Capítulo 9. Estructuras dinámicas lineales

A. J. Pérez

Estructuras dinámicas lineales

Las estructuras dinámicas se caracterizan por tener un número de elementos que no tiene límite teórico durante la ejecución de un programa. Las estructuras dinámicas se contraponen a las estructuras estáticas -típicamente arrays- que requieren conocer el número de elementos antes de poder usar la estructura.

Las estructuras de datos son lineales si en su organización cumplen la condición de que después, o antes, de un elemento puede existir como máximo un solo elemento. Hay que destacar que los arrays son estructuras lineales de naturaleza estática.

Para el tratamiento y almacenamiento de datos en estructuras lineales existen tres tipos básicos:

- *Listas*
- *Colas*
- *Pilas*

Cada una de estas estructuras responden a unas características, necesidades y restricciones específicas según un *tipo de dato abstracto (TDA)* estándar que es independiente de la manera de implementación utilizada.

En general, los *TDA* constituyen una definición del comportamiento de las estructuras desde el punto de vista de las operaciones y propiedades permitidas, sin importar las aplicaciones específicas, las alternativas de implementación y el rendimiento. Podrían por ejemplo, implementarse con características internas estáticas o dinámicas sin afectar a su funcionalidad externa.

Listas

Son una abstracción para representar y organizar colecciones de datos en secuencias y series lineales de elementos, dispuestos con algún criterio.

Se puede decir que una *lista* es una secuencia organizada de elementos para un fin concreto.

Por ejemplo la lista de la compra a realizar en una tienda. En la lista podemos leer cada uno de los elementos (cosas o productos a adquirir), se pueden añadir nuevas cosas en la misma; podemos eliminar (borrar) algo, o se pueden clasificar u ordenar por algún criterio.

Las listas como TDA

Una *lista* es una estructura de datos lineal que contiene una serie de elementos gestionados dinámicamente. Las listas **tienen, como característica, una longitud o tamaño** (número de elementos) **que aumenta o disminuye dinámicamente sin límite máximo teórico; sus elementos siempre tienen una disposición lógica secuencial.**

Una *lista* permite añadir o eliminar elementos en cualquier punto de su estructura y realizar diferentes comprobaciones sobre los elementos almacenados.

Un ejemplo de *TDA* es el proporcionado en la *API Collection* de Java con la *interfaz* `java.util.List`. Teniendo en cuenta que una ***interface*** Java especifica, principalmente, la funcionalidad que proporciona una clase; la interfaz `java.util.List`, fija los principales métodos para el funcionamiento previsto de una lista:

`void add(Object)`	- Añade un elemento al final de la lista.
`void add(int, Object)`	- Inserta un elemento en la posición indicada por un índice.
`void clear()`	- Elimina todos los elementos de la lista.
`boolean contains(Object)`	- Comprueba si el elemento está incluido en la lista.
`Object get(int)`	- Obtiene el elemento de la posición indicada por índice.
`int indexOf(Object)`	- Devuelve la posición del elemento.
`boolean isEmpty()`	- Comprueba si la lista está vacía.
`boolean remove(Object)`	- Elimina el elemento correspondiente.
`Object remove(int)`	- Elimina el elemento de la posición indicada por índice.
`int size()`	- Número de elementos de la estructura.

Como se mencionó anteriormente, un *TDA* puede tener diferentes implementaciones internas; manteniendo la misma funcionalidad externa.

Lista basada en array estático

Los *arrays* disponen directamente de varias de las funciones indicadas en el *TDA Lista*, pero hay una diferencia importante: **Las *listas* deben permitir añadir nuevos elementos**, mientras que los arrays -por su naturaleza estática- tienen tamaño fijo.

Sin embargo, es posible la implementación de una *lista* con un *array* que incremente automáticamente su tamaño. Sería una *lista con implementación interna estática*:

```
public class ListaArray {

     // Atributos
     private Object[] arrayElementos;
```

```
	private int numElementos;
	private static final int TAMAÑO_INICIAL = 4;

	// Métodos
	/**
	 * Inicializa el array de elementos de la lista.
	 */
	public ListaArray() {
		arrayElementos = new Object[TAMAÑO_INICIAL];
		numElementos = 0;
	}

	 /**
	  * @return número de elementos actual en la lista.
	  */
	 public int size() {
		 return numElementos;
	 }

// ...

} // class
```

Se define el array en el que se van a mantener los elementos, así como un contador de los elementos almacenados y la constante para establecer el tamaño inicial.

En el constructor se crea el array con el tamaño inicial y se pone a cero el contador de elementos almacenados.

Las principales operaciones indicadas en el *TDA lista* son :

- `add()` implementa la operación de añadir un nuevo elemento al final de la lista.

```
	/**
	 * Añade un elemento a la lista
	 * @param elemento - el elemento a añadir
	 */
	public void add(Object elemento) {
		if (numElementos == 0) {
			arrayElementos[0] = elemento;
			numElementos++;
		}
		else {
			comprobarLlenado();
			arrayElementos[numElementos] = elemento;
			numElementos++;
```

```
        }
    }

    /**
     * Comprueba si el array si el array interno está casi lleno y lo copia
     * ampliando al doble su tamaño.
     */
    private void comprobarLlenado() {
        // El array interno está casi lleno, se duplica el espacio.
        if (numElementos + 1 == arrayElementos.length) {
            Object[] arrayAmpliado = new Object[arrayElementos.length*2];
            System.arraycopy(arrayElementos, 0,
                             arrayAmpliado, 0, numElementos);
            arrayElementos = arrayAmpliado;
        }
    }

    /**
     * Inserta un elemento en la posición especificada por el índice.
     * @param indice - indica la posición de inserción en la lista.
     * @param elemento - elemento a insertar.
     * @throws IndexOutOfBoundsException
     */
    public void add(int indice, Object elemento) {
        // El índice debe ser válido.
        if (indice >= numElementos || indice < 0) {
            throw new IndexOutOfBoundsException("Índice incorrecto: "
                                                + indice);
        }
        comprobarLlenado();

        // Inserción, desplaza los elementos desde índice indicado.
        if (indice < numElementos) {
            System.arraycopy(arrayElementos, indice, arrayElementos,
                             indice+1, numElementos - indice);
        }
        arrayElementos[indice] = elemento;
        numElementos++;
    }
```

En el método `add()` se comprueba que el array interno (*buffer*) tiene espacio. **Si el array interno está casi lleno, se amplía al doble de la capacidad actual y se mueven todos los elementos del array antiguo al recién creado.**

- `indexOf()` obtiene la *posición, base cero,* de un elemento de la lista.

- `clear()` borra todos los elementos de la lista.
- `contains()` comprueba si un elemento está en la lista.
- `get()` obtiene un elemento de la lista.

```
/**
 * Devuelve el índice de la primera ocurrencia para el objeto especificado.
 * @param elem - el elemento buscado.
 * @return el índice del elemento o -1 si no lo encuentra.
 */
public int indexOf(Object elem) {
      if (elem == null) {
             for (int i = 0; i < arrayElementos.length; i++) {
                    if (arrayElementos[i] == null) {
                           return i;
                    }
             }
       }
       else {
             for (int i = 0; i < arrayElementos.length; i++) {
                    if (elem.equals(arrayElementos[i])) {
                           return i;
                    }
             }
       }
       return -1;
 }

 /**
  * Borra todos los elementos de la lista.
  */
 public void clear() {
     arrayElementos = new Object[TAMAÑO_INICIAL];
     numElementos = 0;
 }

 /**
  * Comprueba si existe un elemento.
  * @param elem – el elemento a comprobar.
  * @return true - si existe.
  */
 public boolean contains(Object elem) {
    return indexOf(elem) != -1;
 }

 /**
  * Obtiene el elemento-dato por índice.
```

```
  * @param indice - posión relativa del nodo que contiene el elemento-dato.
  * @return el dato indicado por el índice de nodo; null si está indefinido.
  *@exception IndexOutOfBoundsException - índice no está entre 0 y numElementos-1.
  */
public Object get(int indice) {
      // El índice debe ser válido para la lista.
      if (indice >= numElementos || indice < 0) {
            throw new IndexOutOfBoundsException("Índice incorrecto: " + indice);
      }
      return arrayElementos[indice];
 }
```

- `remove()`, operaciones de eliminación de elementos (por índice y dado el objeto)

```
 /**
  * Elimina el elemento especificado en el índice.
  * @param indice - del elemento a eliminar.
  * @return - el elemento eliminado.
 * @exception IndexOutOfBoundsException - índice no está entre 0 y numElementos-1.
  */
public Object remove(int indice) {
      // El índice debe ser válido para la lista.
      if (indice >= numElementos || indice < 0) {
            throw new IndexOutOfBoundsException("Índice incorrecto: " + indice);
      }
      // Elimina desplazando uno hacia la izquierda, sobre la posición a borrar.
      Object elem = arrayElementos[indice];
      System.arraycopy(arrayElementos, indice+1, arrayElementos, indice,
      numElementos - (indice+1));

      // Ajusta el último elemento.
      arrayElementos[numElementos-1] = null;
      numElementos--;
      return elem;
}

 /**
  * Elimina el elemento especificado.
  * @param elemento - elemento a eliminar.
  * @return - el índice del elemento eliminado o -1 si no existe.
  */
public int remove(Object elem) {
      int indice = indexOf(elem);

      if (indice != -1) {
```

```
                remove(indice);
        }
        return indice;
}
```

Hay dos modalidades para eliminar un elemento:

- Una, por índice que localiza directamente el elemento a borrar y desplaza los elementos hacia la izquierda. Hay que borrar el duplicado del último elemento y actualizar el `numElementos`.
- Otra, dado el elemento que obtiene el índice y después se delega en el método `remove()` por índice.

Para probar la implementación realizada se crea una lista de compras y se realizan varias operaciones básicas:

```
public static void main(String[] args){
        ListaArray listaCompra = new ListaArray();
        listaCompra.add("Leche");
        listaCompra.add("Miel");
        listaCompra.add("Aceitunas");
        listaCompra.add("Cerveza");
        listaCompra.remove("Aceitunas");
        listaCompra.add(1, "Fruta");
        listaCompra.add(0, "Queso");
        listaCompra.add(4, "Verduras");

        System.out.format("Los %d elementos de la lista de la compra son:\n",
        listaCompra.size());
        for (int i = 0; i < listaCompra.size(); i++) {
                System.out.format("%s\n", listaCompra.get(i));
        }
        System.out.format("¿Hay pan en la lista? %b", listaCompra.contains("Pan"));
}
```

Salida:

```
Los 6 elementos de la lista de la compra son:
Queso
Leche
Fruta
Miel
Verduras
Cerveza
¿Hay pan en la lista? false
```

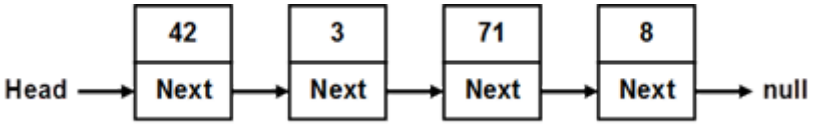
Lista enlazada

Las listas con implementación estática tienen un inconveniente que puede ser importante: Las operaciones de inserción y eliminación de elementos del array requieren la reorganización completa del array interno. Si la inserción o la eliminación son frecuentes, se penaliza bastante el rendimiento.

Las *listas enlazadas* mejoran claramente la eficiencia cuando hay muchos cambios en las posiciones intermedias. **La principal diferencia de las listas enlazadas frente a las estáticas radica en la organización y la estructura de cada elemento, que en la lista estática contiene sólo el dato almacenado, mientras que las listas enlazadas mantienen información sobre el siguiente elemento de la lista dentro de la memoria.**

Las listas enlazadas hacen una implementación independiente de la memoria subyacente utilizando un sistema de enlaces propio.

Así es como se representa lógicamente una *lista enlazada*:



Para la implementación de una lista con elementos enlazados se necesitan dos clases:

- La clase **Nodo** que representa un elemento de la lista junto con la referencia que permite el enlace recursivo al siguiente elemento de la lista con un objeto de la misma clase **Nodo**.
- La clase **`ListaEnlazada`** que contiene el *nodo inicial* o *cabeza*; es el `primero` de la serie de elementos enlazados y un contador para el control del total de elementos almacenados.

```
/**
 * Representa la implementación más sencilla y básica de una lista enlazada simple
 * con acceso sólo al principio de la serie de nodos.
 */
public class ListaEnlazada {
	// Atributos
	private Nodo primero;	// Referencia a nodo
	private int numElementos;

	// Métodos
	/**
	 * Constructor que inicializa los atributos al valor por defecto.
	 * Lista vacía.
	 */
	public ListaEnlazada() {
		primero = null;
		numElementos = 0;
	}
```

```
 //...

 } // class

/**
 * Representa la estructura de un nodo para una lista dinámica con enlace simple.
 */
class Nodo {
	// Atributos
	Object dato;
	Nodo siguiente;
	/**
	 * Constructor que inicializa atributos por defecto.
	 * @param elem - el elemento de información útil a almacenar.
	 */
	public Nodo(Object dato) {
		this.dato = dato;
		siguiente = null;
	}

} // class
```

La mayoría de las operaciones típicas indicadas en el *TDA* pueden son :

- `add()` permite añadir un nuevo elemento al final de la lista. Utiliza un método auxiliar privado, `obtenerNodo()` para obtener una referencia al nodo que hay en una determinada *posición o índice* de la lista.

```
 /**
  * Añade un elemento al final de la lista.
  * @param elem - el elemento a añadir.
  * Admite que el elemento a añadir sea null.
  */
 public void add(Object dato) {
	//variables auxiliares
	 Nodo nuevo = new Nodo(dato);
	 Nodo ultimo = null;
	 if (numElementos == 0) {
		// Si la lista está vacía enlaza el nuevo nodo el primero.
		primero = nuevo;
	  }
	 else {
		// Obtiene el último nodo y enlaza el nuevo.
		ultimo = obtenerNodo(numElementos-1);
		ultimo.siguiente = nuevo;
```

```
        }
        numElementos++;                // Actualiza el número de elementos.
    }

/**
* Obtiene el nodo correspondiente al índice. Recorre secuencialmente la cadena de
enlaces.
* @param indice - posición del nodo a obtener.
* @return - el nodo que ocupa la posición indicada por el índice.
*/
private Nodo obtenerNodo(int indice) {
        assert indice >= 0 && indice < numElementos;
        // Recorre la lista hasta llegar al nodo  de posición buscada.
        Nodo actual = primero;
        for (int i = 0; i < indice; i++)
                actual = actual.siguiente;
        return actual;
}
```

- `remove()` elimina el elemento situado en el índice especificado. Es algo más complicada que `add()`. Delega en `removePrimero()` y `removeIntermedio()` que a su vez también se apoya en el método `obtenerNodo()`.

```
/**
 * Elimina el elemento indicado por el índice. Ignora índices negativos
 * @param indice - posición del elemento a eliminar
 * @return - el elemento eliminado o null si la lista está vacía.
 * @exception IndexOutOfBoundsException - índice no está entre 0 y numElementos-1
 */
public Object remove(int indice) {
        // Lanza excepción si el índice no es válido.
        if (indice >= numElementos || indice < 0) {
                throw new IndexOutOfBoundsException("Índice incorrecto: " + indice);
        }
        if (indice > 0) {
                return removeIntermedio(indice);
        }
        if (indice == 0) {
                return removePrimero();
        }
        return null;
}

/**
```

```
 * Elimina el primer elemento.
 * @return - el elemento eliminado o null si la lista está vacía.
 */
private Object removePrimero() {
	//variables auxiliares
	Nodo actual = null;
	actual = primero;			// Guarda actual.
	primero = primero.siguiente;		// Elimina elemento del principio.
	numElementos--;
	return actual.dato;
}

/**
 * Elimina el elemento indicado por el índice.
 * @param indice - posición del elemento a eliminar.
 * @return - el elemento eliminado o null si la lista está vacía.
 */
private Object removeIntermedio(int indice) {
	assert indice > 0 && indice < numElementos;
	//variables auxiliares
	Nodo actual = null;
	Nodo anterior = null;
	// Busca nodo del elemento anterior correspondiente al índice.
	anterior = obtenerNodo(indice - 1);
	actual = anterior.siguiente;		// Guarda actual.
	anterior.siguiente = actual.siguiente;	// Elimina el elemento.
	numElementos--;
	return actual.dato;
}
```

Otra versión sobrecargada de la operación `remove()` permite eliminar el elemento proporcionado; obtiene primero el índice con el método `indexOf()` para después eliminar por índice.

- `indexOf()` proporciona el índice, *base cero*, del dato proporcionado.

```
/**
 * Elimina el dato especificado.
 * @param dato – a eliminar.
 * @return - el índice del elemento eliminado o -1 si no existe.
 */
public int remove(Object dato) {
	// Obtiene el índice del elemento especificado.
	int actual = indexOf(dato);
	if (actual != -1) {
```

```
                remove(actual);         // Elimina por índice.
        }
        return actual;
}

/**
 * Busca el índice que corresponde a un elemento de la lista.
 * @param dato- el objeto elemento a buscar.
 */
public int indexOf(Object dato) {
        Nodo actual = primero;
        for (int i = 0; actual != null; i++) {
                if ((actual.dato != null && actual.dato.equals(dato))
                                || actual.dato == dato) {
                        return i;
                }
                actual = actual.siguiente;
        }
        return -1;
}
```

- `get()` permite obtener el elemento situado en la posición indicada por el índice. Para obtener el nodo donde está contenido el elemento se utiliza el método privado `obtenerNodo()`.
- `size()` permite obtener el número de elementos de la lista es resuelta con el método público

```
/**
 * @param indice – obtiene un elemento por su índice.
 * @return elemento contenido en el nodo indicado por el índice.
 * @exception IndexOutOfBoundsException - índice no está entre 0 y numElementos-1.
 */
public Object get(int indice) {
        // lanza excepción si el índice no es válido
         if (indice >= numElementos || indice < 0) {
                throw new IndexOutOfBoundsException("índice incorrecto: " + indice);
         }
        Nodo aux = obtenerNodo(indice);
        return aux.dato;
}

/**
 * @return el número de elementos de la lista
 */
```

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

Para una prueba de uso básico de la implementación realizada de la lista dinámica se puede utilizar la siguiente clase de prueba:

```
public class PruebaListaEnlazada {
	public static void main(String[] args) {
		ListaEnlazada listaCompra = new ListaEnlazada();
		listaCompra.add("Leche");
		listaCompra.add("Miel");
		listaCompra.add("Aceitunas");
		listaCompra.add("Cerveza");
		listaCompra.add("Café");
		System.out.println("Lista de la compra:");
		for (int i = 0; i < listaCompra.size(); i++) {
			System.out.println(listaCompra.get(i));
		}
		System.out.println("elementos en la lista: " + listaCompra.size());
		System.out.println("elementos 3 en la lista: " + listaCompra.get(3));
		System.out.println("posición del elemento Miel: " +
		listaCompra.indexOf("Miel"));
		System.out.println("eliminado: " + listaCompra.remove("Miel"));
	}
}
```

Salida:

```
Lista de la compra:
Leche
Miel
Aceitunas
Cerveza
Café
elementos en la lista: 5
elementos 3 en la lista: Cerveza
posición del elemento Miel: 1
eliminado: 1
```

Lista enlazada con acceso al último

Las *listas enlazadas con acceso directo al último* elemento, además del primero, permiten resolver el coste añadido de tener que recorrer toda la lista para llegar al punto de inserción al final. **Mantienen acceso directo al primero y al último elemento de la lista. Se caracterizan por la mejora del rendimiento en las operaciones de inserción y eliminación del principio y del**

final.

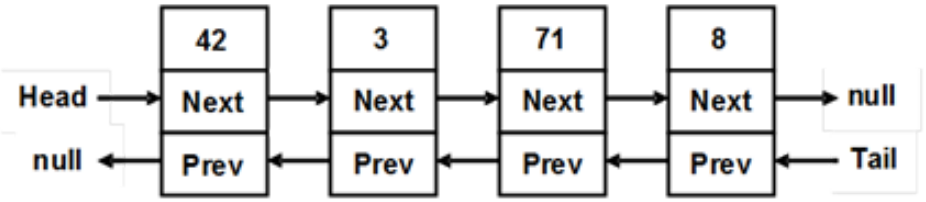
Para la implementación de una lista enlazada con acceso al último elemento se necesita añadir un nuevo atributo de la clase **Nodo** que representa el enlace o referencia al último elemento. Se implementa el método `add()` con índice para evidenciar la mejora que proporciona tener acceso al último elemento. Para la inserción en los puntos intermedios no hay cambio, se sigue utilizando el método auxiliar privado, `obtenerNodo()`, que tendría la misma implementación ya conocida.

```
/**
* Representa la implementación básica de una lista enlazada con acceso al último.
*/
public class ListaEnlazada2 {
	// Atributos
	private Nodo primero;
	private Nodo ultimo;
	private int numElementos;

	// Métodos
	/**
	 * Constructor que inicializa los atributos al valor por defecto.
	 */
	public ListaEnlazada2() {
		primero = null;
		ultimo = null;
		numElementos = 0;
	}

	 /**
	 * Añade un elemento al final de la lista.
	 * @param elem - el elemento a añadir. Admite que sea null.
	 *  @exception IndexOutOfBoundsException - índice no está entre 0 y numElementos-1
	 */
	public void add(Object dato) {
		// variables auxiliares
		Nodo nuevo = null;
		Nodo actual = null;
		Nodo anterior = null;

		// Lanza excepción si el índice no es válido.
		if (indice >= numElementos || indice < 0) {
			throw new IndexOutOfBoundsException("Índice incorrecto: " + indice);
		}

		// Si la lista está vacía el nuevo nodo es primero y último
		if (numElementos == 0) {
```

```
                    primero = new Nodo(dato);
                    ultimo = primero;
                }
                else {
                    // Enlaza el nuevo nodo al final; pasa a ser el último
                    nuevo = new Nodo(dato);
                    ultimo.siguiente = nuevo ;
                    ultimo = nuevo;
                    }
                    numElementos++;     // actualiza el número de elementos
                }
        }

 } // class
```

Lista enlazada doble

Las *listas doblemente enlazadas* están constituidas por nodos con dos referencias que permiten enlazar a los nodos `anterior` y `siguiente` de un elemento dado. **Permiten recorrer la lista hacia adelante y hacia atrás. Se caracterizan por la mejora del rendimiento y la simplificación en las operaciones de inserción y eliminación de elementos.**



Para la implementación de una lista enlazada doble se necesita una clase **Nodo** con `dobles enlaces` recursivos al `anterior` y `siguiente` **Nodo** de la lista.

- La clase **`ListaDoble`** utiliza normalmente enlaces al `primero` y `ultimo` de la serie de elementos enlazados y `numElementos` para el control del total de elementos almacenados.

```
/**
* Representa la implementación básica de una lista doble enlazada.
*/
public class ListaDoble {
       private Nodo primero;
       private Nodo ultimo;
       private int numElementos;

       /**
        * Constructor que inicializa los atributos al valor por defecto.
        */
```

```
	public ListaDoble() {
		primero = null;
		ultimo = null;
		numElementos = 0;
	}

 } // class

/**
 * Representa la estructura de un nodo para una lista enlazada doble.
 */
class Nodo {
	Object dato;
	Nodo anterior;
	Nodo siguiente;
	/**
	 * Constructor que inicializa atributos por defecto.
	 * @param dato - el elemento de información útil a almacenar.
	 */
	public Nodo(Object dato) {
		this.dato = dato;
		anterior = null;
		siguiente = null;
	}

} // class
```

Alguna de las operaciones importantes indicadas en el *TDA* son:

- `add()` con índice permite insertar un nuevo elemento a la lista en la posición indicada. Se utiliza un método auxiliar privado, `obtenerNodo()`, para obtener el nodo que hay en una determinada posición de índice.

```
/**
 * Añade un elemento al final de la lista
 * @param dato - el dato del elemento a añadir que no sea null
 */
 public void add(Object dato) {
	addUltimo(dato);
 }

/**
 * Inserta un elemento en la posición indicada de la lista.
 * @param indice - posición donde insertar el nuevo nodo.
 * @param dato - el dato del elemento a añadir. Admite que sea null.
```

```
 * @exception IndexOutOfBoundsException.
 * índice no está entre 0 y numElementos numElementos-1
 */
 public void add(int indice, Object dato) {
       // Lanza excepción si el índice no es válido.
       if (indice < 0 || indice >= numElementos) {
             throw new IndexOutOfBoundsException("Índice incorrecto: " +
 indice);
       }
       // Nuevo nodo al principio.
       if (indice == 0) {
             insertarPrimero(dato);
       }

       // Nuevo nodo en posiciones intermedias.
       if (indice > 0) {
             insertarIntermedio(indice, dato);
       }
 }

/**
 * Inserta un elemento en una posición intermedia de una lista doble enlazada.
 * @param el índice que ocupará el elemento nuevo.
 * @param el dato del elemento nuevo.
 */
 private void insertarIntermedio(int indice, Object dato) {
       assert indice >= 0 && indice < numElementos;
       Nodo nuevo = new Nodo(dato);
       Nodo actual = obtenerNodo(indice);      // Donde insertar.
       Nodo anterior = actual.anterior;        // Obtiene el anterior.
       actual.anterior = nuevo;                // Enlaza el nuevo nodo.
       anterior.siguiente = nuevo;
       nuevo.anterior = anterior;              // Ajusta enlaces.
       nuevo.siguiente = actual;
       numElementos++;                         // Actualiza tamaño.
 }

/**
 * Inserta un elemento al principio de una lista doble enlazada.
 * @param el dato del elemento nuevo.
 */
 private void insertarPrimero(Object dato) {
       Nodo nuevo = new Nodo(dato);

       // La lista está vacía; el nuevo nodo es primero y último.
```

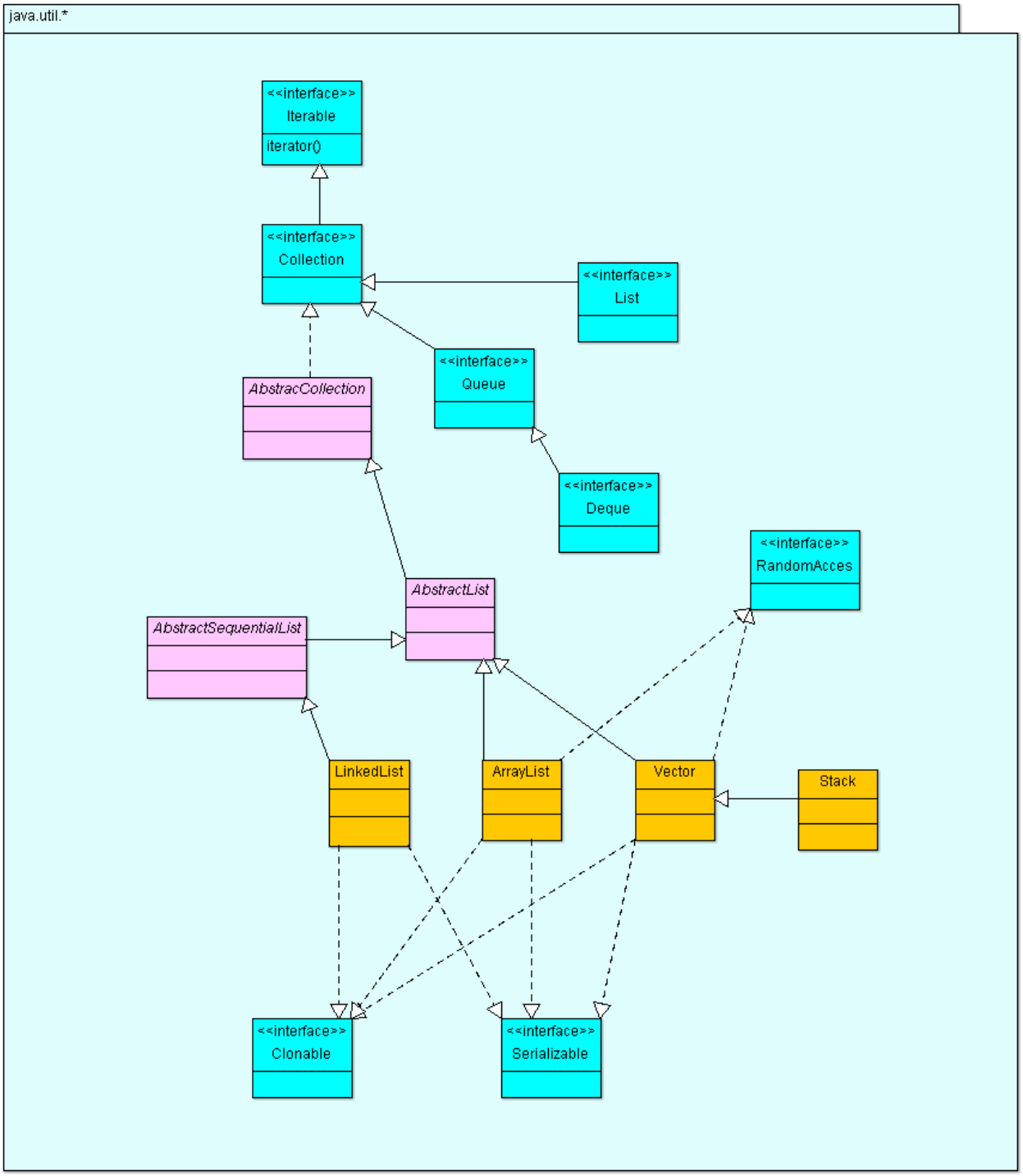
```
        if (numElementos == 0) {
                primero = nuevo;
                ultimo = nuevo;
        }
        // La lista no está vacía; el nuevo nodo pasa a ser el primero.
        else {
                Nodo actual = primero;           // Dónde insertar.
                actual.anterior = nuevo;         // Enlaza el nuevo nodo.
                nuevo.siguiente = actual;       // Ajusta enlace.
                primero = nuevo;                 // Actualiza el nuevo primero.
        }
        numElementos++;                          // Actualiza tamaño.
 }

 /**
  * Añade un elemento al final de una lista doble enlazada.
  * @param el dato del elemento nuevo.
  */
 private void addUltimo(Object dato) {
        Nodo nuevo = new Nodo(dato);
        // La lista está vacía; el nuevo nodo es último y primero.
        if (numElementos == 0) {
                ultimo = nuevo;
                primero = nuevo;
        }
        // La lista no está vacía; el nuevo nodo pasa a ser el último.
        else {
                Nodo actual = ultimo;            // Dónde insertar.
                actual.siguiente = nuevo;        // Enlaza el nuevo nodo.
                nuevo.anterior = actual;        // Ajusta enlace.
                ultimo = nuevo;                  // Actualiza el nuevo último.
                }
        numElementos++;                          // Actualiza tamaño.
 }

/**
 * Obtiene el nodo correspondiente al índice.
 * @param indice - posición del nodo a obtener.
 * @return el nodo que ocupa la posición indicada por el índice.
 */
private  Nodo obtenerNodo(int indice) {
        assert indice >= 0 && indice < numElementos;
        // Recorre la lista hasta llegar a la posición buscada.
        Nodo actual = primero;
        for (int i = 0; i < indice; i++) {
                actual = actual.siguiente;
        }
        return actual;
```

```
}
```

API Java Collection

La API de Colecciones (*Collections*) de Java proporciona un conjunto de clases e interfaces que facilitan la tarea de manejar contenedores de objetos. Las colecciones se caracterizan por su naturaleza dinámica.

La mayoría de las colecciones de Java se encuentran en el paquete `java.util.*`



Iterable

La interfaz **`Iterable`** da origen a la jerarquía de contenedores Java. Todas las clases que implementan la interfaz **`Iterable`** pueden ser usadas con el bucle `for-each`.

```
List lista = new ArrayList();
for (Object objeto : lista) {
       //hace algo con objeto
```

```
}
```

Collection

La interfaz **`Collection`** es origen o raíz de la jerarquía de contenedores Java; hereda de **`Iterable`**. Todos los subtipos de **`Collection`** deben implementar la interfaz **`Iterable`**.
La interfaz **`Collection`** da origen a las siguientes interfaces y clases abstractas relacionadas con las estructuras lineales:

- ❏ **`List`**
- ❏ **`Queue`**
- ❏ **`Deque`**
- ❏ **`AbtractCollection`**
- ❏ **`AbtractList`**
- ❏ **`AbstractSecuentialList`**

Java no proporciona una implementación de la interfaz **`Collection`**, así que para instanciar se deberá utilizar alguna de las clases derivadas finales. La interfaz **`Collection`** especifica una serie de métodos para establecer el comportamiento que compartirán cada uno de los subtipos de dicha interfaz. Este enfoque hace posible la independencia de las particularidades de cada variedad de colección; considerándose como una colección genérica.

```
/** clase que utiliza diferentes variedades de Collection con un método genérico
 */
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;

public class EjemploUsoColeccion {
	/**
	 * método genérico que realiza algún tratamiento sobre una Collection
	 * @param coleccion
	 */
	public static void hacerAlgo(Collection coleccion) {
		Iterator iterador = coleccion.iterator();
		while (it.hasNext()) {
			Object objeto = iterador.next();
			//hace algo con objeto ...
		}
	}

	/**
	 * llamadas al método hacerAlgo() con diferentes subtipos de Collection.
	 */
	public static void main(String[] args) {
```

```
            // No importa la variedad de Collection que se use.
            Stack pila = new Stack();
            List lista = new ArrayList();
            hacerAlgo(pila);
            hacerAlgo(lista);

            // Hay disponibles métodos comunes para añadir y eliminar elementos.
            String cad1 = "Esto es una prueba";
            String cad2 = "Esto es otra prueba";
            Collection coleccion = new LinkedList();
            boolean huboUnCambio = coleccion.add(cad1);
            System.out.println(coleccion.add(cad2));
            if (coleccion.remove(cad1)) {
                  System.out.printLn("Se eliminó...");
            }
            // Añade elementos de una colección a otra.
            lista.addAll(coleccion);

            // Elimina elementos de una colección contenidos en otra.
            lista.removeAll(coleccion);

            // Elimina elementos de una colección que no están en otra.
            lista.retainAll(coleccion);

            if (coleccion.contains("Esto es una prueba")) {
                  System.out.printLn("Se encontró...");
            }
            if (lista.containsAll(coleccion)) {
                  System.out.printLn("Se encontraron...");
            }
            System.out.printLn("Número de elementos: " + coleccion.size());
      }
}
```

- El método `add()` agrega el elemento dado a la colección y devuelve un valor **`true`** si la colección cambia como resultado de la ejecución del método. Un **`Set`** (*conjunto*) (se estudiará en un tema aparte), por ejemplo, podría no haber cambiado ya que, si el conjunto ya contenía el elemento, dicho elemento no se agrega de nuevo. Por otro lado, si el método `add()` es llamado con una **`List`** (lista) y la lista ya contenía el elemento, dicho elemento se agrega de nuevo; existirán dos elementos iguales dentro de la lista.
- El método `remove()` elimina el elemento dado y devuelve un valor **`true`** si el elemento existía en la colección. Si el elemento no existe, el método `remove()` devuelve **`false`**.
- El método `addAll()` añade todos los elementos encontrados en la colección pasada como argumento. El objeto `coleccion` no es agregado, sólamente sus elementos. Si en lugar de utilizar `addAll()` se hubiera utilizado `add()`, entonces el objeto `coleccion` se habría agregado y no sus elementos individualmente.
- El método `removeAll()` elimina todos los elementos encontrados en la colección pasada como argumento. Si la colección pasada como argumento incluye elementos que no se encuentran en la colección objetivo, simplemente se ignoran.
- El método `retainAll()` realiza la operación complementaria de `removeAll()`. En lugar de quitar los elementos pasados como argumento, retiene todos estos elementos y quita cualquier otro que no se encuentre en la colección proporcionada. Hay que tener en

cuenta que, si los elementos ya se encontraban en la colección objetivo, estos se mantienen. Cualquier elemento encontrado en la colección argumento que no se encuentre en la colección objetivo, no se agrega automáticamente, simplemente se ignora.

- ➢ El comportamiento específico de cada variedad de colección depende del subtipo de **`Collection`**, algunos subtipos permiten que el mismo elemento sea agregado varias veces (**`List`**), otros puede que no como en los **`Set`**.
- ➢ El método `contains()` devuelve **`true`** si la colección incluye el elemento y **`false`** si no es así.
- ➢ El método `containsAll()` devuelve **`true`** si la colección contiene todos los elementos de la colección argumento y **`false`** si no es así.
- ➢ El método `size()` devuelve el tamaño de una colección; indica el número de elementos que contiene.

Es posible utilizar **`Collection`** con tipos seguros -homogeneidad forzada con el tipo declarado- de la siguiente manera:

```
// Colección que verifica que todos los elementos sean String
Collection<String> coleccionCadenas = new ArrayList<String>();
```

El **`ArrayList`** `coleccionCadenas` únicamente puede contener instancias u objetos del tipo **`String`**. Si se trata de agregar cualquier otra cosa, o hacer un *cast* en los elementos de la colección, a cualquier otro tipo que no sea **`String`**, se producirá un error en tiempo de compilación.

List

La interfaz **`List`** representa una secuencia de objetos, a los que se puede acceder en un orden específico o mediante un índice. En una lista se pueden añadir elementos repetidos más de una vez. Al ser un subtipo de la interfaz **`Collection`**, todos los métodos de **`Collection`** también se encuentran disponibles en la interfaz **`List`**. Al ser **`List`** una interfaz, es necesario instanciar utilizando alguna de sus clases derivadas para poder utilizarla. Existen varias implementaciones (clases) de la interfaz **`List`** en la API de Java:

- ❏ **`ArrayList`**
- ❏ **`LinkedList`**
- ❏ **`Vector`**
- ❏ **`Stack`**

Para añadir elementos a un lista se utiliza el método `add()`. Este método es heredado y viene ya establecido por la interfaz **`Collection`**. El orden en el que se añaden los elementos a la lista es con el que quedan almacenados, de manera que se pueden acceder en el mismo orden. Para ésto, se puede utilizar el método `get()` o a través de un *iterador* obtenido con el método `iterator()`.

```
List listaA = new ArrayList();
listaA.add("elemento 0");
listaA.add("elemento 1");
```

```
listaA.add("elemento 2");

//accesso con índice.
String elemento0 = listaA.get(0);
String elemento1 = listaA.get(1);
String elemento3 = listaA.get(2);

//acceso mediante un iterador.
Iterator iterador = listaA.iterator();
while (iterador.hasNext()) {
	String elemento = (String) iterador.next();
}
//acceso mediante un bucle for-each.
for (Object o : listaA) {
	String elemento = (String) o;
}
```

ArrayList y Vector

Java proporciona dos estructuras de datos de tipo lista resueltas internamente con arrays que se redimensionan cuando es necesario; similares a **`ListaArray`** implementada en el ejemplo al principio del capítulo. Las dos clase de la API Java basadas en arrays estáticos extensibles son:

- **`ArrayList`** que tiene métodos convencionales.
- **`Vector`** que tiene todos sus métodos **`syncronized`** previstos para la programación concurrente.

Los principales métodos que tienen estas clases son:

`boolean add(Object)`	- Añade un elemento nuevo en el primer sitio libre.
`void add(int, Object)`	- Inserta un elemento en la posición indicada por índice.
`void clear()`	- Elimina todos los elementos de la lista.
`boolean contains(Object)`	- Comprueba si el elemento está incluido en la lista.
`Object get(int)`	- Obtiene el elemento de la posición indicada por índice.
`int indexOf(Object)`	- Devuelve la posición del elemento.
`boolean isEmpty()`	- Comprueba si la lista está vacía.
`boolean remove(Object)`	- Elimina el elemento correspondiente.
`Object remove(int)`	- Elimina el elemento de la posición indicada por índice.
`int size()`	- Devuelve el número de elementos almacenados.

Uno de los principales problemas que tienen que resolver estas implementaciones del *TDA lista* es el cambio del tamaño del array interno cuando se añaden, insertan y eliminan los elementos; se sigue una estrategia de sobredimensionado del array que actúa como un *buffer* extra, lo que permite añadir elementos sin cambiar el tamaño del array en cada operación.

ArrayList y **Vector** son listas para tipo de datos genéricos, por lo que pueden mantener todo tipo de elementos: números, cadenas y otros objetos; mezclados o no.

En el ejemplo se crea un **ArrayList** y un **Vector**. Después se añaden varios elementos de diferentes tipos: **String**, **int**, **double** y **Date**.

```
import java.util.ArrayList;
import java.util.Vector;
import java.util.Date;
public class EjListArray {
	public static void main(String[] args) {
		ArrayList list1 = new ArrayList();
		list1.add("Hola");
		list1.add(5);
		list1.add(3.14159);
		list1.add(new Date());
		System.out.println("ArrayList");
		for (int i = 0; i < list1.size(); i++) {
			Object valor = list1.get(i);
			System.out.format("Índice: %d; Valor: %s\n", i, valor);
		}
		Vector list2 = new Vector();
		list2.add("Hola");
		list2.add(5);
		list2.add(3.14159);
		list2.add(new Date());
		System.out.println("\nVector");
		for (int i = 0; i < list2.size(); i++) {
			Object valor = list2.get(i);
			System.out.format("Índice: %d; Valor: %s\n", i, valor);
		}
	}
}
```

Salida:

```
ArrayList
Índice: 0; Valor: Hola
Índice: 1; Valor: 5
Índice: 2; Valor: 3.14159
Índice: 3; Valor: Mon Jan 06 12:45:45 CET 2014

Vector
Índice: 0; Valor: Hola
```

```
Índice: 1; Valor: 5
Índice: 2; Valor: 3.14159
Índice: 3; Valor: Mon Jan 06 12:45:45 CET 2014
```

En el siguiente ejemplo se crea una lista de números y se procesan para averiguar su suma, hay que convertir cada elemento a tipo **`Integer`** debido a que todo lo que se almacena en el **`ArrayList`** es considerado en realidad un **`Object`**, sin más distinción.

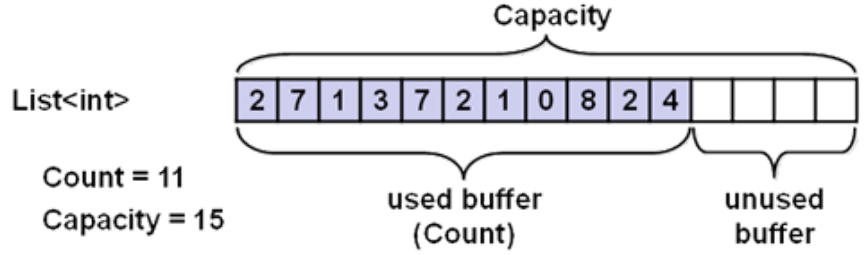
```
ArrayList lista = new ArrayList();
lista.add(2);
lista.add(3);
lista.add(4);
int suma = 0;
for (int i = 0; i < lista.size(); i++) {
    Integer valor = (Integer) lista.get(i);
    suma = suma + valor.intValue();
}
System.out.println("Suma: " + suma);
```

Es posible, y recomendable, utilizar las características de *verificación de tipos* que proporciona Java con los *tipos genéricos* para simplificar el tratamiento de los datos almacenados sin necesidad de conversiones de tipos:

```
ArrayList<Integer> lista = new ArrayList<Integer>();
lista.add(2);
lista.add(3);
lista.add(4);
int suma = 0;
for (int i = 0; i < lista.size(); i++) {
    suma = suma + lista.get(i);
}
System.out.println("Suma: " + suma);
```

Implementación basada en arrays

En los **`ArrayList`** y **`Vector`** los elementos se mantienen en un array estático que está parcialmente lleno. Los elementos libres permiten añadir elementos casi siempre sin necesidad de cambiar el tamaño del array. A veces, por supuesto, el array tiene que ser ampliado. Cada vez que se amplía el tamaño se duplica. El cambio de tamaño ocurre en raras ocasiones y puede ser ignorado en comparación con la frecuencia del resto de operaciones habituales.



La capacidad extra asignada inicialmente al array interno, que contiene los elementos de la clase

`ArrayList` y **`Vector`**, permite que sean estructuras extremadamente eficientes cuando es necesario *añadir al final y extraer elementos accediendo por índice*. Sin embargo resultan bastante lentas en la *inserción y eliminación* de elementos a menos que estos elementos se encuentren en la última posición. Se puede decir que estas implementaciones combinan los aspectos positivos de las listas y los arrays:

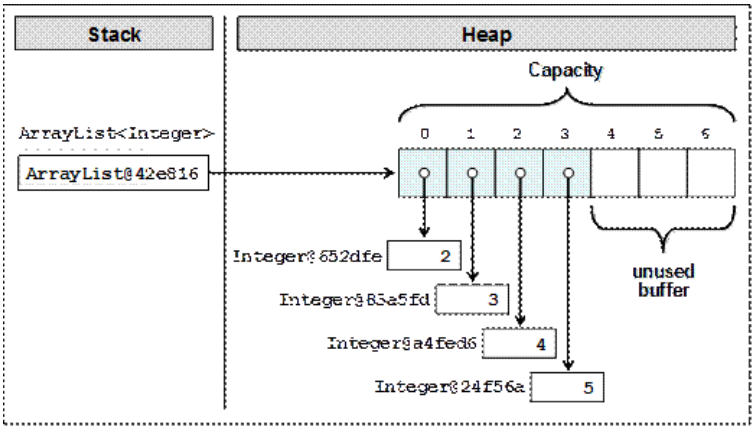
- Añaden eficazmente si es al final de la lista.
- Consiguen gestionar el tamaño dinámicamente de forma razonable.
- El acceso directo por el índice es constante y el más rápido de forma absoluta independientemente del tamaño..

ArrayList en la memoria

Todas las colecciones en Java son en realidad **`Collection`** de objetos y por lo tanto funcionan más lentamente que los arrays de tipos primitivos (por ejemplo, **`int`**`[]`). Al almacenar un nuevo elemento de tipo primitivo en un **`ArrayList`** se traslada a la memoria dinámica (*heap*). Al acceder a un elemento del **`ArrayList`**, se devuelve como un objeto y luego puede ser convertido en un tipo primitivo (por ejemplo, a un número).
Vamos el siguiente código:

```
ArrayList<Integer> lista = new ArrayList<Integer>();
lista.add(2);
lista.add(3);
lista.add(4);
lista.add(5);
```

En la memoria es tendría la siguiente representación:



Cada valor de la lista de números es un objeto de tipo **`Integer`**, que se encuentra en la memoria dinámica (*heap*). En el array no se guardan los valores de los elementos, sino sus direcciones (punteros). Por esta razón, el acceso a los elementos de la **`ArrayList<Integer>`** es más lento que el acceso a los elementos de **`int`**`[]` .

Cuándo utilizar ArrayList

Ya se ha dicho que estas implementaciones de listas se basan en arrays internos que duplican su tamaño cuando llegan a un nivel de llenado; esta característica hace que tengan los siguientes

aspectos buenos y malos:

- **El acceso por índices es muy rápido.** Se accede a la misma velocidad a cada elemento, independientemente del total. Los arrays tienen naturaleza de acceso directo por índice.
- **El acceso por el valor del elemento, tiene el coste adicional de las necesarias comparaciones; resultando finalmente un proceso lento**.
- **La inserción y extracción de elementos es una operación muy lenta**; sobre todo si no están al final de la lista; al tener que reorganizar todos los elementos.
- **A veces es necesario aumentar la capacidad del array interno, que en sí misma es una operación lenta, pero no es muy frecuente.**

Ejemplo: Obtención de los números primos en un intervalo

Para la búsqueda de los números primos en un cierto intervalo se puede tener en cuenta que:
Si un número no es primo tiene al menos un divisor en el intervalo [2 ... raíz cuadrada del número dado].
Para cada número que se valora, se busca un divisor entre 2 y raiz cuadrada del número. Si se encuentra un divisor, entonces el número no es primo y se puede continuar con el siguiente número del intervalo. Poco a poco, se irá completando la lista de los números primos.

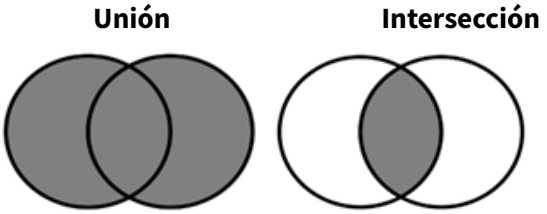
```
import java.util.ArrayList;
public class EjemploListaNumerosPrimos {
        public static ArrayList<Integer> numPrimos(int principio, int fin) {
            ArrayList<Integer> listaDePrimos = new ArrayList<Integer>();
            for (int num = principio; num <= fin; num++) {
                boolean esPrimo = true;
                for (int divisor = 2; divisor <= Math.sqrt(num); divisor++) {
                    if (num % divisor == 0) {
                        esPrimo = false;
                        break;
                    }
                }
                if (esPrimo)
                    listaDePrimos.add(num);
            }
            return listaDePrimos;
        }

        public static void main(String[] args) {
            ArrayList<Integer> listaNumPrimos = numPrimos(150, 300);
            for (int p : listaNumPrimos) {
                System.out.printf("%d ", p);
            }
            System.out.println();
        }
 }
```

Salida:

```
151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257
263 269 271 277 281 283 293
```

Ejemplo: Unión e intersección de dos listas

Dadas dos lista de números, obtener el resultado de la unión y la intersección de esas dos listas tomadas como conjuntos.



Se puede considerar que queremos obtener todos los elementos que están simultáneamente en ambas listas (intersección). Por otro lado también queremos obtener los elementos que están al menos en una de las dos listas sin repeticiones (unión). La lógica del programa sigue las definiciones de unión e intersección de conjuntos. Se utilizan las operaciones de búsqueda de un elemento en una lista.

```
import java.util.ArrayList;
public class EjListaIntersecUnion
	public static ArrayList<Integer> listaUnion(ArrayList<Integer> list1,
                                                ArrayList<Integer> list2) {
		ArrayList<Integer> listaUnion = new ArrayList<Integer>();
		listaUnion.addAll(list1);
		for (Integer item : list2) {
			if (!listaUnion.contains(item)) {
				listaUnion.add(item);
			}
		}
		return listaUnion;
	}

	public static ArrayList<Integer> listaIntersec(ArrayList<Integer> list1,
                                                   ArrayList<Integer> list2) {
		ArrayList<Integer> listaIntersec = new ArrayList<Integer>();
		for (Integer item : list1) {
			if (list2.contains(item)) {
				listaIntersec.add(item);
			}
		}
		return listaIntersec;
	}

	public static void printLista(ArrayList<Integer> lista) {
```

```
            System.out.print("{ ");
            for (Integer elem : lista) {
                  System.out.print(elem);
                  System.out.print(" ");
            }
            System.out.println("}");
      }

      public static void main(String[] args) {
            ArrayList<Integer> lista1 = new ArrayList<Integer>();
            lista1.add(1);
            lista1.add(2);
            lista1.add(3);
            lista1.add(4);
            lista1.add(5);
            System.out.print("Primera lista:\t\t");
            printLista(lista1);

            ArrayList<Integer> lista2 = new ArrayList<Integer>();
            lista2.add(2);
            lista2.add(4);
            lista2.add(6);
            System.out.print("Segunda lista:\t\t");
            printLista(lista2);

            ArrayList<Integer> listaUnion = unionListas(lista1, lista2);
            System.out.print("Lista unión:\t\t");
            printLista(listaUnion);

            ArrayList<Integer> listaIntersec = intersecListas(lista1, lista2);
            System.out.print("Lista intersección:\t");
            printLista(listaIntersec);
      }
}
```

Salida:

```
Primera lista:     { 1 2 3 4 5 }
Segunda lista:     { 2 4 6 }
Lista unión:       { 1 2 3 4 5 6 }
Lista intersección:      { 2 4 }
```

Conversiones entre List y array

La conversión de cualquier lista a un array se resuelve utilizando el método `toArray()` que implementan todas las clases de la interfaz **`List`**. Para la operación opuesta no está resuelto

directamente; se puede utilizar el constructor específico de la clase que interese para crear una lista del tamaño del array y añadir los elemento uno a uno. Hubiese sido una mejor implementación de la *API Collections* de Java, si se proporcionara directamente un constructor específico para este fin. La conversión *lista-array* tampoco resulta suficientemente limpia en su sintaxis. Se debe indicar, además, que se requiere la utilización de objetos y no funciona directamente con arrays de tipos primitivos.
Ejemplo:

```
import java.util.ArrayList;
import java.util.Arrays;
public class ConversionArray {
	public static void main(String[] args) {
		int[] array = new int[] {1, 2, 3};

		//Convierte el array en ArrayList.
		ArrayList lista = new ArrayList(array.length);
		for (int i = 0; i < array.length; i++) {
			lista.add(array[i]);
		}
		//Añade un nuevo elemento al ArrayList.
		lista.add(4);

		//Convierte el ArrayList en array.
		Integer[] numeros = (Integer[]) lista.toArray(new
		Integer[lista.size()]);

		//Muestra el array de Integer.
		System.out.println(Arrays.toString(numeros));
	}
}
```

Salida:

```
 [1, 2, 3, 4]
```

LinkedList

Java proporciona una estructura de datos de tipo lista completamente dinámica implementada con elementos doblemente enlazados internamente. Sus nodos o elementos contienen un *valor* y un *puntero al anterior* y *al siguiente* elemento en la lista. La clase **`LinkedList`** funciona de manera similar a la **`ListaEnlazada`** implementada como ejemplo en este tema.
Uno de los principales detalles que tiene que resolver, esta implementación del *TDA lista,* es el adecuado enlazado y desenlazado de nodos cuando se añaden, insertan y eliminan los elementos. Las Listas enlazadas, en general, son de naturaleza secuencial en su acceso; siendo este su principal punto débil.
`LinkedList` de Java es una lista genérica, por lo que puede mantener todo tipo de elementos:

números, cadenas y otros objetos; mezclados o no permitiendo la verificación de tipos seguros.

Cuándo utilizar LinkedList

En los ejemplos de implentación de **`listaArray`** y **`ListaEnlazada`** se comprueba que cada una de ellas tiene unas características específicas, mejor o peor según las diferentes operaciones. En las listas enlazadas, en general, hay que tener en cuenta que:

- La operación de añadir puede ser muy rápida si en la implementación se contempla la utilización de un *nodo final*.
- La inserción de un nuevo elemento en una posición aleatoria la lista es muy rápido si tenemos un puntero a esta posición, por ejemplo, si insertamos en el inicio de lista o al final la lista.
- La localización de elementos por índice o por valor en **`LinkedList`** es una operación lenta, ya que hay que requiere comparar todos los elementos consecutivamente desde el principio de la lista.
- La eliminación de elementos es lenta porque implica la búsqueda previa.
- La utilización de **`LinkedList`** es buena opción cuando hay que añadir / eliminar elementos en ambos extremos de la lista y cuando se requiere un acceso principalmente secuencial. Sin embargo, cuando se requiere acceso a los elementos por su índice o posición, entonces **`ArrayList`** puede ser una opción más apropiada.

Teniendo en cuenta el consumo de memoria, **`LinkedList`** generalmente toma más espacio, ya que tiene que mantener, en cada nodo, el dato del elemento y uno o varios punteros para cada elemento. **`ArrayList`** también usa espacio adicional de memoria para más elementos de lo que realmente usa.

Operaciones básicas con LinkedList

`LinkedList` tiene los mismos métodos que todas las **`List`** de Java. Los principales son:

`void add(Object)`	- Añade un elemento nuevo en el primer sitio libre.
`void add(int, Object)`	- Inserta un elemento en la posición indicada por índice.
`void clear()`	- Elimina todos los elementos de la lista.
`boolean contains(Object)`	- Comprueba si el elemento está incluido en la lista.
`Object get(int)`	- Obtiene el elemento de la posición indicada por índice.
`int indexOf(Object)`	- Devuelve la posición del elemento.
`boolean isEmpty()`	- Comprueba si la lista está vacía.
`boolean remove(Object)`	- Elimina el elemento correspondiente.
`Object remove(int)`	- Elimina el elemento de la posición indicada por índice.
`int size()`	- Devuelve el número de elementos almacenados.

`Los métodos de` **`LinkedList`** coinciden completamente con los de **`ArrayList`** por lo que son intercambiables.

Stack

Imaginemos un puerto marítimo de mercancías donde los contenedores se colocan uno encima del otro con una grúa para organizar el espacio; se puede poner un contenedor nuevo en la parte superior, así como quitar uno que esté en lo alto. Para mover el contenedor que está en la parte inferior, primero hay que mover los contenedores que tenga por encima.
La estructura de datos clásica de *Stack* (*pila*) funciona de manera que se pueden añadir elementos, uno a uno, en la parte superior y permite que se retire el último elemento que ha sido añadido, de uno en uno, pero no los anteriores (los que están debajo de él). En programación y sistemas operativos, las pilas son estructuras de datos de uso común. Se utilizan internamente para gestionar muchas características; por ejemplo para mantener las *variables del programa,* los *parámetros de los métodos,* las *llamadas a subprogramas* en la *pila de ejecución del programa*.
La pila es una estructura de datos, que implementa el comportamiento *LIFO (Last In - First Out)* (*último en entrar - primero en salir)*. Como con los contenedores de mercancías, se podrían añadir elementos y quitados sólo en la parte superior de la pila.

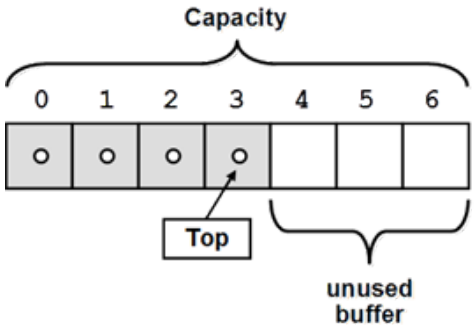
El *TDA* **`Stack`** (*pila*) se caracteriza por tres operaciones fundamentales:

- `push()` - Añade un elemento en la parte superior de la pila.
- `pop()` - Extrae el elemento existente en la parte superior de la pila.
- `peek()` - Lee el elemento de la parte superior de la pila sin eliminarlo.

Una pila pueden tener implementaciones dinámicas y estáticas.

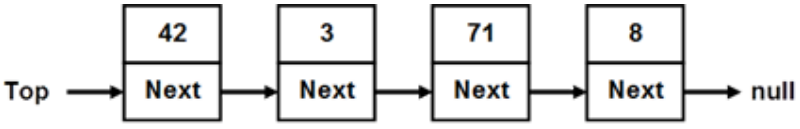
Pila implementada con un array

Como con las listas, se puede utilizar un array para mantener los elementos de la pila. Se puede mantener un índice o un puntero al elemento de la parte superior.
Por lo general, si el array interno se llena, hay que redimensionar su tamaño (por ejemplo duplicando su espacio en memoria).



Pila implementada con una lista enlazada

Para la implementación con una lista enlazada no requiere un *buffer interno*, no se llenan nunca y tiene prácticamente el mismo rendimiento, en las principales operaciones, que implementación estática:



Cuando la pila está vacía, el elemento superior tiene un valor **`null`**. Cuando se añade un elemento, se inserta al principio de la lista enlazada. La extracción consiste en eliminar el primer elemento de la lista.

Operaciones básicas de una pila

La implementación en Java de **`Stack`** tiene, entre otros, los siguientes métodos básicos son:

`void clear()`	- Elimina todos los elementos de la lista.
`boolean contains(Object)`	- Comprueba si el elemento está incluido en la pila.
`boolean isEmpty()`	- Comprueba si la pila está vacía.
`Object peek()`	- Obtiene el elemento de la cima de la pila sin quitarlo.
`Object pop()`	- Extrae el elemento situado en la cima de la pila.
`void push(Object)`	- Añade un elemento en la cima de la pila.
`int size()`	- Devuelve el número de elementos almacenados.
`Object[] toArray()`	- Devuelve un array con todos los elementos de la pila.

Ejemplo: Uso de una pila

Se van a añadir varios nombres de plantas a una pila, se convierte en array y después se muestran en la el array y la pila consola. A medida que se van extrayendo elementos de la pila:

```
import java.util.Stack;
public class EjemploUsoPila {
	public static void main(String[] args) {
		Stack<String> pila = new Stack<String>();
		pila.push("1. Camelia");
		pila.push("2. Azalea");
		pila.push("3. Jazmín");
		pila.push("4. Datura");

		//Convierte el Stack en array.
		Object[] plantas = pila.toArray();

		//Muestra la pila de nombres de plantas.
		System.out.println("Cima de la pila: " + pila.peek());
		while (pila.size() > 0) {
			String nombrePlanta = pila.pop();
			System.out.println(nombrePlanta);
		}
		//Muestra el array de nombres de plantas. La pila ya está vacía.
		System.out.println("Array:\n" + Arrays.toString(plantas));
	}
}
```

Salida:

```
Cima de la pila: 4. Datura
4. Datura
3. Jazmín
2. Azalea
1. Camelia
Array:
[1. Camelia, 2. Azalea, 3. Jazmín, 4. Datura]
```

Ejemplo: Chequeo de paréntesis en expresiones aritméticas

Se trata de un programa que comprueba si en una expresión aritmética se cumplen las reglas aritméticas de colocación de paréntesis. Debe comprobarse si el recuento de los paréntesis de apertura es igual al recuento de los paréntesis de cierre y que todos los paréntesis van bien emparejados.
Cuando se encuentra un paréntesis abierto, se añade un elemento a una pila. Cuando se encuentra con un paréntesis de cierre, se saca un elemento de la pila. Si la pila se vacía antes del final del proceso, en un momento en que debería haber elementos; los paréntesis están mal colocados. Si al final del proceso quedan elementos en la pila; también están mal colocados.

```
import java.util.Stack;
public class EjPilaParentesis {
	public static void main(String[] args) {
		String expresion = "1 + (3 + 2 - (2+3) * 4 - ((3+1)*(4-2)))";
		Stack<Integer> parentesis = new Stack<Integer>();
		boolean correcto = true;
		for (int i = 0; i < expresion.length(); i++) {
			char car = expresion.charAt(i);
			if (car == '(') {
				parentesis.push(i);
			}
			if (car == ')') {
				if (parentesis.isEmpty()) {
					correcto = false;
					break;
				}
				parentesis.pop();
			}
		}
		if (!parentesis.isEmpty()) {
			parentesisCorrectos = false;
		}

		if (parentesisCorrectos) }
			System.out.println("Paréntesis correctos...");
```

```
            {
            else {
                  System.out.println("Paréntesis incorrectos...");
            }
      }
}
```

Salida:

```
Paréntesis correctos...
```

Queue

Las *colas* son estructuras de datos lineales adecuadas para tareas y tratamientos de naturaleza secuencial en el que el orden y la prioridad son importantes, por ejemplo una cola de espera para la impresión de documentos, los procesos de espera para acceder a un recurso común, y otros. En las colas, de manera normal se añaden elementos en la parte del final y recuperan o extraen de la cabeza o frente.

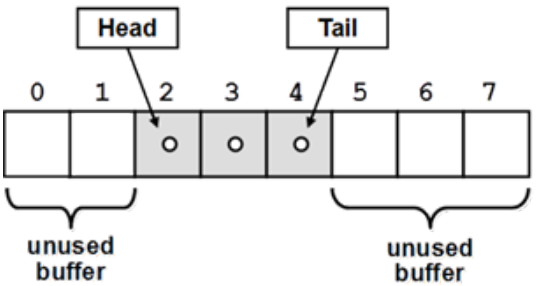
Por ejemplo, para comprar una entrada para un concierto; probablemente haya que hacer cola delante de la taquilla y habrá que esperar a que sean atendidos todos los clientes que tengamos delante.

En Java, una implementación basada en lista enlazada es la interfaz **Queue**. El TDA cola cumple el comportamiento *FIFO (first in - first out) (primero en entrar - primero en salir)*. Los elementos añadidos a una cola, lo hacen por final, y cuando se extraen los elementos, se hace por el principio de la cola (en el orden en que fueron añadidos). Así, la cola se comporta como una lista con dos extremos (cabeza y final).

Una cola pueden tener implementaciones dinámicas y estáticas.

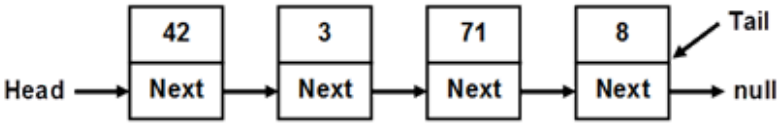
Cola implementada con un array

Como las colas son particularizaciones de las listas, se puede utilizar un array para mantener los elementos de la cola. En las colas se necesita mantener dos índices o punteros respectivamente a los elementos cabeza y final por donde la estructura soporta las dos operaciones fundamentales de extraer y añadir. Si la estructura llega a llenarse añadiendo elementos, se produce una ampliación dinámica del array interno.



Cola implementada con una lista enlazada

La implementación de colas con listas enlazada no requiere un *buffer interno o array*, no tiene límite teórico de elementos.



Operaciones básicas sobre colas

La implementación en Java de **Queue** tiene, entre otros, los siguientes métodos básicos son:

`void clear()`	- Elimina todos los elementos de la lista.
`boolean contains(Object)`	- Comprueba si el elemento está incluido en la pila.
`boolean isEmpty()`	- Comprueba si la cola está vacía
`Object peek()`	- Obtiene el elemento de la cabeza de la cola sin quitarlo.
`Object poll()`	- Extrae el elemento situado en la cabeza de la cola.
`void offer(Object)`	- Añade un elemento al final de la cola.
`int size()`	- Devuelve el número de elementos almacenados.

Ejemplo: Uso de una cola

Un ejemplo sencillo consiste en crear una cola de mensajes para después mostrarlos por la en el mismo orden en fueron insertados:

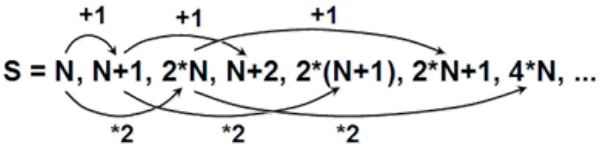
```
import java.util.Queue;
import java.util.LinkedList;
public class EjemploUsoCola {
	public static void main(String[] args) {
		Queue<String> mensajes = new LinkedList<String>();
		mensajes.offer("Mensaje Uno");
		mensajes.offer("Mensaje Dos");
		mensajes.offer("Mensaje Tres");
		mensajes.offer("Mensaje Cuatro");

		while (mensajes.size() > 0) {
			String msg = mensajes.poll();
			System.out.println(mensajes);
		}
	}
}
```

Salida:

```
Cabeza de la cola: Mensaje Uno
Mensaje Uno
Mensaje Dos
Mensaje Tres
Mensaje Cuatro
```

Ejemplo: Secuencia N, N+1, 2*N

Se trata de un proceso que genera una serie de valores utilizando una cola. Partiendo de un valor inicial N en la cola, lo saca y va alternativamente, poniendo al final, N+1 seguido de N*2



Con N = 3 y un límite de 16, la serie buscada se obtiene directamente en la salida de consola:

```
import java.util.Queue;
import java.util.LinkedList;

public class EjColaSerie {
	public static void main(String[] args) {
		int n = 3;
		int p = 16;
		int terminos = 0;
		int actual;
		Queue<Integer> queue = new LinkedList<Integer>();
		queue.offer(terminos);
		System.out.print("Serie: ");
		while (queue.size() > 0) {
			terminos++;
			actual = queue.poll();
			System.out.print(" " + actual);
			if (current == p) {
				System.out.println();
				System.out.println("Términos: " + terminos);
				break;
			}
			queue.offer(actual + 1);
			queue.offer(2 * actual);
		}
	}
}
```

Salida:

```
Serie:  3 4 6 5 8 7 12 6 10 9 16
Términos: 11
```

Hashtable

Es una estructura de datos que contiene entradas del tipo *clave-valor*, **`Hastable`** no admite valores **`null`** en ninguna de las dos partes del par *clave-valor*; si ocurre, se produce una **`NullPointerException`**. Otra característica de este tipo de colección es que es sincronizada, sólo un hilo de tarea puede acceder a la estructura a la vez, esto la hace un poco más lenta que **`HastMap`**.

```
Hashtable cityTable = new Hashtable();

cityTable.put(1, "Lahore");
cityTable.put(2, "Karachi");
cityTable.put(3, null); // NullPointerEcxeption en tiempo de ejecución
System.out.println(cityTable.get(1));
System.out.println(cityTable.get(2));
System.out.println(cityTable.get(3));
```

HashMap

El **`hashMap`** es parecida a **`Hastable`**, también almacena pares *clave-valor*.

Acepta **`null`** para ambos, o sea, se puede poner **`null`** en la clave o en el valor.
No es sincronizado, por lo que tiene mejor rendimiento.

```
HashMap productMap = new HashMap();
productMap.put(1, "Keys");
productMap.put(2, null);
productMap.put(null, "hola");

System.out.println(productMap.containsKey(null));
System.out.println(productMap.containsValue("Keys"));

Salida: true
        true
```

HashSet

Este tipo no soporta valores duplicados. en vez de tener un metodo put, provee un método add(). El Hashset es usado como una lista, no es de tipo lclave-valor. Preferencialmente este tipo de coleccion puede ser usado cuando quieres tener una lista única de objetos.

```
HashSet stateSet = new HashSet();
stateSet.add ("CA");
stateSet.add ("WI");
stateSet.add ("NY");
if (stateSet.contains("PB"))
     System.out.println("Already found");
else
    stateSet.add("PB");
```

Properties

La clase **`Properties`** permite manejar el conjunto de propiedades de un programa, siendo estas persistentes. Esta clase es un tipo especial de *tabla hash* con las siguientes características:

- La *clave* y el *valor* de la tabla son siempre **`Strings`**.
- La tabla puede ser guardada y recuperada de un *stream* con sólo una operación.
- Pueden definirse *valores por defecto* en una tabla secundaria.
- Debido a que **`Properties`** hereda de **`Hashtable`**, los métodos `put()` y `putAll()` están disponibles en los objetos **`Properties`**.

Métodos de la clase **`Properties`**:

- `getProperty(String key)`
  devuelve un **`String`** con el valor de la clave especificada.
- `setProperty(String key)`
  devuelve un **`String`** con el valor de la clave especificada.

Ficheros de Properties

Los ficheros de propiedades se guardan habitualmente con la extensión `.properties`, `.xml`, `.config` o `.txt`; es la mejor forma de reconocerlos.

Por ejemplo en el fichero `configuracion.properties` se puede tener la siguiente información:

```
# Nombre del servidor
servidor.nombre = PC9915
# Usuario para la conexión
servidor.usuario = Pepe
# Password para la conexión
servidor.password = poijasfr
```

Para la recuperación de la información se utiliza a la clase `java.util.Properties` que una vez instanciada, da acceso a las claves y valores del fichero de propiedades según el nombre de la clave, o bien se pueden recorrer todas si no se conoce el nombre:

```
import java.io.*;
import java.util.*;
public class EjemploProperties {
   public static void main(String[] args) throws IOException {
       Properties properties = new Properties();
       InputStream is = null;
       try {
           is = new FileInputStream("configuracion.properties");
           properties.load(is);
       }
       catch(IOException e) {
          e.printStackTrace();
       }

       // Acceso a las propiedades por su nombre
       System.out.println("Propiedades por nombre:");
       System.out.println("----------------------");
       System.out.println(properties.getProperty("servidor.nombre"));
       System.out.println(properties.getProperty("servidor.password"));
       System.out.println(properties.getProperty("servidor.usuario"));

       // Recorre todas; sin conocer los nombres de las propiedades
       System.out.println("Recorrer todas las propiedades:");
       System.out.println("------------------------------");
       for (Enumeration e = properties.keys(); e.hasMoreElements(); ) {
           // Obtiene elemento
           Object objeto = e.nextElement();
           System.out.println(objeto + ": "
                              +  properties.getProperty(objeto.toString()));
       }
   }
}
```

La salida del código anterior es la siguiente:

```
Propiedades por nombre:
-----------------------
PC9915
poijasfr
Pepe
Recorre todas las propiedades:
------------------------------
servidor.password:poijasfr
servidor.nombre:PC9915
servidor.usuario:Pepe
```

Puede ser que en ocasiones se quieran usar ciertas propiedades del sistema donde se ejecuta el programas para evaluar el rendimiento que éste pueda tener o simplemente para propósitos informativos. La clase `java.lang.System` dispone del método `getProperty()` para obtener la siguiente información:

```
java.version : Versión del entorno de desarrollo de Java (JRE).
java.vendor : Nombre de la distribución del JRE.
java.home: Directorio de instalación de la máquina virtual.
 os.name: Nombre del sistema operativo.
os.arch: Arquitectura del sistema operativo.
os.version: Versión del sistema operativo.
user.name : Nombre de usuario del sistema.
user.home : Ruta del directorio del usuario del sistema.
java.vm.name: Nombre de la máquina virtual instalada.
java.vm.version: Versión de la máquina virtual instalada.
```

Clase Collections

La clase **`Collections`** (no confundir con la interfaz **`Collection`**) dispone exclusivamente de métodos estáticos que operan sobre colecciones. Contiene algoritmos polimórficos que operan sobre colecciones, los cuales devuelven una nueva colección obtenida a partir la colección original.

Los métodos de esta clase producen **`NullPointerException`** si las colecciones proporcionadas a los métodos son **`null`**.

Los algoritmos “destructivos” contenidos en esta clase, es decir, los algoritmos que modifican la colección sobre la cual operan, producen **`UnsuportedOperationException`** si la colección original no admite la mutación apropiada de los primitivos.

Algunos ejemplos de métodos de la clase **`Collections`**:

```
void copy(List, List)
```

```
boolean disjoint(Collection, Collection)
Object max(Collection)
void reverse(List)
void sort(List)
```

Ejemplos de utilización de los métodos de la clase:

```
List lista = new LinkedList();

lista.add(“Juan”);
lista.add(“Luis”);
lista.add(“Adrian”);
lista.add(“Cheko”);
lista.add(“Rodolfo”);

Collections.sort(lista);

List lista2 = new LinkedList();

Collections.copy(lista, lista2);
```

Ejercicios

1. ➢ Copia y prueba de forma completa la implementación de lista basada en array que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales.***
   ➢ Haz una prueba completa de todos los métodos de la clase.
   ➢ Documenta con comentarios aclaratorios adicionales.

2. ➢ Copia y prueba de forma completa la implementación de lista enlazada que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales.***
   ➢ Haz una prueba completa de todos los métodos de la clase.
   ➢ Documenta con comentarios aclaratorios adicionales.

3. ➢ Escribe un método, que falta en la implementación de la lista enlazada que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales.*** El método se llamará `public void add(int indice, Object elemento)` sirve para poder insertar elementos en la estructura.
   ➢ Se recomienda echar un vistazo a la implementación hecha en la versión estática de lista.

4. ➢ Escribe un método, que falta en la implementación de la lista enlazada que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales.*** El método se llamará `public void removeAll(ListaEnlazada elementos)`. Sirve para poder eliminar de la lista todos los elementos que se le proporcionan en otra lista pasada como argumento.
   ➢ Se recomienda echar un vistazo a la implementación hecha en la versión de lista enlazada.

5. ➢ Escribe una nueva versión del programa ***Ejemplo: Unión e intersección de dos listas*** que aparece en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales*** para que haga exactamente lo mismo utilizando los métodos disponibles en la clase `ArrayList`:
      - `addAll()`
      - `removeAll()`
      - `retainAll()`
   ➢ Se deben escribir nuevas versiones de los correspondientes métodos especializados en obtener la unión y la intersección sin utilizar bucles.
   ➢ El método `retainAll()` deja en la lista que ejecuta el método, los elementos que aparecen en una lista pasadas como argumento; si algún elemento no está previamente lo ignora.

6.

- Escribe la implementación de un método que se llame `chequearParentesis()` que recibe una expresión aritmética como texto y devuelve **`true`** o **`false`**, según cumpla las reglas aritméticas de colocación de paréntesis. Debe comprobarse si el recuento de los paréntesis de apertura es igual al recuento de los paréntesis de cierre y que todos los paréntesis van bien emparejados.
- Se puede aprovechar el funcionamiento de una pila para registrar los paréntesis abiertos que se van encontrando. Cuando se encuentra con un paréntesis de cierre, se saca un elemento de la pila. Si la pila se vacía antes del final del proceso, en un momento en que debería haber elementos; los paréntesis están mal colocados. Si al final del proceso quedan elementos en la pila; también están mal colocados.

7.

- Escribe una implementación del *TDA pila* basada en un array.
- Puede servir, restringiendo operaciones, la implementación de lista basada en array que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales***.

8.

- Escribe una implementación del *TDA cola* basada en un array.
- Puede servir, restringiendo operaciones, la implementación de lista basada en array que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales***.

9.

- Escribe una implementación del *TDA cola* basada en una lista enlazada.
- Puede servir, restringiendo operaciones, la implementación de lista enlazada que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales***.

10.

- Escribe una implementación del *TDA pila* basada en una lista enlazada.
- Puede servir, restringiendo operaciones, la implementación de lista dinámica que se proporciona en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales***.

11.

- Escribe una versión de *lista doblemente enlazada* a partir de la implementación de la lista enlazada simple que aparece en el **Manual de Java**, en el tema: ***Estructuras dinámicas lineales,*** adaptando toda su funcionalidad básica sin cambiar el interfaz de métodos.
- Se debe probar la lista enlazada proporcionada y comprender su funcionamiento

antes de abordar la modificación y la versión pedida.

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en linea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- GUZMAN NATERAS, L. F. *Colecciones en Java*. [en linea] https://lc.fie.umich.mx/~lfguzman/manuales/notas_collections.pdf
- SÁNCHEZ, J. *Apuntes de fundamentos de programación*. [en línea] http://www.jorgesanchez.net