



# Herencia

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

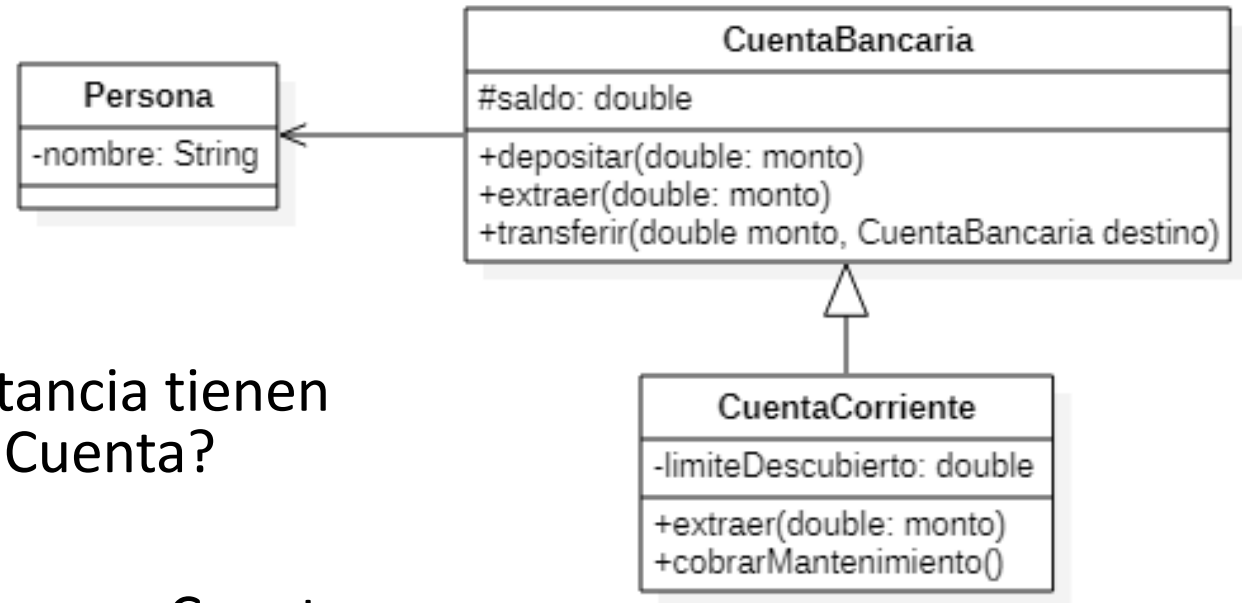
# Herencia

- Mecanismo que permite a una clase “heredar” estructura y comportamiento de otra clase
  - Es una estrategia de reúso de código
  - Es una estrategia de reúso de conceptos/definiciones
- En Java solo tenemos herencia simple
- Es una característica transitiva

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



# Herencia



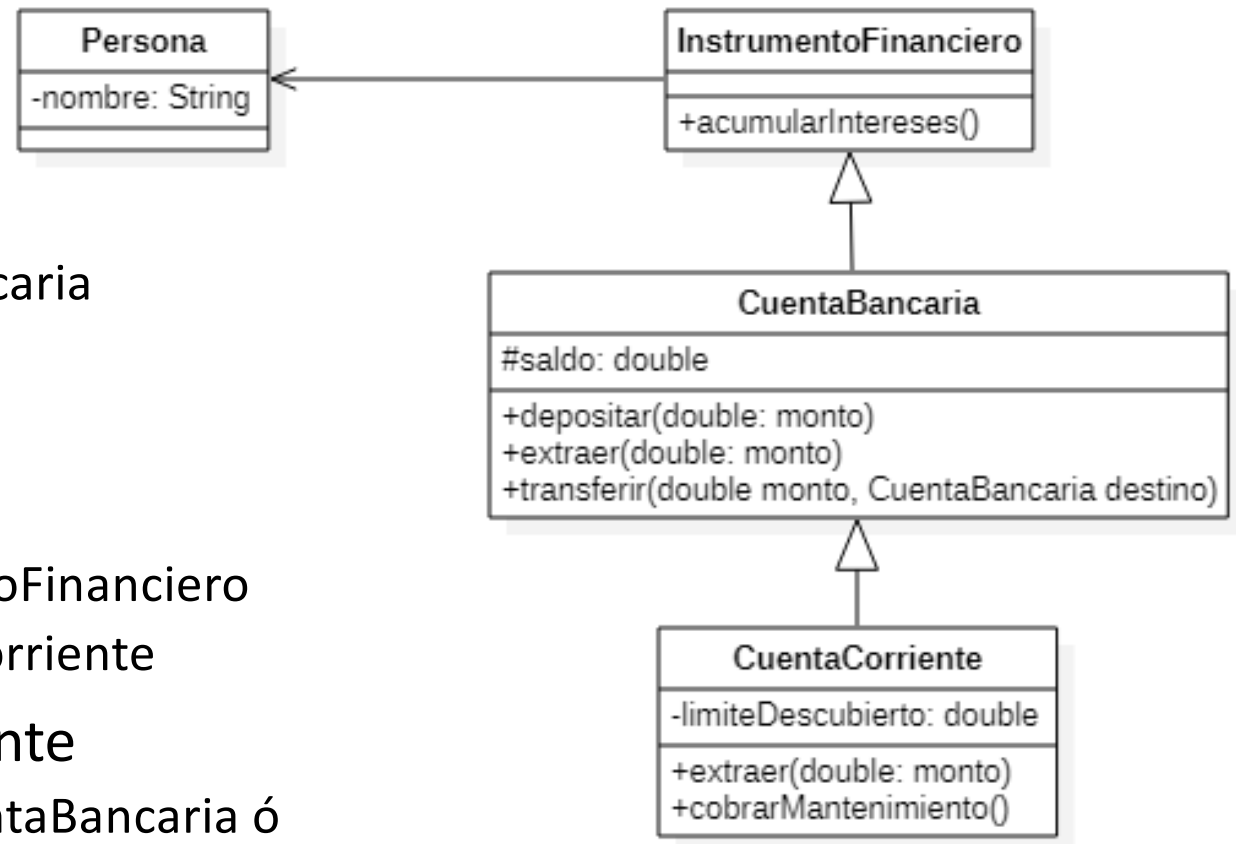
¿Qué variables de instancia tienen unaCuenta y otraCuenta?

¿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

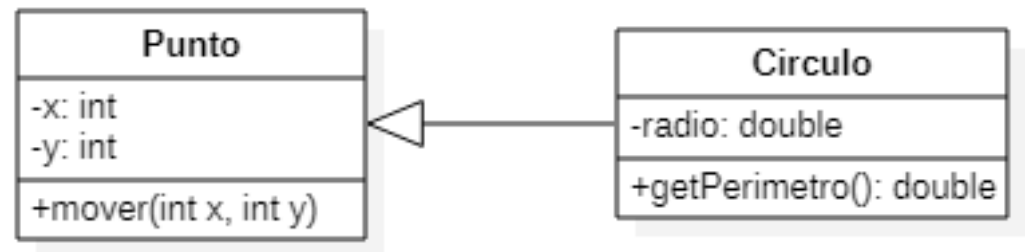


# La prueba “es un” ...

- 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 (herencia para construcción):



Un Círculo NO-ES-UN Punto,  
conoce o tiene un punto



# Principio de substitución de Liskov

- Si en algún lugar de nuestro programa estamos usando una clase, y esta clase es extendida, entonces tenemos que poder usar cualquiera de sus sub-clases y que el programa siga siendo válido (que pase los test).
- Nos ayuda a verificar que los criterios de subclasificación siguen funcionando.
- Es mucho más preciso que el intuitivo “es un”.

Barbara Liskov



# 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)

+ Luz
+encender() +apagar()

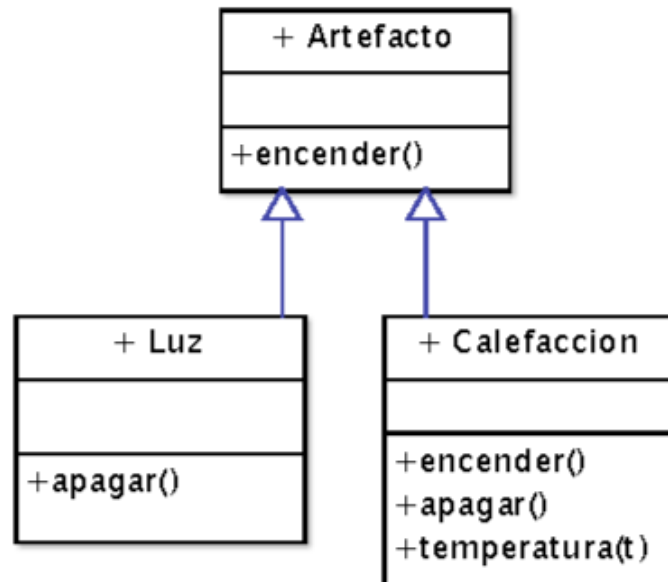
+ Calefaccion
+encender() +apagar() +temperatura(t)

+ Puerta
+abrir() +cerrar()

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

# Method Lookup con herencia

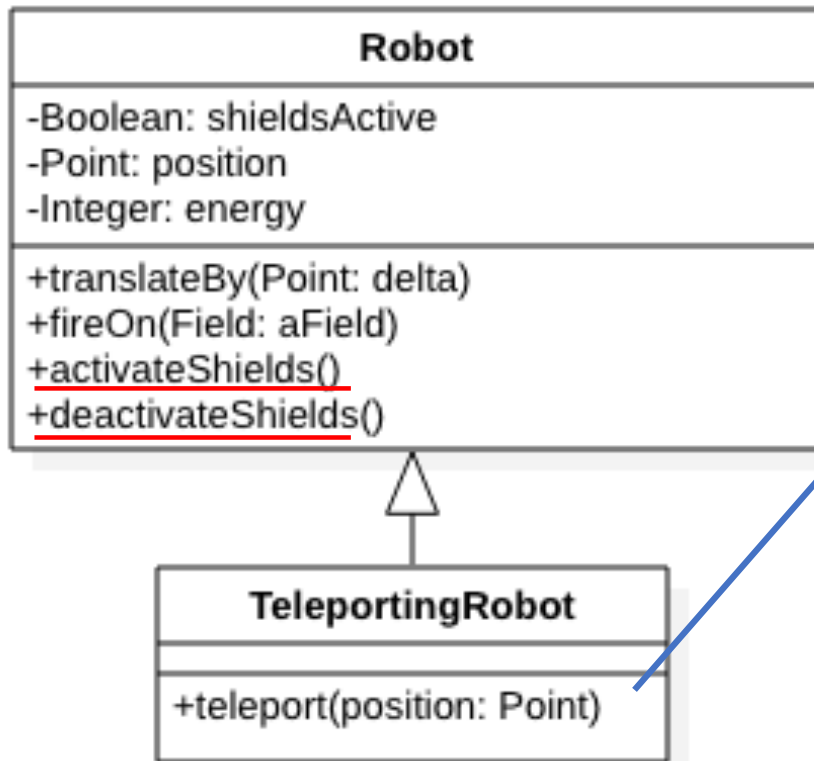
- Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje. Si no lo encuentra, sigue buscando en la superclase de su clase, y en la superclase de esta ...



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



# Aprovechar comportamiento heredado



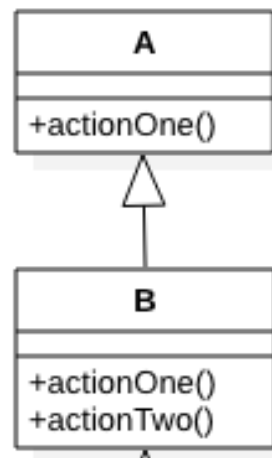
```
public void teleport(Point2D position) {
    this.activateShields();
    this.position = position;
    this.deactivateShields();
}
```

A blue arrow points from the `+teleport(position: Point)` method in the **TeleportingRobot** class to the `teleport` method in the code block, highlighting the implementation of the method by leveraging inherited behavior.

# Sobre escribir métodos (overriding)

- La búsqueda en la cadena de superclases termina tan pronto encuentre 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();
```



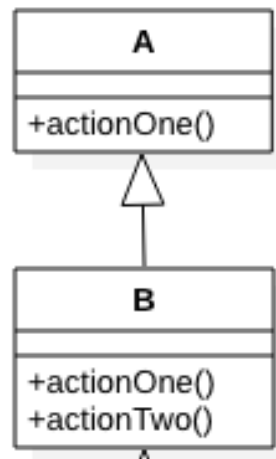
```
public void actionOne() {  
    // Hacer algo como le gusta a A  
}
```

```
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

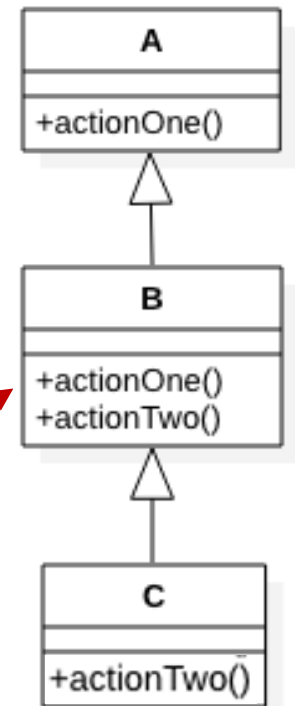
Cuando **super** recibe un mensaje, la búsqueda de métodos comienza en la clase **inmediata superior** a aquella donde está definido el método que envía el mensaje (sin importar la clase del receptor)

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

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

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

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



# Super() en los constructores

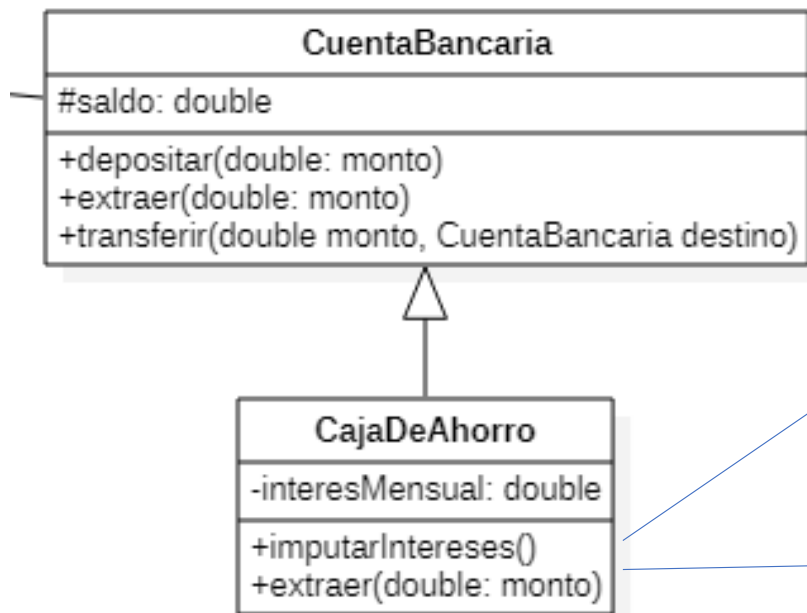
- Los constructores en Java son subrutinas que se ejecutan en la creación de objetos - no se heredan
- Si quiero reutilizar comportamiento de otro constructor debo invocarlo explícitamente - usando `super(...)` al principio

```
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



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

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

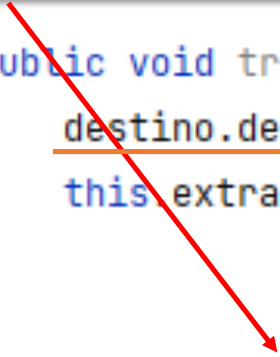
```
public void extraer(double monto) {
    if (monto <= saldo) {
        super.extraer(monto);
    }
}
```

# Clase abstracta

- Una clase abstracta captura comportamiento y estructura que será común a otras clases
- Una clase abstracta no tiene instancias (no es un objeto completo)
- Seguramente será especializada
- Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto

**Gancho que deben implementar las subclasses**

```
public abstract class CuentaBancaria {  
    protected double saldo;  
  
    public void depositar(double monto) {  
        saldo = saldo + monto;  
    }  
  
    public void transferir(double monto, CuentaBancaria  
        destino) {  
        destino.depositar(monto);  
        this.extraer(monto);  
    }  
  
    public abstract void extraer(double monto);  
}
```





# Generalizar

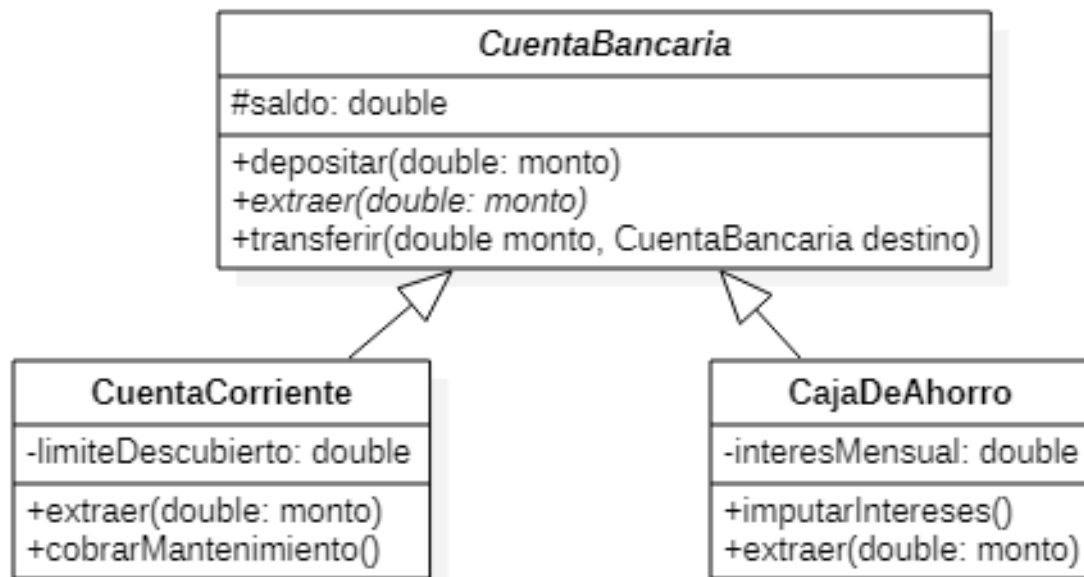
- Generalizar: Introducir una superclase que abstraee aspectos comunes a otras – suele resultar en una clase abstracta

CuentaCorriente
-limiteDescubierto: double -saldo: double
+depositar(double: monto) +transferir(double monto, CuentaCorriente destino) +extraer(double: monto) +cobrarMantenimiento()

CajaDeAhorro
-interesMensual: double -saldo: double
+depositar(double: monto) +transferir(double monto, CajaDeAhorro destino) +extraer(double: monto) +imputarIntereses()

# Generalizar

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



# Situaciones de uso de herencia

- Subclasificar para especializar
  - La subclase extiende métodos método para especializarlos
  - Ambas clases concretas
- Herencia para especificar
  - La superclase es una combinación de métodos concretos y abstractos (clase abstracta)
  - La subclase implementa los métodos abstractos
- Subclasificar para extender
  - La subclase agrega nuevos métodos

# Situaciones de uso de herencia (feas)

- Heredar para construir
  - Heredo comportamiento y estructura, pero cambio el tipo (no-es-un)
  - Se debe evitar, aunque nos vamos a cruzar con ejemplos (en código de otros)
  - Si es posible, reemplazar por composición
- Subclasificar para generalizar (no es lo mismo que generalización)
  - La subclase reimplementa métodos para hacerlos más generales
  - Solo tiene sentido si no puedo reordenar la jerarquía (para especializar)
- Subclasificar para limitar
  - La subclase reimplementa un método para que deje de funcionar / limitarlo
  - Solo tiene sentido si no puedo reordenar la jerarquía (para especializar)

# Situaciones de uso de herencia (feas)

- Herencia indecisa (subclasificación por varianza)
  - Tengo dos clases con un mismo tipo, y algunos métodos compartidos
  - No puedo decidir cuál es la subclase y cual la superclase
  - Resolverlo buscando una superclase común (generalización)

# Tipos en lenguajes OO (recordamos)

- Tipo = Conjunto de firmas de operaciones/métodos (nombre, tipos de argumentos)
- 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)
  - Principio de substitución

# Clases abstractas e interfaces

- Una clase abstracta “es una clase”
  - Puedo usarla como tipo
  - Puede o no tener métodos abstractos
  - Ofrece implementación a algunos métodos
  - Sus métodos concretos pueden depender de sus métodos abstractos (p.e., si se define antes de sus subclases)
- Una interfaz define un tipo
  - Sirve como contrato
  - Puede extender a otras
- Se pueden implementar muchas interfaces pero solo se puede heredar de una clase

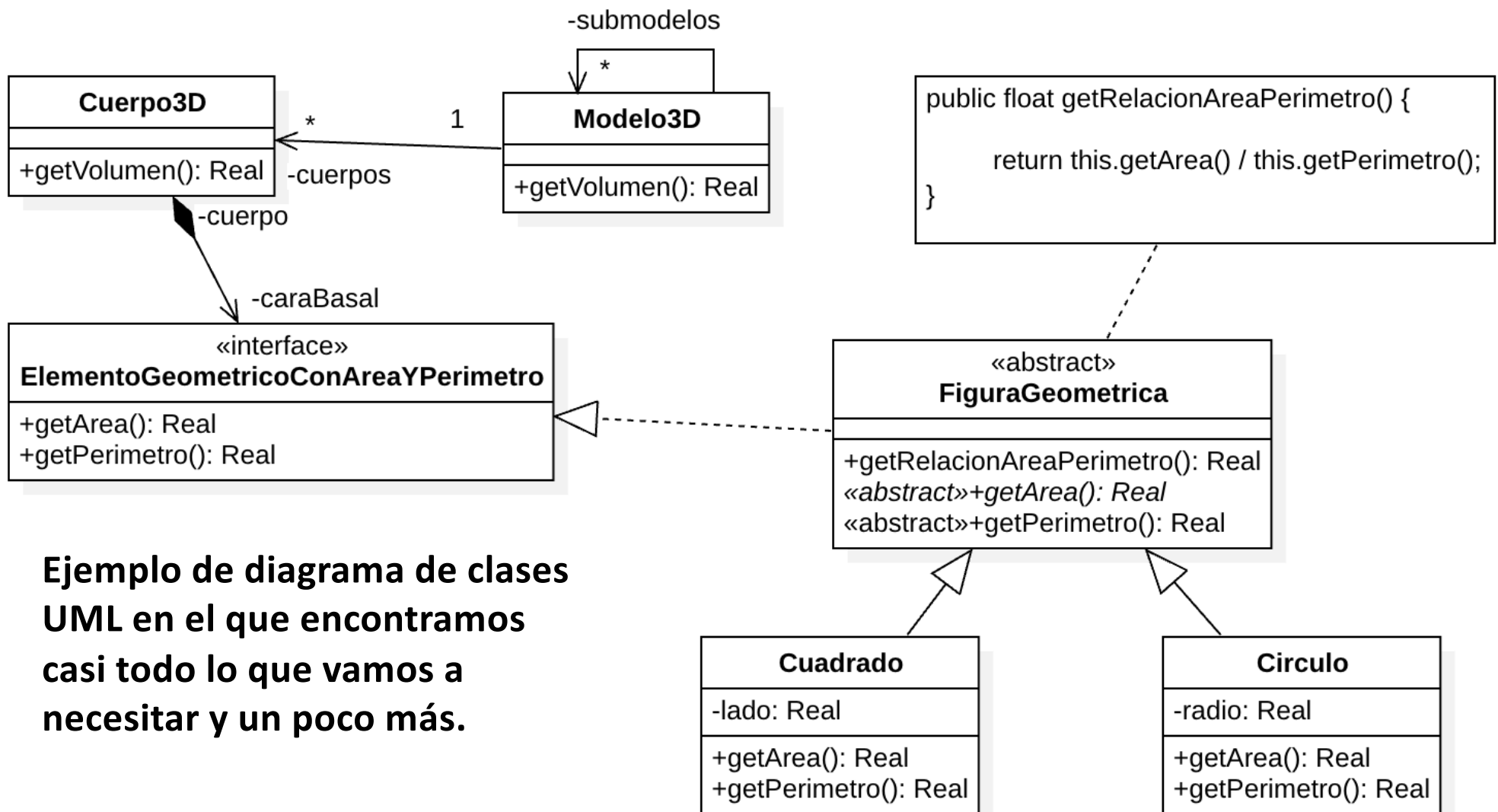
# Modificadores de visibilidad con herencia

- 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?



# Visibilidad, herencia y OO1

- El ocultamiento de información favorece el bajo acoplamiento
- Nos interesa sostener esta premisa incluso en relaciones de herencia
- En OO1 vamos a utilizar la siguiente estrategia:
  - Las variables de instancia son siempre privadas.
  - Si en una subclase, para extender o especializar, dependo de ese conocimiento, agrego accessors (get/set) protegidos.

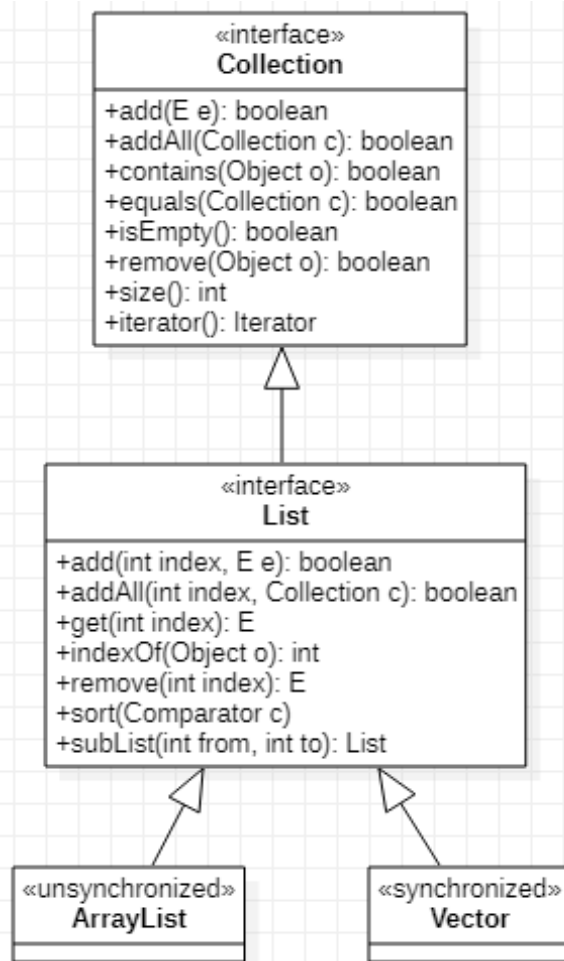


**Ejemplo de diagrama de clases UML en el que encontramos casi todo lo que vamos a necesitar y un poco más.**

Hagamos un poco de ejercicio ..



# Listas ...



```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```
public class Ticket {

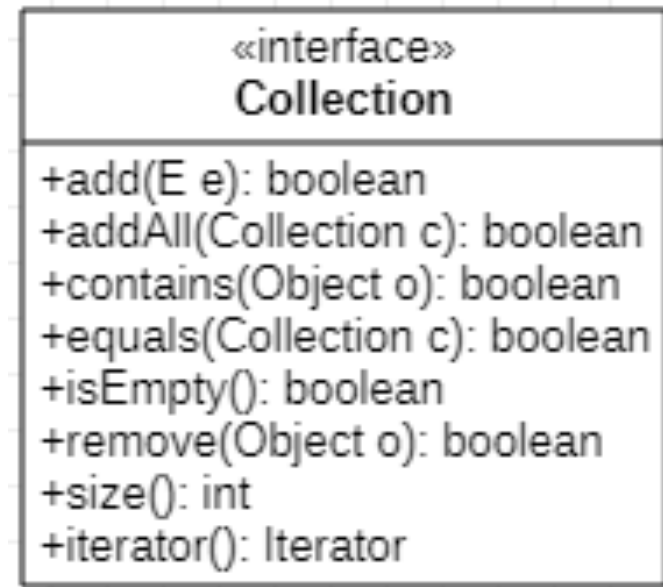
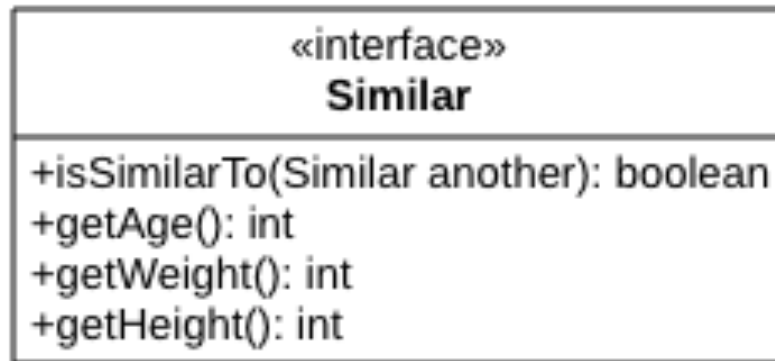
    private List<Item> items;
    private Date fecha;
    private Cliente cliente;

    public Ticket(Cliente cliente) {
        this.cliente = cliente;
        fecha = new Date();
        items = new ArrayList<Item>();
    }

    public void agregarItem(Producto producto, int cantidad) {
        items.add(new Item(producto, cantidad));
    }

    public List<Item> getItems() {
        return items;
    }
}
```

## Lista de parecidos ...



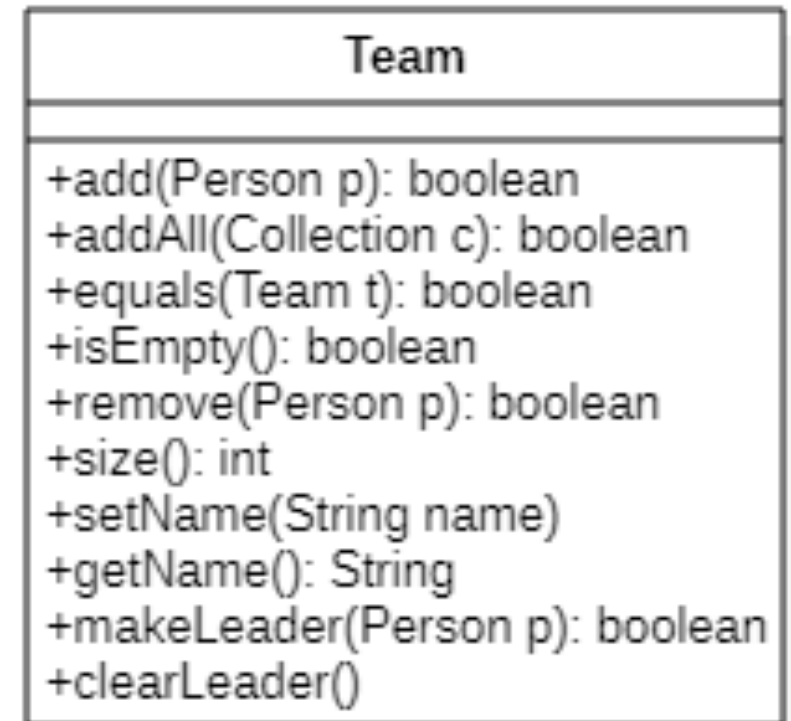
- Queremos una colección, que en su constructor tome un Similar (su ejemplo), y solo acepte objetos que son similares a ese
  - Cuando no acepta (no agrega) a un objeto, levanta IllegalArgumentException
- El objeto no es elemento de la colección si no se lo agrega explícitamente
- No se puede cambiar el ejemplo una vez creada la colección (pero si se lo puede obtener)

## Lista de parecidos ... (para reflexionar)

- ¿Subclasificamos ArrayList (u otra) o conocemos a una instancia de ArrayList (u otra)?
- ¿Se cumple el principio de es-un?
- Si subclasificamos (heredamos) ..,
  - ¿que variables de instancia tenemos que definir?
  - ¿qué métodos debemos implementar y cómo?
- Si conocemos (componemos) ..,
  - ¿que variables de instancia tenemos que definir?
  - ¿qué métodos debemos implementar y cómo?
- ¿Qué pasa con los constructores en cada caso?

# Equipos

- Nos piden implementar la clase Team
- Se pueden agregar, quitar miembros, comparar, etc...
- El líder (si lo tiene) debe ser uno de los miembros
- Si se elimina el líder, el puesto queda vacante
- Los miembros no tienen ningún orden particular

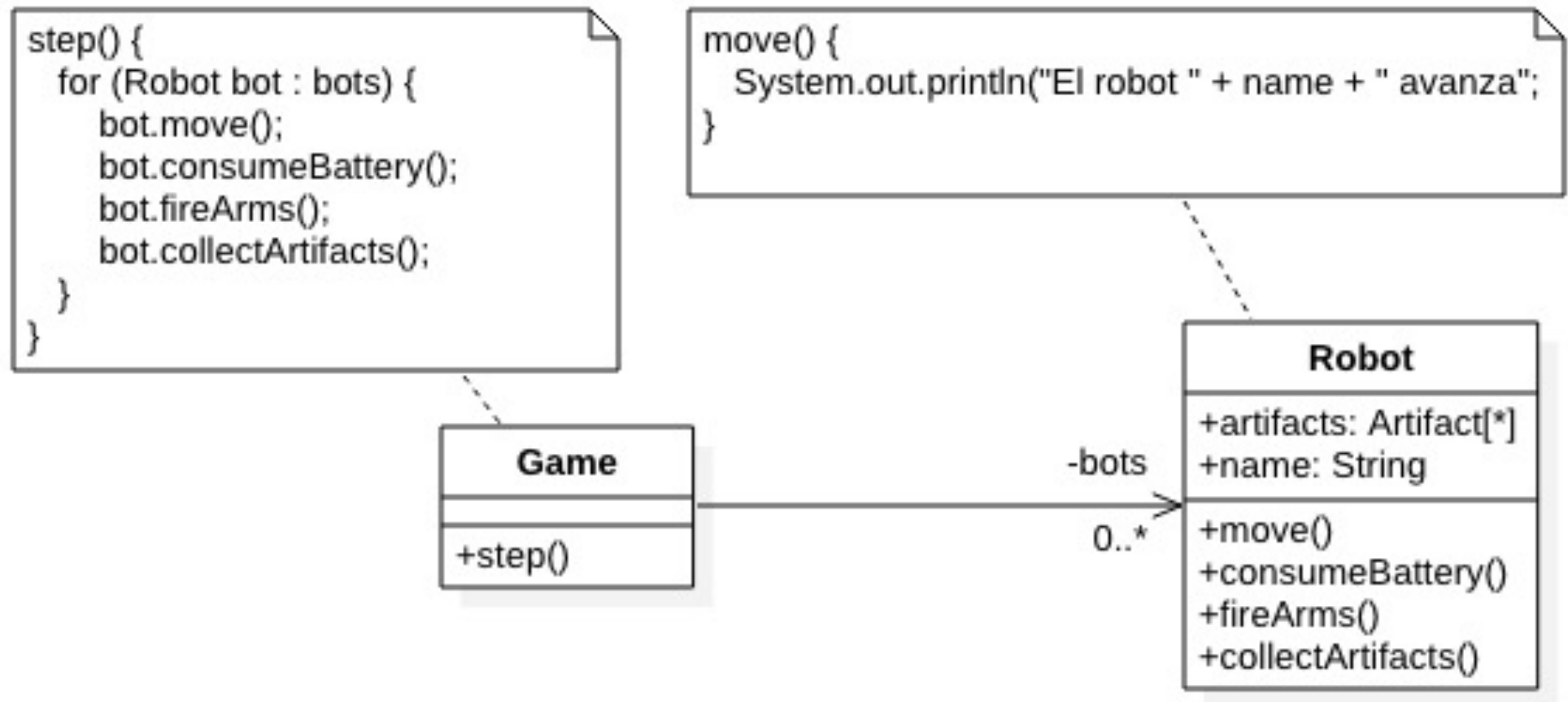


# Equipos... (para reflexionar)

- ¿Subclasificamos ArrayList (u otra) o conocemos a una instancia de ArrayList (u otra)?
- ¿Se cumple el principio de es-un?
- Si subclasificamos (heredamos) ..,
  - ¿que variables de instancia tenemos que definir?
  - ¿qué métodos debemos implementar y cómo?
- Si conocemos (componemos) ..,
  - ¿que variables de instancia tenemos que definir?
  - ¿qué métodos debemos implementar y cómo?
- ¿Qué pasa con los constructores en cada caso?



# Juego de Robots



# Juego de Robots

- Nos piden SolarRobots
  - Son como los Robots pero en consumeEnergy() imprimen "... consume energía solar"
- Nos piden NuclearRobots
  - Son como los Robots pero en consumeEnergy() imprimen "... consume energía nuclear"
- Nos piden CaterpillarRobots
  - Son como los Robots pero en move() imprimen "... avanza sobre orugas"
- Nos piden NuclearCaterpillarRobots y SolarCaterpillarRobots
- Nos piden OvercraftRobot
- Nos piden NuclearOvercraftRobot y SolarOvercraftRobot
- Nos piden CadmiumRobot, FlyingRobot, BombingRobot, LaserFiringRobot y todas las combinaciones que se nos ocurran

# Herencia y Robots

- ¿Qué pasa cuando aparecen nuevas formas de mover, disparar, consumir batería y recolectar?
- ¿Podemos cambiar el comportamiento de un Robot en tiempo de ejecución?
- ¿En los métodos de mis clases de robot, tengo acceso a las variables de instancia del robot?
- ¿Cuántos objetos hacen a un robot?
- ¿Podemos descomponer a los robots en varias clases, reutilizando mejor, y sin usar herencia?

Lectura recomendada

*An Introduction to Object-Oriented Programming,*

Timothy Budd

- Capitulo 8

