

## UT6

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

array = colección de  
valores del mismo tipo

En la unidad de trabajo anterior estudiamos una **estructura de datos de tamaño fijo**, el array. Los arrays permitían **almacenar datos de tipo primitivo y objetos**.

Los arrays son estructuras adecuadas en determinadas ocasiones ya que **permiten un rápido acceso a sus elementos**. Sin embargo tienen desventajas. **Las operaciones de inserción y borrado de sus elementos son lentas (hay que desplazar los valores)** así como la búsqueda y la ordenación. Además sabemos que su **tamaño no puede cambiar**.

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**.

## 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 n° 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 n° 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
{
    // Almacén de notas
    private ArrayList<String> notas;

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

    /**
     * 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 cuantas = notas.size();
    int indice=0;
    while (indice < cuantas) {
        System.out.println((indice + 1) + " " + notas.get(indice));
        indice++;
    }
}

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

```

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 permitió 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>();    // notas = new ArrayList<>();
}

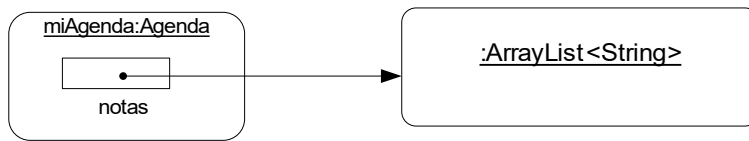
```

A partir de Java 7 si se omite el tipo al instanciar la colección el compilador lo infiere.

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

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

tendremos el siguiente diagrama de objetos:



**Ejer 6.1.** Una clase Banco mantiene dos colecciones ArrayList, una que guarda objetos de tipo Cliente y otra que guarda objetos de tipo Cuenta. Define el esqueleto de la clase con el par de atributos, *listaClientes* y *listaCuentas*, que definen estos valores. Incluye también el constructor.

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

Veamos algunos de los múltiples métodos que proporciona la clase ArrayList.

#### 6.1.2.1- Añadir un objeto a una colección ArrayList

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

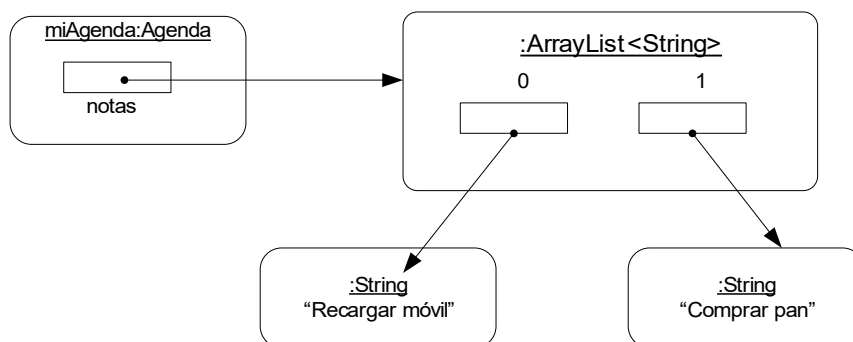
```

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);
    }
    .....
}
  
```

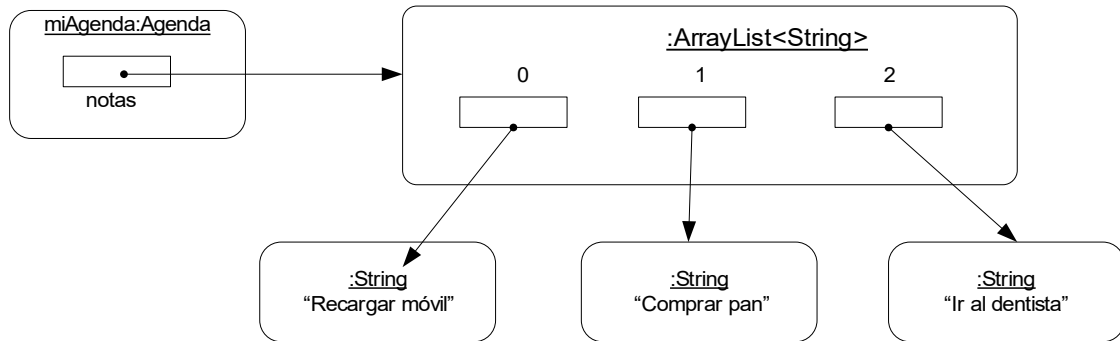
objeto que se añade a la colección ArrayList

```

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.

### 6.1.2.2- 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.

### 6.1.2.3- Numeración dentro de las colecciones

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`.

### 6.1.2.4- 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*”.

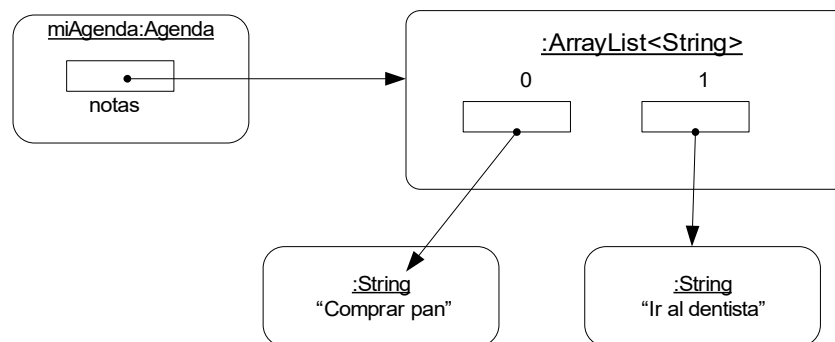
Puesto que el método `get()` no comprueba que el argumento que se le pasa es un valor correcto, el método `mostrarNota()` de la clase `Agenda` hace una comprobación de la posición. Si se intenta recuperar un objeto no existente con `get()` se genera un error (una excepción), `IndexOutOfBoundsException`.

#### 6.1.2.5- 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.

### 6.1.2.6.- Procesando la colección completa

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

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

### 6.1.2.7.- Otros métodos de la clase ArrayList.

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

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

### 6.1.3.- El bucle *for* mejorado en Java 1.5

La versión 1.5 de Java introdujo una extensión para el bucle *for* de tal forma que su utilización 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 a 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.).

### 6.1.4.- 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;`**

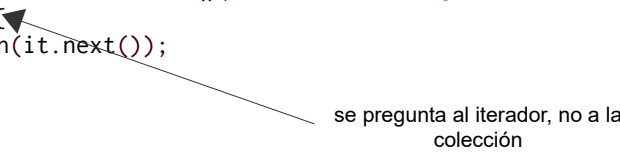
La **clase `Iterator`** proporciona, entre otros, los siguientes métodos para recorrer una colección:

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



se pregunta al iterador, no a la colección

**Ejer 6.2.** 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:

- a) con un iterador
- b) con el bucle `while` (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)

### 6.1.5.- 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.

**Ejer 6.3.** Abre el proyecto `Club` desde BlueJ. 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`.

- a) Define en la clase `Club` el atributo *miembros*
- b) En el constructor de la clase crea la colección. Compila el proyecto.
- c) 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 o

- d) 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);
```

- e) 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.
- f) 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
- g) 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*.
- h) Escribe el método, `public void listarClub()` que visualiza todos los miembros del club. Utiliza un bucle *for* mejorado para recorrer la colección

**Ejer 6.4.** En este ejercicio trabajaremos con el proyecto `Producto` que incluye las siguientes clases:

- **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`. El método `añadirProducto()` añade un nuevo producto a la colección. Modifica esta clase completando los siguientes métodos:
  - a) `escribirDetallesProductos()` - muestra los detalles de cada producto. Haz tres versiones de este método: con iterador, con *for.. each* y con *for* e índices.
  - b) `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.
  - c) `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.
  - d) Modifica el método `añadirProducto()` para que no sea posible añadir un producto con el mismo identificador que otro existente.

- e) `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?
- f) `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.
- g) `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.

## 6.2.- Las clases envoltantes (*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 envoltantes* o *wrapper classes*.

### 6.2.1.- Convirtiendo tipos primitivos en clases

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

Tipo primitivo	Clase envoltante (wrapper)
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>

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 llamando 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');
```

objetoNumero:Integer

int

16

### 6.2.2.- Convirtiendo clases envoltantes en tipos primitivos

Cada clase envolvente 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  $n^{\text{os}}$  enteros hacemos:

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

Si añadimos un  $n^{\text{o}}$  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();
```

A partir de Java 1.5 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 ahora ( en Java 1.5 ) un  $n^{\text{o}}$  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
```

**Ejer 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()
    {

    }

}

```

- x 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.
- x el método `sumar()` devuelve la suma de los valores enteros que almacena la colección. Haz dos versiones, con iterador y con `for.. each`
- x `toString()` devuelve la representación textual de la colección de la forma: 4, 7, 5, 3, 2, .... (utiliza `StringBuilder`)

## 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");
```

**Ejer 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.

**Ejer 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:

isLetter()
isDigit()
isLowerCase()
toUpperCase()

**Ejer 6.8.** Dados el siguiente ejemplo:

```
TextField txtNumero = new TextField();
.....
String strNumero = txtNumero.getText();
int numero =
```

Completa la asignación indicada para obtener el valor numérico correspondiente.

## 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 n° 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 n° 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 n° 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 n°s 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;`

**Ejer 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.

### 6.4.1. 1. – Algunas operaciones habituales en un `HashMap`

<code>put(clave, valor)</code>	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
<code>get(clave)</code>	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.
<code>remove(clave)</code>	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. (**)
<code>values()</code>	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> .
<code>keySet()</code>	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.
<code>entrySet()</code>	devuelve un conjunto ( <i>Set</i> ) con todas las entradas (de tipo <i>Map.Entry</i> ) del <i>map</i> . Es una <i>vista</i> sobre las entradas, no es un conjunto independiente.
<code>size()</code>	devuelve el nº de entradas en el <i>map</i>
<code>isEmpty()</code>	devuelve <i>true</i> si no hay entradas en el <i>map</i>
<code>containsKey(clave)</code>	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());
}
```

**Ejer 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*)

**Ejer 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)

**Ejer 6.12.** Responde a las siguientes cuestiones consultando la documentación de Java:

- a) ¿Qué ocurre si se añade una entrada a un *map* con una clave que ya existe? Pruébalo con el ejemplo anterior
- b) ¿Y si se añade una entrada con un valor que ya existe?
- c) ¿Cómo sabemos si una determinada clave está contenida en el *map*?

- d) ¿Qué ocurre si se intenta buscar un valor y la clave no existe en el *map*? – pruébalo en el ejemplo anterior.
- e) ¿Cómo podemos saber cuántas entradas hay en un *map*?

**Ejer 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");`

## 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()`, `hashNext()`, `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);
    }
}
```

#### Ejer 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*

### 6.4.3. – Las clases **TreeMap** / **LinkedHashMap** y **TreeSet** / **LinkedHashSet**

La clases **TreeMap** y **LinkedHashMap** tienen la misma funcionalidad que **HashMap**:

- **TreeMap** permite recuperar las claves ordenadas
- **LinkedHashMap** permite recuperar las claves en el mismo orden en que se introdujeron

La clases **TreeSet** y **LinkedHashSet** tienen la misma funcionalidad que **HashSet**:

- **TreeSet** permite recuperar los valores del conjunto ordenados
- **LinkedHashSet** permite recuperar los valores del conjunto en el mismo orden en que se introdujeron

## 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

<b>@author</b>	nombre del autor
<b>@version</b>	nº de versión y fecha
<b>@param</b>	nombre del parámetro y descripción
<b>@return</b>	descripción del valor de retorno
<b>@throws</b> <b>@exception</b>	tipo de excepción lanzada y circunstancias (se pueden usar indistintamente)

### 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*.

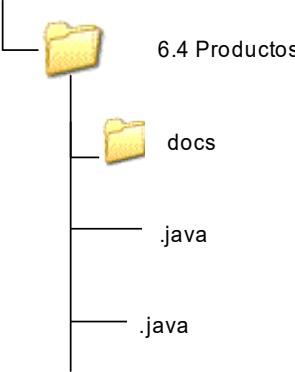
**Ejer 6.15.** Documenta adecuadamente las clases del proyecto realizado en el ejercicio 6.3 (Club) y genera la documentación desde BlueJ. Observa la nueva carpeta creada dentro del proyecto con la documentación generada (la carpeta *doc*).

## 6.5.4. – Utilizando javadoc desde la línea de comandos

**Ejer 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* :



El diagrama muestra una estructura de carpetas: una carpeta principal llamada '6.4 Productos' que contiene una subcarpeta 'docs', un archivo '.java' y otro archivo '.java'. A la derecha, se muestran dos comandos de terminal para ejecutar 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.

## 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:

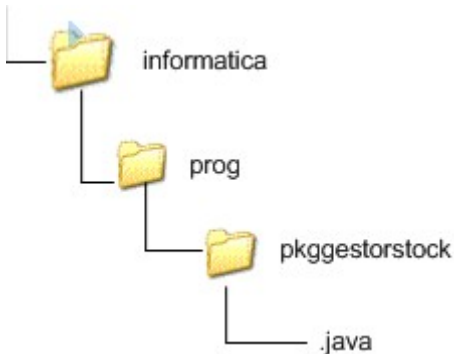
- *para localizar las clases* – las clases con funciones similares se sitúan en el mismo paquete y así se localizan más fácilmente
- *para evitar conflictos de nombre* – cuando se desarrollan clases que van a ser compartidas por varios programadores, colocar las clases en paquetes evita el conflicto 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*.

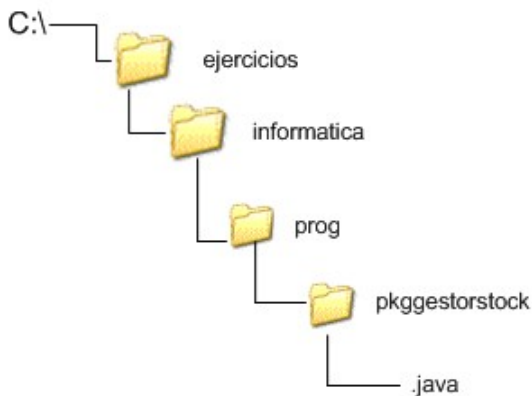
Por convención, los nombres de los paquetes se escriben en minúsculas.

### 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.prog.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 compilador y el intérprete java buscan los paquetes, las clases compiladas *.class* y los *.jar* (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.prog.pkggestorstock;
```

nombre del paquete

como primera sentencia de la clase.

### 6.6.3.- Utilizando las clases del paquete

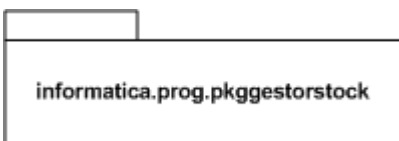
Para utilizar las clases de un paquete las importamos con la sentencia *import*:

```
import informatica.prog.pkggestorstock.*;  
// con * se importan todas las clases del paquete  
import informatica.prog.pkggestorstock.Producto;  
//se importa una sola clase (la forma que usaremos)
```

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

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

### 6.6.4.- Paquetes y BlueJ

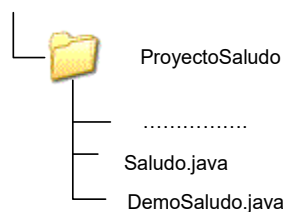


En primer lugar indicaremos cuál es la notación que utiliza UML para denotar un paquete.

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).

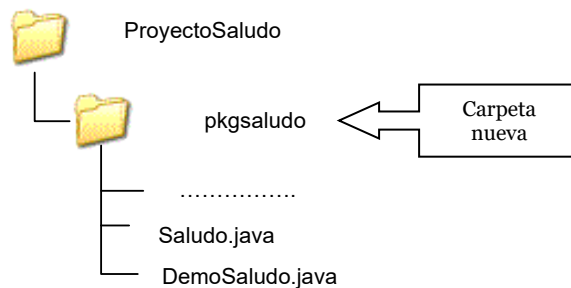
Podemos crear un paquete en BlueJ partiendo de un proyecto que ya incluye una serie de clases de la forma siguiente:

- 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\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 (únicamente en el caso de que el proyecto incluya una parte gráfica, si no es así, hay que ejecutar desde línea de comandos). El ordenador que ejecute un *.jar* ha de tener instalado el JDK o JRE y asociado con ficheros *.jar*.

**Ejer 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).

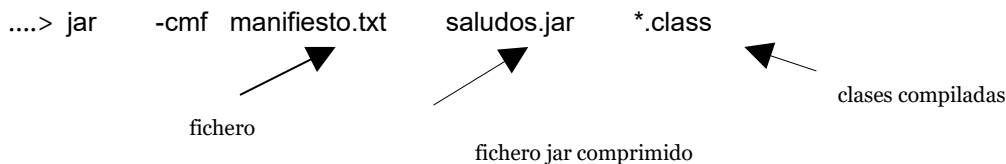
### 6.6.5.2.- Creando ficheros *jar* desde la línea de comandos

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



-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

-e: permite especificar el punto de entrada a la aplicación sin editar o crear el fichero manifiesto

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
```

Otra forma más cómoda de hacerlo usando el flag **-e**:

```
....> jar -cvfe saludos.jar *.class
```

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

**Ejer 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.

### 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:

<i><b>Paquete</b></i>	<i><b>Descripción</b></i>
<b>java.lang</b>	clases centrales de la plataforma (cadenas, números,...). No es necesario incluir la sentencia <i>import</i> cuando se utilizan clases de este paquete
<b>java.util</b>	utilidades varias:fechas, generadores de n <sup>os</sup> 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 <i>awt</i> )
<b>java.applet</b>	clase para crear <i>applets</i>
<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. Desde la aparición de java 2 (a partir de la versión Java 1.2) todas las colecciones fueron reorganizadas. El nuevo 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, TreeSet)
  - a.2) **List** – interfaz que representa a una colección de elementos ordenados que permite duplicados y el acceso por índice (ArrayList, LinkedList)
- 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, TreeMap).

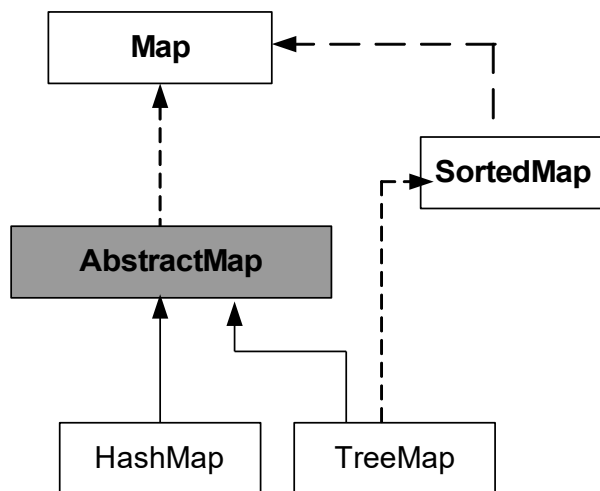


Figura 1

### 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:

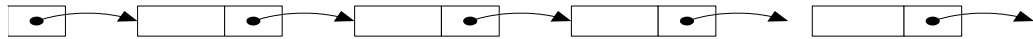
- arrays redimensionables
- listas enlazadas (*linked lists*)
- árboles (*tree*)
- tablas asociativas (*hash table*)

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

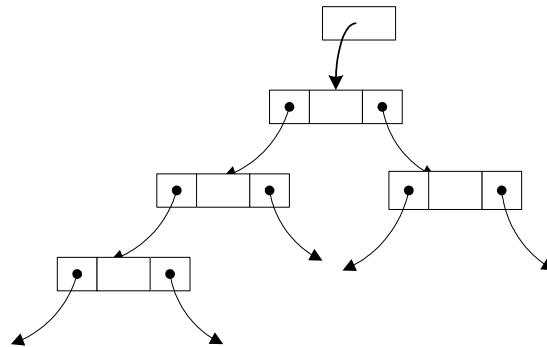
## Implementaciones

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

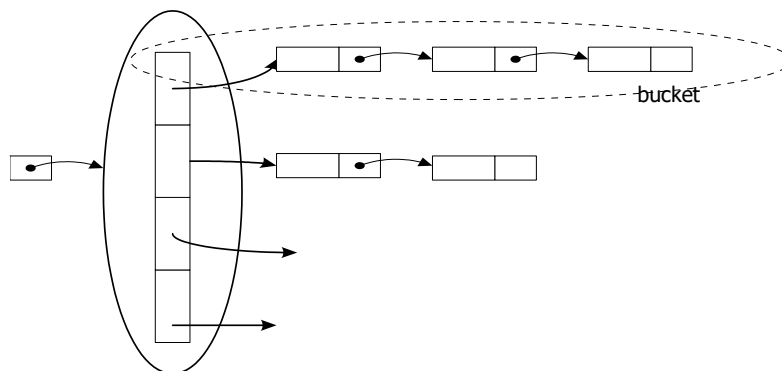
- \* *Lista enlazada* – es una estructura lineal formada por nodos donde cada nodo “apunta” o “enlaza” con el siguiente



- \* *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 ha sido reemplazada por ArrayList a partir de Java 2. Tiene la misma funcionalidad y se utiliza de la misma forma pero Vector es una clase sincronizada al contrario que 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.

**Ejer 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

**Ejer 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.