"Luis Eduardo Abad Diaz","C00023",3.2);

        CuentaRemunerada c002=new CuentaRemunerada(3000,"Alfonso Garcia
Ruiz","C0043",5.0);

        //ingresar dinero en cuentas

        c001.ingresar(1000);

        c002.ingresar(3000);

        //abonar intereses en cuentas

        c001.abonarIntereses();

        c002.abonarIntereses();

        //ver los saldos

        System.out.println("saldo cuenta:"+c001.obtenerCodigoCuenta()+"
"+c001.obtenerSaldo());

        System.out.println("saldo cuenta:"+c002.obtenerCodigoCuenta()+"
"+c002.obtenerSaldo());

    }

}
```

Salida:

```
saldo cuenta:C00023: 2000.0

saldio cuenta:C0043: 6300.0
```

Como se puede comprobar la subclase ha heredado los atributos y métodos de la clase base **CuentaBancaria**

1.3. Análisis de miembros heredados por una subclase

Las subclases **CuentaAhorro** y **CuentaRemunerada** han heredado atributos y comportamiento de la clase base **CuentaBancaria**. Además cada subclase **deber proporcionar una implementación al método abonarIntereses().** Pero ¿Qué hereda exactamente la subclase **CuentaAhorro** y **CuentaRemunerada**? La siguiente tabla muestra un resumen de todo lo heredado por cada subclase.

Atributos	Métodos
private double saldo; protected String titular; protected String codigo; protected double tipoInteres;	void ingresar(double cantidad) double obtenerSaldo() String obtenerCodigoCuenta()
Obligación de sobrescribir el método abonarIntereses()	

Tabla herencia CuentaBancaria y subclases

IMPORTANTE: Los **constructores de una clase base** no son heredados por las subclases. Como se ve en el constructor de la clase base la inicialización correcta de una subclase implica inicializar la clase base primero. Esto se realiza con la siguiente llamada a *super(lista parametros)*.

```
public CuentaAhorro(double saldo, String titular, String codigo, double tipoInteres) {  
    super(saldo, titular, codigo, tipoInteres);  
}
```

Para **instanciar una clase base desde una subclase** en general utiliza la siguiente sintaxis:

```
public <NombreSubclase>(lista parámetros){  
    super(lista parámetros);  
    //Cuerpo del constructor de la subclase  
}
```

1.4. Acceso a los miembros de una clase base

Los **métodos de la subclase** no tienen acceso a los **miembros privados** de la clase base, pero sí a los miembros protegidos y públicos. Por ejemplo en el método **abonarIntereses()** de la subclase **CuentaRemunerada** no tiene acceso directo al atributo **saldo** ya que este es declarado privado (**private**) en la clase base **CuentaBancaria**. Para acceder al atributo saldo la subclase debe usar el método **obtenerSaldo()** como se observa en el siguiente fragmento.

```
public void abonarIntereses () {  
    //Abonar intereses siempre  
    double cant=this.obtenerSaldo()*(tipoInteres/100);  
    this.ingresar(cant);  
}
```

```
}
```

Sin embargo la subclase si tiene acceso al atributo **tipoInteres** de la clase base ya que este es declarado como protegido (**protected**).

Importante: La restricción `private` en los atributos de una clase base puede sorprender pero es necesaria para mantener la propiedad de encapsulación de la información. Si se pudiese tener acceso a los miembros privados de una clase, simplemente se conseguiría heredando. De esta forma se cumple que el interfaz de una clase es el único medio de manipulación de la clase. Luego como regla general es un mejor establecer el modificador **private** para los atributos de una clase base y que las subclases accedan a ellos a través de una interfaz

Tabla general de visibilidad de miembros entre clases base y subclases:

Puede ser accedido desde	Un miembro declarado en una clase como			
	privado	predeterminado	protegido	público
Su misma clase	Sí	Sí	Sí	Sí
Cualquier clase o subclase de su mismo paquete	No	Sí	Sí	Sí
Cualquier clase de otro paquete	No	No	No	Sí
Cualquier subclase de otro paquete	No	No	Sí	Sí

Tabla. Javier Ceballos (Java 2)

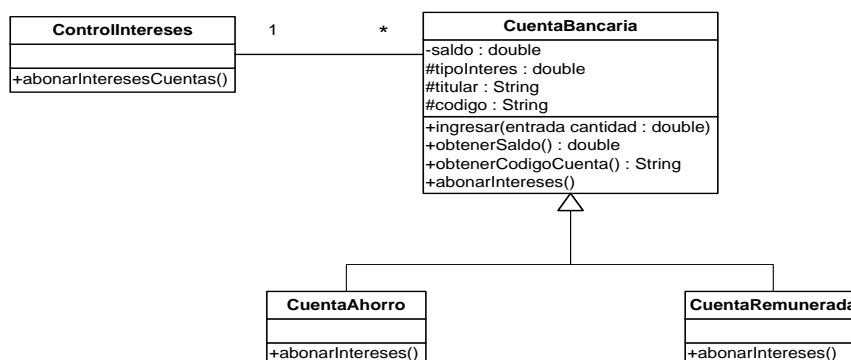
1.5. Estudio del uso del polimorfismo en Java

En ejemplo anterior declarábamos una clase base denominada **CuentaBancaria**. Esta clase era abstracta por que definía un método que no tenía implementación **abstract void abonarIntereses()**. La implementación adecuada de **abonarIntereses()** se proporciona por cada una de las subclases que heredan de esta clase base.

Vamos a suponer que tenemos una clase denominada **ControlIntereses** que expone un único método denominado **abonarInteresesCuentas()** con una lógica de negocio como lo siguiente:

Todos los días uno de cada mes se procede a abonar los interés correspondientes a cada una de las cuentas existente en el banco.

El diseño de clases en UML podría ser el siguiente:



El método **abonarInteresesCuentas()** tiene algunas consideraciones de diseño importantes:

1. La lógica de control de este método considera todas las cuentas existentes de manera homogénea y a para todas aplica el mismo tratamiento **abonarIntereses()**.
2. La lógica de **abonoIntereses()** es propia de cada subclase e independiente de este método.
3. Si en el futuro se añade una nueva cuenta este modulo no debería sufrir cambios.

Definición de la clase *ControladorIntereses.java*

```
import java.util.LinkedList;

public class ControladorIntereses {

    LinkedList cuentas;

    public ControladorIntereses() {

        this.cuentas =new LinkedList();

    }

    public void registrarCuenta(CuentaBancaria cb){

        cuentas.add(cb);

    }

    public void abonarInteresesCuentas(){

        CuentaBancaria cb=null;

        //Obtener nuemro de cuentas

        int tam=cuentas.size();

        for (int i=0;i<tam;i++){

            cb=(CuentaBancaria)cuentas.get(i);

            //Por cada cuenta abonar intereses

            cb.abonarIntereses();

        }

    }

}
```

Ahora probamos el polimorfismo

```
public class ProgramaPrincipal {

    public static void main(String[] args) {

        //crear cuentas bancarias

        CuentaAhorro ca001=new CuentaAhorro(2000,"Luis Eduardo Abad Diaz","CA001",3.2);

    }

}
```

```

        CuentaAhorro ca002=new CuentaAhorro(4000,"Manuel García Solis","CA002",3.2);

        CuentaRemunerada cr001=new CuentaRemunerada(3000,"Alfonso Garcia
Ruiz","CR003",5.0);

        //crear el controlador

        ControladorIntereses ctrl= new ControladorIntereses();

        //Registrar cada una de las cuentas en el controlador

        ctrl.registrarCuenta(ca001);

        ctrl.registrarCuenta(ca002);

        ctrl.registrarCuenta(cr001);

        //abonar intereses en todas las cuentas

        ctrl.abonarInteresesCuentas();

        //Comprobar el pago de intereses

        System.out.println("saldo cuenta:"+ca001.obtenerCodigoCuenta()+":
"+ca001.obtenerSaldo());

        System.out.println("saldo cuenta:"+ca002.obtenerCodigoCuenta()+":
"+ca002.obtenerSaldo());

        System.out.println("saldo cuenta:"+cr001.obtenerCodigoCuenta()+":
"+cr001.obtenerSaldo());

    }

}

```