Ficha 8

Listas sobre arreglos [2.0] - JUnit

1.] Listas implementadas sobre arreglos de soporte (diseño final – 2.0).

Hemos mostrado en una ficha anterior una implementación preliminar de la clase *TSBArrayList* para representar listas soportadas sobre arreglos adaptativos. Esa primera versión se basó en emular tan fielmente como fuese posible la implementación original de la clase *java.util.ArrayList* (nativa de Java), mediante una estrategia que designamos como *clean room*: nos permitimos analizar y estudiar la documentación oficial de la clase, pero no su código fuente original. Y nuestro primer diseño consistió en simplemente copiar las especificaciones de los métodos principales de *java.util.ArrayList*, manteniendo sus cabeceras tal como figuran en ella y luego implementándolos de acuerdo a la funcionalidad descripta para cada uno en la documentación de esa clase.

En esta ficha analizaremos una forma de diseño e implementación definitiva, acorde a las convenciones de Java. El modelo terminado está disponible en el proyecto *TSB-ArrayList02* que acompaña a esta ficha.

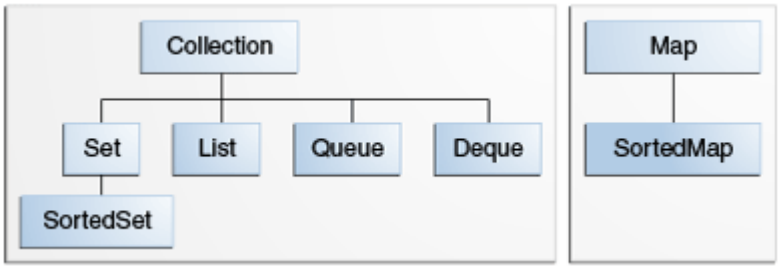
La primera cuestión importante a introducir, es que al definir una clase para representar e implementar a una estructura de datos abstracta (no provista por el lenguaje) o para emular lo hecho por una estructura nativa (que es nuestro caso), la idea central es que esta nueva clase *derive* de alguna de las clases que Java ya tiene previstas para este tipo de desarrollos, y que además *implemente* la o las clases de interfaces que Java también provee para ayudar en el diseño.

La ventaja que se obtiene si se trabaja de esta forma, es que la super clase desde la cual se derive y las interfaces que se implementen imponen un contexto en el que las definiciones de los métodos públicos a implementar están ya previstos, y en ese mismo contexto la nueva clase aprovecha el marco polimórfico de la jerarquía a la que pasa a pertenecer. En lugar de copiar manualmente las cabeceras de los métodos a incluir, es la *herencia* la que impone esas cabeceras. Además, las instancias de la nueva clase pueden usarse *polimórficamente* en cualquier contexto en el que se esperase encontrar instancias de la super clase. Una clase diseñada e implementada por el programador siguiendo estos principios, puede ser usada exactamente en cualquier lugar en donde podría usarse una clase nativa que pertenezca a la misma jerarquía y sea polimórficamente intercambiable con ella. En nuestro caso, si seguimos esta línea de acción, un objeto de nuestra clase *TSBArrayList* podrá usarse en cualquier programa en lugar de un objeto de la clase *java.util.ArrayList*, sin cambiar ninguna linea de código más que aquella en la cual se cree el objeto.

El lenguaje Java dispone de un complejo esquema de jerarquías de clases e interfaces nativas ya previstas para llevar adelante esta estrategia, en el marco de trabajo definido por el package *java.util* que ya hemos citado.

En general, para referirse a cada estructura de datos provista por ese paquete, se usa el nombre *Colección* (o *Collection*). Los objetos *Collection* representan estructuras de datos capaces de contener objetos (y no tipos primitivos). En ese sentido, un objeto *Collection* difiere de los arreglos comunes en cuanto a que estos últimos pueden contener objetos, pero también tipos primitivos. Sin embargo, un arreglo común tiene tamaño fijo, en tanto que un objeto *Collection* puede crecer a medida que lo requiere el programa. Por otra parte, si se desea almacenar valores de tipos primitivos en un objeto *Collection*, se pueden usar las clases de envoltorio (wrapper classes: Integer, Character, Double, etc.) sin problemas (y a partir de las últimas versiones de Java está también disponible el *autoboxing*, que facilita el trabajo del programador: si se desea almacenar valores de tipos primitivos en una colección, simplemente se invoca a los métodos de esa colección para hacerlo y el mecanismo de autoboxing de encarga de convertir esos valores primtivos a objetos *wrapper*).

El paquete *java.util* provee una jerarquía amplia y completa de interfaces y clases que representan a cada estructura de datos básica. Existen dos clases de interface que dan origen a sendas jerarquías para representar estructuras de datos: *Collection* y *Map*. La novedad es que estas jerarquías comienzan con una *jerarquía de interfaces*: desde *Collection* y *Map* se definen nuevas interfaces que *derivan* de *Collection* o *Map*, y a partir de ellas vienen las clases para cada estructura. Como vemos, se comienza con *clases de interfaces que heredan entre ellas*. Un esquema de las dos jerarquías de interfaces es el siguiente[1]:



Mostramos aquí un breve resumen del fundamento de cada una de ellas:

- ***Collection***: es la raiz de una de las jerarquías de colecciones. Una *colección* representa un grupo de objetos que se conocen como sus elementos. La interface *Collection* es el denominador común que muchas de las colecciones implementan y se usa cuando se desea el máximo grado de generalidad en aplicaciones polimórficas. Algunos tipos de colecciones permiten elementos repetidos y otros no. Algunas colecciones son ordenadas y otras no. La plataforma Java no provee clases concrectas que implementen directamente a *Collection*, pero sí provee varias que implementan a sus subinterfaces específicas *Set*, *List*, *Queue* y *Deque*.

[1] La gráfica está tomada directamente desde los tutoriales de Java disponibles en la documentación *javadoc* de la empresa Oracle: http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html.

- ***Set***: una colección que no contiene elementos repetidos. Esta interface modela la abstracción matemática del *conjunto* y se usa para representar conjuntos de diversos tipos de objetos.
- ***List***: una colección o secuencia ordenada (*en el sentido de preservar del orden de inserción*). Puede contener elementos repetidos. Normalmente el programador que usa un objeto *List* tiene control preciso de en qué lugar de la lista se insertan los elementos y puede acceder a ellos mediante sus índices o posiciones.
- ***Queue***: una colección usada para soportar y almacenar múltiples elementos antes de su procesamiento. Una *Queue* típicamente (pero no necesariamente) mantiene sus elementos en orden *FIFO* (*first-in, first-out*). No obstante, puede mantenerse el orden de acuerdo a algún criterio y obtener una *cola de prioridad*. Sea cual sea el orden de almacenamiento usado, el primer elemento de una *Queue* es el elemento que debería removido al invocar al método *remove()* o al método *poll()*. En una *Queue* de tipo *FIFO*, todo nuevo elemento se inserta al final de la estructura. Otros tipos de *Queues* deben especificar sus propias reglas.
- ***Deque***: una colección usada para soportar y almacenar múltiples elementos antes de su procesamiento. Una *Deque* puede ser usada tanto en forma *FIFO* (*first-in, first-out*) como en forma *LIFO* (*last-in, first-out*). En una *deque* todo nuevo elemento puede ser insertado, recuperado y eliminado desde cualquiera de sus dos extremos.
- ***Map***: un objeto que puede mapear ciertas claves hacia ciertos valores. Un *Map* no puede contener claves duplicadas; y cada clave puede mapear al lo sumo a un valor. Las *tablas hash* son posiblemente las formas más conocidas de implementación de un *Map*.
- ***SortedSet***: un *Set* que mantiene sus elementos en orden ascendente. Es simplemente la versión ordenada de la interface *Set*. Provee algunas operaciones adicionales para aprovechar la ventaja de tener sus elementos ordenados.
- ***SortedMap***: un *Map* que mantiene los elementos mapeados en él en orden ascendente de sus claves. Es simplememente la versión ordenada de la interfce *Map*.

Estas jerarquías de interfaces brindan el contexto general y polimórfico básico para la implementación de las estructuras de datos nativas ya provistas por Java, y para el diseño de nuevas estructuras abstractas que pudiera necesitar el programador.

Para aprovechar al máximo este contexto de trabajo, es bueno hacer un paréntesis y poner en claro algunas ideas referidas a las estructuras de datos que por el momento nos ocupan: las *listas*. Estrictamente hablando, una *lista* es una estructura de datos *lineal* (cada elemento tiene o puede tener un único elemento sucesor y un único elemento antecesor) que además es *adaptativa*: a medida que necesita (o deja de necesitar) memoria, puede crecer o decrecer en su tamaño, y de tal forma además que un nuevo elemento puede ser insertado en cualquier lugar de la lista (y no necesariamente al principio o al final).

Esta capacidad adaptativa es la que distingue a las *listas* de los *arreglos*. En muchos de los primeros lenguajes (o incluso en versiones antiguas de lenguajes modernos) los arreglos se definían como colecciones lineales esencialmente no adaptativas (no podían crecer ni decrecer en tamaño) pero que tenían la ventaja del acceso directo a sus componentes en

tiempo constante. Esto se debe a que un arreglo se aloja completo en un bloque contiguo de memoria, y conocida la dirección del primer elemento es simple y directo calcular la dirección de cualquier otro.

Con el tiempo, y a partir de la flexibilidad para el manejo de memoria de lenguajes como *C* o *C++* (en los que se programan muchos de los lenguajes modernos…) todos los lenguajes comenzaron a brindar capacidad adaptativa a sus arreglos (ya sea en forma directa o facilitando funciones de librería o clases especiales para hacerlo), llegando a un estado en el que la línea que separa a los arreglos de las listas se ha vuelto difusa: baste recordar (por ejemplo) que en el lenguaje *Python* directamente no se habla ya de arreglos sino de listas, y que esas listas en *Python* son en todo equivalentes a las *java.util.ArrayList* de *Java*, o a nuestras *TSBArrayList*: estructuras lineales adaptativas, que se alojan en un bloque contiguo de memoria y disponen por tanto de acceso directo en tiempo constante.

De acuerdo a la forma en que una lista se implemente, se podrá contar con ciertas ventajas y desventajas. Esencialmente, hay dos formas básicas de implementar una lista:

a. *Implementación encadenada (o ligada)*: Cada elemento de la lista se almacena en un objeto separado designado como *nodo*. Cada nodo contiene un campo o slot para almacenar al elemento que contiene, y también incluye uno o dos campos más para almacenar las direcciones de su nodo sucesor y de su nodo antecesor. Si sólo se almacena la dirección de uno de ellos, la lista se dice *simplemente vinculada*, y si se almacenan los dos, se dice *doblemente vinculada*. La dirección del primero de los nodos se almacena en una variable separada (y también la del último si la lista es doblemente vinculada). Cuando la lista necesita crecer, se crea un nuevo nodo y se engancha al mismo en el lugar requerido, cambiando los valores de los enlaces al sucesor y/o antecesor. Si la lista necesita decrecer, se sueltan los enlaces que apuntan al nodo que se desea eliminar y se los ajusta para que apunten al nuevo sucesor y/o antecesor. Puede verse que de esta forma, la memoria ocupada por la lista no constituye un bloque contiguo (cada nodo se crea por separado y se aloja en el lugar que esté libre en ese momento) y esto lleva a que el acceso a cada nodo sólo pueda hacerse mediante un recorrido secuencial comenzando desde el principio (o desde el final) y siguiendo los enlaces al sucesor o antecesor. En otras palabras: el acceso a un elemento individual de una lista ligada requiere tiempo lineal (*O(n)*) en el peor caso. Un ejemplo de este tipo de implementación es nuestra clase *TSBSimpleList* (simplemente vinculada) o la clase *java.util.LinkedList* de *Java* (doblemente vinculada).

b. *Implementación sobre un arreglo de soporte*: Toda la lista se almacena sobre un arreglo de forma de ir ocupando ese arreglo de izquierda a derecha, sin saltear casillas. El arreglo de soporte puede contener más casillas que las que realmente está usando la lista en un momento dado, pero como la memoria ocupada es contigua se puede acceder en forma directa y en tiempo constante (*O(1)*) a cada elemento a través del índice que el mismo tiene en el arreglo. Si la lista tiene que crecer y el arreglo de soporte no tiene lugar libre, se crea un nuevo arreglo más grande, se copian los elementos del viejo arreglo al nuevo y se desecha el viejo, y se procede a la inversa cuando el arreglo debe acortarse. Obviamente, los dos ejemplos

más directos que podemos citar son la clase *java.util.ArrayList* de *Java*, y nuestra ya conocida *TSBArrayList*.

Está claro que la ventaja de las *listas ligadas* es que ocupan exactamente la memoria que necesitan y son muy útiles en situaciones de memoria dispersa o fragmentada, pero su desventaja es que el acceso a sus elementos individuales es más lento (por ser secuencial). Por su parte, las *listas implementadas sobre un arreglo* brindan acceso rápido (de tiempo constante) a sus elementos, pero en general ocupan más memoria que la que podrían necesitar, debido a que el arreglo de soporte ocupa espacio adicional durante la mayor parte del tiempo.

Ahora bien: si un programador quiere desarrollar sus propias clases para implementar listas, la idea es que esas clases implementen la interface *List* de *java.util*, y esta interface le dirá qué métodos son los que están previstos para una lista (tanto en su forma ligada como en su forma de arreglo de soporte). Pero el hecho es que la interface *java.util.List* prevé una cantidad considerable de métodos (y por lo tanto, una considerable cantidad de trabajo…) La siguiente tabla está extraida (y traducida) directamente de la documentación *javadoc* de la interface *java.util.List*:

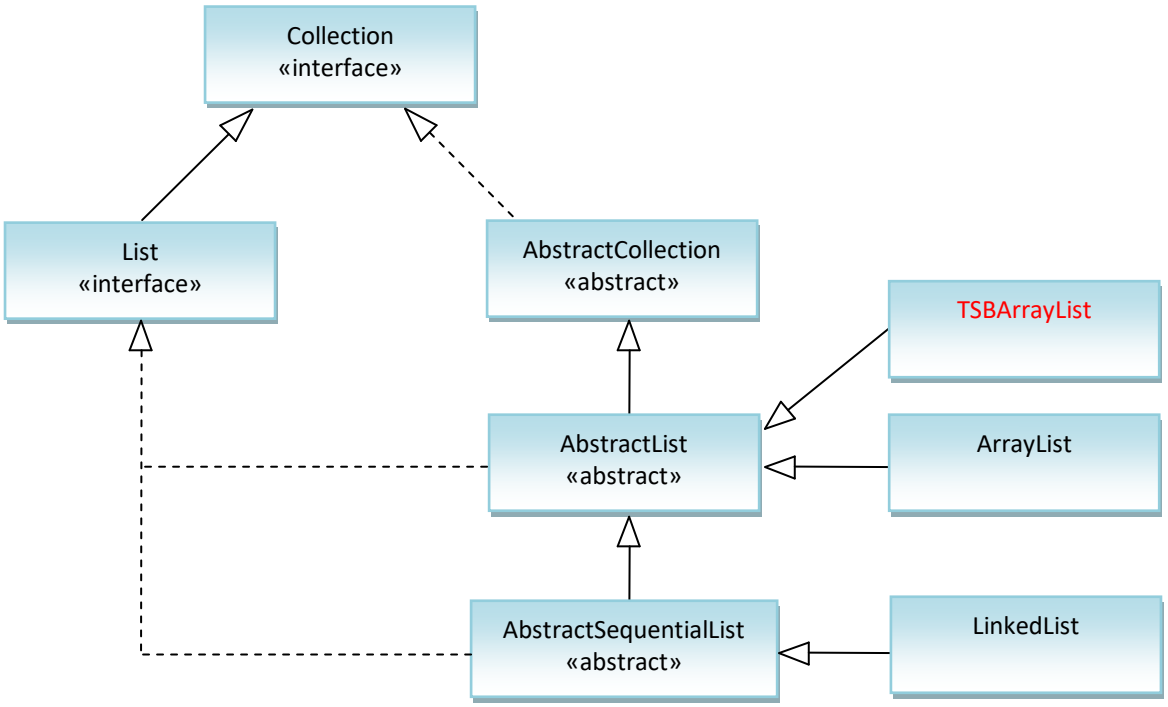
Interface List<E> extends Collection<E>

Tipo de salida	Método – Descripción básica
boolean	add(E e) Agrega el elemento especificado al final de esta lista (operación opcional).
void	add(int index, E element) Inserta el elemento especificado en la posición especificada en esta lista. (operación opcional).
boolean	addAll(Collection<? extends E> c) Agrega todos los elementos de la colección especificada, al final de esta lista, el orden en que son retornados por el iterador de la colección (operación opcional).
boolean	addAll(int index, Collection<? extends E> c) Inserta en esta lista todos los elementos de la colección especificada, a partir de la posición especificada (operación opcional).
void	clear() Elimina todo el contenido de esta lista (operación opcional).
boolean	contains(Object o) Retorna true si esta lista contiene al elemento especificado.
boolean	containsAll(Collection<?> c) Retorna true si esta lista contiene a todos los elementoss de la colección especificada.
boolean	equals(Object o) Compara el objeto especificado, con esta lista, para determinar si son iguales.
E	get(int index) Retorna el elemento en la posición especificada de esta lista.
int	hashCode()

	Retorna un valor de hash code para esta lista.
int	indexOf(Object o) Retorna el índice de la primera ocurrencia del elemento especificado en esta lista, o el valor -1 si la lista no contiene a ese elemento.
boolean	isEmpty() Retorna true si la lista no contiene elementos.
Iterator<E>	iterator() Retorna un iterador sobre los elementos de esta lista, en una secuencia apropiada.
int	lastIndexOf(Object o) Retorna el índice de la última ocurrencia del elemento especificado en esta lista, o el valor -1 si la lista no contiene a ese elemento.
ListIterator<E>	listIterator() Retorna un iterador de lista sobre los elementos de esta lista, en una secuencia apropiada.
ListIterator<E>	listIterator(int index) Retorna un iterador de lista sobre los elementos de esta lista, en una secuencia apropiada, comenzando desde la posición especificada.
E	remove(int index) Elimina el elemento ubicado en la posición especificada en esta lista (operación opcional).
boolean	remove(Object o) Elimina la primera ocurrencia del elemento especificado en esta lista, si está presente (operación opcional).
boolean	removeAll(Collection<?> c) Elimina de esta lista todos los elementos que están contenidos en la colección especificada (operación opcional).
default void	replaceAll(UnaryOperator<E> operator) Reemplaza cada elemento de esta lista con el resyultado de aplicar el operador especificado en ese elemento.
boolean	retainAll(Collection<?> c) Retiene en esta lista sólo los elementos que están contenidos en la colección especificada (operación opcional).
E	set(int index, E element) Reemplaza el elemento ubicado en la posición especificada, con el elemento especificado (operación opcional).
int	size() Retorna el número de elementos de esta lista.
default void	sort(Comparator<? super E> c) Ordena esta lista de acuerdo al orden inducido por el Comparator especificado.
default Spliterator<E>	spliterator() Crea un Spliterator sobre los elementos de esta lista.
List<E>	subList(int fromIndex, int toIndex) Retorna una vista de una porción de esta lista entre el índice *fromIndex*, inclusive, y

	el índice toIndex, exclusive.
Object[]	toArray() Retorna un arreglo conteniendo todos los elementos de esta lista en una secuencia apropiada (desde el primero hasta el últimto).
<T> T[]	toArray(T[] a) Retorna un arreglo conteniendo todos los elementos de esta lista en una secuencia apropiada (desde el primero hasta el últimto); el tipo de tiempo de ejecución del arreglo retornado es el del arreglo especificado.

Para ahorrar algo de trabajo al programador, *Java* provee ciertas clases abstractas que ya implementan la interface *List* y proveen una implementación adecuada de la mayor parte de estos métodos (y de otros que a su vez provienen de la interface *Collection*). Dos de esas clases son muy útiles: *AbstractList* y *AbstractSequentialList* (ver diagrama de clases siguiente):



La clase abstracta *AbstractCollection* provee una implementación esencial de los métodos de la interface *Collection*, y de ella deriva a su vez la clase *AbstractList* que es superclase de la clase *AbstractSequentialList*. Y las dos últimas implementan la interface *List*.

La clase abstracta *AbstractList* provee una implementación básica (a modo de plantilla) de la interface *List* para minimizar el esfuerzo requerido para implementar esa interface cuando se diseñan estructuras lineales basadas en almacenamiento de *acceso directo* (como un arreglo, o una lista soportada sobre un arreglo). Para la implementación de estructuras lineales basadas en *acceso secuencial* (como una lista ligada), debería usarse como base la clase *AbstractSequentialList* en lugar de *AbstractList*.

De hecho, en la implementación nativa de Java para su clase *java.util.ArrayList*, la misma deriva de *AbstractList*, mientras que *java.util.LinkedList* deriva de *AbstractSequentialList* (ver diagrama de clases anterior).

Por lo tanto, si un programador desea implementar alguna clase que permita modelar una lista basada en un soporte de acceso directo, puede emular lo que hace Java con su propia clase *java.util.ArrayList*: derivarla de la clase *AbstractList*, la cual le facilita mucho trabajo ya desarrollado. Y si desea implementar una clase para modelar una lista basada en la idea de lista ligada, debería pues derivarla desde *AbstractSequentialList* y aprovechar lo que esa clase ya provee. En nuestro caso, la clase *TSBArrayList* se ha hecho derivar desde *AbstractList*.

Lo queda es analizar las convenciones y recomendaciones que la propia documentación javadoc provee para cada una de estas clases cuando se hace un trabajo similar al que estamos exponiendo. Veremos que en esencia no es demasiado trabajo:

- ✓ Si se deriva desde *AbstractList* (para una lista con soporte de acceso directo):
    a. Si la lista es *inmodificable*, el programador sólo necesita derivar esta clase y proveer implementaciones para los métodos *get(int)* y *size()*.
    b. Si la lista es modificable (esto es, algún elemento pre-existente puede cambiar por otro, pero sin cambiar el tamaño de la lista), el programador debe ***adicionalmente*** sobre-escribir el método *set(int, E)* (el cual en caso contrario lanzará una excepción de la clase *UnsupportedOperationException* al ser invocado).
    c. Si además el tamaño de la lista puede cambiar (ya sea que aumente o disminuya), entonces el programador debe ***adicionalmente*** sobre-escribir los métodos *add(int, E)* y *remove(int)*.
    d. El programador debería también proveer una implementación del *constructor sin parámetros*, y también del "*constructor collection*" (es decir, del constructor que toma como parámetro a otra colección y crea la lista copiando el contenido de esa otra colección).
    e. A diferencia de otras implementaciones de colecciones abstractas, en este caso el programador *no está obligado a implementar una clase iteradora* ni tampoco a dar una implementación de los métodos *iterator()* y *listIterator()*: la clase *AbstractList* ya provee una implementación apropiada basada en el uso de los métodos de acceso directo *get(int)*, *set(int, E)*, *add(int, E)* y *remove(int)*.
    f. Finalmente un detalle técnico: la clase *AbstractList* dispone de un atributo protegido de tipo *int*, llamado *modCount*. Ese atributo se usa para llevar la cuenta de la cantidad operaciones realizadas sobre la lista que *modificaron su estructura*, es decir, operaciones que *cambiaron el tamaño de la lista*. Usando ese atributo, las clases iteradoras pueden inferir si la lista ha cambiado mientras se está realizando concurrentemente un recorrido con algún iterador, y en ese caso, interrumpir de inmediato el proceso iterador lanzando una excepción de la clase *ConcurrentModificationException*. Si esto

se aplica, se dice que la clase sigue una estrategia de *fail-fast iterator* (o *iterador de respuesta rápida a fallos*). No hay que alarmarse por este detalle: si el programador desea una implementación *fail-fast iterator*, ***lo único que debe hacer es sumar 1 al atributo modCount*** en cada método que efectivamente lleve a cabo una modificación estructural de la lista (operación que implique un cambio de tamaño). El control de la falla en un contexto concurrente y el lanzamiento de la excepción, es llevado a cabo por los métodos ya implementados en la clase *AbstractList*. Y si el programador no desea una implementación *fail-fast iterator*, simplemente puede ignorar el atributo *modCount* y dejar que cada iterador haga lo que pueda cuando recorra una lista y esta pueda cambiar durante el recorrido (en este caso, no hay seguridad de obtener resultados correctos, e incluso podría ocurrir de todos modos una excepción si el iterador llega a un punto (por ejemplo) en el que no pueda recuperar el próximo elemento).

- ✓ Si se deriva desde *AbstractSequentialList* (para una lista ligada):

  a. Para implementar una lista ligada,el programador sólo necesita derivar esta clase y proveer implementaciones para los métodos *listIterator()* y *size()* (y obviamente deberá proveer una clase iteradora que implemente *ListIterator*). Si la lista es *inmodificable*, el programador sólo necesita implementar los métodos *hasNext()*, *next()*, *hasPrevious()*, *previous()* e *index()* de la clase iteradora que programe.

  b. Si la lista es modificable (esto es, algún elemento pre-existente puede ser reemplazado por otro pero sin cambiar el tamaño de la lista) entonces el programador debería ***adicionalmente*** implementar el método *set()* de la clase iteradora para *listIterator()*.

  c. Si la lista es de tamaño variable, entonces el programador debería ***adicionalmente*** implementar los métodos *remove()* y *add()* de la clase iteradora para *listIterator()*.

  d. El programador debería también proveer una implementación del *constructor sin parámetros*, y también del "*constructor collection*" (es decir, del constructor que toma como parámetro a otra colección y crea la lista copiando el contenido de esa otra colección).

  e. También aquí es válido considerar las recomendaciones respecto al atributo *modCount* heredado desde *AbstractList* (ver para ello el *punto f* de las recomendaciones para implentar una lista derivando desde *AbstractList*.)

Tanto si se deriva desde *AbstractList* como si se lo hace desde *AbstractSequentialList*, la documentación para cada uno de los métodos no abstractos en cada clase describe su implementación en detalle. Y por supuesto, cada uno de esos métodos puede ser sobre-escrito si la colección que está siendo diseñada admite implementaciones más eficientes.

Con estos elementos ya presentados, sólo nos queda ver en detalle la implementación final de nuestra clase *TSBArrayList*, que se define esencialmente así:

```
public class TSBArrayList<E> extends AbstractList<E>
             implements List<E>, RandomAccess, Cloneable, Serializable
{
    // 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;

    // resto de la clase aquí...
}
```

Note que los atributos de la clase son los mismos que ya hemos incluido en la versión original de la clase, con las mismas especificaciones y usos.

La interface *RandomAccess* es una *interface vacía o de marcado*, que sirve para que una clase notifique que admite operaciones de acceso directo a sus elementos (normalmente de tiempo constante). De esta forma, un algoritmo genérico (que tome un parámetro polimórfico) puede chequear si una instancia pertenece a una clase que haya implementado *RandoAccess* y modificar el comportamiento de tal algoritmo para aplicar técnicas más veloces que aprovechen el acceso directo. Por el momento, es suficiente con saber que el programador sólo debe indicar que la clase implementa *RandomAccess*. Y algo similar sucede con la interface *Serializable* (ya conocida) que también es una interface vacía cuya implementación se usa para notificar que la clase admite la serialización de sus instancias.

La interface *Cloneable* también es una interface vacía, pero en este caso hay algún trabajo extra para realizar: La implementación de esta interface se usa para notificar que la clase admite la operación de *clonar sus instancias invocando al método clone()* (que NO viene especificado en la interface *Cloneable*, sino en la clase *Object*, por lo que toda clase dispone de la implementación esencial del método). La idea es la siguiente: si una instancia de una clase *X* invoca a *clone()* pero la clase *X* no implementó *Cloneable*, se producirá una excepción de la clase *CloneNotSupportedException*. La clase X ***debe*** implementar *Cloneable* si se tiene pensado invocar a *clone()* con sus instancias.

Si se usa directamente el método *clone()* heredado desde *Object*, este método creará y retornará una *copia superficial* del objeto: todos sus atributos de tipo primitivo serán efectivamente replicados y copiados en el nuevo objeto, pero para los atributos que sean referencias a otros objetos, este método replicará las referencias (sin copiar los objetos en sí mismos). Por lo tanto, posiblemente el programador querrá implementar su propia versión de *clone()* en las clases que implementen *Cloneable*. En ese caso, lo aconsejable es que en la nueva versión del método se comience invocando al método *clone() de la superclase* (y si todas las clases siguen esta convención, finalmente todas invocarán a *Object.clone()*). Con esto se garantiza que se creará un objeto de la clase correcta, con sus atributos primitivos correctamente replicados, quedando para el programador la tarea de modificar en la copia clonada los atributos que sean referencias. En nuestra clase *TSBArrayList* el método *clone()* se redefinió así:

```
    public Object clone() throws CloneNotSupportedException
    {
        TSBArrayList<?> temp = (TSBArrayList<?>) super.clone();
        temp.items = new Object[count];
```

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

        // fail-fast iterator para el nuevo objecto...
        // modCount se hereda desde AbstractList y es protected...
        temp.modCount = 0;

        return temp;
    }
```

El método define en su cabecera que puede lanzar una excepción de la clase *CloneNotSupportedException* (lo cual teóricamente ocurrirá si se invocase a este método sin que la clase *TSBArrayList* implemente *Cloneable*… pero en la práctica eso no pasará, ya que efectivamente nuestra clase hizo esa implementación).

El método comienza creando el nuevo objeto (que finalmente será el que se retorne), pero invocando a *super.clone()* para hacer esa creación. Con esto, el nuevo objeto *temp* tendrá todos sus atributos con los mismos valores que el objeto actual (entre ellos, *initial_capacity* y *count*). Note que *clone()* retorna una referencia de la clase *Object*, y por eso se aplica una operación de casting explícito para convertir el tipo del resultado a *TSBArrayList*. Para evitar que la referencia *temp.items* apunte al mismo arreglo referenciado por *this.item* (que es el efecto de la copia superficial), se crea nuevamente y por separado el arreglo *temp.items* y se usa luego *System.arraycopy()* para copiar las direcciones contenidas en *this.items* (y en este momento, note que *clone()* replicará la lista, pero no los objetos contenidos en ella). Finalmente, se pone a cero el valor del atributo *temp.modCount* (heredado desde *AbstractList*) para activar desde cero el mecanismo *fail—fast iterator* en el objeto recién creado (y evitar que ese atributo tenga en *temp* el mismo valor que tenía en *this*…). Cuando todo esto ha sido concluido, el método termina retornando el nuevo objeto.

Como la clase *TSBArrayLIst* deriva de *AbstractList*, entonces la mayor parte del trabajo de implementar los métodos de la interface *List* está ya realizada (aunque sea a un nivel esencial). El contenido completo de la clase (exceptuando comentarios javadoc) es el que sigue:

```
import java.io.Serializable;
import java.util.AbstractList;
import java.util.Collection;
import java.util.List;
import java.util.RandomAccess;

public class TSBArrayList<E> extends AbstractList<E>
       implements List<E>, RandomAccess, Cloneable, Serializable
{

    private Object[] items;
    private int initial_capacity;
    private int count;

    public TSBArrayList()
    {
        this(10);
    }

    public TSBArrayList(Collection<? extends E> c)
    {
        this.items = c.toArray();
        initial_capacity = c.size();
        count = c.size();
```

```
}

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

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++;

    // fail-fast iterator...
    // modCount se hereda desde AbstractList y es protected...
    this.modCount++;
}

public void clear()
{
    items = new Object[initial_capacity];
    count = 0;

    // fail-fast iterator...
    // modCount se hereda desde AbstractList y es protected...
    this.modCount = 0;
}

public Object clone() throws CloneNotSupportedException
{
    TSBArrayList<?> temp = (TSBArrayList<?>) super.clone();
    temp.items = new Object[count];
    System.arraycopy(this.items, 0, temp.items, 0, count);

    // fail-fast iterator...
    // modCount se hereda desde AbstractList y es protected...
    temp.modCount = 0;

    return temp;
}

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;
    }

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

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

    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;

        // fail-fast iterator...
        // modCount se hereda desde AbstractList y es protected...
        this.modCount++;

        return (E) old;
    }

    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;
    }
```

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

    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();
    }

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

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

Los métodos y líneas de código que se marcaron en color rojo, son aquellos que responden a las convenciones para implementar una lista derivando desde *AbstractList*: *el constructor sin parámetros*, *el constructor collection*, los métodos *add(int, E)*, *remove(int)*, *size()*, *get(int)*, *set(int, E)* y *el incremento en 1 (o su vuelta cero si corresponde) del atributo modCount* heredado como *protected* desde *AbstractList*.

En rigor, los únicos métodos que no están especificados en la interface *List* (ni vienen heredados desde alguna superclase) son el *segundo constructor* y los métodos *ensureCapacity()* y *trimToSize()*: ambos son específicos y definidos *ad hoc* en la clase *java.util.ArrayList* (y nosotros decidimos mantenerlos en nuestra implementación de emulación). Los métodos *clone()* y *toString()* son muy especiales (como ya sabemos…) y vienen heredados desde *Object*.

Del resto de los métodos que figuran en nuestra implementación (*clear()*, *contains()* e *isEmpty()*) sólo *clear()* necesitaba sobre-escribirse (para asegurarnos que al volver a crear el arreglo *items* se lo haga con tamaño *initial_capacity*, lo cual no está especificado que ocurrra en el método *clear()* original…). Los otros dos se mantuvieron simplemente por razones de claridad, pero no era estrictamente necesario.

Nuestra clase *TSBArrayList* esencialmente emula a la clase *java.util.ArrayList* en toda su funcionalidad, salvo por un detalle: la clase *java.util.ArrayList* admite que se inserten o se agreguen en la lista *elementos nulos* (referencias valiendo *null*), mientras que en nuestra implementación de *TSBArrayList* los elementos nulos son rechazados por nuestra versión del método *add(int, E)* (y como este método en última instancia es el que se invoca desde

todo otro método de inserción, el resultado es que los elementos nulos nunca son aceptados).

No deje de observar que en esta versión final de la clase, no hemos incluido ninguna clase iteradora ni hemos implementado tampoco los métodos *iterator()* o *listIterator()*… Según se expuso, la propia clase *AbstractList* se hace cargo de todas estas tareas, proveyendo una clase iteradora propia (que obviamente utiliza los métodos de acceso directo *get(int)* y *remove(int)*, entre otros).

El modelo *TSB-ArrayList02* que acompaña a esta ficha incluye una clase *Test*, que contiene un método *main()* para probar el funcionamiento de (casi) todo lo que aquí hemos presentado. Dejamos el análisis de ese *main()* para el estudiante.

Obviamente, nada impide que el programador tome el control y desarrolle sus propias implementaciones de los métodos de la interface *List* (y posiblemente fuese aconsejable hacerlo en algunos casos, para intentar alguna mejora en los tiempos de ejecución de ciertos métodos), pero la idea central de este desarrollo no era esa, sino mostrar el camino para hacer una implementación rigurosa… lo que efectivamente acabamos de terminar de hacer. ☺…

2.] Fundamentos de Testing Unitario: clases *JUnit*.

Una vez que una clase ha sido desarrollada llega el momento de comprobar si todo está en orden y cada método funciona como se esperaría y hace lo correcto. Esta fase de prueba suele designarse como *fase de testing*, y no siempre es una tarea sencilla u obvia. No es suficiente con comprobar que la clase compila y que en un pequeño programa de prueba las cosas salieron como se esperaba (como es el caso de nuestros *main()* de prueba en cada uno de los modelos que hemos presentado hasta ahora).

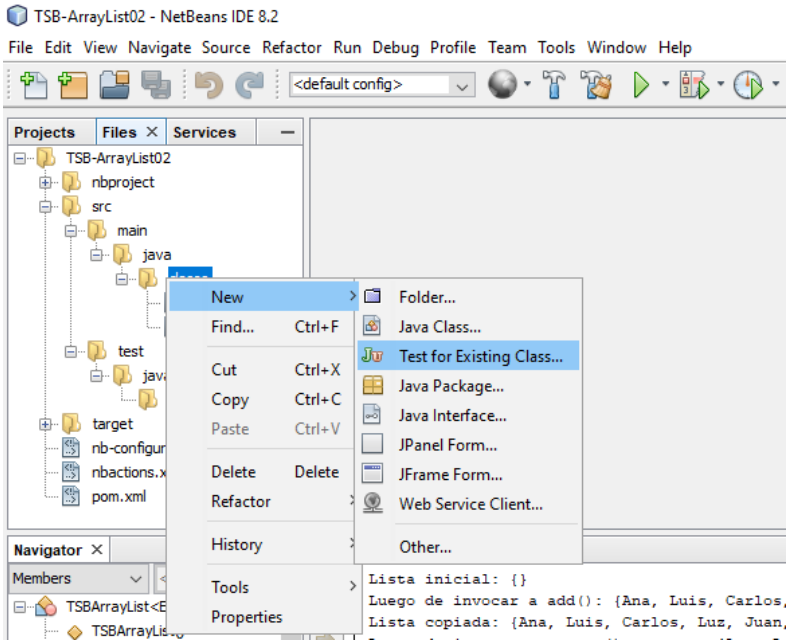
Una clase se diseña para llevar a cabo tareas muy especializadas y en contextos no siempre predecibles por el analista/diseñador/programador original. Cuando la clase efectivamente entre en producción en un sistema complejo, cualquier pequeño error o detalle que podría haber parecido sin importancia, cobrará fuerza y podría replicarse de forma insospechada, haciendo que la aplicación entera falle y provocando más de un disgusto.

Evidentemente, se impone que la *fase de testing* se diseñe ella misma con todo rigor profesional y (si es posible) contando con herramientas de software que ayuden a automatizarla y controlarla en profundidad. En ese sentido, *Java* es más que un simple lenguaje de programación: es una plataforma profesional de desarrollo, que entre sus muchas prestaciones incluye herramientas de testing automático que se designan en general como *herramientas de testing unitario* y se conocen en la plataforma bajo la denominación de *clases JUnit* (o clases de *Unit Test*).

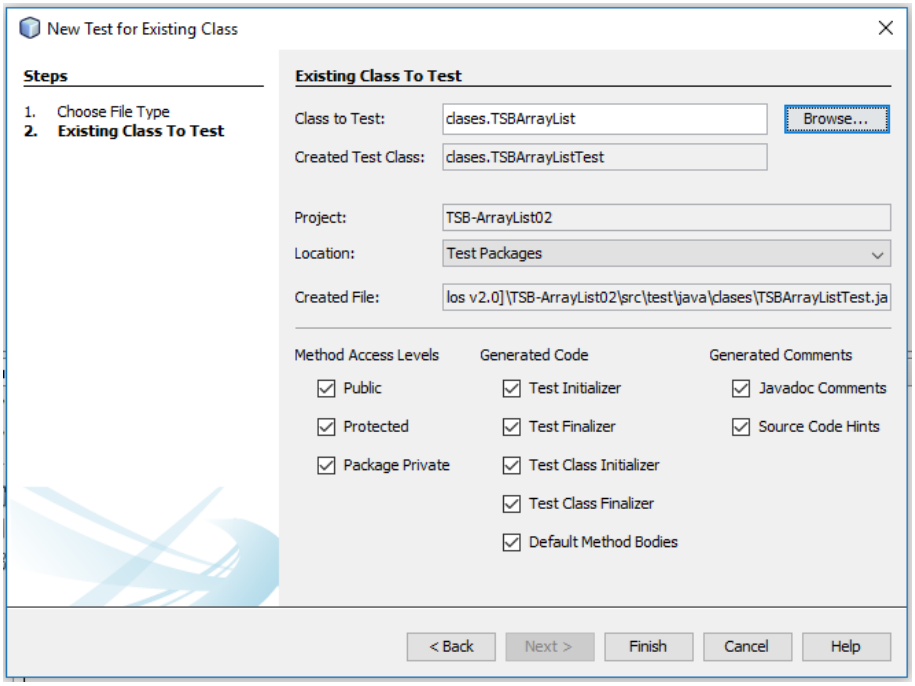
Una *clase JUnit* es una clase *Java* que se crea para permitir el testing de otra clase (que designaremos como la *clase testeada*). Cada método de la *JUnit* está orientado a comprobar el funcionamiento de alguno de los métodos de la clase testeada, y la idea es que Java *brinda mecanismos automáticos de soporte para medir los resultados al ejecutar una JUnit*.

Cada IDE profesional para desarrollo en Java provee mecanismos transparentes para la creación de clases *JUnit*. Supongamos que se desea crear una *JUnit* para nuestra clase

TSBArrayList (provista en el modelo *TSB-ArrayList02* que acompaña a esta ficha)[2]. Lo primero es asegurarse de que el proyecto que contiene a la clase esté compilado y sin errores. Luego, apuntar con el mouse a la carpeta del proyecto que contiene a la clase, hacer click derecho, seleccionar la opción *New* en el menú contextual, y a continuación la opción *Test for Existing Class* (ver imagen a continuación):



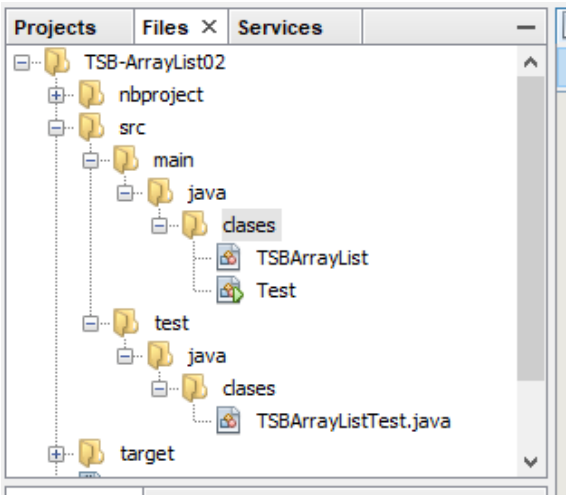
Una vez seleccionada esa opción, la ventana que aparece permite seleccionar la clase a testear, entre otros elementos:



[2] El proyecto *TSB-ArrayList02* fue creado en *NetBeans* mediante el plugin de *Maven*. Los pasos que se describen para crear una *JUnit* en *NetBeans* son apropiados para un proyecto soportado por *Maven*, pero si no fuese el caso, los pasos a seguir en *NetBeans* para un proyecto normal son muy similares.

Es típico que el nombre de la clase *JUnit* coincida con el nombre de la clase testeada, pero agregando al final el sufijo *Test*. Así, una clase *JUnit* para la clase *TSBArrayList* puede llamarse *TSBArrayListTest*. Las casillas de verificación que aparecen al final y a la izquierda permiten seleccionar el nivel de acceso de los ítems de la clase testeada que serán controlados. Las casillas centrales permiten configurar el tipo de código fuente que se agregará en la clase *JUnit*. Y las casillas de la derecha permiten configurar el tipo de comentarios que serán agregados en el código fuente de la *JUnit*. Por ahora, marque todas esas casillas y presione el botón *Finish*.

Dentro de la carpeta del proyecto, se creará una subcarpeta *test* que contendrá otras carpetas. La clase *TSBArrayListTest* estará alojada en una de ellas (*test\java\clases* en nuestro caso):



La primera parte de la clase así generada luce de esta forma:

```
package clases;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class TSBArrayListTest
{

    public TSBArrayListTest()
    {
    }

    @BeforeClass
    public static void setUpClass()
    {
    }

    @AfterClass
    public static void tearDownClass()
    {
```

```
    }

    @Before
    public void setUp()
    {
    }

    @After
    public void tearDown()
    {
    }

    // resto de la clase aquí...
}
```

Los métodos vacíos (que por ahora seguirán vacíos) que se ven en este momento, son métodos especiales que serán ejecutados automáticamente en distintos momentos del proceso de testing, cuando se pida ejecutar esta clase. Las líneas que comienzan con el símbolo *@* antes de la cabecera de cada método son expresiones conocidas como *anotaciones* (o *annotations*), y son usadas por Java para marcar esos métodos como métodos de test, o para configurar esos métodos. Las directivas *import* de la forma:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
```

son las que permiten acceder y usar esas anotaciones en el código fuente de la clase. Las más comunes de esas anotaciones se ven y explican en la tabla que sigue:

Annotation	**Efecto**
@After	Marca un método para ser ejecutado cuando ha finalizado la ejecución de cada método marcado como @Test. Si la clase tiene n métodos marcados como @Test, el método @After se ejecutará n veces.
@Before	Marca un método para ser ejecutado antes de que comience ejecutar a cada método marcado como @Test (típicamente para tareas de inicialización del entorno, creación de alguna estructura común, etc.) Si la clase tiene n métodos marcados como @Test, el método @Before se ejecutará n veces.
@AfterClass	Marca un metodo para ser ejecutado *sólo una vez*, cuando todo el proceso de testing de la clase ha terminado.
@BeforeClass	Marca un metodo para ser ejecutado *sólo una vez*, antes de que el proceso completo de testing de la clase comience.
@Ignore	Marca a un método como deshabilitado (es práctico cuando un método ha dejado de ser útil pero no se desea eliminarlo de la *JUnit*).

@Test	Identifica a un método como un método de testing.
@Test (expected=exception.class)	Provoca que el test del método lance un fallo si no se obtiene la excepción indicada al ejecutarlo. Note que debe citarse la excepción por su nombre, seguida de un punto y el acceso al atributo *class* de la misma…
@Test (timeout = 100)	Provoca que el test del método lance un fallo si su ejecución demora más de 100 milisegundos (o el tiempo que sea que se haya estipulado en *timeout*).

En la clase *JUnit* que estamos analizando, se generaron además varios otros métodos, marcados con la anotación *@Test*. Analicemos el primero de ellos:

```
@Test
public void testAdd() {
    System.out.println("add");
    int index = 0;
    Object e = null;
    TSBArrayList instance = new TSBArrayList();
    instance.add(index, e);
    // TODO review the generated code and remove the call to fail.
    fail("The test case is a prototype.");
}
```

Por cada método disponible en la clase testeada *TSBArrayList*, se generó un método similar a este en la clase JUnit *TSBArrayListTest*. El método *testAdd()* del ejemplo, fue generado para testear el funcionamiento del método *add(int, E)* de la clase *TSBArrayList* (lo cual está anunciado en el comentario javadoc del método, y de todos modos resaltamos marcando la llamada a ese método en color rojo).

El código fuente propuesto hasta aquí para el método *testAdd()* es simplemente una plantilla para que el programador modifique según su necesidad. Por ahora el método sólo muestra un mensaje en pantalla, crea una instancia de la clase *TSBArrayList*, declara una variable *index* que enviará como parámetro a *add()*, así como otra variable *e* con el mismo propósito, e invoca al método *add()*. El comentario que comienza con *// TODO* sugiere revisar y cambiar el código fuente actual por aquel que prefiera el programador, y finalmente eliminar la llamada que por default se hace al método *fail()* al final del bloque. Este método se usa para dar por finalizada la ejecución de un método de testing cuando se ha detectado algún error durante el proceso que haga imposible continuarlo: en este momento el error es que el método no está listo para ser ejecutado porque sólo es una plantilla… El resto de los métodos de la clase tiene un aspecto muy similar.

Lo que debemos hacer ahora, es cambiar las plantillas que hayan sido generadas automáticamente para cada método por el código de testing que creamos conveniente. De los cuatro métodos iniciales (por ahora vacíos) sólo modificaremos el método marcado como *@Before* (en este caso, ese método se llama *setUp()*): este método será ejecutado tantas veces como métodos *@Test* haya en la clase, y siempre antes de invocar al cada método *@Test*. Por ese motivo, es útil que el método *@Before* realice alguna inicialización que pueda ser útil. Sugerimos a modo de ejemplo el siguiente esquema:

```
public class TSBArrayListTest
{
    private TSBArrayList<Integer> instance;
```

```
    public TSBArrayListTest()
    {
    }

    @BeforeClass
    public static void setUpClass()
    {
    }

    @AfterClass
    public static void tearDownClass()
    {
    }

    @Before
    public void setUp()
    {
        System.out.println("* UtilsJUnit4Test: @Before method");
        int data[] = {1, 5, 7, 6, 10, 8, 11};

        instance = new TSBArrayList<>();
        for(int i=0; i<data.length; i++)
        {
            instance.add(i, data[i]);
        }
    }

    @After
    public void tearDown()
    {
    }

    // resto de la clase aquí...
}
```

Hemos incluido en la clase *JUnit* la declaración de un atributo privado *instance* de tipo *TSBArrayList<Integer>*. De esta forma, todos los métodos de la clase tienen disponible una variable del tipo de la clase testeada y se facilita su acceso y uso. Además, hemos incluido en el método *setUp()* (marcado como *@Before*) un bloque de código que crea una lista, asigna su dirección en el atributo *instance*, y luego agrega datos en la lista mediante el método *add(int, E)* (que de todos modos será testeado con el método *testAdd()* más abajo). La idea es que ese código será ejecutado antes de invocar a cada método *@Test*, por lo que cada uno de esos métodos ya tendrá una lista creada de antemano con la que podrá hacer pruebas (en lugar de repetir este código una vez por cada método…) Los otros tres métodos iniciales pueden dejarse vacíos (no es obligatorio programarlos todos).

Luego modificaremos el método *testAdd()* para llevar a cabo pruebas más rigurosas del método *add(int, E)*. La forma de hacer esas pruebas rigurosas, consiste en verificar ciertas condiciones madiante métodos provistos por la clase *org.junit.Assert* (que ya está

importada en nuestro modelo). Un método *assert* permite comparar el resultado entregado por un método con un posible valor prefijado, y en caso de no coincidir, arroja un error de la clase *AssertionError* (derivada de la clase *Error* de la jerarquía de excepciones de Java). Existen varios de esos métodos en la clase *Assert*. En la tabla siguiente mostramos algunos de los más comunes:

Método	**Efecto**
fail(msg)	Hace que el método que lo ejecuta finalice informando un fallo. El parámetro *msg* es la cadena que debe mostrarse cuando el método se interrumpa.
assertTrue(msg, expresión_lógica)	Verifica si la *expresión_lógica* tomada como parámetro es *true*. Si no lo es, lanza un *AssertionError*. El parámetro *msg* es opcional, y contiene el mensaje a mostrar en pantalla si se produce el *AssertionError*.
assertFalse(msg, expresión_lógica)	Verifica si la *expresión_lógica* tomada como parámetro es *false*. Si no lo es, lanza un *AssertionError*. El parámetro *msg* es opcional, y contiene el mensaje a mostrar en pantalla si se produce el *AssertionError*.
assertEquals(msg, esperado, actual)	Verifica si el objeto *esperado* es igual al objeto *actual*. Si no es así, lanza un *AssertionError*. El parámetro *msg* es opcional, y contiene el mensaje a mostrar en pantalla si se produce el *AssertionError*.
assertEquals(msg, esp, act, rango)	Usado para comprobar igualdad con valores de coma flotante. Verifica si el objeto *esp* es igual al objeto *act,* con un radio de tolerancia igual a *rango*. Si no es así, lanza un *AssertionError*. El parámetro *msg* es opcional, y contiene el mensaje a mostrar en pantalla si se produce el *AssertionError*.
assertNull(msg, objeto)	Verifica si el *objeto* tomado como parámetro es *null*. Si no es así, lanza un *AssertionError*. El parámetro *msg* es opcional, y contiene el mensaje a mostrar en pantalla si se produce el *AssertionError*.
assertNotNull(msg, objeto)	Verifica si el *objeto* tomado como parámetro NO es *null*. Si el *objeto* ES null, lanza un *AssertionError*. El parámetro *msg* es opcional, y contiene el mensaje a mostrar en pantalla si se produce el *AssertionError*.

Los métodos *assert* mostrados en la tabla anterior son solamente algunos de los más usados. La clase *org.junit.Assert* dispone de muchos más, que obviamente deberán ser estudiados y analizados por el programador cuando necesite otras funcionalidades de testing (para lo cual sólo hay que revisar la documentación javadoc de la clase *org.junit.Assert*).

Podemos ahora sugerir una estructura para el método *testAdd()*. Como el método *add(int, E)* de la clase *TSBArrayList* no retorna valor alguno pero puede lanzar una excepción de la forma *IndexOutOfBoundsException* en caso de índice fuera de rango, podemos hacer un planteo así:

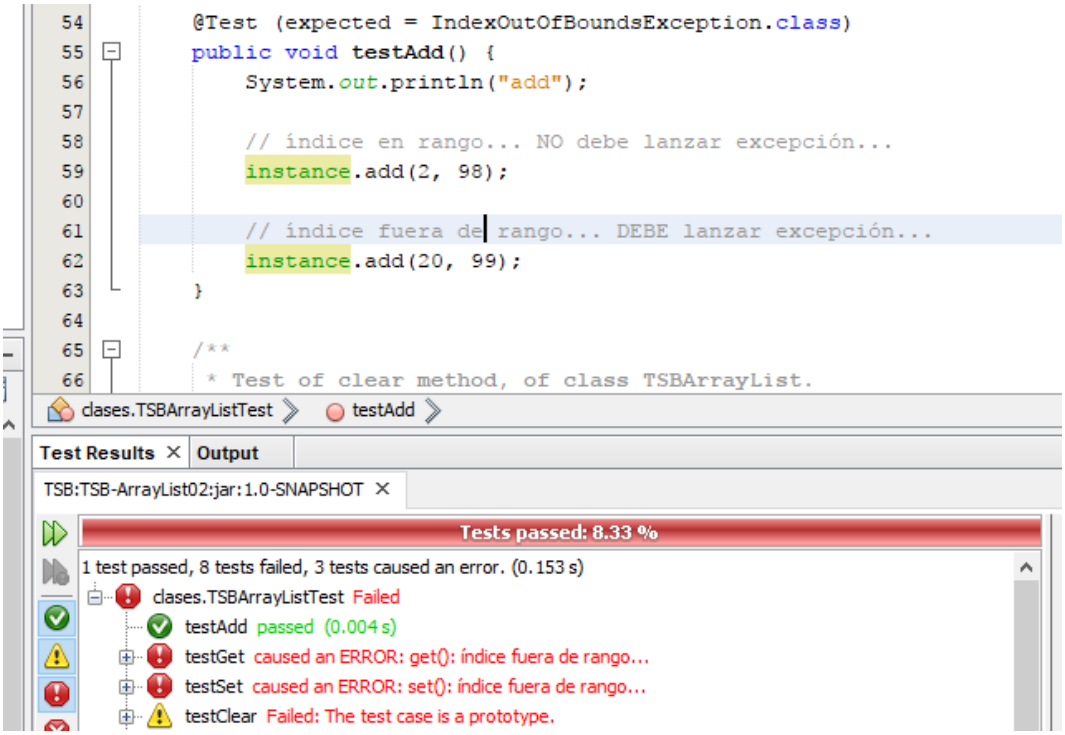
```
@Test (expected = IndexOutOfBoundsException.class)
public void testAdd() {
    System.out.println("add");

    // índice en rango... NO debe lanzar excepción...
    instance.add(2, 98);

    // índice fuera de rango... DEBE lanzar excepción...
    instance.add(20, 99);
}
```

Así planteado, el método define en su anotación *@Test* que espera que se produzca una *IndexOutOfBoundsException*. Si se ejecuta su bloque de acciones y la excepción citada NO se produce, el método acusará un fallo. Recuerde que antes de ejecutarse este método se ejecuta el método *@Before*, que crea la lista con 7 elementos. Como la segunda invocación a *add()* pide agregar un objeto en la posición 20 (claramente fuera de rango) la excepción se producirá (y el test no provocará un fallo).

Para ejecutar una clase *JUnit* (y comenzar con ello un test), simplemente haga click derecho en esa clase y seleccione la opción *Test File*. Con eso el test comenzará a ejecutarse, y en nuestro caso verá una salida como la que sigue:

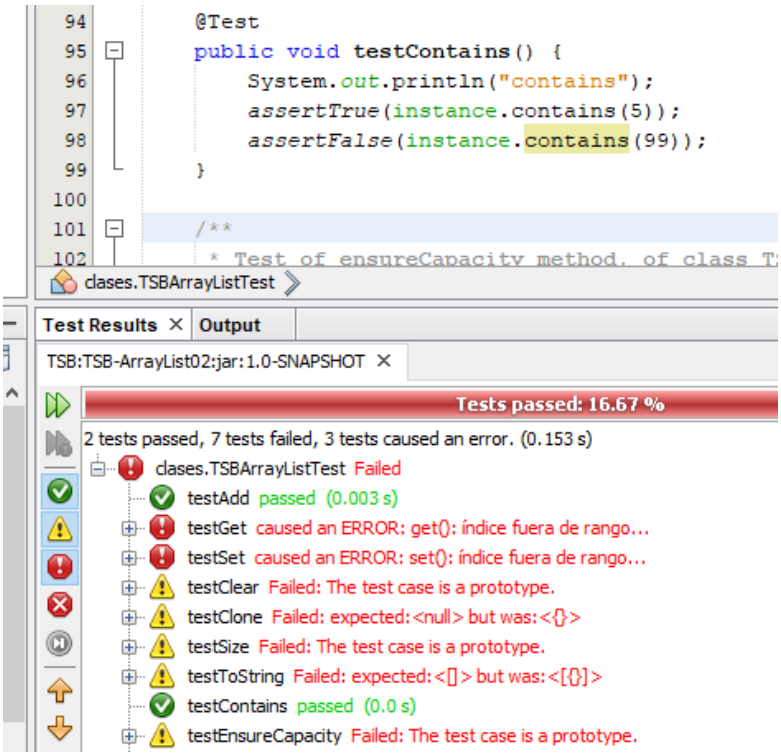


Puede verse en el marco de resultados del test (bloque *Test Results*) que el test para el método *add()* pasó con éxito (está marcado en verde y con la palabra *passed* agregada, además del tiempo de ejecución que tomó el proceso).

Se puede proceder en forma similar con el resto de los métodos. Por ejemplo, el método *contains()* podría ser testeado así:

```
@Test
public void testContains() {
    System.out.println("contains");
    assertTrue(instance.contains(5));
    assertFalse(instance.contains(99));
}
```

Otra vez: antes de ejecutarse el método *testContains()*, se ejecuta el método *@Before* que crea una lista con siete elementos (entre los que está el 5). Estamos usando aquí el método *assertTrue()* para comprobar que el 5 está en la lista, y el método *assertFalse()* para comprobar que el 99 no está. En estas condiciones, el test pasará sin fallos:

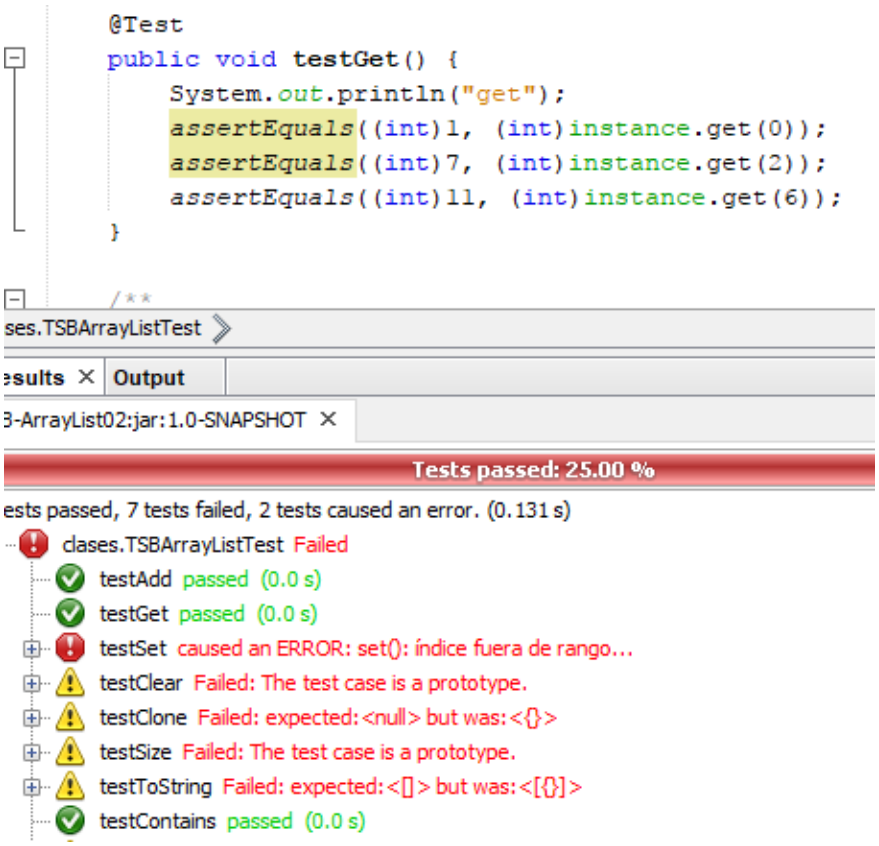


Una versión rápida del método *testGet()* podría ser la que sigue:

```
@Test
public void testGet() {
    System.out.println("get");
    assertEquals((int)1, (int)instance.get(0));
    assertEquals((int)7, (int)instance.get(2));
    assertEquals((int)11, (int)instance.get(6));
}
```

Como se ve, estamos usando tres invocaciones a *assertEquals()* para comprobar si lo que retorna *get()* es lo esperado en cada caso. Los valores deben ser convertidos a *int* con casting explícito para evitar un error de compilación, ya que la clase *Assert* dispone de dos

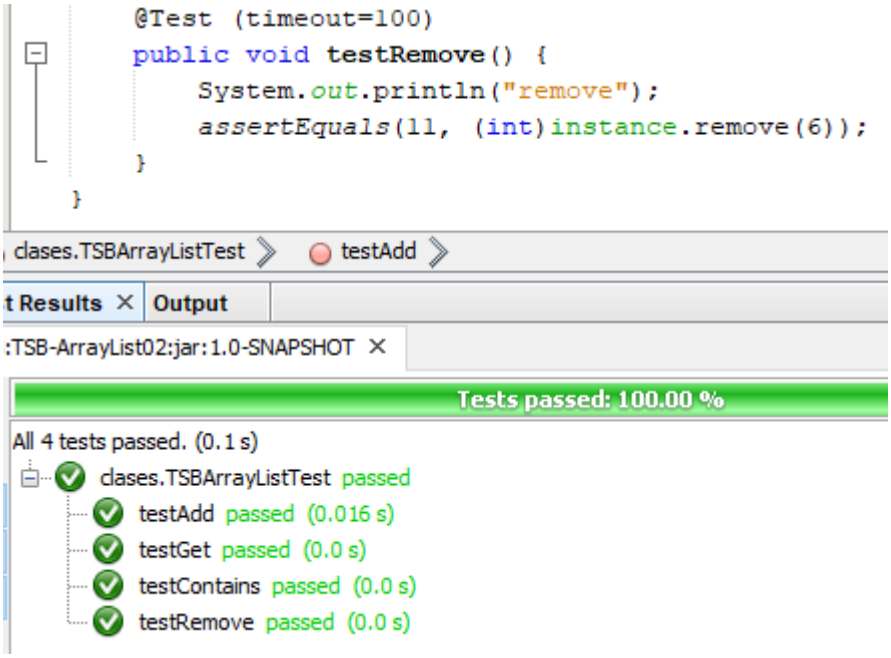
versiones de este método: una con parámetros de tipo *long* y la otra con parámetros de tipo *Object*. Sin el casting explícito que se muestra, el compilador no tiene forma de distinguir entre ambas versiones y acusa un error. Así como está planteado, el método ejecuta sin fallos:



Para terminar estar breve introducción al uso de clases *JUnit*, mostramos el metodo *testRemove()*:

```
@Test (timeout=100)
public void testRemove() {
    System.out.println("remove");
    assertEquals(11, (int)instance.remove(6));
}
```

El método simplemente invoca a *remove()* para eliminar y retornar el elemento de la casilla 6 de la lista, y se usa *assertEquals()* para comprobar que el valor retornado sea el 11 (que es el que efectivamente estaba en la casilla 6). La novedad es que en la anotación *@Test* se agregó la expresión *(timeout=100)*, lo cual significa que se controlará que el proceso completo no insuma más de 100 milisegundos en total. Si el tiempo de ejecución fuese superior a ese valor, el método lanzará un fallo. En las condiciones dadas, el método ejecuta sin fallos (en la captura de pantalla que sigue, mostramos el mismo proceso de testing, pero hemos eliminado de la clase *JUnit* los métodos que no hemos usado en este ejemplo, para simplificar):



Dejamos el desarrollo del resto de los métodos de testing para el estudiante.