ikastaroak.ulhi.net

PROG07.- Estructuras dinámicas de almacenamiento.

68-87 minutos

Ana ha recibido un pequeño encargo de parte de su tutora, **María**. Se trata de que realice un pequeño programita, muy sencillo pero fundamental.

-Hola **Ana** -dice **María**-, hoy tengo una tarea especial para ti.

-¿Sí? -responde **Ana**-. Estoy deseando, últimamente no hay nada que se me resista, llevo dos semanas en racha.

-Bueno, quizás esto se te resista un poco más, es fácil, pero tiene cierta complicación. Un cliente para el que hicimos una aplicación, nos ha dicho si podemos ayudarle. El cliente tiene una aplicación para gestionar los pedidos que recibe de sus clientes. Normalmente, recibe por correo electrónico los pedidos en un formato concreto que todos sus clientes llevan tiempo usando. El cliente suele transcribir el pedido desde el correo electrónico a la aplicación, copiando dato a dato, pero nos ha preguntado si es posible que copie todo el pedido de golpe, y que la aplicación lo procese, detectando posibles errores en el pedido.

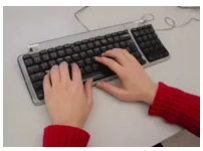
-¿Qué? -dice **María** con cierta perplejidad.

-Me alegra que te guste -dice María esbozando una sonrisa picara-, sé que te gustan los retos.

-Pero, ¿eso cómo se hace? ¿Cómo compruebo yo si el pedido es válido? ¿Y si el cliente ha puesto un producto que no existe?

-No mujer, se trata de comprobar si el pedido tiene el formato correcto, y transformarlo de forma que se genere un documento XML con los datos del pedido. Un documento XML luego se puede incorporar fácilmente a los otros pedidos que tenga el cliente en su base de datos. De la verificación de si hay algún producto que no existe, o de si hay alguna incoherencia en el pedido se encarga otra parte del software, que ya he realizado yo.

-Bueno -dice María justo después de resoplar tres veces seguidas-, empieza por contarme que es el formato XML.



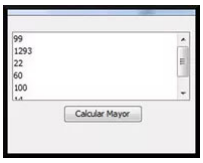
Cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué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

Ya hemos trabajo con arrays y con clases, pero, a veces, los datos tienen estructuras aún más complejas, y son necesarias soluciones adicionales.

En esta UD aprenderemos esas soluciones adicionales que consisten básicamente en la capacidad de poder manejar varios datos del mismo o diferente tipo de forma dinámica y flexible. Aunque hablaremos de diferentes estructuras de datos dinámicas, nos centraremos en manejar las listas y en concretos los ArrayLists.



¿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.

Como ya hemos visto, las estructuras de almacenamiento se pueden clasificar atendiendo a si pueden almacenar datos de diferente tipo o si solo pueden almacenar datos de un solo tipo pero también en función de si pueden o no cambiar de tamaño de forma dinámica, de acuerdo a las necesidades.

Hemos aprendido a 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 hemos aprendido a definir clases. Nos permiten agrupar diferentes tipos de datos pero la estructura es fija una vez creada.

Por ello, en esta UD nos vamos a centrar en 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.

Pregunta

El tamaño de las estructuras de almacenamiento siempre se determina en el momento de la creación. ¿Verdadero o falso?

A **Ana** las listas siempre se le han atragantado, por eso no las usa. Después de darle muchas vueltas, ha pensado que no le queda más remedio y que tendrá que usarlas para almacenar los artículos del pedido. Además, ha concluido que es la mejor forma de gestionar un grupo de objetos, aunque sean del mismo tipo.

No sabe si lo más adecuado es usar una lista u otro tipo de colección, así que ha decido revisar todos los tipos de colecciones disponibles en Java, para ver cuál se adecua mejor a sus necesidades.

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 los métodos definidos en la estructura Collection que explican a continuación.

Las colecciones en Java parten de una serie de interfaces básicas. Cada interfaz define un **modelo** de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados, por lo que es necesario conocerlas.

La **interfaz inicial**, a través de la cual se han construido el resto de colecciones, es la interfaz `java.util.Collection`, que define las **operaciones comunes a todas las colecciones** derivadas.

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 como se usan estos métodos.

Si quieres saber más sobre la interfaz Collections puedes consultar la página de Oracle:

Interfaz Collections

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.

En el siguiente artículo podéis comparar las situaciones en las que se usan cada una de las principales colecciones.

Introducción a Colecciones en Java

Juan se queda pensando después de que Ana le preguntara si sabía los tipos de colecciones que había en Java. Obviamente no lo sabía, son muchos tipos, pero ya tenía una respuesta preparada:

—Bueno, sea lo que sea, siempre puedes utilizar una lista para almacenar lo que sea. Yo siempre las uso, pues te permiten almacenar cualquier tipo de objeto, extraer uno de las lista sin tener que recorrerla entera, buscar si hay o no un elemento en ella, de forma cómoda. Son para mi el mejor invento desde la rueda —dijo Juan.

—Ya, supongo, pero hay dos tipos de listas que me interesan, `LinkedList` y `ArrayList`, ¿cuál es mejor? ¿Cuál me conviene más? —respondió Ana.

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 eliminarnos sin que nosotros tengamos que hacer nada al respecto. De echo, las listas son una de las estructuras de datos fundamentales que te vas ha encontran 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 UD 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 elementos siguientes.
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 elmento 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.

Pregunta

Si M es una lista de números enteros, ¿sería correcto poner "`M.add(M.size() - 1,3);`"?

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.

En la siguiente UD veremos de qué dependen y cómo hacerlos funcionar.

Y, ¿cómo se usan las listas?

Pues para usar una lista haremos uso de su implementación `ArrayList`. El siguiente ejemplo muestra como 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

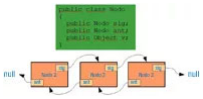
tendremos que sustituir la E por la clase concreta que queremos utilizar.

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

Completa con el número que falta.

Dado el siguiente código:

```
ArrayList<Integer> t = new ArrayList<Integer>();
t.add(t.size() + 1);
t.add(t.size() + 1);
int suma = t.get(0) + t.get(1);
```



¿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.

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 a 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 que 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).

Pregunta

```
Dada la siguiente lista, usada como si fuera una cola de prioridad, ¿cuál es la letra que se mostraría por la pantalla tras su ejecución?
LinkedList<String> tt = new LinkedList<String>();
tt.offer("A"); tt.offer("B"); tt.offer("C");
System.out.println(tt.poll());
```

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 ente objetos mutables e inmutables

A continuación, profundizaremos en estos aspectos.

¿Habéis probado a crear un ArrayList de números enteros? ¿Ha sido posible?

Seguramente al compilar se ha producido un error "unexpected type". Es decir, que el tipo int no era uno de los tipos esperados.

Si repasáis lo que hemos visto sobre las colecciones, veréis 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 boleanos.

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

Las clases que necesitaremos para los tipos primitivos seras:

- Integer
- Double
- Boolean
- Char

Como el resto de clases tendrán sus constructores propios:

```
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 interesantes:

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"

Las clases envoltorio y en especial los métodos parseXxx, son **muy utilizados para leer datos tanto de ficheros como desde el teclado**. Permiten leer todos los datos como texto con next, comprobar que cumplen un patrón concreto y después convertirlos al tipo adecuado.

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 version 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 yJava 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.

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 tiene 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 límitamos 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 ajecutarse.

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 télefono o una fecha. Conviene consultar si existe la expresión que queremos usar antes de empezar a diseñar una. Hay muchos ejemplos en Internet .

Para más información sobre las expresiones regulares en Java puedes consultar el anexo II .

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 indispinsable que los elementos de la estructura de datos se referencien mediante un índice.

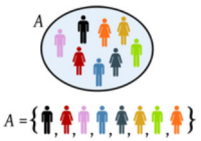
Una estructura parecida nos puede servir también para las listas pero no para el resto de colecciones.

```
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



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 de inicialización.

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 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);
}
```

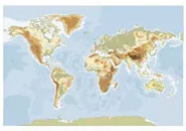
Como ves la estructura for-each es muy sencilla: la palabra `for` seguida de "(tipo nombre : estructura)" y el cuerpo del bucle.

- 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.

Realiza un pequeño 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).



Juan se acerco a la mesa de Ana y le dijo:

—María me ha contado la tarea que te ha encomendado y he pensado que quizás te convendría usar mapas en algunos casos. Por ejemplo, para almacenar los datos del pedido asociados con una etiqueta: nombre, dirección, fecha, etc. Así creo que te será más fácil generar luego el XML.

—La verdad es que pensaba almacenar los datos del pedido en una clase especial llamada Pedido. No tengo ni idea de que son los mapas -dijo Ana-, supongo que son como las listas, ¿tienen iteradores?

—Según me ha contado María, no necesitas hacer tanto, no es necesario crear una clase específica para los pedidos. Y respondiendo a tu pregunta, los mapas no tienen iteradores, pero hay una solución... te explico.

¿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 incomoda 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:

```
while (it.hasNext()) {        // Mientras que haya otro elemento, seguiremos en el bucle.
   String t = it.next();      // Recogemos el siguiente elemento.
   if (t.equals("borrar")) {
       it.remove();           // Si el número es par, eliminar el elemento extraído de la lista.
   }
}
```

¿Qué elementos contendría la lista después de ejecutar el bucle? Efectivamente, todas las palabras menos las que coinciden con "borrar".

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.

Comparación de usos de los iteradores, con o sin conversión de tipos.

Ejemplo indicando el tipo de objeto de iterador	**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<Integer> it = lista.iterator();`
`while (it.hasNext()) {`
`  Integer t = it.next();`
`  if (t % 2 == 0) {`
`  it.remove();`
`  }`
`}`
	`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 esta 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

incomoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.

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, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

Los problemas son debidos a 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.

¿Qué ofrece la `clase java.util.Collections` de Java? Igual que ocurria con los Arrays, en Java existe la clase Collections que ofrece algunas operaciones adicionales a todos los tipos de colecciones. A continuación, se muestran algunos de esos métodos:

Operaciones adicionales sobre listas y arrays.

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 ascente 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	`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:

retornado (no es `ArrayList` ni `LinkedList`), solo se especifica que retorna una lista que implementa la interfaz `java.util.List`.

```
List<Integer> lista =
Arrays.asList(array);
```

Otra operación que no se ha visto hasta ahora es la 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. Es una operación sencilla, pero dado que es necesario conocer el funcionamiento de los arrays y de las expresiones regulares para su uso, no se ha podido ver hasta ahora. 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. ¡Increíble lo que se puede llegar a hacer con solo tres líneas de código!

En este caso también, los métodos que necesitan buscar y comparar elementos como sort o binarySearch, solo funcionarán con los tipos de datos básicos.

En la siguiente UD aprenderemos cómo conseguir que funcionen con cualquier objeto.

Un objeto inmutable es aquel cuyo estado no se puede cambiar una vez construído. Todos sus atributos han sido definidos como final y utiliza copia defensiva para protegerse frente a cambios desde el código cliente.

Por ejemplo, los objetos String e Integer de Java son inmutables. Si echas un vistazo a la documentación de Java de estas clases, verás que todos los métodos que en principio alterarían el estado interno de uno de estos objetos, en realidad devuelven un nuevo objeto sin modificar el actual. Por ejemplo, los método concat(), replace() o trim() de String. El objeto original no es alterado.

No es lo mismo usar las colecciones con objetos inmutables (`Strings`, `Integer`, etc.) que con objetos mutables. Los objetos inmutables no pueden ser modificados después de su creación, por lo que cuando se incorporan a la lista, a través de los métodos `add`, se pasan por copia (es decir, se realiza una copia de los mismos). En cambio los objetos mutables, no se copian, y eso puede producir efectos no deseados.

Imaginate la siguiente clase, que contiene un número:

```
class Test {
    private int num;
    Test (int num) {
        this.num = num;
    }
    public void setNum(int num) {
        this.num = num;
    }
    public String toString() {
        return "Número: " + num;
    }
}
```

La clase de antes es mutable, por lo que no se pasa por copia a la lista. Ahora imagina el siguiente código en el que se crea una lista que usa este tipo de objeto, y en el que se insertan dos objetos:

```
Test p1 = new Test(11);                                 // Se crea un objeto Test donde el entero que contiene vale 11.
Test p2 = new Test(12);                                 // Se crea otro objeto Test donde el entero que contiene vale 12.
LinkedList<Test> lista = new LinkedList<Test>();    // Creamos una lista enlazada para objetos tipo Test.
lista.add(p1);                                          // Añadimos el primero objeto test.
lista.add(p2);                                          // Añadimos el segundo objeto test.
System.out.println(lista.toString());               // Mostramos la lista de objetos.
```

¿Qué mostraría por pantalla el código anterior? Simplemente mostraría los números 11 y 12. Ahora bien, ¿qué pasa si modificamos el valor de uno de los números de los objetos test? ¿Qué se mostrará al ejecutar el siguiente código?

```
p1.setNum(44);
System.out.println(lista);
```

El resultado de ejecutar el código anterior es que se muestran los números 44 y 12. El número ha sido modificado y no hemos tenido que volver a insertar el elemento en la lista para que en la lista se cambie también. Esto es porque en la lista no se almacena una copia del objeto `Test`, sino un apuntador a dicho objeto. Solo hay una copia del objeto a la que se hace referencia desde distintos lugares.

Para resolverlo, tendríamos 2 alternativas:

- Hacer que el objeto Test sea inmutable. Su atributo debería ser final y el método setNum debería desaparecer.
- Añadir una copia de p1 y p2 para que aunque se modifique el original la copia no cambie.

> "Controlar la complejidad es la esencia de la programación."
>
> *Brian Kerniga*

Pregunta

Los elementos de un `ArrayList` de objetos `Short` se copian al

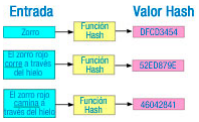
insertarse al ser objetos mutables. ¿Verdadero o falso?

Ana se toma un descanso, se levanta y en el pasillo se encuentra con **Juan**, con el que entabla una conversación bastante amena. Una cosa lleva a otra y al final, Ana saca el tema que más le preocupa:

—¿Cuántos tipos de colecciones hay? ¿Tu te los sabes? —pregunta Ana.

—¿Yo? ¡Que va! Normalmente consulto la documentación cuando los voy a usar, como todo el mundo. Lo que sí creo recordar es que había cuatro tipos básicos: los conjuntos, las listas, las colas y alguno más que no recuerdo. ¡Ah sí¡, los mapas, aunque creo que no se consideraban un tipo de colección. ¿Por qué lo preguntas?

—Pues porque tengo que usar uno y no sé cuál.



Los conjuntos o Set son un tipo de colección que **no admite duplicados**.

La interfaz `java.util.Set` define cómo deben ser los conjuntos, y extiende la interfaz `Collection`, aunque no añade ninguna operación nueva. Contiene únicamente los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos.

Es importante destacar que para comprobar si los elementos son elementos duplicados o no y un Set funcione correctamente, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos **equals y hashCode**. Estos son los métodos que utilizan internamente los conjuntos para saber si 2 elementos son iguales.

Las implementaciones o clases genéricas más usadas que implementan la interfaz `Set` son las siguientes:

- `java.util.HashSet`. Conjunto que almacena los objetos usando **tablas hash**, lo cual acelera enormemente el acceso a los objetos almacenados. Inconvenientes: necesitan bastante memoria y **no almacenan los objetos de forma ordenada**.
- `java.util.LinkedHashSet`. Conjunto que almacena objetos combinando **tablas** hash, para un acceso rápido a los datos, y **listas enlazadas** para conservar el orden. **El orden de almacenamiento es el de inserción**, por lo que se puede decir que es una estructura ordenada a medias. Inconvenientes: necesitan bastante memoria y es algo mas lenta que HashSet.
- `java.util.TreeSet`. Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Son más lentas que los dos tipos anteriores. pero tienen una gran ventaja: **los datos almacenados se ordenan por valor**. Es decir, que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno. Los elementos almacenados deben implementar la interfaz Comparable.

  Veamos un ejemplo de uso básico de la estructura `HashSet` y después, profundizaremos en los `LinkedHashSet` y los `TreeSet`.

  Para crear un conjunto, simplemente creamos el `HashSet` indicando el tipo de objeto que va a almacenar, dado que es una clase genérica que puede trabajar con cualquier tipo de dato debemos crearlo como sigue. No olvides hacer la importación de `java.util.HashSet` primero:

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

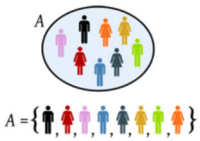
Después podremos ir almacenando objetos dentro del conjunto usando el método `add` (definido por la interfaz `Set`). Los objetos que se pueden insertar serán siempre del tipo especificado al crear el conjunto:

```
String n = new String("Hola");
if (!conjunto.add(n)) {
    System.out.println("Elemento ya en la lista.");
}
```

Si el elemento ya está en el conjunto, **el método `add` retornará `false` indicando que no se pueden insertar duplicados**. **Si todo va bien, retornará `true`.**

Pregunta

¿Cuál de las siguientes estructuras ordena automáticamente los elementos según su valor?



Y ahora te preguntarás, ¿cómo accedo a los elementos almacenados en un conjunto? Los conjuntos, a diferencia de las listas, no referencian sus elementos mediante índices y no hay otra forma de acceder a los elementos de un conjunto directamente. Por ello, para obtener los elementos almacenados en un conjunto hay que usar el **bucle for-each o los iteradores**. Ambis permiten obtener los elementos del conjunto uno a uno de forma secuencial.

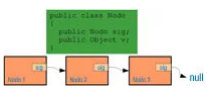
En el siguiente código se usa un bucle for-each. En él el objeto texto va tomando los valores de todos los elementos almacenados en el conjunto hasta que llega al último:

```
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 variable : colección)" y el cuerpo del bucle. Tipo es el tipo del objeto sobre el que se ha creado la colección, variable es la variable donde se almacenará cada elemento de la colección y colección es la colección en sí. Los bucles for-each se pueden usar para todas las colecciones.

```
Iterator<String> it = conjunto.iterator();
while (it.hasNext()) {
    String texto = it.next();
    System.out.println("Elemento almacenado:" + texto);
}
```

Realiza un programa que pregunte al usuario 4 palabras diferentes, las almacene en un `HashSet` y que después muestre la más larga usando un iterador.

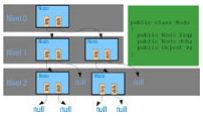


¿En qué se diferencian las estructuras `LinkedHashSet` y `TreeSet` de la estructura `HashSet`? Ya se comento antes, y es básicamente en su funcionamiento interno.

La estructura `LinkedHashSet` es una estructura que internamente funciona como una lista enlazada, aunque usa también tablas hash para poder acceder rápidamente a los elementos. Una lista enlazada es una estructura similar a la representada en la imagen de la derecha, la cual está compuesta por nodos que van enlazándose entre sí. Un nodo contiene dos cosas: el dato u objeto almacenado en la lista y el siguiente nodo de la lista. Si no hay siguiente nodo, se indica poniendo nulo (`null`) en la variable que contiene el siguiente nodo.

Las listas enlazadas tienen un montón de operaciones asociadas en las que no vamos a profundizar: eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, etc. Gracias a las colecciones podremos utilizar listas enlazadas sin tener que complicarnos en detalles de programación.

La estructura `TreeSet`, en cambio, utiliza internamente árboles. Los árboles son como las listas pero mucho más complejos. En vez de tener un único elemento siguiente, pueden tener dos o más elementos siguientes, formando estructuras organizadas y jerárquicas.



Los nodos se diferencian en dos tipos: nodos padre y nodos hijo; un nodo padre puede tener varios nodos hijo asociados (depende del tipo de árbol), dando lugar a una estructura que parece un árbol invertido (de ahí su nombre).

En la figura de la derecha se puede apreciar un árbol donde cada nodo puede tener dos hijos, denominados izquierdo (izq) y derecho (dch). Puesto que un nodo hijo puede también ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el primer nivel (que no tiene padre).

Los árboles son estructuras complejas de manejar y que permiten operaciones muy sofisticadas. Los árboles usados en los `TreeSet`, los árboles rojo-negro, son árboles auto-ordenados, es decir, que al insertar un elemento, este queda ordenado por su valor de forma que al recorrer el árbol, pasando por todos los nodos, los elementos salen ordenados. El ejemplo mostrado en la imagen es simplemente un árbol binario, el más simple de todos.

Nuevamente, no se va a profundizar en las operaciones que se pueden realizar en un árbol a nivel interno (inserción de nodos, eliminación de nodos, búsqueda de un valor, etc.). Nos aprovecharemos de las colecciones para hacer uso de su potencial.

En la siguiente tabla tienes un uso comparado de `TreeSet` y `LinkedHashSet`. Su creación es similar a como se hace con `HashSet`, simplemente sustituyendo el nombre de la clase `HashSet` por una de las otras. Ni `TreeSet`, ni `LinkedHashSet` admiten duplicados, y se usan los mismos métodos ya vistos antes, los existentes en la interfaz `Set` (que es la interfaz que implementan).

Ejemplos de utilización de los conjuntos `TreeSet` y `LinkedHashSet`.

	Conjunto `TreeSet`.	**Conjunto `LinkedHashSet`.**

Ejemplo de uso	`TreeSet<Integer> t;`
	`t = new TreeSet<Integer>();`
	`t.add(new Integer(4));`
	`t.add(new Integer(3));`
		`LinkedHashSet<Integer> t;`
		`t = new LinkedHashSet<Integer>();`
		`t.add(new Integer(4));`
		`t.add(new Integer(3));`

	Conjunto `TreeSet`.	Conjunto `LinkedHashSet`.

	`t.add(new Integer(1));`
	`t.add(new Integer(99));`
	`for (Integer num : t) {`
	`System.out.println(num);`
	`}`
		`t.add(new Integer(1));`
		`t.add(new Integer(99));`
		`for (Integer num : t) {`
		`System.out.println(num);`
		`}`

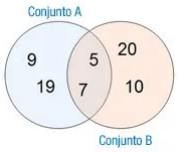
Resultado mostrado por pantalla	1 3 4 99 (el resultado sale ordenado por valor)	4 3 1 99 (los valores salen ordenados según el momento de inserción en el conjunto)

Pregunta

Un árbol cuyos nodos solo pueden tener un único nodo hijo, en realidad es una lista. ¿Verdadero o falso?

¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle `for` y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones.

Partimos del siguiente ejemplo, en el que hay dos colecciones de diferente tipo, cada una con 4 números enteros:

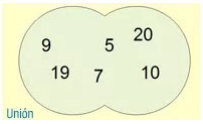



```
TreeSet<Integer> A = new TreeSet<Integer>();
A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7

LinkedHashSet<Integer> B = new LinkedHashSet<Integer>();
B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio `Integer` sin tener que hacer nada, lo cual es una ventaja. Veamos las formas de combinar ambas colecciones:

Tipos de combinaciones.

Combinación.	Código.	Elementos finales del conjunto A.
Unión. Añadir todos los elementos del conjunto B en el conjunto A.	`A.addAll(B)`	Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20. 
Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.	`A.removeAll(B)`	Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19. 

Recuerda, estas operaciones son comunes a todas las colecciones.

Pregunta

Tienes un `HashSet` llamado `vocales` que contiene los elementos "a", "e", "i", "o", "u", y otro, llamado `vocales_fuertes` con los elementos "a", "e" y "o". ¿De qué forma podríamos sacar una lista con las denominadas vocales débiles (que son aquellas que no son fuertes)?

Respuestas

Opción 1

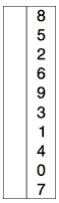
```
vocales.retainAll (vocales_fuertes);
```

Opción 2

```
vocales.removeAll(vocales_fuertes);
```

Opción 3

No es posible hacer esto con `HashSet`, solo se puede hacer con `TreeSet` o `LinkedHashSet`.



Por defecto, los `TreeSet` ordenan sus elementos de forma ascendente, pero, ¿se podría cambiar el orden de ordenación? Los `TreeSet` tienen un conjunto de operaciones adicionales, además de las que incluye por el hecho de ser un conjunto, que permite entre otras cosas, cambiar la forma de ordenar los elementos. Esto es especialmente útil cuando el tipo de objeto que se almacena no es un simple número, sino algo más complejo (un artículo por ejemplo). `TreeSet` es capaz de ordenar tipos básicos (números, cadenas y fechas) pero otro tipo de objetos no puede ordenarlos con tanta facilidad.

Para indicar a un `TreeSet` cómo tiene que ordenar los elementos, debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello, utilizamos la interfaz genérica `java.util.Comparator`, usada en general en algoritmos de ordenación, como veremos en la UD siguiente.

Como adelanto, comentar que lo que tendremos que hacer es añadir un único método denominado compare a la clase. Tiene dos parámetros: los dos objetos a comparar y las reglas son sencillas, a la hora de personalizar dicho método:

- Si el primer objeto (`o1`) es menor que el segundo (`o2`), debe retornar un número entero negativo.
- Si el primer objeto (`o1`) es mayor que el segundo (`o2`), debe retornar un número entero positivo.
- Si ambos son iguales, debe retornar 0.

Por ahora, utilizaremos los TreeSet únicamente para almacenar los tipos básicos.

Juan se quedó pensativo después de la conversación con **Ana**. **Ana** se fue a su puesto a seguir trabajando, pero él se quedó dándole vueltas al asunto...

—Sí que está bien preparada **Ana**, me ha puesto en jaque y no sabía qué responder.

El hecho de no poder ayudar a Ana le frustró un poco.

De repente, apareció María. Entonces Juan aprovecha el momento para preguntar con más detalle acerca del trabajo de Ana. María se lo cuenta y de repente, se le enciende una bombilla a Juan... dice: —Vale, creo que puedo ayudar a Ana en algo, le aconsejaré usar mapas y le explicaré cómo se usan.

¿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.

En Java existe la interfaz `java.util.Map` que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz: `java.util.HashMap`, `java.util.TreeMap` y `java.util.LinkedHashMap`. ¿Te suenan? Claro que si. Cada una de ellas, respectivamente, tiene características similares a `HashSet`, `TreeSet` y `LinkedHashSet`, tanto en funcionamiento interno como en rendimiento.

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. 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.

Método.	Descripción.
`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.

Vemos que como en el caso de las listas, tiene métodos comunes a todas las colecciones y otros especificos de los mapas.

Completa el siguiente código para que al final se muestre el número 40 por pantalla:

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>test();

for (int i = 0; i < 10; i++) {
    mapa.put(i, i);                          // Insertamos
datos de prueba en el mapa.

}
for (Integer llave : mapa.keySet()) {       // Recorremos
el conjunto generado por keySet, contendrá las llaves.

    Integer valor = mapa.get(llave);        // Para cada
llave, accedemos a su valor si es necesario.

}
```

Lo único que tienes que tener en cuenta es que el conjunto generado por **keySet** no tendrá obviamente el método `add` para añadir elementos al mismo, dado que eso tendrás que hacerlo a través del mapa.

Pregunta

¿Cuándo debemos invocar el método `remove()` de los iteradores?

Respuestas

Opción 2

Después de invocar el método `next()`.

Opción 3

Después de invocar el método `hasNext()`.

Opción 4

No es conveniente usar este método para eliminar elementos, es mejor usar el de la colección.

Para trabajar con fechas vamos a utilizar la **clase Date** para lo que necesitamos la librería java.util.Date.

Lo primero que vamos a hacer es **crear la fecha y hora actual** y mostrarla

por consola:

```
// Crear una instancia de la clase Date
// El objeto guardará la fecha y hora actual
Date fechaActual = new Date();
      // Mostrar la fecha con formato predeterminado
System.out.println(fechaActual);
```

Lo que veremos es:

```
Thu Jan 24 15:57:47 CET 2019
```

La fechas se muestra en el **formato predeterminado**.

Si queremos mostrar la fecha y la hora de otra forma tendremos que **definir el formato** que nos interesa utilizando la **Clase SimpleDateFormat** de la siguiente manera. Necesitamos importar la librería java.text.SimpleDateFormat:

```
// Definimos diferentes formatos con la Clase
SimpleDateFormat
SimpleDateFormat formato1 = new SimpleDateFormat("hh:mm:ss
a");
SimpleDateFormat formato2 = new SimpleDateFormat("dd 'de'
MMMM 'de' yyyy. hh:mm:ss zzz");
SimpleDateFormat formato3 = new SimpleDateFormat("EEEE MM
dd yyyy");
// Aplicamos esos formatos a una fecha
System.out.println("Formato 1: " +
formato1.format(fechaActual);
System.out.println("Formato 2: " +
formato2.format(fechaActual));
System.out.println("Formato 3: " +
formato3.format(fechaActual));
```

Lo que veremos es:

```
Formato 1: 04:53:24 PM
Formato 2: 24 de enero de 2019. 04:53:24 CET
Formato 3: jueves 01 24 2019
```

Como se observa, para incluir texto dentro del formato hay que usar las comillas simples. Las letras principales para definir el formato se muestran a continuación y dependiendo del número de veces que aparezca cambiará el valor que se muestre:

y	Año. Por ejemplo: yyyy --> 2018
M	Mes. Por ejemplo: MM --> 01 o MMMM --> enero
d	Día. Por ejemplo: dd --> 24
E	Día de la semana. Por ejemplo: EEEE --> jueves
h	Hora AM/PM. Por ejemplo: hh --> 04
m	Minutos. Por ejemplo: mm --> 34
s	Segundos. Por ejemplo: ss --> 25
H	Hora 0-23. Por ejemplo: hh --> 16
a	AM/PM
z	Zona horaria. Por ejemplo: zzz --> CET (Central European Time)

Y si en vez de la fecha actual **¿quiero crear cualquier otra fecha?**

La clase **SimpleDateFormat** nos permite instanciar un objeto a partir de una cadena de caracteres que cumpla con el formato definido. Por ejemplo, si el formato es "dd-mm-yyyy" cualquier String del estilo "24-01-2019" permitirá crear objetos Date como se ve en el siguiente ejemplo:

```
SimpleDateFormat formatoFecha = new SimpleDateFormat("dd-
mm-yyyy");
try {
    Date fecha1 = formatoFecha.parse("20-08-2017");
    System.out.println("Fecha introducida: " +
formatoFecha.format(fecha1));
} catch (ParseException e) {
    System.out.println("El formato de la fecha es
incorrecto");
}
```

El método que hay que utilizar es **parse** y genera una excepción ParseException de tipo "tested". Por ello, para que el programa se pueda compilar se deberá gestionar mediante la sentencia try-catch o lanzarse usando throws.

¿Y si quisiera comparar 2 fechas?

El método más útil para comparar fechas es el **método compareTo**. Es un método de la clase Date que devuelve un entero. Si la fecha es anterior al fecha que se pasa como parámetro el resultado será negativo, si es posterior devolverá un número positivo y si son iguales devolverá 0.

Por ejemplo:

```
SimpleDateFormat formatoFecha = new SimpleDateFormat("dd-
mm-yyyy");
Date fecha = new Date();
System.out.println("Fecha actual: " +
formatoFecha.format(fecha));
try {
    Date fecha1 = formatoFecha.parse("20-08-2017");
    System.out.println("Fecha introducida: " +
formatoFecha.format(fecha1));
    int comparacion = fecha1.compareTo(fecha);
    if (comparacion > 0) {
        System.out.println("La fecha introducida ocurre
después de la fecha actual");
    } else if (comparacion < 0) {
        System.out.println("La fecha introducida ocurre
antes de la fecha actual");
    } else {
        System.out.println("Ambas fechas son iguales");
    }
} catch (ParseException e) {
    System.out.println("El formato de la fecha es
incorrecto");
}
```

El resultado será:

Lo que veremos es:

```
Fecha actual: 24-09-2019
Fecha introducida: 20-08-2017
La fecha introducida ocurre antes de la fecha actual
```

Si quieres profundizar en las funcionalidades de la clases SimpleDateFormat, puedes consultar la web de Oracle:

Clase SimpleDateFormat

¿Tienen algo en común todos los números de DNI y de NIE? ¿Podrías hacer un programa que verificara si un DNI o un NIE es correcto? Seguro que sí. Si te fijas, 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 como 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.

  Con lo visto hasta ahora ya es posible construir una expresión regular capaz de verificar si una cadena contiene un DNI o un NIE, ¿serías capaz de averiguar cuál es dicha expresión regular?

¿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. Veamoslo 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. Una segunda invocación del método `find()` irá a la

segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método `find()`, para que vuelva a comenzar por la primera coincidencia, invocando el método `m.reset()`.

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".
- "`^[01]+$`". Cuando el símbolo "`^`" aparece al comienzo de la expresión regular, permite indicar comienzo de línea o de entrada. El símbolo "`$`" permite indicar fin de línea o fin de entrada. Usándolos podemos verificar que una línea completa (de principio a fin) encaje con la expresión regular, es muy útil cuando se trabaja en **modo multilínea** y con el método `find()`.
- "`.`". 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]`".

Pregunta

¿En cuáles de las siguientes opciones se cumple el patrón "`A.\\d+`"?

Respuestas

Opción 1

"`GA-99`" si utilizamos el método `find`.

Opción 2

"`GAX99`" si utilizamos el método `lookingAt`.

Opción 3

"`AX99-`" si utilizamos el método `matches`.

Opción 4

"`A99`" si utilizamos el método `matches`.

¿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`, éste 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 (¿Lo recuerdas del apartado anterior?) 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: "`\"`".**

Con lo estudiado hasta ahora, ya puedes utilizar las expresiones regulares y sacarles partido casi que al máximo. Pero las expresiones regulares son un campo muy amplio, por lo que es muy aconsejable que amplíes tus conocimientos con el siguiente enlace:

Tutorial de expresiones regulares en Java (en inglés).

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso	Datos del recurso (1)	Recurso	Datos del recurso (2)
	Autoría: Chealer. Licencia: GNU GPL. Procedencia: http://commons.wikimedia.org /wiki/File:Bash_demo.png		Autoría: Mass Communication Specialist 3rd Class Matthew Jackson. U.S. Navy. Licencia: Dominio público. Procedencia: http://commons.wikimedia.org /wiki/File:Orif_surgery.jpg
	Autoría: LatinStock. Licencia: Uso educativo para plataformas públicas de FPaD. Procedencia: LatinStock.		Autoría: Berteun. Licencia: Dominio público. Procedencia: http://commons.wikimedia.org /wiki/File: Swiss_Army_Knife_Wenger_Opened_20050627.jpg
	Autoría: LatinStock. Licencia: Uso educativo para plataformas públicas de FPaD. Procedencia: LatinStock.		Autoría: Fercufer. Licencia: CC-BY-SA. Procedencia: Montaje sobre http://es.wikipedia.org /wiki/Archivo: Hash_function2-es.svg
	Autoría: Ilustración de persona: AIGA symbol signs collection; trabajo derivado por kismalac. Licencia: CC-BY-SA. Procedencia: http://es.wikipedia.org /wiki/Archivo:PersonsSet.svg		Autoría: Salvador Romero Villegas y los autores de la herramienta de Software Libre NetBeans IDE 7.0 Licencia: GNU GPL (el producto NetBeans IDE 7.0 tiene licencia GNU GPL). Procedencia: Código fuente escrito por Salvador Romero Villegas. Iconografía e interfaz visible elaborada por los participantes y colaboradores del Software Libre NetBeans IDE 7.0.
	Autoría: en:Joestape89. Licencia: CC-BY-SA. Procedencia: http://es.wikipedia.org /wiki/Archivo:Selection-Sort-Animation.gif		Autoría: Java. (usuario wikipedia que subío la imagen: Machro). Licencia: Dominio público. Procedencia: http://commons.wikimedia.org /wiki/File:Java.png
	Autoría: Esterab. Licencia: CC-BY-SA. Procedencia: http://commons.wikimedia.org /wiki/File:Xml.jpg		Autoría: David Vignoni. Licencia: GNU LGPL. Procedencia: http://commons.wikimedia.org /wiki/File:Web-browser.svg
	Autoría: Original photo: User:Fanghong; Derivative work: User:Gnomz007. Licencia: CC-BY-SA. Procedencia: http://es.wikipedia.org /wiki/Archivo:Russian-Matroshka_no_bg.jpg		Autoría: Salvador Romero Villegas y los autores de la herramienta de Software Libre NetBeans IDE 7.0. Licencia: GNU GPL (el producto NetBeans IDE 7.0 tiene licencia GNU GPL). Procedencia: Código fuente escrito por Salvador Romero Villegas. Iconografía e interfaz visible elaborada por los participantes y colaboradores del Software Libre NetBeans IDE 7.0.
	Autoría: Booyabazooka. Licencia: CC-BY-SA.		