Java-Programación Orientada a Objetos

Contenido

Teoría de Objetos:

La programación convencional utiliza toda clase de valores y funciones en una sola clase, lo cual hace que acabe habiendo un código con líneas muy dispersas, repetitivas y, en definitiva, largo e inentendible. Para ello, se utiliza la Programación Orientada a Objetos(POO), que permite ligar diferentes clases con métodos y atributos para poder hacer los códigos más ordenados y legibles.

Uso de objetos y clases:

Los objetos no son ni más ni menos que los accesos de una clase a otra. Por ejemplo, estamos en una clase Main y queremos acceder a una clase llamada SubMain, pues la mejor forma que hay de hacerlo es por medio de un objeto:

Constructor de una clase:

En la clase que se usará como objeto, se debe crear un método para inicializar lo necesario, no es obligatorio hacerlo, porque Java tiene lo que se conoce como un constructor vacío, creado invisible en toda clase para hacer un objeto sin parámetros predefinidos.

Método constructor vacío: clase()

```
public class SubMain {
    public SubMain(){

   }
} //Esto sería un constructor vacío, es decir, sin valores
```

Método constructor: clase(valores)

Cabe recalcar que NO ES NECESARIO QUE ESTÉN ABSOLUTAMENTE TODOS LOS ATRIBUTOS EN EL CONSTRUCTOR, solo los que queramos y nos interesen

```
public class SubMain {
    int numero;
    String frase;
    public SubMain(int numero){
        this.numero = numero; //Pasamos el número del objeto aquí
    }
```

Declaración y creación de un Objeto:

Es parecido a los arrays en cuanto su declaración: Clase nombre = new Clase(valores);

```
public class Main {
    public static void main(String[] args) {
        SubMain subMain = new SubMain();
    }
}
```

En este caso, está creando uno con un constructor vacío, de necesitar pasar valores, el IDE te lo avisará.

```
public class Main {
    public static void main(String[] args) {
        SubMain subMain = new SubMain(1); //Usando el constructor con int
    }
}
```

Llamada de funciones

En la clase: Declaración de un método.

```
public class SubMain {
    public SubMain() {
    }
    public void funcion(){
    }
}
```

Llamada de una función con el objeto:

Se usa un punto '.' para llamar las funciones

```
public class Main {
    public static void main(String[] args) {
        SubMain subMain = new SubMain();
        subMain.funcion();
    }
}
```

Obtención y Declaración de valores dentro de las clases (Getter y Setter)

Los atributos de las clases, para mayor seguridad, deben estar en privado (private). Su modo de acceso cambia de simplemente llamar al objeto (objeto.valor) a tener que usar métodos de obtención y escritura (get/set)

Obtención en la clase:

```
public class SubMain {
    private int numero;
    public int getNumero() {
        return numero;
    }
}
```

Obtención con el objeto:

Devuelve un valor, por lo que **o se almacena o se imprime**

```
public class Main {
    public static void main(String[] args) {
        SubMain subMain = new SubMain();
        System.out.println(subMain.getNumero());
    }
}
```

Escritura en la clase:

```
public class SubMain {
    private int numero;

    public SubMain(int numero) {
        this.numero = numero;
    }

    public void setNumero(int numero) {
        this.numero=numero;
    }
}
```

Escritura con el objeto:

```
public class Main {
    public static void main(String[] args) {
        SubMain subMain = new SubMain(3); //Inicia el numero como 3
        subMain.setNumero(5); //Sobreescribe el valor de numero a 5
    }
}
```

Impresión de todos los valores (que se quieran) de un objeto (toString)

Para no tener que hacer una fila de souts con cada Getter de un objeto, se usa la función toString, generada automáticamente por una clase

Método toString en clases:

```
public class SubMain {
    private int numero;
    public SubMain(int numero) {
        this.numero = numero;
    }
    @Override
    public String toString() {
        return "SubMain{" +
                "numero=" + numero +
                '}';
    }
}
```

Método toString con el objeto:

Devuelve un valor, por lo que hace falta almacenarlo en algún lado o imprimirlo por pantalla

```
public class Main {
    public static void main(String[] args) {
        SubMain subMain = new SubMain(3); //Inicia el numero como 3
        subMain.toString();
    }
}
```

Método Equals:

Por defecto, Objeto.equals(objeto) hace lo mismo que si pusieras Objeto == objeto. Lo que hacen ambas opciones es comparar la referencia que se crea con el objeto (La dirección de memoria). Solo se cumpliría si es igual, pero eso no se cumple casi nunca se cumple ni mucho menos se necesita. Por lo que se sobrecarga el método Equals en la clase.

```
public class SubMain {
    private int numero;
    public SubMain(int numero) {
        this.numero = numero;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        SubMain subMain = (SubMain) o;
        return numero == subMain.numero;
    }
}
```

Interfaces:

Las interfaces se pueden entender como “contratos” con condiciones que han de cumplirse en la clase en la que se implementa, marcando unos mínimos que ésta debe tener. Suele representarse como iClase, es decir, el nombre de la clase con una i al principio.

```
public interface iSubMain {
    public void funcion1();
    public int funcion2(int i);
    public void funcion3();
}
```

Si se implementara esta interfaz en alguna clase, la clase debería tener, como mínimo, esas funciones.

```
public class SubMain implements iSubMain{
    public SubMain() {
    }
    @Override
    public void funcion1() {

    }

    @Override
    public int funcion2(int i) {
        return 0;
    }

    @Override
    public void funcion3() {

    }

   public boolean funcion4(){
        return false;
    }
}
```

Hay que recalcar que las interfaces son ORIENTATIVAS, es decir, que sirve simplemente para ayudar a quien lea el código a ver más fácilmente los métodos utilizados, pudiendo editarlo si es necesario

Interfaz Comparable

Si queremos establecer un criterio de comparación entre objetos de una clase (por ejemplo, para una ordenación), se implementa una interfaz de nombre “Comparable<?>”, que solo tiene un método: **compareTo(objeto)**

```
public class Ejemplo implements Comparable<Ejemplo>{
    int numero;
    String nombre;

    public Ejemplo(int numero) {
        this.numero = numero;
    }

    @Override
    public int compareTo(Ejemplo example2) {
        if (numero<example2.numero){
            return -1;
           //Si da -1, el valor del objeto 1 es menor que el del
objeto 2
        } else if (numero> example2.numero) {
            return 1;
           //Si da 1, el valor del objeto 1 es mayor que el del objeto
2
        } else {
            return 0;
           //Dan el mismo valor
        }
    }
}
```

Aplicación simplisima en main:

```
public class Teoria {
    public static void main(String[] args) {
        Ejemplo example1 = new Ejemplo(5);
        Ejemplo example2 = new Ejemplo(7);
        Ejemplo example3 = new Ejemplo(7);

        System.out.println(example1.compareTo(example2)); //-1
        System.out.println(example2.compareTo(example1)); //1
        System.out.println(example3.compareTo(example2)); //0
    }
}
```

Ordenación de Arrays y ArrayLists

Las APIs de Arrays y Arraylists, a la hora de ordenar objetos con sort(), cuentan con la posibilidad de aplicar la interfaz comparable para hacer un orden natural en función del atributo que elijamos:

```java
public class Ejemplo implements Comparable<Ejemplo>{
    int numero;
    String nombre;

    public Ejemplo(int numero, String nombre) {
        this.numero = numero;
        this.nombre = nombre;
    }

    @Override
    public int compareTo(Ejemplo example2) {
        return nombre.compareTo(example2.nombre);
    }

    @Override
    public String toString() {
        return "Ejemplo{" +
                "numero=" + numero +
                ", nombre='" + nombre + '\'' +
                '}';
    }
}
```

```java
import java.util.Arrays;

public class Teoria {
    public static void main(String[] args) {
        Ejemplo example1 = new Ejemplo(5, "AABE");
        Ejemplo example2 = new Ejemplo(7, "Si");
        Ejemplo example3 = new Ejemplo(7, "AAAA4");

        Ejemplo[] ejemplos = {example1, example2, example3};
        System.out.println(Arrays.toString(ejemplos));
        Arrays.sort(ejemplos);
        System.out.println(Arrays.toString(ejemplos));

    }
}
```

Resultado final:

```
[Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='Si'}, Ejemplo{numero=7, nombre='AAAA4'}]
[Ejemplo{numero=7, nombre='AAAA4'}, Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='Si'}]

Process finished with exit code 0
```

Interfaz Comparator<?>

La interfaz Comparator se usa cuando se tiene más de un criterio para ordenar o cuando se usa en objetos que no son de una clase propia (double, String, Integer...)

Se implementa por medio de import java.util.Comparator

Tiene un método abstracto: int compare(objeto1, objeto2), el cual funciona exactamente igual que el compareto(objeto) del Comparable

```
import java.util.Comparator;
public class ComparadorPorNumero implements Comparator<Ejemplo> {
    @Override
    public int compare(Ejemplo o1, Ejemplo o2) {
        return o1.numero - o2.numero;
    }
}
```

```
import java.util.Arrays;
public class Teoria {
    public static void main(String[] args) {
        Ejemplo example1 = new Ejemplo(5, "AABE");
        Ejemplo example2 = new Ejemplo(7, "Si");
        Ejemplo example3 = new Ejemplo(7, "AAAA4");

        Ejemplo[] ejemplos = {example1, example2, example3};
        System.out.println(Arrays.toString(ejemplos));
        Arrays.sort(ejemplos);
        System.out.println(Arrays.toString(ejemplos));

        ComparadorPorNumero c = new ComparadorPorNumero();
        Arrays.sort(ejemplos, c);
        //c (el objeto) es el criterio sacado del comparador (compare)
        System.out.println(Arrays.toString(ejemplos));
    }
}
```

Resultado:

```
[Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='Si'}, Ejemplo{numero=7, nombre='AAAA4'}]
[Ejemplo{numero=7, nombre='AAAA4'}, Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='Si'}]
[Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='AAAA4'}, Ejemplo{numero=7, nombre='Si'}]
```

Invertir el orden: método reverse()

Cuando se quiera ordenar en sentido decreciente, se puede hacer de dos formas: Invertir manualmente el compare, o usando el método por defecto **reversed:**

```java
import java.util.Arrays;
import java.util.Comparator;

public class Teoria {
    public static void main(String[] args) {
        Ejemplo example1 = new Ejemplo(5, "AABE");
        Ejemplo example2 = new Ejemplo(7, "Si");
        Ejemplo example3 = new Ejemplo(7, "AAAA4");

        Ejemplo[] ejemplos = {example1, example2, example3};
        System.out.println(Arrays.toString(ejemplos));
        Arrays.sort(ejemplos);
        System.out.println(Arrays.toString(ejemplos));

        Comparator<Ejemplo> c = new ComparadorPorNumero();
        Arrays.sort(ejemplos, c);
       //c (el objeto) es el criterio sacado del comparador (compare)
        System.out.println(Arrays.toString(ejemplos));

        Comparator<Ejemplo> c2 = c.reversed();
        //Creo el comparador al revés con este comando
        Arrays.sort(ejemplos, c2);
        System.out.println(Arrays.toString(ejemplos));
    }
}
```

Resultados:

```
[Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='Si'}, Ejemplo{numero=7, nombre='AAAA4'}]
[Ejemplo{numero=7, nombre='AAAA4'}, Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='Si'}]
[Ejemplo{numero=5, nombre='AABE'}, Ejemplo{numero=7, nombre='AAAA4'}, Ejemplo{numero=7, nombre='Si'}]
[Ejemplo{numero=7, nombre='AAAA4'}, Ejemplo{numero=7, nombre='Si'}, Ejemplo{numero=5, nombre='AABE'}]
```

Herencia de clases:

Dentro de las relaciones de las clases, existe la relación de herencia, donde, para juntar clases similares en cuanto atributos y métodos, se hace una jerarquía de padre-hijo.

La clase padre tiene una serie de atributos y métodos que todos los hijos comparten, además de tener otros métodos que luego las clases hijo pueden editar.

Las clases padre (o clases abstractas) se declaran como [visibilidad] abstract class nombre{}

Las clases hijo tendrán la extensión **extends** nombreClase:

```
public abstract class subMain{ //Padre    }
public class Main extends subMain { //Hijo}
```

Atributos protegidos

Los atributos, además de tener la visibilidad de public y private, tienen la visibilidad protegida o **protected**. Esto hace que, a la vista de otras clases, no se pueda acceder, mientras que los hijos de esa clase sí pueden verlo y usarlo. He aquí un ejemplo de dos clases, una padre con protegidos y un hijo usándolos

```
public abstract class subMain{
    protected int num;
    protected String cad;
}
public class Main extends subMain {
    public void dar(){
        num = 1;
        cad = "No estamos declaradas aquí";
    }
}
```

Super

A la hora de hacer constructores, se tiene que tener en cuenta que la clase padre tiene valores que comparte con el hijo. Por ello, se hace un constructor en la clase padre, utilizándola con **super(valores)**, para dar los valores de interés en la clase hijo:

```
public abstract class subMain{
    protected long aLong;
    protected double dou;
    public subMain(long aLong, double dou) {
        this.aLong = aLong;
        this.dou = dou;
    }
}
public class Main extends subMain {
    int num;
    String str;
    boolean bool;
public Main(long aLong,double dou,int num, String str, boolean bool) {
        super(aLong, dou); //Para dar el valor deseado
        this.num = num;
        this.str = str;
        this.bool = bool;
    }
}
```

Uso de métodos abstractos

Los métodos abstractos son métodos creados en la clase padre, que serán sobrescritos en los hijos. No es como los métodos normales, que simplemente pasarían del padre al hijo, aquí en el padre solo se declara, y ya luego el hijo la crea a su antojo.

```
public abstract class subMain{
    public abstract void func1();
}
public class Main extends subMain {
    @Override
    public void func1() {
        System.out.println("Adios");
    }
}
```

Overriding o Sobrecarga:

Dicho de forma rápida, la sobrecarga de un método es un medio para cambiar la estructura de un método en una clase.

Un ejemplo es el método toString: Las clases tienen un método toString por defecto que escriba la dirección de memoria, PEEERO, con la sobrecarga, le damos el aspecto que nos interesa para luego usarlo.

Lo único que hay que hacer es escribir ***@Override*** encima de la función a modificar:

```
interface Inter{
    public int getNumero();
}

abstract class Abst{
    public void getNumero2(){
        System.out.println(1);
    }
    public abstract double getNumero3();
}

public class Main extends Abst implements Inter{
    @Override //Override de una función predefinida de Java
    public String toString(){
        return "Hola";
    }
    @Override //Override de una función de una interfaz
    public int getNumero(){
        return 1;
    }
    @Override //Override de una función heredada normal
    public void getNumero2(){
        System.out.println(5);
    }
    @Override //Override de una función abstracta
    public double getNumero3(){
    return 5.3;
    }
}
```