Diseño de Aplicaciones con Java

Contenido

HERENCIA

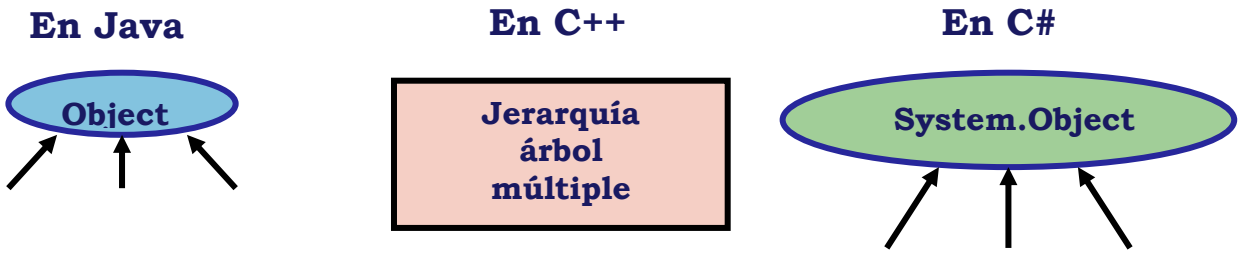
1.- Introducción. Conceptos básicos.

Las clases proporcionan una buena forma de descomposición modular.
Buscamos mecanismos que permitan reutilizar y extender código.
Antes de P.O.O. la reutilización consistía en el uso de bibliotecas de funciones.
En P.O.O. encontramos mecanismos que capturan semejanzas entre grupos de estructuras similares, teniendo en cuanta las diferencias de los casos individuales.

Ejemplo : Modelado de la clase Casa

Generalmente existe una clase que incluye las características comunes a toda clase del sistema (jerarquía de árbol único).
Ej: en Java la clase Object facilita ,entre otros, clone( ) y toString( ).

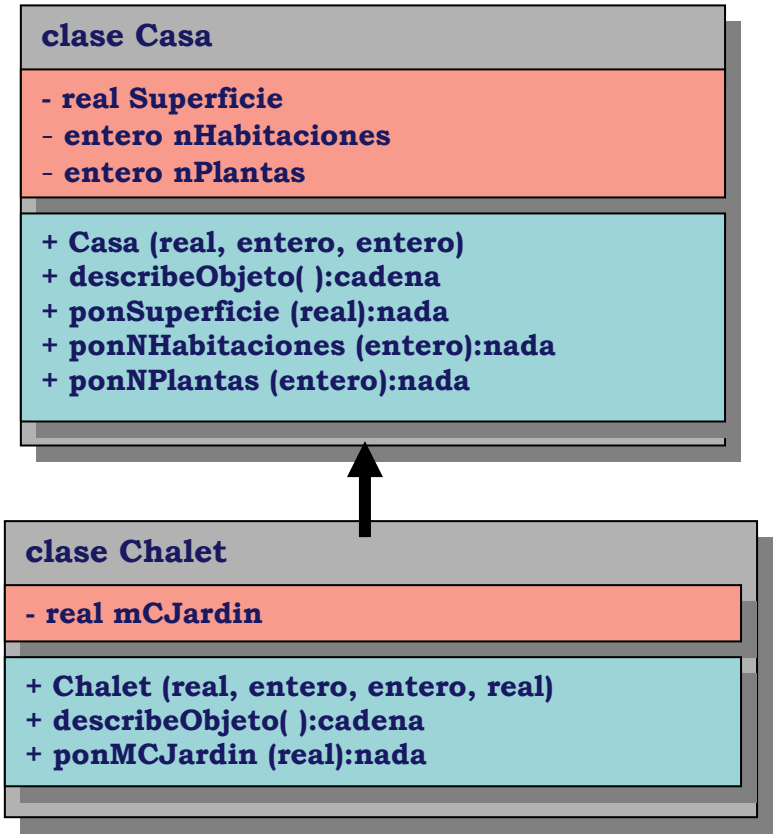
Todos los objetos heredan implícitamente de esa clase (**ANY** en Eiffel, **Object** en Smalltalk)



Por herencia nos referimos a la propiedad de que los ejemplares de una clase hija pueda tener acceso a los datos y los comportamientos asociados a una clase paterna [Budd 95].

Herencia: proceso mediante el cual un objeto adopta las propiedades de otro.
La herencia proporciona todas las herramientas necesarias para crear clases a partir de otras por extensión, especialización o combinación.
Es multigeneracional, transitiva y organiza las clases en una estructura jerárquica.

Ahora modelamos la clase Chalet aprovechando la especificación de Casa



No es únicamente un mecanismo de compartición de código, es una ampliación del sistema de tipos. Aunque parezca una paradoja, el tipo de la clase base es mayor que el de la clase derivada.

Si Chalet hereda de Casa, incorpora:
- estructura (atributos) de Casa.
- comportamiento (métodos) de Casa.

Además Chalet puede adaptar :
- Añadiendo nuevos atributos/métodos
- Redefiniendo métodos heredados
  (refinamiento)
  Ej : clone( )
- Renombrando atributos/métodos
  (sobreescritura /anulación)
  Ej: toString( )
- Implementando métodos diferidos
  (métodos abstractos)

Cuando se envía un mensaje a un objeto, la búsqueda del método correspondiente comienza con los asociados al objeto. Si no se encuentra, se examinan los métodos asociados a la superclase inmediata. Así sucesivamente.
Una **referencia de la clase base** puede conectarse con un **objeto de la clase derivada**, pero sólo podrá acceder a los elementos heredados por la clase derivada.

Para que exista Herencia debe cumplirse el axioma *ObjetoSubClase* ***es un*** *ObjetoSuperclase*

Justificación de la herencia del ejemplo :

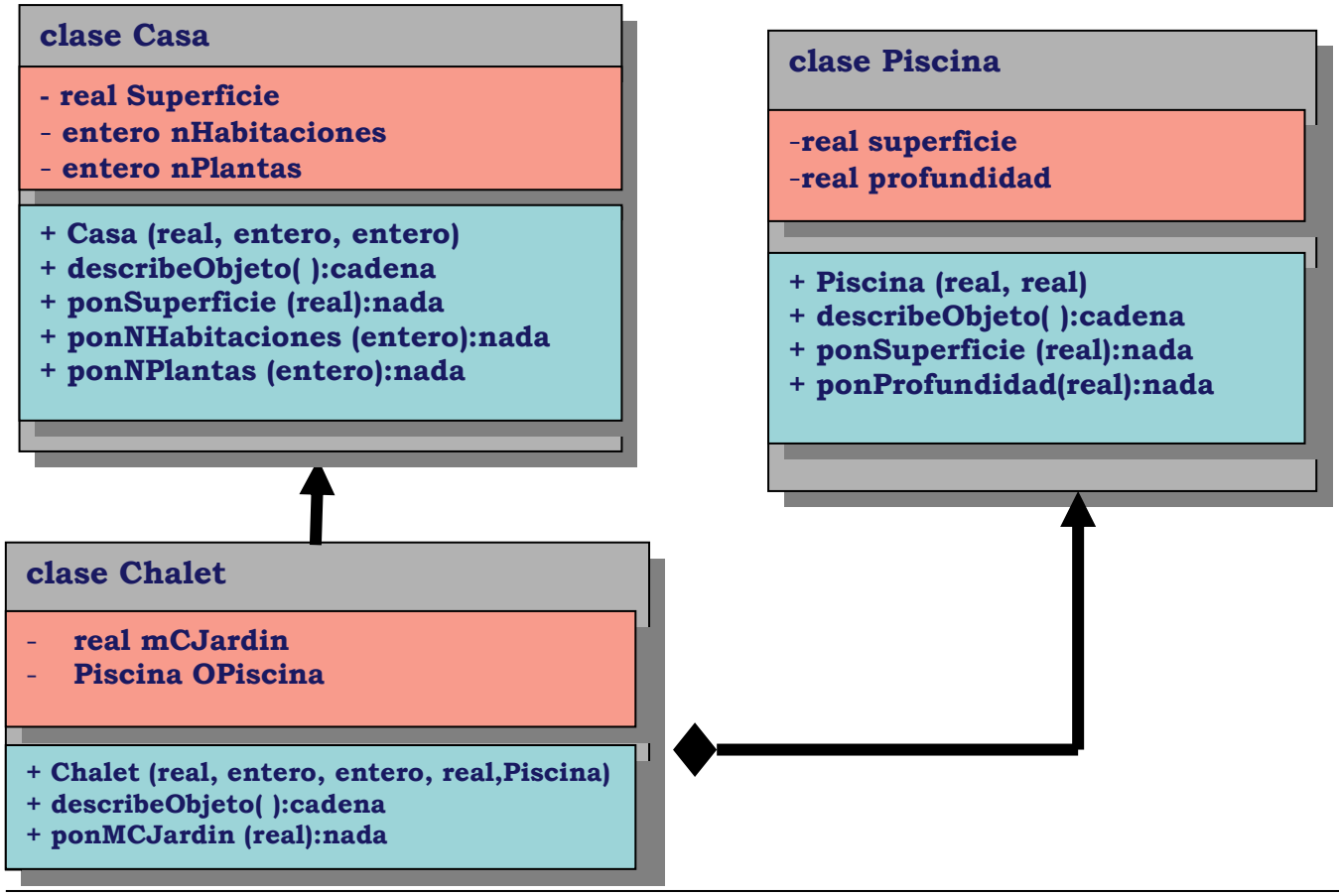
- **Chalet comparte (reutiliza) código de Casa**
- **Amplía el sistema de tipos: Chalet es un subtipo de Casa**
  refCasa=Ochalet
  ~~**refChalet=OCasa**~~
- **"Un Chalet *es una* Casa"**

Clientela

Existe otra forma de crear clases a partir de otras: haciendo que contengan objetos/referencias de otras clases **(clientela).**
Una clase es cliente de otra cuando tiene un objeto de la primera como atributo o utiliza como parámetro u objeto local en un método un objeto de la primera.
Podemos distinguir:
Composición. Dos objetos están estrictamente limitados por una relación complementaria. Uno no se entiende sin el otro, esto es, cada uno por separado no tiene sentido.
Agregación. Dos objetos se consideran usualmente como independientes y aun así están ligados.

Responde al axioma ***tiene un***

Chalet ***es una*** Casa ***tiene una*** Piscina
Chalet es un objeto *compuesto* de otro objeto (Piscina)



Herencia o Clientela

Cuando se puede aplicar herencia, se puede aplicar clientela.

Criterios para elegir una u otra opción:

-Regla del cambio: Si se percibe el axioma "es un", pero los componentes del objeto pueden cambiar en tiempo de ejecución e interesa recibir ese cambio, aplicar clientela.

-Regla del polimorfismo: Si se percibe el axioma "es un", pero entidades de tipo más general necesitan ser conectados a objetos de un tipo más especializado, utilizar herencia.

-Cuando la clase más específica tiene la misma funcionalidad que su hipotética antecesora, la herencia sería lo más recomendable.

Si utilizamos clientela en vez de herencia no creamos nuevos tipos ni una jerarquía de clases. Podría ser un grave problema para implementar polimorfismo.

Cuando utilizamos herencia se instancia automáticamente el objeto de la clase base, aunque no utilicemos sus características.

La clientela permite el ahorro de recursos cuando no se piensa utilizar la instancia de la clase base, ya que sólo nos obliga a la declaración de una referencia, permitiendo crear el objeto cuando se precise.

En herencia el modo de reutilización de código lo hace el compilador, y en clientela lo hace el programador.

2.- Herencia en Java

Para heredar en Java utilizamos la sentencia ***extends*** .

Forma general :

class <nombreSubclase> ***extends*** *<nombreSuperclase> {*
<cuerpo de la clase>
}

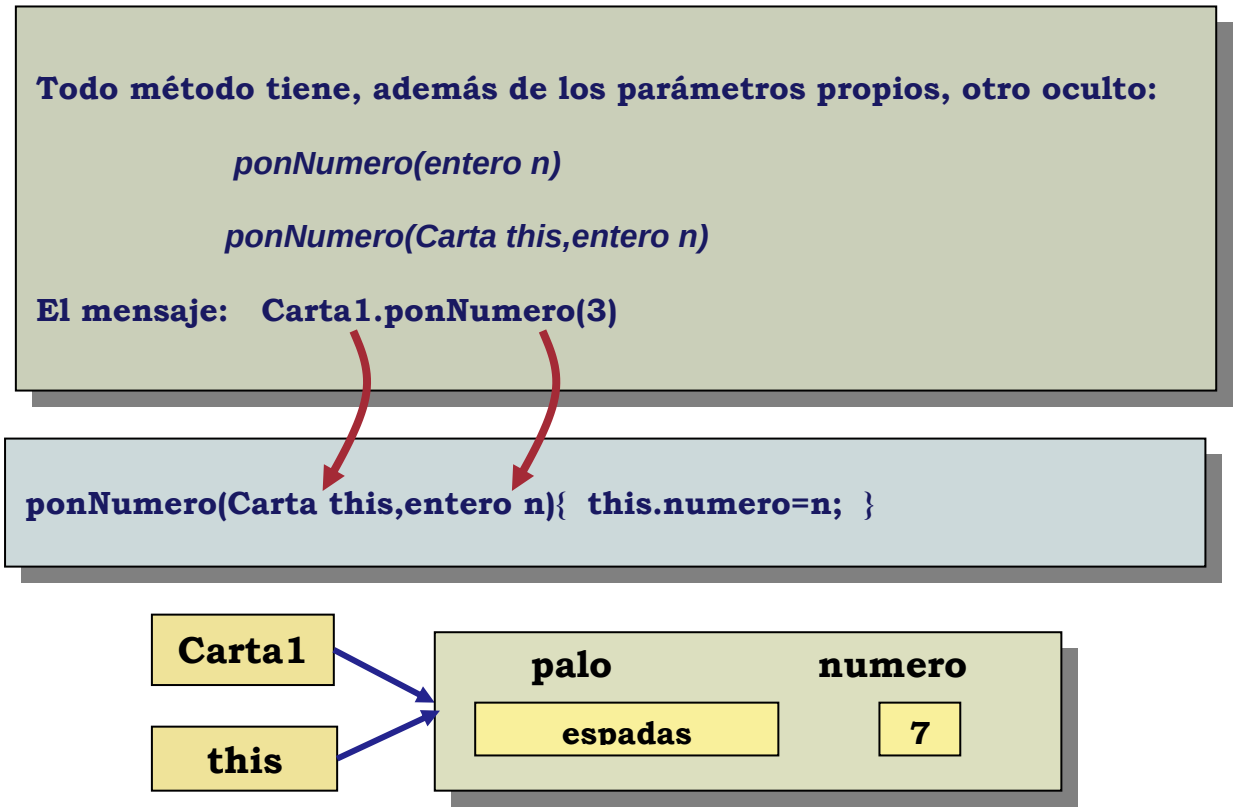
La subclase hereda todos los métodos y los atributos de la clase superior, pero no tiene acceso directo a los elementos private, pero sí a los públicos, protected y friendly.
Podemos acceder a los miembros autorizados de la superclase como si estuvieran definidos en la subclase.

```
/* programa ejemplo de herencia de clases*//*prog_java_16*/
class Caja{
        private double alto,ancho,fondo;
        public double volumen(){
                return alto*ancho*fondo;
        }
        Caja(double al,double an,double fon){     //Si el constructor fuese 'private' no se podría acceder
                alto=al; ancho=an; fondo=fon;
        }
}/* fin clase Caja*/
class CajaConPeso extends Caja{
        private double peso;
        CajaConPeso(double al,double an,double fon,double pe){
                super(al,an,fon);
                peso=pe;
        }
        public double devPeso(){
                return peso;
        }
}/* fin clase CajaConPeso */
public class UsaCaja{
        public static void main(String argv[]){
                Caja C1=new Caja(15,10,32);
                System.out.println("Volumen C1 : "+C1.volumen());
                CajaConPeso Cp1=new CajaConPeso(10,20,15,200);
                //objeto Cp1 que usa un métodos de la superclase//
                System.out.println("Volumen Cp1 : "+Cp1.volumen());
                System.out.println("Peso Cp1 : "+Cp1.devPeso());
                  //intento de C1 de usar métodos de la subclase//
        //System.out.println("Peso C1 : "+C1.devPeso()); //No variable peso defined in class Caja.
                }
}/*fin clase UsaCaja */
/* Salida :
Volumen C1 : 4800.0
Volumen Cp1 : 3000.0
Peso Cp1 : 200.0
*/
```

Una variable de la superclase puede referenciar a un objeto de cualquier subclase suya. Pero sólo puede acceder a los atributos y métodos de la clase a la que pertenece. Se le llama ***upcasting*.**
El tipo de la variable es el que determina a qué elementos podemos acceder. En el programa anterior, C1 no puede acceder al atributo peso, puesto que está definido en la subclase CajaConPeso.

3.- El parámetro this.

En programación procedimental, las llamadas a los procedimientos llevan como argumento la estructura que se debe tratar, en P.O.O. no es así.
Cuando enviamos un mensaje a un objeto, obviamos el argumento de la estructura afectada.
Como los métodos son comunes a todos los objetos y pueden existir varios objetos en memoria, ¿a qué objeto revierte dicho método la acción a realizar?, evidentemente al que le mandamos el mensaje, pero ¿cómo? → parámetro **this**



En Java:
La referencia a un atributo cualquiera dentro de la misma clase se hace con **this.atributo**, aunque en este caso es opcional hacerlo así; podemos suprimir el this. Sólo tiene sentido si en una herencia al haber dos atributos con el mismo nombre queremos recalcar que se trata de el de la propia clase, no el heredado (se verá posteriormente).
Para referirnos a un constructor de una clase desde otro de la misma clase, lo hacemos con la referencia **this([parámetro])**, que buscará el constructor que corresponde con el formato del parámetro pasado.
De igual manera, para referirnos a un método desde otro dentro de la misma clase, se empleará **this.<NombreMétodo([parámetro])>**.

Ejemplo:

```
/* Uso de this en una clase */
class Caja{
        private double alto,ancho,fondo;
        public void setAlto(int _a){this.alto=_a;}
        public double getAlto(){return alto;}
        public void setAncho(int _a){this.ancho=_a;}
        public double getAncho(){return ancho;}
        public void setFondo(int _f){this.fondo=_f;}
        public double getFondo(){return fondo;}
        public double volumen(){
                return alto*ancho*fondo;
        }
        public void dimension(double al, double an, double fon){
                alto=al;
                ancho=an;
                fondo=fon;
                this.volumen(); //además de establecer la dimensión, la devolvemos
        }
}
```

4.- Utilización de super y this en herencia.

Para referirnos desde una subclase a una superclase inmediata utilizamos **super**.
Se utiliza para dos casos:

- Para acceder al constructor de la superclase.
- Para acceder a un miembro de la superclase que ha sido ocultado (sobrescrito) por un miembro de la subclase. Este aspecto se verá posteriormente.

"**super** es a la clase superior lo que **this** a la propia clase"
Si usamos super en un constructor debe ser la primera instrucción de éste.
Cuando tenemos un atributo en la subclase con el mismo nombre que otro de la superclase, el de la superclase quedaría oculto.

Cuando los atributos son privados no puedo acceder a ellos directamente con 'super.'
Conclusión: En herencia, los miembros 'private' no son accesibles, los demás sí lo son.
Como se apuntó anteriormente la referencia a un atributo cualquiera dentro de la misma clase se hace con **this.atributo**, aunque en este caso es opcional hacerlo así; podemos suprimir el this. Sólo tiene sentido si en una herencia al haber dos atributos con el mismo nombre queremos recalcar que se trata de el de la propia clase, no el heredado. Utilizaremos **super.elemento** para referirnos a elementos de la superclase.

```
/* ejemplo de ocultación de atributos *//*prog_java_17*/
class SpClase{
        int i,j;
}
class SubClase extends SpClase{
        int i;    //Cualquier identificador que se llamara 'i' ocultaría al heredado, aunque no fuese de tipo int
        SubClase(int a, int b){
                super.i=a;// referencia al 'i' en la superclase
                i=b;
                j=b;
        }
        void imprimir(){
                System.out.println("super.i :"+super.i);
                System.out.println("i:  "+i+"  j:    "+j);
        }
}/*fin clase SpClase*/
class UsaClases{
        public static void main(String argv[]){
                SubClase ob1=new SubClase(5,10);
                System.out.println("ob1.i : "+ob1.i);
                System.out.println("ob1.j : "+ob1.j);
        //      System.out.println("ob1.super.i : "+ob1.super.i);
                ob1.imprimir();
        }
}
```

No es accesible. super no se puede utilizar en clases que no hayan heredado. Sólo vale para referirse a una superclase.

```
/* Salida :
ob1.i : 10
ob1.j : 10
super.i :5
i:  10  j:    10
*/
```

Nota : Si no se declara otra 'i' en SubClase, la sentencia super.i=a; también sería correcta, puesto que 'i' en Spclase es público en el paquete. En este caso toda la salida tendría el valor 10.

5.- Jerarquía de clases.

Se pueden construir jerarquías con tantos niveles de herencia como se desee.
Cada subclase hereda todas las características de las subclases superiores.

En una jerarquía de clases, al crear un objeto se ejecutan todos los constructores desde la superclase original hasta la subclase de la que hemos creado el objeto.
Por ejemplo, para crear un objeto Chalet, primero se ejecutaría el constructor de Casa( ) y después el propio del Chalet( )

```
/* ejemplo de orden de ejecución con constructores en una jerarquía de clases*/
/*prog_java_18*/

class A{
        A(){
                System.out.println("constructor de A");
        }
}
class B extends A{
        B(){
                //implícitamente hace super( );
                System.out.println("constructor de B");
        }
}
class C extends B{
        C(int i){
                //implícitamente hace super( );
                System.out.println("constructor de C, valor de i :"+i);
        }
}
class UsaConstructores{
        public static void main(String argv[]){
                C C1=new C(5);
        }
}
```

```
/* Salida :
constructor de A
constructor de B
constructor de C, valor de i :5
*/
```

6.- Sobreescritura de métodos.

Cuando un método de una subclase tiene el mismo nombre y parámetros (es idéntico) que uno de la superclase, el método de la subclase ***sobrescribe*** al de la superclase.
También puede darse el caso de ***reutilización*** del método heredado.
Si no tiene los mismos parámetros se trata de sobrecarga, no de sobreescritura.

```
/* ejemplo de sobreescritura de métodos*/
class Cuadro{
        private int alto,ancho;
        public Cuadro(int al, int an){
                alto=al;
                ancho=an;
        }
        public void muestraCuadro(){
                System.out.println("Alto : "+alto+"    Ancho :  "+ancho);
        }
}/* fin clase Cuadro*/
class CuadroColor extends Cuadro{
        private String color;
        public CuadroColor(int al, int an, String co){
                super(al,an);
                color=co;
        }
        public void muestraCuadro(){
                super.muestraCuadro();
                System.out.println("Color :  "+color);
        }
}/* fin clase CuadroColor   */
public class Ejemplo{
        public static void main(String argv []){
       Cuadro C=new Cuadro(4,7);
       CuadroColor CC=new CuadroColor(2,8,"rojo");
       C.muestraCuadro();
       CC.muestraCuadro();
        }
}/* fin clase Ejemplo */
/*Salida
Alto : 4    Ancho :  7
Alto : 2    Ancho :  8
Color :  rojo
*/
```

Como se observa, existe un método en CuadroColor que es idéntico que el de la clase heredada Cuadro : *public void muestraCuadro(),* al existir una referencia *super.muestraCuadro()* reutilizamos el método de la superclase, de no existir esta instrucción habría una sobrescritura (anulación) del método heredado.

7.- Utilización de final y protected con herencia.

Utilizando final con un atributo indica que no se puede modificar (es constante).
Existen otros usos de final:
Si queremos que un método no pueda ser sobrescrito, pondremos **final** al principio de su declaración.
Si no queremos que una clase sea heredada, se pone **final** en su declaración. Esto no implica que no pueda importarse y utilizar sus características.

final public class ClaseNoHeredable{
.....
}

Un miembro **protected** podrá ser utilizado por subclases en toda la jerarquía, pero siempre que esté dentro de un método. Para un objeto de esa subclase sería inaccesible.
Recuerdo que un miembro **private** no puede ser utilizado ni siquiera por métodos de la subclase.

Ejemplo :

```
/**ejemplo de protected */
public class Super {
  private int i;
  protected int j;
  public void f1(int a){i=a;}
  public void f2(int b){j=b;}
  public String toString(){
    String s="  "+i+" "+j+" ";
    return s;
    }
}
public class Sub extends Super{
  private int  k;
  public void f3(int c){k=c;}
  public void f4(int d){j=d;}
  public String toString(){
    String s=super.toString()+" "+k;
    return s;
    }
}
public class Sub2 extends Sub{
  private int l;
  public void f5(int e){
    j=e;
    }
}
```

```
public class UsaProtected {
  public static void main(String[] args) {
    Sub OSub=new Sub();
    OSub.f3(6);
    OSub.f4(2);
    System.out.println("Salida de Sub : "+OSub);
    Sub2 OSub2=new Sub2();
    OSub2.f5(8678);
    System.out.println("Salida de Sub2 : "+OSub2);
  }
}
```

Ejemplo de uso de 'protected' en distintos paquetes :

```
/* ejemplo uso de protected en distintos paquetes *//*ejemplo_protected*/
/*paquete uno*/
package uno;
public class Uno{
        protected int i;
        void metodoUno(int x){
                i=x;
        }
}

/*paquete de la clase Dos */
package dos;
import uno.Uno;
public class Dos extends Uno{
        int j;
        public void metodoDos(int y){
                j=y;
                i=j*2; // como i es protected es accesible //
        }
}
```

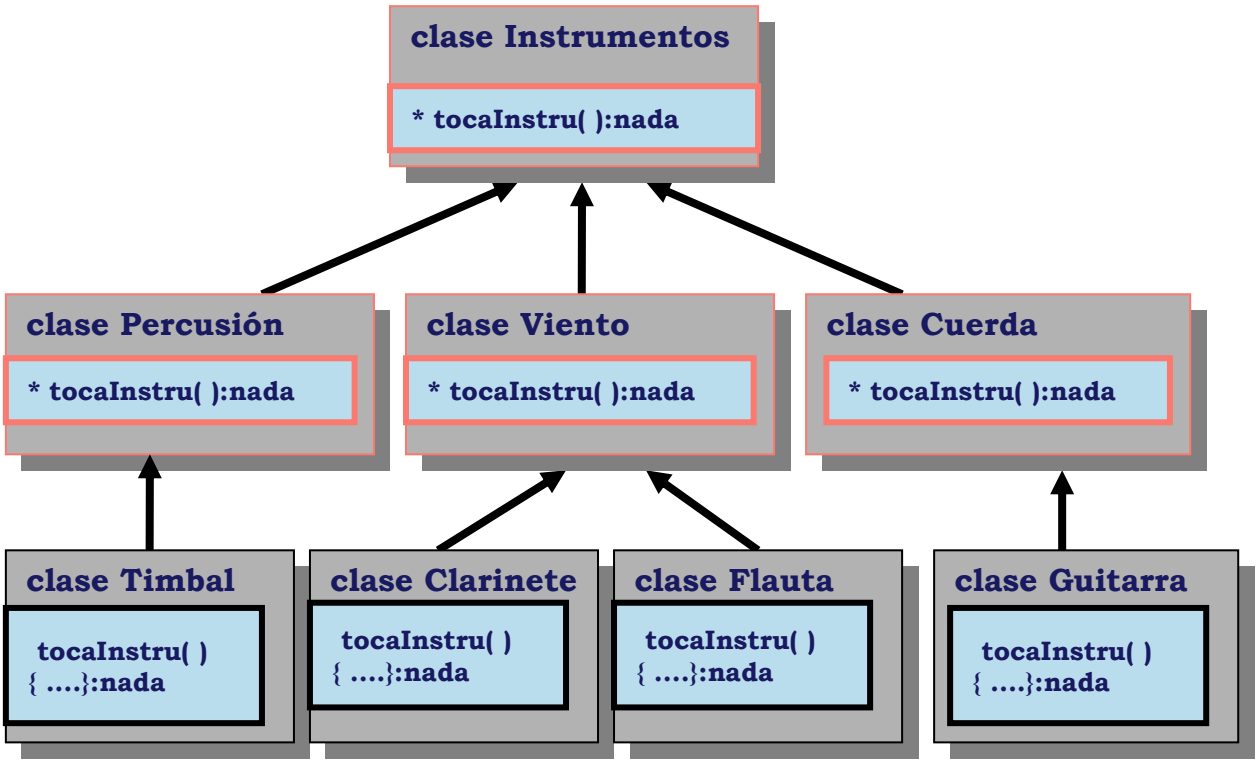
```
/* uso de protected en distintos paquetes */
import dos.Dos;
class Programa{
        public static void main(string argv[]){
                Dos ob=new Dos();
                ob.metodoDos(8);
        //      System.out.println("ob.i:"+ob.i);//Error. 'i' no es accesible por las clases
                                                    que heredan de donde se encuentra 'Uno'
                                                    Si 'i' fuese public, entonces sería correcto. //
                //ob.metodoUno(6);
        }
}
```

Las clases anteriores deben ser públicas. Si el método de la superclase no tiene acceso, al heredar en otro paquete dicho método no es utilizable.
Si el acceso del método es protected, entonces sí se puede usar.

8.- Clases abstractas.

Un aspecto particular de la herencia es la creación de jerarquías de clases abstractas/virtuales puras.
Un método abstracto no se define, solo se declara.
Las clases que tienen un método abstracto son, a su vez, abstractas, pero pueden estar parcialmente implementadas por otros métodos.
Toda clase que herede de una abstracta, debe implementar los métodos declarados como abstractos o ser abstracta.
Sirven para definir métodos que serán comunes a toda la jerarquía en nombre, parámetros, tipo devuelto, etc.
Dictan comportamientos dentro de una jerarquía. Cualquier clase que pertenezca implementará su método particular.
No se pueden crear objetos de clases abstractas, sólo referencias.
Los constructores no pueden ser abstractos.
Los métodos estáticos no pueden ser abstractos.

Estos conceptos también son fundamentales para entender el polimorfismo.



Cualquier clase que quiera pertenecer a esta jerarquía deberá implementar el método tocaInstru{} , con esto conseguimos crear un comportamiento general dentro de este grupo de clases.

En Java para construir una clase abstracta utilizaremos el modificador ***abstract***.

abstract class NombreClase{
....
}

Ejemplo con clases de figuras de dos dimensiones :

```
/* ejemplo de clases abstractas*/
abstract class Figuras{
        double dim1,dim2;
        Figuras(double d1, double d2){
                dim1=d1;
                dim2=d2;
        }
        abstract void area();
}/* fin clase Figuras */
class Rectangulo extends Figuras{
        Rectangulo(double al, double an){
                super(al,an);
        }
        void area() {
                System.out.println("Area del rectángulo : "+dim1*dim2);
        }
}/*fin clase Rectangulo */
class Triangulo extends Figuras{
        Triangulo(double a, double b){
                super(a,b);
        }
        void area() {
                System.out.println("Area del Triángulo : "+(dim1*dim2)/2);
        }
}/*fin clase Triangulo */
class EjemploAreas{
        public static void main(String argv[]){
                Figuras f;      //es una variable, no un objeto
                Rectangulo r=new Rectangulo(10,10);
                Triangulo t=new Triangulo(5,5);
                r.area();
                t.area();
        }
}/* fin clase EjemploAreas */
```

```
/* Salida :
Area del rectángulo : 100.0
Area del Triángulo : 12.5
*/
```

9.- Interfaces.

Una interface es como una clase abstracta pura, es decir una clase en la que sólo se pueden definir cabeceras de métodos sin implementar.
Se utilizan para forzar comportamientos a las clases que las implementan (heredan), ya que Java pierde cierta funcionalidad al renunciar a la herencia múltiple de clases.

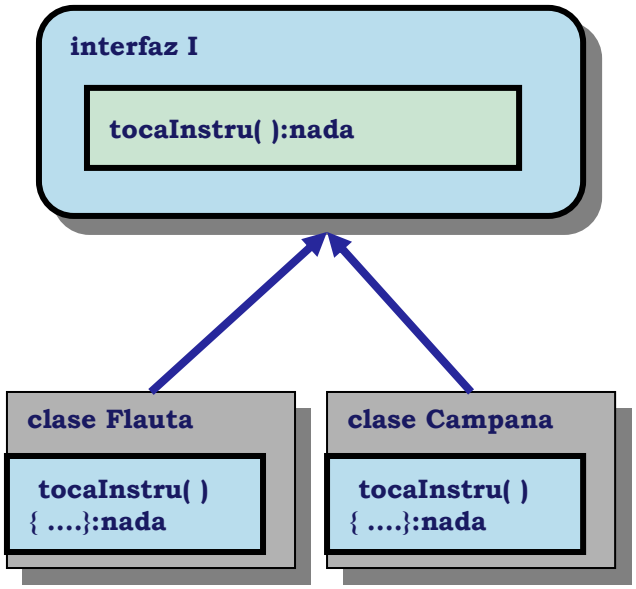
9.1.-Creación de una interfaz:

Forma general :

```
<acceso> interface <Nombre>{
        <tipo> <método1>(<parámetros>);
        <tipo> <método2>(<parámetros>);
        .....
        <tipo> <nombreVariable><Valor>;
}
```

9.2.-Propiedades de interfaz

El acceso puede ser público o por defecto (igual que las clases).
Los métodos de la interface se consideran públicos, no se le indica acceso y no tienen cuerpo. Pero cuando una clase implementa una interface se ha de especificar cada método con acceso público.
Una interface no tiene constructores.
No se pueden instanciar.
Podemos definir atributos, pero estos son implícitamente *public static final* (constantes globales).
Las interfaces ofrecen más operatividad que las clases abstractas, ya que permiten dictar comportamientos a ciertas clases sin necesidad de pertenecer a una determinada jerarquía.
Si por ejemplo queremos usar un método en muchos objetos y no queremos que las clases que los crean hereden de esa, hacemos una interface con ese método y todos los objetos pueden usarla sin haber heredado.



En el ejemplo, la clase Flauta y Campana no tienen por qué pertenecer a la misma jerarquía de clases pero sí necesitar implementar el método tocaInstru( ), en tal caso implementarían la interfaz I.

9.3.-Implementación de una interface

Cualquier clase puede implementar una interface o varias, separadas por comas.
Clases distintas pueden implementar la misma interface.
Si una clase implementa una interface, debe definir todos los métodos o ser abstracta.

```
/* ejemplo de interface *//*prog_java_21*/
public interface Intf{
        void metodo(int p);
}
class ClaseA implements Intf{
        public void metodo(int p){
                System.out.println("p : "+p);
        }
        public void otroMetodo(){
                System.out.println("otroMetodo");
        }
}
class UsaInterface{
        public static void main(String argv[]){
                ClaseA ob=new ClaseA();
                ob.metodo(7);
                ob.otroMetodo();
        }
}
```

Las interfaces pueden ser utilizadas para compartir constantes en múltiples clases:

```
/* ejemplo interface para compartir constantes en múltiples clases*//*prog_java_22*/
interface Constantes{
        double pi=3.1415926;
        int luz=300000;
}
class UsaConstantes implements Constantes{
        public double calculaLongitud(double radio){
                return 2*pi*radio;
        }
}
class ProgConstantes{
        public static void main(String argv[]){
                UsaConstantes ob=new UsaConstantes();
                System.out.println("Longitud circunferencia : "+ob.calculaLongitud(2.5));
        }
}
/* Salida :  Longitud circunferencia :  15.707963 */
```

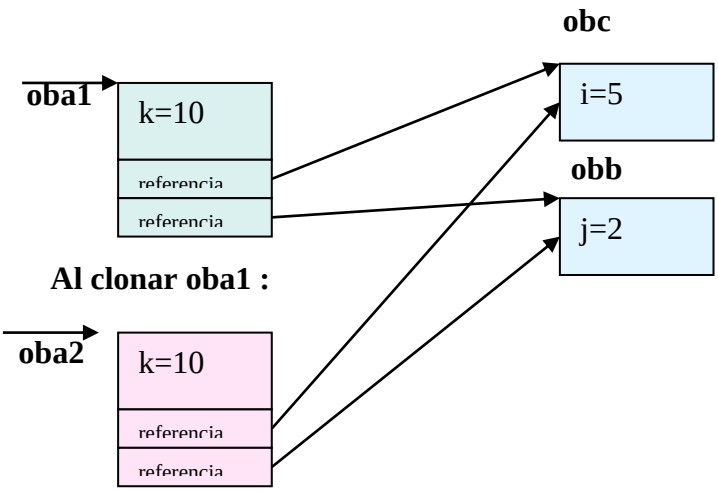
9.4.- La interfaz Cloneable.

Cuando necesitemos tener una copia de un objeto, podemos duplicar éste con el método clone().
Al pasar un objeto a un método, pasa por referencia siempre puesto que es una dirección. Si queremos pasar dicho objeto por valor para no correr el riesgo de modificarlo, debemos hacer una copia del objeto.
Clonar objetos puede tener efectos laterales, sobre todo cuando trabajamos con objetos compuestos. Esto ocurre cuando algún atributo de una clase es una referencia a otro objeto existente.

```
/* ejemplo objetos compuestos*/
class C{
        int i=5;
}
class B{
        int j=2;
}
class A{
        int k=10;
        C obc;          //objeto como atributo de una clase
        B obb;          //idem.
        A(){
                obc=new C();
                obb=new B();
        }
}

class ObjetoCompuesto{
        public static void main(String argv[]){
                A oba1=new A();                         // oba1 es un objeto compuesto//
                System.out.println("oba1.obc.i: "+oba1.obc.i);
                System.out.println("oba1.obb.j: "+oba1.obb.j);
                System.out.println("oba1.k: "+oba1.k);
        }
}
```

```
/* Salida :
oba1.obc.i: 5
oba1.obb.j: 2
oba1.k: 10
*/
```

Con referencia al programa anterior, si consideramos:
.....
A oba1=new A();
A oba2;
oba2=oba1.clone(); *// si se pudiera hacer. Java no lo permite //*
.....
El esquema sería :



En estas circunstancias hacer:
oba2.obc.i=9; // modificamos i del objeto c que ambos comparten, luego se modifica en los dos.

Java pone restricciones para que los objetos no se puedan clonar directamente. El método clone() está definido como protected en la clase Object, luego un objeto sólo podría ser clonado dentro de la propia clase Object o en subclases suyas.

clone( ) genera una excepción(error) si no se puede realizar; esta excepción debe ser capturada.
Java obliga a que una clase cuyos objetos puedan ser clonados implemente la interfaz Cloneable.

Debemos tener especial cuidado cuando clonamos una clase cuyo atributo es una colección de objetos como ArrayList o array, puesto que la clonación de ese atributo supone la clonación de la estructura, pero no la de los objetos que contiene, que son referencias. En este caso debemos clonar previamente los objetos contenidos y después las estructuras.

```
/* ejemplo de efectos laterales al clonar objetos *//*prog_java_26*/
class C{
        int i=5;
}

class B{
        int j=2;
}
class A implements Cloneable{
        int k=10;
        C obc;
        B obb;
        A(){
                obc=new C();
                obb=new B();
        }
        public Object clone() {         //Es obligatorio sobreescribir el método
                try{
                return super.clone(); //invocamos el método clone( ) de Object
                }
                catch(CloneNotSupportedException e){
                  //capturamos la excepción si no puede clonar el objeto//
                        System.out.println("No se puede duplicar el objeto");
                        return null;
                }
        }
}
class ClonaObjeto{
        public static void main(String argv[]){
                A oba1=new A();
                System.out.println("oba1.obc.i: "+oba1.obc.i);
                System.out.println("oba1.obb.j: "+oba1.obb.j);
                System.out.println("oba1.k: "+oba1.k);
                A oba2;
                oba2=(A)oba1.clone(); //devuelve un Object //
                oba2.obc.i=9;//el objeto obc se modifica tanto para oba1 como oba2//
                System.out.println("oba1.obc.i: "+oba1.obc.i);
        }
}
```

```
/* Salida :
oba1.obc.i: 5
oba1.obb.j: 2
oba1.k: 10
oba1.obc.i: 9
*/
```

```
/* Clonación correcta de objetos */
class C implements Cloneable{
        int i=5;
    public Object clone() {
                try{
                return super.clone();
                }
                catch(CloneNotSupportedException e){
                        System.out.println("No se puede duplicar el objeto");
                        return this;
                }}
}
class B implements Cloneable{
        int j=2;
    public Object clone() {
                try{
                return super.clone();}
                catch(CloneNotSupportedException e){
                        System.out.println("No se puede duplicar el objeto");
                        return this;
                }}
}
class A implements Cloneable{
        int k=10;
        C obc;
        B obb;
        A( ){
                obc=new C();
                obb=new B();}
        public Object clone() {
                try{
                return super.clone();}
                catch(CloneNotSupportedException e){
                        System.out.println("No se puede duplicar el objeto");
                        return this;
                }}
}
public class ClonaObjeto {
  public static void main(String[] args) {
          A oba1=new A();
        System.out.println("oba1.obc.i: "+oba1.obc.i);
        System.out.println("oba1.obb.j: "+oba1.obb.j);
        System.out.println("oba1.k: "+oba1.k);
        A oba2;
        oba2=(A)oba1.clone();
        B RefB=(B)oba1.obb.clone();
        C RefC=(C)oba1.obc.clone();
        oba2.obb=RefB;
        oba2.obc=RefC;
        oba2.obc.i=9;
         System.out.println("oba1.obc.i: "+oba1.obc.i);
  }
}
```

POLIMORFISMO

1.- Introducción.

Uno de los factores más importantes de la calidad del software es la extensibilidad.
Buscamos programas extensibles: "Que respondan bien ante cambios en las especificaciones iniciales"
La abstracción es una buena herramienta para conseguir extensibilidad.
La abstracción de código en P.O.O. se consigue con el polimorfismo.
Abstracción: Herramienta intelectual para trabajar con los conceptos independientemente de las instancias particulares de éstos.
Polimorfismo **de tipos**:"Capacidad de una entidad a la que se permite tener valores de tipos diferentes durante la ejecución"
Importante para escribir código genérico (más abstracción)
Tipo estático : el asociado en la declaración.
Tipo dinámico: el correspondiente a la clase del objeto conectado en ejecución
Cuando una referencia puede tener más de un tipo dinámico se dice que puede realizar conexiones polimórficas.

Polimorfismo: "Capacidad para que un mismo mensaje enviado a diferentes objetos de distintas clases provoque un tratamiento diferente"

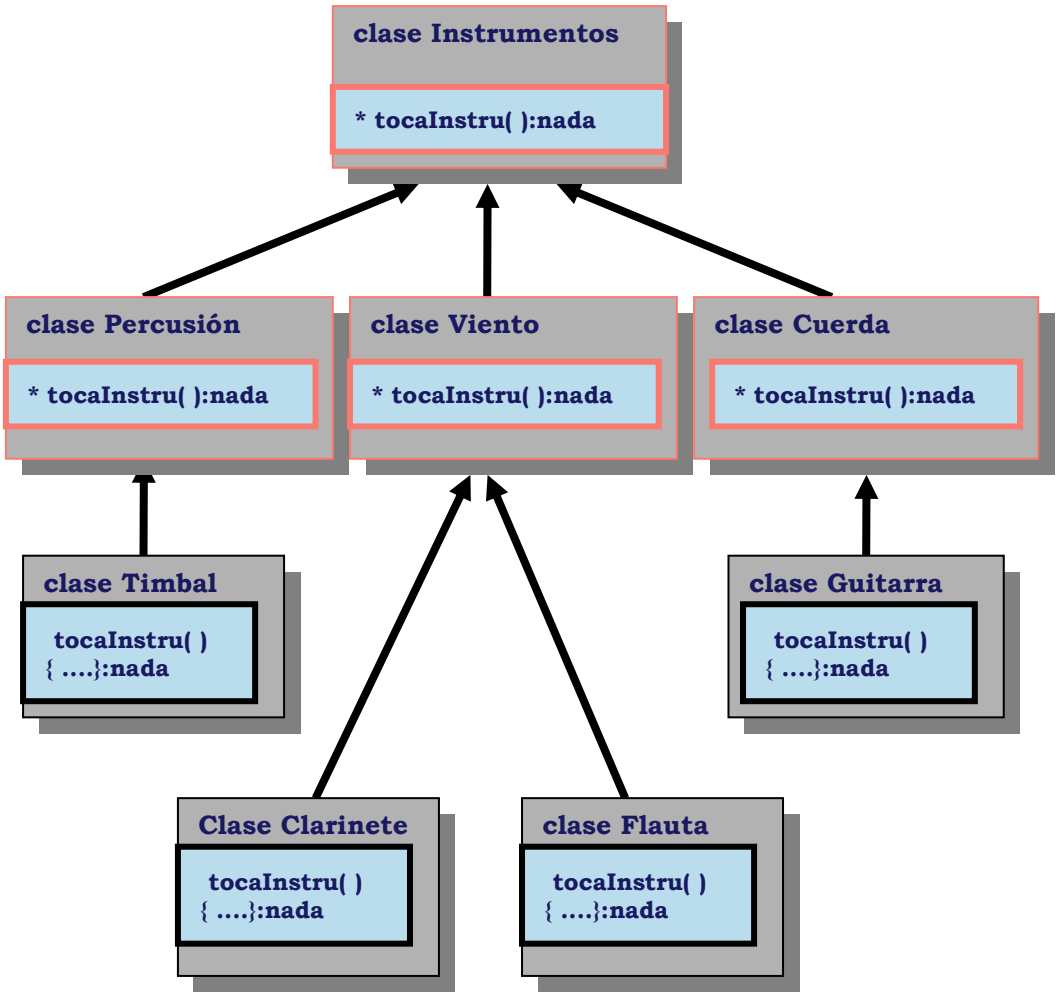
El mensaje debemos mandarlo a una referencia capaz de "conectar" con todos los objetos implicados (*referencia polimórfica*), y dicho mensaje es atendido por el método correspondiente en tiempo de ejecución (*ligadura dinámica*)

En el ejemplo, mandamos el mismo mensaje *"toca el instrumento .."* a distintos objetos t*imbal, clarinete,....* y cada uno responde de manera distinta *"Soy un timbal, ..."*

2.- Ligadura dinámica.

Ligadura estática: "El método invocado en el mensaje se conoce en tiempo de compilación" (paradigma imperativo).
Ligadura dinámica: "En una jerarquía de clases, la versión del método que se ejecutará será la correspondiente al tipo dinámico del objeto/referencia."
Sólo se conoce en ejecución.
Es recomendable la inclusión de clases virtuales/abstractas

Ejemplo :



* métodos abstractos

```
Instrumentos RefInstrumento
....
Timbal OTimbal=new Timbal( )
Clarinete OClarinete=new Clarinete( )
Flauta OFlauta=new Flauta( )
Guitarra OGuitarra=new Guitarra( )
....
RefInstrumento=OTimbal
RefInstrumento.tocaInstru()

RefInstrumento=OClarinete
RefInstrumento.tocaInstru()

RefInstrumento=OFlauta
RefInstrumento.tocaInstru()

RefInstrumento=OGuitarra
RefInstrumento.tocaInstru()
```

Ejemplo en Java :

```
/* ejemplo de clases abstractas y ligadura dinámica*//*prog_java_20*/
abstract class Figuras{
        double dim1,dim2;
        Figuras(double d1, double d2){
                dim1=d1;
                dim2=d2;
        }
        abstract void area();
}/* fin clase Figuras */
class Rectangulo extends Figuras{
        Rectangulo(double al, double an){
                super(al,an);
        }
        void area(){
                System.out.println("Area del rectángulo : "+dim1*dim2);
        }
}/*fin clase Rectangulo */
class Triangulo extends Figuras{
        Triangulo(double a, double b){
                super(a,b);
        }
        void area(){
                System.out.println("Area del Triángulo : "+(dim1*dim2)/2);
        }
}/*fin clase Triangulo */
class EjemploAreas{
        public static void main(String argv[]){
                Figuras f;      //es una variable, no un objeto
                Rectangulo r=new Rectangulo(10,10);
                Triangulo t=new Triangulo(5,5);
                f=r;
                f.area();
                f=t;
                f.area();
        }
}/* fin clase EjemploAreas */
```

```
/* Salida :
Area del rectángulo : 100.0
Area del Triángulo : 12.5
*/
```

En el programa anterior:

.....

f=r; *//Upcasting:Una referencia de una superclase apunta a un objeto de una subclase//*

f.area(); *// area() está definida para Figuras, de lo contrario no sería accesible. Se ejecuta el método correspondiente al objeto que 'apunta' o 'contiene'.*

f=t;

f.area(); *//ahora el método que ejecutará será area() de la clase a la cual pertenece el objeto t*

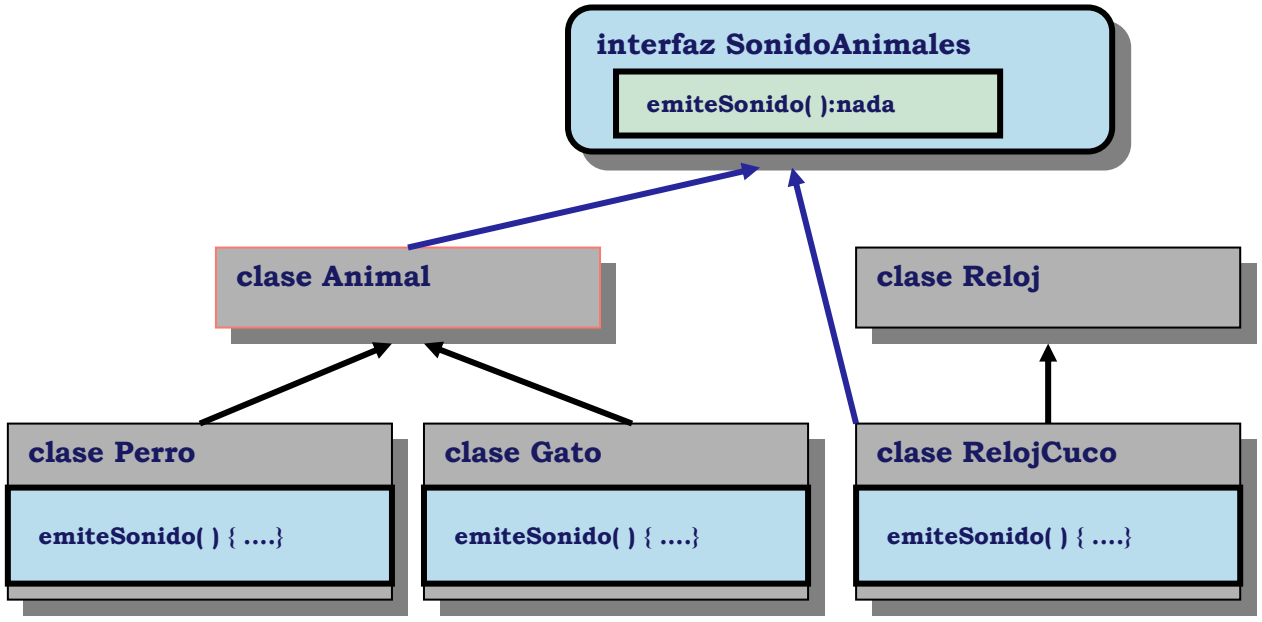
Al utilizar *f.area()* el compilador no sabe qué método tiene que aplicar. Sólo en ejecución se sabe (ligadura dinámica), cuando posee direcciones reales de memoria.

Si *area()* no fuese abstracta o no estuviera en la superclase no se podría utilizar *f.area()*.

3.- Polimorfismo con Interfaces.

Se puede obtener polimorfismo con el uso de interfaces.
También utilizamos ligadura dinámica.
La referencia de la interfaz puede conectar con objetos que hayan implementado sus métodos aunque no pertenezcan a la misma jerarquía de clases.
La ligadura dinámica con interfaces tiene el mismo mecanismo que la ligadura dinámica con clases.

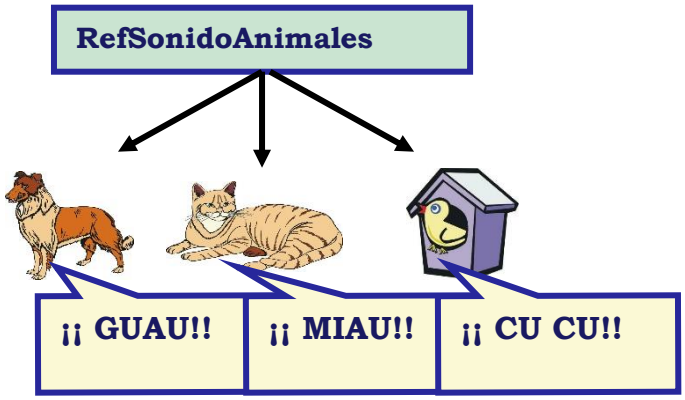
Ejemplo :



Esquema de clases en las que la interfaz SonidoAnimales es implementada a través de una clase abstracta Animal o bien directamente en RelojCuco
Las órdenes sintetizadas serían:

```
SonidoAnimales RefSonidoAnimales;
RefSonidoAnimales=OPerro
RefSonidoAnimales.emiteSonido( )
RefSonidoAnimales=OGato
RefSonidoAnimales.emiteSonido( )
RefSonidoAnimales=ORelojCuco
RefSonidoAnimales.emiteSonido( )
```

El efecto conseguido sobre los objetos :



Ejemplo de aplicación de interfaces para conseguir ligadura dinámica en Java :

```
/* ejemplo de interface  y uso con ligadura dinámica*//*prog_java_21*/
interface Intf{
        void metodo(int p);
}
class ClaseA implements Intf{
        public void metodo(int p){
                System.out.println("p : "+p); }
        public void otroMetodo(){
                System.out.println("otroMetodo");}
}
class UsaInterface{
        public static void main(String argv[]){
                ClaseA ob=new ClaseA();
                ob.metodo(7);
                ob.otroMetodo();
                Intf i;                 // variable de la interface
                i=ob;                   // la variable referencia al objeto
                i.metodo(12);
        //      i.otroMetodo();         //Method otroMetodo() not found in interface Intf.
        } }
/* Salida :
p : 7
otroMetodo
p : 12
*/
```

Podemos definir referencias cuyo tipo sea una interface, estas referencias permiten acceder a cualquier objeto de una clase que implemente la interface.

```
.....
ClaseA ob=new ClaseA();
Intf i;                          // variable de la interface
i=ob;                            // la variable referencia al objeto
.........
```

Si llamamos a un método del objeto a través de una referencia a interface, se realiza ligadura dinámica ejecutándose el método correspondiente a esa clase.

```
.....
i.metodo(12);
.....
```

Una variable que referencia a una interface, sólo puede acceder a los métodos definidos en la interface.

```
.....
i.otroMetodo();        //Method otroMetodo() not found in interface Intf.
.....
```

CONTENEDORES JAVA

1.- Introducción.

En las versiones de Java 1.0/1.1 el único contenedor de objetos que podía expandirse automáticamente era el Vector. Se trataba de un contenedor de Objects cuyo funcionamiento parece ser no era del todo fiable.
Paralelamente se utilizaba la interfaz Enumeration para tratar de manera automática secuencias de objetos.
En Java 2 (ver 1.3 en adelante) se han respetado las implementaciones de estas secuencias, pero a la vez se facilitan dos versiones más eficaces y seguras de los mismos, y por tanto recomendables: **ArrayList** e **Iterator.**

2.-La clase ArrayList.

Se trata de una implementación de array sin límites, que crece automáticamente cuando los objetos se van insertando en el mismo.
Sólo puede contener elementos de tipo Object.
Dispone de una serie de constructores y métodos para manipular dicha estructura.
Se debe importar del paquete java.util.

Constructores :

public ArrayList();
Crea un ArrayList vacío, por defecto 10 elementos.

public ArrayList(int capacidadInicial);
Crea un ArrayList con capacidadInicial de elementos.

Métodos :
public int size()
Devuelve el número de elementos del ArrayList.

public boolean isEmpty()
Devuelve cierto si el ArrayList está vacío (no hay elementos)

public Object get(int index)
Devuelve el elemento de la posición index.

public Object set(int index, Object element)
Devuelve el elemento de la posición *index* y lo reemplaza en esa posición por el objeto *element.*

***public boolean add(Object o*)**
Inserta el objeto o al fnal del ArrayList

public void add(int index, Object element)
Inserta element en la posición index; además desplaza desde index hacia la derecha los demás elementos que hubiera.

public Object remove(int index)
Elimina y devuelve el elemento de la posición *index* y desplaza los elementos a la derecha de index una posición a la izquierda.

public void clear()
Borra todos los elementos del ArrayList.

Muy importante, cuando pasamos referencias de objetos a los métodos de ArrayList, debemos asegurarnos que no serán posteriormente manipulados en el cliente (main, frame , etc..), porque si fuera así, se modificaría el objeto guardado en el ArrayList. Sería, en tal caso, conveniente crear los objetos a partir de las referencias pasadas o guardar un clon del mismo para evitar modificaciones inoportunas.

3.-Ejemplo de ArrayList.

Desarrollo una clase **Pila** con enteros implementada en un ArrayList.

```
/* Pila de enteros implementada en un ArrayList*/
import java.util.ArrayList;
class Pila{
        private ArrayList V;
        Pila(){
                V=new ArrayList();
        }
        public boolean pilaVacia(){
                return V.isEmpty();
        }
        public void apilar(int i){
                V.add(new Integer(i));
        }
        public int desapilar(){
                Integer I=(Integer) V. remove (V.size()-1);
                return I.intValue();
        }
}/* fin clase Pila */
public class UsaPila{
        public static void main(String a[]){
        Pila P=new Pila();
        if(P.pilaVacia())
                System.out.println("La pila está vacía");
        P.apilar(7);
        P.apilar(12);
        P.apilar(3);
        P.apilar(9);
        while(!P.pilaVacia())
                System.out.println("Desapilado : "+P.desapilar());
}
}
```

```
/* Salida :
La pila está vacía
Desapilado : 9
Desapilado : 3
Desapilado : 12
Desapilado : 7
*/
```

4.-La interfaz Iterator.

Define tres métodos que permiten enumerar(obtener uno a uno) los elementos de un conjunto de objetos.

public boolean hasNext()
devuelve true si hay más elementos en la enumeración.

public Object next();
devuelve el objeto siguiente de la enumeración.

public void remove()
borra el último elemento devuelto por la interfaz, no es muy interesante pero es necesario sobrescribirlo.

Ejemplo:

```
/* ejemplo que implementa la interfaz Iterator */
import java.util.Iterator;      //debemos importar la interfaz //
class Iter implements Iterator{
        private int cont=0;
        private boolean mas=true;
        public boolean hasNext(){     //debe sobreescribir el método
                return mas;
        }
        public Object next(){         //debe sobreescribir el método
                cont++;
                if(cont==5)
                        mas=false;
                return new Integer(cont);
        }
        public void remove(){ }
}
class UsaIterator{
        public static void main(String argv[]){
                Iter enum=new Iter();
                while(enum.hasNext()){
                        System.out.println(enum.next());
                }
        }
}/* fin clase UsaIterator */
```

```
/* Salida :
1
2
3
4
5
*/
```

Un objeto de la interfaz Iterator puede estar asignado a cualquier clase, en este caso Iter.
Es un molde para crear secuencias. *El método main() siempre será el mismo.*
La secuencia de objetos puede cambiar, pero la interfaz se comporta igual.

5.-Ejemplo práctico de interfaz Iterator.

```
/* Enumeración de los 10 primeros números primos */

import java.util.Iterator;
class SeriePrimos implements Iterator{
        private int num=1;
        private int cont=1;
        private int limite=10;
        private boolean esPrimo(int i){
                int div=1,resto;
                do{
                        div++;
                        resto=i%div;
                }while((resto!=0)&&(div<=i/2));
                return(!(div<=i/2));
        }
        public boolean hasNext(){
                return (cont<=limite);
        }
        public Object next(){
                while(!esPrimo(num))
                        num++;
                cont++;
                Integer numero=new Integer(num);
                num++;
                return numero;
        }
        public void remove() { }
}/* fin clase SeriePrimos */
```

```
public class ListaPrimos{
        public static void main(String argv[]){
                Iterator e=new SeriePrimos();
                while(e.hasNext())
                        System.out.println(e.next());
        }
}/*fin clase ListaPrimos */

/* Salida :
num: 1
num: 2
num: 3
num: 5
num: 7
num: 11
num: 13
num: 17
num: 19
num: 23
*/
```

GESTION DE EXCEPCIONES

1.- Introducción.

Una **excepción** es una condición anormal(error) que surge en una sentencia ***durante la ejecución*** de un programa. Errores típicos son matemáticos y de índices de arrays.

Cuando se produce un error se genera automáticamente un objeto que representa esa excepción y se envía al método que lo ha provocado.

Esta excepción debe ser capturada y procesada.
Forma general de controlar excepciones:

```
try {
        //bloque de código de riesgo donde pueden surgir excepciones
}
catch(TipoDeExcepción e){   // e es una variable donde se guarda el objeto del error
        //gestión de la excepción
}
...
....
finally {bloque que siempre se ejecuta}
 //tanto si se captura, como si no hay error; incluso si Java corta la ejecución del programa//
```

Si se produce una excepción y el programa no la captura hay un gestor por defecto que lo hace y realizará lo siguiente :

Muestra un mensaje con el tipo de excepción y el lugar donde se produjo (una traza de la pila).
Termina la ejecución del programa.

Ejemplo:

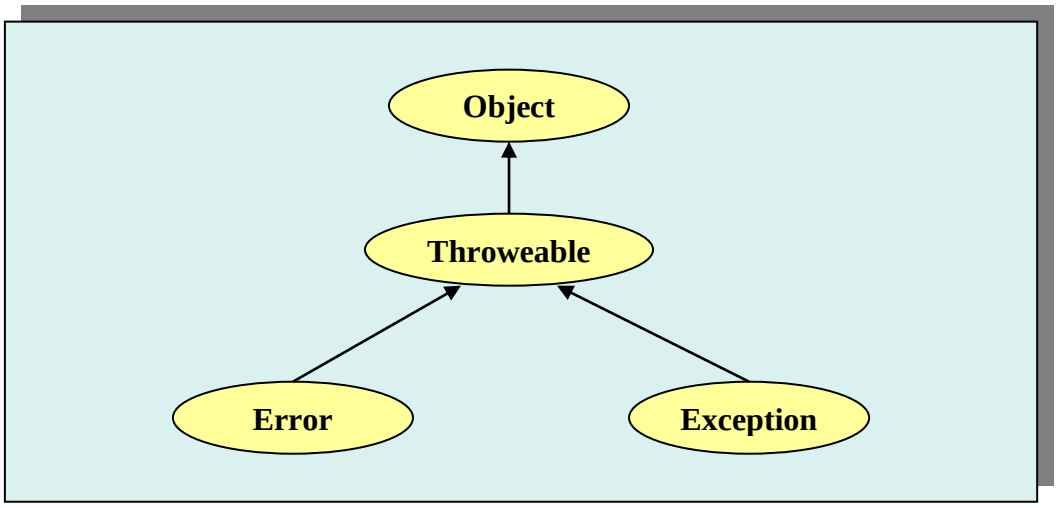
```
/* ejemplo de excepción no capturada *//*prog_java_27*/

class DivPorCero{
        public static void main(String ar[]){
                int d=0;
                int a=7/d;
        }
}
```

```
/* Salida :
java.lang.ArithmeticException: / by zero
        at DivPorCero.main(prog_java_27.java:5)
*/
```

método fichero número de línea

2.- Tipos de excepciones.

Las excepciones son objetos de subclases de la clase **Throweable** (‘lanzable’).



Exception : clases de errores que suelen capturar los programas :
Errores matemáticos
Errores de índices en arrays....

Error : clase de error que no suele capturarse normalmente puesto que son fallos graves.
Desbordamiento de pila ....

El paquete java.lang define una serie de excepciones:

Exception
- RuntimeException
  - ArithmeticException
  - IndexOutOfBoundsException
    - ArrayIndexOutOfBoundsException
    - StringIndexOutOfBoundsException
  - NegativeArraySizeException

Las ‘RuntimeException’ no se comprueban en tiempo de compilación, ni existe obligación de capturarlas. Si se produce una excepción lo más seguro es que sea de alguna de estas clases.

3.- Gestión de excepciones.

```
/* ejemplo de gestión de excepciones *//*prog_java_28*/

import java.util.Random;
class ManejaError{
        public static void main(String argv[]){
                int a=0,b=0,c=0;
                Random r=new Random();
                for(int i=0;i<3200;i++){
                        b=r.nextInt(); // genera nuevos números aleatorios en el rango int//
                        c=r.nextInt();
                        try{
                                a=77/(b/c);
                        }
                        catch(ArithmeticException e){
                        System.out.print("División por cero   ");
                        a=0;
                        }
                        System.out.println("a: "+a);
                }/* fin for */
         }/*fin main */
}/*fin clase ManejaError*/
```

La parte donde se puede producir un error se marca con un bloque 'try'(siempre entre {}), aunque sea una sentencia.
El 'catch' debe ir inmediatamente después del bloque 'try', indicando el tipo de excepción que queremos capturar y una variable donde se guardaría la misma.
Si el error se produce en medio del bloque 'try', se salta directamente al bloque 'catch' sin ejecutar las sentencias que quedaran.
Un 'catch' con una superclase de excepción captura todas las excepciones generadas que sean subclases de ella.

Ejemplo:

```
...
catch(Throweable e){
        //captura cualquier excepción //
}
....
catch(Exception e){
        //captura todas las subclases de Exception//
}
...
catch(RunTimeException e){
        //captura todas las que englobe la clase//
}
```

catch múltiples.
Como un mismo bloque puede producir errores distintos, un 'try' puede tener asociados varias clausulas 'catch'. Estas clausulas se inspeccionan en orden y se ejecuta la primera que coincide con la excepción generada.
Debemos poner antes las subclases que las superclases.

Ejemplo:

```
try{
        .....
        array[a/b] ...
        .....
}
catch(ArithmeticException e1){
        .....
}
catch(IndexOutOfBoundsException e2){
        .....
}
.....
```

4.- Sentencias 'try' anidadas.

Un bloque completo 'try/catch' puede estar dentro de otro bloque 'try'.
Los 'catch' están asociados a su 'try' inmediato.
Si se produce un error dentro de un 'try' y su 'catch' no lo captura, se saltará al siguiente 'catch' más externo.
No es lo mismo que las catch múltiples puesto que en éstas sólo se ejecuta un catch, mientras que las anidadas también se ejecuta una , pero puede no pertenecer a su try inmediato. En try anidadas, lógicamente programaremos en cada try el catch que más posibilidades tenga de producirse.

```
/* ejemplo de sentencias try anidadas */

class TryAnidada {
public static void main(String argv[]){
try{
        int a=argv.length;
        /* si no hay ningún argumento en la línea de órdenes,
        la siguiente sentencia generará una división por cero */
        int b=42/a;
        System.out.println("a="+a);
        try{   //bloque try anidado
                /*Si se utiliza un argumento en la línea de órdenes,
                la siguiente sentencia generará una excepción de
                división por cero */
                if(a==1) a=a/(a-a);   //división por cero

                /* Si se le pasan dos argumentos en la línea de órdenes,
                se genera una excepción al sobrepasar los límites del
                tamaño de la matriz */
                if (a==2){
                        int c[]={1};
                        c[42]=99; //genera una excepción fuera de límites
                }
        }catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Indice fuera de límites:"+e);
        }
        /****/
}catch(ArithmeticException e){
        System.out.println("División por 0:"+e);
}
}
}/* fin clase */
```

Puede producirse anidamiento de 'try/catch' al llamar a un método dentro de un bloque 'try/catch'.

```
/*las sentencias try pueden estar implícitamente anidadas a través de las llamadas a métodos*/
class MetodoAnidado{
        static void metodoAnidado(int a){
                try{
                /*Si se utiliza un argumento en la línea de órdenes,
                la siguiente sentencia generará una excepción de
                división por cero */
                if(a==1) a=a/(a-a);   //división por cero

                /* Si se le pasan dos argumentos en la línea de órdenes,
                se genera una excepción al sobrepasar los límites del
                tamaño de la matriz */
                if (a==2){
                        int c[]={1};
                        c[42]=99; //genera una excepción fuera de límites
                }
        }catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Indice fuera de límites:"+e);
        }
        }
        public static void main(String argv[]){
        try{
                int a=argv.length;
                /* si no hay ningún argumento en la línea de órdenes,
                la siguiente sentencia generará una división por cero */
                int b=42/a;
                System.out.println("a="+a);
                metodoAnidado(a);
        }catch(ArithmeticException e){
                System.out.println("división por 0 :"+e);
        }
        }
}
```

5.- Excepciones explícitas.

Podemos crear y lanzar una excepción mediante la sentencia *throw.*
Forma:
throw ObjetoThrowable

Un objeto throwable es de la clase Throwable o cualquier subclase suya.
Existen dos formas de generar objetos throwables:
Creándolo con un new.
Poniéndolo como parámetro de un catch.

Ejemplo:

```
/* ejemplo de una excepción:crearla,recogerla y volver a lanzarla *//*prog_java_29*/
class ThrowDemo{

        static void ejemplo(){
                try{
                throw new NullPointerException(".....Excepción provocada");
                }
                catch(NullPointerException e){
                System.out.println("Excepción capturada dentro del ejemplo : "+e);
                throw(e);
                }
        }/* fin clase ejemplo()*/

        public static void main(String argv[]){
                try{
                        ejemplo();
                }
                catch(NullPointerException e){
                        System.out.println("Nueva captura:...."+e);
                }
        }/* fin main()*/

}/* fin clase ThrowDemo */
```

```
/* Salida :
Excepción capturada dentro del ejemplo :  java.lang.NullPointerException: .....Excepción provocada
Nueva captura:....java.lang.NullPointerException: .....Excepción provocada
*/
```

Todas las excepciones tienen 2 posibles constructores heredados de Throwable, sin parámetros o con una cadena descriptiva.

6.- Creación de excepciones propias.

Podemos crear nuestros propios tipos de excepciones definiendo subclases de Exception.
Las nuevas clases no necesitan implementar nada, heredan los métodos de la clase Throwable.
Métodos principales:

String getMessage()
devuelve la descripción de la excepción.

void printStackTrace()
imprime la traza de la pila de ejecución.

String toString()
descibe la excepción.

Constructores:
Con String descriptivo.
Sin parámetros.

```
/* ejemplo de creación de una excepción propia *//*prog_java_30*/
class MiExcepcion extends Exception{
        private int codigo;
        MiExcepcion (int a){
                codigo=a;
        }
        public String toString(){
                return "MiExcepcion, código :"+codigo;
        }
}/* fin clase MiExcepcion()*/
class EjMiExcepcion{
        static void lanzadera(int a) throws MiExcepcion{
                if(a>0)
                        throw new MiExcepcion(a);
                else
                        System.out.println("Final normal");
        }
public static void main(String argv[]){
        try{
                lanzadera(-1);
                lanzadera(1);
        }
        catch (MiExcepcion e){
                System.out.println("Excepción capturada : "+e);
        }
}
}/* fin clase EjMiExcepcion()*/
```

```
/* Salida :
Final normal
Excepción capturada : MiExcepcion, código :1
*/
```

Comentarios :

static void lanzadera(int a) throws MiExcepcion{
Si omitimos lo subrayado, el compilador avisa que la excepción o bien debe ir en 'throws' o ser capturada en el mismo método con 'try/catch'.

lanzadera(-1);
lanzadera(1);
Si llamamos antes con valor 1, no se hacen las siguientes instrucciones del 'try', no haría el valor –1.

catch (MiExcepcion e){
Si no se captura hay un error de compilación.

7.- Excepciones no capturadas en métodos.

Si un método puede provocar una excepción que no maneja él mismo, debe avisar para que los métodos que lo utilicen se protejan ante esa excepción.
Para ello se debe incluir la clausula **throws** en la cabecera del método.
La clausula throws debe incluir todos los tipos de excepciones que un método puede lanzar.
No es necesaria para excepciones de la clase Error o RuntimeException o sus subclases.

Ejercicio propuesto 1 :

Determinar la salida que se produciría al ejecutar el siguiente programa.
Corregir previamente los posibles fallos que encontréis en el código:

```
class E1 extends Exception{};
class E2 extends Exception{};

class A{
	void metodo(int i) throws E1,E2{
		if(i<0) throw E1;
		if(i>100) throw E2;
		System.out.println(i);
	}
}

class B{

	A a=new A();
	void metodo(int i) throws E1{
		try{
			a.metodo(i);
		}catch(E2 e){
			System.out.println("E2");
		}
	}
}

class C{
	B b=new B();
	void metodo(int i) throws E1{
		try{
			i=i*2;
			if(i>500) return;
			b.metodo(i);
		}catch(E1 e){
			System.out.println("E1");
			throw(e);
		}
		finally{
			System.out.println("EE");
		}
	}
```

```
public static void main(String argv[]){
        C c=new C();
        try{
                c.metodo(10);
                c.metodo(-80);
                c.metodo(-1);
        }catch(E1 e){
                System.out.println("MM");
        }finally{
                System.out.println("FIN");
        }
}
}
```

Ejercicio propuesto 2 :

Determinar la salida que se produciría al ejecutar el siguiente programa.

```
class E1 extends Exception{};
class E2 extends Exception{};
class A{
        void metodo(int i) throws E1,E2{
                if(i<0) throw new E1();
                if(i>100) throw new E2();
                System.out.println(i);
        }
}

class B{

        A a=new A();
        void metodo(int i) throws E1,E2{
                try{
                        a.metodo(i);
                        throw new E2();
                }catch(E2 e){
                        System.out.println("E2");
                }
        }
}
```

```
class C{
	B b=new B();
	void metodo(int i) throws E1{
		try{
			i=i*2;
			if(i>500) return;
			b.metodo(i);
		}
		catch(E1 e){
			System.out.println("E1");
			throw(e);
		}
		catch(E2 e){
			System.out.println("E2");
		}

		finally{
			System.out.println("EE");
		}
	}

public static void main(String argv[]){
	C c=new C();
	try{
		c.metodo(10);
		c.metodo(80);
		c.metodo(-1);
	}catch(E1 e){
		System.out.println("MM");
	}finally{
		System.out.println("FIN");
	}
}
}
```

PROYECTOS JAVA

Ideas y consejos generales para la realización de Proyectos Java.

1.- Toda clase debe ser programada para una posible reutilización en otra aplicación, debe ser lo más genérica posible. Os debéis imaginar que la podríais vender a un cliente finlandés, por ejemplo.
2.- Los requerimientos particulares del ejercicio se implementan en el main(), usando la clase diseñada. Este main() se supone que lo va a hacer nuestro cliente finlandés usando nuestra clase.
3.- No debéis meter en la clase requerimientos propios del ejercicio en cuestión, pues la haría menos genérica. Claro, porque el cliente finlandés puede querer una cosa de nuestra clase y otro, moldavo, otra distinta. Por ejemplo una clase Estanteria que podría ser usada para meter cajas en una determinada ocasión, la tentación consistiría en declarar como atributo un ArrayList de cajas. Pero esa clase Estantería ya no podría ser usada para otra cosa que no fuese meter cajas, perdería genericidad.
4.- Toda la transacción de datos (entrada/salida) se hace en el método main(), no en métodos particulares de las clases.
5.- Nunca se escribe dentro de un método de una clase, pues el texto acompañaría a la misma. Si el texto está en español, mis clientes al ejecutar el método no entenderían lo que pone. Lo ideal es devolver un String al main() con los datos y en el main() mi cliente los escribe en el idioma que quiere (como el método toString()..).
6.- Los atributos son siempre privados y los métodos públicos, salvo excepciones. Nunca desde el main() podríamos acceder a los atributos de una clase directamente, sino a través de los métodos que yo le he puesto a dicha clase.
7.- Un atributo protected implica confianza en quien va a heredar de tu clase.
La cuestión es la siguiente, si declaras atributos protected, quien herede de la clase puede utilizar directamente esos atributos, sin pasar por los métodos que tú hubieras puesto para su acceso ¿qué significa?, pues que si quien hereda hace un mal uso de los mismos, probablemente las culpas vayan al programador de la clase heredada.
Es una cuestión de confianza y de responsabilidad, si tú haces una clase quieres que funcione perfectamente para ello "proteges" tus atributos.
Por ejemplo, si tienes un peso como atributo, tú te preocupas de que no se le pueda asignar un número negativo al 'double', para ello pones un método que lo detecte, pero si el atributo es protected, quien la hereda va directamente al mismo y puede colocar un valor negativo, y saltar una excepción, etc, etc.
¿quién es el culpable?....
Os falta perspectiva porque vosotros hacéis todas las clases, y además las usáis.
En realidad esto implica a varias personas.
8.- Cuando nuestros atributos sean Objetos, cuidado con las referencias cruzadas, sobre todo en estructuras de datos genéricas (ArrayList, HashMap,List, etc…).

Proyecto 1.- Trenes.

Se precisa implementar una aplicación en Java para simular la gestión que hace Renfe de sus trenes. Nuestro analista nos ha ofrecido una serie de requisitos que deben ser satisfechos por nuestra aplicación y unas especificaciones que deben tener las clases a codificar.

Éstas son :

Existirá una entidad Locomotora que responderá a las siguientes características:

- Código.
- Marca.
- Modelo.
- Tipo , según la energía utilizada para la tracción :
  - Eléctrica.
  - Diesel.
- Potencia de tracción en CV.

Hay dos tipos de vagones, con las características:

- Vagón pasajeros.
  - Código.
  - Marca.
  - Modelo.
  - Número de asientos.
- Vagón mercancías.
  - Código.
  - Marca.
  - Modelo.
  - Capacidad de carga (Toneladas).

Un Tren se especifica como la unión de una locomotora y un número variable de vagones y tipos de éstos (pasajeros y/o mercancías).

Con estos datos se pide:

a) Escribir las clases correspondientes a las entidades antes descritas.
- Describir sólo los métodos imprescindibles para que la simulación pueda funcionar (*quizás no hagan falta todos los métodos set/get*).
- No controlar que la entrada de datos sea correcta.
- Es obligatorio utilizar Herencia o Composición cuando sea justificado.

b) Se ha de dar al usuario la posibilidad de construir objetos de tipo tren formados por un objeto locomotora, 3 objetos vagón de pasajeros y 1 objeto vagón de mercancías. La información se introducirá directamente desde el main().

c) Visualiza estos objetos "tren" con toda la información que posee (descripción completa de locomotora, vagones, etc ....).

Proyecto 2.- Estaciones.

Desde ADIF nos piden que implementemos la gestión de las estaciones de trenes que forman parte de la red nacional de ferrocarriles.
El analista nos ha identificado varias entidades que se orientan a la codificación de cada estación:
Tren que circula, reflejaremos:

- código,
- nombre,
- destino,
- hora llegada
- hora salida
- número de vía que ocupa.

Si el tren es de mercancías se añade el tipo de tren según la carga (automóviles, graneles, intermodal, etc…).
Cuando el tren circulante sea de pasajeros, adjuntaremos el número de plazas del mismo.
Para implementar la entidad estación, tendremos en cuenta el nombre que tiene, la ciudad donde está y que hay una serie de trenes que circulan cada día.

Se pide :
1.- Construir las clases a partir de las especificaciones, empleando el paradigma orientado a objetos (herencia, composición, etc …)
Estará terminantemente prohibido obligar al cliente (main()) a emplear estructuras de Java que no son de su responsabilidad (arrays,listas, etc …).

2.- Con estas clases, construir – como ejemplo- una estación en la que circulen diariamente 5 trenes de pasajeros y 3 de mercancías.

3.- Describir la información completa de la estación (incluidos los trenes que circulan). Los trenes se mostrarán ordenados por hora de salida de la estación.

4.- Obtener la relación de trenes que circulan hacia una determinada ciudad (por ejemplo: trenes con destino “MADRID”).

5.- Crear una excepción propia llamada ***PeligroColisionException*** que salte cuando demos de alta un tren cuya hora de entrada y vía pueda coincidir con la de otro ya registrado. Por ejemplo, si un tren está registrado en la vía 2, llega a las 12:30 y sale a las 12:40, por esa vía no podemos registrar otro tren que tenga prevista su llegada a las 12:35.
Esta excepción se capturará en el main( ) de la aplicación.

6.- Clonar el objeto Estacion creado anteriormente. Para su demostración, implementar un nuevo método quitarTren(<código>), se aplicará a uno de los dos objetos Estacion, y se utilizará para comprobar que la estación original y la clonada no son la misma.

Proyecto 3.- Cesta compra.

Se precisa implementar una aplicación en Java para línea de comandos para simular la realización de la compra de alimentos por parte de una persona.

Nuestro analista nos ha ofrecido una serie de requisitos que deben ser satisfechos por nuestra aplicación y unas especificaciones que deben tener las clases a codificar.

Éstas son :

- Existirá una entidad **Alimento** que responderá a las siguientes características:
  - Código (Cadena de 4 caracteres)
  - Marca (Cadena de 10 caracteres)
  - Descripción (Cadena de 15 caracteres)
  - Precio (Euros).
  - Tipo : Dietético, Ecológico y Normal.
    - Para los alimentos Dietéticos:
      - Número de Calorías por 100 gr.
    - Para los alimentos Ecológicos:
      - Lugar de procedencia.

Ejemplos :

1123	Puleva	Leche semidesnatada	1,18 €

4554	Santiveri	Galletas integrales choco	1,67 €	90 kcal/100 gr

7550	Pitis	Vino tinto crianza 2005	3,80 €	Madrigueras (AB)

- Además para realizar la compra estableceremos un límite de gasto en Euros.

- Con estos datos se pide:

a) Escribir las clases correspondientes a las entidades antes descritas.
  - Describir sólo los métodos imprescindibles para que la simulación pueda funcionar (*quizás no hagan falta todos los métodos set/get*).
  - Es obligatorio utilizar Herencia, Composición y Polimorfismo cuando sea justificado.

b) Introduce varios alimentos y unidades de los mismos en la lista de compra y que se incluya al menos uno de cada tipo.

c) Visualiza la lista de compra con toda la información que posee.

d) Crea una excepción propia llamada ***ExcepcionLimiteGastoSuperado*** que se genere cuando al introducir alimentos en la lista el acumulado supere el límite de gasto establecido en euros. Dar el mensaje oportuno.

Proyecto 4.- Urbanización.

Aunque no corren buenos tiempos para la construcción en este país, el rescate facilitado por la UE ha animado a los bancos a prestar dinero nuevamente a las constructoras.
CONSTRUCCIONES ALBACETE, vuelve a retomar su proyecto de crear una urbanización en el término municipal de Madrigueras, en Albacete y junto al río Júcar.
Esta constructora nos ha pedido que hagamos la simulación de dicha obra, y hemos enviado a nuestro analista, que tras un estudio exhaustivo nos ha facilitado el guión de la misma.
Ha identificado las entidades que deberemos diseñar en Java, y además una simulación que nos detallará posteriormente.

Entidades :

- Vivienda.
  - Reflejaremos la superficie, el número de plantas y el número de ventanas que tiene. Debemos controlar que no admitan valores absurdos ni fuera de la legalidad (superficie mínima 40 m^2 y 1 ventana por lo menos).
- Chalet.
  - Si hacemos chalets serán todos iguales con jardín y piscina (dimensiones mínimas de jardín 1 m^2 y de piscina 1x1x1 metros).
  - Se permiten hasta 3 plantas por chalet.
- Edificio.
  - Se podrán construir edificios hasta de 15 alturas (por ahora, pero ya veremos si se puede negociar).
  - Sólo podrá existir una vivienda por planta.
  - Obligatoriamente tendrán un jardín comunitario de no menos de 100 m^2 y una piscina de 100x50x2 metros como mínimo.
- Urbanización
  - Será un conjunto de edificios y/o chalets, cuyo número se indicará en el constructor de la misma.

Todos los errores se deben controlar con al menos dos excepciones:

DimensionIncorrectaException , para los errores de dimensión descritos anteriormente.
Se debe informar de qué dimensión ha producido el error y cuál ha sido la magnitud que lo ha provocado.

EdificioCompletoException , cuando intentemos añadir un nuevo piso a un edificio y hemos superado el número de plantas.

Simulación:
- Crear la urbanización “Covasyermas” suponiendo que constará de 2 edificios:

- “Las Rocas”.
  - 10 alturas.
  - 200 m^2 de jardín.
  - Una piscina de 100x50x3 m.
  - Cada piso será de 90 m^2 y 4 ventanas.

- "Las Rocas DeLux".
    - 15 alturas.
    - 500 m^2 de jardín.
    - Piscina de 150x75x4 m.
    - Cada piso de 120 m^2 y 8 ventanas.

- 20 chalets tipo:
    - Superficie 250,5 m^2.
    - 8 ventanas.
    - 2 plantas.
    - 20 m^2 de jardín.
    - Piscina de 20x5x2 m.

- Obtener la descripción completa de toda la urbanización.

Proyecto 5.- Subastas.

Tras el desastre que produjo la burbuja inmobiliaria en este país, se han quedado un número ingente de viviendas sin vender. El estado, muy amablemente, se las ha comprado a los bancos para que no corran el riesgo de quiebra. Así ha surgido la Sociedad de Gestión de Activos procedentes de la Reestructuración Bancaria, más conocida por su acrónimo, Sareb. Esta sociedad es la que centraliza todas las viviendas que no se pudieron vender en su momento.
El estado ha decidido sacar a subasta pública dichas viviendas.
La idea es que las personas interesadas en comprar una vivienda puedan pujar por una de ellas.
El Ministerio de Hacienda nos ha encargado una aplicación con **Java** para gestionar dichas pujas.
Se basa en lo siguiente:
- De las viviendas se debe registrar:
    - Código.
    - Tipo (piso, solar, chalet, comercial).
    - Localización (calle, número, población, provincia).
    - Precio de salida o máximo subastado (en Euros).

- De los subasteros se debe registrar:
    - NIF.
    - Apellidos
    - Nombre.

La aplicación debe contemplar los siguientes supuestos:
- Creación de varios objetos de tipo vivienda y subastero.
- Simulación de la subasta, cada subastero puede pujar por varias vivienda pero una sola vez. Si el precio ofrecido es mayor que el registrado para esa vivienda, la misma se grabará en la estructura correspondiente del subastero.
- Se deberá lanzar una excepción *PujaException* con mensaje “Puja ya realizada”, cuando un subastero repita puja por la misma vivienda.
- Si en una puja posterior, algún otro subastero superase el precio anteriormente ofertado, la vivienda aparecería en este nuevo y se borraría del anterior.
- Es posible que se queden viviendas sin adjudicar y subasteros que no hayan conseguido ninguna.

Se pide:
1.- Diseño correcto de las clases.

2.- Registro de subasteros.

3.- Gestión de pujas. Control de excepciones.

4.- Listado final con las viviendas asignadas (todos los datos) y el subastero ganador de la puja (todos los datos). La salida se hará utilizando la interfaz Iterator.

Proyecto 6.- Despedida de soltera.

Dentro de un mes, a pesar de la pandemia, se casa en Madrigueras la Manoli. Como es costumbre, las amigas le han organizado un viaje de despedida de soltera a Benidorm, y más ahora que está barato.
Tomás, el de la agencia de viajes, les ha dicho que deben elegir el hotel, decirle las que son, los días que se van y los extras que quieren contratar cada una en ese hotel. Les da el folleto de Benidorm y se lo llevan para pensarlo.

Al día siguiente ya lo tienen claro.
Le cuentan al de la agencia los planes que tienen:
Se van a ir para hacer solo una noche, del sábado 24 de Octubre al Domingo 25 de Octubre.
El hotel elegido es DGHoteles, que cuesta 45 € pensión completa por día.
Los extras que ofrece son:
- Habitación individual: 15 €.
- Masaje : 25 €.
- Excursión a visitar fábrica de turrones : 20 €.

Los extras que puede ofrecer el hotel son variables y dependen de la temporada, las cuantías también pueden variar, pero en este mes es lo que hay.

Las amigas que van y los extras elegidos son:
- Manoli (la novia), quiere una habitación individual, el masaje y la excursión.
- La Laura y la Jenny no quieren ningún extra y ocuparán una habitación doble.
- La Mercedes quiere el masaje y la excursión.
- La Cathy quiere sólo el masaje y compartirá habitación con la Mercedes.

Como tiene que estar registradas en el hotel, cada una debe aportar los siguientes datos:
- NIF.
- Nombre.
- Número tarjeta crédito.

Por tanto, rellenan la siguiente tabla:

NIF	NOMBRE	NÚMERO TARJETA
32217687S	MANUELA PÉREZ SÁNCHEZ	5470007655432112
09897654R	CATALINA FERNÁNDEZ MILLA	5478766655432680
74565342K	LAURA GARCÍA OROZCO	5400096655432777
34207689L	MARCEDES NIETO RODRÍGUEZ	5478112655432455
67807609J	JENNIFER MARTÍNEZ URREA	4533266655432655

Con estos datos Tomás ya puede hacer la reserva porque necesita:
Destino del viaje, días que se van, hotel elegido y la relación de amigas con sus extras.

El jefe de la agencia, harto de hacer a mano el resumen de la reserva, le ha pedido a un amigo (Juan Carlos) que le haga el diseño de la aplicación, pero por no disponer de tiempo ha hecho un resumen de lo que le haría falta a Tomás.

Este nos contrata para hacer la aplicación en Java teniendo en cuenta las directrices que le ha dejado su amigo Juan Carlos:
- Debemos recabar datos del hotel y de los servicios que presta en este momento.
- Registrar a cada amiga con sus datos personales.
- Asignar a cada amiga los extras que contrata.
- Registrar estas amigas en la reserva del viaje.
- Escribir los datos completos de todos a través de la reserva, con inclusión de los extras de cada una, lo que le cuesta a cada amiga y el total completo.

Para el ejercicio en Java se valorará:
1.- Diseño correcto de clases.
2.- Creación del objeto Hotel.
3.- Creación de los objetos Amiga con sus extras.
4.- Creación del objeto Reserva.
5.- Impresión completa de la reserva. Iterator.

Proyecto 7.- Multicines.

En la ciudad de Albacete, se han abierto unas nuevas salas de un proyecto de multicines.
La idea que nos transmiten es que diseñemos una aplicación en Java que dé respuesta a los requisitos que son inherentes a cualquier complejo de multicines.
Las conclusiones que hemos obtenido con los requerimientos aportados son las siguientes:

- Las películas tendrán:
  - Código.
  - Título.
  - Género.
  - Director/a.
  - Serie de actores/actrices principales. Pueden intervenir en más de una película.
  - Fecha de estreno.
  - Duración (minutos).
  - Calificación moral.
- Del actor/actriz reflejaremos:
  - Nombre.
  - Nacionalidad.
  - Fecha nacimiento.
- La sala donde se proyectan tiene:
  - Un número que la identifica.
  - Aforo.
- Cuando una película se proyecte, identificamos:
  - La película.
  - Hora de proyección (hh:mm).
- El multicine tendrá datos generales del mismo.
  - Nombre.
  - Ubicación.

Aspectos importantes:
- No todos los datos de los actores se han introducido.
- Se deberá lanzar una excepción ***HoraProyeccionException*** consistente solamente en un mensaje que diga "Se solapa con proyección anterior ", cuando al intentar programar una película en una sala a una determinada hora no dé tiempo suficiente a que la anterior haya acabado. Además se contempla un descanso de 15 minutos entre proyección y proyección.
- Simular la proyección de una película en más de una sala como clonación de la misma.
- Se aporta el fichero de prueba [*SalasCineMain.java*] que debéis seguir fielmente para la simulación.

[*SalasCineMain.java]*

```
package SalasCine;
import cines.Actor;
import cines.HoraProyeccionException;
import cines.Multicine;
import cines.Pelicula;
import cines.Sala;
import java.time.LocalTime;
import static java.time.LocalDate.of;
/**
 * @author Daniel
 */
public class SalasCineMain {
    public static void main(String[] args){
        /* Alta Multicine */
        Multicine Avenida=new Multicine("Multicines Avenida", "Albacete");
         /* Creación de las salas */
        Sala Sala1=new Sala(1,250);
        Sala Sala2=new Sala(2,150);
         /* Asignar las salas al multicine*/
        Avenida.setSala(Sala1);
        Avenida.setSala(Sala2);
         /* Alta películas */
        Pelicula Peli1=new Pelicula("1212ASEQ","Greenland: El último refugio","Acción","Ric Roman Waugh" ,
                                    of (2020,8,07) ,120,"M-12");
        Actor Actor1=new Actor("Gerard Butler");
        Actor Actor2=new Actor("Morena Baccarin");
        Peli1.setActor(Actor1);
        Peli1.setActor(Actor2);
        Pelicula Peli2=new Pelicula("0232ASPY","Pinocho","Fantasía","Matteo Garrone",of (2015,12,04) ,
                                    125,"M-7");
        Actor Actor3=new Actor("Roberto Benigni");
        Actor Actor4=new Actor("Gigi Proietti");
        Peli2.setActor(Actor3);
        Peli2.setActor(Actor4);
             /* Asignación de películas a salas */
        Pelicula Copia1Peli1=null;
        try{
            Sala1.proyectar(Peli1, LocalTime.of(16,30));
            Sala1.proyectar(Peli2, LocalTime.of(19,00));
//          Sala1.proyectar(Peli2, LocalTime.of(15,00)); //Simular aquí HoraProyeccionException
            Copia1Peli1=Peli1.clone();
            Sala2.proyectar(Copia1Peli1, LocalTime.of(18,30));
        }catch (HoraProyeccionException HPE){
            System.out.println(HPE);
        }
        System.out.println("CARTELERA");
        System.out.println(Avenida.cartelera());
    /* Prueba de clonación */
    Actor Actor10=new Actor("King Bach");
    Copia1Peli1.setActor(Actor10);
    System.out.println("Copia1Peli1");
    System.out.println(Copia1Peli1);
    System.out.println("Peli1");
    System.out.println(Peli1);
     }
}
```

Se pide:

1.- Hacer el **diseño de las clases** para que sea compatible con el fichero de prueba aportado.
[3 puntos]
2.- Asignación de las películas a las salas tal como se indica.
Control de la excepción *HoraProyeccionException*.
[3 puntos]
Clonación correcta de Películas.
[2 puntos]
3.- Sacar la **cartelera**. Utilizando la interfaz Iterator.
[2 puntos]

Ejemplo salida:

CARTELERA
Multicines Avenida Albacete 06-10-2020
Sala : 1
Título : Greenland: El último refugio
Género : Acción
Director : Ric Roman Waugh
Actores : Actor{nombre=Gerard Butler, nacionalidad=null, fechaNac=null} Actor{nombre=Morena Baccarin, nacionalidad=null, fechaNac=null}
Duración : 120
Calificación : M-12
Hora proyección : 16:30
Título : Pinocho
Género : Fantasía
Director : Matteo Garrone
Actores : Actor{nombre=Roberto Benigni, nacionalidad=null, fechaNac=null} Actor{nombre=Gigi Proietti, nacionalidad=null, fechaNac=null}
Duración : 125
Calificación : M-7
Hora proyección : 19:00
Sala : 2
Título : Greenland: El último refugio
Género : Acción
Director : Ric Roman Waugh
Actores : Actor{nombre=Gerard Butler, nacionalidad=null, fechaNac=null} Actor{nombre=Morena Baccarin, nacionalidad=null, fechaNac=null}
Duración : 120
Calificación : M-12
Hora proyección : 18:30

Proyecto 8.- Cuentas bancarias.

Enunciado

Se pretende desarrollar una aplicación en Java que permita gestionar los distintos tipos de cuentas bancarias que puede tener un cliente en una determinada entidad financiera.

- Una **entidad bancaria** tiene:
  Identificador de empresa (NIF empresa), nombre y un domicilio social.
  Como es evidente, dispone de una relación de clientes que tienen, a su vez, una o más cuentas registradas.
- Un **cliente** está identificado por:
  NIF, Nombre y apellidos, domicilio y teléfono de contacto.
  Deberá tener, al menos, una cuenta en el banco.
- **Entidad domiciliada**:
  Es cualquier empresa o sociedad que puede cargar un recibo mensual por sus servicios (agua, luz, teléfono, etc.)
  Se identifican por NIF empresa y razón social (nombre).

Las **cuentas bancarias** pueden ser de dos tipos y sus características se describen a continuación:

- **Cuenta de ahorro.**
  - Las cuentas de ahorro son remuneradas y tienen un determinado tipo de interés anual.
- **Cuenta corriente.**
  - Las cuentas corrientes no son remuneradas, pero tienen asociada una lista de entidades autorizadas para cobrar recibos domiciliados en la cuenta. Dichas entidades se identifican por su NIF empresa y nombre o razón social (entidad domiciliada).
    Dentro de las cuentas corrientes podemos encontrar a su vez otros dos tipos:
    - Las **cuentas corrientes personales**, que tienen una comisión de mantenimiento (una cantidad fija anual).
    - Las **cuentas corrientes de empresa,** que no tienen comisión de mantenimiento. Como es lógico, para identificar a la empresa tienen un NIF empresa y nombre o razón social. El representante legal debe ser un cliente.
      Además, estas cuentas permiten tener una cierta cantidad de descubierto (máximo dinero permitido que le puedes deber al banco) y por tanto, un tipo de interés por descubierto (el porcentaje es referido al día). También dispone de una comisión fija por cada vez que haya un descubierto.
      Es el único tipo de cuenta que permite tener descubiertos.

Cuando se vaya a abrir una nueva cuenta bancaria, el usuario de la aplicación (empleado del banco) tendrá que solicitar al cliente:

- Datos personales: NIF, nombre y apellidos, domicilio, teléfono contacto, etc.
- Cuando se trate de una cuenta de empresa, el NIF y los datos personales corresponderán al representante legal. Además se deberá aportar el nombre o razón social y NIF de la empresa.
- Cuenta o cuentas que desea abrir: cuenta de ahorro, cuenta corriente personal o cuenta corriente de empresa. Puede tener más de una.

Nota: el NIF de una empresa está formado por 9 caracteres (comienza con una letra que determina el tipo de actividad y 8 números después).

Además de esa información, el usuario de la aplicación deberá también incluir:

- Número de cuenta (CCC) de las nuevas cuentas y saldo inicial.
- Tipo de interés de remuneración, si se trata de una cuenta de ahorro.
- Tipo de interés de mantenimiento, si es una cuenta corriente personal.
- Máximo descubierto permitido, si se trata de una cuenta corriente de empresa.
- Tipo de interés por descubierto, en el caso de una cuenta corriente de empresa.
- Comisión fija por cada descubierto, también para el caso de una cuenta corriente de empresa.

Se pide:

1. Diseño de clases que cumpla con los requisitos. Debe respetar también escrupulosamente todas las normas de un diseño orientado a objetos
   Como es preceptivo, nuestro cliente [main ( )] no podrá ni deberá utilizar estructuras de datos en la simulación.
   [2,5]

2. Crear un banco con nombre y domicilio social.
   [0,1]

3. Crear un cliente con sus datos personales.
   [0,1]

4. Crear una cuenta de ahorro para el cliente anterior. Asignársela.
   [0,2]

5. Crear una cuenta corriente personal.
   [0,1]

6. Crear dos entidades para domiciliar recibos.
   [0,1]

7. Domiciliar estas entidades en la cuenta corriente personal creada anteriormente.
   [0,3]

8. Asignar esta cuenta corriente personal al cliente creado.
   [0,2]

9. Crear una cuenta corriente para empresa.
   [0,1]

10. Crear una entidad para domiciliar los pagos en esta cuenta.
    [0,1]

11. Domiciliar la anterior entidad en la cuenta corriente de empresa.
    [0,2]

12. Asignar esta cuenta al cliente creado.
    [0,1]

13. Describir todos los datos del cliente y los de sus cuentas asociadas.
    No se precisa un formato determinado.
    [0,3]

14. Registrar el cliente en el banco.
    [0,1]

15. Simular el cargo de un recibo que llega al banco.
    Se requiere el número de cuenta donde se va a cargar el importe, el NIF de empresa de la entidad que lo carga y dicho importe. Actualizar saldo.
    Lanzar una excepción *CuentaBancariaException* en caso de que el número de cuenta no corresponda con alguna registrada o no sea una cuenta corriente. Distinguir ambos casos.
    Lanzar excepción también si el NIF de la empresa no se encuentra dentro de las domiciliaciones aceptadas.
    [2,5]

16. Comprobar que el saldo ha cambiado. Obtener el saldo por el número de cuenta.
    [1,5]

17. Clonar la cuenta de empresa creada anteriormente. Comprobar su total independencia de la original.
    Clonar la cuenta y cambiar el nombre de la empresa original. Comprobar que en la cuenta clonada no se ha modificado.
    [1,5]