Reto 3

Estudiante: Jose Javier Bailón Ortiz
Asignatura: Programación de servicios y procesos
Convocatoria: 14/11/2023

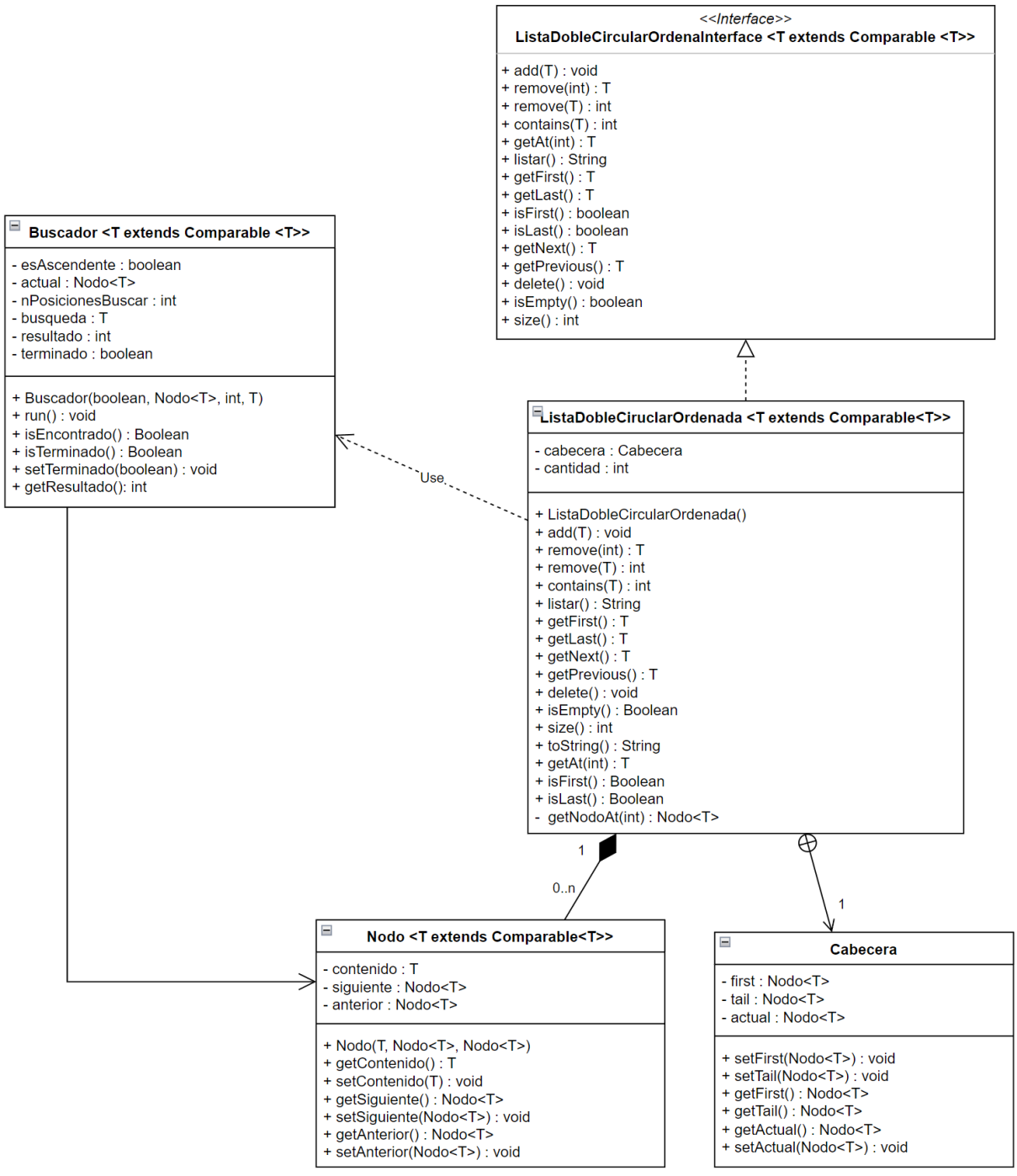
Sumario

A – Lista doblemente enlazada circular ordenada

A.1 Planteamiento del problema

Crear una lista doblemente enlazada circular que tenga la capacidad de buscar un elemento dentro de la lista usando 2 hilos que busquen el elemento buscado encargándose cada hilo de buscar en la mitad de la lista.

A.2 Diagrama de clases



A.3 Estructura de la lista

La lista está implementada en la clase *ListaDobleCircularOrdenada. S*igue las especificaciones de la interfaz ListaDobleCircularOrdenadaInterface y almacena los elementos de la lista en objetos de la clase *Nodo*. Usa dos hilos basados en la clase *Buscador* para la búsqueda de elementos.

Internamente la lista es una sucesión de nodos que tienen referencias al nodo siguiente y al anterior, existiendo ese doble enlace también entre el primer y el último nodo haciéndola circular.

La lista ordena los elementos en el momento de su inserción usando el compareTo del objeto insertado.

Tiene un atributo de la clase *Cabecera* en el que la lista almacena una referencia al primer y al ultimo nodo así como una referencia, a modo de cabeza lectora, que apunta al nodo por el que se va leyendo la lista.

Para recorrer la lista se debe ejecutar primeramente el método getFirst() o getLast(). Con esos métodos se obtiene el primero o el último elemento de la lista. Tras eso se pueden usar los métodos getNext() y getPrevious() para ir avanzando o retrocediendo por la lista. En el momento en el que se usa alguno de esos 4 métodos la cabeza lectora se reposiciona para tener un seguimiento de por donde se va leyendo la lista. Se puede comprobar si se ha llegado al final de la lista ejecutando el método isFirst() o isLast() que devuelven si la posición actual de la cabeza lectora es la primera o ultima.

Por tanto la lista se puede usar de una manera circular indefinidamente o leerla una sola vez al completo en cualquier dirección.

A.4 Búsqueda de elementos con dos hilos

El método **contains(T elemento)** hace una búsqueda de un elemento en la lista.

La lista genera dos hilos de clase *Busqueda* que empiezan a buscar cada uno por un extremo de la lista. Estos hilos van comprobando la existencia del elemento buscado hasta el centro de la lista. El hilo que ha lanzado las dos búsquedas queda esperando en un bucle comprobando si alguno de los hilos buscadores ha terminado. Cuando un hilo buscador ha encontrado el elemento entonces el hilo principal ordena el fin de las búsquedas y su espera termina. Si ambos hilos han terminado sin encontrar el elemento también termina la espera. Cada objeto *Busqueda* almacena el resultado de su búsqueda y el hilo principal de la lista comprueba los mismos para devolver el valor adecuado.

Se puede ver más información respecto al código en el JavaDoc del package *reto3a* y en los comentarios del propio código.

A.5 Ejecutar el código

Versión: Java 21

Paquete: reto3a

Main: reto3a.MainReto3A

Compilación: javac reto3a.MainReto3A.java

Ejecución: java reto3a.MainReto3A

Tras la ejecución se crea una lista y se hacen automáticamente varias operaciones de añadir, listar y borrar para comprobar el funcionamiento de de la lista mostrando en pantalla lo que hace y el estado de la lista tras cada operación.

Tras eso se genera una nueva lista de números enteros aleatorios con la cantidad que elija el usuario y se le mostrará un menú en el que puede elegir qué hacer. Elegir primero, ultimo, ir alante, atrás o buscar.

Se pueden modificar los parámetros de estas pruebas en el método *Main* de la clase *reto3a.MainReto3A*

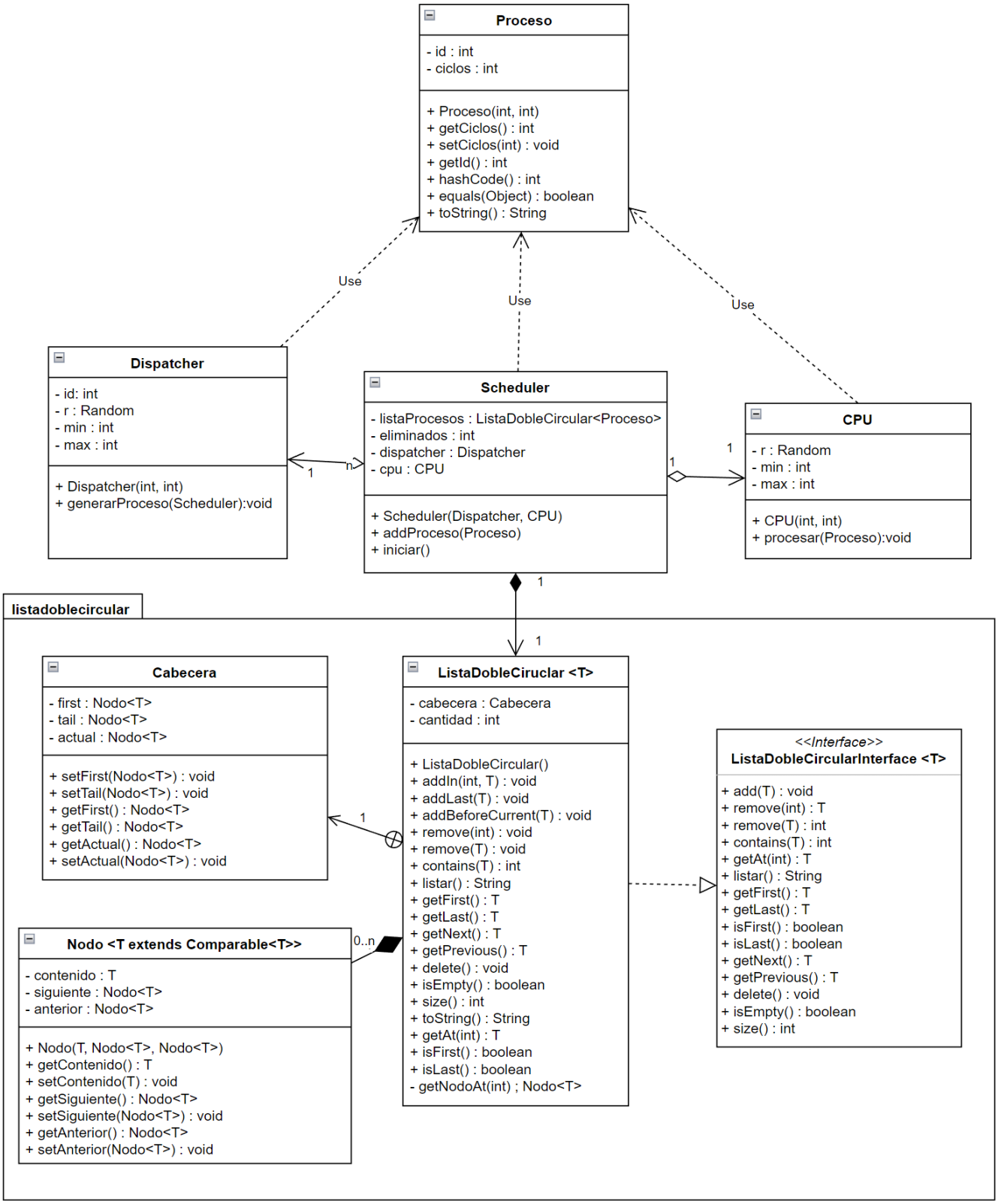
B – Simular consumo de procesos según Round-Robin

B.1 Planteamiento del problema

Simular el consumo de procesos con Round-Robin representado cada proceso con un numero entero que simula su consumo total de ciclos de CPU. La cantidad de ciclos de CPU sera del rango{1000,10000} y cada vez que pasan por CPU consume ciclos de CPU en un rango de {50,200}

- Por cada 100 nodos que terminen de procesarse se agrega 1 nuevo nodo
- Dispatcher:agrega procesos
- Scheduler: controla qué entra en cpu y qué proceso muere por estar terminado
- CPU: consume tiempo del proceso
- Al quedarse la lista de procesos vacía termina el programa

B.2 Diagrama de clases



B.3 Notas sobre la implementación

Las clases principales usadas para implementar la solución son:

- **Proceso:** Representa un proceso. Tiene una id única y unos ciclos de CPU de vida.
- **Dispatcher:** Actúa como creador de procesos y tras crearlos los envía a un *Scheduler.*
- **Scheduler:** Almacena una lista de procesos que luego introduce en una *CPU*
- **CPU:** Se encarga de consumir ciclos de un proceso.
- **ListaDobleCircular:** Es la clase de la lista que utiliza *Scheduler* para almacenar y recorrer los procesos. Esta clase es una lista circular que depende de las otras clases del package *listadocircular.*.
- **MainReto3B:** Es la clase de inicio del programa y se encarga de configurar el estado inicial para luego ordenar el inicio de actividad del Scheduler.

La clase *MainReto3B* al iniciarse genera un objeto *CPU,* otro *Dispatcher,* y otro *Scheduler*. A este ultimo le pasa en su constructor el *Dispatcher* y *CPU* creados para que sea esos los que use.

Tras eso ordena al *Dispatcher* crear 1000 procesos y pasárselos al *Scheduler.* El *Dispatcher* está pensado para que pueda servir procesos a múltiples *Scheduler* aunque en este ejemplo no es el caso.

Después se le ordena a *Scheduler* que inicie su actividad. *Scheduler* va recorriendo la lista circular de *Procesos* que contiene introduciéndolos en la *CPU* la cual consume parte de los ciclos de vida del proceso.Despues pasa al siguiente proceso para repetir esos pasos.

Cuando un proceso sale de *CPU,* el *Scheduler* comprueba si al *Proceso* tiene aún ciclos de vida restantes. Si le quedan ciclos de vida simplemente pasa al siguiente *Proceso* de la lista circular. En caso de que a un *Proceso* no le queden ciclos de vida lo saca de la lista y pide a *Dispatcher* que le envíe un nuevo proceso por cada 100 procesos que haya sacado de la lista. Si se da este caso, Scheduler manda un mensaje a Dispatcher pasándole una referencia a si mismo pidiendo que le envíe un proceso. El nuevo proceso se añadirá en la posición anterior a la cabeza de lectura de la lista circular, quedando así en última posicion respecto al estado actual de ejecución del ciclo a lo largo de la lista.

La lista *reto3b.listadocircular.ListaDobleCircular* es parecida a la lista del apartado A de este reto. Se trata de una lista circular doblemente enlazada que sigue gran parte de lo especificado en la lista del apartado A. En este caso no es ordenada ya que se necesita colocar los elementos en sitios específicos de la lista que no tienen que ver con una posible ordenación del propio objeto a guardar en la lista. Por eso en este caso la lista no tiene opción de añadir y que ya se encargue la lista de colocarlo en su posición, si no que a la hora de añadir el elemento debe especificarse en que indice hay que hacerlo. Puede ser al principio, al final, en un indice concreto o, como es el caso que necesita *Scheduler*, antes de la posición de la cabeza lectora.

Se puede ver más información respecto al código en el JavaDoc de los package reto3b y reto3b.listadoblecircular, y en los comentarios del propio código.

B.4 Ejecutar el código

Versión: Java 21

Paquete: reto3b

Main: reto3b.MainReto3B

Compilación: javac reto3b.MainReto3B.java

Ejecución: java reto3b.MainReto3B

Tras la ejecución se crea automáticamente el estado inicial descrito en el apartado anterior y se ejecuta el consumo de los procesos recibiéndose en consola una retroalimentación de lo que va sucediendo.

Se pueden modificar los parámetros de ejecución en el método *Main* de la clase *reto3b.MainReto3B*