

UT 6. DESARROLLO DE CLASES

Programación de 1DAW

Índice

- 1.- Reutilización de código
- 2.- Relaciones entre clases
 - 2.1.- Composición
 - 2.2.- Semántica de referencia
 - 2.3.- **Herencia**
 - 2.3.1.- Características
 - 2.3.2.- Constructores
 - 2.3.3.- Redefinir métodos
 - 2.3.4.- Invocar a métodos de la superclase
 - 2.3.5.- Modificador Protected
 - 2.3.6.- Modificador final
 - 2.3.7.- Reconocer la herencia
 - 2.3.8.- Ventajas y desventajas
- 3.- Polimorfismo
- 4.- Jerarquias de clases
- 5.- La clase Object
- 6.- Inicialización y carga de clases
- 7.- Combinar Herencia y Composición.
- 8.- Elegir Herencia o Composición.
- 9.- Tabla resumen modificadores de acceso
- 10.- Destrucción de objetos.


Reutilización de código

- Una de las características de Java es la **reutilización** de código.
- Debemos ser capaces de reutilizar el código ya desarrollado, ya sea por nosotros mismos o por otros desarrolladores (librerías).
- Al reutilizar librerías (**programador cliente**) únicamente necesitamos conocer la funcionalidad de la misma, no cómo ha sido implementada (encapsulación).
- El creador de la librería o clase debe poder realizar modificaciones en la misma y mejoras con la certeza de que el código del programador cliente no se verá afectado. El creador de la librería cuenta con los modificadores de acceso para decidir qué partes del código son accesibles o no para el programador cliente. Por convención se debe mantener todo lo que sea posible como `private` y “exponer” solo lo que sea necesario para el programador cliente:
 - Se debe separar la interfaz de la implementación.
 - Recordemos que la interfaz de una clase está formada por sus miembros `public`.

Reutilización de código

- Recordamos:
 - En Java aquellas clases de un paquete que queramos estén disponibles se declararán como public.
 - Solo puede haber una clase public por unidad de compilación (fichero .java) y el nombre de esta clase será el nombre del fichero. La idea es que cada unidad de compilación tenga una sola interfaz pública.
 - Pero podrá tener tantas clases “friendly” como se quiera por unidad de compilación.
 - Recordemos: “friendly” es el término utilizado cuando no declaramos ningún modificador de acceso.

Reutilización de código

- Formas de reutilizar código:
 - **Copiar y pegar** código → puede inducir a errores.
 - **Uso de métodos**, que permiten agrupar bajo un nombre código para resolver una tarea → **modularidad**: técnica presente en todos los paradigmas de programación.
 - Técnicas propias de la OO:
 - **Composición**
 - **Herencia**Relaciones entre clases
- Entre clases existen relaciones conceptuales → Ejemplos:
 - ”Una pila puede definirse a partir de una cola o viceversa”
 - ”Un rectángulo es una especialización de polígono”
 - “Libros y revistas tienen propiedades comunes”

Relaciones entre clases

- ❑ **Clientela:** Cuando una clase utiliza objetos de otra clase (por ejemplo cuando pasamos un objeto como parámetro a un método, o la utilización de clases dentro del método main)
- ❑ **Composición:** Cuando alguno de los atributos de una clase es un objeto de otra clase.
- ❑ **Anidamiento:** Cuando se definen clases en el interior de otra clase (lo veremos en el tema siguiente)
- ❑ **Herencia:** Cuando una clase extiende otra clase (clase base) incorporando todas las características de la clase base.
- ❑ A veces se considera tanto la composición como la anidación casos particulares de clientela, pues en estos casos una clase está haciendo uso de otra.

Composición

- Cuando una clase A declara un atributo cuyo tipo es una clase B.
- Ejemplo:

```
public class Cuenta {  
    ...  
    private final Persona titular;  
    private double[] ultimasOperaciones;  
    public Cuenta(Persona persona) {  
        titular = persona;  
        ultimasOperaciones = new double[20];  
    }  
    public Persona getTitular() {  
        return titular;  
    } ...  
}
```

Composición

- Cuando una clase A declara un atributo cuyo tipo es una clase B.
- Ejemplo:

```
public class Cuenta {  
    ...  
    private final Persona titular;  
    private double[] ultimasOperaciones;  
    public Cuenta(Persona persona) {  
        titular = persona;  
        ultimasOperaciones = new double[20];  
    }  
    public Persona getTitular() {  
        return titular;  
    }  
    ...  
}
```


Composición

- Consiste en crear clases que contienen atributos de otras clases.
- La clase se compone de objetos de otras clases existentes →
Reutilizamos la funcionalidad del código pero no nos interesa los detalles de implementación de esa otra clase.
- **Es una relación de tipo “Tiene un”**
- **Sintaxis:**
 - Declarar atributos que son referencias a objetos de otras clases.
 - Ejemplo:

```
Class Coche{  
    Motor m;  
    Rueda[] ruedas;  
    Puerta pDcha;  
    Puerta pIzda;  
    ...  
}
```

Coche tiene un motor;
pero solo nos interesa
lo que puede hacer un
motor, pero no los
detalles concretos de
cómo lo hace.

Composición

- Cuando utilizamos la **composición** deberemos crear (invocar al constructor) los objetos contenidos.
- En el caso de la composición se ha de tener en cuenta en qué momento se inicializan los objetos declarados. **Puede ser en 3 momentos:**
 - **Definición** del atributo (antes de invocar al constructor)
 - En el **constructor** de la clase
 - Justo antes de su uso (**inicialización tardía o perezosa**). Esta forma puede reducir la sobrecarga en situaciones donde el objeto no necesita ser creado cada vez.
- Ejemplo:

Composición

```
public class Baño {
    //Iniciación en el momento de la definición
    private String s1 = new String("Contento");
    private String s2 = "Contento";
    private String s3,s4;
    private Jabon pastilla;
    int i;
    public Baño(){
        System.out.println("Dentro del baño()");
        //Iniciación dentro del constructor
        s3 = new String("Gozo");
        i = 47;
        pastilla = new Jabon();
    }
    void escribir(){
        //Iniciación tardia
        if(s4==null){
            s4 = new String("Temporal");
        }
        System.out.println("s1 = "+s1);
        System.out.println("s2 = "+s2);
        System.out.println("s3 = "+s3);
        System.out.println("s4 = "+s4);
        System.out.println("i = "+i);
        System.out.println("pastilla = "+pastilla);
    }
}
```

```
class Jabon {
    ...
}
```

Semántica de referencia

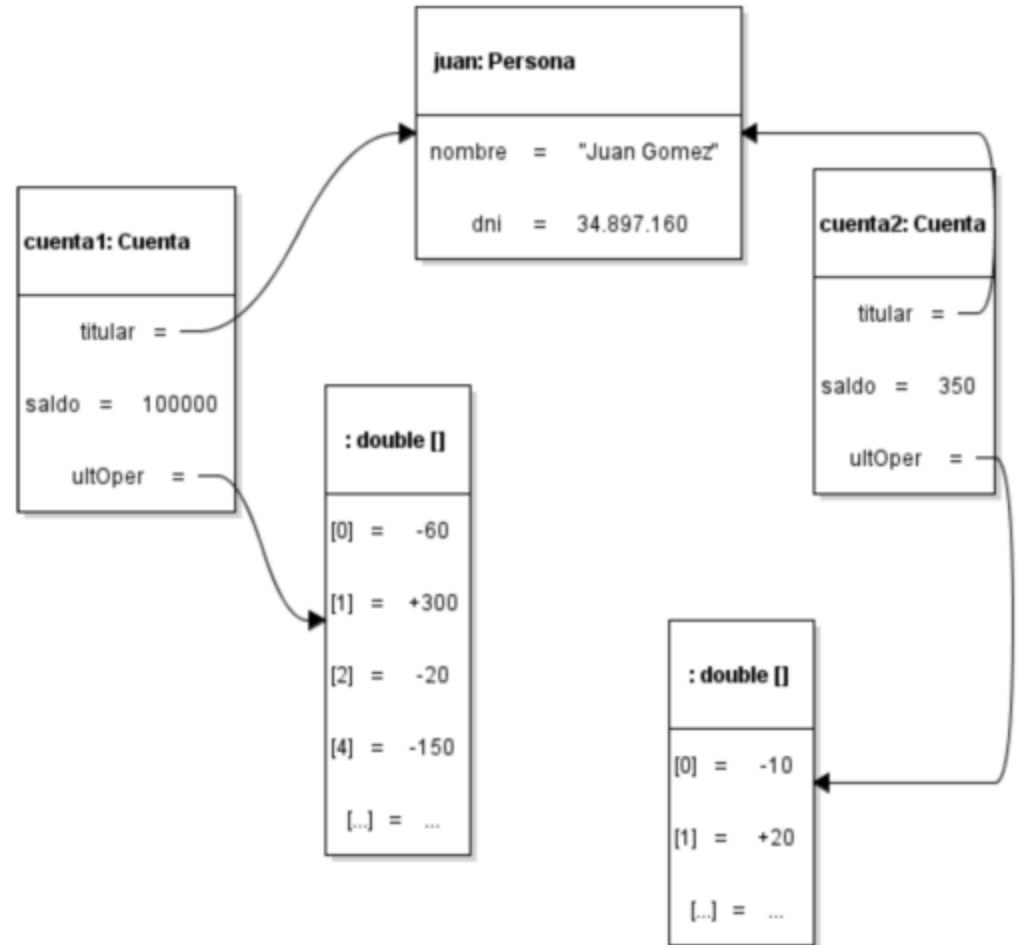
- Los objetos tienen un identificador que los diferencian de otros objetos (oid).
- Los identificadores de objetos almacenan la referencia en memoria donde están almacenados los objetos.
- Una referencia puede estar NO inicializada (null) o conectada (contiene la referencia (oid) a un objeto).

Semántica de referencia

- El manejo de los objetos por referencia tiene **ventajas**:
 - Compartición de objetos → **integridad referencial**
 - **Estructuras recursivas** → objetos que se referencian a si mismos.
 - Resulta más **eficiente** para el manejo de objetos complejos.
 - Los objetos se crean cuando se necesitan y no en su declaración.
 - Soporte para el **polimorfismo**.
- Pero tiene un **inconveniente**:
 - **Aliasing**

Semántica de referencia

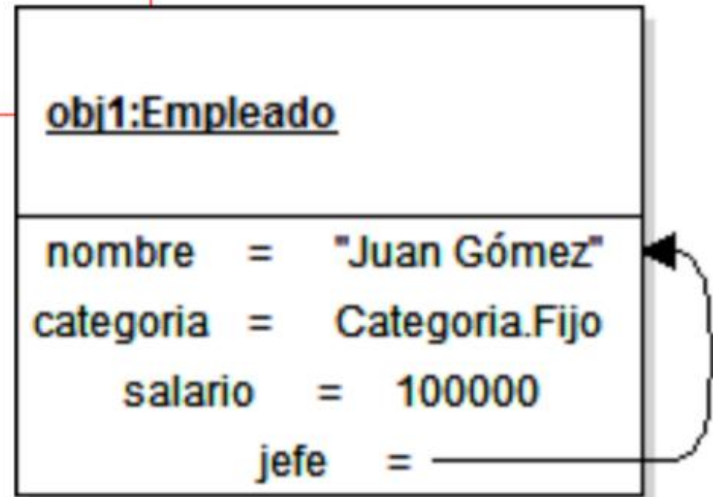
- **Compartición:**
Integridad referencial



Semántica de referencia

□ Estructuras recursivas

```
public class Empleado {  
    private String nombre;  
    private Categoria categoria;  
    private double salario;  
    private Empleado jefe;  
    ...  
}
```

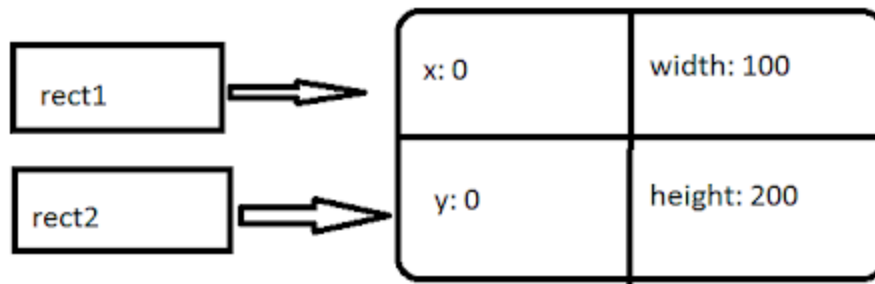


Semántica de referencia

- **Aliasing:** es el fenómeno que se da cuando un objeto es referenciado por más de una variable. Es como si el objeto tuviera más de un nombre (alias).
- Cuando se da el aliasing cualquier cambio que hagamos sobre una variable afecta a la otra.

```
Rectangulo rect1 = new Rectangulo(0, 0, 100, 200);
```


```
Rectangulo rect2 = rect1;
```



Semántica de referencia

- **Aliasing** puede provocar problemas.

```
Cuenta cuenta1;  
Cuenta cuenta2;  
...  
  
double saldoCuenta1 = cuenta1.getSaldo();  
cuenta2 = cuenta1;  
cuenta2.reintegro(1000.0);  
  
// cuenta1.getSaldo() != saldoCuenta1 !!
```



Ahora cuenta2 apunta al
objeto cuenta1
Por tanto
cuenta1.getSaldo()
no es el saldo de cuenta1

Semántica de referencia

- Aliasing – **Ocultación de la información**
 - Hay que prestar atención a los métodos de acceso (get/set), ya que si un atributo de una clase es una referencia (objeto), al devolver la referencia al objeto **se compromete la integridad del objeto**.
 - Esto es importante tenerlo en cuenta cuando utilizamos la Composición o Clientela.
 - Ejemplo → método de consulta de las últimas operaciones de la cuenta

```
public class Cuenta {  
    ...  
    private final Persona titular;  
    private double[] ultimasOperaciones;  
  
    public Cuenta(Persona persona) {  
  
        titular = persona;  
        ultimasOperaciones = new double[20];  
    }  
  
    public Persona getTitular() {  
        return titular;  
    } ...  
}
```

```
public double[] getUltimasOperaciones()  
{  
    return ultimasOperaciones;  
}
```

¡Quien consulte las últimas operaciones tiene acceso al objeto array y podría modificarlo, afectando al estado del objeto Cuenta !

Semántica de referencia

- ❑ **Precaución:** cuando un método de una clase devuelva un objeto, que es un atributo, estamos ofreciendo una referencia a dicho atributo que, probablemente, hayamos definido en la clase como privado → **¡¡ Estamos volviendo a hacer público el atributo !!**
- ❑ Para evitar estas situaciones existen diferentes alternativas.
- ❑ **Recomendación** → cuando se devuelve la referencia a un objeto en un método **público** hay que valorar:
 - ❑ Si el objeto no puede ser modificado (objeto inmutable) como los String que **no** son modificables.
 - ❑ En caso de ser modificable, deberíamos crear y devolver **una copia del objeto** para la consulta.
 - ❑ Evitar devolver un atributo que sea un objeto, a no ser que se trate de un caso en que debe ser así.

Semántica de referencia

- En el momento de la **inicialización de los atributos** de una clase que utiliza **Composición**, se debe tener cuidado con las referencias a objetos que se pasan como parámetros para asignar contenido a los atributos de la clase.
- Es conveniente hacer una copia de estos objetos y utilizar dichas copias en lugar de utilizar la referencia que se ha pasado como parámetro, ya que el código cliente podría tener acceso a la parte interna de la clase sin necesidad de pasar por la interfaz de la clase (**volveríamos a abrir una puerta pública a algo que debería ser privado probablemente**) y ocasionar efectos colaterales.

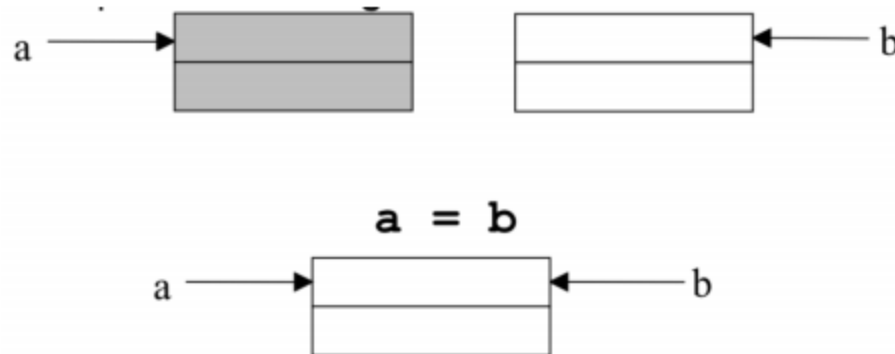
Semántica de referencia

- El aliasing se da también cuando pasamos una referencia como parámetro a un método. En ese caso tenemos dos referencias al mismo objeto.
- Cuando se da esta situación debemos valorar si es lo que se pretende o si se van a producir problemas.
- **Solución al Aliasing:**
 - Hacer una copia del objeto comprometido.
 - Utilizar el método `clone()` → lo veremos más adelante.

Semántica de referencia. Copia de objetos.

□ **Asignación vs copia:**

- Asignación de referencias (operador =)
- No implica la copia de los objetos, sino de los oids
- Se produce aliasing



□ ¿Cómo podemos **copiar los objetos**?

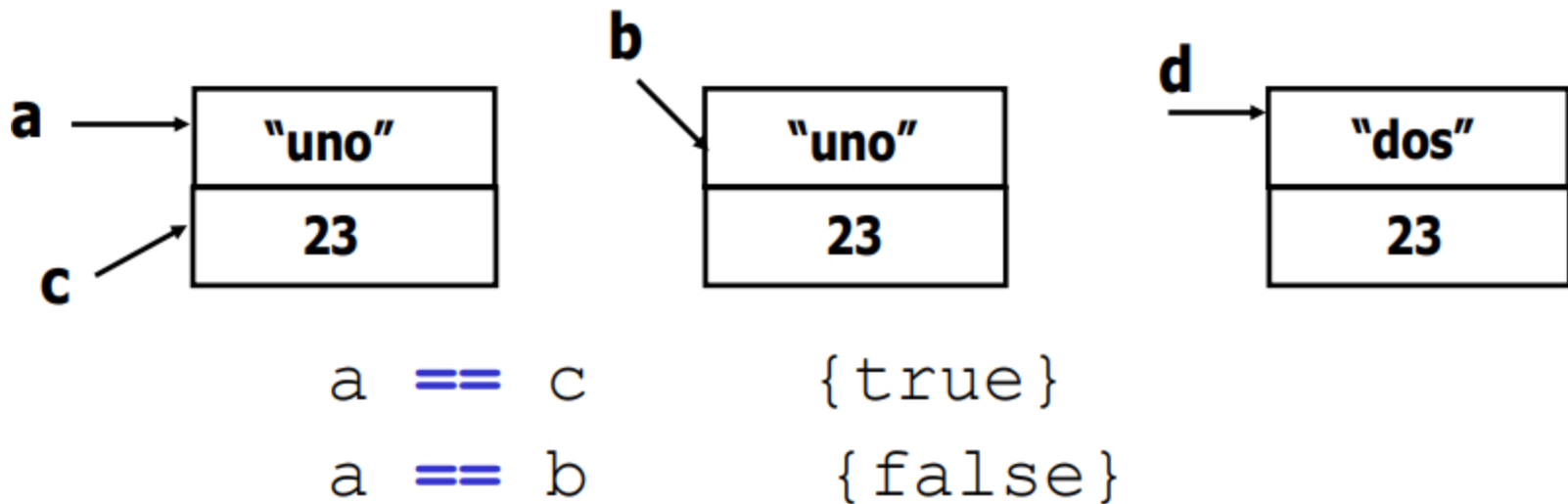
□ **Método “clone”**

- Ej. `copia = obj.clone();` (*) lo estudiaremos más adelante

Semántica de referencia. Identidad de objetos.

□ Identidad vs Igualdad:

- **Identidad:** dos referencias (oid) son iguales si apuntan al mismo objeto (utilizamos operador `==`).
- **Igualdad:** dos objetos se pueden comparar con el método **`equals`** (lo vemos en este tema más adelante)



Herencia

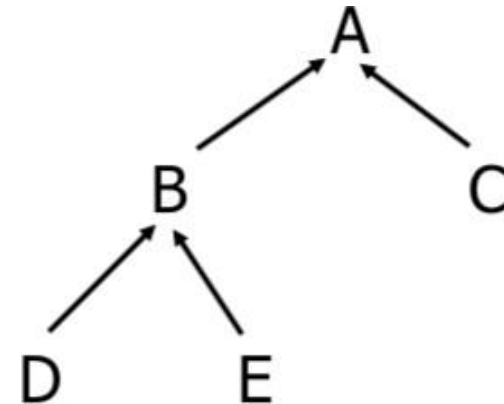
- Es uno de los **principios básicos** de la OO que permite la **reutilización** de código.
- Permite crear **nuevas clases a partir de otras existentes**.
 - Permite definir y utilizar relaciones entre clases.
 - Permite heredar y **redefinir** atributos y métodos.
- La herencia organiza las clases en **Jerarquías de clases**.
- Es consistente con el sistema de tipos.
- Toda clase que no indique explícitamente que hereda de una determinada clase, hereda **implícitamente** de la **clase raíz estándar de Java: Object**.



Herencia: Tipos de Herencia

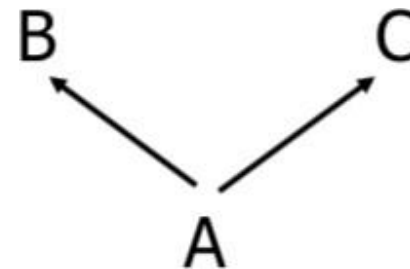
□ Herencia Simple:

- Una clase hereda de una sola clase
- **Java**, C#



□ Herencia Múltiple:

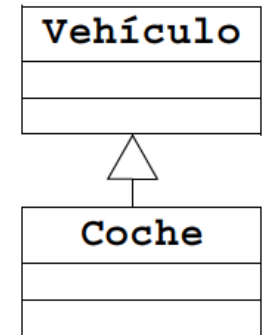
- Una clase hereda de varias clases
- C++



Herencia en Java

- Nomenclatura:

Clase original	Superclase	Padre	Vehículo
Clase extendida	Subclase	Hija	Coche



- Al extender una clase:

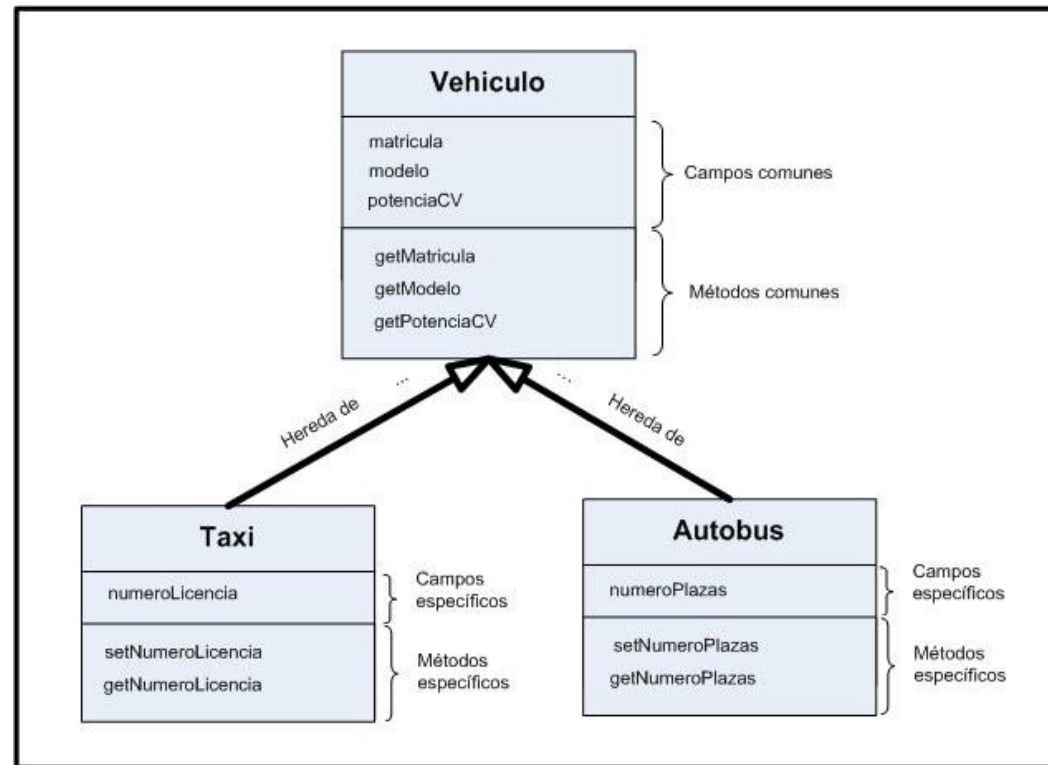
- se **heredan** todos los atributos y métodos de la clase base, teniendo en cuenta los modificadores de acceso.
- se pueden **añadir** nuevos atributos y métodos.

- **Sintaxis:** en Java se utiliza la palabra reservada **extends**

- **Es una relación de tipo “Es un”**

Herencia en Java

- La clase extendida puede:
 - **añadir** nuevos **atributos**
 - **añadir** nuevos **métodos**
 - **redefinir métodos/atributos** heredados (ya sea refinar o reemplazar)
 - **no hacer nada** y heredar tal y como está en la clase base.
- En general, las adaptaciones dependen del lenguaje OO.



- La herencia no permite acceder a **miembros privados** de la superclase ni **sobreescribirlos**.

Herencia en Java

❑ Redefinición de miembros:

- ❑ Se denomina **redefinición** o **superposición** al hecho de modificar el contenido de un método de la clase base.
 - ❑ Para invocar al método original de la clase base desde el método redefinido se ha de utilizar la palabra “super”: (*)
super.nombreMetodo().
- ❑ Los **métodos redefinidos solo** pueden **ampliar** la **accesibilidad** del método original, nunca restringirla. Si un método en la superclase es definido como protected o accesibilidad paquete, podrá ser redefinido como public en la clase derivada, pero no al contrario.
- ❑ **Los métodos estáticos no pueden ser sobreescritos.**
- ❑ Si una subclase sobreescrive un atributo de la clase base, lo reemplaza, siendo necesario utilizar la palabra reservada **super** para poder acceder al atributo de la clase base siempre que la visibilidad lo permita (si es private no es posible).

Herencia en Java

- **Redefinir vs Sobrecargar:**

- Si se declara un método con el mismo nombre que en la clase base pero con distintos parámetros, se sobrecarga el método, no se redefine.

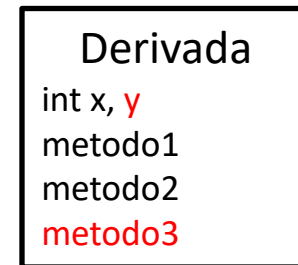
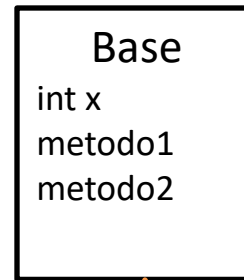
- **Nota:**

- Una subclase que redefine un método heredado, no tiene acceso a su propia versión y a la versión del método de la superclase; pero desde el método redefinido podemos acceder al método original con `super`.

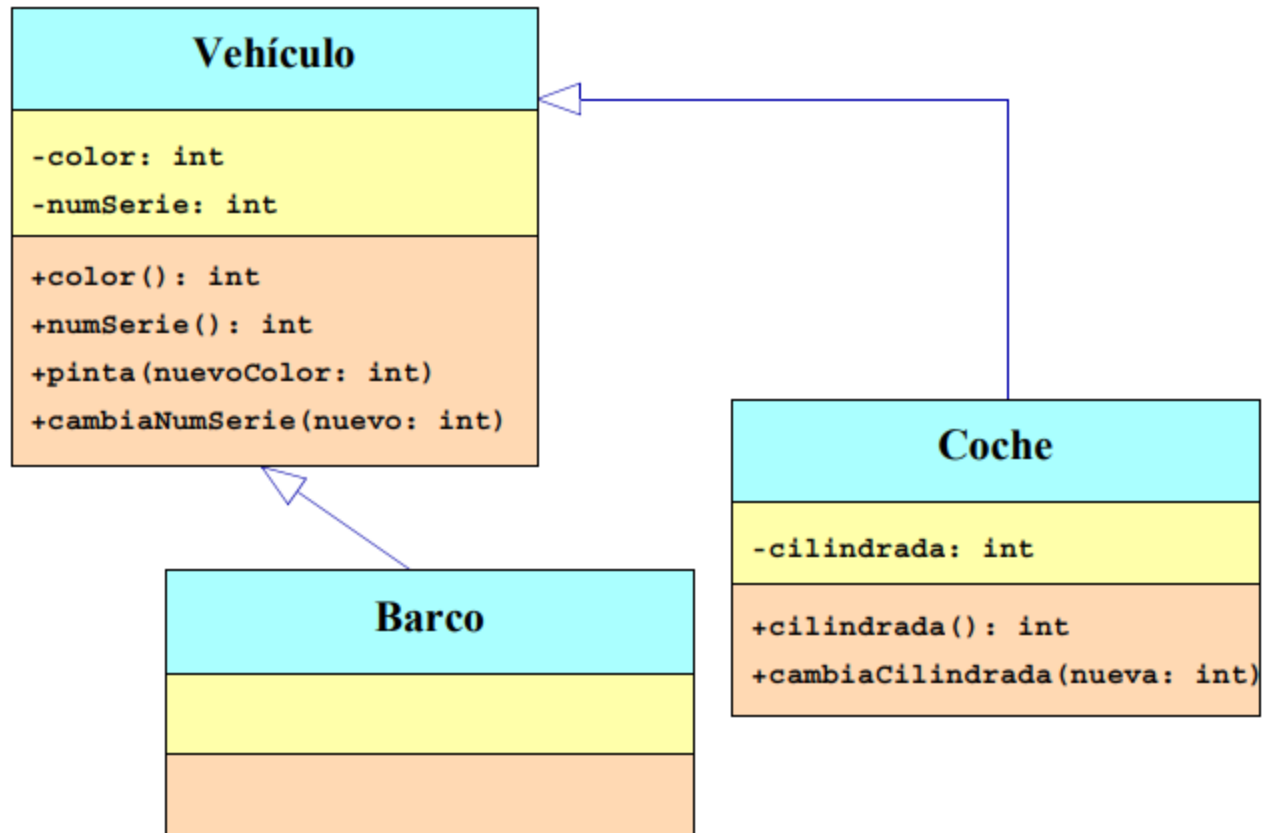
Herencia en Java

```
class Base {  
    int x;  
    Base(int x){this.x=x;}  
    void metodo1(int a){ ...}  
    void metodo2(int a){ ...}  
}
```

```
class Derivada extends Base{  
    int y;  
    public Derivada(int x, int y) {  
        super(x);  
        this.y=y;  
    }  
    void metodo3(int a){ ...}  
}
```

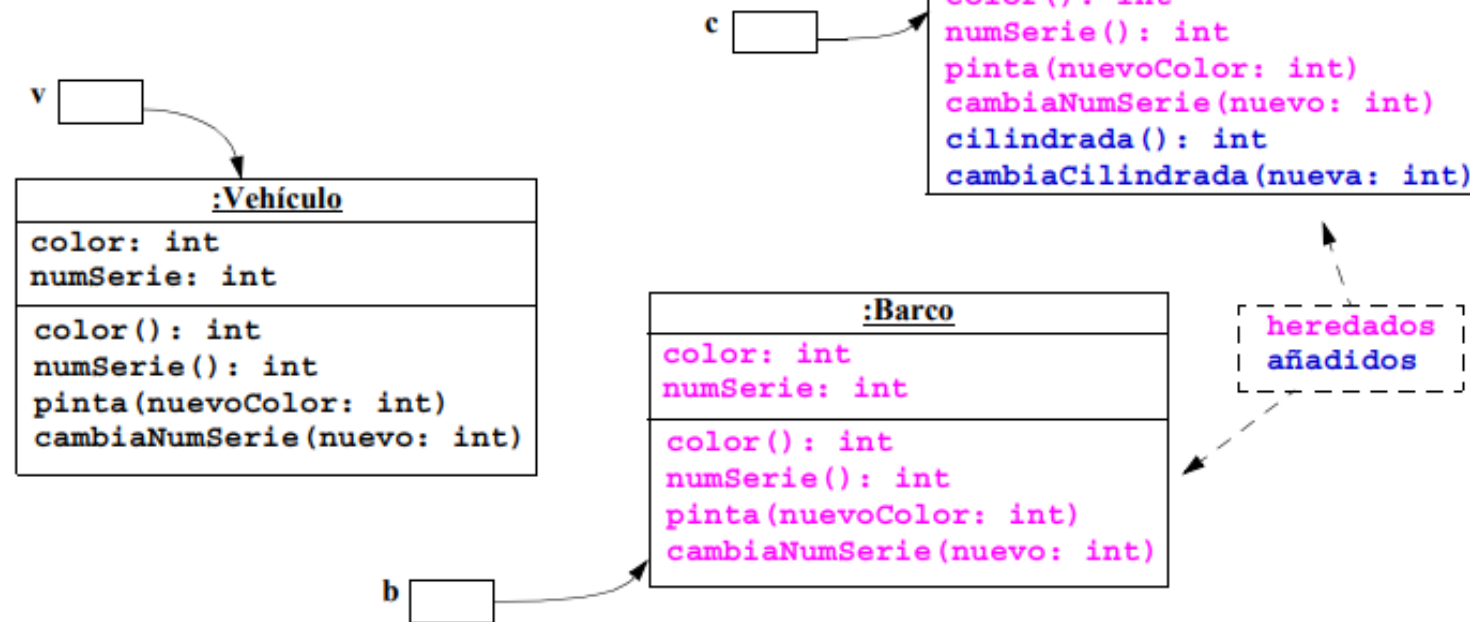


Herencia en Java



Herencia en Java

```
Vehículo v = new Vehículo();  
Coche c = new Coche();  
Barco b = new Barco();
```



Herencia y Constructores

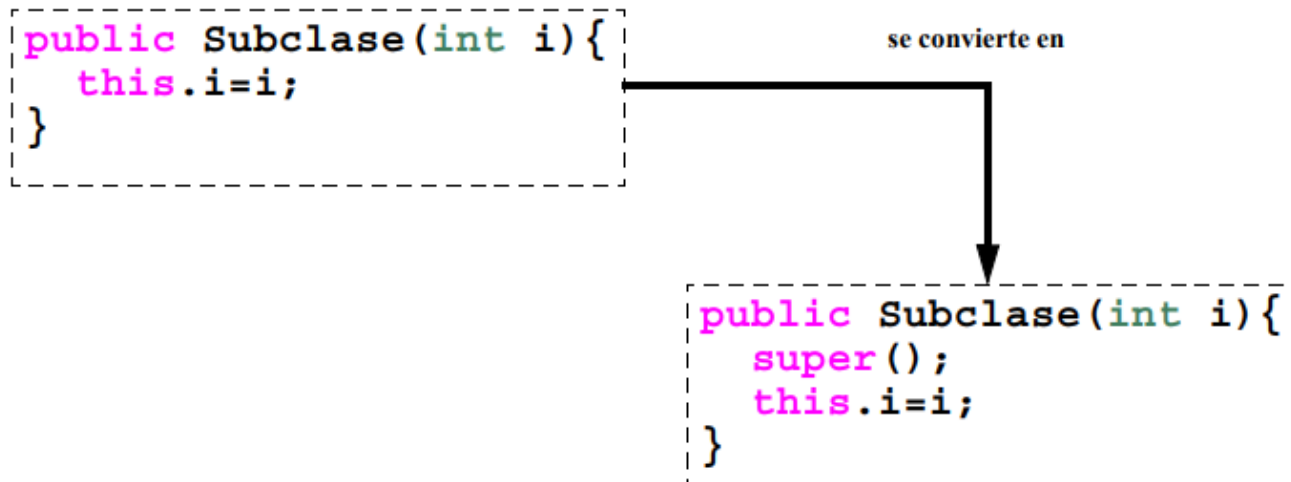
- Los **constructores no se heredan**
 - las subclases deben definir su propio constructor o constructores.
- Si la clase base no tiene constructor o tiene un constructor por defecto, no se necesita invocarlo explícitamente desde la subclase.
- Si es **necesario inicializar atributos de la superclase**, se puede invocar al constructor de la superclase desde el constructor de la subclase con **super(...)**.

```
/** constructor de una subclase */  
public Subclase(parámetros...) {  
    // invoca el constructor de la superclase  
    super(parámetros para la superclase);  
    // inicializa sus atributos  
    ...  
}
```

- **¡Ojo!** La llamada a super **debe ser la primera instrucción** del constructor de la subclase.

Herencia y Constructores

- Si desde el constructor de la subclase no se llama al constructor de la superclase:
 - el compilador invocará al constructor por defecto de la superclase, si es posible.
 - Si la superclase no dispone de un constructor por defecto se producirá un **error de compilación**.



Herencia. Redefinir métodos.

- Una subclase puede **redefinir** (“override”) un método heredado.
 - **¡Ojo!** Si se cambia el número o tipo de los parámetros del método heredado **NO estamos redefiniendo**, sino sobrecargando el método.
- Es conveniente indicarlo con la anotación **@Override** para indicar al compilador que estamos redefiniendo → permite detectar errores.

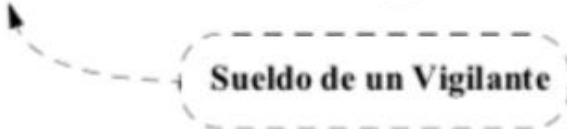
```
@Override  
public String toString()  
    ...  
}
```

Gracias a la anotación **@Override**, el compilador nos informa de que `toString()` NO redefine ningún método de la superclase

Herencia. Invocar métodos de la superclase.

- A veces es necesario invocar desde el método redefinido al método definido en la superclase usando “**super**”:
 - **super.nombreMetodo(parámetros);**
 - Hace referencia a la superclase del objeto actual

```
public class VigilanteNocturno extends Vigilante {  
    ...  
  
    @Override  
    public float sueldo() {  
        return super.sueldo() + PLUS_NOCTURNIDAD;  
    }  
}
```



Herencia en Java

```
class Base {  
    int x=3, y=5;  
    Base(int x){this.x=x;}  
    void metodo1(int a){}  
    void metodo2(int a){}  
    void metodo3(int a){}  
}
```

```
class Derivada extends Base{  
    int x = 7, z = 9;  
    public Derivada() {  
        super(7); //Se invoca al constructor de la clase base  
                //en la clase Base recibe un parámetro  
    }  
    void metodo1(int b){ } //Sobreescribe el método  
    void metodo2(String s){ } //Sobrecarga el método  
    void metodo4() { //Añade un nuevo método  
        this.x=8;  
        super.x=9;  
        super.metodo1(6); //Invoca al método de la clase Base  
        metodo2(5); //Invoca al método 2 de la clase Base  
    }  
}
```

Herencia en Java

```
public class Limpiador {  
    private String s = new String("Limpiador");  
  
    public void append(String a) { s += a; }  
    public void dilute() { append(" diluir()"); }  
    public void apply() { append(" aplicar()"); }  
    public void scrub() { append(" fregar()"); }  
    public void print() { System.out.println(s); }  
    public static void main(String[] args) {  
        Limpiador x = new Limpiador();  
        x.dilute();  
        x.apply();  
        x.scrub();  
        x.print();  
    }  
}
```

Que ambas clases tengan un método main permite probar cada módulo o clase. Posteriormente se puede eliminar.

```
public class Detergente extends Limpiador {  
    public void scrub() {  
        append(" Detergent.scrub()");  
        super.scrub(); // Invoca al constructor de la clase Base  
                        //scrub() simplemente seria una  
                        //llamada recursiva  
    }  
    //Añade métodos al interfaz:  
    public void foam() { append("espuma()"); }  
    //Probamos la clase derivada:  
    public static void main(String[] args) {  
        Detergente x = new Detergente();  
        x.dilute(); //Invoca al método de la clase Base  
        x.apply(); //Invoca al método de la clase Base  
        x.scrub();  
        x.foam();  
        x.print();  
        System.out.println("Probando la clase base:");  
    }  
}
```

Herencia. Modificador “protected”

- Como ya vimos, este modificador declara un miembro accesible para cualquier clase del mismo paquete o subclases desde cualquier paquete.
- En general, **definir atributos protected en Java NO es una buena práctica de programación:**
 - El atributo sería accesible desde cualquier subclase → **complica el mantenimiento de la superclase.**

UnaClase	
+	atrPúblico
-	atrPrivado
~	atrPaquete
#	atrProtegido
+	metPúblico
-	metPrivado
~	metPaquete
#	metProtegido

Herencia. Modificador “protected”

- Uso recomendado del modificador de acceso protected :
 - regla general: todos los campos de una clase son privados
 - se proporcionan métodos públicos para leer y/o cambiar los campos (pero sólo cuando sea necesario)
 - en el caso de que se desee que un campo sólo pueda ser leído y/o cambiado por las subclases se hacen métodos protected.

```
public class Superclase {  
    private int atributo; // atributo privado  
    // método para leer (público)  
    public int atributo() {  
        return atributo;  
    }  
    // método para cambiar (sólo para las subclases)  
    protected void cambiaAtributo(int a) {  
        atributo = a;  
    }  
}
```


Herencia en Java. Modificador “final”

Distintos contextos en los que puede aparecer el modificador final

Lugar	Función
Como modificador de clase.	La clase no puede tener subclases.
Como modificador de atributo.	El atributo no podrá ser modificado una vez que tome un valor. Sirve para definir constantes.
Como modificador al declarar un método	El método no podrá ser redefinido en una clase derivada.
Como modificador al declarar una variable referencia.	Una vez que la variable tome un valor referencia (un objeto), no se podrá cambiar. La variable siempre apuntará al mismo objeto, lo cual no quiere decir que ese objeto no pueda ser modificado internamente a través de sus métodos. Pero la variable no podrá apuntar a otro objeto diferente.
Como modificador en un parámetro de un método.	El valor del parámetro (ya sea un tipo primitivo o una referencia) no podrá modificarse dentro del código del método.

Herencia en Java. Modificador “final”

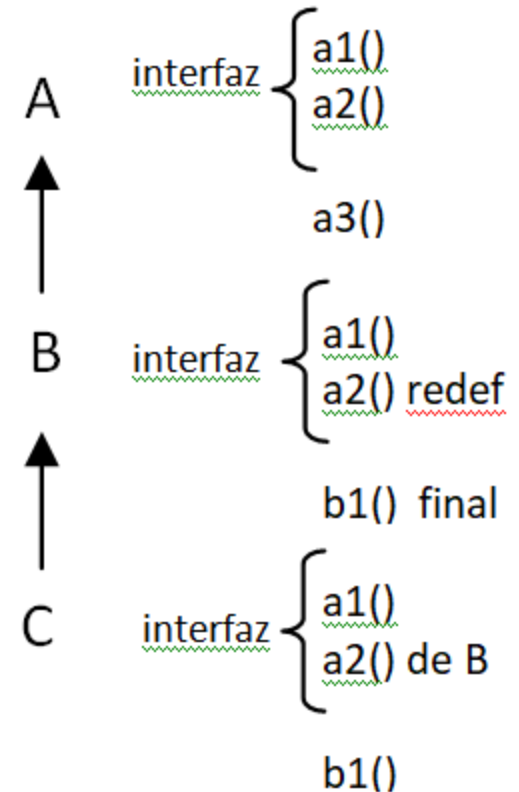
- ❑ La interfaz de una clase la forman los métodos **no privados** de la misma.
- ❑ Un **método private** de una clase es inaccesible por otras clases incluidas las clases derivadas, por tanto no puede ser redefinido → ¡Ojo! Porque el compilador no nos dará un error.
- ❑ Un **método final** no puede ser redefinido en una clase derivada → Obtendremos un error en tiempo de ejecución.

```
--- exec-maven-plugin:1.5.0:exec (default-cli) @ mavenproject1 ---  
Error: A JNI error has occurred, please check your installation and try again  
Exception in thread "main" java.lang.VerifyError: class C overrides final method bl.()V  
    at java.lang.ClassLoader.defineClass1(Native Method)  
    at java.lang.ClassLoader.defineClass(ClassLoader.java:756)  
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)  
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:468)  
    at java.net.URLClassLoader.access$100(URLClassLoader.java:74)  
    at java.net.URLClassLoader$1.run(URLClassLoader.java:369)  
    at java.net.URLClassLoader$1.run(URLClassLoader.java:363)
```

- ❑ Los métodos de una clase final son implícitamente final, ya que no se puede heredar de ella, tampoco se podrá redefinir sus métodos.

Herencia en Java. Modificador “final”

```
class A {  
    A(){ System.out.println("1. Constructor de A");}  
    public void a1(){System.out.println("Metodo a1() de A");}  
    public void a2(){System.out.println("Método a2() de A");}  
    private void a3(){System.out.println("Metodo a3() de A");}  
}  
class B extends A {  
    B(){System.out.println("2. Constructor de B");}  
    //No podemos redefinir y ocultar  
    // private void a2(){System.out.println("Método a2 de B");}  
    public void a2(){System.out.println("Método redefinido a2() en B");}  
    public final void b1(){System.out.println("Método b1() de B");}  
}  
public class C extends B {  
    C(){System.out.println("3. Constructor de C");}  
    //No podemos redefinir un método final  
    // public final void b1() {System.out.println("Redefino b1() en C y bloqueo")}  
    public static void main(String[] args) {  
        C objC = new C();  
        objC.a1(); //heredado de A  
        objC.a2(); //heredado de B y redefinido  
        objC.b1(); //heredado de B y no se puede redefinir en C  
    }  
}
```



Argumentos final

- Java permite que los argumentos de un método sean final. Esto significa que dentro del método no se puede cambiar el argumento, ni su valor ni a donde apunta, en caso de ser una referencia a un objeto.
- ¡Ojo! El objeto referenciado si puede ser modificado.

```
public static void metodo(final Cuenta c){  
    Cuenta c2 = new Cuenta("Pedro Pérez");  
    //c = c2;    Esto nos daría un error de compilación  
    c.setSaldo(1000);    //el objeto referenciado por c si es  
                        //modificable!!!!  
}
```

Clases y métodos final

Distintos contextos en los que puede aparecer el modificador final

Lugar

Función

Como modificador de clase.

La clase no puede tener subclases.

Como modificador de atributo.

El atributo no podrá ser modificado una vez que tome un valor. Sirve para definir constantes.

Como modificador al declarar un método

El método no podrá ser redefinido en una clase derivada.

Como modificador al declarar una variable referencia.

Una vez que la variable tome un valor referencia (un objeto), no se podrá cambiar. La variable siempre apuntará al mismo objeto, lo cual no quiere decir que ese objeto no pueda ser modificado internamente a través de sus métodos. Pero la variable no podrá apuntar a otro objeto diferente.

Como modificador en un parámetro de un método.

El valor del parámetro (ya sea un tipo primitivo o una referencia) no podrá modificarse dentro del código del método.

Reconocer la herencia

- **Especialización:**

- Se detecta que una clase es un caso especial de otra.
- Ejemplo: Rectángulo es un tipo de Polígono.

- **Generalización**

- Se detectan dos clases con características en común y se crea una clase padre con esas características.
- Ejemplo: Libro, Revista: Publicación.

- No hay receta mágica para crear buenas jerarquías de herencia:

- Conocimientos teóricos, práctica, etc.

Buena práctica de programación:

- utilizar **@Override** en los métodos redefinidos

Ventajas y desventajas de la Herencia

□ **Ventajas:**

□ Mejora el diseño:

- permite modelar relaciones de tipo “**es un**” que se dan en los problemas que se pretenden resolver.
- Permite la **reutilización del código**: los métodos de la clase padre se reutilizan en las clases hijas.
- Facilita la **extensión de las aplicaciones**: añadir una nueva subclase no requiere modificar ninguna otra clase de nuestro diseño.

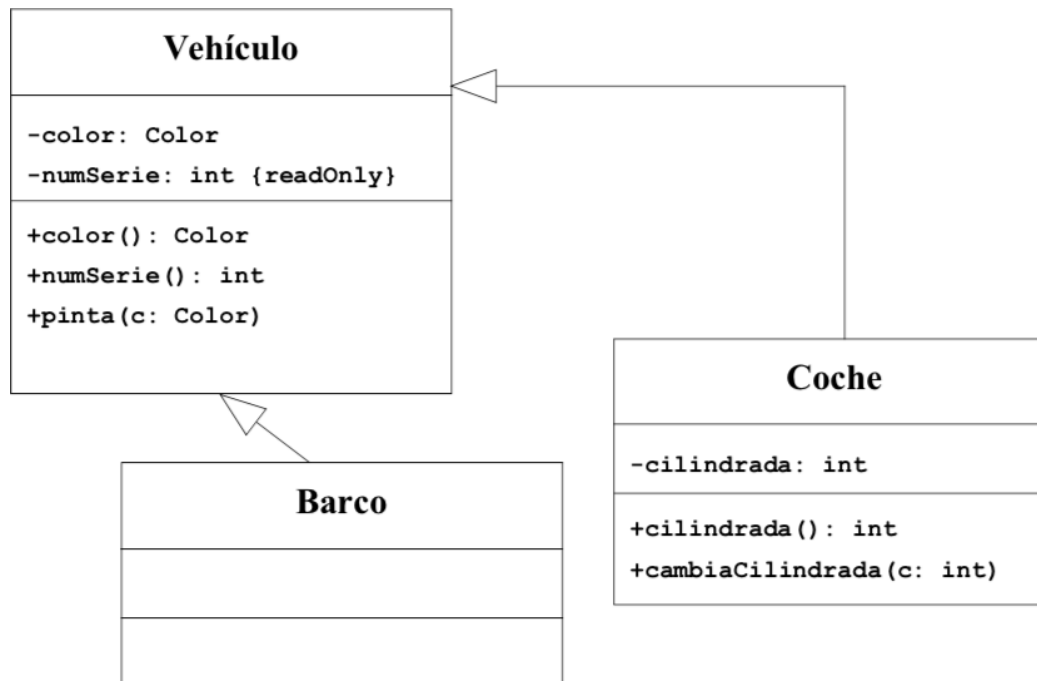
□ **Principal desventaja:**

□ Aumenta el **acoplamiento**:

- las subclases están íntimamente acopladas con la superclase.

Ejercicio de clase.

- Implementa la siguiente jerarquía de clases:



Ejercicio de clase. Solución

Clase Vehículo:

```
/**
 * Clase que representa un vehículo cualquiera
 */
public class Vehículo
{
    // colores de los que se puede pintar un vehículo
    public static enum Color {ROJO, VERDE, AZUL}

    // atributos
    private Color color;
    private final int numSerie;

    /**
     * Construye un vehículo
     * @param color color del vehículo
     * @param numSerie número de serie del vehículo
     */
    public Vehículo(Color color, int numSerie)
    {
        this.color=color;
        this.numSerie=numSerie;
    }
}
```

```
public String toString() { return "Vehiculo -> color =" +color+ "Número de serie " +numSerie;}

}
```

```
/**
 * Retorna el color del vehículo
 * @return color del vehículo
 */
public Color color()
{
    return color;
}

/**
 * Retorna el numero de serie del vehículo
 * @return numero de serie del vehículo
 */
public int numSerie()
{
    return numSerie;
}

/**
 * Pinta el vehículo de un color
 * @param nuevoColor color con el que pintar el vehículo
 */
public void pinta(Color c)
{
    color = c;
}
}
```

Ejercicio de clase. Solución

```
public class Coche extends Vehículo
{
    // cilindrada del coche
    private int cilindrada;

    /** Retorna la cilindrada del coche ... */
    public int cilindrada(){
        return cilindrada;
    }

    /** Cambia la cilindrada del coche ... */
    public void cambiaCilindrada(int c) {
        this.cilindrada=c;
    }

    @Override
    public String toString() {
        return super.toString()+
            ", cilindrada= " + cilindrada;
    }
}
```

```
public class Barco extends Vehículo {

    /**
     * Constructor al que le pasamos el color y
     * el numero de serie
     */
    public Barco(int color, int numSerie) {
        super(color, numSerie);
    }
}
```

Polimorfismo

- El **polimorfismo** es uno de los grandes pilares de la POO junto a la encapsulación y la herencia:
 - La **encapsulación** permite **agrupar características** (atributos) y **comportamientos** (métodos) **dentro de una misma unidad** (clase), pudiendo darles una visibilidad mayor o menor y permitiendo así separar la interfaz de la implementación.
 - La **herencia** proporciona:
 - un mecanismo para que las clases derivadas dispongan de la **interfaz** de la clase base.
 - la posibilidad de tratar a los objetos como pertenecientes a una jerarquía de clases, fundamental para el **polimorfismo**.
- El **polimorfismo** permite mejorar la **organización** y **legibilidad** del código así como la posibilidad de desarrollar **aplicaciones fáciles de ampliar** a la hora de añadir nuevas funcionalidades.

Polimorfismo en Java

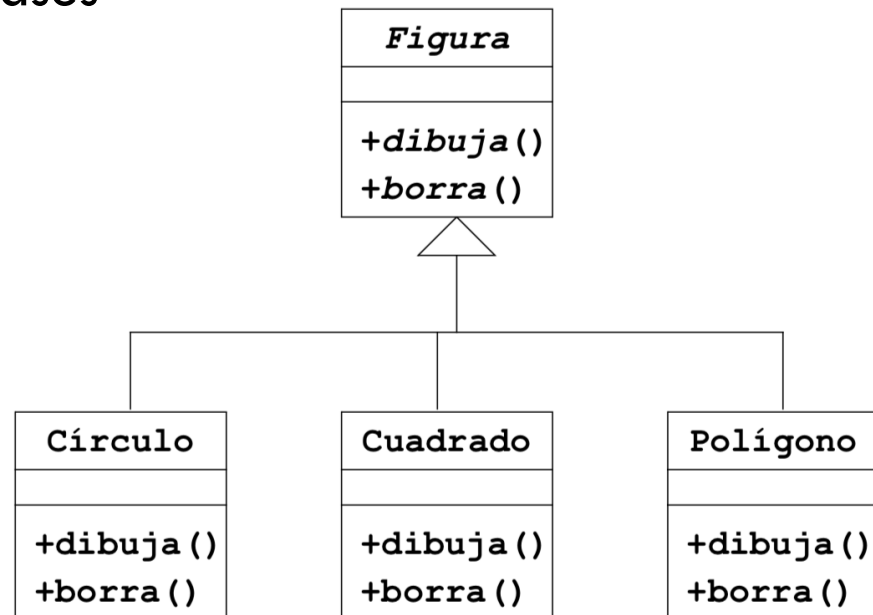
- **Polimorfismo viene de “múltiples formas”.**
- Las operaciones polimórficas son aquellas que hacen funciones similares con objetos diferentes:
- Ejemplo:

Dada la clase *Figura* y sus subclases

- Círculo
- Cuadrado
- Polígono

Todas ellas con las operaciones

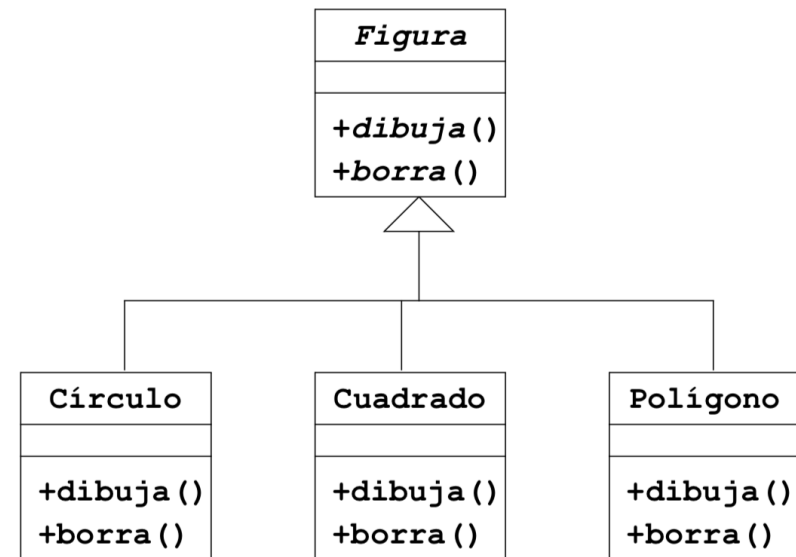
- `dibuja()`
- `borra()`



Polimorfismo en Java

- Nos gustaría hacer la operación polimórfica “mueveFigura()” que opere correctamente con cualquier clase de figura

```
mueveFigura  
  borra  
  dibuja en la nueva posición
```



- Esta operación polimórfica debería:
 - llamar a las operaciones `borra()` y `dibuja()` del **Círculo** cuando la figura sea un círculo
 - llamar a las operaciones `borra()` y `dibuja()` del **Cuadrado** cuando la figura sea un cuadrado
 - etc.

Polimorfismo en Java

- El polimorfismo consiste en la capacidad de poder utilizar una referencia a un objeto de una determinada clase como si fuera de otra clase (subclase).
- El polimorfismo permite escribir programas que procesen objetos de clases que formen parte de la misma jerarquía como si todos fueran objetos de sus superclases.

Polimorfismo en Java

- El polimorfismo en Java consiste en dos propiedades:
1. Una referencia a una superclase puede apuntar a un objeto de cualquiera de sus subclases (**no al revés**):

```
Vehículo v1=new Coche(Vehiculo.rojo,12345,2000);  
Vehículo v2=new Barco(Vehiculo.azul,2345);
```

2. La **operación a ejecutar** se selecciona en base a la **clase del objeto**, no a la de la referencia:

```
v1.toString() ← usa el método de la clase Coche, puesto que v1 es un coche  
v2.toString() ← usa el método de la clase Barco, puesto que v2 es un barco
```

Polimorfismo en Java

- Gracias a estas dos propiedades, el método “moverFigura()” sería...

```
public void mueveFigura(Figura f, Posición pos){  
    f.borra();  
    f.dibuja(pos);           //Ligadura dinámica o tardía  
}
```

- Y podría invocarse de la siguiente forma...

```
Círculo c = new Círculo(...);  
Polígono p = new Polígono(...);  
mueveFigura(c, pos); //Conversión hacia arriba  
mueveFigura(p, pos);
```

- Gracias a la primera propiedad, el parámetro “f” puede referirse a cualquier subclase de Figura. Si se amplía la jerarquía de Figura, el método mueveFigura no se ve afectado.
- Gracias a la segunda propiedad, en “mueveFigura()” se llama a las operaciones “borra()” y “dibuja()” apropiadas.

Polimorfismo en Java

- **Ligadura estática** es la vinculación que se produce en la llamada a un método con la clase a la que pertenece ese método:
 - se produce en tiempo de compilación.
 - se da en los lenguajes no orientados a objetos.
- **Ligadura dinámica o vinculación tardía** es la vinculación que se produce en la llamada a un método con la clase a la que pertenece ese método en tiempo de ejecución.
 - La ligadura dinámica hace posible que sea el tipo del objeto y no el tipo de la referencia lo que **determine** la versión del método que se ejecutará.

Polimorfismo en Java

□ Limitaciones de la ligadura dinámica:

- No se puede acceder a los miembros específicos de la subclase a través de una referencia a una superclase.
- Solo se pueden usar los miembros declarados en la superclase, aunque la definición que finalmente se utilice en su ejecución sea de la subclase.
- En el ejemplo de la jerarquía de Figura, el polimorfismo permite declarar variables de tipo Figura y más tarde referenciar con ellas a objetos de tipo Circulo o Cuadrado, pero no deberíamos intentar acceder con esta variable a métodos que sean específicos de la clase Circulo o de la clase Cuadrado, tan solo a métodos declarados en Figura.
- Se debe proporcionar en las subclases constructores compatibles con alguno de la superclase que permita inicializar todos los atributos incluidos los heredados.

Polimorfismo en Java

- El lenguaje permite que una referencia de una superclase pueda apuntar a un objeto de cualquiera de sus subclases, **pero no al revés...**

```
Vehículo v = new Coche(...); // permitido  
Coche c = new Vehículo(...); // ¡NO permitido!
```

Justificación

- Un coche es un Vehículo: cualquier operación de la clase Vehículo existe (sobrescrita o no) en la clase Coche

```
v.cualquierOperación(...); // siempre correcto
```

- Pero un vehículo no es un Coche: sería un error tratar de invocar la operación:

```
c.cilindrada(); // ERROR: cilindrada() no  
                // existe para un vehículo
```

- Por esa razón el lenguaje lo prohíbe.

Ejemplo de polimorfismo

Ejemplo de polimorfismo y ligadura dinámica:

La clase Instrumento representa un instrumento musical genérico. Las clases Flauta y Piano representan dos instrumentos específicos. Todas las clases tienen un método tocarNota específico de cada subclase.

```
public abstract class Instrumento {  
    public void tocarNota (String nota) {  
        System.out.printf ("Instrumento: tocar nota %s.\n", nota);  
    }  
}
```

```
public class Flauta extends Instrumento {  
    @Override  
    public void tocarNota (String nota) {  
        System.out.printf ("Flauta: tocar nota %s.\n", nota);  
    }  
}
```

```
public class Piano extends Instrumento {  
    @Override  
    public void tocarNota (String nota) {  
        System.out.printf ("Piano: tocar nota %s.\n", nota);  
    }  
}
```

Ejemplo polimorfismo

Ejemplo de polimorfismo y ligadura dinámica:

A la hora de declarar una variable o referencia a un objeto de tipo instrumento, utilizaremos la superclase:

```
Instrumento instrumentol; // Ejemplo de objeto polimórfico (podrá ser Piano o Flauta)
```

Sin embargo, a la hora de instanciar el objeto, utilizaremos el constructor de alguna de las subclases:

```
if (<condición>) {  
    // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Piano)  
    instrumentol= new Piano ();  
}  
else if (<condición>) {  
    // Ejemplo de objeto polimórfico (en este caso va adquirir forma de Flauta)  
    instrumentol= new Flauta ();  
} else {  
    ...  
}
```

A la hora de invocar al método tocarNota, no sabremos qué versión (de qué subclase) se ejecutará, pues dependerá del tipo de objeto (subclase) que se haya instanciado. Se estará utilizando la **ligadura dinámica**:

```
// Interpretamos una nota con el objeto instrumentol  
// No sabemos si se ejecutará el método tocarNota de Piano o de Flauta  
// (dependerá de la ejecución)  
instrumentol.tocarNota ("do"); // Ejemplo de ligadura dinámica (tiempo de ejecución)
```

Casting (conversión de referencias)

- El **Casting** se puede dar entre clases compatibles, por ejemplo, entre una subclase y su superclase.
- Con una referencia de tipo Vehículo, un coche se ve desde el punto de vista de un Vehículo.
 - 'v' solo da acceso a los atributos y métodos definidos en la clase Vehículo.
- Con una referencia de tipo Coche se podrían acceder a todos los atributos y métodos de Coche y de Vehículo.
- Es posible convertir referencias (hacer un casting):

```
Vehículo v=new Coche(...);  
Coche c=(Coche)v;  
  
v.cilindrada(); // ¡ERROR!  
c.cilindrada(); // correcto
```

- A través de "v" le vemos como un Vehículo (y por tanto sólo podemos invocar métodos definidos en la clase Vehículo)
- A través de "c" le vemos como un Coche (y podemos invocar cualquiera de los métodos de esa clase y de sus superclases)

Casting y el operador instanceof

- Hacer una conversión de tipos incorrecta produce una excepción **ClassCastException** en tiempo de ejecución:

```
Vehículo v=new Vehículo(...);  
Coche c=(Coche) v; ← lanza ClassCastException en tiempo de ejecución
```

- Java proporciona el operador **instanceof** que permite conocer la clase de un objeto:

```
if (v instanceof Coche) {  
    Coche c=(Coche) v; ← NUNCA lanza ClassCastException (por que es  
    ... seguro que v es un coche)  
}
```

- “**v instanceof Coche**” retorna true si v apunta a un objeto de la clase Coche o de cualquiera de sus (posibles) subclases

- Un uso excesivo del operador “instanceof”...
 - Elimina las ventajas del polimorfismo
 - Revela un diseño incorrecto de las jerarquías de clases

Operador instanceof

```
public class Circulo{  
    private int radio;  
    Circulo(int r){ this.radio=r }  
}  
@Override  
public boolean equals(Object obj){  
    if(obj instanceof Circulo){  
        Circulo c = (Circulo) obj;  
        return (c.radio==this.radio);  
    } else return false;  
}
```

instanceof es un operador en Java para conocer de que tipo es un determinado objeto.

Casting

Salida: True, ya que el método que hemos redefinido comprueba si son iguales

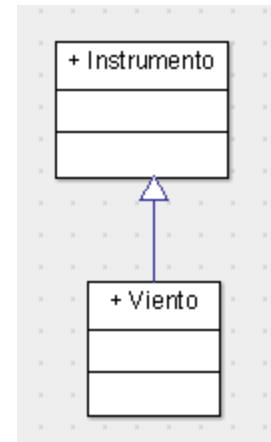
```
public static void main(String[] args) {  
    Circulo c1 = new Circulo(3);  
    Circulo c2 = new Circulo(3);  
    System.out.println("¿Son iguales?" + c1.equals(c2));  
}
```


Upcasting (conversión ascendente)

□ Veámoslo con un ejemplo

```
class Instrumento{
    public void play(){ .... }
    static void sonar (Instrumento i){
        i.play();
    }
}

public class Viento extends Instrumento{
    public static void main(String[] args) {
        Viento flauta = new Viento ();
        Instrumento.sonar(flauta);
    }
}
```



Invocamos al método sonar(Instrumento) y le pasamos una referencia a un objeto Viento
→ Al acto de convertir una referencia a un objeto Derivado en una referencia a un objeto Base se llama **Upcasting (Conversión ascendente)**

Estamos convirtiendo un tipo específico en un tipo más general sin una conversión explícita. Esto es posible ya que el tipo específico tiene al menos la misma estructura y comportamiento que el tipo más general.

Arrays de objetos de distintos tipos

- Gracias al polimorfismo es posible que un array o una colección contenga referencias a objetos de distinto tipo
 - superclase y subclases

Ejemplo: array de figuras

```
Figura[] figuras = new Figura[3];  
figuras[0] = new Círculo(...);  
figuras[1] = new Cuadrado(...);  
figuras[2] = new Polígono(...);
```

```
// borra todas las figuras  
for(int i=0; i<figuras.length; i++)  
    figuras[i].borra();
```

← Llama a la operación borra correspondiente a la clase del objeto

Arrays de objetos de distintos tipos

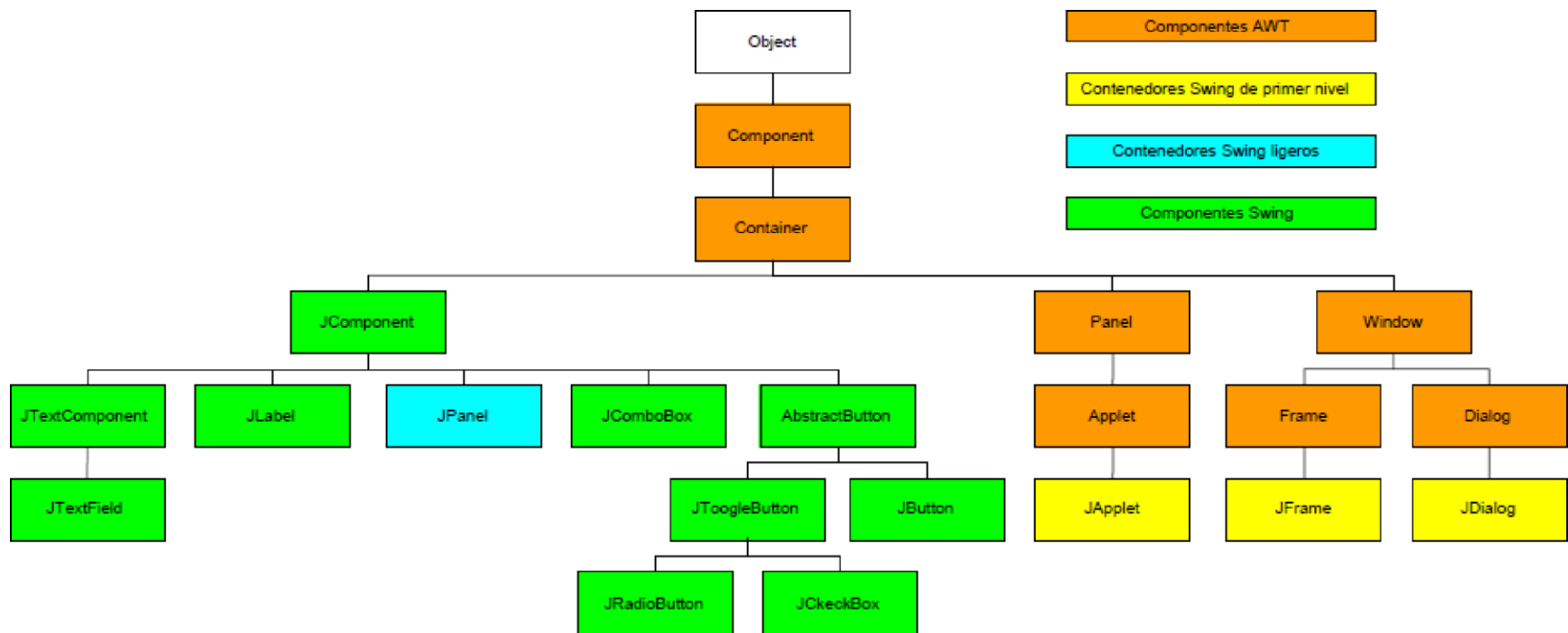
Ejemplo

Debemos hacer casting
para poder ejecutar
métodos
propios de Circulo o
Cuadrado

```
Figura[] lista = new Figura[5];
lista[0] = new Circulo(Color.BLUE, 5);
lista[1] = new Circulo(Color.YELLOW, 3);
lista[2] = new Cuadrado(Color.BLUE, 5);
lista[3] = new Cuadrado(Color.YELLOW, 3);
lista[4] = new Cuadrado(Color.ORANGE, 10);
for(int i = 0; i < lista.length; i++){
    lista[i].dibujar();
    System.out.println("AREA " + lista[i].area());
    System.out.println("PERIMETRO " + lista[i].perimetro());
    if(lista[i] instanceof Circulo){
        Circulo c1 = (Circulo) lista[i];
        System.out.println("Diametro " + c1.diametro());
    } else if(lista[i] instanceof Cuadrado){
        Cuadrado c1 = (Cuadrado) lista[i];
        System.out.println("Diagonal " + c1.diagonal());
    }
}
```

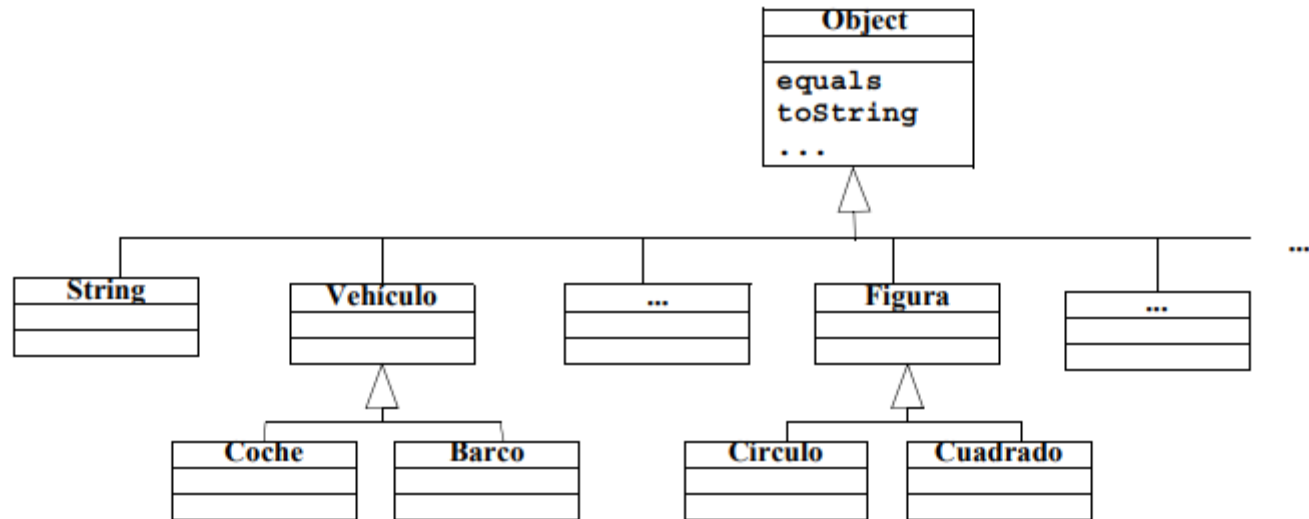
Jerarquías de clases

- La herencia puede aplicarse en sucesivos niveles, creando grandes **jerarquías de clases** (jerarquía Swing)



La Clase Object.

- En Java todas las clases heredan de la superclase **Object**.
 - Esta clase define estados y comportamientos básicos que deben tener todos los objetos.
 - Es como si el compilador añadiera “extends Object” a todas las clases de primer nivel.



La Clase Object.

Método	Sintaxis	Propósito
getClass	class _ getClass()	Obtiene la clase de un objeto en tiempo de ejecución
hashCode	int hashCode	Devuelve el código hash asociado con el objeto invocado
equals	boolean equals(Object obj)	Determina si un objeto es igual a otro
clone	Object clone()	Crea un nuevo objeto que es el mismo que el objeto que se está clonando
toString	String toString()	Devuelve una cadena que describe el objeto
notify	void notify()	Reanuda la ejecución de un hilo esperando en el objeto invocado
notifyAll	void notifyAll()	Reanuda la ejecución de todo el hilo esperando en el objeto invocado
wait	void wait(long timeout)	Espera en otro hilo de ejecución
wait	void wait(long timeout,int nanos)	Espera en otro hilo de ejecución
wait()	void wait()	Espera en otro hilo de ejecución
finalize	void finalize()	Determina si un objeto es reciclado (obsoleto por JDK9)


La Clase Object.

- Método **equals**: definido en la clase Object, se utiliza para saber si dos objetos son del mismo tipo y contienen los mismos datos. El método devuelve true si los objetos son iguales y false en caso contrario.
 - Este método se deberá **redefinir** en cada clase que lo quiera utilizar.
- Ej. Supongamos una clase Coordenada:

```
public class Circulo{  
    private int radio;  
    Circulo(int r){ this.radio =r; }  
}
```

```
public static void main(String[] args) {  
    Circulo c1 = new Circulo(3);  
    Circulo c2 = new Circulo(3);  
    System.out.println("¿Son iguales?" + c1.equals(c2));  
}
```

Salida: False, ya que por defecto equals compara referencias a objetos!!



La Clase Object.

```
public class Circulo{  
    private int radio;  
    Circulo(int r){ this.radio =r; }  
    @Override  
    public boolean equals(Object obj){  
        if(this.getClass() == obj.getClass()){  
            Circulo c = (Circulo) obj;  
            return (c.radio==this.radio);  
        } else return false;  
    }  
}
```

getClass() es un método de Object que devuelve el tipo del objeto

Casting

Salida: True, ya que el método que hemos redefinido comprueba si son iguales

```
public static void main(String[] args) {  
    Circulo c1 = new Circulo(3);  
    Circulo c2 = new Circulo(3);  
    System.out.println("¿Son iguales?" + c1.equals(c2));  
}
```


La Clase Object.

- ❑ Método **toString**: definido en la clase Object, devuelve un String con información del objeto.
 - ❑ Se debe **redefinir** en cualquier clase para que se pueda mostrar correctamente sus datos.
 - ❑ Supongamos una clase Circulo:

```
public class Circulo{  
    private int radio;  
    Circulo(int r){ this.radio =r; }  
}
```

```
public static void main(String[] args) {  
    Circulo c1 = new Circulo(3);  
    System.out.println("Datos del objeto" + c1.toString());  
}
```

Salida: Circulo@15db9742



La Clase Object.

```
public class Circulo{  
    private int radio;  
    Circulo(int r){ this.radio =r; }  
    @Override  
    public String toString(){  
        return "Radio = "+radio;  
    }  
}
```

```
public static void main(String[] args) {  
    Circulo c1 = new Circulo(3);  
    System.out.println("Datos del objeto" + c1);  
}
```

Salida: Radio = 3



Inicialización y carga de clases

- En Java, dado que el código de cada clase se compila por separado (.class) se simplifica el proceso de inicialización.
- En general, el código de una clase no se carga en memoria hasta que no se necesita por primera vez, ya sea porque se crea un objeto de la clase o porque la clase contiene métodos o campos static.
- **Orden de inicialización:** La máquina virtual **primero carga la clase y después instancia los objetos:**
 - **Al cargar la clase:**
 - Primero se carga todo lo **static**, bloques de código y atributos static en orden de aparición y una sola vez.
 - **Inicialización con Herencia:**
 - Si se trata de una clase derivada **primero se carga la clase base** de la que se derive, de forma recursiva, y se aplica el orden de carga de variables de clase especificado arriba.
 - Una vez cargadas las clases ascendientes, se carga la clase que se esta instanciando y se aplica el orden de carga de clases.
 - Luego, se cargan los campos no static de clases ascendientes recursivamente.
 - Después se ejecutan los constructores de clases ascendientes de forma recursiva.
 - Luego se inicializan los campos de la clase que se está instanciando.
 - Finalmente se ejecuta el constructor de la clase instanciada.

Inicialización y carga de clases

```
class Arte {
    Arte(){
System.out.println("1. Constructor de Arte");}
}
class Dibujo extends Arte {
    Dibujo(){
System.out.println("2. Constructor de Dibujo");}
}
public class DibujoAnimado extends Dibujo {
    DibujoAnimado(){
System.out.println("3. Constructor de Dibujo
Animado");
}
    public static void main(String[] args) {
        DibujoAnimado a = new DibujoAnimado();
    }
}
```

```
] --- exec-maven-plugin:1.5.0:exec (de
1. Constructor de Arte
2. Constructor de Dibujo
3. Constructor de Dibujo Animado
-----
```

- Los objetos creados a partir de clases que heredan de otras, contienen atributos heredados de las clases base, y deben ser correctamente inicializados. Por ello desde el constructor de la clase derivada se debe invocar al constructor de la clase base.
- La JVM invoca automáticamente al constructor desde las clases base de forma recursiva, así la clase base es inicializada antes que los constructores de las clases derivadas puedan acceder a ella.
- Si no se añade un constructor para la clase DibujoAnimado, la JVM sintetizaría uno por defecto que invocaría al constructor de la clase base.

Inicialización y carga de clases

```
class Juego{
    Juego(int i){ System.out.println("Constructor de Juego");
    }
}

class JuegoMesa extends Juego{
    JuegoMesa(int i){
        super(i);
        System.out.println("Constructor de JuegoMesa");
    }
}

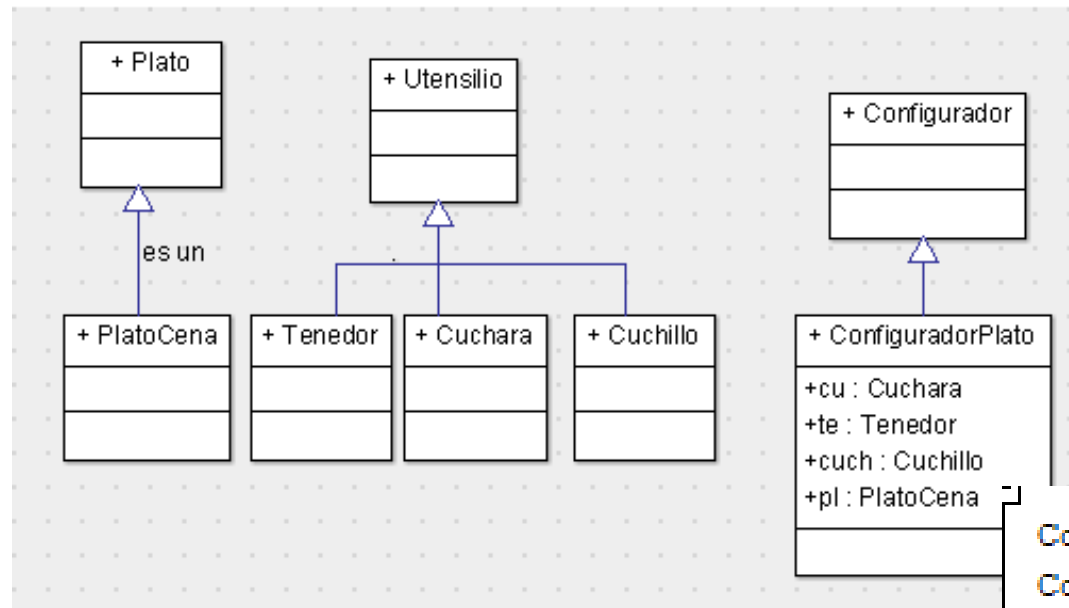
public class Ajedrez extends JuegoMesa{
    Ajedrez(){
        super(11);
        System.out.println("Constructor de Ajedrez");
    }
    public static void main(String[] args) {
        Ajedrez a = new Ajedrez();
    }
}
```

- En este ejemplo vemos como en el constructor de la clase derivada se puede invocar al constructor de la clase base utilizando **super()**. Esta invocación debe ser la primera instrucción del constructor de la clase derivada.

Combinando Herencia y Composición

- Cuando mezclamos Herencia y Composición hemos de tener en cuenta que si bien el compilador fuerza la inicialización de las clases base al comienzo del constructor de las clases derivadas, no se asegura la inicialización de objetos miembros no heredados, por lo que se debe prestar atención a esto.

Combinando Herencia y Composición



exec maven plugin:1.0.0:exec -D

Constructor del Configurador

Constructor Utensilio

Constructor Cuchara

Constructor Utensilio

Constructor Tenedor

Constructor Utensilio

Constructor Cuchillo

Constructor Plato

Constructor PlatoCena

Constructor ConfiguradorCena

Elegir Herencia o Composición

- La **Composición** se utiliza cuando queremos las características de una clase existente.
 - En este caso se modela la relación “**Tiene un**”-> La nueva clase tiene características de otras clases.
 - Normalmente los atributos se declararán privados -> de esta forma ocultamos la interfaz de los objetos incrustados en la nueva clase.
 - Excepcionalmente nos puede interesar declarar los atributos de tipo objeto como public para ayudar al programador cliente a entender cómo usar la clase.
 - En este caso toma sentido utilizar el modificador “protected” en lugar de “public”
- La Herencia se utiliza cuando a partir de una clase construimos una especialización de la misma.
 - En este caso se modela la relación “**Es un**”-> La nueva clase es un tipo de la clase existente.

Elegir Herencia o Composición

- La composición se utiliza más que la Herencia, ya que la clave de la reutilización es empaquetar datos y métodos en una clase y utilizar objetos de dicha clase.
- La Herencia se debe utilizar con moderación, solo cuando sea útil.
- Una forma de determinar si se necesita utilizar composición o herencia es preguntarse si se realizará una conversión ascendente de la nueva clase a la clase base. Si no es necesaria esta conversión entonces se debe analizar profundamente si la herencia es necesaria.

Modificadores de acceso

MODIFICADOR	CLASE	ATRIBUTO	MÉTODO
public	La clase se puede usar en cualquier paquete	El atributo es visible en cualquier paquete	El método se puede usar en cualquier paquete.
protected	Solo para clases internas	En cualquier clase del mismo paquete y en las clases derivadas	En cualquier clase del mismo paquete y clases derivadas
private	Solo para clases internas	En la misma clase	En la misma clase
“friendly”	En el mismo paquete	En el mismo paquete	En el mismo paquete
static	Solo clases internas	Se pueden usar sin instanciar un objeto de la clase. Es compartida por todos los objetos de la clase.	Se pueden invocar sin instanciar un objeto de la clase

Modificadores de acceso

MODIFICADOR	CLASE	ATRIBUTO	MÉTODO
final	No se puede heredar de ella	Convierte el campo en constante de forma que su valor no puede cambiar una vez ha sido inicializado. Si se trata de una referencia a un objeto , no podrá cambiar dicha referencia, pero sí el contenido del objeto. Igual ocurre con los arrays.	No se puede redefinir en una clase derivada. (El mismo efecto se consigue si el método es declarado private en la clase base)
final static	-	Constante de clase: Campo de valor constante y compartido por todos los objetos de la clase.	-
abstract	No se pueden instanciar objetos a partir de la clase.	-	No tiene cuerpo. Convierte a la clase en abstracta.

Destrucción de objetos

- ❑ En Java la destrucción de objetos y liberación de memoria es realizada por el Garbage Collector o recolector de basura.
- ❑ Este proceso se realiza periódicamente y busca objetos que ya no son referenciados y los marca para su eliminación. Posteriormente los irá eliminando de la memoria cuando lo considere oportuno.
- ❑ En Java es posible implementar un método destructor de una clase, denominado `finalize()`.
- ❑ El método `finalize()` es invocado por el recolector de basura cuando va a destruir el objeto (no se sabe cuando sucederá exactamente).
- ❑ Si no existe el método `finalize()` se ejecutará un destructor por defecto (el método `finalize()` de la clase `Object`) que liberará la memoria ocupada por el objeto.
- ❑ Si un objeto utiliza recursos más allá de la memoria que utiliza para su atributos, y no tenemos garantía de que el entorno de ejecución los vaya a liberar (cerrar archivos, cerrar conexiones de red, cerrar conexiones de BBDD, etc), deberemos implementar el método `finalize()` en la clase.

Destrucción de objetos

- Si se necesita garantizar que se ejecute la finalización de un método (finalize) se puede recurrir al método `runFinalization()` de la clase `System` para forzarlo:

```
System.runFinalization();
```

- Este método se encarga de llamar a todos los métodos `finalize` de todos los objetos marcados por el recolector de basura para ser destruidos.
- Si necesitas implementar un destructor, debes tener en cuenta:
 - El nombre del destructor será `finalize()`
 - No puede recibir argumentos ni devolver ningún valor.
 - Solo puede haber un destructor en una clase.