Clase 11 y 12. Introducción al POO

TRABAJADORES INFORMÁTICOS organizados en AGC

Ministerio de Trabajo, Empleo y Seguridad Social Argentina

Objetos

- Muchas veces en programación, interesa representar objetos (del mundo real o no) para poder utilizarlos.
- Por ejemplo: Para hacer un juego de un zoológico, necesitamos representar a los animales. Vamos a arrancar por uno: el Carpincho.

Objetos

- Que **atributos** tiene un carpincho? Qué elementos lo distinguen?
  - Color: Marrón
  - Patas: 4
  - Orejas: 2 chiquitas
  - Energía
  - Hambre
- Qué **comportamiento (métodos)** tiene? Que puede hacer?
  - Comer
  - Nadar
  - Caminar
  - Dormir

Objetos

- Para representar objetos, se eligen los atributos y métodos que nos interesan. Para nuestro caso un carpincho va a tener:
  - Un color
  - Un nivel de energía que se reduce al caminar y se recupera al dormir.
  - Un nivel de hambre que aumenta al caminar y se recupera al comer.

Objetos

- Para representar objetos, se eligen los atributos y métodos que nos interesan. Para nuestro caso un carpincho va a tener:
  - Un color
  - Un nivel de energía que se reduce al caminar y se recupera al dormir.
  - Un nivel de hambre que aumenta al caminar y se recupera al comer.
- Tomarse unos minutos para pensar cómo lo puede implementar.

Objetos

```
carpincho = crear_carpincho("marron")
print(carpincho)
# [10, 0, 'marron']
comer(carpincho)
caminar(carpincho)
caminar(carpincho)
caminar(carpincho)
print(carpincho)
# [7, -2, 'marron']
dormir(carpincho)
print(carpincho)
# [7, -2, 'marron']
```

```
ENERGIA_INICIAL = 10
HAMBRE_INICIAL = 0
POS_ENERGIA = 0
POS_HAMBRE = 1

ENERGIA_POR_CAMINAR = 1
ENERGIA_POR_DORMIR = 10
HAMBRE_POR_CAMINAR = 1
HAMBRE_POR_COMER = 5

def crear_carpincho(color):
    # Primer elemento la energia inicial
    # Segundo elemento el hambre inicial
    # Tercer elemento el color
    return [ENERGIA_INICIAL, HAMBRE_INICIAL, color]

def comer(carpincho):
    carpincho[POS_HAMBRE] -= HAMBRE_POR_COMER

def dormir(carpincho):
    carpincho[POS_ENERGIA] += ENERGIA_POR_DORMIR

def caminar(carpincho):
    carpincho[POS_ENERGIA] -= ENERGIA_POR_CAMINAR
    carpincho[POS_HAMBRE]  += HAMBRE_POR_CAMINAR
```

Objetos

- Cuales son sus problemas tiene esta implementación? Cuales son sus ventajas y desventajas?

```
ENERGIA_INICIAL = 10
HAMBRE_INICIAL = 0
POS_ENERGIA = 0
POS_HAMBRE = 1

ENERGIA_POR_CAMINAR = 1
ENERGIA_POR_DORMIR = 10
HAMBRE_POR_CAMINAR = 1
HAMBRE_POR_COMER = 5

def crear_carpincho(color):
  # Primer elemento la energia inicial
  # Segundo elemento el hambre inicial
  # Tercer elemento el color
  return [ENERGIA_INICIAL, HAMBRE_INICIAL, color]

def comer(carpincho):
  carpincho[POS_HAMBRE] -= HAMBRE_POR_COMER

def dormir(carpincho):
  carpincho[POS_ENERGIA] += ENERGIA_POR_DORMIR

def caminar(carpincho):
  carpincho[POS_ENERGIA] -= ENERGIA_POR_CAMINAR
  carpincho[POS_HAMBRE]  += HAMBRE_POR_CAMINAR
```

Objetos

- Qué problemas tiene esta implementación?
  - Un carpincho es un simple vector -> La persona que va a utilizar un carpincho puede modificar el vector sin problema. (pasa lo mismo si usamos un diccionario por ejemplo).
  - Las funciones tienen que recibir el carpincho.
  - No podemos extender la funcionalidad fácilmente.
  - Cuando queramos agregar más animales, vamos a tener que copiar mucho código ya que otros animales pueden tener atributos y comportamientos similares.

```
ENERGIA_INICIAL = 10
HAMBRE_INICIAL = 0
POS_ENERGIA = 0
POS_HAMBRE = 1

ENERGIA_POR_CAMINAR = 1
ENERGIA_POR_DORMIR = 10
HAMBRE_POR_CAMINAR = 1
HAMBRE_POR_COMER = 5

def crear_carpincho(color):
  # Primer elemento la energia inicial
  # Segundo elemento el hambre inicial
  # Tercer elemento el color
  return [ENERGIA_INICIAL, HAMBRE_INICIAL, color]

def comer(carpincho):
  carpincho[POS_HAMBRE] -= HAMBRE_POR_COMER

def dormir(carpincho):
  carpincho[POS_ENERGIA] += ENERGIA_POR_DORMIR

def caminar(carpincho):
  carpincho[POS_ENERGIA] -= ENERGIA_POR_CAMINAR
  carpincho[POS_HAMBRE]  += HAMBRE_POR_CAMINAR
```

Objetos

- Vamos a implementarlo utilizando el POO. Vamos a crear una **clase** “carpincho” que pueda generar **instancias** de carpinchos.
- Por convención las clases se escriben con la primer letra en mayúscula.

```
class carpincho:
  def __init__(self, color):
    self.color = color
    self.hambre = HAMBRE_INICIAL
    self.energia = ENERGIA_INICIAL

  def comer(self):
    # ???

  def dormir(self):
    # ???

  def caminar(self):
    # ???
```

Objetos

```
carpincho = Carpincho("marron")
print(carpincho)
# Hambre es 0 y energia es 10
carpincho.caminar()
carpincho.caminar()
carpincho.caminar()
carpincho.caminar()
carpincho.comer()
print(carpincho)
# Hambre es -1 y energia es 6
carpincho.dormir()
print(carpincho)
# Hambre es -1 y energia es 16
```

```
ENERGIA_INICIAL = 10
HAMBRE_INICIAL = 0
(constant) ENERGIA_POR_CAMINAR: Literal[1]
ENERGIA_POR_CAMINAR = 1
ENERGIA_POR_DORMIR = 10
HAMBRE_POR_CAMINAR = 1
HAMBRE_POR_COMER = 5

class Carpincho:
  def __init__(self, color):
    self.color = color
    self.hambre = HAMBRE_INICIAL
    self.energia = ENERGIA_INICIAL

  def comer(self):
    self.hambre -= HAMBRE_POR_COMER

  def dormir(self):
    self.energia += ENERGIA_POR_DORMIR

  def caminar(self):
    self.energia -= ENERGIA_POR_CAMINAR
    self.hambre += HAMBRE_POR_CAMINAR

  def __str__(self):
      return "Hambre es " + str(self.hambre) + " y energia es " + str(self.energia)
```

Objetos

- Cuales son sus problemas tiene esta implementación? Cuales son sus ventajas y desventajas?

```
ENERGIA_INICIAL = 10
HAMBRE_INICIAL = 0
 (constant) ENERGIA_POR_CAMINAR: Literal[1]
ENERGIA_POR_CAMINAR = 1
ENERGIA_POR_DORMIR = 10
HAMBRE_POR_CAMINAR = 1
HAMBRE_POR_COMER = 5

class Carpincho:
  def __init__(self, color):
    self.color = color
    self.hambre = HAMBRE_INICIAL
    self.energia = ENERGIA_INICIAL

  def comer(self):
    self.hambre -= HAMBRE_POR_COMER

  def dormir(self):
    self.energia += ENERGIA_POR_DORMIR

  def caminar(self):
    self.energia -= ENERGIA_POR_CAMINAR
    self.hambre += HAMBRE_POR_CAMINAR

  def __str__(self):
      return "Hambre es " + str(self.hambre) + " y energia es " + str(self.energia)
```

Objetos

- Una **clase** es una generadora de **instancias**.
- Los **atributos** de una instancia componen el **estado** de una instancia.
- Los **métodos** definen el comportamiento de una instancia.
- Como siempre, la clase no tiene exponer para su consumidor su estructura interna, ello implica no exponer nunca sus atributos (al menos directamente).

Clase

Instancia. Tiene un estado!

Objetos

- Todos los métodos de las clases tienen que recibirse a sí mismo (**self**).
- El método **__init__** es el **constructor** de la clase. Es llamado para la **creacion** e **inicializacion** de la instancia y la retorna.

```
def __init__(self, color):
    self.color = color
    self.hambre = HAMBRE_INICIAL
    self.energia = ENERGIA_INICIAL
```

Ejercicio

Tiempo ⏱️

- Diseñar y programar una clase *Bicicleta* que tenga **ruedas**, un **color**, una **velocidad actual** y pueda aumentar y disminuir la velocidad. Atención: La velocidad no debería disminuir de 0 ni ser mayor a 80

Funciones comunes

- Cómo funcionan **str**, **len**, **+**, etc ?
- Son funciones que a las que se les puede pasar por parámetro un texto, una lista, etc y funciona… Como **str** para saber convertir a texto un entero, una lista o cualquier tipo?

Funciones comunes

- Cómo funcionan **str**, **len**, **+**, etc ?
- Son funciones a las cuales se les puede pasar por parámetro un texto, una lista, etc y funciona… Como hace **str** para saber convertir a texto un entero, una lista o cualquier tipo?
- No es magia! Cada función de las mencionadas llama a una función particular de las clase pasada por parámetro.

Funciones comunes

- str(instancia) ⇔ instancia.**__str__**
- len(instancia) ⇔ instancia.**__len__**
- instancia1 **+** instancia2 ⇔ instancia1.**__add__**(instancia2)
- instancia1 **-** instancia2 ⇔ instancia1.**__sub__**(instancia2)
- instancia1 ***** instancia2 ⇔ instancia1.**__mul__**(instancia2)
- instancia1 **/** instancia2 ⇔ instancia1.**__div__**(instancia2)
- instancia1 **==** instancia2 ⇔ instancia1.**__eq__**(instancia2)

Ejercicio

Tiempo ⏱️

Implementar los métodos a la bicicleta:

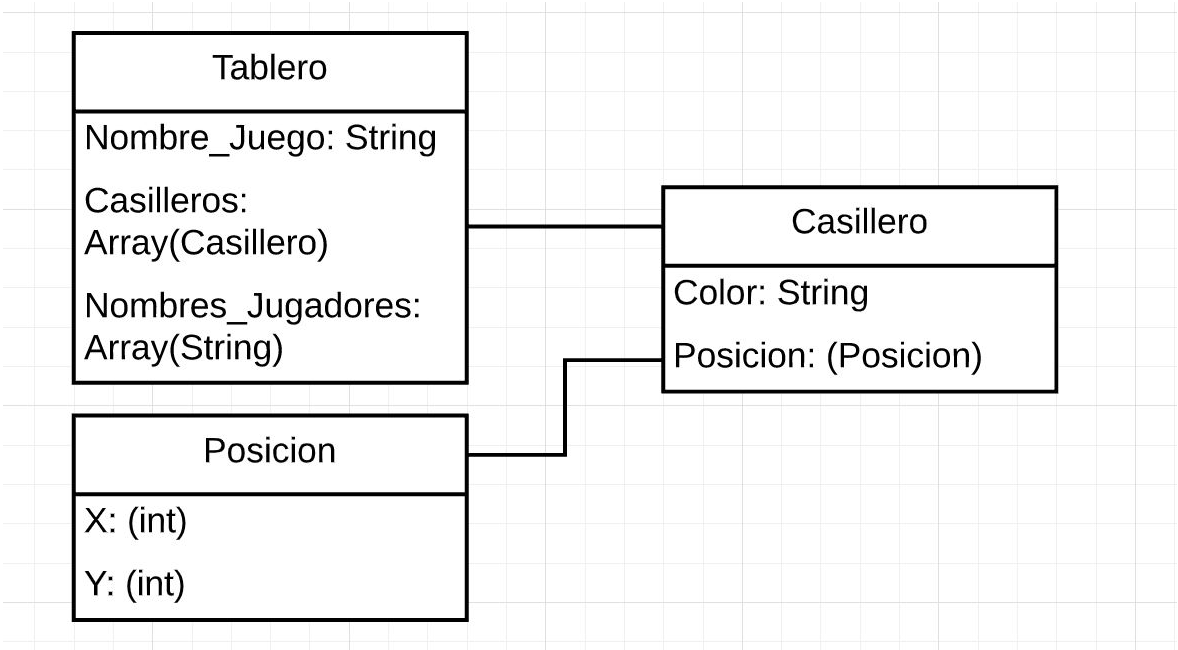
- str: Imprime con un mensaje amigable la cantidad de ruedas y la velocidad actual
- eq: Permita ver si dos bicicletas son exactamente iguales
- multiplicar: retorna una nueva bicicleta donde la cantidad de ruedas es la multiplicación de las ruedas de las bicicletas multiplicadas.

Consumidor vs Implementador

- Se llama **consumidor** de una clase, función, libreria, etc. al rol de la persona que utiliza dicha entidad.
- Se llama **implementador** de una clase, función, libreria, etc. al rol de la persona que implementa dicha entidad.
- El implementador es el que realmente programa la entidad y expone al consumidor la interfaz para que el consumidor sepa cómo utilizarla.
- El implementador **NUNCA** debe exponer cómo está implementada la clase, función, libreria, etc. El consumidor **NUNCA** debe acceder a la estructura interna (por ejemplo acceder directamente a una propiedad).
- Por ejemplo, nosotros solemos ser los consumidores de librería y clases ya implementadas, como las listas y los diccionarios. Nosotros desconocemos totalmente como funcionan por detrás, simplemente sabemos cuales métodos tiene por la interfaz y lo usamos.

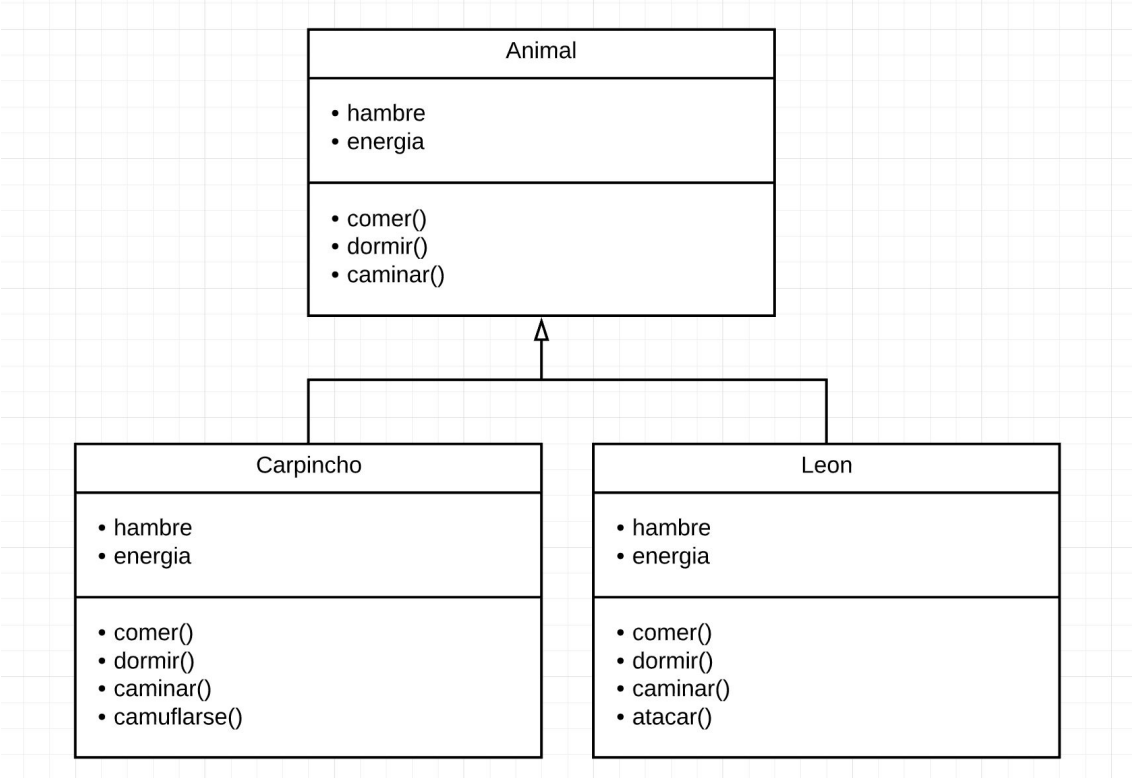
Composición

- Se llama a composición cuando una **propiedad** de una clase es otra clase.
- El tablero está **compuesto** por casilleros.
- Cada casillero está **compuesto** por una posición.



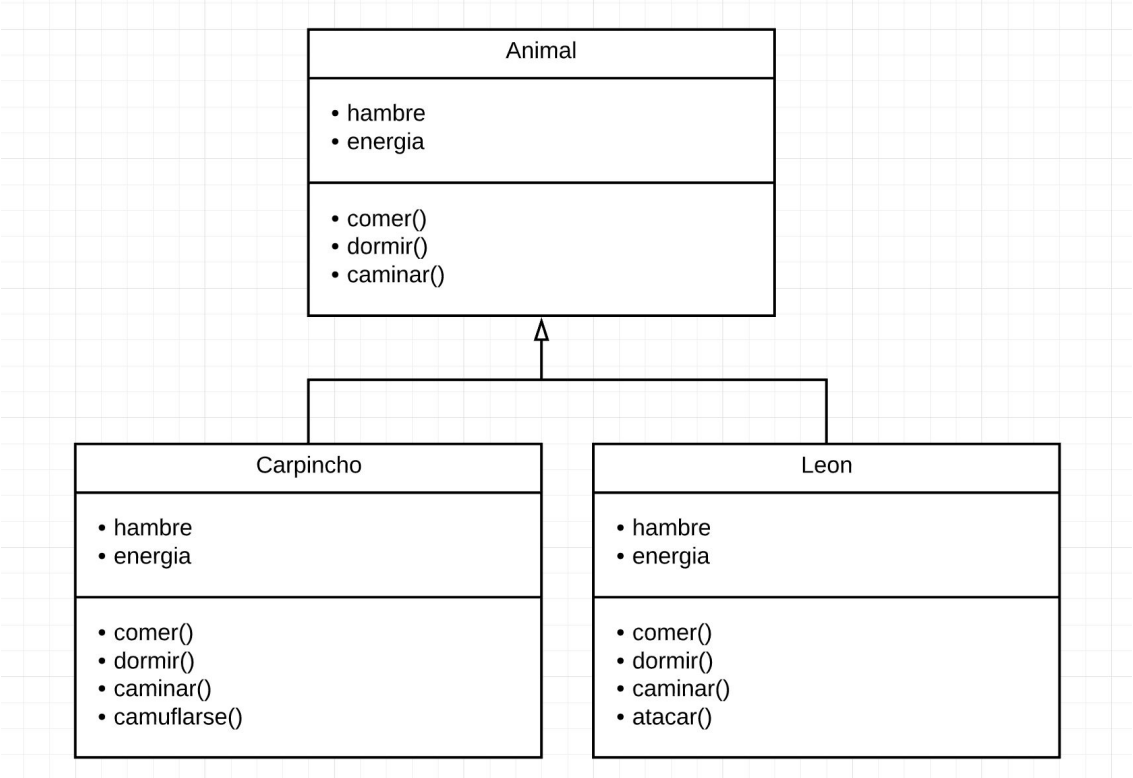
Herencia

- La herencia ocurre cuando se tiene una clase padre y una clase hija. La clase hija **hereda** los métodos y propiedades de la hija.
- La clase hija puede tener más métodos y propiedades que la de la padre.
- Los métodos heredados se pueden sobre escribir y cambiarlos.
- Solo usar herencia cuando **...es un...** Por ejemplo un carpincho **es un** animal



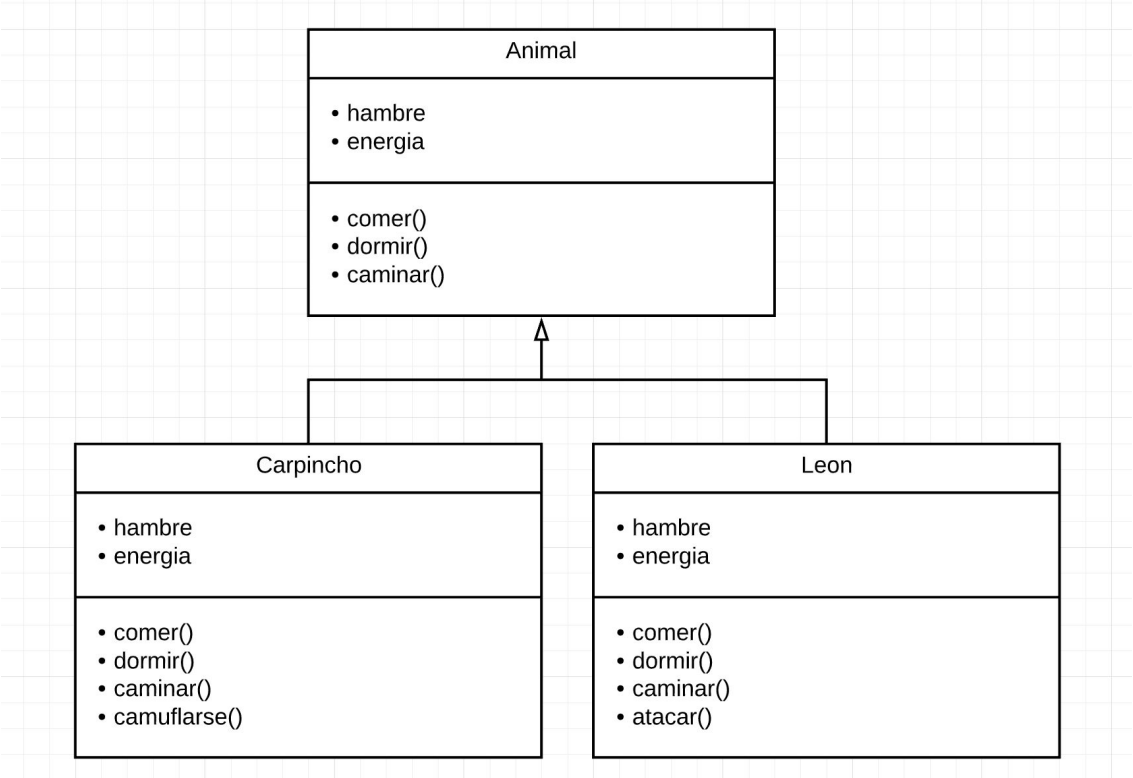
Herencia

- Carpincho y León **heredan** de Animal todas las propiedades (hambre y energía) y sus métodos.
- Carpincho además tiene método *camuflarse* y el león *atacar*.



Herencia

- León conoce cuánta energía suma al dormir el león y lo mismo para el Carpincho.
- El consumidor que usa estas clases, puede llamar al *dormir* de cualquier instancia y ella va saber cómo duerme.



```python
class Animal:
  def __init__(self):
    self.energia = 10

  def caminar(self):
    # Cada hijo tiene que implementar como camina.
    # Si no lanza una excepcion
    raise NotImplementedError()

  def descansar(self):
    self.energia += 10

  def obtenerEnergia(self):
    return self.energia

class Carpincho(Animal):
  # Se cansa 1 por caminar
  def caminar(self):
    self.energia -= 1

class Leon(Animal):
  # Se cansa 5 por caminar
  def caminar(self):
    self.energia -= 5
```

```python
carpincho = Carpincho()
leon = Leon()
carpincho.caminar()
print(carpincho.obtenerEnergia())
# 9
leon.caminar()
print(leon.obtenerEnergia())
# 5
animal = Animal()
animal.caminar()
# Traceback (most recent call last):
#   File "test.py", line 35, in <module>
#     animal.caminar()
#   File "test.py", line 8, in caminar
#     raise NotImplementedError()
# NotImplementedError
```

Polimorfismo

- En programación orientada a objetos, el **polimorfismo** se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.
- En este caso hacemos caminar a varios animales de distinto tipo.

```
class Zoo:
  def __init__(self):
    self.animales = []

  def agregarAnimal(self, animal):
    self.animales.append(animal)

  def caminarTodos(self):
    # Estamos haciendo caminar a todos sin que sepamos el tipo de cada animal!
    for animal in self.animales:
      animal.caminar()

carpincho = Carpincho()
leon = Leon()
zoo = Zoo()
zoo.agregarAnimal(carpincho)
zoo.agregarAnimal(leon)
zoo.caminarTodos()
```

Ejercicio

Tiempo ⏱️

Implementar:

- 3 Clases de **Útiles**: Lapicera, Lápiz y Marcador donde todos puedan escribir con cierto grosor, color y tengan una cantidad de tinta que disminuye al escribir.
- 1 clase Cartuchera donde se puedan guardar **Útiles** . La cartuchera tiene una capacidad y debería poder almacenar hasta una cierta cantidad de útiles.