



# Cadenas de caracteres

## Objetivos

- Utilizar el tipo primitivo char.
- Conocer las funcionalidades que proporciona la clase Character para la manipulación de caracteres.
- Usar la clase String como parte de las aplicaciones que se construyan.
- Profundizar en el uso de operaciones avanzadas con texto y realizar implementaciones acordes a los requisitos del sistema.
- Plantear distintas alternativas y elegir la solución óptima en cada caso, según el problema que se necesite resolver.
- Conocer la API de Java, que permite codificar aplicaciones que utilizan datos de tipo texto.
- Exponer las ventajas e inconvenientes de las posibles herramientas que se usan para la creación y manipulación de texto.

## Contenidos

- 6.1. Tipo primitivo char
- 6.2. Clase Character
- 6.3. Clase String
- 6.4. Cadenas y tablas de caracteres

# Introducción

En los programas escritos hasta ahora hemos utilizado los tipos primitivos: `char`, `byte`, `int`, `float`, `double` y `boolean`. Estos bastan para implementar muchas aplicaciones, sobre todo las relacionadas con datos numéricos, pero proporcionan pocas herramientas para trabajar con texto. El tipo `char`, que almacena un solo carácter, es insuficiente para manejar textos complejos.

Por texto entendemos una palabra, una frase e incluso uno o más párrafos de cualquier longitud. En definitiva, un texto, como por ejemplo este mismo párrafo, es una secuencia de caracteres, de ahí que también se le denomine *cadena de caracteres* o, por economía del lenguaje, simplemente *cadena*.

Para manipular textos disponemos en la API de las clases `Character` y `String` —ambas ubicadas en el paquete `java.lang`—, que proporcionan multitud de funcionalidades para trabajar con un solo carácter la primera y con textos de cualquier longitud la segunda.

## ■ 6.1. Tipo primitivo `char`

De forma general un **carácter** se define como una letra —de cualquier alfabeto—, un número, un ideograma o cualquier símbolo. En Java un carácter o literal carácter se escribe entre comillas simples (''). Algunos ejemplos de ellos son: ''p'', ''ñ'', ''Ψ'', ''7'', ''♡'' o ''#''.

### ■ ■ 6.1.1. Unicode

Mediante un teclado podemos escribir ciertos caracteres, como 'a', pero esto no es posible para otros, como '♡'. Para solventar este problema, un conglomerado de empresas fundó el Unicode Consortium, un organismo que, mediante un comité técnico, diseñó y mantiene un estándar de codificación de caracteres denominado **Unicode**.

Este identifica cada carácter mediante un número entero único, llamado *code point*, cuyo valor se puede representar en decimal o en hexadecimal. Para evitar confusión cuando el code point se representa en hexadecimal se le antepone la secuencia U+ o \u, de forma general, aunque Java solo permite la segunda. Otra particularidad de la representación de un code point en hexadecimal es que siempre se utilizan, como mínimo, 4 dígitos, completando con ceros por la izquierda si fuera necesario.

El esquema de codificación Unicode comprende un total de 1 114 112 posibles code points, que según la representación usada tomarán valores en el rango de 0<sub>dec</sub> a 1114111<sub>dec</sub>, en decimal, o valores en el rango de 0<sub>hex</sub> a 10ffff<sub>hex</sub>, en hexadecimal. Lo que requiere un tamaño de 3 bytes para poder albergar todos los posibles valores.

#### Recuerda

El uso de mayúsculas o minúsculas en los números hexadecimales es indiferente. Por lo tanto, el número 1a<sub>hex</sub> es idéntico a 1A<sub>hex</sub>.



A la hora de seleccionar un carácter es posible usar su codificación Unicode o el propio carácter si es posible escribirlo mediante el teclado. Por ejemplo, el carácter 'a' puede escribirse pulsando la tecla adecuada del teclado o mediante su code point en decimal (97) o en hexadecimal (\u0061). Veamos la forma de asignar 'a' a una variable de tipo `char`,

```
char c;
c = 'a'; //directamente mediante el teclado
c = 97; //usando el code point en decimal
c = '\u0061'; //o con el code point en hexadecimal
```

La única forma de designar el carácter '♥' es mediante su code point.

```
char c = '\u2661'; //o bien, c = 9825;
System.out.println(c); //muestra un ♥
```

Para codificar cualquier code point necesitamos 3 bytes, por lo tanto, ¿cómo es posible que podamos asignar un code point a un tipo primitivo `char` (2 bytes)? La respuesta es que no todos los code points pueden asignarse a `char`. El problema surge porque, en un principio, los code points de Unicode usaban solamente 2 bytes para codificar todos los caracteres; por lo tanto, el tipo `char` se definió acorde a este tamaño. Con el tiempo, el tamaño del code point ha crecido para poder identificar la enorme cantidad de símbolos que se han ido añadiendo.

En consecuencia, solo los code points cuyo valor es inferior o igual a 65 535 —\uFFFF— pueden asignarse a una variable de tipo `char`. Para valores mayores de code point hemos de utilizar variables de tipo `int`, que tiene un tamaño de 4 bytes y dispone de espacio suficiente para albergar cualquier code point. Como veremos en el Apartado 6.1.3, existe una fuerte relación entre los tipos `char` e `int`.

Para conocer la codificación de cualquier carácter necesitamos recurrir a las tablas diseñadas por el Unicode Consortium o bien a las bases de datos de caracteres —véase a modo de ejemplo la Tabla 6.1—. Estas tablas agrupan los caracteres según el alfabeto (latino, braille, etc.) o por el conjunto de símbolos al que pertenecen (matemáticos, jeroglíficos egipcios, etc.). Por ejemplo, si deseamos trabajar en Java con el ideograma ♥, lo primero que tenemos que hacer es buscar su code point en las tablas de caracteres, donde encontramos que el code point que lo identifica es 9825 o \u2661.

**Tabla 6.1.** Ejemplos de codificación Unicode

Carácter	Code point	
	decimal	hexadecimal
A	65	\u0041
a	97	\u0061
Ψ	936	\u03A8
♥	9825	\u2661
7	55	\u0037

## Argot técnico



Otra solución que implementa Java para el problema de que el tipo `char` es demasiado pequeño para albergar todos los símbolos Unicode consiste en codificar los code points en dos variables de tipo `char`, normalmente una tabla de tamaño 2.

### 6.1.2. Secuencias de escape

Un carácter precedido de una barra invertida (\) se conoce como *secuencia de escape*. Al igual que los caracteres escritos mediante la codificación Unicode, representan un único carácter, pero en este caso poseen un significado especial. En la Tabla 6.2 se muestran las secuencias de escape de Java.

**Tabla 6.2.** Caracteres especiales en Java: su nombre y secuencia de escape

Carácter	Nombre
\b	Borrado a la izquierda
\n	Nueva línea
\r	Retorno de carro
\t	Tabulador
\f	Nueva página
\'	Comilla simple
\"	Comilla doble
\\\	Barra invertida

Veamos algunos ejemplos:

```
char c = '\\';
System.out.println(c); //muestra una comilla simple: '
c = '\"';
System.out.println(c); //muestra una comilla doble: "
c = '\t'; //tabulador. Al ser invisible lo representamos con _____
System.out.println("1" + c + "2"); //muestra "1_____2";
```

### 6.1.3. Conversión char ↔ int

Cada code point no es más que un número entero, representado en decimal o en hexadecimal. El hecho de que un carácter se identifique con un número crea una estrecha relación entre el tipo `char` y el tipo `int`. Es posible asignar un valor entero a una variable de tipo `char` (siempre y cuando el valor del entero esté comprendido entre 0 y 65 535) y asignar un carácter a una variable de tipo `int`, ya que Java se encarga de realizar las conversiones oportunas (el tipo `int` representa los números en decimal por defecto).

Veamos un ejemplo: el carácter 'a' tiene asociado el code point \u0061. Si convertimos el número hexadecimal 0061<sub>hex</sub> a decimal, obtenemos 97<sub>dec</sub>.

Aprovechando este mecanismo de conversión podemos realizar asignaciones de la forma:

```
int e = 'a'; //asigna un carácter a una variable int  
System.out.println(e); //muestra 97  
e = '\u0061'; //asigna un carácter a una variable int  
System.out.println(e); //muestra 97  
char c = 97; //asigna un entero a una variable char  
System.out.println(c); //muestra 'a'
```

También es posible forzar una conversión por medio de un cast.

```
char c = 'a';  
System.out.println((int)c); //muestra 97  
int e = 97;  
System.out.println((char) e); //muestra 'a'
```

Se pueden realizar asignaciones de variables del tipo `int` a `char` con un cast.

```
int e = 97;  
char c = (char) e; //c vale 'a'
```

En este último caso hemos hecho una conversión de estrechamiento, ya que `char` se codifica en 2 bytes e `int` necesita 4 bytes. En la variable `c` se guardan los 2 bytes menos significativos de `e`.

Por otra parte,

```
char c = 'a';  
int e = c; //e vale 97
```

Aquí no es necesario el cast porque la conversión es de ensanchamiento.

## ■ ■ 6.1.4. Aritmética de caracteres

La relación existente entre un carácter y su representación numérica en Unicode, ya sea en hexadecimal o en decimal, permite realizar operaciones aritméticas con ellos. En realidad, no es posible operar con un carácter, sino con su representación numérica. Veamos como ejemplo la suma:

```
System.out.println('a' + 1); //se muestra una 'b' en la pantalla
```

Para poder realizar la suma, Java transforma el carácter en su representación numérica, sea en hexadecimal o en decimal, ambas representan el mismo valor. La operación quedaría así:

$$'a' + 1 = 61_{\text{hex}} + 1 = 62_{\text{hex}} = 98_{\text{dec}}$$

Es importante entender que el número obtenido (98) puede ser interpretado, a su vez, como un carácter.

```
char c = 98; //98 es el valor Unicode de la letra b
```

Otra forma de entender la aritmética de caracteres consiste en que, al realizar una suma o una resta, por ejemplo '`x`' ± `n`, el resultado es el carácter situado `n` posiciones delante o detrás, en la codificación Unicode del carácter con el que estamos operando. Veamos un ejemplo en Java:

```
char c = 'e' - 2; //vale 'c'  
c = 'e' + 2; //vale 'g'
```

Es decir, la letra `e` tiene en el código Unicode dos puestos por delante la letra `g` y dos puestos por detrás la letra `c`.

Este comportamiento permite transformar un carácter de minúscula a mayúscula y viceversa.

```
char c = 'h' + 'A' - 'a';  
System.out.println(c) //muestra 'H'
```

`'a' - 'A'` representa la distancia que existe en el código Unicode entre las letras minúsculas y las mayúsculas.

## Actividad resuelta 6.1

Escribir un programa que muestre todos los caracteres Unicode junto a su code point, cuyo valor esté comprendido entre `\u0000` y `\uFFFF`.

### Solución

```
/* Aprovechando la aritmética de caracteres mostraremos todos los símbolos  
 * disponibles en la codificación Unicode, comprendidos entre \u0000 y \uFFFF. */  
public class Main {  
    public static void main(String[] args) {  
        //usamos números en base hexadecimal, lo que se indica anteponiendo 0x  
        //internamente la variable codePoint contiene valores decimales  
        for(int codePoint = 0x0000; codePoint <= 0xFFFF; codePoint++) {  
            String xxxx = Integer.toHexString(codePoint); //convierte un número en su  
            //representación hexadecimal  
            System.out.println("\u" + xxxx + ":" + (char)codePoint);  
        }  
    }  
}
```

## ■ 6.2. Clase Character

El tipo `char` es a todas luces insuficiente, por sí solo, para realizar operaciones con caracteres. La clase `Character` amplía su funcionalidad para trabajar con caracteres simplificando mucho el trabajo. Pensemos en lo engorroso que puede ser, por ejemplo, decidir si un carácter dado es una letra minúscula: para ello, tendríamos que realizar una serie de comprobaciones. Por el contrario, esta funcionalidad se encuentra en `Character`, que dispone de una batería de métodos estáticos, útiles para clasificar y convertir valores de tipo `char`.



## Nota técnica

En esta unidad usamos la clase `Character` que proporciona, mediante sus métodos, una serie de herramientas. Sin embargo, la clase `Character` es un **wrapper** o clase **envoltorio** para el tipo primitivo `char`.

En la Unidad 7 estudiaremos en profundidad las clases.

En la web de la Editorial Paraninfo existe, como recurso digital, un anexo con la descripción de los wrappers.

### 6.2.1. Clasificación de caracteres

Un carácter puede clasificarse dentro de algunos de los grupos siguientes:

- **Dígitos:** este grupo está formado por los caracteres '0', '1' ..., '9'.
- **Letras:** formado por todos los elementos del alfabeto, tanto en minúscula ('a', 'b'...) como en mayúscula ('A', 'B'...).
- **Caracteres blancos:** como el espacio o el tabulador, entre otros.
- **Otros caracteres:** signos de puntuación, matemáticos, etcétera.

## Argot técnico



Como *letras* hemos considerado el alfabeto latino (a, b, c... tanto en mayúsculas como minúsculas), pero en realidad se incluyen las letras de cualquier alfabeto.

Los métodos de `Character` para verificar si un carácter pertenece a alguno de estos grupos devuelven un booleano: `true` en caso de que pertenezca o `false` en caso contrario. Estos métodos son:

- `boolean isDigit(char c)`: indica si el carácter `c` es un dígito. Devuelve `true` en caso afirmativo y `false` en caso contrario.

```
char c1 = '8', c2 = 'p';
boolean b;
b = Character.isDigit(c1); //b vale true, ya que '8' es un dígito
b = Character.isDigit(c2); //b es false, 'p' no es un dígito
```

- `boolean isLetter(char c)`: determina si el carácter pasado como parámetro es una letra (minúscula o mayúscula).

```
Character.isLetter('8'); //false: el carácter '8' no es una letra
Character.isLetter('e'); //true: el carácter 'e' sí es una letra
```

- `boolean isLetterOrDigit(char c)`: indica si el carácter es una letra o un dígito. El conjunto de estos caracteres se conoce como caracteres *alfanuméricos*.

```
boolean b;
b = Character.isLetterOrDigit('%'); //false: '%' no es alfanumérico
```

```
b = Character.isLetterOrDigit('p'); //true: 'p' es una letra
b = Character.isLetterOrDigit('2'); //true: '2' es un dígito
```

- `boolean isLowerCase(char c)`: especifica si `c` es una letra y, además, está en minúscula.

```
char c1 = 'q', c2 = 'Q';
Character.isLowerCase('*'); //false: ni siquiera es una letra
Character.isLowerCase(c1); //true: es una letra en minúscula
Character.isLowerCase(c2); //false: es una letra, pero no minúscula
```

- `boolean isUpperCase(char c)`: funciona igual que el método anterior, pero indicando si el carácter es una letra mayúscula.

```
Character.isUpperCase('t'); //false
Character.isUpperCase('T'); //true
```

- `boolean isSpaceChar(char c)`: devolverá `true` si el carácter utilizado como parámetro de entrada es el espacio (' '), que se consigue pulsando en la barra espaciadora. Como no se ve, lo representaremos de la forma: ' '\_'. En la bibliografía es habitual encontrar otras representaciones, como por ejemplo una letra b tachada ('b') para simbolizar un espacio en blanco.

```
Character.isSpaceChar('_'); //devuelve true
Character.isSpaceChar('a'); //false: es obvio que no es un espacio
```

- `boolean isWhitespace(char c)`: amplía el método anterior y determina si el carácter pasado es cualquier carácter blanco. Los caracteres que hacen que el método devuelva `true` son:

- Espacio en blanco (' '\_'): se teclea mediante la barra espaciadora.
- Retorno de carro ('\r'): dependiendo del sistema operativo, este carácter tendrá distinto comportamiento al imprimirse.
- Nueva línea ('\n'): es el carácter que se consigue al pulsar la tecla Intro.
- Tabulador ('\t'): equivale a varios espacios en blanco. Se genera con la tecla Tab.
- Otros: existen otros caracteres considerados blancos como el tabulador vertical, el separador de ficheros, etc., aunque están en desuso.

Veamos un ejemplo:

```
char c = '\t';
Character.isWhiteSpace(c); //true: tabulador
Character.isWhiteSpace('\n'); //true: carácter nueva línea
Character.isWhiteSpace('a'); //false: evidentemente no es un blanco
```

## 6.2.2. Conversión

Los métodos que realizan conversiones son aquellos que devuelven transformado el valor que se les pasa como parámetro, normalmente un carácter, en otro carácter o en un valor de un tipo distinto. También existen los que realizan la operación inversa, es decir, convierten un valor de otro tipo en un carácter.

## ■■■ Conversión entre caracteres

Son los métodos que transforman un carácter en otro. Cuando la conversión no es posible, por ejemplo, no se puede transformar un número a mayúscula, se devuelve el mismo carácter pasado como parámetro. Disponemos de los siguientes métodos:

- `char toLowerCase(char c)`: si el carácter pasado es una letra, lo devuelve convertido a minúscula. En otro caso, devuelve el mismo.

```
char c1 = 'A', c2;  
c2 = Character.toLowerCase(c1); //la variable c2 toma el valor 'a'  
c2 = Character.toLowerCase('3'); //al no ser una letra, devuelve el  
//mismo valor pasado: '3'
```

- `char toUpperCase(char c)`: similar al anterior método, pero convierte el carácter, si es una letra, a mayúscula. En otro caso devuelve el mismo carácter.

```
char c1 = 'g';  
char c2 = Character.toUpperCase(c1); //a c2 se le asigna el valor 'G'
```

## ■ 6.3. Clase String

Las cadenas, conjuntos secuenciales de caracteres, se manipulan mediante la clase `String`, que funciona de forma dual. Por un lado, de manera general, tiene un funcionamiento no estático; pero a su vez, dispone de algunos métodos que sí lo son. Es posible definir variables de tipo `String` de la forma habitual:

```
String cad; //cad es una variable de tipo cadena
```

Una variable de tipo `String` almacenará una cadena de caracteres, que provendrá de la manipulación de otra cadena o de un literal. Un literal cadena consiste en un texto entre comillas dobles (""). Es posible utilizar cualquier carácter, incluidos los codificados mediante Unicode y las secuencias de escape. Algunos ejemplos de literal cadena son:

```
"Hola\n"  
"En un lugar de la mancha"  
"Un corazón: \u2661"
```

Los literales carácter y cadena se diferencian en el tipo de comillas utilizado; mientras 'a' es un carácter, "a" es una cadena que está compuesta por un único carácter.

Con una cadena de caracteres se pueden realizar multitud de operaciones. Algunas trabajan con la cadena como un todo y otras trabajan carácter a carácter.

### ■■■ 6.3.1. Inicialización de cadenas

De forma análoga a como lo hacemos con la clase `Scanner`, podemos utilizar `new` para crear y asignar un valor a una variable de tipo `String`.

```
cad = new String("literal cadena");
```

Sin embargo, el uso de la clase `String` es tan habitual que Java permite una forma abreviada, funcionalmente idéntica a la anterior:

```
cad = "literal cadena";
```

Ambas formas son equivalentes. Por economía en la escritura del código, utilizaremos habitualmente la segunda forma de asignación.

Cuando queramos usar comillas (") dentro de un literal cadena, dado que es el carácter que inicia y finaliza un texto, disponemos de la secuencia de escape \". Un ejemplo:

```
String cad = "Mi perro \"Perico\" es de color blanco";
System.out.print(cad);
```

que muestra en pantalla: «Mi perro "Perico" es de color blanco».

## Valores de otros tipos

A menudo necesitaremos representar un valor de un tipo primitivo en forma de cadena. Veamos un ejemplo: sea el valor entero 1234 (mil doscientos treinta y cuatro), que podemos representar como la cadena "1234", es decir, la cadena formada por el carácter '1', seguido del carácter '2', el '3' y finalizada con el carácter '4'. De igual manera, podemos representar el valor de cualquier tipo primitivo. El método estático que construye una cadena para representar un valor es:

- `static String valueOf(tipo valor)`: construye y devuelve una cadena con la representación del valor pasado como parámetro. Aquí `tipo` hace referencia a cualquier tipo primitivo. En realidad, el método `valueOf()` es un método sobrecargado para cada tipo de dato primitivo. Veamos varios ejemplos:

```
String cad;
cad = String.valueOf(1234); //cad = "1234"
cad = String.valueOf(-12.34); //cad = "-12.34"
cad = String.valueOf('C'); //cad = "C"
cad = String.valueOf(false); //cad = "false"
```

### Recuerda



No se debe confundir la cadena "1234", formada por cuatro caracteres, cada uno de los cuales ocupa 2 bytes, con el número entero 1234, que se codifica en 4 bytes.

### Argot técnico



El método `valueOf()` de la clase `String`, en realidad, también admite objetos de clases que sean representables por medios de cadenas. Esto está vinculado con el método `toString()`, que veremos en la unidad sobre herencia.

## ■■■ 6.3.2. Comparación

Los operadores de comparación disponibles para números y caracteres igual (==), menor que (<) y mayor que (>) no se encuentran disponibles directamente para comparar cadenas de caracteres, pero en su lugar disponemos de métodos de la clase `String` que realizan las comparaciones oportunas.

### Argot técnico



La comparación de caracteres se realiza según su valor Unicode que, para letras, coincide con el orden alfabético. Por ejemplo, el carácter 'a' es menor alfabéticamente que el carácter 'b', es decir, la comparación 'a' < 'b' es cierta.

## ■■■ Igualdad

Un error común es comparar dos variables de tipo cadena utilizando el operador de comparación (==). Este operador no se puede utilizar con `String` debido a que es una clase y no un tipo primitivo. Por ello, para comparar cadenas usaremos:

- `boolean equals(String otra)`: compara la cadena que invoca el método con otra. El resultado de la comparación se indica devolviendo `true` o `false`, según sean iguales o distintas. Para que las cadenas se consideren iguales deben estar formadas por la misma secuencia de caracteres, distinguiendo mayúsculas de minúsculas. Veamos un ejemplo:

```
String cad1 = "Hola mundo";
String cad2 = "Hola mundo";
String cad3 = "Hola, buenos días"
boolean iguales;
iguales = cad1.equals(cad2); //iguales vale true
iguales = cad1.equals(cad3); //iguales vale false
```

Puede ocurrir que nos interese realizar una comparación, pero sin tener en cuenta las letras mayúsculas y minúsculas, que `equals()` considera distintas. Es decir, poder comparar la cadena «hola» con «HoLa» y que resulten iguales. Para ello, existe el siguiente método:

- `boolean equalsIgnoreCase(String otraCadena)`: funciona igual que `equals()` pero sin distinguir mayúsculas de minúsculas al realizar la comparación. Veamos un ejemplo:

```
String cad1 = "Hola mundo";
String cad2 = "HOLA Mundo";
boolean iguales;
iguales = cad1.equals(cad2); //false, no son iguales
iguales = cad1.equalsIgnoreCase(cad2); //true, sin atender a
                                         //mayúsculas/minúsculas son iguales
```

Los métodos vistos comparan la totalidad de dos cadenas, pero es posible comparar solo una región, o fragmento, de cada cadena. Para ello disponemos de:

- `boolean regionMatches(int inicio, String otraCad, int inicioOtra, int longitud)`: compara dos fragmentos de cadenas: el primero corresponde a la cadena invocante y comienza en el carácter con índice `inicio`; y el segundo corresponde a la cadena `otraCad` y comienza en el carácter con índice `inicioOtra`. Ambos fragmentos tendrán la longitud indicada. El método devuelve `true` o `false` para indicar si las regiones coinciden. Por ejemplo,

```
String cad = "Mi_perro_ladra_mUCHo";
String otra = "Un_bONITO_perro_blanco";
boolean b = cad.regionMatches(3, otra, 10, 5) //cierto
```

compara las regiones “perro” (comienza en el índice 3) de `cad` y “perro” (comienza en el índice 10) de `otra`, ambas de longitud 5.

- `boolean regionMatches(boolean ignora, int ini, String otraCad, int iniOtra, int longitud)`: hace lo mismo que el método anterior con la diferencia de que si el valor del parámetro que `ignora` es `true`, la comprobación se realiza considerando iguales las mayúsculas y minúsculas.

## Comparación alfabética

Otra forma de comparar dos cadenas es alfabéticamente, es decir, según el orden de un diccionario. Una cadena se considera alfabéticamente menor que otra si va antes en un diccionario. El orden lo marca la posición de las letras en el alfabeto. La comparación se lleva a cabo mirando el primer carácter distinto de cada cadena; si, por ejemplo, comparamos “monitor” y “monzón”, la comparación se realiza mirando el cuarto carácter, con índice 3, de cada cadena, que es el primer carácter distinto. Si no hay un carácter distinto pero una cadena es más corta, esta es la menor. Por ejemplo, “monito” va antes que “monitor”.

### Recuerda



El orden de los caracteres los marca su valor Unicode, que para las letras coinciden con el orden alfabético. Las letras mayúsculas son anteriores a las minúsculas.

Los métodos disponibles para comparar cadenas alfabéticamente son:

- `int compareTo(String cadena)`: compara alfabéticamente la cadena invocante y la que se pasa como parámetro, devolviendo un entero cuyo valor determina el orden de las cadenas de la forma:
    - 0: si las cadenas comparadas son exactamente iguales.
    - negativo: si la cadena invocante es menor alfabéticamente que la cadena pasada como parámetro, es decir, va antes por orden alfabético.
    - positivo: si la cadena invocante es mayor alfabéticamente que la cadena pasada, es decir, va después.
- ```
String cad1 = "Alondra";
String cad2 = "Nutria";
```

```

String cad3 = "Zorro";
System.out.println(cad2.compareTo(cad1)) //valor mayor que 0
//"Nutria" está después que "Alondra" alfabéticamente
System.out.println(cad2.compareTo(cad3)) //valor menor que 0
//"Nutria" está antes que "Zorro" alfabéticamente

```

- `int compareToIgnoreCase(String cadena)`: realiza una comparación alfabética sin distinguir entre letras mayúsculas ni minúsculas.

### 6.3.3. Concatenación

El operador `+` sirve para unir o concatenar dos cadenas. Veamos su funcionamiento con un ejemplo:

```

String nombre = "Miguel";
String apellidos = "de_Cervantes_Saavedra";
String nombreCompleto = nombre + apellidos;
System.out.println(nombreCompleto); //"Miguel de Cervantes Saavedra"

```

La concatenación une dos cadenas, pero no inserta nada entre ellas. En nuestro ejemplo, el nombre está completamente pegado a los apellidos. Esto puede evitarse haciendo

```

String nombreCompleto = nombre + " " + apellidos;
System.out.println(nombreCompleto); //"Miguel de Cervantes Saavedra"

```

La conversión de datos de tipo primitivo a tipo `String` permite concatenarlos a una cadena. Esta conversión la realiza Java automáticamente y de forma transparente al programador. Veamos algunos ejemplos:

```

String a = "Resultado: " + 3; //a = "Resultado: 3"
String b = "Resultado: " + true; //b = "Resultado: true"
String c = "Resultado: " + 'a'; //c = "Resultado: a"

```

También es posible usar el operador `+=` para la concatenación de cadenas. Otra forma de concatenar cadenas, idéntica al operador `+`, es mediante el método `concat()`, aunque rara vez se usa.

### 6.3.4. Obtención de caracteres

Todos los caracteres que forman una cadena pueden ser identificados mediante la posición que ocupan, al igual que los elementos de una tabla. Cada carácter se numera con un índice único que comienza en 0. En la Tabla 6.3 se incluye un ejemplo con una cadena junto a los índices que identifican a cada carácter.

**Tabla 6.3.** Identificación de los caracteres que componen una cadena mediante su emplazamiento

| L | I | a | m | a | d | m | e | [ | I | s  | m  | a  | e  | I  |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

En la cadena de la Tabla 6.3, en la posición 2 encontramos el carácter 'a', en la posición 9 el carácter 'l' y en la posición 12, de nuevo, otro carácter 'a'.

Cuando hablamos de extraer uno o varios caracteres de una cadena nos referimos a obtener una copia del carácter o caracteres en cuestión, pero la cadena se mantiene intacta.

## ■■■ Obtención de un carácter

Para conocer qué carácter se encuentra en una posición determinada de una cadena disponemos de:

- `char charAt(int posición)`: devuelve el carácter que ocupa el índice `posición` en la cadena que invoca el método. Hay que tener mucha precaución con no utilizar una posición que se encuentre fuera de rango, ya que esto provocará un error y la terminación abrupta del programa. Veamos un ejemplo:

```
String frase = "Nació con el don de la risa";
System.out.println(frase.charAt(4)); //muestra el carácter 'ó'
char c = frase.charAt(30); //¡error! No existe la posición 30
```

## ■■■ Obtención de una subcadena

Una subcadena es un fragmento de una cadena, es decir, un subconjunto de caracteres contiguos de una cadena. En ocasiones puede ser interesante extraer de una cadena un fragmento. Por ejemplo, si tenemos el nombre y los apellidos de alguien, puede ser útil extraer solo los apellidos. Los métodos que llevan a cabo esto son:

- `String substring(int inicio)`: devuelve la subcadena formada desde la posición `inicio` hasta el final de la cadena. Lo que se devuelve es una copia y la cadena invocante no se modifica.

```
String cad1 = "Una mañana, al despertar de un sueño intranquilo";
String cad2 = cad1.substring(28); //cad2 vale "un sueño intranquilo"
```

- `String substring(int inicio, int fin)`: hace lo mismo que la anterior, devolviendo la subcadena comprendida entre los índices `inicio` y el anterior a `fin`.

```
String cad1 = "Una_mañana,_al_despertar_de_un_sueño_intranquilo";
String cad2 = cad1.substring(15, 36); //cad2 = "despertar de un sueño"
```

Hay que notar que en `cad1` el carácter que ocupa el índice 36 es el espacio en blanco, que va justo antes de *intranquilo* y que este carácter no forma parte de la subcadena devuelta. Esta se forma con los caracteres que se encuentran desde el índice 15 hasta el carácter anterior al 36, es decir, el 35.

### Argot técnico

Es una norma general en Java que, cuando se especifica un rango de índices mediante **inicio** y **fin**, el índice **inicio** esté incluido en el rango y el de **fin**, excluido. La razón es que así la longitud del rango puede calcularse restando: **fin - inicio**.



En ambos métodos ocurre que si utilizamos un índice que se encuentra fuera de rango, es decir, no corresponde a ningún carácter, se produce un error que termina la ejecución del programa.

A veces una cadena leída del teclado o de algún fichero viene acompañada de una serie de espacios en blanco (' ') y tabuladores ('\t', que representaremos '\_\_\_\_\_') al comienzo o al final de la cadena. Para eliminar estos caracteres blancos,

- `String strip()`: devuelve una copia de la cadena eliminando los caracteres blancos del principio y del final. La cadena invocante no se modifica.

```
String cad1 = " _____Mi_perro_se_llama_Perico_____";  
String cad2 = cad1.strip(); //cad2 vale "Mi perro se llama Perico"
```

### Argot técnico



Tradicionalmente se ha usado el método `trim()` para esta operación, pero el resultado no es idéntico. Mientras `strip()` elimina los espacios blancos, `trim()` elimina todos los caracteres no imprimibles.

- `String stripLeading()`: igual que `strip()` pero solo elimina los espacios en blanco del principio.
- `String stripTrailing()`: solo elimina los espacios en blanco del final.

## 6.3.5. Longitud de una cadena

Como hemos visto, en ciertos métodos es necesario utilizar algunos índices para localizar los caracteres que forman una cadena. Para evitar el uso de un índice que se encuentre fuera de rango, existe:

- `int length()`: devuelve el número de caracteres (longitud) de una cadena. Una vez conocida la longitud, podemos usar, sin miedo a generar un error, cualquier índice comprendido entre 0 y el índice del último carácter, que es la longitud de la cadena menos 1.

```
int longitud;  
String cad1 = "Hola", cad2 = "";  
longitud = cad1.length(); //devuelve 4  
longitud = cad2.length(); //devuelve 0
```

### Actividad resuelta 6.2

Introducir por teclado dos frases e indicar cuál de ellas es la más corta, es decir, la que contiene menos caracteres.

#### Solución

```
import java.util.Scanner;  
/* Leeremos dos cadenas (String), y compararemos sus longitudes.  
 * Para obtener el tamaño utilizamos length(). */
```

```

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // leemos las dos frases
        System.out.println("Primera frase:");
        String frase1 = sc.nextLine();
        System.out.println("Segunda frase:");
        String frase2 = sc.nextLine();
        // calculamos la longitud de cada palabra
        int longFrase1 = frase1.length();
        int longFrase2 = frase2.length();
        // comparamos los tamaños
        if (longFrase1 == longFrase2) {
            System.out.println("Son de idéntica longitud");
        } else if (longFrase1 < longFrase2) {
            System.out.println(frase1 + " es más corta que " + frase2);
        } else {
            System.out.println(frase2 + " es más corta que " + frase1);
        }
    }
}

```

## Actividad resuelta 6.3

Diseñar el juego «Acierta la contraseña». La mecánica del juego es la siguiente: el primer jugador introduce la contraseña; a continuación, el segundo jugador debe teclear palabras hasta que la acierte. Realizar dos versiones; en la primera se facilita el juego indicando si la palabra introducida es mayor o menor alfabéticamente que la contraseña. En la segunda, el programa mostrará la longitud de la contraseña y una cadena con los caracteres acertados en sus lugares respectivos y asteriscos en los no acertados.

### Solución a)

```

import java.util.Scanner;
/* El método equals() nos dice si dos cadenas son iguales y el método compareTo()
 * especifica qué cadena es mayor o menor que la otra. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String passwd, palabra;
        System.out.print("Jugador 1. Introduzca la contraseña: ");
        passwd = sc.nextLine(); //leemos la contraseña
        do {
            System.out.print("Jugador 2. Palabra: ");
            palabra = sc.nextLine();
            int comparacion = passwd.compareTo(palabra); //comparamos alfabéticamente
            if (comparacion == 0) {
                System.out.println(";Acertaste!"); //son iguales
            } else if (comparacion < 0) {
                System.out.println("La contraseña es menor que: " + palabra);
            } else {
                System.out.println("La contraseña es mayor que: " + palabra);
            }
        } while (!passwd.equals(palabra));
    }
}

```

**Solución b)**

```

import java.util.Scanner;
/* Las cadenas no se pueden comparar utilizando el operador ==, para
 * realizar comparaciones de cadenas disponemos de equals() y otros métodos. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String passwd, palabra;
        System.out.print("Jugador 1. Introduzca la contraseña: ");
        passwd = sc.nextLine(); //leemos la contraseña
        System.out.println("La contraseña tiene " + passwd.length() + " caracteres");
        System.out.print("Jugador 2. Palabra: ");
        palabra = sc.nextLine(); //leemos una palabra: primer intento
        while (!palabra.equals(passwd)) {//mientras no sean iguales seguimos jugando
            String pista = "";
            //si palabra tiene una longitud menor que la contraseña se producirá
            //un error en tiempo de ejecución. ¿Por qué?
            for (int i = 0; i < passwd.length(); i++) {
                if (passwd.charAt(i) == palabra.charAt(i)) { //si son iguales
                    pista += passwd.charAt(i); //se añade el i-ésimo carácter a la pista
                } else {
                    pista += '*'; //en otro caso, añadimos un *
                }
            }
            System.out.println(pista); //mostramos la pista
            System.out.print("Jugador 2. Introduzca palabra de nuevo: ");
            palabra = new Scanner(System.in).next(); //leemos otra palabra
        }
        System.out.println("¡Acertaste!"); //salir de while significa acertar
    }
}

```

**Actividad resuelta 6.4**

Diseñar una aplicación que pida al usuario que introduzca una frase por teclado e indique cuántos espacios en blanco tiene.

**Solución**

```

import java.util.Scanner;
/* Vamos a recorrer la cadena introducida por el usuario, comprobando carácter a
 * carácter si coincide con un espacio en blanco. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase;
        int numEspaciosBlanco = 0; //contador del número de espacios en blanco
        char c;
        System.out.print("Escriba una frase: ");
        frase = sc.nextLine();
        for (int i = 0; i < frase.length(); i++) {//recorremos del índice 0 a longitud -1
            c = frase.charAt(i); //vemos cual es el i-ésimo carácter
            if (Character.isSpaceChar(c)) { //es equivalente a: c == ' '

```

```
        numEspaciosBlanco++; //incrementamos
    }
}
System.out.println("Tiene: " + numEspaciosBlanco + " espacios en blanco");
}
```

### Actividad resuelta 6.5

Diseñar una función a la que se le pase una cadena de caracteres y la devuelva invertida. Un ejemplo, la cadena «Hola mundo» quedaría «odnum aloH».

## Solución

### 6.3.6. Búsqueda

Dentro de una cadena, entre los caracteres que la forman, es posible buscar un carácter o una subcadena. Disponemos de métodos que realizan la búsqueda de izquierda a derecha, o en sentido contrario a partir de una posición dada. En cualquier caso, los métodos de búsqueda devuelven el índice donde se ha encontrado lo que se buscaba, o un  $-1$  en caso contrario.

- `int indexOf(int c)`: busca la primera ocurrencia del carácter `c` en la cadena invocante empezando por el principio. Si lo encuentra, devuelve su índice, o `-1` en caso contrario.

caso contrario. Obsérvese que el carácter se pasa como un entero; esto no supone ningún problema gracias a la conversión automática entre el tipo `char` e `int`.

```
int pos;
String cad = "Mi_perro_se_llama_Perico";
pos = cad.indexOf('j'); //pos vale -1, no encontramos 'j' en cad
pos = cad.indexOf('e'); //pos vale 4, el índice de la primera 'e'
```

- `int indexOf(String cadena)`: sirve para buscar la primera ocurrencia de una cadena.

```
pos = cad.indexOf("hola"); //pos vale -1, no se encuentra "hola"
pos = cad.indexOf("perro"); //pos vale 3
```

Los métodos anteriores buscan desde el comienzo de la cadena, comenzando en la posición 0 y avanzando, pero es posible comenzar la búsqueda en otra posición.

- `int indexOf(int c, int inicio)`: busca la primera ocurrencia del carácter `c`, pero en lugar de comenzar a buscar en la posición 0, lo hace desde la posición `inicio` en adelante. Devuelve el índice del elemento buscado si lo encuentra o -1 en caso contrario.

```
int pos;
String cad = "Mi_perro_pequines_se_llama_perico";
pos = cad.indexOf('s'); //devuelve 16
pos = cad.indexOf('s', 25); //devuelve -1, a partir de la
                           //posición 25 no se encuentra 's'
```

- `int indexOf(String cadena, int inicio)`: busca la primera ocurrencia de cadena a partir de la posición `inicio`:

```
pos = cad.indexOf("pe"); //devuelve 3
pos = cad.indexOf("pe", 4); //devuelve 9, la posición del primer
                           //"pe" a partir del índice 4
```

Los métodos anteriores realizan la búsqueda de izquierda a derecha, en el sentido de la escritura, pero es posible realizar la búsqueda empezando por el final:

- `int lastIndexOf(int c)`: devuelve el índice de la última ocurrencia de `c`, o -1 en el caso de que no se encuentre.

```
String cad = "su_perro_pequines_se_llama_perico";
int pos = cad.lastIndexOf('s'); //devuelve 18. Busca 's' desde el final
```

- `int lastIndexOf(String cadena)`: funciona igual que el anterior, pero buscando la última ocurrencia de cadena. Un ejemplo:

```
pos = cad.lastIndexOf("pe"); //devuelve 27
```

El método `lastIndexOf()` está sobrecargado para buscar a partir de una posición cualquiera. En este caso, la búsqueda comienza en la posición indicada, de derecha a izquierda.

- `int lastIndexOf(int c, int inicio)`: la búsqueda se realiza desde el final al inicio de la cadena, comenzando en la posición `inicio`.
- `int lastIndexOf(String cadena, int inicio)`: devuelve la posición de cadena en la cadena invocante, comenzando en el índice `inicio` y buscando desde el final hacia el principio. En caso de no encontrar nada, devuelve -1.

## Actividad resuelta 6.6

Escribir un programa que pida el nombre completo al usuario y lo muestre sin vocales (mayúsculas, minúsculas y acentuadas). Por ejemplo, “Álvaro Pérez” se mostrará «lvr Prz».

### Solución

```
import java.util.Scanner;
/* La idea es recorrer el nombre, carácter a carácter, comprobando si es una
 * vocal. En el caso de que no sea una vocal concatenaremos el carácter al final de
 * una segunda cadena, que llamaremos sinVocales. Para comprobar si un carácter
 * es una vocal crearemos la función: esVocal() */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String nombre, sinVocales = "";
        char c;
        System.out.print("Escriba su nombre completo: ");
        nombre = sc.nextLine();
        for (int i = 0; i < nombre.length(); i++) { //recorremos la tabla
            c = nombre.charAt(i);
            if (!esVocal(c)) {
                sinVocales = sinVocales + c;
            }
        }
        System.out.println(sinVocales);
    }

    static boolean esVocal(char c) {
        boolean result; //resultado de la comprobación
        String vocales = "aeiouáéíóúü"; //cadena con todas las vocales posibles
        //en minúsculas
        c = Character.toLowerCase(c); //convertimos c en minúsculas
        if (vocales.indexOf(c) == -1) { //buscamos c en la cadena vocales
            result = false; //si no se encuentra es que no es una vocal
        } else {
            result = true; //en caso contrario: es una vocal
        }
        return result;
    }
}
```

## Actividad resuelta 6.7

Diseñar un programa que solicite al usuario una frase y una palabra. A continuación buscará cuántas veces aparece la palabra en la frase.

### Solución

```
import java.util.Scanner;
/* Buscamos la palabra en la frase usando el método indexOf(), una vez encontrada
 * la primera ocurrencia (en el índice pos) seguiremos buscando, a partir de
 * pos, por si existe otra ocurrencia. Y así sucesivamente hasta que no
 * encontremos más (pos será -1). */
```

```
public class Main {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        String frase, palabra;  
        int veces = 0, pos; //variables contador y posición  
        System.out.print("Introduzca una frase: ");  
        frase = sc.nextLine(); //lee cualquier carácter hasta pulsar Intro  
        System.out.print("Introduzca una palabra: ");  
        palabra = sc.next(); //solo lee una palabra: sin espacios  
        pos = frase.indexOf(palabra); //buscamos la primera ocurrencia  
        while (pos != -1) { //mientras pos no sea -1, no hemos encontrado la palabra  
            veces++; //si hemos encontrado una ocurrencia, incrementamos veces  
            pos = frase.indexOf(palabra, pos + 1); //volvemos a buscar a partir de la  
            //posición siguiente a pos, por si encontramos otra ocurrencia de palabra  
        }  
        //cuando salimos del bucle es que ya no existen más ocurrencias  
        if (veces == 0) { //no hemos encontrado la palabra en la frase  
            System.out.println("'" + palabra + "' no se encuentra en la frase");  
        } else {  
            System.out.println("'" + palabra + "' está " + veces + " veces");  
        }  
    }  
}
```

### ■ ■ ■ 6.3.7. Comprobaciones

Es posible realizar ciertas comprobaciones con una cadena de caracteres, como por ejemplo si está vacía, si contiene cierta subcadena, si comienza con un determinado prefijo o si termina con un sufijo dado, entre otras. Por regla general, los métodos que realizan estas comprobaciones devuelven un booleano que indica el éxito o el fracaso de la consulta.

#### ■ ■ ■ Cadena vacía

Una cadena vacía es aquella que no está formada por ningún carácter, y se representa mediante `""` (comillas dobles seguidas de otras comillas dobles). No contiene ningún carácter, es decir, su longitud es 0. Para asignar la cadena vacía a una variable,

```
String cad = "";
```

Si mostramos una cadena vacía, no aparecerá nada en pantalla. Una operación frecuente es inicializar una variable con la cadena vacía para ir concatenándole otras cadenas. El método para comprobar si una variable contiene la cadena vacía es:

- `boolean isEmpty()`: indica mediante un booleano `true`, si la cadena está vacía, o `false` en caso contrario. Un ejemplo:

```
String cad1 = "", cad2 = "Hola...";  
cad1.isEmpty(); //true  
cad2.isEmpty(); //false
```

## Contiene

Si necesitamos comprobar si una cadena contiene otra subcadena,

- `boolean contains(CharSequence subcadena)`: devuelve `true` si en la cadena invocante se encuentra `subcadena` en cualquier posición. Hay que notar que el parámetro de entrada que se le pasa al método es un objeto `CharSequence`, pero podemos utilizar el método directamente con `String`. Veamos un ejemplo:

```
String frase = "En un lugar de la Mancha";
String palabra = "lugar";
System.out.println(frase.contains(palabra)); //muestra true
System.out.println(frase.contains("silla")); //muestra false
```

## Argot técnico



`CharSequence` es una interfaz implementada por la clase `String`. Para entender mejor estos conceptos debemos esperar a estudiar la unidad sobre interfaces.

## Prefijos y sufijos

Los prefijos y sufijos no son más que subcadenas que van al principio o al final de una cadena respectivamente. Un ejemplo de prefijo en Java para la palabra *programación* es *prog*. En las cadenas de caracteres podemos comprobar si comienzan o terminan con un prefijo o sufijo dado. Para ello disponemos de los siguientes métodos:

- `boolean startsWith(String prefijo)`: comprueba si la cadena que invoca el método comienza con la cadena `prefijo` que se pasa como parámetro.

```
String frase = "Hola mundo...";
boolean empieza = frase.startsWith("Hol"); //true, frase comienza por "Hol"
empieza = frase.startsWith("mun"); //false, frase no comienza por "mun"
```

- `boolean startsWith(String prefijo, int inicio)`: hace lo mismo que el método anterior, comenzando la comprobación en la posición `inicio`. Dicho de otra forma, para realizar la comprobación ignora los caracteres desde el principio de la cadena hasta una posición anterior a `inicio`.

```
String frase = "Hola mundo...", prefijo1 = "Hol", prefijo2 = "mun";
empieza = frase.startsWith(prefijo1, 5); //false
// "Hola mundo..." eliminando los caracteres del 0 al 4: "Hola-mundo..."
// "Hola-mundo..." no empieza por "Hol"
empieza = frase.startsWith(prefijo2, 5); //true
// "Hola-mundo..." comienza por "mun"
```

- `boolean endsWith(String sufijo)`: indica si la cadena termina con el sufijo que le pasamos como parámetro.

```
String frase = "Hola mundo";
b = frase.endsWith("De"); //falso, evidentemente frase no termina en "De"
b = frase.endsWith("undo"); //cierto, frase finaliza con "undo"
```

## Actividad resuelta 6.8

Los habitantes de Javalandia tienen un idioma algo extraño; cuando hablan siempre comienzan sus frases con «Javalín, javalón», para después hacer una pausa más o menos larga (la pausa se representa mediante espacios en blanco o tabuladores) y a continuación expresan el mensaje. Existe un dialecto que no comienza sus frases con la muletilla anterior, pero siempre las terminan con un silencio, más o menos prolongado y la coletilla «javalén, len, len». Se pide diseñar un traductor que, en primer lugar, nos diga si la frase introducida está escrita en el idioma de Javalandia (en cualquiera de sus dialectos), y en caso afirmativo, nos muestre solo el mensaje sin muletillas.

### Solución

```
import java.util.Scanner;
/* Para ver si la frase está escrita en javalandés, miramos si empieza o termina por
 * el prefijo o el sufijo de sus dialectos. Para ello, usamos los métodos startsWith()
 * y endsWith() de la clase String. Para extraer el mensaje, utilizamos dos versiones
 * sobrecargadas de substring() */
public class Main {
    public static void main(String[] args) {
        final String prefijo = "Javalín, javalón"; //constantes con el comienzo y la
        final String sufijo = "javalén, len, len"; //terminación en javalandés
        Scanner sc = new Scanner(System.in);
        System.out.print("Escriba una frase: ");
        String entrada = sc.nextLine(); //texto de entrada al traductor
        boolean idiomaJavalandia = false; //suponemos que entrada no está javalandés
        //Vamos a comprobar si el texto de entrada empieza o termina con alguna
        //muletilla
        if (entrada.startsWith(prefijo)) { //si la frase comienza con prefijo
            idiomaJavalandia = true; //el idioma es javalandés
            entrada = entrada.substring(prefijo.length()); //quitamos el prefijo
            //nos quedamos con los caracteres de entrada a partir del siguiente al
            //prefijo
        } else if (entrada.endsWith(sufijo)) { //si la entrada termina con sufijo
            idiomaJavalandia = true; //es javalandés
            entrada = entrada.substring(0, entrada.length() - sufijo.length()); //quitamos
            //el sufijo. Nos interesa desde el primer carácter de la entrada (0)
            //hasta el carácter antes del sufijo
        }
        if (idiomaJavalandia) {
            entrada = entrada.strip(); // quitamos los espacios antes y después
            System.out.println(entrada); //mostramos
        } else {
            System.out.println("No está escrito en el idioma de Javalandia");
        }
    }
}
```

### 6.3.8. Conversión

Una cadena puede transformarse sustituyendo todas las letras que la componen a minúsculas o a mayúsculas, lo que resulta útil a la hora de procesar, por ejemplo, valores que provienen de un formulario y que cada usuario puede escribir de una forma u otra.

Por homogeneidad se suele trabajar con todos los valores convertidos a un solo tipo de letra. Para realizar esta operación disponemos de:

- `String toLowerCase ()`: devuelve una copia de la cadena donde se han convertido todas las letras a minúsculas.
- `String toUpperCase ()`: similar al método `toLowerCase ()`, convierte todas las letras a mayúsculas.

Veamos un ejemplo de los dos métodos:

```
String frase = "Mi PeRrO: sE lLaMa PeRiCo23 .";
String copia;
copia = frase.toLowerCase(); //mi perro: se llama perico23 .
copia = frase.toUpperCase(); //MI PERRO: SE LLAMA PERICO23 .
```

Solo se convierten las letras; el resto de caracteres se mantiene igual.

## Actividad resuelta 6.9

Introducir por teclado una frase palabra a palabra, y mostrar la frase completa separando las palabras introducidas con espacios en blanco. Terminar de leer la frase cuando alguna de las palabras introducidas sea la cadena «fin» escrita con cualquier combinación de mayúsculas y minúsculas. La cadena «fin» no aparecerá en la frase final.

### Solución

```
import java.util.Scanner;
/* Vamos a leer una serie de palabras que iremos concatenando. Hay que comprobar
 * cada palabra leída por si coincide con alguna combinación de mayúsculas/
 * minúsculas de la cadena "fin" */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase = "", palabra; //frase debe inicializarse con la cadena vacía
        //ya que vamos a concatenarle otra cadena.
        //leemos la primera palabra fuera del bucle por si es "fin"
        System.out.print("Escriba una palabra: ");
        palabra = sc.next(); //solo leemos una palabra
        while (!palabra.toLowerCase().equals("fin")) {
            frase = frase + " " + palabra; //concatenamos la palabra al final de la
            //frase, con un espacio en blanco. La primera vez, frase está
            //inicializada con la cadena vacía. Si no, produciría un error.
            System.out.print("Escriba una palabra: ");
            palabra = sc.next();
        }
        //Sea cual sea la combinación de mayúsculas/minúsculas de palabra, la
        //convertimos a minúscula para compararla con "fin". Se podría convertir a
        //mayúsculas y comparar con "FIN"
        System.out.println(frase); //mostramos el resultado
    }
}
```

## Actividad resuelta 6.10

Realizar un programa que lea una frase del teclado y nos indique si es palíndroma, es decir, que la frase sea igual leyendo de izquierda a derecha, que de derecha a izquierda, sin tener en cuenta los espacios. Un ejemplo de frase palíndroma es: «Dábale arroz a la zorra el abad».

Las vocales con tilde hacen que los algoritmos consideren una frase palíndroma como si no lo fuese. Por esto, supondremos que el usuario introduce la frase sin tildes.

### Solución

```
import java.util.Scanner;
/* La frase "Dabale arroz a la zorra el abad" es palíndroma si no tenemos encuentra
 * los espacios en blanco. Por lo tanto, lo primero que tenemos que hacer, es eliminarlos.
 * A continuación, vamos a construir la frase invertida. Si ambas, original e
 * invertida, coinciden es porque la frase original es palíndroma.
 * Nota: escribiremos las frases sin vocales acentuadas. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase, sinEspacios, invertida;
        System.out.print("Introduzca una frase: ");
        frase = sc.nextLine();
        frase = frase.toLowerCase(); //trabajaremos con las letras en minúsculas
        sinEspacios = eliminaEspacios(frase); //devuelve una cadena sin espacios
        invertida = alReves(sinEspacios); //definida en la Act. Resuelta 6.5
        if (sinEspacios.equals(invertida)) {
            System.out.println("La frase es palíndroma");
        } else {
            System.out.println("La frase no es palíndroma");
        }
    }

    //La función construye y devuelve una cadena idéntica a la pasada, con la
    //diferencia que se han eliminado todos los espacios en blanco
    static String eliminaEspacios(String cadena) {
        String sin = "";
        for (int i = 0; i < cadena.length(); i++) { //recorremos la cadena
            char c = cadena.charAt(i); //miramos el carácter en la i-ésima posición
            if (!Character.isWhitespace(c)) { //si no es un carácter blanco
                sin = sin + c; //construimos la cadena sin con c (que no es un blanco)
            }
        }
        return sin;
    }

    static String alReves(String original) {
        ... //implementada en la Actividad Resuelta 6.5
    }
}
```

El método `replace()` permite sustituir todas las ocurrencias de un carácter de una cadena por otro que se pasa como parámetro.

- `String replace(char original, char otro)`: devuelve una copia de la cadena invocante donde se han sustituido todas las ocurrencias del carácter original por otro. Un ejemplo:

```
String frase = "Hola mundo";
frase = frase.replace('o', '\u2661') // "H\u2661la mund\u2661"
//recordamos que el code point 2661 identifica el carácter ♥
```

- `String replace(CharSequence original, CharSequence otra)`: cambia todas las ocurrencias de la cadena `original` por la cadena `otra`.

### ■ ■ ■ 6.3.9. Separación en partes

Una cadena se puede descomponer en partes si definimos un separador. Por ejemplo, podemos descomponer la frase: «En un lugar de La Mancha», formada por seis palabras separadas por espacios, en una tabla de seis elementos de tipo `String`. Para ello utilizaremos el siguiente método:

- `String[] split(String separador)`: devuelve las subcadenas resultantes de dividir la cadena invocante con el separador pasado como parámetro. La subcadenas resultantes de la división se devuelven como una tabla de `String`.

En nuestro ejemplo, escribiríamos:

```
String frase = "En_un_lugar_de_La_Mancha";
String[] palabras = frase.split("_"); //separador: espacio en blanco
```

La tabla `palabras` tiene una longitud de 6 y sus elementos son: «En», «un», «lugar», «de», «La», «Mancha». Es decir,

```
palabras = ["En", "un", "lugar", "de", "La", "Mancha"]
```

#### Argot técnico



En realidad, el separador que utiliza `split()` es una expresión regular. Las expresiones regulares son complejas y se salen del objetivo de este libro. Baste decir aquí que podemos usar cualquier cadena de caracteres como separador, siempre que no contengan determinados caracteres especiales, como el punto (.), +, \*, \$ o ?, ya que estos se usan como cuantificadores en la sintaxis de las expresiones regulares.

### ■ 6.4. Cadenas y tablas de caracteres

Existe una innegable relación entre las cadenas, clase `String`, y las tablas de caracteres, `char[]`, hasta el punto de que en algunos lenguajes de programación no existe el tipo cadena, sino tablas de caracteres. En Java, ambas, cadenas y tablas de caracteres, pueden convertirse sin problema unas en otras. En aquellas ocasiones en que interese manipular o cambiar de lugar los caracteres dentro de una cadena, resulta más cómodo trabajar con una tabla, cuyo acceso a los elementos es directo. Además, las cadenas en Java no se pueden modificar una vez creadas. Cuando modificamos una cadena lo que ocurre es que se crea una cadena nueva donde se incluyen las modificaciones. Afortunadamente, este proceso es transparente al programador.

El método que crea una tabla de caracteres tomando como base una cadena es:

- `char[] toCharArray()`: crea y devuelve una tabla de caracteres con el contenido de la cadena desde la que se invoca, a razón de un carácter en cada elemento.

```
String frase = "Hola_mundo";
char letras[];
letras = frase.toCharArray();
//la tabla letras contiene ['H', 'o', 'l', 'a', '_', 'm', 'u', 'n', 'd', 'o']
```

**Tabla 6.4.** Ejemplo de relación entre String y char[]

| Tipo   | Descripción          | Ejemplo                                 |
|--------|----------------------|-----------------------------------------|
| String | Cadena de caracteres | "Hola mundo"                            |
| char[] | Tabla de caracteres  | 'H' 'o' 'l' 'a' '_' 'm' 'u' 'n' 'd' 'o' |

El método que realiza el proceso inverso, crear una cadena tomando como base una tabla de caracteres, es:

- `static String valueOf(char[] tabla)`: devuelve un `String` con el contenido de la tabla de caracteres.

```
String cad;
char c[] = {'H', 'o', 'l', 'a'};
cad = String.valueOf(c); //cad vale "Hola"
```

En ocasiones, puede ser interesante obtener una cadena, pero no de una tabla de caracteres al completo, sino de un subconjunto de elementos de la tabla. Al siguiente método se le pasa la posición del primer índice que nos interesa y el número de caracteres que queremos utilizar.

- `static String valueOf(char[] t, int inicio, int cuantos)`: funciona de forma similar al método anterior, con la diferencia de que devuelve la cadena formada por un subconjunto de caracteres consecutivos de la tabla `t`. El parámetro `inicio` es el índice del primer elemento de la tabla que nos interesa y `cuantos` determina el número de caracteres que compondrán la cadena. Veamos cómo funciona:

```
String cad;
char c[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
cad = String.valueOf(c, 2, 4); //cad vale "cdef"
```

## Actividad resuelta 6.11

Se dispone de la siguiente relación de letras:

|             |   |   |   |   |   |   |   |   |   |   |   |
|-------------|---|---|---|---|---|---|---|---|---|---|---|
| conjunto 1: | e | i | k | m | p | q | r | s | t | u | v |
| conjunto 2: | p | v | i | u | m | t | e | r | k | q | s |

Con ella es posible codificar un texto, convirtiendo cada letra del conjunto 1 en su correspondiente del conjunto 2. El resto de las letras no se modifican. Los conjuntos se utilizan tanto para codificar mayúsculas como minúsculas, mostrando siempre la codificación en minúsculas.

Un ejemplo: la palabra «PaquiTo» se codifica como «matqvko».

Realizar un programa que codifique un texto. Para ello implementar la siguiente función:

```
char codifica(char conjunto1[], char conjunto2[], char c)
```

que devuelve el carácter `c` codificado según los conjuntos 1 y 2 que se le pasan.

### Solución

```
import java.util.Scanner;
/* En primer lugar vamos a convertir el texto introducido a minúsculas, para que los
 * alfabetos conjunto 1 y 2 sirvan para las letras mayúsculas y minúsculas.
 * El procedimiento a seguir será recorrer y codificar el texto introducido,
 * carácter a carácter. La codificación se almacenará en una tabla, que nos
 * permite asignar valores (caracteres) a cada uno de sus elementos. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        final char conjunto1[] = {'e', 'i', 'k', 'm', 'p', 'q', 'r', 's', 't', 'u', 'v'};
        final char conjunto2[] = {'p', 'v', 'i', 'u', 'm', 't', 'e', 'r', 'k', 'q', 's'};
        char codificado[]; //tabla que contendrá la codificación del texto introducido
        String texto;
        System.out.print("Introduzca un texto a codificar: ");
        texto = sc.nextLine();
        texto = texto.toLowerCase(); //convertimos el texto a minúscula, para poder
        //codificar las mayúsculas y las minúsculas con el mismo conjunto.
        codificado = new char[texto.length()]; //Creamos una tabla de igual tamaño
  //que texto
        for (int i = 0; i < texto.length(); i++) { // recorremos el texto a codificar
            //codificamos el i-ésimo carácter del texto
            codificado[i] = codifica(conjunto1, conjunto2, texto.charAt(i));
        }
        texto = String.valueOf(codificado); //convertimos la tabla con la codificación
        //en una cadena
        System.out.println(texto);
    }

    //Esta función codifica el carácter c según los alfabetos conjunto 1 y 2.
    //Buscamos el carácter c en conjunto1. Si se encuentra en la posición pos,
    //se devuelve el carácter equivalente en el segundo conjunto: conjunto2[pos].
    //En caso de no encontrar c en conjunto 1 se devuelve c sin codificar.
    static char codifica(char conjunto1[], char conjunto2[], char c) {
        final String conj1 = String.valueOf(conjunto1); //conj1 es un String con los
        //elementos de la tabla conjunto1. Facilita la búsqueda.
        char codificado; //carácter codificado correspondiente a c
        int pos = conj1.indexOf(c); //buscamos c en el conjunto 1. Al ser conj1 una
        //cadena, indexOf() busca por nosotros. En otro caso, tendríamos que
        //buscar mediante un bucle un elemento en una tabla
        if (pos == -1) { //si no hemos encontrado c en conj1
```

```
codificado = c; //no podemos codificar, devolveremos c
} else {
    codificado = conjunto2[pos]; //pos marca la posición de c en conjunto1
    //entonces elegimos el correspondiente en conjunto2
}
return codificado;
}
```

## Actividad resuelta 6.12

Un anagrama es una palabra que resulta del cambio del orden de los caracteres de otra. Ejemplos de anagramas para la palabra *roma* son: *amor*, *ramo* o *mora*. Construir un programa que solicite al usuario dos palabras e indique si son anagramas una de otra.

### Solución

```
import java.util.*;
/* El algoritmo que comprueba si cada letra de la palabra 1 se encuentra en la
 * palabra 2, y lo que es más importante, comprobar que cada letra, tanto de la
 * palabra 1 como de la 2 solo se utilizan una vez. Esto último puede ser algo
 * más laborioso de escribir.
 * Vamos a buscar una propiedad de los anagramas que nos facilite el trabajo.
 * Para que dos palabras sean anagramas tienen que tener la misma longitud y las
 * mismas letras el mismo número de veces. Lo que haremos es ordenar las letras
 * de cada palabra y comprobar si son iguales. Un ejemplo: (sin vocales acentuadas)
 * "esponja" -> ordenamos las letras: "aejnop" -> son iguales
 * "japones" -> ordenamos las letras: "aejnop" -> son iguales */
public class Main {
    public static void main(String[] args) {
        String palabra1, palabra2;
        System.out.println("Escriba una palabra: ");
        palabra1 = new Scanner(System.in).next(); //solo lee una palabra
        palabra1 = palabra1.toLowerCase(); //convertimos a minúsculas
        System.out.println("Escriba otra: ");
        palabra2 = new Scanner(System.in).nextLine();
        palabra2 = palabra2.toLowerCase(); //convertimos a minúsculas
        if (palabra1.length() != palabra2.length()) {//si son de distintos tamaño
            System.out.println("No son anagramas"); //no pueden ser anagramas
        } else {
            char p1[] = palabra1.toCharArray(); //es más fácil ordenar una tabla
            char p2[] = palabra2.toCharArray(); //convertimos las palabras a tablas
            Arrays.sort(p1); //ordenamos ambas tablas
            Arrays.sort(p2);
            if (Arrays.equals(p1,p2)) {//si las tablas son iguales: tienen las
                //mismas letras
                System.out.println("Son anagramas"); //son anagramas
            } else {
                System.out.println("No son anagramas");
            }
        }
    }
}
```

## Actividad resuelta 6.13

Diseñar un algoritmo que lea del teclado una frase e indique, para cada letra que aparece en la frase, cuántas veces se repite. Se consideran iguales las letras mayúsculas y las minúsculas para realizar la cuenta. Un ejemplo sería:

```

Frase: En un lugar de La Mancha.
Resultado:
a: 4 veces
c: 1 vez
d: 1 vez
e: 2 veces
...

```

### Solución

```

import java.util.Scanner;
/* Vamos a utilizar una tabla de contadores (numVeces) donde cada elemento de la tabla
 * corresponde a una letra y donde se almacena el número de veces que aparece en la frase
 * dicha letra. numVeces tendrá tantos elementos como letras tiene el alfabeto, es decir,
 * 'z'-'a'+1 elementos. Las letras del abecedario tienen valores Unicode correlativos.
 * A un carácter cualquiera c, le corresponde el elemento de la tabla con posición c-'a':
 * numVeces[c-'a'], se incrementa cada vez que haya una ocurrencia de c en la frase. */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String frase;
        int[] numVeces; //contador de las ocurrencias de cada letra
        System.out.print("Introduzca una frase: ");
        frase = sc.nextLine();
        //para contabilizar también las mayúsculas pasamos todo a minúsculas
        frase = frase.toLowerCase();
        // Cada posición de numVeces guardará el número de ocurrencias de una letra.
        // numVeces[0] para la 'a', numVeces[1] para la 'b', numVeces[2] para la 'c',...
        numVeces = new int['z' - 'a' + 1];//tantas componentes como letras.
        //La tabla se crea con todos los elementos inicializados a 0
        for (int i = 0; i < frase.length(); i++) { //recorre la frase carácter a carácter
            if (Character.isLetter(frase.charAt(i))) { //si el i-ésimo carácter es una letra
                numVeces[frase.charAt(i) - 'a']++; //incrementamos el contador de esa letra
            }
        }
        for (int i = 0; i < 'z' - 'a' + 1; i++) { //mostramos numVeces
            if (numVeces[i] != 0) { //solo las letras que aparecen en frase
                System.out.println("La letra " + (char) (i + 'a') +
                    " se repite " + numVeces[i] + " veces");
            }
        }
    }
}

```

## Actividad resuelta 6.14

Implementar el juego del anagrama, que consiste en que un jugador escribe una palabra y la aplicación muestra un anagrama (cambio del orden de los caracteres) generado al azar. A continuación, otro jugador tiene que acertar cuál es el texto original. La aplicación no

debe permitir que el texto introducido por el jugador 1 sea la cadena vacía. Por ejemplo, si el jugador 1 escribe «teclado», la aplicación muestra como pista un anagrama al azar, como por ejemplo: «etcloða».

### Solución

```
import java.util.Scanner;
public class Main {

    public static void main(String[] args) {
        String original; //texto original que introduce el jugador 1
        String intento; //intento de acertar la palabra original del jugador 2
        do {
            System.out.print("Jugador 1. Introduzca una palabra: ");
            original = new Scanner(System.in).next();
        } while (original.isEmpty());

        String anagrama = creaAnagrama(original);
        System.out.println("A qué palabra corresponde el anagrama: " + anagrama);
        do {
            System.out.println("Jugador 2. ¿Cuál es el original? ");
            intento = new Scanner(System.in).next();
        } while (!original.equals(intento)); //mientras no acierte el texto original
        System.out.println("Muy bien..."); //si salimos del bucle es que ha acertado
    }

    /* La función creaAnagrama() crea y devuelve un anagrama del texto original
     * pasado como parámetro. El algoritmo para construir el anagrama es:
     * 1. Convertir el String original en una tabla, que es más cómoda para
     *     intercambiar caracteres.
     * 2. Elegir dos caracteres (sus índices) al azar e intercambiarlos.
     * 3. Repetir el punto 2. Cuanta más veces se repita, mayor es el desorden.
     * Repetiremos tantas veces como la longitud del texto original. */
    static String creaAnagrama(String original) {
        char anagrama[] = original.toCharArray(); //una tabla es más cómoda para modificar

        //realizamos un intercambio al azar por cada carácter que forma el texto
        for (int numCambios = 0; numCambios < anagrama.length; numCambios++) {
            int i = (int) (Math.random() * anagrama.length); //índice al azar
            int j = (int) (Math.random() * anagrama.length); //índice al azar
            char aux = anagrama[i]; //intercambiamos anagrama[i] y anagrama[j]
            anagrama[i] = anagrama[j];
            anagrama[j] = aux;
        }
        return String.valueOf(anagrama); //devolvemos un String a partir la tabla
    }
}
```

### Actividad resuelta 6.15

Modificar la Actividad resuelta 6.14 para que el programa indique al jugador 2 cuántas letras coinciden (son iguales y están en la misma posición) entre el texto introducido por él y el original.

**Solución**

```
import java.util.Scanner;
public class Main {

    public static void main(String[] args) {
        ... //código idéntico a la Actividad Resuelta 6.14
        //solo se modifica el bucle do-while:
        do {
            System.out.println("Jugador 2. ¿Cuál es el original? ");
            intento = new Scanner(System.in).next();
            System.out.println("Letras correctas: " + letrasCorrectas(original, intento));
        } while (!original.equals(intento)); //mientras no acierte el texto original
        System.out.println("Muy bien..."); //si salimos del bucle es que ha acertado
    }

    //Comprueba cuántas letras coinciden (son iguales y ocupan la misma posición)
    //entre las dos cadenas pasadas como parámetros.
    static int letrasCorrectas(String a, String b) {
        int longitudMinima = Math.min(a.length(), b.length());
        //usamos la longitud mínima de ambas cadenas para evitar extraer caracteres de más
        int contadorLetrasCorrectas = 0;
        for (int i = 0; i < longitudMinima; i++) {
            if (a.charAt(i) == b.charAt(i)) {
                contadorLetrasCorrectas++;
            }
        }
        return contadorLetrasCorrectas;
    }

    static String creaAnagrama(String original) {
        ... //implementación en la Actividad Resuelta 6.14
    }
}
```