**Herencia**

Hereda de un padre a un hijo, metodos y propiedades.

Usa extends cuando creamos la clase.

**abstractas**

Las clases abstractas no se instancian, son solo para heredar sus metodos y propiedades. Un metodo abstracto obliga a heredar y sobreescribir a las clases hijas, por que estan vacios.  
Aunque no se instancien, los objetos pueden ser tratados como una clase Abstracta, (polimorfismo).

Las clases abastractas empiezan como **“publica abstract class nombre{ }”** y los metodos abstractos declaran igualmente abstract antes del tipo de dato que devuelven, ejemplo:   
**“public abstract string ladrar()”;** sin especificar nada mas por que recordemos es abstracto y viene vacio, obligando a sobreescribir a los hijos.

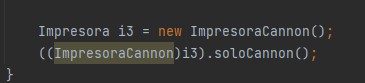
**Polimorfismo**

Impresora i1 = new ImpresoraCannon()

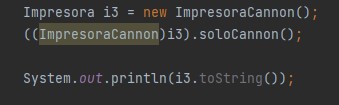
Instancio una nueva ImpresoraCannon pero la trato como si fuera una Impresora.

No necesito casteo si estoy tratando de invocar un metodo que era abstracto de impresora por que impresoracannon lo tiene seguro.

Podemos castear ImpresoraCannon como vemos a continuacion, para tratarla como otro tipo de clase, tiene que ser un objeto de la clase, tiene que haber una relacion del tipo herencia, en este caso ImpresoraCannon hereda de Impresora, asi que existe esa relacion. Tambien podriamos tratarla como un Object, puesto que toda clase hereda de Object, algo util para hacer comparaciones, etc.

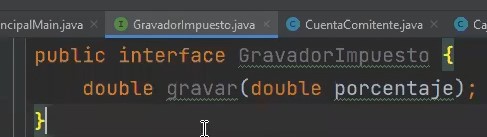


En el siguiente caso, podemos usar el toString por que es un metodo de Object, padre de todas las clases.

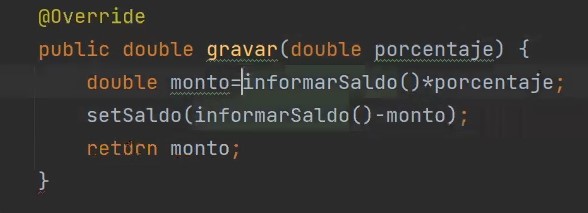


**Interface**

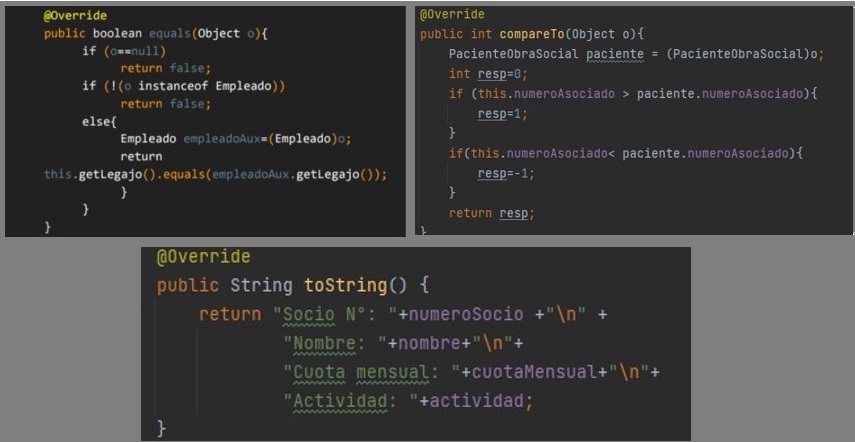
Se inicia con “public interface nombre{}” //Todos los metodos de una interface son abstractos y publicos.



Es un contrato como en las clases abstractas, pero mas fuerte, por que todos los metodos de una interface son abstractos. De esta forma obligamos a sobreescribir metodos que tienen que ser cumplidos. Vamos a sobreescribir el metodo.



Otra ventaja es que no podemos tener una herencia multiple, es decir que una clase no puede tener varios padres, con interface podemos heredar de diferentes clases, sin que sean padre-hijo, es una especie de herencia multiple.**Ejemplo de Override de equals, toString y compareTo QUE SON DE LA CLASE OBJECT, ES DECIR TODAS LAS CLASES LA TIENEN.**

****

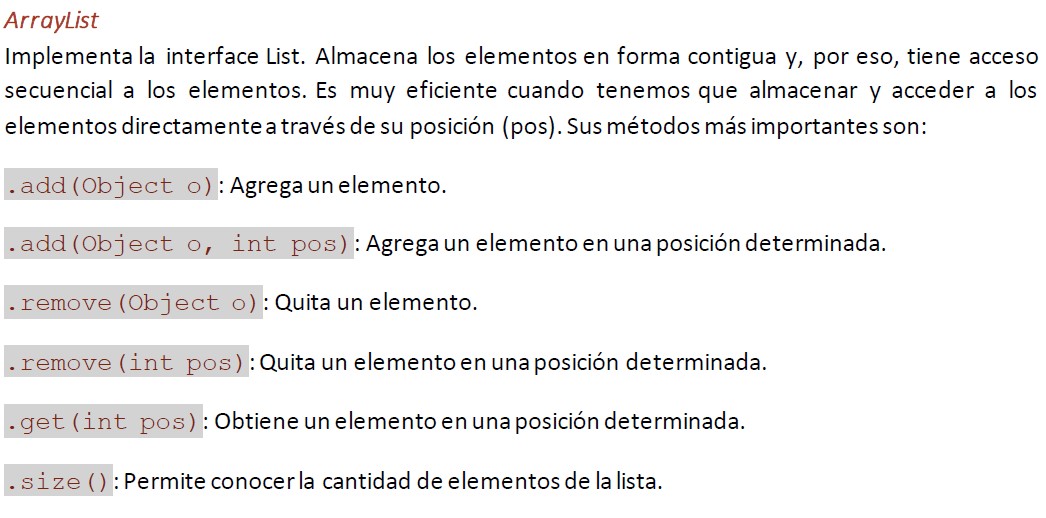
**Colecciones**

Agregar : **Add(objeto)**

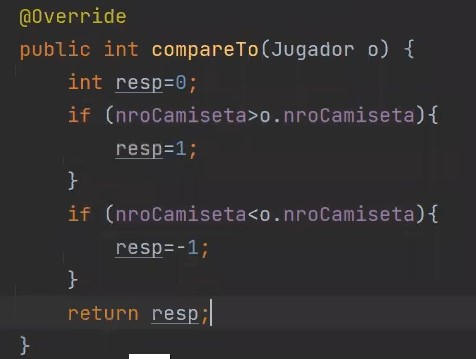
Eliminar : **remove()**

Obtener el tamaño: **size()**

Recorrer objetos almacenado : **iterator()**

****

Podemos declarar que el tipo de un compare sera una clase distinta a Object cuando hacemos un Override. En este ejemplo es para tener un control mas estricto, solo podremos comparar un Jugador.



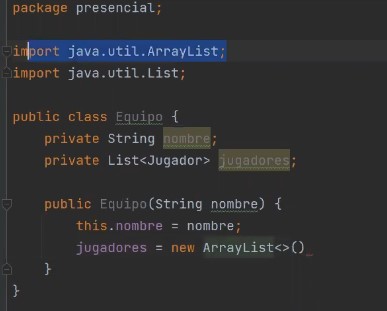
Necesitamos aclarar en la implementacion que sera tratada de otra forma.



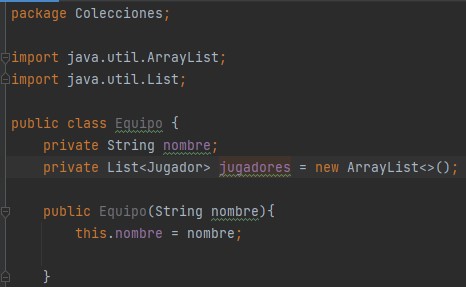
A la hora de crear la lista, la declaramos de la siguiente forma:



Si hacemos un constructor, nos pedira que pasemos un listado, pero hay veces que no tenemos un listado, quiza queremos construir un equipo y despues agregar jugadores a la lista. Asu qye lo inicializamos en el constructor estando vacio.

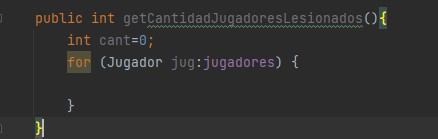


Tambien podriamos instanciar en donde esta el atributo en vez del constructor, funciona de la misma manera.



**forEach**

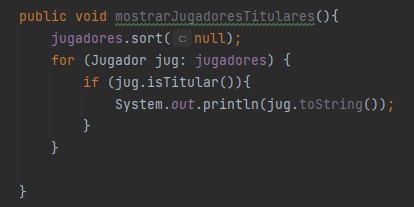
Para recorrer una lista, podemos usar un forEach, tiene la siguiente estructura.



Como vemos, empezamos con un for, seguido por unos parentesis que contienen primero el tipo de objeto que tendra el array, el parametro que se usara para tratar a cada uno de los objetos en cada iteracion y por ultimo despues de los dos puntos, el array que vamos a iterar.

**Sort**

Para ordenear una lista, necesitamos usar un sort, la forma de usarlo es llamar la array y usar el metodo .sort() introduciendo por parametros la forma de ordenamiento, si solo ingresamos la palabra reservada **null**, usar el compareTo de del tipo de Clase que hayamos definido en la respectiva clase. En el siguiente ejemplo ordenamos los jugadores de la lista “jugadores” antes de mostrarlos.



**Ejemplo de un for buscando agregar elementos a un array, desde otro array, contando/iniciando desde atrás.**

public ArrayList<Jugador> obtenerReservas() {  
 ArrayList<Jugador> reservas = new ArrayList<>();  
 for (int i = jugadores.size() - 1; i >= jugadores.size() - 5; i--){  
 reservas.add(jugadores.get(i));  
 }  
 return reservas;  
}

**Excepciones**

Para hacer Excepciones personalizadas, debemos crear una nueva clase y heredar de Exception. A esta clase le creamos un constructor vacio por si es necesario, y un constructor con un mensaje. Y hacemos un Override del toString para mostrar el cartel de la excepcion con el mensaje que uqerramos, tambien usamos getMessage() que es un metodo que heredamos de Exception.

public class ClienteException extends Exception{  
 public ClienteException() {  
 super();  
 }  
 public ClienteException(String message) {  
 super(message);  
 }  
 @Override  
 public String toString(){  
 return "Se produjo la siguiente excepcion: "+getMessage();  
 }  
}

Otro ejemplo de una excepcion personalizada:

public void comprar(double importe) throws ClienteException {  
 if (importe + deuda > limite) {  
 //error  
 throw new ClienteException("Error. Saldo insuficiente, no puede pasar su limite.");  
 } else {  
 //endeudarse  
 deuda += importe;  
 }  
}

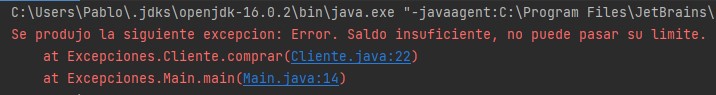
Ahora un ejemplo de una clase Cliente con 2 posibles errores que arrojan excepciones que hemos personalizado:

public class Cliente {  
 private String nombre;  
 private String apellido;  
 private String DNI;  
 //saldoEnCuenta es lo que DEBO  
 private double deuda;  
 private double limite;  
  
 public Cliente(String nombre, String apellido, String DNI, double limite) {  
 this.nombre = nombre;  
 this.apellido = apellido;  
 this.DNI = DNI;  
 this.limite = limite;  
 deuda = 0.0;  
 }  
  
 public void comprar(double importe) throws ClienteException {  
 if (importe + deuda > limite) {  
 //error  
 throw new ClienteException("Error. Saldo insuficiente, no puede pasar su limite.");  
 } else {  
 //endeudarse  
 deuda += importe;  
 }  
 }  
  
 public void saldarDeuda(double importe) throws ClienteException{  
 if(importe <= deuda){  
 //se acepta  
 deuda-=importe;  
 }else{  
 //error  
 throw new ClienteException("Error. El importe es mayor a la deuda.");  
 }  
 }  
}

Al parecer el throw new blablabla, setea el Message, por que cuando llamamos a getMessage, nos devuelve ese error que tiramos.

Una herramienta util es el metodo printStackTrace(), que nos da una trazabilidad de donde ocurre el error. Esto se escribe en el catch.

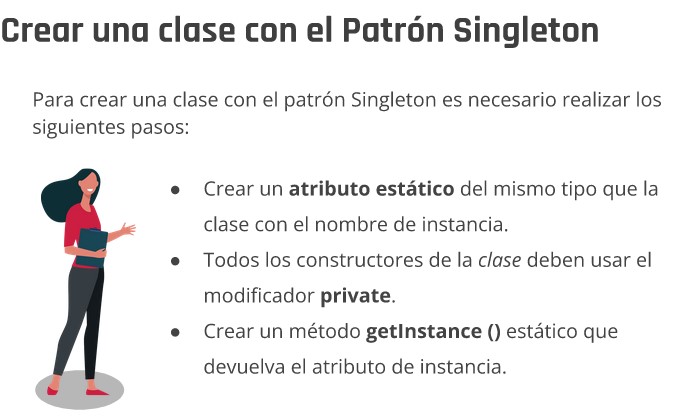
catch(ClienteException e){  
   
 e.printStackTrace();  
}



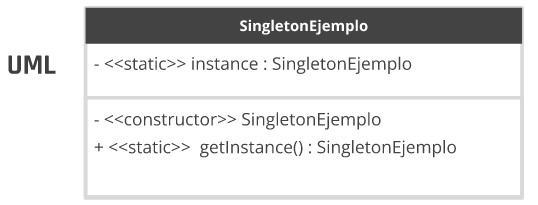
**Patrones**

**Singleton**

Singleton es un patron de diseño de creacion que garantiza que una clase tenga una sola instancia y define un punto de acceso global para ella, ningun otra clase podra instanciarla. La clase siempre ofrecera su propia instancia y si aun no tiene una, crea y devuelve esta instancia recien creada.

****

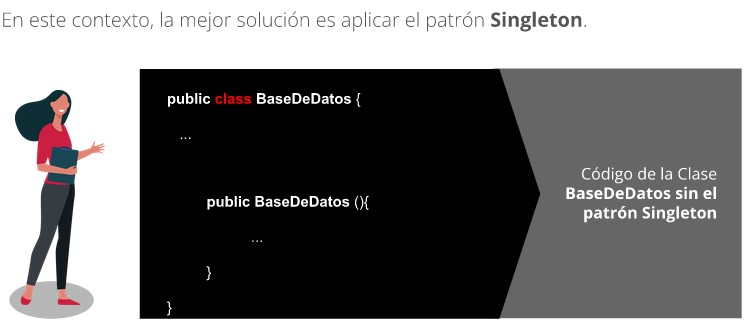
El atributo estatico lo utilizamos para que sea un atributo propio de la clase. Es decir que pueda acceder al mismo sin que tenga que instanciar al objeto, no pertenece a la instancia, pertenece a la clase.

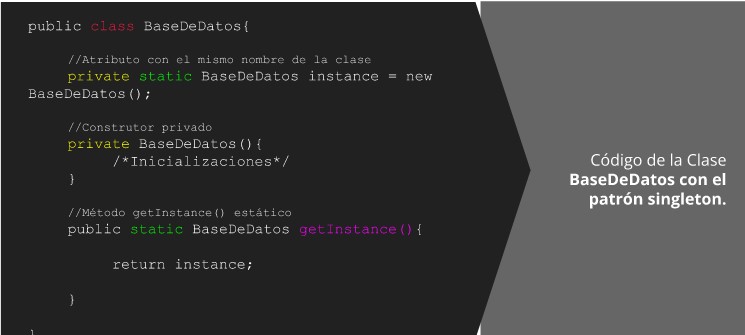
****

El patron singleton se usa cuando necesitas un solo punto para crear instancia de clase y cuando solo necesita una instancia de una clase.

Un ejemplo de singleton es una clas que se necesita para la conexión a la base de datos, mas de una conexión a una base de datos en la misma ejecucion de codigo al mismo tiempo puede generar problemas.

Para evitarlo, bloqueamos la posibilidad de instanciar esta clase fuera de si mismo ( ni siquiera las subclases pueden) y creamos una forma de permitir que exista una sola instancia de ella.





**El siguiente ejemplo es de una inicializacion tardia de un singleton.**

public class EmpleadoFactory {  
  
 private static EmpleadoFactory *instance*;  
  
 //Creamos un constructor en privado para que nadie pueda usarlo.  
 private EmpleadoFactory(){  
  
 }  
  
 //METODO CONSTRUCTOR DE SINGLETON (TARDIO)  
 public static EmpleadoFactory getInstance(){  
 if(*instance*==null){  
 *instance*= new EmpleadoFactory();  
 }  
 return *instance*;  
 }

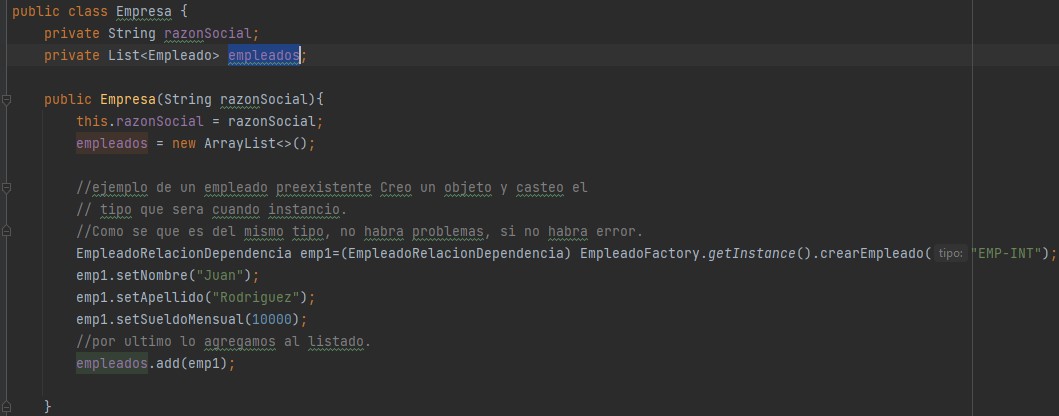
**Factory**

Factory es un metodo de creacion, en el cual establecemos una fabrica (factory) en la cual tendremos el/los metodos para crear otros objetos, es un medio con el cual podemos controlar que solo se instancien objetos en esta fabrica.

Ejemplo de una factory usando a la vez singleton.

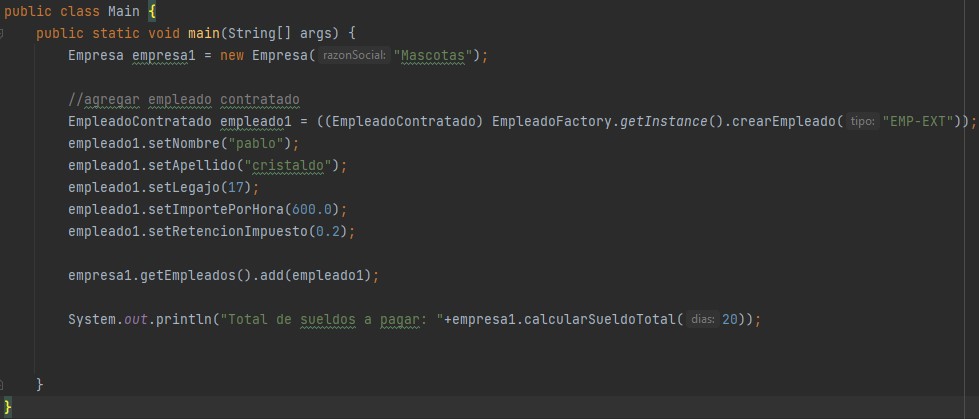
package PatronesSingletonYFactory;  
  
public class EmpleadoFactory {  
  
 private static EmpleadoFactory *instance*;  
   
 //Creamos un constructor en privado para que nadie pueda usarlo.  
 private EmpleadoFactory(){  
 }  
   
 //METODO CONSTRUCTOR DE SINGLETON (TARDIO)  
 public static EmpleadoFactory getInstance(){  
 if(*instance*==null){  
 *instance*= new EmpleadoFactory();  
 }  
 return *instance*;  
 }  
  
  
 //METODO DE CONSTRUCCION FACTORY  
 //Dependiendo del tipo que nos pasen por parametros  
 //sera la clase del objeto que crearemos.  
 public Empleado crearEmpleado (String tipo){  
 Empleado emp=null;  
 switch(tipo){  
 case "EMP-INT":  
 emp= new EmpleadoRelacionDependencia();  
 break;  
 case "EMP-EXT":  
 emp= new EmpleadoContratado();  
 break;  
 }  
 return emp;  
 }  
}

Ejemplo de como podemos crear un empleado preexistente desde la clase empresa, por medio de la factory mientras tambien tiene singleton. Estableciendo sus valores por medio de los setters, muy necesarios en el metodo factory.





**Ejemplo de creacion de empleados desde el main en el patron factory.**

****

**Patron Composite**

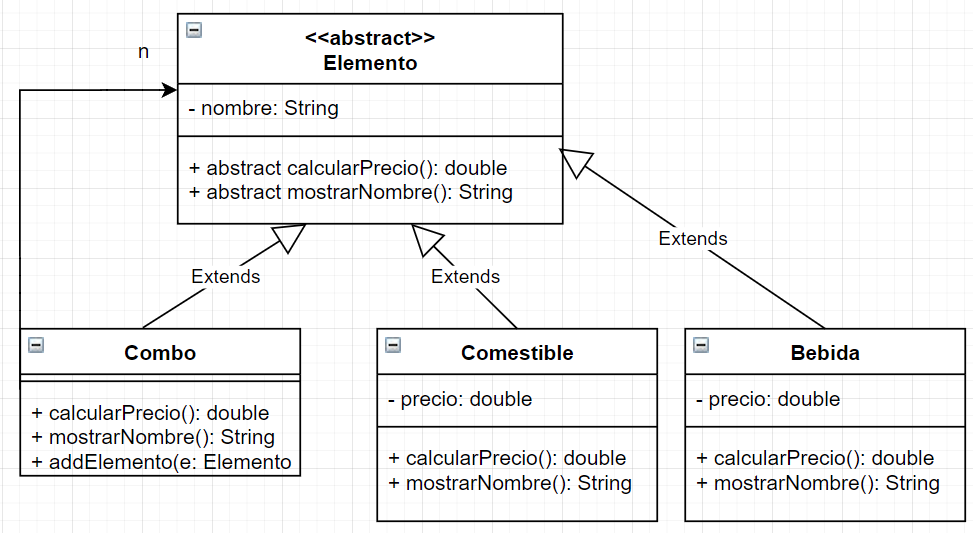
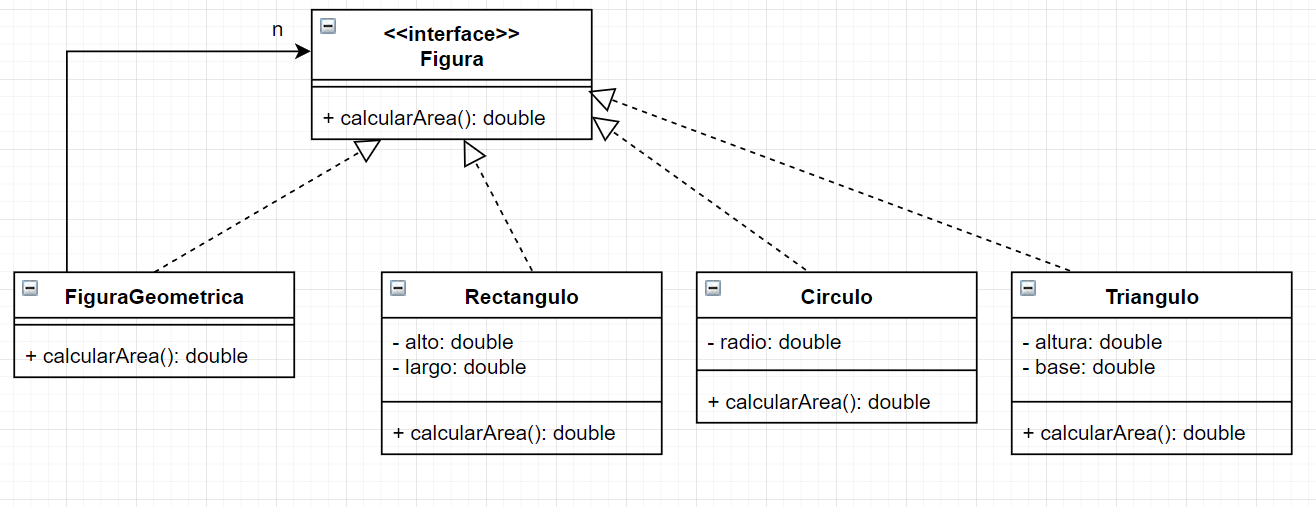
Es un patron categoria estructura, se enfoca en la manera que los objetos esta compuestos para formar estructuras mas complejos.

Su objetivo es componer objetos en estructuras de arbol para representar jerarquias. Permite tratar de manera uniforme objetos indivudales y composiciones de objetos.

Se usa cuando se puede representar jerarquicamente la estructura de objetos o cuando queremos que el codigo cliente trate los objetjos compuestos complejos y simples de la misma manera.



**Ejemplos de UML**



El ejemplo del codigo esta en la carpeta PatronComposite.

En resumen el “combo” tiene una lista de elementos, en el cual puede tener cualquiera de los elementos.