Universidad Católica de Temuco
Departamento de Informática

Informe: Herencia, Clases Abstractas, Polimorfismo e Interfaces en Programación Orientada a Objetos

Denys Rodríguez

7 de Octubre de 2025

Asignatura: Programación II
Profesor: Guido Mellado
Ciudad: Temuco, Chile

1. Introducción

La **Programación Orientada a Objetos (POO)** es un paradigma que busca modelar el mundo real dentro del software, a través de entidades llamadas **clases** y **objetos**. Una clase define las características y comportamientos que tendrán sus objetos, mientras que los objetos son instancias concretas de esas clases.

La POO promueve la reutilización de código, la abstracción, la modularidad y la escalabilidad. En este informe se abordarán los siguientes pilares fundamentales:

- **Herencia:** mecanismo para reutilizar y extender código existente.
- **Clases Abstractas:** estructuras base que definen comportamientos obligatorios.
- **Polimorfismo:** capacidad de diferentes objetos para responder de manera distinta a un mismo método.
- **Interfaces:** contratos que definen métodos sin implementación.
- **Method Resolution Order (MRO):** orden de búsqueda de métodos en jerarquías de herencia múltiple.

2. Herencia

La **herencia** es uno de los pilares fundamentales de la POO. Permite que una clase, denominada **clase hija** o **subclase**, adquiera los atributos y métodos de otra clase, llamada **clase padre** o **superclase**. Este mecanismo facilita la reutilización del código y la creación de jerarquías lógicas entre clases.

Gracias a la herencia, es posible extender o modificar el comportamiento de una clase sin alterar su implementación original, favoreciendo la mantenibilidad del software.

Ejemplo: Nómina

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

    def calcular_salario(self):
        # salario m nimo: solo base
        return self.sueldo_base

class Vendedor(Empleado):
    def __init__(self, nombre, sueldo_base, ventas_mes, comision=0.03):
        super().__init__(nombre, sueldo_base)
        self.ventas_mes = ventas_mes
        self.comision = comision

    def calcular_salario(self):
        # sobrescribe: base + comisi n por ventas
        return super().calcular_salario() + self.ventas_mes * self.
            comision
```

```
class Desarrollador(Empleado):
    def __init__(self, nombre, sueldo_base, horas_extra=0, tarifa_extra
        =8000):
        super().__init__(nombre, sueldo_base)
        self.horas_extra = horas_extra
        self.tarifa_extra = tarifa_extra

    def calcular_salario(self):
        return super().calcular_salario() + self.horas_extra * self.
            tarifa_extra

e1 = Vendedor("Ana", 900000, ventas_mes=12000000)  # + 3%
e2 = Desarrollador("Luis", 1200000, horas_extra=10)
print(e1.calcular_salario())  # 900000 + 360000
print(e2.calcular_salario())  # 1200000 + 80000
```

Análisis

- **Jerarquía:** `Vendedor` y `Desarrollador` heredan de `Empleado`. Comparten el contrato `calcular_salario()`, pero cada uno lo *especializa*.
- **Reutilización con `super()`:** ambas subclases invocan `super().calcular_salario()` para reutilizar el salario base y *sumar* su propia lógica (comisión u horas extra).
- **Encapsulamiento:** los datos relevantes (ventas, comisión, horas, tarifa) se guardan como atributos de instancia, aislando responsabilidades por rol.
- **Extensibilidad:** agregar otra subclase (p. ej., `Gerente` con bono) no rompe el diseño; sólo sobreescribe `calcular_salario()`.
- **Prueba de comportamiento:** `e1` y `e2` demuestran resultados diferentes con el mismo mensaje `calcular_salario()`, lo que ya anticipa polimorfismo por herencia.

3. Clases Abstractas

Las **clases abstractas** son clases que no pueden ser instanciadas directamente. Su propósito es servir como **plantillas** para otras clases, definiendo métodos que deben ser implementados obligatoriamente por sus subclases.

En Python, se definen mediante el módulo `abc` (Abstract Base Classes). Estas clases establecen un marco común para un grupo de clases relacionadas.

Ejemplo: Pasarela de pagos

```
from abc import ABC, abstractmethod

class ProcesadorPago(ABC):
    @abstractmethod
    def validar(self, monto): ...
    @abstractmethod
    def capturar(self, monto): ...
    @abstractmethod
    def reembolsar(self, monto): ...
```

```python
class PagoTarjeta(ProcesadorPago):
    def validar(self, monto):
        if monto <= 0: raise ValueError("Monto inv lido")
        print("[Tarjeta] Validaci n antifraude OK")
    def capturar(self, monto):
        print(f"[Tarjeta] Captura por ${monto:,.0f}")
    def reembolsar(self, monto):
        print(f"[Tarjeta] Reembolso por ${monto:,.0f}")

class PagoCrypto(ProcesadorPago):
    def validar(self, monto):
        if monto < 1000: raise ValueError("M nimo en crypto: 1000")
        print("[Crypto] Validaci n de saldo OK")
    def capturar(self, monto):
        print(f"[Crypto] TX on-chain por ${monto:,.0f}")
    def reembolsar(self, monto):
        print(f"[Crypto] TX de devoluci n por ${monto:,.0f}")

def checkout(procesador: ProcesadorPago, monto):
    procesador.validar(monto)
    procesador.capturar(monto)

checkout(PagoTarjeta(), 150000)
checkout(PagoCrypto(), 200000)
```

Análisis

- **Contrato obligatorio:** `ProcesadorPago` (ABC) define los métodos `validar`, `capturar` y `reembolsar`. No se puede instanciar y obliga a que las subclases implementen el contrato.
- **Sustitución segura:** `PagoTarjeta` y `PagoCrypto` se pueden usar indistintamente en `checkout()`, cumpliendo el Principio de Sustitución de Liskov.
- **Variación por tipo:** cada procesador aplica reglas distintas (mínimos, antifraude, TX on-chain) sin cambiar el *uso* por parte del cliente.
- **Escalabilidad:** incorporar `PagoTransferencia` o `PagoWallet` sólo requiere implementar el contrato; el resto del sistema permanece estable.
- **Manejo de errores:** las validaciones tempranas (`ValueError`) expresan reglas de dominio y previenen estados inválidos.

Beneficios:

- Obligan a mantener una estructura uniforme en las subclases.
- Facilitan el diseño modular y la escalabilidad del sistema.
- Separan la definición de la implementación.

4. Polimorfismo

El **polimorfismo** significa "muchas formas". En la POO, este principio permite que un mismo método tenga diferentes comportamientos según el tipo de objeto que lo invoque.

Esto fomenta la flexibilidad, ya que se puede invocar el mismo método en diferentes clases sin conocer su implementación exacta.

Ejemplo: Exportadores

```
class Exportador:
    def exportar(self, datos): raise NotImplementedError

class ExportarPDF(Exportador):
    def exportar(self, datos):
        return f"[PDF] {len(datos)} registros"

class ExportarCSV(Exportador):
    def exportar(self, datos):
        return "col1,col2\n" + "\n".join(f"{x[0]},{x[1]}" for x in datos
            )

class ExportarJSON(Exportador):
    def exportar(self, datos):
        import json
        return json.dumps({"registros": datos}, ensure_ascii=False)

datos = [(1,"ok"), (2,"fail")]
for exp in (ExportarPDF(), ExportarCSV(), ExportarJSON()):
    print(exp.exportar(datos))
```

Análisis

- **Mismo mensaje, respuestas distintas:** `exportar(datos)` produce salidas diferentes según la clase concreta (PDF/CSV/JSON).
- **Desacoplamiento:** el código cliente itera por `Exportador` sin conocer implementaciones. Permite inyectar nuevos formatos sin editar el bucle.
- **Single Responsibility:** cada clase se concentra en un *formato*. Evita *if/else* por tipo y facilita pruebas unitarias.
- **Extensión sencilla:** añadir `ExportarXML` o `ExportarXLSX` no afecta a `Exportador` ni al cliente.

5. Interfaces

Una **interfaz** es una estructura que define un conjunto de métodos que una clase debe implementar, sin especificar cómo hacerlo. Las interfaces permiten que diferentes clases compartan un mismo conjunto de métodos, asegurando consistencia en el comportamiento.

En Python, se pueden simular mediante clases abstractas con métodos abstractos.

Ejemplo: Notificador

```
from abc import ABC, abstractmethod

class Notificador(ABC):
    @abstractmethod
    def enviar(self, para, mensaje): ...

class EmailNotifier(Notificador):
    def enviar(self, para, mensaje):
        print(f"[Email] A: {para} | {mensaje}")

class SMSNotifier(Notificador):
    def enviar(self, para, mensaje):
        print(f"[SMS] A: {para} | {mensaje}")

class PushNotifier(Notificador):
    def enviar(self, para, mensaje):
        print(f"[Push] A: {para} | {mensaje}")

def alertar(notificador: Notificador, usuarios, mensaje):
    for u in usuarios:
        notificador.enviar(u, mensaje)

alertar(SMSNotifier(), ["+56911111111","+56922222222"], "C digo 2FA:
   839211")
```

Análisis

- **Contrato explícito:** `Notificador` define `enviar(para, mensaje)`. Cualquier canal que lo implemente es compatible con `alertar()`.
- **Inversión de dependencias:** `alertar()` depende de la *abstracción* `Notificador`, no de concretos. Permite cambiar el canal sin tocar la lógica de alertas.
- **Polimorfismo por interfaz:** Email/SMS/Push cumplen el mismo método con detalles distintos (formato, destino).
- **Fácil de testear:** se puede inyectar un `FakeNotifier` para pruebas sin enviar mensajes reales.

Beneficios de las interfaces:

- Aumentan la interoperabilidad entre componentes.
- Favorecen la programación orientada a contratos.
- Facilitan el trabajo en equipos de desarrollo, ya que definen reglas claras.

6. Method Resolution Order (MRO)

El **Method Resolution Order (MRO)** es el mecanismo interno que utiliza Python para determinar el orden en el que se buscan los métodos y atributos dentro de una

jerarquía de clases, especialmente cuando existe **herencia múltiple**. Su objetivo es resolver de manera ordenada y coherente qué método debe ejecutarse cuando varias clases antecesoras definen el mismo nombre de método o atributo.

Importancia del MRO

El MRO evita ambigüedades en la herencia múltiple. Por ejemplo, si una clase hereda de dos clases que a su vez comparten un ancestro común, Python necesita un criterio lógico para decidir qué método ejecutar sin causar conflictos. Sin este orden, el programa podría volverse impredecible o lanzar errores por referencias circulares.

Python utiliza un algoritmo llamado **C3 Linearization** para calcular este orden. Este algoritmo garantiza:

- Que el orden de búsqueda sea **consistente** y **predecible**.
- Que las clases hijas siempre tengan prioridad sobre las clases padres.
- Que se mantenga el orden de declaración de las clases en la definición de la herencia.

Ejemplo: Mixins y MRO cooperativo

```
class BaseService:
    def procesar(self, data):
        # capa base
        return {"data": data, "pasos": ["base"]}

class AuthMixin(BaseService):
    def procesar(self, data):
        res = super().procesar(data)
        res["pasos"].append("auth")
        return res

class LoggerMixin(BaseService):
    def procesar(self, data):
        res = super().procesar(data)
        res["pasos"].append("log")
        return res

class CacheMixin(BaseService):
    def procesar(self, data):
        res = super().procesar(data)
        res["pasos"].append("cache")
        return res

# Orden heredado: izquierda -> derecha define la prioridad en MRO
class Servicio(CacheMixin, LoggerMixin, AuthMixin, BaseService):
    pass

s = Servicio()
print(Servicio.mro())  # observa el orden linealizado
print(s.procesar({"id": 7}))
```

Análisis

- **Herencia múltiple cooperativa:** cada mixin añade una "capa" al método `procesar()` y llama a `super()` para encadenar la ejecución.
- **Orden definido por MRO:** con `class Servicio(CacheMixin, LoggerMixin, AuthMixin, BaseService)`, el MRO típico será:
  `Servicio → CacheMixin → LoggerMixin → AuthMixin → BaseService → object`.
- **Efecto observable:** la lista `pasos` muestra el orden real de invocación (`["base",çache","log",.`
  Cambiar el orden de herencia cambia el orden de pasos.
- **Reglas C3 respetadas:** el algoritmo garantiza prioridad de la subclase, preserva el orden de las bases y evita diamantes ambiguos.
- **Buenas prácticas:** todos los mixins deben llamar a `super()` y devolver el resultado para mantener la cadena cooperativa.

Fórmula general del algoritmo C3

El MRO se calcula siguiendo la fórmula:

$$L(C) = C + merge(L(P_1), L(P_2), ..., P_1, P_2, ...)$$

Donde:

- $L(C)$ representa la linealización (orden) de la clase C.
- $P_1, P_2, ...$ son las superclases directas de C.
- El operador *merge* combina las listas de manera que se respete el orden de precedencia sin duplicar clases.

Comparación con otros lenguajes

En lenguajes como Java o C#, la herencia múltiple directa de clases no está permitida precisamente por la complejidad de la resolución de métodos. Python, en cambio, la admite gracias a este algoritmo MRO, que garantiza un comportamiento estable y bien definido.

Conclusión de la sección

El **MRO** es fundamental para entender cómo Python maneja la herencia múltiple sin ambigüedades. Conocer su funcionamiento permite escribir jerarquías de clases complejas de manera segura y entender el flujo de ejecución de los métodos dentro de un programa orientado a objetos.

7. Conclusión

Los conceptos analizados en este informe constituyen los pilares esenciales de la Programación Orientada a Objetos. La **herencia** fomenta la reutilización, las **clases abstractas**

y las **interfaces** aportan estructura, y el **polimorfismo** otorga flexibilidad. Finalmente, el **MRO** asegura que Python maneje correctamente la herencia múltiple sin ambigüedades.

Dominar estos conceptos permite construir sistemas más organizados, eficientes y fáciles de mantener, aplicables tanto en proyectos pequeños como en grandes arquitecturas de software.