Ficha 7

Serialización – Listas sobre arreglos

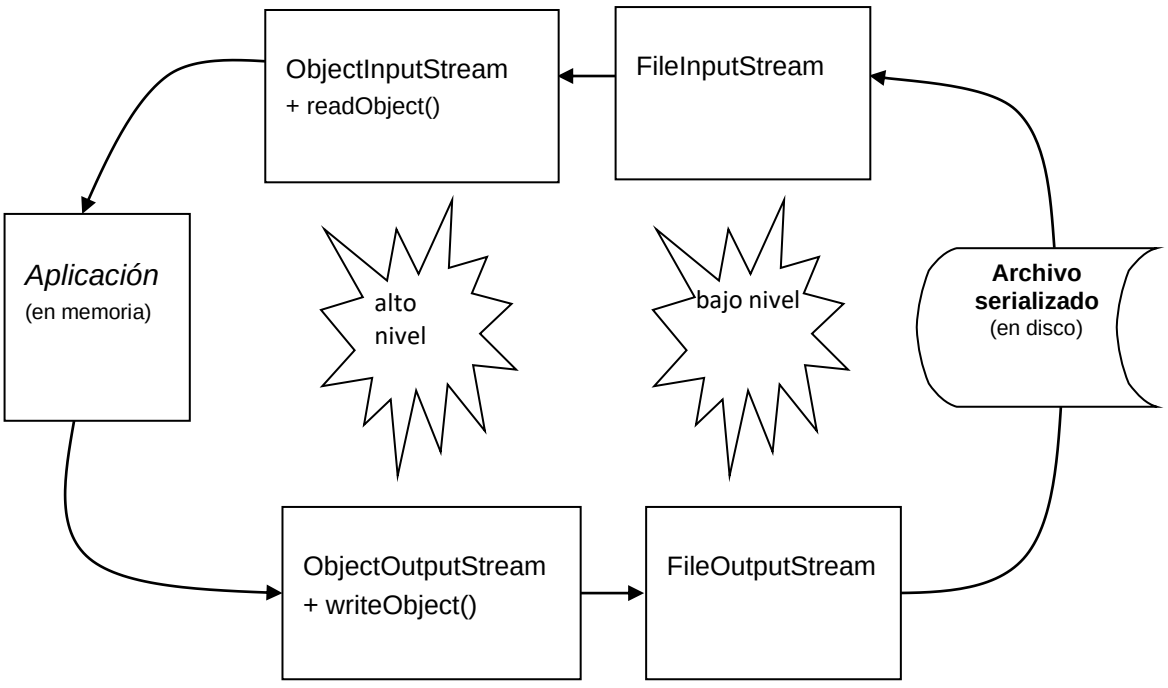
1.] Grabación y recuperación de objetos desde archivos externos en Java: Serialización.

El lenguaje Java dispone de un mecanismo simple de implementar, que permite grabar o recuperar objetos complejos desde archivos almacenados en dispositivos externos invocando a un solo método (y eventualmente capturando excepciones), sin importar la complejidad interna del objeto grabado. Ese mecanismo se conoce como *Serialización*.

Para implementar la serialización, el paquete *java.io* brinda un par de clases *ObjectInputStream* y *ObjectOutputStream*, las cuales permiten leer y grabar objetos completos, respectivamente, sin que el programador deba preocuparse de como "serializar" ese objeto (o sea, el programador no debe preocuparse de cómo se convierte el objeto en una serie de bytes que se envían o se toman del dispositivo externo). Los metodos *readObject()* (de la clase *ObjectInputStream*) y *writeObject()* (de *ObjectOutputStream*) hacen la lectura y la grabación en una sola instruccion (de alto nivel) del programador.

Esas dos clases son de *alto nivel*, y se conectan con otras dos clases de *bajo nivel*: *FileInputStream* y *FileOutputStream*. Ambas clases trabajan leyendo y enviando bytes desde y hacia el dispositivo.

El esquema gráfico que sigue, muestra la relación conceptual entre estas clases:



La forma de crear objetos para lograr la comunicación se muestra en el siguiente ejemplo. Suponemos que lo que se desea grabar es un objeto completo de la clase *TSBSimpleList*, que se toma como parámetro, pero el ejemplo sería el mismo si la clase del objeto a grabar fuera cualquier otra (luego veremos qué requisitos debe cumplir esa otra clase) Notar que es práctica común crear primero un objeto de la clase *File* a modo de "file descriptor", y luego crear el resto de los objetos:

Streams de salida (serialización):

```
public void grabar (TSBSimpleList sl) throws FileNotFoundException, IOException
{
	// creación de objetos para gestión de streams de salida
	File f = new File("lista.dat") ;
	FileOutputStream out  = new FileOutputStream(f);
	ObjectOutputStream ofile = new ObjectOutputStream(out);

	// grabación de lista completa en el archivo serializado
	ofile.writeObject(sl);

	// cierre de los flujos de salida
	ofile.flush();
	out.close();
}
```

Note que el uso de los métodos de estas clases de salida puede lanzar excepciones de la familia de *IOException*, y que esas excepciones son chequeadas. Por lo tanto, el método *grabar()* aquí mostrado debe declarar en su cabecera las posibles excepciones disparadas por los constructores, por *writeObject()* o por los métodos de cierre, o bien usar *try – catch()* para capturarlas.

Streams de entrada (serialización):

```
public TSBSimpleList leer()throws FileNotFoundException, IOException,
ClassNotFoundException
{
	// creación de objetos para gestión de streams de entrada
	File f = new File("lista.dat");
	FileInputStream in = new FileInputStream(f);
	ObjectInputStream ifile = new ObjectInputStream(in);

	// lectura del objeto que estaba en el archivo
	TSBSimpleList sl = (TSBSimpleList) ifile.readObject();

	//cierre de los flujos de entrada
	ifile.close();
	in.close();

	//retorno del objeto leído
	return sl;
}
```

También aquí los métodos usados pueden lanzar excepciones chequeadas y deben ser declaradas en la cabecera del método *leer()* o ser capturadas con *try - catch()*. El método usado para leer un objeto que estaba grabado en el archivo serializado, es *readObject()* (de la clase *ObjectInputStream*). Este método recupera el objeto, pero retorna una referencia a él de tipo *Object*: por lo tanto, debe usarse *casting explícito* a una referencia del tipo

correcto (*TSBSimpleList* en este caso) para poder asignar ese objeto en una referencia del tipo que corresponda.

Sin embargo, el sólo uso de esas clases de IO no basta para poder leer o grabar objetos desde un stream. Las clases cuyos objetos vayan a ser leidos o grabados en streams en forma directa, deberan implementar la interface *Serializable* del paquete *java.io*. Esa clase de interface no pide ningún método: la clase sólo debe indicar que la implementa. Una interface que no pide ningún método se suele conocer como una *interface de marcado* o *interface vacía*: sirve para indicarle a la JVM que la clase que la implementa cumple con cierta propiedad (en el caso de *Serializable*, avisa que el objeto debe ser dividido en sus atributos y grabado "pieza por pieza"). En definitiva, la implementación de la interface *Serializable* sirve a los efectos de indicar semánticamente que una clase está autorizada a ser serializable, y con ello participar de contextos polimórficos que asuman que una clase puede serializarse. La clase *TSBSimpleList* entonces debería definirse así:

```
import java.io.*;
public class TSBSimpleList implements Serializable
{
      private TSBNode frente;

      // resto del contenido de la clase aquí…
}
```

Notar que si una clase es *Serializable*, y esa clase tiene a su vez un atributo de otra clase, para que ese atributo sea a su vez grabado la clase del mismo también debe implementar *Serializable*. En el modelo *TSB-Serializable* que acompaña a esta ficha, la clase *TSBSimpleList* se marca como *Serializable*, y tambien es *Serializable* la clase *TSBNode* (cuyas instancias seran guardadas en objetos de la clase *TSBSimpleList*):

```
import java.io.*;
public class TSBNode implements Serializable
{
      private TSBNode next;
      private Comparable info;

      // resto del contenido de la clase aquí…
}
```

En el mismo modelo TSB-Serializable, en cada nodo se guardarán objetos de la clase *Persona*, y por lo tanto esta última también implementa *Serializable*:

```
import java.io.*;
public class Persona implements Serializable
{
      private String nombre;
      // resto del contenido de la clase aquí…
}
```

Note finalmente, que muchas (casi todas en realidad) de las clases nativas de Java implementan *Serializable*, con lo cual el programador no debe preocuparse si una de sus

clases propias tiene atributos de alguna clase nativa. La clase *Persona* ya citada tiene un atributo *nombre* de tipo *String*, pero como *String* implementa *Serializable* no habrá problema cuando se grabe un objeto *Persona*.

El proyecto *TSB-SimpleList* encapsula el proceso de grabación en una clase *TSBSimpleListWriter*, y el de lectura en otra clase *TSBSimpleListReader*. Tanto el método *write()* de la primera, como el método *read()* de la segunda, controlan el lanzamiento de excepciones de IO, y si alguna aparece la cambian por una excepción propia de la clase *TSBSimpleListIOException*. El método *main()* captura esta excepción con *try - catch()*. Mostramos el código fuente y dejamos el análisis del mismo para el estudiante:

```
/**
 Representa situaciones de error en IO en la serializacion de
 objetos de la clase TSBSimpleList. Estos errores seran casi siempre
 debidos a intentos de Serializacion fallidos, pero puede usarse en
 cualquier otra situacion que implique IO por algun dispositivo
 externo.
 @author Ing. Valerio Frittelli.
 @version Septiembre de 2017.
 */
public class TSBSimpleListIOException extends Exception
{
       private String message = "Problema al serializar la lista";

       /**
        * Inicializa el objeto de excepcion con valores por default.
        */
        public TSBSimpleListIOException()
        {
        }

       /**
        * Inicializa el mensaje a retornar con getMessage().
        * @param msg el mensaje que sera retornado con getMessage().
        * @see getMessage().
        */
        public TSBSimpleListIOException(String msg)
        {
             message = msg;
        }

       /**
        * Retorna una descripcion del error que provocó la excepcion.
        * @return una cadena con la descripcion del error.
        */
        @Override
        public String getMessage()
        {
          return message;
        }
}
```

```
/**
 * Una clase usada para grabar objetos de la clase TSBSimpleList
 * mediante Serializacion.
 * @author Ing. Valerio Frittelli.
 * @version Septiembre de 2017.
 */
import java.io.*;
public class TSBSimpleListWriter
{
      // nombre del archivo serializado...
      private String arch = "lista.dat";

      /**
       Crea un objeto TSBSimpleListWriter. Supone que el nombre del
        archivo a grabar sera "lista.dat".
       */
      public TSBSimpleListWriter()
      {
      }

      /**
       Crea un objeto SimpleListWriter. Fija el nombre del archivo
        que se graba con el nombre tomado como parametro.
       @param nom el nombre del archivo a grabar.
       */
      public TSBSimpleListWriter(String nom)
      {
            arch = nom;
      }

      /**
       * Graba la lista tomada como parametro.
       * @param sl la lista a serializar.
       * @throws TSBSimpleListIOException si hay un error de IO.
       */
      public void write (TSBSimpleList sl) throws
                                          TSBSimpleListIOException
      {
           try
           {
             FileOutputStream ostream = new FileOutputStream(arch);
             ObjectOutputStream p = new ObjectOutputStream(ostream);
             p.writeObject(sl);
             p.flush();
             ostream.close();
           }
           catch ( Exception e )
           {
             throw new TSBSimpleListIOException("Error...");
           }
      }
}
```

```
/**
 Clase que permite recuperar desde un archivo externo un objeto de
 la clase TSBSimpleList que haya sido grabado por Serializacion.
 @author Ing. Valerio Frittelli.
 @version Septiembre de 2017.
 */
import java.io.*;
public class TSBSimpleListReader
{
      private String arch = "lista.dat";

      /**
       Crea un objeto SimpleListReader. Asume que el nombre del
        archivo desde el cual se recupera es "lista.dat".
       */
      public TSBSimpleListReader()
      {
      }

      /**
       Crea un objeto SimpleListReader. Fija el nombre del archivo
        desde el cual se recupera con el nombre tomado como
        parametro.
       @param nom el nombre del archivo a abrir.
       */
      public TSBSimpleListReader(String nom)
      {
          arch = nom;
      }


      /**
       Recupera una SimpleList desde un archivo serializado.
       @throws TSBSimpleListIOException si hay un error de IO.
       @return una referencia al objeto recuperado.
       */
      public TSBSimpleList read() throws TSBSimpleListIOException
      {
           TSBSimpleList sl = null;

           try
           {
             FileInputStream istream = new FileInputStream(arch);
             ObjectInputStream p = new ObjectInputStream(istream);
             sl = ( TSBSimpleList ) p.readObject();
             p.close();
             istream.close();
           }
           catch (Exception e)
           {
             throw new TSBSimpleListIOException("Error…");
           }
```

```
            return sl;
        }
}

import java.util.Scanner;
public class Principal
{
   private static TSBSimpleList<Persona> sl;

   static public void grabar()
   {
       sl = new TSBSimpleList<>();
       Persona a,b,c;
       a = new Persona("Juan", 20);
       b = new Persona("Luis", 30);
       c = new Persona("Ana", 40);
       sl.addFirst(a);
       sl.addFirst(b);
       sl.addFirst(c);
       System.out.println(sl);

       try
       {
           TSBSimpleListWriter slw = new TSBSimpleListWriter();
           slw.write( sl );
       }
       catch(TSBSimpleListIOException e)
       {
            System.out.println("Error: " + e.getMessage());
       }
   }

   static public void leer()
   {
       try
       {
           TSBSimpleListReader slr = new TSBSimpleListReader();
           sl = (TSBSimpleList<Persona>) slr.read();

           System.out.println(sl);
        }
        catch( TSBSimpleListIOException e)
        {
            System.out.println("Error: " + e.getMessage());
        }
   }

   public static void main(String [] args)
   {
       Scanner sc = new Scanner(System.in);
       int op;
       do
```

```
        {
           System.out.println("1. Grabar");
           System.out.println("2. Recuperar");
           System.out.println("3. Probar");
           System.out.println("4. Salir");
           System.out.print("Ingrese opcion: ");
           op = sc.nextInt();
           switch(op)
           {
              case 1:   grabar();
                        break;

              case 2:   leer();
                        break;

              case 3:   System.out.println(sl);
           }
        }
        while(op != 4);
    }
}
```

No toda clase puede ser *Serializable*. Los objetos *streams* no son serializables (es decir, los objetos de clases del package *java.io*). Los atributos marcados como *static* en una clase no son serializables. Finalmente, si el programador desea que algún atributo de la clase no sea serializado, ese atributo debe ser marcado con la palabra *transient* (esto tiene utilidad si por ejemplo, la clase tiene elementos internos sobre los que no se quiere correr el riesgo de hacer que queden expuestos al des-serializar el objeto: un caso típico es un atributo que guarde un password) Por ejemplo, en la clase siguiente, el atributo "x" no será grabado al serializar un objeto de la clase *Ejemplo*:

```
public class Ejemplo implements Serializable
{
      private transient int x;
      private int y;
      ...
}
```

2.] Listas implementadas sobre arreglos de soporte (diseño preliminar – 1.0).

La clase *TSBSimpleList* que hemos mostrado en fichas anteriores sirvió como introducción al diseño e implementación de estructuras de datos propias del programador. Todos los lenguajes de programación ofrecen un kit de estructuras de datos ya implementadas (o *nativas*) que los programadores pueden usar, pero en numerosas situaciones prácticas ocurre que el programador necesita estructuras de datos que su lenguaje no provee (y en este caso se habla de *estructuras de datos abstractas*), o necesita implementar su propia versión (por las razones que sean) de alguna que el lenguaje sí brinda. Y en estos casos, es siempre buena idea seguir las pautas y convenciones que el lenguaje sugiera, de forma de garantizar que las nuevas estructuras que se implementen sean tan reutilizables como se pueda en cualquier contexto.

El lenguaje Java, el package *java.util*, provee una gran cantidad de estructuras de datos nativas, listas para usar e implementadas con criterios profesionales. Muchas de las estructuras de datos que analizaremos en este curso están ya implementadas por alguna de las clases de ese package, y podría parecer entonces que no tiene sentido un curso para diseñar e implementar estructuras que el lenguaje ya ofrece. Hay por lo menos un par de razones por las cuales el estudio, diseño e implementación de tales estructuras es recomedable:

a. Que un lenguaje implemente ciertas estructuras no significa que otros lenguajes implementen las mismas. Cuando el programador trabaje con lenguajes que no incluyan estructuras nativas que pueda necesitar, deberá saber cómo implementarlas.

b. Aún si un lenguaje incluye todas las estructuras que un programador pueda necesitar, siempre es bueno que ese programador conozca los detalles de funcionamiento de cada una, para poder decidir si debe usar una u otra en distintos contextos. Diseñar e emplementar una estructura de datos que el lenguaje ya provee sirve como experiencia y banco de pruebas para que el programador cuente con elementos que le ayuden a decidir. Y ciertos casos y contextos locales, las estructuras implementadas por el propio programador suelen ser más eficientes que las provistas por el lenguaje (ya que en definitiva, fueron diseñadas especialmente para aplicarse en esos contextos).

Nos proponemos aquí mostrar la forma en que puede implementarse una estructura de datos muy conocida en Java, modelada a través de la clase *ArrayList* del package *java.util*. Nuestra propuesta es hacer un diseño propio de esa estructura, a través de una clase que llamaremos *TSBArrayList*, e implementarla paso a paso, primero en una versión preliminar y luego avanzando en versiones cada vez más sofisticadas, hasta emular por completo a la clase original. Una vez concluido el trabajo, nada impide que se realicen pruebas comparativas de rendimiento, que siempre dejarán alguna enseñanza.

Para hacer más desafiante el trabajo, proponemos hacer un diseño y una implementación en modalidad *clean room*[1]: nos permitiremos consultar la documentación javadoc de la clase *java.util.ArrayList* para deducir de allí todo lo que sea posible respecto de su funcionalidad, pero no consultaremos los archivos fuente de esa clase (que por cierto, están accesibles para cualquier programador en el archivo *src.zip* contenido dentro de la carpeta Java donde se haya instalado el JDK). Y al final podremos comparar ambas versiones…

Comencemos: la clase *java.util.ArrayList* representa una lista soportada en un arreglo (exactamente como la clase *list* del lenguaje Python). La clase gestiona un arreglo a modo de buffer, en el que serán almacenados todos los elementos que la lista contenga, y por lo tanto no es necesaria una clase para representar nodos: la dirección de cada elemento en la lista corresponde al índice que ese elemento tiene en el arreglo de soporte. Si un elemento está en la casilla *i* del arreglo, entonces su sucesor está en la casilla *i+1* y su antecesor en la casilla *i-1*.

[1] Las palabras "clean room" significan literalmente "cuarto limpio" o "habitación limpia", y hacen referencia a una estrategia de desarrollo que consiste en emular un trabajo ya hecho pero sin consultar los programas fuente del original. La estrategia *clean room* suele usarse como forma de medir la capacidad de trabajo de un equipo de desarrolladores, o también como forma de testing para un proyecto bajo análisis.

El arreglo de soporte (que en nuestra clase *TSBArrayList* se llamará *items*) se crea con cierto tamaño inicial (llamaremos *initial_capacity* al atributo que representa ese tamaño de partida en nuestra clase), y el método *add(x)* de la clase agregará un elemento x en el arreglo, de forma que x se aloje detrás y a la derecha de los que ya estaban en el arreglo (en otras palabras, add(x) añade el elemento x al final de la lista). Un atributo adicional *count* se utiliza para recordar cuántos elementos ya se insertaron, y por lo tanto, *count* también nos dice el índice del primer casillero libre en el arreglo *items*.

Una primera idea de la clase *TSBArrayList* podría verse así, incluyendo los atributos ya citados y un par de constructores:

```
public class TSBArrayList<E>
{
    // el arreglo que contendrá los elementos...
    private Object[] items;

    // el tamaño inicial del arreglo...
    private int initial_capacity;

    // la cantidad de casillas realmente usadas...
    private int count;

    public TSBArrayList()
    {
        this(10);
    }

    public TSBArrayList(int initialCapacity)
    {
        if (initialCapacity == 0)
        {
            initialCapacity = 10;
        }
        items = new Object[initialCapacity];
        initial_capacity = initialCapacity;
        count = 0;
    }

    // resto de la clase aquí…
}
```

El segundo constructor es el que realmente hace el trabajo: crea el arreglo items con el tamaño que se indique en el parámetro *initialCapacity* (con el que tambien asigna el valor inicial del atributo *initial_capacity*) y ajusta el atributo *count* en 0. Como dijimos, este atributo indica cuántos elementos realmente contiene el arreglo *items*, y su valor también sirve para recordar cuál es el primer casillero libre en ese arreglo: si *count* vale 0, el arreglo no tiene elementos y el primer casillero libre es el cero. El primer constructor simplemente invoca al segundo para crear un arreglo de diez casillas de capacidad.

Note que la clase *TSBArrayList* está parametrizada usando el mecanismo *generics*: el parámetro *<E>* indica que podrán crearse objetos de la clase *TSBArrayList* subordinados a una clase concreta *E* cualquiera que se definirá al crear el objeto. Sin embargo, observe que al crear con *new* el arreglo *items* en el constructor, se indica que el arreglo contendrá referencias polimórficas de tipo *Object* (y no de tipo *E*):

```
private Object[] items;
        . . .
items = new Object[initialCapacity];
```

Esto es así porque el compilador Java no permitirá crear un arreglo cuyo tipo base sea un tipo parametrizado (como el tipo E en nuestro caso). Una instrucción de la forma:

```
items = new E[initialCapacity];
```

no compilará, ya que el identificador *E* representa un tipo que no es identificable en tiempo de ejecución, y el operador *new* no podrá alojar el arreglo sin conocer concretamente ese tipo. Esencialmente, para el compilador no habría mayor problema en dejar pasar esa línea, pero el inconveniente sería trasladado al tiempo de ejecución: el operador *new* fallaría al no poder determinar con precisión qué clase específica es *E*. Como se mostró, una solución consiste en definir el arreglo para contener referencias a *Object*, y crearlo con referencias a *Object*. Como la clase *Object* es la base de todas las jerarquías en Java, entonces una referencia a *Object* puede apuntar polimórficamente a cualquier objeto de cualquier clase *E* que aparezca luego, sea cual sea la clase *E*. Como esa clase será controlada por el compilador antes de ejecutar el programa, el arreglo contendrá objetos de una única clase *E*, aún cuando el tipo base del arreglo no haya sido definido como *E*.

La clase *TSBArrayList* contiene varios otros métodos, cuyas cabeceras fueron tomadas al pie de la letra de la lista de métodos de la clase *java.util.ArrayList* (aunque en esta versión preliminar no se incluyeron todos, sino sólo los más representativos). Por lo pronto, cuenta con dos métodos técnicos que permiten ajustar la capacidad o tamaño del arreglo items si fuese necesario:

```
public void ensureCapacity(int minCapacity)
{
    if(minCapacity == items.length) return;
    if(minCapacity < count) return;

    Object[] temp = new Object[minCapacity];
    System.arraycopy(items, 0, temp, 0, count);
    items = temp;
}

public void trimToSize()
{
    if(count == items.length) return;

    Object temp[] = new Object[count];
    System.arraycopy(items, 0, temp, 0, count);
    items = temp;
}
```

El método *ensureCapacity(minCapacity)* ajusta (si fuese necesario) el tamaño del arreglo *items* para que se garantice que puede contener al menos la cantidad de elementos indicada por el parámetro *minCapacity*. La idea es simple: se crea un segundo arreglo llamado *temp* con el tamaño indicado por *minCapacity*, y se copian en *temp* todos los elementos que estaban en el arreglo original *items*. Al finalizar, se hace que la referencia *items* pase a apuntar al mismo arreglo que apunta *temp* (lo cual hace que el arreglo originalmente apuntado por *items* quede des-referenciado y sea eventualmente liberado por el *garbage collector*).

La copia de todos los elementos de items hacia el arreglo *temp* se ha hecho con el método estático *System.arraycopy()*, que toma cinco parámetros: El primero es el arreglo fuente desde el cual se toman los datos a copiar. El segundo es el índice del casillero del arreglo fuente a partir del cual se debe comenzar la copia. El tercero es el arreglo destino hacia el cual se deben copiar los datos. El cuarto es el índice de la casilla en el arreglo destino a partir de la cual serán copiados los datos. Y el quinto es la cantidad de elementos que serán copiados. Todos esos parámetros son algo difíciles de recordar, pero le dan al método gran versatilidad. Para entender la idea, considere que la instrucción:

```
System.arraycopy(items, 0, temp, 0, count);
```

es equivalente al siguiente bloque de código:

```
for(int i=0; i<count; i++)
{
      temp[i] = items[i];
}
```

Es importante en este momento notar que si bien las dos formas que acaban de mostrarse para copiar un arreglo en otro producen el mismo resultado, en términos de tiempo de ejecución hay una diferencia notable: la copia realizada con el ciclo *for*, tiene un tiempo de ejecución *proporcional a count*, que en el peor caso es igual al tamaño *n* del arreglo. Por lo tanto, ese ciclo ejecuta en un tiempo *O(n)* (lineal) en el peor caso. Pero el método *System.arraycopy()* internamente está implementado en *lenguaje C* (se dice que es un método *nativo*) y la copia se realiza mediante una función específica de ese lenguaje (*memcpy()*) que copia un bloque completo de memoria contigua hacia otro lugar contiguo de memoria, *pero en tiempo constante*… Por lo tanto, la copia con *System.arraycopy()* ejecuta en tiempo *O(1)* (constante) sea cual sea el tamaño del arreglo a copiar[2]. Conclusión: no hay razón práctica para hacer la copia empleando el ciclo, si puede usarse el método *System.arraycopy()* para hacerlo más rápido.

El método *trimToSize()* ajusta (si es necesario) el tamaño del arreglo *items* para que sea igual a la cantidad de elementos *count* que el arreglo efectivamente contiene. Este método puede ser útil en situaciones en las que el programador desee forzar algún ahorro de memoria, eliminando los casilleros sin usar que pudiera tener el arreglo *items* en un momento determinado. La idea es igualmente directa: se crea un segundo arreglo *temp* cuyo tamaño

[2] Es cierto que existen situaciones en las que *System.arraycopy()* podría también ejecutar en tiempo lineal, pero el análisis general de tiempo de ejecución es esencialmente y en la mayor parte de los casos el que hemos indicado aquí.

sea exactamente igual a *count*, se copian en él todos los elementos del arreglo original *items* (usando otra vez *System.arraycopy()*) y finalmente se asigna en la referencia *items* la dirección del arreglo *temp*, des-referenciando al arreglo original.

La clase provee dos versiones del método *add()* (otra vez, tomadas tal cual están especificadas en la documentación javadoc de *java.util.ArrayList*)

```
public boolean add(E e)
{
    if(e == null) return false;

    if(count == items.length) this.ensureCapacity(items.length * 2);

    items[count] = e;
    count++;
    return true;
}


public void add(int index, E e)
{
    if(index > count || index < 0)
    {
        throw new IndexOutOfBoundsException("Fuera de rango...");
    }

    if(e == null) return;

    if(count == items.length)this.ensureCapacity(items.length * 2);

    int t = count - index;
    System.arraycopy(items, index, items, index+1, t);
    items[index] = e;
    count++;
}
```

El primero de estos dos métodos (*add(E e)*) agrega un elemento *e* de la clase *E* al final de la lista (esto es: en la casilla cuyo índice es igual a *count* en el arreglo *items*). Retorna *true* si la operación tuvo éxito o *false* en caso contrario (por ejemplo, si se intentó agregar una referencia nula en la lista). Note que el método controla que el arreglo *items* tenga lugar libre para agregar un nuevo elemento, y en caso de no tenerlo (o sea, si la cantidad de elementos ya contenidos *count* es igual a la capacidad *items.length* del arreglo) se invoca al método *ensureCapacity()* para duplicar el tamaño del arreglo (tal como ya se explicó).

El segundo método (*add(index, e)*) también agrega un elemento *e* de la clase *E* al arreglo, pero lo hace de forma que el nuevo elemento pase a ocupar la casilla cuyo índice es *index*, haciendo que todos los elementos que originalmente ocupaban las casillas desde *index* en adelante, se desplacen un casillero hacia la derecha. De nuevo, si es necesario el método invoca a *ensureCapacity()* para duplicar el tamaño del arreglo *items*, y de nuevo, se usa el método *System.arraycopy()* para copiar un casillero hacia la derecha *en el mismo arreglo* a todos los elementos que deben desplazarse.

Note que en las condiciones dadas, ambos métodos ejecutan en tiempo *O(1)* (constante): el primero agrega un elemento al final con un simple acceso directo (y si debe duplicar el tamaño lo hace en tiempo constante como ya se indicó con el método *ensureCapacity()*). Y el segundo debe desplazar hacia la derecha un subconjunto de elementos que en el peor caso será casi tan grande como todo el arreglo (si el nuevo elemento se inserta adelante), pero eso también se hace con *System.arraycopy()* en tiempo constante. Y si este segundo método *add()* debe aumentar el tamaño del arreglo, también lo hará en tiempo constante invocando a su vez a *ensureCapacity()*.

En forma inversa a los métodos *add()*, el método *remove(index)* elimina de la lista el elemento ubicado en la casilla *index*, y lo retorna:

```
public E remove(int index)
{
    if(index >= count || index < 0)
    {
        throw new IndexOutOfBoundsException("fuera de rango...");
    }

    int t = items.length;
    if(count < t/2) this.ensureCapacity(t/2);

    Object old = items[index];
    int n = count;
    System.arraycopy(items, index+1, items, index, n-index-1);
    count--;
    items[count] = null;
    return (E) old;
}
```

El método controla el espacio disponible en el arreglo *items* antes de proceder al borrado: Si la cantidad de elementos disponibles (indicada por *count*) ya es menor que la mitad de la capacidad del arreglo, entonces se invoca a *ensureCapacity()* para reducir el tamaño a la mitad, permitiendo así que arreglo siempre se mantenga con una capacidad acorde a la verdadera cantidad de elementos que contiene. Luego se usa el método *System.arraycopy()* para mover un casillero hacia la izquierda (en el mismo arreglo) a todos los elementos que estaban originalmente ubicados desde la casilla *index + 1* en adelante. De este modo, el elemento que originalmente estaba en *index* se elimina por des-referencia. Se resta uno al valor de *count* (pues la lista tiene ahora un elemento menos), y como en este momento la casilla *items[count]* contiene una copia del último elemento (que se ha copiado a la casilla *items[count - 1]* cuando todos se movieron a la izquierda), entonces se ajusta en *null* el casillero *items[count]* para des-referenciar esa copia. Y otra vez, el proceso completo ejecuta en tiempo *O(1)* (constante).

Los métodos que siguen son simples y directos: *clear()* elimina el contenido completo de la lista, dejándola vacía, y reajustando el tamaño del arreglo items para que vuelva a tener la capacidad con la que fue originalmente creado:

```
public void clear()
{
    items = new Object[initial_capacity];
```

```
        count = 0;
    }
```

El método *contains()* devuelve *true* si la lista contiene al elemento *e* tomado como parámetro. Si *e* es *null* el método retorna *false*. Puede lanzar una excepción de *ClassCastException* si la clase del objeto *e* no es compatible con el contenido de la lista (en ese sentido, note que la referencia *e* está declarada de tipo *Object* y no de tipo *E*: esto es así debido que en la propia clase *java.util.ArrayList* este método está definido de esta forma). Por supuesto, como este método efectúa un recorrido secuencial en el arreglo, su tiempo de ejecución es *O(n)* (lineal) en el peor caso:

```
public boolean contains(Object e)
{
    if(e == null) return false;

    for(int i=0; i<count; i++)
    {
        if(e.equals(items[i])) return true;
    }
    return false;
}
```

El método *get(index)* retorna el elemento ubicado en la casilla index. Si ese index está fuera de rango, el método lanza una excepción. Note que antes de hacer el return final, el contenido de la casilla *items[index]* (originalmente de tipo *Object*) es convertido al tipo *E* usando el operador de *casting explícito*:

```
public E get(int index)
{
    if (index < 0 || index >= count)
    {
        throw new IndexOutOfBoundsException("fuera de rango...");
    }
    return (E) items[index];
}
```

El método *isEmpty()* retorna *true* si la lista está vacía o *false* en caso contrario:

```
public boolean isEmpty()
{
    return (count == 0);
}
```

El método *set(index, element)* cambia el elemento contenido originalmente en la casilla *index*, por el objeto *element* tomado como parámetro. Si *index* está fuera de rango el método lanza una excepción. Note que *set()* no altera el tamaño del arreglo ni la cantidad de elementos de la lista: sólo reemplaza un objeto pre-existente por otro:

```
public E set(int index, E element)
{
    if (index < 0 || index >= count)
    {
        throw new IndexOutOfBoundsException("fuera de rango...");
    }
```

```
        Object old = items[index];
        items[index] = element;
        return (E) old;
    }
```

El método *size()* retorna la cantidad de elementos que la lista contiene (no confundir con el tamaño o capacidad del arreglo de soporte):

```
    public int size()
    {
        return count;
    }
```

El método *toString()* retorna la ya consabida cadena representando el contenido completo de la lista en forma conveniente:

```
    public String toString()
    {
        StringBuilder buff = new StringBuilder();
        buff.append('{');
        for (int i=0; i<count; i++)
        {
            buff.append(items[i]);
            if(i < count-1)
            {
                buff.append(", ");
            }
        }
        buff.append('}');
        return buff.toString();
    }
```

Lo último (en esta primera versión…) es el mecanismo iterador. La clase *TSBArrayList* implementa la interface *Iterable* para indicar que efectivamente cuenta con un método *iterator()* que retorna un iterador (de paso, digamos que la clase también implementa *Serializable*, con lo que el mecanismo de serialización está disponible para sus instancias):

```
public class TSBArrayList<E> implements Iterable<E>, Serializable
```

A partir de esto, la clase *TSBArrayList* ***debe*** implementar el método *iterator()*, que es el siguiente en nuestro caso:

```
    public Iterator<E> iterator()
    {
        return new TSBArrayListIterator();
    }
```

El método *iterator()* simplemente crea y retorna un objeto de la clase *TSBArrayListIterator*, que es una clase interna de la clase *TSBArrayList*, definida tal como sigue:

```
private class TSBArrayListIterator implements Iterator<E>
{
    private int current;   // índice del elemento que hay que procesar.
    private boolean  next_ok; // true: next fue invocado (para remove()...)
```

```
    public TSBArrayListIterator()
    {
        current = -1;
        next_ok = false;
    }

    /**
     * Indica si queda algun objeto en el recorrido del iterador.
     * @return true si queda algun objeto en el recorrido - false si no
     * quedan objetos.
     */
    @Override
    public boolean hasNext()
    {
         if(isEmpty()) { return false; }
         if(current >= size() - 1) { return false; }
         return true;
    }

    /**
     * Retorna el siguiente objeto en el recorrido del iterador.          *
     * @return el siguiente objeto en el recorrido.
     * @throws NoSuchElementException si la lista está vacia o en la lista
     *         no quedan elementos por recorrer.
     */
    @Override
    public E next()
    {
       if(!hasNext()){ throw new NoSuchElementException("no quedan..."); }

       current++;
       next_ok = true;
       return (E) items[current];
    }

    /**
       Elimina el ultimo elemento que retornó el iterador. Debe invocarse
       una sola vez antes de volver invocar a next(). El iterador queda
       posicionado en el elemento anterior al eliminado.
       @throws IllegalStateException si se invoca a remove() sin haber
               invocado a next(), o si remove() fue invocado mas de una vez
               luego de una sola invocacion a next().
     */
    @Override
    public void remove()
    {
       if(!next_ok){ throw new IllegalStateException("invocar a next()"); }

       E garbage = TSBArrayList.this.remove(current);
       next_ok = false;
       current--;
    }
}
```

La clase iteradora *TSBArrayListIterator* implementa la interface *Iterator*, de la cual toma los métodos *hasNext()*, *next()* y *remove()*. Además, define un atributo *current* para recordar cuál es el índice del último casillero cuyo contenido fue alcanzado por *next()*, y un segundo atributo *next_ok* de tipo *boolean*, que se usa como bandera para avisar que el método *next()*

fue invocado convenientemente antes de invocar al método *remove()*: en las especificaciones javadoc de la interface *Iterator*, está indicado que el método *remove()* de una clase iteradora debe eliminar el último elemento que haya sido alcanzado por *next()*, por lo cual *next()* debe haber sido invocado antes de invocar a *remove()*. De hecho, también se debe controlar que una vez que *remove()* fue invocado, no vuelva a invocarse nuevamente antes de invocar a *next()*.

El constructor de la clase simplemente inicializa los atributos *current* en -1 y *next_ok* en *false*. Con esto, el iterador no está posicionado en ningún casillero en particular, y se avisa que *next()* aún no ha sido invocado.

El método *hasNext()* retorna true si todavía queda algún elemento de la lista que no haya sido alcanzado y retornado por *next()* (lo cual ocurrirá si la lista no está vacía y el valor de *current* es menor que la cantidad de elementos de la lista menos uno).

El método *next()* invoca a *hasNext()* para saber si queda al menos un elemento para ser alcanzado, y en ese caso suma uno a *current*, toma el elemento en esa posición, y lo retorna haciendo casting al tipo *E*. Como el método *next()* se está ejecutando y eso implica que ha sido invocado, entonces antes de finalizar cambia el flag *next_ok* a *true* para dejar marcado que efectivamente fue invocado.

Finalmente, el método *remove()* debe eliminar el último elemento que haya sido alcanzado por *next()* (es decir, el elemento que actualmente esté en la casilla *current*), pero de forma que *current* quede apuntando al elemento que estaba a la izquierda de este (es decir, *current* debe quedar valiendo el índice de la casilla anterior). Por lo tanto, primero debe chequear que next() haya sido invocado (lo cual hace con la condición de la primera línea), lanzando una excepción en caso contrario e interrumpiendo la ejecución del método.

Si todo estaba bien con *next()*, entonces para llevar a cabo efectivamente el borrado simplemente se invoca al método *remove(index)* que ya está implementado en la propia clase *TSBArrayList* (y que ya hemos mostrado): al invocarlo, se le pasa como parámetro el valor *current*, y el método hace el resto, incluyendo la disminución del tamaño del arreglo si fuese necesario y la operación de restar uno al valor de *count* para reflejar el hecho de que la lista ha perdido un componente. Note que para invocar al método remove(index) de la clase TSBArrayList, el método remove() de la clase TSBArrayListIterator accede al mismo de la forma siguiente:

```
TSBArrayList.this.remove(current)
```

Esto es: si desde el interior de un método que pertenece a una clase interna (como el método *remove()* de la clase *TSBArrayListIterator*) se quiere accder a un método o atributo de su clase contenedora (como el método *remove(index)* de la clase *TSBArrayList*) pero usando la referencia *this*, entonces el método de la clase interna *debe anteponer el nombre de la clase contenedora* antes de acceder a *this* (pues de otro modo, estará accediendo al puntero *this* de la propia clase interna…).

El método *remove()* finaliza volviendo a *false* el valor del atributo *next_ok*: la idea es que si *remove()* fue invocado es porque next() había sido invocado a su vez, pero una vez eliminado el elemento actual, *next()* debe volver a invocarse para poder activar nuevamente a *remove()*. Si *remove()* pone *false* en *next_ok*, entonces dos invocaciones seguidas a *remove()* (sin invocar a *next()* entre ambas) provocarán una excepción.

Con esto finaliza la exposición de los detalles de implementación de nuestra versión preliminar de la clase *TSBArrayList*. El proyecto *TSB-ArrayList01* que acompaña a esta ficha incluye una clase *Test* con un método *main()* para hacer una prueba general de todos los métodos aquí explicados. Dejamos su análisis para el estudiante:

```
import java.util.Iterator;

public class Test
{
    public static void main(String args[])
    {
        // un arreglo auxiliar de nombres...
        String nombres[] = {"Ana", "Luis", "Carlos", "Luz", "Juan", "Maria"};

        // creación de un ArrayList...
        TSBArrayList<String> al = new TSBArrayList<>(5);
        System.out.println("Lista inicial: " + al);

        // inserción de cadenas...
        for(int i=0; i<nombres.length; i++)
        {
            al.add(nombres[i]);
        }
        System.out.println("Luego de invocar a add(): " + al);

        // eliminación de una cadena...
        al.remove(3);
        System.out.println("Luego de invocar a remove() una vez: " + al);

        // eliminación masiva usando remove():
        while(al.size()>0)
        {
            al.remove(0);
        }
        System.out.println("Luego de invocar a remove() muchas veces: " + al);

        // regeneración de la lista inicial...
        for(int i=0; i<nombres.length; i++)
        {
            al.add(nombres[i]);
        }
        System.out.println("Luego de invocar a add(): " + al);

        // acceso a un componente individual...
        String nom = (String) al.get(4);
        System.out.println("Nombre recuperado con get(): " + nom);

        // cambio de un componente individual...
        al.set(5, "David");
        System.out.println("Contenido luego de usar set(): " + al);

        al.add(3, "Diana");
        System.out.println("Luego de insertar en casilla 3: " + al);

        int id = al.size();
```

```
        al.add(id, "Diego");
        System.out.println("Luego de insertar en casilla " +  id + ": " + al);

        System.out.println("Contenido usando iterador: ");
        Iterator<String> it = al.iterator();
        while(it.hasNext())
        {
            String x = it.next();
            System.out.println(x);
        }

        System.out.println("Contenido con iterador, eliminando 'Carlos': ");
        Iterator<String> it2 = al.iterator();
        while(it2.hasNext())
        {
            String x = it2.next();
            if(x.equals("Carlos")) it2.remove();
            else System.out.println(x);
        }
        System.out.println("Contenido luego de eliminar a 'Carlos': " + al);


        al.clear();
        // regeneración de la lista inicial...
        for(int i=0; i<nombres.length; i++)
        {
            al.add(nombres[i]);
        }
        System.out.println("Luego de invocar a clear() y regenerarla: " + al);

        String nombre = "Luis";
        boolean r = al.contains(nombre);
        System.out.println("El nombre " + nombre + " está en la lista??? " + r);

        al.clear();
        al.trimToSize();
        System.out.println("Lista luego de clear() y trimToSize(): " + al);
    }
}
```