Tecnología de la Programación **22 de diciembre de 2024**
Duración: 3h:30m Evaluación Final.

Las soluciones tienen que estar razonadas. Comenta el código. Justifica el valor de retorno de los métodos. Se asume que todos los atributos privados tienen propiedades get/set (no los escribas).
ENTREGA EL ENUNCIADO JUNTO CON TUS SOLUCIONES.

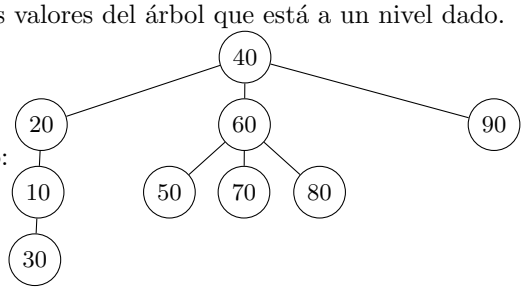
Apellidos y Nombre: .....

Tiempo máximo estimado: Ejercicio 1 y 2 (1h), ejercicio 3 (45'), ejercicio 4 y 5 (1h). Total: 2h 45'. CONTROLA EL TIEMPO.

Ejercicio 1 (1 puntos) Se tiene una lista simplemente enlazada **sin** cabecera. ¿Cuál es el método que permite eliminar elementos duplicados? Usa las esstructuras `ListNode` y `LinkedList`. Máxima calificación si se recorre la lista solo una vez.

Ejercicio 2 (2 puntos) Dado un árbol n-ário, calcular el valor máximo de entre los mínimos de cada nivel. Para ello definirás las estructuras `NAryNode` y `NAryTree` junto con, al menos, 2 funciones auxiliares: la que calcula la profundidad del árbol y la que calcula los valores del árbol que está a un nivel dado.

Ejemplo:



Nivel	Mínimo	Máximo
1	$40 = \min\{40\}$	
2	$20 = \min\{20, 60, 90\}$	40
3	$10 = \min\{10, 50, 70, 80\}$	
4	$30 = \min\{30\}$	

Profundidad = nº de niveles que tiene el árbol. Profundidad del árbol vacío es 0. Profundidad de 90 es 2. Profundidad de 50 es 3. La del 40 es 4.
La profundidad de un nodo es 1+max(profundidad de sus hijos).

Ejercicio 3 (2.5 puntos) *En una estación de tren, hay una ventanilla, una sala de espera y trenes. Una* ***estación*** *tiene un nombre y cuando se abre (al público) se sube el interruptor que enciende las luces de la estación. Una* ***ventanilla*** *también se abre (al público), pero de forma independiente a la apertura de la estación. En la* ***sala de espera*** *se encuentran los pasajeros que posteriormente se montarán en algún tren. Cada* ***tren*** *tiene un cartel con la estación de destino, una locomotora (siempre es la misma y es lo mínimo para tener un tren) y le siguen 0 más vagones que están, desde la locomotora, en el orden inverso en el que fueron enganchados. Cada* ***locomotora*** *está conducida por un maquinista, conocido por un apodo, y tiene un motor diésel (estándar, incorporado de fábrica y no puede sustituirse) que consume una cantidad de litros por Km; además tiene una palanca que, al subirla, permite acelerar el motor. Cada* ***vagón*** *tiene un número, que puede variar, que lo identifica dentro de cada estación. Pueden ser de pasajeros o de mercancías. Cada vagón de* ***pasajeros*** *tienen una cantidad de asientos y cada vagón de mercancías puede transportar un número máximo de toneladas. Existen varios tipos de vagones de* ***mercancías****: los de cisterna (que sólo pueden llevar o líquido o gas) y los de plataforma (cada uno admite cargas de hasta cierta longitud). Hay vagones* ***plataforma*** *que o bien transportan contenedores o bien transportan cargas de naturaleza y tamaños heterogéneos (p.e. maquinaria agrícola, coches, troncos de madera, etc) Ningún vagón con contenedores pueden llevar más de n-contenedores en línea a lo largo del vagón y en el exterior de cada contenedor hay un inventario de su contenido donde, para cada artículo, se conoce su cantidad y el peso total.*
1 Con la información superior construye, **usando la notación de Python** con la máxima **abstracción** posible, todas las clases/interfaces implicadas indicando claramente los decoradores necesarios, el modificador de visibilidad de cada **miembro**, y para cada atributo usa el tipo de dato **que mejor se ajusta a cada atributo**, dejando claro si es de instancia o de clase, y si es constante o no. Indica, para cada atributo, si tendría una propiedad o método set y/o get en forma de comentario.
P.e. `self._salero: Set[Sal] # get, set` representa un atributo protegido y su valor es un conjunto cuyos elementos son granos de sal y tiene sentido hacer un get y set sobre dicho atributo. `self.__notas: Dict[DNI, int] # get` representa un diccionario privado al que solo se puede consultar la calificación numérica asociada a un DNI. 2 Construye los métodos de la situación indicada; pero **NO** construyas los métodos set/get, SALVO que sean para atributos de clase. 3 ¿Ha sido necesario hacer polimorfismo? ¿Dónde? 4 Si el motor de la locomotora fuera eléctrico ¿qué principio SOLID hay que aplicar y cómo?

En lo que sigue, no reescribir lo que ya tengas de ejercicios anteriores, salvo los nombres de las clases. No escribir get/set

Ejercicio 4 (2.5 puntos) *La ventanilla tiene un cartel con los destinos de los trenes y puede expedir una cantidad de billetes para cada tren que hay en la estación, cantidad que coincide con el número de asientos de pasajeros que tienen los vagones de pasajeros de cada tren. Esto se determina exactamente cuando se abre (al público) la ventanilla.* 1 Indica cuáles son los atributos que necesita una ventanilla para resolver esta situación y por qué. 2 Construye el método de apertura de ventanilla para determina la cantidad de billetes para cada tren, construyendo los métodos que necesites de otras clases. Se asume que no hay dos trenes que tengan el mismo destino. 3 Guarda en un fichero los valores de los atributos de la ventanilla usando un diccionario cuyas claves sean los nombres de los atributos, controlando los errores.

No olvides usar los modificadores de visibilidad.

Ejercicio 5 (2 puntos) *Un pasajero tiene un nombre, una cantidad de dinero en metálico y un billete (o no). Todos los billetes tienen el mismo precio (que el pasajero desconoce) y cuando se expiden se les estampa un identificador único y el lugar de destino. El pasajero, al comprar un billete, debe usar la ventanilla y ésta necesita conocer un destino, le quitará al pasajero el dinero equivalente al precio del billete y le entregará un billete al pasajero siempre y cuando: la ventanilla esté abierta, haya billetes para el destino y el pasajero tenga dinero suficiente para comprar el billete.* 1 Escribe la clase Billete. 2 Escribe la clase Pasajero con su constructor y su correspondiente método para comprar un billete en la ventanilla retornando un valor que indique si se pudo comprar o no. 3 Para la ventanilla, escribe el método de venta que expide un billete a un pasajero para un destino.

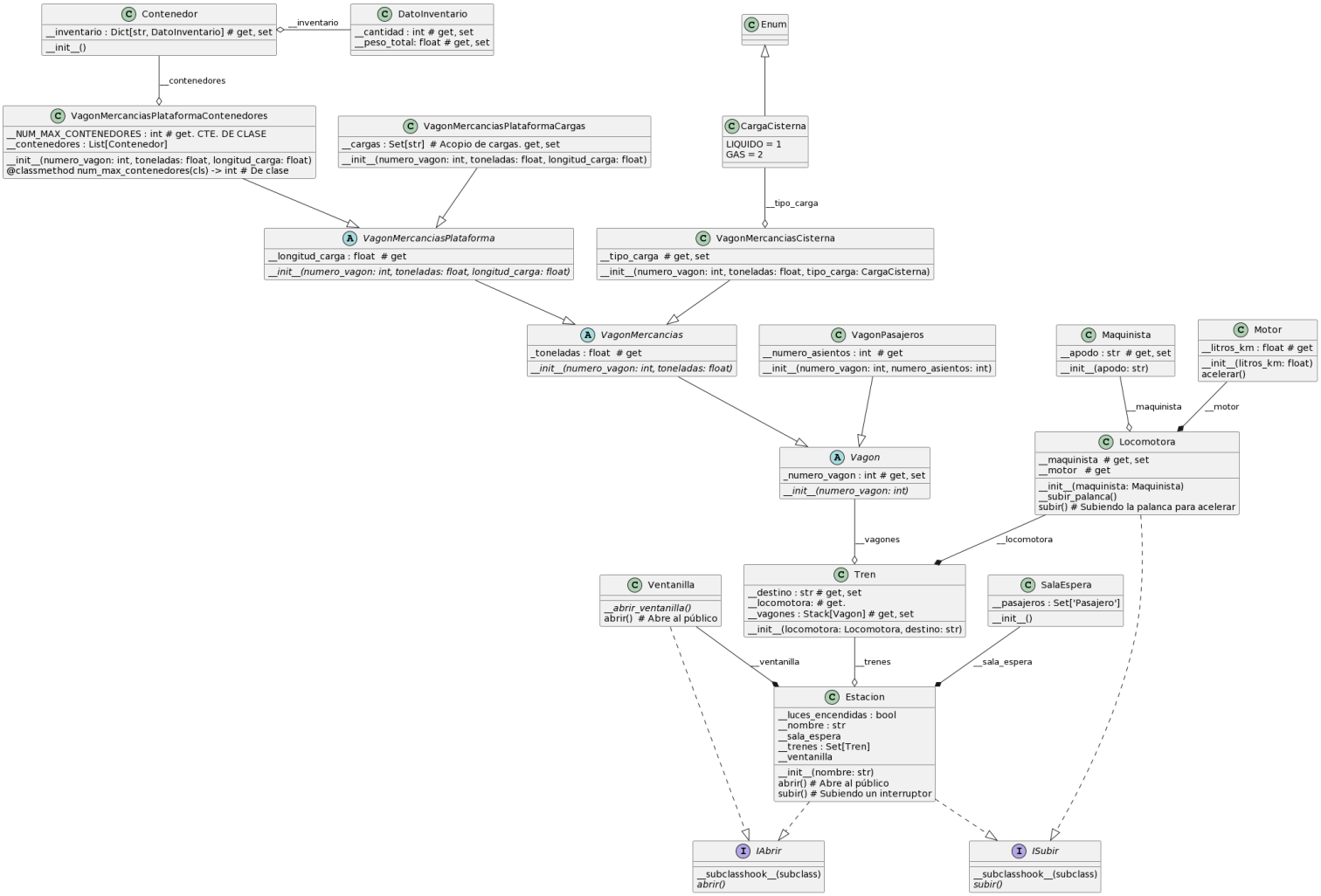
Ejercicio 1

```
class ListNode:
    def __init__(self, value: Any):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.first = None

    def remove_duplicates(self):
        if not self.first:
            return
        unique_values: Set[Any] = set()
        current: ListNode = self.first
        previous: ListNode = None
        while current:  # Si se usa for (iteradores) NO se pueden borrar los elementos
            if current.value in unique_values:
                previous.next = current.next
            else:
                unique_values.add(current.value)
                previous = current
            current = current.next
```

Ejercicio 2

```
from collections import deque # O cualquier otra Cola

class NAryNode:
    def __init__(self, value):
        self.value = value
        self.children: List | None = None

class NAryTree:
    def __init__(self, root=None):
        self.root = root

    def profundidad_del_arbol(self): # Parte NO recursiva
        return 0 if not self.root else self._calcular_profundidad(self.root)

    def _calcular_profundidad(self, nodo): # Parte recursiva
        if not nodo.children:
            return 1
        profundidades_hijos = [self._calcular_profundidad(hijo) for hijo in nodo.children]
        return 1 + max(profundidades_hijos)


    def nodos_en_mismo_nivel(self, nivel_objetivo):
        """Aplicamos búsqueda en anchura (necesita una cola) para obtener los nodos en el nivel objetivo"""
        if self.root is None:
            return []

        cola = deque([(self.root, 1)])  # Tupla: (nodo, nivel). # deque() requiere un iterable
        nodos_nivel_actual = []
        while cola:
            nodo_actual, nivel_actual = cola.popleft() # Extraemos el 1er elem. de la cola
            if nivel_actual == nivel_objetivo:
                nodos_nivel_actual.append(nodo_actual.value)
            cola.extend((hijo, nivel_actual + 1) for hijo in nodo_actual.children)

        return nodos_nivel_actual


    def valor_maximo_por_niveles(self):
        if self.root is None:
            return None
        profundidad = self.profundidad_del_arbol()
        minimos = []
        for nivel in range(1, profundidad + 1):
            minimos.append(min(self.nodos_en_mismo_nivel(nivel)))
        return max(minimos)
```

Ejercicio 3



En este ejercicio se pide **usar la notación de Python** con la máxima **abstracción** posible, todas las clases/interfaces implicadas indicando claramente los decoradores necesarios, el modificador de visibilidad de cada **miembro**, y para cada atributo usa el tipo de dato **que mejor se ajusta a cada atributo**, dejando claro si es de instancia o de clase, y si es constante o no. Indica, para cada atributo, si tendría una propiedad o método set y/o get en forma de comentario.
P.e. `self._salero: Set[Sal] # get, set` representa un atributo protegido y su valor es un conjunto cuyos elementos son granos de sal y tiene sentido hacer un get y set sobre dicho atributo. `self.__notas: Dict[DNI, int] # get` representa un diccionario privado al que solo se puede consultar la calificación numérica asociada a un DNI.
En concreto, se espera encontrar:

- Distinguir entre atributos privados y protegidos.
- Componer la locomotora con el motor (que viene de fábrica). Composición de Estación con Ventanilla y SalaEspera. Agregación de Trenes en la Estación.
- El tren tiene una pila de vagones.
- Usar un atributo de **clase** (y constante) sobre el número de contenedores en los vagones de contenedores y el método get() para dicho atributo. Los contenedores forman una lista en el vagón.
- Usar, al menos, 2 clases abstractas, indicando los métodos abstractos.
- Herencia entre los distintos tipos de vagones.
- Usar, al menos, una interface (abrir(), abrir_al_publico(), o similar)

Ejercicio 4
Para que al abrir la ventanilla se pueda conocer el número de asientos que hay en cada tren (destino) hemos de delegar en ellos para que suministren esa información. Como trenes y ventanilla están en una estación, el punto de unión es la estación. Por tanto, no basta con que la estación conozca a los trenes y a la ventanilla, se necesita que la ventanilla también conozca a la estación. Entonces, por delegación, la ventanilla pedirá dicha información a la estación y la estación se lo pedirá a cada uno de los trenes.
Es fundamental tener en cuenta que al preguntar a los trenes, los vagones se encuentran en una PILA. Los vagones NO están en una lista: no se puede acceder al vagón que está junto a la locomotora sin antes haber quitado todos los de la cola.

```
class Ventanilla(IAbrir):
    # .... resto del código

    def __init__(self):
        self.__estacion: None | 'Estacion' = None # Necesario para acceder a los trenes
```

```
        self.__cartel: List[str] = list()  # get, set. El cartel del enunciado.
        self.__billetes: Dict[str, int] = dict()  # Clave: destino, Valor: cantidad de billetes

    def abrir(self):  # << Dado por la interface
        self.__abrir_ventanilla(). # << Se adapta a esta clase

    def __abrir_ventanilla(self):
        self.__cantidad_billetes_por_tren()
        self.__cartel = list(self.__billetes.keys())

    def __cantidad_billetes_por_tren(self):
        self.__billetes = self.__estacion.cantidad_billetes_por_tren() # Delego en la estación

    def obtener_atributos(self) -> Dict[str, object]:
        return {
            'estacion': self.__estacion.nombre, # get_nombre()
            'cartel': self.__cartel,  # La lista de destinos
            'billetes': self.__billetes, # La cantidad de billetes por tren
        }

    def guardar_en_archivo(self, nombre_archivo: str):
        import json
        atributos = self.obtener_atributos()

        try:
            with open(nombre_archivo, 'w') as archivo:
                json.dump(atributos, archivo, indent=2)
        except Exception as e: # Es mucho más correcto usar logging.
            print("## Error al guardar el archivo", e)
```

```
class Estacion(IAbrir, ISubir):
    # .... resto del código

    def cantidad_billetes_por_tren(self) -> Dict[str, int]:
        billetes: Dict[str, int] = dict()
        for tren in self.__trenes:   # La estación conoce sus trenes, delega en ellos.
            billetes[tren.destino] = tren.numero_de_asientos_para_pasajeros()

        return billetes
```

```
class Tren:
    # .... resto del código
    def numero_de_asientos_para_pasajeros(self) -> int:
        contador: int = 0
        for vagon in self.__vagones:  # << Esto no es correcto porque los vagones no forman una lista
            if isinstance(vagon, VagonPasajeros):  # Solo nos interesa los de pasajeros
                contador += vagon.numero_asientos(). # Este es el método get del atributo. No se pide
        return contador

        # Nota. Lo correcto es usar una pila auxiliar donde guardar los trenes que se van desenganchando.
        ↪  Después se vuelven a enganchar.
```

Ejercicio 5

Esta es una relación de uso del pasajero con la ventanilla, Hay que tener en cuenta que el pasajero no conoce el precio por eso se indica que el pasajero desconoce el precio y, aún más, se indica que para la ventanilla se escriba un método de venta que expide un billete a un **pasajero** para un **destino** (con dos argumentos).

```
class Billete:
    # Atributos de clase. Todos los atributos son privados.
    __PVP: int = 10  # CTE, get Precio del billete que desconoce el pasajero
    __id: int = 0  # get, set. Lo usamos para el identificador único

    @classmethod  # << Para conocer el precio del billete.
    def pvp(cls) -> int:
        return cls.__PVP

    @classmethod. # << Para obtener el identificador
    def get_id(cls) -> int:
        cls.__id += 1
        return cls.__id

    def __init__(self, destino: str):
```

```
        "Se expide el billete con su destino y su identificador único"
        self.__numero: int = self.__class__.get_id()  # get
        self.__destino: str = destino  # get.
```

```
class Ventanilla(IAbrir):
    # .... resto del código
    # Para esta parte se necesita un atributo que determine si la ventanilla está abierta
    def vende_billete_destino(self, destino: str, pasajero: Pasajero):
        if self.__abierta:
            if destino in self.__billetes and self.__billetes[destino] > 0:
                if pasajero.dinero < Billete.pvp():
                    raise ValueError("No tienes dinero suficiente")
                pasajero.dinero -= Billete.pvp() # Quita dinero al pasajero
                self.__billetes[destino] -= 1.   # Quita un billete
                return Billete(destino)          # Expide nuevo billete
            else:
                raise ValueError("No hay billetes para ese destino")
        else:
            raise ValueError("La ventanilla está cerrada")
```

```
class Pasajero:
    def __init__(self, nombre: str, dinero: float):
        self.__nombre: str = nombre  # get
        self.__dinero: float = dinero  # get, set
        self.__billete: Billete | None = None  # get, set

    def compra_billete(self, destino: str, ventanilla: 'Ventanilla') -> bool:
        try:
            self.billete = ventanilla.vende_billete_destino(destino, self)
            return True
        except ValueError as e:
            return False
```