UNIDAD 7
PROGRAMACIÓN ORIENTADA A OBJETOS II
EJERCICIOS

PROGRAMACIÓN
CFGS DAW

UD7. PROGRAMACIÓN ORIENTADA A OBJETOS II

EJERCICIO 1 - PRODUCTO

Supongamos una clase Producto con dos atributos:

- String nombre
- int cantidad

Implementa esta clase con un constructor (con parámetros) además de los getters y setters de sus dos atributos. No es necesario comprobar los datos introducidos.

A continuación, en el programa principal haz lo siguiente:

1. Crea 5 instancias de la Clase Producto.
2. Crea un ArrayList.
3. Añade las 5 instancias de Producto al ArrayList.
4. Visualiza el contenido de ArrayList utilizando Iterator.
5. Elimina dos elemento del ArrayList.
6. Inserta un nuevo objeto producto en medio de la lista.
7. Visualiza de nuevo el contenido de ArrayList utilizando Iterator.
8. Elimina todos los valores del ArrayList.

EJERCICIO 2 - ASTROS

Define una jerarquía de clases que permita almacenar datos sobre los planetas y satélites (lunas) que forman parte del sistema solar.

Algunos atributos que necesitaremos almacenar son:

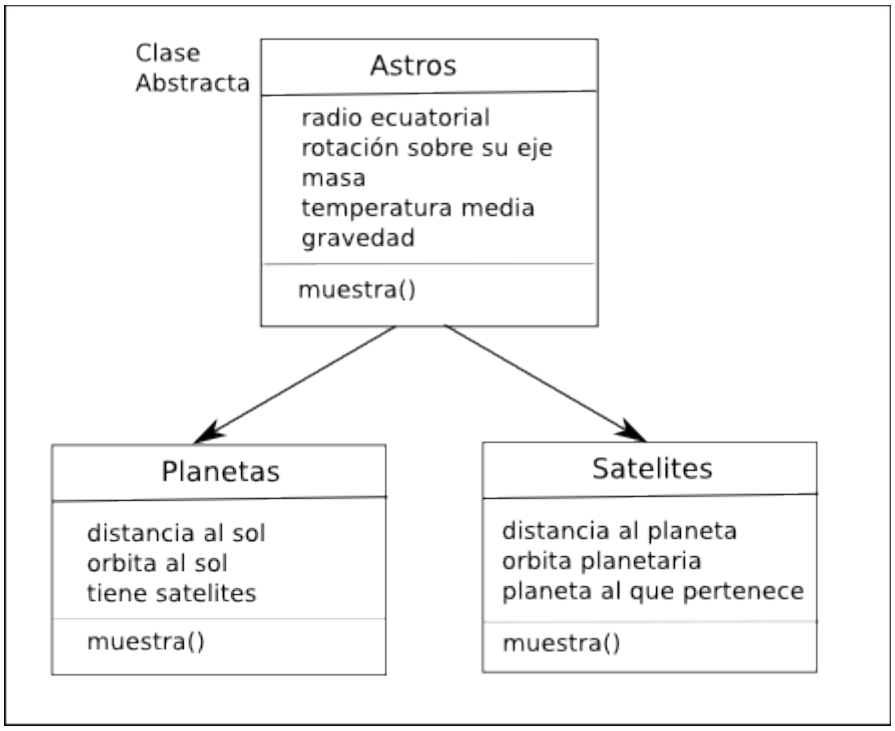
- Masa del cuerpo.
- Diámetro medio.
- Período de rotación sobre su propio eje.
- Período de traslación alrededor del cuerpo que orbitan.
- Distancia media a ese cuerpo.
- etc.

Define las clases necesarias conteniendo:

- Constructores.
- Métodos para recuperar y almacenas atributos.
- Método para mostrar la información del objeto.

Define un método, que dado un objeto del sistema solar (planeta o satélite), imprima toda la información que se dispone sobre el mismo (además de su lista de satélites si los tuviera).

El diagrama UML sería:



Una posible solución sería crear una lista de objetos, insertar los planetas y satélites (directamente mediante código o solicitándolos por pantalla) y luego mostrar un pequeño menú que permita al usuario imprimir la información del astro que elija.

EJERCICIO 3 - MASCOTAS

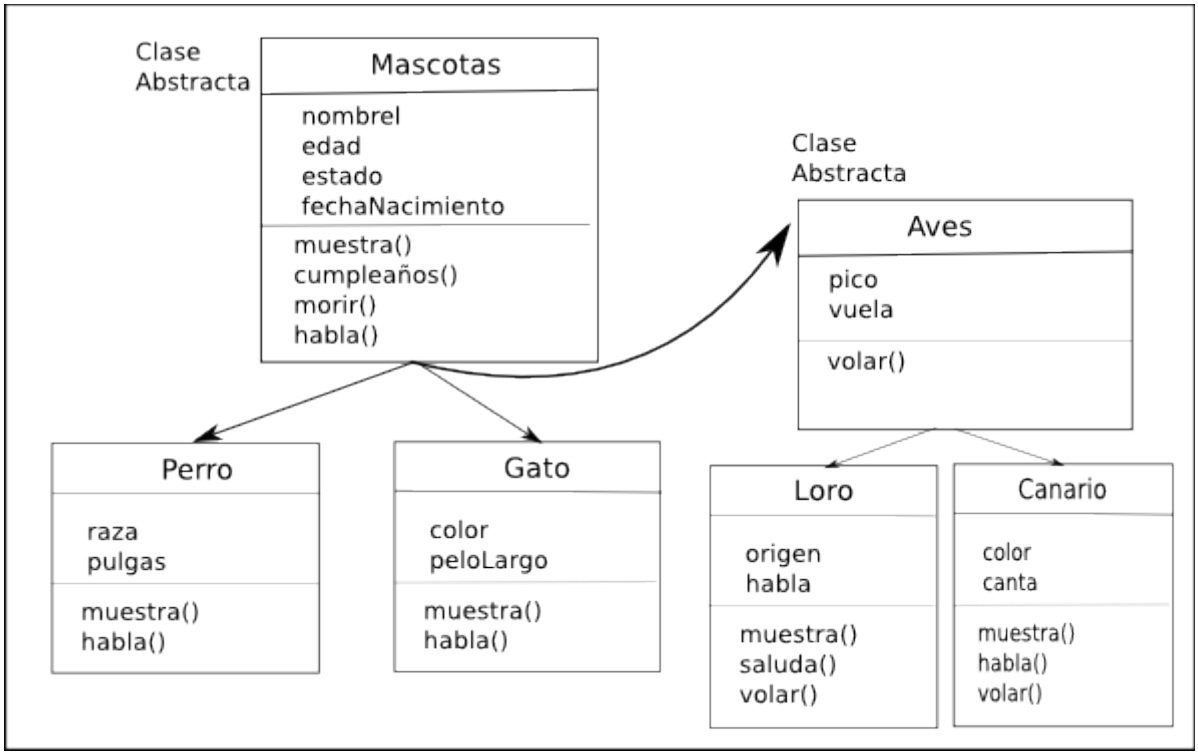
Implementa una clase llamada **Inventario** que utilizaremos para almacenar referencias a todos los animales existentes en una tienda de mascotas.

Esta clase debe cumplir con los siguientes requisitos:

- En la tienda existirán 4 tipos de animales: perros, gatos, loros y canarios.
- Los animales deben almacenarse en un ArrayList privado dentro de la clase **Inventario**.
- La clase debe permitir realizar las siguientes acciones:
  - Mostrar la lista de animales (solo tipo y nombre, 1 línea por animal).
  - Mostrar todos los datos de un animal concreto.
  - Mostrar todos los datos de todos los animales.
  - Insertar animales en el inventario.
  - Eliminar animales del inventario.
  - Vaciar el inventario.

Implementa las demás clases necesarias para la clase Inventario.

El diagrama UML sería:



EJERCICIO 4 – BANCO

Vamos a hacer una aplicación que simule el funcionamiento de un banco.

Crea una clase **CuentaBancaria** con los atributos: **iban** y **saldo**. Implementa métodos para:

- Consultar los atributos.
- Ingresar dinero.
- Retirar dinero.
- Traspasar dinero de una cuenta a otra.

Para los tres últimos métodos puede utilizarse internamente un método privado más general llamado **añadir(...)** que añada una cantidad (positiva o negativa) al saldo.

También habrá un atributo común a todas las instancias llamado **interesAnualBasico**, que en principio puede ser constante.

La clase tiene que ser ***abstracta*** y debe tener un método **calcularIntereses()** que se dejará sin implementar.

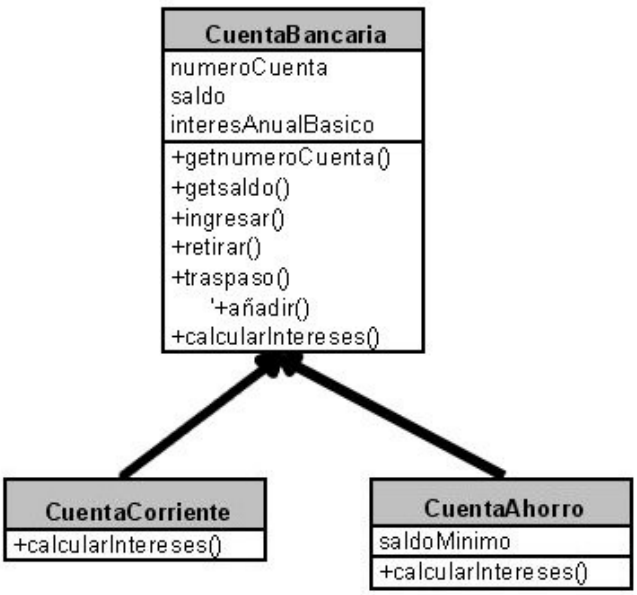
También puede ser útil implementar un método para mostrar los datos de la cuenta.

De esta clase heredarán dos subclases: **CuentaCorriente** y **CuentaAhorro**. La diferencia entre ambas será la manera de calcular los intereses:

- A la primera se le incrementará el saldo teniendo en cuenta el interés anual básico.
- La segunda tendrá una constante de clase llamada **saldoMinimo**. Si no se llega a este saldo el interés será la mitad del interés básico. Si se supera el saldo mínimo el interés aplicado será el doble del interés anual básico.

Implementa una clase principal con función main para probar el funcionamiento de las tres clases: Crea varias cuentas bancarias de distintos tipos, pueden estar en un ArrayList si lo deseas; prueba a realizar ingresos, retiradas y transferencias; calcula los intereses y muéstralos por pantalla; etc.

El diagrama UML sería:



EJERCICIO 5 – EMPRESA Y EMPLEADOS

Vamos a implementar dos clases que permitan gestionar datos de empresas y sus empleados.

Los **empleados** tienen las siguientes características:

- Un empleado tiene nombre, DNI, sueldo bruto (mensual), edad, teléfono y dirección.
- El nombre y DNI de un empleado no pueden variar.
- Es obligatorio que todos los empleados tengan al menos definido su nombre, DNI y el sueldo bruto. Los demás datos no son obligatorios.
- Será necesario un método para imprimir por pantalla la información de un empleado.
- Será necesario un método para calcular el sueldo neto de un empleado. El sueldo neto se calcula descontando del sueldo bruto un porcentaje que depende del IRPF. El porcentaje del IRPF depende del sueldo bruto anual del empleado (sueldo bruto x 12 pagas).**(*)**

Sueldo bruto anual	**IRPF**
Inferior a 12.000 €	20%
De 12.000 a 25.000 €	30%
Más de 25.000 €	40%

Por ejemplo, un empleado con un sueldo bruto anual de 17.000 € tendrá un 30% de IRPF. Para calcular su sueldo neto mensual se descontará un 30% a su sueldo bruto mensual.

Las **empresas** tienen las siguientes características:

- Una empresa tiene nombre y CIF (datos que no pueden variar), además de teléfono, dirección y empleados. Cuando se crea una nueva empresa esta carece de empleados.
- Serán necesarios métodos para:
  - Añadir y eliminar empleados a la empresa.
  - Mostrar por pantalla la información de todos los empleados.
  - Mostrar por pantalla el DNI, sueldo bruto y neto de todos los empleados.
  - Calcular la suma total de sueldos brutos de todos los empleados.
  - Calcular la suma total de sueldos netos de todos los empleados.

Implementa las clases Empleado y Empresa con los atributos oportunos, un constructor, los getters/setters oportunos y los métodos indicados. Puedes añadir más métodos si lo ves necesario. Estas clases no deben realizar ningún tipo de entrada por teclado.

Implementa también una clase Programa con una función main para realizar pruebas: Crear una o varias empresas, crear empleados, añadir y eliminar empleados a las empresas, listar todos los empleados, mostrar el total de sueldos brutos y netos, etc.

(*) El IRPF realmente es más complejo pero se ha simplificado para no complicar demasiado este ejercicio.

EJERCICIO 6 - VEHÍCULOS

Es muy aconsejable hacer el diseño UML antes de empezar a programar.

Debes crear varias clases para un software de una empresa de transporte. Implementa la jerarquía de clases necesaria para cumplir los siguientes criterios:

- Los vehículos de la empresa de transporte pueden ser terrestres, acuáticos y aéreos. Los vehículos terrestres pueden ser coches y motos. Los vehículos acuáticos pueden ser barcos y submarinos. Los vehículos aéreos pueden ser aviones y helicópteros.
- Todos los vehículos tienen matrícula y modelo (datos que no pueden cambiar). La matrícula de los coches terrestres deben estar formadas por 4 números y 3 letras. La de los vehículos acuáticos por entre 3 y 10 letras. La de los vehículos aéreos por 4 letras y 6 números.
- Los vehículos terrestres tienen un número de ruedas (dato que no puede cambiar).
- Los vehículos acuáticos tienen eslora (dato que no puede cambiar).
- Los vehículos aéreos tienen un número de asientos (dato que no puede cambiar).
- Los coches pueden tener aire acondicionado o no tenerlo.
- Las motos tienen un color.
- Los barcos pueden tener motor o no tenerlo.
- Los submarinos tienen una profundidad máxima.
- Los aviones tienen un tiempo máximo de vuelo.
- Los helicópteros tienen un número de hélices.
- No se permiten vehículos genéricos, es decir, no se deben poder instanciar objetos que sean vehículos sin más. Pero debe ser posible instanciar vehículos terrestres, acuáticos o aéreos genéricos (es decir, que no sean coches, motos, barcos, submarinos, aviones o helicópteros).
- El diseño debe obligar a que todas las clases de vehículos tengan un método imprimir() que imprima por pantalla la información del vehículo en una sola línea.

Implementa todas las clases necesarias con: atributos, constructor con parámetros, getters/setters y el método imprimir. Utiliza **bastracción** y **herencia** de la forma más apropiada.

Implementa también una clase Programa para hacer algunas pruebas: Instancia varios vehículos de todo tipo (coches, motos, barcos, submarinos, aviones y helicópteros) así como vehículos genericos (terrestres, acuáticos y aéreos). Crea un ArrayList y añade todos los vehículos. Recorre la lista y llama al método imprimir de todos los vehículos.

EJERCICIO 7 - FIGURAS

Implementa una **interface** llamada **iFigura2D** que declare los métodos:

- double perimetro(): Para devolver el perímetro de la figura
- double area(): Para devolver el área de la figura
- void escalar(double escala): Para escalar la figura (aumentar o disminuir su tamaño). Solo hay que multiplicar los atributos de la figura por la escala (> 0).
- void imprimir(): Para mostrar la información de la figura (atributos, perímetro y área) en una sola línea.

Existen 4 tipos de figuras.

- **Cuadrado**: Sus cuatro lados son iguales.
- **Rectángulo**: Tiene ancho y alto.
- **Triángulo**: Tiene ancho y alto.
- **Círculo**: Tiene radio.

Crea las 4 clases de figuras de modo que implementen la interface iFigura2D. Define sus métodos.

Crea una clase ProgramaFiguras con un main en el que realizar las siguientes pruebas:

1. Crea un ArrayList figuras.
2. Añade figuras de varios tipos.
3. Muestra la información de todas las figuras.
4. Escala todas las figuras con escala = 2.
5. Muestra de nuevo la información de todas las figuras.
6. Escala todas las figuras con escala = 0.1.
7. Muestra de nuevo la información de todas las figuras.