UD05 – TIPOS DE DATOS COMPUESTOS

Contenido

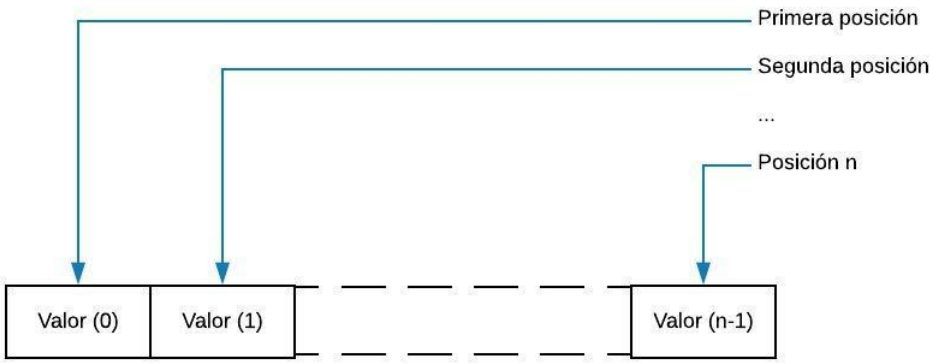
1 Arrays

Supongamos que queremos hacer un programa para analizar las notas finales de la clase. Las operaciones a realizar pueden ser diversas: calcular la nota media, ver cuántos estudiantes han aprobado, suspendido o han superado cierto umbral de nota, hacer gráficas de rendimiento, etc. Además, no descartamos que en el futuro el programa pudiéramos añadir nuevas funcionalidades. Para poder realizar todos estos cálculos y los nuevos que fuéramos a realizar en un futuro, necesitamos disponer dentro del programa del valor de cada nota individual. Tal como hemos planteado hasta ahora la manipulación de datos dentro de un programa, si hay 20 estudiantes en la clase entonces habría que declarar 20 variables. Si se incorporara un estudiante nuevo necesitaríamos añadir una nueva variable. Imaginemos las expresiones que necesitaríamos para calcular la media. O si extendemos la aplicación al instituto, ¿sería factible declarar mil variables?

Ya podemos anticipar que esta aproximación es inviable. Necesitamos una aproximación más flexible para manejar un número de valores que pueda ser distinto en cada ejecución del programa y que no nos obligue a declarar variables individuales para cada elemento. Para resolver esta problemática, los lenguajes de programación de alto nivel ofrecen el tipo de dato compuesto ***array***, (o matriz).

Un **tipo de dato compuesto** es aquel que permite almacenar más de un valor dentro de una única variable. En el caso del **array**, este permite almacenar, en forma de secuencia, una cantidad predeterminada de valores pertenecientes al **mismo tipo** de datos.

Con el fin de diferenciar los diferentes valores almacenados, el array gestiona su contenido de acuerdo con posiciones que siguen un orden numérico: el valor almacenado en la primera posición, en la segunda, la tercera, etc. A efectos prácticos, se puede considerar que cada posición individual se comporta exactamente igual que una variable del tipo de dato elegido. Los arrays permiten tanto consultar el dato en una posición concreta como modificarlo.



En general, los arrays en la mayoría de los lenguajes de programación empiezan en el índice cero, por lo que, si un array contiene n valores, el último lo encontramos en el índice (n-1).

Antes de ver con más detalle cómo funcionan, debemos tener presente que la sintaxis para usar arrays en los diferentes lenguajes de programación puede ser relativamente diferente. Veremos aquí la sintaxis en Java, que tiene ciertas particularidades únicas. Sin embargo, el concepto general de acceso a valores ordenados como una secuencia en forma de *array* es aplicable a cualquier lenguaje.

1.1 Declaración e inicialización de arrays

Para declarar un array, además del indicar el tipo de dato añadiremos **[]**.

```
Tipo[] identificadorArray;
```

Por ejemplo, para declarar un array de enteros

```
int[] myArray;
```

Una forma de inicializar el array dando valor a todos los elementos del array:

```
Tipo[] identificadorArray = {valor1,valor2,valor3,…,valorN};
```

Los elementos del array deben pertenecer al mismo tipo que el array, se ponen entre llaves **{}** y separados por comas. No puede haber datos de tipo mezclados dentro de un array.

El siguiente ejemplo declara un array de enteros con los valores 10, 20, 30, 40 y 50.

```
int[] myArray = {10,20,30,40,50};
```

No siempre vamos a conocer de antemano el valor de los elementos del array, así que una alternativa para inicialización es indicar el número de elementos que contendrá el array. Lo expresaremos de la siguiente manera:

```
Tipo[] identificadorArray = new Tipo[tamaño];
```

En la inicialización se usa la palabra clave ***new***, seguida nuevamente del tipo de los datos del array y, entre corchetes, el tamaño que queramos. Los tipos de datos especificados en la parte izquierda y derecha deben ser idénticos, o habrá un error de compilación.

No hay que hacer nada más para asignar valores a las posiciones del array, Java ya asigna automáticamente el valor predeterminado para el tipo de datos declarado. Ahora bien, esta es una propiedad **específica** de Java y no debe ser así necesariamente en otros lenguajes, en la que el valor inicial de cada posición puede ser indeterminado.

El valor predeterminado para los tipos de datos numéricos corresponde con el cero. Puedes encontrar más información sobre los valores por defecto para cada tipo de dato en este enlace.

Por ejemplo, para un array de reales con 100 posiciones, en cada una de las 100 posiciones en valor del array será 0.0, al ser el valor por defecto para este tipo de dato en Java.

```
double[] datos = new double[100];
```

1.2 Manipulación de datos en los arrays

No existen operaciones entre arrays. No es posible usar el identificador del array directamente para invocar operaciones y así manipular los datos contenidos, tal como se puede hacer con variables de otros tipos. Por ejemplo, no es posible hacer lo siguiente:

```
int[] a = { 7 , 1 , 0 , 40 , -50 } ;
int[] b = { 50 , 25 , 75 , 80 , 10 } ;
int[] c = a + b ;
```

Siempre que se queramos hacer alguna operación con los datos almacenados dentro de un array hay manipularlos de forma individual, posición por posición. En este

aspecto, cada posición de un array tiene exactamente el mismo comportamiento que una variable tal como las hemos vistos hasta ahora.

Cada posición de un array, tiene asignado un **índice**, un valor entero que indica el orden dentro de la estructura. Siempre que queramos referirnos a una de las posiciones, basta con usar el identificador del array junto con el índice de la posición entre corchetes []. La sintaxis exacta es:

```
identificadorArray[índice]
```

El rango de los índices puede variar según el lenguaje de programación. En el caso del Java y para un array de N elementos, estos van de 0, para la primera posición, a (N - 1), para la última. Por ejemplo, para acceder a las posiciones de un *array* de cinco posiciones usaría los índices 0, 1, 2, 3 y 4. A continuación se muestra un ejemplo de cómo se accede a los datos de un *array* mediante los índices de sus posiciones:

```
...
int [] a = { 10 , 20 , 30 , 40 , 50 } ;
int [] b = { 50 , 60 , 70 , 80 , 100 } ;
int [] c = new int[5];
// La variable de tipo entero "r" valdrá 140, 40 + 100.
int r = a[3] + b[4];
// La segunda posición (índice 1) de "c" valdrá la suma de
// las mismas posiciones de los arrays "a" y "b".
c[1] = a[1] + b[1];
...
```

Es muy importante que utilicemos el índice correcto para acceder a los elementos de un array. Si se usa un valor fuera del rango definido para el array, como por ejemplo un valor negativo o un valor igual o superior a su tamaño, habrá un error de ejecución. Concretamente, se producirá una IndexOutOfBoundsException. El código siguiente sería incorrecto

```
...
// "a" tiene tamaño 5.
// Los índices válidos van de 0 a (tamaño - 1), que es 4.
int[] a = { 10 , 20 , 30 , 40 , 50 } ;
// Se asigna un valor a una posición incorrecta (índice 5).
a[5] = 60 ;
// El índice 6 es inválido, no se puede consultar el valor.
int r = a[2] + a[4] + a[6];
...
```

Para controlar esta situación, las variables de tipo array tienen el atributo ***length***, cuyo valor corresponde con el tamaño del array.

```
identificadorArray.length
```

El uso de este valor nos permite determinar si un índice es o no válido para un array

```
...
// "a" tiene tamaño 5. Los índices válidos van de 0 a 4.
int[] a = {10,20,30,40,50};
// "length" le dice tamaño.
// Recuerda que el índice máximo es (tamaño - 1).
if (index>=0 && index<array.length) {
  System.out.println("La posición" + index + "vale" + a[index]);
} Else {
  System.out.println("El índice no es válido");
}
...
```

1.3 Recorrer los datos de un array

Hablamos de recorrido cuando hay que tratar todos los valores del array. Muy a menudo tendremos que hacer cálculos en que estarán implicados todos los valores almacenados. En este caso, hay que hacer un recorrido secuencial de las posiciones e ir procesando cada uno de los valores para obtener un resultado final. Ejemplos de este tipo de acciones puede ser calcular la suma total de valores, obtener la media, el valor máximo de los elementos de un array etc.

En el siguiente ejemplo se calcula la media y la desviación media de una serie de valores que introduce el usuario a través de la consola. Se utilizan dos aproximaciones para recorrer el bucle; con un ***for*** y con un ***foreach***.

```java
import java.util.Scanner;

public class Average {
	// objeto Scanner para gestionar la entrada
	static Scanner sc;

	public static void main(String[] args) {
		// inicicializamos el escanner
		sc = new Scanner(System.in);
		// pedimos el número de elementos del array
		System.out.println("Cálculo de la media aritmética");
		System.out.println("¿Cuántos valores necesita introducir?");
		while (!sc.hasNextInt()) {
			sc.nextLine();
			System.out.println("¿Cuántos valores necesita introducir?”
				+ "(Debe ser un entero)");

		}
		int length = sc.nextInt();
		sc.nextLine(); // limpiar scanner
		// declarar array
		double[] values = new double[length];
		// declarar indice para rellenar el array
		int index = 0;

		// Pedimos introducir los valores
		System.out.println("Introduzca los valores (números reales)");
		System.out.println("Puede hacerlo uno a uno");
		System.out.println("o varios en una línea separados por espacios");
		// vamos leyendo del scanner hasta completar la inserción
		// de la cantida de valores esperada
		while (index < length) {
			while (!sc.hasNextDouble()) {
				sc.nextLine();
			}
			values[index] = sc.nextDouble();
			index++;
		}

		System.out.println("Valores introducidos");
		// recorremos el array con un for
		// y vamos sumando el total
		double total = 0.0;
		for (int i = 0; i < values.length; i++) {
			System.out.println(values[i]);
			total += values[i];
		}
		// calculamos la media
		double avg = total / length;
		System.out.println("Media aritmética: " + avg);

		// recorrer el array para mostrar la desviación media
		// ahora recorremos el array con un foreach
		// Desviación media = abs(valor - avg)
		for (double d : values) {
			System.out.println("Valor: " + d
				+ " - Desviación media: " + Math.abs(d - avg));
		}

		// cerrar scanner
		sc.close();
	}

}
```

1.4 Arrays como parámetros

Vimos en la unidad anterior que en Java los parámetros se pasan a las variables siempre por valor. Esto significa que si modificamos el valor de la variable dentro del método al que hemos pasado la variable no este cambio no tendrá repercusión fuera del método al que hemos llamado. Por otro lado, hemos visto las diferencias entre variables de tipos de dato primitivos y variables de objeto cuando aplicamos el operador ***==***, por ejemplo. Vimos que la principal diferencia es que en una variable de tipo de dato primitivo está asociada al valor que guardado en memoria para esa variable y en los tipos de datos asociados a objetos la variable está asociada al puntero en memoria que guarda dicha variable. Es decir, dos objetos eran iguales si estaban asociados al mismo puntero (a la misma dirección de memoria).

Los arrays en Java son tratados como objetos, esto significa que cuando pasamos un array como parámetro a un método, si dentro del método hacemos una nueva asignación al array esta asignación no tiene repercusión fuera del método. Veamos el siguiente ejemplo:

```
import java.util.Random;

//import java.util
public class DoubleValues {

	public static void main(String[] args) {
		// creamos un array con 5 elementos
		int[] array = { 1, 2, 3, 4, 5 };

		// llamamos al método que cambiará
		// la asignación del array
		modifyArray(array);

		for (int i = 0; i < array.length; i++) {
			System.out.print(array[i] + " ");
		}
		System.out.println();

	}

	// recibimos el array y modificamos su asignación
	// dentro del método para duplicar los valores
	private static void modifyArray(int[] array) {
		int[] a = { 2, 4, 6, 8, 10 };
		array = a;
	}

}
```

En cambio, si lo que hacemos dentro del método es cambiar el valor individual de los elementos del array sí tendrá repercusión fuera del método.

```
import java.util.Random;

//import java.util
public class DoubleValues {

	public static void main(String[] args) {
		// creamos un array con 5 elementos
		int[] array = { 1, 2, 3, 4, 5 };

		// llamamos al método que multiplica por 2
		// cada valor del array
		duplicateArray(array);

		for (int i = 0; i < array.length; i++) {
			System.out.print(array[i] + " ");
		}
		System.out.println();

	}

	// recorre el array y multiplica por 2 cada elemento
	private static void duplicateArray(int[] array) {
		for (int i = 0; i < array.length; i++) {
			array[i] *= 2;
		}

	}

}
```

Esto sucede porque en realidad no ha cambiado el "valor" de la variable *array,* que es un puntero que sigue apuntando a la misma dirección de memoria, lo que ha cambiado son los datos que están almacenados en dicha dirección de memoria.

1.5 Búsqueda secuencial en arrays

Otra tarea que a menudo se hace en trabajar con arrays es buscar valores concretos en su interior. Por eso, hay que ir mirando todas las posiciones hasta encontrarlo. Por ejemplo, buscar si encontramos una ciudad en una lista.

Hasta cierto punto, se puede considerar que una búsqueda no es más que un tipo especial de recorrido. En este caso, el énfasis se hace en la circunstancia de que no siempre hay que recorrer todos los elementos del array, sólo hasta encontrar el valor que buscamos. De hecho, llegar al final del array es lo que indica que no se ha encontrado el valor buscado.

El código siguiente hace la búsqueda de una ciudad introducida por el usuario en conjunto de destinos disponibles en la aplicación. En este código la ejecución sale del bucle en cuanto se encuentra el valor que buscamos. Ya no hay que seguir iterando.

```
import java.util.Scanner;

public class Search {

	public static void main(String[] args) {
		// destinos disponibles
		String[] destinations = {
			"Madrid", "Londres", "Roma", "Tenerife", "París"
			, "Los Angeles", "Boston", "Nueva York"
			, "Filadelfia", "Santiago", "Buenos Aires"
			, "Bogotá", "Sao Paulo", "Atenas", "Lisboa"
			, "Barcelona", "Ginebra", "Bruselas"
		};
		// inicializamos el objeto escaner para la entrada por teclado
		Scanner sc = new Scanner(System.in);

		System.out.println("¿Dónde quieres viajar?");
		String destination = sc.nextLine();

		int i = search(destinations, destination);

		// verificamos si la búsqueda ha tenido éxito
		if (i == -1) {
			System.out.println("El destino no se encuentra en nuestro sistema");
		} else {
			System.out.println(destination + " tiene el código "
				+ i + " en nuestro sistema");
		}

		// cerramos el objeto escaner
		sc.close();

	}

	private static int search(String[] destinations, String destination) {
		int i = 0;
		// recorremos el array para buscar el objeto
		for (int j = 0; j < destinations.length; j++) {
			// comparamos en minúsculas para hacer que la búsqueda sea
			// independiente de mayúsculas y minúsculas
			if (destinations[i].toLowerCase().equals(destination.toLowerCase())) {
				return i;
			}
		}
		// devolvemos -1 si no encontramos nada
		return -1;
	}

}
```

Podríamos mejorar nuestro programa añadiendo una función de búsqueda que permita buscar a partir de un índice dado, de forma que nos permita buscar más ocurrencias en el array para un elemento dado.

1.6 Ordenación de arrays

Hay veces en que resulta muy útil reorganizar los valores que hay dentro del array de manera que estén ordenados de manera ascendente o descendente. En muchos casos, mostrar los datos de manera ordenada es simplemente un requisito de la aplicación, de manera que el usuario las pueda visualizar y extraer conclusiones de manera más cómoda. Pero hay casos en que disponer de los datos ordenados facilita la codificación del programa y la eficiencia.

En general, los datos ordenados resultan de utilidad a la hora de realizar búsquedas o recorridos en que los datos para tratar tienen un rango o un valor concreto, ya que permiten establecer estrategias que minimicen los números de iteraciones de los bucles.

Para ordenar una secuencia de valores hay varios algoritmos. Los más eficientes son bastante complejos y no los abordaremos por ahora. Veremos el algoritmo de la burbuja, que es uno de los más sencillos.

El **algoritmo de la burbuja** sirve para ordenar elementos mediante recorridos sucesivos a lo largo de la secuencia, en la que se comparando parejas de valores. Cada vez que se encuentran dos parejas en orden incorrecto, intercambia su posición.

Básicamente, lo que se hace es buscar el valor más alto, recorriendo todo el array, y ponerlo en la última posición. Con esto ya tenemos la garantía de haber resuelto qué valor debe terminar en la última posición. Entonces se busca el segundo valor más alto, recorriendo la secuencia desde principio hasta la segunda posición por el final, y se pone en esta penúltima posición. Y la operación se va repitiendo hasta llegar a la primera posición. Para hacer esta tarea se opera directamente sobre los valores almacenados en el *array*, intercambiándolos cada vez que encontramos un valor más alto que el que hay en la posición que se está tratando.

En el siguiente enlace está explicado con todo detalle el algoritmo de la burbuja o BubbleSort. También podemos encontrar otros algoritmos de ordenación.

El código sería el siguiente:

```
import java.util.Random;

public class BubbleSort {

	public static void main(String[] args) {
		// generamos un array con valores aleatorios
		// y mostramos el array desordenado
		int length = 10;
		int maxValue = 100;
		Random r = new Random(System.currentTimeMillis());
		int[] a = new int[length];
		System.out.print("[");
		for (int i = 0; i < length; i++) {
			a[i] = r.nextInt(maxValue);
			System.out.print(a[i] + " ");
		}
		System.out.print("]\n");

		// ordenación
		bubleSort(a);

		//Mostramos el array ordenado
		System.out.print("[");
		for (int i = 0; i < a.length; i++) {
			System.out.print(a[i] + " ");
		}
		System.out.print("]\n");
	}

	// ordena por el algoritmo de la burbuja
	private static void bubleSort(int[] a) {

		// primer bucle de 0 a length-1
		// (el penúltimo ya se compara con el último)
		// segundo bucle va de 0 length - indice_primer_bucle - 1
		// ya que en cada paso del primer bucle el último elemento
		// ya está ordenado y es el mayor
		for (int i = 0; i < a.length-1; i++) {
			for (int j = 0; j < a.length-i-1; j++) {
				// comparamos cada elemento con el siguiente
				// si el primero es mayor
				// intercambiamos las posiciones
				if(a[j]>a[j+1]) {
					interchange(a, j, j+1);
				}
			}
		}
	}

	// función para intercambiar los valores de un array
	// para dos posiciones dadas
	// funciona ya que un array es una variable de tipo objeto
	// asociada al puntero a memoria
	private static void interchange(int[] a, int i, int j) {
		int aux = a[j];
		a[j] = a[i];
		a[i] = aux;
	}
}
```

Los algoritmos QuickSort y MergeSort están entre los algoritmos de ordenación más eficientes

1.7 Búsqueda binaria

La búsqueda secuencial puede aplicarse sobre cualquier tabla, sin embargo, si está ordenada podemos usar la búsqueda binaria, que es un algoritmo más óptimo. En la búsqueda binaria, se va al elemento central del array y se compara con el elemento buscado, si coincide termina la búsqueda, si el elemento buscado es menor se buscará en la primera parte del array y si es mayor se buscará en la parte superior. Se continua de forma sucesiva hasta que no haya más elementos que buscar o se encuentre el elemento.

```
// búsqueda binaria
int binarySearch(int[] a, int find) {
	// calculamos el primer y último elemento
	int ini = 0;
	int end = a.length - 1;
	// entramos en bucle mientras el existan elementos
	// en el rango de búsqueda
	while (ini <= end) {
		// calculamos la posición central
		int mid = (ini + end) / 2;
		// verificamos la búsqueda
		if (a[mid] == find) {
			// si se encuentra el elemento devolver el índice
			return mid;
		} else if (find < a[mid]) {
			// si el valor búscado es menor que el elmento central
			// buscar en el rango inferior
			end = mid - 1;
		} else {
			// si el valor búscado es mayor que el elmento central
			// buscar en el rango superior
			ini = mid + 1;
		}
	}
	// devolver -1 si no se encuentra el elemento
	return -1;
}
```

En este caso se ha implementado una solución iterativa, podríamos haber hecho una implementación recursiva de búsqueda.

1.8 Clase Arrays

La clase Arrays presenta una serie de funcionalidades que permite manipular con arrays de una forma más cómoda y eficiente. Veamos algunas de ellas.

El método *copyOf(original, newLength)* crea un nuevo array de tamaño *newLength* a partir de un array existente, si el tamaño de la copia es menor que el original se truncarán los elementos sobrantes, si es mayor se rellenará con el valor por defecto.

```
int[] a = {1,2,3};
int[] c = Arrays.copyOf(a, 5);
```

El array *c* tiene los tres primeros elementos igual que *a* y los dos últimos son cero que es el valor por defecto del tipo *int*.

```
int[] b = {4,5};
System.arraycopy(b, 0, c, 3, 2);
```

Estas líneas permiten copiar parte del array *b* sobre las posiciones indicadas del array *c*.

De manera similar a *Arrays.copyOf* disponemos de *Arrays.copyOfRange*(original, from, to), que devolverá un array de longitud *(from – to)* tomando elementos del array original desde el índice *from* (incluido) hasta *to* (no incluido). Se rellenarán con el valor por defecto del tipo del array si hubiera posiciones sobrantes.

```
int[] d = Arrays.copyOfRange(c, 1, 3);
```

El método *sort* permite ordenar el array

```
Arrays.sort(c);
```

Con el método binarySearch podemos realizar búsquedas en un array ordenado. Es imprescindible que el array esté ordenado para poder realizar una búsqueda binaria.

```
Arrays.binarySearch(c, 3);
```

2 Cadenas de caracteres.

2.1 Objeto String

El tipo de dato compuesto **cadena de texto** o String sirve para representar y gestionar de manera sencilla una secuencia de caracteres. En Java, el tipo de dato compuesto cadena de texto se identifica con la palabra clave *String*.

Crear Strings

La manera más directa de crear un String es:

```
String greeting = “Hello world!”;
```

En este caso, “Hello world!” es un literal de String, una serie de caracteres delimitados por comillas dobles (“). Siempre que el compilador encuentre un literal de String en nuestro código creará un objeto String con el valor correspondiente.

Como con cualquier otro tipo de objeto podemos crear objetos String usando la palabra clave ***new*** y el nombre de la clase. La clase String permite proporcionar el valor inicial del objeto de diferentes formas, por ejemplo, utilizando un array de caracteres y otras cadenas:

```
char[] arrayHello = {‘H’, ‘e’, ‘l’, ‘l’, ’o’};
String hello = new String(arrayHello);
String world = new String(“world”);
String greeting = hello + “ “ + world + “!”;
```

Existe el concepto de cadena vacía, es decir, que no contiene ningún carácter. Se representa poniendo dos comillas dobles sin nada entre ellas:

```
String emptyString = “”;
```

La clase String es inmutable, es decir una vez creada un objeto de tipo String éste no puede ser cambiado. La clase String tiene un conjunto de métodos que parece modificar cadenas. Lo que en realidad hacen es crear y devolver una nueva cadena conteniendo el resultado de la operación. Pero nunca se verá modificado el valor inicial de la cadena de texto original.

Sí podremos asignar un nuevo valor a una variable de tipo String después de inicializarla. Esto remplazará la cadena que guardaba por otra nueva (nuevo puntero a memoria) pero no modificará la cadena original.

Por ejemplo, en la primera línea se declara el objeto str (String) y se inicializa con el valor “first value”, en la segunda línea se instancia el objeto String con el valor “new value” y se asigna este nuevo objeto (puntero) a str. El puntero a la cadena “first value” deja de estar relacionado con la variable str y podrá ser liberado por el GarbageCollector. No se produce una modificación de los datos en memoria a los que apunta str, sino que se reemplaza el puntero para apuntar a otro objeto.

```
String str = “first value”;
str = “new value”;
```

Acceso a los caracteres de una cadena de texto

De igual manera que en un array, en el objeto String existe el método ***length()*** que devuelve la longitud de una cadena de caracteres. Para recorrer los caracteres de una cadena podremos acceder por su índice, que como en el caso de los arrays va de ***0*** a ***length() - 1***. El método que permite acceder a un carácter es ***charAt(index)***.

En el siguiente ejemplo vemos como obtener (de manera ineficiente) el inverso de un texto

```
public class ReverseString {

	public static void main(String[] args) {
		// creamos la cadena de texto
		String str = "Hello world";
		// creamos un array de caracteres
		// de la misma longitud que la cadena
		char[] chars = new char[str.length()];
		// recorremos la cadena accediendo
		// a los caracteres uno a uno
		for (int i = 0; i < str.length(); i++) {
			// vamos colocando los caracteres leidos
			// en el array en orden inverso a la lectura
			chars[str.length()-1-i] = str.charAt(i);
		}
		// creamos una nueva cadena a partir del
		// array de caracteres que hemos construido
		String reverse = new String(chars);
		System.out.println(reverse);
	}

}
```

Podemos obtener directamente un array de caracteres a partir de un String con el método ***getChars***:

```
char[] c = new char[str.length()];
str.getChars(0, str.length(), c, 0);
```

Concatenación de cadenas

A diferencia de los tipos primitivos, cualquier objeto almacenado en una variable perteneciente a una clase (es decir, cualquier valor almacenado en una variable de un tipo de dato compuesto), no puede ser manipulado directamente mediante operadores de ningún tipo. No hay operadores que manipulen o comparen directamente objetos. Esto es válido también para las cadenas de caracteres. Es evidente que no podemos aplicar el producto o la división a cadenas de caracteres, pero tampoco podemos utilizar operadores relacionales como > ó >=. Ya hemos visto como funciona el operador igual (==) con objetos devolviendo si dos objetos son el mismo y no si los valores que almacenan los objetos son iguales.

Tenemos que recordar que el resultado del siguiente bloque de código será *false* ya que los objetos str1 y str2 son distintos, aunque contengan el mismo texto:

```
char[] arrayHello = {‘H’, ‘e’, ‘l’, ‘l’, ’o’};
String str1 = new String(arrayHello);
String str2 = new String(arrayHello);
System.out.println(str1==str2);
```

El único operador permitido para los objetos String es **+**, que para el caso de String realiza la concatenación. Así, el resultado de:

```
"Hello," + " world" + "!"
```

será

```
"Hello, world!"
```

Esta concatenación puede ser una mezcla de cualquier objeto. Para los objetos que no sean de tipo String, se realizará su conversión implícita a String.

```
"3+2=" + (3+2)
```

Da como resultado:

```
"3+2=5"
```

También se puede utilizar el método ***concat***

```
string1.concat(string2);
```

Esta expresión devolverá una nueva cadena resultado de añadir string2 al final de string1.

Dar formato a las cadenas de caracteres

El método estático **format** de la clase *String* permite crear textos con un determinado formato y de manera que este formato sea reutilizable. Vemos un ejemplo:

```
String fs;
fs = String.format("The value of the float " +
                   "variable is %f, while " +
                   "the value of the " +
                   "integer variable is %d, " +
                   " and the string is %s",
                   floatVar, intVar, stringVar);
System.out.println(fs);
```

Vemos que %d se utiliza para dar formato a enteros, %f a reales y %s permite introducir cadenas de texto. Esta misma aproximación puede usarse para imprimir directamente por consola:

```
System.out.printf("The value of the float " +
                  "variable is %f, while " +
                  "the value of the " +
                  "integer variable is %d, " +
                  "and the string is %s",
                  floatVar, intVar, stringVar);
```

Realizar búsquedas en cadenas de caracteres

Los métodos ***indexOf*** y ***lastIndexOf*** permiten buscar caracteres o subcadenas dentro de una cadena de caracteres. El método *indexOf* comenzará desde el inicio de la cadena hacia delante y *lastIndexOf* comenzará desde el final hacia atrás. Estos métodos devolverán el índice de la cadena o carácter encontrado dentro de la cadena de caracteres o -1 en caso de no haber encontrado el elemento buscado.

Además, también disponemos del método *contains* que devolverá verdadero en caso de encontrar una secuencia de caracteres dentro de un *String* y falso en caso contrario. Este método lo utilizaremos si únicamente estamos interesados en saber si la cadena contiene la secuencia y no dónde se encuentra dicha secuencia.

Veamos el siguiente ejemplo de uso de estos métodos:

```
public class SearchString {

	public static void main(String[] args) {
		// declaramos e instanciamos la cadena de texto con información sobre Java
		String info = "The Java programming language" +
				" is a high-level language that can be characterized" +
				" by all of the following buzzwords:\n" +
				"Simple, Object oriented, Distributed, " +
				"Multithreaded, Dynamic, Architecture neutral, " +
				"Portable, High performance, Robust, Secure.";
		// comprobamos si contiene la palabra "Java"
		System.out.println("Phrase contains \"Java\": " + info.contains("Java"));

		// buscamos las palabras object y Object
		// recordamos que Java distingue entre mayúsculas y minúsculas
		search(info, "object");
		search(info, "Object");

		//buscamos la primera, segunda y última posición del caracter 'a'
		int pos = info.indexOf('a');
		System.out.println("First \'a\' appears in the phrase at position: " + pos);
		pos = info.indexOf('a',pos+1);
		System.out.println("Second \'a\' appears in the phrase at position: " + pos);
		pos = info.lastIndexOf('a');
		System.out.println("Last \'a\' appears in the phrase at position: " + pos);

	}

	// este método busca una cadena dentro de otra
	// y muestra la posición de la cadena buscada dentro de la otra
	// o escribe que no la ha encontrado en caso contrario
	static void search(String str, String find) {
		int pos = str.indexOf(find);
		if(pos==-1) {
			System.out.println("\"" + find
				+ "\" has not be found within the phase.");
		} else {
			System.out.println("\"" + find
				+ "\" is in the phase at position " + pos + ".");
		}

	}
}
```

Manipular de cadenas de caracteres

Ya hemos visto que el método ***charAt*** devuelve el carácter presente en un índice concreto de una cadena. El índice del primer carácter es ***0*** y el del último es ***length()-1***.

```
String greeting = "Hello world!";
char c = greeting.charAt(1);
```

0	1	2	3	4	5	6	7	8	9	10	11
H	e	l	l	o		w	o	r	l	d	!

El valor de c resultará ser ‘e’.

Si queremos obtener más de un carácter consecutivo de un String podemos llamar al método ***substring***. Este método tiene 2 versiones:

```
String substring(int beginIndex, int endIndex)
```

Devuelve una nueva cadena que es una subcadena de la cadena original, comenzando en el carácter ***beginIndex*** y terminando en ***endIndex-1.***

```
String substring(int beginIndex)
```

Devuelve una nueva cadena que es una subcadena de la cadena original, comenzando en el carácter ***beginIndex*** y terminando en el final de la cadena de caracteres**.**

Así, retomando el ejemplo anterior,

```
String greeting = "Hello world!";
String word1 = greeting.substring(0,5);
String word2 = greeting.substring(6);
```

Resultará que word1 contiene el texto “Hello” y el valor de word2 es “world!”.

El método split busca el valor pasado por parámetro y divide la cadena de caracteres en un array de tipo String usando el parámetro como separador.

Veamos el siguiente ejemplo:

```
public class SplitText {

	public static void main(String[] args) {
		// creamos dos cadenas con texto
		String phrase1 = "The String class represents "
				+ "character strings";
		String phrase2 = "The class String includes methods "
				+ "for examining individual characters of the "
				+ "sequence, for comparing strings, for "
				+ "searching strings, for extracting substrings"
				+ ", and for creating a copy of a string with "
				+ "all characters translated to uppercase or "
				+ "to lowercase";

		// dividimos la primera frase por los espacios
		// obtendremos un array con las palabras que contiene
		String[] words = phrase1.split(" ");
		for (int i = 0; i < words.length; i++) {
			System.out.println(words[i]);
		}

		System.out.println();

		// dividimos la segunda frase por las comas
		// de manera que obtenemos un array que
		// contiene como elmentos las oraciones
		// sububordinada de la frase
		String sentences[] = phrase2.split(",");
		for (String sentence : sentences) {
			System.out.println(sentence);
		}
	}
}
```

El método ***trim*** devuelve una copia de la cadena eliminando los espacios que pudiera haber al principio y al final de la cadena.

Los métodos ***toLowerCase*** y ***toUpperCase*** devuelven una copia de la cadena convertida a minúsculas y mayúsculas respectivamente.

Veamos un ejemplo:

```
public class ManipulateStrings {

	public static void main(String[] args) {
		// creamos un string con una lista de palabras separadas por comas
		String str = "Clase, Objeto, Instancia, Puntero, variable”
			+ ”, CONSTANTE, literal, método ";

		// pasamos la cadena a mayúsculas y
		// dividimos la cadena por las comas
		// el resultado es un array con las palabras
		// pero contendrá el espacio que hay tras cada coma
		// el método trim eliminará estos espcacios que sobran
		String[] words = str.toUpperCase().split(",");
		for (String word : words) {
			System.out.println(word.trim());
		}
	}

}
```

El método ***replace*** permite reemplazar caracteres o subcadenas dentro de una cadena. Ojo, no remplaza ningún trozo de la cadena, sino que genera una nueva cadena con los remplazos hechos.

```
String str = "Hello world!";
System.out.println(str.replace("Hello", "Bye, bye,"));
System.out.println("gobernment".replace('b','v'));
```

Comparar de cadenas de caracteres

Ya hemos visto que el operador ***==*** no evalúa si el texto contenido en dos variables de tipo *String* es el mismo o no, sino que evalúa si las dos variables referencian al mismo objeto. Para comprobar si dos variables de tipo cadena contienen el mismo texto podemos utilizar el método ***equals***, esta comparación se hará haciendo distinción entre mayúsculas y minúsculas. Si queremos ignorar las diferencias entre mayúsculas y minúsculas podemos usar el método ***equalsIgnoreCase***.

También podemos comprobar si una cadena empieza o termina con una determinada secuencia de caracteres utilizando los métodos ***startsWith*** y ***endsWith*** respectivamente.

El método ***compareTo*** compara dos cadenas aplicando el orden alfabético devolviendo un valor entero que será mayor que cero, igual a cero o menor que cero en función de dicha comparación.

Las siguientes sentencias mostrarán *true*

```
System.out.println("A".compareTo("B")<0);
System.out.println("B".compareTo("B")==0);
System.out.println("C".compareTo("B")>0);
```

Si queremos una comparación independiente de mayúsculas y minúsculas utilizaremos ***compareToIgnoreCase***.

Convertir números a cadenas de caracteres

Estamos familiarizados con la conversión que realiza el operado ***+*** cuando concatena tipos numéricos con cadenas, antes de poder hacer la concatenación, Java convierte de manera intrínseca el valor numérico a String.

```
System.out.println("3+2=” + (3+2));
```

El método estático valueOf permite convertir de tipos numéricos a cadena de caracteres:

```
String s = String.valueOf(5);
```

Observa que llamamos al método directamente a través de la clase String sin declarar ni inicializar un objeto de tipo String.

Veremos más adelante en el apartado Clases Wrapper cómo convertir cadenas de caracteres a tipos numéricos.

2.2 Objeto StringBuilder

Ya hemos estudiado que los objetos de la clase *String* son inmutables, es decir, una vez inicializada una variable no podemos cambiar su contenido. Si asignamos un nuevo valor a una variable de tipo *String* en realidad estamos cambiando la asignación de la variable a un objeto diferente, pero no modificamos el valor del objeto existente. La razón de este comportamiento es para mejorar el rendimiento de los objetos *String* y sus operaciones. Ya que se trata de un tipo muy utilizado.

Este comportamiento es el más adecuado en la mayoría de las situaciones, pero si nuestro programa tiene que realizar un número elevado de transformaciones sobre cadenas de caracteres el uso del tipo *String* resultará poco eficiente. Java dispone del tipo ***StringBuilder***, los objetos de este tipo son similares a los de tipo String salvo que sí que pueden ser modificados. Internamente los objetos de tipo *StringBuilder* son tratados como arrays de longitud variable que contienen una secuencia de caracteres. En cualquier momento, podemos modificar la longitud y el contenido de la secuencia de caracteres guardados en el objeto StringBuilder a través de invocaciones a sus métodos.

La clase *StringBuilder*, como la clase *String*, tiene un método ***length()*** que devuelve la longitud de la secuencia de caracteres guardada en el objeto.

A diferencia de los strings, el objeto StringBuilder dispone del método ***capacity()*** que devuelve el número de caracteres que han sido reservados en memoria, que será siempre mayor o igual a *length()* (normalmente mayor). La capacidad se expandirá automáticamente conforme se vaya necesitando al modificar el contenido del objeto.

Podemos inicializar un objeto StringBuilder de varias formas; creado un objeto vacío (reserva una capacidad de 16 caracteres), indicando una capacidad dada o proporcionando una cadena de texto o una secuencia de caracteres cuyo valor se utilizará para inicializar el objeto.

Por ejemplo:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder(32);
StringBuilder sb3 = new StringBuilder(“Hello world”);
```

Disponemos de los métodos ***append***, ***delete***, ***insert*** para añadir, borrar o insertar caracteres a nuestra cadena.

En la ayuda de Java podemos encontrar los detalles de las opciones de los métodos disponibles en la clase StringBuilder.

El método ***replace*** permitirá realizar sustituciones de subcadenas dentro de la cadena original. El método ***reverse*** hace que la cadena sea sustituida por la secuencia de caracteres en orden inverso.

Como en la clase *String*, disponemos de los métodos de ***indexOf*** y ***lastIndexOf*** para realizar búsquedas.

En la clase *String* podíamos acceder a un carácter de la cadena usando su índice a través del método ***charAt***, pero no podíamos modificarlo. En *StringBuilder*, además de *charAt*, tenemos ***setCharAt*** y ***deleteChatAt*** para modificar y borrar el carácter en una posición dada de la cadena respectivamente.

El método ***toString*** devuelve un objeto *String* con la misma secuencia de caracteres.

Veamos el siguiente ejemplo de uso de *StringBuilder*

```
public class Builder {
	public static void main(String[] args) {
		// creamos un objeto StringBuilder a partir de una cadena
		StringBuilder sb = new StringBuilder("hello");
		// modificamos un caracter individual
		sb.setCharAt(0, 'H');
		// añadimos un nuevo caracter
		sb.append('!');
		// insertamos un string en la posicion que necesitamos
		sb.insert(5, "world");
		// insertamos un guión tras "Hello"
		sb.insert(5, "-");
		System.out.println(sb.toString());
		// reemplazamos el guión por un espacio
		sb.replace(5, 6, " ");
		System.out.println(sb.toString());
		// borramos el signo de exclamación
		sb.deleteCharAt(sb.length() - 1);
		// damos la vuelta a la cadena
		sb.reverse();
		System.out.println(sb.toString());
	}
}
```

En la mayoría de las situaciones nuestros programas no van a modificar cadenas, o lo van a hacer de manera esporádica, por lo que el tipo que debemos utilizar en la mayoría de los casos es *String* para manejar cadenas de caracteres. Si nuestro programa hace modificaciones repetidas de las secuencias de caracteres entonces será más eficiente realizar estas transformaciones utilizando el tipo *StringBuilder*.

3 Paso de argumentos al método principal

Si nos fijamos en la declaración método *main* de una clase Java vemos que estamos definiendo un parámetro de tipo array de *String* llamado *args.* Esta opción permite establecer una entrada de datos en el programa sin tener que interrumpir la ejecución por tener que preguntar algo al usuario y esperar una entrada por teclado.

Esto es muy útil en programas que queremos que se comporten de forma diferente según una entrada de datos y para los que se espera una ejecución desatendida, es decir no se espera interacción por parte del usuario durante su ejecución. Por ejemplo, un programa se ejecuta cada noche para hacer copias de seguridad de una serie de archivos. En este caso, no tiene sentido que el programa esté siempre en marcha comprobando si ha llegado la hora de hacer la copia. Es más lógico programar su ejecución periódica con alguna herramienta ofrecida por el sistema operativo. Además, es evidente que no es factible que alguien esté pendiente del teclado cada vez que se ejecuta, y por tanto para que funcione correctamente debe disponer de cierta información que tampoco puede estar escrita en el código fuente, ya que el valor dependerá como se configure: la lista de directorios que se quiere copiar, dónde se guarda la copia, etc. Así pues, es útil disponer de una forma simple de introducción de datos a un programa sin que tenga que ser siempre por el teclado.

La lista de palabras que conforman los argumentos de un programa en Java se puede obtener directamente a partir de la variable especial que se define en la declaración del método principal: *String[] args*. Se trata de un array de cadenas de texto, en el que estas ocupan las posiciones en el mismo orden que se han escrito los argumentos en ejecutar el programa. Para saber con cuantos argumentos se ha ejecutado el programa hay que usar el atributo *length* asociado al *array*. Si no se ha pasado ningún argumento, su valor será 0.

En el siguiente ejemplo se muestran los argumentos recibidos por consola:

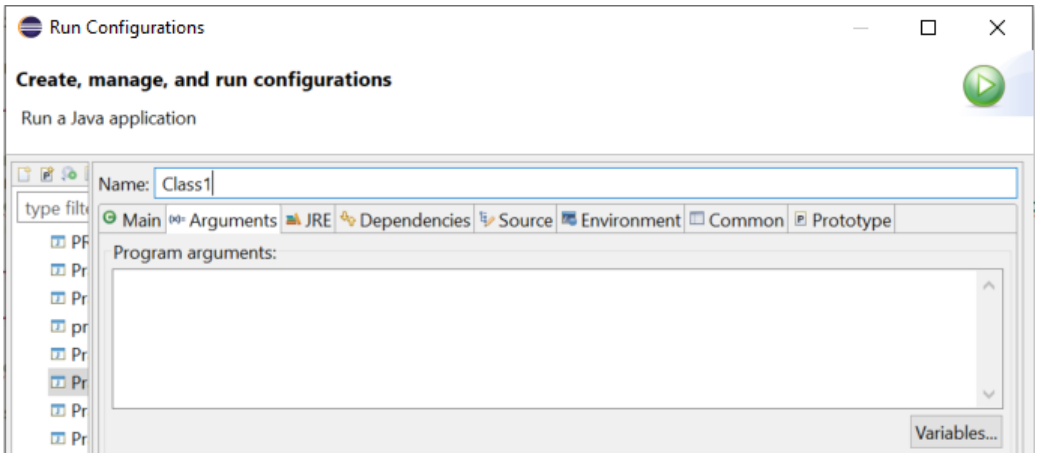
```
public class Ejemplo {

	public static void main(String[] args) {
		for (int i = 0; i < args.length; i++) {
			System.out.println(args[i]);
		}
	}
}
```

La forma de pasar argumentos a la ejecución de un programa Java es introducir un conjunto de palabras separadas por espacios a continuación del nombre del archivo ejecutable. Por ejemplo:

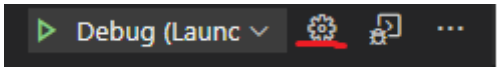
```
java Ejemplo Hola mundo
```

En el ejemplo: *args[0]=”Hola” y args[1]=”mundo”.*

- En ***Eclipse*** podemos ir a *Run → Run Configurations*, vamos a la pestaña *Arguments* y en *Program Arguments* y allí escribimos los argumentos que queremos pasar separados por espacios.



- Cuando estamos trabajando con ***Visual Studio Code*** podemos ir al menú lateral Run, Ctrl + Mayus + D, clickamos en el enlace "*create a launch.json file*", abrimos el fichero json creado y vamos al bloque de la clase a la que queremos pasar los argumentos (parámetros) en incluimos el valor "args" en el json con los argumentos que queremos pasar separados por espacios.




```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "java",
            "name": "Debug (Launch) - Current File",
            "request": "launch",
            "mainClass": "${file}"
        },
        {
            "type": "java",
            "name": "Debug (Launch)-App<PruebasJava_9fc46284>",
            "request": "launch",
            "mainClass": "App",
            "projectName": "PruebasJava_9fc46284",
            "args": "Parametro1 Parametro2 \"Nuevo Parametro\""
        }
    ]
}
```

4 Clases Wrapper

Las clases *wrapper* son objetos que encapsulan a los tipos de datos primitivos en Java. Están incluidas en el paquete *java.lang* por lo que no es necesario importarlas de forma manual. La siguiente tabla muestra los datos primitivos sus clases *wrapper* equivalentes:

Tipo de Dato Primitivo	Clase Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Podemos crear objetos de tipo wrapper simplemente declarando las variables con el tipo de dato de la clase y asignándoles un valor literal del tipo de dato correspondiente

```
Integer myInt = 5;
Double myDouble = 5.99;
Character myChar = 'A';
System.out.println(myInt);
System.out.println(myDouble);
System.out.println(myChar);
```

A la conversión automática de tipos primitivos en objetos wrapper se le conoce como ***autoboxing***. Al proceso contrario, la conversión automática de un objeto wrapper en un tipo de dato primitivo, se le conoce como ***unboxing***. Veamos un ejemplo:

```
Character myChar = 'a';
Char c = myChar;
```

Los siguientes métodos permiten obtener de los objetos tipo *wrapper* los valores primitivos correspondientes: *intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue()*. Así tendríamos:

```
Integer myInt = 5;
Double myDouble = 5.99;
Character myChar = 'A';
System.out.println(myInt.intValue());
System.out.println(myDouble.doubleValue());
System.out.println(myChar.charValue());
```

Un método muy útil es ***toString()***, que permite convertir los objetos *wrapper* en cadenas de caracteres.

```
Integer myInt = 100;
String myString = myInt.toString();
System.out.println(myString.length());
```

El método ***valueOf(String s)*** de los objetos wrapper devuelve un objeto wrapper del tipo correspondiente a partir de una cadena de texto.

Los siguientes métodos permiten obtener los valores primitivos correspondientes a partir del valor presente en una cadena de caracteres: *parseInt, parseByte, parseShort, parseLong, parseFloat, parseDouble, parseBoolean.*

```
float a = Float.parseFloat("1.25");
Float objB = Float.valueOf("3.75");
float b = objB.floatValue();
System.out.println(a+b)
```

Vemos que la conversión se realiza sin aplicar la configuración local del ordenador para el separador decimal.

Para realizar conversiones de texto a números utilizando la configuración regional del ordenador usaremos las clases NumberFormat y Number, por ahora no abordaremos este problema.

Además, las clases wrapper ofrecen una serie de atributos y métodos que facilitan en uso de los datos primitivos correspondientes. Para más información podemos consultar la ayuda de Java: Byte, Short, Integer, Long, Float, Double, Boolean, Character.

5 Introducción a las listas. Ordenación de listas

Ya hemos visto que las posibilidades que nos ofrecen los arrays cuando necesitamos procesar la información de un conjunto de elementos. Aunque hemos visto cuál es su principal limitación; los arrays tienen un tamaño fijo y no podemos modificar este tamaño una vez creado, tenemos que crear un nuevo array si queremos eliminar o añadir elementos o gestionar manualmente arrays parcialmente ocupados.

Estas opciones resultan no ser nada prácticas cuando nos encontramos ante la situación de no saber de antemano cuántos elementos va a tener un conjunto o si vamos a necesitar añadir, insertar o borrar elementos de nuestro conjunto.

Los lenguajes de programación ofrecen alternativas de gestión dinámica de memoria para trabajar con colecciones de datos que pueden cambiar de tamaño y que ofrecen determinados comportamientos para el tratamiento del conjunto. No abordaremos estas cuestiones hasta más avanzado el curso, pero sí vamos a hacer una introducción a la clase *ArrayList* que nos permitirá un uso básico de este tipo de objetos.

La case ***ArrayList*** es un array redimensionable presente en el paquete *java.util*.

A diferencia de un array, cuyo tamaño no se puede modificar, los objetos *ArrayList* permiten añadir y eliminar elementos siempre que necesitemos. Llamamos *listas* de manera habitual y general a los objetos de tipo ArrayList.

Veamos el siguiente ejemplo de uso:

```
import java.util.ArrayList;

public class Cities {

	public static void main(String[] args) {
		// declaramos e inicializamos la lista
		// debemos indicar el tipo de datos que
		// contiene la lista
		ArrayList<String> cities = new ArrayList<String>();
		// añadimos elementos a la colección
		cities.add("Alcañiz");
		cities.add("Calamocha");
		cities.add("Montalbán");
		cities.add("Alcorisa");
		cities.add("Sarrión");
		cities.add("Benasque");
		// borramos un elemento
		cities.remove(5);
		// insertamos un elemento en la posición que queramos
		cities.add(0, "Teruel");

		// recorremos los elementos de la lista
		int length = cities.size();
		for (int i = 0; i < length; i++) {
			// accedemos a los elementos indicando su posición
			System.out.println(cities.get(i));
		}

		System.out.println();
		System.out.println("Clear()");
		// limpiamos la lista
		cities.clear();
		// comprobamos el tamaño de la lista
		if (cities.size() == 0) {
			System.out.println("No hay elementos en el ArrayList");
		}
	}
}
```

Vemos que al crear el objeto *ArrayList* debemos indicar el tipo de datos que contendrá el array entre los signos **< >**.

El método ***add*** permite añadir elementos a la lista.

Para obtener un elemento de la lista llamaremos al método ***get***.

El método ***remove*** permite eliminar un elemento de la colección. Llamaremos al método ***clear*** para borrar todos los elementos de una lista.

El método ***size*** permite obtener el número de elementos que contiene la lista.

No podemos crear objetos *ArrayList* de tipos de datos primitivos, pero sí utilizando objetos las clases *wrapper*. Veamos un ejemplo:

```
import java.util.ArrayList;
import java.util.Collections;

public class Numbers {

	public static void main(String[] args) {
		// declaramos una lista de números enteros
		ArrayList<Integer> numbers = new ArrayList<Integer>();

		// generamos un número aleatorio entre 15 y 20
		int length = (int) (Math.random() * 6 + 15);

		// generamos enteros aleatorios en el rango [0,100)
		// mostramos la lista generada
		for (int i = 0; i < length; i++) {
			int number = (int) (Math.random() * 100);
			numbers.add(number);
			System.out.print(number + " ");
		}
		System.out.println();

		System.out.println();
		// chequeamos la existencia de los números 1, 5 y 10
		if (numbers.indexOf(1) > -1) {
			System.out.println("El 1 está presente en la lista");
		}
		if (numbers.indexOf(5) > -1) {
			System.out.println("El 5 está presente en la lista");
		}
		if (numbers.indexOf(10) > -1) {
			System.out.println("El 10 está presente en la lista");
		}

		// ordenamos la lista
		Collections.sort(numbers);
		System.out.println();
		System.out.println("Lista ordenada");
		length = numbers.size();
		for (int i = 0; i < length; i++) {
			System.out.print(numbers.get(i) + " ");
		}
		System.out.println();
	}
}
```

Vemos en el ejemplo que el método ***sort*** de la clase Collections permite ordenar las listas.

Podemos buscar elementos en la lista con los métodos ***indexOf*** y ***lastIndexOf***.

El método toArray permite obtener un array a partir de un ArrayList. El método asList de la clase Arrays permite convertir un array en una lista. Veámoslo en el siguiente ejemplo:

```
import java.util.ArrayList;
import java.util.Arrays;

public class Rivers {

	public static void main(String[] args) {
		// declaramos un array strings
		String[] a = { "Turia", "Ebro", "Jiloca", "Alfambra" };
		// transformamos el array en una lista
		// pasamos como parámetro el resultado
		// de Arrays.asList
		ArrayList<String> lst =
			new ArrayList<String>(Arrays.asList(a));
		lst.add("Tajo");
		lst.add("Duero");
		lst.add("Júcar");
		// obtenemos un array a partir de la lista
		// pasamos como parámetro un array del tipo
		// y tamaño que necesitamos
		a = (String[]) lst.toArray(new String[lst.size()]);
		// recorremos el array y mostramos
		for (int i = 0; i < a.length; i++) {
			System.out.println(a[i]);
		}

	}

}
```

6 Tipos de dato de fecha y hora

6.1 Clase LocalDate

LocalDate es un objeto inmutable de fecha-hora que representa una fecha. Contiene una fecha sin hora (ni zona horaria) siguiendo el sistema de calendario ISO-8601. El formato ISO de fecha se expresa normalmente como “yyyy-MM-dd”.

Podemos obtener una instancia de la fecha actual a partir de la fecha del sistema de la siguiente manera:

```
LocalDate localDate = LocalDate.now();
```

Para obtener un objeto que representa una fecha en concreto podemos hacer uso de los métodos estáticos *of* y *parse*.

```
LocalDate date1 = LocalDate.of(1989, Month.NOVEMBER, 8);
LocalDate date2 = LocalDate.parse("2011-04-25");
```

La clase *LocalDate* dispone de una serie de métodos que nos permiten obtener información diversa a partir de una fecha:

```
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;
import java.time.temporal.ChronoUnit;

public class TestDate {

	public static void main(String[] args) {

		LocalDate localDate = LocalDate.now();
		// calcular la fecha sumando 3 días
		// a partir de hoy
		System.out.println(localDate.plusDays(3));
		// calcular la fecha de hace 2 meses
		// desde hoy
		System.out.println(localDate.minusMonths(2));
		// mostrar el día de la semana
		System.out.println(localDate.getDayOfWeek());
		// mostrar el día del mes
		System.out.println(localDate.getDayOfMonth());
		// obtener el mes de una fecha
		System.out.println(localDate.getMonth());
		System.out.println(localDate.getMonthValue());
		// obtener el año
		System.out.println(localDate.getYear());

		// comparar 2 fechas
		LocalDate date1 = LocalDate.of(1989, Month.NOVEMBER, 8);
		System.out.println(localDate + " es antes de " + date1
			+ ": " + localDate.isBefore(date1));
		System.out.println(localDate + " es después de " + date1
			+ ": " + localDate.isAfter(date1));

		// Calcular el period transcurrido entre 2 fechas
		// años
		System.out.println(Period.between(date1, localDate).getYears());
		// días del mes
		System.out.println(
			Period.between(
				LocalDate.parse("2019-09-13")
				, LocalDate.parse("2019-12-20")
			).getDays());
		// días del año
		System.out.println(
			ChronoUnit.DAYS.between(LocalDate.parse("2019-09-13")
				, LocalDate.parse("2019-12-20")));

	}

}
```

En el ejemplo anterior hemos usado la clase Period y ChronoUnit para calcular diferencias de tiempo entre dos fechas.

6.2 Clase LocalTime

La clase LocalTime representa tiempo (hora) sin fecha. Trabajaremos con la clase *LocalTime* de manera similar a como trabajamos con *LocalDate*.

Podemos obtener la hora del sistema con el método *now*:

```
LocalTime now = LocalTime.now();
System.out.println(now);
```

Usando los métodos *of* y *period* obtenemos un instante determinado:

```
LocalTime t1 = LocalTime.of(15, 25, 35);
System.out.println(t1);

LocalTime t2 = LocalTime.parse("06:30:00");
System.out.println(t2);
```

Disponemos de las constantes siguientes constantes en *LocalTime*

MAX	El máximo LocalTime soportado, '23:59:59.999999999'
MIDNIGHT	Medianoche, al comienzo del día, '00:00'
MIN	El máximo LocalTime soportado, '00:00'
NOON	Mediodía, '12:00'

En el siguiente ejemplo vemos algunas de las informaciones que podemos obtener de la clase *LocalTime*

```
import java.time.LocalTime;

public class TestTime {

	public static void main(String[] args) {
		// obtenemos objeto con las 9 y media
		LocalTime t1 = LocalTime.of(9, 30);
		System.out.println(t1);
		// añadimos 3 horas
		System.out.println(t1.plusHours(3));
		// restamos 30 minutos
		System.out.println(t1.minusMinutes(30));

		// comparamos dos tiempos
		LocalTime t2 = LocalTime.parse("06:30");
		System.out.println(t1 + " es antes que " + t2 + ": " + t1.isBefore(t2));

		// obtenemos la hora
		System.out.println(t2.getHour());
		// obtenemos los minutos
		System.out.println(t2.getMinute());

	}

}
```

6.3 Clase LocalDateTime

La clase LocalDateTime se usa para representar la combinación de fecha y hora. Se usa de manera similar a como usamos *LocalDate* y *LocalTime*. Veamos un ejemplo:

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class TestDateTime {

	public static void main(String[] args) {
		// obtenemos fecha y hora del sistema
		LocalDateTime now = LocalDateTime.now();
		System.out.println(now);

		// obtenemos una fecha hora concreta
		// a partir de sus elementos
		LocalDateTime newYear = LocalDateTime.of(2020, 01, 01, 00, 00, 00);
		System.out.println(newYear);
		// a partir de una fecha y una hora
		LocalDate lunes = LocalDate.parse("2019-11-25");
		LocalTime clase = LocalTime.of(18, 40);
		LocalDateTime claseLunes = LocalDateTime.of(lunes, clase);
		System.out.println(claseLunes);
		// calculamos periodos de tiempo
		System.out.println("Faltan " + ChronoUnit.HOURS.between(now, claseLunes)
			+ " horas para la clase del lunes");
		System.out.println("Faltan " + ChronoUnit.DAYS.between(now, newYear)
			+ " horas para año nuevo");
		// a partir de una cadena de texto
		LocalDateTime holidays = LocalDateTime.parse("2019-12-19T21:20");
		System.out.println(holidays);
		// obtenemos el dia de la semana
		System.out.println("Empezamos las vacaciones en "
			+ holidays.getDayOfWeek());

	}

}
```