TEMA 7. APLICACIÓN DE LOS MECANISMOS DE ABSTRACCIÓN: CLASES, PAQUETES, SUBCLASES E INTERFACES.

Índice

Jerarquía de Clases de Java. Object

En Java, a diferencia de otros lenguajes de programación orientada a objetos como C++, existe una clase raíz que es la clase **Object** de la que dependen todas las demás clases.

Cuando se crea una nueva clase, por ejemplo la clase Complejo que hemos creado, hereda de la clase Object aunque no lo hayamos indicado. Esto implica que todas las clases que se crean en Java incorporan la funcionalidad de la clase Object.

Biblioteca de Clases Predefinidas en Java

Una de las razones por las que la programación en Java se ha extendido tanto es que Java proporciona por defecto una Biblioteca de Clases Predefinidas que facilitan enormemente la creación de aplicaciones Java.

Por ejemplo, cuando creamos la clase Holamundo.java escribimos la sentencia ***System.out.println("Hola Mundo.");*** y, al ejecutar la aplicación, apareció un mensaje en la pantalla. Esto es posible porque Java nos proporciona la clase **System** que contiene la subclase **out** que contiene el método **println** que recibe un dato de tipo String y lo muestra por pantalla.

Otro ejemplo es la clase Leerentero.java que usa un objeto de la clase Scanner para leer un dato de tipo entero por teclado. Para poder usar la clase Scanner basta con importar desde la biblioteca predefinida de clases de Java a nuestra aplicación la clase Scanner que se encuentra en la clase java subclase util, para lo que escribimos ***import java.util.Scanner;***

La biblioteca de clases predefinidas de Java es muy extensa por lo que el conocimiento de todas las clases es prácticamente imposible.

Cuando se desarrolla una nueva aplicación en Java se realiza un estudio previo para determinar que clases vamos a necesitar. Si ya existen en la biblioteca de clases predefinidas de Java las usamos sin más y, si no existen, se crean desde cero o a partir de otras clases ya existentes.

Encapsulación

El mecanismo de **encapsulación** nos permite al definir una clase especificar a qué partes y de qué modo se puede acceder a los elementos de la misma. Esto minimiza muchos los problemas a la hora de trabajar con la clase. El usuario de la clase sólo tiene que conocer la parte pública de la clase (interfaz) para trabajar con ella.

Modificadores de Acceso

A la hora de definir un elemento de una clase podemos especificar modificadores de acceso. Estos modificadores de acceso indican que elementos tienen acceso al elemento en cuestión.

Existen varios modificadores de acceso:

public. Un elemento definido como public puede ser accedido por cualquier elemento pertenezca o no o la misma clase. Por ejemplo, si queremos que la clase Racional pueda ser accedida desde otra clase a la hora de definirla escribimos

```
public class Racional{…}
```

private. Un elemento definido como private sólo puede ser accedido por elementos que pertenezcan a la misma clase. Se suelen definir como private los atributos de la clase. Por ejemplo, si en la clase Racional queremos que los atributos sólo puedan ser accedidos por miembros de la misma clase escribimos

```
public class Racional{
 private int numerador;
 private int denominador;
}
```

protected. Un elemento definido como protected sólo puede ser accedido por elementos que pertenezcan a la misma clase o a alguna de las clases derivadas de ella. También pueden ser accedidas por las clases que estén en el mismo paquete. Lo veremos más adelante cuando estudiemos la herencia de clases.

default. Es el modificador por defecto si no se indica nada. Se le suele llamar **default** o **package-private**. Un elemento definido como default sólo puede ser accedido por elementos que pertenezcan al mismo paquete (package).

Herencia

La herencia nos permite desarrollar clases a partir de otra u otras previamente definidas. El mecanismo de herencia es uno de los principales motivos por los que han tenido tanto éxito los lenguajes Orientados a Objetos ya que una vez desarrollada una clase y comprobado su correcto funcionamiento la podemos reutilizar a la hora de crear otras clases.

Java proporciona una jerarquía de clases predefinida que nos permite utilizar objetos de clases ya existentes o utilizar las clases ya existentes como base para otras nuevas clases.

Cuando se produce una herencia a la clase de la que derivamos se le denomina **clase base** y a la clase que deriva (hereda) se le denomina **clase derivada**. Para indicar que una clase deriva de otra se utiliza la palabra reservada **extends**.

Por ejemplo, si queremos crear una nueva clase de nombre Alumno que hereda de la clase Persona escribimos

```
public class Alumno extends Persona{…}
```

Clases Derivadas. Efecto de Modificadores de Acceso en la Herencia de Clases

Las clases derivadas reciben todos los elementos de la clase base pero con sus modificadores de acceso modificados.

Clase Base	***Clase Derivada***
Public se convierte en	*Protected*
Protected se convierte en	*Private*
Private	*No se tienen permisos de acceso*

Esto quiere decir que los elementos de la clase derivada pueden acceder a los elementos public y protected de la clase base pero no a los elementos private.

Super

Super es un identificador que permite acceder a los elementos de la clase base desde una clase derivada.

Por ejemplo, si queremos llamar al método toString de la clase base desde la clase derivada escribimos

```
super.toString();
```

El método super se suele utilizar para llamar al constructor de la clase base desde una clase derivada.

Por ejemplo, si queremos llamar al constructor por defecto de la clase Persona desde el constructor por defecto de la clase Alumno escribimos

```
public Alumno(){
 super();
}
```

Ejercicios con Clases Derivadas

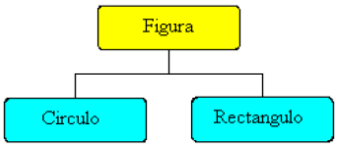
1. Crea la clase Alumno que hereda de la clase Persona. Un alumno es una persona que tiene además un dato de tipo String que se llama grupo. Crea los siguientes métodos para la clase Alumno: constructor por defecto, constructor copia, constructor personalizado, toString, getters and setters, equals, compareTo, leer.

2. Crea la clase AlumnoMain que comprueba el funcionamiento de la clase Alumno.

Jerarquía de Clases: Superclases y Subclases. Clases y Métodos Abstractos

La jerarquía de clases en Java se basa en los mecanismos de herencia. Para explicarla usaremos de ejemplo las figuras planas (rectángulo, círculo, triangulo…).

Clases y métodos abstractos
Consideremos las figuras planas rectángulo y círculo. Las dos figuras comparten características comunes como la posición de su centro, y el área de la figura, aunque el procedimiento para calcular dicha área sea distinto en cada caso. Entonces diseñamos una jerarquía de clases en java, tal que la **clase base** denominada **Figura**, tenga las características comunes y cada clase derivada las específicas. La relación jerárquica se muestra en la figura:



Clase Figura
La clase Figura es la que contiene las características comunes a dichas figuras concretas por tanto, no tiene forma ni tiene área. Esto lo expresamos declarando **Figura** como una **clase abstracta**, declarando el método **area** como **abstract**. Para ello escribimos

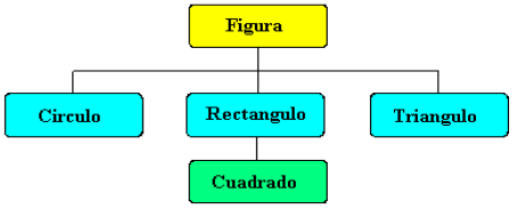
```
public abstract class Figura {
 …
 public abstract double area();
}
```

Las clases abstractas solamente se pueden usar como clases base para otras clases. No se pueden crear objetos pertenecientes a una clase abstracta.

La definición de la clase abstracta Figura, contiene la posición x e y de la figura particular, de su centro, y la función área, que se va a definir en las clases derivadas para cada una, ya que el método para calcularlo cambia. La definición de la clase abstracta Figura queda de la siguiente forma

```
public abstract class Figura {
    protected int x;
    protected int y;
    public Figura(int x, int y) {
        this.x=x;
        this.y=y;
    }
    public abstract double area();
}
```

Las clases derivadas heredan los atributos x e y de la clase base, y definen la función área, declarada abstract en la clase base Figura, ya que cada figura particular tiene una fórmula distinta para calcular su área. Por ejemplo, la clase derivada Rectangulo, tiene como datos, aparte de su posición (x, y) en el plano, sus dimensiones, es decir, su anchura ancho y altura alto.



Clase Rectangulo
La definición de la clase Rectangulo que deriva de Figura queda de la siguiente forma

```
public class Rectangulo extends Figura{
 protected double ancho;
 protected double alto;
 public Rectangulo(int x, int y, double ancho, double alto){
  super(x,y);
  this.ancho=ancho;
  this.alto=alto;
 }
 public double area(){
  return ancho*alto;
 }
}
```

La primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base, para ello se emplea la palabra reservada **super**. El constructor de la clase derivada llama al constructor de la clase base y le pasa las coordenadas del punto x e y. Después inicializa sus atributos ancho y alto. En la definición de la función área, se calcula el área del rectángulo como producto de la anchura por la altura, y se devuelve el resultado.

Clase Circulo
La definición de la clase Circulo que deriva de Figura queda de la siguiente forma

```
public class Circulo extends Figura{
 protected double radio;
 public Circulo(int x, int y, double radio){
  super(x,y);
  this.radio=radio;
 }
 @Override
 public double area(){
  return Math.PI*radio*radio;
 }
}
```

Como vemos, la primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base empleando la palabra reservada super. En la definición de la función área usamos la constante Math.PI.

Clase Triangulo
La clase derivada Triángulo, tiene como datos, aparte de su posición (x, y) en el plano, la base y la altura del triángulo. La definición de la clase Triangulo que deriva de Figura queda de la siguiente forma

```
public class Triangulo extends Figura{
 protected double base;
 protected double altura;
 public Triangulo(int x, int y, double base, double altura){
  super(x, y);
  this.base=base;
  this.altura=altura;
 }
```

```
 public double area(){
  return ((base*altura)/2);
 }
}
```

El constructor de la clase Triangulo llama al constructor de la clase Figura, le pasa las coordenadas x e y de su centro, y luego inicializa los atributos base y altura.

Clase Cuadrado
La clase Cuadrado es una clase derivada de Rectangulo, ya que un cuadrado tiene los lados iguales. El constructor solamente precisa de tres argumentos los que corresponden a la posición de la figura (x,y) y la longitud del lado.

```
public class Cuadrado extends Rectangulo{
 public Cuadrado(int x, int y, double lado){
  super(x, y, lado, lado);
 }
}
```

El constructor de la clase derivada llama al constructor de la clase base y le pasa la posición x e y de la figura, el ancho y alto que tienen el mismo valor. No es necesario redefinir una nueva función area. La clase Cuadrado hereda la función area definida en la clase Rectangulo.

Uso de la Jerarquía de Clases en Java

Para probar el funcionamiento de estas clases creamos una nueva clase de nombre **FiguraMain** en cuyo método main colocamos el código necesario. Por ejemplo, si queremos crear un objeto c de la clase Circulo situado en el punto (0, 0) y de 5.5 unidades de radio escribimos

```
Circulo c=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+c.area());
```

Veamos ahora, una forma alternativa, guardamos el valor devuelto por **new** al crear objetos de las clases derivadas en una variable f del tipo Figura (clase base).

```
Figura f=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+f.area()) ;
f=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+f.area());
```

Ejercicios de Jerarquía de Clases en Java

3. Añade a la clase Figura el método abstracto perimetro que calcula el perímetro de cualquier figura. Añade el método perimetro a todas las clases derivadas de Figura teniendo en cuenta que el perímetro de un círculo es su circunferencia y el del resto de figuras la suma de las longitudes de sus lados. En el caso del triángulo suponemos que es un triángulo rectángulo.

4. Modifica FiguraMain para probar el correcto funcionamiento de las modificaciones realizadas en la clase Figura y sus clases derivadas.

Clases y Métodos Finales

Para evitar que una clase o un método puedan ser heredados los debemos definir incluyendo la palabra reservada **final** en su definición. Todos los métodos incluidos dentro de una clase final son también finales (final).

Por ejemplo, si queremos que no se creen nuevas clases a partir de la clase Cuadrado debemos definir la clase Cuadrado como final. Para ello escribimos

```
final class Cuadrado extends Rectangulo{…}
```

Constructores de las Subclases

A la hora de crear el constructor de una clase derivada podemos usar como base el constructor de la clase base. Para ello usamos el identificador **super**.

Por ejemplo, si queremos llamar al constructor por defecto de la clase Persona desde el constructor por defecto de la clase Alumno escribimos

```
public Alumno(){
 super(); // llamo al constructor por defecto de Persona
this.grupo = "1DW3";
}
```

Ejercicios de Constructores de las Subclases

5. Añade a todas las clases de la jerarquía de Figura un constructor por defecto y un constructor copia (el constructor personalizado ya está creado).

6. Modifica FiguraMain para probar el correcto funcionamiento de las modificaciones realizadas en la clase Figura y sus clases derivadas.

Destructores de las Subclases

Cuando finaliza la ejecución de un objeto de una clase derivada, el recolector de basura invoca al destructor del objeto. Esto inicia una cadena de invocaciones a destructores, en donde el destructor de la clase derivada y los destructores de las clases bases directas e indirectas se ejecutan en orden inverso al que se ejecutaron los constructores, esto es, primero se ejecuta el destructor de la clase derivada y al final se ejecuta el destructor de la clase base ubicada en el nivel superior de la jerarquía. La ejecución de los destructores debe liberar todos los recursos que se hayan asignado al objeto durante su ejecución, antes de que el recolector de basura reclame la memoria de ese objeto.

Cuando el recolector de basura invoca al destructor de un objeto de una clase derivada, ese destructor realiza su tarea y después invoca al destructor de la clase base. El proceso se repite hasta que se invoca al destructor de la clase Object.

Para ello, al igual que en el caso de los constructores, usamos el identificador **super**.

Por ejemplo, si queremos crear un destructor para la clase Alumno que deriva de escribimos

```
public class Alumno extends Persona{
 protected void finalize(){
  super.finalize();
  //código que libera recursos
 }
}
```

Acceso a Métodos de la Superclase

Para acceder a los métodos de la superclase usamos la palabra reservada **super**. Por ejemplo, si queremos llamar al método **toString** de la clase base Persona desde la clase derivada Alumno escribimos

```
super.toString ();
```

Ejercicios de Acceso a Métodos de la Superclase

7. Añade a todas las clases de la jerarquía de Figura el método toString que permite mostrar por pantalla los objetos de las clases con un formato adecuado. El formato para Figura es **"Figura - X:"+x+" Y:"+y**. Para las demás será **"Clase - X:"+x+" Y:"+y+" Propiedades"**. Por ejemplo **"Circulo - X:"+x+" Y:"+y+" Radio:"+radio**.

8. Modifica FiguraMain para probar el correcto funcionamiento de las modificaciones realizadas en la clase Figura y sus clases derivadas.

Redefinición de Métodos de la Superclase

En determinadas ocasiones puede ser necesario redefinir los métodos de la clase base de tal modo que se ajusten al funcionamiento de la clase derivada.

Un ejemplo típico es la redefinición de los métodos **hashCode**, **equals**, y **compareTo** que se usan para comparar objetos del mismo tipo.

Por ejemplo, queda claro que no es lo mismo comparar dos objetos genéricos de tipo Object que dos objetos de la clase complejo.

Para redefinir un método de una clase dentro de otra se antepone @**Override** a la redefinición del método.

Por ejemplo, para redefinir el método equals de la clase Object dentro de la Complejo escribimos

```
@Override
public boolean equals(Object obj) {
 if (this == obj)   // si es el mismo objeto
  return true;
 if (obj == null)   // si el objeto es nulo
  return false;
 if (getClass() != obj.getClass()) // si los objetos no son de la misma clase
  return false;
 Complejo other = (Complejo) obj; // creo un objeto temporal
 if (Double.doubleToLongBits(imaginaria) != Double.doubleToLongBits(other.imaginaria))
 // comparo el primer campo
  return false;
 if (Double.doubleToLongBits(real) != Double.doubleToLongBits(other.real))
 // comparo el segundo campo
  return false;
 return true;
}
```

Ejercicios de Redefinición de Métodos de la Superclase

9. Añade a todas las clases de la jerarquía de Figura los métodos **hashCode**, **equals**, y **compareTo**.

10.Modifica FiguraMain para probar el correcto funcionamiento de las modificaciones realizadas en la clase Figura y sus clases derivadas.

Polimorfismo

El término **Polimorfismo** es una palabra de origen griego que significa “muchas formas”. Este término se utiliza en programación Orientada a Objetos para referirse a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. Es decir, pedir que objetos de tipos distintos reciban el mismo mensaje y respondan según el tipo de objeto que sean.
Si nos basamos en la jerarquía de clases de Figura, podríamos tener un ArrayList de nombre arrayListFiguras que contiene objetos de tipo Figura de los que queremos calcular su área. Para ellos bastaría con ir recorriendo arrayListFiguras y llamando al método área. Sería el compilador el que llamara al método área del objeto que corresponda (Circulo, Rectangulo, …)

```
for (int posicion=0; posicion < arrayListFiguras.size(); posicion ++){
 System.out.println("Área = " +arrayListFiguras.get(posicion).area());
}
```

Ejercicios de Polimorfismo

11.Crea la clase ArrayListFiguras que añade varios objetos distintos de la clase Figura a un ArrayList. De nombre arrayListFiguras y después recorre el ArrayList calculando el aréa de cada uno de los elementos.

12. Crea la clase **VentanaJListFiguras** que modifica VentanaJListNumeros para que la aplicación gestione objetos de la clase Figura y sus clases derivadas. Añade unos cuantos objetos de clases diferentes a la

lista. Crea el método **toString** de las clase Figura que muestra las coordenadas X y Y. Crea el método toString de las clases derivadas para que muestre un texto con el nombre de la clase antes de las coordenadas. Por ejemplo "***Rectangulo X:1 Y:1***"

13. Crea la clase **VentanaJListFigurasSeleccion** que modifica VentanaJListFiguras para que en función de la clase seleccionada en una ComboBox de nombre **cmbFiguras** aparezcan los datos necesarios para crear un objeto de esa clase. Los campos para meter las coordenadas siempre están visibles e incialmente valen 0. Si la clase seleccionada es Rectangulo aparecerán dos cajas de texto para meter el valor del alto y del ancho, si es Circulo aparecerá una caja de texto para meter el radio, si es Triangulo aparecerán dos cajas de texto para meter el valor de la base y de la altura y, si es Cuadrado aparecerá una caja de texto para meter el lado. Al pulsar btnInsertar se creará un nuevo objeto de la clase seleccionada en cmbFiguras y se insertará en la lista. Al pulsar btnBorrar se borrarán todos los elementos seleccionados en la lista, si es que hay alguno seleccionado. Al pulsar btnLimpiar se vaciará la lista, si todavía no está vacía. En la barra de estado aparecerá actualizado en todo momento el número de elementos en la lista y la suma de las áreas de todos los elementos de la lista.

Herencia Múltiple en Java. Interfaces

En Java no existe la herencia múltiple como tal. Todas las clases pueden heredar sólo de una clase. Para poder solucionar el problema de la herencia múltiple en Java se utilizan interfaces.

Un **interface** es una construcción similar a una clase abstracta en Java pero con algunas diferencias.

En el encabezado se usa la palabra clave interface en lugar de class o abstract class. Por ejemplo para definir un interface de nombre miInterface

```
public interface miInterface {…}
```

Todo método de un interface es abstracto y público sin necesidad de declararlo, es decir, no hace falta poner abstract public porque por defecto todos los métodos son abstract public. Por lo tanto un interface en Java no implementa ninguno de los métodos que declara: ninguno de sus métodos tiene cuerpo.

Las interfaces no tienen ningún constructor.

Un interfaz solo admite campos de tipo "public static final", es decir, campos de clase, públicos y constantes. No hace falta incluir las palabras public static final porque todos los campos serán tratados como si llevaran estas palabras. Recordemos que static equivalía a "de clase" y final a "constante". Las interfaces pueden ser un lugar interesante para declarar constantes que van a ser usadas por diferentes clases en nuestros programas.

Una clase puede derivar de un interface de la misma manera en que puede derivar de otra clase. No obstante, se dice que el interface se implementa (**implements**), no se extiende (extends) por sus subclases. Por tanto para declarar la herencia de un interface se usa la palabra clave implements en lugar de extends.

Por ejemplo, si tenemos una clase de nombre claseDerivada que deriva de la clase claseBase y que implementa el interface miInterface escribimos

```
public class claseDerivada extends claseBase implements miInterface {…}
```

Un ejemplo de interface que hemos usado es el interface Comparable que usamos en la clase Racional.

```
public class Racional implements Comparable< Racional>{…}
```

Ejercicios de Herencia Múltiple

14.Crea la clase ArrayListAlumnos que crea unos datos de Alumnos de prueba y los almacena en un ArrayList. Después muestra el contenido del array por pantalla.

15.Crea la clase ArrayListAlumnosOrdenado que modifica la clase ArrayListAlumnos para que los alumnos aparezcan por pantalla en orden según el método de ordenación por defecto de la clase Alumno.

16.Crea la clase ArrayListAlumnosOrdenadoDescendente que modifica la clase ArrayListAlumnosOrdenado para que los alumnos aparezcan por pantalla en orden inverso al método de ordenación por defecto de la clase Alumno.

17.Crea la clase ArrayListAlumnosOrdenadoGrupo que modifica la clase ArrayListAlumnosOrdenado para que los alumnos aparezcan por pantalla en orden ascendente (de menor a mayor) en función de su grupo. Si el grupo es igual los ordena como Personas en orden ascendente.

18.Crea la clase ArrayListAlumnosOrdenadoGrupoDescendente que modifica la clase ArrayListAlumnosOrdenadoGrupo para que los alumnos aparezcan por pantalla en orden descendente (de mayor a menor) en función de su grupo. Si el grupo es igual los ordena como Personas en orden descendente.

19.Crea la clase ArrayListAlumnosMenu que muestra un menú por pantalla con las siguientes opciones
- Añadir Alumno. Pide los datos de un Alumno y lo añade al array, si todavía no existe, en la posición que le corresponda en función de su dni.
- Buscar Alumno. Pide los datos de un Alumno y lo busca en el array. Si lo encuentra lo muestra y si no lo encuentra muestra un mensaje de error.
- Borrar Alumno. Pide los datos de un Alumno y lo elimina, si es que existe, del array. Si no lo encuentra muestra un mensaje de error.
- Listar Alumnos. Muestra todos los elementos del array por pantalla, si es que tiene elementos, ordenados en orden descendente en función de su grupo.
- Salir. Realiza las operaciones necesarias para la correcta finalización del programa.

20.Crea la clase **VentanaJListAlumnos** que modifica VentanaJListPersonas para que ahora la aplicación gestione objetos de la clase Alumno.

21.Crea la clase Empleado que hereda de la clase Persona. Un empleado es una persona que tiene además un dato de tipo String que se llama departamento, otro dato de tipo String que se llama puesto, y un dato de tipo double que se llama salario. La clase Empleado contará con los siguientes métodos:
- Un constructor por defecto que tenga como valores por defecto "" para los datos de tipo String y el valor 0.0 para el salario.
- Un constructor que permita dar valores sólo a todos los atributos de la clase Empleado.
- Un constructor que permita dar valores a los atributos de la clase Empleado y a los de la clase Persona.
- Un constructor copia.
- Setters y Getters necesarios para el correcto funcionamiento de la clase.
- Método toString que devuelve un String que añade al toString de la clase Persona que muestra los datos con el formato **dni + " " + nombre + " " + apellidos + " " + fechanacimiento** los datos propios del Empleado con el formato **departamento + " " + puesto + " " + salario**. Ej: **"**11111111A José María De Miguel Ajamil 1/1/2014 Informática Tutor 100.0**"**
- Métodos equals, hashCode. Dos Empleados son iguales si su dni es el mismo.
- Método compareTo que devuelve los valores correspondientes a comparar únicamente el campo dni de los objetos que se comparan.
- Método **leer** que lea por teclado usando un objeto de la clase Scanner los datos necesarios para crear un objeto de la clase controlando los posibles errores.

22.Crea la clase ArrayListEmpleados que recibe Empleados por teclado hasta que el usuario no quiera continuar. Una vez leídos los Empleados los va almacenando en un ArrayList. Cuando finaliza la introducción muestra el contenido del array por pantalla.

23.Crea la clase ArrayListEmpleadosOrdenado que modifica la clase ArrayListEmpleados para que los alumnos aparezcan por pantalla en orden ascendente (de menor a mayor) en función de su dni.

24.Crea la clase ArrayListEmpleadosOrdenadoDescendente que modifica la clase ArrayListEmpleadosOrdenado para que los alumnos aparezcan por pantalla en orden descendente (de mayor a menor) en función de su dni.

25.Crea la clase ArrayListEmpleadosOrdenadoSalario que modifica la clase ArrayListEmpleados para que los alumnos aparezcan por pantalla en orden ascendente (de menor a mayor) en función de su salario.

26.Crea la clase ArrayListEmpleadosOrdenadoSalarioDescendente que modifica la clase ArrayListEmpleadosOrdenado para que los alumnos aparezcan por pantalla en orden descendente (de mayor a menor) en función de su salario.

27.Crea la clase ArrayListEmpleadosMenu que muestra un menú por pantalla con las siguientes opciones

- Añadir Empleado. Pide los datos de un Empleado y lo añade al array, si todavía no existe, en la posición que le corresponda en función de su dni.
- Buscar Empleado. Pide los datos de un Empleado y lo busca en el array. Si lo encuentra lo muestra y si no lo encuentra muestra un mensaje de error.
- Borrar Empleado. Pide los datos de un Empleado y lo elimina, si es que existe, del array. Si no lo encuentra muestra un mensaje de error.
- Listar Empleados. Muestra todos los elementos del array por pantalla, si es que tiene elementos, ordenados en orden ascendente en función de su dni.
- Listar Empleados Salario. Muestra todos los elementos del array por pantalla, si es que tiene elementos, ordenados en orden descendente en función de su salario.
- Salir. Realiza las operaciones necesarias para la correcta finalización del programa.

28.Crea la clase abstracta Vehiculo que contiene un atributo de tipo String y de nombre nombreVehiculo, un atributo de tipo int y de nombre potenciaMotor (que vale 0 si el vehiculo no tiene motor) y otro atributo de tipo int y de nombre velocidadMaxima. La clase Vehiculo contará con los siguientes métodos:

- Un constructor por defecto que tenga como valores por defecto "" para los datos de tipo String y el valor 0 para los datos de tipo int.
- Un constructor que permita dar valor a un atributo de la clase Vehiculo.
- Un constructor que permita dar valores a todos los atributos de la clase Vehiculo.
- Un constructor copia.
- Getters y Setters necesarios para el correcto funcionamiento de la clase.
- Método toString que devuelve un String con el formato **"Nombre: " + nombreVehiculo + " Potencia: " + potenciaMotor + " CV Velocidad: " + velocidadMaxima + " Km/h"**
- Métodos abstractos de tipo void iniciarDesplazamiento y finalizarDesplazamiento

29.Crea las clases VehiculoTerrestre, VehiculoAcuatico y VehiculoAereo que derivan de Vehiculo y que tienen las siguientes características:

- VehiculoTerrestre. Contiene un atributo de tipo int y de nombre numeroRuedas. No se van a crear objetos de la clase VehiculoTerrestre. Sirve como base para otras clases.
- VehiculoAcuatico. No contiene ninguna característica adicional. No se van a crear objetos de la clase VehiculoAcuatico. Sirve como base para otras clases.
- VehiculoAereo. No contiene ninguna característica adicional. No se van a crear objetos de la clase VehiculoAereo. Sirve como base para otras clases.

30.Crea las siguientes clases que derivan de VehiculoTerrestre que tienen las siguientes características:

- Bici. Implementa un objeto de la clase Vehiculo con las características de una bici. Al iniciar el desplazamiento muestra el mensaje "Pedaleando…" y al finalizar el desplazamiento muestra el mensaje "Parado…"
- Moto. Implementa un objeto de la clase Vehiculo con las características de una moto. Al iniciar el desplazamiento muestra el mensaje "Dando gas…" y al finalizar el desplazamiento muestra el mensaje "Parado…"
- Coche. Implementa un objeto de la clase Vehiculo con las características de un coche. Al iniciar el desplazamiento muestra el mensaje "Arrancando…" y al finalizar el desplazamiento muestra el mensaje "Parado…"

31.Crea las siguientes clases que derivan de VehiculoAcuatico que tienen las siguientes características:

- Barco. Implementa un objeto de la clase Vehiculo con las características de un barco. Al iniciar el desplazamiento muestra el mensaje "Levando anclas…" y al finalizar el desplazamiento muestra el mensaje "Amarrado…"
- Velero. Implementa un objeto de la clase Vehiculo con las características de un velero. Al iniciar el desplazamiento muestra el mensaje "Izando velas…" y al finalizar el desplazamiento muestra el mensaje "Velas arriadas…"

32.Crea las siguientes clases que derivan de VehiculoAereo que tienen las siguientes características:

- Avion. Implementa un objeto de la clase Vehiculo con las características de un avión. Al iniciar el desplazamiento muestra el mensaje "Despegando…" y al finalizar el desplazamiento muestra el mensaje "Tomando tierra…"
- Helicoptero. Implementa un objeto de la clase Vehiculo con las características de un helicóptero. Al iniciar el desplazamiento muestra el mensaje "Encendiendo el rotor…" y al finalizar el desplazamiento muestra el mensaje "Rotor parado…"

33.Crea la clase VehiculoMain para probar el correcto funcionamiento de la jerarquía de clases dependientes de Vehiculo. Crea un ArrayList de objetos de la clase Vehiculo e introduce en él por lo menos un objeto de cada clase derivada. Comprueba el funcionamiento de los métodos de las clases derivadas.

34.Añade a la clase VehiculoMain una linea que muestre los datos de cada objeto. Para ello será necesario que cada clase de la jerarquía Vehiculo incorpore el método **toString**. Los datos se mostrarán con el formato **"Coche - Nombre: MiCoche Potencia: 100 CV Velocidad: 200Km/h Ruedas: 4"**

35.Crea la clase abstracta Animal que contiene un atributo de tipo String y de nombre nombreAnimal, y un atributo de tipo int de nombre esperanzaDeVida. Dispone de los métodos abstactos comer, reproducirse, y dormir. También dispone de los métodos iniciarDesplazamiento y finalizarDesplazamiento.

36.Crea las clases Vertebrado, e Invertebrado que derivan de Animal y que tienen las siguientes características:
- Vertebrado. No contiene ninguna característica adicional. No se van a crear objetos de la clase Vertebrado. Sirve como base para otras clases.
- Invertebrado. No se van a crear objetos de la clase Invertebrado. No contiene ninguna característica adicional. Sirve como base para otras clases.

37.Crea las siguientes clases que derivan de Vertebrado que tienen las siguientes características:
- Mamifero. Implementa un objeto de la clase Animal con las características de un mamífero. Incluye un atributo de tipo int y de nombre numeroDeMamas (que vale 2 por defecto). Sobreescribe el método comer para que muestre el mensaje "Dando de mamar…". Sobreescribe el método reproducirse para que muestre el mensaje "Reproduciéndose…". Sobreescribe el método dormir para que muestre el mensaje "Durmiendo…". Al iniciar el desplazamiento muestra el mensaje "Mamifero en movimiento…" y al finalizar el desplazamiento muestra el mensaje "Mamifero Detenido…"
- Ave. Implementa un objeto de la clase Animal con las características de un ave. Incluye un atributo de tipo int y de nombre numeroDeAlas (que vale 2 por defecto). Al iniciar el desplazamiento muestra el mensaje "Volando…" y al finalizar el desplazamiento muestra el mensaje "Posado…"
- Pez. Implementa un objeto de la clase Animal con las características de un pez. Al iniciar el desplazamiento muestra el mensaje "Nadando…" y al finalizar el desplazamiento muestra el mensaje "Descansando…"

38.Crea las siguientes clases que derivan de Invertebrado que tienen las siguientes características:
- Molusco. Implementa un objeto de la clase Animal con las características de un molusco. Al iniciar el desplazamiento muestra el mensaje "Molusco en movimiento…" y al finalizar el desplazamiento muestra el mensaje "Molusco Detenido…"
- Crustaceo. Implementa un objeto de la clase Animal con las características de un crustáceo. Al iniciar el desplazamiento muestra el mensaje "Crustáceo en movimiento…" y al finalizar el desplazamiento muestra el mensaje " Crustáceo detenido…"

39.Crea las siguientes clases que derivan de Mamifero que tienen las siguientes características:
- Humano. Implementa un objeto de la clase Mamifero con las características de un humano. Al iniciar el desplazamiento muestra el mensaje "Humano en movimiento…" y al finalizar el desplazamiento muestra el mensaje "Humano detenido…"
- Delfin. Implementa un objeto de la clase Mamifero con las características de un delfín. Al iniciar el desplazamiento muestra el mensaje "Delfín Nadando…" y al finalizar el desplazamiento muestra el mensaje " Delfín descansando…"

40.Crea la clase AnimalMain para probar el correcto funcionamiento de la jerarquía de clases dependientes de Animal. Crea un ArrayList de objetos de la clase Animal e introduce en él por lo menos un objeto de cada clase derivada. Comprueba el funcionamiento de los métodos de las clases derivadas.