UD 4: POLIMORFISMO

ÍNDICE

1. INTRODUCCIÓN

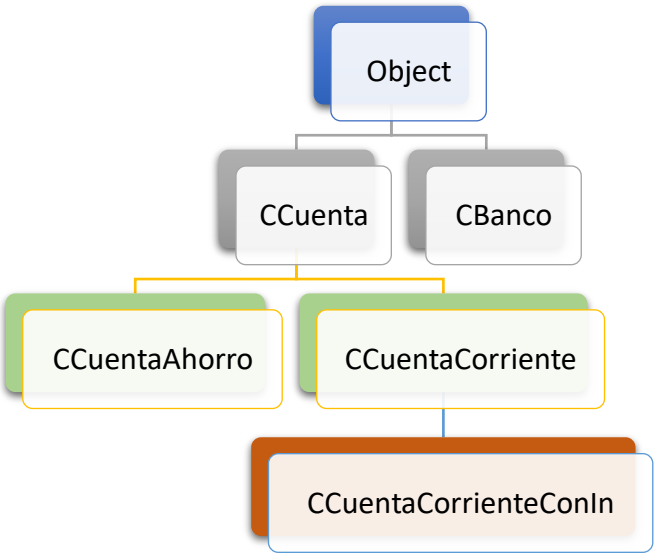
La facultad de llamar a una variedad de métodos utilizando exactamente el mismo medio de acceso, proporcionada por los métodos redefinidos en subclases, es a veces llamada polimorfismo.

La palabra "polimorfismo" significa "la facultad de asumir muchas formas", refiriéndose en Java a la posibilidad de llamar a muchos métodos diferentes utilizando una única sentencia.

Recordemos que cuando se invoca a un método que está definido en la superclase y redefinido en las subclases, la versión que se ejecuta depende de la clase del objeto referenciado, no del tipo de la variable que lo referencia.

Además, también "sabemos" que una referencia a una subclase puede ser convertida implícitamente por Java en una referencia a su superclase directa o indirecta. Esto significa que es posible referirse a un objeto de una subclase utilizando una variable del tipo de su superclase.

Supongamos la siguiente jerarquía de clases:



Pensemos en un array de referencias en la que cada elemento señale a un objeto de alguna de las subclases de la jerarquía construida anteriormente. ¿De qué tipo deben ser los elementos del array? Según nuestro ejemplo, deberían ser del tipo CCuenta, de esta forma el array podrá almacenar indistintamente, referencias a objetos de cualquiera de las subclases. Por ejemplo:

```
public class Test{

      public static void main(String[] args){

            CCuenta [] cliente= new Ccuenta [100];

            cliente [0]= new CCuentaAhorro ("cliente00", "3000123450",
            10000, 2.5,30);

            cliente [1]= new CCuentaCorriente ("cliente01",
            "6000123450", 10000, 2.0,1.0,6);
```

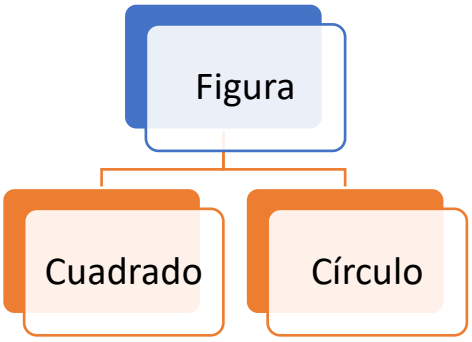
```
            cliente [2]=new CCuentaCorrienteConIn ("cliente02",
            "4000123450", 10000, 3.5, 1.0,6);


            for (int i=0; cliente [i]!=null; i++){
                  System.out.println(cliente[i].obtenerNombre ()+ ":");
                  System.out.println(cliente[i].intereses());
//Donde intereses pertenece a la clase CCuenta y está reescrito en las clases
//hijas CCuentaAhorro, CCuentaCorriente y CCuentaCorrienteConIn
            }
      }
}
```

Se define un array cliente de tipo CCuenta con 100 elementos inicialmente con valor null. Después se crean varios objetos de algunas de las subclases y se almacenan en los primeros elementos del array. Aquí, Java realizará una conversión implícita del tipo de la referencia devuelta por new al tipo CCuenta. Finalmente mostramos el nombre del cliente y los intereses que le corresponderán por mes. Si tenemos el método intereses redefinido de formas diferentes en cada una de las subclases, ¿En qué línea se aplica el polimorfismo?

En la última, porque invoca a las distintas definiciones del método intereses utilizando el mismo medio de acceso: una referencia a CCuenta que es el tipo del array cliente [i]. Así, un objeto tipo CCuenta se está comportando como una cuenta de ahorro, una cuenta corriente y una cuenta con intereses (polimorfismo).

Veremos todo esto con un clásico ejemplo sencillo usando de nuevo, Figura.

2. EJEMPLO 1



La palabra polimorfismo viene de “Múltiples formas”, por lo tanto, las operaciones polimórficas son aquellas que hacen funciones similares con objetos diferentes.

```java
package polimorfismo;
public abstract class Figura {
	private String nombre, color;
	//Es interesante el hecho de que sea una clase abstracta puesto que una figura es
	//un objeto general, no sabemos cómo calcular su área o su perímetro si no
	//concretamos qué tipo de figura es
	//de todas formas se podría hacer perfectamente sin necesidad de que la clase Figura
	//sea abstracta

	public Figura (){
	}
	public Figura(String nombre, String color) {
		super();
		this.nombre = nombre;
		this.color = color;
	}
	@Override
	public String toString() {
		return "Figura [nombre=" + nombre + ", color=" + color + "]";
	}
	public abstract double calcularArea ();
	public abstract double  calcularPerimetro ();

	public void metodoSoloDeFigura (){
		System.out.println("Solo estoy en la clase Figura, sin sobrescribir en las hijas");
	}
}
```

Clase Cuadrado que extiende a Figura

```java
package polimorfismo;
public class Cuadrado extends Figura{
	private double lado;
	public Cuadrado (){
	}
	public Cuadrado(String nombre, String color, double lado) {
		super(nombre, color);
		this.lado=lado;
	}
	public double getLado() {
		return lado;
```

```
	}
	public void setLado(double lado) {
		this.lado = lado;
	}
	@Override
	public String toString() {
		return "Cuadrado [lado=" + lado + ", toString()=" + super.toString() + "]";
	}
	@Override
	public double calcularArea() {
		return lado*lado;
	}
	@Override
	public double calcularPerimetro() {
		return lado*4;
	}
	public void mostrarLados (){
		System.out.println("Solo estoy en la clase Cuadrado porque los demás no tienen lados,
		en concreto tengo 4 lados ");
	}
}
```

Clase Círculo, que también es hija de Figura

```
package polimorfismo;
public class Circulo extends Figura{
	private double radio;
	public Circulo (){
	}
	public Circulo(String nombre, String color, double radio) {
		super(nombre, color);
		this.radio = radio;
	}
	public double getRadio() {
		return radio;
	}
	public void setRadio(double radio) {
		this.radio = radio;
	}
```

```
	@Override
	public String toString() {
		return "Circulo [radio=" + radio + ", toString()=" + super.toString()+"]";
	}
	@Override
	public double calcularArea() {
		// TODO Auto-generated method stub
		return Math.PI*radio*radio;
	}
	@Override
	public double calcularPerimetro() {
		// TODO Auto-generated method stub
		return 2*Math.PI*radio;
	}
	public void mostrarRadianes(){
		System.out.println("Solo estoy en la clase Círculo, porque los demás no pueden tener radianes en concreto tengo 2 pi radianes");
	}
}
```

Una clase normal, sin estar en la jerarquía de herencia para ver el uso de métodos

```
package polimorfismo;
public class OperacionesFiguras {
	public double calcularElAreaDeUnaFigura (Figura f){
		return f.calcularArea();
	}

	/*1) Dentro del método sumarAreas, se está llamando al método
	 * calcularElAreaDeUnaFigura, al que se le pasa como parámetro
	 * una figura del array
	 *2) El método calcularElAreaDeUnaFigura es el que distingue a qué versión de calcularArea hay que llamar
	 *según lo que haya en el listado, un cuadrado o un círculo*/
	public double sumarAreas (Figura [] listado){
		double resultado=0;
		for (int i=0; i<listado.length;i++){
			resultado=resultado+calcularElAreaDeUnaFigura(listado[i]);
		}
		return resultado;
	}
}
```

Clase Test para probar todo

```
package polimorfismo;

public class PruebaPolimorfismo {

	public static void main(String[] args) {

		//Prueba con objetos "sueltos"

		//Figura f1=new Figura (); No se pude crear porque Figura es abstracta

		Cuadrado cu1= new Cuadrado("Primer cuadrado", "Rojo", 1.0);

		Circulo ci1= new Circulo ("Primer Círculo", "Azul", 1.0);

		System.out.println(cu1);

		System.out.println(ci1);

		System.out.println("*********Área y perímetro**********");

		System.out.println("Área del primer cuadrado: "+cu1.calcularArea());

		System.out.printf("Perímetro del primer círculo: %.2f",ci1.calcularArea());


		//Prueba con polimorfismo

		System.out.println("\n\n\n*********************************************");


		Figura f1= new Cuadrado ("segundo cuadrado", "verde", 2.0);

		Figura f2= new Circulo ("Segundo círculo", "Amarillo", 2.0);


		System.out.println(f1);

		System.out.println(f2);

		System.out.println("*********Áreas**********");

		System.out.println("Usan el método de cada clase concreta porque está sobrescrito");

		System.out.println("Área del segundo cuadrado: "+f1.calcularArea());

		System.out.printf("Perímetro del segundo círculo: %.2f",f2.calcularArea());


		System.out.println("\nPero ahora no pueden usar los métodos que solo están en
		cuadrado y círculo porque son Figuras\n\n");

		//System.out.println(f1.mostrarLados ()); Error de compilación

		//System.out.println(f2.contarRadianes ());Error de compilación


		System.out.println("*****************************************************");

		System.out.println("*******Vamos ahora con el array de objetos**********\n\n");


		//Prueba con array de objetos

		Figura lista []= new Figura [4];

		//Cargamos el array con objetos cuadrado y círculo indistintamente

		//No podemos hacerlo con figuras porque la clase Figura es abstracta, si no lo fuera
sí se podría hacer
```

```
                lista[0]= new Cuadrado ("Un mísero cuadrado", "negro", 2.0);

                lista[1]= new Circulo ("Un mísero círculo", "blanco", 2.0);

                lista[2]= new Circulo ("Un círculo grisáceo", "gris", 2.0);

                lista[3]= new Cuadrado ("Un cuadrado desnudo", "transparente", 2.0);


                //Una prueba simple de lo que hay en el array es llamar al toString. Dependiendo de
                //lo que haya guardado en el

                //array se llamará a uno u otro toString


                for (int i=0; i<lista.length;i++){

                        System.out.println(lista[i].toString());

                }


                //Llamamos a los métodos de la clase OperacionesFiguras para calcular las áreas
                //individualmente

                //y la suma de todas las áreas


                //Creamos un objeto de la clase, aunque debería estar creado arriba, se pone aquí
                //para no liar


                OperacionesFiguras of= new OperacionesFiguras ();

                for (int i=0; i< lista.length;i++){

                        System.out.printf("El área del "+ (i+1)+ " es: %.2f
                        \n",of.calcularElAreaDeUnaFigura (lista[i]));

                }

                System.out.printf("La suma de todas las áreas es: %.2f", of.sumarAreas(lista));

                //Dejo para vosotros el probar con el método calcularPerímetro

        }

}
```

3. EJEMPLO 2 (de lectura)

Vamos a escribir un ejemplo, en el que se cree un objeto que represente a una entidad bancaria con un cierto número de clientes. Este objeto estará definido por una clase CBanco y los clientes serán objetos de alguna de las clases de la jerarquía construida en el ejemplo Banco que puedes descargar de la plataforma. *Puedes ver la jerarquía de clases al principio del tema.*

La estructura de datos que represente el banco debe ser capaz de almacenar objetos de CCuentaAhorro, CCuentaCorriente y CCuentaCorrienteConIn. Sabiendo que cualquier referencia a un objeto de una subclase puede convertirse implícitamente en una referencia a un objeto de su superclase, la estructura idónea es un array de referencias a la superclase CCuenta.

La clase banco tendrá:

ATRIBUTOS	
• CCuenta [ ] clientes	Array de referencias de tipo CCuenta

MÉTODOS	
• CBanco	Constructor. Inicia el array con cero elementos
• unElementoMas	Añade un elemento vacío (null) al final del array inrementando su longitud en 1
• unElementoMenos	Elimina un elemento cuyo valor sea null, decrementando su longitud en uno.
• insertarCliente	Asigna un objeto de alguna de las subclases de CCuenta al elemento i del array clientes.
• clienteEn	Devuelve el objeto que está en la posición i del array clientes
• longitud	Devuelve la longitud del array
• eliminar	Elimina el objeto que coincida con el número de cuenta pasado como argumento, poniendo el elemento a valor null.

El código será (el ejemplo está hecho para poder crear un array dinámico, es decir, aumentará en un elemento cuando se añada un objeto):

```
public class CBanco
{
  private CCuenta[] clientes; // matriz de objetos
  private int nElementos; // número de elementos de la matriz

  public CBanco()
  {
    // Crear una matriz vacía
    nElementos = 0;
    clientes = new CCuenta[nElementos];
  }

  private void unElementoMás(CCuenta[] clientesActuales)
  {
    nElementos = clientesActuales.length;
    // Crear una matriz con un elemento más
    clientes = new CCuenta[nElementos + 1];
    // Copiar los clientes que hay actualmente
    for ( int i = 0; i < nElementos; i++ )
```

```
    clientes[i] = clientesActuales[i];
  nElementos++;
}

private void unElementoMenos(CCuenta[] clientesActuales)
{
  if (clientesActuales.length == 0) return;
  int k = 0;
  nElementos = clientesActuales.length;
  // Crear una matriz con un elemento menos
  clientes = new CCuenta[nElementos - 1];
  // Copiar los clientes no nulos que hay actualmente
  for ( int i = 0; i < nElementos; i++ )
    if (clientesActuales[i] != null)
      clientes[k++] = clientesActuales[i];
  nElementos--;
}

public void insertarCliente( int i, CCuenta objeto )
{
  // Asignar al elemento i de la matriz, un nuevo objeto
  if (i >= 0 && i < nElementos)
    clientes[i] = objeto;
  else
    System.out.println("Índice fuera de límites");
}

public CCuenta clienteEn( int i )
{
  // Devolver la referencia al objeto i de la matriz
  if (i >= 0 && i < nElementos)
    return clientes[i];
  else
  {
    System.out.println("Índice fuera de límites");
```

```
    return null;
  }
}

public int longitud() { return nElementos; }

public void añadir(CCuenta obj)
{
  // Añadir un objeto a la matriz
  unElementoMás(clientes);
  insertarCliente( nElementos - 1, obj );
}

public boolean eliminar(String cuenta)
{
  // Buscar la cuenta y eliminar el objeto
  for ( int i = 0; i < nElementos; i++ )
    if (cuenta.compareTo(clientes[i].obtenerCuenta()) == 0)
    {
      clientes[i] = null; // enviar el objeto a la basura
      unElementoMenos(clientes);
      return true;
    }
  return false;
}

public int buscar(String str, int pos)
{
  // Buscar un objeto y devolver su posición
  String nom, cuen;
  if (str == null) return -1;
  if (pos < 0) pos = 0;
  for ( int i = pos; i < nElementos; i++ )
  {
    // Buscar por el nombre
```

```
      nom = clientes[i].obtenerNombre();
      if (nom == null) continue;
      // ¿str está contenida en nom?
      if (nom.indexOf(str) > -1)
        return i;
      // Buscar por la cuenta
      cuen = clientes[i].obtenerCuenta();
      if (cuen == null) continue;
      // ¿str está contenida en cuen?
      if (cuen.indexOf(str) > -1)
        return i;
    }
    return -1;
  }
}
```

Añadir un objeto se hace en dos pasos: uno, incrementar el array en un elemento y dos, asignar la referencia al objeto al nuevo elemento del array (de la misma forma se elimina).

Una clase de prueba para todo esto podría ser:

```
import java.io.*;
public class Test
{
  // Para la entrada de datos se utiliza Leer.class
  public static CCuenta leerDatos(int op)
  {
    CCuenta obj = null;
    String nombre, cuenta;
    double saldo, tipoi, mant;
    System.out.print("Nombre.................: ");
    nombre = Leer.dato();
    System.out.print("Cuenta.................: ");
    cuenta = Leer.dato();
    System.out.print("Saldo..................: ");
    saldo = Leer.datoDouble();
    System.out.print("Tipo de interés........: ");
```

```
    tipoi = Leer.datoDouble();
    if (op == 1)
    {
      System.out.print("Mantenimiento.....: ");
      mant = Leer.datoDouble();
      obj = new CCuentaAhorro(nombre, cuenta, saldo, tipoi, mant);
    }
    else
    {
      int transex;
      double imptrans;
      System.out.print("Importe por transacción: ");
      imptrans = Leer.datoDouble();
      System.out.print("Transacciones exentas..: ");
      transex = Leer.datoInt();
      if (op == 2)
        obj = new CCuentaCorriente(nombre, cuenta, saldo, tipoi,
                                   imptrans, transex);
      else
        obj = new CCuentaCorrienteConIn(nombre, cuenta, saldo,
                                        tipoi, imptrans, transex);
    }
    return obj;
  }

  public static int menú()
  {
    System.out.print("\n\n");
    System.out.println("1. Saldo");
    System.out.println("2. Buscar siguiente");
    System.out.println("3. Ingreso");
    System.out.println("4. Reintegro");
    System.out.println("5. Añadir");
    System.out.println("6. Eliminar");
    System.out.println("7. Mantenimiento");
```

```
    System.out.println("8. Salir");
    System.out.println();
    System.out.print("   Opción: ");
    int op;
    do
      op = Leer.datoInt();
    while (op < 1 || op > 8);
    return op;
  }

  public static void main(String[] args)
  {
    // (Para lectura por teclado) Definir una referencia al flujo estándar de salida: flujoS
    PrintStream flujoS = System.out;

    // Crear un objeto banco vacío (con cero elementos)
    CBanco banco = new CBanco();

    int opción = 0, pos = -1;
    String cadenabuscar = null;
    String nombre, cuenta;
    double cantidad;
    boolean eliminado = false;

    do
    {
      opción = menú();
      switch (opción)
      {
        case 1: // saldo
          flujoS.print("Nombre o cuenta, total o parcial ");
          cadenabuscar = Leer.dato();
          pos = banco.buscar(cadenabuscar, 0);
          if (pos == -1)
```

```
      if (banco.longitud() != 0)
        flujoS.println("búsqueda fallida");
      else
        flujoS.println("no hay clientes");
    else
    {
      flujoS.println(banco.clienteEn(pos).obtenerNombre());
      flujoS.println(banco.clienteEn(pos).obtenerCuenta());
      flujoS.println(banco.clienteEn(pos).estado());
    }
    break;
  case 2: // buscar siguiente
    pos = banco.buscar(cadenabuscar, pos + 1);
    if (pos == -1)
      if (banco.longitud() != 0)
        flujoS.println("búsqueda fallida");
      else
        flujoS.println("no hay clientes");
    else
    {
      flujoS.println(banco.clienteEn(pos).obtenerNombre());
      flujoS.println(banco.clienteEn(pos).obtenerCuenta());
      flujoS.println(banco.clienteEn(pos).estado());
    }
    break;
 case 3: // ingreso
 case 4: // reintegro
    flujoS.print("Cuenta: "); cuenta = Leer.dato();
    pos = banco.buscar(cuenta, 0);
    if (pos == -1)
      if (banco.longitud() != 0)
        flujoS.println("búsqueda fallida");
      else
        flujoS.println("no hay clientes");
    else
```

```
    {
      flujoS.print("Cantidad: "); cantidad = Leer.datoDouble();
      if (opción == 3)
        banco.clienteEn(pos).ingreso(cantidad);
      else
        banco.clienteEn(pos).reintegro(cantidad);
    }
    break;
  case 5: // añadir
    flujoS.print("Tipo de cuenta: 1-(CA), 2-(CC), 3-CCI  ");
    do
      opción = Leer.datoInt();
    while (opción < 1 || opción > 3);
    banco.añadir(leerDatos(opción));
    break;
  case 6: // eliminar
    flujoS.print("Cuenta: "); cuenta = Leer.dato();
    eliminado = banco.eliminar(cuenta);
    if (eliminado)
      flujoS.println("registro eliminado");
    else
      if (banco.longitud() != 0)
        flujoS.println("cuenta no encontrada");
      else
        flujoS.println("no hay clientes");
    break;
  case 7: // mantenimiento
    for (pos = 0; pos < banco.longitud(); pos++)
    {
      banco.clienteEn(pos).comisiones();
      banco.clienteEn(pos).intereses();
    }
    break;
  case 8: // salir
    banco = null;
```

```
        }
      }
      while(opción != 8);
    }
}
```

Dejo para vosotros la lectura detallada del código (en ocasiones antiguo y con trozos que se pueden hacer de manera más sencilla con cosas que todavía no hemos estudiado).

Por ejemplo, podemos observar que el mantenimiento de las cuentas (case 7) resulta sencillo gracias a la aplicación del polimorfismo, esto es, el método comisiones o intereses que se invoca para cada cliente depende del tipo del objeto referenciado por el elemento accedido del array clientes de CCbanco. Se puede ver todo el código en la plataforma, “ejemplo Banco”.