Ejercicios 15-Oct-2025

Programación Orientada a Objetos – Jerarquía Animal

1. Clase base Animal

Crea una clase `Animal` con un campo `nombre` y un método `mover()`.

El método debe imprimir `"El animal se mueve"`.

Pista: Usa un constructor `Animal(this.nombre);` y un método `void mover() { ... }`.

---

2. Subclases Mammal, Bird y Fish

Crea tres clases que hereden de `Animal`: `Mammal`, `Bird` y `Fish`.

Cada una debe redefinir el método `mover()` para imprimir un mensaje distinto.

Pista:

```
class Mammal extends Animal {
  Mammal(super.nombre);
  @override
  void mover() ⇒ print("$nombre camina sobre la tierra");
}
```

---

3. Constructores con super()

Haz que las subclases `Mammal`, `Bird` y `Fish` reciban el nombre en su constructor y lo pasen al de `Animal` usando `super(nombre)`.

En `main()`, crea un `Bird` y un `Fish` e imprime su movimiento.

---

4. Herencia múltiple con mixins

Crea tres mixins:

- `Walk` con un método `caminar()`.
- `Swim` con un método `nadar()`.
- `Fly` con un método `volar()`.

Luego, combínalos en clases concretas:

- `Bat` hereda de `Mammal` e implementa `Walk` y `Fly`.
- `Dolphin` hereda de `Mammal` e implementa `Swim`.

Pista: Usa `class Bat extends Mammal with Walk, Fly`.

---

5. **Comportamientos múltiples**

Crea un `Duck` que herede de `Bird` e implemente los mixins `Walk`, `Swim` y `Fly`.

En `main()`, instancia un `Duck` y llama a sus tres comportamientos.

Pista:

```
class Duck extends Bird with Walk, Swim, Fly {}
```

---

6. **Encapsulación y propiedades privadas**

Agrega a la clase `Animal` un campo privado `_edad` y un getter y setter para controlarlo.

El setter debe impedir que se asigne una edad negativa.

Pista:

```
int get edad ⇒ _edad;
set edad(int e) {
  if (e >= 0) _edad = e;
}
```

---

7. **Polimorfismo**

Crea una lista `List<Animal>` con diferentes tipos (`Cat`, `Duck`, `Shark`, `Bat`).

Recorre la lista y llama al método `mover()` en cada objeto.

Pista: Usa `for (var a in animales) { a.mover(); }`.

---

8. Sobrescritura de `toString()`

En cada clase concreta (`Cat`, `Duck`, `Dolphin`, etc.), sobrescribe `toString()` para mostrar una descripción personalizada.

Pista:

```
@override
String toString() ⇒ "Soy un $nombre que puede volar y caminar";
```

---

9. Clase abstracta Animal

Convierte `Animal` en una **clase abstracta** que defina el método `mover()` sin implementación.

Haz que las subclases implementen el método obligatoriamente.

Pista:

```
abstract class Animal {
  void mover();
}
```

---

10. Ejercicio integrador – Zoológico

Crea un programa que:

1. Declare las clases de toda la jerarquía (`Animal`, `Mammal`, `Bird`, `Fish`).
2. Incluya mixins `Walk`, `Swim`, `Fly`.
3. Cree instancias de `Cat`, `Dolphin`, `Duck`, `Bat`, `Shark`, `FlyingFish`.

4. Almacene todos en una lista y muestre sus capacidades.

Pista: Usa `is` para comprobar habilidades:

```
if (animal is Fly) print("${animal.nombre} puede volar");
```

Programación Orientada a Objetos – Ejemplos alternativos (Dart)

1. Clase base con constructor y atributos

Crea una clase `Animal` con los atributos `nombre` y `tipo`.

El constructor debe inicializar ambos campos, y el método `descripcion()` debe imprimir un texto con esa información.

Pista:

```
Animal(this.nombre, this.tipo);
```

2. Subclases con comportamiento específico

Crea una subclase `Perro` que extienda `Animal` y añada un atributo `raza`.

Sobrescribe el método `descripcion()` para incluir la raza.

Pista: Usa `@override` y `super.descripcion()` si lo necesitas.

3. Uso de getters y setters

Crea una clase `Ave` con un campo privado `_alturaVuelo`.

Agrega un getter `alturaVuelo` y un setter que **solo permita valores positivos**.

Pista:

```
set alturaVuelo(double valor) {
  if (valor >= 0) _alturaVuelo = valor;
}
```

---

4. **Herencia + método extra**

Crea una clase `Pez` que extienda `Animal` y agregue un método `nadar()`.

En `main()`, crea un `Pez` y muestra su nombre y el mensaje de `nadar()`.

Pista:

```
void nadar() ⇒ print("$nombre está nadando.");
```

---

5. **Constructores nombrados**

Crea una clase `Gato` con dos constructores:

- `Gato(this.nombre, this.edad)`
- `Gato.bebe(this.nombre)` que asigne `edad = 0`.

Pista: Usa inicialización por lista:

```
Gato.bebe(this.nombre) : edad = 0;
```

---

6. **Clase abstracta y herencia**

Crea una clase abstracta `Vehiculo` con el método `mover()`.

Crea dos clases `Coche` y `Avion` que la extiendan e implementen el método con diferentes mensajes.

Pista:

```
abstract class Vehiculo {
  void mover();
}
```

---

7. **Polimorfismo**

Declara una lista `List<Vehiculo>` que contenga instancias de `Coche` y `Avion`.

Recórrela con un `for-in` y llama al método `mover()` en cada elemento.

Pista: Muestra distintos mensajes según la clase.

---

8. **Composición (objeto dentro de otro)**

Crea una clase `Corazon` con un método `latir()`.

Crea una clase `Persona` que contenga un `Corazon` como propiedad y un método `vivir()` que llame a `corazon.latir()`.

Pista:

```
class Persona {
  Corazon corazon = Corazon();
  void vivir() ⇒ corazon.latir();
}
```

---

9. **Mixins**

Crea mixins `Caminar`, `Volar` y `Nadar`, cada uno con su método.

Crea clases que los combinen:

- `Pingüino` con `Caminar` y `Nadar`.
- `Murciélago` con `Caminar` y `Volar`.
- `Pato` con los tres.

Pista:

```
class Pato with Caminar, Volar, Nadar {}
```

---

10. **Ejercicio integrador – Zoológico**

Crea una estructura completa:

- Clase abstracta `Animal` con método `mover()`.
- Subclases `Mamifero`, `Ave`, `Pez`.
- Clases concretas (`Gato`, `Murciélago`, `Delfin`, `Pato`, `Tiburón`) que hereden de las anteriores.
- Implementa distintos comportamientos (caminar, volar, nadar).
- Crea una lista de animales y muestra qué puede hacer cada uno.

Pista: Usa condicionales con `is`:

```
if (animal is Volar) print("${animal.nombre} puede volar");
```