Tema 2 – Listado de Ejercicios

Ejercicio 1

Durant aquesta primera meitat del curs, hem vist aspectes de la Programació Orientada a Objectes, com són classes abstractes, interfícies, i la relació d'herència (subclasse i superclasse). També hem treballat amb la interfície Comparable<E>, definida a la *Java Collections Framework*, al Laboratori 2, i també amb genèrics i comodins. Sabem que <? extends E> indica qualsevol tipus que sigui subtipus de E, i que <? super E> indica qualsevol tipus que sigui supertipus de E. Ara volem posar tot això en pràctica.

Suposem que tenim el següent mètode. Aquest mètode ens permet trobar el màxim donada una llista d'elements que siguin subtipus de Comparable.

public static <E extends Comparable<? super E>> E maxim(List<? extends E> l)

```
{
  E candidate = l.iterator().next();
  for (E ele: l) {
   if (candidate.compareTo(ele) < 0) candidate = ele;
  }
  return candidate;
}
```

Volem utilitzar aquesta operació per a un cas concret. Suposem que tenim una classe abstracta Fruita amb dues subclasses, Poma i Pera. Cada fruita té un pes. **Direm que dos fruites són iguals si tenen el mateix pes**. Utilitzant el mètode *maxim* definit abans, i una col·lecció de Fruita (per exemple, un ArrayList), hauríem de ser capaços de trobar quina és la fruita que pesa més.

Definim les nostres classes Fruita, Pera i Poma, de dues maneres diferents. Únicament mostrem els detalls més importants per respondre a les preguntes. Recordeu que el mètode compareTo retorna un nombre enter negatiu, zero o un nombre enter positiu si l'objecte és menor, igual, o major que un altre.

Opció A)

```
public abstract class Fruita { protected int pes = 0; ...}

public class Poma extends Fruita implements Comparable <Poma>{

(...)

   public int compareTo (Poma p){
       return this.pes < p.pes ? -1 : this.pes == p.pes ? 0 : 1;
   }
}
```

```
public class Pera extends Fruita implements Comparable <Pera>{
(...)

   public int compareTo (Pera pr){
       return this.pes < pr.pes ? -1 : this.pes == pr.pes ? 0: 1;
   }

}
```

Opció B)

```
public abstract class Fruita implements Comparable <Fruita>{
 protected int pes = 0;

    (...)

    public int compareTo (Fruita f){
        return this.pes < f.pes ? -1 : this.pes == f.pes ? 0 : 1;
    }
}

public class Poma extends Fruita{
    public Poma (int pes) { super(pes);}
}

public class Pera extends Fruita{
    public Pera (int pes) { super(pes); }
}
```

I ara, al nostre main, cridem al mètode *maxim* (que el tindrem definit dintre del nostre main) de la següent manera:

```
(1) Poma poma1 = new Poma (100);
(2) Poma poma2 = new Poma (90);
//creem més pomes
(3) Pera pera1 = new Pera (10);
(4) Pera pera2 = new Pera (50);
//creem més peres
(5) ArrayList<Poma> arr_pomes = new ArrayList<Poma>();
(6) ArrayList<Pera> arr_peres = new ArrayList<Pera>();
(7) ArrayList<Fruita> arr_fruita = new ArrayList<Fruita>();
//afegim pomes
(8) arr_pomes.add(poma1);
(9) arr_pomes.add(poma2);
//afegim peres
(10) arr_peres.add (pera1);
(11) arr_peres.add(pera2);
//afegim pomes i peres
(12) arr_fruita.add(poma1);
(13) arr_fruita.add(poma2);
(14) arr_fruita.add(pera1);
```

```
(15) arr_fruita.add(pera2);
//cridem al mètode maxim
(16) Fruita frt_max = maxim (arr_fruita);
(17) Poma poma_max = maxim (arr_pomes);
(18) Pera pera_max = maxim (arr_peres);
```

- (1p) L'opció A) no ens permet trobar el màxim a una llista de Peres i Pomes (línia 16). En cavi, la opció B) sí que ens permet trobar el màxim a una llista de Peres i Pomes. Raona la teva resposta en no més de 10 línies.
- (1p) Tenint en compte que la opció B) ens permet trobar el màxim a una llista de Peres i Pomes, també ens permet trobar el màxim a una llista únicament de Peres (línia 18), i a una altra llista, únicament de Pomes (línia 17)? Raona la teva resposta en no més de 10 línies.

Ejercicio 2

Tenim tres classes: *Fruita*, *Pera* i *Poma*. Les classes les hem definit de la següent manera:

```
public abstract class Fruita implements Comparable<Fruita> {...}
public class Poma extends Fruita {...}
public class Pera extends Fruita {...}
```

I tenim la capçalera del següent mètode:

```
public static <E extends Comparable<E>> E max (List<? extends E> l)
```

Quin canvi faries a la capçalera d'aquest mètode per a trobar el màxim a una llista de *Pomes* i *Peres*? Raona la teva resposta. No us demanem implementar el mètode. Tampoc us demanem canviar la definició de les classes *Fruita*, *Pera* i *Poma*.

Ejercicio 3

Tenemos dos implementaciones que definen dos jerarquías de clases. La implementación A es la siguiente:

```
public abstract class MedioTransporte
                          implements Comparable<MedioTransporte>{…}
public class Acuatico extends MedioTransporte{…}
public class Terrestre extends MedioTransporte{…}
public class Barco extends Acuatico{…}
public class Submarino extends Acuatico{…}
public class Bicicleta extends Terrestre{…}
public class Coche extends Terrestre{…}
public class ComparadorMedioTransporte
                          implements Comparator<MedioTransporte>{…}
```

La implementación B es como sigue:

```
public abstract class MedioTransporte{…}
public class Acuatico extends MedioTransporte{…}
public class Terrestre extends MedioTransporte
                      implements Comparable<Terrestre>{…}
public class Barco extends Acuatico{…}
public class Submarino extends Acuatico{…}
public class Bicicleta extends Terrestre{…}
public class Coche extends Terrestre{…}
public class ComparadorMedioTransporte
                           implements Comparator<MedioTransporte>{…}
```

Además, conocemos la cabecera de los siguientes métodos:

```
public static <E> E medidor1(List<E> l)
public static <E> E medidor2(Comparator<E> comp, List<E> l)
public static <E extends Comparable<? super E>> E medidor3(List<E> l)
```

En base a esto, explica de manera razonada, para cada implementación (A y B), si el siguiente código compila o no:

```
List<MedioTransporte> lm = new ArrayList<>();
ComparadorMedioTransporte comp = new ComparadorMedioTransporte();
List<Acuatico> la = new ArrayList<>();
List<Terrestre> lt = new ArrayList<>();
List<Coche> lc = new ArrayList<>();

MedioTransporte mt = medidor1(lm);
Coche co = medidor1(lc);

MedioTransporte mt2 = medidor2(comp, lm);

MedioTransporte mt3 = medidor3(lm);
Terrestre ter3 = medidor3(lt);
Acuatico ac3 = medidor3(la);
```

Ejercicio 4

Explica de manera bien justificada y haciendo uso de al menos, un ejemplo, las siguientes reglas generales aplicables en el uso de comodines:

- Se pueden utilizar para leer u obtener elementos, las estructuras de datos que contienen elementos de tipo *<? extends E>*. No se pueden utilizar para escribir.
- Se pueden utilizar para añadir elementos, las estructuras de datos que contienen elementos de tipo *<? super E>*.

Ejercicio 5

Tenemos implementadas las clases:

```
public abstract class Animal implements Comparable<Animal> { . . . }
public class Cow extends Animal { . . . }
public class Snake extends Animal { . . . }
```

Además, conocemos la cabecera de los siguientes métodos:

```
public static <E extends Comparable <E>> E strongest1(List<E> l)
public static <E extends Comparable <? super E>> E strongest2(List<E> l)
```

En base a esto, responde, explicando de manera razonada, a las siguientes preguntas:
a) ¿A qué método o métodos podemos llamar si tenemos una *List<Snake>*?
b) ¿A qué método o métodos podemos llamar si tenemos una *List<Animal>*?

Ejercicio 6

Tenemos implementadas las clases:

```
public abstract class Food {...}
public abstract class Meat extends Food
                        implements Comparable <Meat> {...}
public class Burger extends Meat {...}
public class BigBurger extends Burger
                       implements Comparable <BigBurger> {...}
public class SmallBurger extends Burger {...}
```

¿Es una implementación válida o existe algún problema en las clases definidas?

Ejercicio 7

És correcta la següent implementació del mètode *equals*? Raona la teva resposta. Si consideres que la implementació és incorrecta, proposa una implementació correcta del mètode.

```
public class MyDate{
    public int month;
    (...)
    @Override
    public boolean equals (Object dt){
        return (dt.month == this.month);
    }
    (...)
}
```

Ejercicio 8

- Los jefes de los departamentos son ingenieros (de la clase Engineer, que es subclase de Person). Implementad un método que reciba una lista de departamentos y una lista de ingenieros y añada los jefes de los departamentos a la lista de ingenieros.
  - La clase Department tiene un método "`public Engineer getHead()`" que devuelve el jefe del departamento.

- Generalizad la cabecera de la función anterior al máximo utilizando comodines.
- Si se desea escribir la lista de ingenieros, ¿se puede utilizar el siguiente método?
  - `void printNames(List<Person> lp)`
- ¿Cómo generalizarías la declaración del método usando comodines?

Ejercicio 9

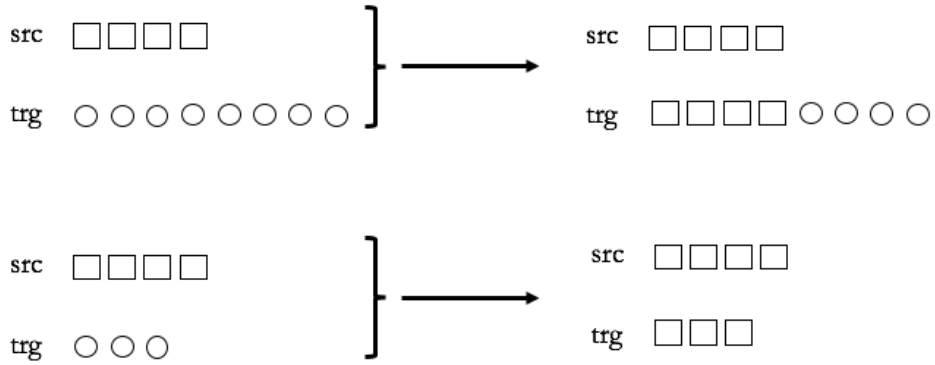
a) Implementa un método que reciba dos *LinkedList<E>* (*src* y *trg*) cuya funcionalidad sea cambiar el prefijo de *trg* con *src*, es decir, debe realizar lo que describe la figura. El resultado no será un nuevo *LinkedList<E>* sino que se debe modificar *trg*.

- **El coste de la función debe ser lineal, es decir, de O(N)**.

- La cabecera del método es

public static <E> void modifyPrefix(LinkedList<E> src, LinkedList<E> tfg)

b) **Generaliza** la cabecera de la función anterior al máximo utilizando comodines.



Ejemplo de funcionamiento método Pregunta 2

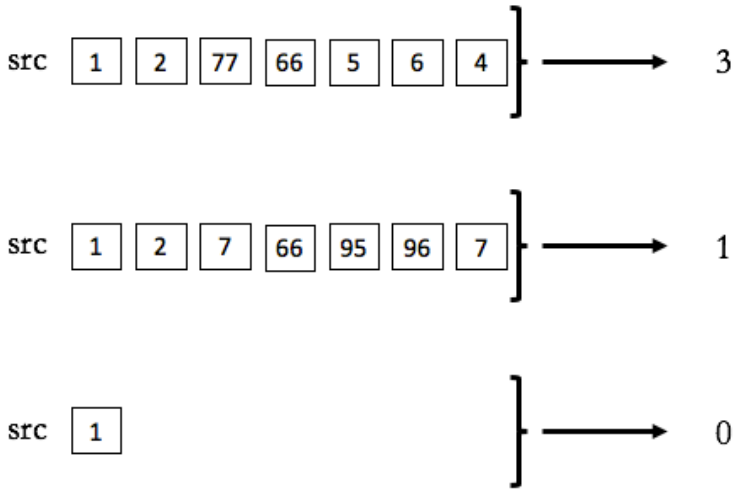
Ejercicio 10

Implementa dos versiones del método *consecutivePairs*, el cual, dado un iterador, obtenga el número de elementos consecutivos que se encuentran desordenados.

Dos elementos se consideran ordenados si el primero es inferior o igual al segundo, tal y como puede observarse en los ejemplos de la figura, en los que se han usado enteros siguiendo el criterio de ordenación creciente.

La primera versión trabajará con elementos comparables (*Comparable*<*E*>) y la segunda, utilizando un comparador (*Comparator*<*E*>) que será pasado como parámetro de la función. Las cabeceras de ambas funciones son las siguientes:

- `public static <E extends Comparable<E>> int consecutivePairs(ListIterator<E> l_iter)`
- `public static <E> int consecutivePairs(Comparator<E> comp, ListIterator<E> l_iter)`

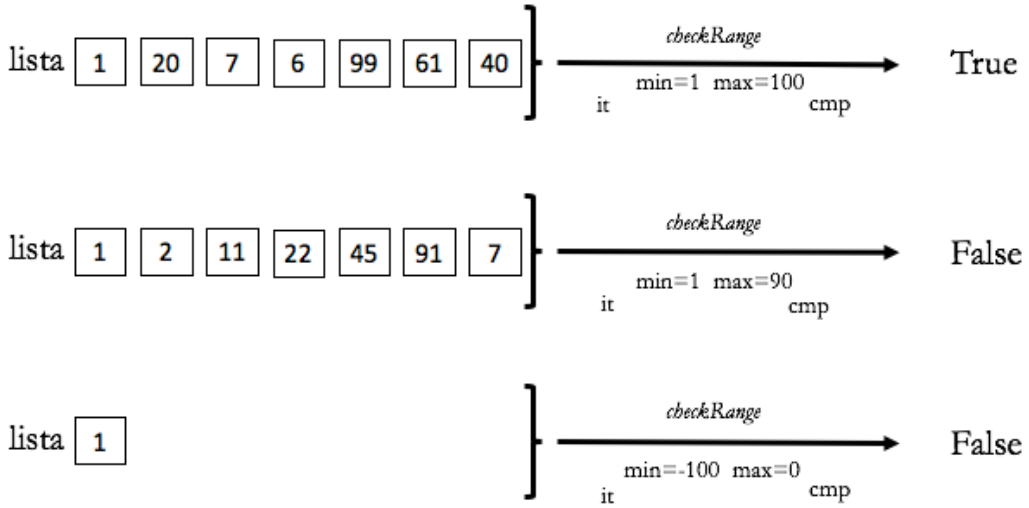


Ejemplo de funcionamiento del método consecutivePairs

Ejercicio 11

Implementa el método *checkRange*, el cual, dado un iterador, devuelva *true* o *false* en base a si todos los elementos accesibles a través del iterador se encuentran dentro del rango definido por dos elementos pasados como parámetros junto al iterador: *[min, max]*. La figura muestra un ejemplo del uso de *checkRange* para un iterador sobre una *List<Integer>*. La cabecera del método será la siguiente:

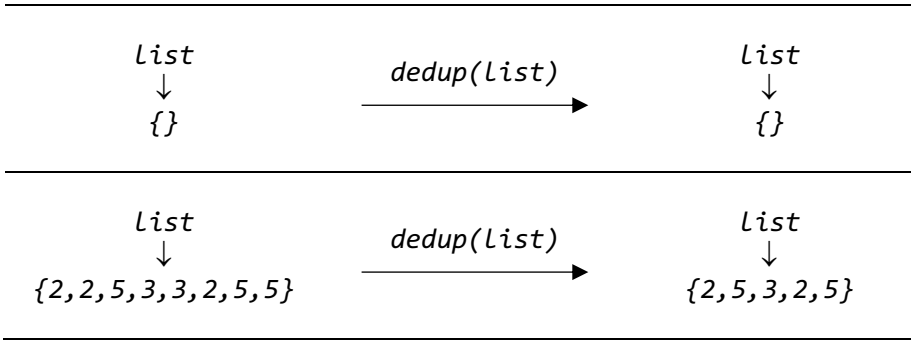
```
public static <E> boolean checkRange(Iterator<E> it,
                                     Comparator<E> cmp,
                                     E min,
                                     E max)
```



Ejemplo de funcionamiento del método checkRange para un iterador sobre una List<Integer> usando el comparador natural (orden creciente)

Ejercicio 12

Implementa el método *dedup* que reciba una lista y elimine de la misma, los elementos consecutivos que se encuentren duplicados. Dicha lista no contiene nulls y tampoco ella misma será null. La figura muestra dos ejemplos del funcionamiento del método *dedup*.



Ejemplos sobre el funcionamiento del método *dedup*

La implementación deberá ser realizada haciendo uso de iteradores.

La cabecera de la función será la siguiente:

```
public static <E> void dedup(List<E> list)
```

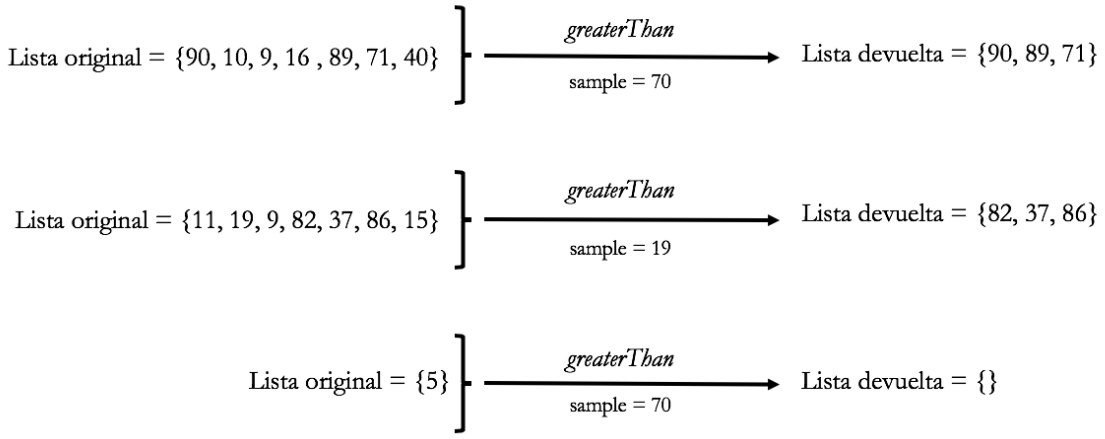
Ejercicio 13

Implementa el método *greaterThan*, el cual, dado un iterador, devuelva una lista con los elementos accesibles a través del iterador que son mayores que otro elemento pasado como parámetro. La figura muestra un ejemplo del uso de *greaterThan* para un iterador sobre una *List<Integer>*.

La implementación debe realizarse en dos versiones. La primera versión trabajará con elementos comparables (*Comparable<E>*) y la segunda, utilizando un comparador (*Comparator<E>*) que será pasado como parámetro de la función. Las cabeceras de ambas funciones son las siguientes:

```
public static <E extends Comparable<? super E>> List<E> greaterThan(Iterator<E>
it, E sample)

public static <E> List<E> greaterThan(Iterator<E> it,
                                      Comparator<? super E> cmp,
                                      E sample)
```



Ejemplo de funcionamiento del método greaterThan para un iterador sobre una List<Integer> usando el comparador natural (orden creciente)

Ejercicio 14

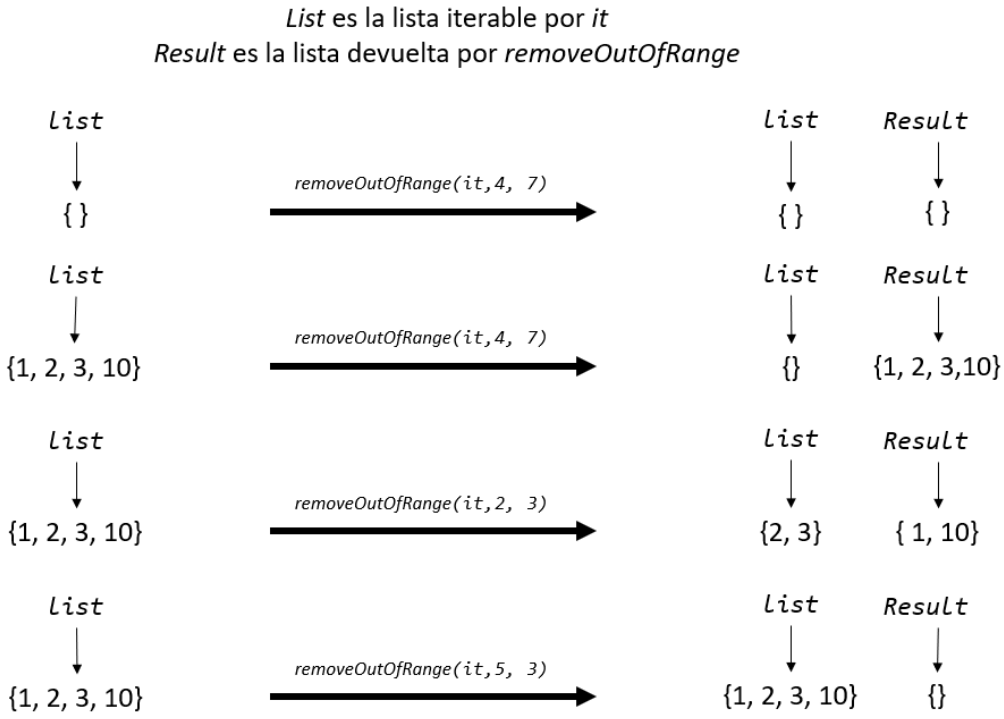
Implementa el método *removeOutOfRange* que, dado un iterador, devuelva una lista con los elementos accesibles con el iterador que se encuentren fuera del rango (que incluye sus extremos) definido por dos valores E pasados como parámetros y que, a su vez, los elimine de la colección que itera con el iterador.

La cabecera, la cual debe ser completada generalizándola al máximo y justificando la respuesta (puntos suspensivos deben adquirir un valor que puede ser diferente) sería:

```
public static <. . .> List<E> removeOutOfRange(Iterator<. . .> it,
                                               E min_range,
                                               E max_range)
```

Si *min_range* es mayor que *max_range*, la lista con los elementos accesibles con el iterador no se ve modificada y la lista devuelta por el método será vacía.

La figura muestra un ejemplo de uso para una lista de elementos Integer:



Ejemplos sobre el funcionamiento del método *removeOutOfRange*

Ejercicio 15

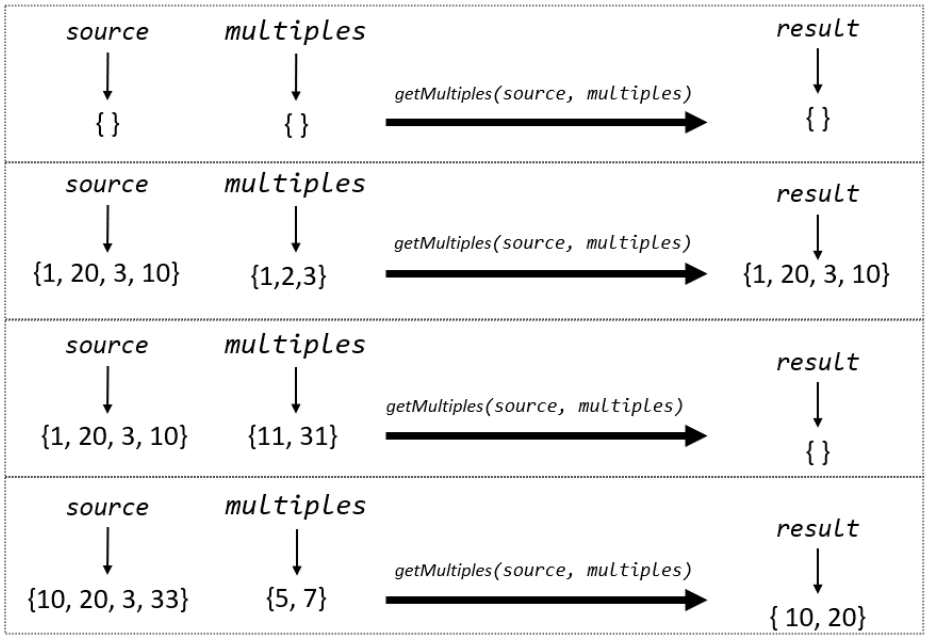
Implementa el método *getMultiples* que, dadas dos listas de enteros *source* y *multiples*, devuelva una lista con los elementos de *source* que sean múltiplos de algún elemento de *multiples*.

El método no modificará ni *source* ni *multiples*.

Si *source* o *múltiples* son listas vacías, la lista devuelta por el método será vacía.

Será **obligatorio** el uso de iteradores para trabajar con los elementos de *source* y *multiples*.

La figura siguiente muestra cuatro ejemplos de uso.



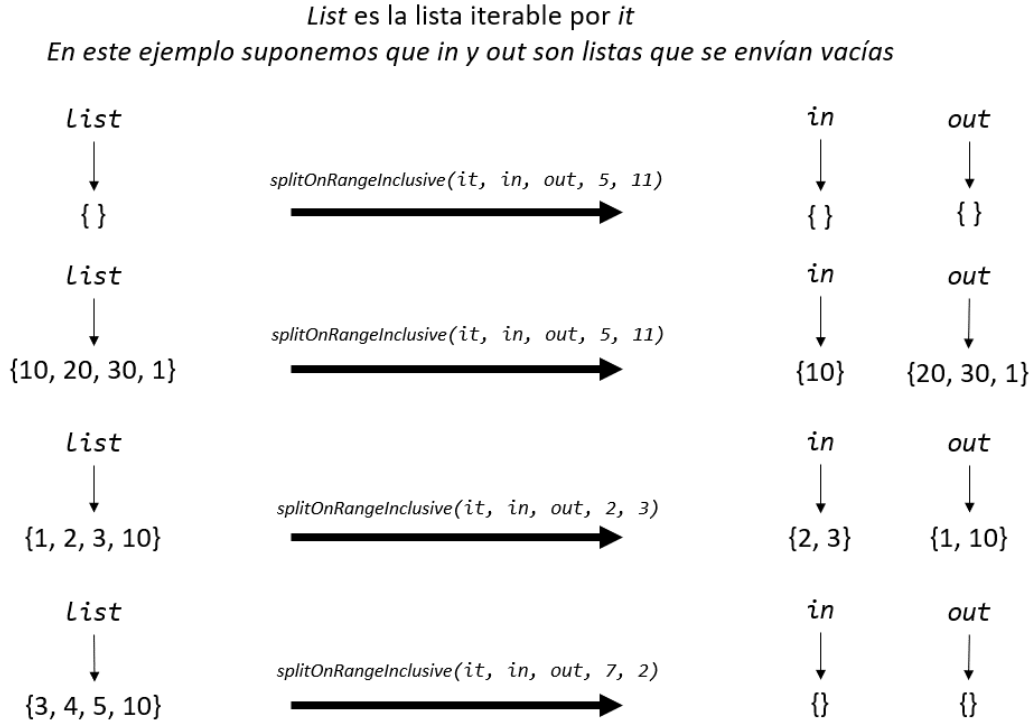
Ejemplos sobre el funcionamiento del método *getMultiples*

Ejercicio 16

Implementa el método *splitOnRangeInclusive* que, dado un iterador y dos listas (*in* y *out*), inserte en *in* los elementos accesibles con el iterador que se encuentren dentro del rango (que incluye sus extremos) definido por dos valores E pasados como parámetros (*min_range* y *max_range*) y que, a su vez, inserte en *out* los elementos accesibles con el iterador que se encuentren fuera del rango (sin incluir sus extremos).

Si *min_range* es mayor que *max_range*, las listas *in* y *out* no se verán modificadas.

La siguiente figura muestra un ejemplo de uso para una lista de elementos Integer:



Ejemplo de uso de *splitOnRangeInclusive*

La cabecera del método es la siguiente:

```
public static <E extends Comparable<? super E>> void
splitOnRangeInclusive
                                                    (Iterator<E> it,
                                                     List<E> in,
                                                     List<E> out,
                                                     E min_range,
                                                     E max_range)
```

Ejercicio 17

Tenemos las siguientes clases:

```
public class Point {
    protected int x;
    protected int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

}

public class ColoredPoint extends Point {
    private Color color;

    public Color(int x, int y, Color color) {
      super(x, y);
      this.color = color;
    }

    public Color getColor() {
      return color;
    }
}
```

Implementa de dos maneras el método *equals* de *Point:* (a) para que no sea interoperable con *ColorPoint*; y (b) para que lo sea. Explica y justifica las diferencias en las implementaciones.

Ejercicio 18

Tenemos las siguientes clases:

```
public class Rectangle {
    final int width;
    final int height;

    public Rectangle (int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

}

public class ColoredRectangle extends Rectangle {
    private Color color;

    public ColoredRectangle(int width, int height, Color color) {
      super(width, height);
      this.color = color;
    }


    public Color getColor() {
      return color;
    }
}
```

Implementa de dos maneras el método *equals* de *Rectangle:* (a) para que no sea interoperable con *ColorRectangle*; y (b) para que lo sea. Explica y justifica las diferencias en las implementaciones.