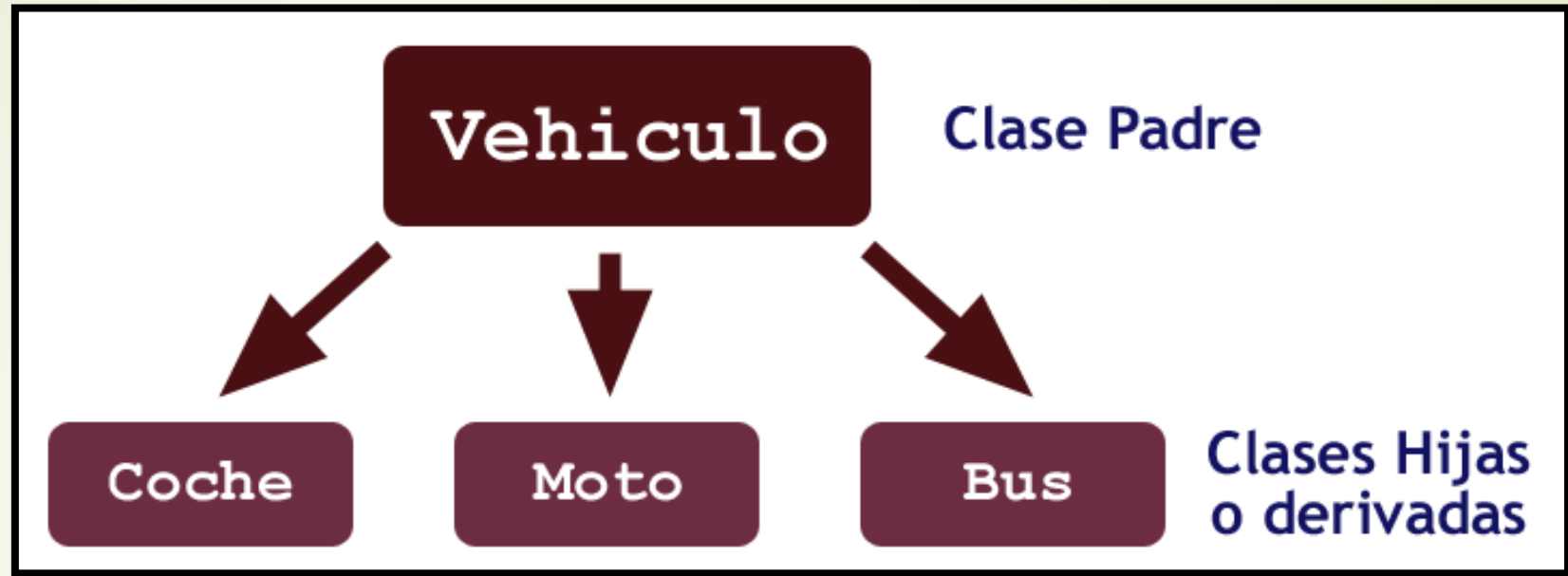




Desarrollo de Aplicaciones Web y Multiplataforma: Programación

DOCENTE: Daniel López Lozano





Tema 7.

P00 Avanzada. Herencia y Polimorfismo

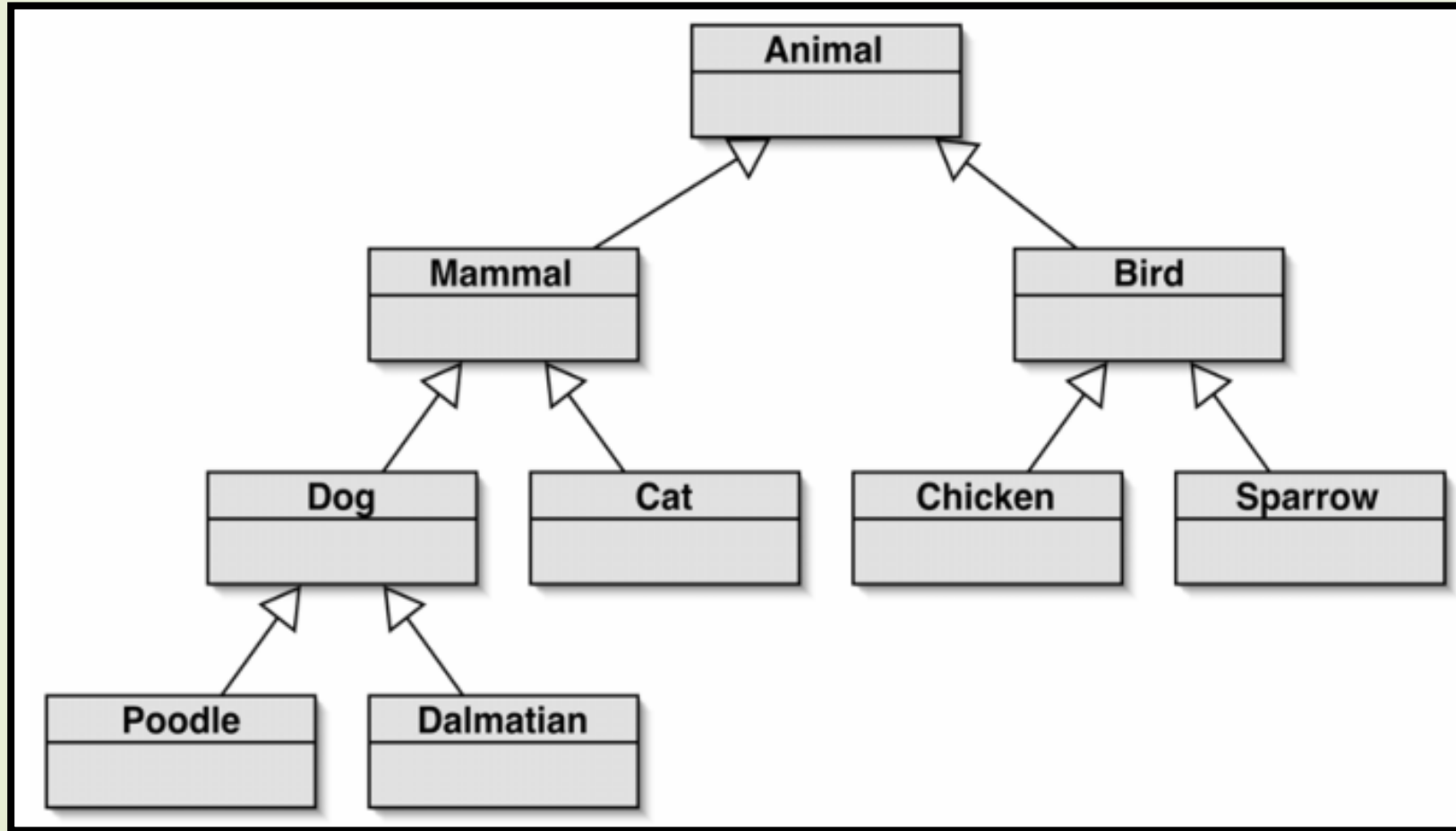
Índice de contenidos

- ❑ Herencia de clases. Superclase y subclases.
- ❑ Jerarquía de clases. Polimorfismo.
- ❑ La clase Object.
- ❑ Clases y métodos abstractos.
- ❑ Herencia de interfaces.

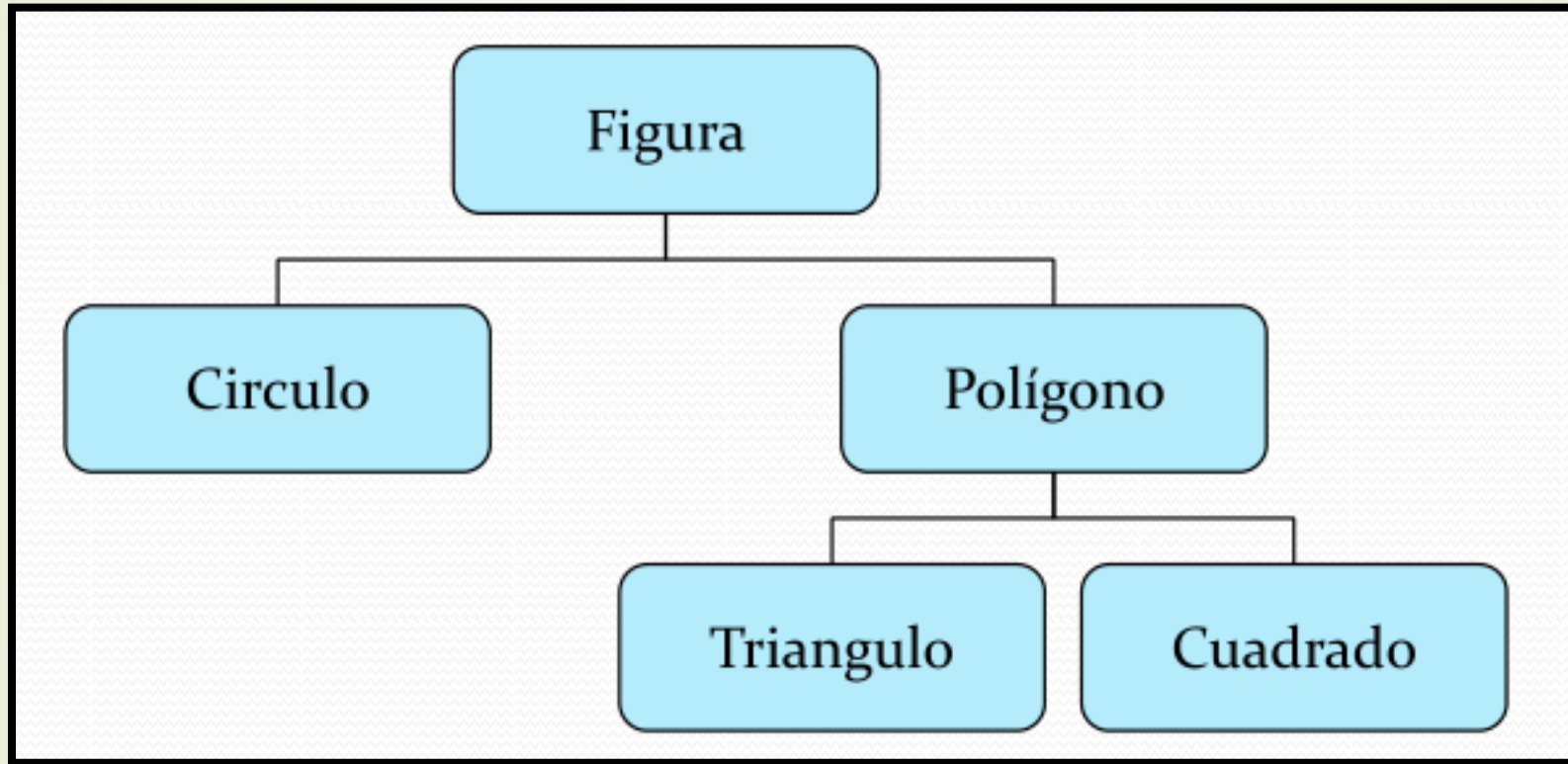
- ❑ En la POO la herencia consiste en la creación de clases a partir de otras ya existentes.
- ❑ De este modo, la nueva clase o clase hija (subclase) “hereda” los atributos y métodos de la clase padre (superclase)
- ❑ Además se pueden añadir nuevos atributos y métodos e incluso modificar los heredados.
- ❑ Es imprescindible para la reutilización de código y la descomposición de problemas en subproblemas.
- ❑ Junto con el polimorfismo es la técnica estrella de la POO.

- ❑ Es la base de la reutilización de código ya que, cuando una clase hija hereda de una clase padre, hereda los atributos y métodos de este.
- ❑ En general todas las subclases no sólo adoptan las variables y comportamientos de las superclases sino que los amplían y/o modifican.
- ❑ La herencia genera una jerarquía de clases que se basa en la existencia de relaciones de generalización y especialización entre clases.
- ❑ Esta jerarquía establece relaciones de compatibilidad entre clases que estando separadas no lo eran.

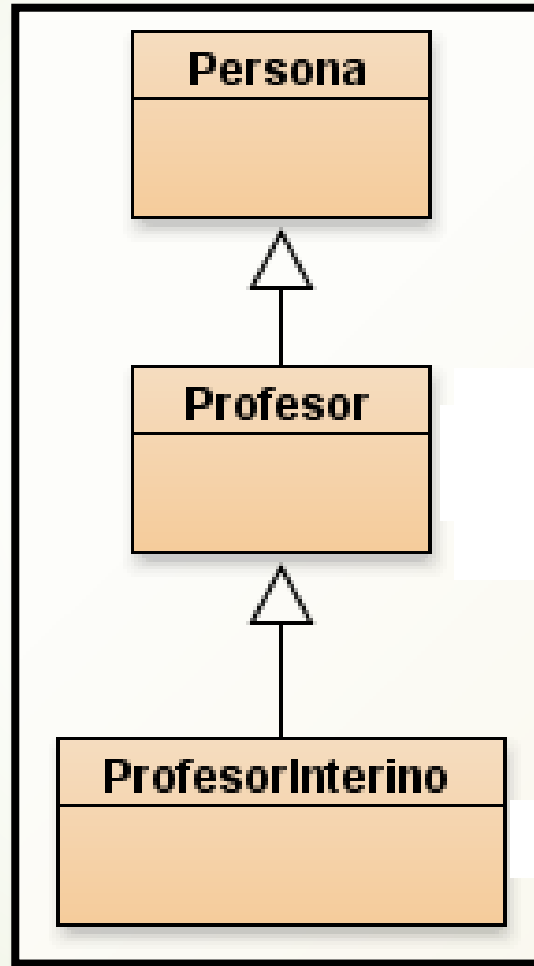
Generalización/Especialización



Generalización/Especialización



La herencia es transitiva



- ❑ La herencia es transitiva, es decir si B hereda de A y C hereda de B, entonces también C hereda de A.
- ❑ C no sólo hereda de B sino que también hereda de A.
- ❑ Los métodos y atributos siempre se pueden heredar exactamente igual que en los antecesores o modificarlos (generalmente modificando su comportamiento pero manteniendo su esencia).
- ❑ Una subclase puede seguir utilizando ambos comportamientos, el heredado y el redefinido.
- ❑ Todo esto es así a menos que se restrinja la herencia mediante modificadores de visibilidad y uso.

- ❑ En Java sólo está permitida la herencia simple, es decir, una clase puede tener muchas subclases pero una clase sólo puede tener una superclase.
- ❑ Cuando una clase hereda de otra, es porque cumple la funcionalidad de que la subclase es un caso específico de la superclase.
- ❑ La herencia contribuye a:
 - ❑ Evitar duplicación de código
 - ❑ Reutilizar código
 - ❑ Mejorar el mantenimiento
 - ❑ Extensibilidad

- ❑ Si un método o atributo es declarado como **private** no podrá ser usado ni modificado por una subclase.
- ❑ La herencia está activada si no ponemos nada (por defecto, nada recomendable), **public** o **protected**.
- ❑ El modificador **protected** es como **private** pero con la herencia permitida.
- ❑ Si queremos que una subclase herede, pero no pueda modificar lo que hereda debemos usar “**final**”.
- ❑ Si hemos modificado en la subclase un método y queremos usar el método original hay que usar la palabra “**super**” como prefijo.

Un ejemplo simple



atributos

edad
numero_patas
métodos
envejecer
andar



atributos

edad
numero_aletas
metodos
envejecer
nadar

Comparten elementos y aportan propios

Un ejemplo algo más interesante



PIRATA

atributos

puntos_vida
puntos_mana
fuerza
agilidad

...

métodos

atacar
moverse
abrir_cerradura



MAGO BLANCO

atributos

puntos_vida
puntos_mana
fuerza
agilidad

...

métodos

atacar
moverse
hechizo_invocación



SANADORA

atributos

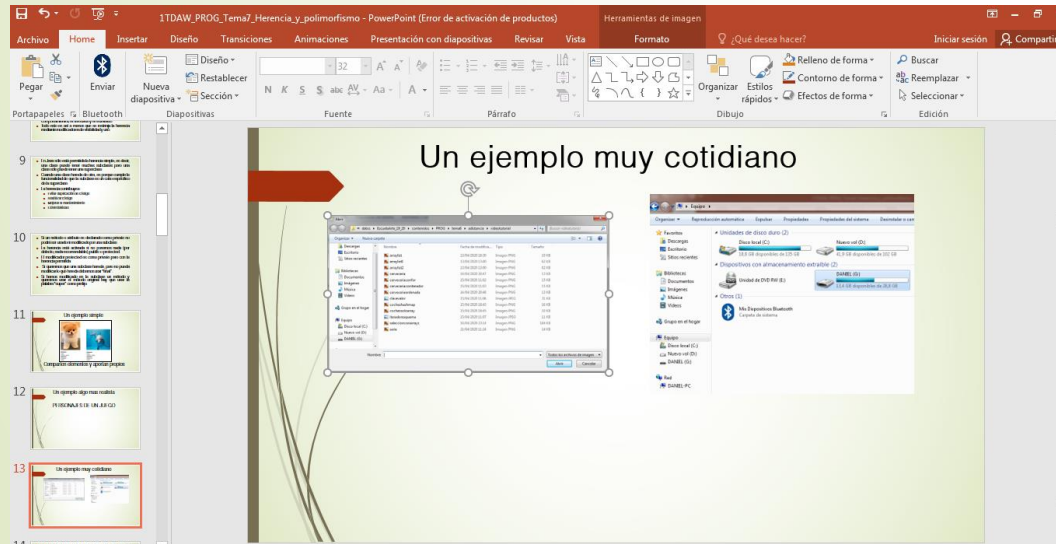
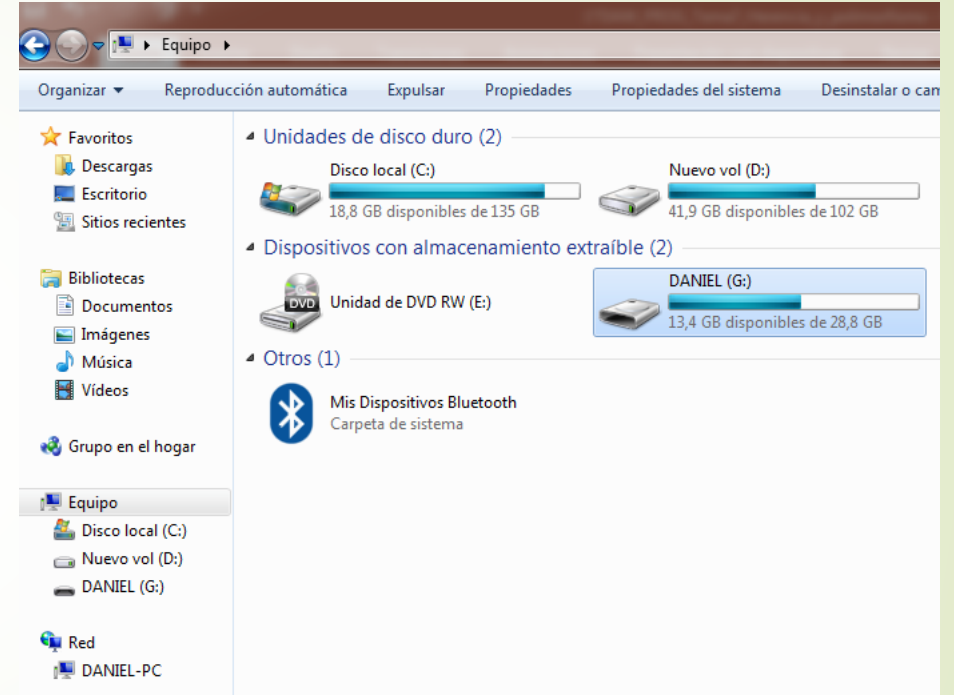
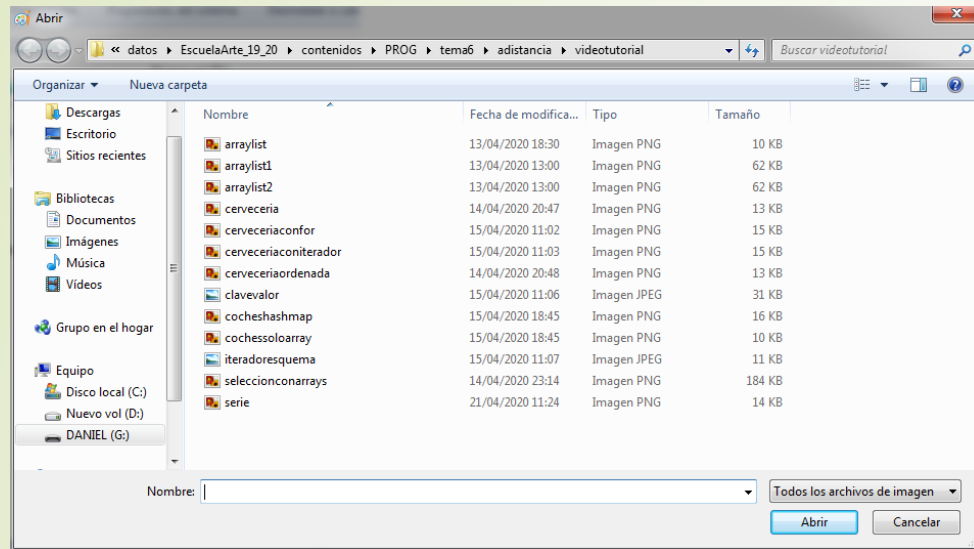
puntos_vida
puntos_mana
fuerza
agilidad

...

metodos

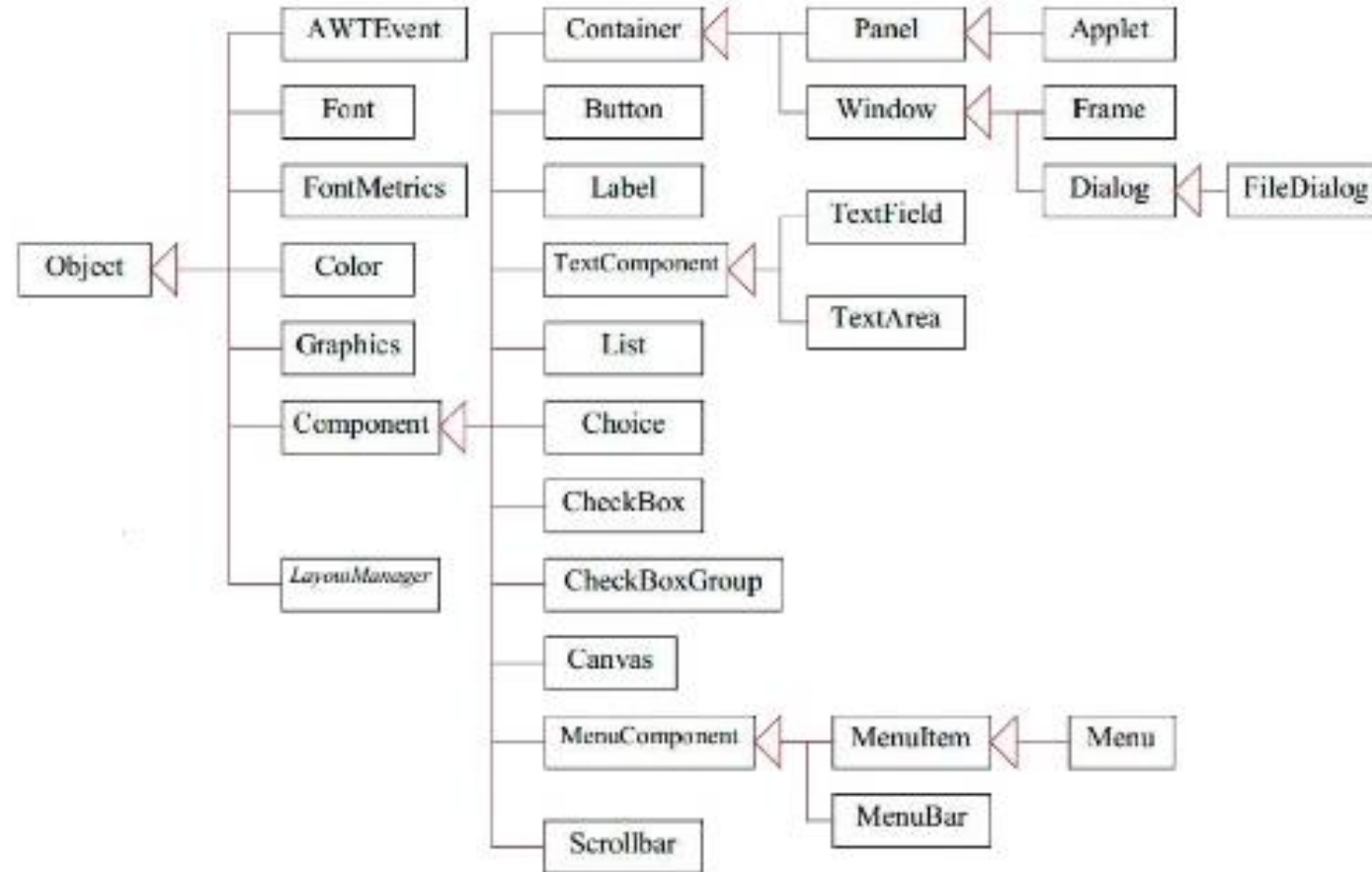
atacar
moverse
hechizo_curación

Un ejemplo muy cotidiano



Herencia transitiva compleja

Jerarquía de clases para las GUI: JComponent



Clase Bombilla

```
public class Bombilla {  
    protected boolean encendida;  
  
    public Bombilla(){  
        this.encendida=false;  
    }  
  
    public void encender(){  
        this.encendida=true;  
    }  
  
    public void apagar(){  
        this.encendida=false;  
    }  
  
    public boolean getEncendida(){  
        return this.encendida;  
    }  
}
```

```
    public String toString(){  
        String res="";  
        if(this.encendida){  
            res="Bombilla encendida\n";  
        }else{  
            res="Bombilla apagada\n";  
        }  
        return res;  
    }  
}
```

Clase BombillaVariable

```
public class BombillaVariable extends Bombilla{  
    protected int intensidad;  
  
    //Constructor que se base en el constructor de la superclase  
    public BombillaVariable(){  
        super();  
        this.intensidad=0;  
    }  
}
```

Recuerda!!

Se usa la palabra **extends** para heredar

Clase BombillaVariable

```
//Metodos que se añaden para los atributos nuevos  
public void aumentaIntensidad(){  
    if(this.encendida && this.intensidad<10){  
        this.intensidad++;  
    }else{  
        System.out.println("No se puede subir la intensidad");  
    }  
}  
  
public void disminuyeIntensidad(){  
    if(this.encendida && this.intensidad>0){  
        this.intensidad--;  
    }else{  
        System.out.println("No se puede bajar la intensidad");  
    }  
}
```

Clase BombillaVariable

```
public int getIntensidad(){
    return this.intensidad;
}

//Metodos redefinidos para añadirles nueva funcionalidad

public String toString(){
    String res=super.toString();

    res+="-----\n";
    res+="Marcador de intensidad en:"+this.intensidad+"\n";

    return res;
}
```

Probar la herencia

```
public static void main(String[] args) {  
    Bombilla b=new Bombilla();  
    BombillaVariable bv=new BombillaVariable();  
  
    b.encender();  
    System.out.println(b.toString());  
  
    bv.encender();  
    bv.aumentaIntensidad();  
    bv.aumentaIntensidad();  
    bv.aumentaIntensidad();  
  
    System.out.println(bv.mostrarBombilla());  
}
```

Clase CuentaCorriente

```
public class CuentaCorriente {  
    protected String nombre;  
    protected String codigo_cuenta;  
    protected String direccion;  
    protected double saldo;  
  
    //Constructores  
    public CuentaCorriente(String nombre,String codigo_cuenta,String direccion){  
        this.nombre=nombre;  
        this.codigo_cuenta=codigo_cuenta;  
        this.direccion=direccion;  
        this.saldo=0;  
    }  
  
    public CuentaCorriente(String nombre,String codigo_cuenta,String direccion,double saldo_inicial){  
        this.nombre=nombre;  
        this.codigo_cuenta=codigo_cuenta;  
        this.direccion=direccion;  
        this.saldo=saldo_inicial;  
    }  
}
```

Clase CuentaCorriente

```
public String obtenerNombre(){
public String obtenerDireccion(){
...
public void ingresar(double cantidad){
public void retirar(double cantidad){
public String toString(){
    String res;

    res="=====\n"+
        "|Nombre propietario: "+nombre+"\n"+
        "|Direccion propietario: "+direccion+"\n"+
        "|Numero de cuenta: "+codigo_cuenta+"\n"+
        "|Saldo en cuenta: "+saldo+"\n"+
        "=====\n";

    return res;
}
```


Clase CuentaCredito

```
public class CuentaCredito extends CuentaCorriente{
    protected double limite_credito;

    public CuentaCredito(String nombre,String codigo_cuenta,String direccion,double sa
        super(nombre, codigo_cuenta, direccion, saldo_inicial);
        this.limite_credito=limite_credito;
    }

    public double getLimite_credito() {
        return this.limite_credito;
    }

    public void setLimite_credito(double limite_credito) {
        this.limite_credito = limite_credito;
    }

    public String toString(){
        String res=super.toString();

        res+="LIMITE DE CREDITO: "+this.limite_credito+"\n";

        return res;
    }
}
```










Clase CuentaCredito

```
public void retirar(double cantidad){
    Scanner teclado=new Scanner(System.in);
    char respuesta;
    if(cantidad<=saldo){
        saldo=-cantidad;
    }else{
        System.out.println("No tienes dinero suficiente en la cuenta");
        System.out.println("¿Quieres sacar a credito?(s/n)");
        respuesta = teclado.next().charAt(0);
        if(respuesta=='s'){
            retirarACredito(cantidad);
        }
    }
}

public void retirarACredito(double cantidad){
    if(cantidad<=limite_credito){
        limite_credito=-cantidad;
    }else{
        System.out.println("Error nos tiene suficiente credito");
    }
}
```

Refactorizar las siguientes clases usando herencia

Futbolista
 id: Integer
 Nombre: String
 Apellidos: String
 Edad: Integer
 dorsal: Integer
 demarcacion: String
 Concentrarse(): void
 Viajar(): void
 jugarPartido(): void
 entrenar(): void

Entrenador
 id: Integer
 Nombre: String
 Apellidos: String
 Edad: Integer
 idFederacion: String
 Concentrarse(): void
 Viajar(): void
 dirigirPartido(): void
 dirigirEntrenamiento(): void

Masajista
 id: Integer
 Nombre: String
 Apellidos: String
 Edad: Integer
 Titulacion: String
 aniosExperiencia: Integer
 Concentrarse(): void
 Viajar(): void
 darMasaje(): void

- ❑ Dentro de nuestro ejemplo de **Animal**, **AnimalTerrestre** o **AnimalAcuatico** vamos a profundizar para hablar de Polimorfismo.
- ❑ Polimorfismo se refiere a la capacidad de los objetos en tener distintas capacidades y diferencias, pero perteneciendo a una misma familia.
- ❑ Esta estrechamente ligado a la herencia para poder aplicarse.
- ❑ Vamos a desarrollar un ejemplo donde de **AnimalTerrestre** hereda **AnimalBipedo** y **AnimalCuadrupedo**.

- ❑ Ambos tienen la capacidad de andar pero de manera distinta obviamente.

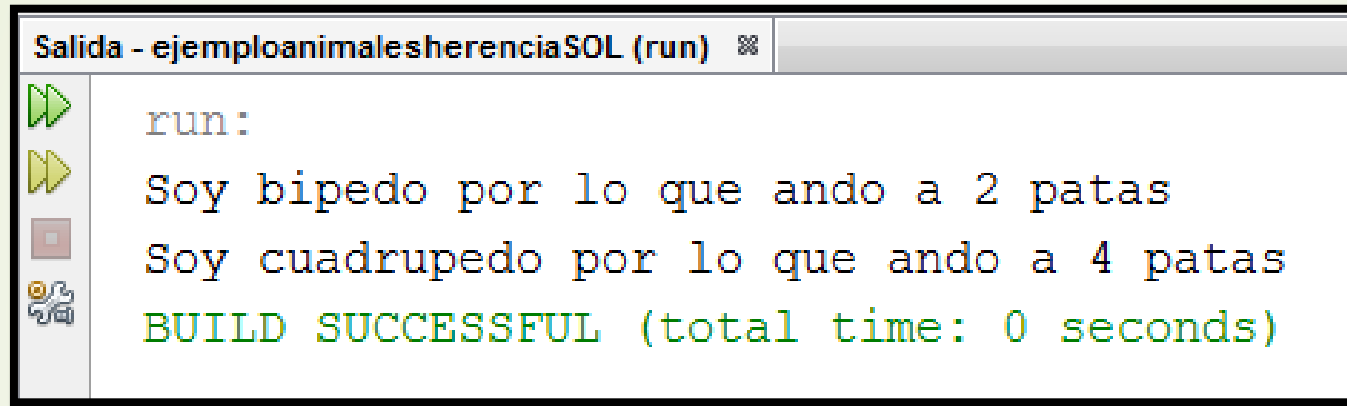
```
public class AnimalBipedo extends AnimalTerrestre{  
    public AnimalBipedo(int edad){  
        super(edad,2);  
    }  
  
    public void andar(){  
        System.out.println("Soy bipedo por lo que ando a 2 patas");  
    }  
}
```

```
public class AnimalCuadrupedo extends AnimalTerrestre{  
    public AnimalCuadrupedo(int edad){  
        super(edad,4);  
    }  
  
    public void andar(){  
        System.out.println("Soy cuadrupedo por lo que ando a 4 patas");  
    }  
}
```

- ❑ Si tenemos una variable de tipo **AnimalTerrestre** podemos guardar en ella objetos de **AnimalBipedo** y **AnimalCuadrupedo** porque al tener una relación de herencia son compatibles.

```
public static void main(String[] args) {  
    AnimalBipedo mono=new AnimalBipedo(20);  
    AnimalCuadrupedo perro=new AnimalCuadrupedo(10);  
  
    AnimalTerrestre referencia;  
  
    //Funciona porque guarda un objeto que hereda de su clase  
    referencia=mono;  
    referencia.andar();  
  
    //Supuestamente es el mismo codigo pero al guardar un objeto cuadrupedo  
    //el resultado cambia  
    referencia=perro;  
    referencia.andar();  
}
```


- ❑ Y lo mas sorprendente es que si llamamos al método andar invoca al método del objeto al que pertenece y no al de la clase AnimalTerrestre.



```
run:
Soy bipedo por lo que ando a 2 patas
Soy cuadrupedo por lo que ando a 4 patas
BUILD SUCCESSFUL (total time: 0 seconds)
```

- ❑ Esto es así porque Java detecta el tipo del objeto no basándose en el tipo de la variable sino en su composición.
- ❑ A esto se llama vinculación dinámica.

- ❑ Vamos hacer un diseño orientado a objetos avanzado.
- ❑ Imaginemos que queremos hacer una aplicación que gestione una base de datos multimedia digital donde en un principio tendremos información de **canciones** y películas.
- ❑ De una **canción** queremos el titulo, el artista, genero, descripción, valoración, estudio donde se grabó y si lo tengo o no en mi colección.
- ❑ De una **película** queremos el titulo, el director, genero, descripción, valoración, productora de cine y si lo tengo o no en mi colección.

Diseño de las clases

```
public class Cancion {  
    private String titulo,autor,genero;  
    private String descripcion;  
    private String estudio_grabacion;  
    private int duracion,valoracion;//De 0 a 5  
    private boolean loTengo;  
  
    public Cancion(String titulo, String autor, String g  
        this.titulo = titulo;  
        this.autor = autor;  
        this.genero = genero;  
        this.descripcion = descripcion;  
        this.estudio_grabacion = estudio_grabacion;  
        this.duracion = duracion;  
        this.valoracion = valoracion;  
        this.loTengo = loTengo;  
}
```

Diseño de las clases

```
public class Pelicula {  
    private String titulo,director,genero;  
    private String descripcion,productora;  
    private int duracion,valoracion; //De 0 a 5  
    private boolean loTengo;  
  
    public Pelicula(String titulo, String director, String  
        this.titulo = titulo;  
        this.director = director;  
        this.genero = genero;  
        this.descripcion = descripcion;  
        this.productora = productora;  
        this.duracion = duracion;  
        this.valoracion = valoracion;  
        this.loTengo = loTengo;  
}
```

Diseño de las clases

```
public String toString(){
    String res="";

    res="-----"+
        "TITULO: "+this.titulo+"\n"+
        "GENERO: "+this.genero+"\n"+
        "DESCRIPCION: "+this.descripcion+"\n"+
        "DURACION: "+this.duracion+" minutos\n"+
        "VALORACION: "+this.valoracion+"\n";
    if(this.loTengo){
        res+="LO TENGO";
    }else{
        res+="NO LO TENGO";
    }
    res+="ARTISTA: "+this.autor+"\n"+
        "ESTUDIO GRABACION: "+this.estudio_grabacion+"\n"+
        "-----";

    return res;
}
```

Diseño de las clases

```
public String toString(){
    String res="";

    res="-----"+
        "TITULO: "+this.titulo+"\n"+
        "GENERO: "+this.genero+"\n"+
        "DESCRIPCION: "+this.descripcion+"\n"+
        "DURACION: "+this.duracion+" minutos\n"+
        "VALORACION: "+this.valoracion+"\n";
    if(this.loTengo){
        res+="LO TENGO";
    }else{
        res+="NO LO TENGO";
    }
    res+="DIRECTOR: "+this.director+"\n"+
        "PRODUCTORA: "+this.productora+"\n"+
        "-----";

    return res;
}
```

Observamos que estamos duplicando código

Diseño de las clases

```
public void reproducir(){  
    System.out.println("Reproduciendo cancion "+this.titulo+" de "+this.autor);  
}
```

```
public void reproducir(){  
    System.out.println("Reproduciendo pelicula "+this.titulo+" de "+this.director);  
}
```

Aquí tenemos dos métodos que son muy parecidos pero solo cambian algunos datos

Clase Database

```
public class Database {  
    private ArrayList<Cancion> lista_canciones;  
    private ArrayList<Pelicula> lista_peliculas;  
  
    public void añadirCancion(String titulo, Str  
    Cancion nueva_cancion=new Cancion(titulo, art  
    lista_canciones.add(nueva_cancion);  
}  
  
    public void añadirPelicula(String titulo, Str  
    Pelicula nueva_pelicula=new Pelicula(titulo,  
    lista_peliculas.add(nueva_pelicula);  
}
```

Clase que contiene datos de ambos tipos

Clase Database

```
public String toString()
{
    String res="Resumen biblioteca multimedia";
    for(Cancion objeto_canciones:lista_canciones) {
        res+=objeto_canciones.toString();
    }
    for(Pelicula objeto_pelicula:lista_peliculas) {
        res+=objeto_pelicula.toString();
    }

    return res;
}
```

También aquí duplicamos código

Clase Database

```
public void reproducirDatabase(){  
    for(Cancion objeto_canciones:lista_canciones) {  
        objeto_canciones.reproducir();  
    }  
    for(Pelicula objeto_pelicula:lista_peliculas) {  
        objeto_pelicula.reproducir();  
    }  
}
```

También aquí duplicamos código

Este diseño presenta varios problemas:

- ❑ En primer lugar estamos repitiendo código exacto.
- ❑ Si queremos añadir distintos tipo de multimedia existen atributos y comportamientos duplicados, cuando la única diferencia es el mecanismo de reproducción.
- ❑ Es difícil de aplicar mecanismos de reutilización o modificación, de formar que si queremos realizar cambios estamos casi avocados a empezar de nuevo.
- ❑ Existe un riesgo muy alto de propagar errores.
- ❑ Todo esto se puede mejorar usando herencia y polimorfismo.

Rediseño de clases. Clase Multimedia

```
public class Multimedia {  
    protected String titulo, genero;  
    protected String descripcion;  
    protected int duracion, valoracion;  
    protected boolean loTengo;  
  
    public Multimedia(String titulo, String genero, String des  
        this.titulo = titulo;  
        this.genero = genero;  
        this.descripcion = descripcion;  
        this.duracion = duracion;  
        this.valoracion = valoracion;  
        this.loTengo = loTengo;  
}
```

Rediseño de clases Cancion y Pelicula

```
public class Cancion extends Multimedia{  
    protected String autor;  
    protected String estudio_grabacion;  
  
    public Cancion(String titulo, String autor, String genero,  
        super(titulo,genero,descripcion,duracion,valoracion);  
        this.autor = autor;  
        this.estudio_grabacion = estudio_grabacion;  
}
```

```
public class Pelicula extends Multimedia {  
    protected String director;  
    protected String productora;  
  
    public Pelicula(String titulo, String director, String genero,  
        super(titulo,genero,descripcion,duracion,valoracion);  
        this.director = director;  
        this.productora = productora;  
}
```

Clase Database nueva

```
public class Database {  
    private ArrayList<Multimedia> lista_multi;  
  
    public void añadirCancion(String titulo, String  
        Cancion nueva_cancion=new Cancion(titulo,  
        lista_multi.add(nueva_cancion);  
    }  
  
    public void añadirPelicula(String titulo, String  
        Pelicula nueva_pelicula=new Pelicula(titulo,  
        lista_multi.add(nueva_pelicula);  
    }  
}
```


Permite simplificar enormemente el código

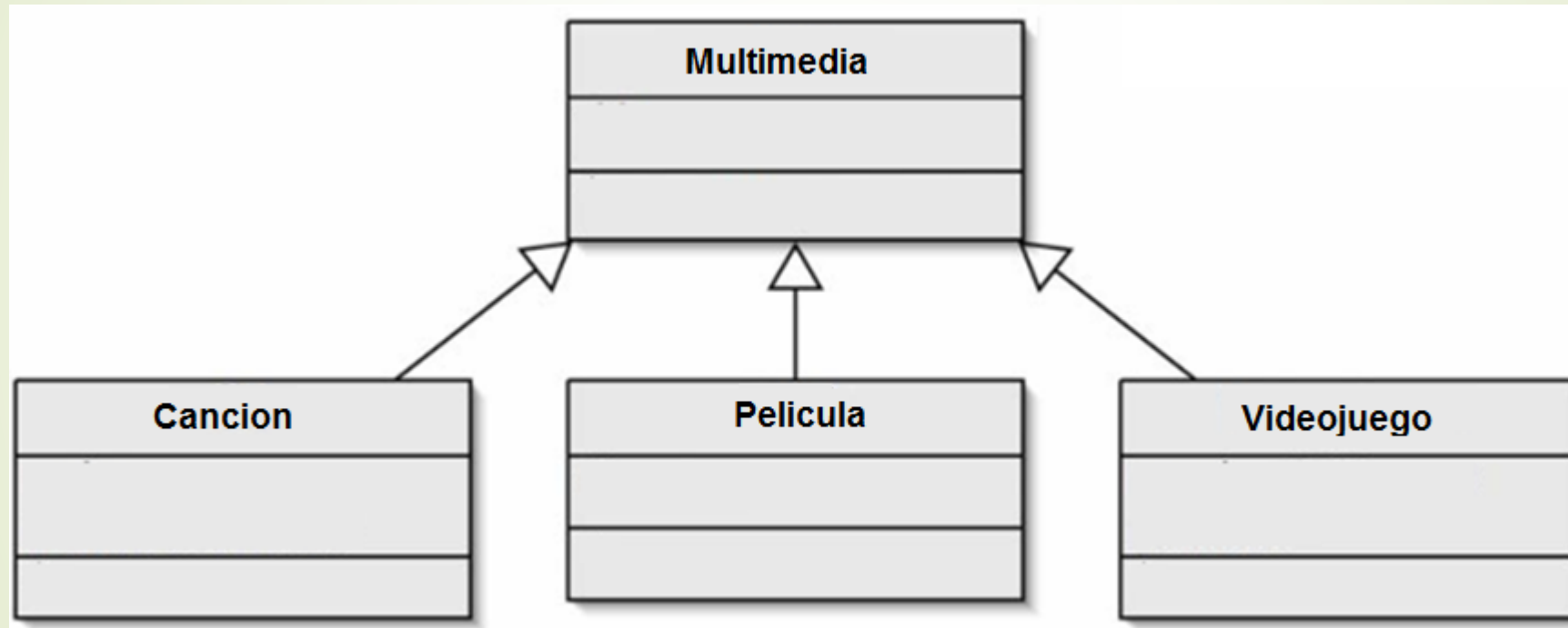
```
public String toString()
{
    String res="Resumen biblioteca multimedia";
    for(Multimedia objeto_multi:lista_multi){
        res+=objeto_multi.toString();
    }

    return res;
}

public void reproducirBiblioteca()
{
    for(Multimedia objeto_multi:lista_multi){
        objeto_multi.reproducir();
    }
}
```

La herencia y la vinculación dinámica hacen
el trabajo por nosotros

Además de poder añadir más tipos en el futuro sin necesidad de cambiar Database



- ❑ Esto esta bien pero si queremos usar una propiedad especifica de una subclase desde una variable de la clase Multimedia da error de compilación.

```
Multimedia objeto_multi=new Cancion("One","Metallica","Heavy","Sobre la guerra","EMI",9,5,true);  
System.out.println("El estudio de grabacion:"+objeto_multi.getEstudio_grabacion());  
//Da error aunque sabemos con seguridad que es un objeto de clase Cancion
```

- ❑ Para ello hay que hacer una operación llamada **casting** que fuerza a convertir el tipo de un objeto a otro, solo funcionando cuando el tipo destino es subclase.

```
Cancion objeto_transformado=(Cancion)objeto_multi;  
System.out.println("El estudio de grabacion:"+objeto_transformado.getEstudio_grabacion());
```

- ❑ Pero si queremos aplicarlo de forma masiva en un for va a dar error de ejecución porque no todos los objetos tienen que ser del mismo tipo.

```
// Contar los discos cuya discografica es Sony Music
public int contarCancionesSony(){
    int res=0;
    Cancion objeto_transformado;
    for(Multimedia objeto_multi:lista_multi){
        objeto_transformado=(Cancion)objeto_multi;
        if(objeto_transformado.getEstudio_grabacion().equals("Sony Music")){
            res++;
        }
    }

    return res;
}
```

```
Exception in thread "main" java.lang.ClassCastException: conpolimorfismo.Pelicula cannot be cast to conpolimorfismo.Cancion
    at conpolimorfismo.Database.contarCancionesSony(Database.java:77)
    at conpolimorfismo.Polimorfismo.main(Polimorfismo.java:89)
C:\Users\Daniel\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
```

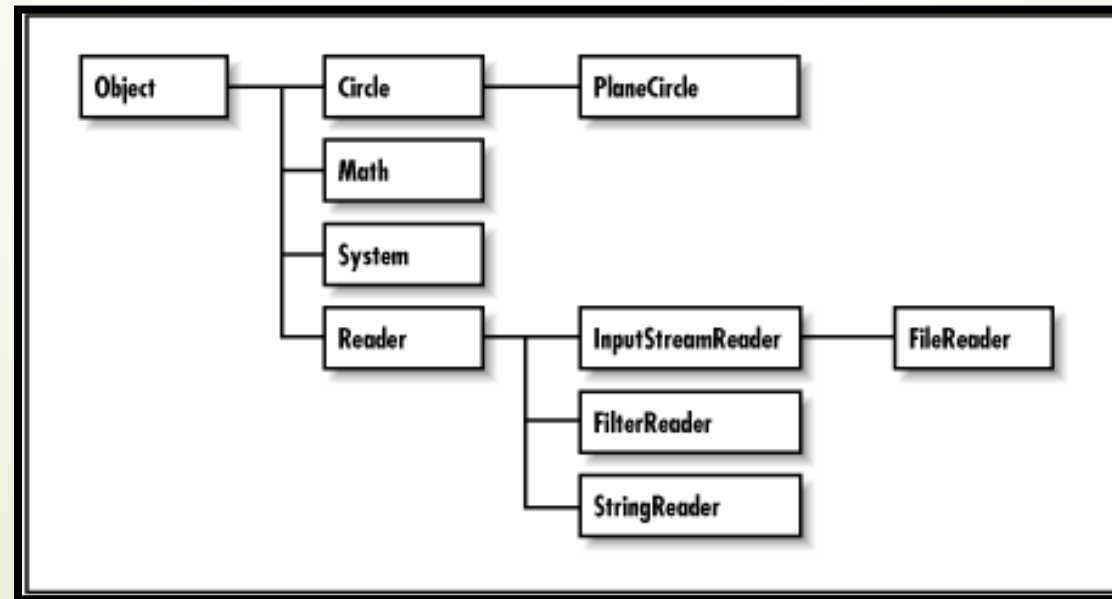
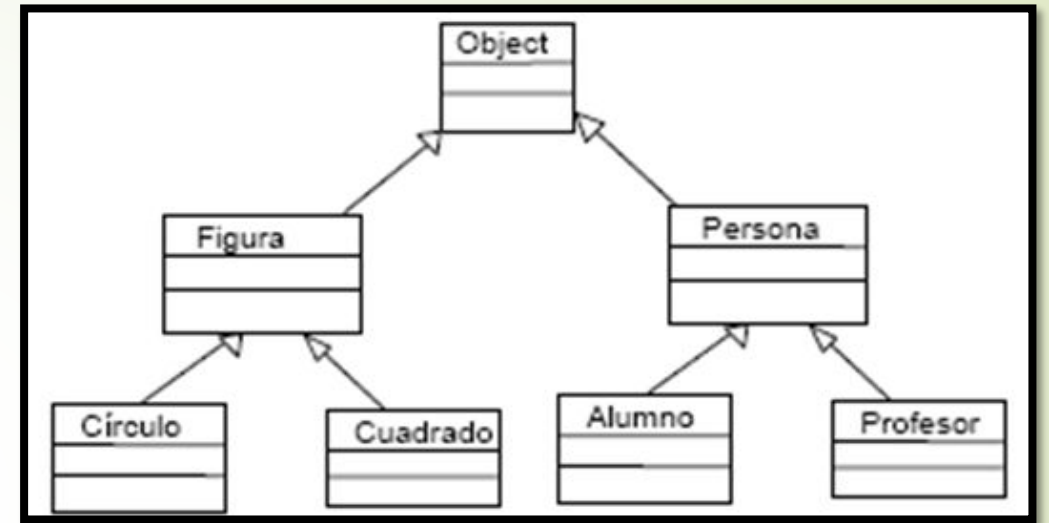
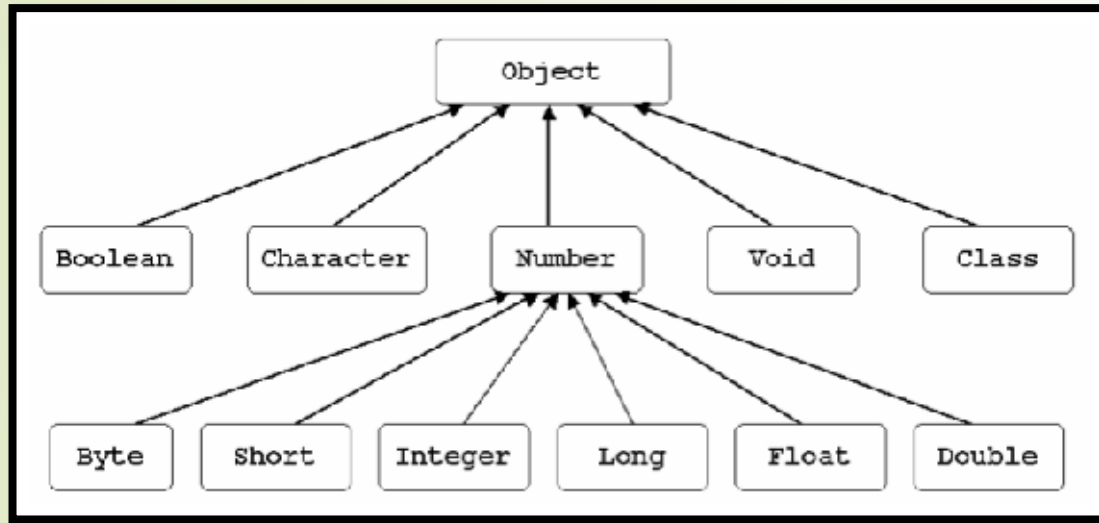
- ❑ Podemos hacer un “apaño” para situación especiales, utilizando la operación `instanceof` que comprueba si el objeto es del tipo que queremos y producir la excepción.

```
//Contar los discos cuya discografica es Sony Music
public int contarCancionesSony(){
    int res=0;
    Cancion objeto_transformado;
    for(Multimedia objeto_multi:lista_multi){
        if(objeto_multi instanceof Cancion){
            objeto_transformado=(Cancion)objeto_multi;
            if(objeto_transformado.getEstudio_grabacion().equals("Sony Music")){
                res++;
            }
        }
    }

    return res;
}
```

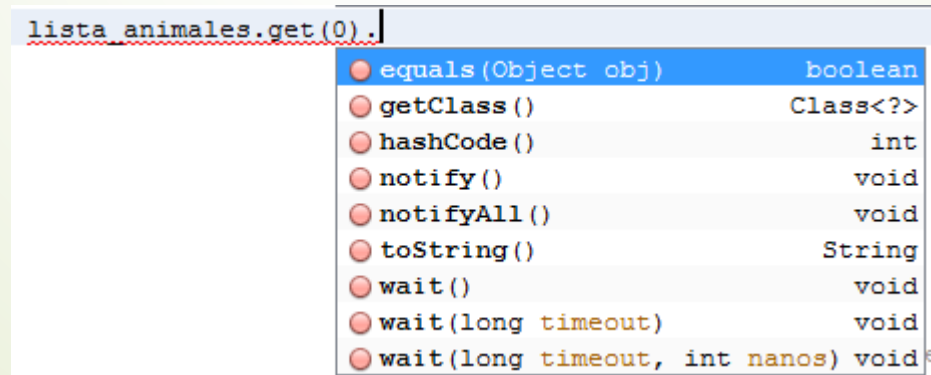
- ❑ Existe un clase universal llamada Object y es la base de la genericidad en Java aunque no lo observemos.
- ❑ Cualquier clase automáticamente hereda de Object y eso quiere decir que el polimorfismo se puede usar en cualquier objeto usando la clase Object.
- ❑ Observamos en Netbeans que cuando exploramos los métodos de una clase que estamos implementando podemos ver los métodos hashCode, toString, etc que son métodos de la clase Object.
- ❑ De esta manera existe una jerarquía de clases con raíz en la clase Object.

Jerarquías de Java basada en Object



```
public static void main(String[] args) {  
  
    AnimalBipedo mono=new AnimalBipedo(20);  
    AnimalCuadrupedo perro=new AnimalCuadrupedo(10);  
  
    AnimalTerrestre ciempies=new AnimalTerrestre(1, 100);  
    AnimalAcuatico delfin=new AnimalAcuatico(3, 4);  
  
    ArrayList<Object> lista_animales=new ArrayList<>();  
    lista_animales.add(mono);  
    lista_animales.add(perro);  
    lista_animales.add(ciempies);  
    lista_animales.add(delfin);  
}
```

- ❑ Pero los métodos disponibles son los de la clase Object



- ❑ El problema que solucionábamos usando el casting e instanceof.

```
AnimalTerrestre casting_terrestre;  
AnimalAcuatico casting_acuatico;  
for(Object dato:lista_animales){  
    if(dato instanceof AnimalAcuatico){  
        casting_acuatico=(AnimalAcuatico)dato;  
        casting_acuatico.nadar();  
    }else if(dato instanceof AnimalTerrestre){  
        casting_terrestre=(AnimalTerrestre)dato;  
        casting_terrestre.andar();  
    }  
}
```

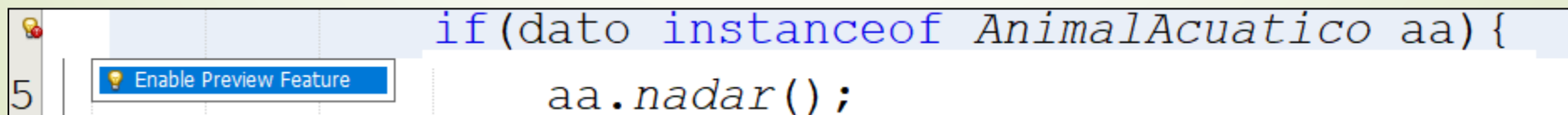
```
//Sin usar variables auxiliares  
for(Object dato:lista_animales){  
    if(dato instanceof AnimalAcuatico){  
        ((AnimalAcuatico) dato).nadar();  
    }else if(dato instanceof AnimalTerrestre){  
        ((AnimalTerrestre) dato).andar();  
    }  
}
```

- ❑ Donde pone **Object** podemos poner **Animal** que es superclase de **AnimalAcuatico** y **AnimalTerrestre**.

- ❑ Versiones nuevas del JDK de Java introducción una sintaxis simplificada para `instanceof` que hace el casting automáticamente, quedando tan simple como:

```
for(Animal dato:lista_animales){  
    if(dato instanceof AnimalAcuatico aa){  
        aa.nadar();  
    }else if(dato instanceof AnimalTerrestre at){  
        at.andar()  
    }  
}
```

- ❑ Si el JDK es anterior al 14 hay que habilitar la sintaxis pulsando a la izquierda la bombilla que aparece



- ❑ El uso es limitado y pertenece a una solución anterior de la genericidad que hoy día se resuelve mejor con los `<>` al definir una colección por ejemplo.
- ❑ Hoy día solo se usa cuando no tenemos muy clara la jerarquía de memoria como por ejemplo en sistemas en red
- ❑ En un sistema de red se comunican datos entre distintos lenguajes, arquitecturas y sistemas operativos.
- ❑ Es un claro ejemplo donde no se pueden asegurar tipos de datos.
- ❑ De eso vemos un ejemplo en el lenguaje Javascript.

- ❑ Al igual que podemos forzar un método para que no se sobrescriba y una clase para que no se herede también podemos hacer lo contrario:
- ❑ Con la palabra reservada `abstract` podemos definir métodos genéricos e incluso vacíos que deberán sobrescribirse a la fuerza en las clases los hereden.
- ❑ Igualmente si definimos una clase como `abstract` esta clase no se podrá instanciar. No se podrán crear objetos de esta clase. Estas clases sólo se pueden heredar.
- ❑ La utilidad de las clases abstractas es crear clases genéricas o plantillas, cuya funcionalidad será implementada por las subclases.

- ❑ De esta manera, todas las subclases tendrán una interfaz común. Es decir funcionarán con los mismos métodos y realizarán las mismas operaciones.
- ❑ Pero cada subclase tendrá una funcionalidad interna diferente.
- ❑ En definitiva abstract es sinónimo de genérico.
- ❑ Vamos a ver un ejemplo con modelando figuras geométricas planas.
- ❑ Una figura geométrica tiene área o perímetro pero no podemos calcular dichos valores a menos que sepamos su tipo (cuadrado, circulo, triangulo).

Clase abstracta Figura

```
public abstract class Figura {  
  
    protected String color;  
  
    public String getColor() {  
        return color;  
    }  
  
    public abstract double areaFigura();  
    public abstract double perimetroFigura();  
}
```

Distintas extensiones de figuras

```
public class Cuadrado extends Figura{  
    protected double lado;  
  
    public double areaFigura(){  
        return lado*lado;  
    }  
  
    public double perimetroFigura(){  
        return 4*lado;  
    }  
}
```

```
public class Triangulo extends Figura{  
    protected double base, altura;  
  
    public double areaFigura(){  
        return base*altura/2;  
    }  
  
    public double perimetroFigura(){  
        return 3*base;  
    }  
}
```

```
public class Circulo extends Figura{  
    protected double radio;  
  
    public double areaFigura(){  
        return radio*radio*Math.PI;  
    }  
  
    public double perimetroFigura(){  
        return 2*Math.PI*radio;  
    }  
}
```

Otro ejemplo pieza de ajedrez

```
public abstract class PiezaAjedrez {  
    protected String color;//blanza o negra  
  
    public abstract void moverse();  
    //Cada pieza del ajedrez tiene su propia forma de moverse  
}
```

Algunas piezas y su uso

```
public class Rey extends PiezaAjedrez{  
    public Rey(String color){  
        this.color=color;  
    }  
  
    public void moverse(){  
        System.out.println("En cualquier direccion un solo paso");  
    }  
}
```

```
public class Alfil extends PiezaAjedrez{  
    public Alfil(String color){  
        this.color=color;  
    }  
  
    public void moverse(){  
        System.out.println("En cualquier diagonal");  
    }  
}
```

```
public class Torre extends PiezaAjedrez{  
    public Torre(String color){  
        this.color=color;  
    }  
  
    public void moverse(){  
        System.out.println("En vertical y horizontal");  
    }  
}
```

```
public class Caballo extends PiezaAjedrez{  
    public Caballo(String color){  
        this.color=color;  
    }  
  
    public void moverse(){  
        System.out.println("Haciendo una L");  
    }  
}
```

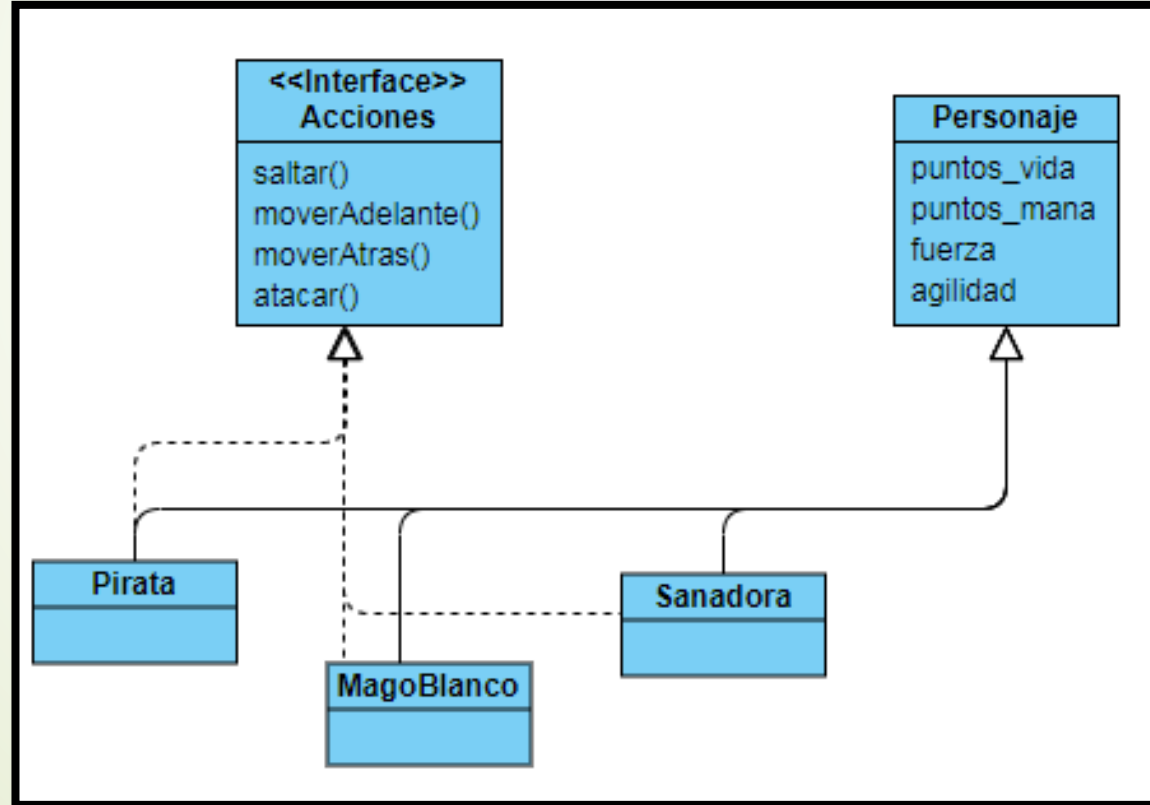
```
ArrayList<PiezaAjedrez> tablero_ajedrez=new ArrayList<>();  
  
Rey r_negro=new Rey("NEGRO");  
Caballo c_blanco=new Caballo("BLANCO");  
  
tablero_ajedrez.add(r_negro);  
tablero_ajedrez.add(c_blanco);  
//...
```

- ❑ En Java sólo se puede heredar de una única clase. En otros lenguajes como Smalltalk y C++. Este tipo de herencia es muy potente pero también es fuente de incoherencias y errores.
- ❑ Para solucionar esto en Java se definen las interfaces, de forma que una clase puede heredar de una única superclase pero además heredar(implementar) todas las interfaces que desee.
- ❑ Una interfaz es un tipo de clase especial en la que no se implementa ninguno de sus métodos, todos son abstractos.

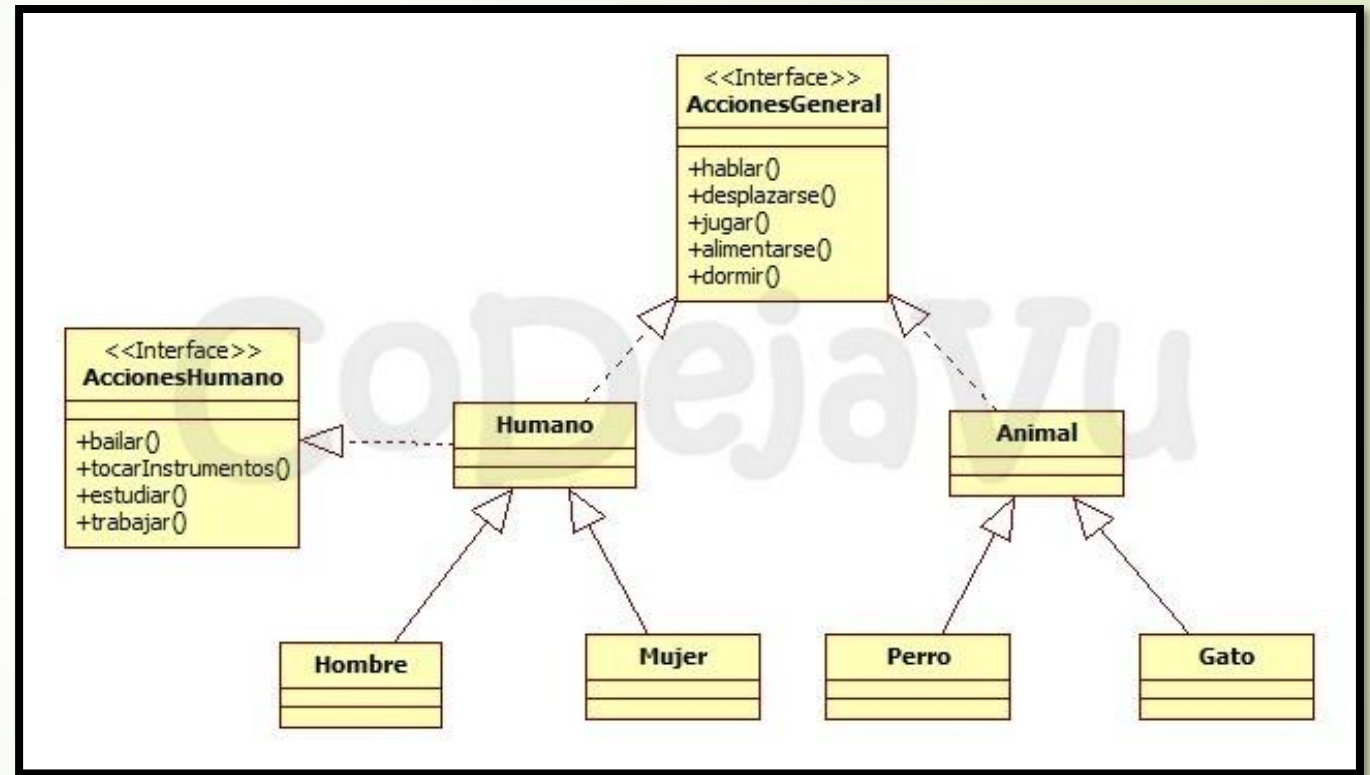
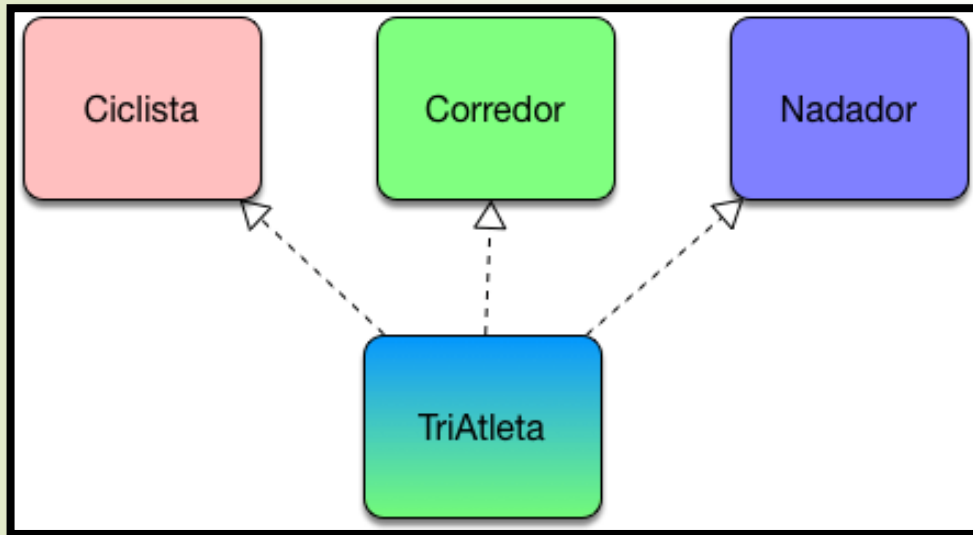
- ❑ Una interfaz sólo define los métodos que se van a utilizar, que parámetros reciben y que tipo de dato devuelven.
- ❑ Dejando libertad para implementar la funcionalidad en cada una de las subclases.
- ❑ Cuando se hereda de una interfaz, se dice que se implementa dicha interfaz, ya que no se hereda ninguna funcionalidad, sino que se lo que se va a hacer es implementar los métodos de esta.
- ❑ Cuando se implementa una interfaz, hay que implementar todos sus métodos o declarar la clase como abstracta.

Todo personaje de un videojuego tiene
características y acciones comunes

Dichas acciones son de obligada implementación



Los interfaces permiten lo que no permite la herencia de Java



Como usar interfaces en Java

```
public interface Corredor {  
    void correr();  
}  
  
public interface Nadador {  
    void nadar();  
}  
  
public interface Ciclista {  
    void montarEnBici();  
}
```

Como hacer herencia múltiple implementando interfaces

```
public class TriAtleta implements Corredor, Nadador, Ciclista{  
    public void montarEnBici(){  
        System.out.println("Montar en bici rapido");  
    }  
  
    public void nadar(){  
        System.out.println("Nadar en piscina no mas");  
    }  
  
    public void correr(){  
        System.out.println("Correr medio rapido");  
    }  
}
```

- ❑ Según los esquemas anteriores la clase Triatleta es compatible con los siguientes métodos ya la implementación de interfaces es lo que la herencia y el polimorfismo, o sea COMPATIBLES.

```
public class Competiciones {  
    public void correrTriatlon(TriAtleta persona){  
        persona.correr();  
        persona.montarEnBici();  
        persona.nadar();  
    }  
  
    public void participarVueltoCiclista(Ciclista persona){  
        persona.montarEnBici();  
    }  
  
    public void participarMaraton(Corredor persona){  
        persona.correr();  
    }  
  
    public void participarTravesia(Nadador persona){  
        persona.nadar();  
    }  
}
```

```
public static void main(String[] args) {  
    TriAtleta deportista=new TriAtleta();  
    Competiciones competi=new Competiciones();  
  
    competi.correrTriatlon(deportista);  
    competi.participarMaraton(deportista);  
    competi.participarTravesia(deportista);  
    competi.participarVueltoCiclista(deportista);  
}
```

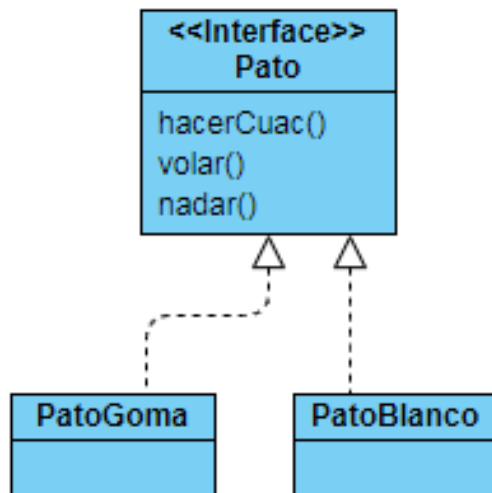

- ❑ Lo mismo pasaría con la clase Humano.
- ❑ Recordad que no existe la herencia múltiple en Java y esta es la manera de pseudo conseguirlo.
- ❑ Por supuesto que hay lenguajes donde existe la herencia múltiple pero se considera algo muy sofisticado de usar y necesario en casos muy concretos.
- ❑ Nuestro objetivo es entender como conseguir compatibilidad y mejor organización de código.
- ❑ Para entenderlo mejor debemos analizarlo al contrario mediante estos dos ejemplos.

- ❑ Supongamos que queremos modelar los patos

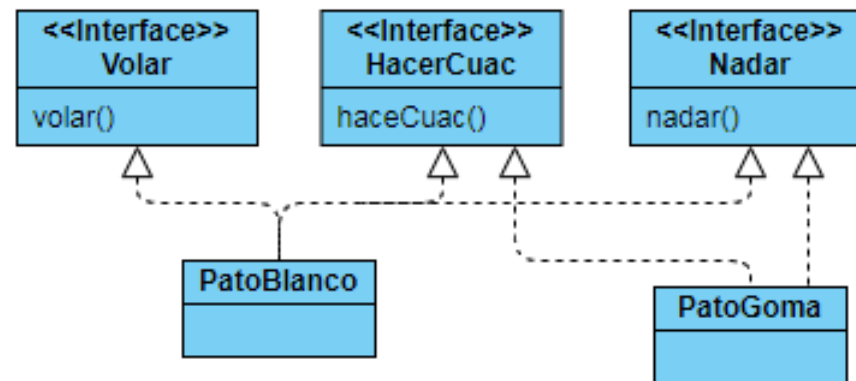


- ❑ Este esquema no es posible porque un pato de goma no puede volar. Por lo que segregamos en interfaces.

**MAL INTERFAZ
NO SIEMPRE SE
PUEDE CUMPLIR**

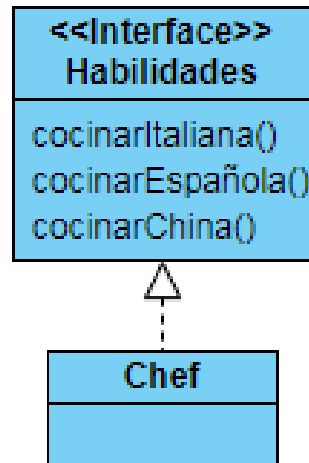


**BIEN. VARIOS INTERFACES
PARA SEPARAR ACCIONES**

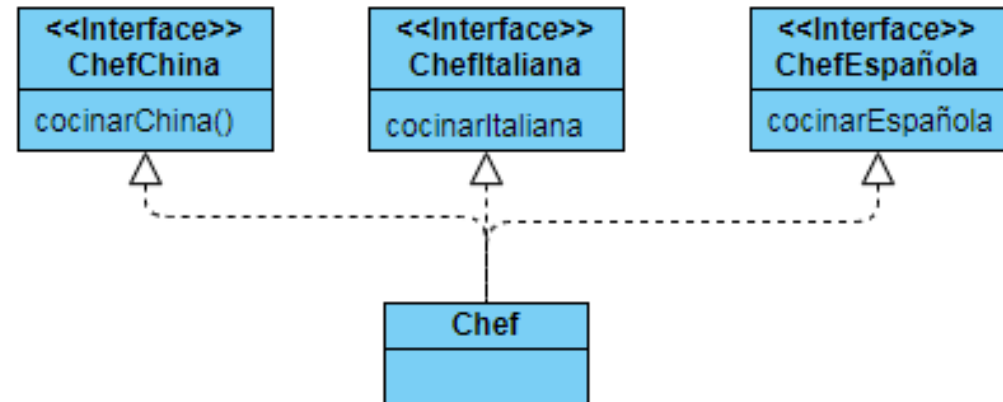


Otro ejemplo de separación de interfaces

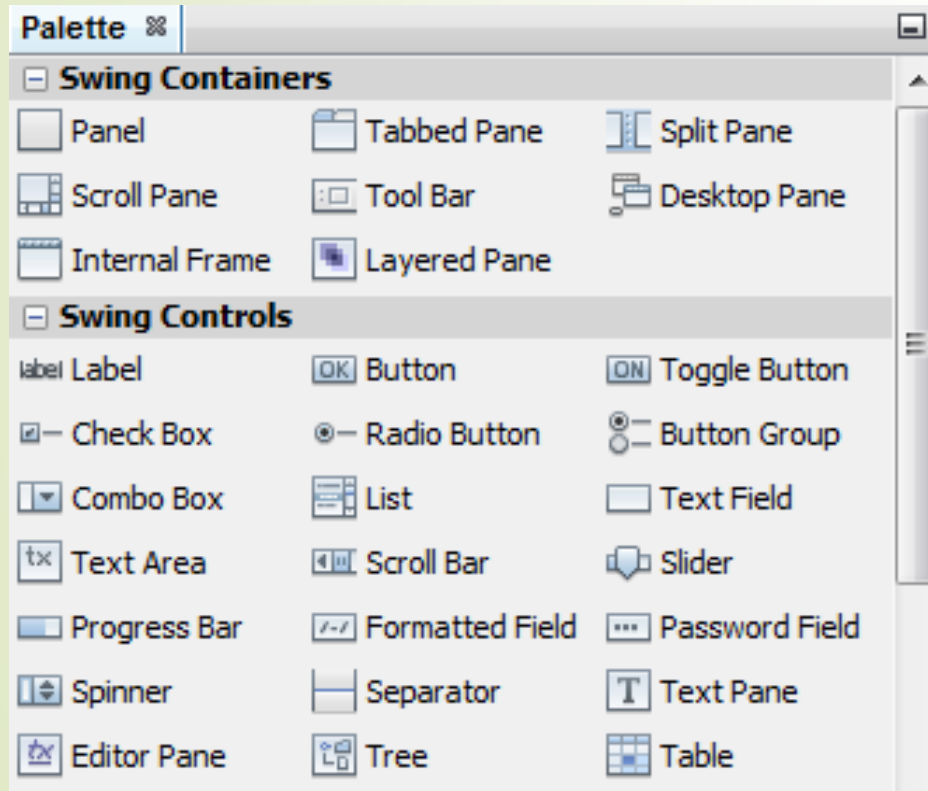
**MAL INTERFAZ
NO SIEMPRE SE
PUEDE CUMPLIR**



**BIEN. VARIOS INTERFACES
PARA SEPARAR ACCIONES**



Un ejemplo real seria la programación de eventos donde un componente puede responder a varias situaciones



Event Listener Interface

FocusListener

KeyListener

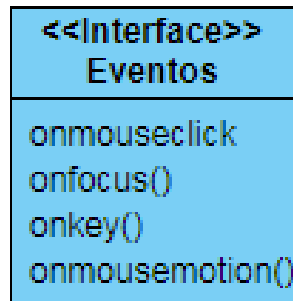
MouseListener

MouseMotionListener

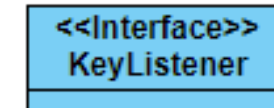
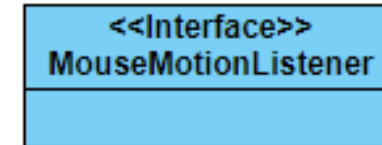
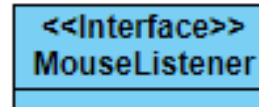
- ❑ El cuadro anterior tiene interfaces segregados según el evento que se quiera controlar sobre un componente.
- ❑ Sobre un botón se puede controlar que hacer cuando se hace clic en él, pasa el ratón por encima, tiene el foco, etc.
- ❑ Sobre una caja de texto se puede controlar que hacer cuando se escribe en ella, pasa el ratón por encima, tiene el foco, etc.
- ❑ Normalmente queremos controlar un solo evento, como mucho dos, no todos los posibles sobre un solo componente.

- ❑ Por eso existe un interfaz para cada evento y no un interfaz que tenga todos los eventos.

MAL UN SOLO INTERFAZ PARA CADA EVENTO



BIEN INTERFACES PARA CADA EVENTO



Bibliografía

- ❑ **García de Jalón, j.:** “Aprende Java como si estuvieras en primero”. Editorial TECNUN. 2000
- ❑ **Holzner, S.:** “La biblia de JAVA 2”. Editorial Anaya Multimedia 2000.
- ❑ **Moreno Pérez, J.C.:** “C.F.G.S Entornos de desarrollo” Editorial RA-MA. 2012
- ❑ **Wikipedia, la enciclopedia libre.** <http://es.wikipedia.org/>
Última visita: Octubre 2018.
- ❑ **López, J.C.:** “Curso de JAVA” <http://www.cursodejava.com.mx>
Última visita: Octubre 2018.
- ❑ **Documentación oficial Java JSE 8** <http://docs.oracle.com/javase/8/>
Última visita: Octubre 2015.
- ❑ **Programación en castellano: Java.** <http://www.programacion.net/java>
Última visita: Octubre 2015.