Juan Menduiña

Trabajo Práctico N° 1:
Estructuras de Control y Estructuras de Datos Básicas en Java. Recursión.

Ejercicio 1.

Escribir tres métodos de clase (static) que reciban por parámetro dos números enteros (tipo int) a y b e impriman todos los números enteros comprendidos entre a; b (inclusive), uno por cada línea en la salida estándar. Para ello, dentro de una nueva clase, escribir un método por cada uno de los siguientes incisos:

- *Que realice lo pedido con un for.*
- *Que realice lo pedido con un while.*
- *Que realice lo pedido sin utilizar estructuras de control iterativas (for, while, do while).*

Por último, escribir, en el método de clase main, el llamado a cada uno de los métodos creados, con valores de ejemplo. En la computadora, ejecutar el programa y verificar que se cumple con lo pedido.

Ejercicio 2.

*Escribir un método de clase que, dado un número n, devuelva un nuevo arreglo de tamaño n con los n primeros múltiplos enteros de n mayores o iguales que 1. Ejemplo: f(5)= [5; 10; 15; 20; 25]; f(k)= {n*k, donde k: 1…k}. Agregar al programa la posibilidad de probar con distintos valores de n ingresándolos por teclado, mediante el uso de System.in. La clase Scanner permite leer, de forma sencilla, valores de entrada.*

Ejercicio 3.

Creación de instancias mediante el uso del operador new.

(a) *Crear una clase llamada Estudiante con los atributos especificados abajo y sus correspondientes métodos getters y setters (hacer uso de las facilidades que brinda eclipse).*

- *nombre*
- *apellido*
- *comision*
- *email*
- *direccion*

(b) *Crear una clase llamada Profesor con los atributos especificados abajo y sus correspondientes métodos getters y setters (hacer uso de las facilidades que brinda eclipse).*

- *nombre*
- *apellido*
- *email*
- *catedra*
- *facultad*

(c) *Agregar un método de instancia llamado tusDatos() en la clase Estudiante y en la clase Profesor, que retorne un String con los datos de los atributos de las mismas. Para acceder a los valores de los atributos, utilizar los getters previamente definidos.*

(d) *Escribir una clase llamada Test con el método main, el cual cree un arreglo con 2 objetos Estudiante, otro arreglo con 3 objetos Profesor, y, luego, recorrer ambos arreglos imprimiendo los valores obtenidos mediante el método tusDatos(). Recordar asignar los valores de los atributos de los objetos Estudiante y Profesor invocando los respectivos métodos setters.*

(e) *Agregar dos breakpoints, uno en la línea donde itera sobre los estudiantes y otro en la línea donde itera sobre los profesores.*

(f) *Ejecutar la clase Test en modo debug y avanzar paso a paso visualizando si el estudiante o el profesor recuperado es lo esperado.*

Ejercicio 4.

Pasaje de parámetros en Java.

(a) *Sin ejecutar el programa en la computadora, sólo analizándolo, indicar qué imprime el siguiente código.*

(b) *Ejecutar el ejercicio en la computadora y comparar el resultado con lo esperado en el inciso anterior.*

(c) *Insertar un breakpoint en las líneas donde se indica: y= tmp y ejecutar en modo debug. ¿Los valores que adoptan las variables x, y coinciden con los valores impresos por consola?*

Ejercicio 5.

Dado un arreglo de valores tipo entero, se desea calcular el valor máximo, mínimo y promedio en un único método. Escribir tres métodos de clase, donde respectivamente:

(a) *Devolver lo pedido por el mecanismo de retorno de un método en Java ("return").*

(b) *Devolver lo pedido interactuando con algún parámetro (el parámetro no puede ser de tipo arreglo).*

(c) *Devolver lo pedido sin usar parámetros ni la sentencia "return".*

Ejercicio 6.

Análisis de las estructuras de listas provistas por la API de Java.

(a) *¿En qué casos ArrayList ofrece un mejor rendimiento que LinkedList?*

- Frecuentes accesos aleatorios a los elementos: *ArrayList* usa un *array* dinámico internamente, por lo que acceder a un elemento por índice es O(1), mientras que, en *LinkedList*, es O(n) porque debe recorrer los nodos secuencialmente.
- Pocas inserciones/eliminaciones al inicio o medio de la lista: Si bien *LinkedList* es más eficiente en la eliminación/inserción en estos casos, si estas operaciones no son muy frecuentes, en general, *ArrayList* sigue siendo más rápido debido a su menor sobrecarga en memoria.
- Poco uso de memoria adicional: *ArrayList* almacena sólo los elementos, mientras que *LinkedList* usa referencias adicionales para enlazar nodos.
- Frecuentes operaciones de iteración: La iteración sobre un *ArrayList* es más rápida debido a la contigüidad de la memoria y la mejor localización en caché.

(b) *¿Cuándo LinkedList puede ser más eficiente que ArrayList?*

- Pocos accesos aleatorios a los elementos: Como *LinkedList* no tiene acceso directo a los elementos, sólo es útil si no se necesita acceder frecuentemente por índices.
- Frecuentes inserciones/eliminaciones al inicio o medio de la lista: *LinkedList* tiene O(1) en inserciones/eliminaciones en estos casos, mientras que *ArrayList* tiene O(n) debido al desplazamiento de elementos.
- Uso de *Iterator* para eliminaciones: Al eliminar elementos con un *Iterator*, *LinkedList* tiene un mejor rendimiento, ya que la eliminación es O(1), mientras que, en *ArrayList*, es O(n) por el desplazamiento.

(c) *¿Qué diferencia se encuentra en el uso de la memoria en ArrayList y LinkedList?*

- *ArrayList*: Usa menos memoria porque sólo almacena los elementos en un *array* contiguo, sin punteros adicionales.
- *LinkedList*: Usa más memoria porque cada nodo almacena el dato junto con dos referencias adicionales (*next* y *prev*), lo que aumenta el consumo de memoria significativamente.

(d) *¿En qué casos sería preferible usar un ArrayList o un LinkedList?*

Sería preferible usar un *ArrayList* cuando:

- Se requiere acceso rápido a los elementos por índice.

- Se realizan pocas inserciones/eliminaciones al inicio o medio de la lista.
- Se realizan recorridos de la lista frecuentemente.
- Se prioriza la eficiencia en el uso de memoria.

Sería preferible usar *LinkedList* cuando:

- No se requiere acceso rápido a los elementos por índice.
- Se realizan muchas inserciones/eliminaciones al inicio o medio de la lista.
- Se quiere usar *Iterator* para modificar la lista mientras se recorre.
- No se realizan recorridos de la lista frecuentemente.

En la mayoría de los casos, *ArrayList* es preferible debido a su menor consumo de memoria y mejor rendimiento general, excepto en escenarios con muchas modificaciones al inicio de la lista.

Ejercicio 7.

Uso de las estructuras de listas provistas por la API de Java. Para resolver este ejercicio, crear el paquete tp1.ejercicio7.

(a) *Escribir una clase llamada TestArrayList cuyo método main recibe una secuencia de números, los agrega a una lista de tipo ArrayList y, luego de haber agregado todos los números a la lista, imprime el contenido de la misma iterando sobre cada elemento.*

(b) *Si en lugar de usar un ArrayList en el inciso anterior se hubiera usado un LinkedList, ¿qué diferencia se encuentra respecto de la implementación? Justificar.*

Si en lugar de usar un *ArrayList* en el inciso anterior se hubiera usado un *LinkedList*, el acceso a los números sería muy ineficiente, ya que hay que recorrer la lista, no es directo.

(c) *¿Existen otras alternativas para recorrer los elementos de la lista del inciso (a)?*

Sí, existen otras alternativas para recorrer los elementos de la lista del inciso (a).

(d) *Escribir un método que realice las siguientes acciones:*

- *Crear una lista que contenga 3 estudiantes.*
- *Generar una nueva lista que sea una copia de la lista anterior.*
- *Imprimir el contenido de la lista original y el contenido de la nueva lista.*
- *Modificar algún dato de los estudiantes.*
- *Volver a imprimir el contenido de la lista original y el contenido de la nueva lista. ¿Qué conclusiones se obtiene a partir de lo realizado?*
- *¿Cuántas formas de copiar una lista existen? ¿Qué diferencias existen entre ellas?*

(e) *A la lista del inciso (d), agregar un nuevo estudiante. Antes de agregar, verificar que el estudiante no estaba incluído en la lista.*

(f) *Escribir un método que devuelva verdadero o falso si la secuencia almacenada en la lista es o no capicúa: public boolean esCapicua(ArrayList<Integer> lista).*

(g) *Considerar que se aplica la siguiente función de forma recursiva. A partir de un número n positivo se obtiene una sucesión que termina en 1:*

$$f(n) = \begin{cases} \frac{n}{2}, si\ n\ es\ par \\ 3n + 1, si\ n\ es\ impar \end{cases}.$$

Escribir un programa recursivo que, a partir de un número n, devuelva una lista con cada miembro de la sucesión.

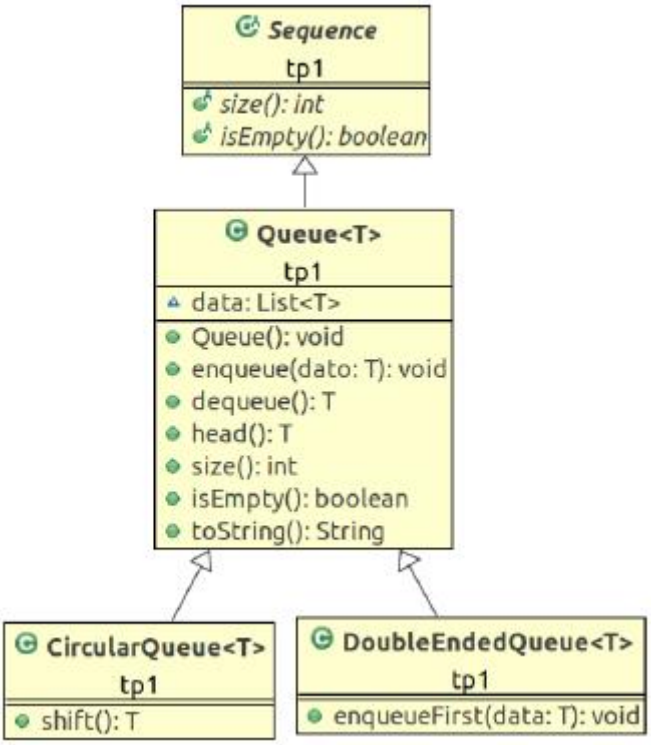
(h) *Implementar un método recursivo que invierta el orden de los elementos en un ArrayList: public void invertirArrayList(ArrayList<Integer> lista).*

(i) *Implementar un método recursivo que calcule la suma de los elementos en un LinkedList: public int sumarLinkedList(LinkedList<Integer> lista).*

(j) *Implementar el método "combinarOrdenado" que reciba 2 listas de números ordenados y devuelva una nueva lista también ordenada conteniendo los elementos de las 2 listas: public ArrayList<Integer> combinarOrdenado(ArrayList<Integer> lista1, ArrayList<Integer> lista2).*

Ejercicio 8.

El objetivo de este punto es ejercitar el uso de la API de listas de Java y aplicar conceptos de la programación orientada a objetos. Sean las siguientes especificaciones de cola, cola circular y cola con 2 extremos disponibles, vistas en la explicación teórica:



(a) *Implementar en JAVA la clase Queue de acuerdo con la especificación dada en el diagrama de clases. Definir esta clase dentro del paquete tp1.ejercicio8.*

- *Queue(): Constructor de la clase.*
- *enqueue(dato: T): Inserta el elemento al final de la cola.*
- *dequeue(): T Elimina el elemento del frente de la cola y lo retorna. Si la cola está vacía, se produce un error..*
- *head(): T Retorna el elemento del frente de la cola. Si la cola está vacía, se produce un error.*
- *isEmpty(): boolean Retorna verdadero si la cola no tiene elementos y falso en caso contrario.*
- *size(): int Retorna la cantidad de elementos de la cola.*
- *toString(): String Retorna los elementos de la cola en un String.*

(b) *Implementar en JAVA la clase CircularQueue de acuerdo con la especificación dada en el diagrama de clases. Definir esta clase dentro del paquete tp1.ejercicio8.*

- *shift(): T Permite rotar los elementos, haciéndolo circular. Retorna el elemento encolado.*

(c) *Implementar en JAVA la clase DoubleEndedQueue de acuerdo con la especificación dada en el diagrama de clases. Definir esta clase dentro del paquete tp1.ejercicio8.*

- *enqueueFirst(): void Permite encolar al inicio.*

Ejercicio 9.

Considerar un string de caracteres S, el cual comprende, únicamente, los caracteres: (,), [,], {, }. Se dice que S está balanceado si tiene alguna de las siguientes formas:

- *S = "" S es el string de longitud cero.*
- *S = "(T)".*
- *S = "[T]".*
- *S = "{T}".*
- *S = "TU".*

Donde ambos T y U son strings balanceados. Por ejemplo, "{()[()]}" está balanceado, pero "([)]" no lo está.

(a) *Indicar qué estructura de datos se utilizará para resolver este problema y cómo se utilizará.*

Se utilizará la estructura de datos *"Stack"*. Por cada signo de apertura, se hará un *PUSH* y, por cada signo de cierre, se hará un *POP*, verificando que sean del mismo tipo (el abrir y cerrar de signo). Si la pila queda vacía, significa que el *String* está balanceado. Además, se tendrán dos listas enlazadas, una con los caracteres de apertura y otra con los caracteres de cierre, para ir viendo si el caracter actual corresponde con alguno de esos seis caracteres y, dependiendo de eso, se realizará la operación correspondiente (*push* o *pop*).

(b) *Implementar una clase llamada tp1.ejercicio9.TestBalanceo, cuyo objetivo es determinar si un String dado está balanceado. El String a verificar es un parámetro de entrada (no es un dato predefinido).*

Ejercicio 10.

Considerar el siguiente problema: Se quiere modelar la cola de atención en un banco. A medida que la gente llega al banco, toma un ticket para ser atendido, sin embargo, de acuerdo a la LEY 14.564 de la Provincia de Buenos Aires, se establece la obligatoriedad de otorgar prioridad de atención a mujeres embarazadas, a personas con necesidades especiales o movilidad reducida y a personas mayores de setenta (70) años. De acuerdo a las estructuras de datos vistas en esta práctica, ¿qué estructura de datos se sugeriría para el modelado de la cola del banco?

Para el modelado de la cola del banco, la estructura de datos que se sugeriría es "DoubleEndedQueue", es decir, una cola de doble extremo, ya que esta estructura permite agregar tanto al principio como al final de la cola, con lo cual las personas con orden de prioridad serán puestas al principio de la cola, mientras que las personas que no tengan este orden serán puestas al final de la cola.

Ejercicio 11.

Considerar el siguiente problema: Se quiere modelar el transporte público de la ciudad de La Plata, lo cual involucra las líneas de colectivos y sus respectivas paradas. Cada línea de colectivos, tiene asignado un conjunto de paradas donde se detiene de manera repetida durante un mismo día. De acuerdo a las estructuras de datos vistas en esta práctica, ¿qué estructura de datos se sugeriría para el modelado de las paradas de una línea de colectivos?

Para el modelado de las paradas de una línea de colectivos, la estructura de datos que se sugeriría es “CircularQueue”, es decir, una cola circular, ya que esta estructura permite que, cuando un colectivo llega a la última parada, automáticamente, vuelve a la primera, sin necesidad de reiniciar la estructura.