Ristras de caracteres

Introducción

Cuando un programa necesita procesar información textual, por ejemplo, para mostrar un mensaje, almacenar un nombre, una dirección, la descripción de un producto, etc., se recurre a las ristras de caracteres. Una ristra es un tipo de datos que representa textos como una sucesión de caracteres puestos uno detrás de otro (una secuencia).

Las ristras de caracteres tienen una longitud, que es el número de caracteres que la forman; un orden interno en virtud del cual cada carácter ocupa un posición determinada, lo que implica que dos ristras solo son iguales si tienen los mismos caracteres en el mismo orden; y son comparables, lo que quiere decir que una ristra puede ser menor, igual o mayor que otra. Especial mención, por su importancia en determinadas operaciones de manipulación de ristras, merece la ristra vacía, que es la que no tiene ningún carácter.

La clase *String* de Java

Declaración y propiedades básicas

Las ristras de caracteres se implementan en Java, preferentemente, como objetos de la clase String:

```
String msg = "Hola mundo";
```

Los valores literales de las ristras se representan encerrándolos entre comillas dobles; de esta manera, no se confunden con nombres de variables o palabras reservadas (que son secuencias de caracteres, pero sin comillas), ni con

valores de tipo carácter, que se encierran entre comillas simples ("a" es una ristra de caracteres compuesta por un único carácter, 'a').

La longitud de la ristra representada por un objeto de la clase *String* puede conocerse usando el método *length*, como muestra el ejemplo:

```
int msgLength = msg.length(); // msgLength toma el valor 10
```

El carácter que ocupa una posición determinada de la ristra representa por un objeto de la clase *String* se puede conocer mediante el método *charAt* (en Java las posiciones de una ristra se numeran empezando en **cero**):

```
char c = msg.charAt(3); // c toma el valor 'a';
```

Comparación de objetos de la clase *String*

La comparación entre ristras de caracteres se basa en los caracteres que las forman. Dos ristras son iguales si tienen exactamente los mismos caracteres en el mismo orden. Si no es así, es menor la que, al ir comparando carácter a carácter desde el principio, tenga un carácter menor, o la que tenga menor longitud si todos sus caracteres coinciden con los del inicio de la más larga. El orden entre los caracteres viene dado por su codificación binaria.

Las ristras representadas como objetos de la clase *String* de Java se pueden comparar, pero no usando los operadores relacionales (==, !=, <, >, <=, >=), sino métodos específicos. El operador == sí puede aplicarse entre variables *String*, pero sólo dará un resultado *true* cuando sus dos operandos están referenciando al mismo objeto. Cuando referencien objetos diferentes, aunque ambos representen la misma ristra, el resultado será *false*.

Para comparar si dos objetos de la clase *String* representan, o no, la misma ristra se usa el método *equals*:

```
if (msg.equals("Hola mundo")) …
```

Existe otro método, *equalsIgnoreCase*, que hace lo mismo que el método *equals*, pero ignorando las diferencias entre mayúsculas y minúsculas (es decir, por ejemplo, el carácter 'A' se considera igual que el carácter 'a').

Si lo que queremos es saber si una ristra es menor o mayor que otra, tendríamos que usar el método *compareTo*:

```
if (msg.compareTo("Hola mundo") < 0) …
```

El método *compareTo* trabaja con dos objetos *String*; supongamos que nos referimos a ellos como *objA* y *objB*. La llamada a *compareTo* toma la forma: *objA.compareTo(objB)*, con *objA* asociado al método como prefijo y *ObjB* como parámetro (en el ejemplo, *objA* corresponde a *msg* y *objB* a *"Hola mundo"*).

El método *compareTo* devuelve un valor de tipo *int*. Si el valor devuelto es cero, significa que las ristras representadas por los dos objetos comparados son iguales. Si es negativo, significa que la ristra representada por *objA* es menor que la representada por *objB*; y si es positivo significa que la ristra representada por *objA* es mayor que la representada por *objB*.

El valor devuelto por el método *compareTo* se calcula comparando los caracteres que ocupan la misma posición en las dos ristras, empezando por el principio. Si se acaban los caracteres de una de las ristras, o de ambas, sin encontrar ninguno diferente, se devuelve la diferencia de longitudes (que, si ambas ristras son iguales, será cero). Si al ir comparando, todos los caracteres van siendo iguales, pero al llegar a una posición, *i*, se encuentra una diferencia, entonces el valor devuelto es:

```
objA.charAt(i) – objB.charAt(i)
```

Existe otro método, *compareToIgnoreCase*, que hace lo mismo que el método *compareTo*, pero ignorando las diferencias entre mayúsculas y minúsculas (es decir, por ejemplo, el carácter 'A' se considera igual que el carácter 'a').

Operaciones básicas de la clase String

Concatenación

La concatenación es una operación básica de manejo de ristras que permite construir una ristra juntando otras preexistentes:

```
String nombre = "Juan";
String apellido1 = "Pérez";
String apellido2 = "Rodríguez";
String nombreCompleto = nombre + ' ' + apellido1 + ' ' + apellido2;
```

En Java, la concatenación de objetos *String* se expresa con el operador +, como se puede ver en el ejemplo. El ejemplo también muestra que no solo es posible concatenar objetos *String* con otros objetos *String*, incluyendo literales, sino que también es posible concatenar objetos *String* con objetos de otros tipos, caracteres en este caso (automáticamente se obtiene una representación como ristra del objeto en cuestión, que es la que se concatena). El resultado de la concatenación del ejemplo es un nuevo objeto *String* que representa el valor:

```
"Juan Pérez Rodríguez"
```

Nótese que los espacios entre el nombre y los apellidos hay que ponerlos explícitamente. La concatenación:

```
String nombreCompleto = nombre + apellido1 + apellido2;
```

da como resultado:

```
"JuanPérezRodríguez"
```

La concatenación no es conmutativa (el orden en que se unen las ristras es relevante). Es asociativa:

```
(s1 + s2) + s3 = s1 + (s2 + s3)
```

También tiene elemento neutro, que es la ristra vacía (""):

```
s + ""  = "" + s = s
```

Localización de una subristra

Podemos usar el método *contains* para ver si una ristra contiene una secuencia determinada de caracteres (lo que llamamos una *subristra*):

```
Posiciones>  01234567890123456789012345678901234567
String s1 = "23894475843108235128031023512375239875";
boolean result = s1.contains("75"); // result toma el valor true
```

Pero si nos interesa saber exactamente en qué posición se encuentra, usaremos el método *indexOf*:

```
Posiciones>  01234567890123456789012345678901234567
String s1 = "2389447584310823512803102351237523987 5";
int pos = s1.indexOf("75"); // pos toma el valor 6
```

El método *indexOf*, si solo se le pasa la *subristra* a buscar, devuelve la posición de comienzo de la primera aparición. Si queremos buscar una aparición posterior, basta con añadir en la llamada al método *indexOf* un segundo parámetro que indique la posición a partir de la cual comenzar la búsqueda. En el siguiente ejemplo se busca la primera a parición de la subristra "75" en una posición igual o posterior a la 7:

```
Posiciones>  01234567890123456789012345678901234567
String s1 = "2389447584310823512803102351237523987 5";
int pos = s1.indexOf("75", 7); // pos toma el valor 30
```

Si lo que queremos es buscar la última aparición, usaremos el método *lastIndexOf*:

```
Posiciones>  01234567890123456789012345678901234567
String s1 = "2389447584310823512803102351237523987 5";
int pos = s1.lastIndexOf("75"); // pos toma el valor 36
```

De forma simétrica a lo que ocurre con *indexOf*, la adición de un parámetro al método *lastIndexOf* permite buscar "la última aparición en una posición igual o anterior a una posición dada":

```
Posiciones>  01234567890123456789012345678901234567
String s1 = "23894475843108235128031023512375239875";
int pos = s1.lastIndexOf("75", 35); // pos toma el valor 30
```

Tanto el método *indexOf* como el *lastIntexOf* devuelven el valor -1 si la ristra buscada no se encuentra en la ristra de búsqueda.

Copia de una subristra

El método *substring* permite crear un nuevo objeto *String* copiando un trozo (una subristra) de una ristra. Para hacerlo se ha de indicar la posición de comienzo y la **posición siguiente** a la última, de la *subristra* a copiar.

```
Posiciones>  01234567890123456789012345678901234567
String s1 = "2389447843108235128031023512375239875";
String s2 = s1.substring(7, 14); // "5843108"
```

Otras operaciones útiles para manipular ristras

La clase *String* ofrece muchas más operaciones, de las cuales, en este momento, creemos interesante destacar las siguientes:

String	**concat(String str)**
	Concatena la ristra especificada al final de la actual ristra.

String	**copyValueOf(char[] data)**
	Devuelve una ristra que representa la secuencia de caracteres del array.

boolean	**endsWith(String sufijo)**
	Comprueba si la ristra actual finaliza con la ristra sufijo.

void	**getChar(int srcBegin, int srcEnd, char[] dst,int dstBegin)**
	Copia caracteres de la ristra actual a un array de caracteres.
int	**indexOf(int char)**
	Devuelve el índice de la primera aparición del carácter especificado.
int	**indexOf(int char, int fromIndex)**
	Devuelve el índice de la primera aparición del carácter especificado, comenzando la búsqueda en el índice especificado.
boolean	**isEmpty()**
	Devuelve verdadero si la ristra está vacía, si length() es cero.
String	**replace(CharSequence[1] target, Charsequence replacement)**
	Crea un nuevo objeto *String* que tiene el mismo valor que la ristra sobre la que se aplica, pero con todas las subristras que coinciden con target sustituidas por replacement.
String	**replaceAll(String actual, String nueva)**
	Reemplaza en la ristra todas las apariciones de la subristra actual por la subristra nueva.
String[]	**split(String separador)**
	Divide la ristra actual en trozos usando el separador.
boolean	**startsWith(String prefix)**
	Comprueba si la ristra actual comienza con la ristra prefijo.
boolean	**startsWith(String prefix, int índice)**
	Comprueba si la ristra actual comienza con la ristra prefijo en el índice especificado.

[1] CharSequence equivale a "ristra de caracteres"

`CharSequence[]`	`subSequence(int beginIndex, int endIndex)`
	Devuelve una nueva secuencia de caracteres correspondiente al rango de los índices especificados.
`char[]`	`toCharArray()`
	Convierte la ristra actual a un nuevo array de caracteres.
`String`	`toLowerCase()`
	Crea un nuevo objeto *String* cuyo contenido es igual al de la ristra sobre la que se aplica con todos los caracteres convertidos a minúsculas.
`String`	`toUpperCase()`
	Crea un nuevo objeto *String* cuyo contenido es igual al de la ristra sobre la que se aplica con todos los caracteres convertidos a mayúsculas.
`String`	`trim()`
	Crea un nuevo objeto *String* cuyo contenido es igual al de la ristra sobre la que se aplica eliminando los espacios al principio y al final.
`static String`	`valueOf(T b)`
	Siendo T uno de los tipos primitivos: ***int***, ***long***, ***float***, ***double***, ***boolean***, ***char***; devuelve un nuevo objeto *String* que es la representación como ristra del valor de b.