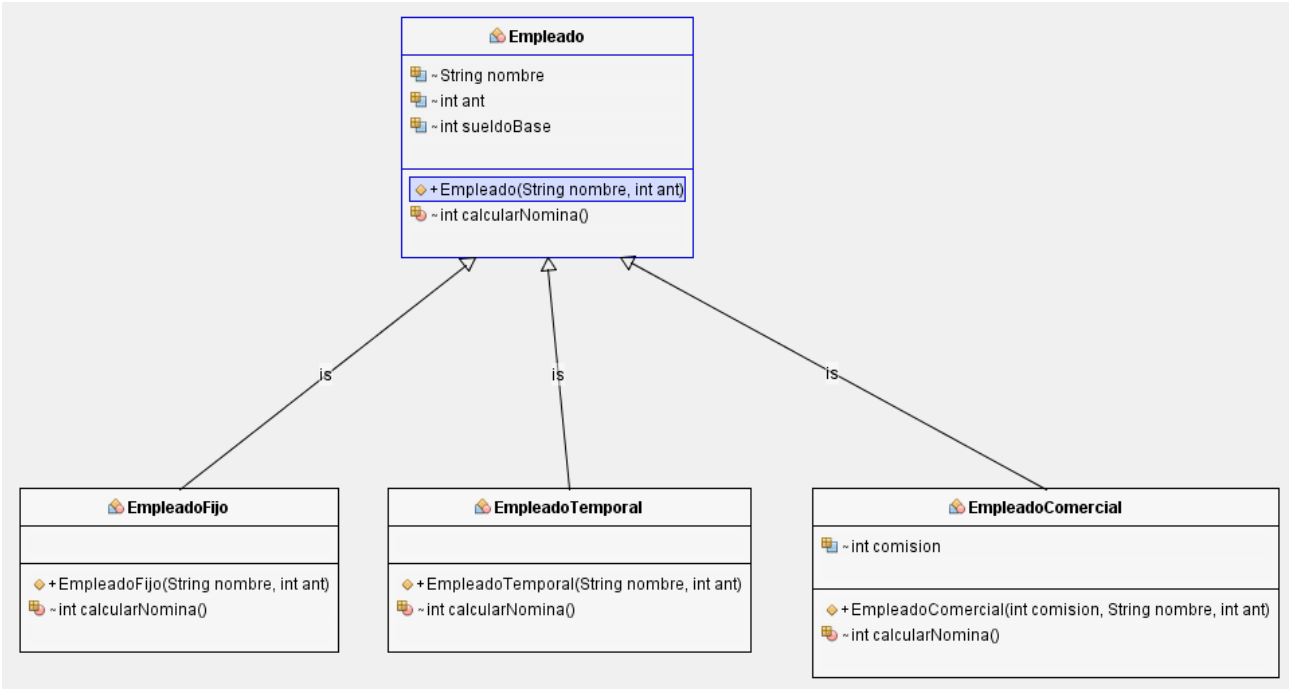
CLASES ABSTRACTAS

http://manuais.iessanclemente.net/index.php/Herdanza#Clases_e_m.C3.A9todos_abstractos

Ejemplo: Escribimos la siguiente jerarquía. De momento no hay nada abstracto.

Cada empleado tiene un nombre y una antigüedad(un entero que almacena los años de antigüedad). Un empleado comercial tiene además una comisión(un valor entero fijo, no hay que calcular nada). Además existe un método para calcular la nómina de cada empleado:

- Empleado fijo: sueldo base+100*ant
- Empleado temporal: sueldo base+50*ant
- Empleado comercial: sueldo base + comisión
- Empleado genérico: sueldo base +10*ant



tenemos que sobrescribir el método calcularNomina()

```
class Empleado{
    String nombre;
    int ant;
    int sueldoBase=1000;

    public Empleado(String nombre, int ant) {
        this.nombre = nombre;
        this.ant = ant;
    }
```

```
    int calcularNomina(){
        return sueldoBase + 10*ant;
    }
 }
class EmpleadoFijo extends Empleado{
    public EmpleadoFijo(String nombre, int ant) {
        super(nombre, ant);
    }

    @Override //esa anotación no es obligatoria pero tiene ventajas usarla, una es  la legibilidad del código
    int calcularNomina() {
        return sueldoBase+100*ant;
    }
}
class EmpleadoTemporal extends Empleado{
    public EmpleadoTemporal(String nombre, int ant) {
        super(nombre, ant);
    }
    int calcularNomina() {
        return sueldoBase+50*ant;
    }
}
class EmpleadoComercial extends Empleado{
    int comision;
    public EmpleadoComercial(int comision, String nombre, int ant) {
        super(nombre, ant);
        this.comision = comision;
    }
    int calcularNomina() {
        return sueldoBase+comision;
    }
}
public class App {
    public static void main(String[] args) {
        Empleado eGenerico=new Empleado("generico",2);
        System.out.println("Nomina de "+ eGenerico.nombre+": "+eGenerico.calcularNomina());
        EmpleadoFijo eFijo=new EmpleadoFijo("fijo",10);
        System.out.println("Nomina de "+ eFijo.nombre+": "+eFijo.calcularNomina());
        EmpleadoTemporal eTemporal=new EmpleadoTemporal("temporal",5);
        System.out.println("Nomina de "+ eTemporal.nombre+": "+eTemporal.calcularNomina());
        EmpleadoComercial eComercial=new EmpleadoComercial(30,"comercial",5);
        System.out.println("Nomina de "+ eComercial.nombre+": "+eComercial.calcularNomina());
    }
}
```

Sobre la necesidad de métodos y clases abstractas

Suponemos ahora, sobre la solución del ejercicio anterior que no existen los "empleados genéricos", solo pueden ser fijos, temporales y comerciales, por tanto, no tiene sentido calcular la nómina de un empleado genérico. Ante esto escribimos:

```
class Empleado{
    String nombre;
    int ant;
    int sueldoBase=1000;

    public Empleado(String nombre, int ant) {
        this.nombre = nombre;
        this.ant = ant;
    }
 }
class EmpleadoFijo extends Empleado{
    public EmpleadoFijo(String nombre, int ant) {
        super(nombre, ant);
    }
    int calcularNomina() {
        return sueldoBase+100*ant;
```

```
    }
}
class EmpleadoTemporal extends Empleado{
    public EmpleadoTemporal(String nombre, int ant) {
        super(nombre, ant);
    }
    int calcularNomina() {
        return sueldoBase+50*ant;
    }
}
class EmpleadoComercial extends Empleado{
    int comision;
    public EmpleadoComercial(int comision, String nombre, int ant) {
        super(nombre, ant);
        this.comision = comision;
    }
    int calcularNomina() {
        return sueldoBase+comision;
    }
}
public class App {
    public static void main(String[] args) {
        EmpleadoFijo eFijo=new EmpleadoFijo("fijo",10);
        System.out.println("Nomina de "+ eFijo.nombre+": "+eFijo.calcularNomina());
        EmpleadoTemporal eTemporal=new EmpleadoTemporal("temporal",5);
        System.out.println("Nomina de "+ eTemporal.nombre+": "+eTemporal.calcularNomina());
        EmpleadoComercial eComercial=new EmpleadoComercial(30,"comercial",5);
        System.out.println("Nomina de "+ eComercial.nombre+": "+eComercial.calcularNomina());
    }
}
```

Es decir, ahora cada subclase tiene un método calcularNomina() pero que no es sobrescriben a la versión de superclase.

NO GUSTA LA SOLUCIÓN ANTERIOR. ¿Y por qué queremos que los calcularNomina() de las subclases sean sobrescrituras del calcularNomina() del Padre y no métodos "sueltos" sin relación de sobrescritura?

- Para poder disfrutar del mecanismo de **polimorfismo** que veremos más adelante. Descubriremos que gracias al polimorfismo se evita duplicar código y se simplifica dicho código. Pero el funcionamiento del polimorfismo se basa en que en las subclases hay métodos que sobrescritos
- Y porqué queremos que el padre garantice que todos sus hijos se vean obligados a calcular una nómina, es decir, a que sobrescriban obligatoriamente calcularNomina() del padre.

Nos creemos las dos razones de arriba y buscamos la siguiente solución que consiste en crear en el padre un método con cualquier código, aunque no tenga sentido

```
int calcularNomina() {
        System.out.println("Esto nunca debe ejecutarse");
       return 1000;
   }
```

Esto es una chapuza pero consigue que los calcularNomina() de las subclases sean una "Redefinición o sobrescritura del método calcularNomina() del padre". Pero esto es una chapuza sin sentido y no obliga a las subclases a sobrescribir obligatoriamente el método calcularNomina()

Solución de la chapuza: ¿Cómo se hace en P.O.O. para tener en el padre un calcularNomina() que no haga nada pero que cree relación de sobrescritura con subclases y además obliga a las subclases a sobrescribir el método?

Declarando calcularNomina() como método abstracto. Lo que implica que la clase Empleado también sea abstracta .

```
abstract class Empleado{
    String nombre;
    int ant;
    int sueldoBase=1000;

    public Empleado(String nombre, int ant) {
        this.nombre = nombre;
        this.ant = ant;
    }
    abstract int calcularNomina();
 }
class EmpleadoFijo extends Empleado{
    public EmpleadoFijo(String nombre, int ant) {
        super(nombre, ant);
    }
    int calcularNomina() {
        return sueldoBase+100*ant;
    }
}
class EmpleadoTemporal extends Empleado{
    public EmpleadoTemporal(String nombre, int ant) {
        super(nombre, ant);
    }
    int calcularNomina() {
        return sueldoBase+50*ant;
    }
}
class EmpleadoComercial extends Empleado{
    int comision;
    public EmpleadoComercial(int comision, String nombre, int ant) {
        super(nombre, ant);
        this.comision = comision;
    }
    int calcularNomina() {
        return sueldoBase+comision;
    }
}
public class App {
    public static void main(String[] args) {
        EmpleadoFijo eFijo=new EmpleadoFijo("fijo",10);
        System.out.println("Nomina de "+ eFijo.nombre+": "+eFijo.calcularNomina());
        EmpleadoTemporal eTemporal=new EmpleadoTemporal("temporal",5);
        System.out.println("Nomina de "+ eTemporal.nombre+": "+eTemporal.calcularNomina());
        EmpleadoComercial eComercial=new EmpleadoComercial(30,"comercial",5);
        System.out.println("Nomina de "+ eComercial.nombre+": "+eComercial.calcularNomina());
    }
}
```

Prueba: añade un nuevo tipo de Empleado y observa cómo al ser el método abstracto estoy obligado a darle una implementación.

método abstracto: método que se declara con abstract y no tiene código en su cuerpo (no tiene implementación).

clase abstracta: una clase que se califica con abstract. Normalmente una clase abstract contiene al menos un método abstracto.

A continuación, a través de ejemplos veremos diversas cuestiones de funcionamiento de las clases abstractas.

Si una clase contiene un método abstracto, dicha clase debe declararse abstracta, en caso contrario, obtenemos error de compilación.

```
class A{
   abstract void saludo();
}
public class App {
   public static void main(String[] args) {

   }
}
```

Si declaramos la clase como abstracta desaparece el error.

```
abstract class A{
   abstract void saludo();
}
public class App {
   public static void main(String[] args) {

   }
}
```

Una clase abstracta no es instanciable.

```
abstract class A{
   abstract void saludo();
}
public class App {
   public static void main(String[] args) {
      A a = new A();
   }
}
```

una subclase de una clase abstracta, si no se declara también como abstracta, debe redefinir los métodos abstractos

```
abstract class A{
   abstract void saludo();
}
class B extends A{
        void imprimeUno(){
                System.out.println("1");
        }
}
public class App {
   public static void main(String[] args) {
```

```
    }
}
```

Si sobrescribo saludo(), no hay error

```
abstract class A{
    abstract void saludo();
}
class B extends A{
        void imprimeUno(){
                System.out.println("1");
        }
        void saludo(){
                System.out.println("hola");
        }
}
public class App {
    public static void main(String[] args) {

    }
}
```

una clase abstracta no tiene porqué contener un método abstracto. Es decir, no hay error de compilación al escribir una clase abstracta sin método abstracto. Es raro que exista una clase abstracta sin ningún método abstracto pero sintácticamente es correcto.

En el siguiente ejemplo observamos que normalmente que una clase abstracta no tenga ningún método abstracto es algo inútil. A es abstracta y por lo tanto como observamos en el main() no es instanciable. B la puede extender y como no tiene ningún método abstracto puede ser vacía, pero B no es abstracta y por lo tanto instanciable.

```
abstract class A{
    void saludo(){
        System.out.println("hola");
    }
}
class B extends A{

}
public class App {
    public static void main(String[] args) {
       // A a= new A(); error
        B b = new B();
        b.saludo();
    }
}
```

ESPECIFICADOR FINAL: Para evitar la sobrescritura de un método.

Puede resultar conveniente desear justamente la situación contraria a la sobrescritura, es decir, en una clase tenemos un método que no sólo no queremos obligar a que se sobrescribe si no que expresamente queremos PROHIBIR que se sobreescriba:

Ejercicio: Comprueba el error del siguiente código. A continuación suprime el especificador *final* y comprueba cómo desaparece el error.

```
class A{
    final void met(){ // con final no se puede sobrescribir
        System.out.println("Este es un método final");
```

```
    }
}
class B extends A{
    void met(){
        System.out.println("Estoy sobre escribiendo met() de A");
    }
}
class App{
    public static void main(String[] args) {
        B b = new B();
        b.met();
    }
}
```

Una clase abstracta puede contener métodos final.

Evidentemente los métodos abstractos tendrán que sobrescribirse y los métodos final no pueden sobrescribirse.

Ejemplo: Ejecuta y entiende el siguiente código. Observa como el objeto b utiliza un método *sobrescrito* de B y otro *final* de A

```
abstract class A{
    final void met1(){
        System.out.println("Este es un método final de A");
    }
    abstract void met2();
}
class B extends A{
    void met2(){
        System.out.println("Estoy sobre sobrescribiendo met2() de A");
    }
}
class App{
    public static void main(String[] args) {
        B b = new B();
        b.met2();
        b.met1();
    }
}
```

ESPECIFICADOR FINAL: Para prohibir la herencia de clases.

Todavía más prohibitivo que no permitir sobrescribir un método de una clase, es directamente no dejar extender una clase con lo cual ya ni siquiera el método llega a heredarse

Ejemplo: Evitar que haya clases que se puedan "extender" de A. Observa el error que produce el siguiente código y a continuación retira el especificador final para ver como desaparece.

```
final class A{
    final void met1(){
        System.out.println("Este es met1() de A");
    }
}
class B extends A{
    void met2(){
        System.out.println("Esto es met2() de B");
    }
}
class App{
    public static void main(String[] args) {
        B b = new B();
        b.met2();
        b.met1();
    }
```

```
}
```

Y para acabar con final, aunque no tiene que ver con clases y herencia repasamos otro uso de final ya visto

Final para indicar que queremos que un atributo funcione como una constante con nombre.

Una constante es un valor que no varía en toda la ejecución del programa. Podemos hacer que el valor de un atributo sea constante utilizando el especificador *final*. El valor constante será aquel que se indica en la inicialización de atributo.

Ejemplo: Descomenta la asignación a la constante y observa el error.

```
class A{
   final int VAR_X=4;
}
class App{
   public static void main(String[] args) {
      A a = new A();
      System.out.println(a.VAR_X);
      //a.VAR_X=9;
   }
}
```

Vimos también que podíamos trabajar con " variables de clase" utilizando el especificador static. Si combinamos final + static podremos trabajar con "variables constantes de clase".

Ejemplo:

```
class A{
   static final int VAR_X=4;
}
class App{
   public static void main(String[] args) {
       System.out.println(A.VAR_X);
      //A.VAR_X=78;
   }
}
```

Entiende porque los atributos final siempre suelen ser además static

Los atributos final normalmente los calificaremos también como static. Es lógico, ya que sin static todos los objetos de esa clase almacenarán ese mismo valor de sólo lectura. Se gasta menos memoria si almacenamos ese valor de sólo lectura en la zona static una única vez

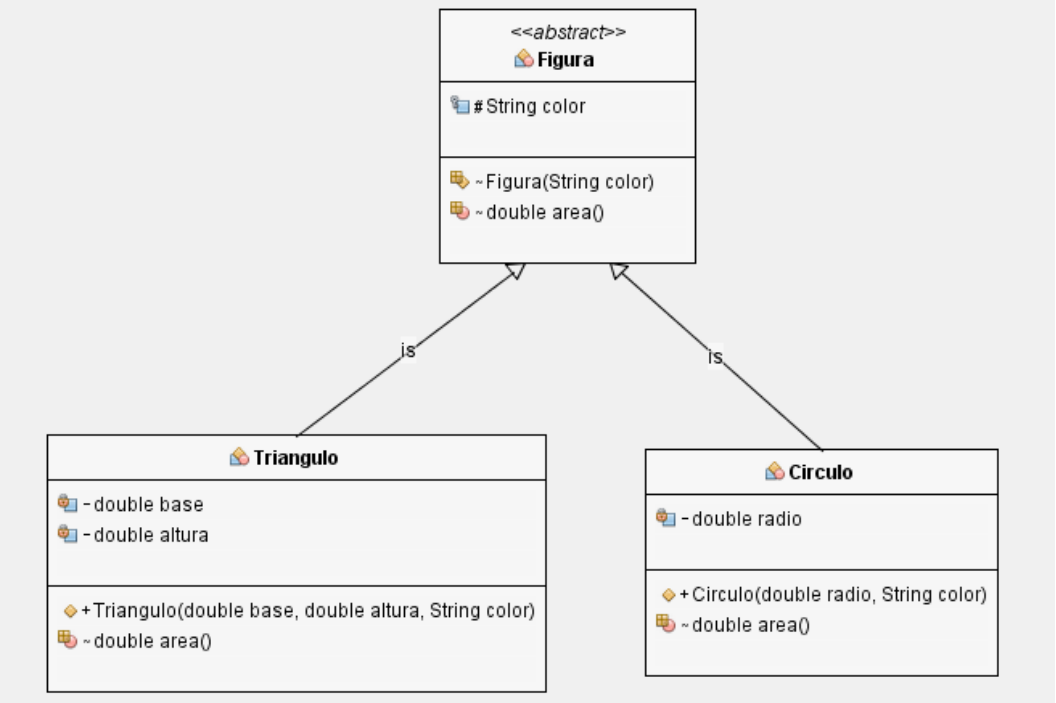
Ejercicio U5_B4_E1: ¿Qué falta en el siguiente código?.

```
abstract class Articulo{
   protected String titulo;
   protected float precio = 5.0f;
   public abstract boolean esAlquilable();
   public float getPrecio() {
      return precio;
   }
}

class Pelicula extends Articulo{
   public boolean esAlquilable() {
      return true;
   }
}

class Libro extends Articulo{
   public float getPrecio() {
      return 0.0f;
```

```
    }
}

class App{
    public static void main (String[] args) {
        Pelicula pelicula = new Pelicula();
        Libro libro = new Libro();
        System.out.println(pelicula.esAlquilable());
        System.out.println(libro.getPrecio());
    }
}
```

Ejercicio U5_B4_E2: Escribe código para implantar la siguiente estructura. Luego en main de App crea un triángulo y un círculo y calcula su área.

El diagrama está generado con el plugin para netbeans "easyUML". El método area() es abstracto en Figura, aunque en el diagrama no se aprecie debes de tenerlo en cuenta para hacer el ejercicio.



Las flechas indican una relación de herencia "un triángulo es una Figura" y un "Círculo es una figura".