Programación sobre Redes - Guía de Ejercicios 3

Nota: Todas las clases e interfaces que se declaren a partir de este punto deben ser declaradas en el package ar.edu.ottokrause.psr.math.

Ejercicio 1:

Crear una clase Point que representa un punto en el plano coordinado. Las instancias de la clase Point deben saber responder los siguientes mensajes:

```
public double getX();
```
Devuelve la coordenada x del punto.

```
public double getY();
```
Devuelve la coordenada y del punto.

```
public Point add(Point aPoint) {
```
Devuelve un nuevo punto que es la suma de las coordenadas de `this` con las coordenadas de aPoint.

La clase Point debe proveer al menos los siguientes constructores:

```
public Point();
```
Crea un punto con sus coordenadas en (0, 0).

```
public Point(double x, double y);
```
Crea un punto con sus coordenadas en (x, y).

Ejercicio 2:

Crear una clase Vector3D que representa un vector en el espacio tridimensional. Las instancias de la clase Vector3D deben saber responder los siguientes mensajes:

```
public double getX();
```
Devuelve la coordenada x del vector.

```
public double getY();
```
Devuelve la coordenada y del vector.

```
public double getZ();
```
Devuelve la coordenada Z del vector.

```
public double norm();
```
Devuelve la norma del vector.

```
public Vector3D add(Vector3D aVector);
```
Devuelve un nuevo vector que es la suma de las coordenadas de `this` con las coordenadas de aVector.

```
public double dotProduct(Vector3D aVector);
```
Devuelve el producto escalar entre el vector representado por **this** y el vector aVector. Más formalmente, si llamamos A a **this** y B a aVector, con N = 3:

$$\lambda = \sum_{i=0}^{N-1} A(i) * B(i)$$

Dónde λ es el producto escalar.

La clase Vector3D debe proveer al menos los siguientes constructores:

```
public Vector3D();
```
Crea un vector con sus coordenadas en (0, 0, 0).

```
public Vector3D(double x, double y, double z);
```
Crea un vector con sus coordenadas en (x, y, z).

Ejercicio 3:

Crear una clase Vector que representa un vector en el espacio n-dimensional (n coordenadas). Las instancias de la clase Vector deben saber responder los siguientes mensajes:

```
public double getCoordinate(int index);
```
Devuelve la coordenada i-ésima del vector.

```
public double norm();
```
Devuelve la norma del vector.

```
public Vector add(Vector aVector);
```
Devuelve un nuevo vector que es la suma de las coordenadas de **this** con las coordenadas de aVector.

```
public double dotProduct(Vector aVector);
```
Devuelve el producto escalar entre el vector representado por **this** y el vector aVector. Más formalmente, si llamamos A a **this** y B a aVector:

$$\lambda = \sum_{i=0}^{N-1} A(i) * B(i)$$

Dónde λ es el producto escalar.

La clase Vector debe proveer al menos los siguientes constructores:

```
public Vector(int coordinateCount);
```
Crea un vector de coordinateCount dimensiones con sus coordenadas en 0.

```
public Vector(double[] coordinates);
```
Crea un vector con sus coordenadas inicializadas con los valores que tiene el arreglo coordinates.

Ejercicio 4:

Crear una clase Circle que representa un círculo en el plano cartesiano. Las instancias de la clase Circle deben saber responder los siguientes mensajes:

```
public double getArea();
```
Devuelve el área del círculo.

```
public Point getCenter();
```
Devuelve el punto que representa el centro del círculo.

```
public double getRadius();
```
Devuelve el radio del círculo.

```
public double getPerimeter();
```
Devuelve la longitud de la circunferencia.

Ejercicio 5:

Crear una clase Rectangle que representa un rectángulo en el plano cartesiano. Las instancias de la clase Rectangle deben saber responder los siguientes mensajes:

```
public double getArea();
```
Devuelve el área del rectángulo.

```
public Point getOrigin();
```
Devuelve el punto que representa la esquina superior izquierda del rectángulo.

```
public Point getCorner();
```
Devuelve el punto que representa la esquina inferior derecha del rectángulo.

```
public double getPerimeter();
```
Devuelve el perímetro del rectángulo.

Ejercicio 6:

Crear una clase Triangle que representa un triángulo en el plano cartesiano. Las instancias de la clase Triangle deben saber responder los siguientes mensajes:

```
public double getArea();
```
Devuelve el área del triángulo.

```
public Point[] getVertices();
```
Devuelve un arreglo que contiene los puntos que forman los vértices del triángulo.

```
public Point getVertex(int index);
```
Devuelve uno de los puntos que forman los vértices del triángulo.

```
public double getPerimeter();
```
Devuelve el perímetro del triángulo.

Listas

Una lista es una colección ordenada, también conocida como secuencia. List<E> es una interfaz. Los usuarios de la interfaz List<E> tienen un control preciso sobre donde se inserta cada elemento en la lista. Los usuarios pueden acceder los elementos usando un índice entero (posición en la lista), y buscar elementos en la lista.

A diferencia de los conjuntos, las listas generalmente permiten la existencia de elementos duplicados. Más formalmente, las listas generalmente permiten pares de elementos e1 y e2 tales que e1.equals(e2), y permiten múltiples elementos nulos si es que permiten elementos nulos.

Java define una interfaz List<E> en el paquete java.util. Esa interfaz declara muchos mensajes y resulta complicada de implementar. Luego, para la materia se va a definir una interfaz alternativa que va a hacer más sencilla su implementación. Para ello no se utilizarán características tales como "generics", con lo cual en lugar de definir una lista que pueda contener cualquier tipo de objetos, nos vamos a limitar a implementar una lista que contenga únicamente valores enteros. Además, se va a reducir la cantidad de método de la interfaz para hacer más simple la implementación. La interfaz lista de enteros que se definirá para la materia soporta el siguiente protocolo:

```
package ar.edu.ottokrause.psr.collections;

public interface IntegerList {

	/** Appends the specified element to the end of this list. */
	boolean add(Integer element);

	/** Inserts the specified element at the specified position
	 * in this list (optional operation). */
	void add(int index, Integer element);

	/** Appends all of the elements in the specified collection
	 * to the end of this list, in the order that they are
	 * returned by the specified collection's iterator. */
	boolean addAll(IntegerList aList);

	/** Inserts all of the elements in the specified
	 * collection into this list at the specified position. */
	boolean addAll(int index, IntegerList aList);

	/** Removes all of the elements from this list. */
	void clear();

	/** Returns true if this list contains the specified element. */
	boolean contains(Integer element);

	/** Returns the element at the specified position in this list. */
	Integer get(int index);

	/** Returns the index of the first occurrence of the specified
	 * element in this list, or -1 if this list does not contain
	 * the element. */
	int indexOf(Integer element);
```

```
        /** Returns true if this list contains no elements. */
        boolean isEmpty();

        /** Returns the index of the last occurrence of the specified
         * element in this list, or -1 if this list does not contain
         * the element. */
        int lastIndexOf(Integer element);

        /** Removes the element at the specified position in this list. */
        int remove(int index);

        /** Replaces the element at the specified position
         * in this list with the specified element. */
        void set(int index, Integer element);

        /** Returns the number of elements in this list. */
        int size();

        /** Returns a view of the portion of this list between
         * the specified fromIndex, inclusive, and toIndex, exclusive. */
        List<E> subList(int fromIndex, int toIndex);

        /** Returns an array containing all of the elements
         * in this list in proper sequence (from first to last element). */
        E[] toArray();
}
```

ArrayList: Listas implementadas sobre arreglos

Nota: Todas las clases e interfaces que se declaren a partir de este punto deben ser declaradas en el package ar.edu.ottokrause.psr.collections.

Ejercicio 7:
Crear una clase IntegerArrayList que implemente la interfaz IntegerList utilizando arreglos como el medio de almacenamiento subyacente. Al momento de agregarse un elemento de una instancia de IntegerArrayList, el arreglo que contiene los elementos debe ser copiado a uno nuevo que tenga lugar para almacenar el nuevo elemento que se desea agregar.

Ejercicio 8:
Modificar la clase IntegerArrayList del ejercicio anterior o crear una clase IntegerArrayList2 que implemente la interfaz IntegerList utilizando arreglos como el medio de almacenamiento subyacente. En este caso, se debe reservar más espacio que el requerido a priori, de manera tal que no sea necesario copiar el arreglo cada vez que el usuario de la clase quiera agregar un elemento al final de la lista.

Ejercicio 9:
Modifica la clase del ejercicio anterior de manera que soporte agregar elementos tanto al principio como al final de la lista sin tener que copiar todo el arreglo cada vez que se agrega un elemento.

LinkedList: Listas implementadas con Nodos Enlazados (Opcional)

Las listas enlazadas son una forma de implementar la interfaz IntegerList que utiliza nodos que contienen los valores y apuntan al siguiente elemento de la lista (simplemente enlazadas) o al siguiente y al anterior (doblemente enlazadas). Podemos definir la clase IntegerNode de la siguiente manera:

```
package ar.edu.ottokrause.psr.collections;

public class IntegerNode {

	private Integer value;
	private IntegerNode next;
	private IntegerNode previous;

	public IntegerNode() {
	}

	public IntegerNode(Integer value) {
		this.value = value;
	}

	public IntegerNode(IntegerNode next) {
		this.next = next;
	}

	public IntegerNode(IntegerNode next, Integer value) {
		this.next = next;
		this.value = value;
	}

	public IntegerNode(IntegerNode next, IntegerNode previous) {
		this.next = next;
		this.previous = previous;
	}

	public IntegerNode(IntegerNode next,
			Integer value, IntegerNode previous) {
		this.next = next;
		this.value = value;
		this.previous = previous;
	}

	public Integer getValue() {
		return this.value;
	}

	public void setValue(Integer value) {
		this.value = value;
	}

	public IntegerNode getNext() {
		return this.next;
	}
```

```
	public void setNext(IntegerNode next) {
		this.next = next;
	}

	public IntegerNode getPrevious() {
		return this.previous;
	}

	public void setPrevious(IntegerNode previous) {
		this.previous = previous;
	}
}
```

Ejercicio 10:
Crear una clase IntegerLinkedList que implemente la interfaz IntegerList utilizando una lista simplemente enlazada.

Ejercicio 11:
Crear una clase IntegerDoubleLinkedList que implemente la interfaz IntegerList utilizando una lista doblemente enlazada.

Matrices

En matemática, una matriz es un arreglo bidimensional de números, símbolos o expresiones. Dado que puede definirse tanto la suma como el producto de matrices, se dice que las matrices son elementos de un anillo.

Ejercicio 12:
Crear una clase Matrix que representa una matriz de números reales. Las instancias de la clase Matrix deben saber responder los siguientes mensajes:

```
public int rows();
```
Devuelve la cantidad de filas de la matriz.

```
public int columns();
```
Devuelve la cantidad de columnas de la matriz.

```
public double get(int row, int column);
```
Devuelve el valor que se encuentra en la fila row y columna column de la matriz.

```
public void set(int row, int column, double value);
```
Establece el valor que se encuentra en la fila row y columna column de la matriz al valor de value.

La clase Matrix debe proveer al menos los siguientes constructores:

```
public Matrix(int rows, int columns);
```
Crea una matriz de rows filas y columns columas con todos los elementos inicializados a 0.

```
public Matrix(double[][] elements);
```
Crea una matriz que contiene los mismos elementos que el arreglo de arreglo de doubles. Se asume que los arreglos que componen el arreglo de arreglos elements tienen todos la misma longitud.

```
public Matrix(Matrix matrix);
```
Crea una matriz que es una copia de la matriz que se le pasa como parámetro.

Ejercicio 14:
Agregar a la clase Matrix un método que devuelva si la misma es diagonal. Una matriz se dice diagonal si los elementos de su diagonal principal son distintos de cero y el resto de los elementos son cero.

Ejercicio 14:
Agregar a la clase Matrix un método que devuelva si la misma es la matriz identidad. La matriz identidad de tamaño N es aquella matriz de tamaño N x N con unos en la diagonal, y ceros en las otras posiciones.

Ejercicio 15:
Agregar a la clase Matrix un método que tome como parámetro un número real y que multiplique todos los elementos de la matriz por ese número.

Ejercicio 16:
Agregar a la clase Matrix un método que tome como parámetro otra matriz y devuelva una nueva matriz que sea la suma de ambas matrices. El método no debe modificar las matrices originales.

Ejercicio 17:
Agregar a la clase Matrix un método que tome como parámetro otra matriz y devuelva una nueva matriz que sea la resta de ambas matrices. El método no debe modificar las matrices originales.

Ejercicio 18:
Agregar a la clase Matrix un método que tome como parámetro otra matriz y devuelva una nueva matriz que sea el producto de ambas matrices. El método no debe modificar las matrices originales.

Ejercicio 19:
Agregar a la clase Matrix un método que transponga la matriz.

Ejercicio 20:
Agregar a la clase Matrix un método que devuelva si la matriz es simétrica.

Ejercicio 21:
Agregar un método a la clase Matrix que rote la matriz a izquierda. Se define rotar la matriz a izquierda como la operación que mueve cada uno de los elementos de la matriz una posición a la izquierda. En caso de que un elemento al ser movido a la izquierda no tenga lugar en la fila en la que se encontraba originalmente, el mismo debe ser puesto en la última posición de la fila

anterior. En caso que el elemento que no tiene lugar en la fila se encontrara en la primera fila, el mismo debe ser llevado a la última posición de la última fila.

Ejercicio 22:

Agregar un método a la clase Matrix que rote la matriz a derecha. Se define rotar la matriz a derecha como la operación que mueve cada uno de los elementos de la matriz una posición a la derecha. En caso de que un elemento al ser movido a la derecha no tenga lugar en la fila en la que se encontraba originalmente, el mismo debe ser puesto en la primera posición de la siguiente fila. En caso que el elemento que no tiene lugar en la fila se encontrara en la última fila, el mismo debe ser llevado a la primera posición de la primera fila.