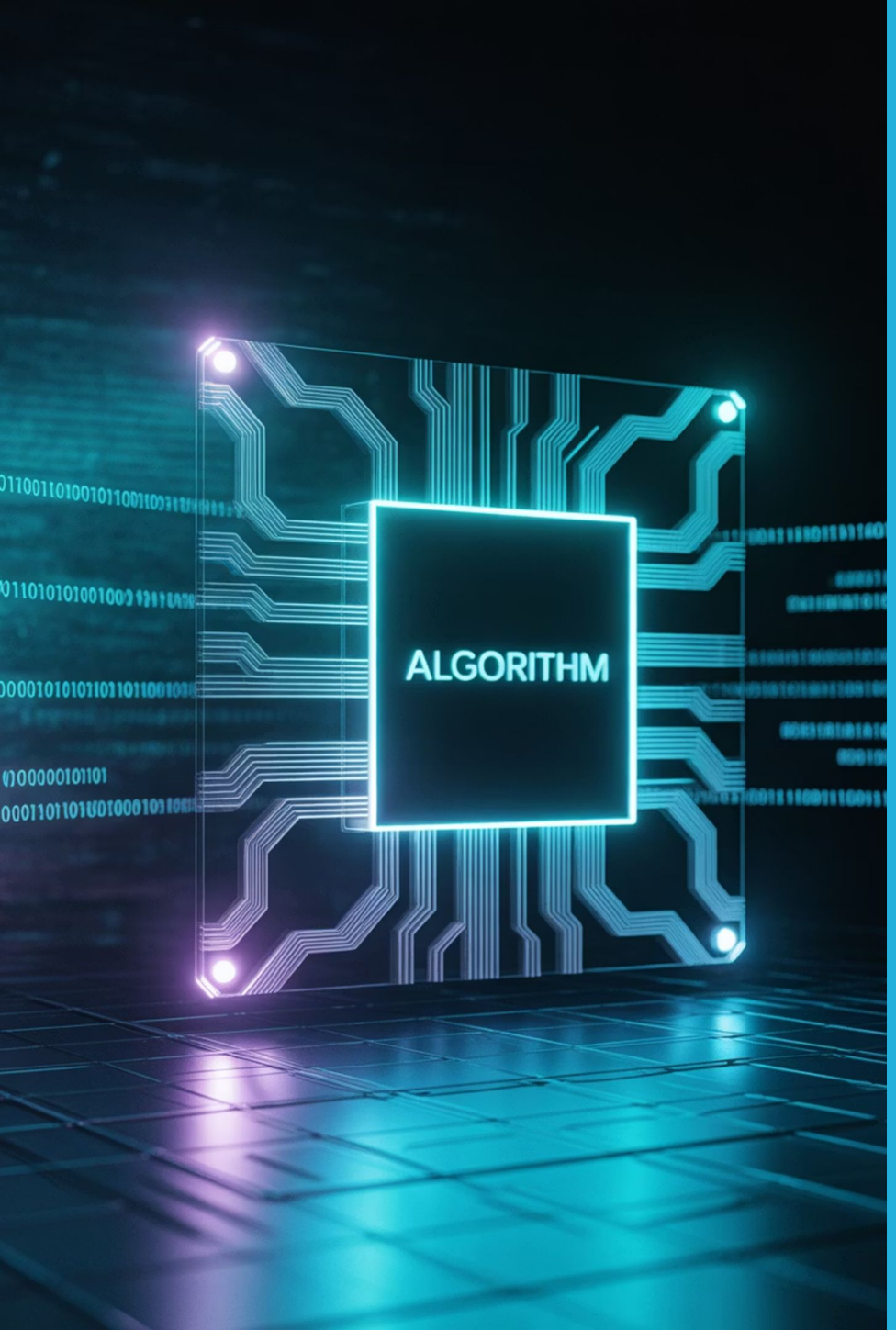
Sobreescritura y Polimorfismo en Python

Módulo 4: Programación Avanzada en Python

Bootcamp Full Stack Python

Prof. Cristian Iglesias - Mauricio Casanova

Fecha: Viernes 1 de agosto



Objetivos de la clase

- **Comprender qué es la sobreescritura de métodos.**
- **Entender el concepto de polimorfismo.**
- **Aplicar ambos conceptos con ejemplos prácticos.**
- **Crear diagramas de clases para visualizar la herencia y la modificación de comportamiento.**

Estos conceptos son fundamentales en la programación orientada a objetos y te permitirán crear código más flexible y reutilizable.

Recordemos la herencia

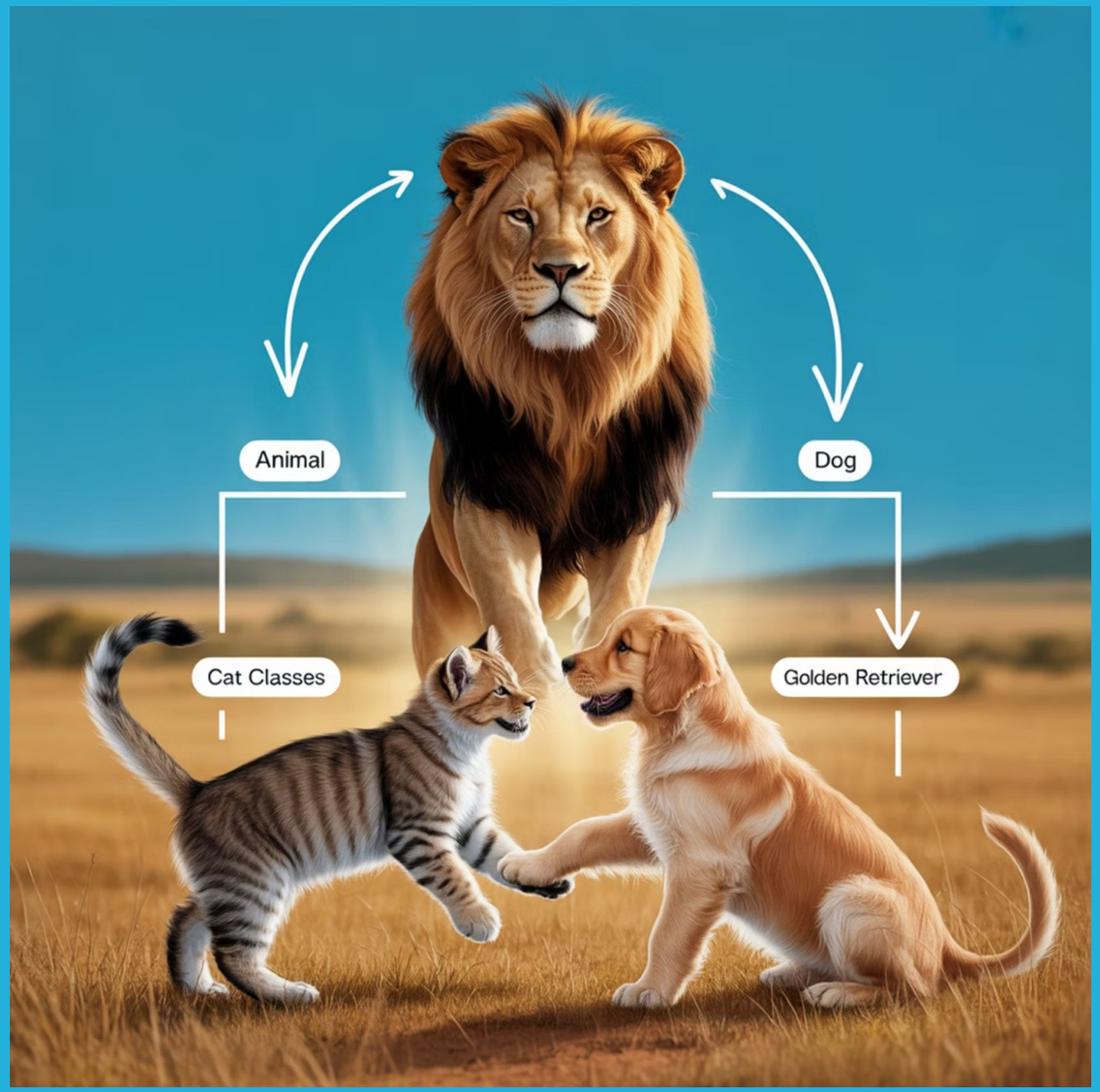
La herencia es uno de los pilares de la programación orientada a objetos que permite:

- **Una clase hija puede heredar atributos y métodos de una clase padre.**
- **Reutilizar código y establecer jerarquías.**

Pero a veces la clase hija necesita comportarse diferente.

Ahí es donde aparece la sobreescritura.

La herencia permite crear estructuras jerárquicas, pero la sobreescritura permite personalizar comportamientos.



Qué es la sobreescritura de métodos

La sobreescritura de métodos ocurre cuando:

- **Una clase hija redefine un método de su clase padre.**
- **El nombre y parámetros son iguales, pero el contenido cambia.**
- **Es como cuando heredas una receta pero le cambias ingredientes.**

Esto permite a las clases hijas personalizar el comportamiento heredado sin cambiar la estructura del programa.


```
class Animal: def hacer_truco(self): print("Hace un truco")class Gato(Animal):
```

Ejemplo completo de sobreescritura

```
class Animal:    def __init__(self, nombre):        self.nombre = nombre
def hacer_truco(self):        print(f"{self.nombre} hace un truco")class
Gato(Animal):    def hacer_truco(self):        print(f"{self.nombre} te
ignora")        print(f"{self.nombre} se rehúsa a hacer el truco")miu =
Gato("Miu")miu.hacer_truco()
```

Salida:

```
Miu te ignoraMiu se rehúsa a hacer el truco
```

En este ejemplo, el gato redefine completamente el comportamiento heredado de Animal. El

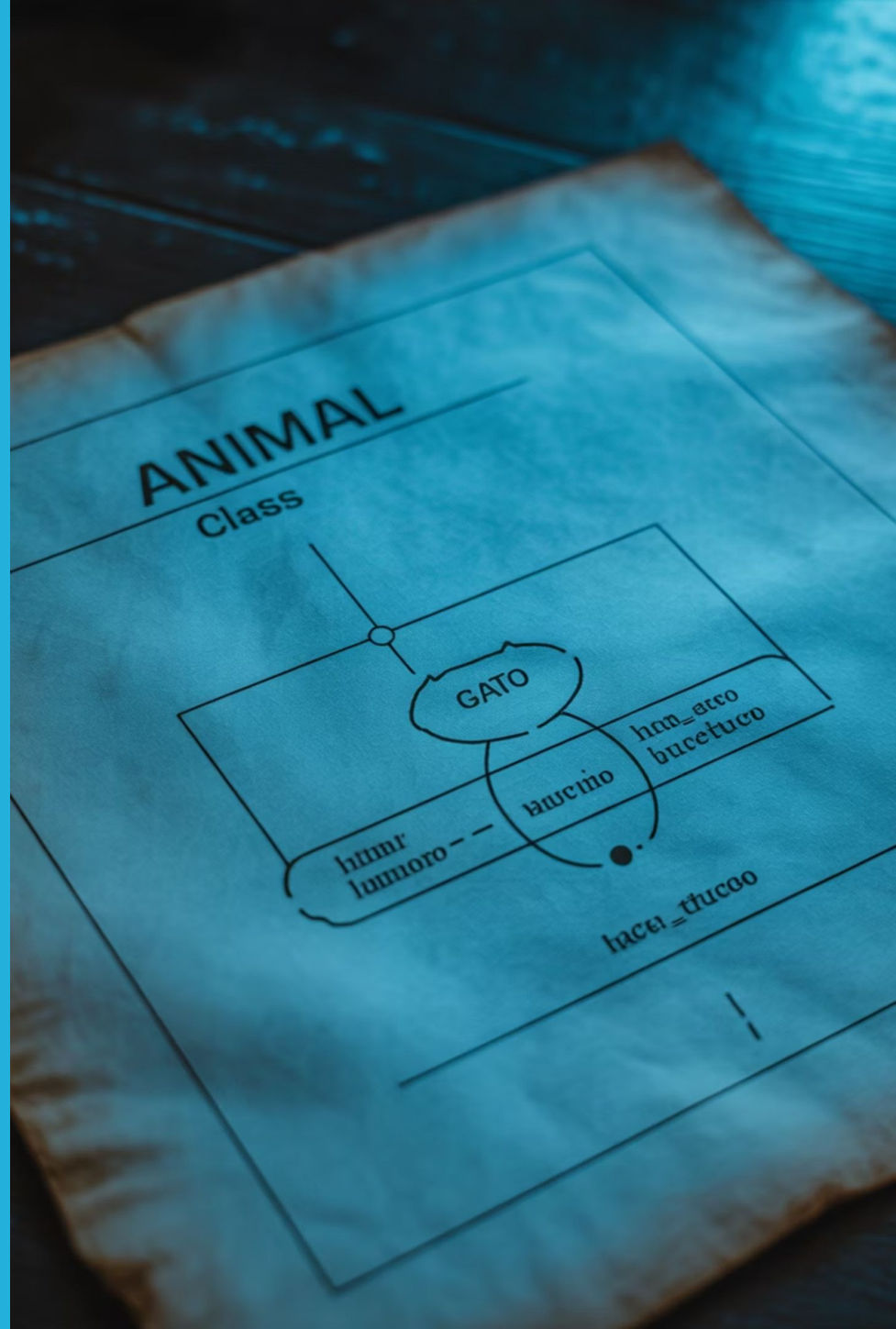
Diagrama UML de la sobreescritura

Los diagramas UML nos ayudan a visualizar la estructura de clases y la sobreescritura de métodos:

En el diagrama podemos ver:

- La clase Animal con el método hacer_truco()
- La subclase Gato que sobreescribe ese método
- Los atributos como nombre compartidos entre ambas clases
- La relación de herencia representada por la flecha

Los diagramas UML son herramientas poderosas para visualizar la estructura de nuestro código antes de implementarlo.



Qué es el polimorfismo

El polimorfismo significa literalmente "muchas formas" y permite:

- **Usar el mismo método pero con comportamientos diferentes según la clase.**
- **El método se llama igual en varias clases relacionadas.**
- **Tratar objetos de diferentes clases de manera uniforme.**

Analogía: El botón "Hablar" hace que un perro ladre, un gato maúlle y un loro hable - misma acción, diferentes comportamientos.

El polimorfismo nos permite escribir código más genérico y flexible, que puede trabajar con

Ejemplo de polimorfismo

Clase base

```
class Animal:    def
hacer_sonido(self):
raise NotImplementedError
```

La clase base define la interfaz que todas las subclases deben implementar.

Implementación Perro

```
class Perro(Animal):    def
hacer_sonido(self):
print("woof woof")
```

Implementación Gato

```
class Gato(Animal):    def
hacer_sonido(self):
print("miiiaaaaauuu")
```

`raise NotImplementedError` indica: "Este método debe implementarse en las subclases". Esto crea una especie de contrato que las clases hijas deben cumplir.

Invocando el comportamiento polimórfico

```
animales = [Perro(), Gato()]for animal in animales:    animal.hacer_sonido()
```

Salida:

```
woof woofmiiiaaaaauuu
```

Todos los animales responden al mismo mensaje: `hacer_sonido()`, pero cada uno a su manera.

Lo importante aquí es que el código que llama a `hacer_sonido()` no necesita saber con qué tipo de animal está trabajando.

Este es el poder del polimorfismo: permite tratar objetos de diferentes clases de manera

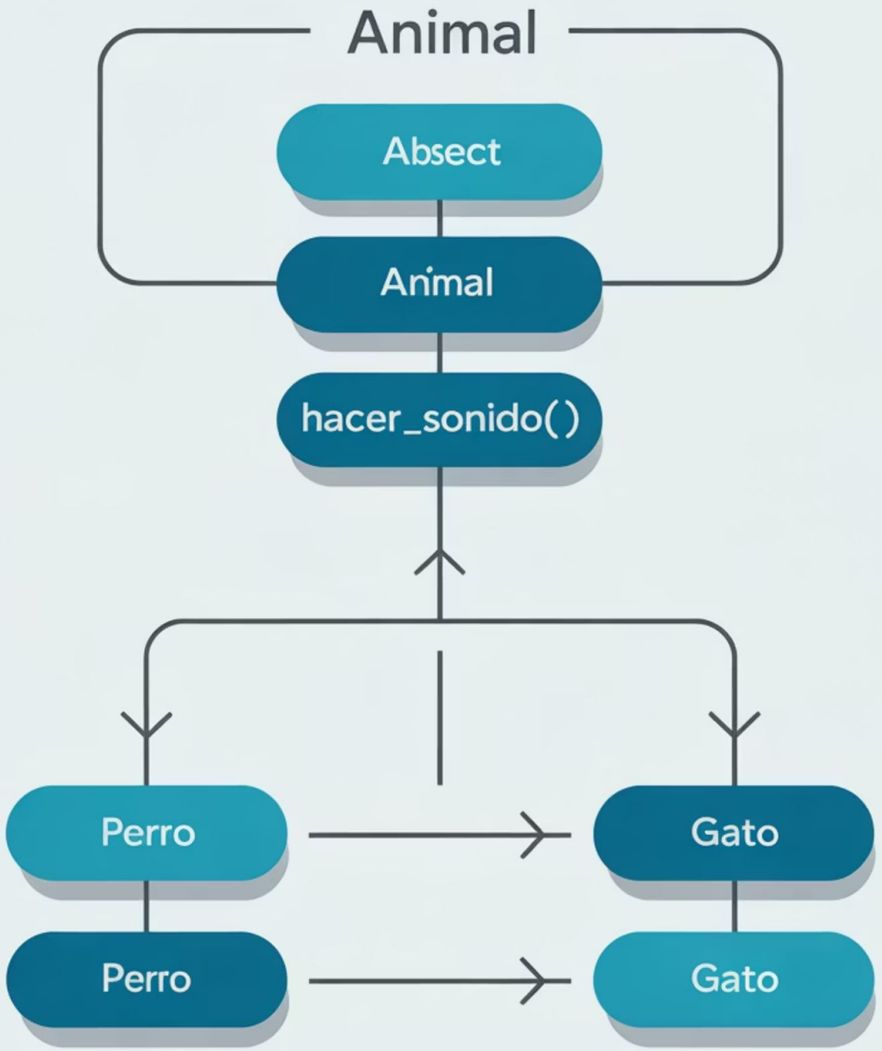


Diagrama UML del ejemplo polimórfico

En este diagrama UML podemos observar:

- **La clase abstracta Animal con el método hacer_sonido() (en cursiva por ser abstracto)**
- **Las subclases Perro y Gato que implementan ese método**
- **La relación de herencia representada por las flechas**
- **Los métodos sobreescritos en cada subclase**

Este tipo de diagrama puede hacerse en Draw.io como tarea grupal para reforzar la comprensión de estas estructuras.

Aplicación práctica en videojuegos

Mago

```
class Mago(Personaje):    def atacar(self):        print("Lanza hechizo")
```

Guerrero

```
class Guerrero(Personaje):    def atacar(self):        print("Usa espada")
```

Arquero

```
class Arquero(Personaje):    def atacar(self):        print("Dispara flecha")
```

El polimorfismo permite manejar a todos los personajes con la misma función `atacar()`, simplificando enormemente el código del juego.

Beneficios del polimorfismo

Reutilización de código

El mismo código puede trabajar con diferentes tipos de objetos, reduciendo la duplicación.

Mayor legibilidad

El código es más claro y conciso, ya que se centra en lo que se hace, no en cómo se hace.

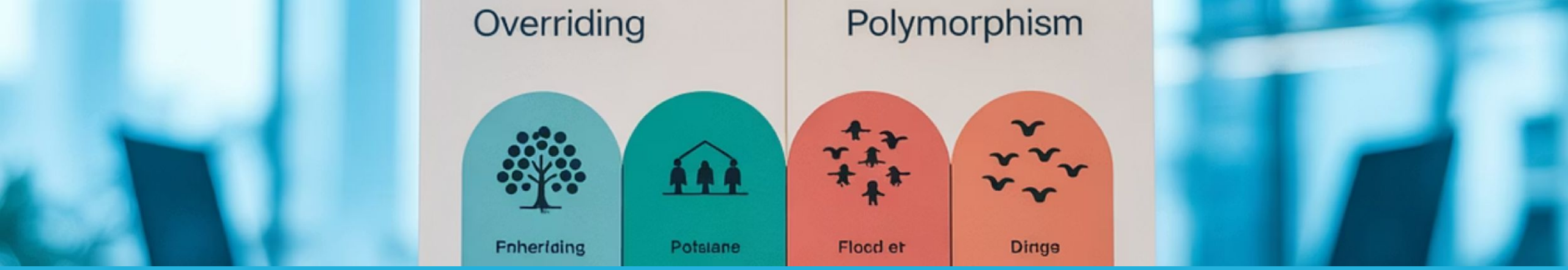
Escalabilidad

Agregar nuevos tipos de objetos no requiere modificar el código existente.

Flexibilidad

Trabajar con listas de objetos diversos sin preocuparse por sus tipos específicos.

Puedes cambiar los objetos sin cambiar el código que los usa, lo que hace que tu programa sea más adaptable y fácil de mantener a largo plazo.



Comparación visual de los conceptos

Sobreescritura

- Cambio específico en una subclase
- Se centra en modificar comportamiento heredado
- Funciona con una jerarquía simple
- Ejemplo: Un gato ignora el comando que un perro sí obedece

Polimorfismo

- Mismo nombre, diferentes comportamientos
- Se centra en la interacción uniforme
- Permite trabajar con colecciones heterogéneas
- Ejemplo: Varios animales responden a "hacer_sonido()" cada uno a su manera

Buenas prácticas al sobreescribir

- **Usa siempre el mismo nombre**

  Mantén la firma del método igual para evitar confusiones.

- **Mismo número de parámetros**

  Conserva los mismos parámetros para mantener la compatibilidad.

- **Usa super() cuando sea apropiado**

  Puedes llamar al método del padre con super() si quieres conservar parte del comportamiento original.

```
class Gato(Animal):    def hacer_truco(self):        # Primero llama al método del padre        super().hacer_truco()        # Luego añade comportamiento propio        print("Pero lo hace a regañadientes")
```

Buenas prácticas con polimorfismo

Define una interfaz común en la clase base

Establece claramente qué métodos deben implementar las subclases para garantizar un comportamiento consistente.

```
class Animal:    def
hacer_sonido(self):        #
Método abstracto        raise
NotImplementedError
```

Usa NotImplementedError cuando sea necesario

Si no quieres proporcionar un comportamiento por defecto, usa esta excepción para forzar la implementación en las subclases.

Trabaja con colecciones de objetos

Aprovecha el polimorfismo para operar sobre listas, tuplas o diccionarios que contengan objetos de diferentes clases.

```
# Operando sobre diferentes
tiposfor animal in [perro,
gato, loro]:
animal.hacer_sonido()
```

Actividad práctica en Draw.io

Crear un diagrama con:

Clase base Instrumento con atributos: nombre, tipo y método tocar()

Subclases: Guitarra, Bateria, Piano

Método tocar() polimórfico en cada una

Cada instrumento debe implementar el método tocar() de forma diferente, reflejando cómo suena cada uno. Incluyan los atributos específicos de cada instrumento (por ejemplo, número de cuerdas para la guitarra).

Implementar el código en VS Code

```
class Instrumento:    def __init__(self, nombre):        self.nombre = nombre
def tocar(self):        raise NotImplementedErrorclass Guitarra(Instrumento):
def tocar(self):        return f"{self.nombre} produce acordes"class
Bateria(Instrumento):    def tocar(self):        return f"{self.nombre} produce
ritmos"class Piano(Instrumento):    def tocar(self):        return
f"{self.nombre} produce melodías"
```

```
# Creando instanciasguitarra = Guitarra("Gibson")bateria = Bateria("Pearl")piano
= Piano("Yamaha")# Lista de instrumentosinstrumentos = [guitarra, bateria,
piano]# Polimorfismo en acciónfor instrumento in instrumentos:
print(instrumento.tocar())
```

Revisión grupal del ejercicio

Comparar diagramas

Analizar las diferentes representaciones UML creadas por los grupos. Identificar fortalezas y áreas de mejora.

Revisar código

Examinar las implementaciones, verificando que el polimorfismo funcione correctamente. Destacar soluciones creativas.

Reforzar conceptos

Discutir cómo el comportamiento compartido facilita el trabajo con colecciones de objetos diferentes.

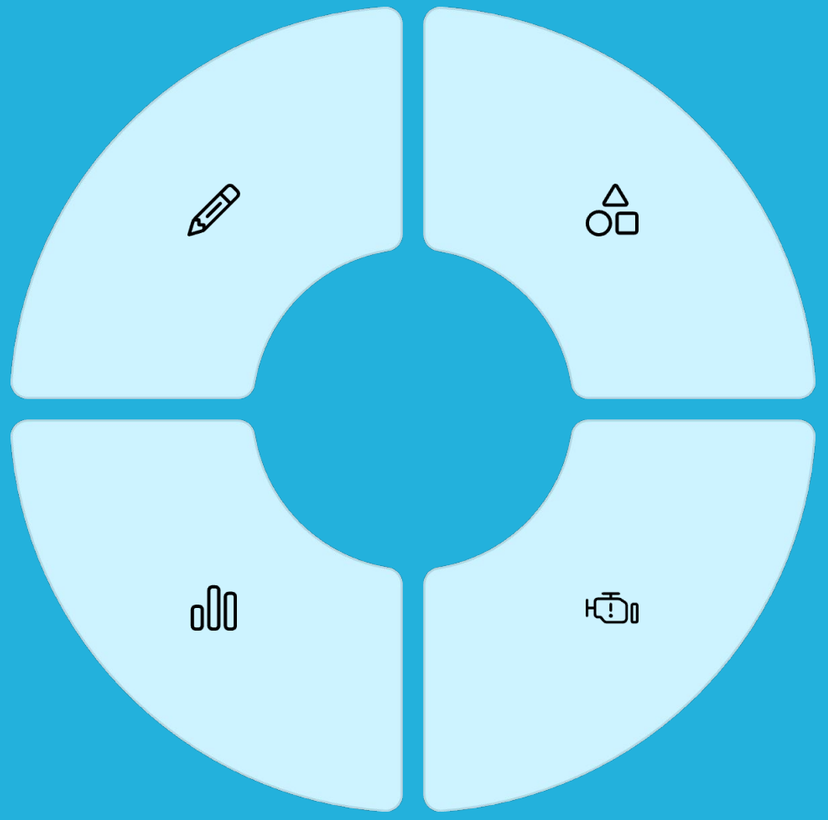
Evaluación formativa express

¿Qué es sobreescritura?

Redefinir un método de la clase padre en la clase hija, manteniendo el mismo nombre pero cambiando su implementación.

¿Cómo representar esto en UML?

Con diagramas de clases que muestren la herencia y los métodos sobreescritos, usando cursiva para métodos abstractos.



¿Qué es polimorfismo?

La capacidad de usar el mismo método en diferentes clases con implementaciones distintas, permitiendo tratar objetos de manera uniforme.

¿Para qué sirve raise NotImplementedError?

Para indicar que un método debe ser implementado en las subclases y no tiene implementación en la clase base.

Cierre y reflexión

Hoy aprendiste:

Sobreescritura = cambiar comportamiento heredado

Polimorfismo = mismos mensajes, respuestas distintas

Diagramas + código = aprendizaje completo

Estos conceptos son fundamentales en la programación orientada a objetos y te permitirán crear sistemas más flexibles y mantenibles.

> **¿Dónde usarías sobreescritura o polimorfismo en tu próximo proyecto?**