Herencia

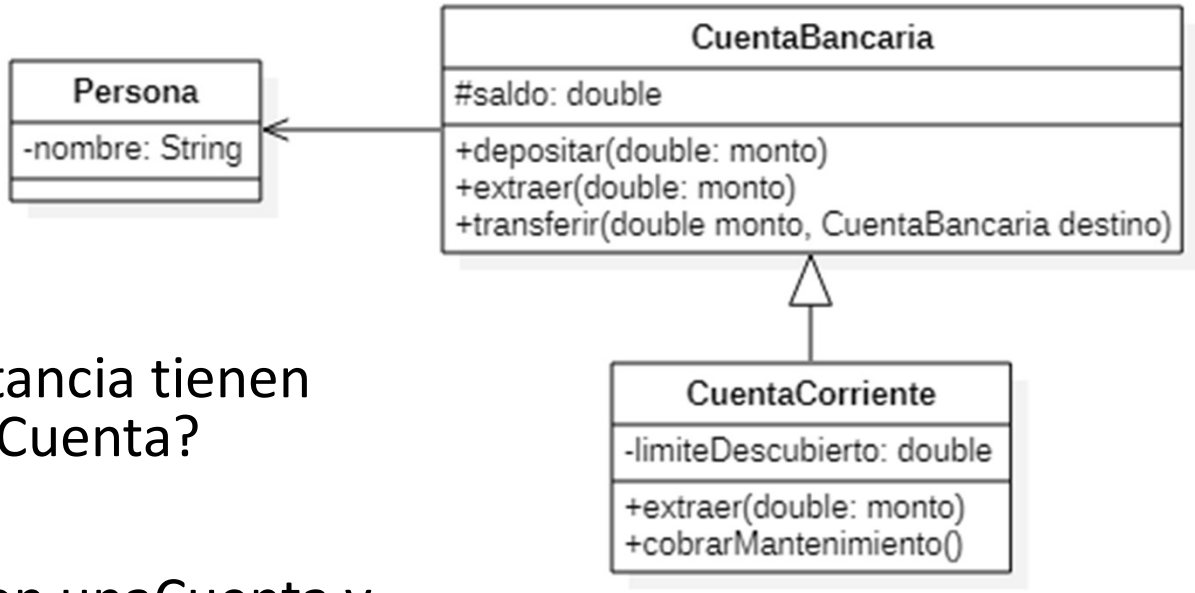
Herencia, clases concretas, clases abstractas, generalización y especialización, this y super

Herencia

- Mecanismo que permite a una clase “heredar” estructura y comportamiento de otra clase
  - Es una estrategia de reúso de código
  - Es una estrategia para reúso de conceptos
- Es una característica transitiva

```
public class CuentaCorriente extends CuentaBancaria {
```

Herencia



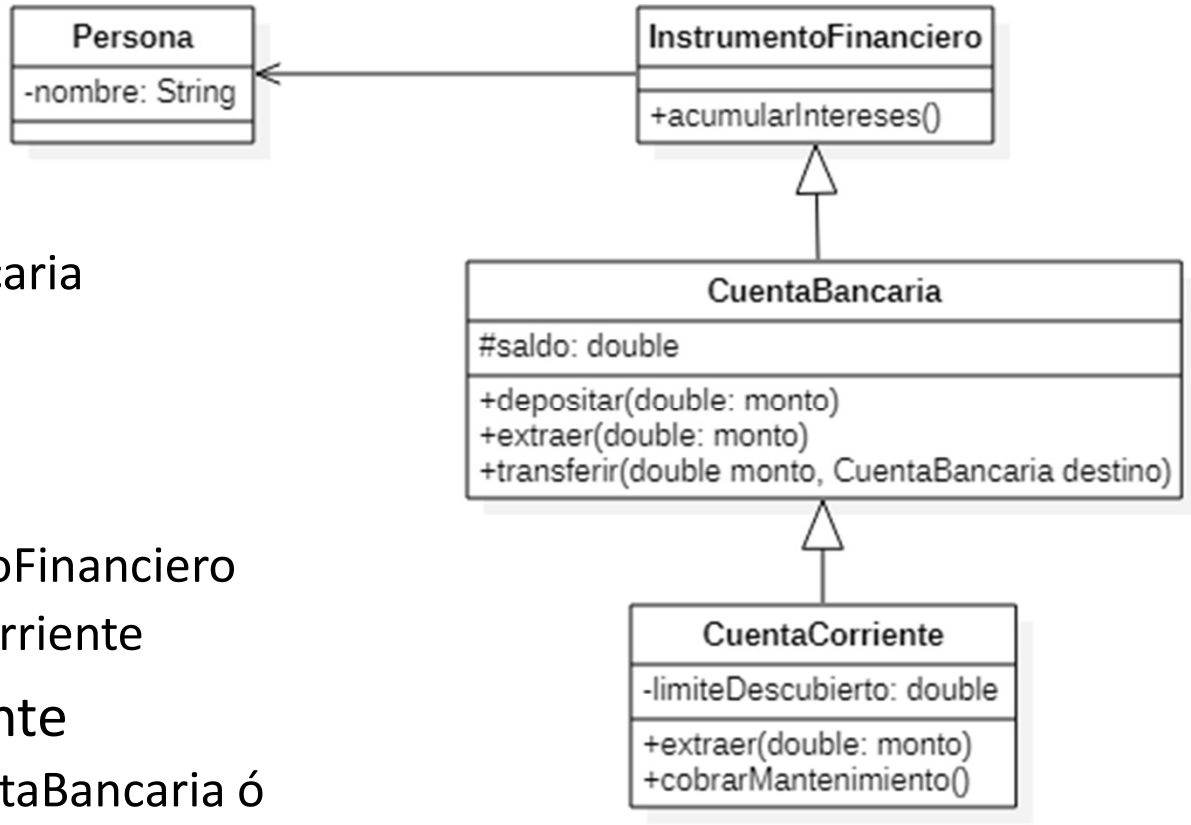
¿Qué variables de instancia tienen unaCuenta y otraCuenta?

¿Qué mensajes entienden unaCuenta y otraCuenta?

```
CuentaBancaria unaCuenta = new CuentaBancaria(alguien);
CuentaCorriente otraCuenta = new CuentaCorriente(alguien, 0, 1000);
```

Vocabulario

- CuentaCorriente
  - es subclase de CuentaBancaria
  - hereda de CuentaBancaria
  - extiende CuentaBancaria
- CuentaBancaria
  - es subclase de InstrumentoFinanciero
  - es superclase de CuentaCorriente
- extraer() en CuentaCorriente
  - extiende extraer() de CuentaBancaria ó
  - redefine extraer() de CuentaBancaria



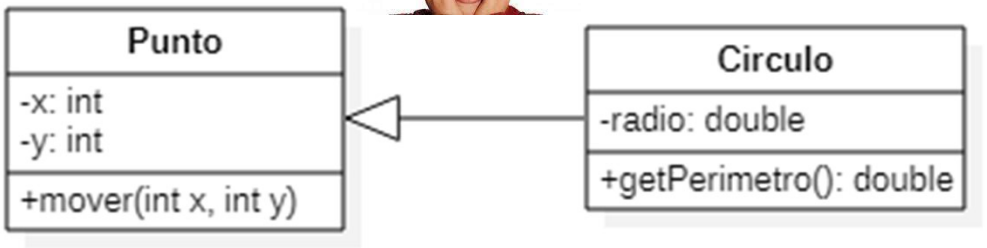
La prueba “es un” ...

- Preguntarse “es-un” es la regla para identificar usos adecuados de herencia
  - Si suena bien en el lenguaje del dominio, es probable que sea un uso adecuado
  - Una cuenta corriente es una cuenta, una ventana de texto es una ventana, un array es una colección, un robot es un agente, ...

- Un anti-ejemplo famoso:

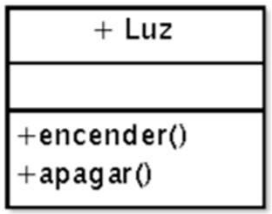
Un Círculo NO-ES-UN Punto,

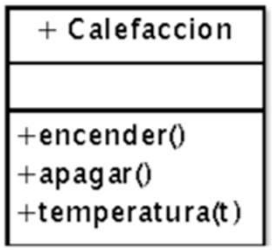
conoce o tiene un punto



Method Lookup (recordamos)

- Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje.
  - En un lenguaje dinámico, podría no encontrarlo (error en tiempo de ejecución)
  - En un lenguaje con tipado estático sabemos que lo entenderá (aunque no sabemos lo que hará)

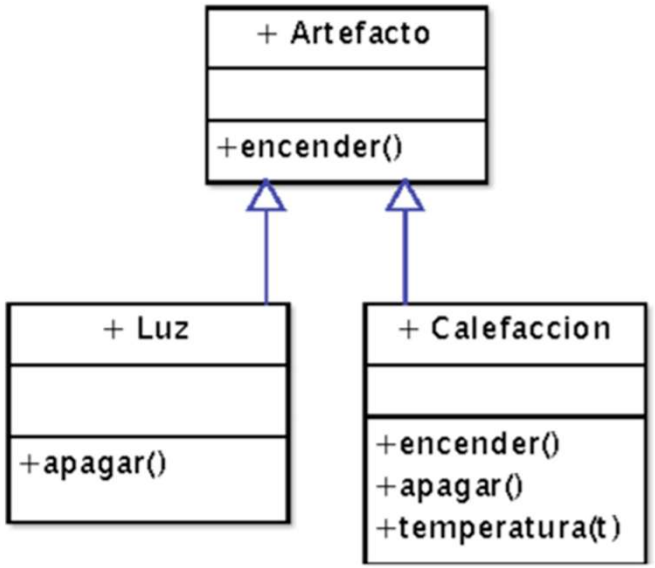





```
(new Luz()).encender();
(new Calefaccion()).encender();
(new Puerta()).encender();
```

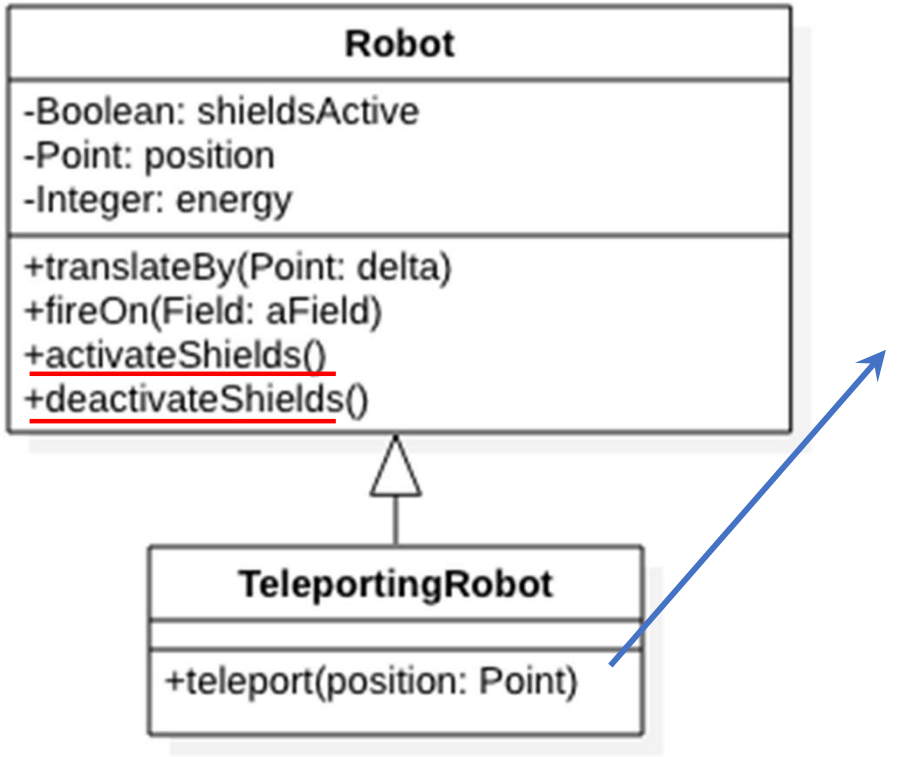
Method Lookup con herencia

- Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje. Si no lo encuentra, sigue buscando en la superclase de su clase, y en la superclase de esta ...




```
(new Luz()).encender();
(new Calefaccion()).encender();
```

Aprovechar comportamiento heredado

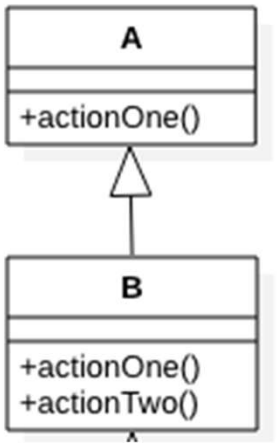



```
public void teleport(Point2D position) {
    this.activateShields();
    this.position = position;
    this.deactivateShields();
}
```

Sobre escribir métodos (overriding)

- La búsqueda en la cadena de superclases termina tan pronto encuentro un método cuya firma coincide con la que busco.
- Si heredaba un método con la misma firma, el mismo queda “oculto” (redefinir / override)
- No es algo que ocurra con frecuencia (puede dar mal olor)

```
(new B()).actionOne();
```

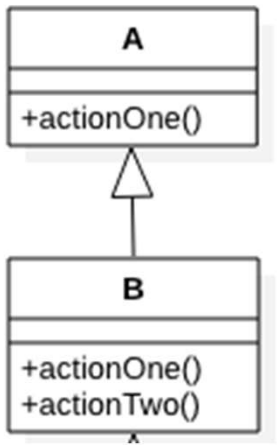



```
public void actionOne() {
    // Hacer algo como le gusta a A
}

public void actionOne() {
    // Hacer algo como le gusta a B
}
```

Extender métodos

- ¿Qué hacemos si queremos aprovechar (extender) ese método que heredábamos?

```
(new B()).actionOne();
```




```
public void actionOne() {
    // Hacer algo como le gusta a A
}
```

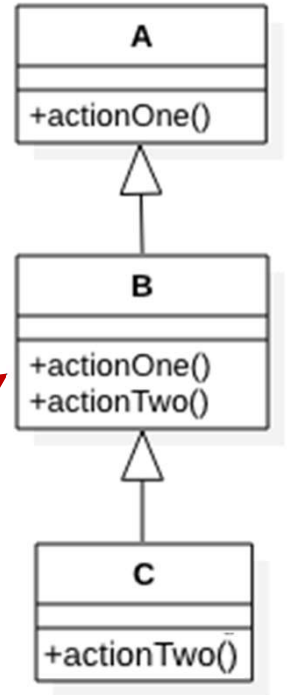
```
public void actionOne() {
    // Hacer algo como le gusta a B
    // Hacer algo como le gusta a A
    // Hacer algo como le gusta a B
}
```

super

- **Podemos pensar a super** como una “pseudo-variable” (como this)
  - no puedo asignarle valor
  - Toma valor automáticamente cuando un objeto comienza a ejecutar un método
- En un método, **“super y this” hacen** referencia al objeto que lo ejecuta (al receptor del mensaje)
- Utilizar super en lugar de this “solo cambia la forma en la que se hace el method lookup”
- Se utiliza para:
  - Solamente para extender comportamiento heredado (reimplementar un método e incluir el comportamiento que se heredaba para él).

super y el method lookup

Cuando **super** recibe un mensaje, la búsqueda de métodos comienza en la clase **inmediata superior** a aquella donde está definido el método que envía el mensaje (sin importar la clase del receptor)




```
(new A()).actionOne();

(new B()).actionOne();

(new C()).actionOne();
```

```
public void actionOne() {
    this.actionTwo();
    super.actionOne();
    this.actionTwo();
}
```

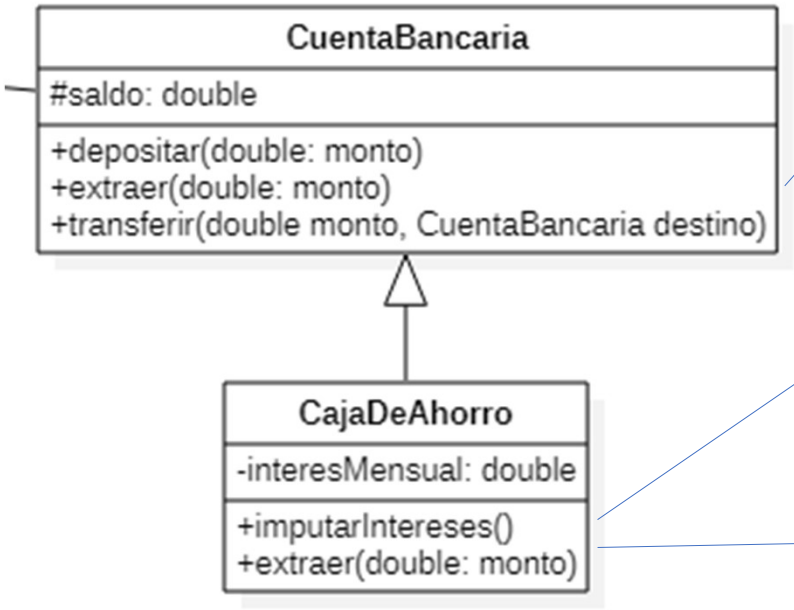
Super() en los constructores

- Los constructores en Java son subrutinas que se ejecutan en la creación de objetos - no se heredan
- Si quiero reutilizar comportamiento de otro constructor debo invocarlo explícitamente - usando super(...) al principio

```
public CuentaBancaria(Persona titular) {
    this.titular = titular;
}

public CuentaCorriente(Persona titular, double saldoInicial, double limiteDescubierto) {
    super(titular);
    saldo = saldoInicial;
    this.limiteDescubierto = limiteDescubierto;
}
```

Especializar

- Especializar: crear una subclase especializando una clase existente




```java
public void extraer(double monto) {
    saldo = saldo - monto;
}

public void imputarIntereses() {
    this.depositar(interesMensual * saldo);
}

    public void extaer(double monto) {
        if (monto <= saldo) {
            super.extraer(monto);
        }
    }
```

Clase abstracta

- Una clase abstracta captura comportamiento y estructura que será común a otras clases
- Una clase ab
  instancias (n
  completo)
- Seguramente será especializada
- Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto

Gancho que deben implementar las subclases

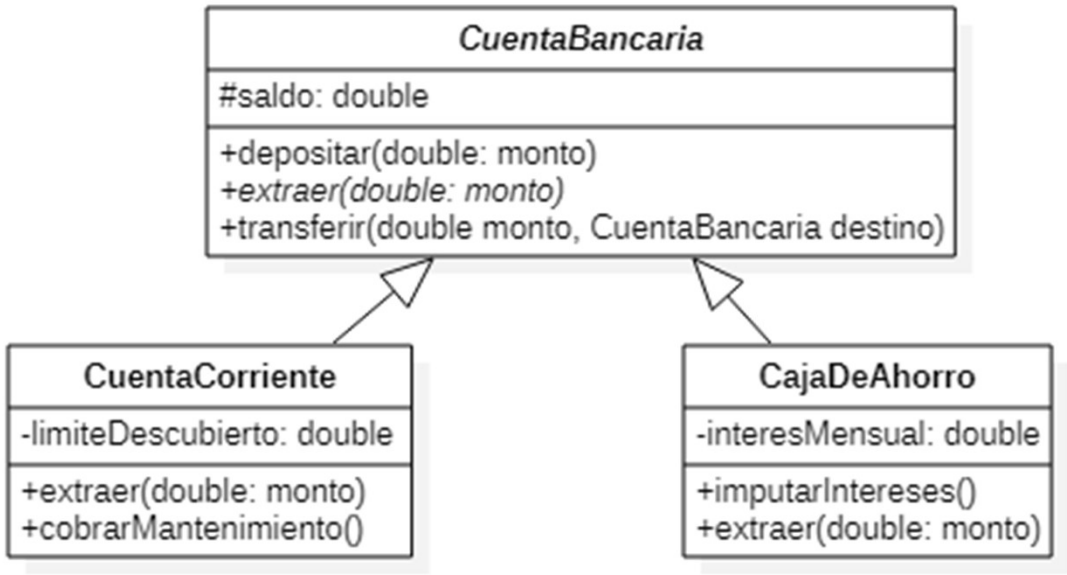
```
public abstract class CuentaBancaria {
    protected double saldo;

    public void depositar(double monto) {
        saldo = saldo + monto;
    }

    public void transferir(double monto, CuentaBancar
        destino.depositar(monto);
        this.extraer(monto);
    }

    public abstract void extraer(double monto);
}
```

Generalizar

- Generalizar: Introducir una superclase que abstrae aspectos comunes a otras – suele resultar en una clase abstracta

Generalizar

- Generalizar: Introducir una superclase que abstrae aspectos comunes a otras – suele resultar en una clase abstracta



Tipos en lenguajes OO (recordamos)

- Tipo = Conjunto de firmas de operaciones/métodos (nombre, tipos de argumentos, tipo del resultado)
- Decimos que un objeto “es de un tipo” si ofrece el conjunto de operaciones definido por el tipo
- Con eso en mente:
  - Cada clase en Java define “explícitamente” un tipo
  - Cada instancia de una clase A (o de cualquier subclase) “es del tipo” definido por esa clase
- Donde espero un objeto de una clase A, acepto un objeto de cualquier subclase de A (lo opuesto no es cierto)

Modificadores de visibilidad con herencia

- Si declaro una variable de instancia como private en una clase A:
  - Las instancias de las subclases de A tendrán esa variable de instancia
  - En los métodos de las subclases de A, no puedo hacer referencia ella
- Si declaro un método “m()” como private en una clase A:
  - Las instancias de sus subclases entenderán el mensaje m()
  - En los métodos de las subclases de A no puedo enviar m() a ”this“
- Si declaro variables y métodos como protected, puedo verlos en las subclases
- ¿Qué impacto tiene restringir la visibilidad de esta manera en términos de acoplamiento?