.UBAfiuba
FACULTAD DE INGENIERÍA

Abstracción, encapsulamiento, herencia y polimorfismo

TB025 - Paradigmas de Programación

Curso Essaya

Abstracción

Proceso cognitivo en el que se **seleccionan las características más relevantes** de un sistema, y se **omiten los detalles no importantes**.

Es la creación de un **modelo simplificado** para representar un **sistema complejo**, concentrándose únicamente en los conceptos relevantes para un **dominio particular**.

En el paradigma procedimental, la abstracción se logra principalmente mediante **funciones** y **procedimientos**.

```
# Alto nivel de abstracción
# Dominio: interacción con el usuario, cálculo de área.
def main():
  radio = pedir_numero("Ingrese el radio: ")
  grados = pedir_numero("Ingrese el ángulo (grados): ")
  area = area_del_arco(radio, grados)
  mostrar_mensaje(f"El área del arco es: {area}")
```

```
# Dominio: interacción mediante la consola
def pedir_numero(mensaje: str) -> float:
    return float(input(mensaje))

def mostrar_mensaje(mensaje: str) -> None:
    print(mensaje)
```

```
# Dominio: cálculos matemáticos
def area_del_arco(radio: float, grados: float) -> float:
  angulo_rad = grados_a_radianes(grados)
  return 0.5 * radio ** 2 * angulo_rad

def grados_a_radianes(grados: float) -> float:
    return grados * math.pi / 180
```

Fuente

Abstracción: interfaces

En el paradigma de objetos, una manera de abstraer objetos es mediante la definición de **interfaces**.

```
public interface Lista {
    void agregar(Object element);
    Object obtener(int index);
    int size();
}
```

```
public void armarListaSupermercado(Lista lista) {
    lista.agregar("mate");
    lista.agregar("café");
    lista.agregar("azúcar");
    lista.agregar("palmitos");
}
```

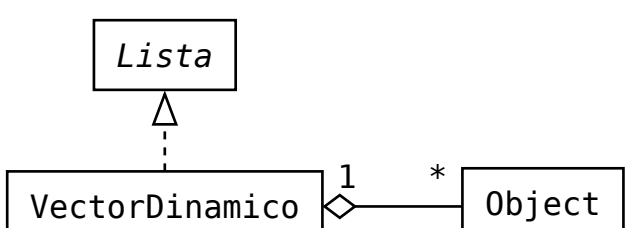
Abstracción: composición y agregación

Otra forma de lograr la abstracción es mediante la **composición** y **agregación** de objetos.

```java
public interface Lista {
    void agregar(Object element);
    Object obtener(int index);
    int size();
}
```

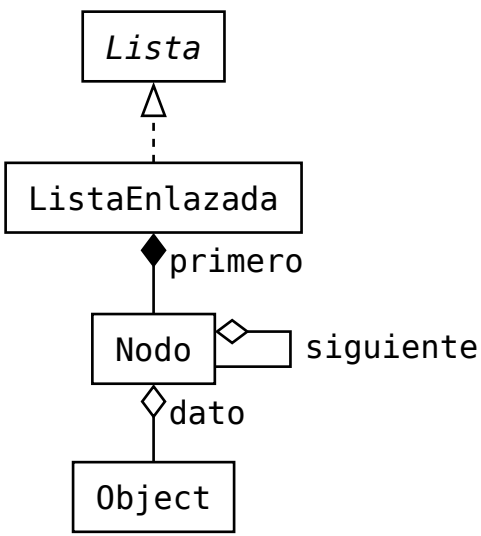
```java
public class VectorDinamico implements Lista {
    // Detalles de bajo nivel de abstracción
    private Object[] elementos;
    private int cantidad;

    public ArrayList() { ... }
    public int capacidad() { ... }
    public void extender(...) { ... }

    // Implementación de la interfaz:
    @Override public void agregar(Object element) { ... }
    @Override public Object obtener(int index) { ... }
    @Override public int size() { ... }
}
```

```java
public void armarListaSupermercado(Lista lista) {
    lista.agregar("mate");
    lista.agregar("café");
    lista.agregar("azúcar");
    lista.agregar("palmitos");
}
```



Abstracción: composición y agregación (cont.)

```java
public interface Lista { ... }

public class ListaEnlazada implements Lista {
    // composición: la ListaEnlazada es dueña del Nodo
    // y sus tiempos de vida están ligados.
    private Nodo primero;

    // ...
}

class Nodo {
    // agregación: el Nodo tiene una referencia a
    // otro Nodo, pero no es responsable de
    // su creación o destrucción.
    private Nodo siguiente;

    // agregación: el Nodo tiene una referencia a la
    // instancia de Object, pero no es responsable de
    // su creación o destrucción.
    private Object dato;

    // ...
}
```



Encapsulamiento

El término **encapsulamiento** tiene dos significados:

- La **agrupación de estado y comportamiento** en una unidad conceptual (la clase).
- La aplicación de **mecanismos de restricción de acceso** a los atributos y métodos, para **ocultar los detalles internos** de un objeto.

En Java la restricción de acceso se logra mediante el uso de **modificadores de acceso**: `private`, `protected` y `public`. [#]

```
public class VectorDinamico implements Lista {
    // Detalles de bajo nivel de abstracción
    private Object[] elementos;
    private int cantidad;

    public ArrayList() { ... }
    public int capacidad() { ... }
    public void extender(...) { ... }

    // Implementación de la interfaz:
    @Override public void agregar(Object element) { ... }
    @Override public Object obtener(int index) { ... }
    @Override public int size() { ... }
}
```

Fuente

Encapsulamiento: getters y setters

Una práctica común es definir **métodos de acceso** (getters) y **métodos de modificación** (setters) para acceder y modificar los atributos de una clase.

Los setters permiten entre otras cosas validar que no se violen la **invariantes** del objeto.

```java
public class Persona {
    private String nombre; // invariante: no es vacío
    private int edad; // invariante: edad >= 0

    public Persona(String nombre, int edad) { ... }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        if (nombre == null || nombre.isEmpty()) {
            throw new IllegalArgumentException("...");
        }
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        if (edad < 0) {
            throw new IllegalArgumentException("...");
        }
        this.edad = edad;
    }
}
```

Polimorfismo

πολλύς μορφή: “muchas formas”.

Es la capacidad de representar **múltiples tipos de datos** diferentes mediante **un único símbolo**.

Comunmente se reconocen las siguientes categorías de polimorfismo:

- **Polimorfismo ad-hoc**: funciona con un número limitado y conocido de tipos, no necesariamente relacionados entre sí.
  - **Polimorfismo por sobrecarga**
  - **Polimorfismo por coerción**
- **Polimorfismo universal**: funciona con un número ilimitado de tipos relacionados entre sí.
  - **Polimorfismo paramétrico**
  - **Polimorfismo por inclusión**

Fuente

Polimorfismo por sobrecarga

Es la capacidad de escribir dos o más **funciones o métodos** con el **mismo nombre** pero **diferente firma**.

```
public class Unidor {
    public int unir(int a, int b) { return a + b; }
    public double unir(double a, double b) { return a + b; }
    public String unir(String a, String b) { return a + b; }

    public static void main(String[] args) {
        Unidor unidor = new Unidor();
        System.out.println(unidor.unir(1, 2));
        System.out.println(unidor.unir(1.5, 2.5));
        System.out.println(unidor.unir("Hola, ", "mundo!"));
    }
}
```

El resultado es un polimorfismo aparente, ya que no se trata de un método que puede recibir muchos tipos de argumentos, sino que hay varios métodos con el mismo nombre, y en tiempo de compilación **se decide cuál usar según el tipo de los argumentos pasados**.

Polimorfismo por coerción

En programación, *coerción* significa **forzar a que un valor de un tipo sea convertido a otro tipo**.

```
public class Duplicador {
    public double duplicar(double x) { return x * 2; }

    public static void main(String[] args) {
        Duplicador duplicador = new Duplicador();
        int n = 5;
        System.out.println(duplicador.duplicar(n)); // int a double
        float f = 2.718f;
        System.out.println(duplicador.duplicar(f)); // float a double
    }
}
```

También se trata de un polimorfismo aparente, ya que la conversión que ocurre no cambia el hecho de que el método, en realidad, trabaja con un único tipo.

Polimorfismo paramétrico

Cuando se usa un **símbolo genérico** que luego puede ser sustituido por un tipo específico.

```
public interface Lista<T> {
    void agregar(T element);
    T obtener(int index);
    int size();
}

public class VectorDinamico<T> implements Lista<T> {
    private T[] elementos;
    private int cantidad;
    // ...
}
```

En Java, a esta variante de polimorfismo se la conoce como **Generics**.

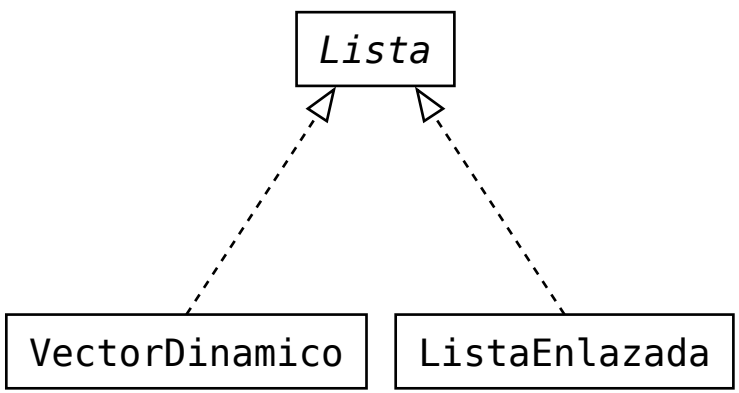
Polimorfismo por inclusión o de subtipos

Se basa en el hecho de que los tipos de datos forman una **jerarquía**, de forma tal que un objeto de un tipo **subtipo** puede ser tratado como un objeto de su tipo **supertipo**.

```
public interface Lista {
  void agregar(Object element);
  Object obtener(int index);
  int size();
}
```

```
public class VectorDinamico implements Lista { ... }
public class ListaEnlazada implements Lista { ... }
```

```
public void armarListaSupermercado(Lista lista) {
  lista.agregar("mate");
  lista.agregar("café");
  lista.agregar("azúcar");
  lista.agregar("palmitos");
}
```

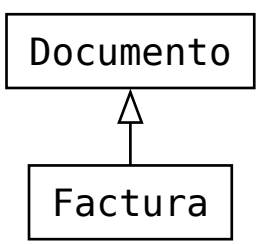


Herencia

Es un mecanismo que permite **crear nuevas clases basadas en clases existentes**, **heredando sus atributos y métodos**, y **agregando o modificando funcionalidad**.

A la clase previamente existente se la conoce como **superclase** (o *clase base*), y a las nuevas como **subclases** (o *clases derivadas*).

Algunos lenguajes de programación (ej: Java) soportan únicamente la **herencia simple**, donde una subclase hereda de una única superclase, y otros (ej: Python) soportan la **herencia múltiple**, donde una subclase puede heredar de múltiples superclases.




```java
class Documento {
    protected String titulo;

    public Documento(String titulo) {
        this.titulo = titulo;
    }

    public void imprimir(String s) {
        System.out.println("Documento: " + titulo);
    }
}

class Factura extends Documento {
    private double monto;

    public Factura(String titulo, double monto) {
        super(titulo);
        this.monto = monto;
    }

    @Override public void imprimir() {
        System.out.println("Factura " + titulo +
            "- Monto: $" + monto);
    }
}
```

Fuente

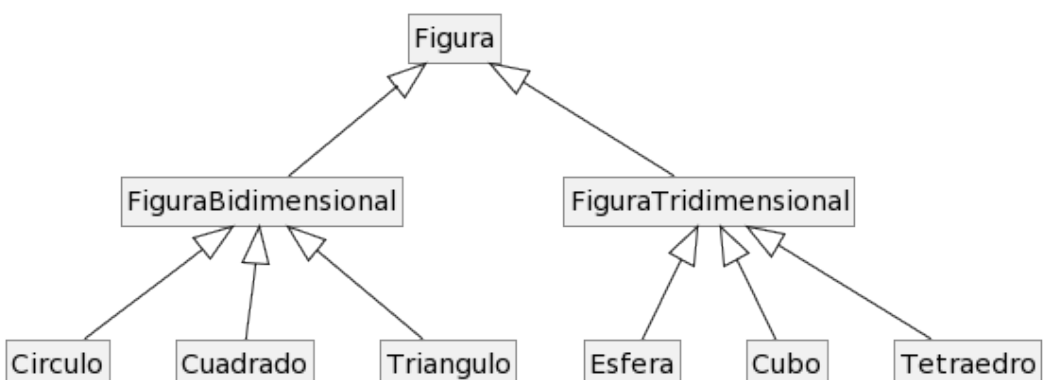
La herencia y las jerarquías de clases

Una subclase **es más específica que su superclase** y representa a un grupo más especializado de objetos.

Generalmente, la subclase exhibe los comportamientos de su superclase junto con otros adicionales específicos de esta subclase. Es por ello que a la herencia se la conoce a veces como **especialización** o **generalización**: la flecha se lee “es un(a)”.

La **superclase directa** es la clase de la cual la subclase hereda en forma explícita.

Una **superclase indirecta** es cualquier clase que esté arriba en la jerarquía de herencia, y de la cual la subclase hereda implícitamente.



Redefinición de métodos

Una subclase puede **sobrescribir** o **redefinir** cualquier método de la superclase con una implementación modificada (*method overriding*).

La palabra clave `super` permite acceder a los atributos y métodos de la superclase.

```
class Producto {
    private final double precioBase;

    public Producto(double precioBase) {
        this.precioBase = precioBase;
    }

    public double calcularPrecio() {
        return precioBase;
    }
}
```

```
class ProductoConDescuento extends Producto {
    private final double descuento;

    public ProductoConDescuento(double precioBase,
                                double descuento) {
        super(precioBase);
        this.descuento = descuento;
    }

    @Override
    public double calcularPrecio() {
        return super.calcularPrecio() * (1 - descuento);
    }
}
```

La clase Object

En Java, todas las clases heredan directa o indirectamente de la clase `Object`. [#]

Entre los métodos heredados se destaca `toString()`, que devuelve una representación en forma de cadena del objeto.

```java
public class Persona {
    private String nombre;
    private String apellido;

    public Persona(String nombre, String apellido) {
        this.nombre = nombre;
        this.apellido = apellido;
    }

    @Override
    public String toString() {
        return nombre + " " + apellido;
    }
}
```

```java
Persona p = new Persona("Juan", "Pérez");
System.out.println(p); // Imprime: Juan Pérez
```

Java: herencia y constructores

Los constructores **no se heredan**.

Si no se define un constructor, todas las clases tienen un **constructor por defecto**: el constructor sin parámetros.

Si en el cuerpo de un constructor no hay una llamada explícita a `super()`, **se invoca implícitamente** como primera instrucción.

De esta manera, cuando se crea un objeto empieza una cadena de llamadas a los constructores, desde la subclase hacia arriba en la jerarquía de herencia, hasta llegar a la clase `Object`.

Si se declara un constructor con parámetros, **el constructor por defecto deja de estar disponible**.

Es posible sobrecargar el constructor para permitir diferentes formas de inicializar el objeto.

Java: herencia y constructores (cont.)

```java
public class Animal {
    public Animal() {
      System.out.println("Soy un animal!");
    }
}

public class Ave extends Animal {
    public Ave() {
      System.out.println("No se me usa!");
    }

    public Ave(String color) {
      System.out.println("Soy un ave de color " + color +
"!");
    }
}
```

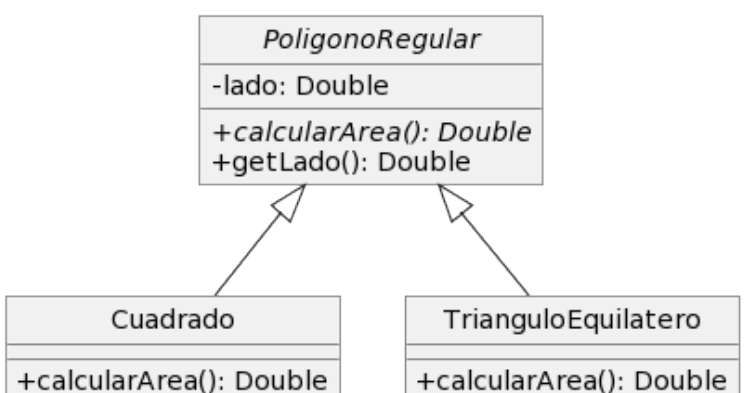
```java
public class Paloma extends Ave {
    public Paloma() {
      super("blanco");
      System.out.println("Soy una paloma!");
    }
}

public class Torcaza extends Paloma {
    public Torcaza(String s) {
      System.out.println("Soy una " + s + "!");
    }
}

public class Main {
    public static void main(String[] args) {
      Torcaza t = new Torcaza("torcaza");
    }
}
```

Clases abstractas

Si una clase o alguno de sus métodos está marcado con la palabra clave `abstract`, entonces la clase es **abstracta** y **no puede ser instanciada directamente**.

```java
public abstract class PoligonoRegular {
    private double lado;
    public PoligonoRegular(double lado) { this.lado = lado; }
    public double getLado() { return lado; }
    public abstract double calcularArea();
}

public class TrianguloEquilatero extends PoligonoRegular {
    public TrianguloEquilatero(double lado) { super(lado); }
    @Override public double calcularArea() {
        return (Math.sqrt(3) / 4) * Math.pow(getLado(), 2);
    }
}

public class Cuadrado extends PoligonoRegular {
    public Cuadrado(double lado) { super(lado); }
    @Override public double calcularArea() {
        return Math.pow(getLado(), 2);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        PoligonoRegular tri = new TrianguloEquilatero(5);
        PoligonoRegular cua = new Cuadrado(4);
        System.out.println("Área del triángulo: " +
            tri.calcularArea());
        System.out.println("Área del cuadrado: " +
            cua.calcularArea());
    }
}
```



Clases abstractas (cont.)

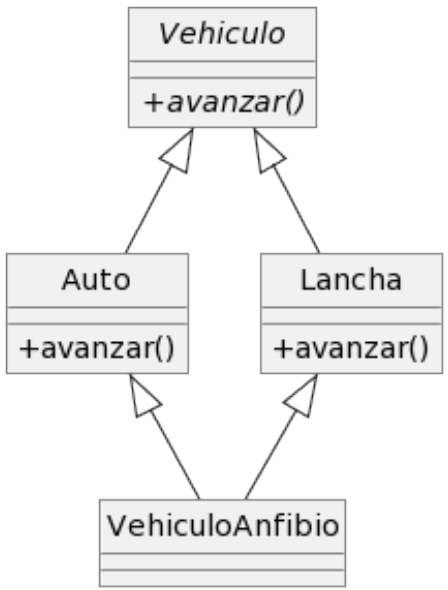
Una clase abstracta puede invocar uno de sus métodos abstractos.

```
public abstract class Producto {
    public abstract String nombre();
    public abstract int stock();
    public abstract double precioUnitario();

    public double calcularPrecioTotal() {
        return stock() * precioUnitario();
    }
}
```

```
public class Pelota extends Producto {
    @Override public String nombre() { return "pelota"; }
    @Override public int stock() { return 23; }
    @Override public double precioUnitario() { return 2500; }
}

public class Raqueta extends Producto {
    @Override public String nombre() { return "raqueta"; }
    @Override public int stock() { return 10; }
    @Override public double precioUnitario() { return 5000; }
}
```

Limitación de la herencia múltiple

El principal problema de la herencia múltiple se conoce como el **problema del diamante**.



¿Qué comportamiento debería tener un vehículo anfibio: el método `avanzar` que hereda de `Auto` o el que hereda de `Lancha`?

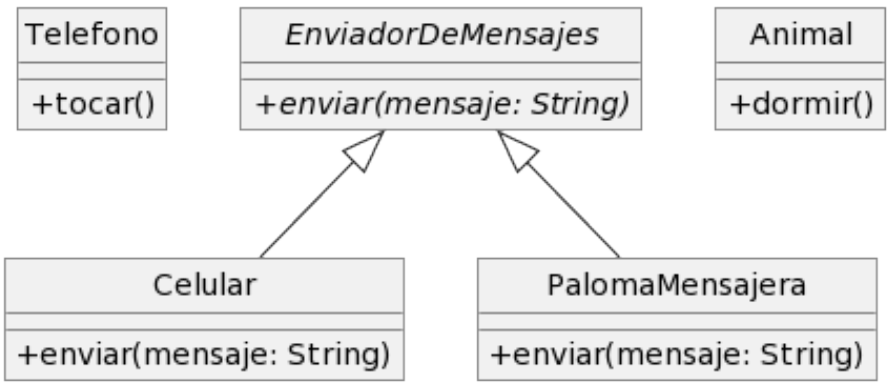
```
class Vehiculo {
    abstract void avanzar();
}

class Auto extends Vehiculo {
    @Override void avanzar() { ... }
}

class Lancha extends Vehiculo {
    @Override void avanzar() { ... }
}

// esto no se permite en Java:
class VehiculoAnfibio extends Auto, Lancha {
}
```

```
VehiculoAnfibio v = new VehiculoAnfibio();
v.avanzar();
```

Fuente

Limitación de la herencia simple

La inexistencia de la herencia múltiple podría resultar limitante para ciertos diseños.

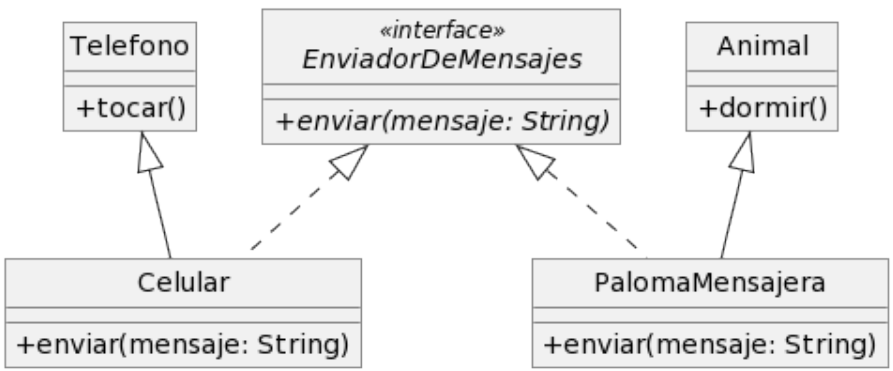


¿Cómo podemos expresar que un `Celular` es al mismo tiempo un `Telefono` y un `EnviadorDeMensajes`?

¿Y que una `PalomaMensajera` es un `Animal` y un `EnviadorDeMensajes`?

Generalización (herencia) vs Realización (interfaces)

La solución consiste modelar como **realizaciones** todas las relaciones con `EnviadorDeMensajes`, que no sería una clase sino una interfaz.



De esta manera se evita el problema del diamante, ya que en todos los casos hay una única implementación del método `enviar`.

Fuente

www.**ingenieria**.uba.ar

/ingenieriauba

/FIUBAoficial