UD10 – UTILIZACIÓN AVANZADA DE CLASES

Contenido

1 Definición de cuerpos de expresión

Para la definición de constructores, propiedades y métodos de clases podemos utilizar una notación compacta llamada definición de cuerpos de expresión.

En **Java**, no existe exactamente una notación de definición de cuerpos de expresión como en **C#**. Sin embargo, podemos lograr un comportamiento equivalente utilizando características como **expresiones lambda**, **interfaces funcionales**, y la implementación directa de métodos compactos.

Podremos utilizarla siempre que la definición del método se componga de una única expresión. Esta notación resulta más compleja, pero al ser más compacta ahorrará código y por lo tanto tiempo de desarrollo. Una vez acostumbrados resultará bastante legible.

Veamos un ejemplo; tenemos la siguiente clase *Circunferencia*

```
public class Circunferencia {

    // Campo radio
    private double radio;

    // Propiedad Radio con cuerpo compacto (get y set)
    public double getRadio() {
        return radio; // Propiedad compacta (Java requiere return y llaves)
    }

    public void setRadio(double radio) {
        this.radio = radio; // Compacto para setters
    }

    // Constructor vacío
    public Circunferencia() {
        // Sin lógica adicional, este constructor es trivial
    }

    // Constructor con el radio, simulando compactación
    public Circunferencia(double radio) {
        this.radio = radio; // Constructor inicializa radio
    }

    // Método CalcularCircunferencia
    public double calcularCircunferencia() {
        return 2 * Math.PI * radio; // Método de una línea
    }

    // Método CalcularArea
    public double calcularArea() {
        return Math.PI * radio * radio; // Método de una línea
    }
}
```

1. **Compactación**:
   - Aunque Java no tiene la notación `=>` de C#, todos los métodos se pueden escribir en una sola línea cuando consisten en una única expresión.
   - Es obligatorio usar `return` y `{}` para métodos regulares.
2. **Limitaciones**:

- Java no elimina la necesidad de palabras clave como `return` en métodos.
- Los constructores y propiedades requieren bloques explícitos incluso cuando son simples.

3. **Legibilidad**:
   - Aunque la notación no es tan compacta como en C#, sigue siendo clara y directa para métodos que realizan una sola operación.

2 Encapsulación y visibilidad

El paradigma de la OOP se basa en el concepto de objeto. Un objeto es aquello que tiene estado (propiedades más valores), comportamiento (acciones y reacciones a mensajes) e identidad (propiedad que lo distingue de los demás objetos). La estructura y comportamiento de objetos de la misma clase están definidos en ésta; los términos instancia y objeto son intercambiables.

Los términos clase y objeto se usan a veces indistintamente, pero, en realidad, las clases describen el tipo de los objetos, mientras que los objetos son instancias de clases que se pueden usar. Así, la acción de crear un objeto se denomina creación de instancias.

Los principios fundamentales de la 00P son: abstracción, encapsulación, modularidad, jerarquía y polimorfismo. Iremos abordando estos principios a lo largo de la unidad.

Abstracción. Se trata de centrarse en las características esenciales de un objeto capturando sus comportamientos. Es decir, se trata de definir los métodos disponibles en el objeto y los parámetros que necesitamos proporcionar a dicho objeto para que lleve a cabo una operación específica. De esta forma, no necesitamos conocer ni comprender cómo están implementados estos métodos ni qué acciones tiene que llevar a cabo para obtener el resultado esperado, lo único que necesitamos conocer es su interfaz pública y no los detalles de su implementación. Por ejemplo, no necesitamos conocer los detalles de implementación de los protocolos de servicios web para abrir un determinado recurso en la red, lo único que necesitamos conocer es su dirección (url), introducimos dicha dirección en el navegador y accedemos a dicho recurso.

Encapsulación. La encapsulación es el proceso de ocultar todos los detalles de un objeto que no contribuyen a sus características esenciales, es decir, separar el aspecto externo del objeto accesible por otros objetos, del aspecto interno del mismo que será inaccesible para los demás. O lo que es lo mismo: la encapsulación consiste en ocultar los atributos y métodos del objeto a otros objetos, estos no deben estar expuestos a los objetos exteriores. Una vez encapsulados, pasan a denominarse atributos y métodos privados del objeto. Por ejemplo, cuando definimos un atributo en una clase como privado y generamos los getter y setter estamos encapsulado dicho atributo.

Ejemplo 1. Ventajas del uso de la abstracción y encapsulamiento. Vamos a desarrollar un programa en el que necesitamos la generación de números aleatorios. Existen listas de números aleatorios que permiten simular distintas condiciones antes las que nuestro sistema debería responder. Para ello vamos a hacer uso de la clase *Random*. Para poder utilizar números aleatorios en nuestro programa sólo necesitamos instanciar un objeto de esta clase y llamar a sus métodos. Al encapsular el código interno de la clase y presentar únicamente una sencilla interfaz pública podemos abstraernos de la complejidad del proceso de obtención de números aleatorios y utilizarlos fácilmente en nuestro programa.

```
import java.util.Random;
import java.util.Arrays;


public class TestRandom {
    public static void testRandom() {
        // Instanciamos un objeto de la clase Random
        Random r = new Random();


        // Creamos un array de 16 bytes
        byte[] sequence = new byte[16];


        // Obtenemos una secuencia de 16 números aleatorios
        r.nextBytes(sequence);


        // Mostramos la secuencia obtenida
        System.out.println(Arrays.toString(sequence));
    }


    public static void main(String[] args) {
        testRandom();
    }
}
```

Ejemplo 2. Esfera. Vamos a crear una clase Esfera que permitirá obtener la superficie y el volumen de una esfera a partir del radio. Encapsularemos una variable privada que contendrá el radio y accederemos a ella a través de una propiedad lo que permitirá validar que el radio establecido es correcto. Abstraemos de esta forma el objeto esfera. Podríamos incluir comportamientos adicionales que modelen propiedades que necesitáramos relativos a una esfera; diámetro, generatriz, etc.

```
public class Esfera {


    // Atributo radio, encapsulado mediante getter y setter
    private double radio;


    // Getter y Setter para la propiedad radio
    public double getRadio() {
        return radio;
    }


    public void setRadio(double radio) {
        // Verificar que el valor recibido es un número real positivo
        if (Double.isNaN(radio) || Double.isInfinite(radio) || radio < 0) {
            throw new IllegalArgumentException("El radio debe ser un número real
positivo");
        }
        this.radio = radio;
    }


    // Propiedad que obtiene el área de la esfera
    public double getArea() {
        return 4.0 * Math.PI * radio * radio;
    }


    // Propiedad que obtiene el volumen de la esfera
    public double getVolumen() {
        return (4.0 / 3.0) * Math.PI * radio * radio * radio;
    }
}
```

```
public class TestEsfera {


    public static void testEsfera() {

        // Crear una instancia de Esfera y establecer el radio

        Esfera e = new Esfera();

        e.setRadio(10);


        // Mostrar los valores en formato con 2 decimales

        System.out.printf("Esfera. Radio = %.2f, Superficie = %.2f, Volumen =
%.2f%n",

                e.getRadio(), e.getArea(), e.getVolumen());

    }


    public static void main(String[] args) {

        testEsfera();

    }

}
```

Todas las clases y miembros de clase pueden especificar el nivel de acceso que proporcionan a otras clases mediante los *modificadores de acceso*.

Están disponibles los siguientes modificadores de acceso:

Modificador	Descripción
public	Acceso no limitado. Cuallquier clase puede acceder a los miembros públicos
protected	Acceso limitado a esta clase o a las clases derivadas de esta clase
private	Acceso limitado solo a la propia clase
(sin modificador)	Package-private o “default”: acceso limitado a las clases dentro del mismo paquete

Relacionado con los términos abstracción y encapsulamiento aparece otro concepto de la OOP que es la **ocultación**. En programación orientada a objetos el principio de ocultación hace referencia a que los atributos privados de un objeto no pueden ser modificados ni obtenidos a no ser que se haga a través del paso de un mensaje (invocación a métodos o propiedades).

En informática, se conoce como principio de ocultación de información a la ocultación de decisiones de diseño en un programa susceptible de cambios con la idea de proteger a otras partes del código si éstos se producen. Proteger una decisión de diseño supone proporcionar una interfaz estable que proteja el resto del programa de la implementación (susceptible de cambios). En los lenguajes de programación modernos el principio de ocultación de información se manifiesta de diferentes maneras, como por ejemplo la encapsulación.

Ejemplo de encapsulación

```
public class Persona {

    // Atributo privado (ocultación de información)

    private String nombre;

    // Constructor

    public Persona(String nombre) {

        this.nombre = nombre;

    }

    // Método público para obtener el nombre (getter)

    public String getNombre() {

        return nombre;

    }

    // Método público para modificar el nombre (setter)

    public void setNombre(String nombre) {

        this.nombre = nombre;

    }

}
```

3 Modularidad

Modularidad. Es la propiedad de una aplicación o de un sistema que ha sido descompuesto en un conjunto de módulos o partes más pequeñas coherentes e independientes. Estos módulos se pueden compilar por separado, pero tienen conexiones con los otros módulos.

Ya hemos visto este concepto. En el diseño de un programa orientado a objetos, es importante considerar la manera en que organizaremos clases e interfaces. Disponemos de los espacios de nombres (namespaces) y ensamblados para organizar los programas. También hemos visto como crear librerías que después podremos utilizar en diferentes aplicaciones.

Ejemplos de Modularidad:

1. Paquete de operaciones matemáticas com.ejemplo.operaciones

```
package com.ejemplo.operaciones;


public class Suma {
    public int sumar(int a, int b) {
        return a + b;
    }
}
```

2. Paquete de utilidades: com.ejemplo.utilidades

```
package com.ejemplo.utilidades;


public class Impresora {
    public void imprimirResultado(int resultado) {
        System.out.println("Resultado: " + resultado);
    }
}
```

3. Aplicación principal: com.ejemplo.app

```
package com.ejemplo.app;


import com.ejemplo.operaciones.Suma;
import com.ejemplo.utilidades.Impresora;


public class Main {
    public static void main(String[] args) {
        Suma suma = new Suma();
        int resultado = suma.sumar(10, 20);


        Impresora impresora = new Impresora();
        impresora.imprimirResultado(resultado);
    }
}
```

Beneficios de la Modularidad:

- Reutilización de código: Los módulos pueden ser reutilizados en diferentes aplicaciones.
- Mantenimiento: Los módulos hacen que el código sea más fácil de mantener, ya que cada parte está aislada y tiene una responsabilidad bien definida.
- Escalabilidad: Permite que el sistema se amplíe sin problemas, añadiendo nuevos módulos según sea necesario.
- Distribución: Los módulos se pueden empaquetar y distribuir como bibliotecas (JARs) que otras aplicaciones pueden utilizar.

4 Herencia

La herencia en la programación orientada a objetos (OOP) es una de las características fundamentales que permite la creación de nuevas clases basadas en clases existentes, facilitando la reutilización del código y la creación de jerarquías de clases.

Al igual que en **C#**, en **Java**, la **herencia** permite que una clase (subclase) herede atributos y métodos de otra clase (superclase o clase base). Esto permite a las subclases especializar o ampliar el comportamiento de las clases base.

Clases Base y Subclases

- Clase base o superclase: Es la clase original de la que otras clases heredan.
- Clase derivada o subclase: Es la clase que extiende o hereda de la clase base, y puede agregar nuevos comportamientos o sobrescribir (modificar) los comportamientos heredados.

Para **heredar de una clase base** en Java, se usa la palabra clave `extends`. La clase derivada (subclase) hereda todos los métodos y propiedades (atributos) de la clase base, salvo que estos sean privados.

Clase Base:

```
public class Animal {
    // Atributo
    private String nombre;

    // Constructor
    public Animal(String nombre) {
        this.nombre = nombre;
    }

    // Método
    public void hacerSonido() {
        System.out.println("El animal hace un sonido");
    }

    // Getter
    public String getNombre() {
        return nombre;
    }
}
```

Clase Derivada

```
public class Perro extends Animal {
    // Constructor de la subclase
    public Perro(String nombre) {
        // Llamada al constructor de la clase base (superclase)
        super(nombre);
    }

    // Sobrescribimos el método hacerSonido
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra");
    }
}
```

Clase principal (test):

```
public class TestHerencia {
    public static void main(String[] args) {
        // Crear objeto de la clase derivada
        Perro perro = new Perro("Fido");

        // Acceder al atributo heredado de la clase base
        System.out.println("Nombre del perro: " + perro.getNombre());

        // Llamar al método sobrescrito en la subclase
        perro.hacerSonido(); // Salida: El perro ladra
    }
}
```

Beneficios de la Herencia:

- Reutilización de código: Las subclases heredan el comportamiento de las clases base, lo que permite evitar la duplicación de código.
- Jerarquía: La herencia permite organizar las clases en una jerarquía, donde las clases más generales están en la parte superior y las clases más específicas se encuentran en la parte inferior.
- Polimorfismo: Las subclases pueden sobrescribir métodos de la clase base, lo que permite que un objeto de tipo base se comporte de manera diferente según su clase derivada.

4.1 Constructores

Utilizaremos los constructores para crear instancias de las clases. Si no declaramos explícitamente ningún constructor el sistema crea un constructor por defecto que no admite parámetros:

```
public class Persona {
    private boolean activo;
    private String nombre;
    private String apellido;
    // Constructor
    public Persona(String nombre, String apellido) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.activo = false; // Estado por defecto
    }
}
```

```
public class TestPersona {
    public static void main(String[] args) {
        // Crear un objeto Persona usando el constructor
        Persona joe = new Persona("Joe", "Fielding");

        // Establecer el estado de 'activo' si es necesario
        joe.setActivo(true);  // Establecer como activo

        // Imprimir el estado de la persona
        System.out.println(joe.getNombre()
            + (joe.isActivo() ? " " : " no ")
            + "está activo");
    }
}
```

Creamos un constructor que incluya el nombre:

```
public class Persona {

    private boolean activo;

    private String nombre;

    private String apellido;

    // Constructor con nombre y apellido

    public Persona(String nombre, String apellido) {

        this.nombre = nombre;

        this.apellido = apellido;

        this.activo = true; // Se puede establecer 'activo' en true si es
necesario

    }


}
```

Con esta implementación de la clase ya no es posible usar un constructor sin parámetros. Para poder usarlo debemos declarar un constructor sin parámetros, por ejemplo:

```
    // Constructor sin parámetros

    public Persona() {

        this.activo = true; // Inicializa 'activo' a true

    }
```

Ahora ya podemos utilizar el constructor sin parámetros de nuevo:

```
public class TestConstructor {
    public static void main(String[] args) {
        // Crear una persona usando el constructor con un solo parámetro
        Persona joe = new Persona("Joe");
        mostrarPersona(joe);

        // Crear una persona usando el setter para nombre
        Persona mary = new Persona("Mary");
        mostrarPersona(mary);
    }

    private static void mostrarPersona(Persona persona) {
        System.out.println(persona.getNombre()
            + (persona.isActivo() ? " " : " no ")
            + "está activo");
    }
}
```

Observamos que dependiendo del constructor usado el objeto persona se inicializa como activo o como inactivo. Si queremos que ambos constructores inicialicen los objetos de manera coherente podemos llamar desde un constructor al otro. De esta forma podemos cambiar nuestro ejemplo para que en ambos casos las personas se inicialicen como activas:

```
public class Persona {
    private boolean activo;
    private String nombre;
    private String apellido;

    // Constructor sin parámetros
    public Persona() {
        this.activo = true; // Inicializa 'activo' a true
    }

    // Constructor con un parámetro 'nombre'
    public Persona(String nombre) {
        this(); // Llamada al constructor sin parámetros
        this.nombre = nombre; // Inicializa 'nombre'
    }

    // Getter y Setter para 'nombre'
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    // Getter y Setter para 'apellido'
    public String getApellido() {
        return apellido;
    }

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    // Getter para 'activo' (solo lectura)
    public boolean isActivo() {
        return activo;
    }
}
```

Vamos a continuar con el ejemplo creando la clase Empleado que hereda de la clase Persona, incluimos el constructor sin parámetros y constructores con el salario y con el nombre y el salario:

```
import java.math.BigDecimal;


public class Empleado extends Persona {
    private BigDecimal salario;


    // Constructor sin parámetros
    public Empleado() {
        super(); // Llama al constructor de la clase base (Persona)
    }


    // Constructor con un parámetro (salario)
    public Empleado(BigDecimal salario) {
        super(); // Llama al constructor de la clase base (Persona)
        this.salario = salario;
    }


    // Constructor con nombre y salario
    public Empleado(String nombre, BigDecimal salario) {
        super(nombre); // Llama al constructor de la clase base (Persona) con el
nombre
        this.salario = salario;
    }


    // Getter y Setter para 'salario'
    public BigDecimal getSalario() {
        return salario;
    }


    public void setSalario(BigDecimal salario) {
        this.salario = salario;
    }
}
```

Hemos visto cómo utilizar constructores para crear objetos nuevos. Las clases pueden tener también un constructor estático para inicializar los valores de los campos estáticos de la clase. Los constructores estáticos no tienen parámetros. El siguiente ejemplo muestra cómo inicializar un campo estático con un constructor estático:

```
// Declaramos la constante estática SalarioMinimo
    public static final BigDecimal SALARIO_MINIMO;


    // Bloque de inicialización estática
    static {
        SALARIO_MINIMO = new BigDecimal("13300"); // Inicializamos la constante
    }
```

4.2 Jerarquía

La **herencia** permite crear clases nuevas que reutilizan, extienden y modifican el comportamiento que se define en otras clases. En Java, una clase derivada solo puede tener una clase base directa.

La herencia establece una jerarquía entre las clases; la herencia es transitiva. Si ClaseC se deriva de ClaseB y ClaseB se deriva de ClaseA, ClaseC hereda los miembros declarados en ClaseB y ClaseA

En el siguiente ejemplo Cualificado hereda de Empleado que a su vez hereda de Persona, Cualificado adquiere el comportamiento de ambas clases Persona y Empleado.

```
import java.math.BigDecimal;


public class Cualificado extends Empleado {

    private BigDecimal bonus;

    private String titulo;

    private String departamento;

    // Constructor sin parámetros

    public Cualificado() {

        super(); // Llama al constructor sin parámetros de la clase base
(Empleado)

    }

    // Constructor con nombre

    public Cualificado(String nombre) {

        super(nombre); // Llama al constructor de la clase base (Empleado) con el
nombre

    }

    // Constructor con nombre y salario

    public Cualificado(String nombre, BigDecimal salario) {

        super(nombre, salario); // Llama al constructor de la clase base
(Empleado) con el nombre y salario

    }

    // Getters y Setters

    public BigDecimal getBonus() {

        return bonus;

    }

    public void setBonus(BigDecimal bonus) {

        this.bonus = bonus;

    }

    public String getTitulo() {

        return titulo;

    }

    public void setTitulo(String titulo) {

        this.titulo = titulo;

    }

    public String getDepartamento() {

        return departamento;

    }

    public void setDepartamento(String departamento) {

        this.departamento = departamento;

    }

}
```

```
import java.math.BigDecimal;


public class TestConstructor {
    public static void main(String[] args) {
        // Crear una instancia de Persona
        Persona joe = new Persona("Joe");
        mostrarPersona(joe);

        // Crear una instancia de Empleado con nombre y salario
        Empleado matt = new Empleado("Matt", new BigDecimal("25000"));
        mostrarPersona(matt);

        // Crear una instancia de Cualificado con nombre y salario
        Cualificado ann = new Cualificado("Ann", new BigDecimal("45000"));
        mostrarPersona(ann);
    }

    private static void mostrarPersona(Persona persona) {
        // Mostrar el nombre y si la persona está activa
        System.out.println(persona.getNombre()
                + (persona.getActivo() ? " " : " no ")
                + "está activo");
    }
}
```

5 Polimorfismo

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después de la encapsulación y la herencia. Polimorfismo es una palabra griega que significa "con muchas formas" y tiene dos aspectos diferentes:

En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos, listas, o arreglos. En estos casos, el tipo declarado del objeto no es idéntico a su tipo real en tiempo de ejecución. Esto se logra en Java utilizando referencias polimórficas.

Las clases base pueden definir e implementar métodos que las clases derivadas pueden sobrescribir mediante la anotación @Override. En tiempo de ejecución, cuando el código llama a un método, la JVM busca el tipo en tiempo de ejecución del objeto e invoca la implementación correspondiente del método sobrescrito. Esto permite que en el código fuente se invoque un método en una referencia de la clase base y que se ejecute una versión definida en la clase derivada.

Los métodos sobrescritos permiten trabajar con grupos de objetos relacionados de manera uniforme. Por ejemplo, retomando una jerarquía de objetos como Persona -> Empleado -> Cualificado, podemos añadir otra rama, como Persona -> ContactoCliente. Supongamos que tenemos una aplicación de gestión que permite al usuario crear varios tipos de personas. En tiempo de compilación, no se sabe qué tipos específicos de Persona creará el usuario. Sin embargo, la aplicación necesita realizar un seguimiento de los objetos creados y actualizarlos en respuesta a las acciones del usuario. El polimorfismo permite gestionar esta situación de la siguiente forma:

1. Se crea una jerarquía de clases en la que cada clase específica deriva de una clase base común.
2. Se utiliza un método sobrescrito (@Override) para invocar el método apropiado en una clase derivada mediante una sola llamada al método de la clase base.

```
// Clase base común
class Persona {
    public void actualizar() {
        System.out.println("Actualizando información de Persona.");
    }
}

// Clase derivada 1
class Empleado extends Persona {
    @Override
    public void actualizar() {
        System.out.println("Actualizando información de Empleado.");
    }
}

// Clase derivada 2
class ContactoCliente extends Persona {
    @Override
    public void actualizar() {
        System.out.println("Actualizando información de ContactoCliente.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Uso polimórfico
        Persona[] personas = new Persona[3];
        personas[0] = new Persona();
        personas[1] = new Empleado();
        personas[2] = new ContactoCliente();

        // Invocación polimórfica
        for (Persona persona : personas) {
            persona.actualizar(); // Llama al método correspondiente según el tipo
en tiempo de ejecución
        }
    }
}
```

Para acceder a los miembros de la clase padre, utilizaremos la palabra clave super seguida de un punto y el nombre del miembro al que queremos acceder. Ejemplo:

```
class Persona {
    public void mostrarInfo() {
        System.out.println("Información desde la clase Persona.");
    }
}

class Empleado extends Persona {
    @Override
    public void mostrarInfo() {
        // Llamada al método de la clase padre
        super.mostrarInfo();
        System.out.println("Información adicional desde la clase Empleado.");
    }
}

public class Main {
    public static void main(String[] args) {
        Empleado empleado = new Empleado();
        empleado.mostrarInfo();
        // Salida:
        // Información desde la clase Persona.
        // Información adicional desde la clase Empleado.
    }
}
```

De manera similar, super también puede ser utilizado para acceder a constructores o atributos de la clase padre:

```
class Persona {
    public Persona(String nombre) {
        System.out.println("Constructor de Persona: " + nombre);
    }
}

class Empleado extends Persona {
    public Empleado(String nombre, String puesto) {
        super(nombre); // Llama al constructor de la clase padre
        System.out.println("Constructor de Empleado: " + puesto);
    }
}

public class Main {
    public static void main(String[] args) {
        Empleado empleado = new Empleado("Juan", "Gerente");
        // Salida:
        // Constructor de Persona: Juan
        // Constructor de Empleado: Gerente
    }
}
```

Impedir la herencia de una clase completa

Se utiliza el modificador final en la declaración de la clase para impedir que otras clases puedan heredarla.

```
final class Persona {
    public void mostrarInfo() {
        System.out.println("Información desde la clase Persona.");
    }
}

// Esto provocará un error de compilación
// class Empleado extends Persona {
// }
```

La palabra clave final en la definición de la clase asegura que ninguna clase puede derivar de esta clase.

Impedir la sobrescritura de miembros

El modificador final también puede aplicarse a métodos para evitar que sean sobrescritos en clases derivadas.

```
class Persona {
    public final void mostrarInfo() {
        System.out.println("Información desde la clase Persona.");
    }
}

class Empleado extends Persona {
    // Esto provocará un error de compilación
    // @Override
    // public void mostrarInfo() {
    //     System.out.println("Información desde la clase Empleado.");
    // }
}
```

La palabra clave `final` en el método `mostrarInfo` asegura que las clases derivadas no puedan sobrescribirlo.

Clases y métodos sellados (sealed) en Java 17 o superior

Desde Java 17, se introdujo el concepto de clases y métodos sellados mediante la palabra clave sealed. Esto permite controlar explícitamente qué clases pueden heredar de una clase base.

```
public sealed class Persona permits Empleado, Cliente {
    public void mostrarInfo() {
        System.out.println("Información desde la clase Persona.");
    }
}

final class Empleado extends Persona {
    @Override
    public void mostrarInfo() {
        System.out.println("Información desde la clase Empleado.");
    }
}

final class Cliente extends Persona {
    @Override
    public void mostrarInfo() {
        System.out.println("Información desde la clase Cliente.");
    }
}
```

6 Clases y métodos abstractos

En Java, puedes crear una clase abstracta y métodos abstractos utilizando la palabra clave abstract. Una clase abstracta no puede instanciarse directamente y sirve como base para otras clases. Los métodos abstractos no tienen cuerpo y deben ser implementados por las clases derivadas.

Aquí está el ejemplo adaptado para Java, transformando la clase Persona en abstracta y definiendo su método mostrarInfo como abstracto:

```
// Clase abstracta
abstract class Persona {
    // Método abstracto
    public abstract void mostrarInfo();
}
// Clase derivada 1
class Empleado extends Persona {
    @Override
    public void mostrarInfo() {
        System.out.println("Información del Empleado.");
    }
}
// Clase derivada 2
class Cliente extends Persona {
    @Override
    public void mostrarInfo() {
        System.out.println("Información del Cliente.");
    }
}
public class Main {
    public static void main(String[] args) {
        // Persona persona = new Persona(); // Esto provoca un error: No se puede
instanciar una clase abstracta
        Persona empleado = new Empleado();
        Persona cliente = new Cliente();
        empleado.mostrarInfo(); // Salida: Información del Empleado.
        cliente.mostrarInfo();  // Salida: Información del Cliente.
    }
}
```

Cuando una clase se declara como abstracta en Java, implica lo siguiente:

1. No se puede instanciar directamente:
   - Una clase abstracta no puede tener objetos creados directamente; solo puede servir como clase base para otras clases.
2. Obligación de implementación:
   - Todas las clases derivadas que no sean abstractas deben implementar todos los métodos abstractos definidos en la clase base.

```
// Clase abstracta
abstract class Persona {
    // Método abstracto
    public abstract void mostrarInfo();
}

// Clase derivada concreta
class Empleado extends Persona {
    @Override
    public void mostrarInfo() {
        System.out.println("Información del Empleado.");
    }
}

// Clase derivada concreta
class Cliente extends Persona {
    @Override
    public void mostrarInfo() {
        System.out.println("Información del Cliente.");
    }
}

// Clase abstracta derivada
abstract class Visitante extends Persona {
    // No implementa mostrarInfo, por lo que también debe declararse como
abstracta
}

public class Main {
    public static void main(String[] args) {
        // Persona persona = new Persona(); // Error: No se puede instanciar una
clase abstracta

        // Instanciando clases concretas
        Persona empleado = new Empleado();
        Persona cliente = new Cliente();

        empleado.mostrarInfo(); // Salida: Información del Empleado.
        cliente.mostrarInfo();  // Salida: Información del Cliente.

        // Visitante visitante = new Visitante(); // Error: También es abstracta
    }
}
```

7 Interfaces

En Java, las interfaces funcionan de manera similar a C#, y permiten que una clase implemente múltiples interfaces, lo que compensa la falta de herencia múltiple de clases. Al igual que en C#, una interfaz no puede contener implementaciones de métodos (excepto a partir de Java 8, donde se permite la implementación predeterminada de métodos), y todas las clases que implementen una interfaz deben proporcionar implementaciones de todos sus métodos.

Propiedades de una interfaz en Java:

3. Definición de una interfaz: Se define con la palabra clave interface.
4. Métodos sin implementación: Los métodos en una interfaz no tienen cuerpo (en versiones anteriores a Java 8). Desde Java 8, pueden tener una implementación predeterminada.
5. No se puede instanciar una interfaz: No se pueden crear instancias de una interfaz directamente.
6. Implementación obligatoria en las clases: Cualquier clase que implemente una interfaz debe proporcionar una implementación para todos sus métodos, a menos que la clase sea abstracta.

```
// Definición de la interfaz
interface Funcionalidad {
    // Método abstracto
    void mostrarInfo();
    // Desde Java 8, podemos tener métodos con implementación predeterminada
    default void metodoConcreto() {
        System.out.println("Este es un método con implementación
predeterminada.");
    }
}

// Clase que implementa la interfaz
class Empleado implements Funcionalidad {
    @Override
    public void mostrarInfo() {
        System.out.println("Información del Empleado.");
    }
}
```

```
// Otra clase que implementa la interfaz
class Cliente implements Funcionalidad {
    @Override
    public void mostrarInfo() {
        System.out.println("Información del Cliente.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Instanciamos las clases
        Empleado empleado = new Empleado();
        Cliente cliente = new Cliente();

        // Llamamos al método implementado por ambas clases
        empleado.mostrarInfo(); // Salida: Información del Empleado.
        cliente.mostrarInfo();  // Salida: Información del Cliente.

        // Llamada al método predeterminado de la interfaz
        empleado.metodoConcreto(); // Salida: Este es un método con implementación
predeterminada.
        cliente.metodoConcreto();  // Salida: Este es un método con implementación
predeterminada.
    }
}
```

Características Importantes:

- Interfaz como clase base abstracta: Las interfaces son similares a las clases base abstractas, pero no pueden contener implementaciones de métodos, a menos que sean métodos predeterminados (a partir de Java 8).
- Implementación múltiple: Java permite que una clase implemente múltiples interfaces, lo que compensa la falta de herencia múltiple de clases.
- Instanciación: Las interfaces no pueden ser instanciadas directamente, pero sus métodos se implementan en las clases que las implementan.

En Java, el polimorfismo también se aplica a las interfaces. Una clase que implementa una interfaz puede ser tratada como una instancia del tipo de la interfaz. Esto permite escribir código más flexible y reutilizable, ya que puedes trabajar con diferentes clases a través de una interfaz común.

```
// Definición de la interfaz
interface IInfo {
    void mostrarInfo(); // Método abstracto que todas las clases deben implementar
}
// Clase base abstracta
abstract class Persona implements IInfo {
    protected String nombre;

    public Persona(String nombre) {
        this.nombre = nombre;
    }
}
// Clase derivada 1
class Empleado extends Persona {
    private String puesto;
    public Empleado(String nombre, String puesto) {
        super(nombre);
        this.puesto = puesto;
    }
    @Override
    public void mostrarInfo() {
        System.out.println("Empleado: " + nombre + ", Puesto: " + puesto);
    }
}
// Clase derivada 2
class Cliente extends Persona {
    private String empresa;
    public Cliente(String nombre, String empresa) {
        super(nombre);
        this.empresa = empresa;
    }
    @Override
    public void mostrarInfo() {
        System.out.println("Cliente: " + nombre + ", Empresa: " + empresa);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Polimorfismo: Trabajar con objetos a través de la interfaz
        IInfo empleado = new Empleado("Carlos", "Gerente");
        IInfo cliente = new Cliente("Ana", "TechCorp");

        // Llamada al método mostrarInfo a través de la interfaz
        empleado.mostrarInfo(); // Salida: Empleado: Carlos, Puesto: Gerente
        cliente.mostrarInfo();  // Salida: Cliente: Ana, Empresa: TechCorp

        // También puedes trabajar con una colección de objetos IInfo
        IInfo[] personas = {
            new Empleado("Pedro", "Analista"),
            new Cliente("Luisa", "Fintech")
        };

        for (IInfo info : personas) {
            info.mostrarInfo();
        }
        // Salida:
        // Empleado: Pedro, Puesto: Analista
        // Cliente: Luisa, Empresa: Fintech
    }
}
```

Beneficios del Polimorfismo con Interfaces:

- Flexibilidad: Puedes escribir código que funcione con diferentes tipos de objetos que implementan la misma interfaz sin preocuparte por sus tipos específicos.
- Extensibilidad: Agregar nuevas clases que implementen la interfaz no requiere modificar el código existente que utiliza esa interfaz.
- Reutilización de código: Permite trabajar con objetos de diferentes clases de una manera uniforme.

8 Clases estáticas

Una clase estática es una clase con todos sus miembros y variables estáticos, no contiene ninguna variable o miembro de instancia. Básicamente es igual que una clase no estática, con la diferencia de que no se pueden crear instancias de una clase estática. En otras palabras, no podemos usar el operador new para crear una variable del tipo de clase. Dado que no hay ninguna variable de instancia, para tener acceso a los miembros de una clase estática, debe usar el nombre de la clase.

Únicamente pueden contener un constructor, que será un constructor privado estático, del tipo que hemos visto en el apartado Constructores.

Las clases estáticas están selladas, por lo que no se puede heredar de ellas.

Una clase estática es un contenedor adecuado para conjuntos de métodos que solo necesitan parámetros de entrada y que no tienen que obtener ni establecer campos de instancias internas. Por ejemplo, en la biblioteca de clases .NET Framework, la clase estática System.Math contiene métodos que realizan operaciones matemáticas, sin ningún requisito para almacenar o recuperar datos que sean únicos de una instancia concreta de la clase Math.

Veamos un ejemplo; la siguiente clase estática permite conversiones de grados centígrados a Fahrenheit y viceversa.

```
public final class ConversionTemperatura {

    // Constructor privado para evitar instanciación

    private ConversionTemperatura() {

        // Este constructor está vacío intencionalmente para evitar la
creación de instancias

    }


    // Método estático para convertir de Celsius a Fahrenheit

    public static double celsiusAFahrenheit(double celsius) {

        return (celsius * 9 / 5) + 32;

    }


    // Método estático para convertir de Fahrenheit a Celsius

    public static double fahrenheitACelsius(double fahrenheit) {

        return (fahrenheit - 32) * 5 / 9;

    }

}
```

```
public class Main {
    public static void main(String[] args) {
        // Uso de los métodos estáticos de la clase "estática"
        double celsius = 25.0;
        double fahrenheit = ConversionTemperatura.celsiusAFahrenheit(celsius);
        System.out.println(celsius + "°C en Fahrenheit es " + fahrenheit + "°F");

        fahrenheit = 77.0;
        celsius = ConversionTemperatura.fahrenheitACelsius(fahrenheit);
        System.out.println(fahrenheit + "°F en Celsius es " + celsius + "°C");
    }
}
```

9 Composición de clases

En Java, la herencia permite crear relaciones jerárquicas entre clases, lo que facilita la reutilización del código y establece una relación de "es un(a)" entre las clases. Este concepto de herencia refuerza la idea de que una clase hija hereda las características (atributos y métodos) de su clase padre, mientras que puede agregar o modificar funcionalidades para ajustarse a un propósito específico.

```
// Clase base: Persona
class Persona {
    protected String nombre;
    protected int edad;
    // Constructor
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    // Método para mostrar información
    public void mostrarInfo() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

```
// Clase hija: Empleado
class Empleado extends Persona {
    protected String puesto;
    // Constructor
    public Empleado(String nombre, int edad, String puesto) {
        super(nombre, edad); // Llama al constructor de la clase padre
        this.puesto = puesto;
    }
    // Sobrescribir el método mostrarInfo
    @Override
    public void mostrarInfo() {
        super.mostrarInfo(); // Llama al método de la clase padre
        System.out.println("Puesto: " + puesto);
    }
}
```

```
// Clase nieta: EmpleadoCualificado
class EmpleadoCualificado extends Empleado {
    private String especializacion;
    // Constructor
    public EmpleadoCualificado(String nombre, int edad, String puesto, String
especializacion) {
        super(nombre, edad, puesto); // Llama al constructor de la clase padre
(Empleado)
        this.especializacion = especializacion;
    }
    // Sobrescribir el método mostrarInfo
    @Override
    public void mostrarInfo() {
        super.mostrarInfo(); // Llama al método de la clase padre
        System.out.println("Especialización: " + especializacion);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Crear objetos de diferentes niveles de la jerarquía
        Persona persona = new Persona("Juan", 40);
        Empleado empleado = new Empleado("Laura", 30, "Gerente");
        EmpleadoCualificado empleadoCualificado = new EmpleadoCualificado("Ana",
35, "Ingeniera", "Desarrollo de Software");

        // Llamar al método mostrarInfo
        System.out.println("Información de Persona:");
        persona.mostrarInfo();

        System.out.println("\nInformación de Empleado:");
        empleado.mostrarInfo();

        System.out.println("\nInformación de Empleado Cualificado:");
        empleadoCualificado.mostrarInfo();
    }
}
```

En Java, la composición es un enfoque de diseño alternativo a la herencia, donde una clase incluye instancias de otras clases como atributos para reutilizar su funcionalidad. En lugar de establecer una relación de "es un(a)" mediante herencia, la composición se basa en la relación de "tiene un(a)". Esto puede simplificar el diseño, reducir las dependencias y permitir mayor flexibilidad.

Aquí tienes cómo resolver el ejemplo utilizando composición:

```
class Persona {
    protected String nombre;
    protected int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre; this.edad = edad;
    }
    public void mostrarInfo() {
        System.out.println("Nombre: " + nombre + ", Edad: " + edad);
    }
}
```

```
class Categoria {
    private String nombreCategoria;
    public Categoria(String nombreCategoria) {
        this.nombreCategoria = nombreCategoria;
    }
    // Método para mostrar información de la categoría
    public String getNombreCategoria() {
        return nombreCategoria;
    }
}
class Retribucion {
    private double salario;
    public Retribucion(double salario) {
        this.salario = salario;
    }
    // Método para mostrar el salario
    public double getSalario() {
        return salario;
    }
}
// Clase Empleado que utiliza composición
class Empleado {
    private Persona persona; // Composición con Persona
    private Categoria categoria; // Composición con Categoria
    private Retribucion retribucion; // Composición con Retribucion
    // Constructor
    public Empleado(Persona persona, Categoria categoria, Retribucion retribucion)
{
        this.persona = persona; this.categoria = categoria; this.retribucion =
retribucion;
    }
    // Método para mostrar información del empleado
    public void mostrarInfo() {
        persona.mostrarInfo(); // Llama al método de la clase Persona
        System.out.println("Categoría: " + categoria.getNombreCategoria());
        System.out.println("Salario: $" + retribucion.getSalario());
    }
}
```

```
// Clase principal para probar el ejemplo
public class Main {
    public static void main(String[] args) {
        // Crear los componentes del Empleado
        Persona persona = new Persona("Carlos", 45);
        Categoria categoria = new Categoria("Gerente");
        Retribucion retribucion = new Retribucion(75000.0);

        // Crear un Empleado utilizando composición
        Empleado empleado = new Empleado(persona, categoria, retribucion);

        // Mostrar información del empleado
        empleado.mostrarInfo();
    }
}
```

Comparación entre Herencia y Composición:

Aspecto	Herencia	Composición
Relación	"Es un(a)"	"Tiene un(a)"
Flexibilidad	Más rígida (modificaciones afectan a toda la jerarquía)	Más flexible (los componentes pueden cambiarse)
Reutilización	A través de clases base	A través de objetos incluidos
Complejidad	Puede generar una jerarquía complicada	Más modular y manejable
Ejemplo en este caso	Empleado es una Persona	Empleado tiene una Persona

La composición permite construir sistemas más modulares y adaptables, especialmente en aplicaciones donde los requisitos pueden cambiar con frecuencia.

Vemos que la composición ofrece una solución más flexible que la jerarquía para representar las categorías y retribuciones de los empleados de una empresa.

Leeremos en muchos artículos que en el diseño de clases debemos utilizar la composición como preferencia frente a la jerarquía. Lo adecuado es pensar si entre las clases y objetos tenemos una relación de “ser” o de “tener” y cuando la relación sea realmente de “ser” utilizar la herencia, pero cuando la relación sea una relación de tener (una característica, una cualidad, etc.) es preferible tener utilizar la composición.

10 Delegación

Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Cuando se crea una instancia de un delegado, puede asociar su instancia a cualquier método mediante una signatura compatible y un tipo de valor devuelto. Podemos llamar al método a través de la instancia del delegado.

Los delegados se utilizan para pasar métodos como argumentos a otros métodos. Los controladores de eventos no son más que métodos que se invocan a través de delegados.

En Java, se puede lograr un comportamiento similar utilizando interfaces funcionales y expresiones lambda, especialmente desde la introducción de Java 8.

Una interfaz funcional en Java es una interfaz que contiene exactamente un método abstracto. Esto permite tratar las implementaciones como referencias a métodos o funciones, lo que es conceptualmente similar a los delegados.

```
// Definir una interfaz funcional que actúa como un delegado
@FunctionalInterface
interface Operacion {
    double ejecutar(double a, double b);
}

public class Main {
    // Método que recibe un delegado como parámetro
    public static void realizarOperacion(double a, double b, Operacion operacion)
{
        double resultado = operacion.ejecutar(a, b);
        System.out.println("Resultado: " + resultado);
    }

    public static void main(String[] args) {
        // Usar lambdas para implementar la interfaz funcional
        Operacion suma = (x, y) -> x + y;
        Operacion resta = (x, y) -> x - y;
        Operacion multiplicacion = (x, y) -> x * y;
        Operacion division = (x, y) -> y != 0 ? x / y : 0;

        // Llamar al método pasando diferentes delegados
        System.out.println("Suma:");
        realizarOperacion(10, 5, suma);

        System.out.println("Resta:");
        realizarOperacion(10, 5, resta);

        System.out.println("Multiplicación:");
        realizarOperacion(10, 5, multiplicacion);

        System.out.println("División:");
        realizarOperacion(10, 5, division);
    }
}
```

11 Indizadores

Un indizador es un miembro de una clase que permite indexar la clase de la misma manera que un array.

Veamos un ejemplo, creamos la clase Nota:

```
public class Nota {
    private double[] calificaciones;

    // Constructor para inicializar el array de calificaciones
    public Nota(int cantidadNotas) {
        if (cantidadNotas <= 0) {
            throw new IllegalArgumentException("La cantidad de notas debe ser
mayor que 0.");
        }
        calificaciones = new double[cantidadNotas];
    }

    // Método para establecer una calificación en un índice específico
    public void setCalificacion(int indice, double valor) {
        if (indice < 0 || indice >= calificaciones.length) {
            throw new IndexOutOfBoundsException("Índice fuera de rango.");
        }
        calificaciones[indice] = valor;
    }

    // Método para obtener una calificación de un índice específico
    public double getCalificacion(int indice) {
        if (indice < 0 || indice >= calificaciones.length) {
            throw new IndexOutOfBoundsException("Índice fuera de rango.");
        }
        return calificaciones[indice];
    }

    // Método para obtener la cantidad de notas
    public int getCantidadNotas() {
        return calificaciones.length;
    }
```

```
public static void main(String[] args) {
        // Crear un objeto Nota con 5 calificaciones
        Nota notas = new Nota(5);

        // Asignar valores usando el "indizador" simulado
        notas.setCalificacion(0, 8.5);
        notas.setCalificacion(1, 9.0);
        notas.setCalificacion(2, 7.5);
        notas.setCalificacion(3, 10.0);
        notas.setCalificacion(4, 6.5);

        // Obtener valores usando el "indizador" simulado
        for (int i = 0; i < notas.getCantidadNotas(); i++) {
            System.out.println("Nota " + i + ": " + notas.getCalificacion(i));
        }
    }
}
```

12 Sobrecarga de operadores

Un tipo definido por el usuario puede sobrecargar un operador de C# predefinido. Es decir, un tipo puede proporcionar la implementación personalizada de una operación cuando uno o los dos operandos son de ese tipo.

Usaremos la palabra clave *operator* para declarar un operador. Una declaración de operador debe cumplir las reglas siguientes:

- Incluir los modificadores *public* y *static*.
- Un operador unario debe un parámetro de entrada. Un operador binario debe tener dos parámetros de entrada. En cada caso, al menos un parámetro debe ser de tipo T o T? donde T es el tipo que contiene la declaración del operador.

En Java no se pueden sobrecargar los operadores. Sin embargo, podemos lograr el mismo comportamiento creando un método específico para sumar dos puntos.

```
public class Punto {
    private int x;
    private int y;

    // Constructor
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Método para sumar dos puntos
    public Punto sumar(Punto otroPunto) {
        return new Punto(this.x + otroPunto.x, this.y + otroPunto.y);
    }

    // Método para mostrar los valores de las coordenadas
    public void mostrar() {
        System.out.println("Resultado: X=" + this.x + ", Y=" + this.y);
    }

    public static void main(String[] args) {
        Punto p1 = new Punto(1, 2);
        Punto p2 = new Punto(3, 4);

        Punto p3 = p1.sumar(p2);  // Llamamos al método para sumar los puntos
        p3.mostrar();  // Mostramos el resultado
    }
}
```

13 Métodos de extensión

Los métodos de extensión permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original. Los métodos de extensión son métodos estáticos, pero se les llama como si fueran métodos de instancia en el tipo extendido.

En Java, no existe un concepto directo de "métodos de extensión" como en C#, ya que Java no permite la adición de métodos a clases existentes sin modificarlas o extenderlas. Sin embargo, se puede simular algo similar mediante el uso de clases utilitarias y métodos estáticos, los cuales proporcionan funcionalidades adicionales a las clases existentes.

Veamos cómo podemos simular métodos de extensión en Java para la clase String

```
// Clase de utilidad para simular métodos de extensión en Java
public class StringUtils {

    // Método para invertir una cadena
    public static String invertir(String str) {
        StringBuilder sb = new StringBuilder(str);
        return sb.reverse().toString();
    }

    public static void main(String[] args) {
        // Ejemplo de uso del método simulado como extensión
        String original = "Hola Mundo";
        String invertida = StringUtils.invertir(original);

        System.out.println("Original: " + original);
        System.out.println("Invertida: " + invertida);
    }
}
```

Ahora en lugar de hacer las llamadas a la clase StringExtensions, podemos llamar directamente a los métodos como si fueran de la clase string en lugar de llamar a los métodos estáticos de StringExtensions

14 Introspección. Reflexión.

En Java, la reflexión es una característica poderosa que permite a un programa inspeccionar y modificar la estructura de clases, interfaces, métodos, campos y otros aspectos de los objetos en tiempo de ejecución. La reflexión en Java se basa principalmente en el paquete java.lang.reflect y proporciona una forma de acceder a información sobre clases, interfaces, constructores y métodos en tiempo de ejecución, incluso si no se conocen en tiempo de compilación.

En Java, se puede usar la clase Class y el paquete java.lang. A continuación, se presenta un ejemplo de cómo usar la reflexión en Java:

```
import java.lang.reflect.Method;
import java.lang.reflect.Field;

public class Persona {
    private String nombre;

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

    public void mostrarInfo() {
        System.out.println("Nombre: " + nombre);
    }

    public static void main(String[] args) throws Exception {
        // Crear una instancia de la clase Persona
        Persona persona = new Persona("Juan");

        // Obtener el tipo de la clase Persona en tiempo de ejecución
        Class<?> tipo = persona.getClass();

        // Invocar el método mostrarInfo mediante reflexión
        Method metodo = tipo.getMethod("mostrarInfo");
        metodo.invoke(persona);

        // Obtener y mostrar los campos
        Field campo = tipo.getDeclaredField("nombre");
        campo.setAccessible(true); // Hacer el campo accesible
        System.out.println("Campo Nombre: " + campo.get(persona));
    }
}
```