	FUNDAMENTOS DE PROGRAMACIÓN **EJERCICIOS DE REPASO DEL BLOQUE 2 DE JAVA. SOLUCIONES**	**Curso: 2024/25**
APELLIDOS:	**NOMBRE:**	**DNI:**

Sean las siguientes definiciones de tipos:

Tipo **Producto**
Propiedades:
- **nombre**: String, consultable
- **precio**: Double, consultable

Tipo **Compra**
Propiedades:
- **fecha**: LocalDate, consultable y modificable
- **hora**: LocalTime, consultable
- **producto**: Producto, consultable y modificable
- **pago**: del tipo enumerado MedioPago (EFECTIVO, TARJETA, OTRO), consultable
- **unidades**: Integer, consultable y modificable
- **precio**: Double, consultable, derivada. El precio de la compra se obtiene multiplicando el precio del producto por el número de unidades

Tipo **Pedido**
Propiedades:
- **usuario**: String, consultable
- **compras**: List<Compra>, consultable

El tipo Producto está implementado mediante un record, y los tipos Compra y Pedido mediante una clase.

1. Sea m un Map que relaciona cada fecha con la lista de compras realizadas en esa fecha. Escriba las expresiones para realizar las siguientes operaciones sobre el Map m:
   a) Añadir una nueva compra c al final de la lista de compras realizadas en la fecha f
   b) Obtener el conjunto de fechas del Map m
   c) Obtener el número de fechas diferentes en las que se han realizado compras
   d) Obtener un valor true si el Map m contiene la fecha f
   e) Obtener el conjunto de parejas del Map m

   a) **`m.get(f).add(c)`**
   b) **`m.keySet()`**
   c) **`m.size()`**
   d) **`m.containsKey(f)`**
   e) **`m.entrySet()`**

2. Sea m un Map que relaciona cada producto con la lista de compras que contienen ese producto. Complete el bucle para que muestre en pantalla el número de compras que contienen cada producto, con el formato siguiente:

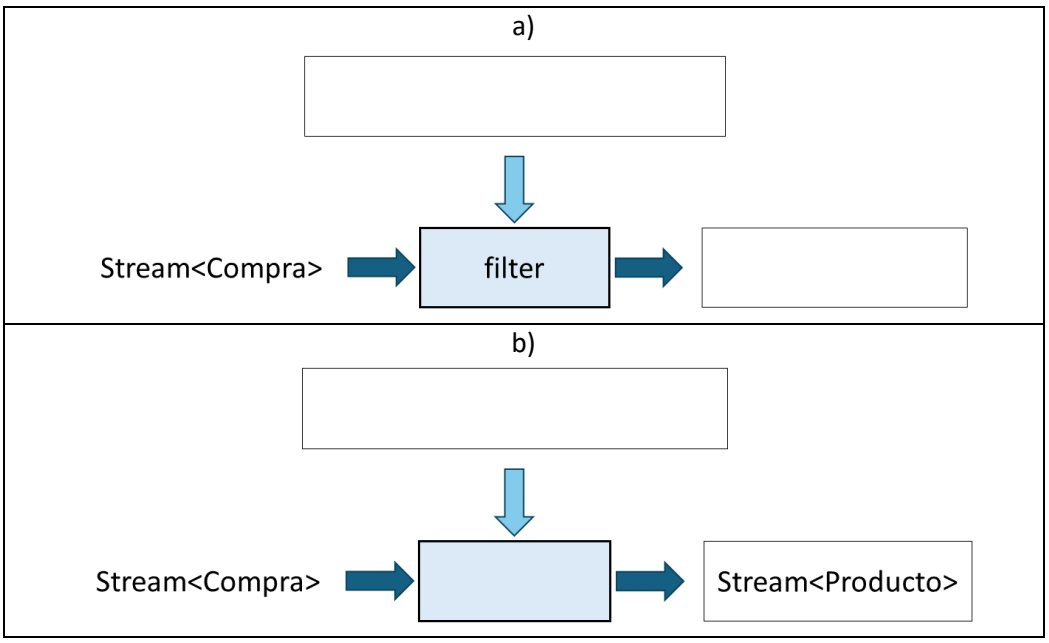
```
El producto Rotulador aparece en 5 compra(s).
El producto Grapadora aparece en 1 compra(s).
```

```
for (Map.Entry<Producto, List<Compra>> e: m.entrySet()) {

}
```

```
for (Map.Entry<Producto, List<Compra>> e: m.entrySet()) {
      System.out.println("El producto " + e.getKey().nombre() +
            " aparece en " + e.getValue().size() + " compra(s).");
}
```

3. Complete los siguientes gráficos añadiendo lo que proceda.



a) `Predicate<Compra>, Stream<Compra>`
b) `Function<Compra, Producto>`

4. Cree las siguientes variables de tipo Predicate:
   a) Un predicado p1 que tome valor true si una compra tiene más de n unidades del producto
   b) Un predicado p2 que tome valor true si una compra se ha realizado con tarjeta
   c) Un predicado p3 que tome valor true si una compra tiene más de n unidades del producto y no se ha realizado con tarjeta. Reutilice los predicados anteriores

   a) `Predicate<Compra> p1 = c -> c.getUnidades() > n;`
   b) `Predicate<Compra> p2 = c -> c.getPago().equals(MedioPago.TARJETA);`
   c) `Predicate<Compra> p3 = p1.and(p2.negate());`

5. Cree las siguientes variables de tipo Function:
   a) Una función f1 que obtenga la fecha en que se realiza una compra
   b) Una función f2 que obtenga el año en que se realiza una compra. Reutilice la función f1
   c) Una función f3 que obtenga el precio del producto de una compra
   d) Una función f4 que obtenga el número de compras de un pedido

   a) `Function<Compra, LocalDate> f1 = Compra::getFecha;`
   b) `Function<Compra, Integer> f2 = f1.andThen(LocalDate::getYear);`
   c) `Function<Compra, Double> f3 = c -> c.getProducto().precio();`
   d) `Function<Pedido, Integer> f4 = p -> p.getCompras().size();`

6. Cree las siguientes variables de tipo Comparator. Puede reutilizar las variables ya creadas si lo desea. Si no se indica otra cosa, se sobreentiende que se usa el orden natural de la propiedad.
   a) Un comparador cmp1 que compare productos por precio, y a igualdad de precio por nombre
   b) Un comparador cmp2 que compare compras por unidades de mayor a menor
   c) Un comparador cmp3 que compare compras por fecha, y a igualdad de fecha por unidades de mayor a menor

   a)
   ```
   Comparator<Producto> cmp1 = Comparator.comparing(Producto::precio)
         .thenComparing(Comparator.comparing(Producto::nombre));
   ```
   b)
   ```
   Comparator<Compra> cmp2 = Comparator.comparing(Compra::getUnidades)
         .reversed();
   ```

c)
```
Comparator<Compra> cmp3 = Comparator.comparing(Compra::getFecha)
      .thenComparing(cmp2);
Comparator<Compra> cmp3 =
      Comparator.comparing(Compra::getFecha).thenComparing(
         Comparator.comparing(Compra::getUnidades).reversed());
```

7. Dado un Stream<Compra>, queremos obtener una lista con los nombres de los productos de las n primeras compras pagadas en EFECTIVO, ordenadas de la más antigua a la más reciente y sin repetir ninguno. Indique la secuencia de métodos que hay que aplicar en el orden correcto para obtener el resultado deseado. Debe indicar el nombre de cada método y el tipo de su parámetro, si lo tiene.

filter con Predicate<Compra>, sorted con Comparator<Compra>, map con Function<Compra, String>, distinct, limit con Integer, collect con Collector<String, List<String>>

8. Complete el método siguiente para que devuelva el número medio de unidades de las compras realizadas en el mes dado como parámetro y pagadas con el medio de pago dado como parámetro.

```
public Double mediaUnidadesComprasMes(MedioPago medio, Month mes) {
      return compras.stream()



}

public Double mediaUnidadesComprasMes(MedioPago medio, Month mes) {
      return compras.stream()
            .filter(a -> a.getPago().equals(medio)
                  && a.getFecha().getMonth().equals(mes))
            .mapToInt(Compra::getUnidades)
            .average()
            .getAsDouble();
}
```

9. Complete el método siguiente para que construya un Map que relacione cada fecha con el número de compras realizadas en esa fecha, a partir de un Map que relaciona cada fecha con la lista de compras de esa fecha.

```
public Map<LocalDate, Integer> numeroComprasPorFecha() {
      Map<LocalDate, Set<Compra>> m = comprasPorFecha();
      return m.keySet().stream()



}

public Map<LocalDate, Integer> numeroComprasPorFecha() {
      Map<LocalDate, Set<Compra>> m = comprasPorFecha();
      return m.keySet().stream()
                  .collect(Collectors.toMap(
                              f -> f,
                              f -> m.get(f).size()));
}
```

10. El siguiente método tiene como objetivo obtener un Map que relaciona cada fecha con la lista de horas de las compras realizadas en esa fecha. Este método puede contener errores. Para cada posible error, indique su causa y explique cómo lo corregiría.

```
public Map<LocalDate, List<LocalTime>> horasPorFecha() {
	return compras.stream()
		.collect(Collectors.groupingBy(
				Compra::getFecha)
		);
}
```

Tal como está escrito el método, los valores son colecciones de compras, no de horas de las compras. Para que se acumulen las horas, hay que añadir al groupingBy un segundo parámetro que transforme la compra en su hora, usando un mapping:

```
public Map<LocalDate, List<LocalTime>> horasPorFecha() {
	return compras.stream()
		.collect(Collectors.groupingBy(
				Compra::getFecha,
				Collectors.mapping(
					Compra::getHora,
					Collectors.toList()
					)
				)
		);
}
```