Ejercicio 1

Se desea crear una colección que ofrezca el método **`toMap`**. Este método recibe una función que indica cómo se debe construir la clave que acompañará a cada valor de la colección en un nuevo mapa que retorna el método. En caso de que se obtengan dos o más claves iguales, en el mapa no quedará una cantidad de entradas igual a los elementos de la colección y no se debe validar.

Implementar <u>todo</u> lo necesario para que, con el siguiente programa de prueba, se obtenga la salida indicada en los comentarios.

```
public class SimpleListTester {

   public static void main(String[] args) {
       SimpleList<Integer> simpleList = new SimpleArrayList<>();

       simpleList.add(1);
       simpleList.add(3);
       simpleList.add(5);
       simpleList.add(7);

       System.out.println(simpleList.size()); // 4

       System.out.println(simpleList.contains(0)); // false

       Map<Integer, Integer> first = simpleList.toMap(element -> element * 2);

       System.out.println(first.get(2)); // 1
       System.out.println(first.get(6)); // 3
       System.out.println(first.get(10)); // 5
       System.out.println(first.get(14)); // 7

       Map<String, Integer> second = simpleList.toMap(element ->
String.format("<%d>", element));

       System.out.println(second.get("<1>")); // 1
       System.out.println(second.get("<3>")); // 3
       System.out.println(second.get("<5>")); // 5
       System.out.println(second.get("<7>")); // 7

       Map<Integer, Integer> third = simpleList.toMap(element -> element % 2);

       System.out.println(third.get(0)); // null
       System.out.println(third.get(1)); // Podría ser 1, 3, 5 ó 7
   }

}
```

Ejercicio 2

Se desea contar con la clase `ReversedListImpl` para contar con una lista cuyo iterador retorne los elementos en el orden inverso al que se encuentran almacenados. Asimismo, se debe ofrecer un método alternativo para iterar por los elementos en el orden habitual.

Implementar todo lo necesario para que, con el siguiente programa,

```
import java.util.Iterator;

public class ReversedListTester {

   public static void main(String[] args) {
       ReversedList<Integer> list = new ReversedListImpl<>();
       list.add(1);
       list.add(2);
       list.add(3);
       list.add(4);
       Iterator<Integer> defaultIterator = list.defaultIterator();
       while(defaultIterator.hasNext()) {
           System.out.print(defaultIterator.next() + " ");
       }
       System.out.println();
       Iterator<Integer> iterator = list.iterator();
       while(iterator.hasNext()) {
           System.out.print(iterator.next() + " ");
       }
   }

}
```

se obtenga la siguiente salida:

```
1 2 3 4
4 3 2 1
```

Ejercicio 3

La clase `FilteredKeyMapIterator` modela un **iterador de mapas**. Permite iterar sobre una instancia de `Map` (que recibe en su constructor), accediendo a cada una de las claves del mismo **que cumplan un criterio** (que recibe en su constructor) con el método `next()` y al valor asociado a esa clave en el mapa mediante el método `getValue()`.

Aclaración: no importa el orden en que se recorren las claves del mapa.

Para modelar el criterio que deben cumplir las claves, se utiliza la interfaz funcional `Predicate<T>` presente en la biblioteca de Java. El cuerpo relevante de la misma es el siguiente:

```
package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Predicate<T> {

   /**
    * Evaluates this predicate on the given argument.
    *
    * @param t the input argument
    * @return {@code true} if the input argument matches the predicate,
    * otherwise {@code false}
    */
   boolean test(T t);

   ...

}
```

Implementar todo lo necesario para que, con el siguiente programa

```
import java.util.HashMap;
import java.util.Map;

public class FilteredKeyMapIteratorTester {

   public static void main(String[] args) {
       Map<Integer, String> map = new HashMap<>();
       map.put(1, "One");
       map.put(2, "Two");
       map.put(3, "Three");
       FilteredKeyMapIterator<Integer, String> mapIterator
                                           = new FilteredKeyMapIterator<>(map, k -> k %
2 == 1);


       while (mapIterator.hasNext()) {
           System.out.println("Key: " + mapIterator.next());
           System.out.println("Value: " + mapIterator.getValue());
       }
       mapIterator = new FilteredKeyMapIterator<>(map, k -> k != null);
       try {
           mapIterator.getValue();
       } catch (Exception ex) {
```

```
            System.out.println(ex.getClass());
        }
        System.out.println(mapIterator.next());
        System.out.println(mapIterator.getValue());
        System.out.println(mapIterator.getValue());
        while (mapIterator.hasNext()) {
            mapIterator.next();
        }
        mapIterator.next();
    }

}
```

se obtenga la siguiente salida:

```
Key: 1
Value: One
Key: 3
Value: Three
class java.util.NoSuchElementException
1
One
One
Exception in thread "main" java.util.NoSuchElementException
...
```

Ejercicio 4

Se desea implementar un sistema para cargar los pasajeros de un vuelo para obtener luego el **orden de llamada para el embarque**. Un pasajero cuenta con su **nombre** (que no se repite en el vuelo), una **fila de asiento** y una **categoría** de membresía de la aerolínea.

Existen dos formas de embarque:

- **Por fila de asiento:** Donde embarcan primero los pasajeros de las filas menores (orden ascendente en función de la fila del asiento).
- **Por categoría del pasajero:** Donde embarcan primero los pasajeros de categorías superiores (orden ascendente en función de la categoría del pasajero donde la categoría superior `EMERALD` tiene el valor más bajo y la categoría inferior `ECONOMY` tiene el valor más alto).

Implementar todo lo necesario para que, con el siguiente programa de prueba

```
public class BoardingFlightTester {

   public static void main(String[] args) {
       PassengerCategory[] categories = PassengerCategory.values();
       System.out.println(Arrays.toString(categories));
       Arrays.sort(categories);
       System.out.println(Arrays.toString(categories));
       System.out.println("-----");
       BoardingFlight rowBoardingFlight = new RowBoardingFlight();
       rowBoardingFlight.addForBoarding("Passenger 3", 10,
PassengerCategory.ECONOMY);
       rowBoardingFlight.addForBoarding("Passenger 2", 5, PassengerCategory.RUBY);
       rowBoardingFlight.addForBoarding("Passenger 1", 15,
PassengerCategory.ECONOMY);
       rowBoardingFlight.addForBoarding("Passenger 4", 5, PassengerCategory.EMERALD);
       Iterator<String> rowIterator = rowBoardingFlight.boardingCallIterator();
       while(rowIterator.hasNext()) {
           System.out.println(rowIterator.next());
       }
       try {
           rowIterator.next();
       } catch (NoSuchElementException ex) {
           System.out.println("No more elements");
       }
       System.out.println("-----");
       BoardingFlight categoryBoardingFlight = new CategoryBoardingFlight();
       categoryBoardingFlight.addForBoarding("Passenger 3", 10,
PassengerCategory.ECONOMY);
       categoryBoardingFlight.addForBoarding("Passenger 2", 5,
PassengerCategory.RUBY);
       categoryBoardingFlight.addForBoarding("Passenger 1", 15,
PassengerCategory.ECONOMY);
       categoryBoardingFlight.addForBoarding("Passenger 4", 5,
PassengerCategory.EMERALD);
       Iterator<String> categoryIterator =
categoryBoardingFlight.boardingCallIterator();
       while(categoryIterator.hasNext()) {
           System.out.println(categoryIterator.next());
       }
       try {
           categoryIterator.next();
       } catch (NoSuchElementException ex) {
```

```
            System.out.println("No more elements");
        }
    }

}
```

se obtenga la siguiente salida

```
[EMERALD, SAPPHIRE, RUBY, ECONOMY]
[EMERALD, SAPPHIRE, RUBY, ECONOMY]
-----
Passenger 2
Passenger 4
Passenger 3
Passenger 1
No more elements
-----
Passenger 4
Passenger 2
Passenger 1
Passenger 3
No more elements
```

Ejercicio 5

Se cuenta con la interfaz `DoubleKeyMap` que modela un mapa con **claves compuestas**.

La clave compuesta, que es única, está dada por dos valores, no necesariamente del mismo tipo. Cada clave compuesta tiene sólo un valor asociado.

```
public interface DoubleKeyMap<K1,K2,V> {

   int size();

   boolean isEmpty();

   boolean containsKey(K1 firstKey, K2 secondKey);

   boolean containsValue(V value);

   V get(K1 firstKey, K2 secondKey);

   void put(K1 firstKey, K2 secondKey, V value);

}
```

<u>Implementar todo lo necesario</u> para que, con el siguiente programa se obtenga la salida indicada en los comentarios:

```
public class DoubleKeyMapTester {

    public static void main(String[] args) {
        DoubleKeyMap<String, String, Integer> doubleKeyMap = new
DoubleKeyHashMap<>();
        doubleKeyMap.put("Juan", "Perez", 49);
        System.out.println(doubleKeyMap.size()); // 1
        doubleKeyMap.put("Lucas", "Gomez", 37);
        doubleKeyMap.put("Lucas", "Lopez", 26);
        doubleKeyMap.put("Juan", "Lopez", 55);
        System.out.println(doubleKeyMap.size()); // 4
        System.out.println(doubleKeyMap.isEmpty()); // false
        System.out.println(doubleKeyMap.containsKey("Juan", "Ramirez")); // false
        System.out.println(doubleKeyMap.containsKey("Juan", "Gomez")); // false
        System.out.println(doubleKeyMap.containsKey("Lucas", "Gomez")); // true
        System.out.println(doubleKeyMap.get("Lucas", "Gomez")); // 37
        System.out.println(doubleKeyMap.get("Lucas", "Lopez")); // 26
        System.out.println(doubleKeyMap.containsValue(26)); // true
        doubleKeyMap.put("Lucas", "Lopez", 27);
        System.out.println(doubleKeyMap.size()); // 4
        System.out.println(doubleKeyMap.containsValue(26)); // false
        System.out.println(doubleKeyMap.get("Lucas", "Lopez")); // 27
        System.out.println(doubleKeyMap.containsKey("Gomez", "Lucas")); // false
        System.out.println(doubleKeyMap.containsValue(10)); // false
    }

}
```

Ejercicio 6

Se cuenta con la interfaz **`Cache`** que modela una forma de consultar y establecer valores asociados a claves de forma similar a un mapa, pero informando qué usuario realizó determinada operación en la fecha indicada. A motivos de simplificar su uso, ambos identificadores usuario y fecha son instancias de `String`.

```
public interface Cache<K, V> extends Map<K,V> {

   void put(String user, String date, K key, V value);

   V get(String user, String date, K key);

}
```

Además ya se cuenta con la implementación **`BaseCache`** que se muestra a continuación:

```
public class BaseCache<K, V> extends HashMap<K, V> implements Cache<K, V> {

   @Override
   public void put(String user, String date, K key, V value) {
       System.out.println(user + " put value " + value + " for key " + key + " on " +
date);
       super.put(key, value);
   }

   @Override
   public V get(String user, String date, K key) {
       V value = super.get(key);
       System.out.println(user + " retrieved value " + value + " for key " + key + "
on " + date);
       return value;
   }

}
```

Se desea contar con una nueva implementación de la interfaz `Cache` que permita **limitar el número de operaciones de lectura y escritura que cada usuario puede realizar por día** según las siguientes dos políticas:

- **Política ilimitada:** no hay ninguna limitación diaria para ambas operaciones.
- **Política limitada:** Se permite un máximo de 2 (dos) lecturas y 1 (una) escritura, por día y por usuario.

Superado el límite por un usuario en un determinado día, si el usuario intenta realizar nuevamente la misma operación en el mismo día se debe arrojar la excepción `RateLimitedException`. Una vez llegado al límite, debe ser posible realizar operaciones en otras fechas y/o para otros usuarios y/o la otra operación para el mismo usuario en el mismo día (si es que aún le queda cuota).

Implementar todo lo necesario para que, con el siguiente programa de prueba

```
public class RateLimitedCacheTester {

   private static final String USER1 = "Alice";
```

```
    private static final String USER2 = "Bob";
    private static final String DATE1 = "11/11/2019";
    private static final String DATE2 = "12/11/2019";

    public static void main(String[] args) {
        RateLimitedCache<Integer, String> cache = new RateLimitedCache<>();

        System.out.println(cache.size());

        cache.register(USER1, QuotaType.LIMITED);

        System.out.println("Testing puts with a limited quota (maximum 1 per date)");
        cache.put(USER1, DATE1, 1, "1");
        try {
            cache.put(USER1, DATE1, 1, "2");
        } catch (RateLimitedException e) {
            System.out.println("Cannot put 1->\"2\" on  " + DATE1);
        }
        System.out.println("-----");
        System.out.println(cache.get(USER1, DATE1, 1));
        System.out.println("-----");

        System.out.println("Testing reads with a limited quota (maximum 2 per date)");
        cache.put(USER1, DATE2, 2, "2");
        System.out.println("-----");
        System.out.println(cache.get(USER1, DATE2, 1));
        System.out.println("-----");
        System.out.println(cache.get(USER1, DATE2, 2));
        System.out.println("-----");
        try {
            System.out.println(cache.get(USER1, DATE2, 3));
        } catch (RateLimitedException e) {
            System.out.println("Cannot read 1 on " + DATE2);
        }

        cache.register(USER2, QuotaType.UNLIMITED);

        System.out.println("Testing puts with a unlimited");
        cache.put(USER2, DATE1, 3, "3");
        cache.put(USER2, DATE1, 3, "4");

        System.out.println("Testing reads with a unlimited");
        System.out.println("-----");
        System.out.println(cache.get(USER2, DATE1, 3));
        System.out.println("-----");
        System.out.println(cache.get(USER2, DATE1, 3));
        System.out.println("-----");
        System.out.println(cache.get(USER2, DATE1, 3));
        System.out.println("-----");

        System.out.println(cache.size());
    }

}
```

se obtenga la siguiente salida

```
0
Testing puts with a limited quota (maximum 1 per date)
Alice put value 1 for key 1 on 11/11/2019
Cannot put 1->"2" on  11/11/2019
-----
Alice retrieved value 1 for key 1 on 11/11/2019
1
-----
Testing reads with a limited quota (maximum 2 per date)
Alice put value 2 for key 2 on 12/11/2019
-----
Alice retrieved value 1 for key 1 on 12/11/2019
1
-----
Alice retrieved value 2 for key 2 on 12/11/2019
2
-----
Cannot read 1 on 12/11/2019
Testing puts with a unlimited
Bob put value 3 for key 3 on 11/11/2019
Bob put value 4 for key 3 on 11/11/2019
Testing reads with a unlimited
-----
Bob retrieved value 4 for key 3 on 11/11/2019
4
-----
Bob retrieved value 4 for key 3 on 11/11/2019
4
-----
Bob retrieved value 4 for key 3 on 11/11/2019
4
-----
3
```

Ejercicio 7

Se desea implementar un conjunto de clases que permitan administrar el préstamo de libros de la **biblioteca** de una universidad a sus alumnos y docentes.

Ya se cuenta con la clase `BookInfo` implementada, la cual relaciona el nombre del libro con el stock actual.

```
public class BookInfo {

   private String name;
   private int stock;

   public BookInfo(String name, int stock) {
       this.name = name;
       this.stock = stock;
   }

   public void borrowBook() {
       if (stock == 0) {
           throw new IllegalStateException();
       }
       stock--;
   }

   public void returnBook() {
       stock++;
   }

}
```

Las reglas de la biblioteca son las siguientes:

- **Se puede prestar únicamente un libro por persona.**
- Si es **alumno**, el préstamo del libro es por 2 días.
- Si es **profesor**, el tiempo del préstamo del libro dependerá de su cargo:
  - Responsable: 12 días
  - Jefe de Trabajos Prácticos: 11 días
  - Ayudante: 10 días.

Implementar todo lo necesario y completar los ..... para que, con el siguiente programa de prueba:

```
import java.time.LocalDate;

public class LibraryTester {

   public static void main(String[] args) {
       // Ejemplo de uso para determinar si una fecha está después que otra fecha
       System.out.println(LocalDate.of(2018,12,3).isAfter(LocalDate.of(2018,12,1)));

       Library library = new Library().addBook("Book 1", 1)
               .addBook("Book 2", 1)
               .addBook("Book 3", 3);

       // El estudiante Student 1 pide el libro Book 1
```

```
        ..................... s1 = .....................................;
        library.borrowBook(s1, "Book 1", LocalDate.of(2018, 12, 1));
        // El estudiante Student 2 pide el libro Book 3

        ..................... s2 = .....................................;
        library.borrowBook(s2, "Book 3", LocalDate.of(2018, 12, 1));
        // El estudiante Student 3 pide el libro Book 3

        ..................... s3 = .....................................;
        library.borrowBook(s3, "Book 3", LocalDate.of(2018, 12, 2));

        // Lista los préstamos vencidos para la fecha recibida
        library.printDueLoansBooks(LocalDate.of(2018, 12, 4));
        library.returnBook(s1); // El estudiante Student 1 devuelve el libro que pidió
        library.returnBook(s2); // El estudiante Student 2 devuelve el libro que pidió
        library.returnBook(s3); // El estudiante Student 3 devuelve el libro que pidió

        // El profesor Professor 1 es Responsable y pide el libro Book 1

        ..................... p1 = .....................................;
        library.borrowBook(p1, "Book 1", LocalDate.of(2018, 12, 4));
        // El profesor Professor 2 es Jefe de Trabajos Prácticos y pide el libro Book
3

        ..................... p2 = .....................................;
        library.borrowBook(p2, "Book 3", LocalDate.of(2018, 12, 4));
        // El profesor Professor 3 es Ayudante y pide el libro Book 3

        ..................... p3 = .....................................;
        library.borrowBook(p3, "Book 3", LocalDate.of(2018, 12, 4));

        library.printDueLoansBooks(LocalDate.of(2018, 12, 10));
        library.printDueLoansBooks(LocalDate.of(2018, 12, 15));
        library.printDueLoansBooks(LocalDate.of(2018, 12, 16));
        library.printDueLoansBooks(LocalDate.of(2018, 12, 17));

        try {
            library.borrowBook(s1, "Other Book", LocalDate.of(2018,12,3));
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }

}
```

se obtenga la siguiente salida:

```
true
Due loans at 2018-12-04
Student 2 (Book 3 - 2018-12-01)
Student 1 (Book 1 - 2018-12-01)

Due loans at 2018-12-10

Due loans at 2018-12-15
Professor 3 (Book 3 - 2018-12-04)
```

```
Due loans at 2018-12-16
Professor 3 (Book 3 - 2018-12-04)
Professor 2 (Book 3 - 2018-12-04)

Due loans at 2018-12-17
Professor 3 (Book 3 - 2018-12-04)
Professor 2 (Book 3 - 2018-12-04)
Professor 1 (Book 1 - 2018-12-04)
Book not found.
```

Ejercicio 8

Una aplicación web destinada a empleados de empresas tiene ya definidas las clases `Menu` y `MenuItem`. Un `MenuItem` representa una opción de menú (por ejemplo "Guardar") que cuenta con un menú padre de tipo `Menu` (por ejemplo "Archivo").

Por defecto, el acceso a los menúes (padres e hijos) es restringido. Para poder acceder a uno de ellos se requiere una autorización. Se desea entonces contar con una forma de otorgar permisos para que los empleados puedan acceder a los ítems para los cuales fueron autorizados.

Un menú puede ser accedido por una o más personas y/o una o más empresas (en este caso, todos los empleados de la empresa pueden acceder al menú). Además un empleado puede pertenecer a una o más empresas.

Para simplificar el mantenimiento, se le puede otorgar el permiso a una persona o empresa directamente a un `Menu`, y en ese caso podrán acceder a todos los `MenuItem` correspondientes.

Implementar todo lo necesario para que, con el siguiente programa de prueba,

```
menu1 = Menu.new('File')
menu2 = MenuItem.new('New', menu1)
menu3 = MenuItem.new('Close', menu1)
menu4 = Menu.new('Help')
menu5 = MenuItem.new('About Us', menu4)
menu6 = MenuItem.new('Find...', menu4)

company1 = Company.new('ACME')
company2 = Company.new('Warner')
employee1 = Employee.new('James', [company1])
employee2 = Employee.new('Annie', [company1, company2])

menu1.authorize(employee1)
menu2.authorize(employee1)
menu5.authorize(employee1)
menu3.authorize(employee2)
menu4.authorize(company2)

[menu1, menu2, menu3, menu4, menu5, menu6].each { |menu| puts menu.access?(employee1)
}

puts '##########'

[menu1, menu2, menu3, menu4, menu5, menu6].each { |menu| puts menu.access?(employee2)
}
```

se obtenga la siguiente salida:

```
true
true
true
false
true
false
##########
false
false
true
true
```

```
true
true
```

Ejercicio 9

Se desea modelar un conjunto de clases para un **sistema de reservación de cocheras de un estacionamiento**. El **estacionamiento** (***parking lot***) cuenta con **pisos**. Cada **piso** es identificado por una letra. No pueden haber dos pisos con la misma identificación. Cada **cochera** (*parking space*) pertenece a un piso y es identificada por un número. No pueden haber dos cocheras con la misma identificación en el mismo piso.

Al crear un estacionamiento éste se crea sin pisos (y por ende sin cocheras). Es responsabilidad del usuario dar de alta cada una de las cocheras de cada uno de los pisos con el método `add_parking_space`. Dando de alta una cochera se da de alta también al piso correspondiente si es que éste no existía.

Cada cochera puede estar o no reservada. El estacionamiento debe ofrecer métodos para reservar o cancelar una reserva de una cochera (métodos `park` y `unpark` respectivamente).

Ya cuenta con la implementación del método `information` de la clase `ParkingLot` que le indica la colección que deberá utilizar para almacenar la información del estacionamiento. Este método muestra el estado actual de todas las cocheras del estacionamiento.

```
class ParkingLot
```

```
...
```

```
 def information
   s = "Parking Lot #{@name}\n"
   @parking_spaces_by_level.keys.sort.each do |level|
     s += "Level #{level}\n"
     @parking_spaces_by_level[level].values.sort.each do |parking_space|
       s += "#{parking_space}\n"
     end
   end
   s
 end

end
```

Implementar todo lo necesario, agregando métodos a `ParkingLot` y/o creando clases nuevas para que, con el siguiente programa, se obtenga la salida indicada.

```
parking_lot =
ParkingLot.new('EstacionARTE')
parking_lot.add_parking_space('A', 1030)
parking_lot.add_parking_space('A', 1000)
parking_lot.add_parking_space('A', 1001)
parking_lot.add_parking_space('B', 1001)
puts parking_lot.information
puts '##########'
begin
 parking_lot.park('Z',1001)
rescue RuntimeError => e
 puts e.message
end
puts '##########'
```

```
Parking Lot EstacionARTE
Level A
#1000: Available
#1001: Available
#1030: Available
Level B
#1001: Available
##########
Invalid Level
##########
Invalid Parking Space
##########
Cannot Park Reserved Parking Space
##########
```

```ruby
begin
 parking_lot.unpark('A',9999)
rescue RuntimeError => e
 puts e.message
end
puts '##########'
parking_lot.park('A',1001)
parking_lot.unpark('A',1001)
parking_lot.park('A',1001)
begin
 parking_lot.park('A',1001)
rescue RuntimeError => e
 puts e.message
end
puts '##########'
begin
 parking_lot.unpark('B',1001)
rescue RuntimeError => e
 puts e.message
end
puts '##########'
parking_lot.park('B',1001)
puts parking_lot.information
```

```
Cannot Unpark Available Parking Space
##########
Parking Lot EstacionARTE
Level A
#1000: Available
#1001: Reserved
#1030: Available
Level B
#1001: Reserved
```

Ejercicio 10

Se desea modelar un sistema de compra de pasajes aéreos. Dicho sistema debe consultar la información de los vuelos utilizando las clases **`FlightCatalog`** y **`Flight`** que se muestran a continuación:

```ruby
class FlightCatalog

 def initialize
   @flights = Hash.new
 end

 def add_flight(flight)
   @flights[flight.code] = flight
 end

 def get_flight(flight_code)
   @flights[flight_code]
 end

end
```

```ruby
class Flight

 attr_reader :code, :airline, :price, :miles

 def initialize(code, airline, price, miles)
   @code = code
   @airline = airline
   @price = price
   @miles = miles
 end

end
```

Se desea contar con la clase **`FlightOperator`** que modela a un operador de pasajes aéreos. Dicha clase permite realizar la compra de un pasaje acumulando las millas que este otorga.

Existen tres categorías de **miembros** del programa de millas:

- **Standard**: Acumula las millas del vuelo y permite acumular hasta 1000 millas.
- **Gold**: Acumula un **10%** más de millas que la categoría Standard (multiplicador 1.10) y permite acumular hasta 2000 millas.
- **Platinum**: Acumula un **25%** más de millas que la categoría Standard (multiplicador 1.25) y no tiene límite de acumulación de millas.

Así cada miembro pertenece a una de estas tres categorías y la misma sirve para todas las aerolíneas.

El método `buy_flight` retorna el **precio del vuelo**, el cual puede ser:

- **El precio original** si no cuenta con la totalidad de millas necesarias para canjearlo. Se deberán además sumarle las millas correspondientes al miembro para dicha aerolínea, con los criterios de su categoría.
- **Cero**, si cuenta con la totalidad de millas necesarias para canjearlo. Se deberán además restarle las millas al miembro para dicha aerolínea.

La clase `FlightOperator` ofrece además el método `miles_status` que dado un miembro y una aerolínea, lista el millaje acumulado en dicha aerolínea.

Implementar todo lo necesario y completar los ….. para que con el siguiente programa de prueba

```
flight_catalog = FlightCatalog.new

flight_catalog.add_flight(Flight.new('LA1','LATAM',200,600))
flight_catalog.add_flight(Flight.new('LA2','LATAM',400,3000))
flight_catalog.add_flight(Flight.new('LA3','LATAM',40,200))
flight_catalog.add_flight(Flight.new('UA1','United',3500,5000))

flight_operator = FlightOperator.new(flight_catalog)

juan = Member.new('Juan', .....................) # Juan tiene Categoría Gold

puts flight_operator.miles_status(juan, 'United') # 0
puts flight_operator.miles_status(juan, 'LATAM') # 0

puts flight_operator.buy_flight('UA1', juan) # 3500

puts flight_operator.miles_status(juan, 'United') # 2000 por el tope de categoría
puts flight_operator.miles_status(juan, 'LATAM') # 0

malena = Member.new('Malena', .....................) # Malena tiene Categoría Platinum

puts flight_operator.miles_status(malena, 'United') # No registra millas
puts flight_operator.miles_status(malena, 'LATAM') # No registra millas

puts flight_operator.buy_flight('UA1', malena) # 3500
puts flight_operator.buy_flight('LA1', malena) # 200

puts flight_operator.miles_status(malena, 'United') # 6250.0 = 5000 * 1.25
puts flight_operator.miles_status(malena, 'LATAM') # 750.0 = 600 * 1.25

puts flight_operator.buy_flight('LA2', malena) # 400

puts flight_operator.miles_status(malena, 'United') # 6250.0
puts flight_operator.miles_status(malena, 'LATAM') # 4500.0 = 750.0 + 3000 * 1.25

puts flight_operator.buy_flight('LA3', malena) # 0 porque tenía 200 millas

puts flight_operator.miles_status(malena, 'United')  # 6250.0
puts flight_operator.miles_status(malena, 'LATAM') # 4300.0 = 4500.0 - 200.0
```

se obtenga la siguiente salida:

```
Juan no registra millas acumuladas en United
Juan no registra millas acumuladas en LATAM
3500
Millas de Juan en United: 2000
Millas de Juan en LATAM: 0
Malena no registra millas acumuladas en United
Malena no registra millas acumuladas en LATAM
3500
200
Millas de Malena en United: 6250.0
Millas de Malena en LATAM: 750.0
```

```
400
Millas de Malena en United: 6250.0
Millas de Malena en LATAM: 4500.0
0
Millas de Malena en United: 6250.0
Millas de Malena en LATAM: 4300.0
```