Tema 1

Califique las siguientes afirmaciones como verdaderas o falsas. En cada caso, justifique apropiadamente su respuesta:

- La operación para obtener el tamaño de cualquier lista se realiza siempre en tiempo de ejecución constante.
- Hay ciertas clases de Java que permiten invocar al método pop de una pila vacía sin producir excepciones de ningún tipo.
- Las colas de prioridad son siempre una estructura de datos de tipo LIFO (Last In, First Out).
- En una cola de prioridad, el método poll devuelve siempre el elemento con la mayor prioridad.
- Un mapa almacena elementos en pares clave-valor y los valores asociados a las claves deben ser siempre únicos. Es decir, distintas claves no pueden asociarse a un mismo valor.

Tema 2

Una tienda en línea desea implementar un sistema para gestionar los pedidos realizados por los clientes. Cada pedido está representado por un objeto de la clase Pedido, que contiene la siguiente información:

```
class Pedido {

    private int numero;
    private String cliente;
    private List<String> productos;

    // Constructores, getters y setters
}
```

La tienda almacena todos los pedidos registrados en una lista. Un ejemplo de cómo luce esta lista se muestra a continuación:

Lista `pedidos`

N°	Cliente	Productos
1	`Cliente1`	{P1, P2}
2	`Cliente2`	{P3, P4, P11}
3	`Cliente1`	{P5, P6}
4	`Cliente3`	{P7, P8}
5	`Cliente1`	{P9, P10}

Implemente un método llamado **`getPedidosPorCliente`** que reciba como parámetro una lista de pedidos y un cliente, y retorne una lista con todos los productos solicitados por ese cliente.

Para la lista de ejemplo indicada arriba, el método **`getPedidosPorCliente`** retorna la lista de productos {`P1, P2, P5, P6, P9, P10`} cuando el cliente enviado como parámetro del método es el `Cliente1`. Si, en cambio, el método se invoca con el `Cliente2`, el método retorna la lista {`P3, P4, P11`}

Tema 3

El juego del apilamiento consiste en construir una torre lo más alta posible apilando bloques uno encima de otro. El juego funciona de la siguiente manera:

- Cada bloque está representado por un número entero positivo que indica su altura.
- Los bloques son colocados en una pila que, al inicio del juego, se encuentra vacía.
- En cada turno, el jugador debe seleccionar un bloque de una colección de bloques disponibles y añadirlo a la torre.
- El bloque seleccionado debe tener una altura menor o igual al bloque superior de la torre actual.
- El juego termina cuando no quedan bloques disponibles para añadir (porque se acabaron, o porque el bloque seleccionado tiene una altura mayor al bloque superior de la torre).
- El puntaje del jugador se determina sumando las alturas de los bloques de la torre que construyó.

Implemente el método **`simularApilamiento`**, que simula el juego descrito arriba. Para esto, el método recibe como parámetro una lista de números enteros, que representa los bloques disponibles. En cada turno del juego, su implementación debe seleccionar un bloque de manera aleatoria y, al final del juego, imprimir el puntaje otorgado al jugador de acuerdo a los bloques presentes en la torre. Internamente, su método debe representar a la torre mediante una pila.

Considere los ejemplos de ejecución mostrados a continuación:

Ejemplo 1:

Turno 1:
Bloques disponibles: [3, 2, 5, 1, 4]
Torre actual: [ ]
Bloque seleccionado: 5

Turno 2:
Bloques disponibles: [3, 2, 1, 4]
Torre actual: [5]
Bloque seleccionado: 2

Turno 3:
Bloques disponibles: [3, 1, 4]
Torre actual: [5, 2]
Bloque seleccionado: 4

El juego termina porque el la altura del bloque seleccionado (4) es mayor a la del bloque superior de la torre (2).
Puntaje total obtenido: 7

Ejemplo 2:

Turno 1:
Bloques disponibles: [3, 2, 5]
Torre actual: [ ]
Bloque seleccionado: 5

Turno 2:
Bloques disponibles: [3, 2]
Torre actual: [5]
Bloque seleccionado: 3

Turno 3:
Bloques disponibles: [2]
Torre actual: [5, 3]
Bloque seleccionado: 2

El juego termina porque ya no hay bloques disponibles (se seleccionaron todos).
Puntaje total obtenido: 10. Torre resultante [5, 3, 2]

Tema 4

Una tienda de departamentos desea identificar sus productos más valiosos en términos de ventas. Un producto se considera valioso cuando su precio es alto y se vende mucho.

En la tienda, cada producto tiene un código único (String), un nombre (String), y un precio (double). La tienda ha implementado un sistema de gestión de inventario que almacena información sobre sus productos y ventas en los siguientes mapas:

> Map<String, Integer> **ventas**: almacena los códigos de los productos como claves y la cantidad de ventas correspondiente como valor. El valor asociado a cada código indica cuántas unidades se han vendido de ese producto.
>
> Map<String, Double> **precios**: almacena los códigos de los productos como claves y el valor asociado a cada clave indica el precio del producto.

Utilizando la información de estos dos mapas, usted debe implementar el método **`obtenerProductosMasVendidos`** que, además, recibe un número entero **`n`** indicando la cantidad de productos más vendidos que se desean obtener. El método retorna una lista de Strings que contiene los códigos de los **`n`** productos más vendidos, ordenados de mayor a menor en términos de los ingresos generados para la tienda.

Asuma que usted cuenta con la clase Producto y que, de ser necesario, puede modificarla. En este tema, usted puede utilizar todas las clases y estructuras de datos provistas por Java.

Considere el ejemplo mostrado a continuación:

Mapa `ventas`

Código	Unidades
"P001"	10
"P002"	5
"P003"	8
"P004"	15
"P005"	20

Mapa `precios`

Código	Precio
"P001"	10.50
"P002"	15.75
"P003"	12.00
"P004"	8.50
"P005"	11.25

Cuando el método ob es invocado con **`n = 3`**, el programa retorna la lista {**“**`P005`**”**, **“**`P004`**”**, **“**`P001`**”**}. Estos productos (en el orden de la lista) son los que más ingresos han generado para la tienda.