UD 09 – Aspectos y usos avanzados de clases.

Contenido

1.- Encapsulación y visibilidad 2
Ejemplo1: 2
Ejemplo 2: Esfera: 3
2.- Modularidad 4
3.- Herencia 4
3.1 Constructores 4
3.1 Jerarquía 5
4.- Polimorfismo 5
4.1- Polimorfismo con métodos: 6
4.2- Polimorfismo con interfaces: 6
5.- Clases y métodos abstractos: 7
6.- Interfaces: 8
7.- Clases estáticas: 9
8. Expresiones Lambda 10

1.- 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 OOP 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.

Ejemplo1:

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

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

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));
}
```

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.

```java
public class Esfera {
    private double radio;

    public double getRadio() {
        return radio;
    }

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

    public double getArea() {
        return 4.0 * Math.PI * radio * radio;
    }

    public double getVolumen() {
        return 4.0 / 3.0 * Math.PI * radio * radio * radio;
    }
}
```

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

1. "**public**": El miembro es accesible desde cualquier otra clase.
2. "**private**": El miembro sólo es accesible dentro de su propia clase.

3. “**protected**”: El miembro es accesible dentro de su propia clase, en cualquier clase dentro del mismo paquete, y también en una subclase en otro paquete.
4. Sin modificador (también conocido como "package-private" o "default"): El miembro es accesible dentro de su propia clase y en cualquier otra clase dentro del mismo paquete.

Estos modificadores de acceso permiten controlar el nivel de visibilidad de los campos, métodos y clases en Java, lo que es fundamental para el encapsulamiento y la ocultación de información en la programación orientada a objetos.

2.- 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.

3.- Herencia

La herencia es una de las características principales de la OOP. La programación orientada a objetos introduce la posibilidad de extender clases, produciendo nuevas definiciones de clases que heredan todo el comportamiento y código de la clase extendida. La clase original se denomina clase padre, clase base o superclase. La nueva clase que se define como una extensión se denomina clase hija, clase derivada o subclase. La clase hija hereda todos los métodos y atributos de la clase padre que extiende. Cada subclase estaría formada por un grupo de objetos más especializados con características comunes que compartirían datos y operaciones. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.

El mecanismo de herencia produce una jerarquía de clases desde la clase raíz hasta todas las clases que derivan de ella.

En Java todas las clases de heredan implícitamente de la clase Object

3.1 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 Alumno {
    private String nombre;

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

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

    public static void main(String[] args) {
        Alumno a = new Alumno();
    }
}
```

Pero cuidado, que si nos declaramos un constructor con parámetros, el constructor vacío ya no se crea automáticamente:

```
public class Alumno {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }

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

    public static void main(String[] args) {
        Alumno a = new Alumno();    //ERROR
    }
}
```

Nosotros deberíamos crear un segundo constructor, en este caso sin parámetros.

3.1 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

4.- Polimorfismo

El polimorfismo es uno de los cuatro principios fundamentales de la programación orientada a objetos (POO). En Java, el polimorfismo permite que una referencia de clase padre pueda

referirse a un objeto de cualquier clase hija. Esto permite que el mismo código se comporte de manera diferente en diferentes contextos, lo que puede simplificar el código y hacerlo más reutilizable.

4.1- Polimorfismo con métodos:

```
class Animal {
    void sound() {
        System.out.println("El animal hace un sonido");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("El perro ladra");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("El gato maúlla");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myAnimal.sound();
        myDog.sound();
        myCat.sound();
    }
}
```

En este ejemplo, “Animal” es la clase padre y “Dog” y “Cat” son las clases hijas. Cada clase tiene un método “sound()”. Cuando llamamos a “sound()” en un objeto “Animal”, “Dog” o “Cat”, el método correspondiente de la clase del objeto se ejecuta.

4.2- Polimorfismo con interfaces:

```
interface Animal {
    void sound();
}

class Dog implements Animal {
    public void sound() {
        System.out.println("El perro ladra");
    }
}

class Cat implements Animal {
    public void sound() {
        System.out.println("El gato maúlla");
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.sound();
        myCat.sound();
    }
}
```

En este ejemplo, “Animal” es una interfaz y “Dog” y “Cat” son clases que implementan la interfaz. Cada clase tiene un método “sound()”. Cuando llamamos a “sound()” en un objeto “Dog” o “Cat”, el método correspondiente de la clase del objeto se ejecuta.

5.- Clases y métodos abstractos:

En Java, una clase abstracta es una clase que no puede ser instanciada y se utiliza como base para otras clases. Las clases abstractas pueden contener métodos abstractos y métodos no abstractos.

Un método abstracto es un método que se declara en una clase abstracta sin una implementación. Las subclases de la clase abstracta deben proporcionar una implementación para estos métodos abstractos.

```
abstract class Animal {
    abstract void sound();

    void eat() {
        System.out.println("El animal come");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("El perro ladra");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("El gato maúlla");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        Cat myCat = new Cat();

        myDog.sound();
        myDog.eat();
```

```
        myCat.sound();
        myCat.eat();
    }
}
```

En este ejemplo, “Animal” es una clase abstracta con un método abstracto “sound()” y un método no abstracto “eat()”. “Dog” y “Cat” son subclases de “Animal” que proporcionan una implementación para el método “sound()”. Cuando llamamos a “sound()” y “eat()” en un objeto “Dog” o “Cat”, los métodos correspondientes de la clase del objeto se ejecutan.

6.- Interfaces:

En Java, una interfaz es una referencia de tipo completamente abstracta que se utiliza para especificar un comportamiento que las clases deben implementar. Son similares a las clases abstractas, pero mientras una clase sólo puede extender una única clase padre, puede implementar múltiples interfaces.

```
interface Animal {
    void sound();
    void eat();
}

class Dog implements Animal {
    public void sound() {
        System.out.println("El perro ladra");
    }

    public void eat() {
        System.out.println("El perro come");
    }
}

class Cat implements Animal {
    public void sound() {
        System.out.println("El gato maúlla");
    }

    public void eat() {
        System.out.println("El gato come");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        Cat myCat = new Cat();

        myDog.sound();
        myDog.eat();

        myCat.sound();
        myCat.eat();
    }
}
```

En este ejemplo, Animal es una interfaz con dos métodos: sound() y eat(). Dog y Cat son clases que implementan la interfaz Animal y proporcionan una implementación para los métodos sound() y eat(). Cuando llamamos a sound() y eat() en un objeto Dog o Cat, los métodos correspondientes de la clase del objeto se ejecutan.

7.- Clases estáticas:

En Java, no puedes crear una clase estática en el sentido más estricto como puedes hacerlo en algunos otros lenguajes de programación. Sin embargo, puedes hacer que una clase se comporte de manera similar a una clase estática al hacer que el constructor sea privado y todos los miembros y funciones sean estáticos.

Una clase estática es una clase que no necesita ser instanciada y todos sus miembros y funciones pueden ser accedidos directamente.

```
public class StaticClass {
    // Variable estática
    private static String staticVariable = "Soy una variable
estática";

    // Constructor privado para prevenir la instanciación
    private StaticClass() {
    }

    // Método estático
    public static void staticMethod() {
        System.out.println("Soy un método estático");
    }

    // Getter para la variable estática
    public static String getStaticVariable() {
        return staticVariable;
    }

    // Setter para la variable estática
    public static void setStaticVariable(String staticVariable) {
        StaticClass.staticVariable = staticVariable;
    }
}

public class Main {
    public static void main(String[] args) {
        // Accediendo al método estático
        StaticClass.staticMethod();

        // Accediendo a la variable estática
        System.out.println(StaticClass.getStaticVariable());

        // Modificando la variable estática
        StaticClass.setStaticVariable("Nuevo valor");
        System.out.println(StaticClass.getStaticVariable());
    }
}
```

En este ejemplo, StaticClass es una clase con un constructor privado, una variable estática y un método estático. En la clase Main, accedemos al método y a la variable estática directamente desde la clase, sin necesidad de crear una instancia de StaticClass.

8. Expresiones Lambda

Una expresión lambda es un bloque corto de código que recibe parámetros y devuelve un valor. Las expresiones lambda son similares a los métodos, pero no necesitan un nombre y pueden implementarse directamente en el cuerpo de un método.

Sintaxis:

```
parameter -> expression

(parameter1, parameter2) -> expression

(parameter1, parameter2) -> { code block }
```

Veamos un ejemplo muy sencillo de una expresión lambda que se utiliza para implementar un Comparator para una lista de enteros:

```
import java.util.Arrays;
import java.util.List;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 2, 3, 1, 4);

        // Usamos una expresión lambda para definir cómo se deben
comparar los números
        Collections.sort(numbers, (n1, n2) -> n1 - n2);

        System.out.println(numbers);
    }
}
```

En este ejemplo, (n1, n2) -> n1 - n2 es una expresión lambda. Toma dos parámetros, n1 y n2, y devuelve el resultado de n1 - n2. Esta expresión se utiliza para determinar el orden de los números en la lista.

Las expresiones lambda pueden ser muy útiles para simplificar tu código, especialmente cuando estás trabajando con interfaces funcionales (interfaces con un solo método abstracto), ya que te permiten implementar el método directamente en tu código sin necesidad de crear una clase anónima.