# CLASES ABSTRACTAS

En el mundo real estamos rodeados de conceptos, algunos de estos los podemos asociar con objetos tangibles como una silla o un ratón, y objetos intangibles como una empresa o una profesión, pero hay muchos otros conceptos que referencian a objetos abstractos y que solo nos sirven como eso, como concepto, no especifican ningún objeto en particular, sino al contrario agrupan a muchos objetos que comparten ciertas características y comportamientos, como por ejemplo el concepto "animal", si tratamos de graficarlo, dibujamos un gato, un perro, pero no podemos dibujar un animal como tal, ya que este es solo un concepto abstracto, que agrupa a un conjunto de objetos que están dentro de esa clasificación. La forma de representar ese objeto abstracto en la POO es a través de las **clases abstractas**.

Una clase abstracta es una clase que solo se utiliza para generar clases hijas, por lo que no puede instanciar objetos. En este tipo de clases tiene las siguientes características:

* Comienzan con la palabra reservada **abstract**
* Se pueden implementar atributos de la misma forma que una clase normal
* Se pueden implementar métodos (normales y/o abstractos)
* No se puede instanciar

Los métodos abstractos son aquellos que solo se declaran, y no se define el cuerpo de estos, delegando esa tarea a las clases hijas. Estos al igual que las clases abstractas se reconocen por la palabra reservada **abstract**.

Si una clase tiene al menos un método abstracto debe ser declarada como abstracta, ya que no provee la implementación para este.

Ejemplo:

El departamento de remuneraciones desea implementar una aplicación para calcular el sueldo de los empleados, clasificándose estos en 2 tipos: a Honorarios, y con Contrato, ambos comparten los mismos datos base, pero se diferencian en el cálculo del sueldo.

Por las características del problema planteado, se puede declarar una clase, que reúna las características comunes de los empleados (clase Empleado), pero debido a que el método de cálculo de sueldo es diferente para los distintos tipos de empleado, este debe ser abstracto para que cada clase hija lo implemente de acuerdo a sus necesidades, por lo tanto la clase también debe ser declarada como abstracta.

Declaración de la clase abstracta:

public **abstract** class Empleado { //Clase Abstracta private String nombre;

private int sueldoBase;

public String getNombre() { return nombre;

}

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

}

public int getSueldoBase() { return sueldoBase;

}

public void setSueldoBase(int sueldo) { this.sueldoBase = sueldo;

}

public **abstract** int calcSueldo();

// Método abstracto

}

Declaración de las clases hijas (Honorarios y Contrato):

public class Contrato extends Empleado{ public Contrato(){}

@Override

public int calcSueldo() {

int sb, desc, afp, salud, liquido;

sb = this.getSueldoBase();

afp = (int)(sb \* 0.13); // 13% prevision (AFP) salud = (int)(sb \* 0.07); // 7% Salud (Fonasa) desc = afp + salud;

liquido = sb ­ desc; return liquido;

}

}

public class Honorario extends Empleado{ public Honorario(){}

@Override

public int calcSueldo() { int sb, desc, liquido;

sb = this.getSueldoBase();

desc = (int)(sb \* 0.1); // Descuenta 10% provisional liquido = sb ­ desc;

return liquido;

}

}

Ejemplo de implementación del main():

public class EjemAbstracta {

public static void main(String[] args) { int aPagar;

Honorario emp\_1 = new Honorario(); Contrato emp\_2 = new Contrato();

emp\_1.setSueldoBase(120000); aPagar = emp\_1.calcSueldo();

System.out.println("Empleado a Honorarios: " + aPagar);

emp\_2.setSueldoBase(120000); aPagar = emp\_2.calcSueldo();

System.out.println("Empleado con Contrato: " + aPagar);

}

}

Si bien ambos trabajadores tienen el mismo sueldo base, por la naturaleza del empleado, obtienen un sueldo liquido distinto.

# INTERFACES

Se puede decir que una interfaz, al igual que una clase o una clase abstracta, define un tipo de dato, con la diferencia que en la interfaz no se pueden declarar variables de instancia o de clase (atributos), ni tampoco se pueden implementar métodos. Las interfaces tienen las siguientes características:

* Permite **declarar** métodos (ojo, solo declararlos)
* Permite declarar constantes (static final)
* Permiten heredar (extends) de otras interfaces
* Una clase puede implementar varias interfaces

Cuando una clase implementa cierta interfaz, debe proporcionar el código para cada uno de los métodos declarados en la interfaz, lo cual garantiza que la clase tendrá determinados métodos, con los cuales las demás clases podrán interactuar, independiente de cómo se implementen, ya que tendrán los mismos parámetros de entrada y la misma salida.

Las interfaces son útiles cuando objetos de distintas clases, para ciertas situaciones, operan de la misma forma (lo que viene siendo herencia múltiple). Por ejemplo, distintos aparatos electrónicos, si bien realizan distintas tareas, comparten la acción de encenderse y apagarse. Hay que tener en cuenta que se está hablando de clases que no comparten la misma clase padre, por lo que no podríamos heredar estos comportamientos comunes, como por ejemplo un tostador eléctrico (clase Electrodomésticos) es encendible, al igual que un automóvil (de la clase Vehículo).

Ejemplo:

Se creará la interfaz "Encendible", que luego será implementada por 2 clases totalmente distintas, pero que comparten las acciones de "encender" y "apagar", además se declaran 2 constantes para indicar el estado de "Encendido" o "Apagado":

public **interface** Encendible {

// Constantes

public static final char ENCENDIDO = 1; public static final char APAGADO = 0;

// Metodos

public String Encender(); public void Apagar();

}

Ahora se implementa la interfaz en la clase "Linterna"

public class Linterna **implements** Encendible { private int pilas;

private int estado;

public Linterna (int numero) { this.pilas = numero;

this.estado = **APAGADO**; // Constante de la interfaz

}

@Override

public String **Encender**() {

this.estado = **ENCENDIDO**; // Constante de la interfaz return("Se hizo la luz!!!");

}

@Override

public void **Apagar**() { this.estado = **APAGADO**;

}

// Constante de la interfaz

// Implementación de los demás metodos de la clase

}

public class Tostador **implements** Encendible { private int pilas;

private int estado;

public Tostador() { this.estado = **APAGADO**;

}

// Constante de la interfaz

@Override

public String **Encender**() {

this.estado = **ENCENDIDO**; // Constante de la interfaz return("Al rico pan tostado!!!");

}

@Override

public void **Apagar**() { this.estado = **APAGADO**;

}

// Constante de la interfaz

// Implementación de los demás metodos de la clase

}

(nota: es un tostador eléctrico)

Implementación de la clase persona, la cual utilizara las interfaces de los objetos de ambas clases para interactuar con ellos:

public class Persona{ public Persona() {}

public void EncenderAparato(Encendible e){ System.out.print(e.Encender()) // Metodo de la interfaz

}

}

Implementación del metodo main(), donde interactua el objeto de la clase "Persona" con los objetos de las clases "Linterna" y "Tostador", a través de su interfaz en común:

public class EjemploInterface {

public static void main(String[] args) { Linterna linterna1 = new Linterna(4); Tostador tostador1 = new Tostador(); Persona juanPerez = new Persona();

// Se ha cortado la luz y Juan Perez procede a encender la linterna juanPerez.EncenderAparato(linterna1);

// Juan Perez procede a prepararse desayuno con unas tostadas juanPerez.EncenderAparato(tostador1);

}

}

### **Problema:**

Se desea implementar dos juegos distintos.

El primero que permita ingresar el nombre de dos jugadores, tirar dos dados y mostrar cuál de los dos ganó.

El segundo juego permita a un jugador adivinar un número entre 1 y 100 que elige la computadora al azar.

Podemos decir que todo juego tiene un inicio, estado de juego propiamente dicho y un fin. Si queremos estandarizar estos dos juegos y posibles nuevos juegos podemos declarar una interface llamada 'Juego' que declare los métodos: iniciar, jugar y finalizar.

Luego cada vez que creemos un juego debe respetar esta interface implementándola.

Como sería el resultado esperado:

*Ingrese el nombre del primer jugador: Blanca*

*Ingrese el nombre del segundo jugador: Alex*

*Blanca le salió el valor 2*

*Alex le salió el valor 6*

*Gano Alex con un valor de 6*

*Adivina un número entre 1 y 100:5*

*El número a adivinar es mayor*

*Adivina un número entre 1 y 100:80*

*El número a adivinar es menor*

*Adivina un número entre 1 y 100:50*

*El número a adivinar es menor*

*Adivina un número entre 1 y 100:20*

*El número a adivinar es mayor*

*Adivina un número entre 1 y 100:35*

*El número a adivinar es menor*

*Adivina un número entre 1 y 100:30*

*El número a adivinar es menor*

*Adivina un número entre 1 y 100:25*

*El número a adivinar es menor*

*Adivina un número entre 1 y 100:22*

*El número a adivinar es menor*

*Adivina un número entre 1 y 100:21*

*Ganaste después de 9 intentos*

# TIPO DE DATOS ENUMERADOS

Los tipos enumerados sirven para restringir el contenido de una variable a una serie de valores predefinidos. Esto suele ayudar a reducir los errores en nuestro código. Estos valores predefinidos son el equivalente a las contantes, solo que, agrupadas bajo un concepto en común, evitando así, tener que definir muchas constantes por separado.

En Java, los tipos enumerados se pueden definir fuera o dentro de una clase, pero no dentro de un método. Otra ventaja que traen los tipos enum de Java, es que al ser una “especie de clase” podemos añadirles métodos, variables de instancia, constructores, etc… lo que los hace muy potentes. Al igual que una clase normal, no se puede invocar el constructor directamente, sino que a través de la creación de la instancia. También se puede utilizar la sobrecarga de constructores.

A continuación, les dejo un pequeño ejemplo que ilustra todos estos conceptos.

package ejemploenum;

public class EjemploEnum {

// Tipo enumerado simple

enum dias{lunes, martes, miercoles, jueves, viernes, sabado, domingo}

// Tipo enumerado complejo enum meses{

enero(31), febrero(28),marzo(31),abril(30),mayo(31),junio(30), julio(31),agosto(31),septiembre(30),octubre(31),noviembre(30), diciembre(31);

private int d; meses (int d){

this.d = d;

}

public int diasxmes(){ return this.d;

}

}

public static void main(String[] args) {

// TODO code application logic here dias diadelasemana = dias.lunes; meses m = meses.julio;

System.out.println("Hoy es dia "+ diadelasemana); System.out.println("El mes de "+m+" tiene "+m.diasxmes()+" dias");

}

}

# CLASES ANIDADAS

Las clases anidadas contienen en su interior la definición de otra clase. Son conocidas como clases anidadas (nested classes) o clases internas (outer class), sólo existen para el compilador, ya que éste las transforma en clases regulares separando la clase externa de la interna con el signo $. Cada clase interna puede heredar independientemente de una implementación. Por consiguiente, la clase interna no está limitada por el hecho de que la clase externa pueda estar ya heredando de una implementación. Hay cuatro tipos de clases internas.

## Clases internas no estáticas.

1. **Clases internas estáticas.**

## Clases internas locales a método.

1. **Clases anónimas.**

## Clases internas no estáticas.

Tiene acceso a todas las variables y métodos de la clase que la contiene. Aquí vemos como crear una instancia de una clase interna desde un método de la clase que la contiene.

public class Externa { void imprime() {

Interna min = new Interna(); min.muestra();

}

class Interna {

public void muestra(){ System.out.println("objeto interno");

}

}

public static void main (String []args) { Externa test = new Externa(); test.imprime();

}

}

Para crear una instancia del objeto interno directamente debemos hacerlo de esta manera. Externa.Interna in = ex.new Interna();

public class Externa { class Interna {

public void muestra(){ System.out.println("objeto interno");

}

}

public static void main (String []args) { Externa ex = new Externa(); Externa.Interna in= ex.new Interna(); in.muestra();

}

}

Se mostrará en pantalla “objeto interno”.

Dentro de la clase interna hace referencia a la instancia en ejecución de la clase interna, para hacer referencia a la instancia de la clase externa debemos utilizar la expresión Externa.this. Los modificadores de acceso que pueden tener son: final, abstract, strictfp, static, private, protected y public

## Clases internas estáticas

Las clases internas estáticas no pueden acceder a las variables y métodos no estáticos de la clase externa directamente sino a través de un objeto.

public class Externa { static class interna{

public void muestra(){

System.out.println("Imprimir clase estática");

}

}

}

class Test {

public static void main (String []args) { Externa.interna Test2= new Externa.interna(); Test2.muestra();

}

}

## Clases internas locales a método.

Se declaran dentro de un método en el cuerpo de la clase externa y sólo puede ser instanciada dentro del metódo después de la declaración de la clase interna.

public class Externa { void proceso() {

class Interna {

public void muestra(){

System.out.println("clase local a método");

}

}

Interna in= new Interna(); in.muestra();

}

public static void main (String []args) { Externa ex = new Externa(); ex.proceso();

}

}

Se mostrar en pantalla “clase local a método”.

Una clase local a método no puede acceder a las variables locales a no ser que sean declaradas como finales. Los modificadores de acceso que pueden tener son: abstract y final

## Clases anónimas

Una clase anónima se define en la misma línea donde se crea el objeto y debe ser una subclase de otra clase o implementar una interfaz. Como la clase no tiene nombre sólo se puede crear un único objeto, ya que las clases anónimas no pueden definir constructores.

class Test {

public void imprimir(){ System.out.println("Imprimir test");

}

}

public class Externa {

Test Test2 = new Test(){//aqui se define la clase anónima public void imprimir(){

System.out.println("Imprimir clase anónima");

}

}; //la definición de la clase termina con ";"

void proceso() {

Test2.imprimir();

}

public static void main (String []args) { Externa Test2= new Externa(); Test2.proceso();

}

}

Se mostrará en pantalla “Imprimir clase anónima”.

En una instancia de clase anónima sólo se puede acceder a los métodos que herede, sobrescriba o implemente de una interfaz.