

# **GUÍA DE PROGRAMACIÓN EN JAVA**

**Gracias a Roberto por la información.**

## CONTENIDOS:

1. Programación II
2. Teoría
3. 4.1 Composición
4. 4.2 Herencia
5. 5. Polimorfismo
6. 6. Genericidad
7. 7. Aspectos Funcionales
8. Ejercicios Tipo Examen del Bloque II
9. Práctica
10. XML

# Teoría



4.1 Composición →

4.2 Herencia →

5. Polimorfismo →

6. Genericidad →

7. Aspectos Funcionales →

Ejercicios Tipo Examen del Bloque II →

# 4.1 Composición

## ¿Qué es la Composición?

La composición es un principio fundamental de la POO que permite crear objetos complejos combinando otros objetos más simples. Se dice que la relación de composición consiste en “componer” o “agregar” objetos dentro de otros, obteniendo así estructuras más complejas.

Se podría decir que “A tiene B”. En la composición, un objeto contiene una o más instancias de otros objetos como parte de sus datos internos.

```
class Punto {  
  
    private final double x;  
    private final double y  
  
    public Punto(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public double calcularDistancia(Punto otroPunto) {  
        return Math.sqrt(Math.pow(otroPunto.x - this.x, 2) +  
            Math.pow(otroPunto.y - this.y, 2))  
    }  
}
```

```
class Linea {  
  
    private final Punto punto1;  
    private final Punto punto2;  
  
    public Linea(Punto punto1, Punto punto2) {  
        this.punto1 = punto1;  
        this.punto2 = punto2;  
  
    }  
    public double calcularLongitud() {  
        return punto1.calcularDistancia(punto2)  
    }  
}
```

```

public class Main {

    public static void main(String[] args) {
        Punto punto1 = new Punto(0, 0);
        Punto punto2 = new Punto(3, 4);
        Linea linea = new Linea(punto1, punto2);

        System.out.println("Distancia entre los puntos: " +
                           punto1.calcularDistancia(punto2));

        System.out.println("Longitud de la línea: " + linea.calcularLongitud());
    }
}

```

## multiplicidad

La multiplicidad en la composición se refiere a la cantidad de instancias de una clase que están asociadas con una instancia de otra clase. En el ejemplo anterior, la multiplicidad entre Linea y Punto es de 1 a 2, ya que cada línea está compuesta por dos punto.

## Tipos de composición

### composición fuerte

La **Composición Fuerte** Implica que un objeto es dueño de los objetos que contiene. Esto significa que los objetos contenidos no pueden existir sin el objeto contenedor. En términos de ciclo de vida, cuando el objeto contenedor se destruye, también se destruyen los objetos contenidos.

### composición débil

En la **Composición débil** los objetos contenidos pueden existir de forma independiente al objeto contenedor. En este caso, los objetos contenidos no se destruyen cuando el objeto contenedor se destruye.



La "**composición fuerte**" se suele referir como "**composición**" propiamente dicha, mientras que la "**composición débil**" se asocia a "**asociación**" o "**agregación**".



En Java, en la **composición fuerte**, los objetos contenidos se destruyen cuando el contenedor se destruye, gracias al recolector de basura de Java. No se observa que Linea destruya los Punto explícitamente porque la gestión de la destrucción está automatizada por el recolector de basura de Java.

!!

Cuando una clase usa a otra al recibirla o devolverla como parámetro en algún método, al hacer new dentro de un método, o al usarlas como variables locales se trata de dependencia, **NO** es composición

# Ejemplos

## composición débil

```
class Linea {  
  
    private Punto punto1;  
    private Punto punto2;  
  
    public Linea(Punto punto1, Punto punto2) {  
        this.punto1 = punto1;  
        this.punto2 = punto2;  
    }  
}
```

## composición fuerte

```
class Linea {  
  
    private final Punto punto1;  
    private final Punto punto2;  
  
    public Linea(int x1, int y1, int x2, int y2) {  
        this.puntoInicio = new Point(x1, y1);  
        this.puntoFin = new Point(x2, y2);  
    }  
}
```

## implementación de dos composiciones a la vez

-> ejemplo de composición débil entre un departamento que tiene varios profesores.

Implementa dos composiciones a la vez: entre el departamento y todos sus profesores y entre el departamento y su director, que es un profesor del departamento. Siempre debe haber un director en el departamento desde el inicio. Lanza excepciones si se viola la invariante.

Emplea arrays primitivos de Java, estilo Profesor[], con maximo 50, pero no rompas la encapsulación, no desveles que estás empleando un array, permite añadir un Profesor al final de la lista, y eliminar un profesor dada su posición. Da acceso a los profesores con un método para saber cuántos hay y otro para obtener un profesor por posición. El director se puede cambiar por otro profesor del departamento. Sin embargo, ten en cuenta esta invariante de clase: el director debe formar siempre parte de la lista de profesores, es decir, ten cuidado al cambiar el director o al eliminar un profesor.

```
public class Departamento {  
    private static final int MAX_PROFESORES = 50;  
    private Profesor[] profesores;  
    private Profesor director;  
    private int contadorProfesores;  
  
    public Departamento(Profesor director) {  
        this.director = director;  
        this.profesores = new Profesor[MAX_PROFESORES];  
        this.contadorProfesores = 0;  
        this.agregarProfesor(director); // Agregar al director como primer pro-  
fesor  
    }  
  
    public void cambiarDirector(Profesor nuevoDirector) throws IllegalArgumentException {  
        if (!esProfesorEnDepartamento(nuevoDirector)) {  
            throw new IllegalArgumentException("El nuevo director no es miembro  
del departamento");  
        }  
  
        this.director = nuevoDirector;  
    }  
  
    public void agregarProfesor(Profesor profesor) throws IllegalStateException {  
        if (contadorProfesores >= MAX_PROFESORES) {  
            throw new IllegalStateException("Se alcanzó el máximo número de pro-  
fesores en el departamento");  
        }  
  
        this.profesores[contadorProfesores++] = profesor;  
    }  
  
    public void eliminarProfesor(int posicion) {  
        if (posicion < 1 || posicion >= contadorProfesores) {  
            throw new IllegalArgumentException("Posición inválida");  
        }  
  
        if (profesores[posicion] == director) {  
            throw new IllegalArgumentException("No se puede eliminar al director  
del departamento");  
        }  
  
        for (int i = posicion; i < contadorProfesores - 1; i++) {  
            profesores[i] = profesores[i + 1];  
        }  
        contadorProfesores--;  
    }  
  
    public int cantidadProfesores() {  
        return contadorProfesores;  
    }  
}
```

```

public Profesor obtenerProfesor(int posicion) {
    if (posicion < 0 || posicion >= contadorProfesores) {
        throw new IllegalArgumentException("Posición inválida");
    }
    return profesores[posicion];
}

private boolean esProfesorEnDepartamento(Profesor profesor) {
    for (int i = 0; i < contadorProfesores; i++) {
        if (profesores[i] == profesor) {
            return true;
        }
    }
    return false;
}
}

```

## composición recursiva

Al igual que ocurre con las excepciones en Java, que pueden encerrar causas (que son excepciones), de forma recursiva, suponen un tipo especial de composiciones, denominadas **composiciones recursivas**. Pon un ejemplo en Java de una Persona, que sea inmutable, y que tiene una madre, que es otra Persona. Haz un main con un ejemplo de uso con una familia de personas, desde el nieto hasta la abuela. Enumera algún otro ejemplo clásico de composiciones recursivas.

```
public class Persona {  
    private final String nombre;  
    private final Persona madre;  
  
    public Persona(String nombre, Persona madre) {  
        this.nombre = nombre;  
        this.madre = madre;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public Persona getMadre() {  
        return madre;  
    }  
  
    @Override  
    public String toString() {  
        return "Persona{" +  
            "nombre='" + nombre + '\'' +  
            ", madre=" + madre +  
            '}';  
    }  
  
    public static void main(String[] args) {  
        // Crear la familia de ejemplo  
        Persona abuela = new Persona("Abuela", null);  
        Persona madre = new Persona("Madre", abuela);  
        Persona hijo = new Persona("Hijo", madre);  
        Persona nieto = new Persona("Nieto", hijo);  
  
        // Mostrar la información de la familia  
        System.out.println("Información de la familia:");  
        System.out.println("Abuela: " + abuela);  
        System.out.println("Madre: " + madre);  
        System.out.println("Hijo: " + hijo);  
        System.out.println("Nieto: " + nieto);  
    }  
}
```

## 4.2 Herencia

### ¿Qué es la herencia?

La herencia es un principio fundamental de la POO que permite crear nuevas clases a partir de clases existentes. Se dice que la relación entre clases heredadas es "A es-un B".

La relación "A es-un B" se refiere a cómo se establece la herencia en la POO. Si decimos "A es-un B", estamos indicando que la Clase A hereda de la Clase B. Es decir, A es una versión especializada o una subclase de B. Consideramos B la superclase de A. Esto implica que la Clase A hereda todas las características y comportamientos de la Clase B, pero a mayores también puede tener sus propias características y comportamientos únicos.

### Compatibilidad de tipo

En Java, la compatibilidad de tipo en herencias permite que un objeto de una subclase pueda ser tratado como un objeto de su superclase. Es decir:

```
Animal a = new Perro();
Animal a = new Animal();
```

### Herencia de estado y comportamiento

Los objetos de una subclase pueden heredar campos y métodos de su superclase. Los métodos de una superclase pueden estar ejecutando sobre instancia de la subclase



- Diseño: (Si no necesitas compatibilidad de tipos)
- No utilizar herencia para reutilizar código de la clase base
  - Favorecer composición frente a herencia
  - La herencia rompe la encapsulación

### Ejemplo

Soldado tiene un nombre (privado), y hay dos subtipos, un Artillero, que es capaz de disparar cohetes y un Zapador que pone minas. El artillero tiene un número de cohetes y el zapador un número de minas, accesibles mediante getters específicos.

```

public abstract class Soldado {
    private String nombre;

    public Soldado(String nombre) {
        this.nombre = nombre;
    }

    public abstract void saludar();
}

public class Artillero extends Soldado {
    private int numCohetes;

    public Artillero(String nombre, int numCohetes) {
        super(nombre);
        this.numCohetes = numCohetes;
    }

    public int getNumCohetes() {
        return this.numCohetes;
    }

    @Override
    public void saludar() {
        System.out.println("Hola, soy un artillero y mi nombre es " + this.nombre);
    }
}

```

```

public class Zapador extends Soldado {
    private int numMinas;

    public Zapador(String nombre, int numMinas) {
        super(nombre);
        this.numMinas = numMinas;
    }

    public int getNumMinas() {
        return this.numMinas;
    }

    @Override
    public void saludar() {
        System.out.println("Hola, soy un zapador y mi nombre es " + this.nombre);
    }
}

```

Ahora aprovechando la **compatibilidad de tipos**, creamos un array de Soldados y recorremos el array para que todos saluden:

```
public static void main(String[] args) {  
    Soldado[] soldados = new Soldado[3];  
    soldados[0] = new Artillero("Juan", 5);  
    soldados[1] = new Zapador("Pedro", 3);  
    soldados[2] = new Artillero("Carlos", 7);  
  
    for (Soldado soldado : soldados) {  
        soldado.saludar();  
    }  
}
```

Cada soldado saludará diciendo su tipo y nombre.

## Preguntas

### **¿Cuántos constructores se ejecutan y en qué orden?**

Al crear soldados concretos, se ejecutan dos constructores: primero el constructor de la clase base (Soldado) y luego el constructor de la subclase (Artillero o Zapador). El orden en que se ejecutan es primero el constructor de la clase base y luego el de la subclase.

### **¿Qué significa "super" dentro de un constructor?**

La palabra "super" dentro de un constructor se usa para llamar al constructor de la superclase. Esto es necesario para asegurar que todos los atributos de la superclase se inicialicen correctamente antes de que el constructor de la subclase haga su trabajo.

### **Si la clase base no tiene visible el constructor sin parámetros, ¿debo llamar a super siempre?**

En ese caso sí, debes llamar a super() en el constructor de la subclase, y debes hacerlo en la primera línea del constructor. Además, debes proporcionar los argumentos necesarios para uno de los constructores de la superclase que sea visible para la subclase.

## Casting

### **¿Qué es castear?**

"Castear" es el término utilizado para referirse a la conversión de valores de un tipo de dato a otro tipo de datos compatible. En herencia es una característica imprescindible.

al castear un objeto Java no convierte realmente los tipos de datos en sí. En cambio, lo que hace es cambiar el tipo de referencia que se utiliza para acceder a dicho objeto en memoria.

## Upcasting

El "upcasting" se refiere a convertir una referencia de un subtipo a un supertipo. El "upcasting" es totalmente seguro y se realiza implícitamente en Java, no necesita ningún tipo de manejo especial y no genera excepciones en tiempo de ejecución

## Downcasting

El "downcasting" es convertir una referencia de un supertipo a un subtipo. A diferencia del "upcasting", el "downcasting" debe hacerse explícitamente.

Cuando haces un downcasting, estás convirtiendo una referencia de un supertipo a un subtipo. Esta conversión no se realiza automáticamente por Java y, por lo tanto, debes indicar en el código que estás realizando esta conversión.

Continuando con el ejemplo de los soldados, es importante aclarar que, al hacer downcasting no especializa el soldado, el soldado ya era especialista (artillero, zapador, etc.) aunque en memoria estuviera referenciado como soldado.

!!

Si haces un **downcasting** a un objeto y dicho objeto no es en realidad del tipo de dato de la subclase, la conversión provocará una excepción de tipo **ClassCastException**

## instanceof

El operador **instanceof** se utiliza para verificar si un objeto es una instancia de una clase determinada. Devuelve **true** si el objeto es una instancia de la clase especificada o una subclase de la misma; de lo contrario, devuelve **false**.

💡

`instanceof` suele ir antes de un **downcasting** para asegurar que es del tipo necesario para la conversión

```
void dispararCohetes(soldado[] soldados){  
    for (Soldado soldado : soldados) {  
        if (soldado instanceof Artillero) {  
            Artillero artillero = (Artillero) soldado; // Downcasting  
            artillero.dispararcohete();  
        }  
    }  
}
```

## Acceso Protegido

El acceso "protegido" es un nivel de acceso que permite que los miembros (métodos y atributos) de una clase sean accesibles desde otras clases dentro del mismo paquete, así como desde las subclases, independientemente de si están dentro del mismo paquete o no. Esto proporciona un nivel de encapsulamiento más flexible que el acceso público, pero más restringido que el acceso privado.

Esto permite que las subclases accedan a esos miembros y los utilicen o los sobrescriban según sea necesario. La palabra clave utilizada para este acceso en Java es "**protected**".

!

puede peligrar la integridad de las invariancias de clase

## Ejemplos de uso

Crear una excepción personalizada

```
public class MiExcepcionPersonalizada extends Exception {  
  
    // Constructor vacío  
    public MiExcepcionPersonalizada() {  
        super();  
    }  
  
    // Constructor que acepta un mensaje de error  
    public MiExcepcionPersonalizada(String mensaje) {  
        super(mensaje);  
    }  
  
    // Otros métodos si es necesario  
}
```

# 5. Polimorfismo

## ¿Qué es el polimorfismo?

El polimorfismo es un principio fundamental de la POO que permite que los objetos puedan tomar muchas formas diferentes. Ayuda a escribir código genérico que pueda funcionar con diferentes tipos de objetos sin conocer su tipo específico en tiempo de compilación.

El polimorfismo permite que un objeto de una clase base pueda representar a objetos de sus subclases. Esto significa que un objeto puede tomar muchas formas diferentes, dependiendo del contexto en el que se utilice.

## Sobreescritura de métodos

La sobreescritura de métodos es una forma de polimorfismo que ocurre cuando una subclase proporciona una implementación específica de un método que ya está definido en su clase base.

Esto permite que las subclases proporcione su propia implementación de un método heredado, lo que permite que los objetos de la subclase se comporten de manera diferente cuando se invoca el mismo método.



El modificador `@Override` se utiliza para indicar que un método en una subclase está sobre escribiendo un método de la superclase.



La ligadura dinámica o enlace tardío es un concepto que se refiere al momento en el que se resuelve la llamada a un método o función durante el tiempo de ejecución en lugar de en tiempo de compilación. En Java es inherente al lenguaje y se maneja implícitamente.

## Ejemplo

Un Soldado, con un método saluda, con dos subclases: Zapador y Artillero, donde Zapador **sobre escribe** el método saludar, sustituyendo por completo su comportamiento. Ilustra el funcionamiento del polimorfismo creando un array de Soldados de dos tipos y luego recorriéndolo empleando referencias de tipo Soldado y llamando a saludar.

```

class Soldado {
    public void saludar() {
        System.out.println("¡Atención!");
    }
}

class Zapador extends Soldado {
    @Override
    public void saludar() {
        System.out.println("¡Volad todo!");
    }
}

class Artillero extends Soldado {
    // No se sobreescribe el método saludar, utiliza la implementación de Soldado
}
}

public class Main {
    public static void main(String[] args) {
        // Creamos un array de Soldados que incluye tanto Zapadores como Artilleros
        Soldado[] ejercito = new Soldado[2];
        ejercito[0] = new Zapador();
        ejercito[1] = new Artillero();

        // Recorremos el array y llamamos al método saludar
        for (Soldado soldado : ejercito) {
            soldado.saludar();
        }
    }
}

```

```

class Zapador extends Soldado {
    @Override
    public void saludar() {
        super.saludar(); // Llama al método saludar() de la clase base Soldado
        System.out.println("ZAPADOR A SUS ORDENES");
    }
}

```

## Ejemplo 2:

Vamos a poner un ejemplo nuevo con polimorfismo. Queremos implementar una clase Punto, con un método calcularDistanciaA, que permite calcular la distancia a otro Punto. Sin embargo, como queremos trabajar con puntos 2D y 3D, haz que ese método sea abstracto y haya dos implementaciones de ese cálculo de distancia. Emplea instanceof y downcasting para verificar que se recibe un punto compatible y poder calcular correctamente la distancia siempre entre puntos del mismo subtipo.

Aprovecha este diseño para crear ahora una clase Linea, que acepta Punto, sin saber de qué tipo es, y es capaz de dar su longitud independientemente de las dimensiones de sus puntos (las cuales desconoce).

```

abstract class Punto {
    public abstract double calcularDistanciaA(Punto otroPunto);
}

class Punto2D extends Punto {
    private double x;
    private double y;

    public Punto2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public double calcularDistanciaA(Punto otroPunto) {
        if (otroPunto instanceof Punto2D) {
            Punto2D punto2D = (Punto2D) otroPunto;
            double dx = this.x - punto2D.x;
            double dy = this.y - punto2D.y;
            return Math.sqrt(dx * dx + dy * dy);
        } else {
            throw new IllegalArgumentException("El punto no es de tipo 2D.");
        }
    }
}

class Punto3D extends Punto {
    private double x;
    private double y;
    private double z;

    public Punto3D(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    @Override
    public double calcularDistanciaA(Punto otroPunto) {
        if (otroPunto instanceof Punto3D) {
            Punto3D punto3D = (Punto3D) otroPunto;
            double dx = this.x - punto3D.x;
            double dy = this.y - punto3D.y;
            double dz = this.z - punto3D.z;
            return Math.sqrt(dx * dx + dy * dy + dz * dz);
        } else {
            throw new IllegalArgumentException("El punto no es de tipo 3D.");
        }
    }
}

class Linea {
    private Punto punto1;

```

```

private Punto punto2;

public Linea(Punto punto1, Punto punto2) {
    this.punto1 = punto1;
    this.punto2 = punto2;
}

public double longitud() {
    return punto1.calcularDistanciaA(punto2);
}
}

public class Main {
    public static void main(String[] args) {
        Punto punto2D1 = new Punto2D(1, 2);
        Punto punto2D2 = new Punto2D(4, 6);
        Linea linea2D = new Linea(punto2D1, punto2D2);
        System.out.println("Longitud de la línea 2D: " + linea2D.longitud());

        Punto punto3D1 = new Punto3D(1, 2, 3);
        Punto punto3D2 = new Punto3D(4, 6, 8);
        Linea linea3D = new Linea(punto3D1, punto3D2);
        System.out.println("Longitud de la línea 3D: " + linea3D.longitud());
    }
}

```

## ¿Qué son las clases abstractas?

Una clase abstracta es una clase que no puede ser instanciada directamente, sino que se utiliza como un modelo para otras clases que la extienden.

Puede contener tanto métodos abstractos como métodos con implementaciones concretas. Los métodos abstractos son declarados sin una implementación específica en la clase abstracta, dejando su implementación a las clases que la extienden.

Para definir una clase abstracta en Java, se utiliza la palabra clave **abstract** antes de la declaración de la clase. Los métodos abstractos se declaran también con la palabra clave **abstract** y no tienen implementación, solo firma del método.

!!

No puedes crear instancias directas de una clase abstracta, pero sí puedes crear instancias de clases que extienden la clase abstracta.

## ¿Qué son las interfaces?

Una interfaz es una colección de métodos abstractos y variables constantes. Los métodos definen un conjunto de acciones que una clase que implementa la interfaz debe proporcionar.

Las interfaces en Java actúan como contratos que especifican qué métodos deben estar presentes en cualquier clase que las implemente, pero no proporcionan ninguna implementación de estos métodos. (métodos vacíos)

Las interfaces se utilizan para lograr la abstracción y el polimorfismo, permitiendo que las clases comparten un conjunto común de métodos sin necesidad de herencia de clase



una clase puede implementar más de una interfaz

## Ejemplo

Programa básico: Crea una sola clase en Java que se llama Programa, con un método “iniciar”, que muestra un menú de tres operaciones. La opción 1 es “saludar”, la opción dos es “que día es hoy” y la operación tres es “salir”.

```
import java.util.Scanner;

class Programa {
    public void iniciar() {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("Menú:");
            System.out.println("1. Saludar");
            System.out.println("2. ¿Qué día es hoy?");
            System.out.println("3. Salir");
            System.out.print("Ingrese una opción: ");

            int opcion = scanner.nextInt();
            scanner.nextLine(); // Limpiar el buffer

            switch (opcion) {
                case 1:
                    System.out.println("Hola, ¡bienvenido!");
                    break;
                case 2:
                    System.out.println("Hoy es domingo.");
                    break;
                case 3:
                    System.out.println("¡Adiós!");
                    return;
                default:
                    System.out.println("Opción no válida. Inténtelo de nuevo.");
            }
        }
    }

    public static void main(String[] args) {
        Programa programa = new Programa();
        programa.iniciar();
    }
}
```

Programa mejorado: empleando polimorfismo, crea una abstracción para las distintas operaciones. Cada operación tiene un texto descriptivo a visualizar en el menú y un método para ejecutarse. Mejora la clase Programa para que ahora reciba un conjunto de estas operaciones en su constructor y que ellas mismas se ejecutan. Demuestra cómo Programa ya no se necesita modificar aunque el programa tenga más operaciones, pero sigue siendo útil para mostrar el menú e interactuar con el usuario.

```

interface Operacion {
    String getDescripcion();
    void ejecutar();
}

class Saludar implements Operacion {
    @Override
    public String getDescripcion() {
        return "Saludar";
    }

    @Override
    public void ejecutar() {
        System.out.println("Hola, ¡bienvenido!");
    }
}

class QueDiaEsHoy implements Operacion {
    @Override
    public String getDescripcion() {
        return "¿Qué día es hoy?";
    }

    @Override
    public void ejecutar() {
        System.out.println("Hoy es domingo.");
    }
}

class ProgramaMejorado {
    private Operacion[] operaciones;

    public ProgramaMejorado(Operacion[] operaciones) {
        this.operaciones = operaciones;
    }

    public void iniciar() {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("Menú:");
            for (int i = 0; i < operaciones.length; i++) {
                System.out.println((i + 1) + ". " + operaciones[i].getDescripcion());
            }
            System.out.println((operaciones.length + 1) + ". Salir");
            System.out.print("Ingrese una opción: ");

            int opcion = scanner.nextInt();
            scanner.nextLine(); // Limpiar el buffer
        }
    }
}

```

```
        if (opcion == operaciones.length + 1) {
            System.out.println("¡Adiós!");
            return;
        } else if (opcion >= 1 && opcion <= operaciones.length)
        { operaciones[opcion - 1].ejecutar();
        } else {
            System.out.println("Opción no válida. Inténtelo de nuevo.");
        }
    }

public static void main(String[] args) {
    Operacion[] operaciones = {new Saludar(), new QueDiaEsHoy()};
    ProgramaMejorado programa = new ProgramaMejorado(operaciones);
    programa.iniciar();
}
```

## Herencia de interfaces

La herencia de interfaces en Java es un mecanismo que permite que una interfaz herede los métodos y las constantes de otra interfaz. Esto significa que una interfaz hija puede incluir métodos adicionales además de los métodos heredados de la interfaz padre.

!

En Java NO existe herencia múltiple de clases, pero una clase si puede implementar múltiples interfaces, lo que permite lograr funcionalidades similares a la herencia múltiple.

### Ejemplo

Interfaz Fichero que tenga un método para leer su contenido en forma de String y luego dicha interfaz sea extendida por otra que sea FicheroEscribible que permita enviar contenido e incluso eliminar el fichero.

```
interface Fichero {
    String leerContenido();
}

interface FicheroEscribable extends Fichero {
    void escribirContenido(String contenido);
    void eliminarFichero();
}

class FicheroTexto implements FicheroEscribable {
    private String contenido;

    @Override
    public String leerContenido() {
        return contenido;
    }

    @Override
    public void escribirContenido(String contenido) {
        this.contenido = contenido;
    }

    @Override
    public void eliminarFichero() {
        this.contenido = null;
    }
}

public class Main {
    public static void main(String[] args) {
        FicheroEscribable fichero = new FicheroTexto();
        fichero.escribirContenido("Hola, mundo!");
        System.out.println("Contenido del fichero: " + fichero.leerContenido());
        fichero.eliminarFichero();
        System.out.println("Fichero eliminado.");
    }
}
```

# 6. Genericidad

## ¿Qué es la Programación Genérica?

La "programación genérica" es un paradigma de programación que permite escribir algoritmos y estructuras de datos que pueden trabajar con tipos de datos variables, sin especificar el tipo de datos concreto en el momento de escribir el código. En lugar de eso, los tipos se especifican más adelante, cuando se utilizan las estructuras o algoritmos.

## Parámetros de tipo

Los "parámetros de tipo", también conocidos como "tipos genéricos" o "argumentos de tipo", son una característica de algunos lenguajes de programación que permiten escribir código que pueda operar sobre tipos de datos específicos sin especificar esos tipos de datos de antemano.

En otras palabras, los parámetros de tipo son placeholders o marcadores que representan tipos de datos reales que se pueden proporcionar más tarde al utilizar una clase, método o estructura de datos genérica.

### Ejemplo

uso de tipos genéricos en Java

```
public class Lista<T> {
    private T[] elementos;

    public Lista(int capacidad) {
        this.elementos = (T[]) new Object[capacidad];
    }

    public void agregar(T elemento) {
        // Implementación para agregar un elemento a la lista
    }

    // Otros métodos...
}
```

En este ejemplo, **T** es un parámetro de tipo que representa el tipo de elementos que contendrá la lista. Al crear una instancia de **Lista**, puedes especificar el tipo de elementos que deseas almacenar, por ejemplo:

```
Lista<Integer> listaDeEnteros = new Lista<>(10);
Lista<String> listaDeCadenas = new Lista<>(20);
```

En este caso, **Integer** y **String** son los tipos que se proporcionan como argumentos de tipo para los parámetros de tipo **T**

## ¿Por qué usar generics?

### Reutilización del código

Podemos escribir un método/clase/interfaz una vez y usarlo para cualquier tipo de dato que queramos.

### Seguridad de tipos

Los genéricos hacen que los errores aparezcan en tiempo de compilación en vez de en tiempo de ejecución (Siempre es mejor conocer los problemas en tu código en tiempo de compilación que esperar a que tu código falle en tiempo de ejecución).

Supongamos que queremos crear un ArrayList que almacene los nombres de los alumnos, y si por error el programador añade un objeto entero en lugar de una cadena, el compilador lo permite. Pero, cuando recuperamos estos datos de ArrayList, causa problemas en tiempo de ejecución.

Además, una vez generalizamos un código podemos hacer que el parámetro genérico herede de un tipo, por ejemplo. Si tenemos la clase Punto<T> y queremos que los puntos contengan números podemos obligar a que dicho tipo sea de tipo numérico con <T extends number>.

### Evitar casteos individuales

El typecasting se puede volver un dolor de cabeza a la hora de recorrer estructuras, con generics podemos (en muchas ocasiones) evitar el casteo de tipos

#### ejemplo

Objetivo: evitar typecasting mediante el uso de generics

```
public interface Punto{  
    public double distanciaA(Punto p);  
}  
  
public class Punto2D implements Punto{  
    private double x,y;  
  
    //constructor  
    .  
    .  
    .  
  
    @override  
    public double distanciaA(Punto P){  
        if (p instance of Punto2D){  
            Punto2D p2d = (Punto2D) p;  
            return ...  
        } else {  
            throw new IllegalArgumentException();  
        }  
    }  
}
```

pasándolo a generics sería:

```
public interface Punto<P extends number>{  
    public double distanciaA(P p);  
}  
  
public class Punto2D implements Punto<Punto2D>{  
    @override  
    public double distanciaA(Punto2D O){  
        return ...  
    }  
}
```

# 7. Aspectos Funcionales

## ¿Qué es la Programación Funcional?

La "funcional" es un paradigma de programación basado en la estructura de las funciones matemáticas, utiliza expresiones lambda y permite declarar expresiones que se evaluarán como funciones. Por medio de las expresiones lambda vamos a poder hacer uso de la inferencia de tipos o más bien dentro de las expresiones lambda vamos a poder prescindir de declarar los tipos de datos y vamos a dejar ese trabajo al compilador por medio de la inferencia de tipos.

## Expresión lambda

Una expresión lambda es un bloque de código que puedes "guardar" de manera que puede ser pasado como parámetro para ser ejecutado más tarde.

Una expresión lambda representa el método abstracto de una interfaz funcional, está caracterizada o su sintaxis es:

parámetro → cuerpo de la expresión lambda.

### Ejemplo:

```
// Clase principal
public class lambdaFunction {

    // Se declara la interface
    interface operacion {
        // el metodo abstracto
        public double suma(double x, double y);
    }

    public static void main(String[] args) {
        // Expresion lambda
        operacion l = (x, y) -> x + y;
        System.out.println(l.suma(8, 30));
    }
}
```

## Interfaces funcionales

Una interfaz funcional es aquella que solo tiene un método abstracto, podemos utilizar métodos default, métodos estáticos y métodos heredados de la clase object y declararlos como métodos abstractos.

```
@FunctionalInterface  
public interface Runnable{  
    public void run();  
}
```

Si la interfaz que estamos declarando contiene la Anotación `@FunctionalInterface` y esta no cumple con los criterios para que sea una interfaz funcional nos dará un error de compilación, esto nos ayuda y es una buena práctica para desarrollar correctamente. Cabe mencionar que una interface con un solo método abstracto como lo hemos comentado sigue siendo una interfaz funcional aunque no tenga la Anotación `@FunctionalInterface`.

## **Consumer<T>**

Representa una operación que acepta un solo argumento de tipo `T` y no devuelve ningún resultado.

```
import java.util.function.Consumer;  
  
public class ConsumerExample {  
    public static void main(String[] args) {  
        Consumer<String> consumer = s -> System.out.println("Mensaje: " + s);  
        consumer.accept("Hola mundo");  
    }  
}
```

El método `.accept(T t)` Se utiliza para realizar acciones secundarias basadas en el valor de entrada, como imprimir mensajes en la consola, modificar estados de variables o realizar cualquier otra operación que no implique devolver un valor específico.

## **Supplier<T>**

Representa un “proveedor” que no toma ningún argumento y produce un resultado de tipo `T`

```
import java.util.function.Supplier;  
  
public class SupplierExample {  
    public static void main(String[] args) {  
        Supplier<String> supplier = () -> "Hola mundo";  
        String resultado = supplier.get();  
        System.out.println(resultado);  
    }  
}
```

El método `.get()` Se utiliza para obtener un valor suministrado por el **Supplier**

## **Function<T, R>**

Representa una función que acepta un argumento de tipo T y produce un resultado de tipo R

```
import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {
        Function<Integer, Integer> funcion = x -> x * 2;
        int resultado = funcion.apply(5);
        System.out.println(resultado);
    }
}
```

El método .apply(T t) Se utiliza para aplicar una función sobre un valor de entrada y obtener un resultado, como por ejemplo realizar operaciones con el valor de entrada

## **Predicate<T>**

Representa un predicado (una condición) que toma un argumento de tipo T y devuelve un valor booleano.

```
import java.util.function.Predicate;

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> esPar = x -> x % 2 == 0;
        System.out.println(esPar.test(4)); // Devuelve true
        System.out.println(esPar.test(5)); // Devuelve false
    }
}
```

El método .test(T t) Se utiliza para probar si el valor de entrada satisface cierta condición

## **Closures**

Una "closure" es un concepto en programación que se refiere a una función que captura variables del ámbito que la rodea. Esto significa que la función puede acceder y utilizar variables que están definidas fuera de su propio ámbito léxico.

Lo que hace especial a una "closure" es su capacidad para capturar y retener el entorno en el que fue creada, incluso después de que el ámbito original haya finalizado su ejecución. Esto significa que una "closure" puede hacer referencia a variables y parámetros de una función externa, y mantener una conexión con ellos incluso cuando esa función ha terminado de ejecutarse.

## ejemplo

```
public class EjemploCLlosure{  
    public static void main(String[] args) {  
        int x = 10; // Variable local  
        // La expresión lambda captura la variable local 'x'  
        Runnable runnable = () -> System.out.println("El valor de x es: " + x);  
        runnable.run();  
    }  
}
```

En este ejemplo, la expresión lambda captura la variable local **x**. Aunque **x** está definida en el método **main()** y su ámbito termina cuando el método **main()** finaliza, la expresión lambda puede seguir haciendo referencia a **x** debido a la "closure".

## ejemplo 2

```
public class EjemploClosure2 {  
    public static void main(String[] args) {  
        // Método que recibe una función y la ejecuta  
        execute(() -> System.out.println("Mensaje desde la función anónima"));  
    }  
  
    public static void execute(Runnable runnable) {  
        System.out.println("Ejecutando...");  
        runnable.run();  
    }  
}
```

En este ejemplo, la función **execute()** toma una interfaz funcional **Runnable** como argumento. Cuando se llama a **execute()**, se pasa una función anónima que captura el comportamiento a ejecutar. La "closure" permite que esta función anónima acceda al contexto en el que fue creada, incluso después de haber salido de la función **main()**.

El segundo ejemplo imprimiría lo siguiente por consola:

```
Ejecutando...  
Mensaje desde la función anónima
```

Este resultado se debe a que el método **execute()** imprime "Ejecutando..." antes de ejecutar la función **run()** de la interfaz **Runnable** que se le pasa como argumento. La función anónima pasada como argumento a **execute()** simplemente imprime "Mensaje desde la función anónima" cuando se ejecuta.

## Métodos referenciados

La expresión lambda puede llamar a un método que ya existe, en estos casos podría ser más claro referirnos al método que ya existe por su nombre.

Es por eso que, cuando queremos utilizar una expresión lambda que solo implica a un método, podemos hacerlo con `::` de modo que el compilador generará una instancia de la interfaz funcional, sobrescribiendo el método abstracto de la interfaz para llamar al método dado.

Una expresión lambda solo podrá ser reescrita como una referencia a un método si la expresión lambda llama a un único método y no hace ninguna otra cosa.

!!

`s -> s.length() == 0` No puedes usar referencia a método porque hay una única llamada a la función `length()` pero tambien compara el resultado con 0

## Referencia a métodos

### Referencia a constructores

#### ejemplos

Método Referenciado	Expresión lambda Equivalente	Notas
<code>separator::equals</code>	<code>x -&gt; separator.equals(x)</code>	es una expresión con un objeto y una instancia de método
<code>String::trim</code>	<code>X -&gt; x.strip()</code>	es una expresión con una clase y una instancia de método
<code>String::contact</code>	<code>(x,y) -&gt; x.contact(y)</code>	el primer parametro de lambda se convierte en un parámetro implícito y el resto de parámetros son pasados al método
<code>Integer::valueOf</code>	<code>x -&gt; Integer.valueOf(x)</code>	es una expresión con un método estático, el parametro de lambda se pasa al método estático
<code>Integer::sum</code>	<code>(x,y) -&gt; Integer.sum(x,y)</code>	es una expresión con un método estático, pero esta vez con dos parámetros, los dos parámetros de lambda se pasan al método estático
<code>String::new</code>	<code>x -&gt; new String(x)</code>	es una referencia al constructor, los parámetros de lambda se pasan al constructor
<code>String[]::new</code>	<code>n -&gt; new String[n]</code>	es una referencia al constructor de un array, la lambda pasa a ser el <code>length</code> del array

# Ejercicios Tipo Examen del Bloque II

## Contenidos

-  [4.1 Composición →](#)
-  [4.2 Herencia →](#)
-  [5. Polimorfismo →](#)
-  [6. Genericidad →](#)
-  [7. Aspectos Funcionales →](#)

## [4.1 Composición →](#)

### Composición fuerte

- unidireccional Receta → Ingredientes
- multiplicidad

|      1 Receta → \* Ingredientes  
      1 Ingrediente → 1 Receta

```

Class IngredienteEnReceta{
    private String ingrediente;
    private int cantidad;

    //Constructor
    public IngredienteEnReceta(String ingrediente, int cantidad){
        this.ingrediente = ingrediente;
        this.cantidad = cantidad;
    }

    //Getters
    public String getIngrediente(){
        return this.ingrediente;
    }

    public int getCantidad(){
        return this.cantidad;
    }
}

Class Receta{
    private String nombre;
    private List<IngredienteEnReceta> ingredientes = new ArrayList<>();

    //...

    public void anadirIngredienteAReceta(String ing, int cant){
        ingredientes.add(new IngredienteEnReceta(ing,cant))
    }
}

```

## Composición débil

- unidireccional Grupo → Alumnos
  - multiplicidad
- 1 Grupo → \* Alumnos  
1 Alumno → 1 Grupo
- invariante: ["No añadir al mismo alumno más de una vez"]

```

class Alumno{
/...
}

Class Grupo{
    private List<Alumno> inscritos = new ArrayList<>();

    public void inscribir(Alumno alumno){
        if (this.inscritos.contains(alumno)){
            throw new IllegalArgumentException("Alumno ya inscrito");
        }
        this.inscritos.add(alumno);
    }
}

```

## Composición fuerte

- unidireccional Universidad → Inscripción
- multiplicidad
  - 1 Universidad → \* Inscripciones
  - 1 Inscripción → 1 Universidad
- invariante: ["No puede inscribirse un Alumno en un grupo si colisiona en horario"]
- Hora y DiaSemana → @overrides equals

```

class Inscripcion{
    Alumno alumno;
    Grupo grupo;
    /...
}

class Universidad{
    private List<Grupo> grupos;
    private List<Alumno> alumnos;
    private List<Inscripcion> inscripciones;

    public void inscribir(Alumno a, Grupo g){
        for(Inscripcion inscripcion : inscripciones){
            if (inscripcion.getAlumno().equals(a) &&
                inscripcion.getGrupo().getDia().equals(g.getDia()) &&
                inscripcion.getGrupo().getHora().equals(g.getHora())){
                throw new IllegalArgumentException("colisión de horario");
            }
        }
        this.inscripciones.add(New Inscripcion(a,g));
    }
}

```

## **4.2 Herencia → && 5. Polimorfismo →**

- Jerarquía:
  - Alumno es abstracto con atributo DNI : String
  - AlumnoUniversitario con atributo Curso : int
  - AlumnoInstituto con atributo OrientacionTecnologica : boolean

```

public abstract class Alumno {
    private String DNI;

    public Alumno(String DNI) {
        this.DNI = DNI;
    }

    public String getDNI() {
        return DNI;
    }

    public abstract void estudiar(); // Metodo abstracto
    // (OBLIGATORIO IMPLEMENTARLO EN LAS CLASES QUE EXIENDAN DE ALUMNO)
}

public class AlumnoUniversitario extends Alumno {
    private int curso;

    public AlumnoUniversitario(String DNI, int curso) {
        super(DNI);
        this.curso = curso;
    }

    public int getCurso() {
        return curso;
    }

    @Override
    public void estudiar() {
        System.out.println("El alumno universitario está estudiando para sus exámenes.");
    }
}

public class AlumnoInstituto extends Alumno {
    private boolean OrientacionTecnologica;

    public AlumnoInstituto(String DNI, boolean orientacionTecnologica) {
        super(DNI);
        this.OrientacionTecnologica = orientacionTecnologica;
    }

    public boolean isOrientacionTecnologica() {
        return OrientacionTecnologica;
    }

    @Override
    public void estudiar() {
        System.out.println("El alumno de instituto está estudiando para sus exámenes.");
    }
}

```

```
    }  
}
```



```
Alumno mialumno = new Alumno(); //genera error
```

Para acceder a la clase abstracta, ésta debe ser heredada de otra clase.

AlumnoUniversitario extends Alumno y a esta si se puede acceder.



```
AlumnoUniversitario mialumno = new AlumnoUniversitario();
```

## 5. Polimorfismo →

Objetivo: Se pretende quitar las estructuras if de la interfaz “guardableEnFichero”

```
public void guardarEnFichero(List<Alumno> alumnos, Fichero fichero){  
    for (Alumno alumno : alumnos){  
        fichero.escribe(alumno.getDNI());  
        if (alumno instanceof AlumnoUniversitario){  
            AlumnoUniversitario alumnoU = (AlumnoUniversitario) alumno;  
            fichero.escribe(alumno.getCurso());  
        } else if (alumno instanceof AlumnoInstituto){  
            AlumnoInstituto alumnoI = (AlumnoInstituto) alumno;  
            fichero.escribe(alumno.isOrdenacionTecnologica());  
        }  
    }  
}
```

```

public interface GuardableEnFichero{
    void guardar(Fichero f);
}

public abstract class Alumno implements GuardableEnFichero{
    private String DNI;
    /...
    @override
    public void guardar(Fichero f){
        f.escribe(DNI);
    }
}

public class AlumnoUniversitario extends Alumno{
    private int curso;
    /...
    @override
    public void guardar(Fichero f){
        super.guardar(f);
        f.escribe(curso);
    }
}

public class AlumnoInstituto extends Alumno{
    private boolean OrientacionTecnologica;
    /...
    @override
    public void guardar(Fichero f){
        super.guardar(f);
        f.escribe(OrientacionTecnologica);
    }
}

```

```

public void guardarEnFichero(List<Alumno> alumnos, Fichero fichero){
    for (Alumno alumno : alumnos){
        alumno.guardar(fichero);
    }
}

```

## 6. Genericidad →

Objetivo: usar generics para poder restringir el tipo de Alumno y evitar downcasting

```

Class ParejaConcurso{
    private Alumno MiembroPrimero;
    private Alumno MiembroSegundo;

    public ParejaConcurso(Alumno a1, Alumno a2){
        //...
    }
    //getters...

    //Main()...
    Alumno a1 = new AlumnoUniversitario(77,1);
    Alumno a2 = new AlumnoUniversitario(88,2);

    ParejaConcurso pareja = new ParejaConcurso(a1,a2);
    /...
    AlumnoUniversitario a1 = (AlumnoUniversitario) pareja.getPrimero();
    sout(a1.getCurso());
}

```

```

Class ParejaConcurso<E extends Alumno> {
    private E MiembroPrimero;
    private E MiembroSegundo;

    public ParejaConcurso(E a1, E a2){
        //...
    }
    //getters...

    //Main()...
    Alumno a1 = new AlumnoUniversitario(77,1);
    Alumno a2 = new AlumnoUniversitario(88,2);

    ParejaConcurso<AlumnoUniversitario> pareja = new ParejaConcurso<>(a1,a2);
    /...
    AlumnoUniversitario a1 = pareja.getPrimero();
    sout(a1.getCurso());
}

```

## 7. Aspectos Funcionales →

### **forEach**

Objetivo: Haz un forEach(Consumer <? super E> action) en el siguiente código

```
public List<Grupo> GrupoDe(Alumno a){  
    List<Grupo> grupoDe = new ArrayList<>();  
    List<Inscripcion> inscripciones = this.inscripciones;  
    for (Inscripcion inscripcion : inscripciones){  
        if (getAlumno().equals(a)){  
            grupoDe.add(inscripcion.getGrupo());  
        }  
    }  
    return grupoDe;  
}
```

```
inscripcion.forEach((inscripcion) -> {  
    if(inscripcion.getAlumno().equals(a)){  
        grupoDe.add(inscripcion.getGrupo());  
    }  
});
```

## Implementación de funcional

Objetivo: Dado el siguiente código declara e implementa obtenerGruposQue...

```
Universidad uni = new Universidad();  
List <Grupo> grupoDeTarde = uni.obtenerGrupoQue(  
(grupo) -> { grupo.getHora().getHora()>=16 });
```

```
public List<Grupo> obtenerGruposQue(Predicate<grupo> filtro){  
    List<Grupo> gruposQue;  
    for (grupo g : this.grupos){  
        if(filtro.test(g)){  
            gruposQue.add(g);  
        }  
    }  
    return gruposQue;  
}
```

# **Práctica de XML (incluye soluciones del profesorado):**

En la clase del objeto:

```

public class contacto{
    // atributos etiquetas
    public static final String ETQ_CONTACTO = "Contacto";
    public static final String ETQ_NOMBRE = "Nombre";
    public static final String ETQ_EMAIL = "EMAIL";
    public static final String ETQ_TELEF = "Telefono";

    // atributos
    private final String nombre;
    private final String email ;
    private final int telefono;

    // constructor
    public Contacto(String nombre, String email, int telefono){
        this.nombre = nombre;
        this.email = email;
        this.telefono = telefono;
    }

    // constructor Elemento -> PARA LECTURA
    public Contacto(Element e) throws ParsingException {
        //Asignar Element a las etiquetas
        Element eltoNombre = e.getFirstChildElement(ETQ_NOMBRE);
        Element eltoEmail = e.getFirstChildElement(ETQ_EMAIL);
        Element eltoTelef = e.getFirstChildElement(ETQ_TELEF);

        // Comprobar que la etiqueta no este vacía
        if(eltoNombre == null){
            throw new ParsingException("Falta el Nombre en el elemento Contacto");
        }
        if(eltoEmail == null){
            throw new ParsingException("Falta el Email en el elemento Contacto");
        }
        if(eltoTelef == null){
            throw new ParsingException("Falta el Telefono en el elemento Contacto");
        }

        // Asignar etiquetas a atributos de la clase
        this.nombre = eltoNombre.getValue().trim();
        this.email = eltoEmail.getValue().trim();
        try {
            this.telefono = Integer.ParseInt(eltoTelef.getValue().Trim());
        } catch (NumberFormatException e) {
            throw new ParsingException("Valor incorrecto para el telefono");
        }
    } // fin del contructor Elemento

    // toDOM -> PARA ESCRITURA

    public Element toDOM() {
        // Crear los Element
        Element raiz = new Element(ETQ_CONTACTO);
        Element eltoNombre = new Element(ETQ_NOMBRE);

```

```
Element eltoEmail = new Element(ETQ_EMAIL);
Element eltoTelef = new Element(ETQ_TELEF);

// Enlazar elementos y atributos
eltoTitulo.appendChild(nombre);
eltoAutor.appendChild(email);
eltoAnho.appendChild(Integer.toString(telefono));

// Enlazar Elemento Raiz con resto
raiz.appendChild(eltoNombre);
raiz.appendChild(eltoEmail);
raiz.appendChild(eltoTelef);

return raiz;
}
```

Clase contenedor:

```

/** Carga una agenda desde XML
 * @param nf la ruta del fichero XML, como String
 */
public Agenda(String nf) throws ParsingException, IOException {
    this();

    Builder parser = new Builder();
    Document doc = parser.build( new File( nf ) );

    Elements contactos = doc.getRootElement().getChildElements();

    for(int i = 0; i < contactos.size(); i++) {
        this.inserta(new Contacto(contactos.get(i)));
    }
}

/** construye el DOM a partir de los objetos almacenados en la agenda
 * @return nodo raiz, de tipo Element
 */
public Element toDom(){
    Element toRet = new Element(ETQ_AGENDA);

    for(int i = 0; i < getNumContactos();i++){
        toRet.appendChild(agenda.get(i).toDOM());
    }
    return toRet;
}

/** Almacena en un fichero XML los contactos de la agenda
 * @param nf ruta del fichero, como un String
 * @throws IOException
 */
public void toXML(String nf) throws IOException
{
    FileOutputStream f = new FileOutputStream( nf );
    Serializer serial = new Serializer( f );
    Document doc = new Document( this.toDom() );
    serial.write(doc);
}

```