TALLER PROGRAMACIÓN ORIENTADA A OBJETOS CON PYTHON

ACTIVIDADES POR DESARROLLAR:

- Comprender Programación Orientada a Objetos en **PYTHON**
- Aplicar Programación Orientada a Objetos en **PYTHON**

EVIDENCIA(S) A ENTREGAR:

EV1 Desarrollar la actividad a desarrollar propuesta en el taller

CONTROL DEL DOCUMENTO

	Nombre	Cargo	Dependencia	Fecha
Autor (es)	JOSE FERNANDO GALINDO SUAREZ	INSTRUCTOR	CGMLTI	13/04/2024

CONTROL DE CAMBIOS (diligenciar únicamente si realizan ajustes al taller)

	Nombre	Cargo	Dependencia	Fecha	Razón del Cambio
Autor (es)					

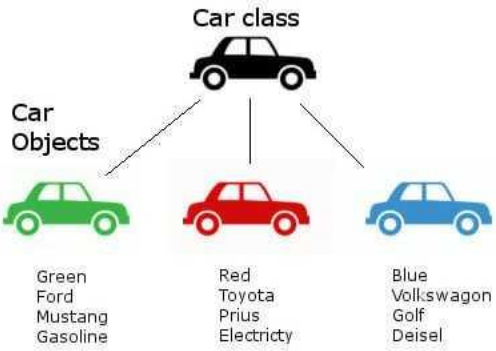
INTRODUCCIÓN

Después de realizar la lectura "programación orientada a objetos", podrá desarrollar los ejercicios dispuestos en este taller.

La programación orientada a objetos es un paradigma de la programación y es el más actualizado en la actualidad, aumentando así la modularidad y la reutilización de código en la programación, rompiendo el paradigma de la programación estructurada, aunque se puede combinar las dos al desarrollar aplicaciones.

Se acerca a la manera como se tratan los objetos en la realidad empezando con la fase de análisis, luego su implementación será la forma más adecuada para su desarrollo.

Clases y objetos.



- **Los objetos son representaciones (simples/complejas) (reales/imaginarias) de cosas: reloj, avión, coche.**
- **No todo puede ser considerado como un objeto, algunas cosas son simplemente características atributos de los objetos: color, velocidad, nombre**



Abstracción funcional

Hay cosas que sabemos que los coches hacen, pero no cómo lo hacen:

- avanzar

- parar
- girar a la derecha
- girar a la izquierda

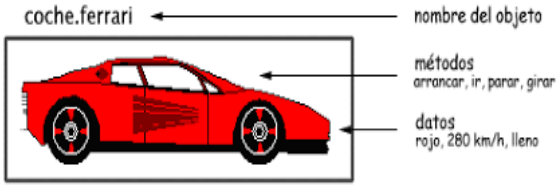
Abstracción de datos

Un coche tiene además ciertos atributos:

- color
- velocidad
- tamaño
- etc.

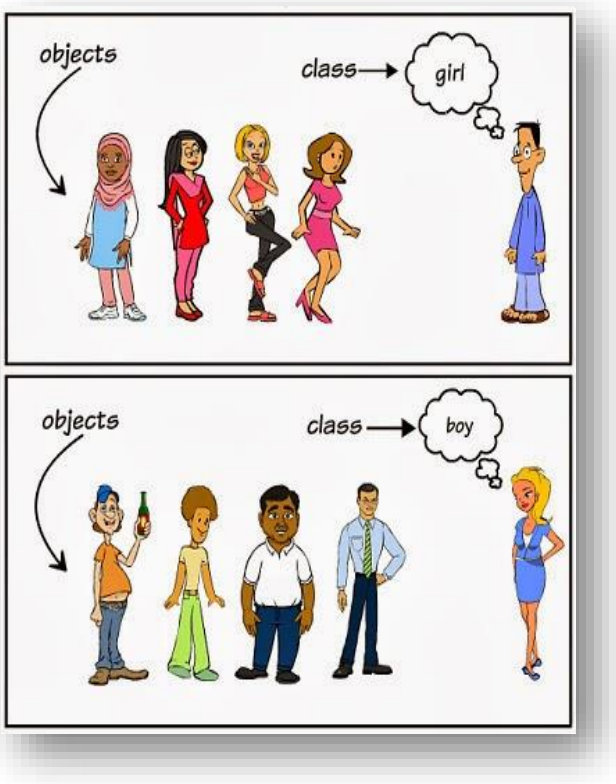
- **Es una forma de agrupar un conjunto de datos (estado) y de funcionalidad (comportamiento) en un mismo bloque de código que luego puede ser referenciado desde otras partes de un programa**

- **La clase a la que pertenece el objeto puede considerarse como un nuevo tipo de datos**



Los objetos permiten tener un control total sobre 'quién' o 'qué' puede acceder a sus variables y métodos, es decir, pueden tener componentes públicos, a los que podrán acceder otros objetos, o componentes privados, a los que únicamente puede acceder él.

Los componentes pueden ser tanto las variables como los métodos de ese objeto.



La programación orientada a objetos (Object Oriented Programming OOP) es un modelo de lenguaje de programación organizado por objetos constituidos por datos y funciones, entre los cuales se pueden crear relaciones como herencia, cohesión, abstracción, polimorfismo y encapsulamiento.

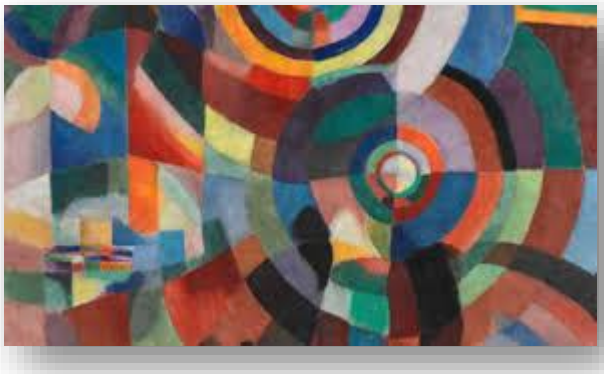
Esto permite que haya una gran flexibilidad y se puedan crear objetos que pueden heredarse y transmitirse sin necesidad de ser modificados continuamente.

JERARQUÍA DE COMPOSICIÓN

El objeto está compuesto por otros objetos con comportamientos distintos.

Esto sirve para representar uno varios objetos que están dentro de otro que los contiene.

DEFINICIÓN DE ABSTRACCIÓN.



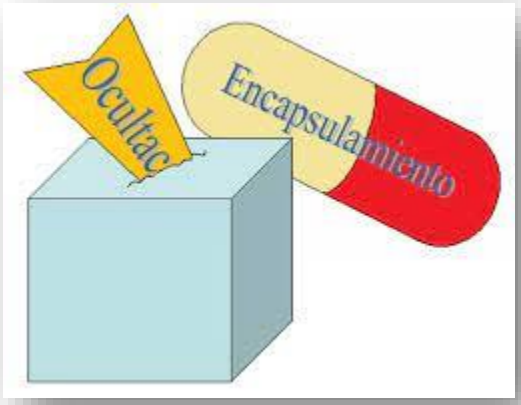
Nos da una visión simplificada de una realidad de la que sólo consideramos determinados aspectos esenciales:

¿qué entendemos por ...?
¿... color de un semáforo?
¿... estado de una cuenta bancaria?
¿... estado de una bombilla?
¿qué necesitamos conocer de un coche para utilizarlo?

La abstracción como técnica de programación

La programación es una tarea compleja.
Mediante la abstracción es posible elaborar software que permita solucionar problemas cada vez más grandes.

DEFINICIÓN DE ENCAPSULAMIENTO



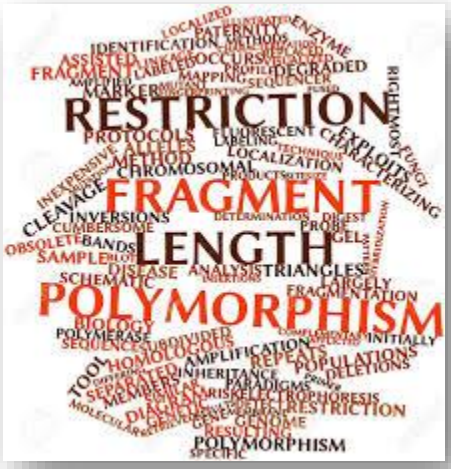
Proceso de ocultamiento de todos los detalles de una entidad que no contribuyen a sus características esenciales.

Abstracción nos centramos en la visión externa.
Encapsulamiento nos centramos en la visión interna.
El acceso a los datos y las operaciones se realiza
mediante una interfaz bien definida.

DEFINICIÓN DE POLIMORFISMO

El polimorfismo se refiere al hecho que un método adopte múltiples formas.

Esto se consigue por medio de la sobrecarga de métodos:
un mismo nombre de método para distintas funcionalidades.
a = Sumar (c,d); a = Sumar (c,d,5);
Sobrecarga de operadores:
un mismo operador con distintas funcionalidades.



En la sobrecarga de funciones se desarrollan distintas funciones con un mismo nombre, pero distinto código.
Las funciones que comparten un mismo nombre deben tener una relación en cuanto a su funcionalidad.
Aunque comparten el mismo nombre, deben tener distintos parámetros. Éstos pueden diferir en:

- El número
- El tipo
- El orden
- El tipo del valor de retorno de una función no es válido como distinción.

Define los datos y el comportamiento, llamado atributos y métodos.

```
'''
nombre #atributo público
_nombre #atributo protegido
__nombre #atributo privado
'''
class Alumno:
  def __init__(self, nombre, nota):
    self.__nombre = nombre
    self.__nota = nota
    print ("Estoy dentro del constructor")

  def mostrar_info(self, maximo=5):
    print ("nombre", self.__nombre, "nota:", self.__nota, "entre ", maximo)
    print ("Objeto actual ", self)
########################################
a1 = Alumno ("Juan", 3)

a1.mostrar_info()


a2 = Alumno ("Margarita", 3)
a2.mostrar_info(4)
print(a2)
print (a2)
```

Character	Icon for field	Icon for method	Visibility
-			private
#			protected
~			package private
+			public

La visibilidad de una propiedad, un método o una constante se puede definir anteponiendo a su declaración una de las palabras reservadas public, protected o private.

- public la variable/función se accede desde cualquier lugar y otras instancias de esa misma clase.
- private la variable/función solamente se accede desde la misma clase que la define.
- protected la variable/función se accede desde la clase que las define y las clases que se herede de ella.

Cuando un método tiene dos guiones bajos al principio y al final de sus nombres se denomina un método mágico. Los métodos mágicos no se pueden llamar directamente desde el código como lo hacemos con los demás métodos, son "mágicos" porque Python los llama automáticamente en determinadas situaciones.

```
__metodo_magico__()   #el doble guión bajo antes y después del nombre del método lo identifica como método mágico
```

El parámetro self

Cada clase da origen a diferentes objetos. El parámetro **self** que se recibe en los constructores o métodos de una clase hace referencia al objeto en particular que está accediendo al comportamiento definido en la clase en un momento determinado.

Todos los métodos en Python deben recibir como primer parámetro el parámetro **self** de esta manera se tiene una referencia al objeto actual que está haciendo uso de los miembros de la clase.

```
obtener_nombre(self):
    return self.nombre      #se devuelve el nombre del objeto actual
```

Ejemplo:

```
class Persona:
      def saludar(self, nombre):
    print("hola ", nombre)
    print(self)
##############################################
p = Persona()
p.saludar("Juan")
p1 = Persona()
p1.saludar("Lina")

print("estoy por fuera y soy :", p)
```

El constructor

Si en Python no se especifica un constructor, de igual manera cada clase tendrá un constructor por defecto que permite instanciar objetos de esa clase.

Así:

```
class Persona:
      pass

juan = Persona()
print (juan)
```

En el código anterior, se creó un objeto de la clase Persona, el objeto se llama **juan** y no hace nada, ya que la clase Persona no tiene código. Pero el objeto existe y tiene memoria asignada para su almacenamiento.

```
class Persona:
      pass

juan = Persona()
print ("memoria juan:", juan) #muestra la posición de memoria
print ("tipo juan:", type(juan)) #muestra de qué tipo es juan

pedro = Persona()
print ("memoria pedro:", pedro) #muestra la posición de memoria
print ("tipo pedro:", type(pedro)) #muestra de qué tipo es juan
```

__ init__ ()

Para definir el constructor de una clase en Python, se usa el método mágico **init**

```
class Persona:
      def __init__(self):
    print ("Me estoy creando ", self)
################################################

juan = Persona()   #creamos objetos de la clase Persona
pedro = Persona()
```

Complementando:

```
class Persona:
      def __init__(self, ced, nom, ed):
    print ("Me estoy creando ", self)
```

```
    self.cedula = ced
    self.nombre = nom
    self.edad = ed
################################################



obj1 = Persona(123, "Luis", 35)   #creamos objetos de la clase Persona
obj2 = Persona(345, "Lina", 20)
print (obj1.cedula, obj1.nombre, obj1.edad)
print(obj1)
print (obj2.cedula, obj2.nombre, obj2.edad)
print(obj2)
```

En Python no es necesario declarar explícitamente los atributos de una clase. Ellos van a estar siempre almacenados en el parámetro self

```
class Persona:
      def __init__(self, n, e, pepito):
    self.nombre = n
    self.edad = e
    self.ciudad = pepito

  def mostrar_informacion(self):  #mediante el self el método reconoce al
objeto que está invocando el método
    print(self.nombre, self.edad, self.ciudad)
################################################

juan = Persona("Juan Villa", 23, "Medellín")
pedro = Persona("Pedro Valencia", 40, "Pereira")

pedro.pasaporte = "AX4743"
pedro.novia = "Florencia"

juan.alergia = "mariscos"

print (juan.ciudad)
print (pedro.ciudad)

print (juan.alergia)

#juan.mostrar_informacion()
#pedro.mostrar_informacion()
```

En Python se pueden crear atributos "al vuelo", es decir, después de crear un objeto se le pueden asignar los atributos deseados y estos formarán parte de los miembros de este objeto, los cuales recordemos podemos acceder a través del parámetro **self**

```
class Persona:
      def mostrar_informacion(self):
    print(vars(self))      # devuelve un diccionario con los atributos y
valores del objeto actual

  def __str__(self):
     return "Hola soy el objeto con esta información: " + str(vars(self))
################################################

juan = Persona()
juan.altura = 180
juan.peso = 85

juan.mostrar_informacion()
print (juan)

obj = Persona()
obj.sexo = "masculino"
obj.salario = 1000000
obj.tipo_sangre = "o+"
obj.mostrar_informacion()

print (obj)
```

Otros métodos mágicos

Las clases tienen por defecto algunos métodos mágicos que podemos sobrescribir para alterar su funcionamiento por defecto.

__ str__ ()

El método mágico **str** en una clase tiene el comportamiento por defecto de retornar una cadena con la clase a la que pertenece el objeto y su dirección de memoria. Este método mágico es llamado cada vez que a la función **str()** le enviamos por parámetro un objeto.

```
class Lenguaje:
      def __init__(self, nombre):
    self.nombre = nombre

  def __str__(self):
```

```
    return self.nombre
##########
l1 = Lenguaje("Java")
print(l1)

l2 = Lenguaje("Python")
print(l2)

l3 = Lenguaje("C#")
print(l3)
```

De lo contrario, se imprime me la dirección de memoria

```
class Persona:
    def __init__(self):
      print (str(self))
################################################

juan = Persona()   #creamos objetos de la clase Persona
print(juan)
```

En toda clase se puede sobrescribir el método mágico **str** para obtener otro tipo de resultado al imprimir un objeto de dicha clase

```
class Persona:
      def __str__(self):
    return "Este es un objeto cool otra vez"


################################################

juan = Persona()   #creamos objetos de la clase Persona y nos ejecuta el
constructor init
print (juan)
```

Se imprime el nombre.

```
class Persona:
      def __init__(self, nombre, edad, direccion):
    self.nombre = nombre
    self.edad = edad
    self.direccion = direccion

  def __str__(self):
    return self.nombre
```

```
###############################################

objeto1 = Persona("Andres Julian Valencia", 23, "calle 3")
print (objeto1)
objeto2 = Persona("Ximena Diaz", 35, "calle 66")
print (objeto2)
```

Atributos de Instancia y de Clase

Los atributos de los ejemplos anteriores son atributos de instancia, esto quiere decir que cada instancia (objeto) tiene su propio conjunto de atributos con sus propios valores. Estos atributos son accedidos a través del parámetro **self** y el cambio de valor en un atributo de instancia no tiene efecto en los demás objetos. Incluso, gracias a los atributos dinámicos en Python, dos instancias de la misma clase pueden tener atributos diferentes con valores por supuesto diferentes.

```
class Persona:
      def __init__(self):
    pass
  def mostrar_informacion(self):
    print(vars(self))      # devuelve un diccionario con los atributos y
valores del objeto actual
###############################################

juan = Persona()
juan.altura = 180
juan.peso = 85

pedro = Persona()
pedro.altura = 170
pedro.telefono = "300232323"

print ("Información del objeto juan:")
juan.mostrar_informacion()
print ("Información del objeto pedro:")
pedro.mostrar_informacion()
print ("Cambiamos la altura de pedro")
pedro.altura = 200
print ("Información del objeto juan:")
juan.mostrar_informacion()
print ("Información del objeto pedro:")
pedro.mostrar_informacion()
```

Un atributo de clase por su parte, es un atributo cuyo valor es el mismo para todas las instancias de una clase pues pertenece a la clase y no a sus instancias, aunque estas lo puedan acceder. En el siguiente código vamos a usar un atributo de clase para llevar la cuenta del número total de personas creadas:

```
class Persona:
      personas_total = 0 #este es un atributo de clase
  def __init__(self):
    Persona.personas_total += 1  #cada vez que se crea un objeto se incrementa
el atributo de clase
################################################

juan = Persona()
print ("personas_total a través de juan: ", juan.personas_total)
pedro = Persona()
print ("personas_total a través de pedro: ", pedro.personas_total)
print ("personas_total a través de juan: ", juan.personas_total)
luis = Persona()print ("personas_total a través de luis: ",
luis.personas_total)print ("personas_total a través de juan: ",
juan.personas_total)

print ("personas_total a través de pedro: ", pedro.personas_total)
```

Métodos

Los métodos se definen igual que una función con la diferencia de siempre recibir el primer parámetro **self** que como vimos antes hace referencia al objeto actual.

Sintaxis:

```
def mi_metodo(self):
    cuerpo del método
def mi_metodo(self, parametro1, parametro2, parametro_n):
    cuerpo del método
```

Ejemplo:

```
class Persona:
      def __init__(self, n, e, c):
    self.nombre = n
    self.edad = e
    self.ciudad = c

  def cumplir_anos(self):
    self.edad += 1
```

```
  def get_edad(self):
    return self.edad

  def set_edad(self, x):
    self.edad = x

  def puede_votar(self):
    if self.edad >= 18:
      return True
    else:
      return False

  def cambiar_ciudad(self, nueva_ciudad):
    self.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {self.ciudad}.
Podrá votar??? {self.puede_votar()}")
################################################################################
##################
p = Persona("Raul Diaz", 17, "Manizales")
p.mostrar_info()
p.cumplir_anos()
p.cambiar_ciudad("Cali")
p.mostrar_info()
```

Métodos de Clase

De la misma manera que existen los atributos de clase, también existen los métodos de clase. Un método de clase es un método que tiene acceso a la clase (atributos de clase u otros métodos de clase).

Para declarar un método como método de clase debemos poner el decorador **@classmethod** antes de la declaración del método de clase.

Estos métodos en lugar de recibir como primer parámetro self, reciben como primer parámetro **cls** que representa a la clase y a través de este parámetro puede acceder a los miembros de la clase.

```
class Persona:
      ciudad = "Manizales"  #atributo de clase

  def __init__(self, nombre, edad):
```

```
    self.nombre = nombre
    self.edad = edad

  def cumplir_anos(self):
    self.edad += 1


  @classmethod    #este es un método de clase
  def cambiar_ciudad(cls, nueva_ciudad):
    cls.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {Persona.ciudad}
")
################################################################################

p1 = Persona("Hugo", 35)
p2 = Persona("Paco", 20)
p3 = Persona("Luis", 29)

p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()


p1.cambiar_ciudad("Cali")  #la invocación de un método de clase se realiza a
través de la clase

p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()

Persona.ciudad = "Bogotá"
p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()
```

Métodos Estáticos

Los métodos estáticos no acceden ni modifican el estado de los objetos (self) ni de las clases (cls). Son métodos que cumplen una función particular y que quedan asociados a una determinada clase.

Para declarar un método como método estático debemos poner el decorador **@staticmethod** antes de la declaración del método estático.

Estos métodos no reciben ni el parámetro self ni el parámetro cls ya que como lo dijimos antes no conocen ni modifican el estado de la clase ni de sus objetos.

```
import random
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  @staticmethod    #este es un método estático
  def mensaje_hoy():
    mensajes =["La vida es bella", "Mejor tarde que nunca", "Dios te
bendiga", "Vive para servir"]
    print (random.choice(mensajes))

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años")
#############################################################################
########################
Persona.mensaje_hoy()
p1 = Persona ("Pablo", 38)
p1.mostrar_info()
Persona.mensaje_hoy()
```

APROPIACIÓN

1. Caso No 1

- Cree una clase llamada **Estudiante** con los siguientes atributos de instancia: **nombre**, **nota1** y **nota2**.
- Defina un constructor que reciba como parámetros el nombre, la nota1 y la nota2 del estudiante.
- Defina los siguientes métodos de instancia: **obtener_nota_promedio()** este método devuelve la nota promedio del estudiante y **mostrar_informacion()** este método muestra en pantalla todos los datos del estudiante (nombre, nota1, nota2 y nota promedio).
- Pruebe el funcionamiento de la clase.

2. Caso No 2

Encapsule la clase **Estudiante** creada en el punto anterior, de tal manera que todos sus atributos de instancia queden **privados**. Agregue los métodos

necesarios dentro de la clase para que continúe su funcionamiento y adicionalmente solo acepte notas entre 0 y 5.

3. Caso No 3

   A la clase **Estudiante** del punto anterior realice las adaptaciones necesarias para que al imprimir un objeto completo de esta clase se presente su nombre y nota promedio.

4. Caso No 4

   Se requiere que cualquier estudiante creado pertenezca a la misma institución de tal manera que si un estudiante cambia de institución, todos deben hacerlo. Lleve además un control del número de estudiantes de la institución. Realice los ajustes a la clase **Estudiante** del punto anterior para lograr estos objetivos.

5. Caso No 5

   Realice el método **ver_escala()** que al ser invocado en la clase **Estudiante** presente una tabla con la escala de calificación de todos los estudiantes de la institución. Esta escala es la siguiente:

Nota	Escala
0 a 2.9	Baja
3 a 3.9	Media
4 a 4.5	Alta
4.6 a 5	Sobresaliente

   Permita que al invocar al método desde la clase **Estudiante.ver_escala()** se muestre la tabla anterior. Utilice el módulo tabulate para que la presentación sea la indicada.

RELACIONES (Fuente)

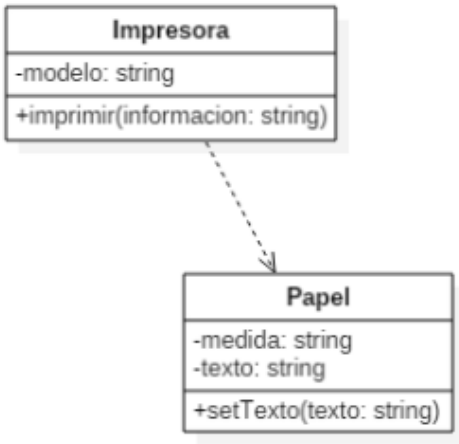
Asociación

- La asociación es una relación donde los objetos tienen su propio ciclo de vida y no hay propietario.
- La frase para comprobar una relación de este tipo es A es una parte de B.

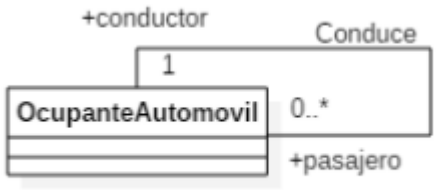
Tomemos un ejemplo de Instructor y Aprendiz.
Varios Aprendices pueden asociarse con un solo Instructor y un solo Aprendiz puede asociarse con varios Instructores. Ambos se pueden crear y eliminar de forma independiente.

Composición

Dependencia



Asociación reflexiva

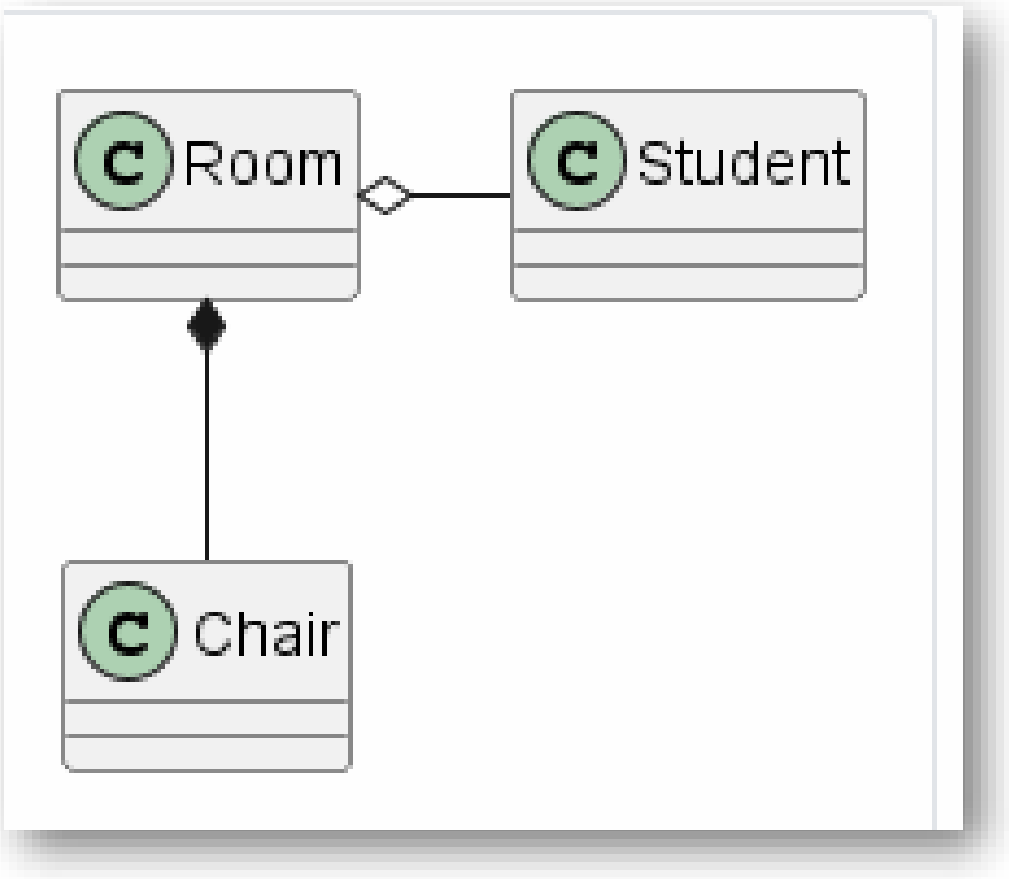


● La agregación es una forma especializada de asociación donde todos los objetos tienen su propia existencia, pero hay propiedad y los objetos secundarios no pueden pertenecer a otro objeto principal.

Un ejemplo: Área e Instructor. Un solo Instructor no puede pertenecer a múltiples Áreas, pero si eliminamos el Área, el objeto Instructor no se destruirá. Podemos pensarlo como una relación "tiene una".

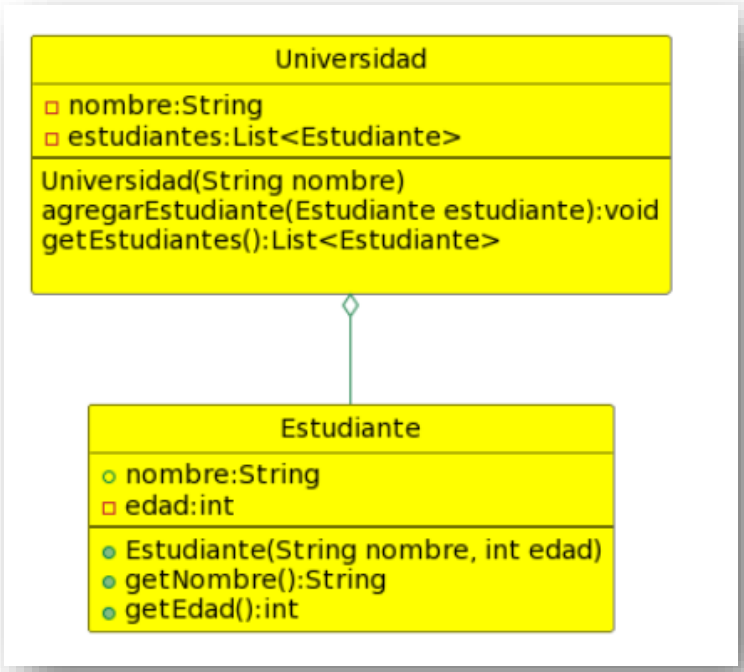
● La composición es una forma especializada de Agregación donde el objeto secundario no tiene su ciclo de vida propio y si se elimina el objeto principal, también se eliminarán todos los objetos secundarios.

La relación entre Preguntas y Opciones. Las preguntas individuales pueden tener múltiples opciones y la opción no puede pertenecer a otra pregunta.



AGREGACIÓN

La agregación: es un tipo de relación entre clases en la programación orientada a objetos, donde una clase contiene a otra, pero ambas pueden existir de manera independiente. Vamos a ver un ejemplo sencillo en Java para entender mejor el concepto de agregación.



Supongamos que queremos modelar una relación entre una clase Universidad y una clase Estudiante. La Universidad tiene una lista de estudiantes, pero los estudiantes pueden existir sin estar asociados a una universidad específica.

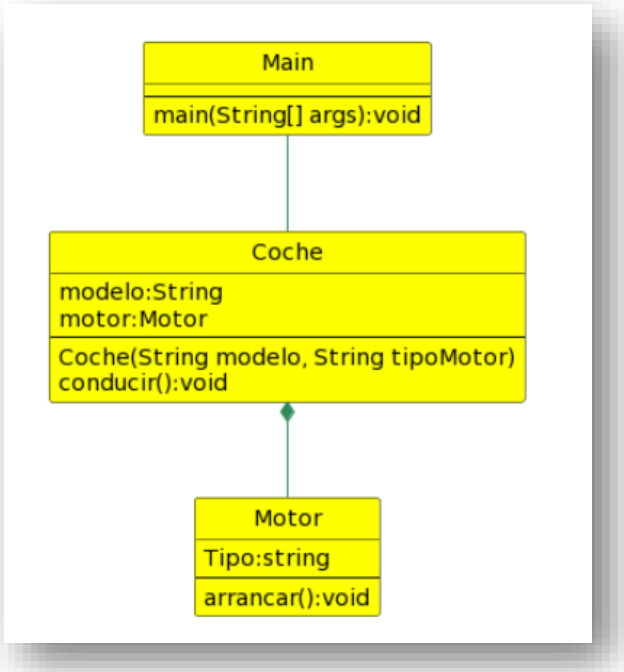
```
class Estudiante:
  def __init__(self,nom,edad):
    self.__nombre=nom
    self.__edad=edad

  def getNombre(self):
        return self.__nombre
  def getEdad(self):
        return self.__edad

class Universidad:

  def __init__(self,nombre):
    self.__nombre=nombre
    self.__estudiantes=[]

  def agregarEstudiante(self,estudiante:Estudiante):
    self.__estudiantes.append(estudiante)
```

```
  def getEstudiantes(self):
    for x in self.__estudiantes:
          print(f"Nombre: {x.getNombre()},Edad: {x.getEdad()}")


estudiante1=Estudiante("Juan",20)
estudiante2=Estudiante("Maria",22)
universidad=Universidad("Universidad xyz")
universidad.agregarEstudiante(estudiante1)
universidad.agregarEstudiante(estudiante2)

universidad.getEstudiantes()
```

En este ejemplo, la clase Universidad contiene una lista de estudiantes (estudiantes). La relación entre Universidad y Estudiante es de agregación, ya que los estudiantes pueden existir de forma independiente y no se destruyen si la universidad deja de existir. La universidad simplemente mantiene una lista de estudiantes asociados.

COMPOSICIÓN

La composición en programación orientada a objetos se refiere a la relación entre dos clases donde una clase contiene una instancia de otra clase. Aquí tienes un ejemplo sencillo en Java que ilustra la composición:



Supongamos que queremos modelar una relación entre una clase Motor y una clase Coche. En este caso, un coche tiene un motor, y utilizaremos la composición para representar esta relación.

```
// Definición de la clase Motor
class Motor:
  def __init_(self,tipo):
    self.tipo=tipo
  def arrancar(self):
    print("Motor arrancado")

class Coche:
  def __init__(self,modelo,tipoMotor):
    self.__modelo=modelo
    self.__tipoMotor=tipoMotor
  def getModelo(self):
```

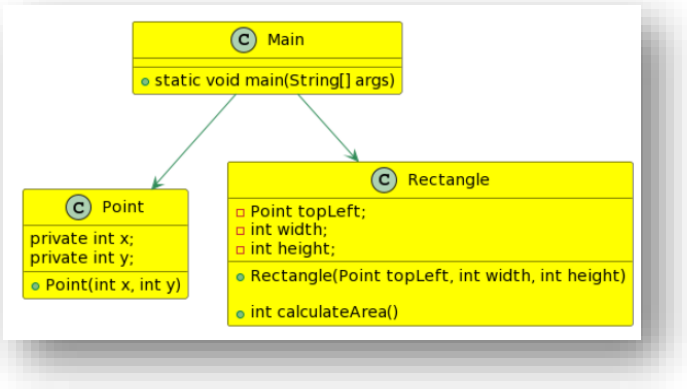
```
        return self.__modelo
  def conducir(self):
    print("Conduciendo el coche modelo: "+self.getModelo())

miCoche=Coche("Sedan","Gasolina")
miCoche.conducir()
```

En este ejemplo, la clase Coche tiene una instancia de la clase Motor. Al crear un objeto Coche, también se crea un objeto Motor dentro de él. La composición permite que el coche utilice las funcionalidades del motor, como el método arrancar().

DEPENDENCIA

En Java, la dependencia de clases se refiere a la relación en la cual una clase utiliza o depende de otra clase para llevar a cabo ciertas operaciones o funcionalidades. Estas dependencias se manifiestan a través de la utilización de instancias de otras clases, llamadas a métodos de otras clases o referencias a constantes y variables de otras clases.

Cuando una clase A depende de otra clase B, significa que la clase A utiliza o requiere la funcionalidad proporcionada por la clase B para realizar alguna tarea específica. La dependencia entre clases es un concepto fundamental en la programación orientada a objetos y es esencial para construir aplicaciones modularizadas y mantenibles.

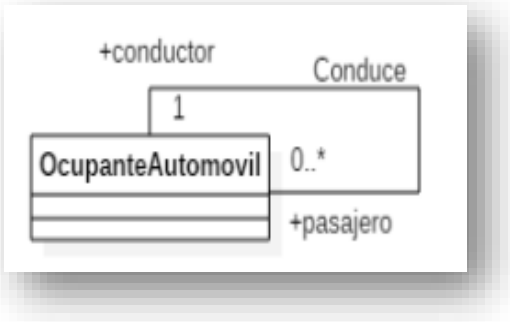



```
// Clase Rectangle que depende de la clase Point
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
```

```
        this.x = x;
        this.y = y;
    }

    // Métodos getter y setter para x e y
}

class Rectangle {
    private Point topLeft;
    private int width;
    private int height;

    public Rectangle(Point topLeft, int width, int height) {
        this.topLeft = topLeft;
        this.width = width;
        this.height = height;
    }

    // Método para calcular el área del rectángulo
    public int calculateArea() {
        return width * height;
    }
}

public class Main {
    public static void main(String[] args) {
        // Crear una instancia de Point
        Point point = new Point(10, 20);

        // Crear una instancia de Rectangle que depende de Point
        Rectangle rectangle = new Rectangle(point, 30, 40);
        // Calcular y mostrar el área del rectángulo
        int area = rectangle.calculateArea();
        System.out.println("Área del rectángulo: " + area);
    }
}
```

En este ejemplo, la clase Rectangle depende de la clase Point para representar las coordenadas del vértice superior izquierdo del rectángulo. La relación de dependencia se establece al crear una instancia de Point y pasarlo como argumento al constructor de Rectangle. La clase Rectangle utiliza la funcionalidad proporcionada por la clase Point para realizar sus operaciones.

ASOCIACIÓN



La asociación reflexiva en Java se refiere a una relación entre una clase y sí misma. En otras palabras, una instancia de la clase puede estar relacionada con otra instancia de la misma clase. Este tipo de relación se utiliza cuando necesitas modelar una relación entre objetos del mismo tipo. Puedes implementar asociaciones reflexivas a través de atributos de instancia que son instancias de la misma clase o a través de métodos que devuelven instancias de la misma clase.

```
public class Persona {
    private String nombre;
    private int edad;
    private Persona mejorAmigo;  // Asociación reflexiva: Una persona puede
tener otra persona como mejor amigo

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Método para establecer el mejor amigo
    public void establecerMejorAmigo(Persona amigo) {
        this.mejorAmigo = amigo;
    }

    // Método para obtener el mejor amigo
    public Persona obtenerMejorAmigo() {
        return mejorAmigo;
    }

    public static void main(String[] args) {
        Persona persona1 = new Persona("Alice", 25);
        Persona persona2 = new Persona("Bob", 30);

        // Establecer una asociación reflexiva
```

```
        persona1.establecerMejorAmigo(persona2);

        // Obtener el mejor amigo
        Persona amigoDePersona1 = persona1.obtenerMejorAmigo();
        System.out.println(persona1.getNombre() + "'s mejor amigo es " +
amigoDePersona1.getNombre());
    }

    // Métodos getter para nombre y edad
    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }
}
```

En este ejemplo, la clase Persona tiene un atributo llamado mejorAmigo que es de tipo Persona, lo que indica que una persona puede tener otra persona como su mejor amigo. Los métodos establecerMejorAmigo y obtenerMejorAmigo se utilizan para establecer y obtener la relación de mejor amigo entre instancias de la clase Persona. La asociación reflexiva se crea cuando se establece que una instancia de Persona puede tener otra instancia de Persona como su mejor amigo.

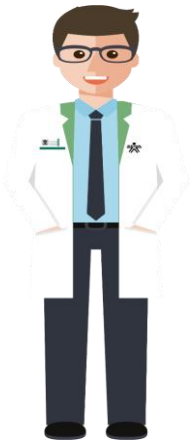
DELEGACIÓN

En el contexto de Java, la delegación se refiere a un patrón de diseño en el cual un objeto transfiere la responsabilidad de realizar ciertas tareas a otro objeto. Esto se logra mediante la creación de una asociación entre los objetos, donde el objeto que delega no implementa directamente la funcionalidad requerida, sino que pasa la solicitud al objeto delegado.

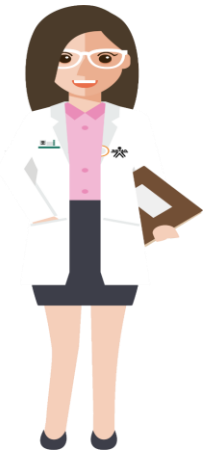
La delegación permite la reutilización de código y promueve la modularidad y la separación de preocupaciones en el diseño de software. En lugar de que un objeto realice todas las operaciones por sí mismo, puede delegar ciertas tareas a otros objetos especializados en esas operaciones.

En términos de implementación en Java, la delegación se logra mediante la creación de una instancia del objeto delegado dentro del objeto que delega y luego llamando a los métodos del objeto delegado cuando sea necesario realizar una operación específica. Esto puede lograrse mediante la composición de objetos, donde el objeto que delega contiene una referencia al objeto delegado y lo utiliza para completar ciertas tareas.

```java
package facturas;
import java.util.*;
public class Facturas {
    static List<String> testList = new ArrayList<String>();
      void addDetalle(FacturaDetalle detalle){
          testList.add(detalle.nombre);
    }
     static public void MostrarDetalle(){
         System.out.println("->"+testList);
     }
    static public class FacturaDetalle{
        static String nombre;
        FacturaDetalle(String Que){
            this.nombre=Que;
        }
    }
    public static void main(String[] args) {
        // TODO code application logic here
        Facturas factura=new Facturas();
        factura.addDetalle(new FacturaDetalle("Juan"));
        factura.addDetalle(new FacturaDetalle("Maria"));
        factura.addDetalle(new FacturaDetalle("Carlos"));
        MostrarDetalle();
        System.out.println("Oprima una tecla para continuar....");
    }
}
```



```
run:
->[Juan, Maria, Carlos]
Oprima una tecla para continuar....
BUILD SUCCESSFUL (total time: 0 seconds)
```



Diferencia entre Agregación y Composición

- Las relaciones en una composición son requeridas, en la agregación son opcionales.
- En la composición una clase partícula no puede ser compartida por otras clases compuestas, en la agregación esto es posible.
- La relación de vida de la clase partícula y la clase contenedora, es muy fuerte, de hecho es la relación más fuerte; tanto así que si un objeto de la clase contenedora es destruido la clase partícula también lo será. Esto en la agregación no ocurre.

Visibilidad

La visibilidad de una propiedad, un método o una constante se puede definir anteponiendo a su declaración una de las palabras reservadas *public*, *protected* o *private*. A los miembros de clase declarados como 'public' se puede acceder desde donde sea; a los miembros declarados como 'protected', solo desde la misma clase, mediante clases heredadas o desde la clase padre. A los miembros declarados como 'private' únicamente se puede acceder desde la clase que los definió.

Visibilidad de propiedades

Las propiedades de clases deben ser definidas como 'public', 'private' o 'protected'. Si se declaran usando *var*, serán definidas como 'public'.

HERENCIA DE OBJETOS

Esto es útil para la definición y abstracción de la funcionalidad y permite la implementación de funcionalidad adicional en objetos similares sin la necesidad de Re implementar toda la funcionalidad compartida.

En el contexto de programación en Java, la herencia es un concepto fundamental que permite la creación de nuevas clases basadas en clases ya existentes. La herencia es un mecanismo que permite a una clase heredar propiedades y comportamientos de otra clase, lo que facilita la reutilización del código y la creación de una jerarquía de clases.

En Java, la herencia se implementa mediante la palabra clave extends. Una clase que hereda de otra se llama subclase o clase hija, y la clase de la cual se hereda se llama superclase o clase padre.

```java
// Superclase o clase padre
class Animal {
    void comer() {
        System.out.println("El animal está comiendo");
    }
}

// Subclase o clase hija que hereda de Animal
class Perro extends Animal {
    void ladrar() {
        System.out.println("El perro está ladrando");
    }
}

public class Main {
    public static void main(String[] args) {
        // Crear una instancia de la subclase
        Perro miPerro = new Perro();
```

```
        // Llamar a métodos de la superclase
        miPerro.comer();

        // Llamar a métodos de la subclase
        miPerro.ladrar();
    }
}
```

En este ejemplo, la clase **`Perro`** hereda de la clase **`Animal`**. Esto significa que la clase **`Perro`** tiene automáticamente el método **`comer`** de la clase **`Animal`**, además de tener su propio método **`ladrar`**. La herencia permite organizar el código de una manera que refleje la relación "es un/a" entre las clases, lo que facilita la comprensión y el mantenimiento del código.

CLASES TRAIT.

Los TRAIT son clases que agrupan funcionalidades diseñadas y reutilizadas en otras clases para evitar la herencia simple. Estas no se instancian, sólo permiten utilizar sus funciones específicas en otras clases.

En Java, las clases trait no son una característica nativa del lenguaje. Sin embargo, a partir de Java 8, se introdujeron las interfaces funcionales que proporcionan una funcionalidad similar a los traits en otros lenguajes de programación.

En programación orientada a objetos, un trait es una unidad de comportamiento que se utiliza para componer clases y agregar funcionalidades a ellas. En Java, las interfaces funcionales se introdujeron para admitir programación funcional y permitir la creación de funciones lambda. Una interfaz funcional es aquella que tiene un solo método abstracto y puede contener métodos default y estáticos.

A continuación, se muestra un ejemplo simple de una interfaz funcional en Java:

```
@FunctionalInterface
```

```java
public interface MiTrait {
    void miMetodoAbstracto();

    default void miMetodoDefault() {
        System.out.println("Este es un método default en la interfaz
funcional.");
    }

    static void miMetodoEstatico() {
        System.out.println("Este es un método estático en la interfaz
funcional.");
    }
}
```

En este ejemplo, MiTrait es una interfaz funcional que contiene un método abstracto (miMetodoAbstracto) y dos métodos con implementación predeterminada (miMetodoDefault y miMetodoEstatico). La anotación @FunctionalInterface es opcional pero se utiliza para indicar que la interfaz es funcional.

Las interfaces funcionales en Java 8 y versiones posteriores permiten una forma de composición de comportamiento similar a los traits en otros lenguajes de programación. Además, con la introducción de las expresiones lambda, puedes utilizar estas interfaces para lograr un estilo de programación más funcional en Java.

OPERADOR DE RESOLUCIÓN DE ÁMBITO (.)

"El Operador de Resolución de Ámbito (también denominado Paamayim Nekudotayim) o en términos simples, es un token que permite acceder a elementos estáticos, constantes, y sobrescribir propiedades o métodos de una clase."

En Java, el "operador de resolución de ámbito" se denota por el símbolo de punto (.) y se utiliza para acceder a miembros de una clase, como variables de instancia,

métodos o variables estáticas. Este operador se conoce también como el "operador de acceso de miembro". La sintaxis general es:

```
objeto.metodo();  // Acceso a un método de instancia
objeto.variableDeInstancia;  // Acceso a una variable de instancia
Clase.metodoEstatico();  // Acceso a un método estático
Clase.variableEstatica;  // Acceso a una variable estática
```

Aquí, objeto puede ser una instancia de una clase y Clase es el nombre de la clase. El operador de resolución de ámbito permite a los programadores acceder a los miembros de una clase y llamar a métodos o acceder a variables tanto de instancias como estáticas.

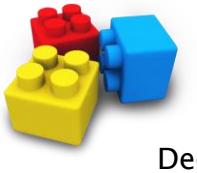
Ejemplo:

```
public class Ejemplo {
    // Variable de instancia
    private int variableInstancia;

    // Método de instancia
    public void metodoInstancia() {
        System.out.println("Método de instancia");
    }

    // Variable estática
    public static int variableEstatica;

    // Método estático
    public static void metodoEstatico() {
        System.out.println("Método estático");
    }

    public static void main(String[] args) {
        // Uso del operador de resolución de ámbito
        Ejemplo objeto = new Ejemplo();
        objeto.variableInstancia = 42;  // Acceso a variable de instancia
        objeto.metodoInstancia();  // Llamada a método de instancia
```

```
        Ejemplo.variableEstatica = 10;  // Acceso a variable estática
        Ejemplo.metodoEstatico();  // Llamada a método estático
    }
}
```

En este ejemplo, se muestra cómo usar el operador de resolución de ámbito para acceder a variables y métodos tanto de instancia como estáticos. Es fundamental entender este operador para trabajar efectivamente con objetos y clases en Java.

DESDE EL INTERIOR DE LA DEFINICIÓN DE LA CLASE



Las tres palabras claves especiales *self*, *parent* y *static* son utilizadas para acceder a propiedades y métodos desde el interior de la definición de la clase.

Declarar static a un método o atributo permite hacerlos accesibles sin la necesidad de instanciar la clase contenedora. Cuando se declara static no se puede acceder mediante una clase instanciada. La seudo variable this. no está disponible dentro de los métodos y atributos declarados static, se debe utilizar self.

INVOCANDO A UN MÉTODO PARENT.

En Java, el término "parent" no es un concepto específico del lenguaje de programación en sí mismo. Sin embargo, puede hacer referencia a algunas situaciones o contextos en el desarrollo de software en Java. Aquí hay algunas interpretaciones posibles:

Clase Padre (Superclass): En la programación orientada a objetos, especialmente en Java, se utiliza el término "clase padre" para referirse a una clase de la cual otra clase hereda atributos y métodos. La clase que hereda se llama "clase hija" o "subclase". En este contexto, a veces se usa la palabra "parent" para describir la clase de la cual otra está heredando.

```
class Animal {
    // ...
```

```
}

class Mamifero extends Animal {
    // ...
}
```

En este ejemplo, Animal sería la "clase padre" de Mamifero.

Jerarquía de Directorios (Padre/Directorio Superior): En el manejo de archivos y directorios, el término "parent" se puede usar para referirse al directorio superior o padre de un directorio particular. Java proporciona métodos para trabajar con rutas de archivos y directorios, y el método getParent() puede utilizarse para obtener el directorio padre de una ruta dada.

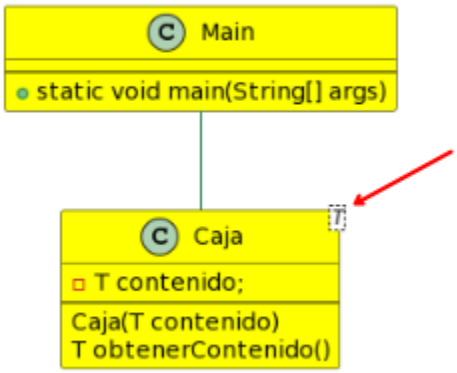
```
import java.nio.file.Path;
import java.nio.file.Paths;

public class DirectorioParent {
    public static void main(String[] args) {
        Path ruta = Paths.get("/home/usuario/carpeta/archivo.txt");
        Path directorioPadre = ruta.getParent();
        System.out.println("Directorio Padre: " + directorioPadre);
    }
}
```

En este ejemplo, getParent() se utiliza para obtener el directorio padre de la ruta "/home/usuario/carpeta/archivo.txt".

CLASE GENÉRICA

Es un recurso que podemos encontrar en la mayoría de los lenguajes de programación. Esta clase genérica es una clase que no tiene ninguna propiedad ni método como solemos ver en otras clases predefinidas. Puede parecer poco práctico, pero en realidad es de gran ayuda cuando se quiere crear un objeto e ir añadiendo las propiedades que queramos para, por ejemplo, añadirlo a un archivo JSON.

En Java, una clase genérica es una plantilla que define un conjunto de tipos que pueden ser utilizados como parámetros al crear instancias de esa clase. Las clases genéricas permiten escribir código que sea reutilizable y que funcione con diferentes tipos de datos sin tener que repetir el código para cada tipo.



La principal ventaja de las clases genéricas es que proporcionan un tipo de seguridad de tipo durante la compilación, lo que significa que el compilador puede detectar y evitar errores de tipo en tiempo de compilación. Esto ayuda a escribir código más robusto y menos propenso a errores.

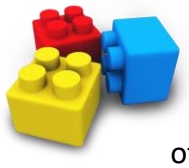
Para definir una clase genérica en Java, se utiliza la sintaxis <T>, donde T es un tipo de parámetro que puede ser cualquier tipo de dato (clase, interfaz, tipo primitivo, etc.). Este tipo de parámetro se puede utilizar dentro de la clase para definir variables, métodos y otros elementos. Cuando se instancia la clase genérica, se proporciona el tipo concreto que se utilizará en lugar del parámetro T.

```java
// Definición de una clase genérica
class Caja<T> {
    private T contenido;

    // Constructor
    public Caja(T contenido) {
        this.contenido = contenido;
    }
```

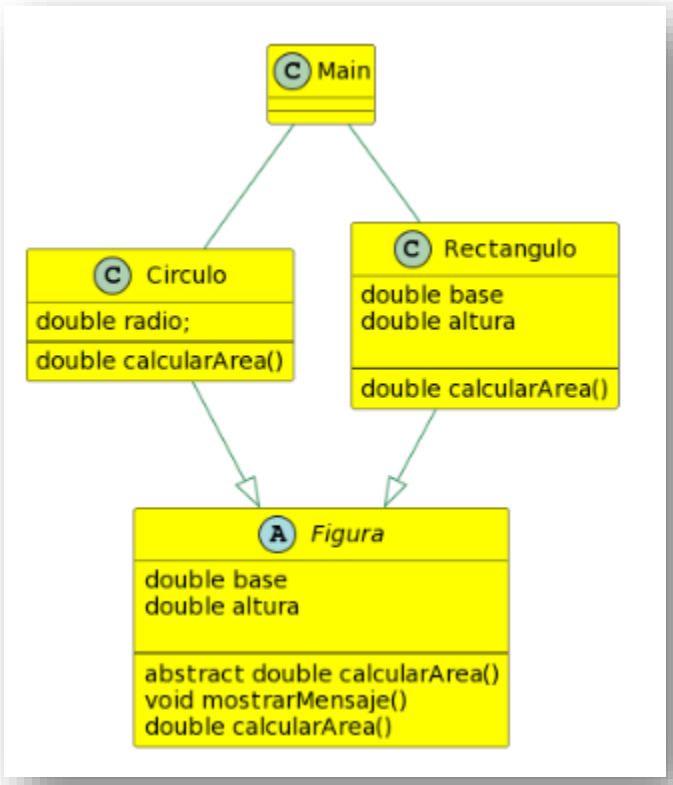
```
    // Método para obtener el contenido
    public T obtenerContenido() {
        return contenido;
    }
}

// Clase principal
public class Main {
    public static void main(String[] args) {
        // Crear una instancia de la clase genérica con un tipo concreto
(String en este caso)
        Caja<String> cajaString = new Caja<>("Hola Mundo");
        // Obtener y mostrar el contenido
        System.out.println("Contenido de la caja: " +
cajaString.obtenerContenido());

        // Crear una instancia de la clase genérica con otro tipo concreto
(Integer en este caso)
        Caja<Integer> cajaInteger = new Caja<>(42);
        // Obtener y mostrar el contenido
        System.out.println("Contenido de la caja: " +
cajaInteger.obtenerContenido());
    }
}
```

En este ejemplo, Caja es una clase genérica que utiliza el tipo de parámetro T. La clase tiene un constructor que toma un argumento del tipo T y un método obtenerContenido() que devuelve el contenido almacenado en la caja. Al crear instancias de la clase Caja, se proporcionan tipos concretos para el parámetro T. En el primer caso, se crea una caja de tipo String, y en el segundo caso, se crea una caja de tipo Integer.

CLASES ABSTRACTAS



Las clases abstractas son aquellas que por sí mismas no se pueden identificar con algo 'concreto' (no existen como tal en el mundo real), pero sí poseen determinadas características que son comunes en otras clases que pueden ser creadas a partir de ellas. .



Una clase abstracta en Java es una clase que no se puede instanciar directamente, es decir, no se pueden crear objetos de ella. La finalidad principal de una clase abstracta es proporcionar una estructura común para clases que comparten ciertas características, pero que pueden tener comportamientos específicos diferentes.

En Java, se define una clase abstracta utilizando la palabra clave **abstract** antes de la palabra **class**. Además, una clase abstracta puede contener métodos abstractos y métodos concretos. Un método abstracto es un método que no tiene una implementación definida en la clase abstracta, sino que debe ser implementado por las clases hijas (subclases) que heredan de ella. Los métodos concretos son aquellos que tienen una implementación definida en la clase abstracta y que pueden ser heredados y utilizados directamente por las subclases.

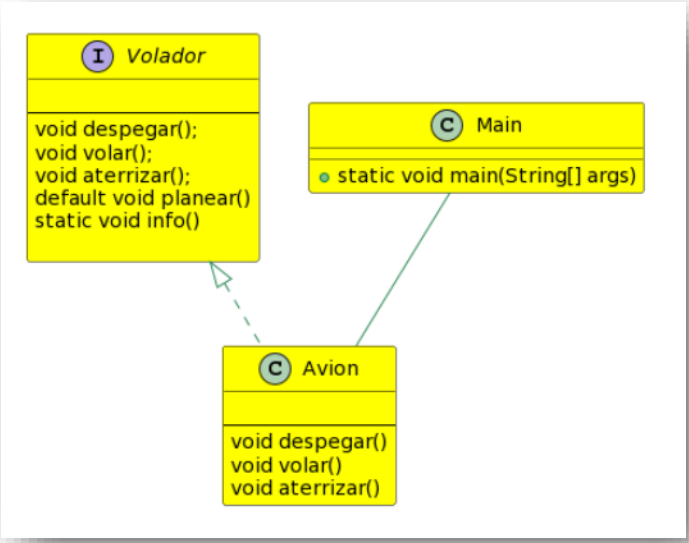
Para declarar una **clase** o método como abstractos, se utiliza la palabra reservada abstract. Una **clase abstracta** no se puede instanciar(es decir no se pueden volver en objetos), pero si se puede heredar y las **clases** hijas serán las encargadas de implementar la funcionalidad a los métodos abstractos.

```
abstract class Figura {
    // Método abstracto para calcular el área
    abstract double calcularArea();

    // Método concreto para imprimir un mensaje
    void mostrarMensaje() {
        System.out.println("Esta es una figura.");
    }
}

class Rectangulo extends Figura {
    // Implementación del método abstracto
    double calcularArea() {
        // Implementación específica para calcular el área de un rectángulo
        return base * altura;
    }

    // Variables específicas de un rectángulo
    double base;
    double altura;
}

class Circulo extends Figura {
    // Implementación del método abstracto
    double calcularArea() {
        // Implementación específica para calcular el área de un círculo
```

```
        return Math.PI * radio * radio;
    }

    // Variable específica de un círculo
    double radio;
}

public class Main {
    public static void main(String[] args) {
        // No se puede crear un objeto de tipo Figura directamente
        // Figura f = new Figura(); // Esto daría un error

        // Se pueden crear objetos de las subclases
        Rectangulo rectangulo = new Rectangulo();
        rectangulo.base = 5;
        rectangulo.altura = 3;
        System.out.println("Área del rectángulo: " +
rectangulo.calcularArea());
        rectangulo.mostrarMensaje();

        Circulo circulo = new Circulo();
        circulo.radio = 2.5;
        System.out.println("Área del círculo: " + circulo.calcularArea());
        circulo.mostrarMensaje();
    }
}
```

En este ejemplo, Figura es una clase abstracta que define un método abstracto calcularArea() y un método concreto mostrarMensaje(). Las clases Rectangulo y Circulo son subclases de Figura que implementan el método abstracto calcularArea() con su propia lógica específica.

INTERFACES



Una interfaz (interface) es sintácticamente similar a una clase abstracta, en la que puede especificar uno o más métodos que no tienen cuerpo ({}). Esos métodos deben ser implementados por una clase para que se definan sus acciones.

Por lo tanto, una interfaz especifica qué se debe hacer, pero no cómo hacerlo. Una vez que se define una interfaz, cualquier cantidad de clases pueden implementarla. Además, una clase puede implementar cualquier cantidad de interfaces.

Las interfaces en Java se definen utilizando la palabra clave interface. Una clase implementa una interfaz proporcionando una implementación para todos los métodos definidos en la interfaz. Una clase puede implementar múltiples interfaces, lo que permite la implementación de múltiples conjuntos de comportamientos.

Además de los métodos abstractos, una interfaz puede contener métodos predeterminados (default methods) y métodos estáticos, que tienen implementaciones predeterminadas. Los métodos predeterminados permiten agregar nuevos métodos a las interfaces sin romper la compatibilidad con las clases que ya las implementan. Los métodos estáticos proporcionan funcionalidad relacionada con la interfaz, pero no dependen de ninguna instancia específica de la clase.

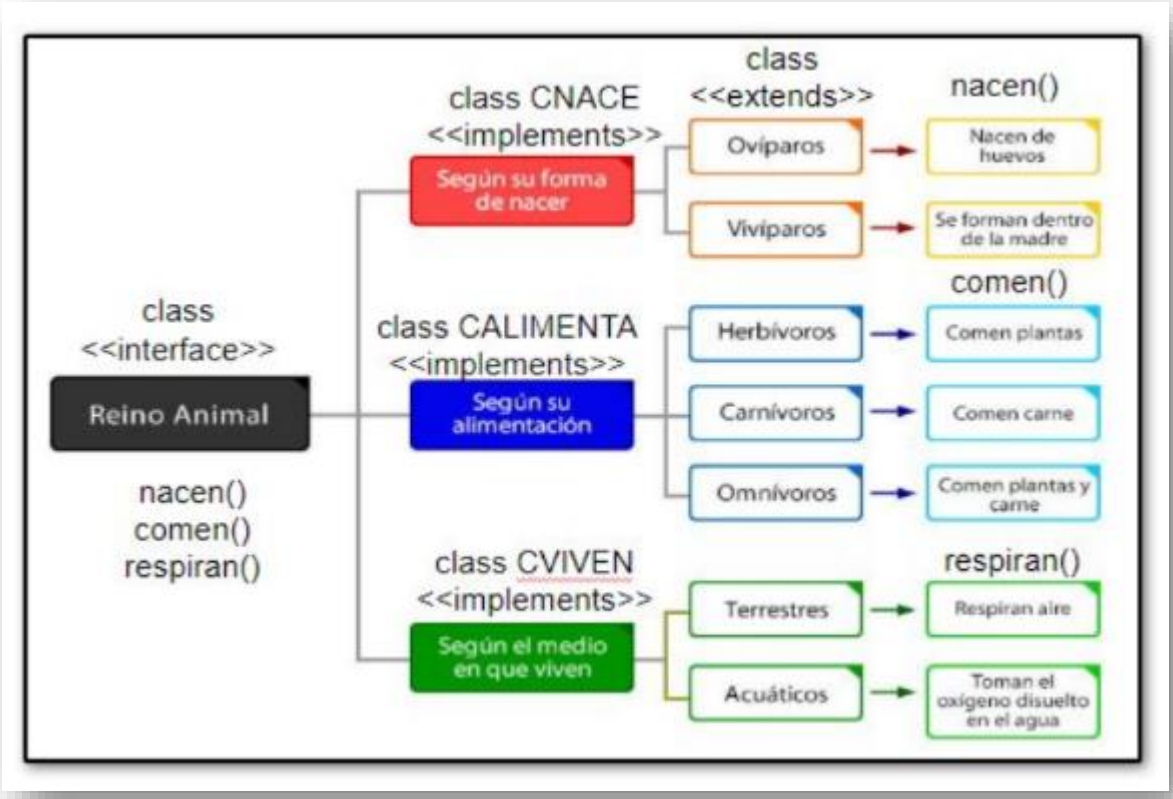
```java
// Definición de la interfaz
interface Volador {
    void despegar();
    void volar();
    void aterrizar();

    // Método predeterminado
    default void planear() {
        System.out.println("Planeando...");
    }
```

```
    // Método estático
    static void info() {
        System.out.println("Esta es una interfaz para objetos voladores.");
    }
}

// Clase que implementa la interfaz
class Avion implements Volador {
    public void despegar() {
        System.out.println("Avión despegando.");
    }

    public void volar() {
        System.out.println("Avión volando.");
    }

    public void aterrizar() {
        System.out.println("Avión aterrizando.");
    }
}

// Clase principal
public class Main {
    public static void main(String[] args) {
        // Crear un objeto de la clase Avion
        Avion avion = new Avion();

        // Llamar a los métodos de la interfaz a través del objeto del tipo
de la interfaz
        avion.despegar();
        avion.volar();
        avion.aterrizar();
        avion.planear(); // Llamada al método predeterminado

        // Llamada al método estático de la interfaz
        Volador.info();
    }
}
```

En este ejemplo, Volador es una interfaz que define varios métodos abstractos (despegar(), volar(), aterrizar()), un método predeterminado (planear()) y un método

estático (info()). La clase Avión implementa la interfaz Volador, proporcionando implementaciones para todos los métodos definidos en la interfaz.

PRACTICA




```
interface ReinoAnimal{
    void comen(int tipo);
    void nacen(int tipo);
}

class CNACEN implements ReinoAnimal{
    public void comen(int tipo){}
    public void nacen(int tipo){
        if(tipo==1)
        System.out.println("OVIPAROS Nace de huevos");
        else if(tipo==2)
        System.out.println("VIVIPAROS Se forman dentro de la madre");
    }
```

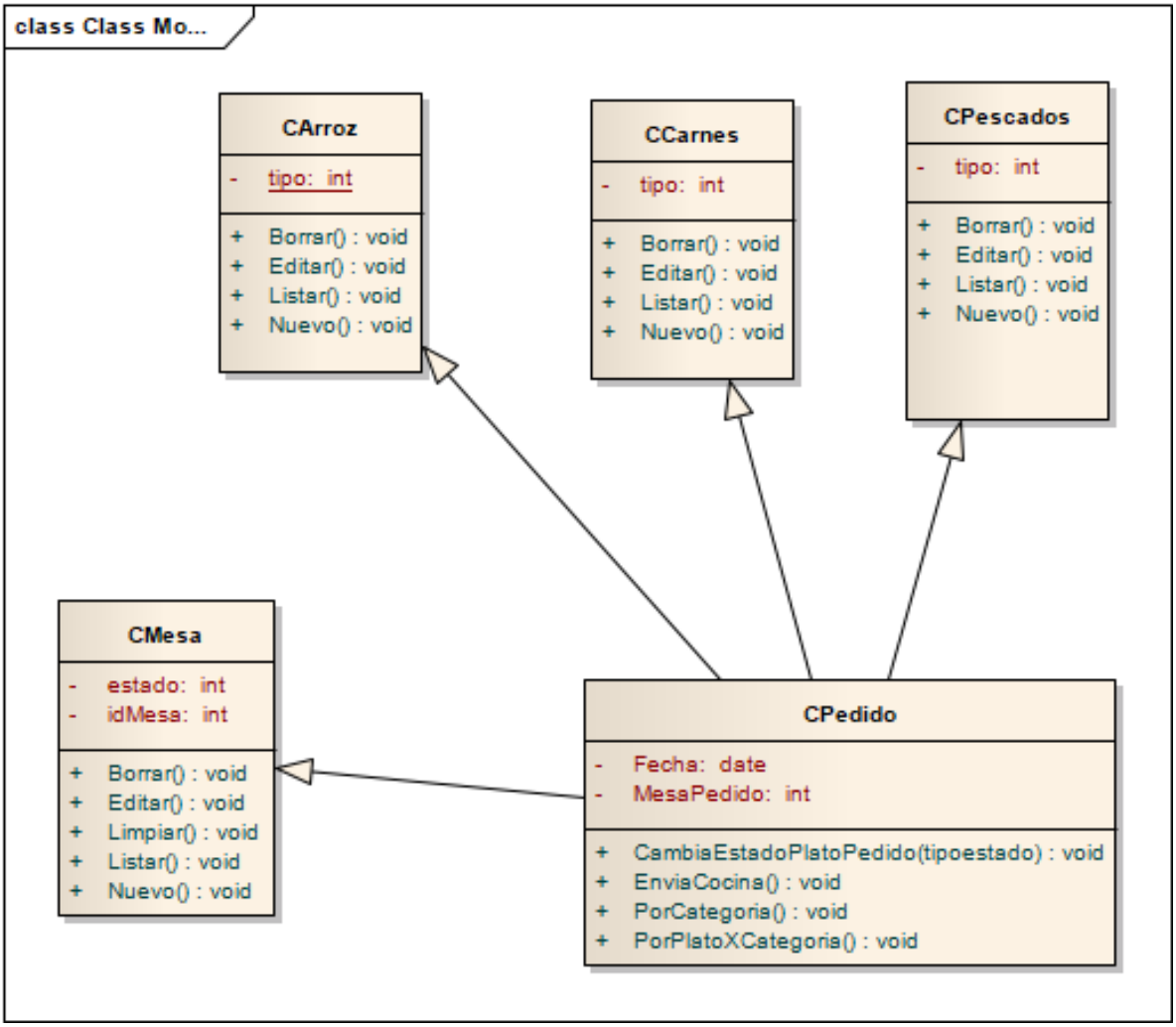
```
}
class CALIMENTAN implements ReinoAnimal{
    public void comen(int tipo){
        if(tipo==1)
        System.out.println("HERVIVOROS comen hiervas");
        else if(tipo==2)
        System.out.println("CARNIVOROS comen carne");
        else if(tipo==3)
        System.out.println("OMNIVOROS comen hierva y carne ");
    }
    public void nacen(int tipo){}
}
class Oviparos extends CNACEN{
    Oviparos(){
        nacen(1);
    }
}
class Viviparos extends CNACEN{
    Viviparos(){
        nacen(2);
    }
}
class Hervivoros extends CALIMENTAN{
    Hervivoros(){
        comen(1);
    }
}
class Carnivoros extends CALIMENTAN{
    Carnivoros(){
        comen(2);
    }
}
class Ovnivoros extends CALIMENTAN{
    Ovnivoros(){
        comen(3);
    }
}

class Main{
    public static void main(String[] args){
        Viviparos viviparo = new Viviparos();
        Oviparos oviparo = new Oviparos();
```

```
        Hervivoros hervivoros=new Hervivoros();
        Carnivoros carnivoros=new Carnivoros();
        Ovnivoros ovnivoros=new Ovnivoros();
    }
}
```

```
VIVIPAROS Se forman dentro de la madre
OVIPAROS Nace de huevos
HERVIVOROS comen hiervas
CARNIVOROS comen carne
OMNIVOROS comen hierva y carne
```

RETO: Termine la codificación de: "Según el medio en que viven"

Practica en clase:

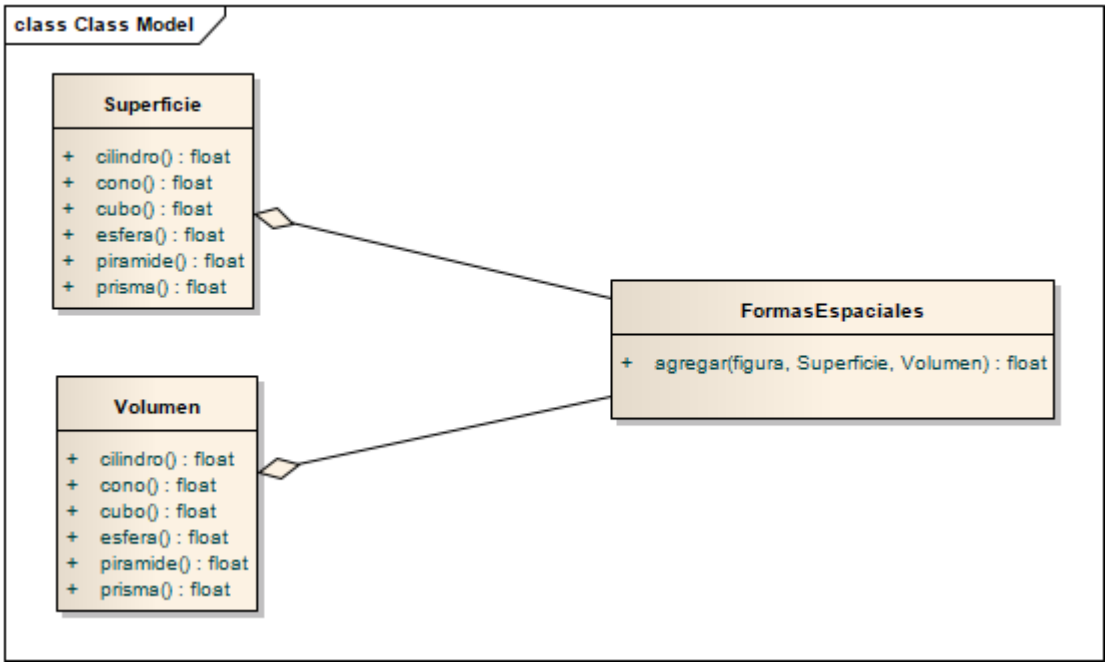


Desarrollar en JAVA el anterior diagrama de clase, junto al instructor que dará las orientaciones para su construcción, no olvide guardar una copia en el portafolio de evidencias.

Condiciones de la práctica:

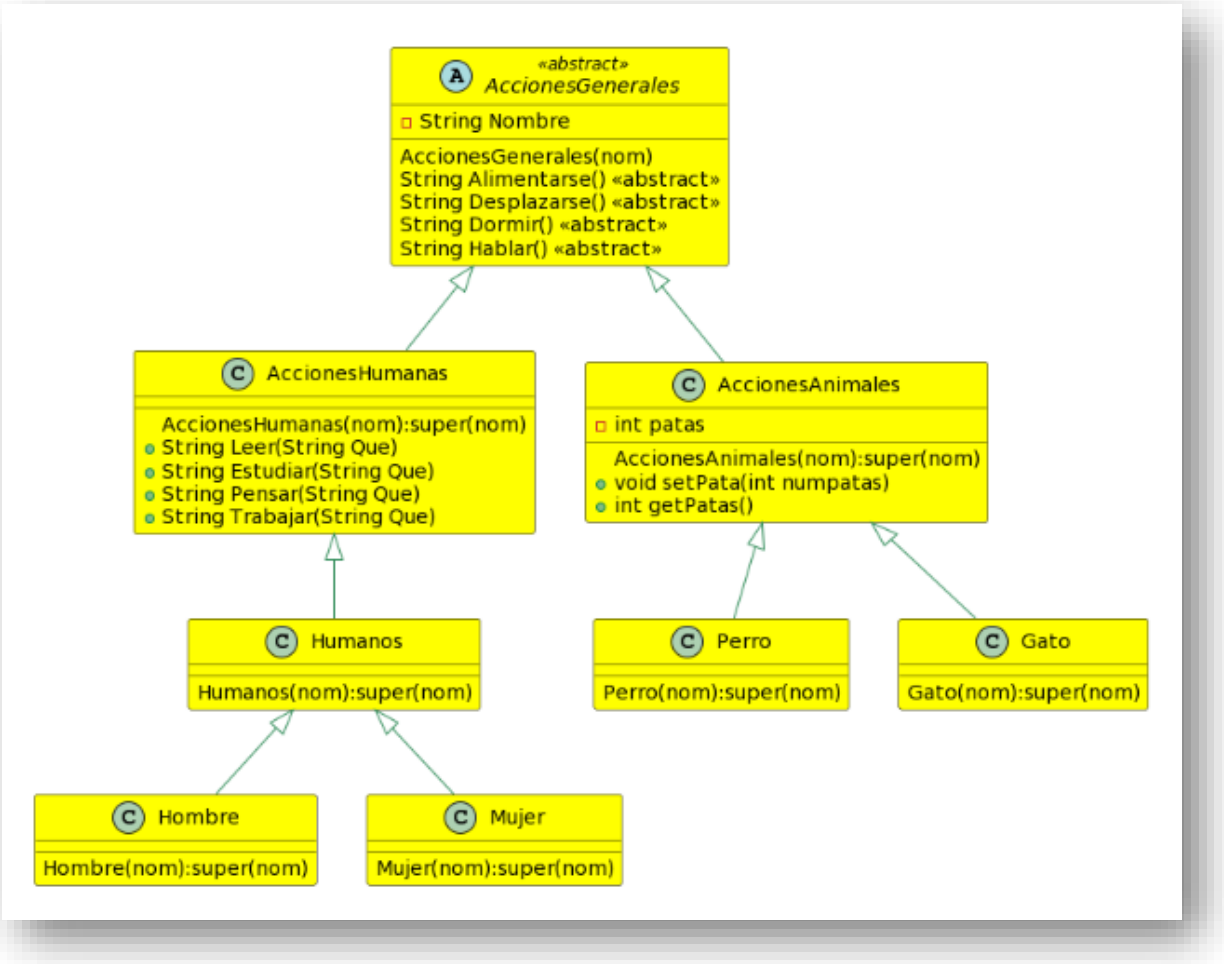
- No se permite cambiar el nombre a ningún método ni atributo de la práctica.
- Todos los métodos deben escribir el origen y el método por ejemplo “Desde la clase CArroz y el método Nuevo”
- No es permitido utilizar NameSpace

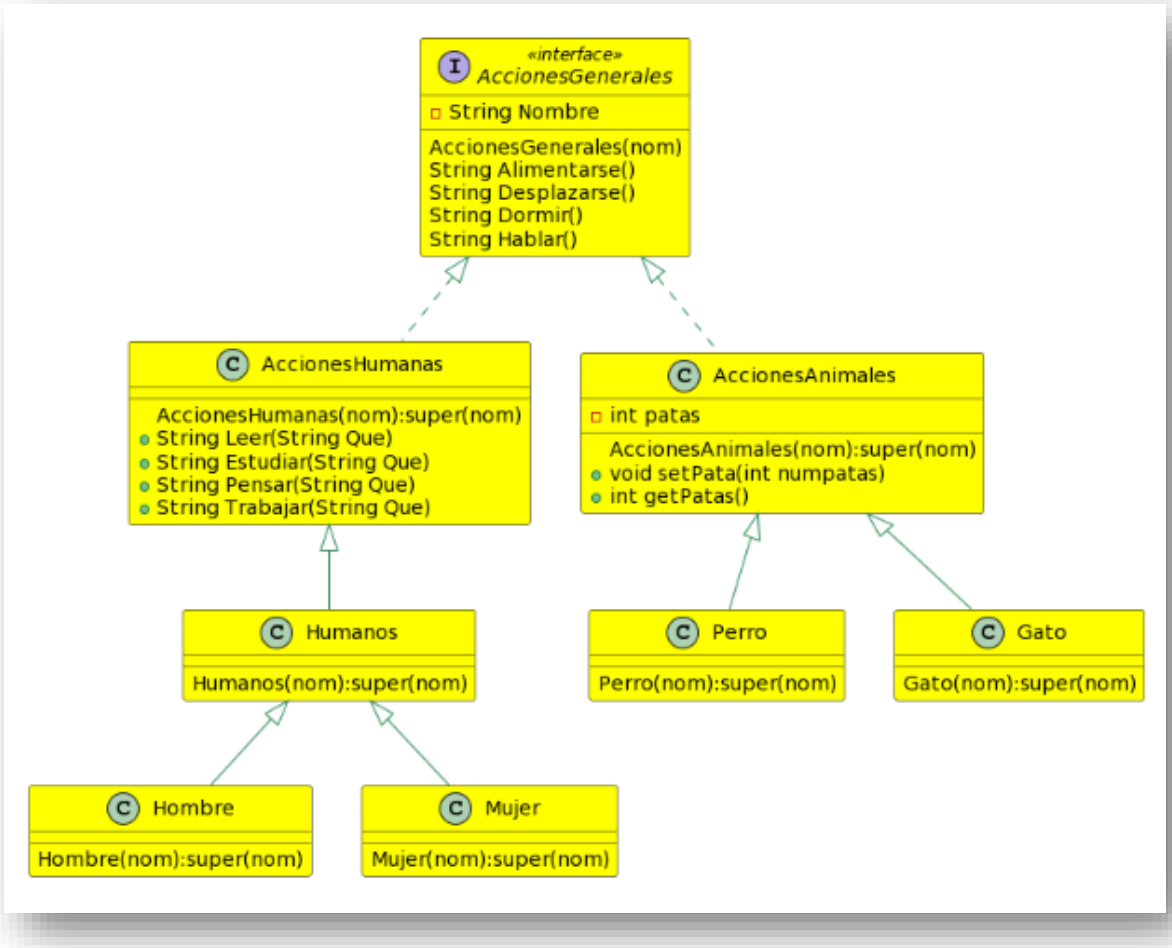
Ejemplo de Agregación


class Class Model
Superficie
+ cilindro() : float
+ cono() : float
+ cubo() : float
+ esfera() : float
+ piramide() : float
+ prisma() : float
Volumen
+ cilindro() : float
+ cono() : float
+ cubo() : float
+ esfera() : float
+ piramide() : float
+ prisma() : float
FormasEspaciales
+ agregar(figura, Superficie, Volumen) : float

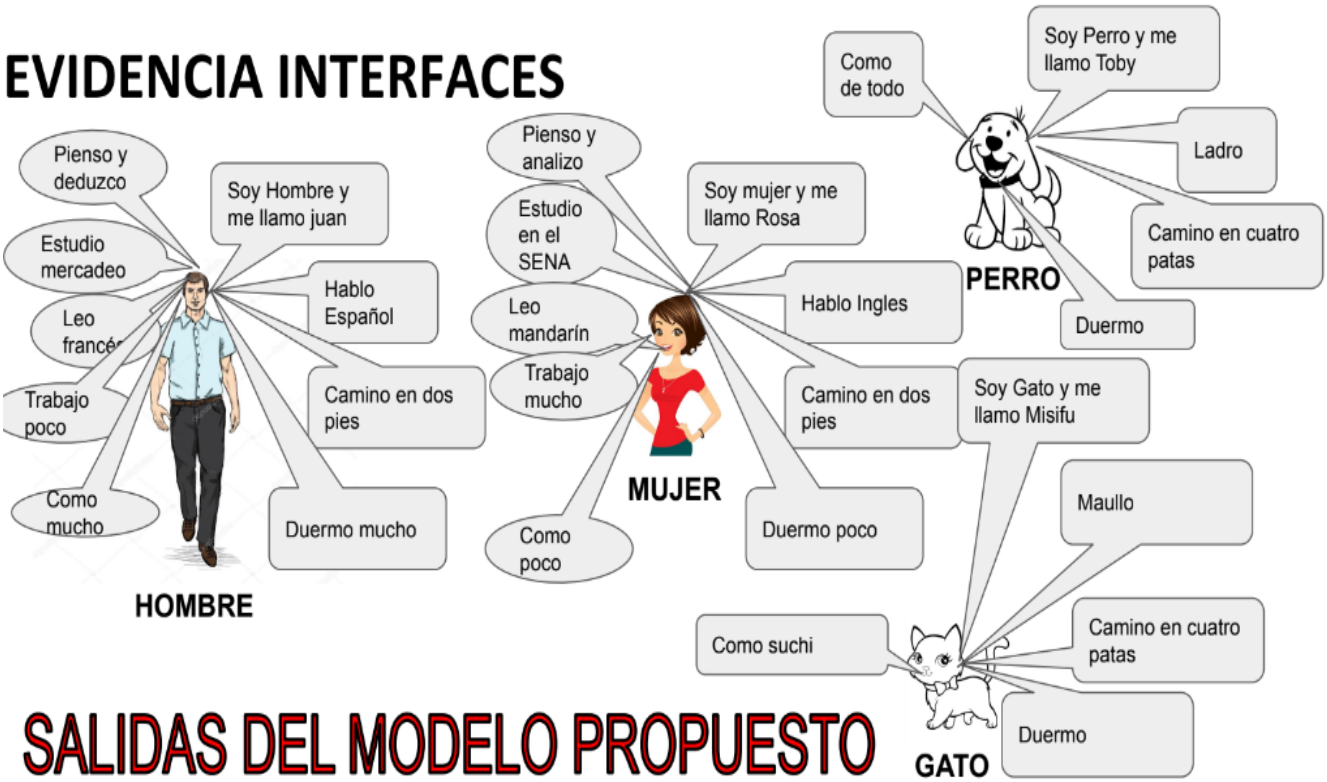
Ejemplo de composición.

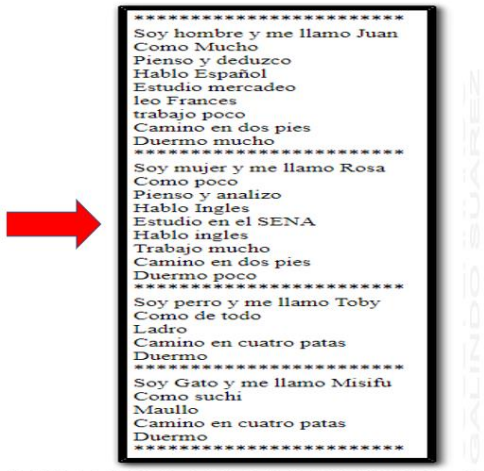
PRACTICA No 1:



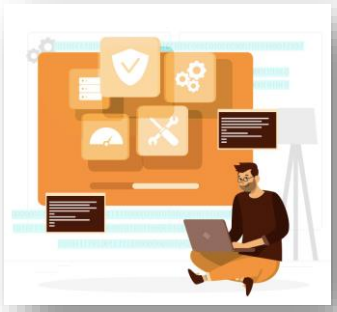

«interface»
AccionesGenerales
String Nombre
AccionesGenerales(nom)
String Alimentarse()
String Desplazarse()
String Dormir()
String Hablar()
AccionesHumanas
AccionesHumanas(nom):super(nom)
String Leer(String Que)
String Estudiar(String Que)
String Pensar(String Que)
String Trabajar(String Que)
AccionesAnimales
int patas
AccionesAnimales(nom):super(nom)
void setPata(int numpatas)
int getPatas()
Humanos
Humanos(nom):super(nom)
Perro
Perro(nom):super(nom)
Gato
Gato(nom):super(nom)
Hombre
Hombre(nom):super(nom)
Mujer
Mujer(nom):super(nom)

Desarrollar la evidencia de interfaces de acuerdo a la salida de la siguiente grafica:





SALIDAS DEL MODELO PROPUESTO



Desarrolle esta evidencia en archivo , de acuerdo con la gráfica anterior y teniendo en cuenta las salidas del modelo propuesto, en la herramienta de su preferencia y envíela al instructor, guarde una copia en el portafolio del aprendiz; este taller se debe realizar con los integrantes del grupo de proyecto de formación.

Recuerde enviarlo en un archivo ZIP no RAR ni otra extensión de comprimido y colocar como nombre "**T3_poo.zip**" y subirlo al LMS individualmente.