Estructuras lineales de datos

Objetivos

- Conocer las diferentes formas de acceso de datos.
- Utilizar los punteros a datos.
- Conocer y poner en práctica el uso de arreglos unidimensionales y multidimensionales.
- Emplear los tipos de ordenamiento.
- Conocer y poner en práctica la Clase String.
- Conocer y poner en práctica las distintas formas de conversiones a otros tipos.

Contenidos

Arreglos unidimensionales (vectores). Arreglos multidimensionales (matrices). Operaciones básicas con arreglos: declarar, recorrer, cargar y mostrar arreglos. Arreglos como parámetros de una función. Contar y sumar elementos de un arreglo. Bucle For mejorado.

Búsqueda secuencial y binaria. Algoritmos de ordenamiento de vectores básicos (Burbuja).

La clase String. Métodos básicos. Comparar cadenas con equals(). Conversiones a otros tipos.

Arreglos

Para empezar con este tema presentamos un ejemplo. Hay que realizar un programa que le pida al usuario seis tiempos, en segundos, correspondientes a las vueltas de cierta carrera para que la máquina calcule cuántas vueltas superaron el tiempo promedio de vueltas.

Con lo que conocemos, no queda más alternativa que declarar seis variables enteras, cuyos nombres pueden ser vuelta1, vuelta2, vuelta3, vuelta4, vuelta5 y vuelta6. Operaciones simples como leer un dato desde consola y asignar a cada variable deberían repetirse seis veces. Necesitamos una manera de agrupar en una única variable datos relacionados del mismo tipo.

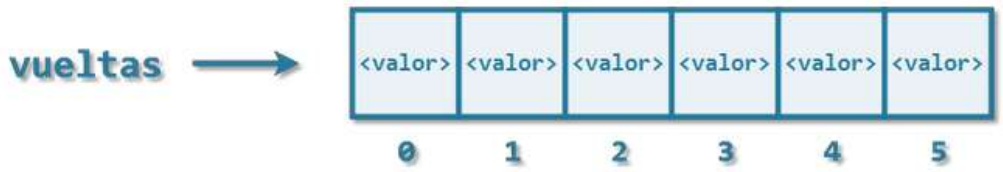
Entre múltiples y variadas estructuras de datos existentes que permitirían guardarlos, veremos la más sencilla: un **arreglo unidimensional**.

Un arreglo unidimensional (en inglés, "array"), también llamado **vector**, es una estructura de datos que consiste en agrupar elementos del mismo tipo en celdas contiguas, permitiendo acceder a tales celdas, y, por ende, a los datos, mediante un número de índice, cuyo primer valor es el 0.

Es unidimensional porque solo consta de una dimensión.

Existen también los arreglos bidimensionales (dos dimensiones), a los cuales se les suele denominar matrices.

Para el caso anterior, podríamos tener una variable de tipo arreglo de enteros llamada **vueltas** que guarde cada uno de los tiempos en una celda, como muestra la ilustración:



En Java, los arreglos no son tipos de datos primitivos, sino que son objetos.

A las variables de tipo objeto también se las llama de referencia, pues en realidad, una variable de tipo objeto no tiene como valor a un objeto en sí sino una referencia a él.

A continuación, mostramos diferentes formas de declarar una variable de tipo arreglo.

Declaración

1) Sin inicializar , La sintaxis para declarar un arreglo es la siguiente:

```
<tipo>[] <identificador>;
```

Donde **<tipo>** es el tipo de dato que junto a los corchetes de apertura y cierre indican que se trata de un conjunto de datos de ese tipo. **<identificador>** es el nombre del arreglo (siguiendo el mismo criterio de nomenclatura para cualquier variable).

La siguiente instrucción declara un arreglo de enteros llamado vueltas:

```
int[] vueltas;
```

Hasta aquí, vueltas está sin inicializar.

Por lo general, los arreglos se llaman con un sustantivo plural, pues permiten guaradr varios datos similares.

2) Inicializado por defecto

Para que vueltas tenga una referencia a un arreglo, es necesario crear el mismo y asignarlo.

La sintaxis es la siguiente:

```
<tipo>[] <identificador> = new <tipo>[<longitud>];
```

El operador de asignación permite asignar una nueva referencia a un arreglo, que se crea mediante la palabra **new**, seguida del tipo **<tipo>** (debe coincidir con el de la declaración) y un par de corchetes. Dentro de los corchetes se escribe un número **<longitud>** que representa la dimensión del arreglo, es decir, la cantidad de celdas contiguas que poseerá.

Continuando con el ejemplo anterior, crearemos un arreglo de seis enteros, que será referenciado por la variable **vueltas**.

Podemos declarar y asignar:

```
int[] vueltas = new int[6];
```

También podemos primero declarar en una línea y asignar después:

```
int[] vueltas; vueltas = new int[6];
```

Bien, vueltas apunta hacia un arreglo de seis valores. Pero ¿cuánto valen esos valores por defecto? La respuesta depende del tipo de dato del cual se definió el arreglo:

Para arreglos de tipos int, char y double, el valor por defecto de cada una de las celdas es 0.

Para arreglos de tipo boolean, el valor por defecto de cada una de las celdas es false.

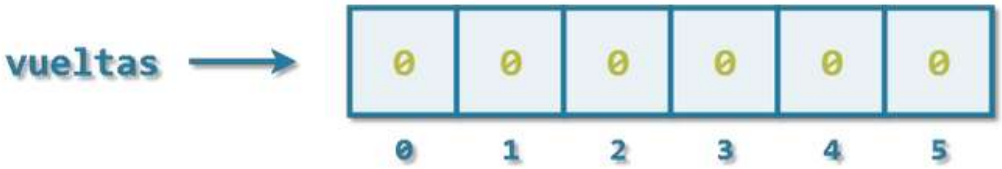
Para arreglos de tipos por referencia (por ejemplo, un arreglo de cadenas String), el valor por defecto de cada una de las celdas es null.

El valor **null** es especial. Indica que la referencia es nula, es decir, no existe. Cualquier intento de hacer operaciones con un valor null genera un error.

Por lo tanto, al haber hecho la siguiente declaración y asignación:

```
int[] vueltas; vueltas = new int[6];
```

El arreglo al que apunta vueltas queda así:



3) Inicializado con valores

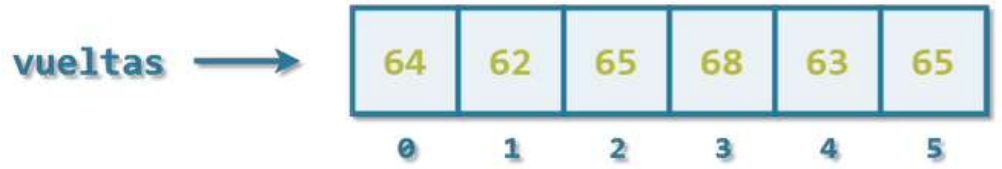
Existe una tercera manera de crear un arreglo con valores constantes definidos en el propio código. Esta forma es útil para hacer algunas pruebas sobre el funcionamiento de los arreglos, pero, en un programa de uso práctico, los valores debería introducirlos el usuario y no estar escritos estáticamente en el código.

Vamos a crear un arreglo llamado vueltas con los valores de tiempos 64, 62, 65, 68, 63 y 65:

```
int[] vueltas = {64,62,65,68,63,65};
```

Como habrás observado, en un par de llaves se listan, separados por comas, los valores deseados. Java asume que se trata de un arreglo de 6 elementos, por lo que implícitamente lo crea con dicha longitud.

El arreglo al que apunta vueltas queda así:



Operaciones básicas con arreglos

Obtener un valor

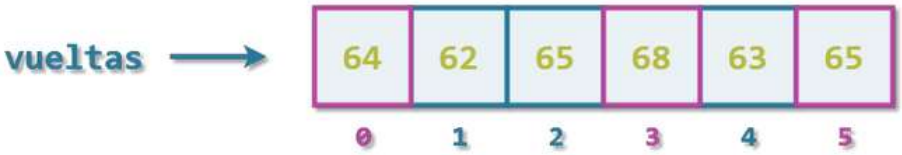
Para obtener un valor de un arreglo, se debe escribir el nombre de este seguido de un par de corchetes. Dentro de los corchetes, se escribe el número de índice del dato que se desea mostrar (recordando que la cuenta empieza desde 0).

El siguiente programa muestra los tiempos de la primera, la cuarta y la última vuelta del arreglo vueltas:

Ejemplo

```
1   package prueba;
2   public class Prueba {
3       public static void main(String[] args) {
4           int[] vueltas = {64,62,65,68,63,65};
5           System.out.println("1º vuelta: " + vueltas[0] );
6           System.out.println("4º vuelta: " + vueltas[3] );
7           System.out.println("Última vuelta: " + vueltas[5] );
8       }
9   }
```

Una representación abstracta de lo que acabamos de hacer sería la siguiente:



Hay algo de lo anterior que a un programador debería hacerle ruido: el índice de la última vuelta fue el número 5 porque se tiene un arreglo de 6 posiciones.

¿Qué sucede si esas mismas instrucciones las reutilizo para mostrar otro arreglo de 10 posiciones?

Resulta que las primeras dos instrucciones de salida funcionan bien, pero la tercera, que imprime vueltas[5] no está imprimiendo el valor de la última posición, sino la sexta. El índice de la última vuelta en un arreglo de 10 posiciones es 9.

Intentar obtener el valor de un arreglo escribiendo un índice fuera de rango, como un valor negativo o mayor que el último índice, genera un error.

Habrás notado que, para obtener el valor de la última posición de cualquier arreglo, se debe conocer su longitud.

Obtener longitud del arreglo

Todo arreglo incorpora un atributo entero llamado **length**, que representa su longitud. Para obtener el valor de la longitud de cualquier arreglo se escribe el nombre de este seguido de un punto y a continuación la palabra length.

El siguiente programa muestra por consola cuánto vale la longitud del arreglo **vueltas**:

Ejemplo

```
1   package prueba;


2   public class Prueba {
3       public static void main(String[] args) {
4           int[] vueltas = {64,62,65,68,63,65};
5           System.out.println("Longitud: " + vueltas.length );
6       }
7   }
```

Gracias a poder conocer la longitud, podemos ahora obtener la última posición de cualquier arreglo: el índice del último elemento es la longitud del arreglo menos 1 (porque el índice comienza en 0).

El siguiente programa muestra el valor de la última vuelta del arreglo vueltas, pero usando una salida que funcionaría para cualquier longitud:

Ejemplo

```
1   package prueba;
2   public class Prueba {
3       public static void main(String[] args) {
4           int[] vueltas = {64,62,65,68,63,65};
5           int ult = vueltas.length-1;
6           System.out.println("Última vuelta: " + vueltas[ult]);
7       }
8   }
```

Obtener todos los valores

Para obtener todos los valores de un arreglo, simplemente hay que pasar por todos sus índices. Como ya sabemos, el índice de cualquier arreglo comienza en 0 y termina en la longitud menos 1.

El siguiente programa muestra todos los valores de cualquier arreglo (uno por línea) utilizando un ciclo **for**:

Ejemplo

```
1   package prueba;
2   public class Prueba {
3       public static void main(String[] args) {
4           int[] vueltas = {64,62,65,68,63,65};
5           for (int i = 0; i < vueltas.length; i++) {
6               System.out.println( i + "º valor: " + vueltas[i] );
7           }
8       }
9   }
```

Pero si el que visualiza esa salida es un usuario, no debería ver los elementos listados desde 0, por lo tanto, manipulamos la variable i en la salida para que se vea incrementada una unidad (en vez de mostrar de 0 a 5, mostrará de 1 a 6), pero seguiremos accediendo a los valores del arreglo utilizando el valor original de i:

Ejemplo

```
1   package prueba;
2   public class Prueba {
3       public static void main(String[] args) {
4           int[] vueltas = {64,62,65,68,63,65};
5           for (int i = 0; i < vueltas.length; i++) {
6               System.out.println( (i+1) + "º valor: " + vueltas[i] );
7           }
8       }
9   }
```

Establecer un valor

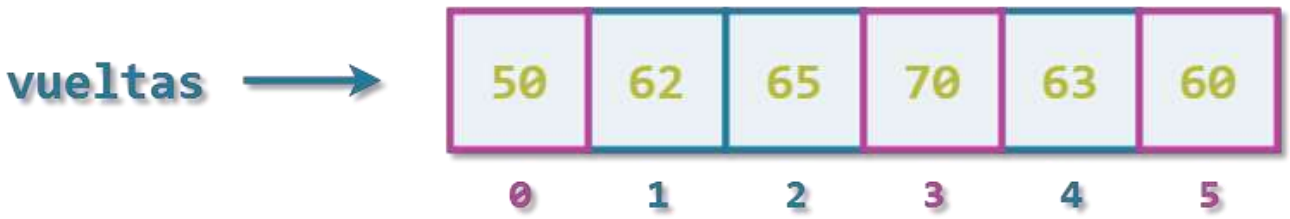
Para establecer un valor en un arreglo, se debe escribir el nombre de este seguido de un par de corchetes. Dentro de los corchetes, se escribe el número de índice del dato que se desea reemplazar (recordando que la cuenta empieza desde 0). A través del operador de asignación, se puede establecer un nuevo valor, por supuesto, del mismo tipo que la definición del arreglo.

El siguiente programa reemplaza los tiempos de la primera, la cuarta y la última vuelta del arreglo vueltas. A través del procedimiento **mostrarArreglo()**, se imprime dos veces el mismo arreglo, antes y después de las asignaciones:

Ejemplo

```
1   package prueba;
2   public class Prueba {
3       public static void main(String[] args) {
4           int[] vueltas = {64,62,65,68,63,65};
5           int ultimaPosicion = vueltas.length - 1;
6           System.out.println("Arreglo original");
7           mostrarArreglo(vueltas);
8           vueltas[0] = 50;
9           vueltas[3] = 70;
10          vueltas[ultimaPosicion] = 60;
11          System.out.println("Arreglo modificado");
12          mostrarArreglo(vueltas);
13      }

14      static void mostrarArreglo (int[] a) {
15          for (int i = 0; i < a.length; i++) {
16              System.out.print( a[i] + " " );
17          }
18          System.out.println(""); // Línea en blanco
19      }
```

Una representación abstracta de lo que acabo de hacer sería la siguiente:



Establecer todos los valores

Para establecer todos los valores de un arreglo, por ejemplo, cuando se lo crea con valores por defecto y luego se espera que el usuario los ingrese, la metodología es similar a la de mostrar todos los valores: pasar por todos los índices.

Lo ideal es modularizar, y por ello, haremos una función llamada **crearArregloDeEnteros()**, que recibe como parámetro un número que representa la longitud deseada para el nuevo arreglo. El procedimiento crea un nuevo arreglo y va guardando números enteros a medida que se van leyendo desde la consola (usando otra función llamada **leerEntero()**). Por último, se devuelve el arreglo a quien haya invocado a la función.

Ejemplo :

```
1   package prueba;
2   import java.util.Scanner;
3   public class Prueba {

4       public static void main(String[] args) {
5           int[] numeros = crearArregloDeEnteros(5);
6           System.out.println("El arreglo queda:");
7           mostrarArreglo(numeros);
8       }

9       static void mostrarArreglo (int[] a) {
10          for (int i = 0; i < a.length; i++) {
11              System.out.print(a[i] + " ");
12          }
13          System.out.println(""); // Línea en blanco
14      }

15      static int[] crearArregloDeEnteros (int tam) {
16          int[] a = new int[tam];
17          for (int i = 0; i < a.length; i++) {
18              a[i] = leerEntero("Valor " + (i+1) + " de " + a.length +
": ");
19          }
20          return a;
21      }
```

```
22          static int leerEntero (String cartel) {
23                  Scanner entrada = new Scanner (System.in);
24                  System.out.print(cartel);
25                  double x = entrada.nextDouble();
26                  while ( (int) x != x ) {
27                          System.out.print("ERROR. " + cartel);
28                          x = entrada.nextDouble();
29                  }
30                  return (int) x;
31          }
32  }
```

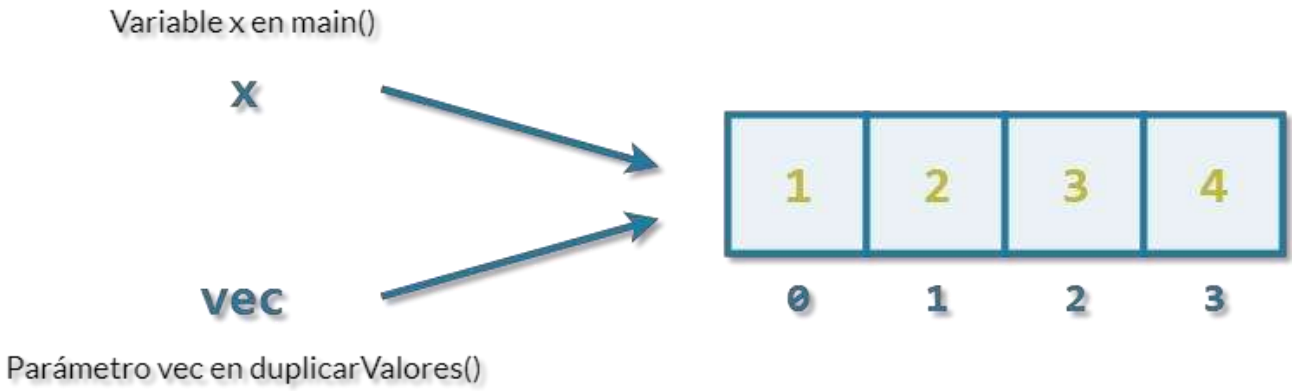
Arreglos como argumentos

En la unidad de Funciones y Procedimientos, observamos que Java trabaja con pasaje de argumentos por valor. Veremos a continuación un caso similar cuando el argumento se trata de un arreglo:

```
1   package prueba;


2   public class Prueba {
3       public static void main(String[] args) {
4           int[] x = {1,2,3,4};
5           System.out.println("Muestro arreglo x antes de invocar:");
6           mostrarArreglo(x);
7           duplicarValores(x);
8           System.out.println("Muestro arreglo x luego de invocar:");
9           mostrarArreglo(x);
10      }


11      static void duplicarValores(int[] vec) {
12          for (int i = 0; i < vec.length; i++) {
13              vec[i] *= 2;
14          }
15      }


16      static void mostrarArreglo (int[] a) {
17          for (int i = 0; i < a.length; i++) {
18              System.out.print(a[i] + " ");
19          }
20          System.out.println(""); // Línea en blanco
21      }
22  }
```

En este caso, las salidas no son similares. En la línea 7 se invoca al procedimiento que permite mostrar el arreglo, imprimiendo los valores 1, 2, 3 y 4. Sin embargo, tras la invocación del procedimiento **duplicarValores()**, el **mostrarArreglo()** de la línea 10 imprime los valores 2, 4, 6 y 8.

La explicación para tal efecto es que el pasaje sigue siendo por valor, pero los arreglos son objetos, y, como vimos anteriormente, una variable de tipo objeto no guarda más que una referencia hacia un objeto. Esto hace que cuando se invoca al procedimiento **duplicarValores()** enviando como argumento la variable x (de tipo arreglo), lo que se esté enviando como copia es la dirección hacia donde está el arreglo x.



Por consiguiente, la dirección del arreglo x llega como parámetro a la definición del procedimiento **duplicarValores()** con el nombre vec. A partir de allí, todo cambio que se haga sobre vec, tiene efecto para todas las variables que apunten hacia el mismo arreglo. Por ello, cuando se retorna a main() y se intenta mostrar el arreglo (usando siempre la misma referencia), se observa que han cambiado sus valores.

Lo explicado anteriormente funciona exactamente igual con cualquier pasaje como argumentos de variables de tipo objeto, como los arreglos o los String.

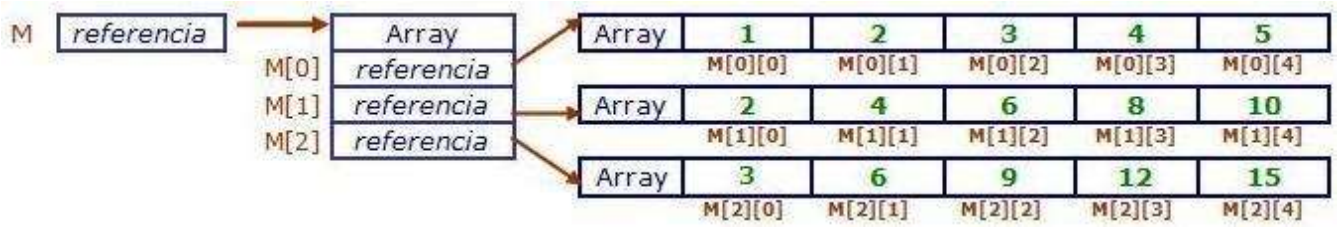
Estos conceptos se entenderán mejor cuando veamos el paradigma orientado a objetos.

Un arreglo en Java puede tener más de una dimensión. El caso más general son los arreglos bidimensionales también llamados matrices.La dimensión de un arreglo la determina el número de índices necesarios para acceder a sus elementos.Los vectores que ya hemos visto son arreglos unidimensionales porque solo utilizan un índice para acceder a cada elemento.Una matriz necesita dos índices para acceder a sus elementos. Gráficamente podemos representar una matriz como una tabla de n filas y m columnas cuyos elementos son todos del mismo tipo.La siguiente figura representa un array M de 3 filas y 5 columnas:

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Pero en realidad una matriz en Java es un array de arrays.

Gráficamente podemos representar la disposición real en memoria del array anterior así:



La longitud del array M (M.length) es 3.

La longitud de cada fila del array (M[i].length) es 5.

Para acceder a cada elemento de la matriz se utilizan dos índices. El primero indica la fila y el segundo la columna.

Crear matrices en Java

Se crean de forma similar a los arrays unidimensionales, añadiendo un índice. Por ejemplo: matriz de datos de tipo int llamado ventas de 4 filas y 6 columnas:

```
int [][] ventas = new int[4][6];
```

matriz de datos double llamado temperaturas de 3 filas y 4 columnas:

```
double [][] temperaturas = new double[3][4];
```

En Java se pueden crear arrays irregulares en los que el número de elementos de cada fila es variable. Solo es obligatorio indicar el número de filas.

Por ejemplo:

```
int [][] m = new int[3][];
```

crea una matriz m de 3 filas.

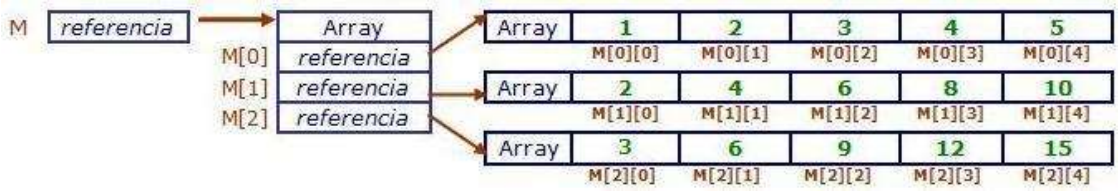
A cada fila se le puede asignar un número distinto de columnas:

Un arreglo en Java puede tener más de una dimensión. El caso más general son los rreglos bidimensionales también llamados matricesLa dimensión de un arreglo la determina el número de índices necesarios para acceder a sus elementos.Los vectores que ya hemos visto son arreglos unidimensionales porque solo utilizan un índice para acceder a cada elementoUna matriz necesita dos índices para acceder a sus elementos. Gráficamente podemos representar una matriz como una tabla de n filas y m columnas cuyos elementos son todos del mismo tipo.La siguiente figura representa un array M de 3 filas y 5 columnas:

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Pero en realidad una matriz en Java es un array de arrays.

Gráficamente podemos representar la disposición real en memoria del array anterior así:



La longitud del array M (M.length) es 3.

La longitud de cada fila del array (M[i].length) es 5.

Para acceder a cada elemento de la matriz se utilizan dos índices. El primero indica la fila y el segundo la columna.

Crear matrices en Java

Se crean de forma similar a los arrays unidimensionales, añadiendo un índice. Por ejemplo: matriz de datos de tipo int llamado ventas de 4 filas y 6 columnas:

Crear matrices en Java

Se crean de forma similar a los arrays unidimensionales, añadiendo un índice. Por ejemplo: matriz de datos de tipo int llamado ventas de 4 filas y 6 columnas:

```
int [][] ventas = new int[4][6];
```

matriz de datos double llamado temperaturas de 3 filas y 4 columnas:

```
double [][] temperaturas = new double[3][4];
```

En Java se pueden crear arrays irregulares en los que el número de elementos de cada fila es variable. Solo es obligatorio indicar el número de filas.

Por ejemplo:

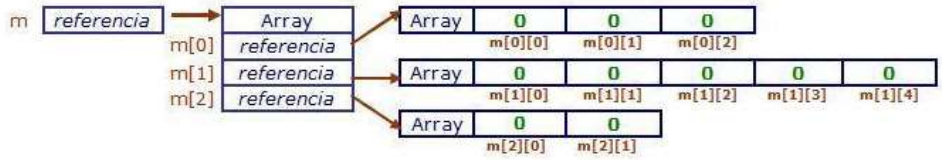
```
int [][] m = new int[3][];
```

crea una matriz m de 3 filas.

A cada fila se le puede asignar un número distinto de columnas:

```
m[0] = new int[3];
m[1] = new int[5];
m[2] = new int[2];
```

Gráficamente podemos representar la disposición real en memoria del array anterior así:



Inicializar matrices

El número de valores determina el tamaño de la matriz.

Por ejemplo:

```
int [][] numeros = {{6,7,5}, {3,8,4}, {1,0,2}, {9,5,2}};
```

Crea la matriz números de tipo int, de 4 filas y 3 columnas, y se le asignan esos valores iniciales.

Asignando valores iniciales se pueden crear también matrices irregulares.

```
int [][] a = {{6,7,5,0,4}, {3,8,4}, {1,0,2,7}, {9,5}};
```

Crea una matriz irregular de 4 filas. La primera de 5 columnas, la segunda de 3, la tercera de 4 y la cuarta de 2.

Recorrer matrices

Para recorrer una matriz se anidan dos bucles for. En general para recorrer un arreglo multidimensional se anidan tantas instrucciones for como dimensiones tenga el arreglo.

Ejemplo de recorrido de una matriz en Java:

Programa que lee por teclado números enteros y los guarda en una matriz de 5 filas y 4 columnas. A continuación muestra los valores leídos, el mayor y el menor y las posiciones que ocupan.

```
1   package prueba;
2   import java.util.*;
3   public class Prueba {
4       public static void main(String[] args) {
5           int filas = 5, columnas = 4;
6           Scanner sc = new Scanner(System.in);
7           int i, j, mayor, menor;
8           int filaMayor, filaMenor, colMayor, colMenor;
9           int[][] A = new int[filas][ columnas];
10          System.out.println("Lectura de elementos de la matriz: ");
11          for (i = 0; i < filas; i++) {
12              for (j = 0; j < columnas; j++) {
13                  System.out.print("A[" + i + "][" + j + "]= ");
14                  A[i][j] = sc.nextInt();
15              }
16          }
17          System.out.println("valores introducidos:");
18          for (i = 0; i < A.length; i++) {
19              for (j = 0; j < A[i].length; j++) {
20                  System.out.print(A[i][j] + " ");
21              }
22              System.out.println();
23          }
```

```
24                  mayor = menor = A[0][0];//se toma el primero como mayor y
menor
25                  filaMayor = filaMenor = colMayor = colMenor = 0;
26
27                  for (i = 0; i < A.length; i++) { //
28                          for (j = 0; j < A[i].length; j++) {
29                                  if (A[i][j] > mayor) {
30                                          mayor = A[i][j];
31                                          filaMayor = i;
32                                          colMayor = j;
33                                  } else if (A[i][j] < menor) {
34                                          menor = A[i][j];
35                                          filaMenor = i;
36                                          colMenor = j;
37                                  }
38                          }
39                  }
40                  System.out.print("Elemento mayor: " + mayor);
41                  System.out.println(" Fila: "+ filaMayor + " Columna: " +
colMayor);
42                  System.out.print("Elemento menor: " + menor);
43                  System.out.println(" Fila: "+ filaMenor + " Columna: " +
colMenor);
44          }
45      }
```

En este ejemplo se han utilizado dos formas distintas para recorrer la matriz:

- utilizando en el for el número de filas y columnas
- utilizando en el for el atributo length

Para recorrer arreglos irregulares como el siguiente:

int [][] a = {{6,7,5,0,4}, {3, 8, 4}, {1,0,2,7}, {9,5}};

Usaremos siempre length para obtener el número de columnas que tiene cada fila:

EjEMPLO:

Ejemplo

```
1   package prueba;
2   public class Prueba {
3       public static void main(String[] args) {
4           int [][] a = {{6,7,5,0,4}, {3, 8, 4}, {1,0,2,7}, {9,5}};
5           for (int i = 0; i < a.length; i++) { //número de filas
6               for (int j = 0; j < a[i].length; j++) { //número de columnas de cada fila
7                   System.out.print(a[i][j] + " ");
8               }
9               System.out.println();
10          }
11      }
12  }
```

Búsqueda secuencial y binaria

Búsqueda secuencial

Los dos elementos fundamentales a tener en cuenta son: un arreglo con datos objeto de la búsqueda y un elemento o criterio de búsqueda.

El método de búsqueda secuencial consiste en ir comparando el elemento o criterio de búsqueda con cada uno de los elementos en el arreglo, esto se hace recorriendo el arreglo y deteniéndose en cada elemento y hacer la comparación, en caso de ser verdadera la comparación, guardar la posición el elemento o dato.

EJEMPLO:

```
 package prueba;
2       public class Prueba {
3               public static int busquedaSecuencial(int []arreglo, int dato){
4                       int posicion = -1;
5                       for(int i = 0; i < arreglo.length; i++){ // recorremos todo el
arreglo
6                               if(arreglo[i] == dato){ // comparamos el elemento en el
arreglo con el buscado
7                                       posicion = i; // si es verdadero guardamos la
posición
8                                       break; // detiene el ciclo
9                               }
10                      }
11                      return posicion; // si encuentra, devuelve la 1ra posición
encontrada sino -1
12              }
13
14              public static void main(String[] args) {
15                      int [] a = {6,7,5,0,4};
16                      System.out.println(busquedaSecuencial(a, 5)); // encuentra en la
posición 2
17                      System.out.println(busquedaSecuencial(a, 10)); // no encuentra,
devuelve -1
18              }
19      }
```

Este método nos halla la posición del elemento o dato buscado en su primero coincidencia, si queremos que nos halle la posición de la última coincidencia, lo único que tenemos que hacer es eliminar la línea donde aparece **break**.

Si el resultado del método anterior es -1, significa que el elemento no se encuentra en el arreglo.

Ahora cabe la pregunta: ¿y si el elemento que deseamos buscar aparece varias veces en el arreglo y deseamos conocer cada una de estas posiciones, como hacemos?

Lo que hacemos es deshacernos de la línea **break** para que el vector sea recorrido en su totalidad, y de alguna forma ir almacenando cada una de las posiciones resultantes de las comparaciones verdaderas.

EJEMPLO:

```
package prueba;
2       public class Prueba {
3               public static String busquedaSecuencial2(int []arreglo, int valor){
4                       String posicion = "";
5                       for(int i = 0; i < arreglo.length; i++){
6                               if(arreglo[i] == valor){
7                                       posicion += i+" ";
8                               }
9                       }
10                      return posicion;
11              }
12
13              public static void main(String[] args) {
14                      int [] a = {6,7,5,5,4};
```

```
15                  System.out.println(busquedaSecuencial2(a, 5)); // encuentra en la posición 2 y 3

16                  System.out.println(busquedaSecuencial2(a, 10)); // no encuentra, devuelve una cadena en blanco

17              }

18      }
```

Búsqueda binaria

Antes de iniciar con la explicación hagamos un ejercicio mental, imaginemos que un amigo desea que adivinemos un número que tiene en un papel anotado y que solo él conoce, antes que empecemos nos advierte que el número está comprendido del 0 al 100, y por último por cada intento que hagamos él nos dirá si el número es mayor, menor o igual al número que tiene anotado en el papel.

Digamos que el número secreto a adivinar es el 81.

Bueno, si somos sistemáticos, empezaremos intentando con el 50.

[1,2,3,.....,58,59,50,51,52,.....,98,99,100]

Nuestro amigo nos dirá que el número que deseamos adivinar es mayor que 50.

A éste punto ya debemos deducir que el número está entre el 51 y el 100.

Observemos como ya no tenemos que gastar intentos con los números comprendidos de 0 a 50.

Ahora intentemos con el 75.

[51,52,53,.....,73,74,75,76,77,.....,98,99,100]

Nuestro amigo nos dirá que el número que deseamos adivinar es mayor que el 75.

A éste punto ya debes deducir que el número está entre el 76 y el 100.

Observemos como ya no tenemos que gastar intentos con los números comprendidos de 0 a 75.

Ahora intentemos con el 87.

[76,77,78,.....,85,86,87,88,89,.....,98,99,100]

Nuestro amigo nos dirá que el número que deseamos adivinar es menor que el 87.

A éste punto ya debemos deducir que el número está entre el 76 y el 86.

Observemos como ya no tenemos que gastar intentos con los números comprendidos de 0 a 75 y 87 a 100.

Ahora intentemos con el 81.

[76,77,78,79,80,81,82,83,84,85,86]

Nuestro amigo nos dirá que hemos acertado.

La técnica para adivinar el número es dividiendo el rango en dos partes, si el número a adivinar es mayor tomamos el rango que nos ha quedado a la derecha, si el número a adivinar es menor tomamos el rango que nos ha quedado a la izquierda.

La técnica anteriormente descrita es análoga a la técnica para la **búsqueda binaria**.

Como podemos observar y asumiendo la analogía, uno de los requisitos para el algoritmo de búsqueda binaria es que los datos estén previamente ordenados.

Este algoritmo se utiliza cuando el arreglo en el que queremos determinar la existencia de un elemento está previamente ordenado. El algoritmo reduce el tiempo de búsqueda considerablemente, ya que disminuye exponencialmente el número de iteraciones necesarias.

Para implementar este algoritmo se compara el elemento a buscar con un elemento cualquiera del arreglo (normalmente el elemento central): si el valor de éste es mayor que el del elemento buscado se repite el procedimiento en la parte del arreglo que va desde el inicio de éste hasta el elemento tomado, en caso contrario se toma la parte del arreglo que va desde el elemento tomado hasta el final. De esta manera obtenemos intervalos cada vez más pequeños, hasta que se obtenga un intervalo indivisible. Si el elemento no se encuentra dentro de este último entonces se deduce que el elemento buscado no se encuentra en todo el arreglo.

EJEMPLO:

```
1   package prueba;
2   public class Prueba{
3       public static int busquedaBinaria(int vector[], int dato){
4           int n = vector.length;
5           int centro, inf = 0, sup = n-1;
6           while(inf <= sup) {
```

```
7                       centro = (sup+inf)/2;
8                       if(vector[centro] == dato)
9                               return centro;
10                      else if(dato < vector[centro] )
11                              sup = centro-1;
12                      else
13                              inf = centro+1;
14              }
15              return -1;
16      }
17
18      public static void main(String []args){
19                                              int[]vector = {1,4,7,8,9,14,23,47,56,60,61,63,65,66,68,69,70,73,76,77,79,80,82};
20              int valorBuscado = 9;
21              System.out.println(busquedaBinaria(vector,valorBuscado));
22      }
23  }
```

Algoritmo de ordenamiento de tipo Burbuja

El algoritmo de la **burbuja** es uno de los métodos de ordenamiento más conocidos.

Consiste en comparar pares de elementos adyacentes en un arreglo y si están desordenados intercambiarlos hasta que estén todos ordenados.

Si A es el arreglo a ordenar, se realizan A.length-1 pasadas. Si la variable i es la que cuenta el número de pasadas, en cada pasada i se comprueban los elementos adyacentes desde el primero hasta A.length-i-1 ya que el resto hasta el final del arreglo están ya ordenados. Si los elementos adyacentes están desordenados se intercambian.

El método de ordenamiento de la burbuja en Java para ordenar un arreglo A es el siguiente:

EJEMPLO:

```
1   package prueba;
2   public class Prueba{
3       public static void burbuja(int[] A) {
4           int i, j, aux;
5           for (i = 0; i < A.length - 1; i++) {
6               for (j = 0; j < A.length - i - 1; j++) {
7                   if (A[j + 1] < A[j]) {
8                       aux = A[j + 1];
9                       A[j + 1] = A[j];
10                      A[j] = aux;
11                  }
12              }
13          }
14      }
15      public static void mostrarVector(int[] v) {
16          for (int i : v) {
17              System.out.print(i + " ");
18          }
19          System.out.println("");
20      }
21      public static void main(String []args){
22          int[]vector = {50,26,7,9,15,27};
```

```
23                  mostrarVector(vector);
24                  burbuja(vector);
25                  mostrarVector(vector);
26              }
27      }
```

Para mostrar el arreglo (línea 16) utilizamos un **bucle for mejorado**, tema que abordaremos más adelante.

Ejemplo de ejecución:

Arreglo Inicial:

50	26	7	9	15	27

Primera pasada:

26	50	7	9	15	27
26	7	50	9	15	27
26	7	9	50	15	27
26	7	9	15	50	27
26	7	9	15	27	50

Segunda pasada:

7	26	9	15	27	50
7	9	26	15	27	50
7	9	15	26	27	50

Bucle for mejorado (foreach o enhanced for)

Esta estructura nos permite recorrer una colección o un arreglo de elementos de una forma sencilla, evitando el uso de un bucle for normal.

De la forma tradicional podríamos recorrer un arreglo de la siguiente forma:

EJEMPLO:

```
1       package prueba;
2       public class Prueba{
3               public static void main (String[] args){
4                       String a[] = {"Buenos Aires", "Santa Fé", "Córdoba", "Mendoza", "Jujuy",
"Chubut"};
5                       for (int x=0; x<a.length; x++){
6                               System.out.println(a[x]);
7                       }
8               }
9       }
```

En este caso nos estamos apoyando en el tamaño del arreglo, con la propiedad length y en una variable contador, la cual vamos incrementando hasta que llegue a el tamaño del arreglo.

El **bucle foreach** en Java nos permite realizar estas mismas operaciones de una forma muy sencilla. La estructura del bucle foreach sería la siguiente:

```
for (TipoBase variable: ArrayDeTiposBase) {...}
```

EJEMPLO:

```
1       package prueba;
2       public class Prueba{
3               public static void main (String[] args){
4                       String a[] = {"Buenos Aires", "Santa Fé", "Córdoba", "Mendoza",
"Jujuy", "Chubut"};
5                       for (String elemento: a){
6                               System.out.println(elemento);
7                       }
8               }
9       }
```

La clase String

En Java, todo conjunto de caracteres se denomina cadena (en inglés, “String”) y como abstracción podemos decir que una cadena es un arreglo unidimensional de variables de tipo char.

Hay operaciones básicas para cualquier cadena de caracteres cuyos algoritmos son tediosos y llenos de combinaciones. Recordemos que un caracter simple puede ser una letra minúscula, una letra mayúscula, un número o un símbolo. Por eso, Java provee la clase String que posee muchos métodos para manejar cadenas.

Cuando escribimos la siguiente sentencia:

```
String mensaje = "Hola";
```

Lo que está pasando es que estamos creando un objeto de la clase String cuya referencia es guardada por la variable mensaje. Implícitamente, Java está haciendo lo siguiente:

```
String mensaje = new String("Hola");
```

Java intenta “emular” a las cadenas como si se tratasen de un dato primitivo, por lo que permite asignar directamente una cadena entre comillas y ocultar la creación de la instancia, pero, en realidad, debemos saber que las cadenas son objetos.

Concatenar cadenas

Es posible unir varias cadenas en una sola más grande, utilizando el operador de concatenación.

Cualquier otro tipo de dato primitivo que se concatene con una cadena, se convertirá primero en su representación en formato String. El resultado es una cadena.

La siguiente tabla lo ejemplifica:

Concatenación	Devuelve
`"Hola" + " mundo"`	`"Hola mundo"`
`"Tengo " + 2 + " cuadernos"`	`"Tengo 2 cuadernos"`
`"Pi vale " + 3.14`	`"Pi vale 3.14"`
`"Empieza con " + 'A'`	`"Empieza con A"`
`"¿Difícil?” + false`	`"¿Difícil? false"`

Comparar cadenas

Hay que tener cuidado cuando se realizan comparaciones entre objetos, pues lo que Java compara no son precisamente sus valores (en realidad, un objeto no tiene valores sino estado, determinado por sus atributos), sino que compara las referencias de cada objeto. A no ser un caso particular, que sería el de comparar un objeto con sí mismo, todas las comparaciones entre objetos usando el operador == resultarán false, pues se supone que cada objeto, por más que tenga el mismo estado, es único. Es como preguntarse si dos personas gemelas son la misma persona: **falso**.

Sin embargo, como Java tiene un tratamiento especial para los String, la siguiente comparación resulta **true**:

```
String cadena = "Hola";
String otraCadena = "Hola";
System.out.println( cadena == otraCadena ); // Devuelve true
```

Sin embargo, exactamente el mismo proceso utilizando el operador new de forma explícita resulta **false**:

```
String cadena = new String("Hola");
String otraCadena = new String("Hola");
System.out.println( cadena == otraCadena ); // Devuelve false
```

La explicación de este fenómeno es bastante técnica y tiene que ver con cómo Java trata de manera especial a los String. Lo que queda claro es que en el primer caso compara realmente los valores **"Hola"**, los cuales al ser idénticos provocan que el resultado de compararlos resulte **true**.

En el segundo caso se están comparando las referencias de cada una de las cadenas. Las variables cadena y otraCadena tienen cada una referencia unívoca, que al compararse, por supuesto, provocan que el resultado sea **false**.

El método equals

Para estar siempre seguro de que lo que se está comparando no son las referencias, sino los verdaderos valores de los String, se debe utilizar el **método equals()** que incorpora todo objeto. El método equals() espera recibir como argumento un String y devolverá un valor booleano que indica si el String recibido como parámetro es idéntico al String por el cual se invocó al método.

Procesar cadenas

Como las cadenas son objetos de la clase String, traen incorporados métodos muy útiles para procesarlas.

Lo que es importante destacar es que los String son inmutables, es decir, que una vez definidos, no pueden cambiar su estado. Cualquier operación sobre un String, por ejemplo, convertirlo a mayúsculas, no modifica al actual, sino que devuelve un nuevo String con el resultado.

A continuación, las operaciones más típicas:

Conocer la longitud de una cadena: como toda cadena en realidad es un arreglo, todos los String cuentan con el método **length()**, que devuelve un entero que indica la longitud de la misma:

```
String cadena = "Cadena de prueba";
System.out.println(cadena.length()); // Devuelve 16
```

A diferencia del atributo público length que incorporaba todo arreglo (notar que accedías sin usar los paréntesis), la longitud de una cadena es privada, es decir, que no es posible acceder a ella directamente, sino a través de un método que devuelve su valor. En la programación orientada a objetos, esto se denomina **encapsulamiento**.

Obtener un caracter de una cadena: para obtener un único caracter de cierta cadena, se utiliza el método **charAt()**, que recibe un número entero como argumento que representa la posición del caracter requerido (comenzando a contar desde 0) y devuelve un char, que representa el caracter en la posición correspondiente.

```
String cadena = "Cadena de prueba";
System.out.println(cadena.charAt(0)); // Devuelve 'C'
System.out.println(cadena.charAt(2)); // Devuelve 'd'
System.out.println(cadena.charAt(cadena.length() - 1)); // Devuelve
la última 'a'
```

Extraer una porción de la cadena: para extraer una subcadena de otra, se utiliza el método **substring()** que presenta dos variantes. Se puede invocar a **substring()** con un solo argumento de tipo entero. Devuelve la subcadena correspondiente entre la posición recibida como argumento (incluyendo a ese caracter) y el final de la cadena:

```
String cadena = "Sacacorchos";
System.out.println(cadena.substring(4)); // Devuelve "corchos"
```

La otra variante es utilizando dos argumentos enteros. En tal caso, devuelve la subcadena correspondiente entre la posición recibida como primer parámetro y la posición recibida como segundo parámetro, sin incluir a este último:

```
String cadena = "Sacacorchos";
System.out.println( cadena.substring(0,4) ); // Devuelve "Saca"
```

Otras operaciones:

La siguiente tabla resume las operaciones que incorpora todo String:

Método	Parámetro(s)	Devuelve	Descripción
charAt()	int	char	Devuelve el carácter que se encuentre en la posición recibida como parámetro.
equals()	String	boolean	Devuelve si la cadena es igual a la recibida por parámetro.
equalsIgnoreCase()	String	boolean	Devuelve si la cadena es igual a la recibida por parámetro, ignorando mayúsculas y minúsculas.
indexOf()	String	int	Devuelve la posición donde comienza la cadena recibida por parámetro, desde el principio. Si no existe, devuelve -1.
indexOf()	char	int	Devuelve la posición del carácter recibido por parámetro, desde el principio. Si no existe, devuelve -1.
indexOf()	String , int	int	Devuelve la posición donde comienza la cadena recibida como primer parámetro, desde el valor del segundo parámetro. Si no existe, devuelve -1.
indexOf()	char , int	int	Devuelve la posición del carácter recibido como primer parámetro, desde el valor del segundo parámetro. Si no existe, devuelve -1.
isEmpty()		boolean	Devuelve si la cadena está o no vacía.
lastIndexOf()	String	int	Devuelve la posición donde comienza la cadena recibida por parámetro, desde el final. Si no existe, devuelve -1.
lastIndexOf()	char	int	Devuelve la posición del carácter recibido por parámetro, desde el final. Si no existe, devuelve -1.
lastIndexOf()	String , int	int	Devuelve la posición donde comienza la cadena recibida como primer parámetro,

			desde el valor del segundo parámetro hacia atrás. Si no existe, devuelve -1.
lastIndexOf()	char , int	int	Devuelve la posición del carácter recibido como primer parámetro, desde el valor del segundo parámetro hacia atrás. Si no existe, devuelve -1.
length()		int	Devuelve la longitud de una cadena.
replace()	char , char	String	Devuelve una cadena, donde todas las ocurrencias del carácter recibido como primer parámetro son reemplazadas por el carácter recibido como segundo parámetro.
replace()	String , String	String	Devuelve una cadena, donde todas las ocurrencias de la cadena recibida como primer parámetro son reemplazadas por la cadena recibida como segundo parámetro.
replaceFirst()	String , String	String	Devuelve una cadena, donde la primera ocurrencia de la cadena recibida como primer parámetro es reemplazada por la cadena recibida como segundo parámetro.
split()	String	String[]	Separa la cadena por la cadena recibida como parámetro y devuelve cada segmento en un arreglo de cadenas.
starsWith()	String	boolean	Devuelve si una cadena comienza con la cadena recibida como parámetro.
starsWith()	String , int	boolean	Devuelve si una cadena comienza con la cadena recibida como parámetro, desde la posición recibida como segundo parámetro.
substring()	int	String	Devuelve la cadena resultante entre la posición recibida como primer parámetro y el final de la cadena.
substring()	int , int	String	Devuelve la cadena resultante entre la posición recibida como primer parámetro y la posición recibida como segundo parámetro (ésta última sin incluir).
toCharArray()		char[]	Devuelve la cadena como un arreglo de caracteres.
toLowerCase()		String	Devuelve la cadena en minúsculas.
toUpperCase()		String	Devuelve la cadena en mayúsculas.
trim()		String	Devuelve la cadena sin espacios en blanco al principio y al final.

Conversiones

Hay ocasiones donde es necesario convertir un valor representado por una cadena en el verdadero valor primitivo y viceversa.

Clases envoltorio (wrapper)

Todo tipo primitivo tiene en Java una clase asociada, que integra operaciones y valores especiales para el correspondiente tipo de dato. La siguiente tabla asocia cada dato primitivo con su correspondiente clase envoltorio:

Dato primitivo	Clase envoltorio
char	Character
int	Integer
double	Double
boolean	Boolean

Convertir un primitivo en una cadena

Para convertir cualquier dato primitivo a una cadena, se utiliza el método toString() de la correspondiente clase envoltorio, cuyo argumento es el tipo de dato primitivo. El método devuelve el argumento representado en una cadena.

```
String enteroACadena = Integer.toString(50); // Convierte el 50 en "50"

String doubleACadena = Double.toString(1.44); // Convierte el 1.44 en
"1.44"

String charACadena = Character.toString('x'); // Convierte la 'x' en "x"

String booleanACadena = Boolean.toString(true); // Convierte true en
"true"
```

Aunque hay un "truco" más sencillo, que implica saber que todo primitivo concatenado con una cadena, resulta en una cadena. Por lo tanto, podrías a cada primitivo concatenarlo con una cadena nula, obteniendo el mismo efecto:

```
String enteroACadena = Integer.toString(50); // Convierte el 50 en "50"

String doubleACadena = Double.toString(1.44); // Convierte el 1.44 en
"1.44"

String charACadena = Character.toString('x'); // Convierte la 'x' en "x"

String booleanACadena = Boolean.toString(true); // Convierte true en
"true"
```

```
String enteroACadena = Integer.toString(50); // Convierte el 50 en "50"

String doubleACadena = Double.toString(1.44); // Convierte el 1.44 en
"1.44"

String charACadena = Character.toString('x'); // Convierte la 'x' en "x"

String booleanACadena = Boolean.toString(true); // Convierte true en
"true"
```

Aunque hay un "truco" más sencillo, que implica saber que todo primitivo concatenado con una cadena, resulta en una cadena. Por lo tanto, podrías a cada primitivo concatenarlo con una cadena nula, obteniendo el mismo efecto:

```
String enteroACadena = Integer.toString(50); // Convierte el 50 en "50"

String doubleACadena = Double.toString(1.44); // Convierte el 1.44 en
"1.44"

String charACadena = Character.toString('x'); // Convierte la 'x' en "x"

String booleanACadena = Boolean.toString(true); // Convierte true en
"true"
```

Cualquier intento de parseo de una cadena que no represente un valor numérico a un tipo de dato primitivo cono **int** o **double** provocará una excepción.