UNIVERSITAT JAUME·I

# 1º Ingeniería Informática
# 1º Matemática Computacional
## EI1008/MT1008 Programación II
Examen final. Primera convocatoria. (05 de junio de 2014)

### Pregunta 1 (5,5 puntos)

Considera el sistema que gestiona los préstamos y devoluciones de libros de una biblioteca. Cuando un usuario quiere tomar prestado un libro y no existe ningún ejemplar disponible en ese momento, el sistema permite realizar una reserva de ese libro. A medida que los libros solicitados vuelvan a estar disponibles, el sistema irá atendiendo las reservas según el orden en que se realizaron.

Para ello, el sistema dispone de la clase `Reserva`, en la que se definen los siguientes atributos y métodos:

- Atributos:
  - `private String DNI`, que almacena el DNI de la persona que realiza la reserva.
  - `private int idLibro`, que almacena un identificador numérico del libro reservado.
- Métodos:
  - `public Reserva(String unDNI, int unIdLibro)`, constructor de objetos de la clase.
  - `public String getDNI()`, devuelve el atributo `DNI`.
  - `public int getIdLibro()`, devuelve el atributo `idLibro`.
  - `public boolean equals(Object obj)`, compara objetos de la clase.

Queremos implementar una clase `ReservasBiblioteca` para gestionar las reservas de libros en la biblioteca. De momento, disponemos de la siguiente definición parcial:

```
public class ReservasBiblioteca {

    private Reserva[] vectorReservas; // Almacena los datos de las reservas
    private int ocupados; // Indica las posiciones realmente ocupadas en el vector

    public ReservasBiblioteca() {
        ocupados = 0; // Inicialmente no hay reservas
        vectorReservas = new Reserva[10]; // Crea un vector con capacidad para 10 reservas
    }
}
```

Como puedes ver, la representación interna de la clase `ReservasBiblioteca` contiene como atributos un vector de reservas (`vectorReservas`) y un valor entero (`ocupados`) que indica la cantidad de reservas en la biblioteca. De esta manera, se distingue entre la longitud (o capacidad máxima) del vector y la cantidad de reservas que realmente contiene.

Se pide implementar los siguientes métodos públicos:

a) `boolean añadir(String DNI, int idLibro)` **(1,25 puntos)**

Si ya existe una reserva igual en la biblioteca, devuelve como resultado `false`. Si no, el método añade una nueva reserva y devuelve como resultado `true`.

Para añadir la reserva, primero debes comprobar si el vector está lleno, en cuyo caso debes duplicar su longitud. Después, debes añadir la reserva en la primera posición libre del vector.

b) `String servirLibro(int idLibro)` **(1,25 puntos)**

Elimina la primera reserva correspondiente al identificador de libro dado y devuelve el DNI de la persona que había realizado esa reserva. Si no hay ninguna reserva para ese identificador, el método debe devolver `null`.

Para eliminar una reserva del vector, todas las reservas posteriores a la misma deben desplazarse una posición a la izquierda en el mismo vector, de modo que se respete su orden de llegada.

c) `String usuarioMásReservas()` **(1,5 puntos)**

Devuelve el usuario que tiene más reservas pendientes en la biblioteca. En caso de empate entre varios usuarios, puedes devolver cualquiera de ellos. Si no existe ninguna reserva en la biblioteca, el método debe devolver como resultado `null`.

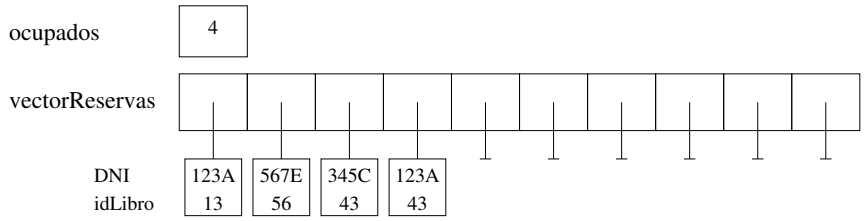
Para resolver este apartado, te puede resultar útil definir un método privado auxiliar que tenga como parámetro el DNI de un usuario y devuelva la cantidad de reservas que ese usuario tiene en la biblioteca.

d) `int anularReservas(String DNI)` **(1,5 puntos)**

Elimina todas las reservas que existan para el DNI dado y devuelve la cantidad de reservas anuladas. Si no hay ninguna reserva para ese DNI, el método debe devolver cero. Al igual que en el método `servirLibro`, los desplazamientos necesarios se realizarán sobre el propio vector (es decir, sin crear vectores auxiliares de reservas).

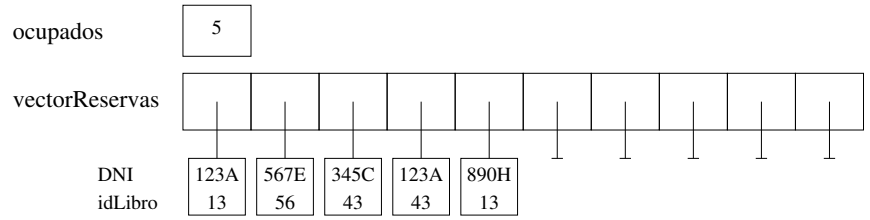
**Ejemplo**

Considerando que `bib` es un objeto de la clase `ReservasBiblioteca` cuyo estado inicial se muestra en la siguiente figura:

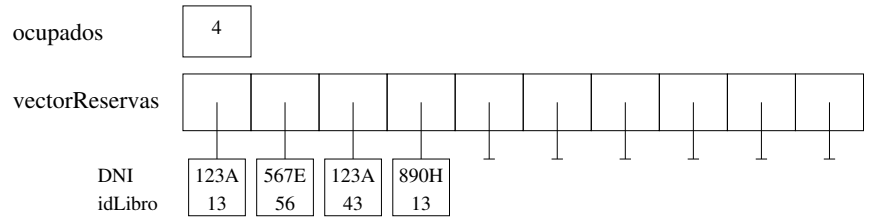



los siguientes gráficos ilustran el comportamiento de cada uno de los métodos pedidos:

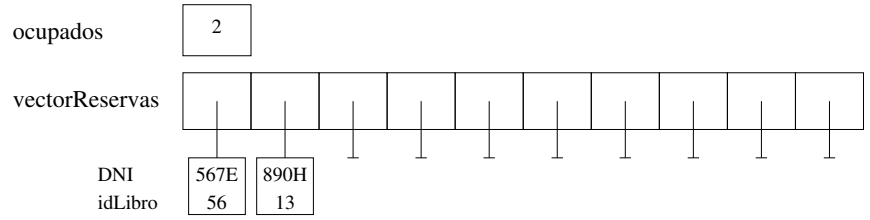
- La llamada `bib.añadir("890H", 13)` devolvería `true` y el estado final de `bib` sería:




- La llamada `bib.servirLibro(43)` devolvería `"345C"` y el estado final de `bib` sería:




- La llamada `bib.usuarioMásReservas()` devolvería `"123A"`.
- La llamada `bib.anularReservas("123A")` devolvería `2` y el estado final de `bib` sería:



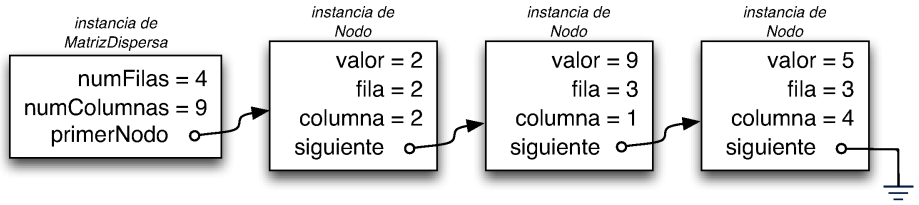
## Pregunta 2 (4,5 puntos)

Una *matriz dispersa* es una matriz de gran tamaño en la que la mayor parte de sus elementos son cero. Para almacenar este tipo de información, el uso de matrices nativas de Java requiere una gran cantidad de memoria, la mayor parte de ella destinada a almacenar ceros. Para solucionar este problema, te proponemos definir una clase `MatrizDispersa` que permita representar matrices dispersas almacenando internamente la cantidad de filas y de columnas de la matriz y una *lista de nodos simplemente enlazada* en la que únicamente almacenamos aquellos valores de la matriz distintos de cero, junto con su posición en la misma (es decir, su fila y su columna).

Por ejemplo, la matriz dispersa:

```
int[][] matriz = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                   { 0, 0, 2, 0, 0, 0, 0, 0, 0 },
                   { 0, 9, 0, 0, 5, 0, 0, 0, 0 } };
```

que, como ves, tiene 4 filas y 9 columnas, tiene únicamente tres elementos distintos de cero: el valor 2 en la posición $(2, 2)$, el valor 9 en la posición $(3, 1)$ y el valor 5 en la posición $(3, 4)$. Con nuestra clase `MatrizDispersa`, esta matriz quedaría representada de la siguiente manera:




Ten en cuenta que tanto las filas como las columnas se numeran empezando en cero y que el orden de los nodos en la lista no importa.

Siguiendo estas ideas hemos implementado ya parte de la clase `MatrizDispersa`:

```
public class MatrizDispersa {
    private static class Nodo {
        // Atributos de Nodo:
        int valor;
        int fila;
        int columna;
        Nodo siguiente;

        // Constructor de Nodo:
        Nodo(int v, int f, int c, Nodo s) {
            valor = v;
            fila = f;
            columna = c;
            siguiente = s;
        }
    }

    // Atributos de MatrizDispersa:
    private int numFilas;
    private int numColumnas;
    private Nodo primerNodo;

    // Constructor y resto de métodos:
    ...
}
```

Implementa los siguientes métodos públicos de la clase `MatrizDispersa`:

a) `MatrizDispersa(int[][] matriz)` **(1,25 puntos)**

Este constructor tiene como parámetro una matriz nativa de Java y crea el objeto equivalente de la clase `MatrizDispersa`.

b) `int obtenerValor(int fila, int columna)` **(0,5 puntos)**

Devuelve el valor del elemento correspondiente a la posición dada como parámetro (`fila` y `columna`).

Si la posición dada no está dentro del rango válido, el método lanzará una excepción del tipo `ExcepciónFueraDeRango`[1].

Si la posición dada no se encuentra en ningún nodo, el método debe devolver cero (recuerda que sólo se guardan en la lista los valores distintos de cero).

c) `void sumarValor(int valor, int fila, int columna)` **(2 puntos)**

Suma el `valor` dado al elemento de la posición dada como parámetro (`fila` y `columna`).

Si la posición dada no está dentro del rango válido, el método lanzará una excepción del tipo `ExcepciónFueraDeRango`[1].

Si existe un nodo correspondiente con la posición dada, a su atributo `valor` se le sumará el `valor` dado como parámetro. Ten en cuenta que si el resultado de esa suma es cero, el nodo debe ser borrado. Por otro lado, si no existe ningún nodo correspondiente con la posición dada y el `valor` dado no es cero, se deberá crear un nuevo nodo.

d) `void sumar(MatrizDispersa otraMatriz)` **(0,75 puntos)**

Suma dos matrices y almacena el resultado en la primera. Por ejemplo, la llamada `m1.sumar(m2)` debe almacenar en `m1` la suma de `m1` y `m2`.

Recuerda que la suma de matrices requiere que ambas tengan las mismas dimensiones. De no ser así, el método lanzará una excepción del tipo `ExcepciónDimensiónNoVálida`[1].

Como ambas matrices son dispersas, bastará con sumar cada elemento no nulo de `m2` en la posición correspondiente de `m1`. Para esto, debes utilizar el método `sumarValor` del apartado anterior.

[1] Considera que las clases `ExcepciónFueraDeRango` y `ExcepciónDimensiónNoVálida` ya están definidas y son del tipo `RunTimeException`.