A.E.D. Laboratorio 1

Guillermo Román
guillermo.roman@upm.es

Lars-Åke Fredlund
lfredlund@fi.upm.es

Manuel Carro
mcarro@fi.upm.es

Julio García
juliomanuel.garcia@upm.es

Tonghong Li
tonghong@fi.upm.es

Normas

- Fechas de entrega y penalización:

Hasta el martes 30 de septiembre, 16:00 horas	0 %
Hasta el miércoles 1 de octubre, 16:00 horas	20 %
Hasta el jueves 2 de octubre, 16:00 horas	40 %

  Después la puntuación máxima será 0
- Se comprobará plagio y se actuará sobre los detectados.
- Usad las horas de tutoría para preguntar sobre programación – son oportunidades excelentes para aprender.

Entrega

- Todos los ejercicios de laboratorio se deben entregar a través de

  `http://deliverit.fi.upm.es`

- El/los fichero(s) que hay que subir es/son `Almacen.java`.
- La clase debe estar en el paquete `aed.almacen` .
- La documentación de la API de `aedlib.jar` está disponible en `http://costa.ls.fi.upm.es/teaching/aed/docs/aedlib/`

Tarea: implementar la lógica de un almacén

- El almacén almacena productos (“helado”,”libro”, etc)
- Un **productor** envía productos al almacén (p.e. 100 “helado”)
- Un **cliente** compra productos (p.e. compra 5 “libros”)

Tarea: terminar la implementación de la clase `Almacen`

```
public class Almacen implements
  ClienteAPI,       // Llamadas de clientes
  ProductorAPI,     // Llamadas de productores
  AlmacenAPI        // Acceso a datos del almacen
{
  // Lista de compras realizadas
  private ArrayIndexedList<Compra> compras;
  // Lista de productos en almacen
  private ArrayIndexedList<Producto> productos;

  public Almacen() { ... }

  // A escribir codigo aqui ...
}
```

¡Documentación detallada en los interfaces!

La clase Producto guarda información sobre productos

```
public class Producto implements Comparable<Producto> {
  private String productoId;          // Nombre
  private int cantidadDisponible;     // Cantidad disponible

  // Constructor
  public Producto(String productoId, int cantidad) { ... }

  public String getProductoId() { ... }
  public int getCantidadDisponible() { ... }
  public void setCantidadDisponible(int cantidad) { ... }

  ...
}
```

La clase Compra guarda información sobre compras

```
public class Compra implements Comparable<Compra> {

  private Integer compraId;              // Id de la compra
  private String clienteId;              // Quien compra
  private String productoId;             // Que compra
  private int cantidad;                  // Cuantas unidades

  // Constructor
  public Compra(String clienteId,
                String productoId, int cantidad) { ... }

  // Setters y Getters
  ...
}
```

ClienteAPI: peticiones de compra de un cliente

```
Integer pedir(String clienteId,     // Quien compra?
              String productoId,    // Que producto?
              int cantidad);        // Cuanto compra?
```

- Comprueba si hay suficientes productos en el almacen
- Si hay suficientes, crea una objeto de la clase `Compra`
- Devuelve la identidad de la compra realizada – `compra.getCompraId()` – o `null` si no había cantidad suficiente
- Guarda la información de la compra dentro el atributo `compras`

¡Documentación detallada en los interfaces!

ProductorAPI: un productor manda productos al almacén

```
void reabastecerProducto(String productoId, // Que producto?
                         int cantidad);     // Cuantos?
```

- Si no existía ya un producto con el `productoId` mencionado, crea una objeto de la clase `Producto` y lo guarda dentro de la lista `productos`
- Si ya existía, aumenta la cantidad de dicho producto

¡Documentación detallada en los interfaces!

AlmacenAPI: acceso a datos del almacén

```
// Devuelve un producto con el Id especificado
public Producto getProducto(String productoId);
// Devuelve una compra con el Id especificado
public Compra getCompra(Integer compraId);
// Devuelve todos los productos
public IndexedList<Producto> getProductos();
// Devuelve todas las compras
public IndexedList<Compra> getCompras();
// Devuelve todas las compras realizadas por un cliente
public IndexedList<Compra> comprasCliente(String clienteId);
// Devuelve todas las compras de un producto
public IndexedList<Compra> comprasProducto(String productoId);
```

¡Documentación detallada en los interfaces!

Un Ejemplo

```
// Crea el almacen
Almacen a = new Almacen();
// Hay un producto "helado"?
a.getProducto("helado");                    // => null
// Llegan 10 "helado" desde el productor
a.reabastecerProducto("helado",10);
a.getProducto("helado");                    // => Producto("helado",10)

// "yo" compra 3 helados
a.pedir("yo","helado",3);                  // => 1 (la id de la compra)
a.getProducto("helado");                    // => Producto("helado",7); quedan 7 helados

// "tu" intenta comprar 8 "helado"
a.pedir("tu","helado",8);                  // => null (solo hay 7)
```

Un Ejemplo (Cont.)

```
// Llegan 10 helados mas
a.reabastecerProducto("helado",10);

// La compra de "tu" ahora funciona
a.pedir("tu","helado",8);              // => 2 (la id de la compra)

// Devuelve todas las compras realizadas
a.getCompras();                        // => [Compra(1,"yo",...),Compra(2,"tu",...)]
// Devuelve las compras realizadas por "tu"
a.comprasCliente("tu");                // => [Compra(2,"tu",...)]
// Intenta acceder a la compra 1
a.getCompra(1);                        // => Compra(1,"yo",...)
```

Consejos

- Notad que el constructor de una `Compra` genera y asigna un `compraId` único; no hay que especificarlo cuando se crea una `Compra`:

  ```
  Compra(String clienteId, String productoId, int cantidad)
  ```

- Para obtener una puntación máxima:
  - Los objetos dentro el atributo `productos` en la clase `Almacen` deberían estan ordenados según el campo `productoId`, en orden ascendente.
  - Y las operaciones de buscar y añadir un producto en dicha estructura deberían utilizar el algoritmo de "búsqueda binaria"
- Notad que la clase `Producto` implementa el interfaz `Comparable` para permitir que los productos estén ordenados segun `productoId`
- Se puede añadir metodos auxiliares (en `Almacen.java`)
- **No** se pueden añadir más atributos (en `Almacen.java`)

Ideas sobre métodos auxiliares útiles

- `int busquedaBinariaEnProductos(String id)`
  - Devuelve el índice donde se encuentra el producto con `id`
  - Si no existe un producto con `id`, *devuelve el indíce donde se debería insertar un producto con* `id`. Esto incrementa su utilidad.

Para descubrir cosas nuevas...

- Seguro que os preguntáis si se pueden/conviene almacenar las compras ordenadas y la respuesta sería: "depende del uso que se haga con las compras almacenadas..."
- En caso de querer almacenar las compras ordenadas os surgirá un dilema: ¿podría reutilizar la búsqueda binaria que ya tengo hecha?
- Parece que sí, pero no... al no cuadrar los tipos de datos no puedes reutilizar el método (siempre puedes copiar y pegar, pero duplicas código muy similar)
- Si os sobra tiempo y tenéis curiosidad, podéis explorar la posibilidad de usar métodos auxiliares más generales usando genéricos y *funciones*

- ```
  static <E,F extends Comparable<F>> int busquedaBinaria
      (IndexedList<E> l, // Lista de elementos tipo E
       F id,             // Identidad buscada
       Function<E,F> f)  // Function que extrae
                         // la identidad de un E
  ```
  ```

Notas Generales

- El proyecto debe compilar sin errores y debe cumplirse la especificación de los métodos a completar
- Debe pasar todos los test `TesterLab1` correctamente sin mensajes de error
- **Nota:** una ejecución sin mensajes de error y que pase todas las pruebas **no** significa que la implementación sea correcta (es decir, que funcione bien para cada posible entrada)
- Todos los ejercicios se corrigen manualmente antes de dar la nota final

Evaluación

¡Seguir estos consejos os permitirá conseguir mejores resultados!

- Corrección
- Ausencia de código repetido con la misma funcionalidad (podéis usar métodos auxiliares para evitarlo)
- Concisión y claridad del código
- Legibilidad, incluida selección de nombres descriptivos para variables y métodos
- El código debe estar correctamente indentado y con comentarios *útiles* cuando lo veáis necesario
- Eficiencia:
  - Se valorará la complejidad computacional del código
  - Se valorará no iterar innecesariamente en los recorridos de las estructuras de datos