Unidad 6

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 11

Cadenas y Expresiones Regulares

Cadenas y Expresiones Regulares

En Este Capítulo

- La Clase String
- Las Clases StringBuffer y StringBuilder
- La clase StringTokenizer
- Introducción a las expresiones regulares
- Clase para pruebas
- Cadena de literales
- Expresiones regulares
- Patrones
- Creando un objeto analizador del tipo Matcher
- Buscando una cadena
- Definiendo conjunto de caracteres
- Operadores lógicos y expresiones regulares
- Cuantificadores

La Clase String

Los objetos String son secuencias inmutables de caracteres Unicode, esto quiere decir, una vez creado no se pueden alterar. Cualquier intento de modificarlo generará la creación de un nuevo objeto del tipo y el anterior será seleccionable para el recolector de basura (garbage collector).

Las operaciones que pertenecen a este tipo de objeto pueden crear nuevos Strings, como ser: concat, replace, substring, toLowerCase, toUpperCase y trim. Lo que obviamente se debe hacer es asignarlo a otra referencia del mismo tipo si no se quiere perder el que generó la invocación

Posee operaciones de:

- **Búsqueda**: endsWith, startsWith, indexOf y lastIndexOf.
- **Comparaciones**: equals, equalsIgnoreCase y compareTo.
- **Otras**: charAt y length.

Las Clases StringBuffer y StringBuilder

Los objetos del tipo StringBuffer y StringBuilder almacenan en su interior secuencias de caracteres Unicode que pueden cambiar. Los objetos de estos tipos deben ser utilizados cada vez que se quiera trabajar con el formato de un String cuyo tamaño pueda ser variable. La principal diferencia entre ambas clases es que StringBuilder es segura respecto de los threads, es decir, sincroniza la memoria compartida para que un thread no escriba en dicha memoria mientras otros puedan estar operando en ella. Esto, evidentemente, tiene un costo mayor en tiempo de ejecución.

Constructores

Para StringBuffer

Constructor	Descripción
StringBuffer()	Crea un buffer vacío
StringBuffer(int capacity)	Crea un buffer vacío con una capacidad inicial
StringBuffer(CharSequence seq)	Construye un búfer de cadena que contiene los mismos caracteres que la CharSequence especificado
StringBuffer(String initialString)	Crea un buffer vacío con un String inicial

Para StringBuilder

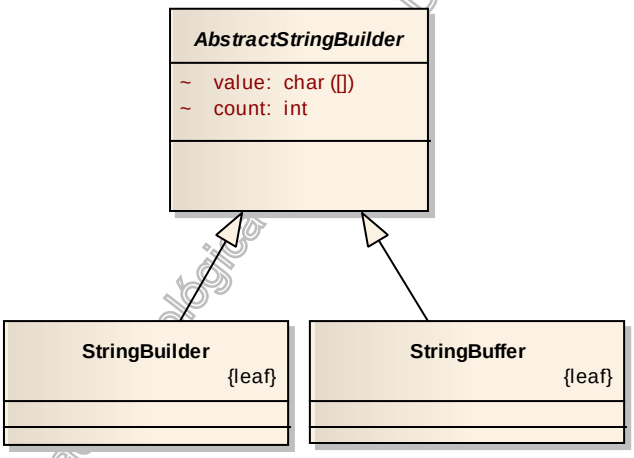
Constructor	Descripción
StringBuilder()	Crea un buffer vacío

Constructor	Descripción
StringBuilder(int capacity)	Crea un buffer vacío con una capacidad inicial
StringBuilder(CharSequence seq)	Construye un búfer de cadena que contiene los mismos caracteres que la CharSequence especificado
StringBuilder(String initialString)	Crea un buffer vacío con un String inicial

La clase define varias operaciones para hacer modificaciones: append, insert, reverse, setCharAt y setLength.

Elementos comunes

Ambas clases heredan de AbstractStringBuilder y es esta clase la que almacena en un vector los caracteres que pertenecen al String que manejan las subclases.



StringBuffer esta "a salvo de threads" y es su principal diferencia respecto de StringBuilder. Por ejemplo, se puede apreciar la diferencia de declaración para un mismo método.

StringBuffer

```
public synchronized StringBuffer append(String str) {
        super.append(str);
        return this;
    }
```

StringBuilder

```
public StringBuilder append(Object obj) {
        return append(String.valueOf(obj));
```

```
}
```

Si bien la sincronización existe es sumamente extraño que se tenga necesidad de ella. Cuando dos threads compiten por el acceso a memoria compartida (race condition) difícilmente lo hagan dentro del contenido de AbstractStringBuilder que utilizan ambas clases y es poco probable que surjan problemas de sincronización. En cambio, debido al mecanismo de sincronización que posee una clase respecto de la otra, si hay un efecto visible en el rendimiento. Diferentes verificaciones encontraron que cuando se realizan más de 1000 operaciones sobre ambas clases, StringBuilder puede ser hasta un 30% más rápido.

La clase StringTokenizer

El procesamiento de texto a menudo consiste en analizar una cadena de entrada que posee un formato. El análisis por interpretación es la división del texto en un conjunto de partes discretas, o cortes (tokens), que en una cierta secuencia puede tener un significado semántico. La clase StringTokenizer proporciona el primer paso en este proceso de análisis, a menudo llamada el analizador de léxico (análisis lexicográfico) o escáner. StringTokenizer implementa la interfaz Enumeration. Por lo tanto, dada una cadena de entrada, se pueden enumerar los tokens individuales contenidos en la misma utilizando StringTokenizer.

Para usar StringTokenizer, se especifica una cadena de entrada y una cadena que contiene delimitadores. Los delimitadores son los caracteres que se usan para separar los elementos. Cada caracter en la cadena de delimitadores se considera uno válido, por ejemplo, ",;:" establece los delimitadores de una coma, punto y coma, y dos puntos. El conjunto predeterminado de delimitadores se compone de los caracteres que implican espacios en blanco: espacio, tabulador, nueva línea y retorno de carro.

Los constructores StringTokenizer se muestran a continuación:

```
public StringTokenizer(String str, String delim, boolean returnDelims)
public StringTokenizer(String str, String delim)
public StringTokenizer(String str)
```

En todas las versiones, str es la cadena que se corta. En la primera versión, se utilizan los delimitadores por defecto. En las versiones segunda y tercera, los delimitadores son una cadena que los especifica.

Los delimitadores no se retornan como tokens por los dos primeros constructores. Una vez que se haya creado un objeto StringTokenizer, el método nextToken se utiliza para extraer tokens consecutivos. El método hasMoreTokens retorna verdadero mientras haya más tokens para ser extraídos. Como StringTokenizer implementa una enumeración, se implementan los métodos los hasMoreElements y nextElement, y actúan de la misma forma que lo hacen hasMoreTokens y nextToken, respectivamente.

Introducción a las expresiones regulares

¿Qué son las expresiones regulares?

Las expresiones regulares son una forma de describir un conjunto de cadenas sobre la base de características comunes compartidas por cada cadena en dicho conjunto. Pueden ser utilizadas para buscar, editar o manipular el texto y datos. Se puede aprender una sintaxis específica para crear expresiones regulares pero esto va más allá de la sintaxis normal del lenguaje de programación Java. Las expresiones regulares varían en complejidad, pero una vez que se entienden los conceptos básicos de cómo están construidas, se es capaz de descifrar (o crear) una expresión regular.

Se verá la sintaxis de expresiones regulares con la ayuda de la java.util.regex API y seilustrar. En el mundo de las expresiones regulares, hay muchos sabores diferentes para elegir, tales como grep, Perl, Tcl, Python, PHP, y awk. La sintaxis de expresiones regulares en el java.util.regex API es muy similar a la que se encuentra en Perl.

¿Cómo son representadas las expresiones regulares en el paquete java.util.regex?

El paquete java.util.regex se compone fundamentalmente de tres clases: Pattern , Matcher y PatternSyntaxException .

- **Pattern**: un objeto de este tipo es una representación compilada de una expresión regular. La clase Pattern no tiene constructores públicos. Para crear un patrón, primero se debe invocar a alguno de sus métodos **`public static`** compile, que luego devuelven un objeto del tipo Pattern. Estos métodos aceptan una expresión regular como primer argumento.
- **Matcher**: un objeto de este tipo es el motor que interpreta al patrón y realiza operaciones de reconocimiento sobre una cadena de entrada. Al igual que Pattern la clase, Matcher no define constructores públicos. Se obtiene un objeto del tipo Matcher mediante la invocación del método matcher en un objeto del tipo Pattern.
- **PatternSyntaxException**: un objeto de este tipo es una excepción no verificada que indica un error de sintaxis en un patrón de la expresión regular.

Cadena de literales

La forma más básica de coincidencia de patrones con el apoyo de esta API es la búsqueda en una cadena literal. Por ejemplo, si la expresión regular es “hola” y la cadena de entrada es “hola”, entonces la búsqueda tendrá éxito porque ambas cadenas son idénticas.

Ejemplo

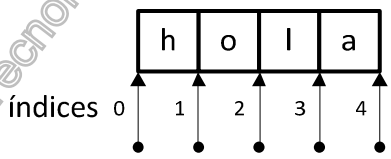
```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class VerificaExpresion {
	public static void main(String[] args) {
		String expresion = "hola";
```

```
            String cadena = "hola";

            boolean encontrado = false;
            Pattern patron = Pattern.compile(expresion);

            Matcher coincidencias = patron.matcher(cadena);

            int lugar = 0;
            while (coincidencias.find(lugar)) {
                  System.out.printf("Se encontró el texto \"%s\" " +
                              "comenzando en el "
                              + "índice %d y finalizando en el índice %d.%n",
                              coincidencias.group(), coincidencias.start(),
                              coincidencias.end());
                  lugar = coincidencias.end();
                  encontrado = true;
            }
            if (!encontrado) {
                  System.out.printf("No se encontró correspondencia.");
            }
            encontrado = false;
      }
}
```

La salida que se obtiene es

```
Se encontró el texto "hola" comenzando en el índice 0 y finalizando en el índice
4.
```

Esta búsqueda fue un éxito. Obsérvese que mientras que la cadena de entrada es de 4 caracteres, el índice inicial es 0 y el índice final es 4. Por convenio, los rangos son inclusivos del índice de comienzo y exclusivos del índice final, como se muestra en la figura siguiente:



Patrones

En su forma más elemental, una expresión regular sólo hace una búsqueda simple de una sub cadena. Por ejemplo, si se desea buscar en una cadena la palabra hola, la expresión regular es exactamente esa. La cadena que define esta expresión regular es "hola". Para crear un objeto patrón de la expresión "hola" como esto:

```
Pattern patron = Pattern.compile("hola");
```

El método estático compile de la clase Pattern retorna una referencia a un objeto que contiene la expresión regular compilada. El método genera una PatternSyntaxException si el argumento no es válido. No es necesario capturar esta excepción, ya que es una subclase de RuntimeException y por lo tanto no es verificada, pero es una buena idea hacerlo para asegurarse de que el patrón de

la expresión regular es válido. El proceso de compilación prepara una cadena normal para que se utilice con una máquina de estado representada por un objeto del tipo Matcher.

Una versión más completa del método compile, que recibe dos argumentos, permite controlar más de cerca cómo el modelo se aplica en la búsqueda de una coincidencia. El segundo argumento es un valor de tipo `int` que especifica uno o más de las siguientes banderas definidas en la clase Pattern:

Bandera	Descripción
CASE_INSENSITIVE	Coincidencias ignorando mayúsculas o minúsculas, pero asume sólo se buscan coincidencias con caracteres del tipo US-ASCII.
MULTILINE	Permite encontrar como coincidencia el principio o final de las líneas. Sin esta bandera, sólo el comienzo y el final de la secuencia entera se consideran una coincidencia.
UNICODE_CASE	Cuando esto se especifica, junto con CASE_INSENSITIVE , las coincidencias sin tener en cuenta mayúsculas y minúsculas son consistentes con el estándar Unicode.
DOTALL	Hace que la expresión coincida con cualquier caracter, incluyendo terminaciones de línea.
LITERAL	Hace que la cadena que especifica el patrón sea tratado como una secuencia de caracteres literales, por lo que las secuencias de escape, por ejemplo, no son reconocidas como tales.
CANON_EQ	Coincidencias que tienen en cuenta la equivalencia canónica de caracteres combinados. Por ejemplo, algunos caracteres que tienen signos diacríticos pueden ser representados como un caracter único o como un solo carácter con un diacrítico seguido de un carácter diacrítico. Este indicador trata esto como una coincidencia.
COMMENTS	Permite espacios en blanco y comentarios en un patrón. Los comentarios en un patrón comienzan con # hasta el final de la línea.
UNIX_LINES	Activa el modo de líneas de UNIX, donde sólo el '\ n' es reconocido como un terminador de línea.
UNICODE_CHARACTER_CLASS	Activa la versión Unicode de clases de caracteres predefinidos.

Todas estas banderas tienen activado un único bit dentro de un valor de tipo **int** para que se puedan combinar con el operador OR o por simple adición. Por ejemplo, se puede especificar las banderas CASE_INSENSITIVE y UNICODE_CASE con la siguiente expresión:

```
Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE
```

O también se puede escribir como:

```
Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE
```

Se debe tener cuidado cuando se quiere añadir una bandera a una variable que representa un conjunto existente de éstas.

Si la bandera ya existe, produce un resultado erróneo debido al acarreo en la adición de los dos bits correspondientes al siguiente bit. Utilizar el operador OR siempre produce el resultado correcto.

Si se desea hacer coincidir "hola" ignorando mayúsculas y minúsculas, se puede crear el patrón con la siguiente declaración:

```
Pattern patron = Pattern.compile("hola", Pattern.CASE_INSENSITIVE);
```

Además de la excepción producida por la primera versión del método, esta versión arroja una IllegalArgumentException si el segundo argumento tiene valores de bits establecidos que no corresponden a ninguna de las constantes de bandera definidas en la clase de Pattern.

Comparadores

Después de tener un objeto patrón, se puede crear un objeto del tipo Matcher en el cual buscar una cadena especifica, como por ejemplo:

```
String oracion = "Juan, dice a Pedro hola hola 'hola', hola hola 'hola hola'.";
Matcher matchHola = patron.matcher(oracion);
```

La primera declaración define la cadena de la oración en la que se desea buscar. Para crear el objeto del tipo Matcher, se llama al método matcher en el objeto patrón con la cadena que se va a analizar como argumento.

Esto retorna un objeto Matcher que puede analizar la cadena que se le ha pasado. El parámetro para el método matcher es en realidad de tipo CharSequence. Esta es una interfaz que es implementada por las clases String, StringBuffer y StringBuilder, por lo que se puede pasar una referencia a cualquiera de estos tipos como argumento del método. La clase java.nio.CharBuffer también implementa CharSequence, por lo que también se puede pasar el contenido de un objeto dek tipo CharBuffer al método. Esto significa que si se utiliza un CharBuffer para almacenar los datos de caracteres leídos desde un archivo, se los puede pasar directamente al método matcher.

Una de las ventajas de la implementación de Java de las expresiones regulares es que se puede reutilizar un objeto del tipo Pattern para crear objetos del tipo Matcher para buscar el patrón en

una variedad de cadenas. Para utilizar el mismo patrón para buscar en otra cadena, sólo se cambia el argumento del método matcher en el objeto patrón con la nueva cadena como argumento. De esta manera se tiene un objeto Matcher nuevo que se puede utilizar para buscar en la nueva cadena.

También se puede cambiar la cadena que un objeto Matcher utiliza para la búsqueda llamando al método reset que éste tiene con la nueva cadena como argumento. Por ejemplo:

```
matchHola.reset(“Hola Mundo. Esta la nueva cadena que tiene la palabra hola.”);
```

Esto remplaza la cadena anterior, oracion, en el objeto Matcher, de manera que ahora es capaz de buscar en la nueva cadena. Al igual que el método matcher en la clase de Pattern, el tipo de parámetro para el método reset es del tipo CharSequence, por lo que se puede pasar una referencia de tipo String, StringBuffer, StringBuilder, o CharBuffer.

Búsqueda de una cadena

Una vez que se tiene un objeto del tipo Matcher, se puede usar para buscar en la cadena. Al llamar al método find del objeto del tipo Matcher se busca en la cadena la siguiente ocurrencia del patrón. Si se encuentra el patrón, el método almacena información acerca de dónde se encuentra en el objeto Matcher y devuelve true. Si no encuentra el patrón, devuelve false. Cuando el patrón se ha encontrado, llamando al método start del objeto Matcher se obtiene la posición del índice en la cadena donde se encontró el primer carácter del patrón. Al llamar al método end retorna la posición del índice después del último carácter del patrón. Los dos valores del índice se devuelven como tipo **int**. Por lo tanto, se podría buscar la primera ocurrencia del patrón de esta manera:

```
if (comparador.find()) {
	System.out.println("Patrón encontrado. Comienzo: "
		+ comparador.start() + " Fin: " + comparador.end());
} else {
	System.out.println("Patrón no encontrado.");
}
```

Tener en cuenta que no se debe llamar a start o end en el objeto de tipo Matcher antes de haber tenido éxito en encontrar el patrón. Hasta que un patrón haya sido encontrado, el objeto del tipo Matcher se encuentra en un estado indefinido y llamar a cualquiera de estos métodos da como resultado una excepción IllegalStateException.

Por lo general, se quiere encontrar todas las ocurrencias de un patrón en una cadena. Cuando se llama al método find, la búsqueda se inicia ya sea desde el comienzo de la región de este comparador, o en el primer carácter no registrado por una llamada previa a find. Por lo tanto, se pueden encontrar fácilmente todas las ocurrencias del patrón mediante la búsqueda en un bucle de la siguiente manera:

```
while (comparador.find()) {
	System.out.println("Patrón encontrado. Comienza en: "
			+ comparador.start() + " y termina en " + comparador.end());
}
```

Al final de este ciclo la posición de índice es el carácter que sigue la última aparición del patrón en la cadena. Si se desea restablecer la posición de índice a cero, se debe llamar a una versión sobrecargada de reset para el objeto Matcher que no tiene argumentos.

```
comparador.reset();// Reiniciar este comparador
```

Búsqueda de una sub cadena

El siguiente es un ejemplo completo para buscar en una cadena un patrón. En primer lugar, hay que definir una cadena, expresionRegular, que contiene la expresión regular, y una cadena, oracion, sobre la que se busca:

```
String expresionRegular = "hola";
String oracion = "Juan, dice a Pedro hola hola 'hola', hola hola 'hola hola'.";
```

También se crea un vector, marcador, de tipo **char**[] con el mismo número de elementos que la cadena, el cual se utiliza para indicar los lugares en los que el patrón se encuentra en la cadena:

```
char[] marcador = new char[oracion.length()];
```

Se llena los elementos del vector marcador inicialmente con espacios utilizando el método estático fill de la clase Array:

```
Arrays.fill(marcador, ' ');
```

Después se remplazan algunos de los espacios del vector con el carácter "^" para indicar en donde se ha encontrado el patrón dentro de la cadena original.

Después de compilar la expresión regular RegEx en un objeto del tipo Pattern, al que se llama patron, se crea un objeto del tipo Matcher, llamado comparador, utilizando el patrón, el cual se aplicará a la cadena oracion:

```
Pattern patron = Pattern.compile(expresionRegular);
Matcher comparador = patron.matcher(oracion);
```

A continuación, se llama al método find de comparador en la condición de un ciclo **while** para encontrar todas las ocurrencias:

```
while (comparador.find()) {
	System.out.println("Patrón encontrado. Comienza en: "
		+ comparador.start() + " y termina en " + comparador.end());
	Arrays.fill(marcador, comparador.start(), comparador.end(), '^');
}
```

Este ciclo continúa mientras el método find retorne **true**. En cada iteración se genera la salida de los valores de los índices devueltos por los métodos start y end, que reflejan la posición de índice en el que se encuentra el primer carácter del patrón, y la posición del índice después del último carácter. También insertar el carácter "^" en el vector marcador en las posiciones en las que el índice indica donde se encontró el patrón utilizando nuevamente el método fill. El ciclo termina

cuando el método find retorna **false**, lo que implica que no hay más ocurrencias del patrón en la cadena.

Ejemplo

```
package patrones;

import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
package patrones;

import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ProbarRegex {
	public static void main(String args[]) {
		// Una expresión regular y una cadena en la cual buscar
		String expresionRegular = "hola";
		String oracion =
		"Juan, dice a Pedro hola hola 'hola', hola hola 'hola hola'.";
		// Se marcan los lugares donde se encuentra el patrón
		// Funciona bien si la letra es siempre del mismo tamaño
		char[] marcador = new char[oracion.length()];
	// Llenar la cadena de espacios para luego colocar el caracter que marca
		Arrays.fill(marcador, ' ');

		// Obtener el compador requerido
		Pattern patron = Pattern.compile(expresionRegular);
		Matcher comparador = patron.matcher(oracion);
		// Buscar cada coincidencia y marcarla
		while (comparador.find()) {
			System.out.println("Patrón encontrado. Comienza en: "
			+ comparador.start() + " y termina en " + comparador.end());
			Arrays.fill(marcador, comparador.start(),
				comparador.end(), '^');
		}
		// Mostrar el resultado con las marcas generadas
		System.out.println(oracion);
		System.out.println(marcador);
	}
}
```


La salida que se obtiene del programa es la siguiente:

```
Patrón encontrado. Comienza en: 19 y termina en 23
Patrón encontrado. Comienza en: 24 y termina en 28
Patrón encontrado. Comienza en: 30 y termina en 34
Patrón encontrado. Comienza en: 37 y termina en 41
Patrón encontrado. Comienza en: 42 y termina en 46
Patrón encontrado. Comienza en: 48 y termina en 52
Patrón encontrado. Comienza en: 53 y termina en 57
Juan, dice a Pedro hola hola 'hola', hola hola 'hola hola'.
```

```
^^^^ ^^^^   ^^^^    ^^^^ ^^^^   ^^^^ ^^^^
```

Definiendo un conjuntos de caracteres

Una expresión regular puede estar formada por caracteres ordinarios, la cuales son letras mayúsculas o minúsculas y números, además de secuencias de meta-caracteres, que son aquellos que tienen un significado especial. El patrón en el ejemplo anterior era sólo la palabra "hola", pero si se quiere buscar en una cadena ocurrencias de conjuntos de caracteres que se encuentren contiguos unos con otros, se debe especificar la expresión regular de otra forma

Una opción es para especificar un carácter intermedio como un comodín mediante el uso de un punto, el cual es un ejemplo de un meta-carácter. Este coincide con cualquier carácter, excepto el del final de línea, por lo que una expresión regular "s.l", representa cualquier secuencia de tres caracteres que comienza con "s" y termina con "l".

Ejemplo

```
package patrones;

import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ProbarRegex {
	public static void main(String args[]) {
		// Una expresión regular y una cadena en la cual buscar
		String expresionRegular = "s.l";
		String oracion = "Sólo soldados, sólo humanos solitarios en " +
			"la tristeza que deja solamente la guerra";
		// Se marcan los lugares donde se encuentra el patrón
		// Funciona bien si la letra es siempre del mismo tamaño
		char[] marcador = new char[oracion.length()];
	// Llenar la cadena de espacios para luego colocar el caracter que marca
		Arrays.fill(marcador, ' ');

		// Obtener el compador requerido
		Pattern patron = Pattern.compile(expresionRegular);
		Matcher comparador = patron.matcher(oracion);
		// Buscar cada coincidencia y marcarla
		while (comparador.find()) {
			System.out.println("Patrón encontrado. Comienza en: "
					+ comparador.start() + " y termina en " +
					comparador.end());
			Arrays.fill(marcador, comparador.start(), comparador.end(),
					'^');
		}
		// Mostrar el resultado con las marcas generadas
		System.out.println(oracion);
		System.out.println(marcador);
	}
}
```

La salida obtenida es:

```
Patrón encontrado. Comienza en: 5 y termina en 8
Patrón encontrado. Comienza en: 15 y termina en 18
Patrón encontrado. Comienza en: 28 y termina en 31
Patrón encontrado. Comienza en: 63 y termina en 66
Sólo soldados, sólo humanos solitarios en la tristeza que deja solamente la guerra
     ^^^       ^^^          ^^^                                  ^^^
```

Notar que la palabra Sólo del principio no se considera un resultado y la razón es debida a que la expresión regular se encuentra en minúsculas y esta palabra no. Sin embargo, este no es el principal problema. Una expresión regular armada de esta manera simplemente ignora comprobar el carácter del medio como una restricción para la salida de la ocurrencia, lo cual implica que si hay cualquier otro carácter, será considerado como verdadero en la comparación. Se puede limitar la expresión regular para encontrar sólo aquellas ocurrencias en las cuales el carácter del medio sea una vocal, con lo cual se puede generar una tercera versión del programa, que arroja un resultado similar dado que su expresión regular es reducida respecto a las ocurrencias que considere verdaderas.

Ejemplo

```
package patrones;

import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ProbarRegex2 {
	public static void main(String args[]) {
		// Una expresión regular y una cadena en la cual buscar
		String expresionRegular = "s[aeiou]l";
		String oracion = "Sólo soldados, sólo humanos solitarios en " +
			"la tristeza que deja solamente la guerra";
		// Se marcan los lugares donde se encuentra el patrón
		// Funciona bien si la letra es siempre del mismo tamaño
		char[] marcador = new char[oracion.length()];
	// Llenar la cadena de espacios para luego colocar el caracter que marca
		Arrays.fill(marcador, ' ');

		// Obtener el compador requerido
		Pattern patron = Pattern.compile(expresionRegular);
		Matcher comparador = patron.matcher(oracion);
		// Buscar cada coincidencia y marcarla
		while (comparador.find()) {
			System.out.println("Patrón encontrado. Comienza en: "
					+ comparador.start() + " y termina en " +
					comparador.end());
			Arrays.fill(marcador, comparador.start(), comparador.end(),
					'^');
		}
		// Mostrar el resultado con las marcas generadas
		System.out.println(oracion);
		System.out.println(marcador);
```

```
        }
}
```

La salida del programa es:

```
Patrón encontrado. Comienza en: 5 y termina en 8
Patrón encontrado. Comienza en: 28 y termina en 31
Patrón encontrado. Comienza en: 63 y termina en 66
Sólo soldados, sólo humanos solitarios en la tristeza que deja solamente la guerra
     ^^^                   ^^^                                 ^^^
```

Notar que en este caso la palabra que tiene una “ó” entre la “s” y la “l” no se considera un resultado verdadero. Esto se debe a que el conjunto de caracteres no incluye las letras con acento.

Se pueden definir conjuntos de caracteres de diferentes maneras, a los cuales se referencian como clases de tipo Character, en una expresión regular. Estos conjuntos se utilizan para definir la condición por la cual se filtra una determinada ocurrencia en una posición específica. La siguiente tabla muestra algunos ejemplos.

Clase	Descripción
[aeiou]	Este es un ejemplo simple en el cual cualquier caracter que sea una vocal en minúscula cumple la condición
[^aeiou]	Esto representa cualquier carácter excepto los que aparecen entre los corchetes a la derecha de ”^“. Por lo tanto se reconoce cualquier carácter que no es una vocal minúscula.
[a-e]	Esto define un rango inclusivo simple. Cualquiera de las letras "a" a "e" en este caso
[a-cs-zA-E]	Esto define un rango inclusivo múltiple. Esto se corresponde con cualquier carácter de "a" a "c", de "s" a "z" o de "A" a "E".

Valiéndonos de la tabla y el ejemplo anterior, se puede construir un ejemplo un poco más complejo para la expresión regular de manera de obtener en el resultado cualquier combinación de las letras s, o y l más allá del caso o la acentuación.

Ejemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ProbarRegex3 {
      public static void main(String args[]) {
            // Una expresión regular y una cadena en la cual buscar
            String expresionRegular = "[sS][aeiouóO][lL]";
            String oracion = "Sólo soldados bajo el SOL, sólo humanos” +
                  “solitarios en la tristeza que deja solamente la guerra";
            // Se marcan los lugares donde se encuentra el patrón
            // Funciona bien si la letra es siempre del mismo tamaño
```

```
            char[] marcador = new char[oracion.length()];
      // Llenar la cadena de espacios para luego colocar el caracter que marca
            Arrays.fill(marcador, ' ');

            // Obtener el compador requerido
            Pattern patron = Pattern.compile(expresionRegular);
            Matcher comparador = patron.matcher(oracion);
            // Buscar cada coincidencia y marcarla
            while (comparador.find()) {
                  System.out.println("Patrón encontrado. Comienza en: "
                              + comparador.start() + " y termina en " +
                              comparador.end());
                  Arrays.fill(marcador, comparador.start(), comparador.end(),
                              '^');
            }
            // Mostrar el resultado con las marcas generadas
            System.out.println(oracion);
            System.out.println(marcador);
      }
}
```

El resultado ahora obtenido es:

```
Patrón encontrado. Comienza en: 0 y termina en 3
Patrón encontrado. Comienza en: 5 y termina en 8
Patrón encontrado. Comienza en: 22 y termina en 25
Patrón encontrado. Comienza en: 27 y termina en 30
Patrón encontrado. Comienza en: 40 y termina en 43
Patrón encontrado. Comienza en: 75 y termina en 78
Sólo soldados bajo el SOL, sólo humanos solitarios en la tristeza que deja solamente la guerra
^^^  ^^^             ^^^  ^^^          ^^^                              ^^^
```

Operadores lógicos en las expresiones regulares

Usted puede utilizar el operador && para combinar las clases que definen los conjuntos de caracteres. Esto es particularmente útil cuando se utiliza en combinación con el operador de negación, ^, que aparece en la segunda fila de la tabla en la sección anterior. Por ejemplo, si se desea especificar que cualquier consonante minúscula es una coincidencia, se podría escribir la expresión regular como:

[b-df-hj-np-tv-z]

Sin embargo, esto no sólo no es claro sino que además es trabajoso. Utilizando operadores lógicos para definir la expresión es mucho más simple para definir coincidencias:

[a-z && [^ aeiou]]

Cuando se escribe una expresión como la anterior, se específica que la coincidencia válida se considera para cualquier carácter desde a hasta z pero que ADEMÁS NO sean la vocales entre los corchetes. El operador && determina la intersección de ambas condiciones al igual que lo hace en una expresión lógica del lenguaje.

Análogamente existe el operador | (OR) que funciona como el "o" incluyente. Los operadores relaciones se pueden combinar de la manera que resulte más conveniente para generar condiciones sobre los caracteres de las expresiones regulares.

Conjuntos predefinidos de caracteres

También se puede utilizar una serie de caracteres predefinidos que proporcionan una notación abreviada para los conjuntos comúnmente utilizados. La siguiente tabla muestra algunos de ellos.

Caracter	Descripción
.	Representa cualquier carácter, como ya se ha visto.
\d	Representa cualquier dígito y por lo tanto, es la abreviatura de [0-9].
\D	Representa cualquier carácter que no es un dígito. Por tanto, es equivalente a [^ 0-9].
\s	Representa cualquier espacio en blanco. Se entiende como tal a una tabulación '\ t', un carácter de nueva línea '\ n', un carácter de avance de línea '\ f', un retorno de carro '\ r', o un salto de página 'x0B'.
\S	Representa cualquier carácter que no sea un blanco y por lo tanto equivalente a [^ \ s].
\w	Esto representa a un posible caracter en una palabra, que corresponde a una letra mayúscula o minúscula, un dígito o guion bajo. Por tanto, es equivalente a [a-zA-Z_0-9].
\W	Esto representa cualquier carácter que no es un caracter en una palabra, por lo que es equivalente a [^ \ w].

El tratamiento de cadenas que tiene Java obliga a tratar los caracteres especiales de las expresiones regulares de manera diferente a la que se puede realizar en otros entornos. Por ejemplo, el uso de la barra invertida en el lenguaje indica un carácter de escape, con lo cual, para indicar una barra invertida en la expresión regular, la misma se debe duplicar para que el lenguaje la interprete como una sola cuando analiza la cadena en tiempo de compilación. De esta manera, si se quieren indicar tres dígitos, la cadena de la expresión regular tendría el siguiente formato:

"\\d\\d\\d"

Otro ejemplo importante es jugar con los espacios en blanco para determinar longitudes exactas en las palabras. En los ejemplos anteriores se buscaron coincidencias con respecto de la cadena "sol" en diferentes combinaciones, pero si se quiere buscar palabras de tres letras que empiecen con una S y terminen con L, se pueden utilizar los caracteres en blanco para delimitarla y así obtener coincidencias exactas:

"\\sS.L[\\s,]"

Ejemplo

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ProbarRegex3 {
	public static void main(String args[]) {
		// Una expresión regular y una cadena en la cual buscar
```

```
            String expresionRegular = "\\sS.L[\\s|,]";
            String oracion = "Sólo soldados bajo el SOL, sólo humanos" +
                  "solitarios en la tristeza que deja solamente la guerra";
            // Se marcan los lugares donde se encuentra el patrón
            // Funciona bien si la letra es siempre del mismo tamaño
            char[] marcador = new char[oracion.length()];
      // Llenar la cadena de espacios para luego colocar el caracter que marca
            Arrays.fill(marcador, ' ');

            // Obtener el compador requerido
            Pattern patron = Pattern.compile(expresionRegular);
            Matcher comparador = patron.matcher(oracion);
            // Buscar cada coincidencia y marcarla
            while (comparador.find()) {
                  System.out.println("Patrón encontrado. Comienza en: "
                              + comparador.start() + " y termina en " +
                              comparador.end());
                  Arrays.fill(marcador, comparador.start(), comparador.end(),
                              '^');
            }
            // Mostrar el resultado con las marcas generadas
            System.out.println(oracion);
            System.out.println(marcador);
      }
}
```

La salida obtenida es:

```
Patrón encontrado. Comienza en: 21 y termina en 26
Sólo soldados bajo el SOL, sólo humanos solitarios en la tristeza que deja solamente la guerra
                    ^^^^^
```

Notar que la comparación se realiza con los cinco caracteres definidos, por lo tanto *todos* son parte de la coincidencia.

Límites en las coincidencias

Hasta ahora, se ha tratado de encontrar la presencia de un patrón en cualquier parte de una cadena. En muchas situaciones se debe ser más específico. Es posible intentar buscar un patrón que aparezca en el comienzo de una línea en una cadena, pero no en cualquier otro lugar, o tal vez sólo en el final de ésta. Como se vio en el ejemplo anterior, puede existir la necesidad de encontrar una coincidencia exacta, que sea por ejemplo “sol” y no “solamente”. La expresión regular se puede simplificar utilizando un conjunto de meta caracteres específicos. Algunos de ellos son los que se muestran en la siguiente tabla.

Caracter	Descripción
^	Especifica el comienzo de una línea. Por ejemplo, para encontrar la palabra Java en el principio de cualquier línea se puede utilizar la expresión "^Java".
$	Especifica el final de una línea. Por ejemplo, para encontrar la palabra Java al final de cada línea se puede utilizar la expresión "Java$". Por supuesto, si se espera un punto al final de una línea de la expresión sería "Java\\.$".

\b	Especifica el límite en una palabra. Para encontrar palabras de tres letras que comienzan con "s" y terminan con 'l', se puede utilizar la expresión "\\bs.l\\b".
\B	Las palabras no son un límite - el complemento de \b.
\A	Especifica el inicio de la cadena buscada. Para encontrar la palabra El, al principio de una cadena en la que se busca, se puede utilizar la expresión "\\AEl\\b". El \\b en el extremo de la expresión regular es necesario para evitar encontrar una coincidencia como Elegido o Elías al comienzo de la cadena.
\z	Especifica el final de una cadena que se busca. Para encontrar la palabra “eliminación” seguida de un punto al final de una cadena, se puede usar la expresión "\\beliminación\\.\\z".
\Z	El extremo final de la cadena a analizar, excepto para el carácter terminador final. Un terminador final es un carácter de nueva línea ('\n') si se establece Pattern.UNlx_LlNES. De lo contrario, también puede ser un retorno de carro ('\r'), un retorno de carro seguido de un carácter de nueva línea, un carácter de nueva línea ('\u0085'), un separador de línea ('\u2028'), o un separador de párrafo ('\u2029').

Uso de cuantificadores

Los cuantificadores permiten identificar caracteres que se repiten según un formato especificado creando una secuencia de caracteres. Por ejemplo, un valor numérico es una secuencia de números que se puede describir mediante un cuantificador. Si se quisiera buscar un valor numérico en una cadena, por ejemplo un entero que es el caso más simple, se puede utilizar como cuantificador el meta caracter “+”. De esta manera, sumando a que cada dígito se puede especificar el la expresión regular como”\\d”, una secuencia de dígitos se puede especificar como “\\d+”.

Para casos más complejos, un número puede incluir una coma decimal. Sin embargo, los números decimales pueden incluir o no una coma decimal, pero será una única ocurrencia. Por lo tanto esta puede ocurrir una vez o ninguna por cada secuencia de números que se analice. Además, la secuencia poseerá números luego de la coma. Esto se puede resolver mediante el meta caracter “?” y generar una expresión regular como la siguiente:

“\\d+.?\\d+”

Se puede leer la expresión anterior como: “se busca una secuencia de números en la cual puede o no aparecer un punto y luego de él se pueden encontrar más dígitos”. Esta expresión se puede mejorar aún más. Como la parte decimal puede ser opcional, se pueden utilizar paréntesis como meta caracteres y combinarlos con el signo de pregunta para indicar que su aparición puede ser opcional:

“\\d(\\.\\d+)?”

Esta expresión indica que una coincidencia válida es si la cadena posee una secuencia de caracteres numéricos y también si opcionalmente aparece un punto seguido de más dígitos. Para

mejorar la capacidad de encontrar coincidencias, se puede agregar que se consideren válidas las ocurrencias con signo, aparezca el mismo o no. De esta manera, la expresión regular queda:

"[+|-]?\\d+(\\.\\d+)?"

Si la cadena tiene un número que empieza con una coma (esto es, da la parte entera como un cero supuesto por la omisión, ej: ,23 en lugar de 0,23), la expresión anterior no sirve. En este caso hay que considerar que:

- Puede empezar con un signo
- Puede empezar con una coma
- Puede ser un número sin parte decimal
- Puede ser un número con parte decimal y coma

Para que un número empiece con coma y signo, la expresión regular sería:

"[+|-]?\\.\\d+"

Esta expresión encontraría como ocurrencia aquellos números que comiencen con una coma y muestres sólo su parte decimal (como ,23). Se debe combinar esta opción con la anterior para cumplir con todos los puntos enumerados:

"[+|-]?(\\d+(\\.\\d+)?)|\\.\\d+)"

Notar que se resolvió como si fuera una expresión aritmética para combinar ambas expresiones. Se saco el factor común que es la resolución del signo fuera de los paréntesis y luego se colocó ambas expresiones unidas mediante el operador lógico OR.

Ejemplo

```
package patrones;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class NumerosYRegEx {
	public static void main(String args[]) {
		String regEx = "[+|-]?(\\d+(\\.\\d+)?)|(\\.\\d+)";
		String str= "256 es el cuadrado de 16 y -2.5 al cuadrado es 6.25 "+
		"y -.243 es menor que 0.1234.";
		Pattern pattern = Pattern.compile(regEx);
		Matcher m = pattern.matcher(str);
		while (m.find ()) {
			System.out.println(m.group());
		}
	}
}
```

La salida que se obtiene es:

```
256
```

```
16
-2.5
6.25
.243
0.1234
```