

# Tema 6. Colecciones de tamaño flexible: ArrayList. Otras colecciones: HashMap, HashSet

---

## Principales conceptos que se abordan en este tema

Aprenderemos en esta unidad a utilizar colecciones de objetos de tamaño flexible, en particular, la colección representada por la clase ArrayList.

No es esta la única clase proporcionada por Java para trabajar con colecciones dinámicas (de tamaño no definido). Entre otras están las clases HashMap, HashSet, LinkedList, Stack, Queue, ....

Ya que las colecciones solo permiten almacenar objetos tendremos que “convertir” tipos primitivos en objetos utilizando para ello las clases envolventes o clases “wrapper”.

Por último, sabemos leer la documentación de una clase, ahora escribiremos nuestra propia documentación.

## Número de sesiones

Se estiman un total de 44 horas

## 6.1 Agrupando objetos en colecciones de tamaño flexible: la clase ArrayList

---

Cuando escribimos programas necesitamos con frecuencia agrupar objetos en colecciones (un instituto que mantiene un registro de los alumnos matriculados, una agenda electrónica que permite guardar anotaciones, una librería que almacena una serie de libros, ...). Es habitual, además, que el nº de elementos en la colección varíe, que haya que añadir, borrar, ...

Las colecciones son una parte vital de cualquier lenguaje de programación (los arrays son realmente colecciones).

Una de las colecciones más usadas en Java es la que representa la clase **ArrayList**.

## 6.1.1 La clase ArrayList

---

La clase ArrayList es una clase que modela una colección que:

Permite almacenar un nº arbitrario de elementos, cada uno de ellos es un objeto (las colecciones flexibles, sean de la clase que sean, guardan objetos, no tipos primitivos, a menos que estos se “envuelvan” en una clase wrapper)

- crece / decrece automáticamente a medida que se añaden / borran elementos de la colección
- mantiene un contador privado que indica cuántos elementos tiene la colección
- está en el paquete java.util
- mantiene el orden de los elementos (tal como se insertaron) , es decir, los elementos de la colección se recuperan en el orden en que se introdujeron, por eso, se dice que una colección ArrayList es una secuencia ordenada de elementos

El siguiente ejemplo de una agenda personal nos permitirá ilustrar todos estos conceptos. La clase Agenda tiene las siguientes características:

- permite guardar notas (cada una de las anotaciones que hacemos en una agenda)
- no tiene límite en el nº de notas que puede almacenar
- permite mostrar notas individuales
- permite indicar cuántas notas hay almacenadas

```
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Una clase que mantiene una lista
 * con un nº arbitrario de notas.
 * Las notas se numeran de forma externa
 * por el usuario
 */

public class Agenda {

    private ArrayList<String> notas; // Almacén de notas

    /**
     * Constructor
     */

    public Agenda() {
        notas = new ArrayList<String>();
    }

    /**
     * Almacenar una nueva nota
     * @param nota La nota que se almacena
     */
    public void añadirNota(String nota) {
        notas.add(nota);
    }
}
```

```

/**
 * @return El nº de notas actualmente almacenadas
 */
public int numeroNotas() {
    return notas.size();
}

/**
 * Mostrar una nota
 * @param numeroNota El nº de nota a mostrar
 */
public void mostrarNota(int numeroNota) {
    if (numeroNota >= 0 && numeroNota < numeroNotas())
        System.out.println((numeroNota + 1) + " " + notas.get(numeroNota));
    else
        System.out.println("Índice incorrecto");
}

/**
 * Borrar una nota
 */
public void borrarNota(int numeroNota) {
    if (numeroNota >= 0 && numeroNota < numeroNotas())
        notas.remove(numeroNota);
    else
        System.out.println("Índice incorrecto, " + " el índice máximo es " +
(notas.size() - 1));
}

/**
 * Mostar todas las notas
 */
public void listarNotas() {
    int indice , cuantas;
    cuantas = notas.size();
    indice=0;
    while (indice < cuantas) {
        System.out.println((indice + 1) + " " + notas.get(indice));
        indice++;
    }
}

public void listarNotasConIterator() {
    Iterator<String> it;
    it = notas.iterator();
    while (it.hasNext())
        System.out.println(it.next());
}
}

```

Para utilizar la clase lo primero que hacemos es importarla: **import java.util.ArrayList;**

En principio, una colección `ArrayList` permite almacenar objetos de cualquier tipo (incluso los objetos que guarda una determinada colección pueden ser de tipos diferentes).

Lo habitual es utilizar colecciones que almacenan un determinado tipo de objetos. A partir de la versión 1.5 Java permite las colecciones genéricas en las que es posible especificar el tipo de objetos de una clase colección, en nuestro caso `ArrayList`.

**`private ArrayList<String> notas;`**

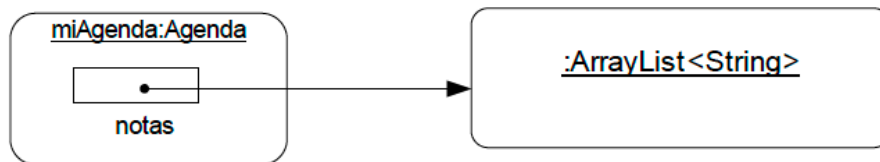
En la llamada al constructor de la colección se indica también este tipo:

```
public Agenda() {  
    notas = new ArrayList<String>();  
}
```

Si hacemos, bien desde BlueJ o desde alguna otra clase del proyecto:

```
miAgenda = new Agenda();
```

tendremos el siguiente diagrama de objetos:



## 6.1.2 Trabajando con ArrayList: add(), size(), get(), remove()

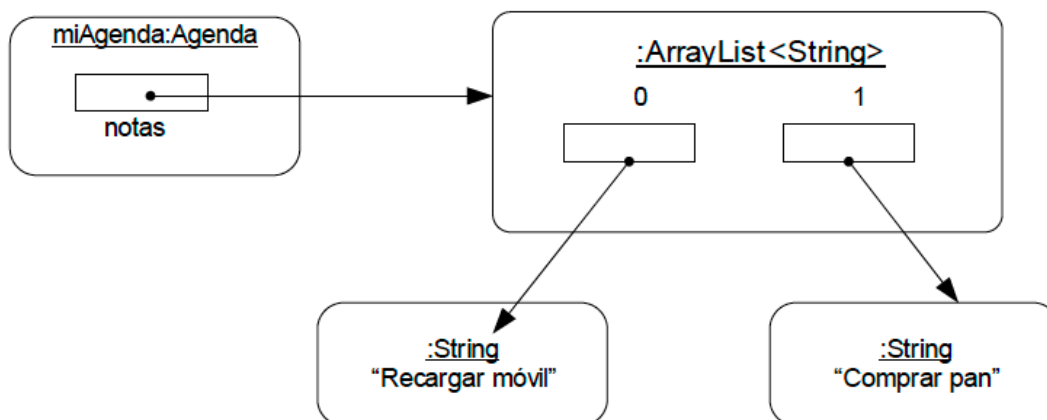
### Añadir un objeto a una colección ArrayList: método add()

El método add() añade un objeto a la colección.

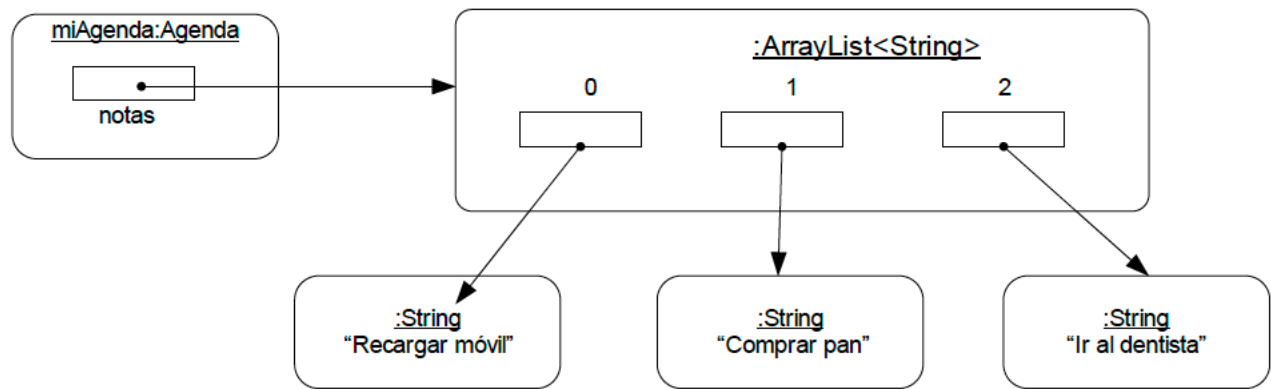
```
import java.util.ArrayList;
public class Agenda
{
    .....
    /**
     * Almacenar una nueva nota
     * @param nota La nota que se almacena
     */
    public void añadirNota(String nota)
    {
        notas.add(nota);
    }
    .....
}
```

objeto que se añade a la colección ArrayList

```
miAgenda = new Agenda();
miAgenda.añadirNota("Recargar móvil");
miAgenda.añadirNota("Comprar pan");
```



Si ahora hacemos, `miAgenda.añadirNota("Ir al dentista");`



Cada vez que se llama al método `add()` el objeto `ArrayList` (`notas`) incrementa automáticamente su tamaño.

### **Calculando el tamaño de una colección ArrayList: método `size()`**

El método `size()` devuelve el tamaño actual de la colección, el nº de objetos que almacena (en nuestro ejemplo el nº de strings).

```

/**
 * @return El nº de notas actualmente almacenadas
 */
public int numeroNotas(){
    return notas.size();
}

```

No es necesario que la clase `Agenda` tenga un atributo para guardar el nº de notas, con el método `size()` podemos conocerlo.

Los objetos almacenados en una colección tienen una numeración implícita (o posición) que comienza en 0. Esta posición es el **índice**. El primer elemento añadido posee índice 0, el 2º índice 1, el último elemento añadido tendrá como índice `size() - 1`.

### **Recuperando elementos de una colección: método `get()`**

El índice de una colección `ArrayList` nos permitirá recuperar directamente un elemento con el método `get()`. El método `get()` toma como parámetro un entero (una posición) y devuelve el objeto que está en esa posición.

```

/**
 * Mostrar una nota
 * @param numeroNota El nº de nota a mostrar
 */
public void mostrarNota(int numeroNota) {
    if (numeroNota >= 0 && numeroNota < numeroNotas())
        System.out.println((numeroNota + 1) + " " + notas.get(numeroNota));
    else
        System.out.println("Índice incorrecto");
}

```

Si hacemos, `miAgenda.mostraNota(1)`, el resultado mostrado en pantalla es “Comprar pan”.

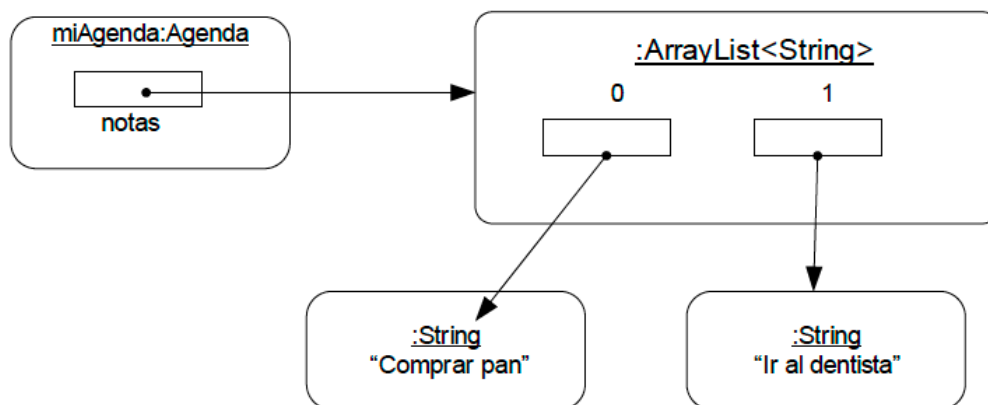
Puesto que el método `get()` no comprueba que el argumento que se le pasa es un valor correcto, el método `mostrarNota()` de la clase `Agenda` hace una comprobación de la posición. Si se intenta recuperar un objeto no existente con `get()` se genera un error (una excepción), `IndexOutOfBoundsException`.

### **Borrando un elemento de la colección: método `remove()`**

El método `remove()` toma como parámetro un valor entero que representa una posición en la colección (ha de estar entre 0 y `size() - 1`) y borra el elemento de esa posición.

```
/**
 * Borrar una nota
 * @param numeroNota el nº de nota a borrar
 */
public void borrarNota(int numeroNota) {
    if (numeroNota >= 0 && numeroNota < numeroNotas())
        notas.remove(numeroNota);
    else
        System.out.println("Índice incorrecto, " + " el índice máximo es " + (notas.size() - 1));
}
```

Si hacemos, `miAgenda.borrarNota(0)`, la colección queda:



Cuando se borra un elemento de la colección ésta decrece automáticamente, el índice cambia, todos los elementos a la derecha del elemento borrado se desplazan y pasan a tener un valor de índice decrementado en 1.



## 6.1.3 Procesando la colección completa

---

Para recorrer todos los elementos de una colección iteramos, con un bucle while por ejemplo, sobre ella.

```
/**
 * Mostar todas las notas
 */
public void listarNotas() {
    int indice , cuantas;
    cuantas = notas.size();
    indice = 0;
    while (indice < cuantas) {
        System.out.println((indice + 1) + " " + notas.get(indice));
        indice++;
    }
}
```

## 6.1.4 Otros métodos de la clase ArrayList.

---

La clase ArrayList incluye otros métodos adicionales para operar con ella, entre otros:

<b>public boolean isEmpty()</b>	devuelve <i>true</i> si la colección no contiene elementos
<b>public E set(int indice, E elemento)</b>	reemplaza el valor de la posición <i>indice</i> por <i>elemento</i> , devuelve el elemento que estaba previamente en esa posición
<b>public boolean contains(Object o)</b>	devuelve <i>true</i> si el objeto <i>o</i> está contenido en la colección
<b>public int indexOf(Object o)</b>	devuelve el índice de la primera ocurrencia del objeto <i>o</i> en la colección
<b>public void add(int indice, E elemento)</b>	inserta el elemento <i>E</i> en la posición especificada por <i>índice</i>
<b>public Iterator&lt;E&gt; iterator()</b>	devuelve un objeto iterador para recorrer los elementos de la colección en secuencia

## 6.1.5 El bucle for mejorado

---

El bucle for mejorado hace más fácil recorrer todos los elementos de una colección (o de un array) sin tener que hacer explícito el índice de cada elemento.

El formato general es:

```
for (tipo elemento: colección) {  
    .....  
}
```

donde:

- tipo – representa el tipo de los objetos almacenados en la colección (String, Cliente, ...). Si se trata de un array es el tipo de los elementos del array
- elemento – es el nombre de la variable asociada al bucle que tomará uno a uno los valores de la colección
- colección – es la colección (o el array) que va ser recorrida

El método `listarNotas()` de nuestra colección quedaría de la siguiente manera con el bucle for mejorado:

```
public void listarNotas() {  
    for (String nota: notas)  
        System.out.println(nota);  
}
```

El siguiente ejemplo muestra cómo utilizar la nueva sentencia for para recorrer todos los elementos de un array:

```
public double calcularMedia(int[] numeros) {  
    int suma = 0;  
    for (int num: numeros)  
        suma += num;  
    return suma / (double) numeros.length;  
}
```

Esta versión no puede utilizarse si la colección (o el array) va a modificarse (añadiendo o borrando elementos, por ej.).

## 6.1.6 Utilización de un iterador para recorrer una colección de un tipo específico

El recorrido completo de una colección es una operación tan común que la clase `ArrayList` proporciona un mecanismo especial para iterar sobre sus elementos (en realidad no es específico de esta clase sino de muchas otras colecciones).

El método `iterator()` de `ArrayList` devuelve un objeto `Iterator`. La clase `Iterator` está definida en el paquete `java.util` y, por tanto, hay que importarla si se quiere utilizar: **`import java.util.Iterator;`**

La **clase `Iterator`** proporciona, entre otros, los siguientes métodos para recorrer una colección:

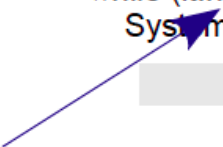
- `hasNext()` – devuelve `true` si queda aún elementos en la colección que se está recorriendo
- `next()` – devuelve el siguiente elemento de la colección
- `remove()` – borra el último elemento de la colección que fue obtenido con `next()`

El método `listarNotas()` utilizando un iterador quedaría así:

```
import java.util.ArrayList;
import java.util.Iterator;
.....
public void listarNotasConIterador()
{
    Iterator<String> it;

    it = notas.iterator(); //devuelve un objeto Iterator<String>
    while (it.hasNext())
        System.out.println(it.next());
}

```



se pregunta al iterador, no a la colección

### Ejercicio 6.1

Dada la siguiente colección:

```
private ArrayList<Estudiante> listaEstudiantes;
```

.....

```
listaEstudiantes = new ArrayList<Estudiante>();
```

Escribe el método **`public void borrarMenoresDeEdad()`** que borra de la colección los alumnos menores de edad.

La clase `Estudiante` proporciona el accesor **`getEdad()`**.

Construye el método con un *iterador* (no se puede utilizar for en ninguna de sus versiones - comprueba que al utilizar una instrucción for da error no de sintaxis sino de ejecución)

#### Solución

```
public void borrarMenoresDeEdad(){
    Iterator <Estudiante> it;
    it=listaEstudiantes.iterator();
    while(it.hasNext()){
        if(it.next().getEdad()<18)
            it.remove();
    }
}
```

## Ejercicio 6.2

Realiza el método del ejercicio anterior sin iteradores, con un bucle while.

#### Mostrar retroalimentación

```
public void borrarMenoresDeEdad(){
    int i=0;
    while(i==listaEstudiantes.size()){
        if(listaEstudiantes.get(i)<18)
            listaEstudiantes.remove(i);
    }
}
```

## 6.1.7 Acceso a través de índice o utilizando iteradores

Ya hemos vistos varias maneras de recorrer un ArrayList, llamando al método `get()` con un índice ( o `remove()` si queremos borrar) o utilizando un objeto `Iterator` o incluso con la sentencia `for` evitando el uso de índices.

Cualquiera de las soluciones es válida. Sin embargo hay que saber que Java proporciona muchas otras colecciones además de ArrayList. Con algunas colecciones el acceso a elementos individuales de la colección utilizando un índice es muy ineficiente. Sin embargo la utilización de un iterador está disponible en todas las colecciones del lenguaje.

### Ejercicio 6.3

Abre el proyecto Club desde BlueJ. ([Enlace para el proyecto Club](#)) Completa la clase Club. Esta clase se construye para almacenar en una colección ArrayList los miembros de un club. Cada miembro es un objeto de la clase Miembro.

- Define en la clase Club el atributo `miembros`
- En el constructor de la clase crea la colección. Compila el proyecto.
- Completa el método `numeroMiembros()`. Este método devuelve la cantidad de miembros que forman parte del club (es decir, el nº de elementos de la colección). Prueba el método desde BlueJ y comprueba que devuelve 0
- Analiza el código de la clase Miembro. Esta clase no necesita modificarse. Una instancia de esta clase representa a un miembro del club. Para añadir un nuevo miembro al club la clase Club incluye un método con la siguiente signatura: `public void añadir(Miembro miembro)` que añade un nuevo miembro al club. Prueba el método desde BlueJ (recuerda que para añadir un nuevo miembro has de crearlo antes en el Object Bench o bien al llamar al método `añadir()` y escribir como parámetro en el cuadro de diálogo, `new Miembro("Pepe", 11, 1997)`, creando así un objeto anónimo en la llamada).

```
club.añadir(new Miembro("Pepe", 11, 1997));
```

//es lo mismo que:

```
nuevoMiembro = new Miembro("Pepe", 11, 1997);
```

```
club.añadir(nuevoMiembro);
```

- Define un nuevo método en la clase, `public int incorporadoEnMes(int mes)`, que devuelve el nº de miembros que se incorporaron al club en el mes que se especifica como parámetro. Si el mes está fuera de rango escribe un mensaje de error y devuelve -1. Comenta el método y pruébalo.
- Añade el método: `public ArrayList<Miembro> borrar(int mes, int año)` El método borra de la colección los miembros que se incorporaron en el mes y año dados como parámetros. Los elementos borrados se devuelven en una nueva colección Si el mes no es correcto se emite un mensaje de error

y se devuelve una colección sin objetos. Construye el método utilizando un iterador

- Añade una nueva versión del método anterior que codifique el método con un while y utilizando índices para acceder a la colección. Incluye otro método idéntico pero con for. ¿Da los mismo hacerlo con while o con for?
- Escribe el método, **public void listarClub()** que visualiza todos los miembros del club. Utiliza un bucle for mejorado para recorrer la colección

Solución

[Enlace](#) de la solución del ejercicio

## Ejercicio 6.4

En este ejercicio trabajaremos con el proyecto Producto que incluye las siguientes clases ([Enlace al proyecto Producto](#)):

□ **clase Producto** – modela un producto vendido por una compañía. La clase registra el identificador, nombre y cantidad actual del producto en stock. El método `incrementarCantidad()` incrementa el nivel de stock del producto. El método `vender()` indica que el producto se vende y reduce la cantidad en 1. La clase no necesita modificarse.

□ **clase GestorStock** – almacena una serie de productos en una colección `ArrayList`.

Modifica esta clase completando los siguientes métodos:

- a. El método **`añadirProducto()`** añade un nuevo producto a la colección.
- b. **`escribirDetallesProductos()`** - muestra los detalles de cada producto. Haz tres versiones de este método: con iterador, con for.. each y con for e índices.
- c. **`localizarProducto()`** – recibe como parámetro un identificador de producto y devuelve el producto que coincide con ese identificador. Si no hay ninguno devuelve null. Prueba el método.
- d. **`cantidadEnStock()`** – dado un identificador de producto como argumento localiza el producto (utilizando para ello el método anterior) y devuelve la cantidad actual que hay en stock de ese producto. Si no hay ningún producto con ese identificador devuelve -1.
- e. Modifica el método **`añadirProducto()`** para que no sea posible añadir un producto con el mismo identificador que otro existente.
- f. **`localizarProducto()`** - dado un nombre de producto lo localiza en la colección. ¿Es posible tener dos métodos con el mismo nombre? ¿Cómo se dice qué son? ¿Qué es lo que les diferencia?
- g. **`recibirProducto()`** – dado un identificador y una cantidad se recibe una entrega de ese producto. Habrá que localizar el producto para poder añadir la cantidad recibida. Si no existe el producto se emite un mensaje de error.

h. *escribirMenorQue()* – escribe los detalles de productos cuyo nivel de stock está por debajo de un cierto nivel pasado como parámetro

Completa la clase StockDemo que se te proporciona.

Solución

[Enlace](#) con la solución



## 6.2 Las clases envolventes (Wrapper classes)

---

Las clases que representan colecciones como ArrayList permiten almacenar objetos. Pero Java distingue entre tipos primitivos y tipos referencia. ¿Qué podemos hacer si queremos guardar valores de tipo int, por ejemplo, en una colección? La solución a esto son las clases envolventes o wrapper classes.

## 6.2.1 Convirtiendo tipos primitivos en clases

---

Cada tipo simple (primitivo) en Java tiene su correspondiente clase envolvente que representa al mismo tipo pero en realidad es un objeto.

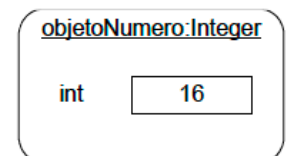
Tipo primitivo	Clase envolvente (wrapper)
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Cada clase tiene un constructor que toma como argumento un valor del tipo primitivo que representa. Los valores de los tipos primitivos se convierten en objetos llamado al constructor apropiado.

Los objetos de las clases wrapper son inmutables, es decir, no cambian una vez han sido creados.

**Ej.**     `int numero = 16;`  
         `Integer objetoNumero = new Integer(numero);`

`Character objetoCaracter = new Character('M');`



## 6.2.2 Convirtiendo clases envoltantes en tipos primitivos

---

Cada clase envoltante tiene un método que obtiene el valor original del tipo primitivo que almacena. El método es `intValue()` para la clase `Integer`, `floatValue()` para la clase `Float`, `charValue()` para la clase `Character`, ....

**Ej.**

```
Integer objetoNumero = new Integer(7);  
int numero = objetoNumero.intValue(); // devuelve 7
```

## 6.2.3 Colecciones y tipos primitivos

Para definir una colección de nos enteros hacemos:

```
ArrayList<Integer> listaEnteros = new ArrayList<Integer>();
```

Si añadimos un nº entero a la colección:

```
listaEnteros.add(new Integer(6));
```

Para recuperar un elemento de la colección, por ej, el primero:

```
int numero = listaEnteros.get(0).intValue();
```

Los valores de tipo primitivo pueden ser almacenados y recuperados de las colecciones sin especificar explícitamente el proceso de wrapping / unwrapping. Esto se permite gracias al mecanismo de autoboxing / unboxing que realiza directamente el compilador y no el programador.

Para añadir un nº entero a la colección del ejemplo anterior:

```
listaEnteros.add(6);
```

y para recuperar el valor entero:

```
int numero = listaEnteros.get(0);
```

El proceso de **autoboxing** (envolver – de tipo primitivo a objeto) se aplica cuando:

- se pasa un valor de tipo primitivo como parámetro a un método que espera un tipo objeto de una clase wrapper
- un valor de un tipo primitivo se asigna a una variable de tipo wrapper.

A la inversa, el proceso de **unboxing** (desenvolver – de tipo objeto a tipo primitivo) se aplica cuando:

- un valor de una clase wrapper se pasa como parámetro a un método que espera un tipo primitivo
- se almacena en una variable de tipo primitivo un valor de tipo wrapper

**Atención!** - El proceso de *unboxing* no funciona con `==` ni con `!=`. Si tenemos:

```
ArrayList<Integer> lista1, lista2; y hacemos
```

```
if (lista1.get(0) == lista2.get(0)) se comparan referencias, no se hace unboxing  
convirtiéndolo a int)
```

## Ejercicio 6.5

Completa la siguiente clase:

```
import java.util.ArrayList;

public class ColeccionEnteros {

    private ArrayList<Integer> miLista;

    public ColeccionEnteros() {

    }

    private void inicializarColeccion() {

    }

    public int sumar() {

    }

    public String toString() {

    }

}
```

| el constructor crea la colección y llama al método inicializarColeccion() que inicializa la lista con valores aleatorios comprendidos entre 1 y 20 (utiliza el método random() de la clase Math importándola). La inicialización termina cuando se genera un 0 ó cuando se hayan guardado 10 enteros.

| el método sumar() devuelve la suma de los valores enteros que almacena la colección. Haz dos versiones, con iterador y con for.. each

| toString() devuelve la representación textual de la colección de la forma: 4, 7, 5, 3, 2, .... (utiliza StringBuilder)

**Solución**

[Enlace](#) con la solución del ejercicio

## 6.2.4 Strings y clases envolventes

A veces interesa convertir un string que representa un valor numérico en un objeto wrapper.

Cada una de las clases envolventes proporcionan un método estático muy útil, `valueOf(String s)`, que crea un nuevo objeto inicializándolo al valor representado por el string.

**Ej.** `Double obj = Double.valueOf("12.4");` //Se crea el objeto obj con el valor 12.4  
`Integer objOtro = Integer.valueOf("13");` //Se crea el objeto objOtro con el valor 13

Las conversiones generan un error, la excepción `NumberFormatException`, si el string no representa un n° válido. Por ejemplo, `Integer.valueOf("5,45")` genera un error.

Si lo que queremos es convertir un string directamente en un tipo primitivo haremos:

```
int numero = Integer.valueOf("653").intValue();  
// o int numero = Integer.valueOf("653") haciendo unboxing
```

Pero las clases envolventes en Java proporcionan métodos estáticos que obtienen directamente el tipo primitivo que corresponde a un string: **`parseInt()`**, **`parseDouble()`**, .....

```
int numero = Integer.parseInt("653");
```

### Ejercicio 6.6

Sabemos que a partir de Java 1.5 se hacen las conversiones automáticamente sin hacer explícito el proceso de boxing (de un valor de tipo primitivo a objeto wrapper) o unboxing (convertir un objeto wrapper en un valor de tipo primitivo). Los siguientes ejemplos son, por tanto, correctos:

- a. `Integer unEntero = 2;`
- b. `Integer[] arrayEnteros = {1, 2, 3};`

Escribe las sentencias de los apartados a) y b) haciendo explícita la conversión.

**Solución**

```
Integer unEntero = new Integer(2);  
Integer[] arrayEnteros = new Integer{1,2,3};
```

### Ejercicio 6.7

Consulta en la API de Java la documentación de la clase `Character` e indica qué hacen los siguientes métodos estáticos poniendo un ejemplo:

<code>isLetter()</code>
<code>isDigit()</code>
<code>isLowerCase()</code>
<code>toUpperCase()</code>

## Ejercicio 6.8

Dados los siguientes ejemplos:

a) `String strNumero = JOptionPane.showInputDialog(null, "Teclea un número");`

`double numero =`

b) `TextField txtNumero = new TextField(25);`

`String strNumero = txtNumero.getText();`

`int numero =`

Completa las asignaciones indicadas en ambos casos para obtener los valores numéricos correspondientes a los strings tecleados.

### Solución

a) `String strNumero = JOptionPane.showInputDialog(null, "Teclea un número");`

`double numero = Double.parseDouble(strNumero);`

b) `TextField txtNumero = new TextField(25);`

`String strNumero = txtNumero.getText();`

`int numero = Integer.parseInt(strNumero);`

## 6.3 ArrayList versus arrays

---

Los objetos ArrayList crecen y decrecen dinámicamente, la colección no impone límites sobre cuántos objetos puede almacenar. Frente a los arrays, que sí imponen límites en cuanto a su tamaño, parece que las colecciones flexibles son mucho mejores y más útiles. ¿Por qué utilizar arrays si podemos utilizar un objeto ArrayList en su lugar?

Hay que apuntar, que por varias razones, las colecciones de tamaño fijo, los arrays, son más eficientes, las operaciones sobre los arrays son más rápidas (inserciones, borrados, ....). La medida de la eficiencia de las diferentes colecciones en Java (no solo ArrayList) es un tema importante a tener en cuenta. Sin embargo estas operaciones sobre una colección flexible son más fáciles de implementar.

Si de antemano se sabe el tamaño a ocupar por una colección de valores es mejor utilizar un array. Los arrays, además, permiten almacenar valores de tipo primitivo (las colecciones sólo objetos).

Un ArrayList se llama así por que, en principio, es una lista, la clase ArrayList hereda del interfaz List, y esta clase es una lista, es decir, una colección flexible. Pero hay muchos tipos de listas como veremos después, una de ellas es ArrayList que utiliza índices, pero hay otras como LinkedList que no los utiliza.

En realidad, un ArrayList es una lista que se implementa como un array. ¿Cómo puede una colección flexible implementarse como una colección de tamaño fijo? Cuando se crea un ArrayList realmente se crea un array de tamaño 10. Si nuestra colección ArrayList crece por encima de los 10 objetos el array se reemplaza por uno nuevo de mayor tamaño, si vuelve a llenarse se vuelve a reemplazar por otro mayor.



## 6.4 Otras colecciones

---

Analizaremos con detalle dos nuevas colecciones proporcionada por la API de Java, las que representan las clases **HashMap** y **HashSet**. Ambas son especializaciones de las clases (interfaces) *Map* y *Set* respectivamente.

## 6.4.1 La clase HashMap

---

Una **clase HashMap** es una especialización del interface Map (implementa el interface Map). La clase HashMap se implementa utilizando una tabla hash (no nos preocuparemos acerca de lo que es una tabla así).

Un **map** es una colección de objetos que almacena pares clave / valor. Como un ArrayList, un map guarda un nº flexible de entradas pero, a diferencia del ArrayList, cada entrada de un map no es un objeto sino un par de objetos. Este par consiste en un objeto clave y un objeto valor.

Un par es una entrada (Entry).

Las claves en una colección HashMap no están ordenadas (si queremos claves ordenadas habrá que utilizar la clase **TreeMap**).

En lugar de indicar un índice para acceder a un objeto (como en un array o en un ArrayList) en un map se indica el objeto clave (la clave) para obtener el valor (el valor).

Un ejemplo simple de lo que puede ser un map es un listín telefónico donde los nombres son las claves y los números de teléfono los valores. Si queremos localizar el teléfono de una persona buscamos su nombre y a partir de él obtenemos su nº de teléfono.

Si los pares clave/valor están ordenados de acuerdo a la clave es muy fácil localizar un par utilizando la clave (el problema es buscar el par a partir del valor). Un map es muy útil para búsquedas de una sola dirección, por ejemplo, dada una clave obtener el valor asociado a ella.

Las claves en un map han de ser únicas, no hay claves duplicadas.

En el listín telefónico a un nombre (la clave) le corresponde un nº de teléfono (el valor). Las búsquedas aquí se realizan por nombre.

**HashMap** es una colección genérica . De la misma manera que cuando creamos un ArrayList indicamos el tipo de objetos que va a contener la colección, en un HashMap hay que hacer lo mismo, solo que ahora daremos los nombres de dos clases: la clave y el valor.

Por ejemplo, si creamos el listín telefónico como un HashMap haremos (asumimos que está definida la clase Telefono para los nos de teléfono):

```
HashMap<String, Telefono> listin = new HashMap<String, Telefono>();
```

En el ejemplo listin es una colección Map, un HashMap, en la que todas las claves son objetos String y todos los valores son objetos de la clase Telefono.

Para utilizar un HashMap hay que importar la clase: *import java.util.HashMap;*

### Ejercicio 6.9

Queremos mantener una base de datos de estudiantes matriculados en un curso y para ello vamos a utilizar una colección HashMap en la que se asocian identificadores de estudiantes, representados por objetos Integer, con objetos Estudiante. Define la colección e instánciala.

### Solución

```
import java.util.HashMap;  
....  
HashMap<Integer, Estudiante> curso = new HashMap<Integer, Estudiante>(
```

## 6.4.1.1 Algunas operaciones habituales en un HashMap

---

<b>put(clave, valor)</b>	almacena una entrada <i>clave / valor</i> asociando el objeto <i>clave</i> con el objeto <i>valor</i> . Si la clave existía se sobrescribe. Devuelve el valor previo (o <i>null</i> ) asociado a esa clave
<b>get(clave)</b>	devuelve el objeto <i>valor</i> correspondiente a clave o <i>null</i> si no hay correspondencia (la clave no existe). La clave proporcionada puede ser cualquier tipo referencia.
<b>remove(clave)</b>	borra de la colección el objeto <i>valor</i> asociado a la clave dada. Devuelve el valor previo (o <i>null</i> ) asociado a esa clave. La clave proporcionada puede ser cualquier tipo referencia. (**)
<b>values()</b>	devuelve una colección ( <i>Collection</i> ) conteniendo todos los <i>valores</i> del <i>map</i> . Es una <i>vista</i> de los <i>valores</i> del <i>map</i> .
<b>keySet()</b>	devuelve un conjunto ( <i>Set</i> ) que es el conjunto de todas las <i>claves</i> del <i>map</i> . Es una <i>vista</i> sobre las claves, no es un conjunto independiente.
<b>entrySet()</b>	devuelve un conjunto ( <i>Set</i> ) con todas las entradas (de tipo <b>Map.Entry</b> ) del <i>map</i> . Es una <i>vista</i> sobre las entradas, no es un conjunto independiente.
<b>size()</b>	devuelve el n° de entradas en el <i>map</i>
<b>isEmpty()</b>	devuelve <i>true</i> si no hay entradas en el <i>map</i>
<b>containsKey(clave)</b>	devuelve <i>true</i> si el <i>map</i> contiene la clave indicada. La clave puede ser de cualquier tipo referencia. (**)

(\*\*) Las clases String, Integer, Character, ... (clases definidas en la API) tienen redefinido el método equals() y hashCode(). El método containsKey() funciona bien con estos tipos. Con clases propias (clase Telefono, clase Estudiante, ...) hay que redefinir equals() y hashCode().

## 6.4.1.2 Iterando sobre un HashMap

### **Map.Entry** (interface)

Cada elemento en un map es un par clave / valor, un objeto de tipo Map.Entry. El conjunto de entradas del map se obtiene con el método entrySet(). Para iterar sobre un map hay que hacerlo sobre este conjunto de entradas, ya que el map, la clase HashMap no incluye ningún iterador (ningún método iterator()).

Si asumimos que entrada es un objeto de tipo Map.Entry entonces:

- **entrada.getKey()** devuelve la clave de la entrada
- **entrada.getValue()** devuelve el valor de la entrada

Si queremos recorrer un map de forma completa podremos hacer:

a) Obtener el conjunto de claves con el método keySet() y luego iterar sobre este conjunto

```
HashMap<String, Telefono> listin = new HashMap<String, Telefono>();
.....
Set<String> conjuntoClaves = listin.keySet();
Iterator<String> it = conjuntoClaves.iterator();
while (it.hasNext()) {
    .....
}
```

b) Obtener el conjunto de entradas con el método entrySet() y luego iterar sobre este conjunto

```
HashMap<String, Telefono> listin = new HashMap<String, Telefono>();
.....
Set<Map.Entry<String, Telefono>> entradas = listin.entrySet();
Iterator<Map.Entry<String, Telefono>> it = entradas.iterator();
while (it.hasNext()) {
    Map.Entry<String, Telefono> entrada = it.next();
    System.out.println(entrada.getKey() + " - " + entrada.getValue());
}
```

## Ejercicio 6.10

Sea la siguiente definición de una colección en una clase Banco:

```
HashMap<String, Cuenta> cuentasBancarias = new HashMap<String,
Cuenta>(); // la clave es el identificador de un cliente, el valor la cuenta
asociada
```

La clase Cuenta mantiene el nº y balance de cada cuenta.

Escribe los siguientes métodos:

- a. **public void addCuenta(String nombre, int numCuenta, int balance)** – añade una nueva cuenta a la colección
- b. **public Cuenta getCuenta(String nombre)** - devuelve la cuenta del cliente cuyo identificador se proporciona como parámetro
- c. **public void listarClientes()** – lista en pantalla los identificadores de los clientes del banco (para ello obtiene un conjunto – set - de todas las claves en el map)

#### Solución

```
public void addCuenta(String nombre, int numCuenta, int balance){
    cuentasBancarias.put(nombre, new Cuenta(numCuenta, balance));
}

public Cuenta getCuenta (String nombre){
    return cuentasBancarias.get(nombre);
}

public void listarClientes() {
    Set<Map.Entry<String, Cuenta>> clientes =
    cuentasBancarias.entrySet();
    Iterator<Map.Entry<String, Cuenta>> it = clientes.iterator();
    while(it.hasNext()) {
        Map.Entry<String, Cuenta> cliente = it.next();
        System.out.println(cliente.getKey() + " - " + cliente.getValue());
    }
}
```

## Ejercicio 6.11

Crea una clase ListinTelefonico que guarda un listín de teléfonos implementado como un HashMap. El map asocia nombres con números de teléfono, ambos de tipo String. Añade a esta clase dos métodos:

- a. **public void introducirNumero(String nombre, String numero)** – añade una nueva entrada al listín de teléfonos
- b. **public String buscarNumero(String nombre)** – devuelve el nº de teléfono correspondiente al nombre proporcionado como parámetro
- c. **public void escribirListin()** – escribe el listín (utiliza un bucle for genérico)

#### Solución

```
import java.util.HashMap;
```

```

public class ListinTelefonico {
    private HasMap<String, String> listin;
    public ListinTelefonico ( ) {
        listin = new HashMap<String, String> ( );
    }
    public void introducirNumero (String nombre, String numero){
        listin.put(nombre, numero);
    }
    public String buscarNumero (String nombre){
        return listin.get(nombre);
    }
    public void escribirListin( ){
        for (ListinTelefonico agenda: listin){
            System.out.println(agenda.getKey(  ) + " - " +
            agenda.getValue( ));
        }
    }
}

```

## Ejercicio 6.12

Responde a las siguientes cuestiones consultando la documentación de Java:

1. ¿Qué ocurre si se añade una entrada a un map con una clave que ya existe? Pruébalo con el ejemplo anterior
2. ¿Y si se añade una entrada con un valor que ya existe?
3. ¿Cómo sabemos si una determinada clave está contenida en el map?
4. ¿Qué ocurre si se intenta buscar un valor y la clave no existe en el map? – pruébalo en el ejemplo anterior.
5. ¿Cómo podemos saber cuántas entradas hay en un map?

### Solución

1. ¿Qué ocurre si se añade una entrada a un map con una clave que ya existe? Pruébalo con el ejemplo anterior
  - Se **sobreescribe**
2. ¿Y si se añade una entrada con un valor que ya existe?
  - Se **modifica** el valor
3. ¿Cómo sabemos si una determinada clave está contenida en el map?
  - Usando el método **containsKey(clave)**

4. ¿Qué ocurre si se intenta buscar un valor y la clave no existe en el map? – pruébalo en el ejemplo anterior.
  - Devuelve **null**
5. ¿Cómo podemos saber cuántas entradas hay en un map?
  - Con el método **size()**

## Ejercicio 6.13

Indica si es correcto o no y por qué.

- a) `HashMap<String, String> m = new HashMap<String, String>();`  
`m.add("hola");`
- b) `HashMap<String, String> m = new HashMap<String, String>();`  
`ArrayList<String> lista = new ArrayList<String>();`  
`m.put("hola", lista);`
- c) `HashMap<String, String> m = new HashMap<String, String>();`  
`String s = "27";`  
`m.put("hola", s);`  
`m.put("adios", s);`
- d) `HashMap<String, String> m = new HashMap<String, String>();`  
`m.put("hola", "27");`  
`m.put("hola", "327");`

### Solución

Indica si es correcto o no y por qué.

- a) **INCORRECTO**. El método para añadir elementos al map es put, no add
- b) **INCORRECTO**. El tipo de datos del HashMap es String, no Array<String>
- c) **CORRECTO**.
- d) **CORRECTO**.



## 6.4.2 La clase HashSet

---

Un **HashSet** es una colección de tamaño flexible que almacena objetos. En realidad es una especialización del interface set. La particularidad del set es que almacena una colección no ordenada de objetos únicos, es decir, no se permiten los duplicados (un objeto no puede aparecer dos veces en un set) y tampoco se puede ordenar. Un HashSet se implementa utilizando una tabla hash, de ahí su nombre.

Puesto que no mantiene ningún orden específico (a diferencia de ArrayList) no se indica ningún índice para acceder a sus elementos. No hay método get() para acceder a un set.

**HashSet** es una colección genérica . De la misma manera que cuando creamos un ArrayList o un HashMap indicamos el tipo de objetos que va a contener la colección, en un HashSet hay que hacer lo mismo. (Un set modela la abstracción “conjunto matemático”)

Por ejemplo, si creamos un conjunto de nombres como un HashSet:

```
HashSet<String> conjuntoNombres = new HashSet<String>();
```

La sentencia anterior crea un nuevo objeto HashSet que va a contener objetos String.

Para utilizar un HashSet hay que importar la clase: *import java.util.HashSet;*

## 6.4.2.1 Algunas operaciones habituales en un HashSet

---

<b>add(objeto)</b>	añade el <i>objeto</i> al <i>set</i>
<b>remove(objeto)</b>	borra <i>objeto</i> del conjunto(**)
<b>contains(objeto)</b>	devuelve <i>true</i> si <i>objeto</i> está dentro del <i>set</i> (**)
<b>size()</b>	devuelve el nº de objetos en el <i>set</i>
<b>isEmpty()</b>	devuelve <i>true</i> si el conjunto está vacío
<b>iterator()</b>	devuelve un objeto <i>Iterator</i> sobre los elementos del conjunto

(\*\*) Las clases `String`, `Integer`, `Character`, ... (clases definidas en la API) tienen redefinido el método `equals()` y `hashCode()`. El método `contains()` funciona bien con estos tipos. Con clases propias (clase `Telefono`, clase `Estudiante`, ...) hay que redefinir `equals()` y `hashCode()`.

**Ej:**

```
import java.util.HashSet;
.....
HashSet<String> conjuntoNombres = new HashSet<String>();
conjuntoNombres.add("Hola");
conjuntoNombres.add("Adiós");
```

## 6.4.2.2 Recorriendo un HashSet

Al recorrer un set se recuperan los elementos aleatoriamente, no necesariamente en el orden en que fueron añadidos.

Para recorrer un set podemos utilizar:

a) un **bucle for genérico (un for..each)** –

```
public void escribirNombres(HashSet<String> conjuntoNombres) {  
    for (String n: conjuntoNombres)  
        System.out.println(n);  
}
```

b) un **iterador** - tal como vimos al estudiar la colección ArrayList un iterador es un objeto que nos permite recorrer una colección. Para obtener el iterador se aplica el método `iterator()` sobre la colección, en nuestro caso el set, y a través de los métodos `next()`, `hasNext()`, `remove()` del objeto iterador se recorre la colección conjunto.

```
public void escribirNombres(HashSet<String> conjuntoNombres) {  
    Iterator<String> iter = conjuntoNombres.iterator();  
    while (iter.hasNext()) {  
        String n = iter.next();  
        System.out.println(n);  
    }  
}
```

### Ejercicio 6.14

- Define una clase **ConjuntoEnteros** que incluye el atributo enteros que es un HashSet de objetos Integer
  - Incluye el constructor, *public ConjuntoEnteros(int tamaño)*, tamaño es la cantidad de números a añadir. El constructor crea el conjunto y añade los números enteros del 1 al valor de tamaño
  - Implementa el método: *public HashSet<Integer> getPares( )* que devuelve el conjunto de números pares. Utiliza un iterador para recorrer el set.
  - Escribe el método *public void addNumero(int n)* que añade un nuevo n° entero al conjunto
  - Define el método *public void printConjunto( )* que escribe los valores del conjunto utilizando un for genérico
  - Define el método *public void borrarPares( )* que borra los números pares del conjunto utilizando un iterador

Solución

```
public class ConjuntoEnteros{
```

```

private HashSet<Integer> enteros;
public ConjuntoEnteros (int tamaño) {
    enteros = new HashSet<Integer>( );
    for(int i=1; i<=tamaño; i++){
        addNumero(i);
    }
}

public HashSet<Integer> getPares( ) {
    HashSet<Integer> pares = new HashSet<Integer>( );
    Iterator <Integer> it = enteros.iterator( );
    while(it.hasNext( )){
        int num=it.next( );
        if(num % 2 == 0){
            pares.add(num);
        }
    }
    return pares;
}

public void addNumero (int n) {
    enteros.add(n);
}

public void printConjunto( ) {
    for(Integer pares: enteros){
        System.out.println(pares);
    }
}

public void borrarPares( ) {
    Iterator <Integer> it = enteros.iterator( );
    while(it.hasNext( )){
        int num=it.next( );
        if(num % 2 == 0){
            it.remove( );
        }
    }
}
}

```

## 6.5 Escribiendo la documentación de las clases

---

Escribir buena documentación para las clases e interfaces de un proyecto es una tarea muy importante que complementa a la escritura de código fuente de calidad. Si no se proporciona una buena documentación puede ser muy duro para otros programadores la comprensión del código (cientos de líneas de código) de nuestras clases.

La documentación de una clase debería ser lo suficientemente detallada para que otros programadores puedan utilizarla sin necesidad de leer la implementación. Por eso es de particular importancia la documentación de los elementos públicos de la clase (su interfaz).

De la misma manera que utilizamos las clases de la librería de Java (la API) una vez hemos leído la documentación que nos proporciona el interfaz de las mismas (y no su implementación) así deberían poder utilizarse nuestras clases.

## 6.5.1 Comentarios javadoc

---

Java utiliza una herramienta, javadoc, que genera a partir de los comentarios incluidos en el código fuente (los denominados comentarios javadoc) el interfaz de una clase. La documentación generada por esta herramienta está en formato HTML (la API de Java está documentada utilizando esta herramienta).

Los comentarios javadoc se escriben con un símbolo de comentario especial:

```
/**  
 * esto es un comentario javadoc  
 */
```

Los comentarios se abren con `/**` y se cierran con `*/`. Han de empezar con `/**` para que sean reconocidos como javadoc.

Entre estos símbolos de comentario se incluye:

- una descripción general de la clase o método que se está documentando
- una serie de etiquetas (tag) que comienzan por `@`. Pueden aparecer o no y siguen a la descripción anterior

`@author`      nombre del autor

`@version`    nº de versión y fecha

`@param`      nombre del parámetro y descripción

`@return`     descripción del valor de retorno

`@throws`    tipo de excepción lanzada y circunstancias (se pueden usar indistintamente)

`@exception`

## 6.5.2 ¿Qué documentamos?

---

Se documentan los elementos públicos de una clase o interfaz.

La documentación de una clase debería incluir al menos:

- nombre de la clase
- propósito general de la clase y características (cómo utilizar la clase)
- nº de versión
- autor/es
- documentación para cada constructor y método.

Por cada constructor y método:

- nombre del método
- tipo de valor de retorno
- nombres de los parámetros y tipos
- descripción del propósito del método
- descripción de cada parámetro
- descripción del valor de retorno

Además cada proyecto debería incluir un comentario general de proyecto, a menudo contenido en un fichero de texto (Readme.txt de BlueJ).

## 6.5.3 Utilizando javadoc desde BlueJ

---

El entorno BlueJ utiliza javadoc para crear la documentación de las clases. En la ventana principal a través de Herramientas / Generar documentación (Tools / Project Documentation) se genera la documentación de todo el proyecto en formato HTML y se visualiza en el navegador.

En BlueJ podemos cambiar la vista del código fuente de una clase a su documentación cambiando la opción Implementación a vista Interfaz.

### Ejercicio 6.15

Documenta adecuadamente las clases del proyecto realizado en el ejercicio 6.3 (Club) y genera la documentación desde BlueJ. Observa la nueva carpeta creada dentro del proyecto con la documentación generada (la carpeta doc).

Solución

[Enlace](#) con el proyecto solucionado



## 6.5.4 Utilizando javadoc desde la línea de comandos

### Ejercicio 6.16

Generaremos ahora la documentación del proyecto Producto del ejercicio 6.4. Primero documentaremos adecuadamente las dos clases del proyecto, Producto.java y GestorStock.java.

Las clases a documentar están dentro de una carpeta, que es el nombre del proyecto, (en el ejemplo, la carpeta es 6.4 Productos). Este además es el directorio activo. Dentro de esta carpeta crearemos otra (podemos hacerlo desde el DOS o desde el explorador de Windows) que llamaremos doc (o también docs) y en la que dejaremos toda la documentación generada: D:.....\6.4 Productos\docs\

Abriremos una ventana de comandos DOS y llamaremos a la herramienta Java javadoc :

**D:\...>javadoc -d .\docs Producto.java**

El parámetro **-d** se usa para indicar a javadoc dónde colocar la documentación generada, en nuestro caso, en .\docs

Se puede especificar más de una clase en la línea de comandos.

Por defecto javadoc no utiliza la información de las etiquetas **@author** y **@versión**. Para que lo haga pondremos:

**D:\...>javadoc -author -version -d .\docs Producto.java GestorStock.java**

javadoc genera un fichero HTML por cada fichero .java y paquete que encuentra.

Completa el ejercicio.

**Solución**

[Enlace](#) a la solución del ejercicio

## 6.6 Creando paquetes

---

Los paquetes se utilizan para agrupar clases relacionadas (recordemos que toda la API de Java está organizada en paquetes y para utilizar una determinada clase en nuestro proyecto importamos la clase del paquete con la sentencia `import`).

Todas las clases pertenecen a un paquete. Si no se especifica ninguno explícitamente las clases pertenecen al paquete por defecto (**default package**). Cada una de las clases que nosotros hemos creado hasta ahora pertenecen al paquete por defecto.

Es posible indicar explícitamente que una clase va a pertenecer a un determinado paquete.

Hay varias razones para utilizar paquetes:

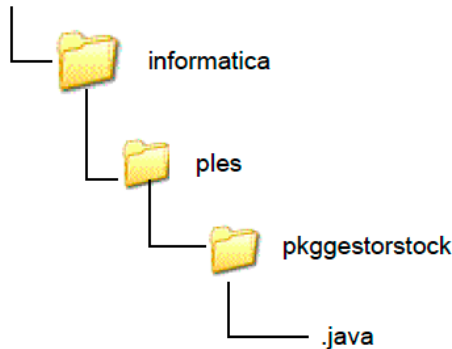
- para localizar las clases – las clases con funciones similares se sitúan en el mismo paquete y así se localizan más fácilmente
- para evitar conflictos de nombre – cuando se desarrollan clases que van a ser compartidas por varios programadores, colocar las clases en paquetes evita el conflicto a la hora de nombrarlas (ej, dos clases con el mismo nombre pero en distinto paquete)
- para distribuir el software más fácilmente – habitualmente en ficheros jar
- para proteger las clases – los paquetes proporcionan protección (por defecto, si no se indica nada los miembros de una clase son accesibles únicamente dentro del paquete al que pertenecen, se dice que tienen visibilidad de paquete, `package` )

Los paquetes son jerárquicos. Dentro de un paquete podemos tener otro. *java.lang.Math* expresa que la clase **Math** está en el paquete **lang** que a su vez es un paquete del paquete `java`.

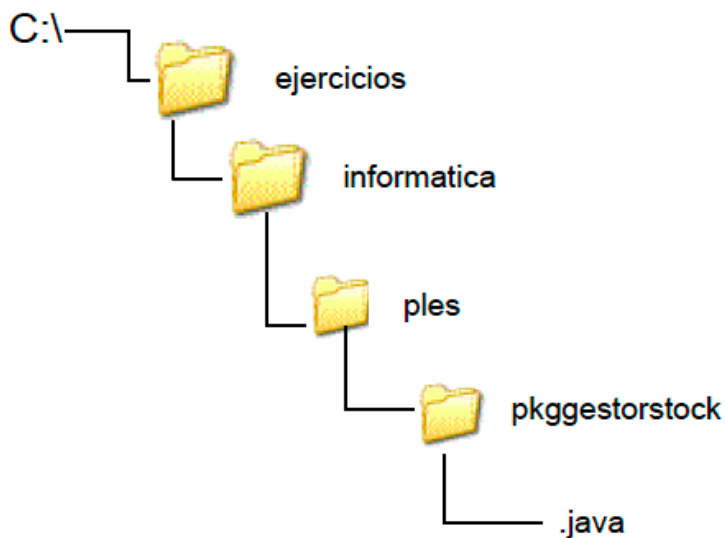
Por convención, los nombres de los paquetes se escriben en **minúsculas**.

## 6.6.1 Paquetes y directorios

Un paquete se corresponde con una carpeta (un directorio) que contiene una serie de clases. Si creamos un paquete con el nombre, `informatica.ples.pkggestorstock` la estructura de directorios que deberíamos crear sería:



Para que Java sepa dónde está nuestro paquete en el sistema de ficheros hay que modificar la variable `classpath` para que apunte al directorio en el cual reside nuestro paquete.



En este caso, `classpath = .; C:\ejercicios`

Además del directorio actual se indica la ruta base (el directorio raíz) a partir del cual se buscarán los paquetes y ficheros `.class`.

**classpath** – es una variable de entorno del sistema que define las rutas en las que el intérprete java busca los paquetes (como el path para los ficheros `.bat` y `.exe`).

El directorio actual (`.`) normalmente está incluido en el `classpath`. Ahí es donde la JVM busca los paquetes por defecto y las clases compiladas.

Si se importan paquetes que no están en el `.` hay que indicar al compilador y a la JVM dónde buscar.

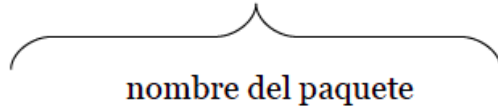
## 6.6.2 Colocando clases en paquetes

---

Cada clase pertenece a un paquete. La clase se añade a un paquete cuando se compila. Si no se indica nada la clase pertenece al paquete por defecto, que es el directorio actual (si trabajamos con BlueJ es la carpeta que contiene el proyecto).

Si queremos colocar una o varias clases en un paquete específico hay que añadir la sentencia,

```
package informatica.ples.pkggestorstock;
```



nombre del paquete

como primera sentencia de la clase.

## 6.6.3 Utilizando las clases del paquete

---

Para utilizar las clases de un paquete las importamos con la sentencia import:

```
import informatica.ples.pkggestorstock.*; // con * se importan todas las clases del  
paquete  
import informatica.ples.pkggestorstock.Producto; //se importa una sola clase
```

Se puede utilizar una clase de un paquete sin incluir la sentencia import si al usar la clase incluimos su nombre calificado completo.

```
informatica.ples.pkggestorstock.Producto          p          =          new  
informatica.ples.pkggestorstock.Producto();  
java.util.Scanner teclado = new java.util.Scanner(System.in);
```

## 6.6.4 Paquetes y BlueJ

---

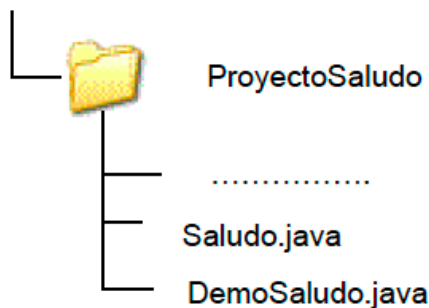
En primer lugar indicaremos cuál es la notación que utiliza UML para denotar un paquete.



En BlueJ un proyecto puede incluir uno o varios paquetes y dentro de cada paquete se incluye una o más clases. Hasta ahora cada proyecto incluía una o varias clases pero no hemos especificado ningún paquete dentro de él, por tanto, se ha utilizado el paquete por defecto (un paquete sin nombre).

Podemos crear un paquete en BlueJ partiendo de un proyecto que ya incluye una serie de clases de la forma siguiente:

❑ Imaginemos un proyecto formado por las clases Saludo.java y DemoSaludo.java. La estructura de directorios que habrá creado BlueJ para este proyecto es:



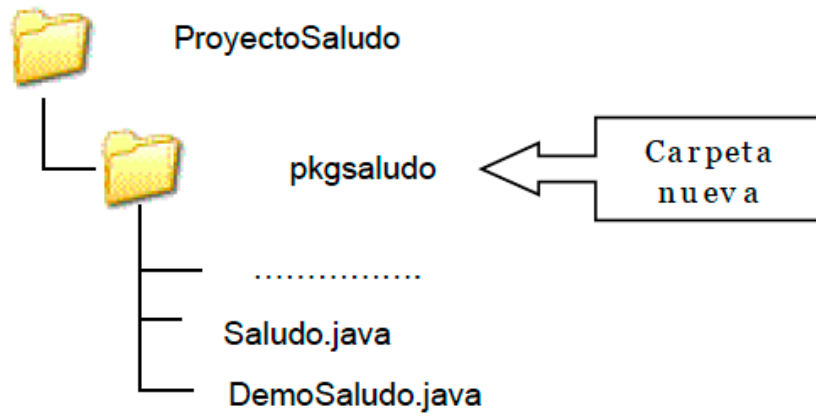
Abrimos el proyecto desde BlueJ

❑ Creamos un paquete de nombre pkgsaludo haciendo Edición / Nuevo paquete, (Edit / New Package)

❑ Se añaden las clases al paquete incluyendo en cada clase la sentencia: package pkgsaludo; al principio de la clase

❑ Compilamos cada clase y BlueJ nos preguntará si queremos mover la clase al nuevo paquete. Diremos que sí.

❑ Físicamente se habrá creado una carpeta nueva dentro del directorio ProyectoSaludo



□ Para que desde otro proyecto se pueda utilizar este paquete :

- crearemos un fichero jar (saludo.jar) y lo guardaremos en C:\BlueJ 2.2.0\lib\userlib
- indicaremos en: Herramientas / Preferencias / Librerías / Agregar la ruta al paquete. (Tools / Preferences / Libraries / Add )

## 6.6.5 Creando ficheros jar (java archive executable)

---

Es más fácil distribuir una aplicación Java si la aplicación completa está almacenada en un único fichero. Java permite crear ficheros de formato **.jar**, así se comprimen todos los ficheros de una aplicación en uno único (similar a los ficheros .zip). Las clases de la API están almacenadas así.

Además podemos hacer que un fichero .jar sea ejecutable especificando la clase de la aplicación que contiene el método main(). Para eso se incluye un fichero de texto dentro del fichero .jar, el fichero manifiesto. Este fichero contiene una línea de texto con la siguiente directiva:

*Main-Class: nombre de la clase que incluye el método main*

Un jar con fichero manifiesto que incluye la clase con main() se puede utilizar, además de cómo ejecutable, como librería. En este último caso podemos añadir las clases que contiene a nuestro proyecto.



## 6.6.5.1 Ficheros jar y BlueJ

---

Desde BlueJ podemos crear un fichero jar seleccionado Proyecto / Exportar (Project / Create Jar File). BlueJ nos pide a través de un cuadro de diálogo la clase que contiene el método main().

Una vez creado, el fichero jar puede ser ejecutado haciendo doble clic sobre él. El ordenador que ejecute un .jar ha de tener instalado el JDK o JRE y asociado con ficheros .jar .

### Ejercicio 6.17

Crea un proyecto BlueJ que contenga las clases siguientes:

- **clase Saludo** – emite saludos personalizados. Tiene un atributo nombre de tipo String que es el nombre de la persona a saludar. Incluye el constructor con un parámetro de tipo String y un método saludar() que emite un saludo a la persona cuyo nombre está en el atributo (visualiza el mensaje utilizando algún método de la clase JOptionPane)
- **clase AppSaludo** - es la clase que contiene el main(). Crea un objeto Saludo y llama al método saludar().
- Documenta adecuadamente las dos clases y genera la **documentación** del proyecto desde BlueJ
- Incluye las dos clases en un paquete **pkgsaludos**
- Crea el fichero **saludos.jar** y ejecútalo (doble clic).

Solución

[Enlace](#) con la solución del ejercicio

## 6.6.5.2 Creando ficheros jar desde la línea de comandos

Java incluye la herramienta **jar** en el JDK para crear archivos .jar.

Si queremos crear desde la línea de comandos, por ejemplo, un fichero jar ejecutable de nombre saludos que incluya las clases *Saludo.java* y *AppSaludo.java* (que es la que incluye el *main()*) haremos:

```
....> java Saludo.java AppSaludo.java
```

Compilamos las clases de la aplicación

```
....> jar -cmf manifiesto.txt saludos.jar *.class
```

fichero manifiesto      fichero jar comprimido      clases compiladas

- **-c**: indica crear nuevo archivo jar
- **-f**: se especificará nombre del archivo jar
- **-m**: se especificará nombre del fichero manifiesto
- **-v**: si se añade esta opción se muestra en la ventana de comandos información del proceso

Previamente hemos creado con un editor de texto el fichero manifiesto.txt con el siguiente texto (puede tener cualquier otro nombre):

*Main-Class: AppSaludo*

Este fichero debe incluir una línea en blanco al final.

Una vez creado saludos.jar desde la línea de comandos se puede ejecutar:

```
....> java -jar saludos.jar
```

### Ejercicio 6.18

Añade al proyecto creado en el ejercicio 6.5 (Colección de enteros) una clase Test que incluya el *main()* y las sentencias adecuadas para probar la clase ColeccionEnteros. Crea con el comando jar el ejecutable de la aplicación.

Solución

[Enlace](#) con la solución del proyecto



## 6.6.6 La API de Java

---

La biblioteca de clases standard de Java incluye cientos de clases organizadas en multitud de paquetes, entre otros:

<u>Paquete</u>	<u>Descripción</u>
<b>java.lang</b>	clases centrales de la plataforma (cadenas, números,...). No es necesario incluir la sentencia import cuando se utilizan clases de este paquete
<b>java.util</b>	utilidades varias:fechas, generadores de nos aleatorios, listas dinámicas, ...
<b>java.io</b>	clases para realizar operaciones de E/S (entrada / salida, uso de ficheros)
<b>java.awt</b>	clases para crear interfaces gráficas y dibujar figuras e imágenes
<b>javax.swing</b>	clases para crear GUI de usuario con componentes 100% escritos en Java (no nativos como los de awt)
<b>java.applet</b>	clase para crear applets
<b>java.net</b>	clases que permiten implementar aplicaciones distribuidas

## 6.7 Más sobre colecciones

---

Las colecciones son una parte vital de cualquier lenguaje de programación. En Java, además de las colecciones `ArrayList`, `HashMap`, `HashSet`, hay muchas otras útiles para otros propósitos.

El conjunto de colecciones se conoce como Java Collections Framework.

Un framework es un conjunto de interfaces y clases (abstractas y concretas) que permiten organizar y manipular los datos eficientemente:

- **interfaces** (tipos) – describen tipos lógicos (representan funcionalidad, son las abstracciones de la implementación)
- **clases** (implementaciones) – implementan el tipo lógico (incluyen además en algunos casos métodos estáticos – algoritmos – de utilidad para trabajar con diferentes tipos de colecciones). Las clases abstractas proporcionan una implementación parcial y las clases concretas implementan los interfaces con estructuras de datos concretas organizadas en una jerarquía.

Las ventajas de disponer de un framework son:

- reducen el esfuerzo de programación
- incrementan la velocidad de desarrollo y la calidad
- hay que hacer menos esfuerzo para utilizar colecciones
- se reutiliza el código

Todas las colecciones en Java están en el paquete **`java.util`**.

## 6.7.1 Java Collections Framework

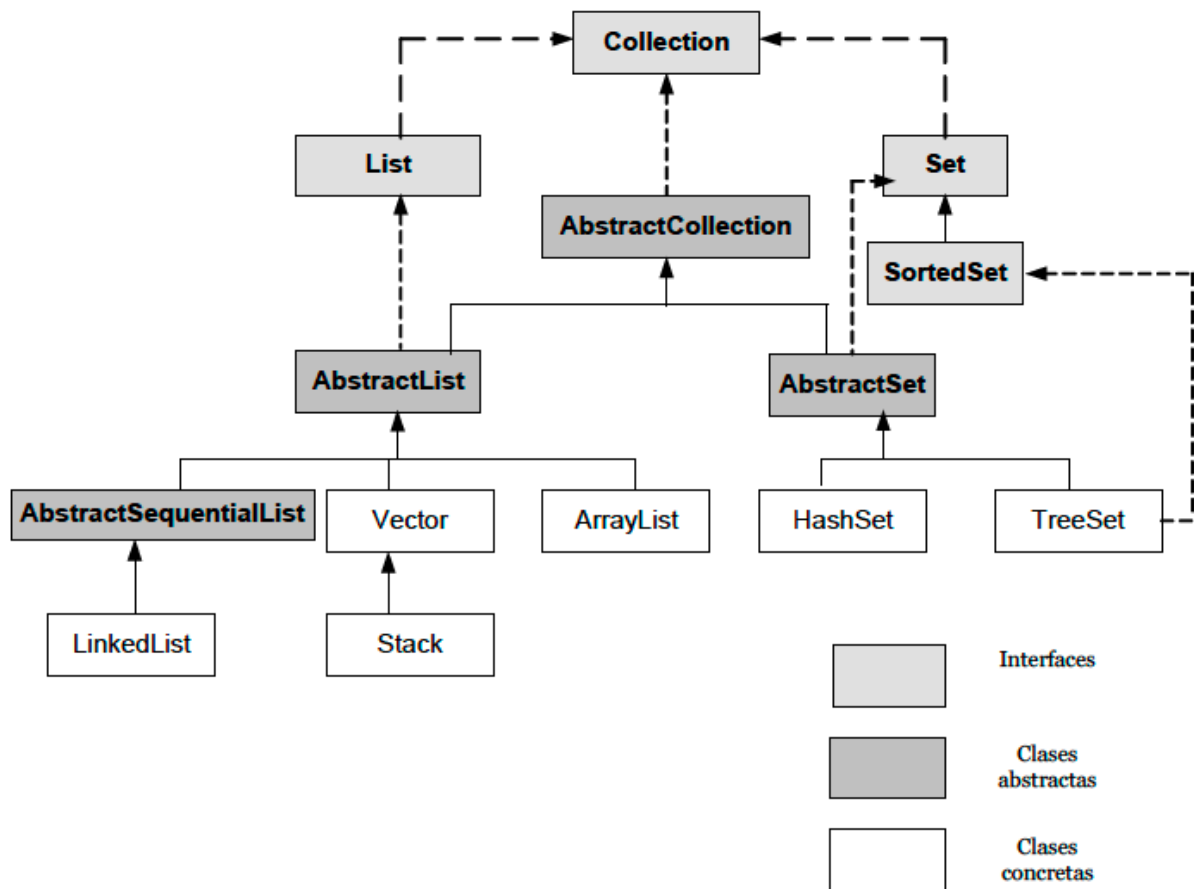
El diagrama de la Figura 1 muestra la jerarquía de colecciones Java. Java soporta tres tipos de colecciones (toda colección es un contenedor de objetos llamados elementos): List, Set y Map. Quedan definidas por tres interfaces:

a) **Collection** – interfaz raíz que representa a una colección de elementos

a.1) **Set** – interfaz que describe una colección de elementos no duplicados y sin orden (HashSet)

a.2) **List** – interfaz que representa a una colección de elementos ordenados que permite duplicados y el acceso por índice (ArrayList)

b) **Map** – describe una colección de objetos en la que se asocia una clave con un objeto. Las claves son como los índices, en List hay índices que son enteros, en Map las claves son objetos. Un map no contiene claves duplicadas (HashMap).



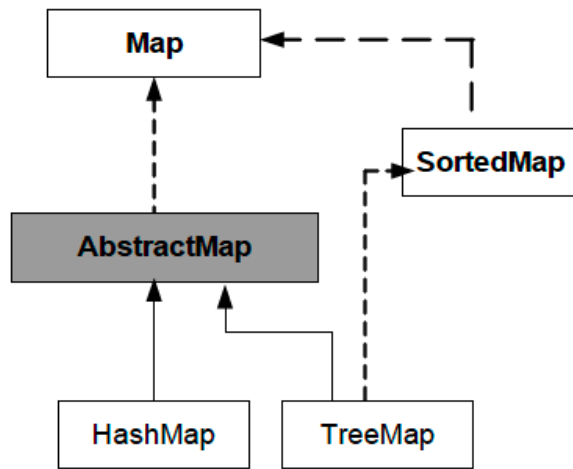


Figura 1

## 6.7.2 Implementación de las colecciones

Las clases concretas implementan los interfaces (ArrayList es una implementación de List, HashMap de Map, HashSet de Set). Las estructuras subyacentes utilizadas en estas clases que

representan colecciones son:

- arrays redimensionables
- listas enlazadas (linked lists)
- árboles (tree)
- tablas asociativas (hash table)

Cualquiera de estas estructuras se basa en el uso , bien de arrays, bien de enlaces (referencias) (listas enlazadas) o una combinación de ambas.

### Implementaciones

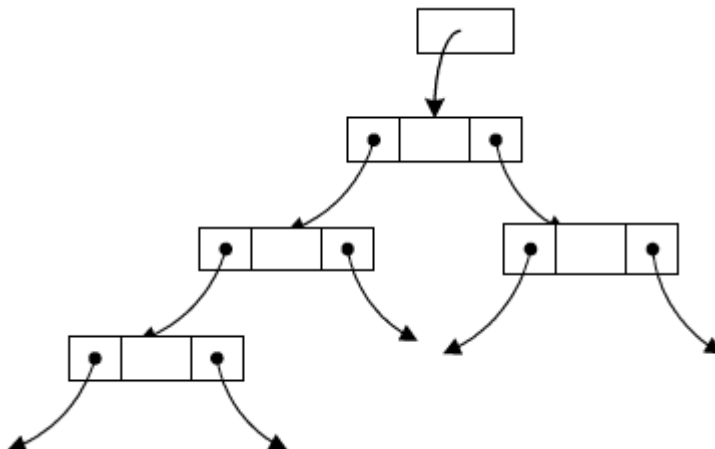
Interfaces	hashtable	array	tree	linkedList
<b>List</b>		ArrayList		LinkedList
<b>Set</b>	HashSet		TreeSet	
<b>Map</b>	HashMap		TreeMap	

□ Lista enlazada – es una estructura lineal formada por nodos dónde cada nodo “apunta” o “enlaza” con el siguiente



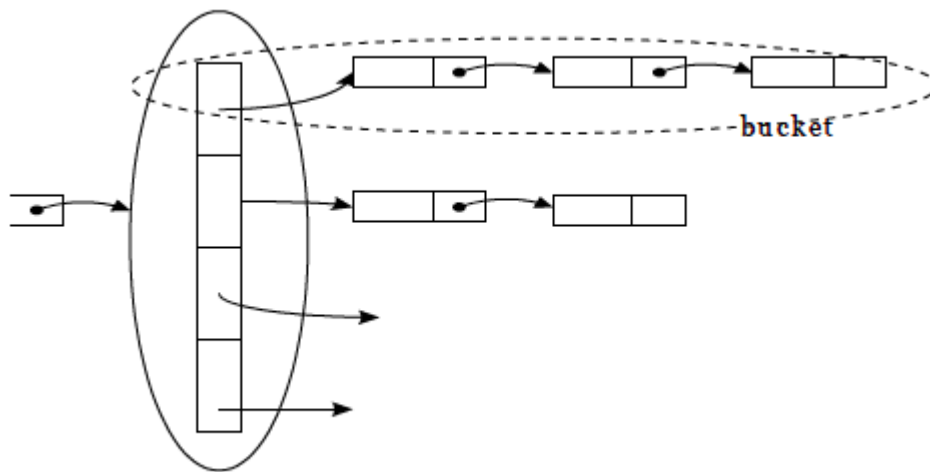
□ array redimensionable – array que “crece”

□ árbol (tree) - estructura no lineal de nodos enlazados en forma de árbol (generalmente binario)





□ hashtable – combinación de array y lista enlazada



## 6.7.3 Otras colecciones: Vector, Stack, LinkedList. La clase Collections

---

**Clase Vector** – esta clase tiene la misma funcionalidad y se utiliza de la misma forma que un ArrayList. ArrayList es más eficiente.

**Clase Stack** – define una colección de elementos que se gestionan como una “pila” (estructura LIFO). Esta clase en Java es una extensión de la clase Vector.

**Clase LinkedList** – así como ArrayList es una implementación concreta de List cuya estructura subyacente es un array, LinkedList se implementa como una estructura de datos dinámica, la lista enlazada. Esto permite que las inserciones y los borrados sean más rápidos.

**Clase Collections** – esta clase no representa a ninguna colección, simplemente contiene utilidades (métodos estáticos) para trabajar con colecciones de diferentes tipos.

### Ejercicio 6.19

Consulta la documentación de las clases anteriores en la API.

De la clase Collections busca los métodos: *max()*, *reverse()*, *shuffle()*, *sort()* y lee lo que hacen.

## 6.7.4 ¿Qué colección utilizar?

---

Cada colección tiene sus ventajas y sus desventajas: por ejemplo, ArrayList permite accesos más rápidos que LinkedList pero las inserciones y borrados son más lentos.

A la hora de utilizar una u otra hay que:

- pensar en el problema a resolver y elegir el interface (funcionalidad) más adecuado (decidir si lo que importan son los accesos, o hay muchas operaciones de inserción y borrado, ...)
- elegir la clase (implementación) que implemente el interfaz

### Ejercicio 6.20

Define una clase **DetectorPalindromos** cuyo atributo es una cadena y detecta si es o no palíndroma (se lee igual de izquierda a derecha que de derecha a izquierda) utilizando para ello una pila, la clase **Stack** de Java. La pila será de caracteres.

Solución

[Enlace](#) con la solución del ejercicio

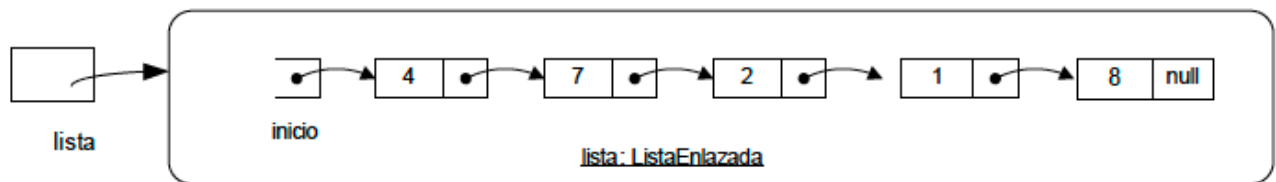
## 6.8 Listas enlazadas (Linked list)

Una lista enlazada simple es una estructura de datos lineal dinámica (crece / decrece) formada por una serie de nodos. Cada nodo consta de:

- una parte de información (el dato a almacenar, bien de un tipo primitivo o de un tipo referencia)
- un enlace o referencia al nodo que sigue en la lista

La estructura queda definida por una referencia inicial al primer nodo de la lista, la cabecera o principio de la lista. A partir de este nodo es posible recorrer la lista puesto que cada nodo

“apunta” al siguiente en la lista. El último nodo, al que no le sigue ningún otro, tiene una referencia null al siguiente. A veces, para mejorar la eficiencia se incluye una referencia al último nodo de la lista.



```
ListaEnlazada lista = new ListaEnlazada();
```

En una lista enlazada, al contrario que en un array, los accesos son más lentos, sin embargo, las inserciones y borrados son más eficientes.

Java ya incluye en la API una clase `LinkedList` pero nosotros vamos a ver cómo podemos definir nuestra propia clase que encapsule a una lista enlazada y las operaciones asociadas a la misma.

## 6.8.1 Definición de una clase

### ListaEnlazada. Operaciones (métodos) asociados

---

La siguiente clase implementa una lista enlazada:

```
/**
 * Representa a un objeto ListaEnlazada
 * que encapsula a una estructura dinámica simple
 * lista enlazada
 */
public class ListaEnlazada {

    private Nodo inicio; // la cabecera de la lista
    private int tamaño; //el n° de elementos,opcional

    /**
     * Constructor de la clase ListaEnlazada
     */
    public ListaEnlazada() {
        inicio = null;
        tamaño = 0;
    }

    /**
     * @return el n° de nodos actualmente en la lista
     */
    public int getTamaño() {
        return tamaño;
    }

    /**
     * @return true si la lista está vacía
     */
    public boolean listaVacía() {
        return inicio == null;
    }

}

/**
 * Modela un objeto Nodo de una lista enlazada
 * de enteros
 */
public class Nodo {
```

```
private int elemento; //el dato a guardar
private Nodo sgte; // la referencia al siguiente objeto Nodo
```

```
/**
 * Constructor de la clase Nodo
 */
public Nodo(int e, Nodo s) {
    elemento = e;
    sgte = s;
}
```

```
/**
 * @return el dato guardado en el nodo
 */
public int getElemento() {
    return elemento;
}
```

```
/**
 * @return el nodo que sigue al actual
 */
public Nodo getSiguiente() {
    return this.sgte;
}
```

```
/**
 * @param e el nuevo dato a guardar en el nodo
 */
public void setElemento(int e) {
    elemento = e;
}
```

```
/**
 * @param s el nodo que sigue al actual
 */
public void setSiguiente(Nodo s) {
    this.sgte = s;
}
```

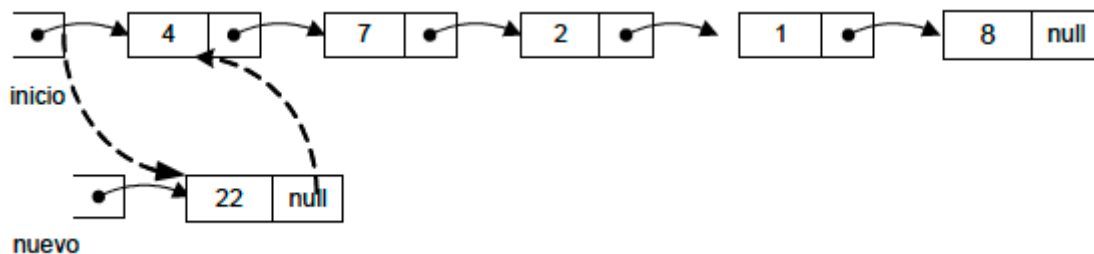
```
}
```

Nodo
+ Nodo(e:int,s:Nodo)
+ getElemento():int
+ getSiguiente():Nodo
+ setElemento(e:int)
+ setSiguiente(s:Nodo)

## 6.8.2 Operaciones sobre una lista enlazada

ListaEnlazada
+ getTamaño():int + listaVacia():boolean + añadirPrincipio(valor:int) + añadirFinal(nodo:Nodo) + borrarPrimero() + añadirPrincipio(nodo:Nodo) + añadirEntreNodos(nuevo:Nodo)

□ Añadir un nuevo nodo al principio de la lista

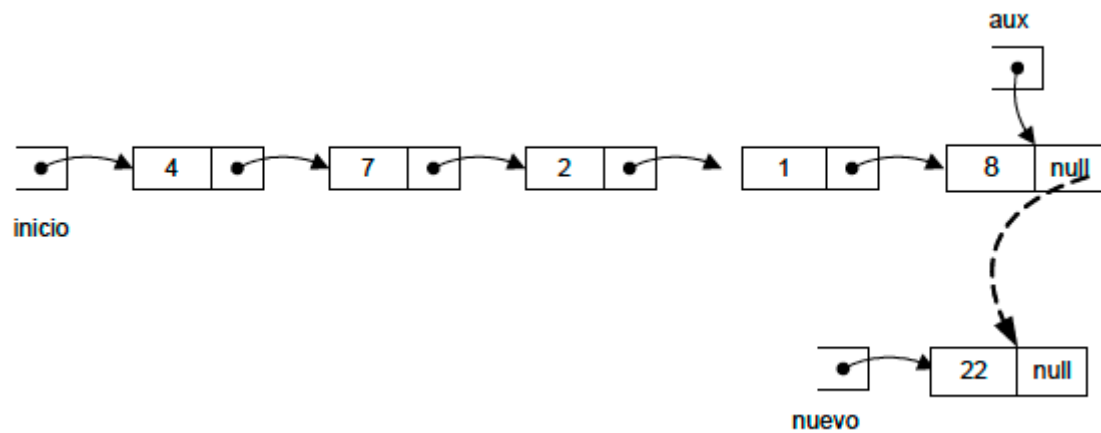


```
/**
 * @param nuevo el nodo a añadir
 */
public void añadirPrincipio(Nodo nuevo)
{
    nuevo.setSiguiente(inicio);
    inicio = nuevo;
    tamaño++;
}
```

```
/**
 *
 * @param n el dato a añadir
 */
public void añadirPrincipio(int n)
{
    Nodo nuevo = new Nodo(n, null);
    nuevo.setSiguiente(inicio);
    inicio = nuevo;
    tamaño++;
}
```

podemos llamar a añadirPrincipio(nuevo);

□ Añadir un nuevo nodo al final de la lista

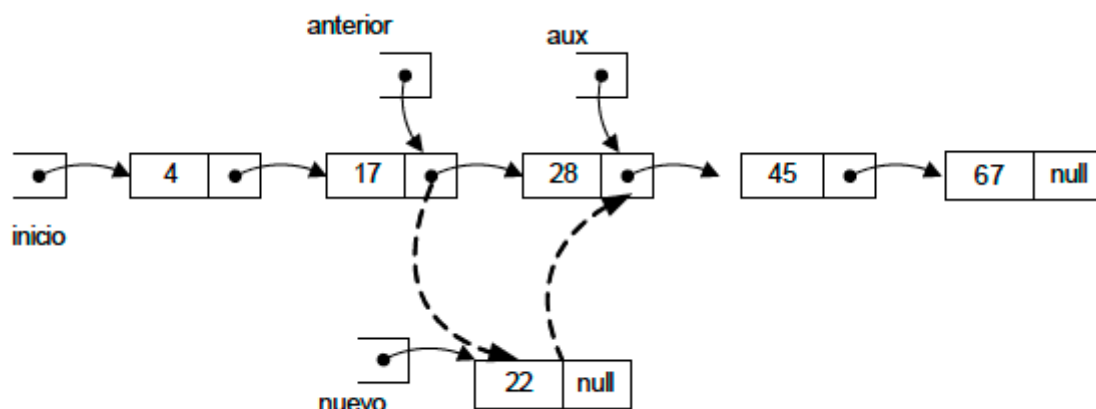


```

/**
 *
 * Requiere recorrer la lista hasta posicionarse en
 * el último elemento
 * @param el nuevo nodo a añadir
 */
public void añadirFinal(Nodo nuevo)
{
    if (listaVacia())
        inicio = nuevo;
    else
    {
        Nodo aux = inicio;
        while (aux.getSiguiente() != null)
            aux = aux.getSiguiente();
        aux.setSiguiente(nuevo);
    }
    tamaño++;
}

```

□ Añadir un nuevo nodo entre nodos (en orden)



```

/**
 * Añade un nuevo nodo entre nodos (en orden)
 * @param el nuevo nodo a añadir

```



```

*/
public void añadirEntreNodos(Nodo nuevo) {

    if (listaVacia())

        inicio = nuevo;

    else {

        Nodo anterior = null;

        Nodo aux = inicio;

        while ( aux != null && nuevo.getElemento() > aux.getElemento()) {
            anterior = aux;
            aux = aux.getSiguiente();
        }

        nuevo.setSiguiente(aux);

        if (anterior == null)
            inicio = nuevo;
        else
            anterior.setSiguiente(nuevo);
    }

}

```

□ Comprobar si la lista está vacía

```

/**
 *
 * @return true si la lista está vacía
 */
public boolean listaVacia() {

    return inicio == null; // o return tamaño == 0;
}

```

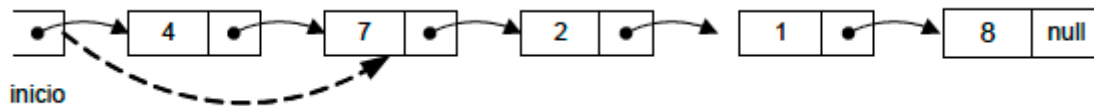
## Ejercicio 6.21

Implementa *public void añadirFinal(int e)* que añade también un nodo al final de la lista pero a partir de un entero que recibe como parámetro. (La creación del nuevo nodo se hace dentro de *añadirFinal()*).

### Solución

```
public void añadirFinal(int e) {
    Nodo nuevo=new Nodo(e,null);
    if(listaVacia( ))
        inicio=nuevo;
    else{
        Nodo aux=inicio;
        while (aux.getSiguiende( )!=null)
            aux=aux.getSiguiende( );
        aux.setSiguiende(nuevo);
    }
    tamaño++;
}
```

□ Borrar el primer nodo de una lista enlazada

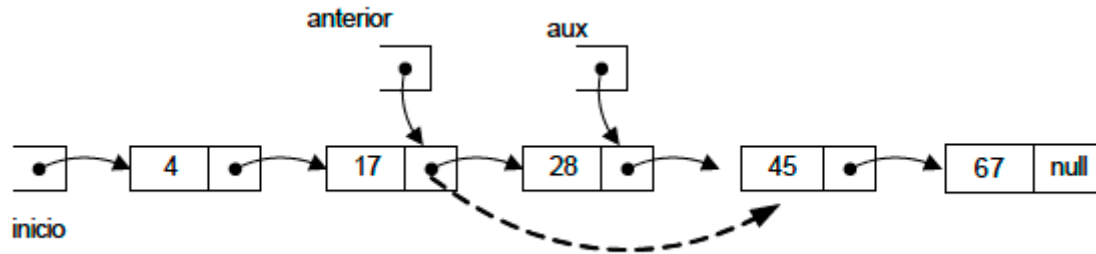


```
/**
 * Borra el nodo que está en el inicio o
 * cabecera de la lista
 *
 * @return devuelve el elemento borrado (esto es opcional)
 */
public int borrarPrimero() {

    if (listaVacia())
        throw new RuntimeException("Lista vacía");
    Nodo primero = inicio;
    inicio = inicio.getSiguiende();
    primero.setSiguiende(null);
    tamaño--;
    return primero.getElemento();

}
```

□ Borrar un nodo entre nodos (en orden)



```

/**
 * Borra un nodo entre nodos (en orden)
 * @param el nodo a borrar
 */
public void borrarEntreNodos(Nodo nuevo) {

    if (!listaVacia())
        throw new RuntimeException("Lista vacia");
    Nodo aux = inicio;
    Nodo anterior = null;
    while (aux != null && aux.getElemento() != nuevo.getElemento()) {
        anterior = aux;
        aux = aux.getSiguiente();
    }
    // aux es el nodo a borrar
    if (aux != null) // no esta el nodo si aux == null {

        if (anterior == null) //es el primero
            inicio = inicio.getSiguiente();
        else
            anterior.setSiguiente(aux.getSiguiente());
        aux.setSiguiente(null);
        tamaño--;

    } else
        System.out.println("No esta el valor buscado " + nuevo.getElemento());

}

```

□ Recorrer una lista enlazada

```

/**
 * Visitar cada uno de los nodos
 *
 * @return una cadena que es la representación
 * textual de la lista
 */
public String toString() {

    StringBuilder sb = new StringBuilder();
    Nodo aux = inicio;
    while (aux != null) {
        sb.append(aux.getElemento() + "\t");
    }
}

```

```

        aux = aux.getSiguiente();
    }
    return sb.toString();
}

```

## Ejercicio 6.22

Completa el método `testListaEnlazada()` de la clase siguiente:

```

public class DemoListaEnlazada {
    private ListaEnlazada lista;

    public DemoListaEnlazada () {
        lista = new ListaEnlazada();
    }

    public void testListaEnlazada () {
        ...
    }
}

```

- añade los siguientes valores a la lista: 3 4 5 9
- borra el primero de la lista
- escribe la lista
- añade los valores 17 y 18 al final de la lista (prueba las dos versiones)
- escribe la lista
- vacía la lista

### Solución

```

public class DemoListaEnlazada {
    private ListaEnlazada lista;

    public DemoListaEnlazada () {
        lista = new ListaEnlazada();
    }

    public void testListaEnlazada () {
        //añade los siguientes valores a la lista: 3 4 5 9
        lista.añadirPrincipio(3);
        lista.añadirPrincipio(4);
    }
}

```

```

lista.añadirPrincipio(5);
lista.añadirPrincipio(9);
//borra el primero de la lista
lista.borraPrimero( );
//escribe la lista
System.out.println(lista.toString( ));
//añade los valores 17 y 18 al final de la lista (prueba las dos
versiones)
lista.añadirFinal(17);
Nodo nuevo=new Nodo(18, null);
lista.añadirFinal(nuevo);
//escribe la lista
System.out.println(lista.toString( ));
//vacía la lista
lista.borrarLista( );
    }
}

```

- [Enlace](#) con todo el proyecto ListaEnlazada completo

## Ejercicio 6.23

Crea un proyecto **Pila** y **Cola** con listas.

- Añade una clase Pila junto con sus operaciones asociadas: *push()*, *pop()*, *estaVacía()*, *getCumbre()*.
- La pila se implementará como una lista enlazada en la que las inserciones y borrados se hacen siempre por el mismo lado, el top o la cumbre de la pila.
- Añade una clase *DemoPila* que incluye sólo el *main()* y testea la pila.
- Añade también una clase Cola junto con sus operaciones: *añadir()*, *borrar()*, *estaVacía()* y pruébala.
- La cola se implementa también como una lista enlazada. Las inserciones se hacen siempre por un extremo de la cola, el final, y los borrados por el otro extremo, el frente.

```

public class Pila
{
    private Nodo top;

    public Pila ()
    {
        top = null;
    }
}

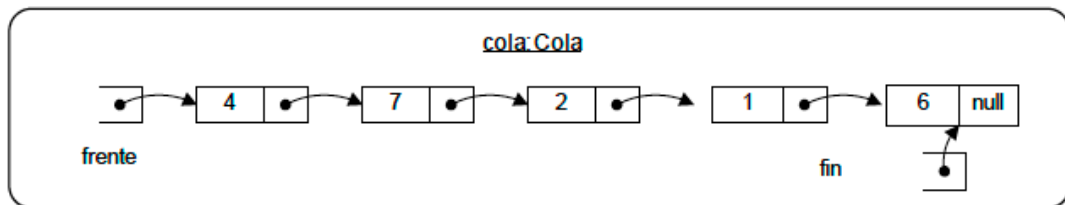
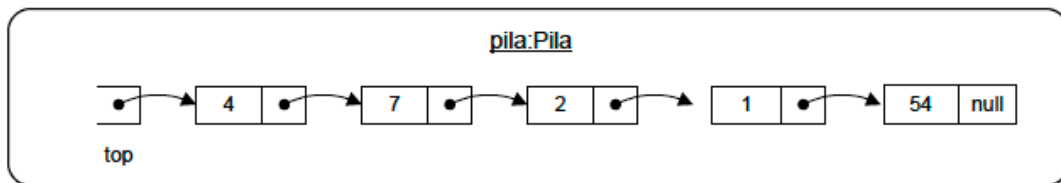
```

```

public class Cola
{
    private Nodo frente;
    private Nodo fin;

    public Cola ()
    {
        frente = null;
        fin = null
    }
}

```



Solución

[Enlace](#) con la solución del ejercicio

## Ejercicio 6.24

Consulta en la API la documentación de la clase `LinkedList`. Esta es la clase que proporciona Java para manejar listas enlazadas.

## 6.9 Árboles binarios

---

Un árbol es una estructura de datos lógica no lineal que modela una jerarquía. Consiste en una serie de nodos entre los que existe una relación de padre-hijo. Permiten representar jerarquías.

Los árboles tienen muchas aplicaciones en informática: sistemas de ficheros, búsqueda en bases de datos, procesos de decisión, ....

## 6.9.1 Terminología de árboles

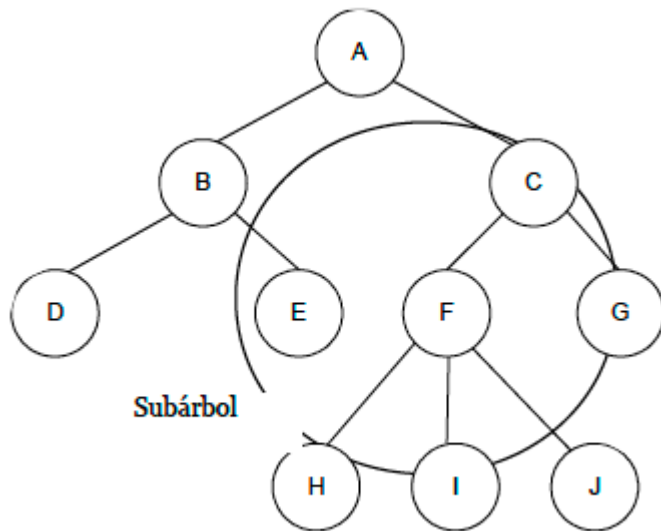
---

La relación entre los nodos del árbol es la de **padre-hijo**. Si hay una rama del nodo  $n$  al nodo  $m$ , entonces  $n$  es el padre de  $m$  y  $m$  es hijo de  $n$  (A es el padre de B y C, B y C son hijos de A).

Los hijos del mismo padre se llaman **hermanos** (B y C son hermanos).

Cada nodo del árbol tiene al menos un padre y existe un único nodo, el nodo **raíz** (A), que no tiene padre.

Un nodo que no tiene hijos se llama **hoja** del árbol (D, E, H, I, J y G son hojas).



La relación padre-hijo entre nodos se generaliza a las relaciones ascendente y descendiente (A es un **antecesor** de D y D es un **descendiente** de A).

Un **subárbol** de un árbol es cualquier nodo junto con todos sus descendientes. **Camino** es una secuencia de nodos conectados entre sí.

La **profundidad** de un nodo es el nº de nodos que se encuentran entre la raíz y el propio nodo (su nº de ascendientes). La profundidad de H es 3, la profundidad de D es 2.

La **altura** de un árbol es la máxima profundidad de cualquier nodo. En el ejemplo es 3.

El **grado** de un árbol es el maxº nº de hijos de sus nodos ( en el ejemplo el grado del árbol es 3).

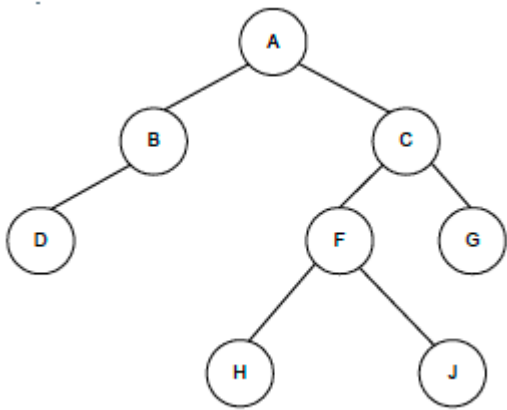


## 6.9.2 Árboles binarios

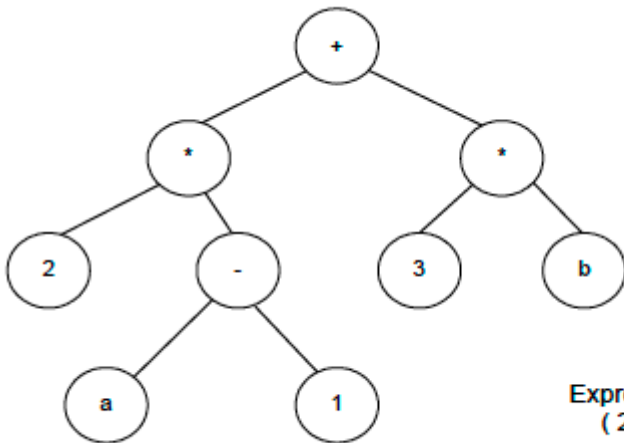
Un **árbol binario** es un árbol en el que cada nodo no puede tener más de dos hijos (es un árbol de grado 2).

En particular, un árbol binario es un conjunto de nodos que es ó bien vacío ó consta de un nodo raíz y dos árboles binarios denominados **subárbol izquierdo** y **subárbol derecho** (definición recursiva).

En un árbol binario los hijos se conocen como hijo izquierdo (subárbol izquierdo) e hijo derecho (subárbol derecho).



Los árboles binarios tienen múltiples aplicaciones, permiten, por ejemplo, representar expresiones aritméticas.

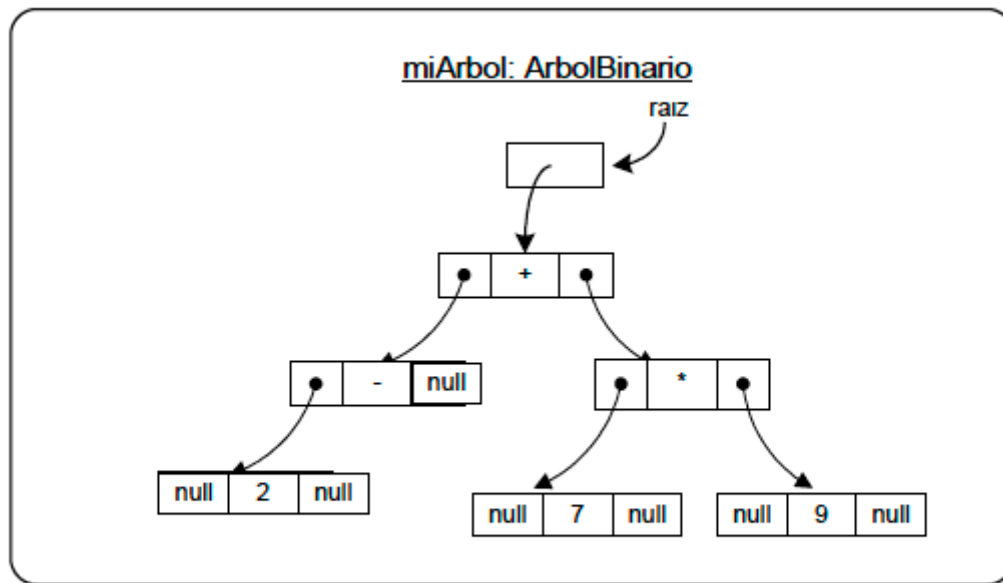


Expresión correspondiente:  
 $(2 * (a - 1) + (3 * b))$

## 6.9.3 Representación de un árbol binario

Un árbol binario puede implementarse mediante una estructura dinámica como una lista enlazada no simple (también con un array) formada por una serie de nodos en la que cada nodo contiene:

- un parte de información que guarda el dato
- dos referencias (punteros) que son enlaces a los hijos izquierdo (subárbol izquierdo) y derecho (subárbol derecho) respectivamente.



`ArbolBinario miArbol = new ArbolBinario();`

```
public class ArbolBinario
{
    private Nodo raiz;

    /**
     * Constructor de la clase ArbolBinario
     */
    public ArbolBinario()
    {
        raiz = null;
    }
}
```

```

public class Nodo
{
    private int elemento;
    private Nodo izdo;
    private Nodo dcho;

    public Nodo(int e)
    {
        elemento = e;
        izdo = null;
        dcho = null;
    }

    public Nodo(int e, Nodo i, Nodo d)
    {
        elemento = e;
        izdo = i;
        dcho = d;
    }

    public int getElemento()
    {
        return elemento;
    }

    public Nodo getIzquierdo()
    {
        return izdo;
    }
}

```

```

    public Nodo getDerecho()
    {
        return dcho;
    }

    public void setElemento(int e)
    {
        elemento = e;
    }

    public void setIzquierdo(Nodo i)
    {
        izdo = i;
    }

    public void setDerecho(Nodo d)
    {
        dcho = d;
    }
}

```

## 6.9.4 Recorridos de un árbol binario

El proceso de acceder ó visitar a todos los nodos de un árbol se conoce normalmente como **recorrido del árbol**.

Los tres recorridos más típicos se clasifican de acuerdo al momento en que se visita la raíz en relación con la visita a los subárboles.

□ **Recorrido en PREORDEN** (raíz – izquierdo – derecho) (escribir la estructura de un documento)

1. Visitar la raíz
2. Recorrer el subárbol izquierdo
3. Recorrer el subárbol derecho

□ **Recorrido en ENORDEN** (izquierdo – raíz – derecho) (escribir una expresión aritmética)

1. Recorrer el subárbol izquierdo
2. Visitar la raíz
3. Recorrer el subárbol derecho

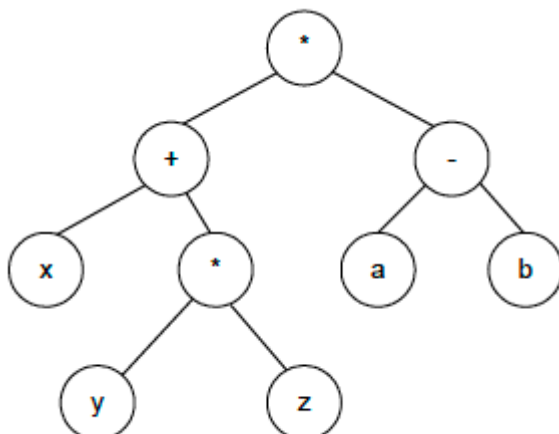
□ **Recorrido en POSTORDEN** (izquierdo – derecho – raíz) (calcular el espacio utilizado por los ficheros de un directorio y sus subdirectorios)

1. Recorrer el subárbol izquierdo
2. Recorrer el subárbol derecho
3. Visitar la raíz

Los algoritmos recursivos son más fáciles para implementar estos recorridos que los iterativos.

### Ejercicio 6.25

Establece la secuencia de nodos visitada en el siguiente árbol binario en los tres recorridos posibles:



```

/**
 * Recorrido en inorden de un árbol binario
 */

public void recorrerInorden() {
    recorrerInorden(raiz);
}

/**
 * Recorrido en inorden de un árbol binario
 * el método es recursivo
 */

private void recorrerInorden(Nodo raiz) {
    if (raiz != null) {
        recorrerInorden(raiz.getIzquierdo());
        System.out.println(raiz.getElemento() + "\t");
        recorrerInorden(raiz.getDerecho());
    }
}

```

#### Solución

Inorden: x, +, y, \*, z, \*, a, -, b  
 Postorden: x, +, y, \*, z, a, -, b, \*  
 Preorden: \*, x, +, y, \*, z, a, - b

## Ejercicio 6.26

Escribe los métodos `recorrerPreorden()` y `recorrerPostorden()` de forma análoga al método anterior.

#### Solución

[Enlace](#) con la solución del ejercicio



## 6.9.5 Árboles binarios de búsqueda

---

Un **árbol binario de búsqueda** es un árbol binario en el que, dado un nodo, todos los valores del subárbol izquierdo de ese nodo son menores a él y los del subárbol derecho mayores a él.

El recorrido en **enorden** de un árbol binario de búsqueda permite obtener los valores en una secuencia ascendente.

## 6.9.5.1.- Algunas operaciones sobre un árbol binario de búsqueda

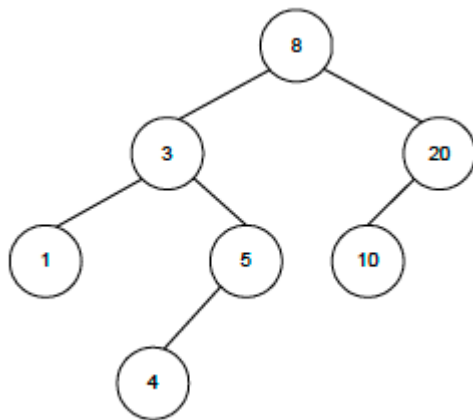
---

### Insertar un valor en el árbol

Dada una secuencia de valores, por ejemplo, 8 3 1 20 10 5 4 , la construcción de un árbol binario de búsqueda a partir de ella consiste en insertar cada uno de esos valores en el árbol:

- el primer valor de la secuencia será la raíz del árbol inicial, en nuestro ejemplo, 8
- el valor 3, el siguiente en la secuencia, es menor que 8, luego pasará a ser un nodo del subárbol izquierdo
- 1 es menor que 8, debe formar parte del subárbol izquierdo. También es menor que 3 luego será un hijo izquierdo de 3
- 20 es mayor que la raíz del árbol (8), pasa a formar parte del subárbol derecho
- el proceso se repite hasta terminar con la secuencia

Los árboles binarios de búsqueda no contienen valores duplicados por lo que antes de insertar normalmente se comprueba si ya existe un nodo con el valor que se quiere añadir.



La construcción de un árbol binario de búsqueda es relativamente sencilla si se hace de forma recursiva (ocurre lo mismo si queremos localizar un determinado valor dentro del árbol).



```

/**
 * @param e el valor a añadir
 */
public void añadir(int e)
{
    Nodo nuevo = new Nodo(e, null, null);
    raiz = añadirRec(raiz, nuevo);
}

/**
 *
 * @param raiz la raiz del árbol considerado en cada llamada
 *           recursiva
 * @param nuevo el nuevo nodo a añadir
 * @return la nueva raíz
 */
private Nodo añadirRec(Nodo raiz, Nodo nuevo)
{
    Nodo aux = raiz;
    if (aux == null)
        aux = nuevo;
    else if (aux.getElemento() < nuevo.getElemento()) //añadir en subárbol derecho
        aux.setDerecho(añadirRec(aux.getDerecho(), nuevo));
    else
        aux.setIzquierdo(añadirRec(aux.getIzquierdo(), nuevo));
    return aux;
}

```

### **Buscar un valor en el árbol**

La búsqueda en un árbol binario de búsqueda es dicotómica ya que en cada examen de un nodo se elimina aquel de los subárboles que no contiene el valor buscado.

Si queremos buscar un valor X se compara con la raíz del árbol. Si no coincide se pasa al subárbol izquierdo ó derecho según el resultado de la comparación y se repite la búsqueda en ese subárbol. El proceso acaba cuando se encuentra el valor X ó hasta encontrar un subárbol vacío.

## **Ejercicio 6.27**

Escribe el método recursivo `esta(Nodo raiz, int e)` correspondiente a la búsqueda de un valor dentro de un árbol binario de búsqueda. Los parámetros indican la raíz del árbol a partir de dónde hay que empezar a buscar y el valor a buscar, e.

```

public boolean esta(int e) {

    return esta(raiz, e);

}

```

Solución

[Enlace](#) con la solución del ejercicio

## Ejercicio 6.28

Escribe el método public boolean esta(Nodo raiz, int e) correspondiente a la búsqueda de un valor dentro de un árbol binario de búsqueda. Los parámetros indican la raíz del árbol a partir de dónde hay que empezar a buscar y el valor a buscar, e. La búsqueda ahora será iterativa.

Solución

[Enlace](#) con la solución del ejercicio

## Ejercicio 6.29

Diseña un método recursivo *int contarHojas(Nodo raiz)* que cuenta el nº de nodos hoja en un árbol.

Solución

[Enlace](#) con la solución del ejercicio

