Programación

(curso 2020-2021)

UT6 Aplicación de las estructuras de almacenamiento

1. Introducción

Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?

Lo mismo ocurre en otros casos. Para poder realizar ciertas aplicaciones se necesita poder manejar datos que van más allá de meros **datos simples** (números y letras). A veces, los datos que tiene que manejar la aplicación son **datos compuestos**, es decir, datos que están compuestos a su vez de varios datos más simples. Por ejemplo, un pedido está compuesto por varios datos, los datos podrían ser el cliente que hace el pedido, la dirección de entrega, la fecha requerida de entrega y los artículos del pedido.

Los datos compuestos son un tipo de estructura de datos, y en realidad ya los has manejado. **Las clases son un ejemplo de estructuras de datos que permiten almacenar datos compuestos**, y el objeto en sí, la instancia de una clase, sería el dato compuesto. Pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales.

Aquí podrás aprender esas soluciones adicionales. Esas soluciones consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible.

2. Estructuras de almacenamiento

¿Cómo almacenarías en memoria un listado de números del que tienes que extraer el valor máximo? Seguro que te resultaría fácil. Pero, ¿y si el listado de números no tiene un tamaño fijo, sino que puede variar en tamaño de forma dinámica? Entonces la cosa se complica. Un listado de números que aumenta o decrece en tamaño es una de las cosas que podemos hacer utilizando estructuras de datos.

Respecto a las clases y los objetos, debes saber que las clases en sí mismas son la evolución de un tipo de estructuras de datos conocidas como datos compuestos (también llamadas **registros**). Las clases, además de aportar la ventaja de agrupar datos

relacionados entre sí en una misma estructura (característica aportada por los datos compuestos), permiten agregar métodos que manejen dichos datos, ofreciendo una herramienta de programación sin igual.

Las estructuras de almacenamiento, en general, se pueden clasificar de varias formas. Por ejemplo, atendiendo a si pueden almacenar datos de diferente tipo, o si solo pueden almacenar datos de un solo tipo, se pueden distinguir:

- Estructuras con capacidad de **almacenar varios datos del mismo tipo:** varios números, varios caracteres, etc. Ejemplos de estas estructuras son los arrays, las cadenas de caracteres, las listas y los conjuntos.
- Estructuras con capacidad de **almacenar varios datos de distinto tipo:** números, fechas, cadenas de caracteres, etc., todo junto dentro de una misma estructura. Ejemplos de este tipo de estructuras son las clases.

Otra forma de clasificar las estructuras de almacenamiento va en función de si pueden o no cambiar de tamaño de forma dinámica:

- **Estructuras cuyo tamaño se establece en el momento de la creación o definición** y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multimensionales).
- **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las listas, árboles, conjuntos y, como veremos también, el caso de algunos tipos de cadenas de caracteres.

Por último, atendiendo a la forma en la que los datos se ordenan dentro de la estructura, podemos diferenciar varios tipos de estructuras:

- **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los arrays.
- **Estructuras ordenadas**. Se trata de estructuras que al incorporar un dato nuevo a todos los datos existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc.

Ya hemos tenido un primer contacto con los arrays en la UT4 no obstante vamos a ver un rápido repaso:

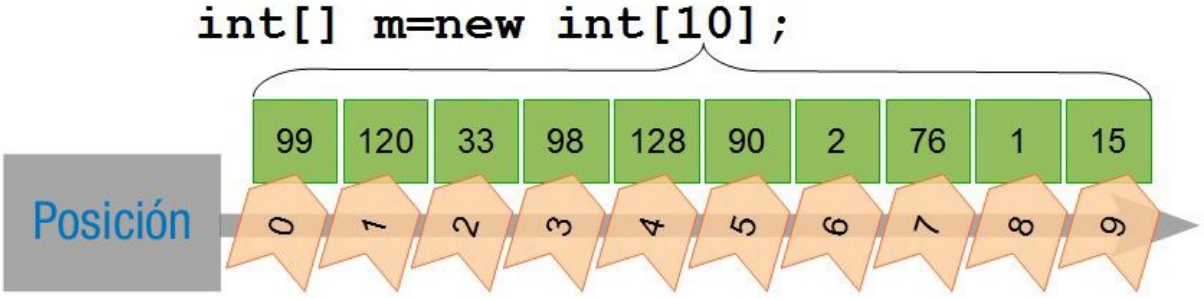
Los arrays permiten almacenar una colección de objetos o datos del mismo tipo. Son muy útiles y su utilización es muy simple:

- **Declaración del array**. La declaración de un array consiste en decir "esto es un array" y sigue la siguiente estructura: "tipo[] nombre;". El tipo será un tipo de variable o una clase ya existente, de la cual se quieran almacenar varias unidades.
- **Creación del array**. La creación de un array consiste en decir el tamaño que tendrá el array, es decir, el número de elementos que contendrá, y se pone de la siguiente forma: "nombre=new tipo[dimension]", donde dimensión es un número entero positivo que indicará el tamaño del array. Una vez creado el array este no podrá cambiar de tamaño.

Un ejemplo de su uso sería :

```
int[] n; // Declaración del array.
n = new int[10]; //Creación del array reservando para el un espacio en memoria.
int[] m = new int[10]; // Declaración y creación en un mismo lugar.
```

Una vez hecho esto, ya podemos almacenar valores en cada una de las posiciones del array, usando corchetes e indicando en su interior la posición en la que queremos leer o escribir, teniendo en cuenta que la primera posición es la cero y la última el tamaño del array menos uno. En el ejemplo anterior, la primera posición sería la 0 y la última sería la 9.



3. Volcado de datos entre ficheros y arrays

En la unidad anterior vimos cómo trabajar con ficheros, veamos un ejemplo de arrays con ficheros. Imaginemos el fichero parcial.txt en el que el primer valor indica el número de datos que almacena y a continuación vienen cada uno de esos datos. Para leer el fichero, utilizaremos las clases File y Scanner. En este caso, mientras leemos el fichero, también recorreremos el array para ir introduciendo cada dato en su posición.

El código sería:

```
Scanner leerFich = new Scanner(new File("parcial.txt"));

// Leemos el primer dato: el número de valores que hay en el fichero
// Creamos el array de ese tamaño
int total = leerFich.nextInt();
int[] valores = new int[total];
```

```
// Recorremos el array y vamos guardando cada uno de los valores
for (int i = 0; i < valores.length; i++) {
    valores[i] = leerFich.nextInt();
}
```

En este caso, el array se crea en tiempo de ejecución y su longitud dependerá del primer valor del fichero. Se creará un array con tantos elementos como indique el primer elemento del fichero.

Una vez que hemos trabajado con el array si lo que queremos es guardar sus valores en un fichero deberíamos usar las clases File y PrintStream. Recorreremos el array y cada uno de sus elementos, en vez de mostrarlos por consola, los guardaremos en el fichero salida.txt

El código sería:

```
PrintStream escribirFich = new PrintStream(new File("salida.txt"));

int total = 0;
for (int i = 0; i < valores.length; i++) {
   escribirFich.print(valores[i] + " ");
       total += valores[i];
}
escribirFich .println(" = " + total);
```

En este caso, además de guardar los datos de array, al final del fichero se guarda la suma de todos los valores.

4. Estructuras dinámicas de almacenamiento

Hemos visto cómo trabajar con arrays, estructuras que almacenan un número fijo de datos siempre del mismo tipo. Podemos guardar números enteros, palabras u objetos pero siempre definimos su número cuando se crea el array y es imposible cambiarlo a posteriori.

También sabemos definir clases. Nos permiten agrupar diferentes tipos de datos pero la estructura es fija una vez creada.

Ahora, vamos a ver las **estructuras cuyo tamaño es variable**, conocidas como estructuras dinámicas. Su tamaño crece o decrece según las necesidades de forma dinámica.

Aprenderemos a trabajar con listas pero también mencionaremos otras estructuras como árboles o conjuntos.

4.1. Colecciones

El manejo de las estructuras de datos dinámicas es una tarea muy importante en el desarrollo de software. Sin embargo, su manejo, creando y manipulando directamente sus elementos y las referencias a ellos, podría considerarse un trabajo de bajo nivel.

Java incluye un conjunto de interfaces y clases genéricas, conocido como el **Java Collection Framework** (marco de trabajo de colecciones de Java), el cuál contiene estructuras de datos, interfaces y algoritmos pre empaquetados para manipular estructuras de datos tales como listas, pilas, colas, conjuntos y mapas clave – valor. **Podríamos considerarlo como la librería de las estructuras dinámicas**.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permiten manejar grupos de objetos, todo ello enfocado a potenciar la reusabilidad del software y facilitar las tareas de programación. Te parecerá increíble el tiempo que se ahorra empleando colecciones y cómo se reduce la complejidad del software usándolas adecuadamente. Las colecciones permiten almacenar y manipular grupos de objetos que, a priori, están relacionados entre sí (aunque no es obligatorio que estén relacionados, lo lógico es que si se almacenan juntos es porque tienen alguna relación entre sí), pudiendo trabajar con cualquier tipo de objeto.

Collection es la interfaz raíz en la jerarquía de colecciones. Es decir, define los métodos básicos que permitirán manejar todos los tipos de colecciones. A partir de Collection se derivan otras estructuras de datos como:

- List: define una colección que puede contener elementos duplicados.
- Set: define una colección que no puede contener duplicados
- Map: define una colección que asocia claves con valores y no puede contener claves duplicadas.

De estas a su vez se derivan otras pero como se ha dicho todas ellas compartirán la misma interfaz (los métodos definidos en la estructura Collection). A continuación se muestran las operaciones más importantes definidas por esta interfaz. Ten en cuenta que Collection es una interfaz genérica donde la **letra E** se utiliza para representar **cualquier clase** y al utilizarse se deberá sustituir por una clase concreta.

Método	Descripción
int size()	Devuelve el número de elementos de la colección.

boolean isEmpty()	Devuelve true si la colección está vacía.
boolean contains(Object objeto)	Devuelve true si la colección tiene el elemento pasado como parámetro.
boolean add(E elemento)	Permitirá añadir elementos a la colección. Devuelve true si se añade correctamente.
boolean remove(Object objeto)	Permitirá eliminar elementos de la colección. Devuelve true si se borra correctamente.
Iterator<E> iterator()	Permitirá crear un iterador para recorrer los elementos de la colección. Esto se ve más adelante, no te preocupes.
Object[] toArray()	Permite pasar la colección a un array de objetos tipo Object
void clear()	Vacía la colección

En todos los tipos de colecciones en Java dispondremos de estos métodos comunes más otros particulares dependiendo de sus funcionalidades. Más adelante veremos cómo se usan estos métodos.

¿Qué colección elegimos a la hora de trabajar? En este curso nos centraremos en trabajar con la colección ArrayList.

Sin embargo, cuando vayamos a desarrollar una nueva aplicación es importante tener en cuenta los siguientes puntos:

- ¿Qué información queremos guardar?
- ¿Puede haber datos repetidos?
- ¿Es importante que los datos estén ordenados?

En función de las respuestas que demos, existirán colecciones que debido a su estructura y funcionamiento interno, serán más eficientes que otras y deberemos tenerlo en cuenta.

4.2 Listas

Las listas son una estructura de datos que nos recuerdan a los arrays pero que proporcionan mayor flexibilidad ya que podemos añadir y eliminar elementos sin preocuparnos por el tamaño de la lista. La lista crece según añadimos elementos y se reduce cuando los eliminamos sin que nosotros tengamos que hacer nada al respecto. De hecho, las listas son una de las estructuras de datos fundamentales que te vas a encontrar en programación.

Sus características son las siguientes:

- Pueden almacenar elementos duplicados. Si no queremos duplicados, hay que verificar manualmente que el elemento no esté en la lista antes de su inserción.
- Permiten acceso posicional. Es decir, podemos acceder a un elemento indicando su posición en la lista.
- Es posible buscar elementos en la lista y obtener su posición.
- Es posible la extracción de sublistas. Es decir se puede obtener una lista que contenga solo una parte de los elementos de forma muy sencilla.

Para ello, además de los métodos heredados de Collection, añade métodos que permiten esas funcionalidades.

Dentro de las listas podemos encontrar ArrayList y LinkedList. Las 2 son muy parecidas de manejar estando su diferencia en la estructura y funcionamiento interno. Cuando la lista vaya a cambiar frecuentemente, es decir cuando tengamos que introducir elementos nuevos y borrar otros de forma habitual, las LinkedList serán más eficientes. Para la mayoría de las soluciones sin embargo, los ArrayLists son suficientes y por ello vamos a centrar esta unidad en su manejo.

En Java, para las listas se dispone de una interfaz llamada java.util.List, y dos implementaciones básicas, java.util.LinkedList y java.util.ArrayList, con diferencias significativas entre ellas.

Los métodos de la interfaz List, que obviamente estarán en todas las implementaciones, y que permiten las operaciones anteriores son:

Método	Descripción
E get(int index)	Permite obtener un elemento partiendo de su posición (index).
E set(int index, E element)	Permite cambiar el elemento almacenado en una posición de la lista (index), por otro (element).

void add(int index, E element)	Otra versión del método add. Inserta un elemento (element) en la lista en una posición concreta (index), desplazando los siguientes elementos.
E remove(int index)	Otra versión del método remove. Elimina un elemento indicando su posición (index) en la lista.
boolean add(E element)	Añade un elemento al final de la lista
void clear()	Elimina todos los elementos de la lista
int size()	Devuelve el número de elementos de una lista
String toString()	Devuelve los elementos de una lista formateados como los arrays: ["hola", "kaixo", "agur", "adios"]
Object[] toArray()	Devuelve un array con los elementos de la lista en el mismo orden.

Fíjate que las listas conservan los métodos de las colecciones (add, clear, size...) y de la clase Object (toString) y añade otras más para posibilitar las funcionalidades descritas.

Al igual que los arrays, los elementos de una lista empiezan a numerarse por 0. Es decir, que el primer elemento de la lista es el 0.

Recuerda también que List es una interfaz genérica, podemos crear listas con elementos de cualquier clase, por lo que <E> se corresponderá con la clase usada para crear esa lista.

Hay otros métodos que para funcionar correctamente necesitan encontrar un elemento en la lista. Funcionan con los tipos básicos, enteros, double, String.. pero no con el resto de objetos:

Método	**Descripción**
E remove(Object o)	Elimina un elemento indicando de la lista.
int indexOf(Object o)	Permite conocer la primera aparición (índice) de un elemento. Si dicho elemento no está en la lista retornará -1.
boolean contains(Object o)	Devuelve true si el objeto indicado está en la lista, false en caso contrario

int lastIndexOf(Object o)	Permite conocer la última aparición (índice) de un elemento. Si dicho elemento no está en la lista retornará -1.

4.3 Manejo de Listas.

Y, ¿cómo se usan las listas? Pues para usar una lista haremos uso de su implementación ArrayList. El siguiente ejemplo muestra cómo usar un ArrayList pero valdría también para LinkedList.

No olvides importar las clases java.util.LinkedList y java.util.ArrayList según sea necesario para poder utilizar estas clases.

En este ejemplo se usan los métodos de acceso posicional a la lista:

```
ArrayList<String> t = new ArrayList<String>();  // Crea un ArrayList de cadenas de
caracteres.
t.add("hola");                                  // Añade el valor "hola" al final de la
lista.
t.add("Agur");                                  // Añade "Agur" al final de la lista.
t.add(1, "Adios");                 // Añade "Adios" en la posición 1 de la lista (la
segunda).
t.remove(0);                                    // Elimina el primer elementos de la lista.
t.set(1, "kaixo");                              // Modifica el valor del elemento 1

// Muestra los elementos de la lista.
for (int i = 0; i < t.size(); i++) {
   System.out.println("Elemento:" + t.get(i));
}

t.set(t.indexOf("kaixo"), "Agur");    // Busca el elemento "kaixo" y lo sustituye por
"Agur".
// Muestra los elementos de la lista mediante el método toString de las clases
System.out.println(t);
```

La lista ArrayList<E> representa una familia de listas que se diferencian en la clase dede elemento que almacenan. Usaremos ArrayList<String> para almacenar una lista de cadenas de caracteres. ArrayList<Punto> guardará diferentes elementos todos ellos de la clase Punto y ArrayList<Cliente> será una lista de Clientes.

Fíjate que nunca podemos declarar algo de la clase ArrayList<E>. Siempre tendremos que sustituir la E por la clase concreta que queremos utilizar.

En el ejemplo anterior, se realizan muchas operaciones, ¿cuál será el contenido de la lista al final? Pues será "Adios" y "Agur".

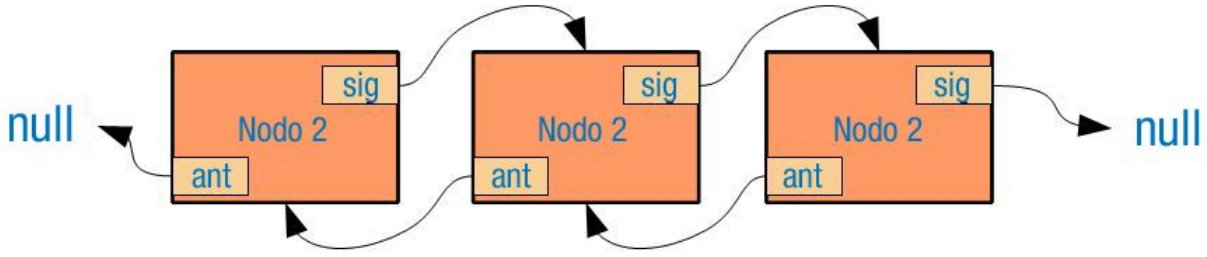
4.4 Cómo funcionan los diferentes tipos de listas

¿Y en qué se diferencia una LinkedList de una ArrayList? Los LinkedList utilizan listas doblemente enlazadas.

Las listas enlazadas sus elementos se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Cuando queremos añadir un elemento al final solo tenemos que enlazarlo al último elemento. Para eliminar un elemento de una lista, solo hay que "puentearlo". Es decir, hay que cambiar el enlace del elemento anterior para que conecte directamente con el siguiente, dejando el elemento a borrar fuera de la lista.

Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y también, información de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null para ambos casos.

```
public class Nodo
{
  public Nodo sig;
  public Nodo ant;
  public Object v;
}
```



En el caso de los ArrayList, éstos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente para nosotros, no nos enteramos cuando se produce, pero eso redunda en una diferencia de rendimiento notable dependiendo del uso.

Los ArrayList son más rápidos en cuanto a acceso a los elementos. Acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada que exige recorrer la lista. En cambio, eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada de cualquier tipo.

¿Y esto qué quiere decir? **Que si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada (LinkedList), pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (ArrayList)**.

LinkedList tiene otras ventajas que nos puede llevar a su uso. Implementa las interfaces java.util.Queue y java.util.Deque. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

Las colas, también conocidas como colas de prioridad, son una lista pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla? Pues igual. Se trata de que el que primero llega es el primero en ser atendido (FIFO). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (add y offer), sacar y eliminar el elemento más antiguo (poll), y examinar el elemento al principio de la lista sin eliminarlo (peek).

Las pilas, mucho menos usadas, son todo lo contrario a las listas. Una pila es igual que una montaña de hojas en blanco, para añadir hojas nuevas se ponen encima del resto, y para retirar una se coge la primera que hay, encima de todas. En las pilas el último en llegar es el primero en ser atendido. Para ello se proveen de tres métodos: meter al principio de la pila (push), sacar y eliminar del principio de la pila (pop), y examinar el primer elemento de la pila (peek, igual que si usara la lista como una cola).

4.5 Trabajando con colecciones

Cuando trabajemos con colecciones hay una serie de aspectos que es importante tener en cuenta:

1. ¿Cómo crear colecciones de datos de los tipos primitivos (int, double, char o boolean)?
2. ¿Cómo recorrer una colección para trabajar con sus elementos?
3. ¿Qué posibilidades ofrecen los métodos estáticos de las diferentes colecciones?
4. Diferencias entre objetos mutables e inmutables.

¿Qué pasa si creamos un ArrayList de números enteros? al compilar se produce un error "unexpected type". Es decir, que el tipo int no era uno de los tipos esperados.

Si repasamos lo que hemos visto sobre las colecciones, veremos que son estructuras de datos que pueden almacenar elementos de cualquier tipo de clase. No nos dice nada de los tipos primitivos pero ya vemos que no los admite. Las colecciones son clase genéricas y pueden almacenar cualquier objeto o tipo referenciado (como las clases, arrays...). Los tipos primitivos (int, double, char o boolean) no se pueden usar como tipo de dato en las colecciones.

Entonces ¿qué hacemos si necesitamos almacenar números pero en una colección no podemos almacenar tipos primitivos? La respuestas son las denominadas clases **Wrapper** o envoltorio.

Wrapper o envoltorio es el calificativo que se da a unas clases especiales cuyo único objetivo es almacenar los tipo primitivos como clases. Es decir, son clases que tendrán un único atributo que coincidirá con el tipo y el valor del tipo primitivo. De esta manera cuando necesitemos trabajar con objetos podremos seguir manejando números, letras y booleanos.

Además, se les ha añadido una serie de métodos que pueden resultar especialmente útiles.

Las clases más utilizadas son:

- Integer
- Double
- Boolean
- Char

Como el resto de clases tendrán sus constructores propios, ejemplo:

```
Integer x = new Integer(34);
Double y = new Double("3.58");
int z = 61;
Integer w = new Integer(z);
Boolean bo = new Boolean("false");
Character co = new Character('a');
```

Además, las clases Wrapper proporcionan los siguientes métodos útiles que seguramente hemos utilizado en anteriores ejercicios:

Ejemplos	Descripción

int a = x.intValue();
 	
double b = y.doubleValue();
 	
boolean c = bo.booleanValue());
 	
char d = co.charValue();
	Métodos de instancia para extraer el dato numérico del envoltorio. xxxValue()
	
	Permiten pasar de un objeto a un tipo primitivo. Se habla de "Unboxing"

int i = Integer.parseInt("123");
 	
double d = Double.parseDouble("34.89");
	Métodos estáticos de clase para crear números a partir de cadenas de caracteres. Xxx.parseXxx(String);
	
	Permiten leer texto por teclado o de un fichero y luego convertirlo a su tipo primitivo.

Integer x = Integer.valueOf("123"); Double y = Double.valueOf("34.89");	Métodos estáticos de clase para crear envoltorios de números a partir de cadenas de caracteres. Xxx.valueOf(String); Pasamos de un texto a un objeto de una de las clases envoltorio. Se habla de "Boxing"

Ya conocemos las clases envoltorio. ¿Cómo las usamos para crear colecciones?

Para crear una colección, las usaremos igual que lo hemos hecho con cualquier otra clase en Java:

```
ArrayList<Integer> ListaInt = ArrayList<Integer>;
ArrayList<Double> ListaDouble = ArrayList<double>;
ArrayList<Char> ListaChar = ArrayList<Char>;
```

A la hora de añadir y leer elementos podemos utilizar los métodos vistos en el apartado anterior:

```
// Añadir un elemento. Creamos un objeto Integer y lo añadimos
Integer x = new Integer(34);
listaInt.add(x);
// Leer un valor. Obtenemos un objeto Integer y lo pasamos a int.
x = listaInt.get(0);
int num = x.intValue();
```

La buena noticia es que a partir de la versión 5 de Java este proceso lo realiza Java automáticamente y podemos escribir:

```
ArrayList<Integer> ListaInt = ArrayList<Integer>;
// Añadir un elemento. Añadimos directamente un int y Java lo convierte en
un objeto Integer
listaInt.add(34);

// Leer un valor. Obtenemos un objeto Integer y Java lo pasa a int para
poderlo almacenar
int num = listaInt.get(0);
```

De esta manera, solo tendremos que usar la clase envoltorio para crear la colección. En el resto de operaciones podemos trabajar con los tipos primitivos directamente y Java se encargará de realizar las conversiones necesarias.

4.6 Patrones

Hemos visto que las clases envoltorio permiten convertir un texto a int o double mediante los métodos parseInt y parseDouble. Pero, ¿podríamos **comprobar si esa cadena de caracteres es realmente un número entero** antes de convertirla para evitar que se produzca una excepción?

Para ello, podríamos utilizar las **expresiones regulares** o regex de Java.

Vamos a ver las características que tienen los números int. Son números de 32-bit que van del -2^{31} al $2^{31}-1$. Es decir toman valores comprendidos entre el -2147483648 y el 2147483647. Identificar todos estos valores con una expresión regular es difícil pero si los limitamos a valores entre -999999999 y +999999999 la cosa se simplifica. Estaríamos descartando algunos números enteros pero evitaríamos excepciones.

¿Cómo sería la expresión regular para ese rango de valores?

1. Puede tener signo o no. Si lo tiene siempre será -. La expresión sería: -? que significa que el signo - puede aparecer o no.
2. Todos los dígitos pueden tener valores entre 0 y 9. La expresión sería: [0-9] o \\d que significa que los caracteres que pueden aparecer en la cadena son los dígitos del 0 al 9.
3. Siempre debe aparecer al menos un dígito y como máximo 9. La expresión sería: {1,9} que significa que un carácter puede aparecer entre 1 y 9 veces.

Si las juntamos, la expresión completa será:

```
-?[0-9]{1,9} ó -?\\d{1,9}
```

Es importante **no dejar ningún espacio en blanco**, ya que producirá un error al ejecutarse.

La forma de aplicar esta expresión a una cadena de caracteres será:

```
Scanner leerDatos = new Scanner(System.in);
// Pide palabras hasta que el texto introducido cumpla con el patrón
String dato = null;
Matcher comparaFormato = null;
Pattern formatoInt = Pattern.compile("-?[0-9]{1,9}");  // Genera la
expresión regular para enteros
do {
       System.out.println("Introduce un entero: ");
       dato = leerDatos.next();
       comparaFormato = formatoInt.matcher(dato);
} while (!comparaFormato.matches());
// Convierte el texto a un int
int numero = Integer.parseInt(dato);
System.out.println(numero + " es un entero");
```

Las expresiones regulares las podemos usar también para comprobar que el texto introducido es un DNI válido, un correo electrónico, un teléfono o una fecha. Conviene consultar si existe la expresión que queremos usar antes de empezar a diseñar una. Hay muchos ejemplos en Internet .

4.7 Recorriendo una colección

Para recorrer un array hemos usado el siguiente código basado en un bucle for:

```
String[] palabras = {"Hola", "Kaixo", "Hello"};
for (int i = 0; i < palabras.length; i++) {
   System.out.println("Elemento: " + palabras[i]);
}
System.out.println(Arrays.toString(palabras));
```

Para ello, es indispensable que los elementos de la estructura de datos se referencien mediante un índice.

Una estructura parecida nos puede servir también para las listas pero no para el resto de colecciones.

```
ArrayList<String> lista = new ArrayList<String>();
lista.add("Hola");
lista.add("Kaixo");
lista.add("Hello");
for (int i = 0; i < lista.size(); i++) {
     System.out.println("Elemento: " + lista.get(i));
}
System.out.println(lista);
```

Por ello, vamos a ver otras 2 maneras de recorrer colecciones expresamente diseñadas para ellas. Estas son:

1. El bucle for-each
2. La clase Iterator

BUCLE FOR-EACH

El bucle "for-each" o bucle "para cada", se parece mucho a un bucle for con la diferencia de que no hace falta una variable i que utilizamos para gestionar los índices.

Existe a partir de Java 5 y en principio puede resultar más cómoda y compacta que el uso de la clase Iterator. Sin embargo, como veremos, tendrá sus limitaciones y en algunos casos deberemos recurrir obligatoriamente a los iteradores.

En el siguiente código se usa un bucle for-each, en el que el texto va tomando los valores de todos los elementos almacenados en el conjunto hasta que llega al último. En este caso, no se necesita ningún índice para recorrer la estructura de datos. La sentencia for-each se encarga de pasar por cada uno de los elementos y guardarlo en texto. Fíjate que se llama for-each pero solo se escribe for:

```
for (String texto : conjunto) {
    System.out.println("Elemento almacenado: " + texto);
}
```

Como ves la estructura for-each es muy sencilla: la palabra for seguida de "(**tipo nombre** : **estructura**)" y el cuerpo del bucle.

- **tipo** es el tipo de dato que se ha utilizado para crear la estructura de datos. Puede ser una colección pero también un Array.
- **nombre** es el nombre del objeto o la variable donde se almacenará cada elemento de la estructura.
- **estructura** es el nombre de la colección en sí.

Los bucles for-each se pueden usar para todas las colecciones y también para los arrays pero no permiten modificar la colección dentro del bucle. Es decir, obtenemos el valor de cada elemento, podemos trabajar con él pero no podríamos borrarlo. Para ello, habría que recurrir a la clase Iterator.

TAREA:

Codificar en java un programa que pregunte al usuario 5 números diferentes (almacenándolos en un ArrayList), y que después calcule la suma de los mismos (usando un bucle for-each).

LA CLASE ITERATOR

¿Qué son los iteradores? Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura.

Cuando queremos modificar una colección mientras la estamos recorriendo, en concreto cuando queremos borrar el último elemento que hemos procesado, necesitaremos utilizar iteradores. Además, los podemos encontrar en programas de versiones antiguas de Java, anteriores a la aparición del bucle for-each.

Ahora la pregunta es, ¿cómo se crea un iterador? Pues creando un objeto de la clase Iterator a partir de la colección que queremos recorrer. Es decir, invocando el método "iterator()" de cualquier colección.

Veamos un ejemplo en el que t es una colección cualquiera:

```
Iterator<String> it = t.iterator();
```

Fijate que se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo "<String>" después de Iterator). Esto es porque los iteradores son también clases genéricas (podemos tener iteradores de cualquier clase), y es necesario especificar el tipo base que contendrá el iterador. Sino se especifica el tipo base del iterador, igualmente nos permitiría recorrer la colección, pero retornará objetos tipo Object (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

Para recorrer y gestionar la colección, el iterador ofrece tres métodos básicos:

- boolean hasNext(). Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.
- E next(). Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (NoSuchElementException para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.
- remove(). Elimina de la colección el último elemento retornado en la última invocación de next (no es necesario pasarselo por parámetro). Cuidado, si next no ha sido invocado todavía, saltará una incómoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy parecida a como leemos datos por teclado y un fichero. Un bucle mientras (while) con la condición hasNext() nos permite hacerlo del siguiente modo:

```
// Mientras que haya otro elemento, seguiremos en el bucle.
while (it.hasNext()){

  String t = it.next();  //Recogemos el siguiente elemento.

  //Si el elemento coincide, eliminar el elemento extraído de la lista.
  if (t.equals("borrar")) {
      it.remove();
  }
}
```

Las listas permiten acceso posicional a través de los métodos get y set, y acceso secuencial a través de iteradores, ¿cuál es para tí la forma más cómoda de recorrer todos los elementos? ¿Un acceso posicional a través un bucle "for (int i = 0; i < lista.size(); i++)" o un acceso secuencial usando un bucle "while (iterador.hasNext())"?

¿Qué inconvenientes tiene usar los iteradores sin especificar el tipo de objeto? En el siguiente ejemplo, se genera una lista con los números del 0 al 10. De la lista, se eliminan aquellos que son pares y solo se dejan los impares. En el ejemplo de la izquierda se especifica el tipo de objeto del iterador, en el ejemplo de la derecha no, observa el uso de la conversión de tipos en la línea 6.

Ejemplo indicando el tipo de objeto de iterador

```
ArrayList <Integer> lista = new ArrayList<Integer>();
for (int i = 0; i < 10; i++) {
    lista.add(i);
}
Iterator<Integer> it = lista.iterator();
while (it.hasNext()) {
    Integer t = it.next();
    if (t % 2 == 0) {
        it.remove();
    }
}
```

Ejemplo no indicando el tipo de objeto del iterador

```
ArrayList <Integer> lista = new ArrayList<Integer>();
for (int i = 0; i < 10; i++)  {
    lista.add(i);
}
Iterator it = lista.iterator();
while (it.hasNext()) {
    Integer t = (Integer) it.next();
    if (t % 2 == 0) {
        it.remove();
    }
}
```

Un iterador es seguro porque está pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden repercutir en errores de software. Pero realmente se convierte en inseguro cuando es necesario hacer la operación de conversión de tipos. Si la colección no contiene los objetos esperados, al intentar hacer la conversión, saltará una incómoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.

IMPORTANTE: Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método remove del iterador y no el de la colección. Si eliminas los elementos utilizando el método remove de la colección, mientras estás dentro

de un bucle de iteración, o dentro de un bucle for-each, se producirán errores ya que el método remove del iterador elimina el elemento de dos sitios: de la colección y del iterador en sí (que mantiene interiormente información del orden de los elementos). Si usas el método remove de la colección, la información solo se elimina de un lugar, de la colección.

4.8 Clase Collections

La clase Collections ofrece algunas operaciones adicionales a **todos** los tipos de colecciones. Algunos de los más utilizados con listas son los siguientes:

Operación	Descripción	Ejemplos
Desordenar una lista.	Desordena una colección, este método no está disponible para arrays.	Collections.shuffle(lista);
Rellenar una lista.	Rellena una colección copiando el mismo valor en todos los elementos de la colección. Útil para reiniciar una colección.	Collections.fill(lista, elemento);
Dar la vuelta.	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	Collections.reverse(lista);
Ordenar una lista	Ordena la lista en orden ascendente según el orden natural de los elementos	Collections.sort(lista, elemento);
Búsqueda binaria.	Permite realizar búsquedas rápidas en un una colección ordenada. Es necesario que la colección esté ordenada, si no lo está, la búsqueda no tendrá éxito.	Collections.binarySearch(lista, elemento);

Convertir un array a lista.	Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es ArrayList ni LinkedList), solo se especifica que retorna una lista que implementa la interfaz java.util.List.	List lista = Arrays.asList(array); Si el tipo de dato almacenado en el array es conocido (Integer por ejemplo), debemos especificar el tipo de objeto de la lista: List<Integer> lista = Arrays.asList(array);
Copiar el contenido de una lista	Copia el contenido de una lista origen a una lista destino.	Collections.copy(ArrayList destino, ArrayList origen);

Otra operación útil es dividir una cadena en partes. Cuando una cadena está formada internamente por trozos de texto claramente delimitados por un separador (una coma, un punto y coma o cualquier otro), es posible dividir la cadena y obtener cada uno de los trozos de texto por separado en un array de cadenas. Para poder realizar esta operación, usaremos el método split de la clase String. El delimitador o separador es una expresión regular, único argumento del método split, y puede ser obviamente todo lo complejo que sea necesario:

```
String texto = "Z,B,A,X,M,O,P,U";
String []partes = texto.split(",");
Arrays.sort(partes);
```

En el ejemplo anterior la cadena texto contiene una serie de letras separadas por comas. La cadena se ha dividido con el método split, y se ha guardado cada carácter por separado en un array. Después se ha ordenado el array.

4.8 Conjuntos de pares clave/valor

¿Cómo almacenarías los datos de un diccionario? Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existen precisamente los **asociativos**. Un tipo de array asociativo son los mapas o diccionarios, que permiten almacenar pares de valores conocidos como clave y valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

Los **mapas utilizan clases genéricas** para dar extensibilidad y flexibilidad, y permiten definir un tipo base para la clave, y otro tipo diferente para el valor. En java existen 2 clases Map:

- HashMap
- TreeMap

Bases para la comparación	**HashMap**	**TreeMap**
Básico	HashMap no mantiene el orden de inserción.	TreeMap mantiene el orden de inserción.
Estructura de datos	HashMap usa Hash Table como una estructura de datos subyacente.	TreeMap utiliza el árbol rojo-negro como estructura de datos subyacente.
Claves nulas y valores	HashMap permite clave nula.	TreeMap no permite la clave nula, pero permite valores.
Extiende e implementa	HashMap extiende la clase AbstractMap e implementa la interfaz Map.	TreeMap extiende la clase AbstractMap e implementa la interfaz SortedMap y NavigableMap.
Actuación	HashMap funciona más rápido.	TreeMap en comparación con HashMap funciona más lento.

Veamos un ejemplo de como crear un mapa, que es extensible a los otros dos tipos de mapas:

```
HashMap<String, Integer> t = new HashMap<String, Integer>();
```

El mapa anterior permite usar cadenas como llaves y almacenar de forma asociada a cada llave, un número entero. Veamos los métodos principales de la interfaz Map, disponibles en todas las implementaciones. En los ejemplos, V es el tipo base usado para el valor y K el tipo base usado para la llave.

Métodos principales de los mapas:

Método.	Descripción.
V put(K key, V value);	Inserta un par de objetos llave (key) y valor (value) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará null.
V get(Object key);	Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará null.
V remove(Object key);	Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe.
boolean containsKey(Object key);	Retornará true si el mapa tiene almacenada la llave pasada por parámetro, false en cualquier otro caso.
boolean containsValue(Object value);	Retornará true si el mapa tiene almacenado el valor pasado por parámetro, false en cualquier otro caso.
int size();	Retornará el número de pares llave y valor almacenado en el mapa.
boolean isEmpty();	Retornará true si el mapa está vacío, false en cualquier otro caso.
void clear();	Vacía el mapa.

ITERADORES CON MAPS

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método entrySet que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien, el método keySet para generar un conjunto con las llaves existentes en el mapa. Veamos como sería para el segundo caso, el más sencillo:

```
HashMap<Integer, Integer> mapa = new HashMap<Integer, Integer>();

for (int i = 0; i < 10; i++) {
   mapa.put(i, i);                 // Insertamos datos de prueba en el mapa.
}
 // Recorremos el conjunto generado por keySet, contendrá las llaves.
for (Integer llave : mapa.keySet()) {
   Integer valor = mapa.get(llave);  // Para cada llave, accedemos a su valor si es
necesario.
}
```

5. Expresiones regulares

Si nos fijamos, los números de DNI y los de NIE tienen una estructura fija: X1234567Z (en el caso del NIE) y 1234567Z (en el caso del DNI). Ambos siguen un **patrón** que podría describirse como: una letra inicial opcional (solo presente en los NIE), seguida de una secuencia numérica y finalizando con otra letra. ¿Fácil no?

Pues esta es la función de las expresiones regulares: **permitir comprobar si una cadena sigue o no un patrón preestablecido**. Las expresiones regulares son un mecanismo para describir esos patrones, y se construyen de una forma relativamente sencilla. Existen muchas librerías diferentes para trabajar con expresiones regulares, y casi todas siguen, más o menos, una sintaxis similar, con ligeras variaciones. Dicha sintaxis nos permite indicar el patrón de forma cómoda, como si de una cadena de texto se tratase, en la que determinados símbolos tienen un significado especial. Por ejemplo "[01]+" es una expresión regular que permite comprobar si una cadena conforma un número binario.

Veamos cuáles son las reglas generales para construir una expresión regular:

- Podemos indicar que una cadena contiene un conjunto de símbolos fijo, simplemente poniendo dichos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán un carácter de escape como veremos más adelante. Por ejemplo, el patrón "aaa" admitirá cadenas que contengan tres aes.
- "[xyz]". Entre corchetes podemos indicar opcionalidad. Solo uno de los símbolos que hay entre los corchetes podrá aparecer en el lugar donde están los corchetes. Por ejemplo, la expresión regular "aaa[xy]" admitirá como válidas las cadenas "aaax" y la

cadena "aaay". **Los corchetes representan una posición de la cadena que puede tomar uno de varios valores posibles.**

- "[a-z]" "[A-Z]" "[a-zA-Z]". Usando el guión y los corchetes podemos indicar que el patrón admite cualquier carácter entre la letra inicial y la final. Es importante que sepas que se diferencia entre letras mayúsculas y minúsculas, no son iguales de cara a las expresiones regulares.
- "[0-9]". Y nuevamente, usando un guión, podemos indicar que se permite la presencia de un dígito numérico entre 0 y 9, cualquiera de ellos, pero solo uno.

Con las reglas anteriores podemos indicar el conjunto de símbolos que admite el patrón y el orden que deben tener. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, entonces la cadena no encajará con el patrón. Veamos ahora cómo indicar repeticiones:

- "a?". Usaremos el interrogante para indicar que un símbolo puede aparecer una vez o ninguna. De esta forma la letra "a" podrá aparecer una vez o simplemente no aparecer.
- "a*". Usaremos el asterisco para indicar que un símbolo puede aparecer una vez o muchas veces, pero también ninguna. Cadenas válidas para esta expresión regular serían "", "aa", "aaa" o "aaaaaaaa".
- "a+". Usaremos el símbolo de suma para indicar que otro símbolo debe aparecer al menos una vez, pudiendo repetirse cuantas veces quiera.
- "a{1,4}" . Usando las llaves, podemos indicar el número mínimo y máximo de veces que un símbolo podrá repetirse. El primer número del ejemplo es el número 1, y quiere decir que la letra "a" debe aparecer al menos una vez. El segundo número, 4, indica que como máximo puede repetirse cuatro veces.
- "a{2,}". También es posible indicar solo el número mínimo de veces que un carácter debe aparecer (sin determinar el máximo), haciendo uso de las llaves, indicando el primer número y poniendo la coma (no la olvides).
- "a{5}". A diferencia de la forma anterior, si solo escribimos un número entre llaves, sin poner la coma detrás, significará que el símbolo debe aparecer un número exacto de veces. En este caso, la "a" debe aparecer exactamente 5 veces.
- "[a-z]{1,4}[0-9]+". Los indicadores de repetición se pueden usar también con corchetes, dado que los corchetes representan, básicamente, un símbolo. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

Por ejemplo, una expresión regular que permite verificar si una cadena contiene un DNI o un NIE es la siguiente: "[XYxy]?[0-9]{1,9}[A-Za-z]"; aunque no es la única solución.

¿Y cómo uso las expresiones regulares en un programa? Pues de una forma sencilla. Para su uso, Java ofrece las clases Pattern y Matcher contenidas en el paquete java.util.regex.*. La clase Pattern se utiliza para procesar la expresión regular y "compilarla",

lo cual significa verificar que es correcta y dejarla lista para su utilización. La clase Matcher sirve para comprobar si una cadena cualquiera sigue o no un patrón. Veámoslo con un ejemplo:

```
Pattern p = Pattern.compile("[01]+");
Matcher m = p.matcher("00001010");
if (m.matches()) {
    System.out.println("Si, contiene el patrón");
} else {
   System.out.println("No, no contiene el patrón");
}
```

En el ejemplo, el método estático compile de la clase Pattern permite crear un patrón, dicho método compila la expresión regular pasada por parámetro y genera una instancia de Pattern (*p* en el ejemplo). El patrón *p* podrá ser usado múltiples veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método matcher, el cual combina el patrón con la cadena de entrada y genera una instancia de la clase Matcher (m en el ejemplo). La clase Matcher contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- m.matches(). Devolverá true si toda la cadena (de principio a fin) encaja con el patrón o false en caso contrario.
- m.lookingAt(). Devolverá true si el patrón se ha encontrado al principio de la cadena. A diferencia del método matches(), la cadena podrá contener al final caracteres adicionales a los indicados por el patrón, sin que ello suponga un problema.
- m.find(). Devolverá true si el patrón existe en algún lugar la cadena (no necesariamente toda la cadena debe coincidir con el patrón) y false en caso contrario, pudiendo tener más de una coincidencia. Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos m.start() y m.end(), para saber la posición inicial y final donde se ha encontrado.

Veamos algunas construcciones adicionales que pueden ayudarnos a especificar expresiones regulares más complejas:

- "[^abc]". El símbolo "^", cuando se pone justo detrás del corchete de apertura, significa "negación". La expresión regular admitirá cualquier símbolo diferente a los puestos entre corchetes. En este caso, cualquier símbolo diferente de "a", "b" o "c".
- ".". El punto simboliza cualquier carácter.
- "\\d". Un dígito numérico. Equivale a "[0-9]".
- "\\D". Cualquier cosa excepto un dígito numérico. Equivale a "[^0-9]".
- "\\s". Un espacio en blanco (incluye tabulaciones, saltos de línea y otras formas de espacio).
- "\\S". Cualquier cosa excepto un espacio en blanco.

- "\\w". Cualquier carácter que podrías encontrar en una palabra. Equivale a "[a-zA-Z_0-9]".

¿Te resultan difíciles las expresiones regulares? Al principio siempre lo son, pero no te preocupes. Hasta ahora has visto como las expresiones regulares permiten verificar datos de entrada, permitiendo comprobar si un dato indicado sigue el formato esperado: que un DNI tenga el formato esperado, que un email sea un email y no otra cosa, etc. Pero ahora vamos a dar una vuelta de tuerca adicional.

Los paréntesis, de los cuales no hemos hablado hasta ahora, tienen un significado especial, permiten indicar repeticiones para un conjunto de símbolos, por ejemplo: "(#[01]){2,3}". En el ejemplo anterior, la expresión "#[01]" admitiría cadenas como "#0" o "#1", pero al ponerlo entre paréntesis e indicar los contadores de repetición, lo que estamos diciendo es que la misma secuencia se tiene que repetir entre dos y tres veces, con lo que las cadenas que admitiría serían del estilo a: "#0#1" o "#0#1#0".

Pero los paréntesis tienen una función adicional, y es la de permitir definir grupos. Un grupo comienza cuando se abre un paréntesis y termina cuando se cierra el paréntesis. Los grupos permiten acceder de forma cómoda a las diferentes partes de una cadena cuando esta coincide con una expresión regular. Lo mejor es verlo con un ejemplo (seguro que te resultará familiar):

```
Pattern p = Pattern.compile("([XY]?)([0-9]{1,9})([A-Za-z])");
Matcher m = p.matcher("X123456789Z Y00110011M 999999T");
while (m.find()) {
   System.out.println("Letra inicial (opcional):" + m.group(1));
   System.out.println("Número:" + m.group(2));
   System.out.println("Letra NIF:" + m.group(3));
}
```

Usando los grupos, podemos obtener por separado el texto contenido en cada uno de los grupos. En el ejemplo anterior, en el patrón hay tres grupos: uno para la letra inicial (grupo 1), otro para el número del DNI o NIE (grupo 2), y otro para la letra final o letra NIF (grupo 3). Al ponerlo en grupos, usando el método group(), podemos extraer la información de cada grupo y usarla a nuestra conveniencia.

Ten en cuenta que el primer grupo es el 1, y no el 0. Si pones m.group(0) obtendrás una cadena con toda la ocurrencia o coincidencia del patrón en la cadena, es decir, obtendrás la secuencia entera de símbolos que coincide con el patrón.

En el ejemplo anterior se usa el método find, este buscará una a una, cada una de las ocurrencias del patrón en la cadena. Cada vez que se invoca, busca la siguiente ocurrencia del patrón y devolverá true si ha encontrado una ocurrencia. Si no encuentra en una iteración ninguna ocurrencia es porque no existen más, y retornará false, saliendo del

bucle. Esta construcción while es muy típica para este tipo de métodos y para las iteraciones, que veremos más adelante.

Lo último importante de las expresiones regulares que debes conocer son las secuencias de escape. Cuando en una expresión regular necesitamos especificar que lo que tiene que haber en la cadena es un paréntesis, una llave, o un corchete, tenemos que usar una secuencia de escape, dado que esos símbolos tienen un significado especial en los patrones. Para ello, simplemente antepondremos "\\" al símbolo. Por ejemplo, "\\(" significará que debe haber un paréntesis en la cadena y se omitirá el significado especial del paréntesis. Lo mismo ocurre con "\\[", "\\]", "\\)", etc. Lo mismo para el significado especial del punto, éste tiene un significado especial, salvo que se ponga "\\.", que pasará a significar "un punto" en vez de "cualquier carácter". **La excepción son las comillas, que se pondrían con una sola barra: "\"".**