

Programación 1

sanchezih@gmail.com

<https://github.com/sanchezih/ort-p1>

https://www.youtube.com/playlist?list=PLCXOBUYA4iuYtB7bzFtw3_8HL29mNQpLG

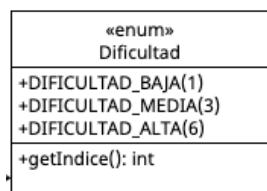
<https://drive.google.com/drive/folders/1DjunXmoyAm1DRtUkLGvmjyq9ojvNBmlW>

Consejos

- -una clase abstracta puede tener constructor y métodos desarrollados
- -una interfaz no puede tener métodos abstractos
- -se corrige la interpretación del problema y su resolución
- -saber llamar a la clase padre en métodos y atributos desde el hijo es IMPORTANTE
- -los enums tienen métodos muy útiles como: ordinal y values
- -método para agregar algo en la primera posición libre en un array se hace con while
- -no explotar los métodos de las interfaces como POP, PEEK, PUSH etc
- -cola dentro de una lista ordenada
- -método abstracto en la clase padre que vía polimorfismo devuelva lo que deba de cada tipo de clase hija, el cual debería ser utilizado desde el método único en la clase padre
- -Si hubieras implementado un método concreto en la clase padre que llame a todos los métodos de las clases hijas, podrías haber reutilizado la llamada al método que te da lo que quieras. En ese caso solo tendrías que haber hecho return lo que quieras. → pabre.metodo()
- -Deberías haber hecho new int[CANT_OBRAS][Dificultad.values().length].
- -los enum tienen un número, si pones Enum.values().ordinal te tira el orden de cada enum, 0...1...2
- -matriz de enteros: matriz [0] [0]++ → le estas sumando uno a la posición [0] [0] → [0] [1] → [1] [1]
- -Si uso un get o set lo debo diagramar en el UML
- -cuando el contenido de una matriz es un objeto en algún momento debo instanciarlo y cuando es un primitivo toma valor por defecto
- -todos los métodos que usó en el NS debo ponerlos en el UML y los debo explotar solo si los uso
- -para acceder a un atributo del padre tengo que hacer un GER SIEMPRE
- Una constante siempre debe ser de un tipo de algo
- si tengo una lista ordenada debo explotar los métodos compare y compareByKey
- El compareByKey lo uso la lista cuando hace búsqueda

Tener en cuenta

- Modificadores de acceso: ok.
- Uso de clases abstractas: ok.
- Uso de interfaces: ok.
- Uso de constantes: ok.
- Uso de enum:



- UML

- El método estimar debería haber estado 1 sola vez, en la superclase. De esta manera reutilizas el pedido de dificultad y con polimorfismo pedis la duración de cada tipo de obra.
- Relaciones de uso

UML	NS+
<p>1. Modificadores de acceso: ok.</p> <p>2. Uso de clases abstractas: Empleado debería haberse modelado como Abstract Class.</p> <p>3. Uso de interfaces: ok.</p> <p>4. Uso de enum: Correcto el enum NivelEducativo. No es necesario modelar el enum Turno, ya que el mismo estaba representado por las filas de la matriz. Lo mismo para día laborable.</p> <p>5. UML: Mal modelada la matriz de cargos que debería tener un profesor. Deberías haber hecho -cargos[DIAS_SEMANA][TURNOS] directamente.</p>	<p>El método empleadosEnFalta debía:</p> <ol style="list-style-type: none"> 1. Recorrer la cola de empleados (Profesores y Auxiliares) 2. Por cada empleado, si tiene alguna irregularidad, lo saco de la cola y lo pongo en la lista ordenada. Caso contrario, lo vuelvo a meter en la cola. <p>irregularidad de Auxiliar es correcto. Se entiende la lógica que intentas seguir en irregularidad de Profesor, pero lo podrías haber simplificado haciendo en 1 sola línea con algo como return !cargosValidosSegunNivel() !cumpleCantCargos(NOCTURNO, MAX_CARGOS_NOCTURNOS); En un método recorres la matriz comparando los niveles del profesor contra el del cargo, siempre utilizando el método ordinal. En el otro método recorrieras solo la fila 2 de la matriz y cuentas la cantidad de ocurrencias.</p> <p>No implementas compare ni compareByKey</p>

Comentario:

UML	NS+
<p>1. Modificadores de acceso: ok.</p> <p>2. Uso de clases abstractas: Empleado debería haberse modelado como Abstract Class.</p> <p>3. Uso de interfaces: ok.</p> <p>4. Uso de enum: Correcto el enum NivelEducativo. No es necesario modelar el enum Turno, ya que el mismo estaba representado por las filas de la matriz. Lo mismo para dia laborable.</p> <p>5. UML: Mal modelada la matriz de cargos que debería tener un profesor. Deberías haber hecho -cargos[DIAS_SEMANA][TURNOS] directamente.</p>	<p>El metodo empleadosEnFalta debia:</p> <ol style="list-style-type: none"> 1. Recorrer la cola de empleados (Profesores y Auxiliares) 2. Por cada empleado, si tiene alguna irregularidad, lo saco de la cola y lo pongo en la lista ordenada. Caso contrario, lo vuelvo a meter en la cola. <p>irregularidad de Auxiliar es correcto. Se entiende la logica que intentas seguir en irregularidad de Profesor, pero lo podrias haber simplificado haciendo en 1 sola linea con algo como return !cargosValidosSegunNivel() !cumpleCantCargos(NOCTURNO, MAX_CARGOS_NOCTURNOS); En un metodo recorres la matriz comparando los niveles del profesor contra el del cargo, siempre utilizando el metodo ordinal. En el otro metodo recorrieras solo la fila 2 de la matriz y contas la cant de ocurrencias.</p> <p>No implementas compare ni compareByKey</p>

El paradigma de programación orientada a objetos
Se compone de 4 elementos

- | | |
|---|--|
| <ul style="list-style-type: none"> ● Clases ● Propiedades ● Métodos ● Objetos | <ul style="list-style-type: none"> ● Encapsulamiento ● Abstracción ● Herencia ● Polimorfismo |
|---|--|

- Analizar
 - problemas
 - Observación
 - Entendimiento
 - Lectura
- Plasmar
 - Análisis del problema
 - Diagramas
- Programar
 - Diagramas
- Lenguaje de programación

Final

Static

A aquellos miembros que sean estáticos DE LA CLASE se los debe subrayar para diferenciarlos con los otros miembros de la instancia

Enum

Los enumerations pueden tener atributos, métodos y constructores, como se muestra:

```
public enum Day {
    MONDAY("Lunes"), SUNDAY("Domingo");
    private String spanish; -> atributo
    private Day(String s) { -> constructor
        spanish = s; }
    private String getSpanish() {
        return spanish;
    } -> Y para utilizarlo lo podemos hacer así:
    } -> System.out.println(Day.MONDAY);
```

Diagramas de clases

Estos son los niveles de visibilidad que puedes tener:

- private : Solo pueden acceder los objetos de la misma clase
 + public : Acceden todos los objetos
 # protected : Acceden los objetos de la misma clase, subclases o clases del mismo paquete
 ~ default :

<<Enum>>
<<Interfaz>>

Clases

Es el modelo sobre el cual se construirá nuestro objeto

Se representan así:

Clase
+ attribute1:type = defaultValue
+ attribute2:type
- attribute3:type
+ operation1(params):returnType
- operation2(params)
- operation3()

En la parte superior se colocan los atributos y debajo las operaciones de la clase el primer carácter con el que empiezan es un símbolo. Este denotará la visibilidad del atributo o método

Objetos

cuando tenga un problema lo primero es identificar objetos. son aquellos que tienen propiedades y comportamientos así los distinguimos en todo el escenario hay objetos físicos o conceptuales

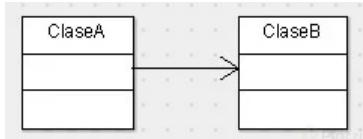
- Entender el contexto de nuestros objetos - Intentar resolver el problema paso a paso

-El encapsulamiento es una propiedad que tienen los objetos dentro de este paradigma. Esta propiedad dice que los **elementos internos del objeto son propios de él y de nadie más**. Por esto, **nadie** desde afuera las puede **alterar**, o siquiera **observar**, sin el "permiso" de su "dueño".

Asociación



Como su nombre lo dice, notarás que cada vez que esté referenciada este tipo de flecha significa que ese elemento contiene al otro en su definición. La flecha apuntará hacia la dependencia.



Con esto vemos que la ClaseA está asociada y depende de la ClaseB.

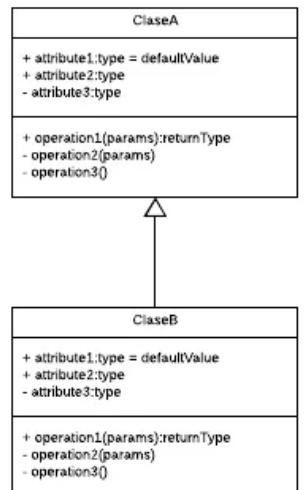
Herencia



Siempre que veamos este tipo de flecha se estará expresando la herencia.

La dirección de la flecha irá desde el hijo hasta el padre.

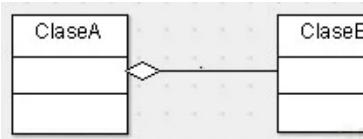
Con esto vemos que la ClaseB hereda de la ClaseA



Agregación

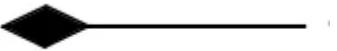


Un elemento dependerá de muchos otros. Aquí tomamos como referencia la multiplicidad del elemento. Lo que comúnmente es Relación uno a muchos.

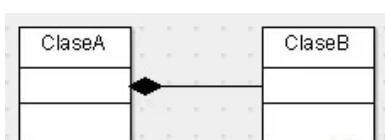


Con esto decimos que la ClaseA contiene varios elementos de la ClaseB. Estos últimos son comúnmente representados con listas o colecciones de datos.

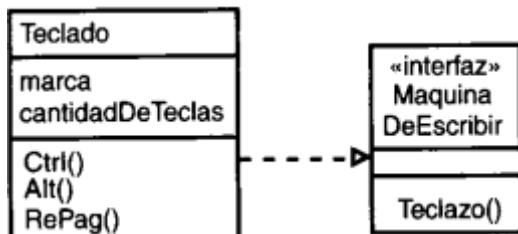
Composición



su relación es totalmente compenetrada de tal modo que conceptualmente una de estas clases no podría vivir si no existiera la otra.



Con esto terminamos nuestro primer módulo. Vamos al siguiente para entender cómo podemos hacer un análisis y utilizar estos elementos para construir nuestro diagrama de clases de Uber.



Modularidad

significa: subdividir un sistema en partes más pequeñas de nombre "módulos" pueden funcionar de forma independiente y nos ayudarán a componer un sistema.

Don't repeat yourself

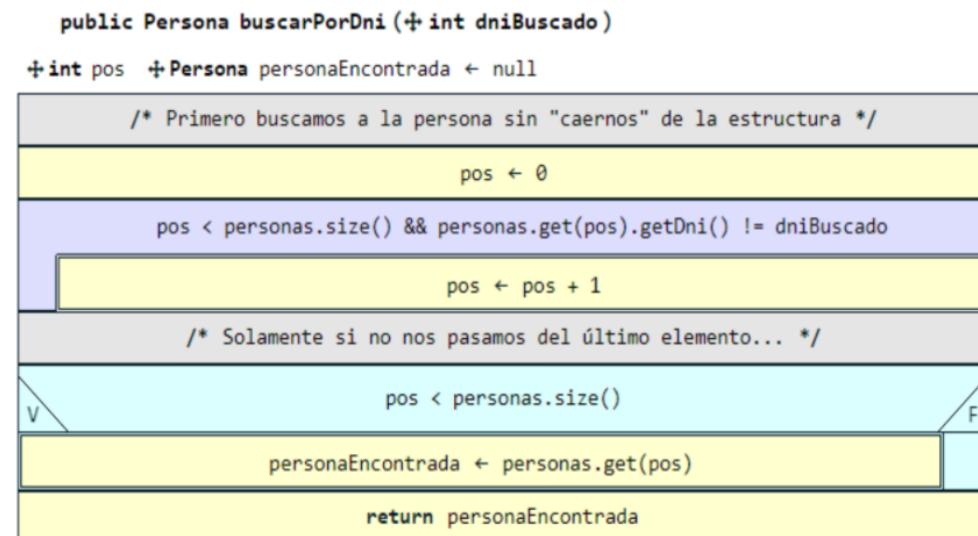
Cohesión y acoplamiento:

Cohesión: grado de relación que tiene el objeto con sus responsabilidades. En la medida que el objeto implemente exclusivamente las responsabilidades que le corresponde mayor será el nivel de cohesión que tendrá y por ende más independiente será del resto.

Acoplamiento: es fundamental que la implementación del objeto está oculta dentro del objeto y que no sea visible a los componentes externos.

–Los modelos que construyamos intentarán estar conformados por objetos altamente cohesivos y con el menor acoplamiento entre sí.

- El sistema no debe tener un estado compartido entre distintos objetos.
 - Si se divide una clase cohesiva las clases resultantes estarán fuertemente acopladas.
 - Cualquier objeto debe ser reemplazable por otro con la misma interfaz.
 - Si se juntan clases con bajo acoplamiento la clase resultante tendrá baja cohesión.
 - Las asociaciones bidireccionales aumentan el acoplamiento.
 - Crear conexiones estrechas. Número de mensajes que un objeto envía a otro y la cantidad de parámetros deben ser mínimos.
 - **Acoplamiento de contenido**: Se da cuando un objeto accede a los atributos del otro sin pasar por su interfaz.
 - **Acoplamiento de control**: Un objeto le envía un mensaje a otro conteniendo parámetros de control.
 - **Acoplamiento estampado**: Un objeto le envía un mensaje a otro conteniendo una estructura de datos.
 - **Acoplamiento por datos**: Un objeto le envía un mensaje a otro conteniendo datos.
- Cuantos más argumentos tiene un método mayor es el acoplamiento. Evitar el uso de datos "excursionistas", es decir, datos que viajan de un objeto a otro sin ser utilizados hasta que alcanzan su destino.
- **Acoplamiento de subclases**: Una clase cliente tiene una referencia a la subclase en lugar de tener la referencia a la superclase. Aumenta la dependencia.



Abstracto

(abstract)

Método abstracto: no lleva cuerpo (llaves) - Declaración: public abstract void importeSalarial();

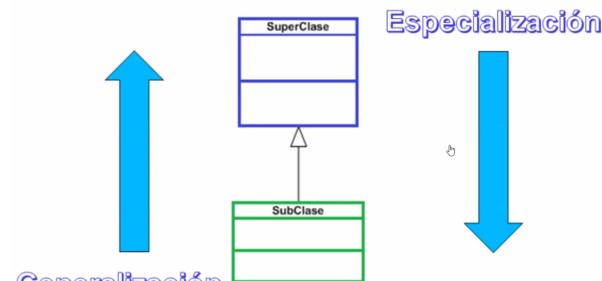
abstract public/private/protected TipoRetorno/void (parámetros...);

Sólo puede existir dentro de una clase abstracta... Tendrán que ser sobreEscritos en los subclases (hijos)

Herencia

(extends)

La herencia siempre involucra dos o más clases, la clase **de la cual**



se hereda la llamaremos **SuperClase** y la o las que heredan las llamaremos **Sub Clases**

-**SuperClase:** Clase cuyas características se heredan.

-**SubClase:** Clase que hereda la otra clase. puede agregar sus propios campos y métodos además de los que posee la superclase.

-**Reutilización:** Cuando queremos crear una clase nueva y ya hay una clase que incluye parte del código, podemos derivar nuestra nueva clase de la clase existente.

Siempre preguntarme si “es un” si es ‘si’ puedo hacer herencia

-**herencia simple:** Una clase **sólo puede heredar de una clase base** y de ninguna otra.

-**herencia múltiple:** Una clase **puede heredar las características de varias clases base**, puede tener varios padres. (Java NO)

-**constructores:** Los constructores no son miembros, por lo que **no son heredados por subclases**, pero el constructor de la superclase puede invocarse desde la subclase.

-**herencia de miembros privados:** una subclase no hereda los miembros privados de su superclase, si esta tiene métodos públicos o protegidos (como getters y setters) para acceder a sus campos privados, estos podrán ser utilizados por la subclase.

¿Qué se puede hacer en una Subclase?

1. Los campos heredados se pueden usar directamente, al igual que cualquier otro campo.
2. Los métodos heredados se pueden usar directamente.
3. Podemos escribir un nuevo método en la subclase que tenga la misma firma que el de la superclase (Overriding).
4. Cualquier cambio que se realice en la superclase tendrá efectos en las clases hijas

- La palabra clave **super** se usa para referir objetos de clase padre se usa Con variables, métodos y constructores
- El objetivo de **instanceof** es conocer si un objeto es de un tipo determinado —> empleado instanceof persona

-implementación:

```
class SubClase :  
    public void SubClase (+ int valor )  
        super(valor)  
        this.v2 ← -1
```

-En UML su nombre se indica en letra **cursiva**

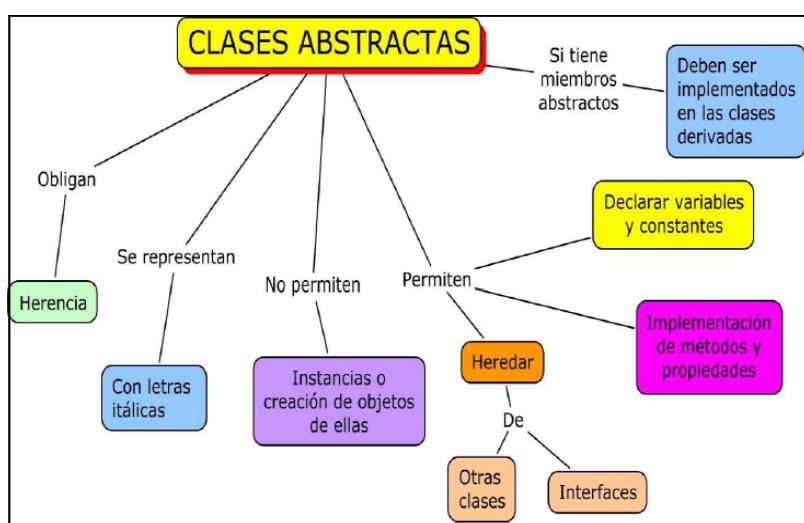
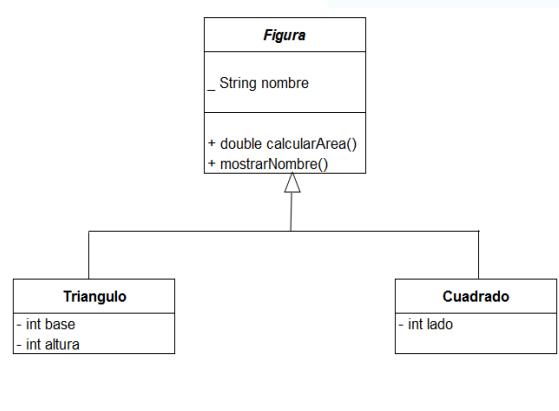
-**public abstract class nombreClase{...}**

-En este modelo vemos que todo Auto tiene (has-a) un Motor y que además es un (is-a) tipo específico de Vehículo.

Asociación

→ has-a (tiene un/a)

Generalización



Interfaces

(implements)

Las Interfaces son un tipo de clase con solo constantes y definiciones de métodos, son de gran ayuda para definir los comportamientos que son redundantes y queremos reutilizar más de una clase, incluso cuando tenemos muchas clases y no todas pertenecen a la misma “familia”.

Las interfaces establecen la forma de las clases que la implementan, así como sus nombres de métodos, listas de argumentos y listas de retorno, pero NO sus bloques de código, eso es responsabilidad de cada clase.

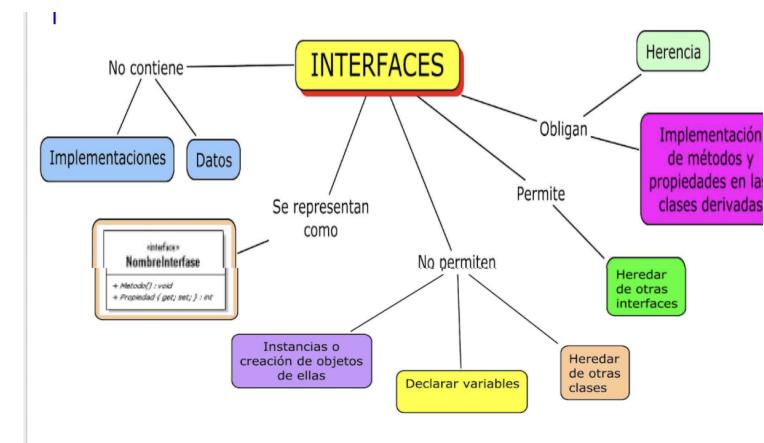
DECLARACIÓN: `public interface NombreInterface {...}`

1. Todo método es abstracto y público aunque no esté declarado
2. Las interfaces no tienen ningún constructor.
3. Si contiene atributos, todos deben ser public static final,
4. no se puede crear un objeto del tipo definido por una interface.
5. Si una clase implementa una interface: implemente los métodos de la interface sobreescribiéndolos (clase concreta).
6. Que no implemente los métodos de la interface: obligatoriamente será una clase abstracta

¿Cuándo deberíamos crear una interface?

1. Si necesitamos que una clase herede de más de una superclase
2. Si en un escenario es igual de viable definir una clase como interface que como clase abstracta

La clase SuperClase no se puede instanciar porque es abstracta, por lo tanto no es necesario implementar en ella los métodos de la interface; de ser este el caso, será obligatorio implementarlos en sus clases hijas



operador ternario:

```
return si_esto_da_true ? devuelvo_esto : devuelvo_esto_otro
```



`void addElemento(objet obh) : añade un elemento al final del vector`

`object elementoAt(ind indice) : devuelve el lemento de la poosicion del vectos indicada por el indice`

void insertElementAt (Objetc ob, int indice) : insera un elemento en la posicion indicada

boolean removeElement(Objet obj) : elimina el elemento indicado del vector, devolviendo true si dicho elemento estaba contenido en el vecto y false en caso contrario

void removeElementAt(int indeice) : elimina el lemento de la posicon indicada en el indice

void setElementAt(Objt ob, int indice) : sobreescibe el elmentos de l aposcion indicada con e objeto espeoficifcafo

int size() " devuelave el numerode elementos del vector,

boolean add(ojt o) : añade un elemnto (objeto) a la colección. nos devuleve true si tras añadir el elemtno la colección ha cambiado, es decir el elemento se ha añadiro correctamento, o false en claro contrario

void clear() : elinia todos los elemtnso de la colección

voolean contains(objeto o) : indica si la colección contienen el lementos (objeto) indicado

boolean isEmpty() : indica si la colección esta vacia (no tiene ninfun elemento)

boolean remove(objeto o) : eliminan un determinado (objeti) de la colección, devolviendo treu si dichi eementos estaba contenido en la colección y false en caso contrario

Los principios SOLID

Los 5 principios SOLID de diseño de aplicaciones de software son:

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)



4. Principio de Segregación de la Interfaz

"Make fine grained interfaces that are client specific."

En el cuarto principio de SOLID, el tío Bob sugiere: “**Haz interfaces que sean específicas para un tipo de cliente**”, es decir, para una finalidad concreta.



En este sentido, según el **Interface Segregation Principle (ISP)**, es preferible contar con muchas interfaces que definan pocos métodos que tener una interface forzada a implementar muchos métodos a los que no dará uso.

Array - Vector - Matriz - arreglo

Son estructuras de datos estáticas que permiten guardar elementos del mismo tipo de forma continua. permite acceso a los elementos de forma aleatoria a través de un índice numérico entero que comienza en 0.

length contiene el tamaño de la matriz

Tipo de dato ‘arra’

Creación de un array

```

int valor;

int[] valores;

boolean[] banderas;
char[] letras;
String[] palabras;
NaveEspacial[] naves;

```

```

int[] valores = new int[5]; // Una variable que apunta a un array de 5 enteros

boolean[] banderas = new boolean[3]; // Una variable que apunta a un array de 3 booleanos

int tam = 7;
char[] letras = new char[tam]; // Una variable que apunta a un array de 7 caracteres

int n = (new Scanner(System.in)).nextInt(); // Lee un entero desde La consola
String[] palabras = new String[n]; // Una variable que apunta a un array de n cadenas

```

¿Cuándo usar arrays?

"Un torneo consta de 20 equipos..."



"Un auto tiene 4 ruedas..."



"Una guardería almacena como máximo 10 embarcaciones..."



Recorrer todos los elementos

```
char[] letras = {'A', 'P', 'R', 'E', 'N', 'D', 'I'}
```

```
int i = 0, letras.length - 1, 1
```

S

letras[i]

Usando ciclo for

```

char[] letras = {'A', 'P', 'R', 'E', 'N', 'D', 'I'};
for (int i = 0; i < letras.length; i++) {
    System.out.println(letras[i]);
}

```

```
char[] letras ← {'A', 'P', 'R', 'E', 'N', 'D', 'I'}
```

```
char letra : letras
```

S

letra

Usando ciclo foreach

```
char[] letras = {'A', 'P', 'R', 'E', 'N', 'D', 'I'};  
for (char letra: letras) {  
    System.out.println(letra);  
}
```

MATRIZ

En **matemática**, una matriz es un conjunto de números distribuidos en filas y columnas.

En **informática**, podemos representar una matriz de números, o cualquier otro tipo de dato, como un array de dos dimensiones.

- Array de arrays
 - Array bidimensional
 - Matriz (a secas)
- }
- Sinónimos**

Recorrer todos los elementos

```
char[][] matLetras ← {{'D', 'E', 'B', 'O'}, {'U', 'S', 'A', 'R'}, {'G', 'U', 'I', 'A'}}
```

```
int i ← 0 , matLetras.length - 1 , 1    int j ← 0 , matLetras[i].length - 1 , 1    S    matLetras[i][j]
```

Usando ciclo for

```
char[][] matLetras = {{'D', 'E', 'B', 'O'}, {'U', 'S', 'A', 'R'}, {'G', 'U', 'I', 'A'}};  
for (int i = 0; i < matLetras.length; i++) {  
    for (int j = 0; j < matLetras[i].length; j++) {  
        System.out.println(matLetras[i][j]);  
    }  
}
```

Búsqueda en matriz

```
public boolean existeLetra (char letra)
boolean existe ← false int i int j
char[][] matLetras ← {{'D','E','B','O'}, {'U','S','A','R'}, {'G','U','I','A'}}
    i ← 0
        i < matLetras.length && !existe
            j ← 0
                j < matLetras[i].length && !existe
                    matLetras[i][j] == letra
                        V F
                        existe ← true
                        j++
                    i++
                return existe
            }
```

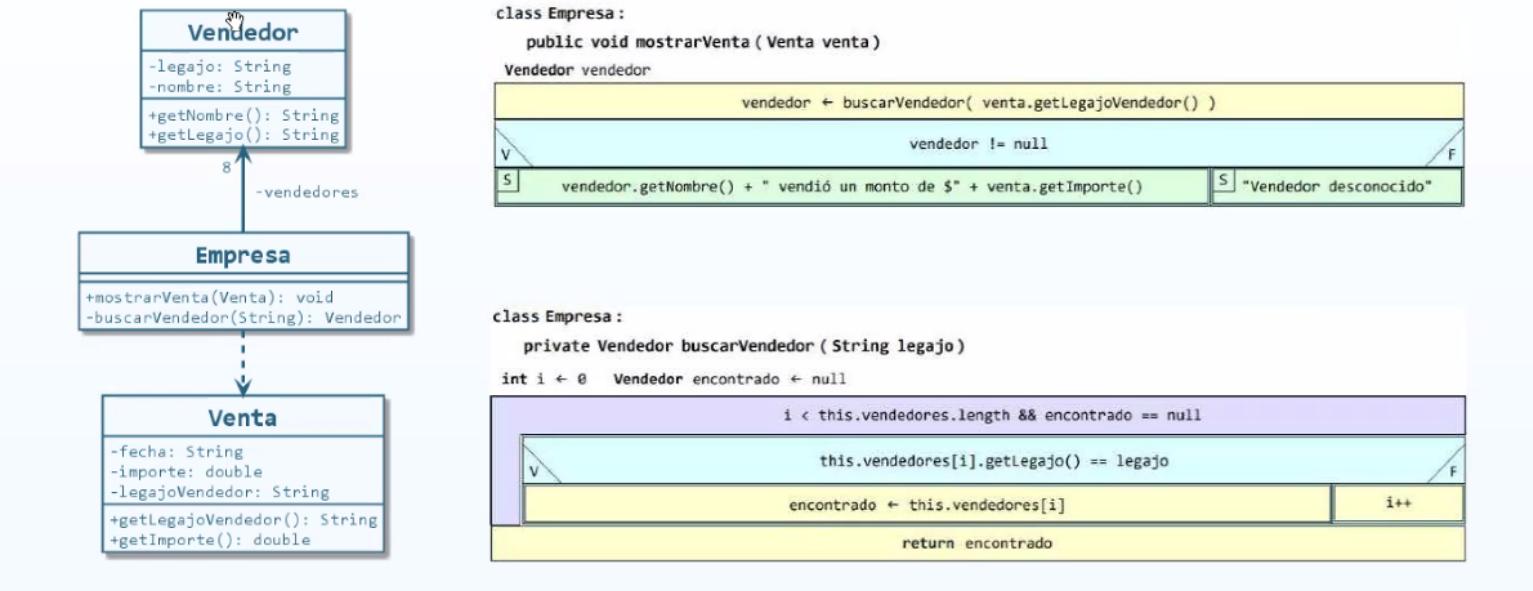
```
public boolean existeLetra (char letra) {
    char[][] matLetras = {{'D','E','B','O'},
                           {'U','S','A','R'},
                           {'G','U','I','A'}};
    boolean existe = false; int i, j;
    i = 0;
    while(i < matLetras.length && !existe) {
        j = 0;
        while(j < matLetras[i].length && !existe) {
            if(matLetras[i][j] == letra) existe = true;
            j++;
        }
        i++;
    }
    return existe;
}
```

```
private Vendedor buscarVendedor(String legajo) {
    int i = 0;
    Vendedor encontrado = null;

    while (i < this.vendedores.length && encontrado == null) {
        if (this.vendedores[i].getLegajo() == legajo) {
            encontrado = this.vendedores[i];
        } else {
            i++;
        }
    }
    return encontrado;
}
```

Posicionamiento indirecto

Cuando no se puede calcular la posición de determinado valor en un array, debe buscarse en la estructura.



```

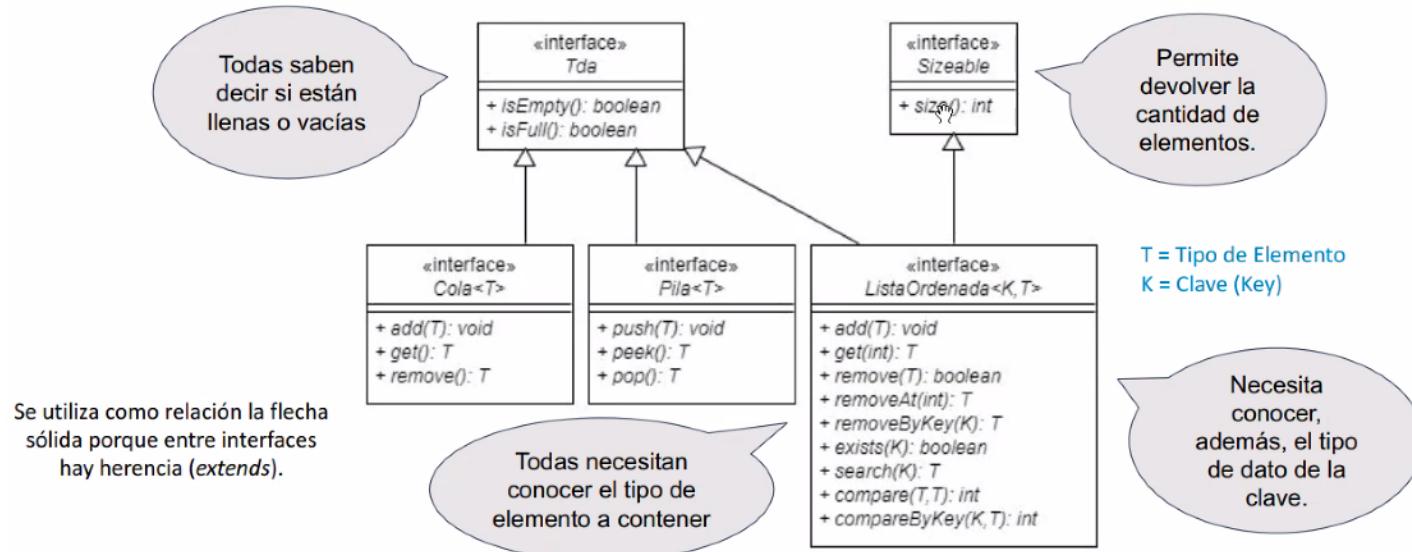
private void activarAlgunCursor(Componente comp) {
    if (!this.hayCursorActivo && comp instanceof CampoDeTexto) {
        ((CampoDeTexto) comp).activarCursor(); // downcasting
        this.hayCursorActivo = true;
    }
}

public VentanaDeError(String textoDelError, String color, boolean habilitado, Tamanio tamano, Posicion posicion,
                      boolean hayCursorActivo, Estado estado) {
    super(color, habilitado, tamano, posicion, hayCursorActivo, estado);

    this.etiquetaError = new Etiqueta(null, null, true, null, null, null);
    this.botonAceptar = new Boton(null, null, true, new Tamanio(640, 480), new Posicion(100, 100));
    etiquetaError.setTexto(textoDelError);
    botonAceptar.setTexto("Aceptar");
    agregar(etiquetaError);
    agregar(botonAceptar);
}
    
```

TDA'S

TADs que usaremos (declaración de Interfaces en Java usando Generics)



PILA

Pila: Último que entra primero que sale → Sabe apilar elementos **sin importar su tipo**

push: agregar algo a la pila

- Push
 - Crea un nodo con el dato recibido y lo pone delante de los nodos existentes
- Pop
 - Extrae el primer nodo de la cadena, actualiza el primer nodo FIRST y actualiza el tamaño
- Peek
 - Devuelve el primer nodo del atributo FIRST y no actualiza el estado

-si quiero ver los elementos que están en una pila WHILE (la pila siempre se recorre con auxiliar y se debe dejar como estaba)

-RECORRER UNA PILA: si tenemos que sacar un elemento de la pila recorremos la pila y si NO encontramos el elemento buscado agregamos los demás a una pila AUXILIAR y si lo encontramos lo retornamos y después volver a recorrer la pila para dejarlo como estaba

```
+Pila pilaAux = new Pila()
    /* MIENTRAS la pila No este vacia */

    !pilaQueReciboPorParametro_O_pilaDeMiClase.isEmpty()

        /* saco un libro con POP() */

        variableTemporal miVariableAux = pila.pop()

        /* lo apilo o guardo en la pila auxiliar */

        pilaAux.push(miVariableAux)

        /* aca hago el proceso */

    /* tengo que volver a poner la pila como estaba */

    !pilaAux.isEmpty()

        pilaQueReciboPorParametro.push(pilaAux.pop())
```

comentario:

[8:27 p. m., 12/7/2022] Federico Ort: por que al procesar reclamos tenes que recorrer esa pila

[8:27 p. m., 12/7/2022] Federico Ort: y ver si los datos que tiene el reclamo estan bien , o sea si existe la persona en el registro y si coincide con los importes que tiene

[8:28 p. m., 12/7/2022] Federico Ort: si no hacia otra cosa creo

[8:28 p. m., 12/7/2022] Federico Ort: Al procesar estos reclamos se verifica que el ciudadano exista y que el monto reclamado sea distinto al importe del impuesto determinado para dicho ciudadano

[8:29 p. m., 12/7/2022] Federico Ort: En el caso de detectarse una inconsistencia (el monto reclamado sea distinto al monto determinado) el reclamo se derivará a un área especial la cual procesará todos los reclamos en el orden que fueron procesados(Cola)

COLA

Cola: El primero que entra es el primero que sale -> Sabe encolar elementos sin importar su tipo FIFO

- Add
 - El dato se agrega al final de la cola
 - Mantiene los nodos FIRST y LAST
 - Actualiza el tamaño de la cola
- Remove
 - Extraer el primer nodo de la cadena
 - Actualiza el atributo FIRST
 - Decrementa en 1 el tamaño de la cola
- Get
 - Devuelve el nodo referenciado por FIRST sin extraerlo

las colas deben quedar igual a como estaban (no se necesita auxiliar) pero se usa un “centinela”

LISTA ORDENADA

TDAs - Listas Ordenadas

La Lista Ordenada debe *aprender* a ordenar su contenido comparando las *claves* de sus elementos

Para eso utilizará la implementación de los métodos **compare(...)** y **compareByKey(...)** a implementar en cada lista.

▶ `public int compare(T elemento1, T elemento2)`

Se utiliza para comparar dos elementos entre sí. Es utilizado internamente durante la inserción de nuevos elementos.

▶ `public int compareByKey(K clave, T elemento)`

Se utiliza para comparar el valor de una clave con un valor determinado. Es utilizado internamente para la búsqueda por clave de un elemento en el método `search(...)`.

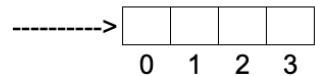
En ambos casos el valor numérico devuelto determina el “peso” de la clave y el posterior ordenamiento de los elementos:

- Si el valor returnedo es 0 (cero) se asume que los elementos son iguales o equivalentes.
- Si el valor devuelto es mayor a cero, significa que la clave del primer elemento es mayor que la clave del segundo (en el `compare`), o que el valor recibido como clave de búsqueda es mayor que la clave del elemento (en el `compareByKey`).
- Si el valor devuelto es menor que cero, significa que la clave del primero es menor a la del segundo (`compare`), o que la clave de búsqueda recibida es mayor a la clave del elemento (`compareByKey`).

Comportamiento de compare

Premisas	El 1er parametro, es el elemento nuevo El 2do parametro, es el elemento que extraigo de la estructura
----------	--

	devuelve 0	devuelve negativo (-)	devuelve positivo (+)
compare	Se asume que los elementos son iguales	El elemento nuevo quedara a la izquierda del que se saco de la estructura para comparar	El elemento nuevo quedara a la derecha del que se saco de la estructura para comparar

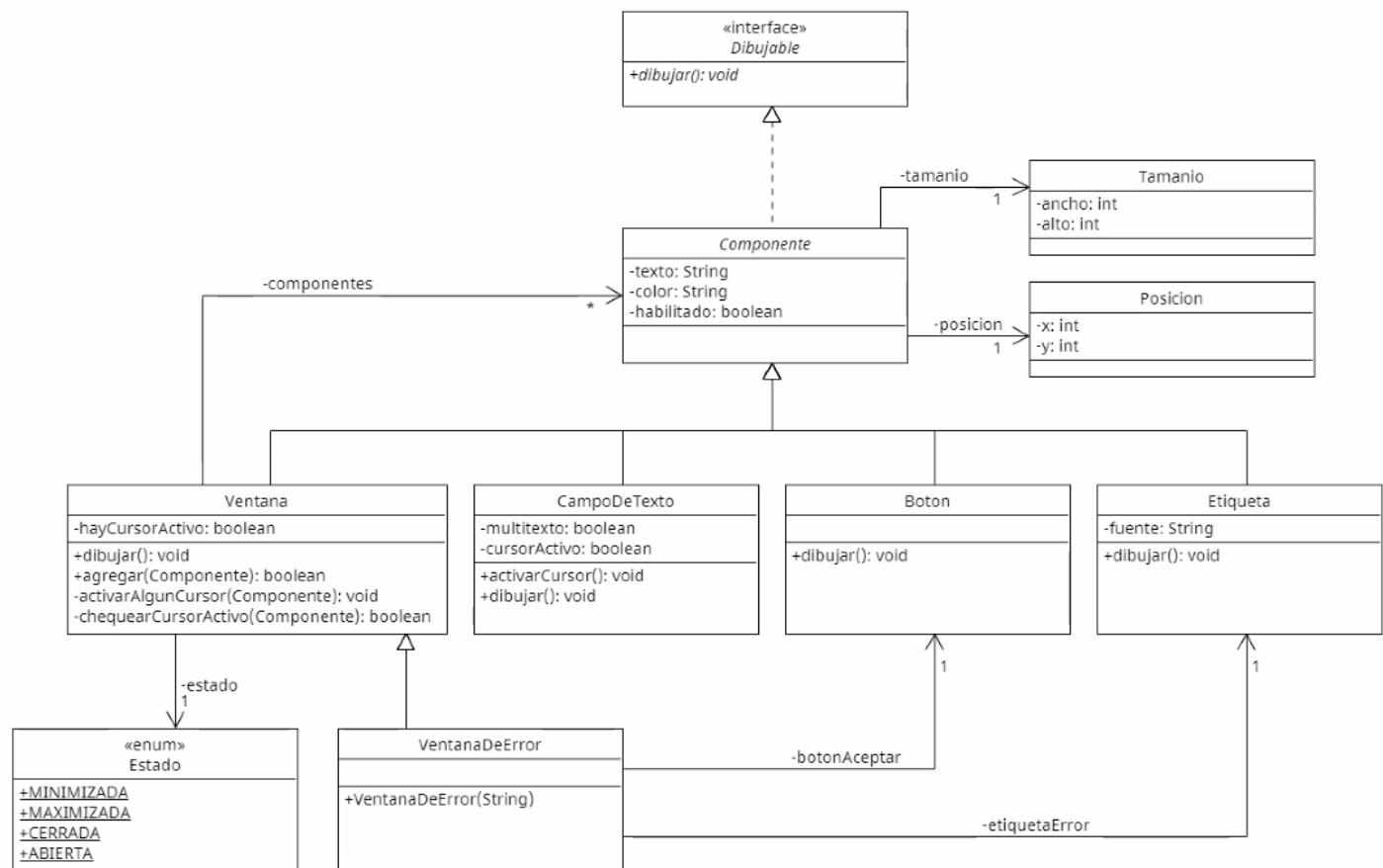
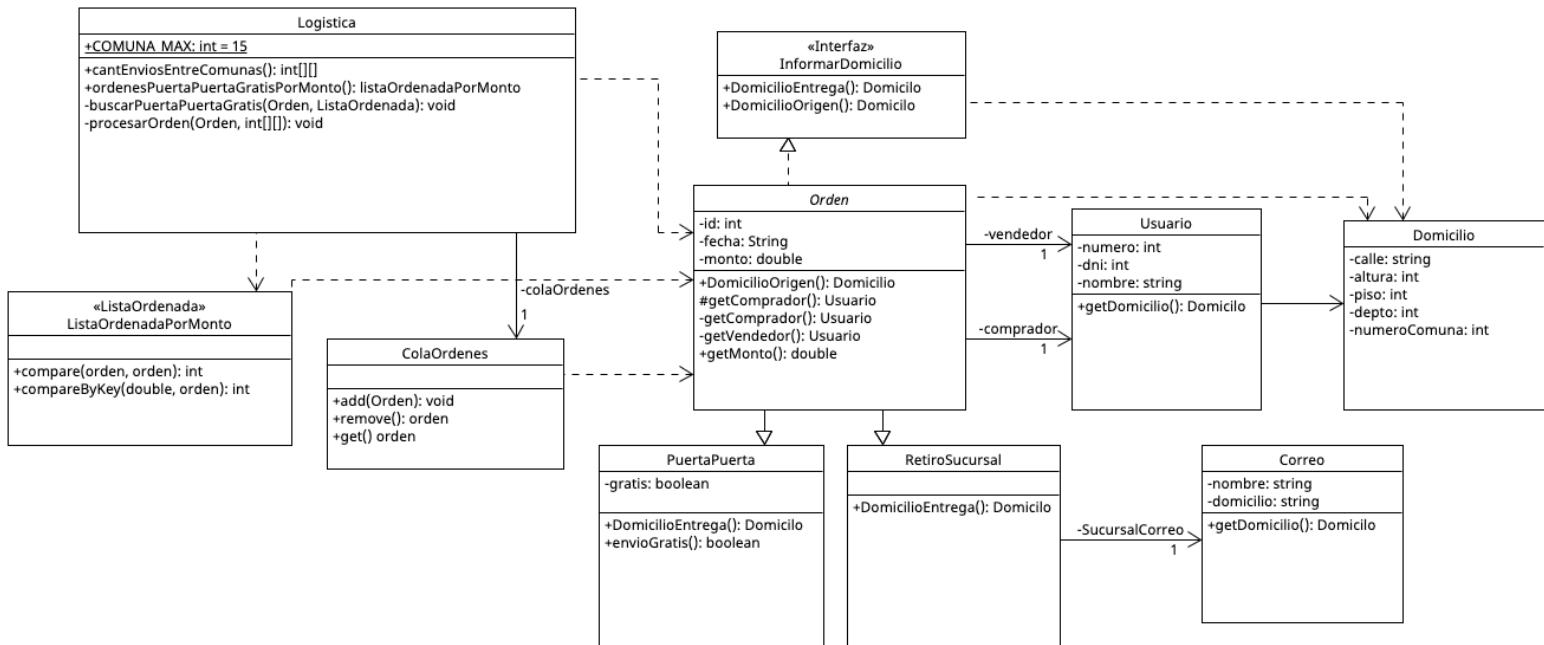


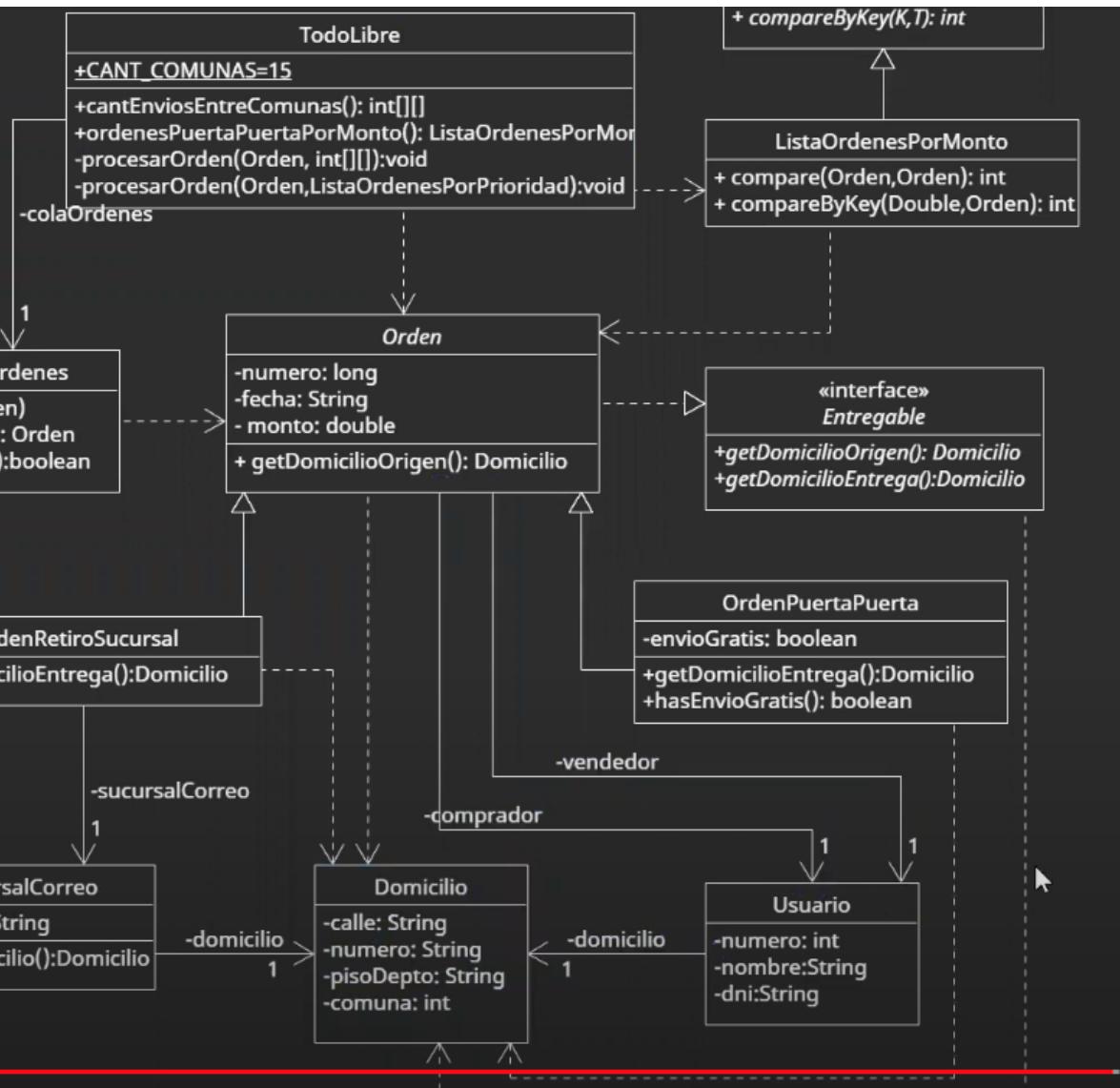
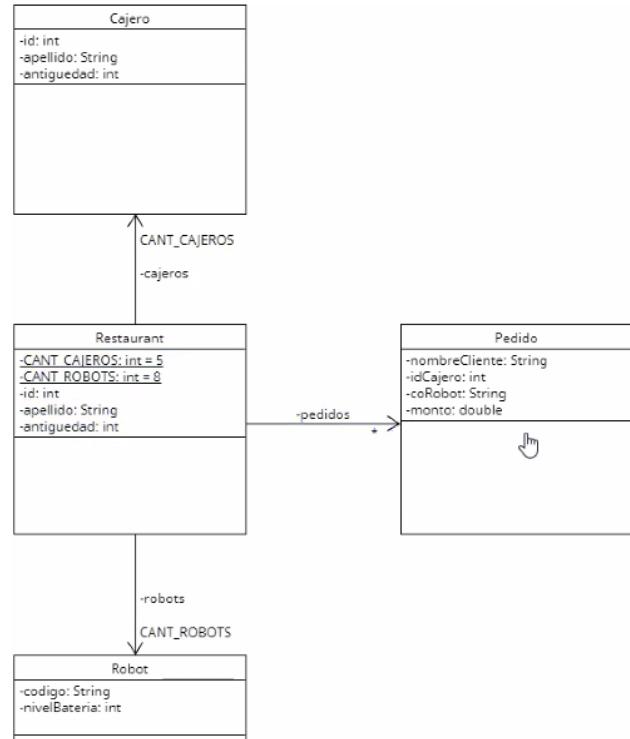
El siguiente caso muestra la implementación de estos métodos con claves *comparables* (por ejemplo Strings):

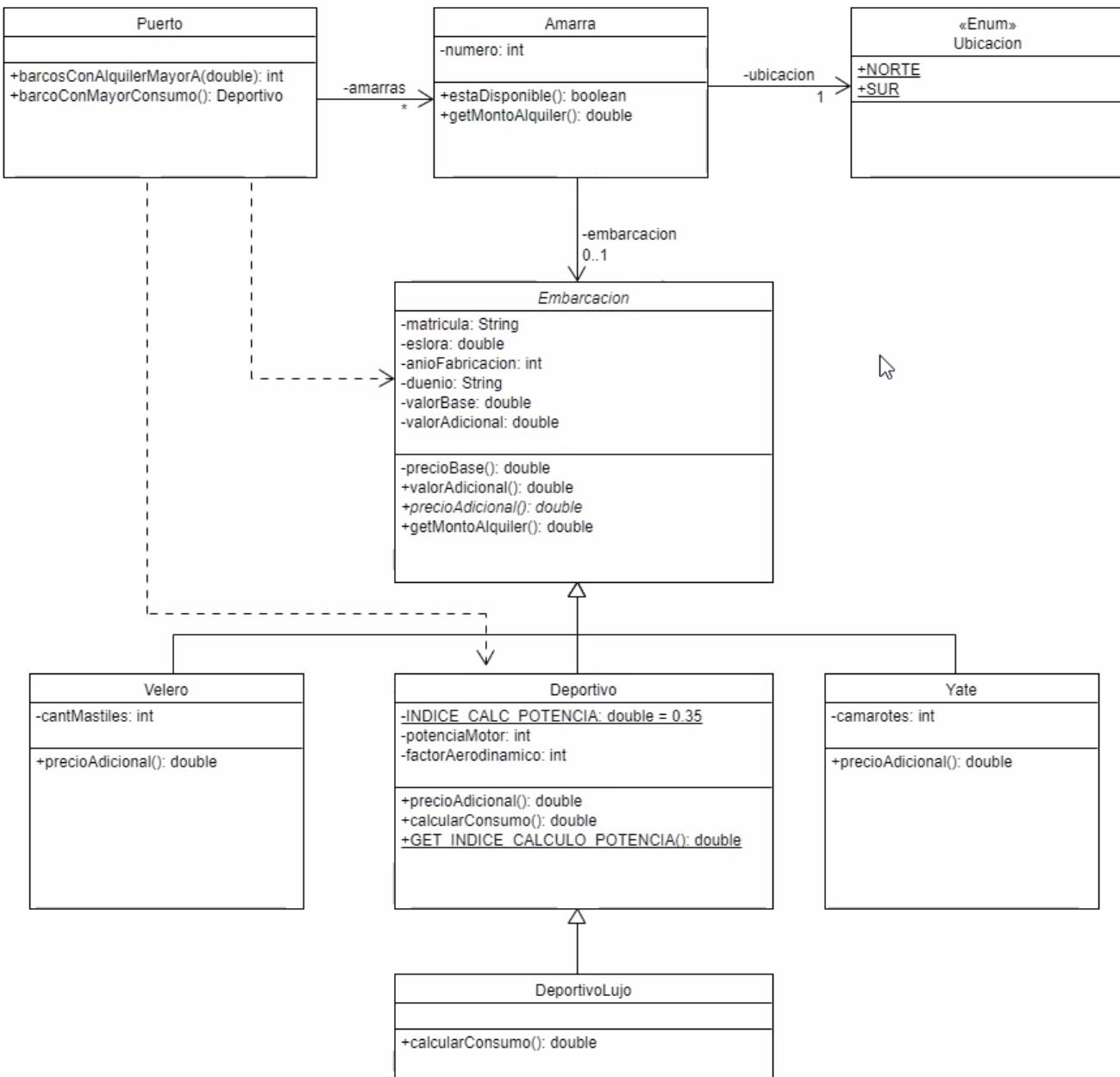
```
public class OrdenadaPorString extends ListaOrdenada<String, Elemento> {  
  
    @Override  
    public int compare(Elemento elemento1, Elemento elemento2) {  
        return elemento1.getClave().compareTo(elemento2.getClave());  
    }  
  
    @Override  
    public int compareByKey(String clave, Elemento elemento) {  
        return clave.compareTo(elemento.getClave());  
    }  
}
```

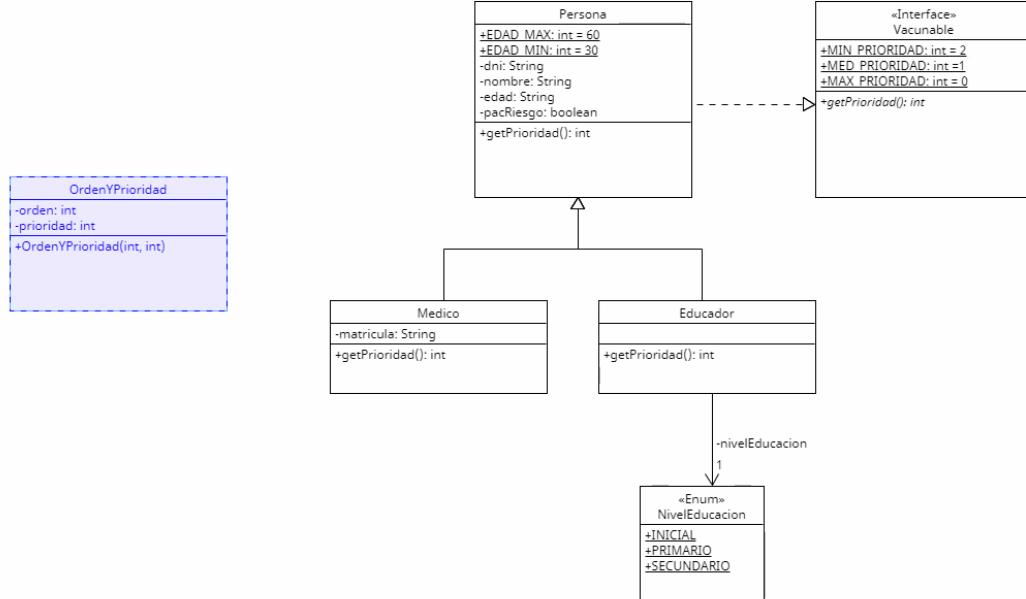
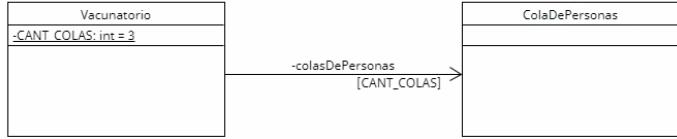
```
public class ListaAlumnosPorEdad extends ListaOrdenadaNodos<Integer, Alumno> {  
  
    @Override  
    public int compare(Alumno a1, Alumno a2) {  
        return a2.getEdad() - a1.getEdad();  
    }  
  
    @Override  
    public int compareByKey(Integer clave, Alumno a) {  
        return clave - a.getEdad();  
    }  
}
```

FORMA de hacer los uml









```

class Sede :
    public informe devolverInforme() ()

```

Tipo e : examenes

e instanceof ExamenAuto



```

class SedeDeLicencias :
    public double promTiempoExaAutosAprobados ()
    +int acuTiempos ← expresión +tipo nombre ← expresión

```

Examen e : this.examenes

e instanceof ExamenDeAuto && e.isAprobado()

acuTiempos += ((ExamenDeAuto) e).getTiempo()

contTiempos++

return (contTiempos > 0 ? (double) acuTiempo / contTiempos : 0)

F

```

class Area :
    public int obtenerCantidadObservacionesExternas ()
    +int cantidad ← 0

```

Obesercacion observacion : observaciones

/* uso de instanceof */

observacion instanceof ObservacionExterna

cantidad++

return cantidad

F

```

class Restaurant:
    public InfoPedido[] informeDePedidos ()
    +InfoPedido[] informes ← new InfoPedido[pedidos.size()]

    int i ← 0 ,
    informes.length-1 , 1
    Pedido p ← this.pedidos.get(i)
    apellido ←
        this.cajeros[p.getIdCajero()-1].getApellido()
    Robot robot ← buscarRobot(p.getCodRobot())
    nivel ← robot.getNivelBateria()
    informes[i] ←
        new InfoPedido(p.getMonto(), apellido, nivel)

    return informes

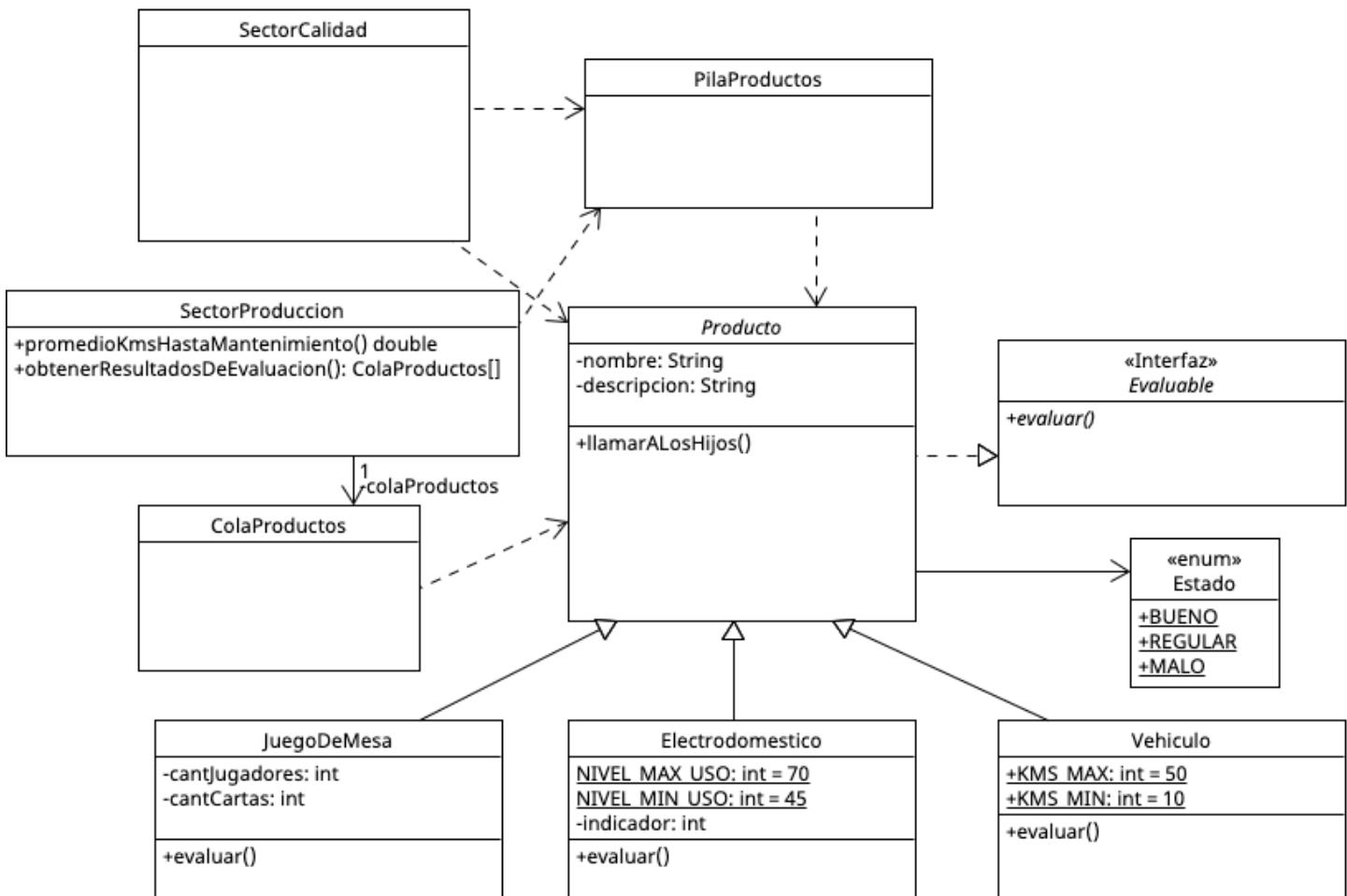
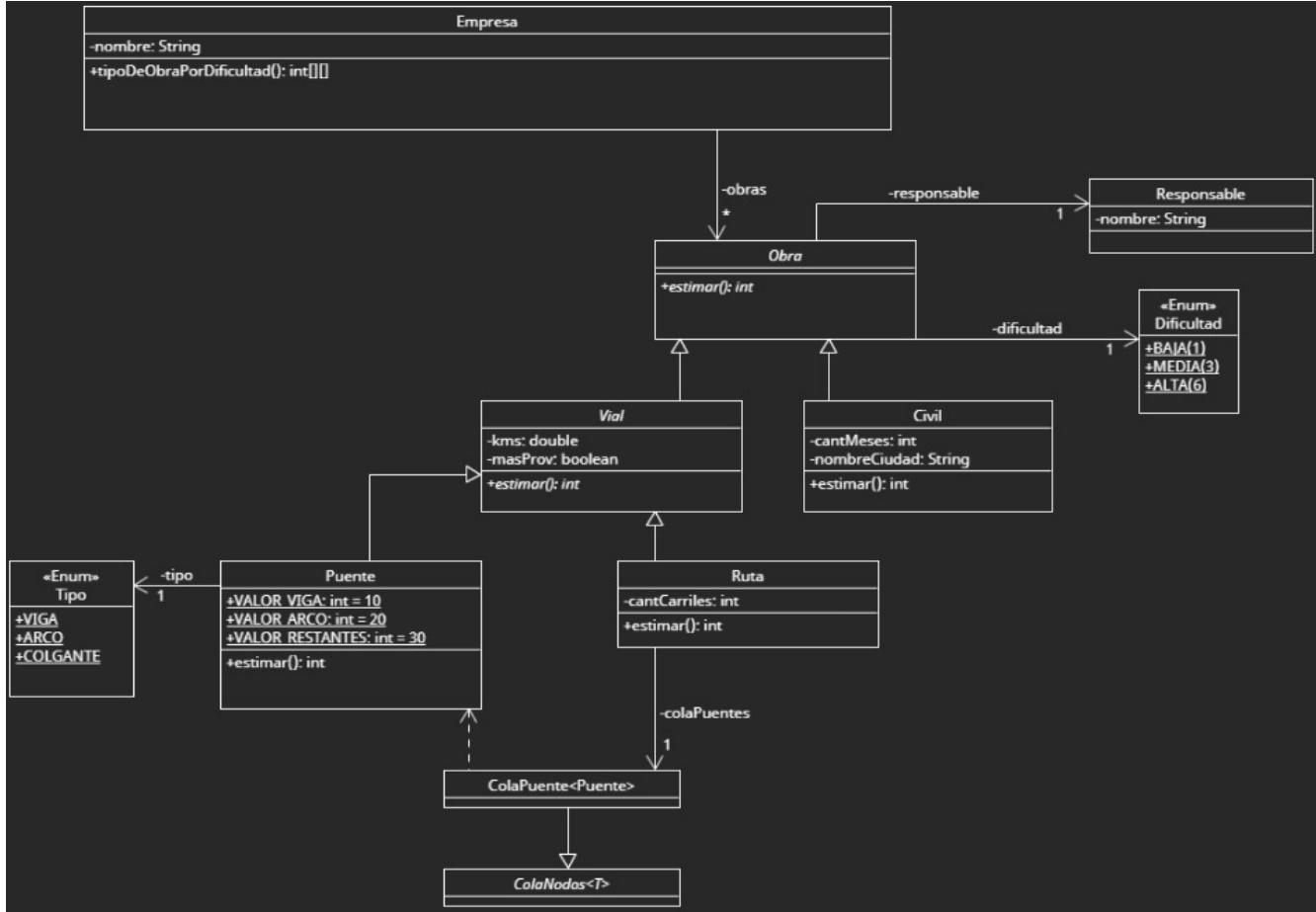
class Restaurant:
    public void mostrarPedidosConMasRiesgo ()
    +int cantPedidos ← this.pedidos.size()  +Robot robotConMenorNivelDeBateria
    robotConMenorNivelDeBateria ← buscarRobotConMenorNivelDeBateria()

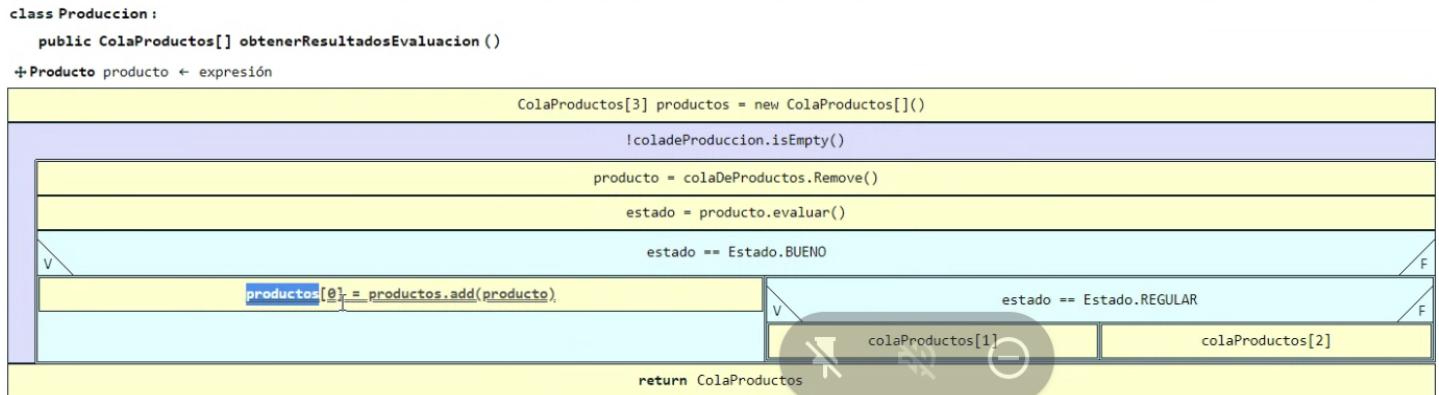
    int p ← 0 , cantPedidos-1 , 1
    Pedido pedido ← this.pedidos.get(p)
    pedido.getCodRobot() == robotConMenorNivelDeBateria.getCodigo()

class ListaOrdenesPorMonto:
    public int compare (+Orden o1 , +Orden o2 )
    +double dif  +int r ← 0
    dif ← o2.getMonto() - o1.getMonto()

    if dif > 0
        r ← 1
    else if dif < 0
        r ← -1
    return r

```





Posicionamiento directo en una matriz

```

class TodoLibre:
    public int[][] cantEnviosEntreComunas ()
        + int[][] informe + Orden ordenCent
            informe ← new int[CANT_COMUNAS][CANT_COMUNAS]
            ordenCent ← new OrdenPuertaPuerta()
            colaOrdenes.add(orden)
            orden ← colaOrdenes.remove()
            orden != ordenCent
            procesarOrden(orden, informe)
            colaOrdenes.add(orden)
            orden ← colaOrdenes.remove()
            return informe

```

```

class TodoLibre:
    private void procesarOrden (+ Orden orden , + int[][] informe)
        + int desde + int hasta
            desde ← orden.getDomicilioOrigen().getComuna() - 1
            hasta ← orden.getDomicilioEntrega().getComuna() - 1
            informe[desde][hasta]++

```

```

class Orden:
    public int informarDomicilioSalida ()
        return vendedor.getDomicilio()

class PuertaPuerta:
    public Domicilio informarDomicilioEntrega ()
        return getComprador().getDomicilio()

class TodoLibre:
    public ListaOrdenesPorMonto ordenesPuertaPuertaPorMonto ()
        + ListaOrdenesPorMonto lista + Orden ordenCent + Orden orden
        lista ← new ListaOrdenesPorMonto()
        ordenCent ← new OrdenPuertaPuerta()
        colaOrdenes.add(ordenCent)
        orden ← colaOrdenes.remove()
        orden != ordenCent
        procesarOrden(orden, lista)
        colaOrdenes.add(orden)
        orden ← colaOrdenes.remove()
        return lista

    private void procesarOrden (+ Orden orden, + ListaOrdenesPorMonto lista)
        + OrdenPuertaAPuerta ordenPaP
        orden instanceof OrdenPuertaAPuerta
        ordenPaP ← (OrdenPuertaAPuerta) orden
        ordenPaP.hasEnvioGratis()
        lista.add(ordenPaP)

```

COMPARE DE LISTA ORDENADA PARA DOUBLE DE FORMA DESCENDENTE (SI QUIERO DE FORMA ASCENDENTE SOLO CAMBIO EL PRIMER 1 Y LO PONGO -1 Y EL SEGUNDO 1 QUE TIENE -1 LO PONGO 1)

```

class ListaOrdenadaPorMonto :
    public int compare ( Orden o1 , Orden o2 )
        return (o1.getMonto() - o2.getMonto() < 0) ? 1 : (o1.getMonto() - o2.getMonto() > 0) ? -1 : 0

```

COMPAREBYKEY DE DOUBLE EN FORMA ASCENDENTE

```

class ListaOrdenadaPorMonto :
    public int compareByKey ( Double clave , Orden o )
        return (clave - o.getMonto() < 0) ? -1 : (clave - o.getMonto() > 0) ? 1 : 0

```

COMPAREBYKEY DE DOUBLE EN FORMA DESCENDENTE

```

class ListaOrdenadaPorMonto :
    public int compareByKey ( Double clave , Orden o )
        return (clave - o.getMonto() < 0) ? 1 : (clave - o.getMonto() > 0) ? -1 : 0

```

Método que se le pasa por parámetro una lista ordenada y un objeto y agrega ese objeto a la lista solo si es del tipo que queremos

```

class TodoLibre :
    private void buscarPuertaPuertaGratis ( Orden o , ListaOrdenadaPorMonto lista )
        V
            o instanceof OrdenPuertaPuerta
        V
            ((OrdenPuertaPuerta)o).envioGratis()
        lista.add(o)
        /* version ultra pro */
        V
            o instanceof OrdenPuertaPuerta && ((OrdenPuertaPuerta)o).envioGratis()
        lista.add(o)

```

```

class TodoLibre :
    public ListaOrdenadaPorMonto ordenesPuertaPuertaGratisPorMonto ()
        Orden cent ← null    Orden orden    ListaOrdenadaPorMonto lista ← new ListaOrdenadaPorMonto()
        colaDeOrdenes.add(cent)
        orden ← colaDeOrdenes.remove()
        orden != cent
        buscarPuertaPuertaGratis(orden, lista)
        colaDeOrdenes.add(orden)
        orden ← colaDeOrdenes.remove()
        return lista

```

```

private void ordenar(Cola<Producto> cola) {
    Producto c = null;
    Producto p;
    cola.add(c);
    p = cola.remove();
    while (p != c) {
        p.pedir();
        cola.add(p);
        I p = cola.remove();
    }
}

```

Enunciado

Todos los **empleados** de un **instituto** educativo, de los cuales se sabe el **DNI** y su **año efectivo de ingreso**, deben ponerse en **fila** para ser fiscalizados.

Para este prototipo se pone énfasis sólo en los **auxiliares** (se sabe además su cantidad de horas de trabajo semanales) y en los **profesores**, de los que se sabe el nivel de estudios alcanzado (terciario, universitario o posgrado) y los cargos que actualmente tiene vigentes, de los que se conoce el nombre de la institución donde se desempeña, el nivel (terciario, universitario o posgrado) y la cantidad de horas reloj.

Los cargos de un profesor están dispuestos en un calendario semanal (**días hábiles**) donde además se ven discriminados por turno (mañana, tarde o noche). Si un profesor no tiene algún cargo para cierto día y turno, se guarda **null**.

	Lunes	Martes	Miércoles	Jueves	Viernes
M					
T					
N					

/*hola, toma a cargo como una entidad que tiene día+horario+lugar

y luego trabaja con la matriz de días por turnos... que contenga a los cargos*/

De acuerdo a la normativa vigente, estas son las condiciones necesarias para considerar alguna irregularidad en algún empleado particular:

- Auxiliares: Cuando tienen más de 10 años de antigüedad* o si trabajan menos de 20 horas semanales.

- Profesores: Cuando poseen algún cargo de mayor nivel que sus estudios** o si tienen más de 3 cargos nocturnos.

* Asumí que contás con la clase **Fecha** y un método estático **añoActual()** ya implementado que retorna el año actual para calcular la antigüedad.

** Por ejemplo, si el profesor tiene un cargo de posgrado pero su título es de menor nivel (terciario o universitario)

Basado en el enunciado descrito, realiza:

- A. El diagrama de clases que lo modelice, con sus relaciones, atributos y métodos.
- B. El método **empleadosEnFalta()**, de la clase que corresponda, que debe devolver (no mostrar por consola) a los empleados que hayan presentado alguna irregularidad, en una colección que permita accederlos rápidamente conociendo su DNI. Deben desarrollarse también los métodos asociados que se usen, sean o no de la misma clase.

Enunciado

La Administración Federal de Ingresos Pùblicos nos contrató para realizar un sistema para gestionar los reclamos que le ingresan debido al cobro de impuestos a los bienes personales. Dado que el proyecto se implementará de manera escalonada para esta primera versión se contempla su uso solamente en la ciudad de buenos aires.

De cada ciudadano la AFIP conoce nombre, apellido, CUIL/CUIT y una lista de bienes tributables que pueden ser Vehículos, Inmuebles o Armas.

- Los vehículos declaran la marca, modelo y cilindrada. El Impuesto a abonar por los mismos se determina por un valor base que equivale al 2% del valor de la cilindrada, más un adicional que depende del tipo del vehículo:
 - si es un auto se le suma un 2% adicional,
 - si es una moto, se le suma un adicional fijo dependiendo del tipo de moto con el que fue registrada: CALLE (1600\$) DEPORTIVAS (6000\$), CHOPERAS (3000\$).
- De los inmuebles se conoce la superficie, la dirección y el valor fiscal. Abonan el 3% de su valor fiscal.
- Las armas poseen un número de serie, marca y modelo: el impuesto determinado es un valor fijo de 600\$.

Cuando un ciudadano detecta que le cobraron erróneamente se dirige a la agencia de AFIP y solicita la carga de un reclamo. El empleado de la mesa de entradas carga en el formulario de reclamo el CUIL/CUIT del ciudadano reclamante y el monto cobrado y lo pone en una bandeja sobre los reclamos anteriores; los mismos se procesarán todos juntos al final del día.

Al procesar estos reclamos se verifica que el ciudadano exista y que el monto reclamado sea distinto al importe del impuesto determinado para dicho ciudadano, que se calcula adicionando los impuestos determinados de cada bien tributable que posea el contribuyente. En el caso de detectarse una inconsistencia (el monto reclamado sea distinto al monto determinado) el reclamo se derivará a un área especial la cual procesará todos los reclamos en el orden que fueron procesados.

Basado en el enunciado descrito, realiza:

- A. El diagrama de clases que lo modelice, con sus relaciones, atributos y métodos.
- B. El método **determinarImpuesto()**, de la/s clase/s que corresponda, que debe **devolver** (no mostrar por consola) el monto a abonar en impuestos.
- C. El método **procesarReclamos()** de la clase que corresponda, que debe procesar los reclamos de los ciudadanos devolviendo una estructura que contenga los reclamos a procesar según lo indicado.

Entrega

Al terminar el ejercicio generá el archivo Nassi-Shneiderman de NS+ (el .nsplus) y los archivos de UMLetino (.uxf y png) y adjuntalos al examen en un único archivo .zip cuyo nombre será SEDE_CURSO_APELLIDO_NOMBRE.zip (por ejemplo YA_PR1A_PEREZ_JUAN.zip).

Exporta como NSPLUS

Una vez adjuntados los archivos al examen, "entregá" asegurándote de que los archivos hayan llegado al profesor a cargo (seguí las instrucciones de la plataforma para cerrar correctamente el examen usando los botones "Terminar Intento" y "Enviar todo y terminar"). La no entrega del .nsplus y/o de los archivos de UMLetino invalida el examen.

IMPORTANTE: Sólo se tendrán en cuenta los archivos de NS+ generados desde la versión de NS+ accesible desde el Aula Virtual (las otras no son compatibles).

Criterios

Para considerar aprobado el examen, el mismo debe demostrar la correcta aplicación de los siguientes conceptos de la programación orientada a objetos:

- Correcta definición de clases y asignación adecuada de sus responsabilidades.
- Encapsulamiento, ocultamiento de información y uso de getters y setters sólo cuando corresponda.
- Modularización reutilizable y mantenible con uso de métodos con correcta parametrización.
- Correcta aplicación de miembros de instancia y de clase.
- Correcta aplicación de herencia y polimorfismo.
- Correcta aplicación conceptual de las relaciones entre clases.
- Correcta aplicación de TADs vistos en clase