



P00

Programación Orientada a Objetos

01 {

[POO (Continuación)]

- Del
- Metodo __str__()
- Pilares de POO
- Herencia
- Polimorfismo
- Composicion

}

Del {

En Python, la **palabra clave del** se usa para eliminar referencias a objetos.

Podemos usarla para eliminar objetos y atributos.



```
carro1 = Carro2('Toyota', 'Corolla', 2023, 'rojo', 4, 5)
del carro1.modelo #Elimina el atributo modelo
del carro1 #Elimina el objeto carro1
```

Método `__str__()` {

El método `__str__()` define cómo se representará el objeto, cuando se requiera que esté sea un **string**.

Es particularmente útil usarlo en conjunto con la función `print()`.

Si el método `__str__()` no está definido en una clase, al imprimir un objeto de esta nos retornará su **dirección de memoria**.

El método `__str__()` se llama automáticamente cuando se intenta imprimir un objeto en la consola.

}

1 Método `__str__()` {

2

3

4

5

6

7

8

9

10

11

12

13


14

}



```
def __str__(self):  
    return f"Alumno: {self.nombre}. Nota: {self.nota}."
```

Método `__str__()` {



```
class Alumno:
    # inicializamos los atributos
    def __init__(self,nombre,nota):
        self.nombre = nombre
        self.nota = nota

    # Metodo __str__()
    def __str__(self):
        return f"Alumno: {self.nombre}. Nota: {self.nota}."
```

```
1  Método __str__() {
```



```
6  alumno1 = Alumno("Ivan", 9)
7  print(alumno1)
```

```
11  Alumno: Ivan. Nota: 9.
```

```
12  }
13
14
```

Ejercicio 27 {

Realizar un programa que conste de una clase llamada **Alumno** que tenga como atributos el nombre y la nota del alumno.

Definir los métodos para inicializar sus atributos, imprimirlos y mostrar un mensaje con el resultado de la nota y si ha aprobado o no.

}

Llamar métodos dentro de métodos {



```
# funcion para imprimir todos los datos
def imprimir_todo(self):
    print("Nombre: " + self.nombre)
    print("Nota: " + str(self.nota))
    self.resultado()

# funcion para obtener el resultado
def resultado(self):
    if self.nota < 10:
        print("El alumno ha suspendido.")
    else:
        print("El alumno ha aprobado.")
```

Llamar métodos dentro de métodos {

```
class Alumno:
    # inicializamos los atributos
    def __init__(self,nombre,nota):
        self.nombre = nombre
        self.nota = nota

    # funcion para imprimir todos los datos
    def imprimir_todo(self):
        print("Nombre: " + self.nombre)
        print("Nota: " + str(self.nota))
        self.resultado()

    # funcion para obtener el resultado
    def resultado(self):
        if self.nota < 10:
            print("El alumno ha suspendido.")
        else:
            print("El alumno ha aprobado.")
```

Llamar métodos dentro de métodos {

```
from Alumno import *
```

```
alumno1 = Alumno("Ivan", 9)  
alumno1.imprimir_todo()
```

```
Nombre: Ivan  
Nota: 9  
El alumno ha suspendido.
```

Ejercicio 28 {

Se le pide que cree un algoritmo que permita el **registro de personas**.

Debe guardar cada persona registrada en una **lista**, también se le solicita que use clases.

La información que se desea guardar es el *nombre*, *apellido*, *edad* y *DNI* de cada persona que se registre en el sistema.

Además, debe crear un método **mostrar()** que imprima toda la información de la persona, y un método **esMayor()** que retorne **verdadero** si la persona tiene 18 años o más, y que retorne **falso** en caso contrario.

}

Ejercicio 29 {

Cree una clase llamada **Cuenta** que tenga los siguientes atributos: titular (que es una persona) y cantidad (puede tener decimales). El titular será obligatorio y la cantidad es opcional.

Construya los siguientes **métodos** para la clase:

- Un **constructor**.
- **mostrar()** que muestra los datos de la cuenta.
- **ingresar(cantidad)** que ingresa una cantidad a la cuenta, si la cantidad introducida es negativa, se le indica al usuario que ingreso un monto invalido.
- **retirar(cantidad)** que retira una cantidad a la cuenta. La cuenta puede estar en números rojos.

¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS?

Es un **paradigma** de programación que **organiza** las **funciones** en **entidades** llamadas **objetos**.

- Los **objetos** se crean a partir de una **plantilla** llamada **clase**. Cada objeto es una **instancia** de su clase.

CLASE



INSTANCIACIÓN

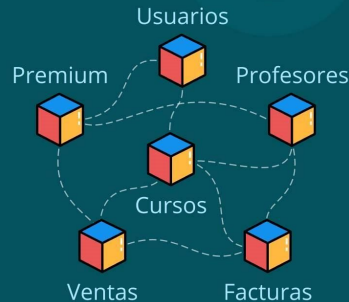
OBJETO



- Los objetos tienen **datos (atributos)** y **funcionalidades (métodos)**.



- En una aplicación los objetos están separados **pero se comunican entre ellos**.



ATRIBUTOS

Nombres
Apellidos
Correo
Contraseña
Premium



MÉTODOS

Editar perfil
Iniciar sesión
Cerrar sesión
Cambiar contraseña
Pasar a premium



Puedes programar con este paradigma **en la mayoría de lenguajes**.



P00 vs Programación Estructurada (Secuencial) {

La **Programación Orientada a Objetos (P00)** y la **Programación Estructurada (PS)** son dos paradigmas de programación diferentes.

La Programación Estructurada (PS) se basa en el **concepto de secuencia de instrucciones**, mientras que la Programación Orientada a Objetos (P00) se basa en el **concepto de objetos**.

}

P00 vs Programación Estructurada (Secuencial) {

En la **Programación Estructurada (PS)**, el código se organiza en funciones o procedimientos.

Cada función tiene una tarea específica y se llama cuando es necesaria.

La **Programación Estructurada (PS)** es un paradigma de programación muy simple y fácil de aprender. Sin embargo, puede ser difícil de usar para crear programas grandes y complejos.

}

P00 vs Programación Estructurada (Secuencial) {

En la **Programación Orientada a Objetos (P00)**, el código se organiza en objetos. Los objetos son entidades que tienen datos y comportamiento.

Los objetos se pueden utilizar para crear programas más organizados y fáciles de entender.

La **Programación Orientada a Objetos (P00)** también es un paradigma de programación muy poderoso y se puede utilizar para crear programas grandes y complejos.

}

Programación Orientada a Objetos	Programación Estructurada
Poco código en varios lugares.	Mucho código en varios lugares.
Permite reutilizar código, priorizando la eficiencia.	Tiende a repetir código, lo que llega a ser ineficiente.
Enfoque por objetos.	Enfoque por bloques de código.
Más fácil de depurar.	Más difícil de depurar.

Pilares de la P00{

La programación orientada a objetos como paradigma, se basa en cuatro pilares fundamentales: **abstracción**, **encapsulamiento**, **polimorfismo** y **herencia**.

Estos cuatro pilares son los conceptos básicos de la P00.

}

Pilar Abstraccion {

La P00 nos permite reutilizar código mediante la abstracción. La abstracción es el proceso en el cual nos preguntamos **qué atributos y métodos puede necesitar nuestra clase.**

La abstracción oculta al usuario final la funcionalidad interna de la aplicación. Un ejemplo de esto sería cómo sabemos usar nuestro teléfono, pero probablemente no sabemos exactamente lo que ocurre dentro de este cada vez que se abre una aplicación.

}

Pilar Encapsulamiento {

El encapsulamiento es el **proceso de ocultar los datos internos de un objeto** de los usuarios del objeto.

Los usuarios del objeto solo pueden acceder a los datos externos del objeto, que se denominan interfaz.

El encapsulamiento se utiliza para proteger los datos de un objeto de ser modificados accidentalmente o maliciosamente.

}

Pilar Encapsulamiento {

La encapsulación es el proceso en el cual protegemos la integridad interna de los datos en una clase.

Les puede interesar el siguiente artículo:

[Encapsulación en Python](#)

}

Pilar Herencia {

La herencia nos permite **definir una clase que hereda todos los métodos y los atributos de otra clase.**

La **clase principal** es la clase de la que se hereda, también llamada **clase base** o **clase padre**.

La **clase secundaria** es la clase que hereda de otra clase, también llamada **clase derivada** o **clase hijo**.

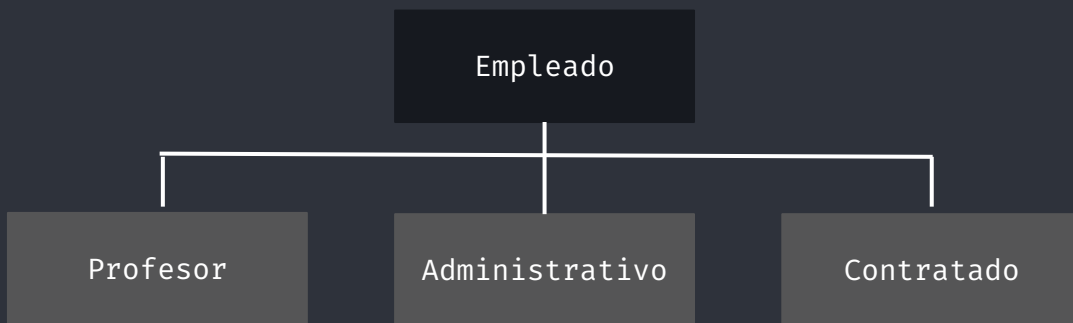
De esta forma, la herencia nos permite **definir múltiples subclases a partir de una clase ya definida.**

}

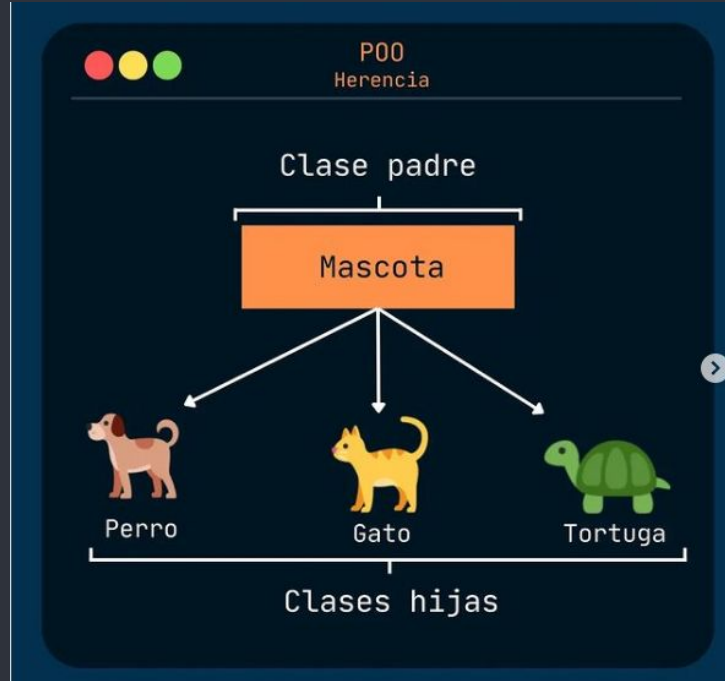
Pilar Herencia {

En resumen, la herencia es la propiedad que nos permite definir nuevas clases usando como base clases ya existentes.

La **clase hijo** hereda los atributos y el comportamiento de la **clase padre**.



Pilar Herencia {



Pilar Herencia {

El propósito principal es seguir el principio **"Don't repeat yourself" (DRY)**. Podemos reutilizar mucho código implementando todos los componentes compartidos en clases principales y secundarias.

Se puede ver cómo el concepto de herencia genética en la vida real.

Los **hijos (subclases)** son el resultado de la herencia de **padres (superclases)**. Heredan **todas las características físicas (atributos)** y **algunos comportamientos comunes (métodos)**.

}

Clase principal (padre) {

Cualquier clase puede ser una clase principal porque se crea igual que como se indico antes.

```
class Persona:
```

```
    def __init__(self, nombre, apellido, cedula, correo = "", telefono = ""):
        self.nombre = nombre
        self.apellido = apellido
        self.cedula = cedula
        self.correo = correo
        self.telefono = telefono
```

```
}
```

poo.py



```
class Persona:
```

```
    def __init__(self, nombre, apellido, carnet, cedula, materias, correo = "", telefono = ""):
        self.nombre = nombre
        self.apellido = apellido
        self.cedula = cedula
        self.correo = correo
        self.telefono = telefono

    def imprimirDatos(this):
        print("Persona: {} {} \nCedula: {} \nCorreo: {} \nTelefono: {}".format(this.nombre, this.apellido, this.cedula, this.correo, this.telefono))
```

Clase secundaria (hijo) {

Para crear una clase que herede de otra clase, hay que **enviar la clase principal como parámetro a la clase secundaria.**

Se debe tener en cuenta que al agregar la **funcion `__init__()`** a la clase hijo, esta clase ya no hereda esa **función `__init__()`** de la clase padre.

Si queremos que siga heredando el constructor pero a su vez queremos que la subclase tenga sus propios atributos, entonces tenemos que llamar al constructor de la clase padre dentro del constructor de la clase hijo.

}

Clase secundaria (hijo) {

```
from Persona import *
```

```
class Estudiante(Persona):
```

```
    def __init__(self, nombre, apellido, cedula, correo = "", telefono = "", carnet = "", materias = []):  
        Persona.__init__(self, nombre, apellido, cedula, correo, telefono)  
        self.carnet = carnet  
        self.materias = materias
```

```
}
```

Clase secundaria (hijo) {

```
from Persona import *
```

```
# Se define la clase hijo indicando la clase padre
```

```
class Estudiante(Persona):
```

```
    # Se pasan los atributos de ambas clases
```

```
    def __init__(self, nombre, apellido, cedula, correo = "", telefono = "", carnet = "", materias = []):
```

```
        # Llamamos al constructor de la clase padre
```

```
        Persona.__init__(self, nombre, apellido, cedula, correo, telefono)
```

```
        # Definimos los atributos de la clase hijo
```

```
        self.carnet = carnet
```

```
        self.materias = materias
```

```
}
```

Clase secundaria (hijo) {

```
from Estudiante import *

persona = Persona("Stefani", "Perez", 1, "s@", "0424-.....")
estudiante = Estudiante("Carlos", "Contreras", 2, "c@", "0412-.....", 2, ["Algoritmos", "Estructuras de Datos"])
print(type(persona))
persona.imprimirDatos()
print(type(estudiante))
estudiante.imprimirDatos()
```

}

Función `super()` {

En Python, **`super()`** es una función que se utiliza para **llamar a un método de una clase padre desde una clase hijo**.

También, nos permite tanto **acceder a los atributos de la clase padre**, como llamar al **constructor de la clase padre**.

}

1 Función super(). Clase Padre. {
2
3
4
5
6
7
8
9
10
11
12
13 }
14



```
class Persona:
```

```
    def __init__(self, nombre, apellido, cedula, correo = "", telefono = ""):  
        self.nombre = nombre  
        self.apellido = apellido  
        self.cedula = cedula  
        self.correo = correo  
        self.telefono = telefono
```

Función super(). Clase Hijo. {

```
from Persona import *
```

```
class Estudiante(Persona):
```

```
    def __init__(self, nombre, apellido, cedula, correo = "", telefono = "", carnet = "", materias = []):  
        super().__init__(nombre, apellido, cedula, correo, telefono)  
        self.carnet = carnet  
        self.materias = materias
```

```
}
```

Función super() {

```
from Estudiante import *

persona = Persona("Stefani", "Perez", 1, "s@", "0424-.....")
estudiante = Estudiante("Carlos", "Contreras", 2, "c@", "0412-.....", 2, ["Algoritmos", "Estructuras de Datos"])
print(type(persona))
persona.imprimirDatos()
print(type(estudiante))
estudiante.imprimirDatos()
```

}

Pilar Polimorfismo {

El polimorfismo nos permite **modificar los métodos de las clases hijo** previamente definidos en la clase padre.

De esta forma, construimos **métodos con el mismo nombre pero con diferente funcionalidad**.

Una **clase hijo** puede heredar un comportamiento definido **imprimirDatos()** pero de una manera ligeramente diferente, por ejemplo, imprimir otros atributos que no sean los de la **clase padre**.


}

Pilar Polimorfismo {

En resumen, **dos objetos de diferentes clases pueden tener métodos con el mismo nombre**, y ambos métodos pueden ser llamados con el mismo código, dando respuestas diferentes.

}

poo.py



```
class Persona:
    def __init__(self, nombre, apellido, carnet, cedula, materias, correo = "", telefono = ""):
        self.nombre = nombre
        self.apellido = apellido
        self.cedula = cedula
        self.correo = correo
        self.telefono = telefono

    def imprimirDatos(this):
        print("Persona: {} {} \nCedula: {} \nCorreo: {} \nTelefono: {}".format(this.nombre, this.apellido, this.cedula, this.correo, this.telefono))
```

poo.py



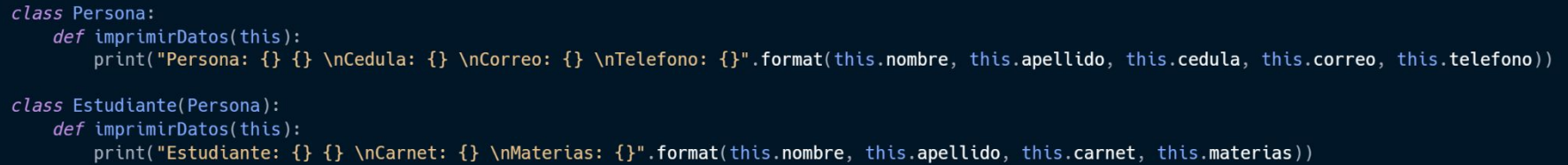
```
from Persona import *

class Estudiante(Persona):

    def __init__(self, nombre, apellido, cedula, correo = "", telefono = "", carnet = "", materias = []):
        super().__init__(nombre, apellido, cedula, correo, telefono)
        self.carnet = carnet
        self.materias = materias

    def imprimirDatos(this):
        print("Estudiante: {} {} \nCarnet: {} \nMaterias: {}".format(this.nombre, this.apellido, this.carnet, this.materias))
```


poo.py



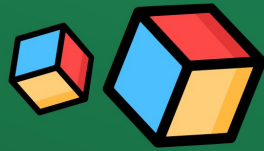
```
class Persona:
    def imprimirDatos(this):
        print("Persona: {} {} \nCedula: {} \nCorreo: {} \nTelefono: {}".format(this.nombre, this.apellido, this.cedula, this.correo, this.telefono))

class Estudiante(Persona):
    def imprimirDatos(this):
        print("Estudiante: {} {} \nCarnet: {} \nMaterias: {}".format(this.nombre, this.apellido, this.carnet, this.materias))
```

Ejercicio 30 {

- Crea una **clase padre** llamada **Libro** que tenga atributos para el *título*, *autor* y *año de publicación* del libro.
- Luego, crea una **clase hijo** llamada **LibroEducativo** que herede de **Libro** y agregue un atributo adicional para el *tema del libro educativo*. Por ejemplo: matemáticas, historia, ciencias...
- Asegúrate de que ambas clases tengan un **método info()** que muestre la información relevante del libro o del libro educativo.
- Finalmente, crea un objeto de la clase **LibroEducativo** y llama al **método info()** para mostrar la información del libro educativo.

}



¡PILARES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS!

ABSTRACCIÓN



Es el proceso de **definir los atributos y los métodos** de una clase.



ENCAPSULAMIENTO



Protege la información de manipulaciones no autorizadas.

POLIMORFISMO



Da la misma orden a varios objetos para que respondan de diferentes maneras.

HERENCIA



Las clases hijo heredan atributos y métodos de las clases padre.

Según el paradigma, la programación orientada objetos, se basa en estos 4 pilares. **Estos definen la simplicidad y la funcionalidad del código.**

Composicion {

La composición en Python se refiere a una **relación entre clases donde una clase se compone de una o más instancias de otras clases.**

En otras palabras, una **clase es un contenedor y otra clase es contenido.**

Tendríamos una **clase que contiene uno o más objetos de otra clase.**

}

Composicion {

```
class Producto:
```

```
    def __init__(self, nombre, precio):  
        self.nombre = nombre  
        self.precio = precio
```

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

```
    def getPrecio(self):  
        return self.precio
```

```
    def setNombre(self, nuevo_nombre):  
        self.nombre = nuevo_nombre
```

```
    def setPrecio(self, nuevo_precio):  
        self.precio = nuevo_precio
```

```
class Carrito:
```

```
    def __init__(self):  
        self.productos = []
```

```
    def getProductos(self):  
        return self.productos
```

```
    def agregar_producto(self, producto):  
        self.getProductos().append(producto)
```

```
    def calcular_total(self):  
        total = 0  
        for producto in self.productos:  
            total += producto.precio  
        return total
```

Ejercicio 31 {

En una escuela, se desea llevar un registro de las notas de los estudiantes. Para ello, se te pide que crees las siguientes clases:

- La **clase Estudiante** debe tener los atributos *nombre* y *notas(lista)*, y los métodos **getNombre**, **getNotas**, **agregarNota** y **promedio**.
- La **clase Nota** debe tener los atributos *materia* y *nota*, y los métodos **getMateria**, **getNota**, **setMateria** y **setNota**.
- La **clase Salon** debe tener el atributo *estudiantes(lista)* y el método **promedio_salon**.

}

Ejercicio 31 {

Una vez creadas las clases, escribe un programa en Python que:

- Cree una **instancia de la clase Salon** con 3 estudiantes.
- **Agregue una nueva nota** a uno de los estudiantes.
- Calcule el **promedio del salón**.

}

Ejercicio 32 {

Ahora se le pide que cree una “**Cuenta Joven**”, para ello debe crear una nueva clase **CuentaJoven** que herede de la **clase Cuenta (Ejercicio 29)**. Cuando se crea esta nueva clase, además del *titular* y la *cantidad*, se debe guardar una *bonificación* que estará expresada en tanto por ciento.

Construye los siguientes métodos para la clase:

- Un **constructor**.
- Los **setters** y **getters** para cada uno de los nuevos atributos.
- **esTitularValido()** que devuelve verdadero si el titular es mayor de edad pero menor de 25 años y falso en caso contrario.
- **retirar(cantidad)** que retira una cantidad a la cuenta si el titular es válido. La cuenta puede estar en números rojos.
- **mostrar()** que ahora debe devolver el mensaje de “**Cuenta Joven**” y la bonificación de la cuenta.

Ejercicio 33 {

Crear un Sistema de Registro de personal y alumnado para una Universidad.

- Debe existir una **clase Universidad**, con tres atributos: una *lista de Estudiantes*, una *lista de Profesores*, y una *lista de PersonalAdmin*.
- **Estudiantes**, **Profesores** y **PersonalAdmin** deben ser clases hijas de una **clase padre Persona**, la **clase Persona** y sus hijas deben tener un método llamado **mostrar**, donde se impriman los datos del objeto.
- Las **personas** tienen *nombre* y *apellido*, los **estudiantes** tienen *carrera*, los **profesores** tienen *materia* y *salario*, y los del *personal administrativo* tienen *departamento asignado*.
- La **clase Universidad** debe tener un método **mostrarIntegrantes**, en dónde se ejecuta el método **mostrar** de los estudiantes, empleados y profesores.

Ejercicio 33 {

- Utilizando las **clases Universidad, Estudiante, Profesor y PersonalAdmin**, crea una instancia de la **clase Universidad** y luego agrega a esta instancia un estudiante, un profesor y un miembro del personal administrativo.
- El **estudiante** debe tener un *nombre, apellido y carrera diferente*.
- El **profesor** debe tener un *nombre, apellido, materia y salario diferente*.
- El **miembro del personal administrativo** debe tener un *nombre, apellido y departamento diferente*.
- Luego, muestra información sobre todos los integrantes de la universidad utilizando el método **mostrar_integrantes** de la clase Universidad.

}

Ejercicio 34 {

Cree una **clase Cadena** que tenga como atributo un **string**. Tenga en cuenta que la clase debe tener un **constructor**.

Debe tener los métodos:

- **invertir()** que retorna la cadena invertida.
- **convertirMayusculas()** que retorna la cadena en mayúsculas.
- **convertirMinusculas()** que retorna la cadena en minúsculas.
- **esNumero()** que retorna verdadero si la cadena es un número y falso en caso contrario.
- **conseguirTamano()** que retorna el tamaño de la cadena.
- **conseguirCadena()** que retorna el atributo cadena.
- **imprimir()** que imprime la cadena.
- **cambiarCadena()** que cambia el valor del atributo cadena.