

# UNIDAD 5.

## POO Avanzada y excepciones.



# Índice

1. Ficha unidad didáctica.....	1
2. Contenidos.....	2
2.1. Introducción.....	2
2.2. POO Avanzada.....	3
2.2.1 Herencia.....	3
2.2.2 Tipos de herencia.....	4
2.2.3 Clases y métodos abstractos.....	6
2.2.4 Herencia en Java.....	7
2.2.4.1 Herencia básica.....	7
2.2.4.2 Palabra reservada final y la herencia.....	19
2.2.4.3 Interfaces e Implements.....	19
2.2.4.4 Clases y métodos abstractos .....	24
2.2.4.5 Polimorfismo.....	27
2.2.4.5.1. Clase Object.....	27
2.2.4.6 Variables poliformicas.....	35
2.2.4.7 Llamando a métodos de la clase padre.....	36
2.2.4.8 Anotaciones.....	37
2.2.5 Herencia en otros lenguajes OO.....	39
2.2.6 Relaciones entre clases y UML.....	40
2.2.6.1 UML y clases.....	41
2.2.6.2 Asociación.....	42
2.2.6.3 Agregación.....	43
2.2.6.4 Composición.....	44
2.2.6.5 Generalización/especialización.....	44
2.3. Excepciones.....	45
2.3.1 Concepto de excepción.....	45
2.3.2 Excepciones en Java (Jerarquía).....	45
2.3.3 Manejo y captura.....	45
2.3.4 Excepciones personalizadas.....	45
2.3.5 Lanzamiento de excepciones.....	45
2.3.6 Aserciones.....	45

### 3. Actividades y ejercicios.....45



# 1. Ficha unidad didáctica.

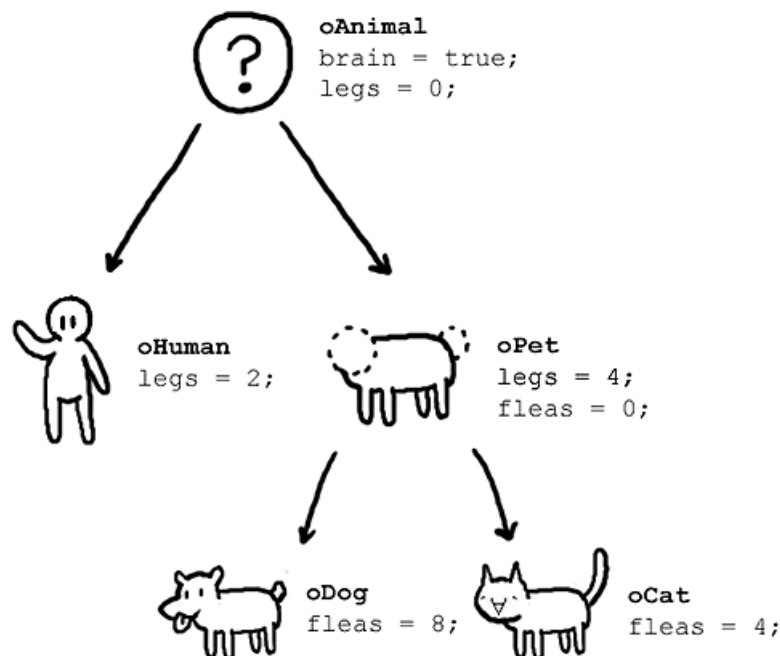
OBJETIVOS DIDÁCTICOS	
<p>OD1: Enunciar los tipos de herencia existentes.</p> <p>OD2: Exponer la características de las clases abstractas.</p> <p>OD3: Usar la herencia e interfaces de forma razonada dependiendo de los problemas a resolver.</p> <p>OD4: Elaborar jerarquía de clases que apliquen sobrecarga y sobreescritura métodos.</p> <p>OD5: Modificar y analizar código fuente que utilice herencia y jerarquía de clases.</p> <p>OD6: Reconocer las situaciones donde controlar excepciones.</p> <p>OD7: Elaborar software tolerante a fallos usando excepciones.</p> <p>OD8: Analizar código con uso de excepciones, o añadiendo estas para prever fallos.</p> <p>OD9: Desarrollar nuevas excepciones en función de las necesidades.</p> <p>EV1. Analizar el derecho a propiedad intelectual de programas, código y librerías, no haciendo un uso indebido o ilícito de estos.</p> <p>EV3. Caracterizar la importancia del trabajo en grupo en el ámbito empresarial.</p> <p>EV4. Valorar el trabajo por su calidad, no por la persona que lo realiza.</p> <p>RL2. Evaluar las consecuencias de una vida y trabajo sedentario para la salud.</p> <p>TIC2. Utilizar el uso de Internet como forma de actualización en nuevas tecnologías relacionadas con el ámbito laboral de la informática.</p>	
<b>RESULTADOS DE APRENDIZAJE</b>	RA3, RA7
CONTENIDOS	
<p>Relaciones entre clases. Composición.</p> <p>Herencia.</p> <p>Poliformismo.</p> <p>Herencia en otros lenguajes OO.</p> <p>UML y clases.</p> <p>Concepto de excepción.</p> <p>Excepciones en Java (Jerarquía)</p> <p>Manejo y captura.</p> <p>Excepciones personalizadas.</p> <p>Lanzamiento de excepciones.</p> <p>Aserciones.</p>	
ORIENTACIONES METODOLÓGICAS	
<p>Se hace ver al alumno la relación entre las actividades y su futura vida laboral.</p> <p>Relacionar los contenidos y actividades con otros módulos del ciclo.</p> <p>Las actividades guían a habituar a la documentación de los trabajos realizados y a la lectura y comprensión de documentación técnica.</p>	
<b>CRITERIO DE EVALUACIÓN</b>	3d, 7a,7b, 7c, 7d, 7e, 7f, 7g, 7h

## 2. Contenidos.

### 2.1. Introducción.

En el tema anterior se han tratado los principios de abstracción, encapsulación y ocultación de información de la programación orientada a objetos. Quedan por tratar la herencia y el polimorfismo.

La herencia es una de los principios más potentes de la POO, permite la reutilización de código y la gestión de conjunto de objetos diferentes pero con un nexo común, por ejemplo tener un conjunto de figuras geométricas, todas ellas con ciertos atributos y métodos comunes como “dibujar” pero otros diferentes y que es interesante administrar de forma conjunta por ejemplo un vector de figuras. La herencia va a permitir tener el código común a todas las clases en un único punto (clase padre) además de poder tratarlos como si de la clase padre se tratara a pesar de no ser así (por ejemplo un vector de objetos de la clase padre pero que almacena clases hijas).



Muy relacionado con la herencia se encuentra el polimorfismo, existiendo dos tipos, el primero de funciones o métodos que permite tener el mismo nombre de función o método

con parámetros diferentes.



**En los lenguajes con polimorfismo de funciones o métodos estos no se definen únicamente con su nombre sino también con el número de parámetros y el tipo de cada uno de ellos, que no han de coincidir en número y mismo orden de tipos.**

El otro tipo de polimorfismo es el de objeto, en la herencia se puede tener métodos en clases diferentes pero con una herencia común, el caso típico es la clase padre figura, y los hijos rectángulo y círculo, todos ellos con un método llamado pintar, si se tiene un vector de figuras que contiene círculos, rectángulos y figuras. ¿A qué método llamar al pintar? ¿Al de figura, al de círculo o al de rectángulo?. Esto es conocido como ligadura dinámica y se resuelve en tiempo de ejecución (cuando el programa se encuentra ejecutándose).

El desarrollo de software se ha estandarizado, pudiendo modelar el comportamiento de las clases y objetos entre otros conceptos por la relación entre las clases y por ende objetos, de forma que se pueda diseñar previamente las relaciones y comportamiento de las clases y los objetos y como interacciona con otros objetos del programa.

Por último se tratan las excepciones que permiten gestionar situaciones anómalas o errores en los programas, estas excepciones se basan en la existencia de una jerarquía de herencia.



¿Qué implicación tiene definir los atributos y métodos en las clases base como private en las clases que heredan? ¿Cómo solucionarlo?

## 2.2. POO Avanzada.

### 2.2.1 Herencia.

La herencia permite definir clases que hereden los atributos y métodos de otras clases de forma que no sea necesario volver a reescribir el código, esto permite conseguir dos objetivos:

Reutilización: No es necesario volver a escribir una y otra vez el mismo código en

diferentes clases, de igual forma el mantenimiento y depuración se centran en un único punto.

**Extensibilidad:** Permite ampliar la funcionalidad básica de una clase, manteniendo intacta la base, por ejemplo modelar un coche en una clase y definir a partir de este un coche 4x4 que tendrá un comportamiento similar excepto por ejemplo la aplicación de reductora.

Cuando una clase B herede de otra clase A entonces:

- B incorpora la estructura (atributos) y el comportamiento (métodos) de A.
- B puede:
  - Añadir nuevos métodos y atributos.
  - Cambiar la visibilidad de algunos métodos y/o atributos.
  - Redefinir o reescribir métodos de la clase A.

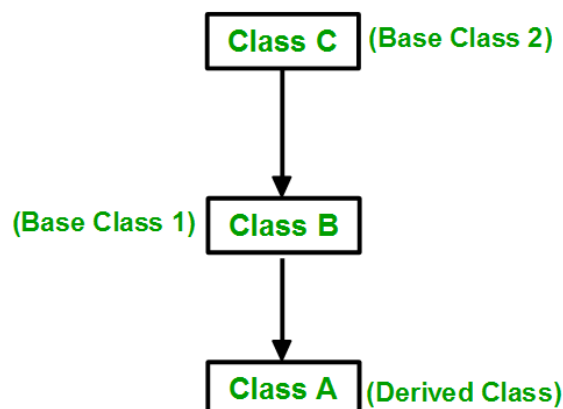
Se suele llamar a la clase A superclase y la la B subclase, en caso de existir una C que hereda de B, se llama a B y C subclases de A, B descendiente directo de A y C descendiente indirecto de A.



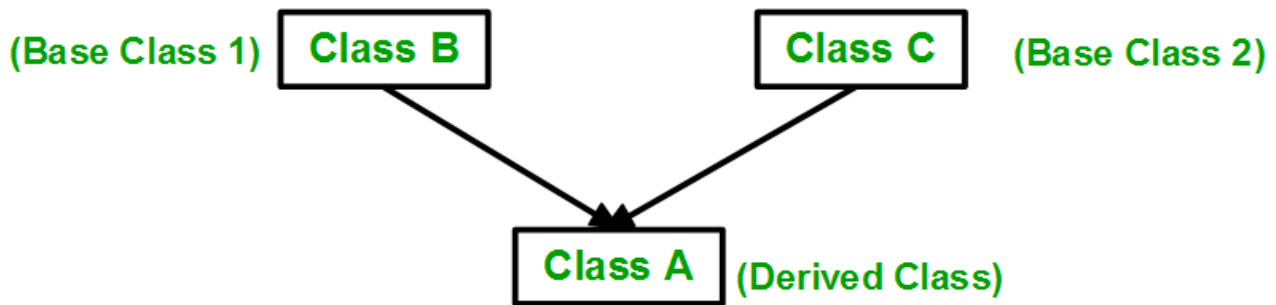
### 2.2.2 Tipos de herencia.

Existen 2 tipos de herencia en los lenguajes de programación, cada uno opta por una u otra opción y por supuesto cada una de ellas tiene sus ventajas y desventajas.

- **Herencia simple.** El caso más común en los lenguajes más modernos es la herencia simple, consiste en que una clase solo puede heredar de una y solo una clase. Java y C# implementan la herencia simple.



- **Herencia múltiple.** Las clase pueden heredar de más de una clase, en principio puede pensarse que es una ventaja, pero surgen algunos problemas en el que puede llegar a ser confuso y problemático la elección de qué método llamar al poder tomar diferentes caminos para llegar al método, además ¿que sucede si dos métodos o atributos se llaman igual en las clases de las que hereda?. El lenguaje típico de la herencia múltiple es C++



Por ejemplo, se desea crear un juego de estrategia, el juego se componen de unidades, cada unidad posee capacidades (atributos) como ataque, defensa, capacidad de trabajo, movilidad o velocidad, y toda unidad puede moverse, atacar o defenderse. Las unidades se clasifican en arqueros, infantería, caballería, artillería y trabajador.

La clase base se puede llamar unidad, con los atributos:

- #entero nivel\_ataque;
- #entero nivel\_defensa;
- #entero nivel\_trabajo;
- #entero nivel\_movilidad;
- #entero nivel\_velocidad;

Y los métodos:

- + boolean Desplazarse(int x, int y);
- + void Atacar(Unidad unidad);
- +void Defenderse();
- +Recurso Trabajar(Terreno r);



Las clases hijas serán Arquero, Infante, Caballero, Artillero y Trabajador, que heredan los atributos y métodos, añadiendo otros, por ejemplo el arquero ha de tener una distancia de



disparo y el redefinir el método atacar y defenderse, al igual que no puede trabajar.

Arquero hereda de Unidad.

- #entero distancia\_arco;
- #Arco[] arcosdisponibles;
- #Arco arco\_seleccionado;

Se crean los métodos:

- +boolean Dispara (int x, int y);
- +boolean Dispara(Unidad u);
- +void AnyadirArcho(Arco a);
- +void CambiarArco(int index);

Y se redefinen en:

- + boolean Desplazarse(int x, int y);
- + void Atacar(Unidad unidad);
- +void Defenderse();
- +Recurso Trabajar(Terreno r);

Un arquero se desplaza a una velocidad menor que un caballero, y no puede atravesar ciertos lugares.

Destacar que al usar herencia es necesario definir en el constructor de la clase hija una llamada al constructor de la clase base, dependiendo del lenguaje esto se puede hacer de forma **explícita** (se ha de llamar en el constructor de la clase derivada) con el nombre de la clase padre (C++) o con super(Java, Python) o se llama de forma **implícita**, llamando al constructor por defecto.

### 2.2.3 Clases y métodos abstractos.

Es posible indicar que algún método de la clase padre no sea implementado en esta, definiéndose este tipo de métodos como abstractos (no se pueden llamar ya que no existe implementación) y deberá ser las clases que hereden de esta las que implementen el método abstracto. En el ejemplo anterior, no tiene mucho sentido implementar el método

atacar en la clase Unidad.

También existe abstracción a nivel de clase, una clase abstracta es:

- La que se define como abstracta.
- La que posee al menos un método abstracto.



**No es posible crear objetos de clases abstractas ya que están incompletas, pero si declarar variables de ese tipo de clase abstracta.**



¿Qué utilidad puede tener definir variables de clases abstractas ? ¿Razonar si es necesario tener constructores en clases abstractas?

## 2.2.4 Herencia en Java.

Los conceptos básicos de herencia se han tratado en el punto anterior, cada lenguaje implementa de una u otra forma los conceptos anteriores. Java posee una herencia simple (solo es posible heredar de una clase base en la clase hija) aunque implementa mecanismos para simular la herencia múltiple.

### 2.2.4.1 Herencia básica.

Para extender de una clase se usa la palabra reservada **extends** y la clase de la que se hereda, además al crear el objeto que hereda se ha de llamar al constructor de la clase base, en caso de no hacerlo se llama al constructor por defecto.

Es importante destacar la **visibilidad de los métodos y atributos heredados**, en caso de que sean declarados en la clase base como privados, no estarán disponibles en la clase heredada.

Modificador (palabra reservada)	Misma clase	Mismo paquete	Subclase de otro paquete	Resto
<i>private</i>	✓	✗	✗	✗
(defecto)	✓	✓	✗	✗
<i>protected</i>	✓	✓	✓	✗
<i>public</i>	✓	✓	✓	✓

Para que sean visibles se han de definir como `protected` o `public`, no estando aconsejado `public` quedando únicamente `protected` para tener acceso a los métodos y atributos externos en caso de ser necesario.

Un ejemplo de herencia: Se desea automatizar la gestión de las máquinas de un gimnasio. A las máquinas poseen una API REST (enviar comandos usando JSON y HTTP). Se tiene las 3 tipos de máquinas:

- Cinta de correr.
- Bicicleta de spinning.
- Máquina de remo.

Todas las máquinas tienen:

- Fecha instalación.
- Versión del software instalado.
- Horas en funcionamiento.
- Identificador
- Estado. (Encendida, apagada...)
- Fechas de revisión. Máximo 50, posee la fecha y un comentario.
- `Encender()`.
- `Apagar()`.
- `Instalar()`;
- `Resetear()`.
- `Pintardisplay()`;

La clase `Maquina` será:

```
1. package pedro.ieslaencanta.com.maquinagimnasio;  
2.  
3. import java.util.Date;  
4.  
5. /**  
6.  *  
7.  * @author Pedro
```

```
8. */
9. enum EstadoMaquina {
10.     ENCENDIDA,
11.     APAGADA,
12.     AVERIADA,
13.     MANTENIMIENTO
14. }
15.
16. public class Maquina {
17.
18.     protected Date fecha_instalacion;
19.     protected String version_software;
20.     protected int horas_funcionando;
21.     protected Revision[] revisiones;
22.     protected EstadoMaquina estado;
23.     protected String identificador;
24.
25.     public Maquina() {
26.         System.out.println("Soy el constructor por defecto de
máquina");
27.         this.fecha_instalacion = new Date();
28.         this.revisiones = new Revision[50];
29.         this.estado = EstadoMaquina.APAGADA;
30.         this.identificador = "";
31.     }
32.
33.     public Maquina(String version, int horas) {
34.         System.out.println("Soy el constructor sobrecargado de
máquina");
35.         this.fecha_instalacion = new Date();
36.         this.revisiones = new Revision[50];
37.         this.version_software = version;
38.         this.horas_funcionando = horas;
39.         this.estado = EstadoMaquina.APAGADA;
40.         this.identificador = "";
41.     }
42.
43.     public void Encender() {
44.         this.estado = EstadoMaquina.ENCENDIDA;
45.     }
46.
47.     public void Apagar() {
```

```
48.         this.estado = EstadoMaquina.APAGADA;
49.     }
50.
51.     /**
52.      * Actualiza el software
53.      *
54.      * @param ruta
55.      * @param version
56.      */
57.     public void Instalar(String ruta, String version) {
58.         this.version_software = version;
59.         this.horas_funcionando = 0;
60.     }
61.     /**
62.      * pone el contador a 0
63.      */
64.     public void Resetear() {
65.         this.estado = EstadoMaquina.MANTENIMIENTO;
66.         this.horas_funcionando = 0;
67.         this.estado = EstadoMaquina.APAGADA;
68.     }
69.
70.     public String pintardisplay() {
71.         String vuelta;
72.         vuelta = "Máquina:" + this.identificador + " software:" +
this.version_software
73.             + " horas funcionamiento:" + this.horas_funcionando
+ " estado:" + this.estado;
74.         return vuelta;
75.     }
76.     /**
77.      * @return the fecha_instalacion
78.      */
79.     public Date getFecha_instalacion() {
80.         return fecha_instalacion;
81.     }
82.
83.     /**
84.      * @return the version_software
85.      */
86.     public String getVersion_software() {
87.         return version_software;
```

```
88.         }
89.
90.         /**
91.          * @return the horas_funcionando
92.          */
93.         public int getHoras_funcionando() {
94.             return horas_funcionando;
95.         }
96.
97.         /**
98.          * @return the revisiones
99.          */
100.        public Revision[] getRevisiones() {
101.            return revisiones;
102.        }
103.
104.        /**
105.         * @return the estado
106.         */
107.        public EstadoMaquina getEstado() {
108.            return estado;
109.        }
110.
111.        /**
112.         * @return the identificador
113.         */
114.        public String getIdentificador() {
115.            return identificador;
116.        }
117.
118.        /**
119.         * @param fecha_instalacion the fecha_instalacion to set
120.         */
121.        public void setFecha_instalacion(Date fecha_instalacion) {
122.            this.fecha_instalacion = fecha_instalacion;
123.        }
124.
125.        /**
126.         * @param version_software the version_software to set
127.         */
128.        public void setVersion_software(String version_software) {
```

```
129.         this.version_software = version_software;
130.     }
131.
132.     /**
133.      * @param horas_funcionando the horas_funcionando to set
134.      */
135.     public void setHoras_funcionando(int horas_funcionando) {
136.         this.horas_funcionando = horas_funcionando;
137.     }
138.
139.     /**
140.      * @param revisiones the revisiones to set
141.      */
142.     public void setRevisiones(Revision[] revisiones) {
143.         this.revisiones = revisiones;
144.     }
145.
146.     /**
147.      * @param estado the estado to set
148.      */
149.     public void setEstado(EstadoMaquina estado) {
150.         this.estado = estado;
151.     }
152.
153.     /**
154.      * @param identificador the identificador to set
155.      */
156.     public void setIdentificador(String identificador) {
157.         this.identificador = identificador;
158.     }
159. }
```

La clase Revision:

```
package pedro.ieslaencanta.com.maquinagimnasio;

import java.util.Date;

/**
 *
 * @author Pedro
 */
```

```
public class Revision {  
    private Date fecha;  
    private String comentario;  
    public Revision() {  
        this.fecha= new Date();  
        this.comentario=null;  
    }  
    public Revision(Date fecha, String comentario){  
        this.fecha=fecha;  
        this.comentario=comentario;  
    }  
  
    /**  
     * @return the fecha  
     */  
    public Date getFecha() {  
        return fecha;  
    }  
  
    /**  
     * @return the comentario  
     */  
    public String getComentario() {  
        return comentario;  
    }  
  
    /**  
     * @param fecha the fecha to set  
     */  
    public void setFecha(Date fecha) {  
        this.fecha = fecha;  
    }  
  
    /**  
     * @param comentario the comentario to set  
     */  
    public void setComentario(String comentario) {  
        this.comentario = comentario;  
    }  
}
```



Ahora se hereda y se implementa por ejemplo la cinta de correr, la cinta además de lo anterior a de tener:

- Programas. Cada programa se define como un vector de pares que tiene tiempo y potencia.
- Inclinación. Angulo de inclinación.
- Velocidad: Velocidad actual.
- Tiemposesion. Cuanto tiempo lleva el usuario o usuaria.
- Subirvelocidad()
- Bajarvelocidad();
- Subirinclinacion();
- BajarInclinacion();
- ParadaEmergencia();

La clase Cintacorrer es la siguiente:

```
1. package pedro.ieslaencanta.com.maquinagimnasio;
2.
3. /**
4.  *
5.  * @author Pedro
6.  */
7. public class Cintacorrer extends Maquina {
8.
9.     private ProgramaCinta programas[];
10.     private int velocidad;
11.     private int inclinacion;
12.     private int minutos_sesion;
13.     private ProgramaCinta programa_actual;
14.
15.     public Cintacorrer() {
16.         this.inclinacion = 0;
17.         this.velocidad = 0;
18.         this.setEstado(EstadoMaquina.APAGADA);
19.         this.programa_actual = null;
20.         System.out.println("Soy el constructor de la cinta");
```

```
21.         }
22.
23.     public void Subirvelocidad() {
24.         this.velocidad++;
25.     }
26.
27.     public void Bajarvelocidad() {
28.         this.velocidad--;
29.     }
30.
31.     public void Subirinclinacion() {
32.         this.inclinacion++;
33.     }
34.
35.     public void Bajarinclinacion() {
36.         this.inclinacion--;
37.     }
38.
39.     public void ParadaEmergencia() {
40.         this.estado = EstadoMaquina.APAGADA;
41.     }
42.
43.     public void SetPrograma(int index) {
44.         if (index > 0 && index < this.programas.length) {
45.             this.programa_actual = this.programas[index];
46.         }
47.     }
48.
49. }
```

Observar en la línea 7 como se hereda de la clase Maquina de forma que el código de la clase base se puede usar en la clase que hereda, por ejemplo en la línea 39.



¿Qué sucede si se declara el atributo estado como privado en la clase padre? ¿Es posible acceder a ese atributo en la clase hija? ¿Cómo solucionarlo?

Las otras clases desarrolladas Puntoprograma y programa son:

```
package pedro.ieslaencanta.com.maquinagimnasio;
```

```
/**
 *
 * @author Pedro
 */
public class Puntoprograma {
    private int tiempo;
    private int velocidad;
    public Puntoprograma() {
        this.tiempo=-1;
        this.velocidad=-1;
    }
    public Puntoprograma(int tiempo, int velocidad){
        this.tiempo=tiempo;
        this.velocidad=velocidad;
    }

    /**
     * @return the tiempo
     */
    public int getTiempo() {
        return tiempo;
    }

    /**
     * @return the velocidad
     */
    public int getVelocidad() {
        return velocidad;
    }

    /**
     * @param tiempo the tiempo to set
     */
    public void setTiempo(int tiempo) {
        this.tiempo = tiempo;
    }

    /**
     * @param velocidad the velocidad to set
     */
    public void setVelocidad(int velocidad) {
```

```
        this.velocidad = velocidad;
    }
}
```

```
package pedro.ieslaencanta.com.maquinagimnasio;

/**
 *
 * @author Pedro
 */
public class ProgramaCinta {
    private Puntoprograma puntos[];
    private String descripcion;
    public ProgramaCinta() {
        this.puntos= new Puntoprograma[20];
        this.descripcion="";
    }
    public ProgramaCinta(String descripcion){
        this.puntos= new Puntoprograma[20];
        this.descripcion=descripcion;
    }

    /**
     * @return the puntos
     */
    public Puntoprograma[] getPuntos() {
        return puntos;
    }

    /**
     * @return the descripcion
     */
    public String getDescripcion() {
        return descripcion;
    }

    /**
     * @param puntos the puntos to set
     */
}
```

```
public void setPuntos(Puntoprograma[] puntos) {  
    this.puntos = puntos;  
}  
  
/**  
 * @param descripcion the descripcion to set  
 */  
public void setDescripcion(String descripcion) {  
    this.descripcion = descripcion;  
}  
}
```



Queda como ejercicio implementar las máquinas restantes

Ahora ya es posible definir cintas de correr que heredan de máquina.


```
1. /**  
2.  *  
3.  * @author Pedro  
4.  */  
5. public class Principal {  
6.     public static void main (String args[]){  
7.         Cintacorrer cinta= new Cintacorrer();  
8.         System.out.println(cinta.pintardisplay());  
9.  
10.    }  
11. }
```


Observar en la línea 8 que se llama a pintardisplay de una cinta de correr, pero esta clase no implementa el método, pero si su clase padre, que es el método que realmente se ejecuta.

Al ejecutar se imprime por pantalla:

```
Soy el constructor por defecto de máquina  
Soy el constructor de la cinta  
Máquina: software:null horas funcionamiento:0 estado:APAGADA
```

Lo primero que se imprime es la frase que se ha puesto en el constructor por defecto de la clase Maquina y a continuación la frase de la cinta de correr, esto es debido a que **por defecto el constructor de la clase heredada llama al constructor por defecto de la clase de la que hereda.**

 Es conveniente llamar al constructor de la clase base con `super()` o en caso de querer llamar a un constructor sobrecargado, `super` y los parámetros, a pesar de que se llama al constructor por defecto.

 `Super` ha de ser la primera instrucción del constructor de la clase que hereda, en caso de que no se incluya esta llamada, Java lo incluye de forma automática y en caso de que la clase base no disponga de ese constructor dará error.

Por ejemplo en la clase `Maquina` se comenta el constructor por defecto, y al ejecutar el programa se obtiene:

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code
- Erroneous sym type: pedro.ieslaencanta.com.maquinagimnasio.Maquina.<init>
    at
pedro.ieslaencanta.com.maquinagimnasio.Cintacorrer.<init>(Cintacorrer.java:19)
    at Principal.main(Principal.java:15)
```

#### 2.2.4.2 Limitaciones a la herencia.

Con la palabra reservada **final** es posible limitar la herencia y no permite sobreescritura de la clase en la que se usa. Se puede usar tanto a nivel de clase (evita la herencia sobre esa clase) como a nivel de método (no permite la sobreescritura, se tratará en puntos posteriores).

A partir de Java 15, se definen las clases “**sealed**” en las que se puede indicar de forma explícita las clases a las que se permite heredad.

```
public sealed class Vehicle permits Car, Truck {
    protected final String registrationNumber;
    public Vehicle(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }
    public String getRegistrationNumber() {
        return registrationNumber;
    }
}
```

```
public final class Truck extends Vehicle {
    private final int loadCapacity;
```

```
public Truck(int loadCapacity, String registrationNumber) {  
    super(registrationNumber);  
    this.loadCapacity = loadCapacity;  
}  
public int getLoadCapacity() {  
    return loadCapacity;  
}  
public int getMaxServiceIntervalInMonths() {  
    return 18;  
}  
}
```

### 2.2.4.3 Interfaces e Implements.

En ocasiones es necesario que una clase pueda heredar de varias clases, pero Java no permite esto (C++ si), para posibilitar esta opción se definen las interfaces. Una interfaz básica se puede ver como un contrato que obliga a la clase que implementa esa interfaz a implementar los métodos definidos en dicha interfaz pudiendo implementar todas las interfaces que sean necesarias.

Para ello usar la palabra reservada implements a continuación del nombre de la clase o en caso de herencia despues de usa extends clasepadre, seguido de una lista separada por comas de las interfaces a implementar:

```
public class BicicletaSpinnig extends Maquina implements Interfaz1, Interfaz2 {  
    ...  
}
```

Una interfaz muy útil es la interfaz Comparable, que permite utilizar el método estático Array.Sort con objetos que implementan esa interfaz

## sort

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. **All elements in the array must implement the Comparable interface.** Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than  $n \lg(n)$  comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately  $n$  comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to  $n/2$  object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (*TimSort*). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

### Parameters:

`a` - the array to be sorted

### Throws:

`ClassCastException` - if the array contains elements that are not *mutually comparable* (for example, strings and integers)

`IllegalArgumentException` - (optional) if the natural ordering of the array elements is found to violate the `Comparable` contract

Al leer la interfaz Comparable en la documentación <https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html> se ve que define un único método que las clases han de implementar:

## Method Detail

### compareTo

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure  $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$  for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive:  $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0)$  implies  $x.\text{compareTo}(z) > 0$ .

Finally, the implementor must ensure that  $x.\text{compareTo}(y) == 0$  implies that  $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$ , for all `z`.

It is strongly recommended, but *not* strictly required that  $(x.\text{compareTo}(y) == 0) == (x.\text{equals}(y))$ . Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation  $\text{sgn}(\text{expression})$  designates the mathematical *signum* function, which is defined to return one of -1, 0, or 1 according to whether the value of *expression* is negative, zero or positive.

### Parameters:

`o` - the object to be compared.

### Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

### Throws:

`NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.

Siguiendo con el ejemplo anterior, se hace que la clase base implemente la interfaz Comparable, se utiliza la palabra reservada **implements** indicando el tipo de clase que se ha de comparar (se tratará en temas posteriores). Normalmente le IDE muestra un error al no estar los métodos de la interfaz definidos (la clase no cumple el contrato), al hacer click



en el error, el id crea de forma automática el método, se opta por ordenar por horas de funcionamiento, aunque es posible ordenar por cualquier atributo o conjunto de atributos que se considere:

```
public class Maquina implements Comparable<Maquina> {

    protected Date fecha_instalacion;
    protected String version_software;
    protected int horas_funcionando;
    protected Revision[] revisiones;
    protected EstadoMaquina estado;
    protected String identificador;

    public Maquina() {
        System.out.println("Soy el constructor por defecto de máquina");
        this.fecha_instalacion = new Date();
        this.revisiones = new Revision[50];
        this.estado = EstadoMaquina.APAGADA;
        this.identificador = "6666";
    }


    public Maquina(String version, int horas) {
        System.out.println("Soy el constructor sobrecargado de máquina");
        this.fecha_instalacion = new Date();
        this.revisiones = new Revision[50];
        this.version_software = version;
        this.horas_funcionando = horas;
        this.estado = EstadoMaquina.APAGADA;
        this.identificador = "";
    }

    @Override
    public int compareTo(Maquina o) {
        return this.horas_funcionando-o.horas_funcionando;
    }
}
```

La clase principal, en el que se crea un array de máquinas y se ordena de forma sencilla al implementar la interfaz:


```
public class Principal {
    public static void main (String args[]){
        Maquina m[]= new Maquina[3];
        m[0]= new Cintacorrer();
    }
}
```

```
m[0].setHoras_funcionando(1000);  
m[1]= new Cintacorrer();  
m[1].setHoras_funcionando(2000);  
m[2]= new Cintacorrer();  
m[2].setHoras_funcionando(500);  
//se llama a ordenar con el algoritmo ya implementado  
Arrays.sort(m);  
//se muestran los resultados  
for(int i=0;i<m.length;i++)  
    System.out.println(m[i].pintardisplay());  
}  
}
```

 Durante el curso se ha indicado que no es posible añadir a un vector definido de un tipo una variable de otro tipo, pero en este caso se añade a un vector de tipo máquina objetos de tipo Cintacorrer. ¿Cómo es posible?

Algunas de las interfaces más utilizadas en Java son:

- Serializable. Permite pasar un objeto a un flujo de bytes, para almacenar o transmitir por red.
- Cloneable. Para clonar objetos.
- Observable. Modelo vista-controlador y también la base para los eventos (temas posteriores).
- Iterable. Permite recorrer objetos con la instrucción foreach, por ejemplo con vectores de objetos (ya usados en el curso).
- Runnable. Para hacer que un objeto se convierta en un hilo (funciona como un proceso/hilo independiente).

 ¿Qué métodos son necesarios implementar en caso de hacer una clase serializable?

Ya es posible usar interfaces definidas previamente, pero es posible también definir las propias interfaces, es necesario crear un nuevo fichero sustituyendo la palabra reservada class por interface, a continuación definir los métodos que ha de implementar la clase que implementen la interfaz. Se pueden definir las interfaces y los métodos con los modificadores de acceso tratados en puntos anteriores. Por ejemplo se desea crear un

programa para reproducir ficheros multimedia, estos pueden ser Audio, Video o Imágenes de cualquier tipo, cada una de las clases ya heredan de otras clases por lo que en Java no es posible heredar de una segunda, y se desea poder realizar las opciones básicas de un reproductor:

```
1. public interface ReproductorMultimedia {
2.     public enum EstadoReproductor{
3.         REPRODUCIENDO,
4.         PARADO,
5.         ERROR
6.     }
7.     public void start();
8.     public void stop();
9.     public EstadoReproductor state();
10.
11. }
```

Ahora en las clases concretas simplemente usar la palabra reservada implements e implementar los métodos concretos.

Indicar que es posible implementar más de una interfaz en una clase, por ejemplo, si se desea que el reproductor sea un hilo e implemente la interfaz ReproductorMultimedia además de heredar de una imaginaria clase ReproductorBase:

```
public class ReproductorAudio extends ReproductorBase implements
ReproductorMultimedia, Runnable{

    @Override
    public void start() {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }

    @Override
    public void stop() {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }

    @Override
    public EstadoReproductor state() {
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }
}
```

```
}  
  
@Override  
public void run() {  
    throw new UnsupportedOperationException("Not supported yet."); //To  
change body of generated methods, choose Tools | Templates.  
}  
}
```

A partir de Java 8 es posible definir métodos por defecto en las interfaces, es decir, en caso de no implementarse en las clases que lo implementas se usa el método definido en la interfaz. Siguiendo con el ejemplo anterior, se quiere poder mostrar un mensaje con información del estado del reproductor, en Java 8 se puede definir un método que las clases podrán o no implementar:

```
default String mensaje(){ return "reproductor...";
```

De igual forma es posible definir métodos estáticos a nivel de interfaz, pero solo puede hacer referencia a otros métodos estáticos y a los parámetros que se le pasa en la llamada:

```
1. public static int metodo_estadico(int valor){  
2.     return valor*2;  
3. }
```

#### 2.2.4.4 Clases y métodos abstractos .

Una de los principios de la POO es la capacidad de abstracción y reusabilidad de código usan entre otra la herencia. Al realizar la labor de abstracción puede suceder que sea necesario definir clases que no pueden ser instanciadas ya que no tiene sentido. El ejemplo clásico es el de un programa de dibujo, en el que se tiene círculos, elipses cuadrados, rectángulos o rutas entre otros , todas estas clases heredan de una clase principal llamada Figura, que define el método void pintar() y String toSVG() pero que no tiene sentido poder crear un objeto de la clase Figura y tampoco llamar al método pintar.

Para estos casos se definen las clases abstractas, que son clases que no pueden ser instanciadas, es decir no se pueden crear objetos si la clase es abstracta.

Una clase es abstracta si se cumple al menos uno de los siguientes casos:

1. Se declara la clase con la palabra reservada abstract.

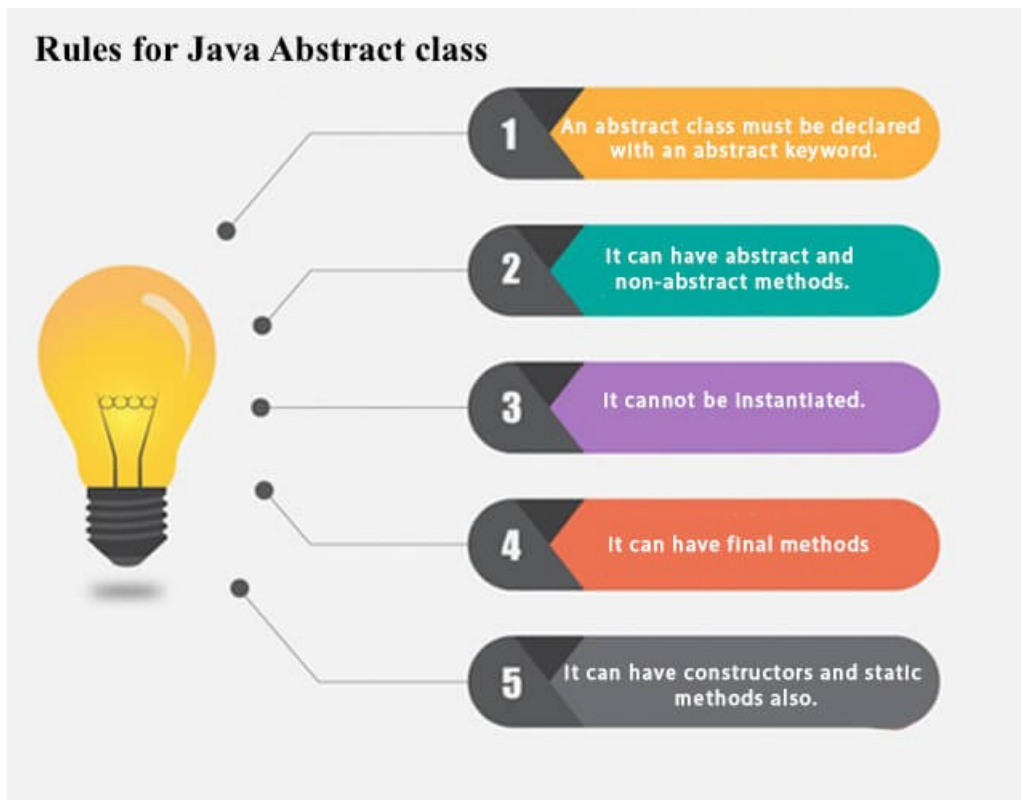
2. Posee al menos un método definido con la palabra reservada `abstract`.
3. Hereda de alguna clase que sea abstracta y no implementar todos los métodos definidos como abstractos en la clase padre.

Una mezcla de los casos 1 y 2:

```
public abstract class Figura {  
    protected String name;  
    protected int x, y;  
    public Figura() {  
        this.x=-1;  
        this.y=-1;  
        this.name="Figura";  
    }  
    public void moverA(int x, int y){  
        this.x=x;  
        this.y=y;  
    }  
    public String aSVG(){  
        return "Pendiente implementar en las clases hijas";  
    }  
    public abstract void pintar();  
}
```

Cuando se intenta crear un objeto de esa clase el resultado es:

```
error: Figura is abstract; cannot be instantiated  
    Figura f= new Figura();
```



Ahora se puede crear clases que hereden de la clase abstracta y poder hereda, por ejemplo la clase rectángulo:

```
public class Rectangulo extends Figura{  
    protected int alto;  
    protected int ancho;  
    public Rectangulo(){  
        super();  
        this.name="Rectangulo";  
    }  
    public Rectangulo(int alto,int ancho, int x,int y){  
        super();  
        this.alto=alto;  
        this.ancho=ancho;  
        this.x=x;  
        this.y=y;  
        this.name="Rectangulo";  
    }  
    public String aSVG(){  
        return "<rect x='"+this.x+"' y='"+this.y+"' width='"+this.ancho+"' height='"+this.alto+"' style='fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)' />";  
    }  
}
```

```
@Override
public void pintar() {

}

public static void main(String[] args) {
    Figura f= new Rectangulo(45,45,5,5);
    System.out.println(f.aSVG());
}
}
```

Observar como se define una variable de tipo Figura llamada f y es posible asignarla a un objeto de tipo Rectangulo y llamar a los métodos abstractos pero que funcionan ya que se han definido en la clase que hereda.



**En la clase abstracta anterior se han definido los atributos como protected.**

**¿Qué implica en relación con la herencia que se declaren private?**

La salida del programa es:

```
<rect x='5' y='5' width='450' height='45' style='fill:rgb(0,0,255);stroke-
width:3;stroke:rgb(0,0,0)' />
```



Vídeo de la UPV sobre herencia en Java. [https://www.youtube.com/watch?](https://www.youtube.com/watch?v=MG0hOuk6fqU)

[v=MG0hOuk6fqU](https://www.youtube.com/watch?v=MG0hOuk6fqU)



Ejercicio práctico. Implementar la clase cuadrado heredando de rectángulo, la clase elipse heredando de Figura, la clase círculo heredando de elipse. Implementar las interfaces necesarias para ordenar las figuras según su coordenadas siendo menores la que estén más arriba en la pantalla y las que estén en la misma coordenada y la que estén más a la izquierda. Crear un programa sencillo con un vector de figuras y varias figuras, imprimir por pantalla el fichero en SVG y probar en un navegador que funciona. Proteger la clase Cuadrado para que no se puede heredar de esta.

## 2.2.4.5 Polimorfismo.

### 2.2.4.5.1. Clase Object

Java posee una implementación curiosa (aunque otros lenguajes como C# también lo implementan), por defecto y de forma implícita **toda clase que no herede de otra hereda**

por defecto de la clase **Object**, y si hereda de otra también hereda de **Object**. Esto permite a Java gestionar los objetos de forma más sencilla y usar los métodos en la VM.

Los métodos que implementa son:

**Object clone()**

void notifyAll()

**boolean equals()**

**String toString()**

void finalize()

void wait()

**Class<?>getClass()**

void wait(long timeout)

**int hashCode()**

void wait(long timeout,int nanos)

void notify()

De los anteriores los más destacados son:

**Clone**, permite crear una copia del objeto, por defecto el método es protected, con lo que no es posible usarlo por defecto, para poder usarlo es necesario reescribirlo en la clase y hacerlo público, el caso más sencillo es:

```
public class Rectangulo extends Figura{
    protected int alto;
    protected int ancho;
    public Rectangulo(){
        super();
        this.name="Rectangulo";
    }
    public Rectangulo(int alto,int ancho, int x,int y){
        super();
        this.alto=alto;
        this.ancho=ancho;
        this.x=x;
        this.y=y;
        this.name="Rectangulo";
    }
    public String aSVG(){
        return "<rect x='"+this.x+"' y='"+this.y+"' width='"+this.ancho+"0'
height='"+this.alto+"' style='fill:rgb(0,0,255);stroke-
width:3;stroke:rgb(0,0,0)' />";
    }
    public Object clone(){
        return this;
    }
}
```



```
}  
  
@Override  
public void pintar() {  
    }  
  
public static void main(String[] args) {  
    Figura r= new Rectangulo(45,45,5,5);  
    Figura r_igualado=r;  
    Figura r_clonado=(Figura) ((Rectangulo)r).clone();  
    System.out.println("Al realizar el igual el original es "+r.hashCode()  
+" y la igualdad "+r_igualado.hashCode());  
    System.out.println("Al realizar la clonacion el original  
es"+r.hashCode()+" y la clonación "+r_clonado.hashCode());  
    }  
}
```

Al ejecutar el código se puede ver que los 3 objetos son los mismos:

```
Al realizar el igual el original es 990368553 y la igualdad 990368553  
Al realizar la clonacion el original es990368553 y la clonación 990368553
```

Es recomendable usar la interfaz cloneable, para implementar el método, mencionar además que se copian las referencias, este caso se denomina **copia superficial**, y se limita a asignar las referencias. Al usar este tipo de clonación se produce el efecto conocido como **Aliasing** que consiste en la modificación de objetos referenciados desde diferentes puntos sin desearlo.

En la clonación por defecto los datos primitivos se copia el valor y en el caso de objetos se copian referencias, pudiendo producirse el efecto aliasing.

Se modifica el ejemplo anterior para que la posición X,Y sea un objeto y ver como funciona la copia por defecto superficial. La clase Coordenada:

```
public class Coordenada {  
    private int x;  
    private int y;  
    public Coordenada() {  
    }  
    public Coordenada(int x,int y){  
        this.x=x;  
        this.y=y;  
    }  
  
    /**
```

```
* @return the x
*/
public int getX() {
    return x;
}

/**
 * @return the y
 */
public int getY() {
    return y;
}

/**
 * @param x the x to set
 */
public void setX(int x) {
    this.x = x;
}

/**
 * @param y the y to set
 */
public void setY(int y) {
    this.y = y;
}
}
```

La modificación en la clase Figura, y por tanto en las clases que heredan de la misma:

```
public abstract class Figura {
    protected String name;
    protected Coordenada coordenada;
    public Figura() {
        this.coordenada= new Coordenada() ;
        this.name="Figura";
    }
    public void moverA(int x, int y){
        this.coordenada= new Coordenada() ;
        this.coordenada.setX(x) ;
        this.coordenada.setY(y) ;
    }
}
```

```
public String aSVG() {  
    return "Pendiente implementar en las clases hijas";  
}  
  
public abstract void pintar();  
}
```



Se ha modificado parte de la implementación de la clase base. ¿Es necesario modificar algo en las clases que heredan de la clase Figura?

Ahora se clona el objeto con el método de la clase Object y se muestra la referencia a la coordenada:

```
public class Rectangulo extends Figura implements Cloneable{  
    protected int alto;  
    protected int ancho;  
    public Rectangulo(){  
        super();  
        this.name="Rectangulo";  
    }  
    public Rectangulo(int alto,int ancho, int x,int y){  
        this.alto=alto;  
        this.ancho=ancho;  
        this.coordenada.setX(x);  
        this.coordenada.setY(y);  
        this.name="Rectangulo";  
    }  
    public String aSVG(){  
        return "<rect x='"+this.getCoordenada().getX()+"'  
y='"+this.getCoordenada().getY()+"' width='"+this.ancho+"0'  
height='"+this.alto+"' style='fill:rgb(0,0,255);stroke-  
width:3;stroke:rgb(0,0,0)' />";  
    }  
    @Override  
    protected Object clone() {  
        Object tempo=null;  
        try {  
            tempo= super.clone();  
        } catch (CloneNotSupportedException ex) {  
            Logger.getLogger(Rectangulo.class.getName()).log(Level.SEVERE,  
null, ex);  
        }  
        return tempo;  
    }  
}
```

```
@Override
public void pintar() {
    }

    public static void main(String[] args) {
        Figura r= new Rectangulo(45,45,5,5);
        Figura r_clonado=(Figura) ((Rectangulo)r).clone();
        System.out.println("Al realizar la clonacion la coordenada original
es"+r+" y la coordenada clonación "+r_clonado);
        //efecto aliasing
        r.getCoordenada().setX(77);
        r.getCoordenada().setY(100);
        System.out.println(r_clonado.aSVG());
    }
}
```

La salida del programa, teniendo en cuenta que se ha modificado la coordenada del objeto original, no del objeto clonado, imprimiendo los valores para SVG:

```
<rect x='77' y='100' width='450' height='45' style='fill:rgb(0,0,255);stroke-
width:3;stroke:rgb(0,0,0)' />
```

Para evitar el aliasing se han copiar de forma manual los atributos que sean objetos, ya sea de forma manual o implementando en las clases de estos objetos el método clon de copia profunda, definiéndose copia profunda como la copia de los valores y no de las referencias de los objetos contenidos. En el ejemplo anterior, se reescribe el método clone:

```
protected Object clone() {
    Object tempo = new
Rectangulo(this.alto,this.ancha,this.getCoordenada().getX(),this.getCoordenada(
).getY());
    return tempo;
}
```

Al ejecutar de nuevo el código:

```
<rect x='5' y='5' width='450' height='45' style='fill:rgb(0,0,255);stroke-
width:3;stroke:rgb(0,0,0)' />
```

Se puede observar que ya no se produce el efecto de aliasing.

**Equals**. Otro método de la clase Object que todos los objetos implementan por defecto y que devuelve si un objeto es igual a otro, el método de la clase Object usa a su vez el método hashCode, comparando el hash de hashCode. En el ejemplo anterior, al hacer un equals de las dos variables de tipo rectángulo :

```
public static void main(String[] args) {  
    Figura r = new Rectangulo(45, 45, 5, 5);  
    Figura r_clonado = (Figura) ((Rectangulo) r).clone();  
    System.out.println(r.equals(r_clonado));  
}
```

Siendo la salida:

```
false
```

Ambos objetos son iguales en valores pero no en referencia que es lo que se utiliza para comparar.

Para poder comparar valores y no referencias, se sobreescribe en la clase Rectangulo el método boolean equals(Object o).

```
1. @Override  
2. public boolean equals(Object o) {  
3.     boolean vuelta = false;  
4.     if (!(o instanceof Rectangulo)) {  
5.         vuelta = false;  
6.     } else {  
7.         Rectangulo r=(Rectangulo) o;  
8.         if(r.alto==this.alto && r.ancho==this.ancho &&  
           r.coordenada==this.coordenada)  
9.             vuelta=true;  
10.    }  
11.    return vuelta;  
12. }
```

Ahora al ejecutar se obtiene el resultado:

```
false
```



**Sigue evaluándose a falso, ¿qué está sucediendo? ¿cómo solucionarlo?**

En el método se ha usado el operador **instanceof** que permite devuelva cierto si el objeto que aparece en la parte izquierda es, extiende o implementa la clase colocada en la parte derecha. En el siguiente código se puede observar su uso sobre un objeto de tipo Rectangulo:

```
1. Figura r = new Rectangulo(45, 45, 5, 5);  
2. System.out.println(r instanceof Rectangulo);  
3. System.out.println(r instanceof Figura);  
4. System.out.println(r instanceof Cloneable);
```

```
5. System.out.println(r instanceof Runnable);
```

Con la salida:

```
true  
true  
true  
false
```

**ToString**. Cuando se concatena un objeto con una cadena, lo que realmente sucede es que se llama al método public String toString() de la clase Object que devuelve el paquete y el nombre de la clase junto con el hash producido por el método hashCode()

```
herencia.Rectangulo@41629346
```

Esto aporta poca información, además en caso de querer mostrar por pantalla, en un fichero o un documento no resulta útil. Es posible sobrescribir el método toString:

```
@Override  
public String toString() {  
    return "Es un rectangulo de nombre:"+this.name+ " con  
    alto:"+this.alto+", ancho:"+this.ancho+ " y posición: "+this.coordenada;  
}
```

La salida es:

```
Es un rectangulo de nombre:Rectangulo con alto:45, ancho:45 y posición:  
herencia.Coordenada@10f87f48
```



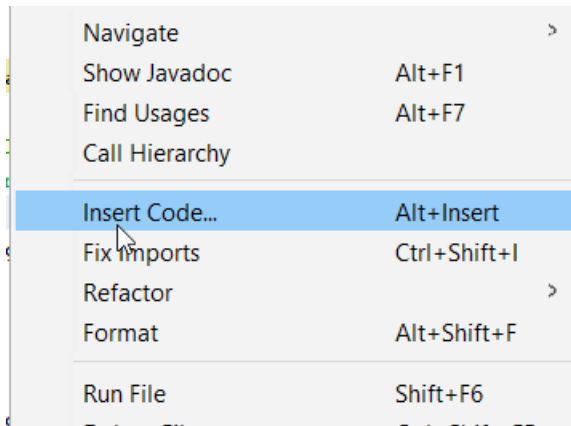
**¿Cómo evitar que las coordenadas aparezcan con el paquete, la clase y el hash?. ¿La solución es válida en caso de tener un vector de 1000 coordenadas? ¿Y si se cambia el sistema de coordenadas?**

**GetClass**. Otro de los métodos que implementa la clase Object, permite obtener diferente información sobre la clase, no sobre el objeto. Devuelve una clase genérica (se trata en temas posteriores) con información variada como el nombre de la clase, las clases de las que hereda, obtener el método constructor, listado de métodos, atributos, interfaces...

Es interesante el método newInstance que crea un objeto de la clase, a partir de la versión 9 se considera "deprecated", utilizándose en primer lugar el método que obtiene el constructor de la clase y a continuación newInstance.

HashCode. Este método devuelve un entero con la función resumen de un objeto (bytes), se usa por defecto al compara objetos con equals y es recomendable sobrescribir ambos para que la comparación de los objetos sea lo más eficiente posible. Existen diferentes

formas de crear métodos hashCode personalizados, incluso los IDE's poseen funcionalidades para crear de forma automática dichas funciones resumen



```
@Override
public int hashCode() {
    int hash = 5;
    hash = 23 * hash + this.alto;
    hash = 23 * hash + this.ancho;
    return hash;
}
```

No se explica en profundidad la generación de funciones Hash, ya que no es el objetivo del módulo.

#### 2.2.4.6 Variables poliformicas.

Una de las ventajas de la herencia es que es posible definir variables del tipo base y que referencien a objetos de tipos que heredan de esta clase, por supuesto no es posible llamar a métodos o atributos de la clase que hereda, solo de la clase de la que se define la variable.




Vídeo de la UPV sobre polimorfismo <https://www.youtube.com/watch?v=8HDLKH3KF2U>

En el siguiente código se añade un método a la clase Rectangulo que hereda de figura, se crea una variable de tipo Rectangulo y se referencia con new un nuevo objeto, llamando a ese nuevo método, también se referencia al objeto con una variable de tipo figura y se observa como no es posible llamar al método que no existe en la clase Figura, pero si al resto de métodos definidos en esta.:

```
public class Rectangulo extends Figura implements Cloneable {
    ...
    public void SoloclaseRectangulo() {
        System.out.println("Solo se puede llamar desde el rectangulo");
    }

    public static void main(String[] args) {
```

```
Rectangulo r = new Rectangulo(45, 45, 5, 5);  
//esto se puede hacer ya que Rectangulo hereda de f  
Figura f= r;  
r.SoloclasseRectangulo();  
//no es posible usar f.SoloclasseRectangulo ya que no existe  
//f.SoloclasseRectangulo();  
  
//si se hace un cast es posible, aunque peligroso, ya que puede que f  
no sea un rectangulo en memoria  
//por ejemplo puede ser un círculo  
((Rectangulo)f).SoloclasseRectangulo();  
}  
}
```

 Vídeo de la UPV sobre enlaces estáticos y dinámicos.  
<https://www.youtube.com/watch?v=y2eCjadS8x8>

#### 2.2.4.7 Llamando a métodos de la clase padre.

Es posible llamar a métodos de la clase padre desde métodos de las clases hijas, usando la palabra `super`, que referencia a la clase base. En la clase `Rectangulo` en el método `toString` se puede llamar a la clase base `Figura`:

```
@Override  
public String toString() {  
    return "Se puede llamar al padre:"+super.toString()+" Es un rectangulo  
de nombre:" + this.name + " con alto:" + this.alto + ", ancho:" + this.ancho +  
" y posición: " + this.coordenada;  
}
```

La salida al usar el método es:

```
Se puede llamar al padre:herencia.Rectangulo@41629346 Es un rectangulo de  
nombre:Rectangulo con alto:45, ancho:45 y posición:  
herencia.Coordenada@7291c18f
```

Las llamadas a la clase base se suelen hacer cuando parte de la funcionalidad del método de la clase hija ya se ha realizado EN PARTE en la clase padre, con lo que no es necesario volver a escribir todo el código.



### 2.2.4.8 Anotaciones.

En el desarrollo del tema se ha utilizado en los métodos sobreescritos de las clases hijas el símbolo @ y la palabra override (anular en inglés), siendo esta un tipo de anotación incorporada.

Una anotación añade información extra (metadatos) que puede ser usada en tiempo de ejecución y compilación, como puede ser indicaciones al compilador acciones a desarrollar, realizar tareas adicionales o definir configuraciones sin usar ficheros entre muchas otras opciones.

Su formato es: @nombreanotacion (parametro="valor", parametro="valor")

Se definen múltiples anotaciones, las predefinidas en el paquete java.lang son:

- @Deprecated, indica que ese método está en desuso y puede ser que en próximas versiones ya no se encuentre disponible.
- @Override, informa al compilador de que ha de sobrescribir el método en la clase que hereda.
- @SafeVarargs, aunque no se ha tratado en el curso, es posible pasar un conjunto indeterminado de parámetros a un método, esto provoca un aviso por parte del compilador, para evitar que se produzca este aviso se usa esta anotación.

```
@SafeVarargs
public final void safe(T... toAdd) {
    for (T version : toAdd) {
        versions.add(version);
    }
}
```

- @FunctionalInterface. Relacionado con las funciones lambda y la programación funcional, no tratados en este tema, permite desactivar los avisos del compilador. Esta anotación indica que la interfaz posee un solo método abstracto y que es una interfaz funcional.
- @SuppressWarnings. Desactiva para ese método o variables algunos avisos de compilación como variable no utilizada, puede recibir una lista de parámetros que indique que advertencias no mostrar.

```
@SuppressWarnings("unused") public void foo() {
```

```
String s;  
}
```

Algunos parámetros:

- all para suprimir todos los avisos
- boxing para suprimir los avisos relativos a las operaciones de empaquetado/desempaquetado
- cast para suprimir avisos relativos a las operaciones de conversión temporal
- dep-ann para suprimir avisos relativos a las anotaciones de desuso
- deprecation para suprimir avisos relativos al desuso
- fallthrough para suprimir avisos relativos a los enlaces faltantes en las sentencias switch
- finally para suprimir avisos relativos al bloque finally que retornan
- hiding para suprimir avisos relativos a locales que ocultan la variable

Es posible crear anotaciones propias con la palabra reservada `@interface` en vez de `class` o `interface`, y seguir algunas normas como que los valores devueltos han de ser primitivos, cadenas y/o arrays, además los métodos pueden tener valores por defectos.


Con estas anotaciones es posible tomar decisiones sobre la compilación o en tiempo de ejecución dependiendo de en que lugar sea visible (Código, compilación o ejecución). Un ejemplo sencillo extraído de la documentación oficial de Java:

```
@MetadataDefinition  
@Name("com.oracle.Severity")  
@Label("Severity")  
@Description("Value between 0 and 100 that indicates " +  
    "severity. 100 is most severe.")  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ ElementType.TYPE })  
public @interface Severity {  
    int value() default 50;  
}
```

Muchos frameworks definen sus propias anotaciones, usándose para configurar diferentes aspectos, del mismo desplazando a la definición de esta configuración en ficheros, por ejemplo Spring Framework( uno de los más usados en Java y entorno empresarial para

Web) define algunas como:

- @Required
- @Autowired
- @Qualifier
- @Configuration
- @Bean
- ...

 Introducción a la configuración de Spring en OpenWebinars.  
<https://www.youtube.com/watch?v=yWDxgzDZzwo>

## 2.2.5 Herencia en otros lenguajes OO.

La herencia es ampliamente utilizada en los lenguajes mas usados en la actualidad.

El primer lenguaje orientado a objetos de gran uso fue C++, se caracteriza por:

- Herencia múltiple, es decir una clase puede heredar de más de una clase al mismo tiempo.
- Clases abstractas.
- A los métodos no implementados se les denomina virtuales.
- Se han de definir destructores y en ellos se ha de llamar de forma explícita al destructor de la clase base.
- No define interfaces.

Otro de los lenguajes más usados es C#

- No soporta la herencia múltiple.
- Soporta interfaces.
- Notación similar a C++.
- Posible definir clases y métodos abstractos.
- Para referirse a métodos del padre se usa base y no super().

Python también soporta herencia con

- Soporta herencia múltiple.
- Puede tener clases y métodos abstractos.
- Las interfaces se pueden simular con clases abstractas.
- Para llamar a métodos del padre se usa la palabra super, en caso de tener varias clases de las que hereda, se utiliza el nombre de la clase.

Javascript en las primeras versiones utiliza herencia basada en prototipos, un verdadero dolor de cabeza, a partir del 2015 con ES6 se cambia a un desarrollo más tradicional.

- Usa la palabra reservada extends
- No soporta interfaces.
- Permite herencia múltiple
- Para llamar a métodos de la clase padre se utiliza la palabra super, pero con precedencia.
- No posee clases abstractas.
- No posee métodos abstractos.

## 2.2.6 Relaciones entre clases y UML.

Las clases se relacionan entre si, colaboran, se componen o asocian entre ellas. Se ha desarrollado un lenguaje propio que permite el modelado de sistemas orientados a objetos, definiendo diferentes diagramas agrupados en estructurales y de comportamiento.

Estructurales: Clases, componentes, despliegue, objetos, paquetes, perfiles.

Comportamiento: Actividades, uso, estados, secuencia, comunicación, tiempo e interacciones.

Todos son importantes, pero uno de los principales es el diagrama de clases que permite ver que clases componen la aplicación y/o librerías, con los atributos principales, los métodos, su visibilidad y cómo se relacionan entre ellos.

### 2.2.6.1 UML y clases.

Una clase se define en UML con un rectángulo y dependiendo de la precisión del diagrama de clases con todos los atributos y los métodos que lo componen junto con los parámetros de cada uno de ellos y los valores devueltos.

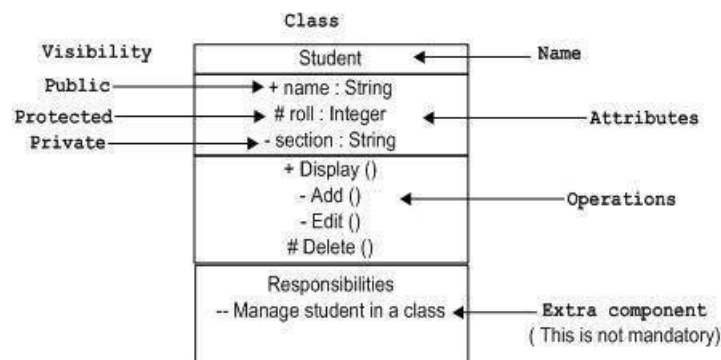
Los modificadores de acceso se definen con los símbolos:

+ Público, el elemento será visible tanto desde dentro como desde fuera de la clase.

- Privado, el elemento de la clase solo será visible desde la propia clase.

#Protegido, el elemento no será accesible desde fuera de la clase, pero podrá ser manipulado por los demás métodos de la clase o de las subclases.

~ paquete/defecto, define la visibilidad del paquete que puede ser utilizada por otra clase mientras esté en el mismo paquete



Diseñar el UML de la clase Figura con componentes x e y en entero, buscar información de cómo se expresan las clases abstractas.

Las relaciones entre clases son:

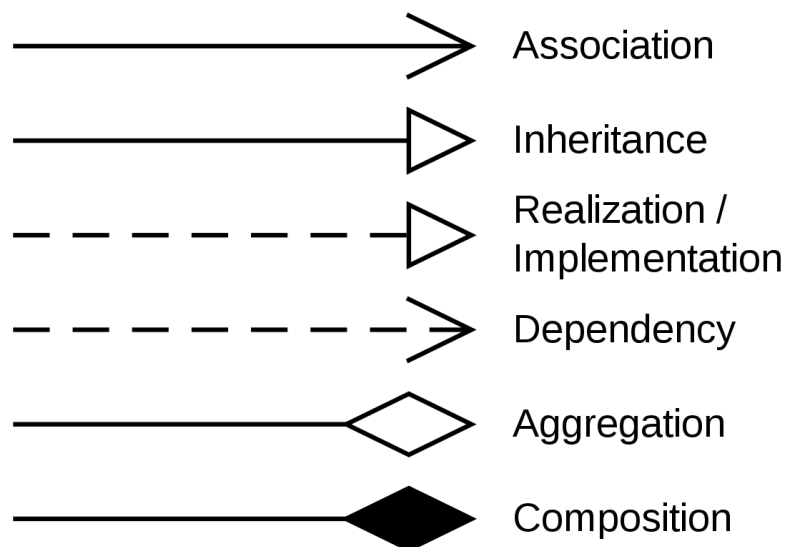
- Asociación.
- Agregación.
- Composición
- Generalización/especialización.

Y al igual que el modelo entidad relación posee cardinalidades:

- 1 Uno y sólo uno

- 0..1 Cero o uno
- N..M Desde N hasta M
- \* Cero o varios
- 0..\* Cero o varios
- 1..\* Uno o varios (al menos uno)

La cardinalidad se puede indicar en ambos extremos de la relación, y pueden indicarse el rol que cumple dicha cardinalidad en cada clase. Además las terminaciones de las relaciones dan información del tipo y navegabilidad de la relación:

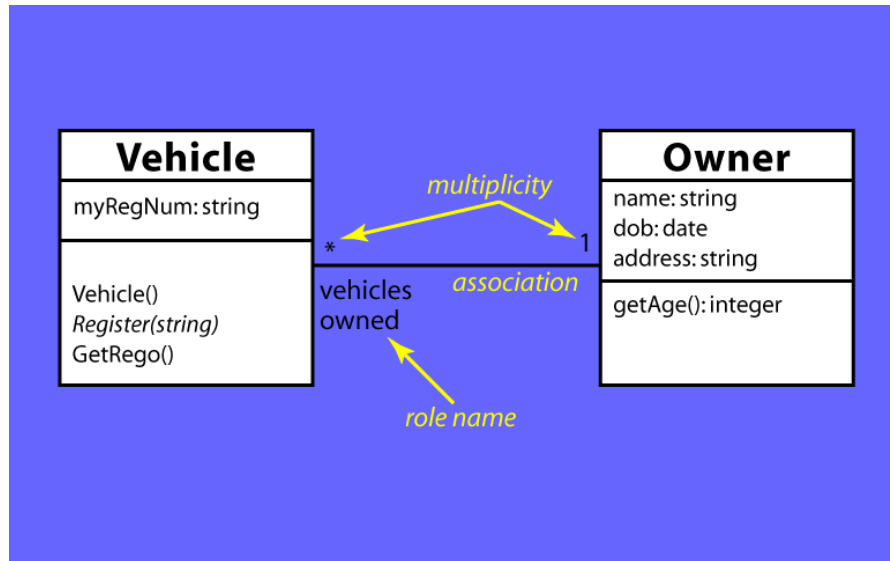


### 2.2.6.2 Asociación.

La primera de las relaciones, es estructura y describe una conexión entre clases, por ejemplo entre Cliente y Dirección, un cliente vive en una dirección, o una Cuenta bancaria y un Cliente, pudiendo establecer cardinalidades.

En el caso bancario un cliente puede tener varias cuentas y una cuenta pertenecer a un único cliente. Estableciendo también la navegabilidad, es decir si se puede acceder desde una clase a la otra implicada en la relación, un cliente tiene que poder navegar (acceder a su cuenta), ¿y desde la cuenta acceder al cliente?. La navegabilidad se representa con la relación terminada en flecha, en caso de no indicar nada, es bidireccional

Un ejemplo de asociación:



Las relaciones navegables se expresan en código con atributos, en caso de ser como máximo uno con un atributo de tipo clase hacia la que se quiere navegar, en caso de muchos con una estructura de datos, en este caso solo se han visto vectores (en temas posteriores se trataran otros)



Pasar a código la asociación anterior.

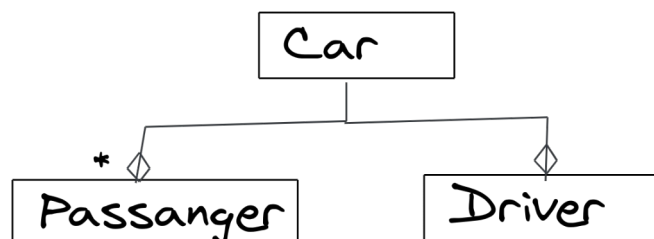
### 2.2.6.3 Agregación.

Un caso particular de la asociación es la agregación, se representa con un rombo sin color de fondo en una de las clases. Representa que una clase es parte de otra, de forma débil, la parte puede existir sin la principal y puede relacionarse con otras.



Por supuesto en la agregación se pueden definir navegabilidad y cardinalidad.

Aggregation



AlgoDaily

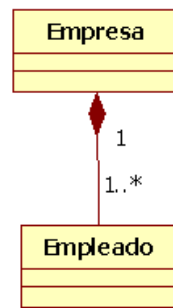
Se han de añadir elementos para la gestión de las relaciones, en la imagen anterior por

ejemplo sería necesario tener un método `anyadirPasajero(Pasajero p)` o `boolean quitarPasajero(String nombre)`

#### 2.2.6.4 Composición.

Otra especialización de la asociación, pero en este caso las partes no pueden existir sin el todo, por ejemplo si se tiene una silla formada por 4 patas, al eliminar la silla también se han de eliminar las patas, **por supuesto esto es un significado semántico y particular de cada aplicación.**

Se representa con un rombo con fondo negro, y se puede añadir navegabilidad y cardinalidad.



En la imagen anterior una empresa se compone de un conjunto de empleados, si se elimina la empresa del sistema también los empleados.

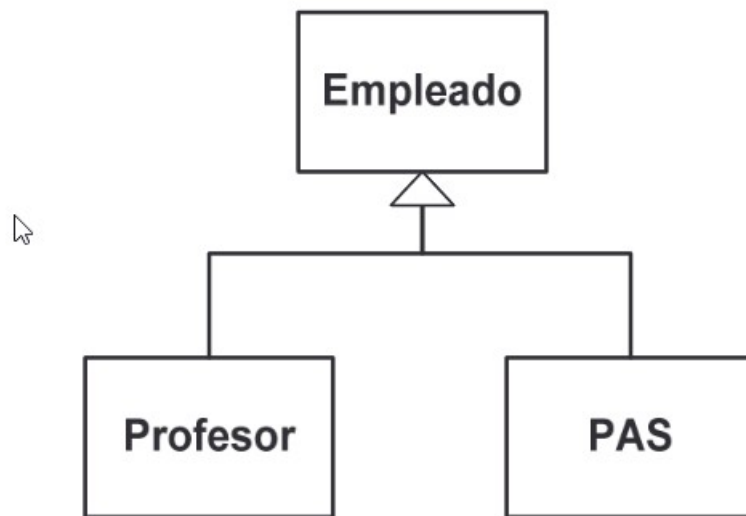


Implementar en código la composición de la imagen, ¿Qué diferencia existe en código con respecto a la agregación? ¿Y semánticamente?

#### 2.2.6.5 Generalización/especialización.

Existen otras relaciones como realización o dependencia que serán tratadas en otros módulos, la última relación de este tema es la generalización/especialización, se representa con una flecha en un extremo de la relación y se implementa con **herencia**.





Se utiliza cuando las clases tienen atributos similares o comportamiento parecido.

### 3. Actividades y ejercicios.

1. ¿Qué dos tipos de herencia existen?
2. De forma genérica, sin importar el lenguaje, ¿qué sucede cuando se crean objetos de clases que heredan de otras?.
3. ¿Qué significa que una clase es abstracta? ¿Y un método?
4. ¿Cómo se hereda en Java? ¿Qué sucede si se tiene un atributo private en la clase base? ¿Y en caso de un método que no define modificador de acceso?
5. ¿Que sucede si se llama a un método que no existe en la clase hija, pero si en la clase padre como privada?
6. ¿Y si tampoco se implementa en la clase padre sino en una clase de la que hereda el padre?
7. Al crear un objeto en Java que utiliza herencia,. ¿Se llama al constructor de la clase de la que hereda? ¿Si posee varios constructores a cuál se llama?
8. Se desea llamar a un constructor sobrecargado del padre en Java, ¿cómo hacerlo?
9. ¿Qué es una interfaz?
10. ¿Cuántas interfaces puede implementar una clase? ¿De cuantas clases puede heredar?
11. Indicar las modificaciones introducidas a partir de Java 8 en las interfaces.

12. Al implementar una clase una interfaz ¿qué ha de cumplir la clase que lo implementa?

13. ¿Qué es una clase abstracta? ¿Qué sentido tiene crear clases abstractas?

14. Indicar un ejemplo de clase abstracta que no sea el de los apuntes.

15. Si una clase posee un único método abstracto. ¿La clase se puede instanciar?

16. ¿Una variable de un tipo abstracto puede referenciar a un objeto que herede de la clase abstracta?

17. ¿De quién heredan todas las clases en Java?

18. ¿Para qué sirve el método clone? Por defecto ¿cómo actual?

19. ¿Qué es una copia superficial? ¿Y una profunda?

20. Explicar que es aliasing con algún ejemplo.

21. ¿Para qué sirve el método equals? ¿Internamente que otro método utiliza?

22. Definir la clase Persona con nombre, apellidos y fecha de nacimiento, reescribiendo los métodos clone con copia profunda y equals.

23. ¿Qué información aporta el método getClass?

24. ¿Qué es una variable polimórfica?

25. Sea una clase A que hereda de una clase B y ambas con el método prueba(), que imprime “soy A” y “soy B” por pantalla, si se tiene el siguiente código:

```
A objetoA= new A();  
B objeto B= objetoA;  
objetoA.prueba();  
objetoB.prueba();  
( (A) objetoB ).prueba();
```

Indicar la salida por pantalla.

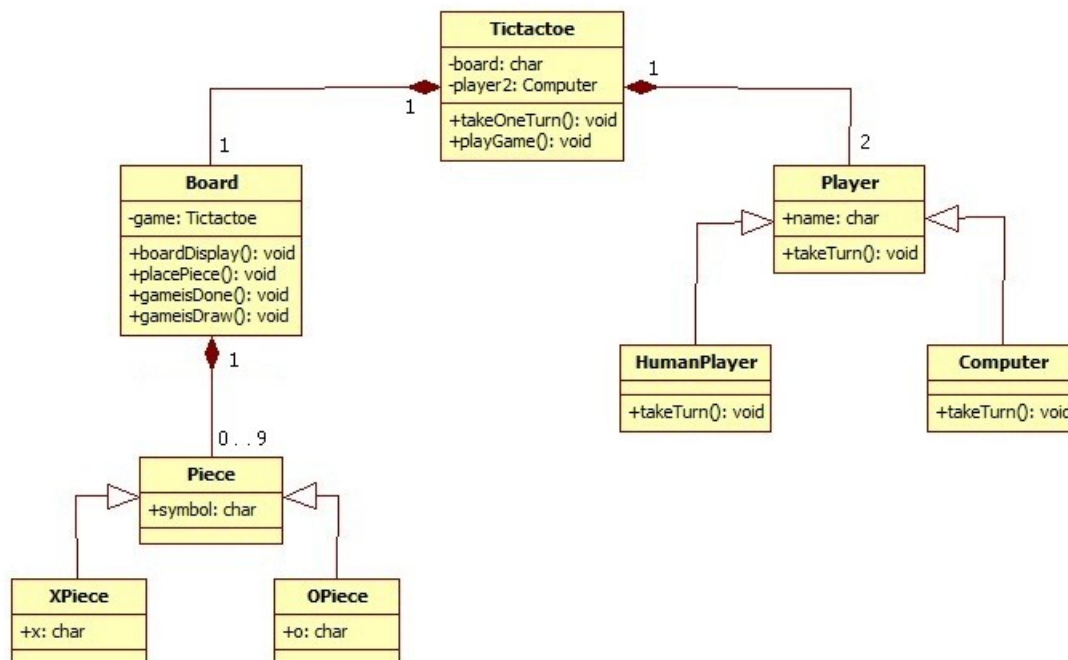
26. Sea una clase A que implementa metodoA() y que hereda de una clase B, si se tiene el siguiente código:

```
A objetoA= new A();  
B objeto B= objetoA;  
objetoA.metodoA();  
( (A) objetoB ).metodoA();  
objetoB.metodoA();
```

Indicar la salida por pantalla.

27. ¿Cómo se llama a un método de la clase padre?

28. ¿Que son las anotaciones?
29. Cuándo en un método se usa la anotación @override ¿Qué se está indicando?
30. Realizar un pequeño resumen del vídeo de OpenWebinar sobre el uso de anotaciones.
31. ¿Qué lenguajes utilizan herencia múltiple?
32. En caso de no indicarse navegabilidad en una asociación, ¿qué implica?
33. ¿Cómo se implementa una cardinalidad \* en una clase?
34. Diferencias entre agregación y composición?
35. Indicar las clases, las relaciones, sus tipos y cardinalidades del siguiente diagrama UML.



36. Escribir las clases de Player, HumanPlayer, Computer, Piece, Xpiece y Opiece, ¿alguna de ellas es abstracta?
37. ¿Cómo se implementaría la relación entre Board y Piece? ¿Qué estructura de datos usar para representar las piezas en el tablero?