Algoritmos y Estructura de datos III - Año 2018 – 1er semestre
Prof. Cristian Cappo
Clase #3 - 22-FEBRERO-2018
Ejercitario para Laboratorio (*Desarrolle con su compañero de grupo (el definido en EDUCA)*).

Cerciorarse que posee instalado el compilador e intérprete de Java. Todos los ejercicios deben hacerse desde la linea de comandos (la compilación y la ejecución). Para ello configure en su entorno la variable PATH agregando el camino donde se encuentra la instalación del compilador de Java. Puede utilizar la guía de configuración del entorno que se encuentra en EDUCA.

Una vez que haya terminado sus ejercicios use EDUCA para alzarlos. Coloque el nombre de los integrantes en cada archivo de solución. Comprima los archivos y solo alce los archivos fuente (.java). Recuerde que el archivo comprimido debe tener como prefijo el identificador de su grupo. Solo se considerará a los presentes.

```
Para compilar : c:> javac <Nombre_archivo_fuente_con_extension_java>
Para ejecutar : c:> java <Nombre_Clase>
```

Para editar los programas fuentes puede utilizar cualquier editor de texto. Un posible editor puede bajarlo del propio sitio de la asignatura en EDUCA (Notepad++).(Obtener de: http://notepad-plus-plus.org)

Información importante:

Descripción del lenguaje Java (Especificación del lenguaje)	http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf (versión 7) http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (versión 8) http://docs.oracle.com/javase/specs/jls/se9/jls9.pdf (versión 9)
Documentación del API de Java	http://docs.oracle.com/javase/7/docs/api/ (versión 7) http://docs.oracle.com/javase/8/docs/api/ (versión 8) http://docs.oracle.com/javase/9/docs/api/ (versión 9)

Una breve ayuda sobre Generic puede encontrar en EDUCA tomada del libro de "Java How to Program, 7/e " Deitel & Deitel

*En Java 9 tiene la posibilidad de utilizar la herramienta **jshell** que le permite definir y ejecutar código java en forma interactiva. Mas info en https://docs.oracle.com/javase/9/tools/language-shell.htm. Es lo que vimos en clase.*

EJERCICIO 0 (para probar el ambiente, el compilador y otras cuestiones del entorno)

Configure el entorno de compilación y pruebe el siguiente código en Java. Recuerde que el nombre de la clase debe ser *Ejercicio0.java (el mismo nombre de la clase).*

```
public class Ejercicio0 {
      public static void main ( String [] args ) {
          System.out.println("Hola Algoritmos III");
      }
}
```

Suele existir varias versiones del mismo compilador, asegurarse que la versión del compilador coincida con la del interprete. Para ver la versión simplemente escriba "*javac - version*" para el compilador y "*java -version*" para el intérprete.

EJERCICIO 1

a) Compile y ejecute el siguiente ejercicio sencillo en Java. Imprime cada parámetro leído de la linea de comandos en una linea separada (solo en el caso de haberse pasado al menos un parámetro).

```
public class Test  {
      public static void main ( String [] args ) {
         if ( args.length > 0 ) {
            for ( int k=0; k < args.length ; k++ )  {
                System.out.printf(">> %s\n", args[k]);
            }
         }
      }
}
c:> javac Test.java
c:> java Test uno dos tres
>> uno
>> dos
>> tres
```

b) Ahora modifique el ejercicio de ***a)*** para que ahora, controlando a través del mecanismo de **Manejo de excepción (**vea las diapositivas de la clase pasada – Parte VII **)**, todos los parámetros deben ser números enteros. Imprimir un mensaje cuando algún parámetro de la linea de comandos no lo sea. Luego imprimir de forma inversa la lista de parámetros. El efecto debería ser:

```
c:> java Test
No se pasó ningún parámetro por la linea de comandos.

c:> java Test 1 2
2 1

c:> java Test 1 2 3 4 5
5 4 3 2 1

c:> java Test 1 2 3 4 5 algoritmo
Todos los parámetros deben ser numéricos
```

TIP: para saber que nombre de excepción utilizar puede generar la excepción y de ahí extraer la información requerida. Para conversión de cadena a entero utilice el método *parseInt()* de la clase *Integer*.

¿Qué debió comprender con este ejercicio?

Como compilar un programa Java.
Como funciona la captura de parámetros desde la linea de comandos.
Como funciona la captura de excepciones, al menos en su forma básica.

EJERCICIO 2 - Definición de clase

Baje el fuente denominado *CtaBancaria.java* que modela una cuenta bancaria sencilla, permite mantener el saldo, información básica e imprimir el movimiento de la misma. Lea el código y los comentarios luego compile el código.

El objetivo es que pueda observar la sencillez de definición de un nuevo tipo de dato.

Conteste las preguntas que están dentro del código. Las respuestas en el mismo codigo.

EJERCICIO 3 – Uso de *interface*

Baje el conjunto de fuentes denominado "*ejemplo-interface*" de EDUCA. Notará que es el ejemplo mostrado en las diapositivas de la clase pasada.

a) Ahora intente compilar *UsoCiudad.java* . ¿Donde esta y cuál es el error? . ¿Qué falta? Probablemente se necesita especificar exactamente en que paquete se encuentra la clase *ArrayList.* Utilice la documentación del API (https://docs.oracle.com/javase/9/docs/api/) para encontrar el lugar exacto donde se encuentra la clase *ArrayList*. Una vez que lo encontró puede comprobar con la herramienta *javap* si la clase existe en el lugar indicado.

```
c:> javap <nombre-clase>
```

Una completa referencia de las herramientas puede encontrar en https://docs.oracle.com/javase/9/tools/main-tools-create-and-build-applications.htm. Más información de ***javap*** puede encontrar en https://docs.oracle.com/javase/9/tools/javap.htm. Se explica como se buscan las clases, como se configura el camino de búsqueda, la estructura de archivo de las librerías, los comandos, y otras cuestiones más complejas. (Por supuesto, todo esta en inglés).

b) Corrija *Ciudad.java* e intente compilar de vuelta *UsoCiudad.java.* Ejecute el código.
c) Agregue ahora una nueva clase *Carrito* que implemente la interface EmiteRuido.
d) En *UsoCiudad.java* cree dos instancias de *Carrito* y ejecute el código.

¿Qué debió comprender con este ejercicio?

¿Como se define una clase?
¿Para qué es útil la herramienta *javap y jar*? ¿Para qué sirve la claúsula *import* ?
¿Qué se gana utilizando interfaces?
¿Cómo funciona la compilación en Java cuando existen varios fuentes relacionados? ¿Se recuerda como es esto en C?

EJERCICIO 4 – Tipo Abstracto de Dato, Generic e Iteradores

Complete la clase *ListaEnlazadaSimple* cuyo fuente fue mostrado en clase, y cuyo dato es un entero que se indica en la diapositiva de la clase pasada. Trate de entender el código y antes que nada compilarlo y ejecutarlo. Conteste las preguntas que se encuentren en el código, responda debajo de cada una.

Implemente:

a) La operación de ***agregar_ordenado*** ()

b) La operación de ***existe***. Debe retornar *true* o *false* de acuerdo a que el elemento se encuentre o no en la lista.

c) La operación ***eliminar*** *() que elimine todos los elemento con el valor dado.*

d) Una clase (no la misma que *ListaEnlazadaSImple*) que utilice *ListaEnlazadaSimple*, acepte algunos datos de la linea de comandos y los liste de acuerdo a como fueron introducidos a la misma. Estos datos deben ser numéricos.

e) Agregar la posibilidad de que sea *Generic* (vea el ejemplo en la diapositiva mostrada en la clase y el material que es proveído en Educa). Para ello cree una nueva clase *ListaEnlazadaSimpleG* que tenga la posibilidad de ser utilizada para cualquier tipo de dato.

f) Hacer que funcione la siguiente expresión de Java (implica uso de *java.util.iterator*) (ver la breve descripción de como utilizar el Iterador del API de java al final de este documento).

```
for (int i : lista )
 System.out.println(i); /* Imprime cada elemento de lista */
```

¿Qué debió comprender con este ejercicio?

Como definir un sencillo TAD
Como definir un TAD Genérico.
Como funciona y se implementa un iterador.

EJERCICIO 5 – Más sobre TAD

Desarrolle una clase que implemente la clase Pila con las operaciones de apilar(), desapilar(), tope(), esVacia(), size().

Hágalo de dos formas posibles.

Para ello implemente una interface *Pila* y dos implementaciones, una con arreglo y otra con una lista enlazada (utilizar el código de *ListaEnlazadaSimple*).

```
public interface Pila {
      ...
}

public class PilaArreglo implements Pila{
      ...
}

public class PilaLista implements Pila{
      ...
}
```

Tarea #2 (A entregar el jueves 1 de marzo vía EDUCA, hasta las 23:55 hs)

Ejercicios prácticos sobre construcción de clases en Java. Cada ejercicio en un subdirectorio separado. En los fuentes debe indicar el nro. de tarea, la fecha de entrega y los integrantes. ***TP = 30***

---

1. (10p) 1.a) Complete la implementación del ejercicio número 5 y haga que la clase sea genérica.
   Además utilice excepciones para todo el control de errores (pila vacía, pila llena, etc).

   1.b) Como ejercicio de uso de esta clase Pila (*cualquier implementación*), construya una clase que verifique el balanceo de paréntesis, llaves o corchetes. El balanceo consiste en que por cada paréntesis, llave o corchete derecho se debe tener el correspondiente izquierdo. La clase a ser desarrollada debe estar separada de la clase Pila.

   La clase debe recibir una cadena (sin espacios) que contenga solo como caracteres paréntesis, corchetes o llaves izquierdo y/o derecho. Luego debe informar si el balanceo es correcto o no, indicar en que posición (de la forma que se muestra en el ejemplo) ocurrió el error (solo la primera posición).

   Ejemplos:

```
a){[]})     balanceo incorrecto (se indica el error con ^ )
      ^
b){()}      balanceo correcto

c)[()       balanceo incorrecto (se indica el error con ^ )
  ^
d){}{}]     balanceo incorrecto (se indica el error con ^ )
       ^
```

   1.c) Ahora, en otra clase separada, denominada *PilaC* que solo acepta elementos de la interface *Comparable*, aparte de los métodos usuales de Pila, agregue dos métodos denominados *max()* y *min()* que debe retornar el máximo y el mínimo de la Pila pero realizando un número constante de operaciones, es decir, que no dependa de la cantidad de elementos.

2. (10p) Implemente el ejercicio nro. 2 de la tarea #1 (Al menos implemente un TAD propio) con las mismas funcionalidades solicitadas en ese ejercicio.

3. (10p) Implemente los siguientes TAD en JAVA:

   3.a) ***ColaDoble* :** es una TAD que permite insertar o remover elementos del comienzo o del final de la lista de elementos. Las siguientes operaciones deben implementarse:
   - *push(x)*: inserta el elemento x en el frente de la ColaDoble
   - *pop()*: remueve el elemento del frente de la ColaDoble y lo retorna
   - *inyecta(x)*: inserta el elemento x al final de la ColaDoble
   - *eyectar()*: remueve el elemento del final de la ColaDoble y lo retorna.
   - size(): retorna la cantidad de elementos de la ColaDoble

   3.b) **Conjunto** con las operaciones de agregar, eliminar, listar, unión e intersección.

   Por cada TDA debe proveerse :

   a) Una interface y una implementación que no tiene que restringir arbitrariamente el número de elementos

```
public interface Conjunto {
 ...
}

public class ConjuntoA implements Conjunto{
 ...
}
```

   b) Utilizar *Generic* en todos los casos. De esta manera se debe soportar cualquier tipo de dato.

   c) Debe agregar las excepciones necesarias para el manejo de errores. Derive las excepciones de *RuntimeException (¿Cuál es la diferencia con derivar de Exception?)*

   d) Ambos TDA deben ser iterables vía Iterator del API de Java. Esto permite listar los componentes de esta forma:

```
Cola<String> cs = new Cola<String>();
cs.agregar("Hola");
cs.agregar("Que tal");
cs.agregar("Como estan");

for (String s : cs )
    System.out.println(s); /* Imprime cada elemento de la cola cs */
```

   e) Debe proveerse dos clases de uso de *ColaDoble* (UsoColaDoble) y de *Conjunto* (UsoConjunto) donde se encuentren todas las pruebas posibles de funcionalidad solicitados por cada TDA.

Anexo - Iteración en Java – ejemplo

A continuación tenemos la clase Cola con un iterador implementado :

```
public class Cola <Item> implements Iterable <Item> {
    private int N;
    private Nodo first;
    private Nodo last;

    private class Nodo {
       private Item dato;
       private Nodo next;
    }

    public Iterator<Item> iterator()  {
        return new ListIterator();
    }

    private class ColaIterator implements Iterator<Item> {
        private Node current = first;

        public boolean hasNext()  { return current != null;                     }
        public void remove()      { throw new UnsupportedOperationException();  }

        public Item next() {
            if (!hasNext()) throw new NoSuchElementException();
            Item item = current.item;
            current = current.next;
            return item;
        }
    }

}
```

Algunas veces el cliente necesita acceder a todos los elementos de una colección (a la cola en este ejemplo), uno a la vez, sin borrarlos. De forma a mantener el principio de encapsulación, necesitamos no revelar la representación interna de la Cola (basada en arreglos o en lista enlazada) al cliente. De esta manera necesitamos desacoplar las cosas necesarias para recorrer la lista, de los detalles de obtener cada elemento de ella. Resolvemos este desafío utilizando la interface de Java denominada: *java.util.Iterator* que mostramos a continuación:

```
public interface Iterator<Item> {
    boolean hasNext();
    Item next();
    void remove();      // opcional
}
```

Esto es, cualquier tipo de dato que implementa la interface *Iterator* "promete" implementar dos métodos: *hasNext()* y *next()*. El cliente utiliza estos métodos para acceder a la lista de elementos uno a la vez utilizando la siguiente estrategia:

```
Cola<String> queue = new Cola<String>();
...
Iterator<String> i = queue.iterator();
while (i.hasNext()) {
   String s = i.next();
   System.out.println(s);
}
```

El código de la clase Cola ilustra como implementar un *Iterator* cuando los elementos son almacenados en una lista enlazada. Esto descansa en una clase anidada privada denominada *ColaIterator* que implementa la interface *Iterator*. El método *iterator()* crea una instancia del tipo *ColaIterator* y lo retorna como un *Iterator*. Esto fuerza a la abstracción de la iteración (puesto que el cliente no conoce la implementación de *ColaIterator*) y así el cliente solo accede a los elementos a través de los métodos *hasNext()* y *next()*. El cliente no tiene acceso a la parte interna de *Cola* y menos a *ColaIterator*. La única responsabilidad del cliente es NO agregar elementos a la lista mientra el iterador se encuentre funcionando.

Mejora para la iteración. Java provee una forma más compacta para esta iteración, conocida como *enhanced for loop*. Esta forma compacta permite recorrer los elementos de una colección o de un arreglo.

```
Iterator<String> i = c.iterator();
while (i.hasNext()) {
   String s = i.next();
   System.out.println(s);
}

for (String s : queue)
    System.out.println(s);
```

Para tomar ventaja de esta forma compacta de iteración, el tipo de dato debe implementar la interface *Iterable*.

```
public interface Iterable<Item> {
    Iterator<Item> iterator();
}
```

Esto es, el tipo de dato debe implementar un método llamado iterator() que retorna un Iterator creada por la colección en cuestión. Debido que nuestro TAD Cola ahora incluye tal método, simplemente necesitamos declararla como implementando la interface Iterable y estamos listos para utilizar la notación de "*foreach*".

```
public class Cola<Item> implements Iterable<Item>
```