Programación 2

Objetos y Clases. Implementación. Verificación. Cambios en el diseño Diagrama de objetos. Identidad, igualdad y equivalencia.

En esta clase

- Repaso
- Caso de Estudio: Cuenta Bancaria
- El diseño de una clase
- La implementación en Python
- La verificación de una clase
- Objetos, variables y referencias
- Mensajes y métodos
- Parámetros y resultados
- Cambios en el diseño
- Cambios en la implementación
- Identidad, igualdad y equivalencia

Repaso - Aclaraciones parámetros opcionales

Para declarar parámetros opcionales simplemente le asignamos al parámetro un valor por defecto.

Reglas para declarar parámetros opcionales:

- Deben declararse después de los parámetros obligatorios. Si no genera un error de sintaxis.
- Documentar claramente los parámetros opcionales y sus valores predeterminados en la cadena de documentación (docstring) de la función.
- Los valores predeterminados deben ser del tipo correcto y el valor predeterminado no debe ser mutable (como listas o diccionarios). Esto se debe a que en Python evalúa los valores predeterminados de los parámetros sólo una vez al definir la función, no cada vez que se la llama.

  Ej:

```
def procedimiento(param1, param2=[]):  # Mala práctica
    pass

def procedimiento(param1, param2=None):  # Mejor práctica
    if param2 is None:
        param2 = []
    pass
```

Parámetros mutables – ej. Mala práctica

```python
class Personaje:
    def __init__(self, nombre:str, habilidades:list=[]):
        self.__nombre = nombre
        self.__habilidades = habilidades

    def agregarHabilidad(self, habilidad:str):
        self.__habilidades.append(habilidad)

    def __str__(self):
        return f"{self.__nombre}, habilidades: {self.__habilidades}"
 #ejecutamos pruebas:
p1 = Personaje("Jugador 1")
p2 = Personaje("Jugador 2")
print(p1)
print(p2)

p1.agregarHabilidad("invisible")
p1.agregarHabilidad("inteligencia")
p2.agregarHabilidad("resistencia")
p2.agregarHabilidad("fuerza")
 print(p1)
print(p2)
```

Salida:

Jugador 1, habilidades: []

Jugador 2, habilidades: []

Jugador 1, habilidades: ['invisible', 'inteligencia', 'resistencia', 'fuerza’] ✖

Jugador 2, habilidades: ['invisible', 'inteligencia', 'resistencia', 'fuerza'] ✖

Parámetros mutables – ej. Mala práctica

¿Por qué tenemos esa salida?

La primera vez que se llama al constructor, se crea una lista vacía y se asigna al parámetro **habilidades**.

Cuando se crea el segundo objeto se reutiliza la misma lista vacía, ya que no se ha proporcionado un valor diferente para el parámetro **habilidades**.

Cuando los distintos objetos agregan habilidades a su lista, la modificación se refleja en la lista original, ya que **ambas variables apuntan al mismo objeto en memoria**.

Se soluciona evitando el uso de objetos mutables como parámetros opcionales.

Parámetros mutables – ej. Buena práctica

```
class Personaje:
    def __init__(self, nombre:str, habilidades:list=None):
        self.__nombre = nombre
        if habilidades != None:
            self.__habilidades = habilidades
        else:
            self.__habilidades = []

    def agregarHabilidad(self, habilidad:str):
        self.__habilidades.append(habilidad)

    def __str__(self):
        return f"{self.__nombre}, habilidades: {self.__habilidades}"
 #ejecutamos las mismas pruebas:
p1 = Personaje("Jugador 1")
print(p1)
p2 = Personaje("Jugador 2")
print(p2)
p1.agregarHabilidad("invisible")
p1.agregarHabilidad("inteligencia")
p2.agregarHabilidad("resistencia")
p2.agregarHabilidad("fuerza")
 print(p1)
print(p2)
```

Salida:

Jugador 1, habilidades: []

Jugador 2, habilidades: []

Jugador 1, habilidades: ['invisible', 'inteligencia']

Jugador 2, habilidades: ['resistencia', 'fuerza']

Caso de estudio: cuenta bancaria

Un banco ofrece **cuentas corrientes** a sus clientes.

*Los clientes pueden realizar **depósitos**, **extracciones** y **consultar el saldo** de su cuenta corriente.*

En el momento que se crea una cuenta corriente se establece su código y el saldo se inicializa en 0.

También es posible crear una cuenta corriente estableciendo su código y saldo inicial.

El código no se modifica, el saldo cambia con cada depósito o extracción.

Una cuenta bancaria puede tener un saldo negativo hasta un máximo establecido por el banco.

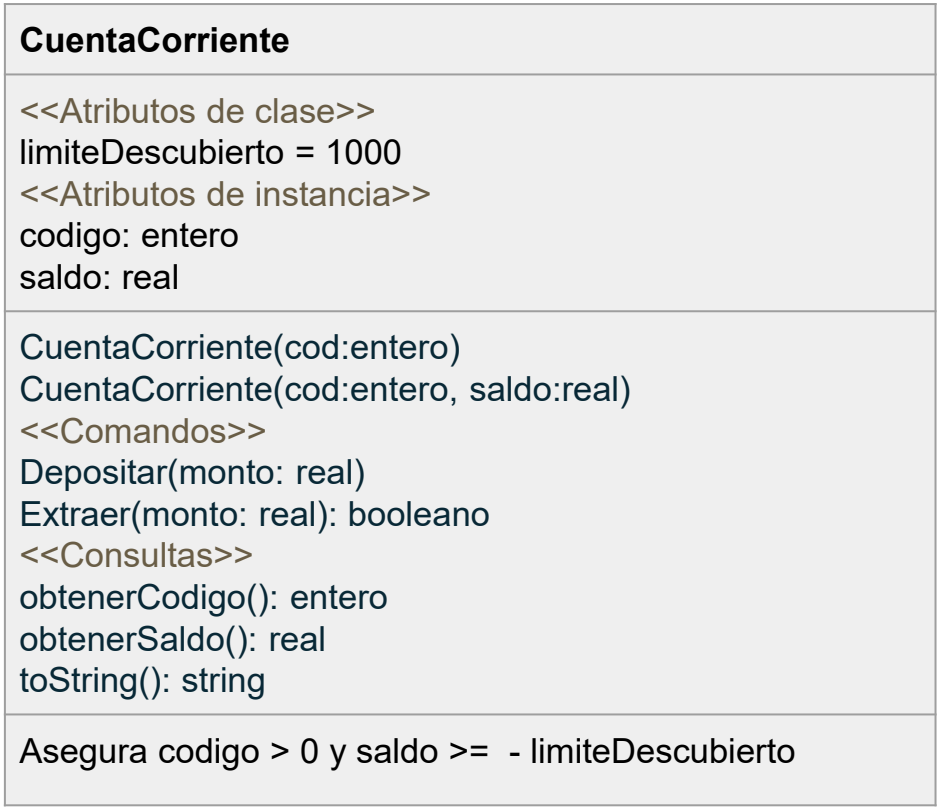
Caso de estudio: cuenta bancaria - Diagrama

CuentaCorriente(cod:entero)
requiere cod > 0

CuentaCorriente(cod:entero, saldo:real)
requiere cod > 0 y saldo >= 0

Depositar(monto: real)
requiere monto > 0

Extraer(monto: real): boolean
requiere monto>0
Si monto > saldo+limiteDescubierto
extraer retorna false y la extracción no se realiza



Aclaración

En los próximos ejemplos, el código de los métodos estará sin validaciones de tipo y rango para facilitar la comprensión del concepto a explicar.

Recordemos que **las clases deben validar los datos que reciben antes de realizar operaciones con ellos**.

Cuando validas solo en el front

Implementación en python

```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        """
        Inicializa una nueva cuenta corriente.
        Parámetros:
        - codigo: El código único de la cuenta.
        - saldo: El saldo inicial de la cuenta (default: 0.0).
        """
        self.__codigo = codigo
        self.__saldo = saldo
```

Las variables **codigo** y **saldo** son los atributos de instancia de la clase y pueden ser usados en cualquiera de los servicios provistos por la clase CuentaCorriente.

Implementación en python

```
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        """
        Inicializa una nueva cuenta corriente.
        Parámetros:
        - codigo: El código único de la cuenta.
        - saldo: El saldo inicial de la cuenta (default: 0.0).
        """
        self.__codigo = codigo
        self.__saldo = saldo
```

Como se declaran **privados**, sus valores sólo pueden ser accedidos desde el exterior por los servicios públicos que brinda la clase.

Implementación en python

```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        """
        Inicializa una nueva cuenta corriente.
        Parámetros:
        - codigo: El código único de la cuenta.
        - saldo: El saldo inicial de la cuenta (default: 0.0).
        """
        self.__codigo = codigo
        self.__saldo = saldo
```

La variable **LIMITE_DESCUBIERTO** es un atributo de clase, todos los objetos de la clase comparten un mismo valor.

Implementación en python

```
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000

    #atributos de instancia
    def __init__(self, codigo: int, saldo: float = 0.0):
        """
        Inicializa una nueva cuenta corriente.
        Parámetros:
        - codigo: El código único de la cuenta.
        - saldo: El saldo inicial de la cuenta (default: 0.0).
        """
        self.__codigo = codigo
        self.__saldo = saldo
```

La clase **CuentaCorriente** no lee ni muestra datos, toda la entrada y salida la hacen las clases que usan a **CuentaCorriente**.

Implementación en python

```
def depositar(self, monto: float):
        """
        Deposita una cantidad en la cuenta corriente. Requiere monto > 0.
        Parametros:
        - monto: La cantidad a depositar.
        """
        self.__saldo += monto
```

El comando **depositar** modifica el valor del **atributo de instancia** saldo.
Según las especificaciones de esta clase, es responsabilidad de la clase que usa a **CuentaCorriente** asegurar que monto > 0

Implementación en python

```
    def extraer(self, monto: float)->bool:
            """
            Extrae una cantidad de la cuenta corriente.
            Parametros:
            - monto: La cantidad a extraer.
            Retorna:
            - True si la extracción fue exitosa, False en caso contrario.
            """
            puedeExtraer = False
            if self.__saldo + CuentaCorriente.__LIMITE_DESCUBIERTO >= monto:
                self.__saldo -= monto
                puedeExtraer= True

            return puedeExtraer
```

La variable local **puedeExtraer** se crea cuando se inicia la ejecución del método y solo puede ser accedida en ese **bloque** de código.

Implementación en python

```
    def extraer(self, monto: float)->bool:
            """
            Extrae una cantidad de la cuenta corriente.
            Parametros:
            - monto: La cantidad a extraer.
            Retorna:
            - True si la extracción fue exitosa, False en caso contrario.
            """
            puedeExtraer = False
            if self.__saldo + CuentaCorriente.__LIMITE_DESCUBIERTO >= monto:
                self.__saldo -= monto
                puedeExtraer= True

            return puedeExtraer
```

La variable **monto** es un parámetro **formal**.
Cuando se inicia la ejecución del método se crea una nueva variable y se inicializa con el valor del **parámetro real (o efectivo)**.

Implementación en python

```
    def extraer(self, monto: float)->bool:
            """
            Extrae una cantidad de la cuenta corriente.
            Parametros:
            - monto: La cantidad a extraer.
            Retorna:
            - True si la extracción fue exitosa, False en caso contrario.
            """
            puedeExtraer = False
            if self.__saldo + CuentaCorriente.__LIMITE_DESCUBIERTO >= monto:
                self.__saldo -= monto
                puedeExtraer= True

            return puedeExtraer
```

Al terminar la ejecución de extraer las variables **puedeExtraer** y **monto** se destruyen.

Implementación en python

```
def obtenerSaldo(self)->float:
    """Devuelve el saldo de la cuenta corriente."""
    return self.__saldo

def obtenerCodigo(self)->int:
    """Devuelve el código de la cuenta corriente."""
    return self.__codigo
```

En cada consulta el tipo de datos que definimos como retorno del método es compatible con el tipo del resultado que retorna.

Implementación en python

```
def __str__(self):
    return f"Cuenta Corriente Código {self.__codigo} - Saldo: ${self.__saldo}"
```

En python definimos el método especial **__str__()** para establecer cómo se debe representar una instancia de la clase como una cadena de texto legible por humanos.

El método __str__ debe devolver una cadena que represente el estado del objeto de una manera significativa para el usuario.

Observación: *En otros lenguajes de programación encontrarán el método toString().*
El nombre toString() es estándar para referirse a una consulta que retorna una cadena de caracteres cuyo valor es la concatenación de los valores de los atributos del objeto que recibe el mensaje.

La clase tester

La **clase tester** verifica que la clase cumple con sus **responsabilidades** y los servicios se comportan de acuerdo a la **funcionalidad** y las **restricciones** especificadas, para un conjunto de **casos de prueba**.

Los casos de prueba pueden ser:

- Fijos.
- Leídos de un archivo.
- Ingresados por el usuario por consola o a través de una interfaz gráfica.
- Generados al azar.

Implementación en python

```python
class TestSaldos:
    @staticmethod
    def test():
        cuenta_1 = CuentaCorriente(1, 1000)
        cuenta_2 = CuentaCorriente(2)
        cuenta_1.depositar(100)
        cuenta_2.depositar(100)
        print(cuenta_1) # llamada a __str__
        print(cuenta_2) # llamada a __str__
        cuenta_1.extraer(500)
        cuenta_2.extraer(900)
        print(cuenta_1)
        print(cuenta_2)
            ...

if __name__ == "__main__":
    TestSaldos.test()
```

Salida:

```
Cuenta Corriente Código 1 - Saldo: $1100.00
Cuenta Corriente Código 2 - Saldo: $100.00

[ … ]

Cuenta Corriente Código 1 - Saldo: $600.00
Cuenta Corriente Código 2 - Saldo: $-800.00
...
...
...
```

if __name__ == "__main__": iniciará la ejecución del método test() de la clase TestSaldos en caso que el archivo de python sea ejecutado directamente y no cuando se importa como módulo.

Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        ...
```

Crea un objeto de software ligado a la variable **cuenta** de clase **CuentaCorriente**.

Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        cuenta.depositar(100)
        ...
```

Envía el **mensaje depositar** al objeto ligado a la variable **cuenta**.
El objeto ejecuta el **método depositar** y el saldo se incrementa en **100**.

Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        cuenta.depositar(100)
        if not cuenta.extraer(500):
            print("No se pudo extraer 500 de la cuenta.")

        ...
```

Envía el **mensaje extraer** al objeto ligado a la variable **cuenta** con parámetro **500**.
El objeto ejecuta el **método extraer** y retorna un valor booleano.

Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        cuenta.depositar(100)
        if not cuenta.extraer(500):
            print("No se pudo extraer 500 de la cuenta.")
        if not cuenta.extraer(1500):
            print("No se pudo extraer 1500 de la cuenta.")

        ...
```

Envía el **mensaje extraer** al objeto ligado a la variable **cuenta** con parámetro **1500**.
El objeto ejecuta el **método extraer** y retorna un valor booleano.

Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        cuenta.depositar(100)
        if not cuenta.extraer(500):
            print("No se pudo extraer 500 de la cuenta.")
        if not cuenta.extraer(1500):
            print("No se pudo extraer 1500 de la cuenta.")
        print(cuenta)
        ...
```

Envía el **mensaje __str__()** al objeto ligado a la variable **cuenta**.
Muestra en consola la cadena de caracteres que retorna luego de que el objeto ejecuta el método **__str__** de la clase **CuentaCorriente**.

Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        cuenta.depositar(100)
        if not cuenta.extraer(500):
            print("No se pudo extraer 500 de la cuenta.")
        if not cuenta.extraer(1500):
            print("No se pudo extraer 1500 de la cuenta.")
        print(cuenta)
        ...
```

No produce salida por pantalla (condición evaluada como false)

Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        cuenta.depositar(100)
        if not cuenta.extraer(500):
            print("No se pudo extraer 500 de la cuenta.")
        if not cuenta.extraer(1500):
            print("No se pudo extraer 1500 de la cuenta.")
        print(cuenta)
        ...
```

Condición evaluada como true, salida:

```
No se pudo extraer 1500 de la cuenta.
```

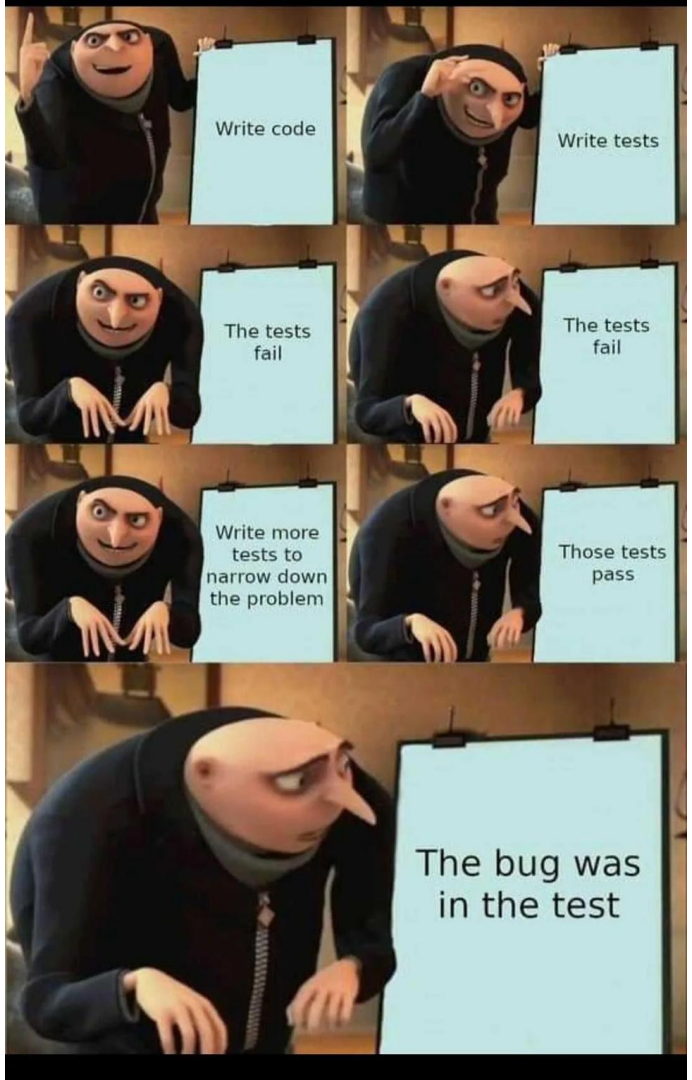
Objetos, mensaje y métodos

```
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta = CuentaCorriente(1, 700)
        cuenta.depositar(100)
        if not cuenta.extraer(500):
            print("No se pudo extraer 500 de la cuenta.")
        if not cuenta.extraer(1500):
            print("No se pudo extraer 1500 de la cuenta.")
        print(cuenta)
        ...
```

Salida:

```
No se pudo extraer 1500 de la cuenta.
Cuenta Corriente Código 1 - Saldo: $300.00
```


Write code
Write tests
The tests fail
The tests fail
Write more tests to narrow down the problem
Those tests pass
The bug was in the test

Variables, objetos y referencias

La instrucción:

```
cuenta = CuentaCorriente(1, 600)
```

- **Declara** la variable cuenta.
- **Crea** un objeto de clase CuentaCorriente.
- **Liga** el objeto a la variable.

La creación de un objeto provoca:

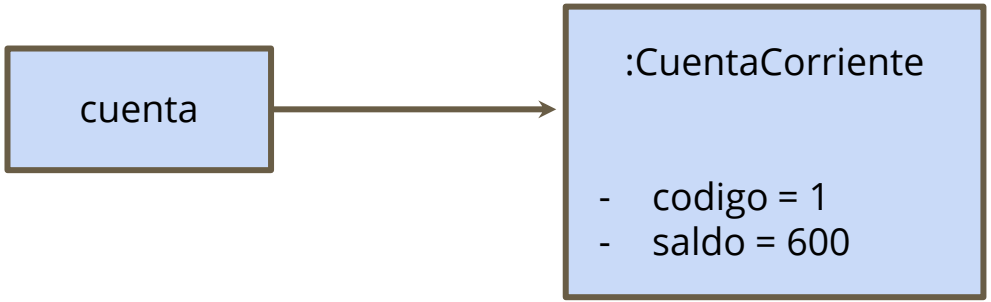
- **Reservar** espacio en memoria para almacenar el estado interno del objeto.
- **Ejecutar** constructor.

Variables, objetos y referencias

La instrucción:

```
cuenta = CuentaCorriente(1, 600)
```

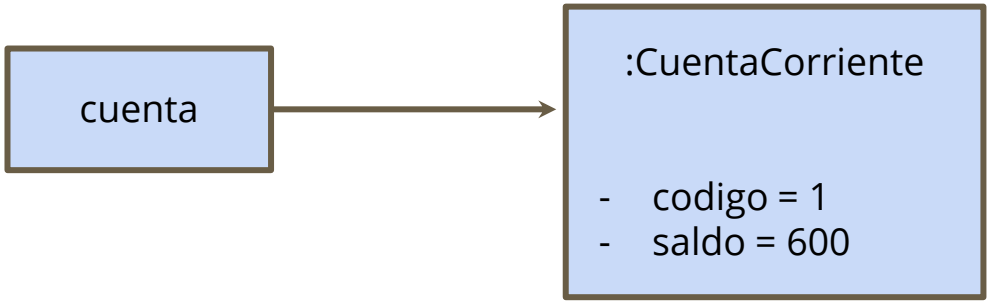
Se puede graficar a través de un **diagrama de objetos**:



Variables, objetos y referencias

El valor de la **variable** cuenta es una **referencia** a un **objeto** de clase CuentaCorriente.

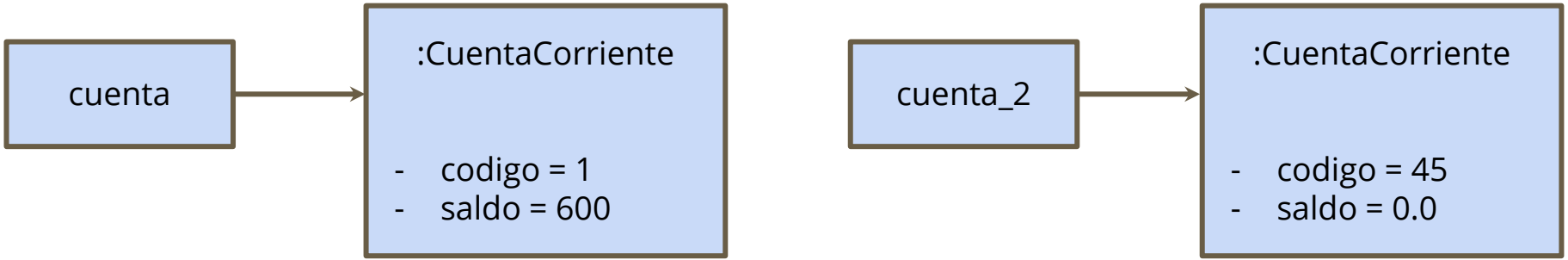
El **estado interno** del objeto almacena los valores de dos variables que corresponden a los atributos de instancia del objeto, determinados por su clase.



Variables, objetos y referencias

La estructura del **estado interno** de los objetos de clase CuentaCorriente está conformada por las variables **código** y **saldo**.

```
cuenta = CuentaCorriente(1, 600)
cuenta_2 = CuentaCorriente(45)
```



Alternativas de diseño

Cambiar la signatura del método, la clase y las responsabilidades impacta en la clase cliente. El cliente debe adaptarse a estos cambios y cambiar parte de su código.

CuentaCorriente(cod:entero)
requiere cod > 0

CuentaCorriente(cod:entero, saldo:real)
requiere cod > 0 y saldo >= 0

Depositar(monto: real)
requiere monto > 0

Extraer(monto: real)
requiere monto>0 y
puedeExtraer(monto) == true

CuentaCorriente

<<Atributos de clase>>
limiteDescubierto = 1000
<<Atributos de instancia>>
codigo: entero
saldo: real

CuentaCorriente(cod:entero)
CuentaCorriente(cod:entero, saldo:real)
<<Comandos>>
Depositar(monto: real)
Extraer(monto: real)
<<Consultas>>
puedeExtraer(monto: real): booleano
obtenerCodigo(): entero
obtenerSaldo(): real

Asegura codigo > 0 y saldo >= - limiteDescubierto

Alternativas de diseño

```python
def extraer(self, monto: float):
    """
    Extrae una cantidad de la cuenta corriente.
    Requiere monto > 0 y que el saldo + limite descubierto sea mayor o igual al monto.

    Parametros:
    - monto: La cantidad a extraer.
    """
    if self.puedeExtraer(monto):
        self.__saldo -= monto

def puedeExtraer(self, monto: float)->bool:
    """Requiere monto > 0.
    Retorna True si el saldo + limite descubierto es mayor o igual al monto."""
    return self.__saldo + CuentaCorriente.__LIMITE_DESCUBIERTO >= monto
```

Alcance de las variables

```python
class CuentaCorriente:
    #atributos de clase
    __LIMITE_DESCUBIERTO = 1000
            ...
    def depositar(self, monto: float):
        """
        Deposita una cantidad en la cuenta corriente.
        Requiere que el monto sea mayor a 0.
        Parametros:
        - monto: La cantidad a depositar.
        """
        self.__saldo += monto
```

```python
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta_1 = CuentaCorriente(1, 1000)
        cuenta_2 = CuentaCorriente(2)
        cuenta_1.depositar(100)
        cuenta_2.depositar(100)
        print(f"Saldo cuenta 1: {cuenta_1.obtenerSaldo()}") # 1100
        print(f"Saldo cuenta 2: {cuenta_2.obtenerSaldo()}") # 100
```

- *depositar()* puede acceder a las variables monto, __saldo, __codigo y __LIMITE_DESCUBIERTO
- *test()* puede acceder a las variables cuenta_1 y cuenta_2

Alternativas de diseño

```python
from CuentaCorriente import CuentaCorriente

class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta_1 = CuentaCorriente(1, 1000)
        cuenta_2 = CuentaCorriente(2)
        cuenta_1.depositar(100)
        cuenta_2.depositar(100)
        print(f"Saldo cuenta 1: {cuenta_1.obtenerSaldo()}") # 1100
        print(f"Saldo cuenta 2: {cuenta_2.obtenerSaldo()}") # 100
        if  cuenta_1.puedeExtraer(500):
            cuenta_1.extraer(500)
            print(f"Extracción exitosa. Saldo cuenta 1: {cuenta_1.obtenerSaldo()}") #600
        else:
            print(f"No se pudo extraer 500 de cuenta_1. Saldo cuenta 1: {cuenta_1.obtenerSaldo()}")
        if cuenta_2.puedeExtraer(900):
            cuenta_2.extraer(900)
            print(f"Extracción exitosa. Saldo cuenta 2: {cuenta_2.obtenerSaldo()}") #-800
        else:
            print(f"No se pudo extraer 900 de cuenta_2. Saldo cuenta 2: {cuenta_2.obtenerSaldo()}")
```

En este diseño cada clase que usa a CuentaCorriente asume la responsabilidad de controlar que sea posible extraer el monto, antes de enviar el mensaje extraer.

Cambios en el diseño

Un banco ofrece **cuentas corrientes** a sus clientes.

Los clientes pueden realizar **depósitos**, **extracciones** y **consultar el saldo** de su cuenta corriente.

En el momento que se crea una cuenta corriente se establece su código y el saldo se inicializa en 0. También es posible crear una cuenta corriente estableciendo su código y saldo inicial.

El código no se modifica, el saldo cambia con cada depósito o extracción.

Una cuenta bancaria puede tener un saldo negativo hasta un máximo establecido por el banco.

La clase brinda servicios para determinar el código de la cuenta con mayor saldo entre dos cuentas y cuál es la cuenta con mayor saldo, entre dos cuentas.

Cambios en el diseño

CuentaCorriente(cod:entero)
requiere cod > 0

CuentaCorriente(cod:entero, saldo:real)
requiere cod > 0 y saldo >= 0

Depositar(monto: real)
requiere monto > 0

Extraer(monto: real): boolean
requiere monto>0
Si monto > saldo+limiteDescubierto
extraer retorna false y la extracción no se realiza

CuentaCorriente
<<Atributos de clase>> limiteDescubierto = 1000 <<Atributos de instancia>> codigo: entero saldo: real
CuentaCorriente(cod:entero) CuentaCorriente(cod:entero, saldo:real) <<Comandos>> Depositar(monto: real) Extraer(monto: real): booleano <<Consultas>> obtenerCodigo(): entero obtenerSaldo(): real toString(): string **codCuentaMayorSaldo(otraCuenta: CuentaCorriente): entero** **cuentaMayorSaldo(otraCuenta: CuentaCorriente): CuentaCorriente**
Asegura codigo > 0 y saldo >= - limiteDescubierto

Cambios en la implementación

```
def codMayorSaldo(self, otraCuenta:"CuentaCorriente")->int:
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve el código de la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self.__codigo
    else:
        return otraCuenta.obtenerCodigo()
```

El método **codMayorSaldo** recibe como parámetro un objeto de la clase **CuentaCorriente**.

Compara el saldo del objeto que recibe el mensaje con el saldo del objeto recibido como parámetro.
Retorna el código de la cuenta con mayor saldo.

Cambios en la implementación

```
def codMayorSaldo(self, otraCuenta:"CuentaCorriente")->int:
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve el código de la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self.__codigo
    else:
        return otraCuenta.obtenerCodigo()
```

Tanto si la expresión lógica computa **true** o **false**, el resultado es un valor **entero**.
El diseñador no indicó qué retorna si las cuentas tienen el mismo saldo, la decisión la tomó el programador.

Cambios en la implementación

```
def codMayorSaldo(self, otraCuenta:"CuentaCorriente")->int:
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve el código de la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self.__codigo
    else:
        return otraCuenta.obtenerCodigo()
```

El método tiene **dos puntos de salida**. Aún así, no se compromete la legibilidad del código.

Cambios en la implementación

```
def codMayorSaldo(self, otraCuenta:"CuentaCorriente")->int:
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve el código de la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self.__codigo
    else:
        return otraCuenta.obtenerCodigo()
```

El método solo accede directamente a los atributos de instancia del objeto que recibe el mensaje.
Los atributos del objeto ligado a la variable *otraCuenta* son accedidos a través de los servicios provistos por su clase.

Cambios en la implementación

```
def cuentaMayorSaldo(self, otraCuenta:"CuentaCorriente")->"CuentaCorriente":
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self
    else:
        return otraCuenta
```

El resultado es un **objeto** de clase **CuentaCorriente**, o mejor dicho, devuelve una **referencia a un objeto de clase CuentaCorriente**.

Cambios en la implementación

```
def cuentaMayorSaldo(self, otraCuenta:"CuentaCorriente")->"CuentaCorriente":
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self
    else:
        return otraCuenta
```

cuentaMayorSaldo retorna una **referencia** a un objeto de clase **CuentaCorriente**.

La palabra reservada ***self*** permite nombrar al objeto que recibe el mensaje.

Cambios en la implementación

```
class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta_1 = CuentaCorriente(1, 1000)
        cuenta_2 = CuentaCorriente(2)
        cuenta_1.depositar(100)
        cuenta_2.depositar(1000)
        print(f"Saldo cuenta 1: {cuenta_1.obtenerSaldo():.2f}") # 1100
        print(f"Saldo cuenta 2: {cuenta_2.obtenerSaldo():.2f}") # 1000
        cuenta_mayor = cuenta_1.cuentaMayorSaldo(cuenta_2)
        print(f"La cuenta con mayor saldo es: {cuenta_mayor}")
        print(f"El código de la cuenta con mayor saldo es: {cuenta_1.codMayorSaldo(cuenta_2)}")
```

El cambio en la especificación de los requerimientos y en el diseño, obliga a modificar también la clase tester o crear una nueva, para **verificar los nuevos servicios**.

Cambios en la implementación

```
class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta_1 = CuentaCorriente(1, 1000)
        cuenta_2 = CuentaCorriente(2)
        cuenta_1.depositar(100)
        cuenta_2.depositar(1000)
        print(f"Saldo cuenta 1: {cuenta_1.obtenerSaldo():.2f}") # 1100
        print(f"Saldo cuenta 2: {cuenta_2.obtenerSaldo():.2f}") # 1000
        cuenta_mayor = cuenta_1.cuentaMayorSaldo(cuenta_2)
        print(f"La cuenta con mayor saldo es: {cuenta_mayor}")
        print(f"El código de la cuenta con mayor saldo es: {cuenta_1.codMayorSaldo(cuenta_2)}")
```

Envía el mensaje **codMayorSaldo** al objeto ligado a la variable **cuenta_1**.
El **parámetro real** es una variable que **referencia a un objeto de clase CuentaCorriente**.

Cambios en la implementación

```
print(f"El código de la cuenta con mayor saldo es: {cuenta_1.codMayorSaldo(cuenta_2)}")
```

```
    def codMayorSaldo(self, otraCuenta:"CuentaCorriente")->int:
        """Requiere que 'otraCuenta' esté ligada (no sea None).
        Devuelve el código de la cuenta con mayor saldo."""
        if self.__saldo > otraCuenta.obtenerSaldo():
            return self.__codigo
        else:
            return otraCuenta.obtenerCodigo()
```

La consulta accede al atributo propio de forma directa, y mediante los servicios provistos por la clase accede al atributo del objeto referenciado.
Retorna un valor entero que corresponde al código de la cuenta con mayor saldo.

Cambios en la implementación

```
class TestCuentaCorriente:
    # verifica los servicios de la clase CuentaCorriente
    @staticmethod
    def test():
        cuenta_1 = CuentaCorriente(1, 1000)
        cuenta_2 = CuentaCorriente(2)
        cuenta_1.depositar(100)
        cuenta_2.depositar(1000)
        print(f"Saldo cuenta 1: {cuenta_1.obtenerSaldo():.2f}") # 1100
        print(f"Saldo cuenta 2: {cuenta_2.obtenerSaldo():.2f}") # 1000
        cuenta_mayor = cuenta_1.cuentaMayorSaldo(cuenta_2)
```

Envía el mensaje **cuentaMayorSaldo** al objeto ligado a la variable **cuenta_1**.
El **parámetro real** es una variable que **referencia a un objeto de clase CuentaCorriente**.

Cambios en la implementación

```
cuenta_mayor = cuenta_1.cuentaMayorSaldo(cuenta_2)
```

```
def cuentaMayorSaldo(self, otraCuenta:"CuentaCorriente")->"CuentaCorriente":
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self
    else:
        return otraCuenta
```

La consulta retorna una **referencia** a un objeto de clase **CuentaCorriente.**
Si el saldo del objeto que recibe el mensaje tiene mayor saldo, entonces retorna la referencia a sí mismo.
Si el saldo del objeto ligado al parámetro tiene mayor saldo, entonces retorna la referencia a ese objeto

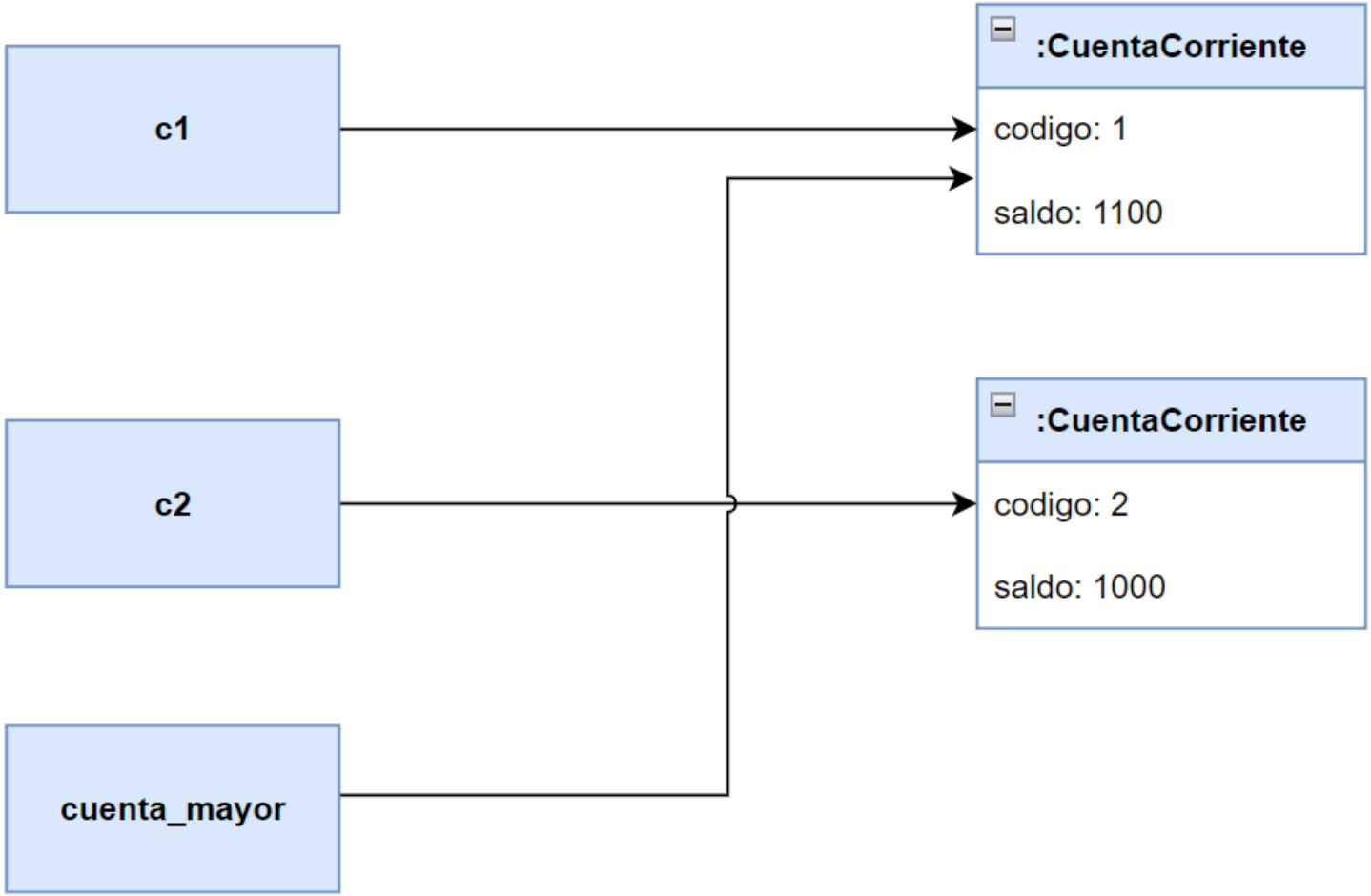
Cambios en la implementación

```
cuenta_mayor = cuenta_1.cuentaMayorSaldo(cuenta_2)
```

```
def cuentaMayorSaldo(self, otraCuenta:"CuentaCorriente")->"CuentaCorriente":
    """Requiere que 'otraCuenta' esté ligada (no sea None).
    Devuelve la cuenta con mayor saldo."""
    if self.__saldo > otraCuenta.obtenerSaldo():
        return self
    else:
        return otraCuenta
```

El valor del **parámetro real *cuenta_2*** declarada como variable local en la clase Tester, se asigna al **parámetro formal *otraCuenta***, que sólo es visible dentro de **cuentaMayorSaldo**.

La variable ***otraCuenta*** solo es visible y puede ser usada durante la ejecución de **cuentaMayorSaldo**. Cuando el método termina, la variable se destruye.


c1
c2
cuenta_mayor
:CuentaCorriente
codigo: 1
saldo: 1100
:CuentaCorriente
codigo: 2
saldo: 1000

Cambios en la implementación

```
print(f"La cuenta con mayor saldo es: {cuenta_mayor}")
```

Envía el mensaje ***__str__*** al objeto ligado a la variable ***cuenta_mayor.***

La variable ***cuenta_mayor*** está ligada con la referencia que retornó como resultado el mensaje **cuentaMayorSaldo** enviado en la instrucción anterior.

El método **__str__** retorna la concatenación de valores de los atributos del objeto ligado a la variable ***cuenta_mayor***.

Identidad, igualdad y equivalencia

Cada objeto de software tiene una **identidad**, una **propiedad** que lo distingue de los demás.

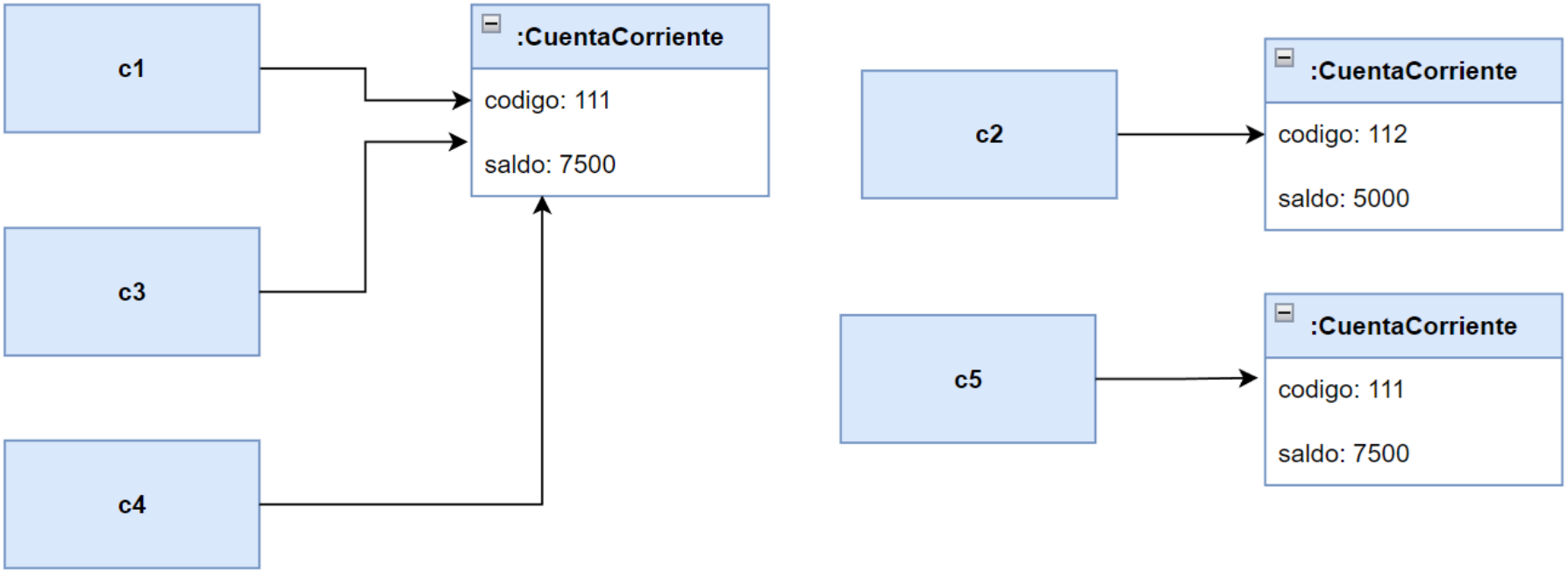
La **referencia** a un objeto puede ser usada como propiedad para identificarlo.

Si dos variables son iguales, significa que mantienen una misma referencia, y por lo tanto están ligadas a un mismo objeto.

Cuando dos objetos mantienen el mismo estado interno, decimos que son **equivalentes**, aún cuando tienen diferente identidad.

Identidad, igualdad y equivalencia

```
class TestReferencias:
    @staticmethod
    def test():
        c1 = CuentaCorriente(111,7500)
        c2 = CuentaCorriente(112,5000)
        c3 = c1.cuentaMayorSaldo(c2)
        c4 = c1
        c5 = CuentaCorriente(111,7500)
        igualdad1 = c1 == c3
        igualdad2 = c1 == c4
        igualdad3 = c1 == c5
        print(f"c1 == c3 : {igualdad1}")
        print(f"c1 == c4 : {igualdad2}")
        print(f"c1 == c5 : {igualdad3}")
```




```
igualdad1 = c1 == c3
igualdad2 = c1 == c4
igualdad3 = c1 == c5
```

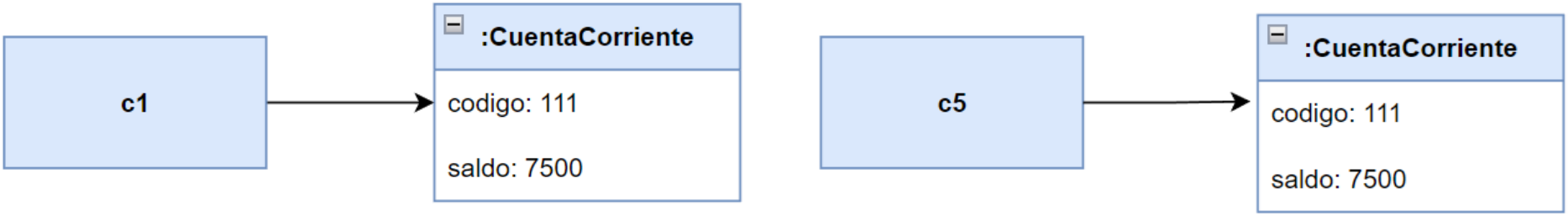
En este caso el operador relacional **==** compara **referencias** entre objetos.

Identidad, igualdad y equivalencia

```
if c1.obtenerCodigo() == c3.obtenerCodigo() and c1.obtenerSaldo() == c3.obtenerSaldo():
        print("El estado interno es igual")
```

En este caso el operador relacional **==** compara valores de **variables elementales**.

Identidad, igualdad y equivalencia



Dos objetos que tienen el mismo estado interno son **equivalentes**.