Ejercicio 1

Se cuenta con la clase **`Exam`** que modela la **hoja de inscripción de un examen de una materia**. Esta clase permite **inscribir y desinscribir alumnos**, y se define de la siguiente manera:

```
public class Exam {

   private String name;

   private String[] enrolled;
   private int dim;
   private static final int INITIAL_DIM = 5;

   public Exam(String name) {
       this.name = name;
       this.enrolled = new String[INITIAL_DIM];
   }

   public void enroll(String student) {
       if(dim == enrolled.length) {
           resize();
       }
       enrolled[dim++] = student;
       System.out.println("Enrolled " + student);
   }

   public void unenroll(String student) {
       for(int i = 0; i < dim; i++) {
           if(enrolled[i].equals(student)) {
               System.arraycopy(enrolled, i + 1, enrolled, i, dim - 1 - i);
               dim--;
               System.out.println("Unenrolled " + student);
           }
       }
   }

   public boolean isEnrolled(String student) {
       for(int i = 0; i < dim; i++) {
           if(enrolled[i].equals(student)) {
               return true;
           }
       }
       return false;
   }

   public int getEnrolledCount() {
       return dim;
   }

   public String[] getEnrolledStrudents() {
       return Arrays.copyOf(enrolled, dim);
   }

   private void resize() {
       enrolled = Arrays.copyOf(enrolled, enrolled.length + INITIAL_DIM);
   }

}
```

Ahora se desean modelar **dos hojas de inscripción más**:

- **Hoja de inscripción única (`UniqueExam`):** Al momento de inscribir un alumno, se debe verificar que el mismo no se encuentre ya inscripto en el mismo.

- **Hoja de inscripción única con cupo máximo (`LimitedExam`)**. Al momento de inscribir un alumno, además de verificar que el mismo no se encuentre ya inscripto en el mismo, se debe controlar que aún quede cupo disponible.
  En caso de que no hubiera cupo, se debe registrar que el alumno intentó inscribirse para otorgarle la próxima vacante disponible.
  Cuando se desinscribe un alumno, en caso de que hubieran alumnos esperando que se libere alguna vacante, se debe inscribir automáticamente al primero que lo haya intentado.

Implementar todo lo necesario (sin modificar la clase Exam) para que, con el siguiente programa, se obtenga la salida indicada.

```
Exam exam = new Exam("Primer Parcial PI");
exam.enroll("Matias");
exam.enroll("Matias");
exam.enroll("Natalia");
System.out.println("Enrolled Students: " + Arrays.toString(exam.getEnrolledStrudents()));
exam.unenroll("Matias");
System.out.println("Enrolled Students: " + Arrays.toString(exam.getEnrolledStrudents()));
System.out.println("##########");
UniqueExam uniqueExam = new UniqueExam("Primer Parcial POO");
uniqueExam.enroll("Matias");
uniqueExam.enroll("Matias");
uniqueExam.enroll("Natalia");
System.out.println("Enrolled Students: " + Arrays.toString(uniqueExam.getEnrolledStrudents()));
uniqueExam.unenroll("Matias");
System.out.println("Enrolled Students: " + Arrays.toString(uniqueExam.getEnrolledStrudents()));
System.out.println("##########");
LimitedExam limitedExam = new LimitedExam("TPE POD", 2);
limitedExam.enroll("Matias");
limitedExam.enroll("Matias");
limitedExam.enroll("Natalia");
limitedExam.enroll("Solange");
limitedExam.enroll("Jose");
limitedExam.enroll("Micaela");
System.out.println("Enrolled Students: " + Arrays.toString(limitedExam.getEnrolledStrudents()));
System.out.println("Enrolled Students: " + Arrays.toString(limitedExam.getPendingStudents()));
limitedExam.unenroll("Matias");
System.out.println("Enrolled Students: " + Arrays.toString(limitedExam.getEnrolledStrudents()));
System.out.println("Enrolled Students: " + Arrays.toString(limitedExam.getPendingStudents()));
limitedExam.unenroll("Jose");
limitedExam.unenroll("Natalia");
System.out.println("Enrolled Students: " + Arrays.toString(limitedExam.getEnrolledStrudents()));
System.out.println("Enrolled Students: " + Arrays.toString(limitedExam.getPendingStudents()));
```

```
Enrolled Matias
Enrolled Matias
Enrolled Natalia
Enrolled Students: [Matias, Matias, Natalia]
```

```
Unenrolled Matias
Enrolled Students: [Matias, Natalia]
##########
Enrolled Matias
Enrolled Natalia
Enrolled Students: [Matias, Natalia]
Unenrolled Matias
Enrolled Students: [Natalia]
##########
Enrolled Matias
Enrolled Natalia
Enrolled Students: [Matias, Natalia]
Pending Students: [Solange, Jose, Micaela]
Unenrolled Matias
Enrolled Solange
Enrolled Students: [Natalia, Solange]
Pending Students: [Jose, Micaela]
Unenrolled Natalia
Enrolled Micaela
Enrolled Students: [Solange, Micaela]
Pending Students: []
```

Ejercicio 2

Un **iterador paralelo** es un iterador que, a partir de dos vectores, permite recorrerlos paralelamente a ambos hasta que se alcance el final de una de las colecciones.

La clase **`ParallelIterator`** recibe en su constructor las dos colecciones a iterar.

Implementar todo lo necesario para que el siguiente programa de prueba imprima la salida indicada:

```
public class ParallelIteratorTester {

   public static void main(String[] args) {
       String[] v1 = new String[]{"hola", "mundo", "adios"};
       String[] v2 = new String[]{"hello", "world"};

       ParallelIterator<String> myIterator = new ParallelIterator<>(v1, v2);
       System.out.println(myIterator.next());
       System.out.println(myIterator.next());
       try {
           System.out.println(myIterator.next());
       } catch (NoSuchElementException ex) {
           System.out.println(ex.getClass());
       }

       System.out.println("##########");

       try {
           new ParallelIterator<>(null, v2);
       } catch (Exception ex) {
           System.out.println(ex.getMessage());
       }

       System.out.println("##########");

       try {
           new ParallelIterator<>(v1, null);
       } catch (Exception ex) {
           System.out.println(ex.getMessage());
       }
   }

}
```

```
{hola,hello}
{mundo,world}
java.util.NoSuchElementException
##########
First collection missing
##########
Second collection missing
```

Ejercicio 3

Se quiere modelar el menú de una pizzería.

Se tienen dos tipos de pizza: al horno y a la parrilla y varios agregados (*toppings*) que se le pueden poner encima a la pizza: extra queso, tomate y cebolla.

Los precios de los tipos de pizza son:

- **Pizza al horno:** $100
- **Pizza a la parrilla:** $150

Los precios de los *toppings*:

- **Extra queso:** $20
- **Tomate:** $30
- **Cebolla:** $10

Completar los … e implementar todo lo necesario para que, con el siguiente programa de prueba

```
public class PizzaTester {

  public static void main(String[] args) {
     // Pizza a la parrilla con Tomate
     Pizza pizza1 = ...............................................;
     // Pizza a la parrilla con Tomate con Cebolla con Extra queso
     Pizza pizza2 = ...............................................;
     // Pizza al horno con Cebolla con Extra queso
     Pizza pizza3 = ...............................................;
     System.out.println(pizza1);
     System.out.println(pizza2);
     System.out.println(pizza3);
  }

}
```

se obtenga la siguiente salida:

```
Pizza a la parrilla con Tomate: $180.00
Pizza a la parrilla con Tomate con Cebolla con Extra queso: $210.00
Pizza al horno con Cebolla con Extra queso: $130.00
```

Ejercicio 4

Dada la siguiente jerarquía de clases, con los métodos de instancia indicados para cada una, se cuenta con tres instancias homónimas a la clase a la cual pertenecen.

Completar el cuadro de doble entrada (clase y mensaje) indicando qué se obtiene al enviar cada uno de los mensajes a instancias de cada una de las clases.

	m1	m2	m3
A			
B			
C			

```
class A {

 int m1() {
  return 7;
 }

 int m2() {
  return this.m3();
 }

 int m3() {
  return m1();
 }

}
```

```
class B extends A {

 int m1() {
  return 9;
 }

 int m3() {
  return super.m3();
 }

}
```

```
class C extends B {

 int m1() {
  return super.m1();
 }

 int m2() {
  return this.m2();
 }

 int m3() {
  return m3();
 }

}
```