ADSOF: Final exam, ordinary period, non-cont. eval. (v1) - 28/05/2020

Exercise 1 (3 points)

We need to develop an application for the management of news. A news item is composed of a title, content, publication date and time, and has associated a topic. The application handles a limited set of topics, each of them having a name and a description. The topics are organized as a taxonomy; that is, a topic may have certain subtopics (children topics) and/or a supertopic (parent topic).

The content of a news item consists of a list of elements, which are enumerated and could be a text, an image or a video. The texts are in HTML format, and the images and videos have associated an URL.

A user of the application has assigned a unique numeric identifier, and can request the automatic reception of breaking news. For such purpose, the user can state the news topics she is interested in, and the transmitter types for which she will receive the news: email (given her email address), SMS (given her phone number) or Twitter (given her username).

A user can modify her registration, adding or removing topics or transmitters.

Once a breaking news item is received, the application sends it to those users who are interested in the topic of such item. The items are sent through all the transmitters recorded by each user.

You are requested to:

a. Draw the UML class diagram with the design of the described application. In the diagram, mark the **design patterns** used, including the roles of the elements in such patterns. (**2 points)**

   In the diagram, do not consider constructors, getters and setters, but include those methods needed to satisfy the constraints described in the exercise.

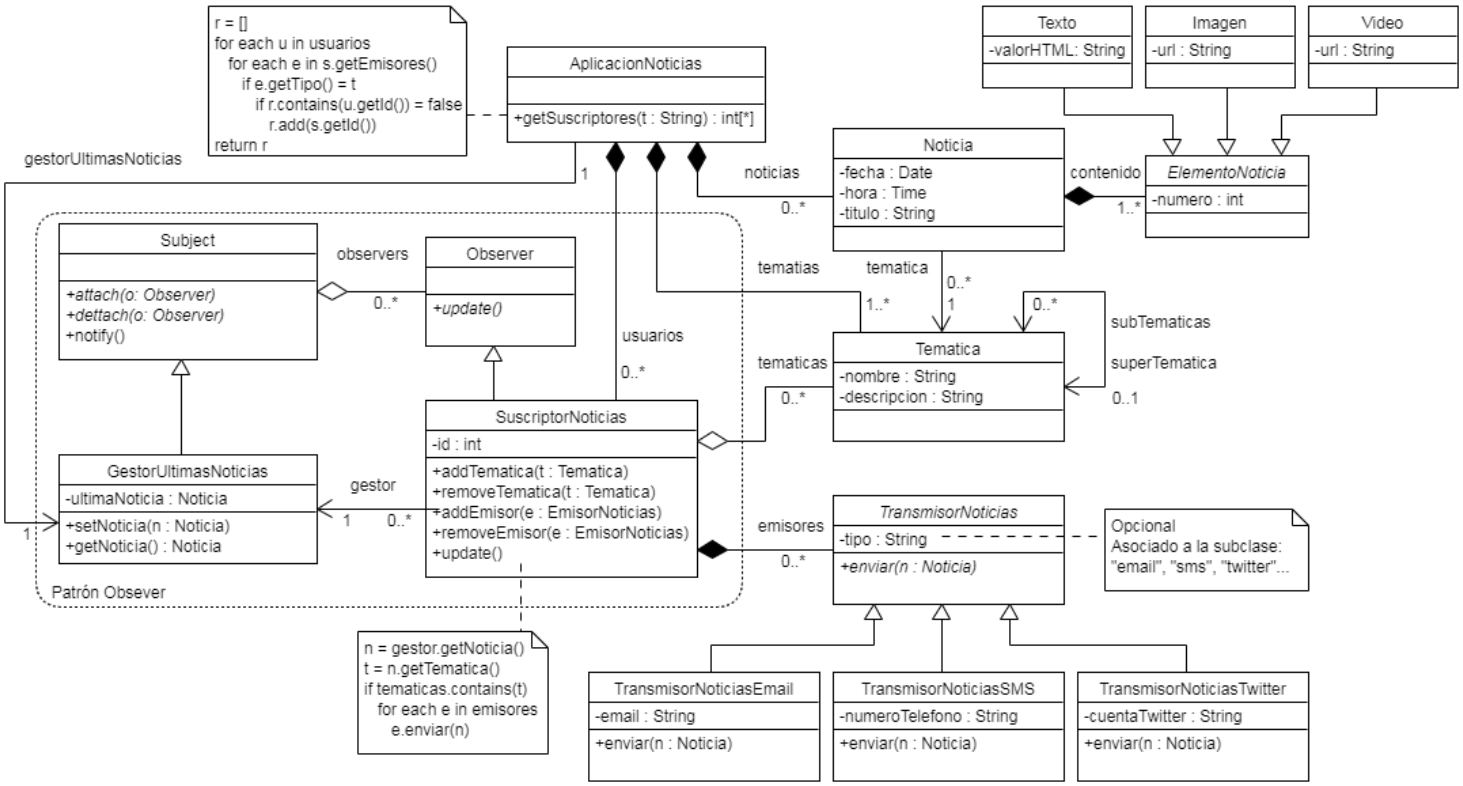
b. Include in the diagram and provide the pseudocode of the following methods: **(1 point)**

   b.1. Obtaining the identifiers of the users who receive news items through a given transmitter.

   b.2. Sending a breaking news item to the corresponding users.

Submit your solution in a single file, with PDF, PNG or JPG format, and named as "**surnames_name.pdf**", where "surnames" and "names" are your surnames and name, respectively; for instance, GarcíaPérez_JoseMaría.pdf. Open the file before submission to check it is correct.

Solution:



(evaluation criteria on the following page)

Evaluation criteria

CE = Continuous evaluation; NCE = Non-continuous evaluation

- General elements of the diagram (0.25 points for CE; 0.2 points for NCE).
  - Correct use of the UML standard: symbols for classes, enumerations, interfaces, comments and relations (aggregations, associations); format of attribute names and types (including arrays []) and method prototypes.
  - Correct use of names for classes and attributes: understandable meanings, singular vs. plural, uppercase vs. lowercase.
- Relations (0.25 points for CE; 0.2 for NCE).
  - Existence/lack of relevant relations.
  - Existence of aggregation and composition relations. Certain solutions different to the given solution have been considered as corrects, such as an associative relation with cardinalities in the Observer patterns, and an aggregation (white rhombus) for NewsItem-NewsItemElement and Offer-Evaluation (both from the non-continuous evaluation exam).
  - Naming of associations.
  - Correct cardinalities in relations.
  - No duplicated data: attribute and relation that reflect the same data.
- Observer pattern (0.5 points for CE; 0.5 for NCE).
  - Identification of the pattern.
  - Correct assignment of class roles in the pattern.
  - Adding methods to the corresponding classes of the pattern.
- User/Subscriber class (0.5 points for CE; 0.4 for NCE).
  - Adding attributes/relations and methods associated to the exercise requirements: add/remove categories of interest, add/remove senders/transmitters.
- Sender class (0.5 points for CE; 0.4 for NCE).
  - Considering subclasses for the different types of senders, and definition of the Sender class as abstract.
  - Adding attributes and “send” method in the Sender class and subclasses.
- Category class (0.3 points for NCE).
  - Correct modeling of the category-subcategory taxonomy.
- Additional classes: news content and offer evaluations (0.3 points for NCE).
  - Correct modeling of classes (including their attributes) satisfying the additional constraints of the NCE exercise.
- Prototype of method of exercise b.1 (0.2 points for CE; 0.1 for NCE).
- Pseudocode of method of exercise b.1 (0.3 points for CE; 0.2 for NCE).
- Prototype of method of exercise b.2 (0.2 points for CE; 0.1 for NCE).
- Pseudocode of method of exercise b.2 (0.3 points for CE; 0.2 for NCE).

ADSOF: Ordinary call - 28/05/2020 Versión 1 – NON CONTINUOUS

Exercise 2 (3 points)

An application must handle medical *therapies*, each one of them identified by a unique name. Some therapies are single *drug prescriptions*, and others are *treatments* consisting of multiple therapies. Each drug prescription has, in addition to its name, a *number of doses*, and a *base risk*, whose *scale* can be: none, low, medium, high and severe. If the drug prescription is not explicitly given a number of doses, 1 will be assumed by default; and if no risk is specified, low will be assumed. However, at least one of those two data must be explicitly provided. Treatments are created empty (only with their name) to add therapies one by one to them. However, the application should avoid the possibility to add (*directly*) two therapies with same name to the same treatment.

We must be able to calculate the *actual risk* of each therapy. For a drug prescription with a number of doses of 1, its actual risk is equal to its standard risk. If it has number of doses higher than 1, the actual risk is the next one greater than its standard risk on the risk scale. For a treatment, its actual risk is the highest among all the therapies included in it.

Finally, we must have a mechanism to count the total number of created drugs.

Note: you may ignore errors due to negative or meaningless numbers, empty or null strings, and cyclic structures.

You are asked to:

(a) **Design and implement in Java** the code needed to solve the requirements stated above, so that the tester program shown below **produces the expected output.** You must apply **object-oriented principles to your design, making it general, reusable and extensible, keeping your code concise and clear**. [2,5 points]

(b) **Identify the design pattern(s)** you have used in your design, **and state the roles that your classes, methods and attributes play in each pattern used.** [0,5 points]

Expected output:

```
TR-1:[AAS(1,NONE), IBUPRO(1,HIGH)] actual risk: HIGH
TR-2:[TR-1:[AAS(1,NONE), IBUPRO(1,HIGH)], ANFE(2,SEVERE), AAS(1,NONE)] actual risk: SEVERE
Total number of drugs: 4
```

```
public class Ej2v1enNonCont {
  public static void main(String[] args) {
        Therapy te1 = new Drug("AAS", 1, Risk.NONE);
        Therapy te2 = new Drug("AAS", 3);
        Therapy te3 = new Drug("IBUPRO", Risk.HIGH);
        Therapy te4 = new Drug("ANFE", 2, Risk.HIGH); // actual risk: SEVERE
        Therapy tr1 = new Treatment("TR-1").add(te1)
                                           .add(te3)  // actual risk: HIGH
                                           .add(te2); // won´t be added
        Therapy tr2 = new Treatment("TR-2").add(tr1)
                                           .add(te4) // actual risk: SEVERE
                                           .add(tr1) // not added
                                           .add(te1); // added
        System.out.println( tr1 + " actual risk: " + tr1.actualRisk() );
        System.out.println( tr2 + " actual risk: " + tr2.actualRisk() );
        System.out.println( "Total number of drugs: " + Drug.count() );
  }
}
```

SOLUCIÓN Y PUNTUACIÓN, Ejercicio 2, Versión 1 NO Continua, 28 Mayo 2020, 3 puntos.

El reparto de puntos se refleja con la siguiente notación:

`[n] = valor aproximado sobre 30, a dividir por 10 para 3 puntos`

Además de las puntuaciones `[n]` asignadas a cada parte de la solución, se aplican penalizaciones por defectos relativos **principios de orientación a objetos en el diseño, así como su generalidad, reusabilidad, extensibilidad y la concisión y claridad,** como por ejemplo, código repetido innecesariamente, instrucciones if/else en cascada (o switch) para valores individuales de la enumeración, atributos no privados sin justificación válida, soluciones innecesariamente más complejas, etc.

Apartado (b): 0,5 puntos

`[1]` Se usa el patrón **Composite.**

`[2]` La clase **Therapy/Terapia** es la clase abstracta **Component** del patrón.
La clase **Drug/Receta** es la clase **Leaf** del patrón
La clase **Treatment/Tratamiento** es la clase **Composite** del patrón.

`[1]` El método actualRisk() es el método **operation**() del patrón.
El método add() es el método **add**() del patrón.

`[1]` El atributo **components** de Treatment es **children** en el patrón.

Apartado (a): 2,5 puntos

```
enum Risk {
      NONE, LOW, MEDIUM, HIGH, SEVERE;   [1] // mejor sin valores internos
      public Risk nextHigher() {         [1] // metodo para obtener el siguiente
            return Risk.values()[ Math.min(Risk.values().length-1,  this.ordinal()+1) ];
      }
}

abstract class Therapy {  // Terapia es la clase Component en el patrón Composite

      private String name;    [1] // atributo privado sin repetir en subclases
      public Therapy(String descr) { name = descr; }

      public Therapy add(Therapy t) { return this; } [1] // add() en Component del patrón

      public abstract Risk actualRisk(); [1] // operation() en Component del patrón

      @Override
      public final boolean equals(Object obj) { [2]
            return (obj instanceof Therapy) && this.name.equals( ((Therapy)obj).name );
      }
      @Override
      public final int hashCode() { return this.name.hashCode(); } [1]

      @Override
      public String toString() { return name; } [1]
}
```

```
class Treatment extends Therapy {// Tratamiento es la clase Composite en el patrón Composite
	private Set<Therapy> components = new LinkedHashSet<>();
	public Treatment(String descr) { super(descr); } [1]
	@Override
	public Therapy add(Therapy t) {
		this.components.add(t); [1]
		return this;            [1]
	}
	@Override
	public Risk actualRisk() {//operation() implementado en Composite del patrón
		Risk resultado = Risk.values()[0]; [1] // mejor que Risk.NONE;
		for (Therapy t : components) {     [1]
			Risk aux = t.actualRisk();
			if (aux.compareTo(resultado) > 0) [1]// mejor que comparar ordinal
				resultado = aux;
		}
		return resultado;
		/* O también en estilo funcional, pero sin olvidar orElse()
		return this.components.stream().map(Therapy::actualRisk)
		                              .max(Comparator.naturalOrder())
		                              .orElse(Risk.values()[0]); //mejor que Risk.NONE;
		*/
	}
	@Override
	public String toString() {    [1]
	 return super.toString() // mejor que getDescription() en superclase
	        + ":" + this.components.toString(); }
}

class Drug extends Therapy {  // Receta/Drug es la clase Leaf en el patrón Composite
	private static int count = 0;
	public static int count() { return count; } [1]

	private int doses;
	private Risk baseRisk;
	public Drug(String descr, int dosis, Risk r) { // constructor principal 3 parámetros
		super(descr);  [1] // usar super() para tener atributos private en superclase
		this.doses = dosis; baseRisk = r;  [2]
		Drug.count++;  [1]
	}
	public Drug(String descr, int dosis) { // constructor con riesgo BAJO por defecto
		this(descr, dosis, Risk.LOW);  [1] // usar this() para no repetir código
	}
	public Drug(String descr, Risk r) { // constructor con dosis = 1 por defecto
		this(descr, 1, r);   [1]  // usar this() para no repetir código
	}
	@Override
	public Risk actualRisk()  { // operation() implementado en Leaf del patrón
		return (this.doses == 1) ?  this.baseRisk : this.baseRisk.nextHigher(); [1]
	}
	@Override
	public String toString() {  [1]
		 return super.toString() // mejor que getDescription() en superclase
		        + "(" + this.doses + "," + this.actualRisk()+")"; }
}
```

ADSOF: Ordinary call - 28/05/2020 NON CONTINUOUS

Exercie 3 (2.5 points)

You need to design an application for auctions ("*subastas*"). For this purpose, you need to create a class `Auction` to store the values of the bids ("*pujas*") associated to the items to be auctioned. `Auction` should be highly reusable, with any item that implements the `IValuable` generic interface. This interface should be parameterized with an appropriate type to store a value for the item, such as for example `Long`, `Double` or `Integer`, but for generality, it will be open to any type that allows comparing its objects. When designing this interface, you should facilitate its implementation as much as possible. Next, we provide an incomplete fragment of the interface, which you should complete or modify as you see fit.

```
/** complete or modify this interface: you should facilitate its implementation as much as possible */
interface IValuable <T> {
	T getValue();
	String getDesc();
}
```

Complete the following program (***including their numbered holes***) so that it produces the output below. Please note that:

1. Variable `store` has type `Auction`, which has been parameterized with `Item`. This is a class that implements `IValuable`, storing its value (the price of the item) in a `Long`.
2. The `Item` constructor receives a description of the item, and its starting auction price.
3. `Auction` maintains its items ordered by starting price, and all their bids are stored in increasing order.
4. When adding an `Item` to the `Auction`, a sell condition should be given, which receives the bid value as argument.
5. If we try to bid (calling method `bid`) for an item that does not exist, an exception is raised. A bid that is lower than the starting price of the item, or that does not exceed the value of the highest bid so far, is ignored. Method `bid` returns *true* if the bid satisfies the sell condition, and *false* otherwise.

We will especially value the use of object-oriented principles in the design, as well as its generality, reusability, extensibility, and the conciseness and clarity of the code.

```
public class AuctionMain {
  public static void main(String... args) ____(1)_______{/* complete if needed */
	Auction<Long, Item> store = new Auction<>();
	__(2)__ {  /* complete */
		// A Picasso painting, for 200.000€. Sell if bid is over 230.000€
		store.addItem(new Item("Picasso painting", 200000L), bid -> bid > 230000L);
		// A bike for 200€. Sell if bid is over 300€, or if it is the second valid bid
		store.addItem(new Item("Bike", 200L), bid -> bid > 300L || store.bidsFor("Bike").size() == 2);

		// This bid satisfies the bid condition
		if (store.bid("Picasso painting", 250000L))
			System.out.println("Painting sold!");

		store.bid("Bike", 190L);	// This bid is ignored, because it is lower than the starting price
		store.bid("Bike", 220L);	// Valid bid, but does not satisfy the sell condition
		if (store.bid("Bike", 225L)) System.out.println("Bike sold!");  // Second bid, satisfies sell condition

		store.bid("ADSOF lab P4 and P5 assignment solutions", 1000L);   // Strangely enough, the item is not available
	} ______(3)_______ {  /* complete */
		_______(4)_______ /* complete */
	}
	System.out.println(store);	// Prints the object
  }
}
```

Expected output:

```
Painting sold!
Bike sold!
auctionsSimple.ItemNotFoundException: Item ADSOF lab P4 and P5 assignment solutions not found
{Bike=[220, 225], Picasso painting=[250000]}
```

Solution:
(1): nothing (2): try (3): catch(ItemNotFoundException exc) (4): System.out.println(exc);

```
interface IValuable<T extends Comparable<T>> extends Comparable<IValuable<T>>{
        T getValue();
        default String getDesc() { return toString(); }
        @Override default int compareTo(IValuable<T> o) { return getValue().compareTo(o.getValue()); }
}

class Item implements IValuable<Long>{
        private String name;
        private Long   initPrice;
        public Item (String i, Long price) {
                this.name = i;
                this.initPrice = price;
        }
        @Override public String toString() { return this.name; }
        @Override public Long getValue() { return initPrice; }
}

class ItemNotFoundException extends Exception {
        public ItemNotFoundException(String item) {
                super("Item "+item+" not found");
        }
}

public class Auction<T extends Comparable<T>, I extends IValuable<T>> {
        private Map<I, SortedSet<T>> bids = new TreeMap<>();
        private Map<I, Predicate<T>> sellConditions = new HashMap<>();

        public void addItem(I item, Predicate<T> pred) {
                this.bids.put(item, new TreeSet<>() );
                this.sellConditions.put(item, pred);
        }

        public boolean bid(String item, T p) throws ItemNotFoundException{
                I bidItem = this.itemDesc(item);
                SortedSet<T> itemBids = this.bidsFor(item);
                if (itemBids==null) throw new ItemNotFoundException(item);
                if (!highestBid(itemBids, p, bidItem)) return false;
                itemBids.add(p);
                if (this.sellConditions.get(bidItem).test(p)) {
                        return true;
                }
                return false;
        }

        private boolean highestBid(SortedSet<T> itemBids, T bid, I it) {
                if (itemBids.size()==0) return it.getValue().compareTo(bid)<0;
                if (itemBids.tailSet(bid).size()>0) return false;//no the highest bid so far
                return true;
        }

        @Override public String toString() {
                return this.bids.toString();
        }

        private I itemDesc(String desc) {
                for (I itm : this.bids.keySet())
                        if (itm.getDesc().equals(desc)) return itm;
                return null;
        }

        public SortedSet<T> bidsFor(String itemDesc) {
                I item = this.itemDesc(itemDesc);
                if (item == null) return null;
                return this.bids.get(item);
        }
}
```

ADSOF: Ordinary Call – NON-Continuous evaluation

Exercise 3 v1 (2.5 p)

Solution:

Holes: (1): nothing (2): try (3): catch(ItemNotFoundException exc) (4): System.out.println(exc); ➔**0,25p**

Total:0,5p

```
interface IValuable<T extends Comparable<T>> extends Comparable<IValuable<T>>{    ➔ (requirements over T and IValuable) 0,25p
        T getValue();
        default String getDesc() { return toString(); }
        @Override default int compareTo(IValuable<T> o) { return getValue().compareTo(o.getValue()); } ➔ default methods 0,25p
}
```

```
class Item implements IValuable<Long>{
        private String name;
        private Long   initPrice;
        public Item (String i, Long price) {
                this.name = i;
                this.initPrice = price;
        }
        @Override public String toString() { return this.name; }
        @Override public Long getValue() { return initPrice; }
}
```

Total:0,25p

(Common errors:
-- Considering Item as generic, when it is not
-- Item not generic, but uses a generic parameter
-- Add to Item the bids or the conditions (it is better to add this in Auction, because otherwise we would need to reimplement in every class implementing IValuable)

```
class ItemNotFoundException extends Exception {
        public ItemNotFoundException(String item) {
                super("Item "+item+" not found");
        }
}
```

Total:0,25p

(Common errors:
-- Storing here an Item, which would make this exception dependent on Item. This is a mistake, because Auction is independent of Item, which is just an example of class implementing IValuable.)

Total: 1,25p

```
public class Auction<T extends Comparable<T>, I extends IValuable<T>> {   ➔ Requirements over T and I: 0,2
        private Map<I, SortedSet<T>> bids = new TreeMap<>();               ➔ 0,2
        private Map<I, Predicate<T>> sellConditions = new HashMap<>();     ➔ 0,2

        public void addItem(I item, Predicate<T> pred) {                   ➔ 0,2
                this.bids.put(item, new TreeSet<>() );
                this.sellConditions.put(item, pred);
        }

        public boolean bid(String item, T p) throws ItemNotFoundException{        ➔ 0,25 (and auxiliary methods)
                I bidItem = this.itemDesc(item);
                SortedSet<T> itemBids = this.bidsFor(item);
                if (itemBids==null) throw new ItemNotFoundException(item);
                if (!highestBid(itemBids, p, bidItem)) return false;
                itemBids.add(p);
                if (this.sellConditions.get(bidItem).test(p))
                        return true;
                return false;
        }

        private boolean highestBid(SortedSet<T> itemBids, T bid, I it) {
                if (itemBids.size()==0) return it.getValue().compareTo(bid)<0;
                if (itemBids.tailSet(bid).size()>0) return false;//no the highest bid so far
                return true;
        }

        @Override public String toString() { return this.bids.toString();         }       ➔-0,1 if absent

        private I itemDesc(String desc) {
                for (I itm : this.bids.keySet())
                        if (itm.getDesc().equals(desc)) return itm;
                return null;
        }

        public SortedSet<T> bidsFor(String itemDesc) {                    ➔0.2
                I item = this.itemDesc(itemDesc);
                if (item == null) return null;
                return this.bids.get(item);
        }
}
```

(continues on next page)

Some comments on the solution:

- **Holes:**
  - (1) should be empty, since the exception is caught
- **Interface IValuable:**
  - T and IValuable should be comparable
  - We add default methods (compareTo, getDesc) to facilitate its implementation
- **Class Item:**
  - Item is not generic, as the code in the main method shows
  - Item should implement IValuable<Long>
  - If IValuable was not comparable, Item should implement Comparable and implement the methods. Therefore, it is better if IValuable provides this support already.
  - Declare here attributes and methods for the bids or the predicates is not optimal, since we would have to reimplement them in classes that implement IValuable. Auction does not have dependencies with Item, but with IValuable.
- **Exception class ItemNotFoundException**:
  - It should implement Exception
  - It cannot extend RuntimeException, because there would be no reasonable way to fill in the holes in the main
  - It needs to have that name, because of the message printed in the console.
  - auctionsSimple is the name of the package, not the name of the class
- **Class Auction:**
  - It has two generic parameters: one must be comparable and the other must extend IValuable.
  - It is a very serious conceptual error to put in the generic parameters things like Item, Double or similar, since they are names of concrete types
  - Declaring that a class throws an exception is a serious misconception. Exceptions are thrown by methods.
  - Bids must be associated with the items, and these must be ordered by starting price. For this reason, the best option is to use an ordered map, where the natural order is the starting price. Bids must be ordered in ascending order, and so the most appropriate structure would be an ordered set. Using a String as the map key is not valid, since the order criteria is not satisfied.
  - Predicates must be associated to the elements, for this purpose a map is required (although TreeSet is unnecessary).
  - Common errors in method bid: not declaring that it throws the exception, not throwing it, not checking that it is the highest bid, not saving the bid
- **Other common errors**
  - Using generic types neglecting the parameters
  - Not adding private access control to the attributes (basic concept error)
  - A constructor with 0 parameters without code (for example in Auction) is unnecessary.
  - Trying to copy from partners is always a mistake

ADSOF: Ordinaria call - 28/05/2020 NON CONTINUOUS

Exercise 4 (1.5 points)

We want to build a generic `BiStream` class that supports operating on the elements of two `Streams` simultaneously (advancing one by one on the elements of the two streams and applying the operation). In particular, we are interested in creating two methods:

- A `map` method, which can apply to each pair of stream elements a function of two arguments and produce a new `Stream` with the result. The type of the elements of the resulting `Stream` will be determined by the result of the function that `map` applies.
- A `filter` method, which will apply to each pair of elements of the streams a predicate of two arguments, and produces a new `Stream` with the individual objects that satisfy it.

In both cases, keep in mind that the streams can have different lengths, in which case `map` generates a `Stream` whose length is that of the shortest `Stream`.

Implement the `BiStream` class so that the following program produces the output indicated below. Please note that:

1. We should be able to construct a `BiStream` from any two collection types.
2. You must strive for the maximum generality of your class, allowing functions with compatible types of arguments and return value, as you can see in the second invocation to `map` in the listing below.

We will especially value the use of object-oriented principles in the design, as well as its generality, reusability, extensibility, and the conciseness and clarity of the code.

```
public class BiStreams {
  public static void main(String[] args) {
    List<String> aList   = Arrays.asList("a",  "list", "of", "words");
    Set<String>  aSet    = new LinkedHashSet<>(Arrays.asList("another", "collection", "of", "more", "Strings"));

    BiStream<String, String> bstr = new BiStream<>(aList, aSet);  // creating a BiStream from a list and a set

    System.out.println(bstr.map((x, y) -> x+" - "+y).
                        collect(Collectors.toList()));            // apply the map operation to every pair of strings

    // We create a BiStream with a list of Strings, and another of Integer, with the string lengths
    // The map function is defined over pairs of objects, and should be applicable to Strings and Integers
    System.out.println(new BiStream<Object, Integer>( aList,
                                                      aList.stream().map(x -> x.length()).collect(Collectors.toList())).
                        map((Object x, Object y) -> x.toString()+" - "+y).
                        collect(Collectors.toList()));

    // Filter let pass the strings of each collection starting by the same letter
    System.out.println(new BiStream<>(aList, aSet).
                        filter((x, y) -> x.charAt(0)==y.charAt(0)).
                        collect(Collectors.toList()));
  }
}
```

Expected output:

```
[a - another, list - collection, of - of, words - more]
[a - 1, list - 4, of - 2, words - 5]
[a, another, of, of]
```

SOLUCIÓN Y PUNTUACIÓN, Ejercicio 4, NO Continua, 28 Mayo 2020, 1,5 puntos.

El reparto de puntos se refleja con la siguiente notación:

`[n] = valor aproximado sobre 30, a dividir por 20 para 1,5 puntos`

Además de las puntuaciones `[n]` asignadas a cada parte de la solución, se aplican penalizaciones por defectos relativos **principios de orientación a objetos en el diseño, así como su generalidad, reusabilidad, extensibilidad y la concisión y claridad,** como por ejemplo, código repetido innecesariamente, copiado innecesario de estructuras de datos, atributos no privados sin justificación válida, soluciones innecesariamente más complejas … y **especialmente falta de generalidad en los parámetros genéricos.**

```
import java.util.function.*;
import java.util.stream.*;
import java.util.*;

class BiStream<T1, T2 extends T1> { [2]
      private Stream<? extends T1> str1; // Collection <? extends T1>
      private Stream<? extends T2> str2; // Collection <? extends T2>   [2]
                                                                  [3]
      public BiStream(Collection<? extends T1> str1, Collection<? extends T2> str2){
            this.str1 = str1.stream();
            this.str2 = str2.stream();   [1]
      }
            [2]                          [4]
      public <R> Stream<R> map(BiFunction<? super T1, ? super T2, ? extends R> fun) {
            Collection<R> res = new LinkedList<R>();
            // this.parallelFoldWith((T1 x, T2 y) -> res.add(fun.apply(x, y)));
            Iterator<? extends T1> it1 = str1.iterator();
            Iterator<? extends T2> it2 = str2.iterator();
            while (it1.hasNext() && it2.hasNext())   [0] Ver opción: No repetir código
              [2] res.add(  fun.apply(it1.next(), it2.next()) ); [2]
            return res.stream();   [1] incluida declaracion e inicialización
      }
                                    [3]
      public Stream<T1> filter(BiPredicate<? super T1, ? super T2> fun) {
            Collection<T1> res = new LinkedList<T1>();
            // this.parallelFoldWith((T1 x, T2 y) ->
            //                          {if (fun.test(x, y)) {res.add(x);res.add(y);}});
            Iterator<? extends T1> it1 = str1.iterator();
            Iterator<? extends T2> it2 = str2.iterator();
            while (it1.hasNext() && it2.hasNext()) {   [0] Ver opción: No repetir código
              T1 x = it1.next(); T2 y = it2.next();  [2]
              if ( fun.test(x, y) ) {  res.add(x);  res.add(y); } [2]
            }
            return res.stream();   [1] incluida declaracion e inicialización
      }

      // Opción para no repetir código:
      private void parallelFoldWith(BiConsumer<T1, T2> oper) {    [3]
            Iterator<? extends T1> it1 = str1.iterator();
            Iterator<? extends T2> it2 = str2.iterator();
            while (it1.hasNext() && it2.hasNext())
                  oper.accept(it1.next(), it2.next());
      }
}
```