

{ LUIS JOSÉ SÁNCHEZ }

APRENDE
JAVA
CON EJERCICIOS



[INCLUYE MÁS DE 200 EJERCICIOS RESUELTOS]

Aprende Java con Ejercicios

Un manual para aprender a programar en Java desde cero, con multitud de ejemplos y más de 200 ejercicios resueltos

Luis José Sánchez González

Este libro está a la venta en <http://leanpub.com/aprendejava>

Esta versión se publicó en 2015-09-14



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2014 - 2015 Luis José Sánchez González

Índice general

Sobre este libro	i
Sobre el autor	ii
Centros en los que se usa este libro	iii
Introducción	iv
Instalación y configuración del entorno de programación Java	v
1. ¡Hola mundo! - Salida de datos por pantalla	1
1.1 ¡Hola mundo! - Mi primer programa	1
1.2 Coloreado de texto	4
1.3 Ejercicios	6
2. Variables	8
2.1 Definición y tipos de variables	8
2.1.1 Enteros (int y long)	9
2.1.2 Números decimales (double y float)	9
2.1.3 Cadenas de caracteres (String)	10
2.2 Operadores aritméticos	12
2.3 Asignación de valores a variables	12
2.4 Conversión de tipos (<i>casting</i>)	13
2.5 Ejercicios	15
3. Lectura de datos desde teclado	16
3.1 Lectura de texto	16
3.2 Lectura de números	17
3.3 La clase Scanner	18
3.4 Ejercicios	22
4. Sentencia condicional (if y switch)	24
4.1 Sentencia if	24
4.2 Operadores de comparación	27
4.3 Operadores lógicos	28

ÍNDICE GENERAL

4.4	Sentencia switch (selección múltiple)	30
4.5	Ejercicios	34
5.	Bucles	37
5.1	Bucle for	37
5.2	Bucle while	38
5.3	Bucle do-while	39
5.4	Ejercicios	42
6.	Números aleatorios	47
6.1	Generación de números aleatorios con y sin decimales	47
6.2	Generación de palabras de forma aleatoria de un conjunto dado	51
6.3	Ejercicios	53
7.	Arrays	55
7.1	Arrays de una dimensión	55
7.2	Arrays bidimensionales	59
7.3	Recorrer arrays con for al estilo foreach	63
7.4	Ejercicios	65
7.4.1	Arrays de una dimensión	65
7.4.2	Arrays bidimensionales	68
8.	Funciones	71
8.1	Implementando funciones para reutilizar código	71
8.2	Comentarios de funciones	73
8.3	Creación de bibliotecas de rutinas mediante paquetes	74
8.4	Ámbito de las variables	78
8.5	Paso de parámetros por valor y por referencia	78
8.6	Ejercicios	83
9.	Programación orientada a objetos	86
9.1	Clases y objetos	86
9.2	Encapsulamiento y ocultación	87
9.3	Métodos	88
9.3.1	Creí haber visto un lindo gatito	88
9.3.2	Métodos <i>getter</i> y <i>setter</i>	92
9.3.3	Método <i>toString</i>	96
9.4	Ámbito/visibilidad de los elementos de una clase - <i>public</i> , <i>protected</i> y <i>private</i>	99
9.5	Herencia	100
9.5.1	Sobrecarga de métodos	104
9.6	Atributos y métodos de clase (<i>static</i>)	106
9.7	Interfaces	108
9.8	Arrays de objetos	114
9.9	Ejercicios	119

ÍNDICE GENERAL

9.9.1	Conceptos de POO	119
9.9.2	POO en Java	120
9.9.3	Arrays de objetos	123
10.	Colecciones y diccionarios	125
10.1	Colecciones: la clase ArrayList	125
10.1.1	Principales métodos de ArrayList	125
10.1.2	Definición de un ArrayList e inserción, borrado y modificación de sus elementos	127
10.1.3	ArrayList de objetos	133
10.1.4	Ordenación de un ArrayList	134
10.2	Diccionarios: la clase HashMap	138
10.2.1	Principales métodos de HashMap	138
10.2.2	Definición de un HasMap e inserción, borrado y modificación de entradas . .	139
10.3	Ejercicios	144
11.	Ficheros de texto y paso de parámetros por línea de comandos	147
11.1	Lectura de un fichero de texto	148
11.2	Escritura sobre un fichero de texto	151
11.3	Lectura y escritura combinadas	152
11.4	Otras operaciones sobre ficheros	154
11.5	Paso de argumentos por línea de comandos	155
11.6	Combinación de ficheros y paso de argumentos	158
11.7	Procesamiento de archivos de texto	159
11.8	Ejercicios	163
12.	Aplicaciones web en Java (JSP)	165
12.1	Hola Mundo en JSP	165
12.2	Mezclando Java con HTML	167
12.3	Recogida de datos en JSP	171
12.4	POO en JSP	175
12.5	Ejercicios	180
13.	Acceso a bases de datos	183
13.1	Socios de un club de baloncesto	183
13.2	Preparación del proyecto de ejemplo	186
13.2.1	Activar la conexión a MySQL	186
13.2.2	Incluir la librería MySQL JDBC	188
13.3	Listado de socios	188
13.4	Alta	191
13.5	Borrado	193
13.6	CRUD completo con Bootstrap	196
13.7	Ejercicios	198

ÍNDICE GENERAL

Apéndice A. Ejercicios de ampliación	204
-------------------------------------------------------	------------

Sobre este libro

“Aprende Java con Ejercicios” es un manual práctico para aprender a programar en Java desde cero.

No es necesario tener conocimientos previos de programación. La dificultad del libro es gradual, empieza con conceptos muy básicos y ejercicios muy sencillos y va aumentando en complejidad y dificultad a medida que avanzan los capítulos.

La práctica hace al maestro. Como de verdad se aprende a programar es programando desde el principio y esa es la filosofía que sigue este libro. En cada capítulo se explican una serie de conceptos de programación y se ilustran con ejemplos. Al final de cada uno de los capítulos se plantean ejercicios de diferente dificultad.

Este libro contiene más de 200 ejercicios. Tanto las soluciones a los ejercicios como los ejemplos están disponibles en GitHub: <https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios> y tanto los unos como los otros siguen los [estándares de programación de Google para el código fuente escrito en el lenguaje de programación Java](#)¹.

“Aprende Java con Ejercicios” es un libro hecho casi a medida de la asignatura *“Programación”* que forma parte del currículo del primer curso de los ciclos formativos DAW (Desarrollo de Aplicaciones Web) y DAM (Desarrollo de Aplicaciones Multiplataforma) pero igualmente puede ser utilizado por estudiantes de Ingeniería Informática, Ingeniería de Telecomunicaciones o Ciencias Matemáticas en la asignatura de *“Programación”* de los primeros cursos.

Si tienes alguna duda o quieres hacer algún comentario en relación a este libro, tienes a tu disposición el grupo de Google <https://groups.google.com/d/forum/aprende-java-con-ejercicios>. Cualquier sugerencia para mejorar este manual será bienvenida.

¹<http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

Sobre el autor

Luis José Sánchez González es Ingeniero Técnico en Informática de Gestión por la Universidad de Málaga (España) y funcionario de carrera de la Junta de Andalucía desde 1998. En su trabajo de profesor de Informática combina sus dos pasiones: la enseñanza y la programación.

En el momento de publicar este libro, es profesor del **I.E.S. Campanillas (Málaga)** e imparte clases en el **Ciclo Superior de Desarrollo de Aplicaciones Web**.

Puedes ponerte en contacto con el autor mediante la dirección de correo electrónico luisjoseprofe@gmail.com o mediante LinkedIn (<https://es.linkedin.com/pub/luis-josé-sánchez/86/b08/34>).

Centros en los que se usa este libro

- IES Campanillas, Málaga (España)
- IES Polígono Sur, Sevilla (España)
- IES Murgi, El Ejido - Almería (España)

Si eres profesor y utilizas este libro para enseñar Java, pídele al autor que añada en esta sección el nombre de tu centro educativo o empresa, escribiendo un correo a la dirección luisjoseprofe@gmail.com indicando la localidad y el país del centro.

Introducción

Java es un lenguaje de programación; es de hecho, desde hace más de 10 años, el lenguaje de programación más utilizado en el mundo según el [índice PYPL](http://pypl.github.io/PYPL.html)²(Popularity of Programming Language index).

Los ordenadores no entienden - por ahora - el español, ni el inglés, ni ningún otro idioma natural. De una forma muy simplificada podríamos decir que un lenguaje de programación es un idioma que entiende el ordenador³. Cualquier aplicación - procesador de textos, navegador, programa de retoque fotográfico, etc. - está compuesta de una serie de instrucciones convenientemente empaquetadas en ficheros que le dicen al ordenador de una manera muy precisa qué tiene que hacer en cada momento.

Java es un lenguaje de programación estructurado y, como tal, hace uso de variables, sentencias condicionales, bucles, funciones... Java es también un lenguaje de programación orientado a objetos y, por consiguiente, permite definir clases con sus métodos correspondientes y crear instancias de esas clases. Java no es un lenguaje de marcas como HTML o XML aunque puede interactuar muy bien con ellos⁴.

Las aplicaciones Java se suelen compilar a *bytecode*, que es independiente del sistema operativo, no a binario que sí depende del sistema operativo. De esta forma, el *bytecode* generado al compilar un programa escrito en Java debería funcionar en cualquier sistema operativo que tenga instalada una máquina virtual de java (JVM).

Cualquier editor simple como **Nano** o **GEdit** es suficiente para escribir código en Java aunque se recomiendan IDEs⁵ como **NetBeans** o **Eclipse** ya que tienen algunas características que facilitan mucho la programación como el chequeo de errores mientras se escribe, el autocompletado de nombres de variables y funciones y mucho más.

²<http://pypl.github.io/PYPL.html>

³En realidad un ordenador no entiende directamente los lenguajes de programación como Java, C, Ruby, etc. Mediante una herramienta que se llama compilador o traductor según el caso, lo que hay escrito en cualquiera de estos lenguajes se convierte en código máquina - secuencias de unos y ceros - que es el único lenguaje que de verdad entiende el ordenador.

⁴El lector deberá tener unos conocimientos básicos de HTML para entender bien y sacar provecho del capítulo relativo a JSP de este mismo libro. Una web excelente para aprender HTML es <http://w3schools.com>

⁵IDE: Integrated Development Environment (Entorno de Desarrollo Integrado)

Instalación y configuración del entorno de programación Java

Todos los ejemplos que contiene este libro así como las soluciones a los ejercicios se han escrito sobre **Linux**, en concreto sobre **Ubuntu** (<http://www.ubuntu.com/>). No obstante, todo el código debería compilar y funcionar sin ningún problema en cualquier otra plataforma.

El software necesario para seguir este manual es el siguiente⁶:

OpenJDK

El **Open Java Development Kit** (OpenJDK) contiene una serie de componentes que permiten desarrollar y ejecutar aplicaciones escritas en Java. Incluye la máquina virtual de Java (JVM) que permite ejecutar *bytecode*. Como mencionamos en el apartado [Introducción](#), el *bytecode* es el ejecutable en Java. El **OpenJDK** también incluye el compilador `javac` que permite compilar el código fuente para generar el *bytecode*. En **Ubuntu**, basta ejecutar una línea en la ventana de terminal para instalar el **OpenJDK**:

```
$ sudo apt-get install openjdk-8-jdk
```

Para los sistemas operativos **Mac OS X** y **Windows**, se encuentran disponibles para su descarga los archivos de instalación del **OpenJDK** en la página [Unofficial OpenJDK installers for Windows, Linux and Mac OS X](#)⁷. Otra opción consiste en utilizar el **JDK** oficial de Oracle, que se puede descargar en el apartado [Java SE Development Kit 8 - Downloads](#)⁸ de la web de esta misma empresa. No vamos a explicar las diferencias entre el **OpenJDK** y el **JDK** ya que se escapa al ámbito de este libro. A efectos prácticos, todos los ejemplos contenidos en este manual funcionan tanto con el **OpenJDK** como con el **JDK** y los ejercicios se pueden realizar sin problema sea cual sea el kit de desarrollo instalado.

Editor de textos simple

Para seguir este manual recomendamos utilizar al principio un editor de textos sencillo para probar los primeros programas y realizar los primeros ejercicios, compilando en línea de comandos. De esta manera, el lector se va familiarizando con el proceso de edición de código, compilación y ejecución y va entendiendo lo que sucede en cada paso. En **Ubuntu** viene instalado por defecto el editor **GEdit**, el sistema operativo **Mac OS X** viene con **TextEdit** y en **Windows** podemos usar el programa **Bloc de notas**.

⁶En la página <http://helloworldcollection.de/> se muestra el programa *Hola mundo* escrito en más de 400 lenguajes de programación.

⁷<https://github.com/alexkasko/openjdk-unofficial-builds>

⁸<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Geany

Una vez que el lector ha superado la barrera de escribir sus primeros programas, recomendamos utilizar el entorno de programación **Geany** (por ejemplo a partir del [capítulo 2](#)). Se trata de un IDE muy ligero que permite realizar aplicaciones sencillas con rapidez. Mediante **Geany** se puede escribir el código, chequear si contiene errores, compilar y ejecutar el *bytecode* generado. Para programas pequeños - por ej. calcular la media de varios números - es recomendable utilizar **Geany** en lugar de un IDE más complejo como **Netbeans**. Mientras con **Geany** da tiempo a escribir el código, compilar y ejecutar, **Netbeans** todavía estaría arrancando.

Instalación de **Geany** (y los *plugins* adicionales) en **Ubuntu**:

```
$ sudo apt-get install geany
$ sudo apt-get install geany-plugins
```

En la [sección de descargas de la página oficial de Geany](#)⁹ se encuentran los ficheros de instalación de este programa tanto para **Mac OS X** como para **Windows**.

Una vez instalado **Geany** es conveniente realizar algunos ajustes en la configuración. Presiona **Control + Alt + P** para abrir la ventana de preferencias. Haz clic en la pestaña **Editor** y luego en la pestaña **Sangría**. Establece las opciones tal y como se indican a continuación.

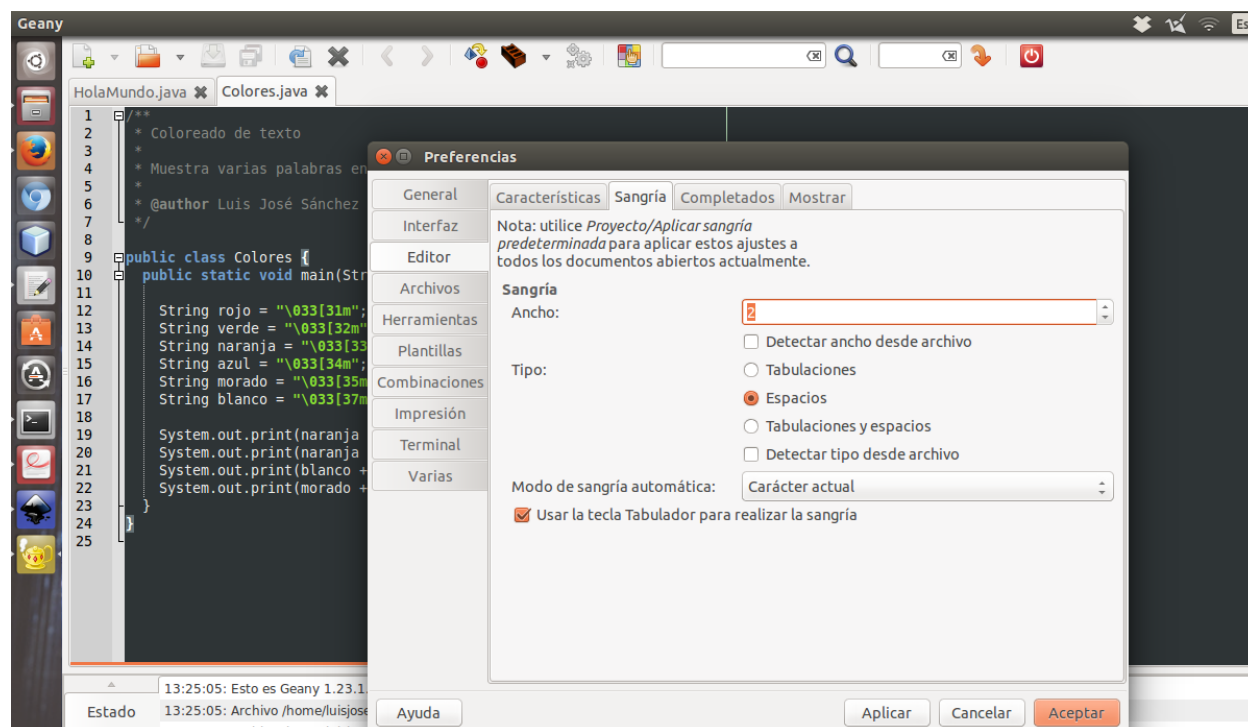


Figura 0.1: Configuración de Geany (sangría)

⁹<http://www.geany.org/Download/Releases>

También es recomendable marcar las casillas **Mostrar guías de sangría** y **Mostrar números de línea**. Estas opciones se activan en la pestaña **Mostrar**.

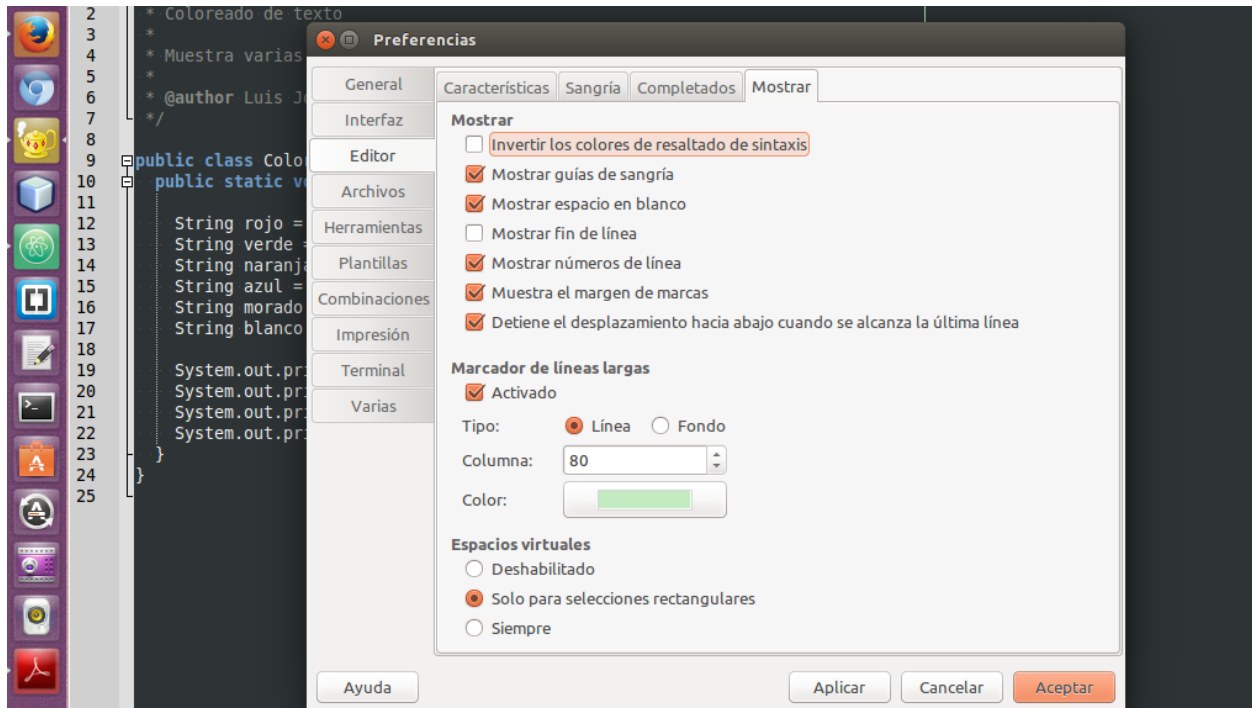


Figura 0.2: Configuración de Geany (guías de sangría y números de línea)

Netbeans

Cuando las aplicaciones a desarrollar son más complejas y, sobre todo, cuando éstas constan de varios ficheros, se recomienda utilizar **Netbeans** de la empresa **Oracle**. En el [capítulo 9](#), el lector ya podría sacarle mucho partido al uso de este IDE. Se trata de un entorno integrado muy potente y muy usado a nivel profesional que incluye el autocompletado de sintaxis y permite una depuración avanzada paso a paso. **Netbeans** se puede descargar de forma gratuita y para cualquier plataforma desde <https://netbeans.org/downloads/>.

Después de la instalación de **Netbeans**, procedemos a configurar el editor. Para ello seleccionamos **Herramientas** → **Opciones** → **Editor**. A continuación seleccionamos la pestaña **Formato** y en el menú desplegable correspondiente al lenguaje de programación seleccionamos **Java**. La configuración debería quedar como se indica en la imagen.

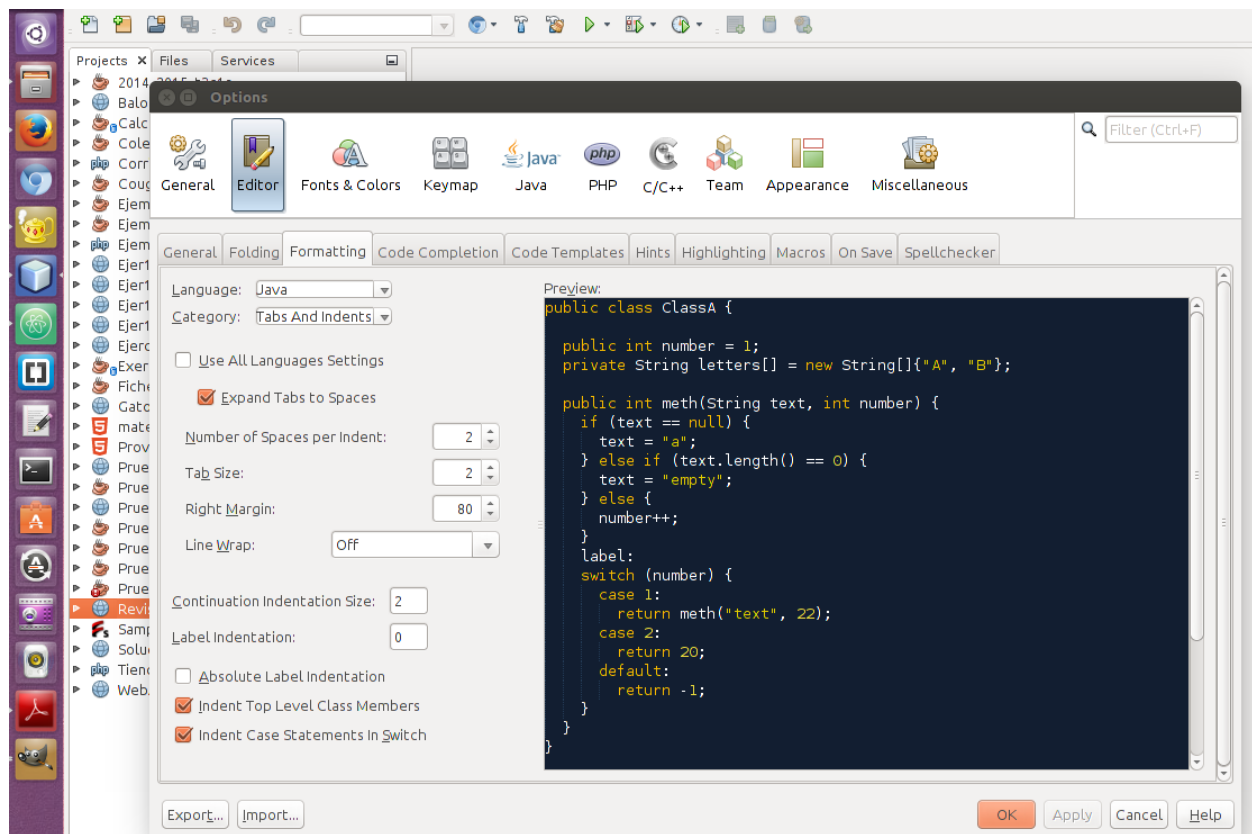


Figura 0.3: Configuración de Netbeans



Es muy importante que los editores/IDEs que se utilicen para escribir código en Java se configuren adecuadamente. Hemos visto cómo configurar Geany y Netbeans, pero si el lector decide utilizar otras herramientas deberá configurarlas igualmente. En definitiva, la configuración/preferencias de las herramientas que se utilicen deben establecerse de tal forma que se cumplan estos requisitos:

- La tecla TAB debe insertar espacios, no debe insertar el carácter de tabulación.
- La indentación debe ser de 2 caracteres (por defecto suele estar a 4).
- La codificación de caracteres debe ser UTF-8.

1. ¡Hola mundo! - Salida de datos por pantalla

1.1 ¡Hola mundo! - Mi primer programa

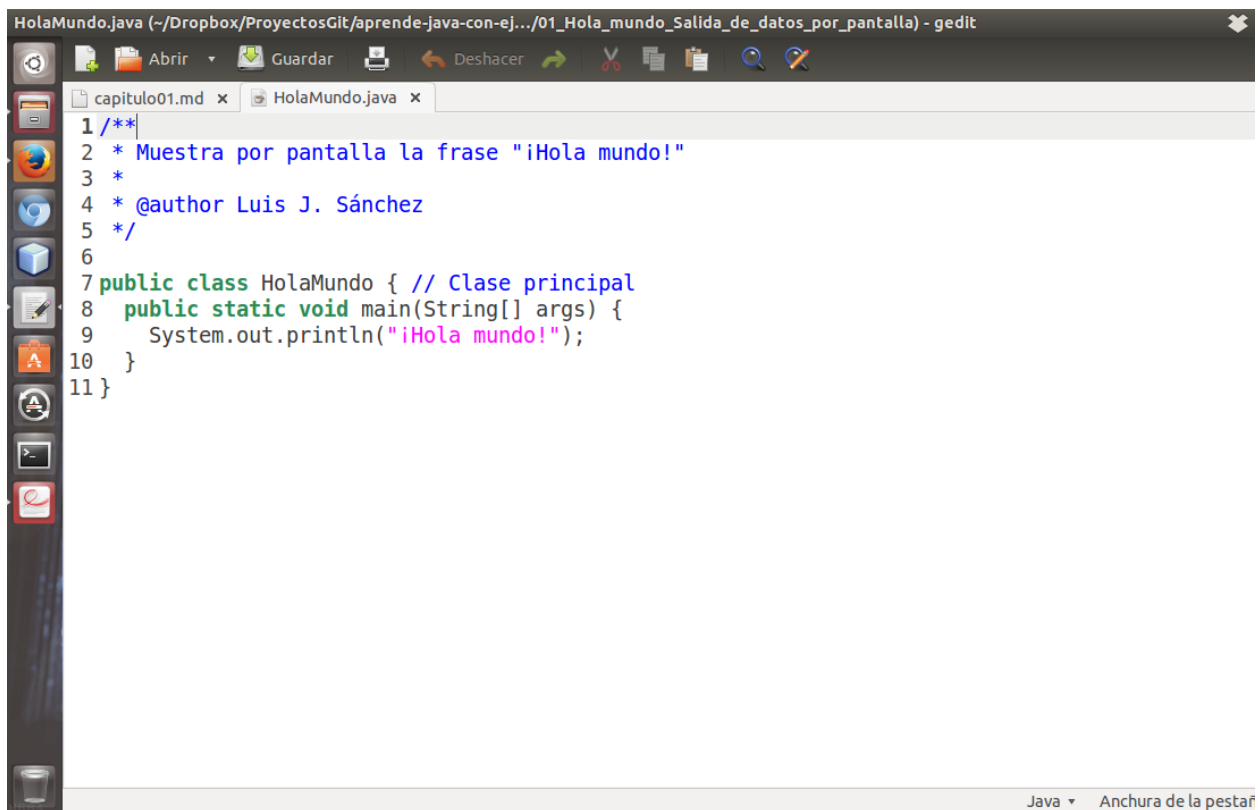
El primer programa que aprende a hacer cualquier aspirante a programador es un **Hola mundo** ¹. Es seguramente el programa más sencillo que se puede escribir. Se trata de un programa muestra el mensaje “Hola mundo” por pantalla. Abre el programa **GEdit** que, como hemos comentado en la sección [Instalación y configuración del entorno de programación Java](#), viene instalado por defecto en Ubuntu y teclea el siguiente código:

```
/**
 * Muestra por pantalla la frase "¡Hola mundo!"
 *
 * @author Luis J. Sánchez
 */

public class HolaMundo { // Clase principal
    public static void main(String[] args) {
        System.out.println("¡Hola mundo!");
    }
}
```

Verás que, en principio, el texto del código no aparece en colores. El coloreado de sintaxis es muy útil como comprobarás más adelante, sobre todo para detectar errores. Graba el programa como `HolaMundo.java`. Al grabar con la extensión `.java`, ahora el programa **GEdit** sí sabe que se trata de un programa escrito en Java y reconoce la sintaxis de este lenguaje de forma que puede colorear los distintos elementos que aparecen en el código.

¹En la página <http://helloworldcollection.de/> se muestra el programa **Hola mundo** escrito en más de 400 lenguajes de programación.

Figura 1.1: Programa `Hola mundo` escrito en el editor GEdit

Fíjate que el nombre que le damos al fichero es exactamente igual que la clase principal seguido de la extensión `.java`. Abre una ventana de terminal, entra en el directorio donde se encuentra el fichero y teclea lo siguiente:

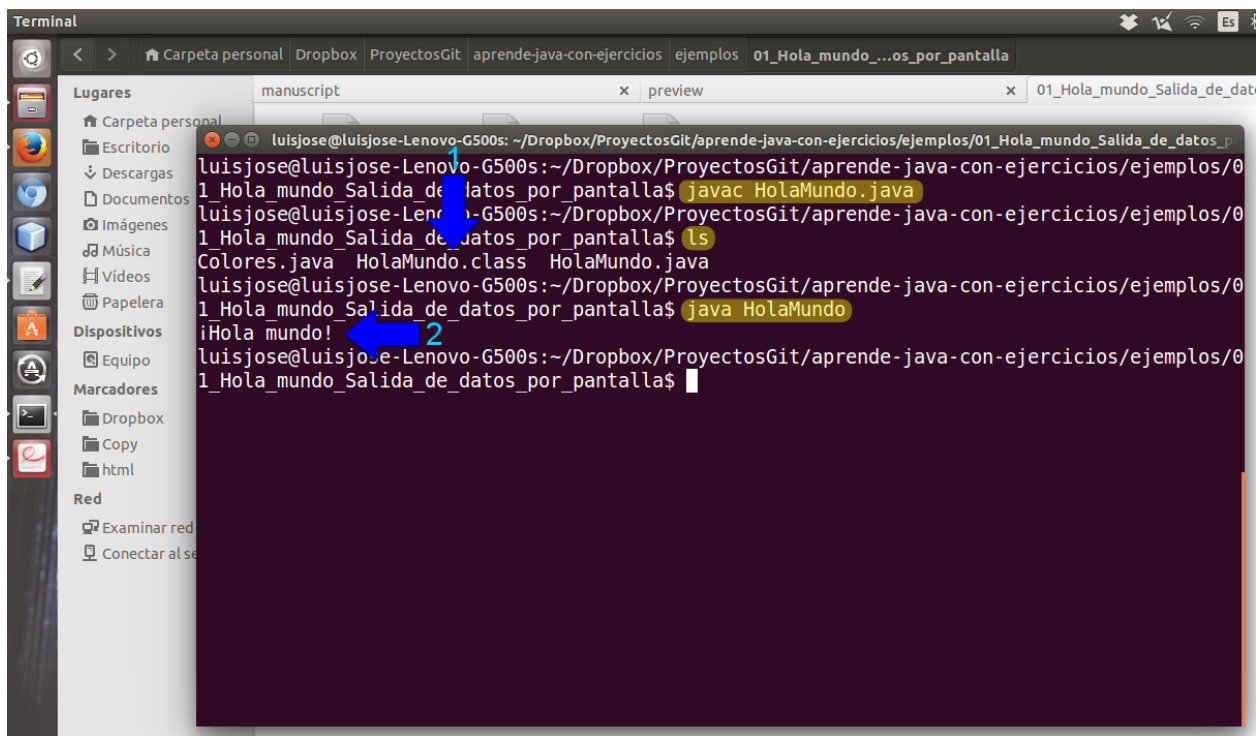
```
$ javac HolaMundo.java
```

Este comando crea `HolaMundo.class` que es el *bytecode*. Para ejecutar el programa, teclea:

```
$ java HolaMundo
```

¡Enhorabuena! Acabas de realizar tu primer programa en Java.

En la siguiente captura de pantalla puedes ver la ventana de terminal y los comandos que acabamos de describir.



```
luisjose@luisjose-Lenovo-G500s: ~/Dropbox/ProyectosGit/aprende-java-con-ejercicios/ejemplos/01_Hola_mundo_Salida_de_datos_p
luisjose@luisjose-Lenovo-G500s:~/Dropbox/ProyectosGit/aprende-java-con-ejercicios/ejemplos/01_Hola_mundo_Salida_de_datos_p$ javac HolaMundo.java
luisjose@luisjose-Lenovo-G500s:~/Dropbox/ProyectosGit/aprende-java-con-ejercicios/ejemplos/01_Hola_mundo_Salida_de_datos_p$ ls
Colores.java  HolaMundo.class  HolaMundo.java
luisjose@luisjose-Lenovo-G500s:~/Dropbox/ProyectosGit/aprende-java-con-ejercicios/ejemplos/01_Hola_mundo_Salida_de_datos_p$ java HolaMundo
¡Hola mundo!
```

Figura 1.2: `HolaMundo.class` es el *bytecode* generado (1). El resultado del programa es una línea de texto escrita en pantalla (2).

La instrucción que hemos utilizado para mostrar una frase por pantalla es `System.out.println()`, colocando la frase entre paréntesis. También se puede volcar texto por pantalla mediante `System.out.print()`. La única diferencia radica en que esta última no añade un salto de línea al final, pruébalo en `HolaMundo.java` y compruébalo por ti mismo. Si lo que quieres mostrar es una palabra o una frase, como en este ejemplo, es importante que el texto esté entrecomillado.



- El programa propiamente dicho está dentro de lo que llamamos “clase principal”.
- El fichero debe tener el mismo nombre que la clase principal seguido por la extensión `.java`
- Los comentarios de varias líneas se colocan entre `/*` y `*/`
- Los comentarios de línea se indican mediante `//`
- Para mostrar información por pantalla se utilizan `System.out.println()` y `System.out.print()`.

1.2 Coloreado de texto

Mostrar siempre texto blanco sobre fondo negro puede resultar muy aburrido. El texto que se muestra por pantalla se puede colorear; para ello es necesario insertar unas secuencias de caracteres - que indican el color con el que se quiere escribir - justo antes del propio texto.

Prueba el siguiente programa y no te preocupes si todavía no lo entiendes del todo, en el próximo capítulo se explican los diferentes tipos de datos que se pueden utilizar en Java, entre estos tipos está la cadena de caracteres o `String`.

```
/**
 * Coloreado de texto
 *
 * Muestra varias palabras en el color que corresponde.
 *
 * @author Luis José Sánchez
 */

public class Colores {
    public static void main(String[] args) {

        String rojo = "\033[31m";
        String verde = "\033[32m";
        String naranja = "\033[33m";
        String azul = "\033[34m";
        String morado = "\033[35m";
        String blanco = "\033[37m";

        System.out.print(naranja + "mandarina" + verde + " hierba");
        System.out.print(naranja + " saltamontes" + rojo + " tomate");
        System.out.print(blanco + " sábanas" + azul + " cielo");
        System.out.print(morado + " nazareno" + azul + " mar");
    }
}
```

Como puedes ver en los dos ejemplos que hemos mostrado, algunas líneas están ligeramente desplazadas hacia la derecha, es lo que se llama **sangría** o **indentación**. En programación es muy importante sangrar (indentar) bien porque da una idea de qué partes del código son las que contienen a otras. En este último ejemplo tenemos un programa que tiene unos comentarios al principio y luego la clase principal marcada por la línea `public class Colores {`. Dentro de la clase `Colores` está el `main` o bloque del programa principal que tiene sangría por estar dentro de la definición de la clase `Colores`. A su vez, dentro del `main` hay una serie de líneas de código que igualmente tienen sangría.

En este libro, tanto el código de los ejemplos como el de las soluciones a los ejercicios siguen el

estándar de Google para el código fuente escrito en el lenguaje de programación Java², por tanto, cada vez que se aplique la sangría será exactamente de dos espacios.

²<http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

1.3 Ejercicios



Ejercicio 1

Escribe un programa que muestre tu nombre por pantalla.



Ejercicio 2

Modifica el programa anterior para que además se muestre tu dirección y tu número de teléfono. Asegúrate de que los datos se muestran en líneas separadas.



Ejercicio 3

Escribe un programa que muestre por pantalla 10 palabras en inglés junto a su correspondiente traducción al castellano. Las palabras deben estar distribuidas en dos columnas y alineadas a la izquierda. Pista: Se puede insertar un tabulador mediante `\t`.



Ejercicio 4

Escribe un programa que muestre tu horario de clase. Puedes usar espacios o tabuladores para alinear el texto.



Ejercicio 5

Modifica el programa anterior añadiendo colores. Puedes mostrar cada asignatura de un color diferente.



Ejercicio 6

Escribe un programa que pinte por pantalla una pirámide rellena a base de asteriscos. La base de la pirámide debe estar formada por 9 asteriscos.



Ejercicio 7

Igual que el programa anterior, pero esta vez la pirámide estará hueca (se debe ver únicamente el contorno hecho con asteriscos).



Ejercicio 8

Igual que el programa anterior, pero esta vez la pirámide debe aparecer invertida, con el vértice hacia abajo.



Ejercicio 9

Escribe un programa que pinte por pantalla alguna escena - el campo, la habitación de una casa, un aula, etc. - o algún objeto animado o inanimado - un coche, un gato, una taza de café, etc. Ten en cuenta que puedes utilizar caracteres como *, +, <, #, @, etc. ¡Échale imaginación!

2. Variables

2.1 Definición y tipos de variables

Una variable es un contenedor de información. Imagina una variable como una cajita con una etiqueta que indica su nombre, una cajita en la que se puede introducir un valor. Las variables pueden almacenar valores enteros, números decimales, caracteres, cadenas de caracteres (palabras o frases), etc. El contenido de las variables puede cambiar durante la ejecución del programa, de ahí viene el nombre de “variable”.

Java es un lenguaje fuertemente tipado, es decir, es necesario declarar todas las variables que utilizará el programa, indicando siempre el nombre y el tipo de cada una.

El nombre que se le da a una variable es muy importante; intenta usar siempre nombres significativos que, de alguna forma, identifiquen el contenido. Por ejemplo, si usas una variable para almacenar el volumen de una figura, una buena opción sería llamarla `volumen`; si tienes una variable que almacena la edad de una persona, lo mejor es llamarla `edad`, y así sucesivamente.

Un programa se lee y se depura mucho mejor si los nombres de las variables se han elegido de forma inteligente.

¿Podrías averiguar qué hace la siguiente línea de código?

```
x = vIp3 * Weerty - zxc;
```

Ahora observa el mismo código pero con otros nombres de variables.

```
precioTotal = cantidad * precio - descuento;
```

Se entiende mucho mejor ¿verdad?

Escribiremos los nombres de variables en formato *lowerCamelCase*. La primera letra se escribe en minúscula y, a continuación, si se utiliza más de una palabra, cada una de ellas empezaría con mayúscula. Por ejemplo, `edadMin` es un buen nombre para una variable que almacena la edad mínima a la que se puede acceder a un sitio web. Observa que hemos usado una mayúscula para diferenciar dos partes (`edad` y `Min`). Puede que en algún libro también encuentres nombres de variables con el carácter de subrayado (`_`) de tal forma que el nombre de la variable sería `edad_min`, pero como hemos comentado anteriormente, en este libro nos ceñimos al [estándar de Google](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html)¹, que exige el formato *lowerCamelCase*.

¹<http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

No se permiten símbolos como \$, %, @, +, -, etc. Puedes usar números en los nombres de variables pero nunca justo al principio; 5x no es un nombre válido pero x5 sí lo es.

No se debe poner una letra mayúscula al comienzo del nombre de una variable para no confundirla con una clase (los nombres de las clases comienzan por mayúscula).

2.1.1 Enteros (int y long)

Las variables que van a contener números enteros se declaran con int. Veamos un ejemplo.

```
/**
 * Uso de variables enteras
 *
 * @author Luis J. Sánchez
 */

public class VariablesEnteras {
    public static void main(String[] args) {
        int x; // Se declara la variable x como entera

        x = 5; // Se asigna el valor 5 a la variable x
        System.out.println("El valor actual de mi variable es " + x);

        x = 7; // Se asigna el valor 7 a la variable x
        System.out.println("y ahora es " + x);
    }
}
```

Si pretendemos almacenar valores muy grandes en una variable, usaremos el tipo long en lugar de int.

2.1.2 Números decimales (double y float)

Usamos los tipos double o float cuando queremos (o esperamos) almacenar números con decimales en las variables.

Ciñéndonos al estándar de Google, no daremos por válidas definiciones de variables como la siguiente (aunque funcionaría sin ningún problema).

```
double x, y;
```

Cada variable se debe definir en una línea diferente. Lo siguiente sí sería correcto.

```
double x;  
double y;
```

A continuación tienes un ejemplo completo.

```
/**  
 * Uso de variables que contienen números decimales  
 */  
* @author Luis J. Sánchez  
*/  
  
public class VariablesConDecimales {  
    public static void main(String[] args) {  
        double x; // Se declaran las variables x e y  
        double y; // de tal forma que puedan almacenar decimales.  
  
        x = 7;  
        y = 25.01;  
  
        System.out.println(" x vale " + x);  
        System.out.println(" y vale " + y);  
    }  
}
```

Como puedes ver, también se pueden almacenar números enteros en variables de tipo double.

2.1.3 Cadenas de caracteres (String)

Las cadenas de caracteres se utilizan para almacenar palabras y frases. Todas las cadenas de caracteres deben ir entrecomilladas.

```
/**  
 * Uso del tipo String  
 */  
* @author Luis J. Sánchez  
*/  
  
public class UsoDeStrings {  
    public static void main(String[] args) {  
        String miPalabra = "cerveza";  
        String miFrase = "¿dónde está mi cerveza?";  
  
        System.out.println("Una palabra que uso con frecuencia: " + miPalabra);  
    }  
}
```



```
        System.out.println("Una frase que uso a veces: " + miFrase);  
    }  
}
```

Como puedes ver, en una cadena de caracteres se pueden almacenar signos de puntuación, espacios y letras con tildes.

2.2 Operadores aritméticos

En Java se puede operar con las variables de una forma muy parecida a como se hace en matemáticas. Los operadores aritméticos de Java son los siguientes:

OPERADOR	NOMBRE	EJEMPLO	DESCRIPCIÓN
+	suma	20 + x	suma dos números
-	resta	a - b	resta dos números
*	multiplicación	10 * 7	multiplica dos números
/	división	altura / 2	divide dos números
%	resto (módulo)	5 % 2	resto de la división entera
++	incremento	a++	incrementa en 1 el valor de la variable
--	decremento	a--	decrementa en 1 el valor de la variable

A continuación tienes un programa que ilustra el uso de los operadores aritméticos.

```
/**
 * Uso de los operadores aritméticos
 *
 * @author Luis J. Sánchez
 */

public class UsoDeOperadoresAritmeticos {
    public static void main(String[] args) {
        int x;
        x = 100;

        System.out.println(x + " " + (x + 5) + " " + (x - 5));
        System.out.println((x * 5) + " " + (x / 5) + " " + (x % 5));
    }
}
```

2.3 Asignación de valores a variables

La sentencia de asignación se utiliza para dar un valor a una variable. En Java (y en la mayoría de lenguajes de programación) se utiliza el símbolo igual (=) para este cometido. Es importante recalcar que una asignación no es una ecuación. Por ejemplo `x = 7 + 1` es una asignación en la cual se evalúa la parte derecha `7 + 1`, y el resultado de esa evaluación se almacena en la variable que se coloque a la izquierda del igual, es decir, en la `x`, o lo que es lo mismo, el número 8 se almacena en `x`. La sentencia `x + 1 = 23 * 2` no es una asignación válida ya que en el lado izquierdo debemos tener únicamente un nombre de variable.

Veamos un ejemplo con algunas operaciones y asignaciones.

```
/**
 * Operaciones y asignaciones
 *
 * @author Luis J. Sánchez
 */

public class Asignaciones {
    public static void main(String[] args) {

        int x = 2;
        int y = 9;

        int sum = x + y;
        System.out.println("La suma de mis variables es " + sum);

        int mul = x * y;
        System.out.println("La multiplicación de mis variables es " + mul);
    }
}
```

2.4 Conversión de tipos (*casting*)

En ocasiones es necesario convertir una variable (o una expresión en general) de un tipo a otro. Simplemente hay que escribir entre paréntesis el tipo que se quiere obtener. Experimenta con el siguiente programa y observa los diferentes resultados que se obtienen.

```
/**
 * Conversión de tipos
 *
 * @author Luis J. Sánchez
 */

public class ConversionDeTipos {
    public static void main(String[] args) {

        int x = 2;
        int y = 9;
        double division;

        division = (double)y / (double)x;
        //division = y / x; // Comenta esta línea y
```

```
        // observa la diferencia.

        System.out.println("El resultado de la división es " + division);
    }
}
```

2.5 Ejercicios



Ejercicio 1

Escribe un programa en el que se declaren las variables enteras x e y . Asigna los valores 144 y 999 respectivamente. A continuación, muestra por pantalla el valor de cada variable, la suma, la resta, la división y la multiplicación.



Ejercicio 2

Crea la variable nombre y asígnale tu nombre completo. Muestra su valor por pantalla de tal forma que el resultado del programa sea el mismo que en el ejercicio 1.



Ejercicio 3

Crea las variables nombre, direccion y telefono y asígnale los valores correspondientes. Muestra los valores de esas variables por pantalla de tal forma que el resultado del programa sea el mismo que en el ejercicio 2.



Ejercicio 4

Realiza un conversor de euros a pesetas. La cantidad en euros que se quiere convertir deberá estar almacenada en una variable.



Ejercicio 5

Realiza un conversor de pesetas a euros. La cantidad en pesetas que se quiere convertir deberá estar almacenada en una variable.



Ejercicio 6

Escribe un programa que calcule el total de una factura a partir de la base imponible (precio sin IVA). La base imponible estará almacenada en una variable.

3. Lectura de datos desde teclado

Hemos visto en los capítulos anteriores cómo mostrar información por pantalla y cómo usar variables. Veremos ahora algo muy importante que te permitirá realizar programas algo más funcionales, que tengan una utilidad real; aprenderás a leer la información que introduce un usuario mediante el teclado.

El funcionamiento de casi todos los programas se podría resumir en los siguientes puntos:

1. Entrada de datos desde teclado (o desde cualquier otro dispositivo de entrada)
2. Procesamiento de los datos de entrada para producir un resultado
3. Visualización de los resultados por pantalla

De momento, hemos visto cómo realizar los puntos 2 y 3. Vamos a ver a continuación cómo se recoge la información desde el teclado.

3.1 Lectura de texto

Para recoger datos por teclado usamos `System.console().readLine()`. Cuando llegamos a esta sentencia, el programa se detiene y espera que el usuario introduzca información mediante el teclado. La introducción de datos termina con la pulsación de la tecla INTRO. Una vez que el usuario presiona INTRO, todo lo que se ha tecleado se almacena en una variable, en el siguiente ejemplo esa variable es `nombre`.

```
/**
 * Lectura de datos desde teclado
 *
 * @author Luis J. Sanchez
 */

public class DimeTuNombre {
    public static void main(String[] args) {
        String nombre;
        System.out.print("Por favor, dime cómo te llamas: ");
        nombre = System.console().readLine();
        System.out.println("Hola " + nombre + ", ¡encantado de conocerte!");
    }
}
```



Mediante `System.console().readLine()` se recoge una línea de texto introducida por teclado.

Cuando se piden datos por teclado es importante que el programa especifique claramente cuál es exactamente la información que requiere por parte del usuario. Date cuenta que este programa muestra el mensaje “Por favor, dime cómo te llamas:”. Imagina que se omite este mensaje, en la pantalla aparecería únicamente un cursor parpadeando y el usuario no sabría qué tiene que hacer o qué dato introducir.

También es importante reseñar que los datos introducidos por teclado se recogen como una cadena de caracteres (un `String`).

3.2 Lectura de números

Si en lugar de texto necesitamos datos numéricos, deberemos convertir la cadena introducida en un número con el método adecuado. Como se muestra en el ejemplo, `Integer.parseInt()` convierte el texto introducido por teclado en un dato numérico, concretamente en un número entero.

```
/**
 * Lectura de datos desde teclado
 *
 * @author Luis J. Sánchez
 */

public class LeeNumeros {
    public static void main(String[] args) {

        String linea;

        System.out.print("Por favor, introduce un número: ");
        linea = System.console().readLine();
        int primerNumero;
        primerNumero = Integer.parseInt( linea );

        System.out.print("introduce otro, por favor: ");
        linea = System.console().readLine();
        int segundoNumero;
        segundoNumero = Integer.parseInt( linea );

        int total;
        total = (2 * primerNumero) + segundoNumero;

        System.out.print("El primer número introducido es " + primerNumero);
```

```
        System.out.println(" y el segundo es " + segundoNumero);
        System.out.print("El doble del primer número más el segundo es ");
        System.out.print(total);
    }
}
```

Este último programa se podría acortar un poco. Por ejemplo, estas dos líneas

```
int total;
total = (2 * primerNumero) + segundoNumero;
```

se podrían quedar en una sola línea

```
int total = (2 * primerNumero) + segundoNumero;
```

De igual modo, estas tres líneas

```
linea = System.console().readLine();
int primerNumero;
primerNumero = Integer.parseInt( linea );
```

también se podrían reducir a una sola tal que así

```
int primerNumero = Integer.parseInt( System.console().readLine() );
```

Es muy importante que el código de nuestros programas sea limpio y legible. A veces, abreviando demasiado el código se hace más difícil de leer; es preferible tener unas líneas de más y que el código se entienda bien a tener un código muy compacto pero menos legible.

3.3 La clase Scanner

El método `System.console().readLine()` funciona bien en modo consola (en una ventana de terminal) pero puede provocar problemas cuando se trabaja con IDEs como **Eclipse**, **Netbeans**, **JavaEdit**, etc. Para evitar estos problemas puedes usar la clase `Scanner` cuando necesites recoger datos desde teclado. La clase `Scanner` funciona tanto en entornos integrados como en una ventana de terminal.


```
/**
 * Lectura de datos desde teclado usando la clase Scanner
 *
 * @author Luis J. Sánchez
 */

import java.util.Scanner;

public class LeeDatosScanner01 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);

        System.out.print("Introduce tu nombre: ");
        String nombre = s.nextLine();

        System.out.print("Introduce tu edad: ");
        int edad = Integer.parseInt(s.nextLine());

        System.out.println("Tu nombre es " + nombre + " y tu edad es " + edad);
    }
}
```

Fíjate que en el programa anterior la sentencia

```
s.nextLine()
```

sería el equivalente a

```
System.console().readLine()`
```

Mediante el uso de la clase Scanner es posible leer varios datos en una misma línea. En el programa anterior se pedía un nombre y una edad, en total dos datos que había que introducir en líneas separadas. Observa cómo en el siguiente ejemplo se piden esos dos datos en una sola línea y separados por un espacio.

```
/**
 * Lectura de datos desde teclado usando la clase Scanner
 *
 * @author Luis J. Sánchez
 */

import java.util.Scanner;

public class LeeDatosScanner02 {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);

        System.out.print("Introduce tu nombre y tu edad separados por un espacio: ");
        String nombre = s.next();
        int edad = s.nextInt();

        System.out.println("Tu nombre es " + nombre + " y tu edad es " + edad);
    }
}
```

Fíjate cómo se ha utilizado `s.next()` para leer una cadena de caracteres y `s.nextInt()` para leer un número entero, todo ello en la misma línea.

El siguiente programa de ejemplo calcula la media de tres números decimales. Para leer cada uno de los números en la misma línea se utiliza `s.nextDouble()`.

```
/**
 * Lectura de datos desde teclado usando la clase Scanner
 *
 * @author Luis J. Sánchez
 */

import java.util.Scanner;

public class LeeDatosScannerMedia {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);

        System.out.print("Introduce tres números (pueden contener decimales) separados por espacios: ");
        double x1 = s.nextDouble();
        double x2 = s.nextDouble();
        double x3 = s.nextDouble();

        double media = (x1 + x2 + x3) / 3;
    }
}
```

```
    System.out.println("La media de esos tres números es " + media);  
  }  
}
```



Recuerda que **¡a programar se aprende programando!** Es muy importante que pruebes los ejemplos - si los modificas y experimentas, todavía mejor - y que realices los ejercicios. Para aprender a programar no basta con leer el libro y entenderlo, es una condición necesaria pero no suficiente. Igual que para ser piloto hacen falta muchas horas de vuelo, para ser programador hacen falta muchas horas picando código, probando, corrigiendo errores... ¡Ánimo! ¡El esfuerzo merece la pena!

3.4 Ejercicios



Ejercicio 1

Realiza un programa que pida dos números y que luego muestre el resultado de su multiplicación.



Ejercicio 2

Realiza un conversor de euros a pesetas. La cantidad de euros que se quiere convertir debe ser introducida por teclado.



Ejercicio 3

Realiza un conversor de pesetas a euros. La cantidad de pesetas que se quiere convertir debe ser introducida por teclado.



Ejercicio 4

Escribe un programa que sume, reste, multiplique y divida dos números introducidos por teclado.



Ejercicio 5

Escribe un programa que calcule el área de un rectángulo.



Ejercicio 6

Escribe un programa que calcule el área de un triángulo.



Ejercicio 7

Escribe un programa que calcule el total de una factura a partir de la base imponible.



Ejercicio 8

Escribe un programa que calcule el salario semanal de un empleado en base a las horas trabajadas, a razón de 12 euros la hora.

**Ejercicio 9**

Escribe un programa que calcule el volumen de un cono según la fórmula $V = \frac{1}{3}\pi r^2 h$

**Ejercicio 10**

Realiza un conversor de Mb a Kb.

**Ejercicio 11**

Realiza un conversor de Kb a Mb.

4. Sentencia condicional (if y switch)

Una sentencia condicional permite al programa bifurcar el flujo de ejecución de instrucciones dependiendo del valor de una expresión.

4.1 Sentencia if

La sentencia `if` permite la ejecución de una serie de instrucciones en función del resultado de una expresión lógica. El resultado de evaluar una expresión lógica es siempre verdadero (`true`) o falso (`false`). Es muy simple, en lenguaje natural sería algo como "si esta condición es verdadera entonces haz esto, sino haz esto otro".

El formato de la sentencia `if` es el siguiente:

```
if (condición) {  
  
    instrucciones a ejecutar si la condición es verdadera  
  
} else {  
  
    instrucciones a ejecutar si la condición es falsa  
  
}
```

A continuación se muestra un ejemplo del uso de la sentencia `if`.

```
/**  
 * Sentencia if  
 *  
 * @author Luis J. Sánchez  
 */  
  
public class SentenciaIf01 {  
    public static void main(String[] args) {  
        System.out.print("¿Cuál es la capital de Kiribati? ");  
        String respuesta = System.console().readLine();  
  
        if (respuesta.equals("Tarawa")) {  
            System.out.println("¡La respuesta es correcta!");  
        }  
    }  
}
```

```

    } else {
        System.out.println("Lo siento, la respuesta es incorrecta.");
    }
}
}

```

En el programa se le pregunta al usuario cuál es la capital de Kiriwati. La respuesta introducida por el usuario se almacena en la variable respuesta. A continuación viene la sentencia condicional `if`.

```
if (respuesta.equals("Tarawa"))
```

Llegado a este punto, el programa evalúa la expresión `respuesta.equals("Tarawa")`. Observa que para comparar dos cadenas de caracteres se utiliza `equals()`. Imaginemos que el usuario ha introducido por teclado Madrid; entonces la expresión `"Madrid".equals("Tarawa")` daría como resultado `false` (falso).

Si la expresión hubiera dado como resultado `true` (verdadero), se ejecutaría la línea

```
System.out.println("¡La respuesta es correcta!");
```

pero no es el caso, el resultado de la expresión ha sido `false` (falso), todo el mundo sabe que la capital de Kiriwati no es Madrid, por tanto se ejecutaría la línea

```
System.out.println("Lo siento, la respuesta es incorrecta.");
```

Vamos a ver otro ejemplo, esta vez con números. El usuario introducirá un número por teclado y el programa dirá si se trata de un número positivo o negativo.

```

/**
 * Sentencia if
 *
 * @author Luis J. Sánchez
 */

public class SentenciaIf02 {
    public static void main(String[] args) {
        System.out.print("Por favor, introduce un número entero: ");
        String linea = System.console().readLine();
        int x = Integer.parseInt( linea );

        if (x < 0) {
            System.out.println("El número introducido es negativo.");
        } else {

```

```
        System.out.println("El número introducido es positivo.");
    }
}
```

El siguiente bloque de código

```
if (x < 0)
    System.out.println("El número introducido es negativo.");
else
    System.out.println("El número introducido es positivo.");
```

compilaría y funcionaría sin problemas - fíjate que hemos quitado las llaves - ya que antes y después del `else` hay una sola sentencia y en estos casos no es obligatorio poner llaves. Sin embargo, nosotros siempre usaremos llaves, es una exigencia del estándar de Google al que nos ceñimos en este manual.



Llaves egipcias (egyptian brackets)

Fíjate en la manera de colocar las llaves dentro del código de un programa en Java. La llave de apertura de bloque se coloca justo al final de la línea y la llave de cierre va justo al principio de la línea. Se llaman **llaves egipcias**¹ por la similitud entre las llaves y las manos de los egipcios que aparecen en los papiros.

¹<http://blog.codinghorror.com/new-programming-jargon/>

4.2 Operadores de comparación

En el ejemplo anterior, usamos el operador `<` en la comparación `if (x < 0)` para saber si la variable `x` es menor que cero. Hay más operadores de comparación, en la siguiente tabla se muestran todos.

OPERADOR	NOMBRE	EJEMPLO	DESCRIPCIÓN
<code>==</code>	igual	<code>a == b</code>	a es igual a b
<code>!=</code>	distinto	<code>a != b</code>	a es distinto de b
<code><</code>	menor que	<code>a < b</code>	a es menor que b
<code>></code>	mayor que	<code>a > b</code>	a es mayor que b
<code><=</code>	mayor o igual que	<code>a >= b</code>	a es mayor o igual que b

El siguiente ejemplo muestra el uso de uno de estos operadores, concretamente de `>=` (mayor o igual). El usuario introduce una nota; si esta nota es **mayor o igual a 5** se le mostrará un mensaje diciendo que ha aprobado y en caso de que no se cumpla la condición se mostrará un mensaje diciendo que está suspenso.

```
/**
 * Sentencia if
 *
 * @author Luis J. Sánchez
 *
 */

public class SentenciaIf03 {
    public static void main(String[] args) {
        System.out.print("¿Qué nota has sacado en el último examen? ");
        String line = System.console().readLine();
        double nota = Double.parseDouble( line );

        if (nota >= 5) {
            System.out.println("¡Enhorabuena!, ¡has aprobado!");
        } else {
            System.out.println("Lo siento, has suspendido.");
        }
    }
}
```

4.3 Operadores lógicos

Los operadores de comparación se pueden combinar con los operadores lógicos. Por ejemplo, si queremos saber si la variable *a* es mayor que *b* y además es menor que *c*, escribiríamos `if ((a > b) && (a < c))`. En la siguiente tabla se muestran los operadores lógicos de Java:

OPERADOR	NOMBRE	EJEMPLO	DEVUELVE VERDADERO CUANDO...
<code>&&</code>	y	<code>(7 > 2) && (2 < 4)</code>	las dos condiciones son verdaderas
<code> </code>	o	<code>(7 > 2) (2 < 4)</code>	al menos una de las condiciones es verdadera
<code>!</code>	no	<code>!(7 > 2)</code>	la condición es falsa

Vamos a ver cómo funcionan los operadores lógicos con un ejemplo. Mediante `if ((n < 1) || (n > 100))` se pueden detectar los números que no están en el rango de 1 a 100; literalmente sería “si *n* es menor que 1 o *n* es mayor que 100”.

```
/**
 * Operadores lógicos
 *
 * @author Luis J. Sánchez
 */

public class OperadoresLogicos01 {
    public static void main(String[] args) {
        System.out.println("Adivina el número que estoy pensando.");
        System.out.print("Introduce un número entre el 1 y el 100: ");
        String linea = System.console().readLine();
        int n = Integer.parseInt( linea );

        if ((n < 1) || (n > 100)) {
            System.out.println("El número introducido debe estar en el intervalo 1 - 100.");
            System.out.print("Tienes otra oportunidad, introduce un número: ");
            linea = System.console().readLine();
            n = Integer.parseInt( linea );
        }

        if (n == 24) {
            System.out.println("¡Enhorabuena!, ¡has acertado!");
        } else {
            System.out.println("Lo siento, ese no es el número que estoy pensando.");
        }
    }
}
```

```
}
```

En el siguiente programa puedes ver el uso de operadores lógicos combinado con operadores relacionales (operadores de comparación). Intenta adivinar cuál será el resultado mirando el código.

```
/**
 *
 * Operadores lógicos y relacionales
 *
 * @author Luis J. Sánchez
 *
 */

public class OperadoresLogicos02 {
    public static void main(String[] args) {
        boolean a = true;
        boolean b = false;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("a && b = " + (a && b));
        System.out.println("a || b = " + (a || b));
        System.out.println("!a = " + !a);
        System.out.println("a || (6 > 10) = " + (a || (6 > 10)));
        System.out.println("((4 <= 4) || false) && (!a) = " + (((4 <= 4) || false) && (!a)));
    }
}
```

4.4 Sentencia switch (selección múltiple)

A veces es necesario comparar el valor de una variable con una serie de valores concretos. La selección múltiple es muy parecida (aunque no es exactamente igual) a una secuencia de varias sentencias if.

El formato de switch es el que se muestra a continuación. En lenguaje natural sería algo así como “Si variable vale valor1 entonces entra por case valor1:, si variable vale valor2 entonces entra por case valor2:,... si variable no vale ninguno de los valores que hay en los distintos case entonces entra por default:.

```
switch(variable) {  
    case valor1:  
        sentencias  
        break;  
  
    case valor2:  
        sentencias  
        break;  
    .  
    .  
    .  
  
    default:  
        sentencias  
}
```

A continuación tienes un ejemplo completo en Java. Se pide al usuario un número de mes y el programa da el nombre del mes que corresponde a ese número.

```
/**  
 * Sentencia múltiple (switch)  
 *  
 * @author Luis José Sánchez  
 */  
  
public class SentenciaSwitch {  
    public static void main(String[] args) {  
  
        System.out.print("Por favor, introduzca un numero de mes: ");  
        int mes = Integer.parseInt(System.console().readLine());  
  
        String nombreDelMes;
```

```
switch (mes) {
    case 1:
        nombreDelMes = "enero";
        break;
    case 2:
        nombreDelMes = "febrero";
        break;
    case 3:
        nombreDelMes = "marzo";
        break;
    case 4:
        nombreDelMes = "abril";
        break;
    case 5:
        nombreDelMes = "mayo";
        break;
    case 6:
        nombreDelMes = "junio";
        break;
    case 7:
        nombreDelMes = "julio";
        break;
    case 8:
        nombreDelMes = "agosto";
        break;
    case 9:
        nombreDelMes = "septiembre";
        break;
    case 10:
        nombreDelMes = "octubre";
        break;
    case 11:
        nombreDelMes = "noviembre";
        break;
    case 12:
        nombreDelMes = "diciembre";
        break;
    default:
        nombreDelMes = "no existe ese mes";
}

System.out.println("Mes " + mes + ": " + nombreDelMes);
}
```

Observa que es necesario introducir un break después de la asignación de la variable nombreDelMes. En caso de no encontrarse el break, el programa continúa la ejecución en la línea siguiente.

El bloque que corresponde al default se ejecuta cuando la variable no coincide con ninguno de los valores de los case. Escribiremos siempre el default al final de la sentencia switch aunque no sea necesario.

La sentencia switch se utiliza con frecuencia para crear menús.

```
/**
 * Ejemplo de un menú con switch
 *
 * @author Luis José Sánchez
 */

public class MenuConSwitch {
    public static void main(String[] args) {

        System.out.println(" CÁLCULO DE ÁREAS");
        System.out.println(" -----");
        System.out.println(" 1. Cuadrado");
        System.out.println(" 2. Rectángulo");
        System.out.println(" 3. Triángulo");
        System.out.print("\n Elija una opción (1-3): ");

        int opcion = Integer.parseInt(System.console().readLine());

        double lado;
        double base;
        double altura;

        switch (opcion) {
            case 1:
                System.out.print("\nIntroduzca el lado del cuadrado en cm: ");
                lado = Double.parseDouble(System.console().readLine());
                System.out.println("\nEl área del cuadrado es " + (lado * lado) + " cm2");
                break;

            case 2:
                System.out.print("\nIntroduzca la base del rectángulo en cm: ");
                base = Double.parseDouble(System.console().readLine());
                System.out.print("Introduzca la altura del rectángulo en cm: ");
                altura = Double.parseDouble(System.console().readLine());
                System.out.println("El área del rectángulo es " + (base * altura) + " cm2");
                break;
```

```
    case 3:
        System.out.print("\nIntroduzca la base del triángulo en cm: ");
        base = Double.parseDouble(System.console().readLine());
        System.out.print("Introduzca la altura del triángulo en cm: ");
        altura = Double.parseDouble(System.console().readLine());
        System.out.println("El área del triángulo es " + ((base * altura) / 2) + " cm2");
        break;

    default:
        System.out.print("\nLo siento, la opción elegida no es correcta.");
}
}
```

4.5 Ejercicios



Ejercicio 1

Escribe un programa que pida por teclado un día de la semana y que diga qué asignatura toca a primera hora ese día.



Ejercicio 2

Realiza un programa que pida una hora por teclado y que muestre luego buenos días, buenas tardes o buenas noches según la hora. Se utilizarán los tramos de 6 a 12, de 13 a 20 y de 21 a 5, respectivamente. Sólo se tienen en cuenta las horas, los minutos no se deben introducir por teclado.



Ejercicio 3

Escribe un programa en que dado un número del 1 a 7 escriba el correspondiente nombre del día de la semana.



Ejercicio 4

Vamos a ampliar uno de los ejercicios de la relación anterior para considerar las horas extras. Escribe un programa que calcule el salario semanal de un trabajador teniendo en cuenta que las horas ordinarias (40 primeras horas de trabajo) se pagan a 12 euros la hora. A partir de la hora 41, se pagan a 16 euros la hora.



Ejercicio 5

Realiza un programa que resuelva una ecuación de primer grado (del tipo $ax + b = 0$).



Ejercicio 6

Realiza un programa que calcule el tiempo que tardará en caer un objeto desde una altura h . Aplica la fórmula $t = \sqrt{\frac{2h}{g}}$ siendo $g = 9.81m/s^2$



Ejercicio 7

Realiza un programa que calcule la media de tres notas.



Ejercicio 8

Amplía el programa anterior para que diga la nota del boletín (insuficiente, suficiente, bien, notable o sobresaliente).



Ejercicio 9

Realiza un programa que resuelva una ecuación de segundo grado (del tipo $ax^2 + bx + c = 0$).



Ejercicio 10

Escribe un programa que nos diga el horóscopo a partir del día y el mes de nacimiento.



Ejercicio 11

Escribe un programa que dada una hora determinada (horas y minutos), calcule los segundos que faltan para llegar a la medianoche.



Ejercicio 12

Realiza un minicuestionario con 10 preguntas tipo test sobre las asignaturas que se imparten en el curso. Cada pregunta acertada sumará un punto. El programa mostrará al final la calificación obtenida. Pásale el minicuestionario a tus compañeros y pídeles que lo hagan para ver qué tal andan de conocimientos en las diferentes asignaturas del curso.



Ejercicio 13

Escribe un programa que ordene tres números enteros introducidos por teclado.



Ejercicio 14

Realiza un programa que diga si un número introducido por teclado es par y/o divisible entre 5.



Ejercicio 15

Escribe un programa que pinte una pirámide rellena con un carácter introducido por teclado que podrá ser una letra, un número o un símbolo como *, +, -, \$, &, etc. El programa debe permitir al usuario mediante un menú elegir si el vértice de la pirámide está apuntando hacia arriba, hacia abajo, hacia la izquierda o hacia la derecha.



Ejercicio 16

Realiza un programa que nos diga si hay probabilidad de que nuestra pareja nos está siendo infiel. El programa irá haciendo preguntas que el usuario contestará con verdadero o falso. Cada pregunta contestada como verdadero sumará 3 puntos. Las preguntas contestadas con falso no suman puntos. Utiliza el fichero `test_infidelidad.txt` para obtener las preguntas y las conclusiones del programa.



Ejercicio 17

Escribe un programa que diga cuál es la última cifra de un número entero introducido por teclado.



Ejercicio 18

Escribe un programa que diga cuál es la primera cifra de un número entero introducido por teclado. Se permiten números de hasta 5 cifras.



Ejercicio 19

Realiza un programa que nos diga cuántos dígitos tiene un número entero que puede ser positivo o negativo. Se permiten números de hasta 5 dígitos.



Ejercicio 20

Realiza un programa que diga si un número entero positivo introducido por teclado es capicúa. Se permiten números de hasta 5 cifras.

5. Bucles

Los bucles se utilizan para repetir un conjunto de sentencias. Por ejemplo, imagina que es necesario introducir la notas de 40 alumnos con el fin de calcular la media, la nota máxima y la nota mínima. Podríamos escribir 40 veces la instrucción que pide un dato por teclado `System.console().readLine()` y convertir cada uno de esos datos a un número con decimales con 40 instrucciones `Double.parseDouble()`, no parece algo muy eficiente. Es mucho más práctico meter dentro de un bucle aquellas sentencias que queremos que se repitan.

Normalmente existe una condición de salida, que hace que el flujo del programa abandone el bucle y continúe justo en la siguiente sentencia. Si no existe condición de salida o si esta condición no se cumple nunca, se produciría lo que se llama un bucle infinito y el programa no terminaría nunca.

5.1 Bucle for

Se suele utilizar cuando se conoce previamente el número exacto de iteraciones (repeticiones) que se van a realizar. La sintaxis es la siguiente:

```
for (expresion1 ; expresion2 ; expresion3) {  
  
    sentencias  
  
}
```

Justo al principio se ejecuta `expresion1`, normalmente se usa para inicializar una variable. El bucle se repite mientras se cumple `expresion2` y en cada iteración del bucle se ejecuta `expresion3`, que suele ser el incremento o decremento de una variable. Con un ejemplo se verá mucho más claro.

```
/**  
 * Bucle for  
 *  
 * @author Luis José Sánchez  
 */  
public class EjemploFor {  
    public static void main(String[] args) {  
  
        for (int i = 1; i < 11; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

En este ejemplo, `int i = 1` se ejecuta solo una vez, antes que cualquier otra cosa; como ves, esta expresión se utiliza para inicializar la variable `i` a 1. Mientras se cumpla la condición `i < 11` el contenido del bucle, o sea, `System.out.println(i);` se va a ejecutar. En cada iteración del bucle, `i++` hace que la variable `i` se incremente en 1. El resultado del ejemplo es la impresión en pantalla de los números del 1 al 10.

Intenta seguir mentalmente el flujo del programa. Experimenta inicializando la variable `i` con otros valores, cambia la condición con `>` o `<=` y observa lo que sucede. Prueba también a cambiar el incremento de la variable `i`, por ejemplo con `i = i + 2`.

5.2 Bucle `while`

El bucle `while` se utiliza para repetir un conjunto de sentencias siempre que se cumpla una determinada condición. Es importante reseñar que la condición se comprueba al comienzo del bucle, por lo que se podría dar el caso de que dicho bucle no se ejecutase nunca. La sintaxis es la siguiente:

```
while (expresion) {  
  
    sentencias  
  
}
```

Las sentencias se ejecutan una y otra vez mientras `expresion` sea verdadera. El siguiente ejemplo produce la misma salida que el ejemplo anterior, muestra cómo cambian los valores de `i` del 1 al 10.

```
/**  
 * Bucle while  
 *  
 * @author Luis José Sánchez  
 */  
  
public class EjemploWhile {  
    public static void main(String[] args) {  
  
        int i = 1;  
  
        while (i < 11) {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

En el siguiente ejemplo se cuentan y se suman los números que se van introduciendo por teclado. Para indicarle al programa que debe dejar de pedir números, el usuario debe introducir un número negativo; esa será la condición de salida del bucle. Observa que el bucle se repite mientras el número introducido sea mayor o igual que cero.

```
/**
 * Bucle while que termina cuando se introduce por teclado un
 * número negativo.
 *
 * @author Luis José Sánchez
 */

public class CuentaPositivos {

    public static void main(String[] args) {

        System.out.println("Por favor, vaya introduciendo números y pulsando INTRO.");
        System.out.println("Para terminar, introduzca un número negativo.");

        int numeroIntroducido = 0;
        int cuentaNumeros = 0;
        int suma = 0;

        while (numeroIntroducido >= 0) {
            numeroIntroducido = Integer.parseInt(System.console().readLine());
            cuentaNumeros++; // Incrementa en uno la variable
            suma += numeroIntroducido; // Equivale a suma = suma + NumeroIntroducido
        }

        System.out.println("Has introducido " + (cuentaNumeros - 1) + " números positivos.");
        System.out.println("La suma total de ellos es " + (suma - numeroIntroducido));
    }
}
```

5.3 Bucle do-while

El bucle do-while funciona de la misma manera que el bucle while, con la salvedad de que expresión se evalúa al final de la iteración. Las sentencias que encierran el bucle do-while, por tanto, se ejecutan como mínimo una vez. La sintaxis es la siguiente:

```
do {  
  
    sentencias  
  
} while (expresion)
```

El siguiente ejemplo es el equivalente do-while a los dos ejemplos anteriores que cuentan del 1 al 10.

```
/**  
 * Bucle do-while  
 *  
 * @author Luis José Sánchez  
 */  
  
public class EjemploDoWhile {  
    public static void main(String[] args) {  
  
        int i = 1;  
  
        do {  
            System.out.println(i);  
            i++;  
        } while (i < 11);  
    }  
}
```

Veamos otro ejemplo. En este caso se van a ir leyendo números de teclado mientras el número introducido sea par; el programa parará, por tanto, cuando se introduzca un número impar.

```
/**  
 * Bucle do-while que termina cuando se introduce por teclado un  
 * número impar.  
 *  
 * @author Luis José Sánchez  
 */  
  
public class TerminaCuandoEsImpar {  
  
    public static void main(String[] args) {  
  
        int numero;  
  
        do {
```

```
System.out.print("Dime un número: ");
numero = Integer.parseInt(System.console().readLine());

if (numero % 2 == 0) {// comprueba si el número introducido es par
    System.out.println("Qué bonito es el " + numero);
} else {
    System.out.println("No me gustan los números impares, adiós.");
}
} while (numero % 2 == 0);
}
}
```

Te invito a que realices una modificación sobre este último ejemplo. Después de pedir un número, haz que el programa diga ¿Quiere continuar? (s/n). Si el usuario introduce una s o una S, el programa deberá continuar pidiendo números.

Cualquier bucle que un programador quiera realizar en Java lo puede implementar con for, while o do-while; por tanto, con una sola de estas tres opciones tendría suficiente. No obstante, según el programa en cuestión, una u otra posibilidad se adapta mejor que las otras y resulta más elegante. Con la experiencia, te irás dando cuenta cuándo es mejor utilizar cada una.

5.4 Ejercicios



Ejercicio 1

Muestra los números múltiplos de 5 de 0 a 100 utilizando un bucle `for`.



Ejercicio 2

Muestra los números múltiplos de 5 de 0 a 100 utilizando un bucle `while`.



Ejercicio 3

Muestra los números múltiplos de 5 de 0 a 100 utilizando un bucle `do-while`.



Ejercicio 4

Muestra los números del 320 al 160, contando de 20 en 20 hacia atrás utilizando un bucle `for`.



Ejercicio 5

Muestra los números del 320 al 160, contando de 20 en 20 hacia atrás utilizando un bucle `while`.



Ejercicio 6

Muestra los números del 320 al 160, contando de 20 en 20 utilizando un bucle `do-while`.



Ejercicio 7

Realiza el control de acceso a una caja fuerte. La combinación será un número de 4 cifras. El programa nos pedirá la combinación para abrirla. Si no acertamos, se nos mostrará el mensaje “Lo siento, esa no es la combinación” y si acertamos se nos dirá “La caja fuerte se ha abierto satisfactoriamente”. Tendremos cuatro oportunidades para abrir la caja fuerte.



Ejercicio 8

Muestra la tabla de multiplicar de un número introducido por teclado.



Ejercicio 9

Realiza un programa que nos diga cuántos dígitos tiene un número introducido por teclado.



Ejercicio 10

Escribe un programa que calcule la media de un conjunto de números positivos introducidos por teclado. A priori, el programa no sabe cuántos números se introducirán. El usuario indicará que ha terminado de introducir los datos cuando meta un número negativo.



Ejercicio 11

Escribe un programa que muestre en tres columnas, el cuadrado y el cubo de los 5 primeros números enteros a partir de uno que se introduce por teclado.



Ejercicio 12

Escribe un programa que muestre los n primeros términos de la serie de Fibonacci. El primer término de la serie de Fibonacci es 0, el segundo es 1 y el resto se calcula sumando los dos anteriores, por lo que tendríamos que los términos son 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144... El número n se debe introducir por teclado.



Ejercicio 13

Escribe un programa que lea una lista de diez números y determine cuántos son positivos, y cuántos son negativos.



Ejercicio 14

Escribe un programa que pida una base y un exponente (entero positivo) y que calcule la potencia.



Ejercicio 15

Escribe un programa que dados dos números, uno real (base) y un entero positivo (exponente), saque por pantalla todas las potencias con base el número dado y exponentes entre uno y el exponente introducido. No se deben utilizar funciones de exponenciación. Por ejemplo, si introducimos el 2 y el 5, se deberán mostrar 2^1 , 2^2 , 2^3 , 2^4 y 2^5 .



Ejercicio 16

Escribe un programa que diga si un número introducido por teclado es o no primo. Un número primo es aquel que sólo es divisible entre él mismo y la unidad.



Ejercicio 17

Realiza un programa que sume los 100 números siguientes a un número entero y positivo introducido por teclado. Se debe comprobar que el dato introducido es correcto (que es un número positivo).



Ejercicio 18

Escribe un programa que obtenga los números enteros comprendidos entre dos números introducidos por teclado y validados como distintos, el programa debe empezar por el menor de los enteros introducidos e ir incrementando de 7 en 7.



Ejercicio 19

Realiza un programa que pinte una pirámide por pantalla. La altura se debe pedir por teclado. El carácter con el que se pinta la pirámide también se debe pedir por teclado.



Ejercicio 20

Igual que el ejercicio anterior pero esta vez se debe pintar una pirámide hueca.



Ejercicio 21

Realiza un programa que vaya pidiendo números hasta que se introduzca un número negativo y nos diga cuantos números se han introducido, la media de los impares y el mayor de los pares. El número negativo sólo se utiliza para indicar el final de la introducción de datos pero no se incluye en el cómputo.



Ejercicio 22

Muestra por pantalla todos los números primos entre 2 y 100, ambos incluidos.



Ejercicio 23

Escribe un programa que permita ir introduciendo una serie indeterminada de números mientras su suma no supere el valor 10000. Cuando esto último ocurra, se debe mostrar el total acumulado, el contador de los números introducidos y la media.



Ejercicio 24

Escribe un programa que lea un número n e imprima una pirámide de números con n filas como en la siguiente figura:

```
1
121
12321
1234321
```



Ejercicio 25

Realiza un programa que pida un número por teclado y que luego muestre ese número al revés.



Ejercicio 26

Realiza un programa que pida primero un número y a continuación un dígito. El programa nos debe dar la posición (o posiciones) contando de izquierda a derecha que ocupa ese dígito en el número introducido.



Ejercicio 27

Escribe un programa que muestre, cuente y sume los múltiplos de 3 que hay entre 1 y un número leído por teclado.



Ejercicio 28

Escribe un programa que calcule el factorial de un número entero leído por teclado.



Ejercicio 29

Escribe un programa que muestre por pantalla todos los números enteros positivos menores a uno leído por teclado que no sean divisibles entre otro también leído de igual forma.



Ejercicio 30

Realiza una programa que calcule las horas transcurridas entre dos horas de dos días de la semana. No se tendrán en cuenta los minutos ni los segundos. El día de la semana se puede pedir como un número (del 1 al 7) o como una cadena (de “lunes” a “domingo”). Se debe comprobar que el usuario introduce los datos correctamente y que el segundo día es posterior al primero. A continuación se muestra un ejemplo:

```
Por favor, introduzca la primera hora.
```

```
Día: lunes
```

```
Hora: 18
```

```
Por favor, introduzca la segunda hora.
```

```
Día: martes
```

```
Hora: 20
```

```
Entre las 18:00h del lunes y las 20:00h del martes hay 26 hora/s.
```

6. Números aleatorios

Los números aleatorios se utilizan con frecuencia en programación para emular el comportamiento de algún fenómeno natural, el resultado de un juego de azar o, en general, para generar cualquier valor impredecible *a priori*.

Por ejemplo, se pueden utilizar números aleatorios para generar tiradas de dados de tal forma que, de antemano, no se puede saber el resultado. Antes de tirar un dado no sabemos si saldrá un 3 o un 5; se tratará pues de un número impredecible; lo que sí sabemos es que saldrá un número entre el 1 y el 6, es decir, podemos acotar el rango de los valores que vamos a obtener de forma aleatoria.

6.1 Generación de números aleatorios con y sin decimales

Para generar valores aleatorios utilizaremos `Math.random()`. Esta función genera un número con decimales (de tipo `double`) en el intervalo `[0 - 1)`, es decir, genera un número mayor o igual que 0 y menor que 1.

El siguiente programa genera diez números aleatorios:

```
/**
 * Generación de números aleatorios.
 *
 * @author Luis José Sánchez
 */

public class Aleatorio01 {
    public static void main(String[] args) {

        System.out.println("Diez números aleatorios:\n");

        for (int i = 1; i < 11; i++) {
            System.out.println(Math.random());
        }
    }
}
```

La salida del programa puede ser algo como esto:

```
0.30830376099567813
0.8330493363981046
0.2322483682676435
0.3130746053770094
0.34192944433558736
0.9636470440975567
0.3398896383918959
0.79825461825305
0.9868509622870223
0.9967893773101499
```

Te invito a que ejecutes varias veces el programa. Podrás observar que cada vez salen números diferentes, aunque siempre están comprendidos entre 0 y 1 (incluyendo el 0).

Pensarás que no es muy útil generar números aleatorios entre 0 y 1 si lo que queremos es por ejemplo sacar una carta al azar de la baraja española; pero en realidad un número decimal entre 0 y 1 es lo único que nos hace falta para generar cualquier tipo de valor aleatorio siempre y cuando se manipule ese número de la forma adecuada.

Por ejemplo, si queremos generar valores aleatorios entre 0 y 10 (incluyendo el 0 y sin llegar a 10) simplemente tendremos que correr la coma un lugar o, lo que es lo mismo, multiplicar por 10.

```
/**
 * Generación de números aleatorios.
 *
 * @author Luis José Sánchez
 */

public class Aleatorio02 {
    public static void main(String[] args) {

        System.out.println("20 números aleatorios entre 0 y 10");
        System.out.println(" sin llegar a 10 (con decimales):");

        for (int i = 1; i <= 20; i++) {
            System.out.println( Math.random()*10 + " ");
        }
    }
}
```

El programa anterior genera una salida como ésta:

```

0.07637674702636321
1.025787417143682
1.854993461897163
5.690351111720931
3.82310645589797
5.518007662236258
9.23529380254256
3.9201032643833376
5.836554253122096
6.224559064261578
7.652976185871555
4.9922807025365135
7.498156441347868
8.743251697509109
9.727764845406675
2.929766691797686
0.05801413446517634
2.1575652936687284

```

Si queremos generar números enteros en lugar de números con decimales, basta con hacer un casting para convertir los números de tipo `double` en números de tipo `int`. Recuerda que `(int)x` transforma `x` en una variable de tipo entero; si `x` era de tipo `float` o `double`, perdería todos los decimales.

```

/**
 * Generación de números aleatorios.
 *
 * @author Luis José Sánchez
 */

public class Aleatorio03 {
    public static void main(String[] args) {

        System.out.println("20 números aleatorios entre 0 y 9 (sin decimales):");

        for (int i = 1; i <= 20; i++) {
            System.out.print((int)(Math.random()*10) + " ");
        }

        System.out.println();
    }
}

```

La salida del programa debe ser algo muy parecido a esto:

```
20 números aleatorios entre 0 y 9 (sin decimales):
0 8 0 3 8 8 7 3 2 0 8 2 1 2 9 0 6 4 5 4
```

¿Y si en lugar de generar números enteros entre 0 y 9 queremos generar números entre 1 y 10? Como habrás podido adivinar, simplemente habría que sumar 1 al número generado, de esta forma se “desplazan un paso” los valores generados al azar, de tal forma que el mínimo valor que se produce sería el $0 + 1 = 1$ y el máximo sería $9 + 1 = 10$.

```
/**
 * Generación de números aleatorios.
 *
 * @author Luis José Sánchez
 */

public class Aleatorio04 {

    public static void main(String[] args) {

        System.out.println("20 números aleatorios entre 1 y 10 (sin decimales):");

        for (int i = 1; i <= 20; i++) {
            System.out.print( (int)(Math.random()*10 + 1) + " ");
        }

        System.out.println();
    }
}
```

Vamos a ponerlo un poco más difícil. Ahora vamos a generar números enteros entre 50 y 60 ambos incluidos. Primero multiplicamos `Math.random()` por 11, con lo que obtenemos números decimales entre 0 y 10.9999... (sin llegar nunca hasta 11). Luego desplazamos ese intervalo sumando 50 por lo que obtenemos números decimales entre 50 y 60.9999... Por último, quitamos los decimales haciendo casting y *voilà*, ya tenemos números enteros aleatorios entre 50 y 60 ambos incluidos.

```
/**
 * Generación de números aleatorios.
 *
 * @author Luis José Sánchez
 */

public class Aleatorio05 {
    public static void main(String[] args) {

        System.out.println("20 números aleatorios entre 50 y 60 (sin decimales):");
```



```
for (int i = 1; i <= 20; i++) {  
    System.out.print(((int)(Math.random()*11) + 50 ) + " ");  
}  
  
System.out.println();  
}  
}
```

6.2 Generación de palabras de forma aleatoria de un conjunto dado

Hemos visto cómo generar números aleatorios con y sin decimales y en diferentes intervalos. Vamos a producir ahora de forma aleatoria una palabra - piedra, papel o tijera - generando primero un número entero entre 0 y 2 y posteriormente haciendo corresponder una palabra a cada número.

```
/**  
 * Generación de números aleatorios.  
 *  
 * @author Luis José Sánchez  
 */  
  
public class Aleatorio06 {  
  
    public static void main(String[] args) {  
  
        System.out.println("Genera al azar piedra, papel o tijera:");  
  
        int mano = (int)(Math.random()*3); // genera un número al azar  
                                           // entre 0 y 2 ambos incluidos  
  
        switch(mano) {  
            case 0:  
                System.out.println("piedra");  
                break;  
            case 1:  
                System.out.println("papel");  
                break;  
            case 2:  
                System.out.println("tijera");  
                break;  
            default:  
                break;  
        }  
    }  
}
```

¿Cómo podríamos generar un día de la semana de forma aleatoria? En efecto, primero generamos un número entre 1 y 7 ambos inclusive y luego hacemos corresponder un día de la semana a cada uno de los números.

```
/**
 * Generación de números aleatorios.
 *
 * @author Luis José Sánchez
 */

public class Aleatorio07 {
    public static void main(String[] args) {

        System.out.println("Muestra un día de la semana al azar:");

        int dia = (int)(Math.random()*7) + 1; // genera un número aleatorio
                                                // entre el 1 y el 7

        switch(dia) {
            case 1:
                System.out.println("lunes");
                break;
            case 2:
                System.out.println("martes");
                break;
            case 3:
                System.out.println("miércoles"); break;
            case 4:
                System.out.println("jueves");
                break;
            case 5:
                System.out.println("viernes");
                break;
            case 6:
                System.out.println("sábado");
                break;
            case 7:
                System.out.println("domingo");
                break;
            default:
        }
    }
}
```

6.3 Ejercicios



Ejercicio 1

Escribe un programa que muestre la tirada de tres dados. Se debe mostrar también la suma total (los puntos que suman entre los tres dados).



Ejercicio 2

Realiza un programa que muestre al azar el nombre de una carta de la baraja francesa. Esta baraja está dividida en cuatro palos: picas, corazones, diamantes y tréboles. Cada palo está formado por 13 cartas, de las cuales 9 cartas son numerales y 4 literales: 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K y A (que sería el 1). Para convertir un número en una cadena de caracteres podemos usar `String.valueOf(n)`.



Ejercicio 3

Igual que el ejercicio anterior pero con la baraja española. Se utilizará la baraja de 40 cartas: 2, 3, 4, 5, 6, 7, sota, caballo, rey y as.



Ejercicio 4

Muestra 20 números enteros aleatorios entre 0 y 10 (ambos incluidos) separados por espacios.



Ejercicio 5

Muestra 50 números enteros aleatorios entre 100 y 199 (ambos incluidos) separados por espacios. Muestra también el máximo, el mínimo y la media de esos números.



Ejercicio 6

Escribe un programa que piense un número al azar entre 0 y 100. El usuario debe adivinarlo y tiene para ello 5 oportunidades. Después de cada intento fallido, el programa dirá cuántas oportunidades quedan y si el número introducido es menor o mayor que el número secreto.



Ejercicio 7

Escribe un programa que muestre tres apuestas de la quiniela en tres columnas para los 14 partidos y el pleno al quince (15 filas).



Ejercicio 8

Modifica el programa anterior para que la probabilidad de que salga un 1 sea de $1/2$, la probabilidad de que salga x sea de $1/3$ y la probabilidad de que salga 2 sea de $1/6$. Pista: $1/2 = 3/6$ y $1/3 = 2/6$.



Ejercicio 9

Realiza un programa que vaya generando números aleatorios pares entre 0 y 100 y que no termine de generar números hasta que no saque el 24. El programa deberá decir al final cuántos números se han generado.



Ejercicio 10

Realiza un programa que pinte por pantalla diez líneas formadas por caracteres. El carácter con el que se pinta cada línea se elige de forma aleatoria entre uno de los siguientes: *, -, =, ., |, @. Las líneas deben tener una longitud aleatoria entre 1 y 40 caracteres.



Ejercicio 11

Escribe un programa que muestre 20 notas generadas al azar. Las notas deben aparecer de la forma: suspenso, suficiente, bien, notable o sobresaliente. Al final aparecerá el número de suspensos, el número de suficientes, el número de bienes, etc.



Ejercicio 12

Realiza un programa que llene la pantalla de caracteres aleatorios (a lo Matrix) con el código ascii entre el 32 y el 126. Puedes hacer casting con (`char`) para convertir un entero en un carácter.



Ejercicio 13

Escribe un programa que simule la tirada de dos dados. El programa deberá continuar tirando los dados una y otra vez hasta que en alguna tirada los dos dados tengan el mismo valor.



Ejercicio 14

Realiza un programa que haga justo lo contrario a lo que hace el ejercicio 6. El programa intentará adivinar el número que estás pensando - un número entre 0 y 100 - teniendo para ello 5 oportunidades. En cada intento fallido, el programa debe preguntar si el número que estás pensando es mayor o menor que el que te acaba de decir.

7. Arrays

7.1 Arrays de una dimensión

Un *array* es un tipo de dato capaz de almacenar múltiples valores. Se utiliza para agrupar datos muy parecidos, por ejemplo, si se necesita almacenar la temperatura media diaria en Málaga durante el último año se pueden utilizar las variables `temp0`, `temp1`, `temp2`, `temp3`, `temp4`, ... y así hasta 365 variables distintas pero sería poco práctico; es mejor utilizar un *array* de nombre `temp` y usar un índice para referenciar la temperatura de un día concreto del año.

En matemáticas, un *array* de una dimensión se llama vector. En este libro no utilizaremos este término para no confundirlo con la clase `Vector` de Java.

Veamos con un ejemplo cómo se crea y se utiliza un *array*.

```
/**
 * Ejemplo de uso de arrays
 *
 * @author Luis José Sánchez
 */

public class Array01 {
    public static void main(String[] args) {

        int[] n; // se define n como un array de enteros
        n = new int[4]; // se reserva espacio para 4 enteros

        n[0] = 26;
        n[1] = -30;
        n[2] = 0;
        n[3] = 100;

        System.out.print("Los valores del array n son los siguientes: ");
        System.out.print(n[0] + ", " + n[1] + ", " + n[2] + ", " + n[3]);

        int suma = n[0] + n[3];
        System.out.println("\nEl primer elemento del array más el último suman " + suma);
    }
}
```

Observa que el índice de cada elemento de un *array* se indica entre corchetes de tal forma que `n[3]` es el cuarto elemento ya que el primer índice es el 0.

Array n	
Índice	Valor
0	26
1	-30
2	0
3	11

Fíjate que la definición del *array* y la reserva de memoria para los cuatro elementos que la componen se ha realizado en dos líneas diferentes. Se pueden hacer ambas cosas en una sola línea como veremos en el siguiente ejemplo.

```
/**
 * Ejemplo de uso de arrays
 *
 * @author Luis José Sánchez
 */

public class Array02 {
    public static void main(String[] args) {

        // definición del array y reserva de memoria en la misma línea
        int[] x = new int[5];

        x[0] = 8;
        x[1] = 33;
        x[2] = 200;
        x[3] = 150;
        x[4] = 11;

        System.out.println("El array x tiene 5 elementos ¿cuál de ellos quiere ver?");
        System.out.print("Introduzca un número del 0 al 4: ");
        int indice = Integer.parseInt(System.console().readLine());
        System.out.print("El elemento que se encuentra en la posición " + indice);
        System.out.println(" es el " + x[indice]);
    }
}
```

El *array* *x* del ejemplo anterior se ha ido rellenando elemento a elemento. Si se conocen previamente todos los valores iniciales del array, se puede crear e inicializar en una sola línea de la siguiente forma

```
int[] x = {8, 33, 200, 150, 11};
```

Cada elemento del *array* se puede utilizar exactamente igual que cualquier otra variable, es decir, se le puede asignar un valor o se puede usar dentro de una expresión. En el siguiente ejemplo se muestran varias operaciones en las que los operandos son elementos del *array* `num`.

Para recorrer todos los elementos de un *array* se suele utilizar un bucle `for` junto con un índice que va desde 0 hasta el tamaño del *array* menos 1.

```
/**
 * Ejemplo de uso de arrays
 *
 * @author Luis José Sánchez
 */

public class Array03 {
    public static void main(String[] args) {

        int[] num = new int[10];

        num[0] = 8;
        num[1] = 33;
        num[2] = 200;
        num[3] = 150;
        num[4] = 11;
        num[5] = 88;
        num[6] = num[2] * 10;
        num[7] = num[2] / 10;
        num[8] = num[0] + num[1] + num[2];
        num[9] = num[8];

        System.out.println("El array num contiene los siguientes elementos:");

        for (int i = 0; i < 10; i++) {
            System.out.println(num[i]);
        }
    }
}
```

En Java, a diferencia de otros lenguajes como Ruby o PHP, todos los elementos de un *array* deben ser del mismo tipo; por ejemplo, no puede haber un entero en la posición 2 y una cadena de caracteres en la posición 7 del mismo *array*. En el siguiente ejemplo se muestra un *array* de caracteres.

```
/**
 * Ejemplo de uso de arrays
 *
 * @author Luis José Sánchez
 */

public class Array04 {
    public static void main(String[] args) {

        char[] character = new char[6];

        character[0] = 'R';
        character[1] = '%';
        character[2] = '&';
        character[3] = '+';
        character[4] = 'A';
        character[5] = '2';

        System.out.println("El array character contiene los siguientes elementos:");

        for (int i = 0; i < 6; i++) {
            System.out.println(character[i]);
        }
    }
}
```

En el siguiente ejemplo se muestra un *array* de números de tipo `double` que almacena notas de alumnos.

```
/**
 * Ejemplo de uso de arrays
 *
 * @author Luis José Sánchez
 */

public class Array05 {

    public static void main(String[] args) {

        double[] nota = new double[4];

        System.out.println("Para calcular la nota media necesito saber la ");
        System.out.println("nota de cada uno de tus exámenes.");

        for (int i = 0; i < 4; i++) {
```



```

        System.out.print("Nota del examen nº " + (i + 1) + ": ");
        nota[i] = Double.parseDouble(System.console().readLine());
    }

    System.out.println("Tus notas son: ");

    double suma = 0;

    for (int i = 0; i < 4; i++) {
        System.out.print(nota[i] + " ");
        suma += nota[i];
    }

    System.out.println("\nLa media es " + suma / 4);
}

```

7.2 Arrays bidimensionales

Un *array* bidimensional utiliza dos índices para localizar cada elemento. Podemos ver este tipo de dato como un *array* que, a su vez, contiene otros *arrays*. También se puede ver como una cuadrícula en la que los datos quedan distribuidos en filas y columnas.

En el siguiente ejemplo se muestra la definición y el uso de un *array* de dos dimensiones.

```

/**
 * @author Luis José Sánchez
 *
 * Ejemplo de uso de arrays bidimensionales
 */

public class ArrayBi01 {
    public static void main(String[] args)
        throws InterruptedException { // Se añade esta línea para poder usar sleep

        int[][] n = new int[3][2]; // array de 3 filas por 2 columnas

        n[0][0]=20;
        n[1][0]=67;
        n[1][1]=33;
        n[2][1]=7;

        int fila, columna;
    }
}

```

```

    for(fila = 0; fila < 3; fila++) {

        System.out.print("Fila: " + fila);

        for(columna = 0; columna < 2; columna++) {
            System.out.printf("%10d ", n[fila][columna]);
            Thread.sleep(1000); // retardo de un segundo
        }
        System.out.println();
    }
}

```

Mediante la línea `int[][] n = new int[3][2]` se define un *array* bidimensional de 3 filas por 2 columnas, pero bien podrían ser 2 filas por 3 columnas, según el objetivo y el uso que del *array* haga el programador.

Los valores del *array* bidimensional se pueden proporcionar en la misma línea de la definición como se muestra en el siguiente ejemplo.

```

/**
 * @author Luis José Sánchez
 *
 * Ejemplo de uso de arrays bidimensionales
 */

public class ArrayBi02 {
    public static void main(String[] args)
        throws InterruptedException { // Se añade esta línea para poder usar sleep

        int fila, columna;
        int[][] n = {{20, 4}, {67, 33}, {0,7}};

        for(fila = 0; fila < 3; fila++) {

            System.out.print("Fila: " + fila);

            for(columna = 0; columna < 2; columna++) {
                System.out.printf("%10d ", n[fila][columna]);
                Thread.sleep(1000); // retardo de un segundo
            }
            System.out.println();
        }
    }
}

```

Los *arrays* bidimensionales se utilizan con frecuencia para situar objetos en un plano como por ejemplo las piezas de ajedrez en un tablero, o un personaje de video-juego en un laberinto.

En el siguiente programa se colocan una mina y un tesoro de forma aleatoria en un cuadrante de cuatro filas por cinco columnas. El usuario intentará averiguar dónde está el tesoro indicando las coordenadas (x, y).

```
/**
 * Minijuego "Busca el tesoro"
 *
 * Se colocan una mina y un tesoro de forma aleatoria en un cuadrante de
 * cuatro filas por cinco columnas. El usuario intentará averiguar dónde
 * está el tesoro.
 *
 * @author Luis José Sánchez
 */

public class BuscaTesoro {

    // se definen constantes para representar el
    // contenido de las celdas
    static final int VACIO = 0;
    static final int MINA = 1;
    static final int TESORO = 2;
    static final int INTENTO = 3;

    public static void main(String[] args) {

        int x;
        int y;
        int[][] cuadrante = new int[5][4];

        // inicializa el array
        for(x = 0; x < 4; x++) {
            for(y = 0; y < 3; y++) {
                cuadrante[x][y] = VACIO;
            }
        }

        // coloca la mina
        int minaX = (int)(Math.random()*4);
        int minaY = (int)(Math.random()*3);
        cuadrante[minaX][minaY] = MINA;

        // coloca el tesoro
        int tesoroX;
```

```

    int tesoroY;
    do {
        tesoroX = (int)(Math.random()*4);
        tesoroY = (int)(Math.random()*3);
    } while ((minaX == tesoroX) && (minaY == tesoroY));
    cuadrante[tesoroX][tesoroY] = TESORO;

    // juego
    System.out.println("¡BUSCA EL TESORO!");
    boolean salir = false;
    String c = "";
    do {
        // pinta el cuadrante
        for(y = 3; y >= 0; y--) {
            System.out.print(y + "|");
            for(x = 0; x < 5; x++) {
                if (cuadrante[x][y] == INTENTO)
                    System.out.print("X ");
                else
                    System.out.print("  ");
            }
            System.out.println();
        }
        System.out.println("  -----\n  0 1 2 3 4\n");

        // pide las coordenadas
        System.out.print("Coordenada x: ");
        x = Integer.parseInt(System.console().readLine());
        System.out.print("Coordenada y: ");
        y = Integer.parseInt(System.console().readLine());

        // mira lo que hay en las coordenadas indicadas por el usuario
        switch(cuadrante[x][y]) {
            case VACIO:
                cuadrante[x][y] = INTENTO;
                break;
            case MINA:
                System.out.println("Lo siento, has perdido.");
                salir = true;
                break;
            case TESORO:
                System.out.println("Enhorabuena, has encontrado el tesoro.");
                salir = true;
                break;
            default:
        }
    }

```

```

    } while (!salir);

    // pinta el cuadrante
    for(y = 3; y >= 0; y--) {
        System.out.print(y + " ");
        for(x = 0; x < 5; x++) {
            switch(cuadrante[x][y]) {
                case VACIO:
                    c = " ";
                    break;
                case MINA:
                    c = "* ";
                    break;
                case TESORO:
                    c = "€ ";
                    break;
                case INTENTO:
                    c = "x ";
                    break;
                default:
            }
            System.out.print(c);
        }
        System.out.println();
    }
    System.out.println(" -----\n  0 1 2 3 4\n");
}
}

```

7.3 Recorrer arrays con for al estilo foreach

Al trabajar con *arrays* es muy frecuente cometer errores utilizando los índices. El error más típico consiste en intentar acceder a un elemento mediante un índice que se sale de los límites. Por ejemplo, si tenemos el *array* *n* definido de la siguiente forma `int[] n = new int[10]`, cuando intentamos acceder a `n[-1]` o a `n[10]` obtenemos un error en tiempo de ejecución.

Para recorrer un *array* de un modo más práctico y sencillo, sin que tengamos que preocuparnos de los límites, podemos utilizar el bucle `for` con el formato `foreach`. De esta forma indicamos simplemente el nombre del *array* que queremos recorrer y en qué variable se va a ir colocando cada elemento con cada iteración del bucle. No hay que especificar con qué índice comienza y termina el bucle, de eso se encarga Java.

A continuación se muestra el ejemplo `Array05.java` visto anteriormente pero, esta vez, utilizando el `for` a la manera `foreach`.

```
/**
 * Recorre un array con un for al estilo foreach.
 *
 * @author Luis José Sánchez
 */

public class ArrayForEach {
    public static void main(String[] args) {

        double[] nota = new double[4];

        System.out.println("Para calcular la nota media necesito saber la ");
        System.out.println("nota de cada uno de tus exámenes.");

        for (int i = 0; i < 4; i++) {
            System.out.print("Nota del examen nº " + (i + 1) + ": ");
            nota[i] = Double.parseDouble(System.console().readLine());
        }

        System.out.println("Tus notas son: ");

        double suma = 0;

        for (double n : nota) { // for al estilo foreach
            System.out.print(n + " ");
            suma += n;
        }

        System.out.println("\nLa media es " + suma / 4);
    }
}
```

Fíjate en el segundo for; en este caso no se utiliza ningún índice; simplemente decimos “ve sacando uno a uno los elementos del array nota y deposita cada uno de esos elementos en la variable n que es de tipo double”.

7.4 Ejercicios

7.4.1 Arrays de una dimensión



Ejercicio 1

Define un *array* de 12 números enteros con nombre `num` y asigna los valores según la tabla que se muestra a continuación. Muestra el contenido de todos los elementos del *array*. ¿Qué sucede con los valores de los elementos que no han sido inicializados?

Índice	0	1	2	3	4	5	6	7	8	9	10	11
Valor	39	-2			0		14		5	120		



Ejercicio 2

Define un *array* de 10 caracteres con nombre `simbolo` y asigna valores a los elementos según la tabla que se muestra a continuación. Muestra el contenido de todos los elementos del *array*. ¿Qué sucede con los valores de los elementos que no han sido inicializados?

Índice	0	1	2	3	4	5	6	7	8	9
Valor	'a'	'x'			'@'		' '	'+'	'Q'	



Ejercicio 3

Escribe un programa que lea 10 números por teclado y que luego los muestre en orden inverso, es decir, el primero que se introduce es el último en mostrarse y viceversa.



Ejercicio 4

Define tres *arrays* de 20 números enteros cada una, con nombres `numero`, `cuadrado` y `cubo`. Carga el *array* `numero` con valores aleatorios entre 0 y 100. En el *array* `cuadrado` se deben almacenar los cuadrados de los valores que hay en el *array* `numero`. En el *array* `cubo` se deben almacenar los cubos de los valores que hay en `numero`. A continuación, muestra el contenido de los tres *arrays* dispuesto en tres columnas.



Ejercicio 5

Escribe un programa que pida 10 números por teclado y que luego muestre los números introducidos junto con las palabras “máximo” y “mínimo” al lado del máximo y del mínimo respectivamente.



Ejercicio 6

Escribe un programa que lea 15 números por teclado y que los almacene en un *array*. Rota los elementos de ese *array*, es decir, el elemento de la posición 0 debe pasar a la posición 1, el de la 1 a la 2, etc. El número que se encuentra en la última posición debe pasar a la posición 0. Finalmente, muestra el contenido del *array*.



Ejercicio 7

Escribe un programa que genere 100 números aleatorios del 0 al 20 y que los muestre por pantalla separados por espacios. El programa pedirá entonces por teclado dos valores y a continuación cambiará todas las ocurrencias del primer valor por el segundo en la lista generada anteriormente. Los números que se han cambiado deben aparecer entrecomillados.



Ejercicio 8

Realiza un programa que pida la temperatura media que ha hecho en cada mes de un determinado año y que muestre a continuación un diagrama de barras horizontales con esos datos. Las barras del diagrama se pueden dibujar a base de asteriscos o cualquier otro carácter.



Ejercicio 9

Realiza un programa que pida 8 números enteros y que luego muestre esos números junto con la palabra “par” o “impar” según proceda.



Ejercicio 10

Escribe un programa que genere 20 números enteros aleatorios entre 0 y 100 y que los almacene en un *array*. El programa debe ser capaz de pasar todos los números pares a las primeras posiciones del *array* (del 0 en adelante) y todos los números impares a las celdas restantes. Utiliza *arrays* auxiliares si es necesario.



Ejercicio 11

Realiza un programa que pida 10 números por teclado y que los almacene en un *array*. A continuación se mostrará el contenido de ese *array* junto al índice (0 – 9) utilizando para ello una tabla. Seguidamente el programa pasará los primos a las primeras posiciones, desplazando el resto de números (los que no son primos) de tal forma que no se pierda ninguno. Al final se debe mostrar el *array* resultante.

Por ejemplo:

Índice	0	1	2	3	4	5	6	7	8	9
Valor	20	5	7	4	32	9	2	14	11	6

Array inicial

Índice	0	1	2	3	4	5	6	7	8	9
Valor	5	7	2	11	20	4	32	9	14	6

Array final



Ejercicio 12

Realiza un programa que pida 10 números por teclado y que los almacene en un *array*. A continuación se mostrará el contenido de ese *array* junto al índice (0 – 9). Seguidamente el programa pedirá dos posiciones a las que llamaremos “inicial” y “final”. Se debe comprobar que inicial es menor que final y que ambos números están entre 0 y 9. El programa deberá colocar el número de la posición inicial en la posición final, rotando el resto de números para que no se pierda ninguno. Al final se debe mostrar el *array* resultante.

Por ejemplo, para inicial = 3 y final = 7:

Índice	0	1	2	3	4	5	6	7	8	9
Valor	20	5	7	4	32	9	2	14	11	6

Array inicial

Índice	0	1	2	3	4	5	6	7	8	9
Valor	6	20	5	7	32	9	2	4	14	11

Array final

7.4.2 Arrays bidimensionales



Ejercicio 1

Define un *array* de números enteros de 3 filas por 6 columnas con nombre *num* y asigna los valores según la siguiente tabla. Muestra el contenido de todos los elementos del *array* dispuestos en forma de tabla como se muestra en la figura.

Array num	Columna 0	Columna 1	Columna 2	Columna 3	Columna 4	Columna 5
Fila 0	0	30	2			5
Fila 0	75				0	
Fila 0			-2	9		11



Ejercicio 2

Escribe un programa que pida 20 números enteros. Estos números se deben introducir en un *array* de 4 filas por 5 columnas. El programa mostrará las sumas parciales de filas y columnas igual que si de una hoja de cálculo se tratara. La suma total debe aparecer en la esquina inferior derecha.

					Σ fila 0
					Σ fila 0
					Σ fila 0
					Σ fila 0
Σ columna 0	Σ columna 1	Σ columna 2	Σ columna 3	Σ columna 4	TOTAL



Ejercicio 3

Modifica el programa anterior de tal forma que los números que se introducen en el *array* se generen de forma aleatoria (valores entre 100 y 999).



Ejercicio 4

Modifica el programa anterior de tal forma que las sumas parciales y la suma total aparezcan en la pantalla con un pequeño retardo, dando la impresión de que el ordenador se queda “pensando” antes de mostrar los números.



Ejercicio 5

Realiza un programa que rellene un *array* de 6 filas por 10 columnas con números enteros positivos comprendidos entre 0 y 1000 (ambos incluidos). A continuación, el programa deberá dar la posición tanto del máximo como del mínimo.



Ejercicio 6

Modifica el programa anterior de tal forma que no se repita ningún número en el *array*.



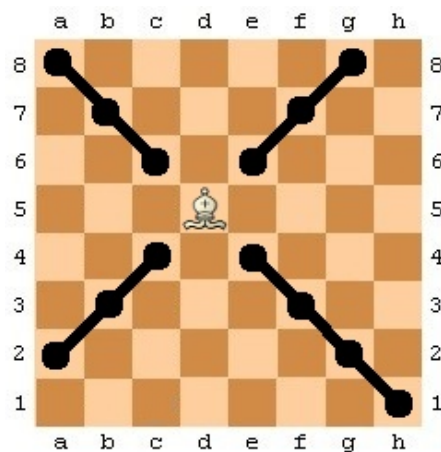
Ejercicio 7

Mejora el juego “Busca el tesoro” de tal forma que si hay una mina a una casilla de distancia, el programa avise diciendo ¡Cuidado! ¡Hay una mina cerca!



Ejercicio 8

Escribe un programa que, dada una posición en un tablero de ajedrez, nos diga a qué casillas podría saltar un alfil que se encuentra en esa posición. Como se indica en la figura, el alfil se mueve siempre en diagonal. El tablero cuenta con 64 casillas. Las columnas se indican con las letras de la “a” a la “h” y las filas se indican del 1 al 8.



Ejemplo:

Introduzca la posición del alfil: d5

El alfil puede moverse a las siguientes posiciones:

h1 a2 g2 b3 e3 c4 e4 c6 e6 b7 f7 a8 g8



Ejercicio 9

Realiza un programa que sea capaz de rotar todos los elementos de una matriz cuadrada una posición en el sentido de las agujas del reloj. La matriz debe tener 12 filas por 12 columnas y debe contener números generados al azar entre 0 y 100. Se debe mostrar tanto la matriz original como la matriz resultado, ambas con los números convenientemente alineados.



Ejercicio 10

Realiza el juego de las tres en raya.

8. Funciones

8.1 Implementando funciones para reutilizar código

En programación es muy frecuente reutilizar código, es decir, usar código ya existente. Cuando una parte de un programa requiere una funcionalidad que ya está implementada en otro programa no tiene mucho sentido emplear tiempo y energía en implementarla otra vez.

Una función es un trozo de código que realiza una tarea muy concreta y que se puede incluir en cualquier programa cuando hace falta resolver esa tarea. Opcionalmente, las funciones aceptan una entrada (parámetros de entrada) y devuelven una salida.

Observa el siguiente ejemplo. Se trata de un programa que pide un número por teclado y luego dice si el número introducido es o no es primo.

```
/**
 * Dice si un número es o no es primo (sin funciones)
 *
 * @author Luis José Sánchez
 */

public class NumeroPrimo {
    public static void main(String[] args) {

        System.out.print("Introduce un número entero positivo: ");
        int n = Integer.parseInt(System.console().readLine());

        boolean esPrimo = true;
        for (int i = 2; i < n; i++) {
            if ((n % i) == 0) {
                esPrimo = false;
            }
        }

        if (esPrimo) {
            System.out.println("El " + n + " es primo.");
        } else {
            System.out.println("El " + n + " no es primo.");
        }
    }
}
```

Podemos intuir que la tarea de averiguar si un número es o no primo será algo que utilizaremos con frecuencia más adelante así que podemos aislar el trozo de código que realiza ese cometido para usarlo con comodidad en otros programas.

```
/**
 * Dice si un número es o no es primo usando una función
 *
 * @author Luis José Sánchez
 */

public class NumeroPrimoConFuncion {

    // Programa principal //////////////////////////////////////

    public static void main(String[] args) {

        System.out.print("Introduzca un número entero positivo: ");
        int n = Integer.parseInt(System.console().readLine());

        if (esPrimo(n)) {
            System.out.println("El " + n + " es primo.");
        } else {
            System.out.println("El " + n + " no es primo.");
        }
    }

    // Funciones //////////////////////////////////////

    /**
     * Comprueba si un número entero positivo es primo o no.
     * Un número es primo cuando únicamente es divisible entre
     * él mismo y la unidad.
     *
     * @param x un número entero positivo
     * @return <code>true</code> si el número es primo
     * @return <code>false</code> en caso contrario
     */
    public static boolean esPrimo(int x) {

        for (int i = 2; i < x; i++) {
            if ((x % i) == 0) {
                return false;
            }
        }
    }
}
```

```
    return true;
}
}
```

Cada función tiene una cabecera y un cuerpo. En el ejemplo anterior la cabecera es

```
public static boolean esPrimo(int x)
```

De momento no vamos a explicar qué significa `public static`, lo dejaremos para cuando veamos el capítulo 9 “Programación orientada a objetos”, por ahora lo escribiremos tal cual cada vez que definamos una función. A continuación se escribe el tipo de dato que devuelve la función, en este caso es `boolean` porque la función devolverá siempre `true` (verdadero) o `false` falso. Lo último que lleva la cabecera son los parámetros encerrados entre paréntesis. Esos parámetros son los valores que se le pasan a la función para que realice los cálculos. En este caso concreto, el parámetro que se pasa es `x`, o sea, el número que queremos saber si es primo o no. Es necesario indicar siempre el tipo de cada parámetro; en esta ocasión, el parámetro que se pasa es de tipo entero (`int`).

8.2 Comentarios de funciones

Los comentarios tienen un papel muy importante en los programas ya que, aunque no se compilan ni se ejecutan, permiten describir, aclarar o dar información relevante sobre qué hace exactamente el código. Un programa bien comentado es mucho más fácil de corregir, mantener y mejorar que un programa mal comentado o que simplemente no tiene comentarios.

Existe una herramienta llamada `javadoc` que crea unos documentos en HTML con información sobre programas escritos en Java. Para ello, `javadoc` utiliza los comentarios que se han hecho y, además, la meta-información que se incluye en esos comentarios. Esta meta-información es muy fácil de distinguir pues va precedida siempre de un carácter `@`.

Te habrás dado cuenta que en los comentarios de los programas que hemos visto en este manual, se indica el nombre del autor mediante `@author`.

Si los comentarios de los programas en general son importantes, debemos prestar especial atención a los comentarios de las funciones ya que posiblemente serán usadas por otros programadores que querrán saber exactamente cómo se utilizan.

Con la etiqueta `@param` seguida de un nombre de parámetro se indica qué parámetro espera como entrada la función. Si una función acepta varios parámetros, se especificarán varias etiquetas `@param` en los comentarios.

Mediante la etiqueta `@return` especificamos qué devuelve exactamente la función. Aunque el tipo de dato que se devuelve ya viene indicado en la cabecera de la función, mediante la etiqueta `@return` podemos explicarlo con más detalle.

En los comentarios se pueden incluir elementos de HTML como `<code>`, ` `, etc. ya que los entiende perfectamente javadoc.

Puedes hacer una prueba generando la documentación del ejemplo ejecutando la siguiente línea en una ventana de terminal. Como tenemos tildes en los comentarios, si queremos que salgan bien en las páginas de documentación, debemos indicar la codificación de caracteres.

```
javadoc -encoding UTF-8 -charset UTF-8 -docencoding UTF-8 NumeroPrimoConFuncion.java
```

Para aprender más sobre javadoc puedes consultar [la documentación oficial de Oracle](http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html)¹.

8.3 Creación de bibliotecas de rutinas mediante paquetes

Si la función `esPrimo()` va a ser usada en tres programas diferentes se puede copiar y pegar su código en cada uno de los programas, pero hay una solución mucho más elegante y práctica.

Las funciones de un determinado tipo (por ejemplo funciones matemáticas) se pueden agrupar para crear un paquete (package) que luego se importará desde el programa que necesite esas funciones.

Cada paquete se corresponde con un directorio. Por tanto, si hay un paquete con nombre `matematicas` debe haber un directorio llamado también `matematicas` en la misma ubicación del programa que importa ese paquete (normalmente el programa principal).

Las funciones se pueden agrupar dentro de un paquete de dos maneras diferentes. Puede haber subpaquetes dentro de un paquete; por ejemplo, si quisiéramos dividir las funciones matemáticas en funciones relativas al cálculo de áreas y volúmenes de figuras geométricas y funciones relacionadas con cálculos estadísticos, podríamos crear dos directorios dentro de `matematicas` con nombres `geometria` y `estadistica` respectivamente. Estos subpaquetes se llamarían `matematicas.geometria` y `matematicas.estadistica`. Otra manera de agrupar las funciones dentro de un mismo paquete consiste en crear varios ficheros dentro de un mismo directorio. En este caso se podrían crear los ficheros `Geometria.java` y `Estadistica.java`.

Entenderemos mejor todos estos conceptos con un ejemplo completo. Vamos a crear un paquete con nombre `matematicas` que contenga dos clases: `Varias` (para funciones matemáticas de propósito general) y `Geometria`. Por tanto en el disco duro, tendremos una carpeta con nombre `matematicas` que contiene los ficheros `Varias.java` y `Geometria.java`. El contenido de estos ficheros se muestra a continuación.

¹<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>


```
/**
 * Funciones matemáticas de propósito general
 *
 * @author Luis José Sánchez
 */

package matematicas;

public class Varias {

    /**
     * Comprueba si un número entero positivo es primo o no.
     * Un número es primo cuando únicamente es divisible entre
     * él mismo y la unidad.
     *
     * @param x un número entero positivo
     * @return <code>true</code> si el número es primo
     * @return <code>false</code> en caso contrario
     */
    public static boolean esPrimo(int x) {

        for (int i = 2; i < x; i++) {
            if ((x % i) == 0) {
                return false;
            }
        }

        return true;
    }

    /**
     * Devuelve el número de dígitos que contiene un número entero
     *
     * @param x un número entero
     * @return la cantidad de dígitos que contiene el número
     */
    public static int digitos(int x) {

        if (x == 0) {
            return 1;
        } else {
            int n = 0;
            while (x > 0) {
                x = x / 10;
                n++;
            }
        }
    }
}
```

```

    }
    return n;
}
}
}

```

Se incluye a continuación `Geometría.java`. Recuerda que tanto `Varías.java` como `Geometría.java` se encuentran dentro del directorio `matematicas`.

```

/**
 * Funciones geométricas
 *
 * @author Luis José Sánchez
 */

package matematicas;

public class Geometria {

    /**
     * Calcula el volumen de un cilindro.
     * Tanto el radio como la altura se deben proporcionar en las
     * mismas unidades para que el resultado sea congruente.
     *
     * @param r radio del cilindro
     * @param h altura del cilindro
     * @return volumen del cilindro
     */
    public static double volumenCilindro(double r, double h) {
        return Math.PI * r * r * h;
    }

    /**
     * Calcula la longitud de una circunferencia a partir del radio.
     *
     * @param r radio de la circunferencia
     * @return longitud de la circunferencia
     */
    public static double longitudCircunferencia(double r) {
        return 2 * Math.PI * r;
    }
}

```

Observa que en ambos ficheros se especifica que las clases declaradas (y por tanto las funciones que se definen dentro) pertenecen al paquete `matematicas` mediante la línea `package matematicas`. Aho-

ra ya podemos probar las funciones desde un programa externo. El programa `PruebaFunciones.java` está fuera de la carpeta `matematicas`, justo en un nivel superior en la estructura de directorios.

```
/**
 * Dice si un número es o no es primo usando una función
 *
 * @author Luis José Sánchez
 */

import matematicas.Varias;
import matematicas.Geometria;

public class PruebaFunciones {
    public static void main(String[] args) {

        int n;

        // Prueba esPrimo()

        System.out.print("Introduzca un número entero positivo: ");
        n = Integer.parseInt(System.console().readLine());

        if (matematicas.Varias.esPrimo(n)) {
            System.out.println("El " + n + " es primo.");
        } else {
            System.out.println("El " + n + " no es primo.");
        }

        // Prueba digitos()

        System.out.print("Introduzca un número entero positivo: ");
        n = Integer.parseInt(System.console().readLine());

        System.out.println(n + " tiene " + matematicas.Varias.digitos(n) + " dígitos.");

        // Prueba volumenCilindro()

        double r, h;

        System.out.println("Cálculo del volumen de un cilindro");
        System.out.print("Introduzca el radio en metros: ");
        r = Double.parseDouble(System.console().readLine());
        System.out.print("Introduzca la altura en metros: ");
        h = Double.parseDouble(System.console().readLine());
```

```
        System.out.println("El volumen del cilindro es " + matematicas.Geometria.volumenCilind\
ro(r, h) + " m3");
    }
}
```

Las líneas

```
import matematicas.Varias;
import matematicas.Geometria;
```

cargan las clases contenidas en el paquete `matematicas` y, por tanto, todas las funciones contenidas en ellas. No vamos a utilizar el comodín de la forma `import matematicas.*`; ya que está desaconsejado su uso en el estándar de codificación en Java de Google.

El uso de una función es muy sencillo; hay que indicar el nombre del paquete, el nombre de la clase y finalmente el nombre de la función con los parámetros adecuados. Por ejemplo, como vemos en el programa anterior, `matematicas.Varias.digitos(n)` devuelve el número de dígitos de `n`; siendo `matematicas` el paquete, `Varias` la clase, `digitos` la función y `n` el parámetro que se le pasa a la función.

8.4 Ámbito de las variables

El ámbito de una variable es el espacio donde “existe” esa variable o, dicho de otro modo, el contexto dentro del cual la variable es válida.

Seguramente has utilizado muchas veces variables con nombres como `x`, `i`, `max` o `aux`. El ámbito de cada una de esas variables era el programa en el que estaban declaradas. Fíjate que la `x` del primer programa que aparece en este libro no interfiere para nada con la `x` del siguiente programa que usa una variable con ese mismo nombre. Aunque se llamen igual son variables diferentes.

Cuando se implementan funciones hay que tener muy claro que las variables utilizadas como parámetros (por valor) o las variables que se definen dentro de la función son locales a esa función, es decir, su ámbito es la función y fuera de ella esas variables no existen. Presta atención al programa `Varios.java` y observa que las dos funciones que hay definidas dentro de la clase utilizan sendas variables con nombre `x`. Pues bien, cada una de esas variables son completamente independientes, se trata de dos variables distintas que tienen el mismo nombre pero que son válidas cada una en su propio ámbito.

8.5 Paso de parámetros por valor y por referencia

En Java, como en la mayoría de lenguajes de programación existen dos maneras de pasar parámetros: por valor y por referencia.

Cuando se pasa un parámetro por valor, en realidad se pasa una copia de la variable, únicamente importa el valor. Cualquier modificación que se le haga a la variable que se pasa como parámetro dentro de la función no tendrá ningún efecto fuera de la misma. Veamos un ejemplo.

```
/**
 * Paso de parámetros por valor
 *
 * @author Luis José Sánchez
 */
public class PruebaParametros1 {
    public static void main(String[] args) {

        int n = 10;

        System.out.println(n);
        calcula(n);
        System.out.println(n);
    }

    public static void calcula(int x) {
        x += 24;
        System.out.println(x);
    }
}
```

La salida del programa anterior es la siguiente:

```
10
34
10
```

A pesar de que la variable que se pasa como parámetro se modifica dentro de la función, los cambios no tienen ningún efecto en el programa principal.

Cuando se pasa un parámetro por referencia, por el contrario, si se modifica su valor dentro de la función, los cambios se mantienen una vez que la función ha terminado de ejecutarse.

En la mayoría de los lenguajes de programación es el programador quien decide cuándo un parámetro se pasa por valor y cuándo se pasa por referencia. En Java no podemos elegir. Todos los parámetros que son de tipo `int`, `double`, `float`, `char` o `String` se pasan siempre por valor mientras que los *arrays* se pasan siempre por referencia. Esto quiere decir que cualquier cambio que efectuemos en un array que se pasa como parámetro permanece cuando termina la ejecución de la función, por lo que hay que tener especial cuidado en estos casos.

```
/**
 * Paso de un array como parámetro
 *
 * @author Luis José Sánchez
 */
public class PruebaParametrosArray {
    public static void main(String[] args) {

        int n[] = {8, 33, 200, 150, 11};
        int m[] = new int[5];

        muestraArray(n);
        incrementa(n);
        muestraArray(n);
    }

    public static void muestraArray(int x[]) {
        for (int i = 0; i < x.length; i++) {
            System.out.print(x[i] + " ");
        }
        System.out.println();
    }

    public static void incrementa(int x[]) {
        for (int i = 0; i < x.length; i++) {
            x[i]++;
        }
    }
}
```

La salida del programa anterior es la siguiente:

```
8 33 200 150 11
9 34 201 151 12
```

Comprobamos, por tanto, que el array que se ha pasado como parámetro se ha modificado en la función y se han conservado los cambios en el programa principal.

Por último, se muestra un programa que contiene funciones que operan con arrays de dos dimensiones.

```
/**
 * Paso de un array bidimensional como parámetro
 *
 * @author Luis José Sánchez
 */

import matematicas.Varias;

public class ArrayBiFunciones {
    public static void main(String[] args) {

        int[][] n = new int[6][9];

        for(int i = 0; i < 6; i++) {
            for(int j = 0; j < 9; j++) {
                n[i][j] = (int)(Math.random()*100000);
            }
        }

        muestraArrayIntBi(n);
    }

    // Funciones //////////////////////////////////////

    /**
     * Devuelve el número de filas de un array bidimensional de
     * números enteros.
     *
     * @param x un array bidimensional de números enteros
     * @return número de filas del array
     */
    public static int filasArrayIntBi(int x[][]) {
        return x.length;
    }

    /**
     * Devuelve el número de columnas de un array bidimensional
     * de números enteros.
     *
     * @param x un array bidimensional de números enteros
     * @return número de columnas del array
     */
    public static int columnasArrayIntBi(int x[][]) {
```

```
    return x[0].length;
}

/**
 * Devuelve el máximo de un array bidimensional
 * de números enteros.
 *
 * @param x un array bidimensional de números enteros
 * @return el valor máximo encontrado en el array
 */
public static int maximoArrayIntBi(int x[][]) {

    int maximo = Integer.MIN_VALUE;

    for (int f = 0; f < filasArrayIntBi(x); f++) {
        for (int c = 0; c < columnasArrayIntBi(x); c++) {
            if (x[f][c] > maximo) {
                maximo = x[f][c];
            }
        }
    }

    return maximo;
}

/**
 * Muestra por pantalla el contenido de un array bidimensional
 * de números enteros.
 *
 * @param x un array bidimensional de números enteros
 */
public static void muestraArrayIntBi(int x[][]) {

    String formatoNumero = "%" + matematicas.Varias.digitos(maximoArrayIntBi(x)) + "d";

    for (int f = 0; f < filasArrayIntBi(x); f++) {
        for (int c = 0; c < columnasArrayIntBi(x); c++) {
            System.out.printf(formatoNumero + " ", x[f][c]);
        }
        System.out.println();
    }
}
```


8.6 Ejercicios



Ejercicios 1-14

Crea una biblioteca de funciones matemáticas que contenga las siguientes funciones. Recuerda que puedes usar unas dentro de otras si es necesario.

1. **esCapicua**: Devuelve verdadero si el número que se pasa como parámetro es capicúa y falso en caso contrario.
2. **esPrimo**: Devuelve verdadero si el número que se pasa como parámetro es primo y falso en caso contrario.
3. **siguientePrimo**: Devuelve el menor primo que es mayor al número que se pasa como parámetro.
4. **potencia**: Dada una base y un exponente devuelve la potencia.
5. **digitos**: Cuenta el número de dígitos de un número entero.
6. **voltea**: Le da la vuelta a un número.
7. **digitoN**: Devuelve el dígito que está en la posición n de un número entero. Se empieza contando por el 0 y de izquierda a derecha.
8. **posicionDeDigito**: Da la posición de la primera ocurrencia de un dígito dentro de un número entero. Si no se encuentra, devuelve -1.
9. **quitaPorDetras**: Le quita a un número n dígitos por detrás (por la derecha).
10. **quitaPorDelante**: Le quita a un número n dígitos por delante (por la izquierda).
11. **pegaPorDetras**: Añade un dígito a un número por detrás.
12. **pegaPorDelante**: Añade un dígito a un número por delante.
13. **trozoDeNumero**: Toma como parámetros las posiciones inicial y final dentro de un número y devuelve el trozo correspondiente.
14. **juntaNumeros**: Pega dos números para formar uno.



Ejercicio 15

Muestra los números primos que hay entre 1 y 1000.



Ejercicio 16

Muestra los números capicúa que hay entre 1 y 99999.



Ejercicio 17

Escribe un programa que pase de binario a decimal.



Ejercicio 18

Escribe un programa que pase de decimal a binario.



Ejercicio 19

Une y amplía los dos programas anteriores de tal forma que se permita convertir un número entre cualquiera de las siguientes bases: decimal, binario, hexadecimal y octal.



Ejercicios 20-28

Crea una biblioteca de funciones para arrays (de una dimensión) de números enteros que contenga las siguientes funciones:

1. **generaArrayInt**: Genera un array de tamaño *n* con números aleatorios cuyo intervalo (mínimo y máximo) se indica como parámetro.
2. **minimoArrayInt**: Devuelve el mínimo del array que se pasa como parámetro.
3. **maximoArrayInt**: Devuelve el máximo del array que se pasa como parámetro.
4. **mediaArrayInt**: Devuelve la media del array que se pasa como parámetro.
5. **estaEnArrayInt**: Dice si un número está o no dentro de un array.
6. **posicionEnArray**: Busca un número en un array y devuelve la posición (el índice) en la que se encuentra.
7. **volteaArrayInt**: Le da la vuelta a un array.
8. **rotaDerechaArrayInt**: Rota *n* posiciones a la derecha los números de un array.
9. **rotaIzquierdaArrayInt**: Rota *n* posiciones a la izquierda los números de un array.



Ejercicio 29-34

Crea una biblioteca de funciones para arrays bidimensionales (de dos dimensiones) de números enteros que contenga las siguientes funciones:

1. **generaArrayBiInt**: Genera un array de tamaño $n \times m$ con números aleatorios cuyo intervalo (mínimo y máximo) se indica como parámetro.
2. **filaDeArrayBiInt**: Devuelve la fila i -ésima del array que se pasa como parámetro.
3. **columnaDeArrayBiInt**: Devuelve la columna j -ésima del array que se pasa como parámetro.
4. **coordenadasEnArrayBiInt**: Devuelve la fila y la columna (en un array con dos elementos) de la primera ocurrencia de un número dentro de un array bidimensional. Si el número no se encuentra en el array, la función devuelve el array $\{-1, -1\}$.
5. **esPuntoDeSilla**: Dice si un número es o no punto de silla, es decir, mínimo en su fila y máximo en su columna.
6. **diagonal**: Devuelve un array que contiene una de las diagonales del array bidimensional que se pasa como parámetro. Se pasan como parámetros fila, columna y dirección. La fila y la columna determinan el número que marcará las dos posibles diagonales dentro del array. La dirección es una cadena de caracteres que puede ser "nose" o "neso". La cadena "nose" indica que se elige la diagonal que va del noroeste hacia el sureste, mientras que la cadena "neso" indica que se elige la diagonal que va del noreste hacia el suroeste.

9. Programación orientada a objetos

9.1 Clases y objetos

La programación orientada a objetos es un paradigma de programación que se basa, como su nombre indica, en la utilización de objetos. Estos objetos también se suelen llamar instancias.

Un objeto en términos de POO no se diferencia mucho de lo que conocemos como un objeto en la vida real. Pensemos por ejemplo en un coche. Nuestro coche sería un objeto concreto de la vida real, igual que el coche del vecino, o el coche de un compañero de trabajo, o un deportivo que vimos por la calle el fin de semana pasado... Todos esos coches son objetos concretos que podemos ver y tocar.

Tanto mi coche como el coche del vecino tienen algo en común, ambos son coches. En este caso mi coche y el coche del vecino serían **instancias** (objetos) y coche (a secas) sería una **clase**. La palabra coche define algo genérico, es una abstracción, no es un coche concreto sino que hace referencia a unos elementos que tienen una serie de propiedades como matrícula, marca, modelo, color, etc.; este conjunto de propiedades se denominan **atributos** o **variables de instancia**.



Clase

Concepto abstracto que denota una serie de cualidades, por ejemplo **coche**.

Instancia

Objeto palpable, que se deriva de la concreción de una clase, por ejemplo **mi coche**.

Atributos

Conjunto de características que comparten los objetos de una clase, por ejemplo para la clase **coche** tendríamos **matrícula**, **marca**, **modelo**, **color** y **número de plazas**.

En Java, los nombres de las clases se escriben con la primera letra en mayúscula mientras que los nombres de las instancias comienzan con una letra en minúscula. Por ejemplo, la clase coche se escribe en Java como Coche y el objeto “mi coche” se podría escribir como miCoche.

Definiremos cada clase en un fichero con el mismo nombre más la extensión .java, por tanto, la definición de la clase Coche debe estar contenida en un fichero con nombre Coche.java

Vamos a definir a continuación la clase Libro con los atributos isbn, autor, titulo y numeroPaginas.

```
/**
 * Libro.java
 * Definición de la clase Libro
 * @author Luis José Sánchez
 */

public class Libro {

    // atributos
    String isbn;
    String titulo;
    String autor;
    int    numeroPaginas;
}
```

A continuación creamos varios objetos de esta clase.

```
/**
 * PruebaLibro.java
 * Programa que prueba la clase Libro
 * @author Luis José Sánchez
 */

public class PruebaLibro {
    public static void main(String[] args) {

        Libro lib = new Libro();
        Libro miLibrito = new Libro();
        Libro quijote = new Libro();
    }
}
```

Hemos creado tres instancias de la clase libro: lib, miLibrito y quijote. Ya sabemos definir una clase indicando sus atributos y crear instancias.



Las **variables de instancia** (atributos) determinan las **cualidades** de los objetos.

9.2 Encapsulamiento y ocultación

Uno de los pilares en los que se basa la Programación Orientada a Objetos es el **encapsulamiento**. Básicamente, el **encapsulamiento** consiste en definir todas las propiedades y el comportamiento de

una clase dentro de esa clase; es decir, en la clase `Coche` estará definido todo lo concerniente a la clase `Coche` y en la clase `Libro` estará definido todo lo que tenga que ver con la clase `Libro`.

El **encapsulamiento** parece algo obvio, casi de perogrullo, pero hay que tenerlo siempre muy presente al programar utilizando clases y objetos. En alguna ocasión puede que estemos tentados a mezclar parte de una clase con otra clase distinta para resolver un problema puntual. No hay que caer en esa trampa. Se deben escribir los programas de forma que cada cosa esté en su sitio. Sobre todo al principio, cuando definimos nuestras primeras clases, debemos estar pendientes de que todo está definido donde corresponde.

La **ocultación** es una técnica que incorporan algunos lenguajes (entre ellos Java) que permite esconder los elementos que definen una clase, de tal forma que desde otra clase distinta no se pueden “ver las tripas” de la primera. La **ocultación** facilita, como veremos más adelante, el **encapsulamiento**.

9.3 Métodos

Un coche arranca, para, se aparca, hace sonar el claxon, se puede llevar a reparar... Un gato puede comer, dormir, maullar, ronronear...

Las acciones asociadas a una clase se llaman métodos. Estos métodos se definen dentro del cuerpo de la clase y se suelen colocar a continuación de los atributos.



Los **métodos** determinan el **comportamiento** de los objetos.

9.3.1 Creí haber visto un lindo gatito

Vamos a crear la clase `GatoSimple`. La llamamos así porque más adelante crearemos otra clase algo más elaborada que se llamará `Gato`. Para saber qué atributos debe tener esta clase hay que preguntarse qué características tienen los gatos. Todos los gatos son de un color determinado, pertenecen a una raza, tienen una edad, tienen un determinado sexo - son machos o hembras - y tienen un peso que se puede expresar en kilogramos. Éstos serán por tanto los atributos que tendrá la clase `GatoSimple`.

Para saber qué métodos debemos implementar hay que preguntarse qué acciones están asociadas a los gatos. Bien, pues los gatos maullan, ronronean, comen y si son machos se pelean entre ellos para disputarse el favor de las hembras. Esos serán los métodos que definamos en la clase.

```
/**
 * GatoSimple.java
 * Definición de la clase GatoSimple
 * @author Luis José Sánchez
 */

public class GatoSimple {

    // atributos //////////////////////////////////////

    String color, raza, sexo;
    int edad;
    double peso;

    // métodos //////////////////////////////////////

    // constructor
    GatoSimple (String s) {
        this.sexo = s;
    }

    // getter
    String getSexo() {
        return this.sexo;
    }

    /**
     * Hace que el gato maulle
     */
    void maulla() {
        System.out.println("Miauuuu");
    }

    /**
     * Hace que el gato ronronee
     */
    void ronronea() {
        System.out.println("mrrrrrrr");
    }

    /**
     * Hace que el gato coma.
     * A los gatos les gusta el pescado, si le damos otra comida
     * la rechazará.
     */
}
```

```

    * @param comida la comida que se le ofrece al gato
    */
    void come(String comida) {
        if (comida.equals("pescado")) {
            System.out.println("Hmmm, gracias");
        } else {
            System.out.println("Lo siento, yo solo como pescado");
        }
    }
}

/**
 * Pone a pelear dos gatos.
 * Solo se van a pelear dos machos entre sí.
 *
 * @param contrincante es el gato contra el que pelear
 */
void peleaCon(GatoSimple contrincante) {
    if (this.sexo.equals("hembra")) {
        System.out.println("no me gusta pelear");
    } else {
        if (contrincante.getSexo().equals("hembra")) {
            System.out.println("no peleo contra gatitas");
        } else {
            System.out.println("ven aquí que te vas a enterar");
        }
    }
}
}

```



Constructor

El método constructor tiene siempre el mismo nombre que la clase y se utiliza normalmente para inicializar los atributos.

Los atributos de la clase `GatoSimple` - color, raza, sexo, edad y peso - se declaran igual que las variables que hemos venido usando hasta ahora, pero hay una gran diferencia entre estos atributos y las variables que aparecen en el `main` (programa principal). Una variable definida en el cuerpo del programa principal es única, sin embargo cada uno de los objetos que se crean en el programa principal tienen sus propios atributos; es decir, si en el programa principal se crean 20 objetos de la clase `GatoSimple`, cada uno tiene sus valores para los atributos color, raza, etc.

Fíjate en la cabecera del método `peleaCon`:


```
void peleaCon(GatoSimple contrincante)
```

Como puedes comprobar, es posible pasar un objeto como parámetro. A continuación se muestra un programa que prueba la clase `GatoSimple`. Te recomiendo seguir el programa línea a línea y observar atentamente la salida que produce.

```
/**
 * PruebaGatoSimple.java
 * Programa que prueba la clase GatoSimple
 * @author Luis José Sánchez
 */

public class PruebaGatoSimple {
    public static void main(String[] args) {

        GatoSimple garfield = new GatoSimple("macho");

        System.out.println("hola gatito");
        garfield.maulla();
        System.out.println("toma tarta");
        garfield.come("tarta selva negra");
        System.out.println("toma pescado, a ver si esto te gusta");
        garfield.come("pescado");

        GatoSimple tom = new GatoSimple("macho");

        System.out.println("Tom, toma sopita de verduras");
        tom.come("sopa de verduras");

        GatoSimple lisa = new GatoSimple("hembra");

        System.out.println("gatitos, a ver cómo maulláis");
        garfield.maulla();
        tom.maulla();
        lisa.maulla();

        garfield.peleaCon(lisa);
        lisa.peleaCon(tom);
        tom.peleaCon(garfield);
    }
}
```

Observa cómo al crear una instancia se llama al constructor que, como decíamos antes, tiene el mismo nombre de la clase y sirve para inicializar los atributos. En este caso se inicializa el atributo

sexo. Más adelante veremos constructores que inicializan varios atributos e incluso definiremos distintos constructores en la misma clase.

```
GatoSimple garfield = new GatoSimple("macho");
```

Como puedes ver hay métodos que no toman ningún parámetro.

```
garfield.maula();
```

Y hay otros métodos que deben tomar parámetros obligatoriamente.

```
garfield.come("tarta selva negra");
```

9.3.2 Métodos *getter* y *setter*

Vamos a crear la clase Cubo. Para saber qué atributos se deben definir, nos preguntamos qué características tienen los cubos - igual que hicimos con la clase GatoSimple. Todos los cubos tienen una determinada capacidad, un color, están hechos de un determinado material - plástico, latón, etc. - y puede que tengan asa o puede que no. Un cubo se fabrica con el propósito de contener líquido; por tanto otra característica es la cantidad de litros de líquido que contiene en un momento determinado. Por ahora, solo nos interesa saber la capacidad máxima y los litros que contiene el cubo en cada momento, así que esos serán los atributos que tendremos en cuenta.

```
/**
 * Cubo.java
 * Definición de la clase Cubo
 * @author Luis José Sánchez
 */

public class Cubo {

    // atributos //////////////////////////////////

    int capacidad; // capacidad máxima en litros
    int contenido; // contenido actual en litros

    // métodos //////////////////////////////////

    // constructor
    Cubo (int c) {
        this.capacidad = c;
    }
}
```

```
// métodos getter
int getCapacidad() {
    return this.capacidad;
}

int getContenido() {
    return this.contenido;
}

// método setter
void setContenido(int litros) {
    this.contenido = litros;
}

// otros métodos
void vacia() {
    this.contenido = 0;
}

/**
 * Llena el cubo al máximo de su capacidad.
 */
void llena() {
    this.contenido = this.capacidad;
}

/**
 * Pinta el cubo en la pantalla.
 * Se muestran los bordes del cubo con el carácter # y el
 * agua que contiene con el carácter ~.
 */
void pinta() {
    for (int nivel = this.capacidad; nivel > 0; nivel--) {
        if (this.contenido >= nivel) {
            System.out.println("#~~~#");
        } else {
            System.out.println("#    #");
        }
    }
    System.out.println("#####");
}

/**
 * Vuelca el contenido de un cubo sobre otro.
```

```

    * Antes de echar el agua se comprueba cuánto le cabe al
    * cubo destino.
    */
void vuelcaEn(Cubo destino) {
    int libres = destino.getCapacidad() - destino.getContenido();

    if (libres > 0) {
        if (this.contenido <= libres) {
            destino.setContenido(destino.getContenido() + this.contenido);
            this.vacia();
        } else {
            this.contenido -= libres;
            destino.llena();
        }
    }
}
}
}

```

Observa estos métodos extraídos de la clase Cubo:

```

int getCapacidad() {
    return this.capacidad;
}

int getContenido() {
    return this.contenido;
}

```

Se trata de métodos muy simples, su cometido es devolver el valor de un atributo. Podrían tener cualquier nombre pero en Java es costumbre llamarlos con la palabra *get* (obtener) seguida del nombre del atributo.

Fijémonos ahora este otro método:

```

void setContenido(int litros) {
    this.contenido = litros;
}

```

Ahora estamos ante un *setter*. Este tipo de métodos tiene el cometido de establecer un valor para un determinado atributo. Como puedes ver, *setContenido* está formada por *set* (asignar) más el nombre del atributo.

Podrías preguntarte: ¿por qué se crea un método *getter* para extraer el valor de una variable y no se accede a la variable directamente?

Parece más lógico hacer esto

```
System.out.print(miCubo.capacidad);
```

que esto otro

```
System.out.print(miCubo.getCapacidad());
```

Sin embargo, en Java se opta casi siempre por esta última opción. Tiene su explicación y lo entenderás bien cuando estudies el apartado [Ámbito/visibilidad de los elementos de una clase - public, protected y private](#)

Probamos, con el siguiente ejemplo, la clase Cubo que acabamos de definir. Crea tus propios cubos, pasa agua de unos a otros y observa lo que se muestra por pantalla.

```
/**
 * PruebaCubo.java
 * Programa que prueba la clase Cubo
 * @author Luis José Sánchez
 */

public class PruebaCubo {
    public static void main(String[] args) {

        Cubo cubito = new Cubo(2);
        Cubo cubote = new Cubo(7);

        System.out.println("Cubito: \n");
        cubito.pinta();

        System.out.println("\nCubote: \n");
        cubote.pinta();

        System.out.println("\nLleno el cubito: \n");
        cubito.llena();
        cubito.pinta();

        System.out.println("\nEl cubote sigue vacío: \n");
        cubote.pinta();

        System.out.println("\nAhora vuelco lo que tiene el cubito en el cubote.\n");
        cubito.vuelcaEn(cubote);

        System.out.println("Cubito: \n");
        cubito.pinta();
```

```

        System.out.println("\nCubote: \n");
        cubote.pinta();

        System.out.println("\nAhora vuelco lo que tiene el cubote en el cubito.\n");
        cubote.vuelcaEn(cubito);

        System.out.println("Cubito: \n");
        cubito.pinta();

        System.out.println("\nCubote: \n");
        cubote.pinta();
    }
}

```

9.3.3 Método toString

La definición de la clase Cubo del apartado anterior contiene el método pinta que, como su nombre indica, permite pintar en pantalla un objeto de esa clase. Se podría definir el método ficha para mostrar información sobre objetos de la clase Alumno por ejemplo. También sería posible implementar el método imprime dentro de la clase Libro con el propósito de mostrar por pantalla los datos de un libro. Todos estos métodos - pinta, ficha e imprime - hacen básicamente lo mismo.

En Java existe una solución muy elegante para mostrar información sobre un objeto por pantalla. Si se quiere mostrar el contenido de la variable entera x se utiliza System.out.print(x) y si se quiere mostrar el valor de la variable de tipo cadena de caracteres nombre se escribe System.out.print(nombre). De la misma manera, si se quiere mostrar el objeto miPiramide que pertenece a la clase Piramide, también se podría usar System.out.print(miPiramide). Java sabe perfectamente cómo mostrar números y cadenas de caracteres pero no sabe *a priori* cómo se pintan pirámides. Para indicar a Java cómo debe pintar un objeto de la clase Piramide basta con implementar el método toString dentro de la clase.

Veamos un ejemplo muy sencillo de implementación de toString. Definiremos la clase Cuadrado con el atributo lado, el constructor y el método toString.

```

/**
 * Cuadrado.java
 * Definición de la clase Cuadrado
 * @author Luis José Sánchez
 */

public class Cuadrado {

    int lado;

    public Cuadrado(int l) {

```

```

        this.lado = 1;
    }

    public String toString() {

        int i, espacios;
        String resultado = "";

        for (i = 0; i < this.lado; i++) {
            resultado += "□□";
        }
        resultado += "\n";

        for (i = 1; i < this.lado - 1; i++) {
            resultado += "□□";
            for (espacios = 1; espacios < this.lado - 1; espacios++) {
                resultado += "  ";
            }
            resultado += "□□\n";
        }

        for (i = 0; i < this.lado; i++) {
            resultado += "□□";
        }
        resultado += "\n";

        return resultado;
    }
}

```

Observa que el método `toString()` devuelve una cadena de caracteres. Esa cadena es precisamente la que mostrará por pantalla `System.out.print()`. En el programa que se muestra a continuación y que prueba la clase `Cuadrado` puedes comprobar que se pinta un cuadrado igual que si se tratara de cualquier otra variable.

```

/**
 * PruebaCuadrado.java
 * Programa que prueba la clase Cuadrado
 * @author Luis José Sánchez
 */

public class PruebaCuadrado {
    public static void main(String[] args) {

        Cuadrado miCuadradito = new Cuadrado(5);
    }
}

```

```
        System.out.println(miCuadrado);  
    }  
}
```


9.4 Ámbito/visibilidad de los elementos de una clase - `public`, `protected` y `private`

Al definir los elementos de una clase, se pueden especificar sus ámbitos (*scope*) de visibilidad o accesibilidad. Un elemento `public` (público) es visible desde cualquier clase, un elemento `protected` (protegido) es visible desde la clase actual y desde todas sus subclases y, finalmente, un elemento `private` (privado) únicamente es visible dentro de la clase actual.

En el siguiente apartado veremos qué son las subclases. De momento fíjate en el ámbito del atributo y de los métodos en la definición de la clase `Animal`.

```
/**
 * Animal.java
 * Definición de la clase Animal
 * @author Luis José Sánchez
 */

public abstract class Animal {

    private Sexo sexo;

    public Animal () {
        sexo = Sexo.MACHO;
    }

    public Animal (Sexo s) {
        sexo = s;
    }

    public Sexo getSexo() {
        return sexo;
    }

    public String toString() {
        return "Sexo: " + this.sexo + "\n";
    }

    /**
     * Hace que el animal se eche a dormir.
     */
    public void duerme() {
        System.out.println("Zzzzzzz");
    }
}
```

Fíjate que el atributo `sexo` se ha definido como `private`.

```
private Sexo sexo;
```

Eso quiere decir que a ese atributo únicamente se tiene acceso dentro de la clase `Animal`. Sin embargo, todos los métodos se han definido `public`, lo que significa que se podrán utilizar desde cualquier otro programa, por ejemplo, como veremos más adelante, desde el programa `PurebaAnimal.java`.



Salvo casos puntuales se seguirá la regla de declarar `private` las variables de instancia y `public` los métodos.

El sexo de un animal solo puede ser macho, hembra o hermafrodita. Una forma de delimitar los valores que puede tomar un atributo es definir un tipo enumerado.



Tipo enumerado

Mediante `enum` se puede definir un tipo enumerado, de esta forma un atributo solo podrá tener uno de los posibles valores que se dan como opción. Los valores que se especifican en el tipo enumerado se suelen escribir con todas las letras en mayúscula.

```
/**
 * Sexo.java
 * Definición del tipo enumerado Sexo
 * @author Luis José Sánchez
 */

public enum Sexo {
    MACHO, HEMBRA, HERMAFRODITA
}
```

9.5 Herencia

La herencia es una de las características más importantes de la POO. Si definimos una serie de atributos y métodos para una clase, al crear una subclase, todos estos atributos y métodos siguen siendo válidos.

En el apartado anterior se define la clase `Animal`. Uno de los métodos de esta clase es `duerme`. A continuación podemos crear las clases `Gato` y `Perro` como subclases de `Animal`. De forma automática, se puede utilizar el método `duerme` con las instancias de las clases `Gato` y `Perro` ¿no es fantástico?

La clase `Ave` es subclase de `Animal` y la clase `Pinguino`, a su vez, sería subclase de `Ave` y por tanto hereda todos sus atributos y métodos.



Clase abstracta (abstract)

Una clase abstracta es aquella que no va a tener instancias de forma directa, aunque sí habrá instancias de las subclases (siempre que esas subclases no sean también abstractas). Por ejemplo, si se define la clase `Animal` como abstracta, no se podrán crear objetos de la clase `Animal`, es decir, no se podrá hacer `Animal mascota = new Animal()`, pero sí se podrán crear instancias de la clase `Gato`, `Ave` o `Pinguino` que son subclases de `Animal`.

Para crear en Java una subclase de otra clase existente se utiliza la palabra reservada `extends`. A continuación se muestra el código de las clases `Gato`, `Ave` y `Pinguino`, así como el programa que prueba estas clases creando instancias y aplicándoles métodos. Recuerda que la definición de la clase `Animal` se muestra en el apartado anterior.

```
/**
 * Gato.java
 * Definición de la clase Gato
 * @author Luis José Sánchez
 */

public class Gato extends Animal {

    private String raza;

    public Gato (Sexo s, String r) {
        super(s);
        raza = r;
    }

    public Gato (Sexo s) {
        super(s);
        raza = "siamés";
    }

    public Gato (String r) {
        super(Sexo.HEMBRA);
        raza = r;
    }

    public Gato () {
        super(Sexo.HEMBRA);
        raza = "siamés";
    }

    public String toString() {
```

```
        return super.toString()
            + "Raza: " + this.raza
            + "\n*****\n";
    }

    /**
     * Hace que el gato maulle.
     */
    public void maulla() {
        System.out.println("Miauuuu");
    }

    /**
     * Hace que el gato ronronee
     */
    public void ronronea() {
        System.out.println("mrrrrrr");
    }

    /**
     * Hace que el gato coma.
     * A los gatos les gusta el pescado, si le damos otra comida
     * la rechazará.
     *
     * @param comida la comida que se le ofrece al gato
     */
    public void come(String comida) {
        if (comida.equals("pescado")) {
            System.out.println("Hmmm, gracias");
        } else {
            System.out.println("Lo siento, yo solo como pescado");
        }
    }

    /**
     * Pone a pelear dos gatos.
     * Solo se van a pelear dos machos entre sí.
     *
     * @param contrincante es el gato contra el que pelear
     */
    public void peleaCon(Gato contrincante) {
        if (this.getSexo() == Sexo.HEMBRA) {
            System.out.println("no me gusta pelear");
        } else {
            if (contrincante.getSexo() == Sexo.HEMBRA) {
```

```

        System.out.println("no pelea contra gatitas");
    } else {
        System.out.println("ven aquí que te vas a enterar");
    }
}
}
}
}

```

Observa que se definen nada menos que cuatro constructores en la clase `Gato`. Desde el programa principal se dilucida cuál de ellos se utiliza en función del número y tipo de parámetros que se pasa al método. Por ejemplo, si desde el programa principal se crea un gato de esta forma

```
Gato gati = new Gato();
```

entonces se llamaría al constructor definido como

```

public Gato () {
    super(Sexo.HEMBRA);
    raza = "siamés";
}

```

Por tanto `gati` sería una gata de raza siamés. Si, por el contrario, creamos `gati` de esta otra manera desde el programa principal

```
Gato gati = new Gato(Sexo.MACHO, "siberiano");
```

se llamaría al siguiente constructor

```

public Gato (Sexo s, String r) {
    super(s);
    raza = r;
}

```

y `gati` sería en este caso un gato macho de raza siberiano.

El método `super()` hace una llamada al método equivalente de la superclase. Fíjate que se utiliza tanto en el constructor como en el método `toString()`. Por ejemplo, al llamar a `super()` dentro del método `toString()` se está llamando al `toString()` que hay definido en la clase `Animal`, justo un nivel por encima de `Gato` en la jerarquía de clases.

A continuación tenemos la definición de la clase `Ave` que es una subclase de `Animal`.

```
/**
 * Ave.java
 * Definición de la clase Ave
 * @author Luis José Sánchez
 */

public class Ave extends Animal {

    public Ave(Sexo s) {
        super(s);
    }

    public Ave() {
        super();
    }

    /**
     * Hace que el ave se limpie.
     */
    public void aseate() {
        System.out.println("Me estoy limpiando las plumas");
    }

    /**
     * Hace que el ave levante el vuelo.
     */
    public void vuela() {
        System.out.println("Estoy volando");
    }
}
```

9.5.1 Sobrecarga de métodos

Un método se puede **redefinir** (volver a definir con el mismo nombre) en una subclase. Por ejemplo, el método `vuela` que está definido en la clase `Ave` se vuelve a definir en la clase `Pinguino`. En estos casos, indicaremos nuestra intención de sobrescribir un método mediante la etiqueta `@Override`.

Si no escribimos esta etiqueta, la sobrescritura del método se realizará de todas formas ya que `@Override` indica simplemente una intención. Ahora imagina que quieres sobrescribir el método `come` de `Animal` declarando un `come` específico para los gatos en la clase `Gato`. Si escribes `@Override` y luego te equivocas en el nombre del método y escribes `comer`, entonces el compilador diría algo como: “¡Cuidado! algo no está bien, me has dicho que ibas a sobrescribir un método de la superclase y sin embargo `comer` no está definido”.

A continuación tienes la definición de la clase `Pinguino`.

```
/**
 * Pinguino.java
 * Definición de la clase Pinguino
 * @author Luis José Sánchez
 */

public class Pinguino extends Ave {

    public Pinguino() {
        super();
    }

    public Pinguino(Sexo s) {
        super(s);
    }

    /**
     * El pingüino se siente triste porque no puede volar.
     */
    @Override
    public void vuela() {
        System.out.println("No puedo volar");
    }
}
```

Con el siguiente programa se prueba la clase `Animal` y todas las subclases que derivan de ella. Observa cada línea y comprueba qué hace el programa.

```
/**
 * PruebaAnimal.java
 * Programa que prueba la clase Animal y sus subclases
 * @author Luis José Sánchez
 */

public class PruebaAnimal {
    public static void main(String[] args) {

        Gato garfield = new Gato(Sexo.MACHO, "romano");
        Gato tom = new Gato(Sexo.MACHO);
        Gato lisa = new Gato(Sexo.HEMBRA);
        Gato silvestre = new Gato();

        System.out.println(garfield);
        System.out.println(tom);
        System.out.println(lisa);
    }
}
```

```
System.out.println(silvestre);

Ave miLoro = new Ave();
miLoro.aseate();
miLoro.vuela();

Pinguino pingu = new Pinguino(Sexo.HEMBRA);
pingu.aseate();
pingu.vuela();
}
```

En el ejemplo anterior, los objetos `miLoro` y `pingu` actúan de manera **polimórfica** porque a ambos se les aplican los métodos `aseate` y `vuela`.



Polimorfismo

En Programación Orientada a Objetos, se llama **polimorfismo** a la capacidad que tienen los objetos de distinto tipo (de distintas clases) de responder al mismo método.

9.6 Atributos y métodos de clase (static)

Hasta el momento hemos definido atributos de instancia como raza, sexo o color y métodos de instancia como `maulla`, `come` o `vuela`. De tal modo que si en el programa se crean 20 gatos, cada uno de ellos tiene su propia raza y puede haber potencialmente 20 razas diferentes. También podría aplicar el método `maulla` a todos y cada uno de esos 20 gatos.

No obstante, en determinadas ocasiones, nos puede interesar tener atributos de clase (variables de clase) y métodos de clase. Cuando se define una variable de clase solo existe una copia del atributo para toda la clase y no una para cada objeto. Esto es útil cuando se quiere llevar la cuenta global de algún parámetro. Los métodos de clase se aplican a la clase y no a instancias concretas.

A continuación se muestra un ejemplo que contiene la variable de clase `kilometrajeTotal`. Si bien cada coche tiene un atributo `kilometraje` donde se van acumulando los kilómetros que va recorriendo, en la variable de clase `kilometrajeTotal` se lleva la cuenta de los kilómetros que han recorrido todos los coches que se han creado.

También se crea un método de clase llamado `getKilometrajeTotal` que simplemente es un *getter* para la variable de clase `kilometrajeTotal`.


```
/**
 * Coche.java
 * Definición de la clase Coche
 * @author Luis José Sánchez
 */

public class Coche {

    // atributo de clase
    private static int kilometrajeTotal = 0;

    // método de clase
    public static int getKilometrajeTotal() {
        return kilometrajeTotal;
    }

    private String marca;
    private String modelo;
    private int kilometraje;

    public Coche(String ma, String mo) {
        marca = ma;
        modelo = mo;
        kilometraje = 0;
    }

    public int getKilometraje() {
        return kilometraje;
    }

    /**
     * Recorre una determinada distancia.
     *
     * @param km distancia a recorrer en kilómetros
     */
    public void recorre(int km) {
        kilometraje += km;
        kilometrajeTotal += km;
    }
}
```

Como ya hemos comentado, el atributo `kilometrajeTotal` almacena el número total de kilómetros que recorren todos los objetos de la clase `Coche`, es un único valor, por eso se declara como `static`. Por el contrario, el atributo `kilometraje` almacena los kilómetros recorridos por un objeto concreto y tendrá un valor distinto para cada uno de ellos. Si en el programa principal se crean 20 objetos de

la clase Coche, cada uno tendrá su propio kilometraje.

A continuación se muestra el programa que prueba la clase Coche.

```
/**
 * PruebaCoche.java
 * Programa que prueba la clase Coche
 * @author Luis José Sánchez
 */

public class PruebaCoche {
    public static void main(String[] args) {

        Coche cocheDeLuis = new Coche("Saab", "93");
        Coche cocheDeJuan = new Coche("Toyota", "Avensis");

        cocheDeLuis.recorre(30);
        cocheDeLuis.recorre(40);
        cocheDeLuis.recorre(220);
        cocheDeJuan.recorre(60);
        cocheDeJuan.recorre(150);
        cocheDeJuan.recorre(90);
        System.out.println("El coche de Luis ha recorrido " + cocheDeLuis.getKilometraje() + "\
Km");
        System.out.println("El coche de Juan ha recorrido " + cocheDeJuan.getKilometraje() + "\
Km");
        System.out.println("El kilometraje total ha sido de " + Coche.getKilometrajeTotal() + \
"Km");
    }
}
```

El método `getKilometrajeTotal()` se aplica a la clase Coche por tratarse de un método de clase (método `static`). Este método no se podría aplicar a una instancia, de la misma manera que un método que no sea `static` no se puede aplicar a la clase sino a los objetos.

9.7 Interfaces

Una **interfaz** contiene únicamente la cabecera de una serie de métodos (opcionalmente también puede contener constantes). Por tanto se encarga de especificar un comportamiento que luego tendrá que ser implementado. La **interfaz** no especifica el “cómo” ya que no contiene el cuerpo de los métodos, solo el “qué”.

Una **interfaz** puede ser útil en determinadas circunstancias. En principio, separa la definición de la implementación o, como decíamos antes, el “qué” del “cómo”. Tendremos entonces la menos

dos ficheros, la **interfaz** y la clase que implementa esa **interfaz**. Se puede dar el caso que un programador escriba la **interfaz** y luego se la pase a otro programador para que sea éste último quien la implemente.

Hay que destacar que cada **interfaz** puede tener varias implementaciones asociadas.

Para ilustrar el uso de interfaces utilizaremos algunas clases ya conocidas. La superclase que va a estar por encima de todas las demás será la clase `Animal` vista con anterioridad. El código de esta clase no varía, por lo tanto no lo vamos a reproducir aquí de nuevo.

Definimos la **interfaz** `Mascota`.

```
/**
 * Mascota.java
 * Definición de la interfaz Mascota
 *
 * @author Luis José Sánchez
 */
public interface Mascota {
    String getCodigo();
    void hazRuido();
    void come(String comida);
    void peleaCon(Animal contrincante);
}
```

Como puedes ver, únicamente se escriben las cabeceras de los métodos que debe tener la/s clase/s que implemente/n la **interfaz** `Mascota`.

Una de las implementaciones de `Mascota` será `Gato`.

```
/**
 * Gato.java
 * Definición de la clase Gato
 *
 * @author Luis José Sánchez
 */
public class Gato extends Animal implements Mascota {

    private String codigo;

    public Gato (Sexo s, String c) {
        super(s);
        this.codigo = c;
    }

    @Override
```

```
public String getCodigo() {
    return this.codigo;
}

/**
 * Hace que el gato emita sonidos.
 */
@Override
public void hazRuido() {
    this.maula();
    this.ronronea();
}

/**
 * Hace que el gato maulle.
 */
public void maulla() {
    System.out.println("Miauuuu");
}

/**
 * Hace que el gato ronronee
 */
public void ronronea() {
    System.out.println("mrrrrrr");
}

/**
 * Hace que el gato coma.
 * A los gatos les gusta el pescado, si le damos otra comida
 * la rechazará.
 *
 * @param comida la comida que se le ofrece al gato
 */
@Override
public void come(String comida) {

    if (comida.equals("pescado")) {
        super.come();
        System.out.println("Hmmm, gracias");
    } else {
        System.out.println("Lo siento, yo solo como pescado");
    }
}
```

```

/**
 * Pone a pelear al gato contra otro animal.
 * Solo se van a pelear dos machos entre sí.
 *
 * @param contrincante es el animal contra el que pelear
 */
@Override
public void peleaCon(Animal contrincante) {
    if (this.getSexo() == Sexo.HEMBRA) {
        System.out.println("no me gusta pelear");
    } else {
        if (contrincante.getSexo() == Sexo.HEMBRA) {
            System.out.println("no peleo contra hembras");
        } else {
            System.out.println("ven aquí que te vas a enterar");
        }
    }
}
}
}

```

Mediante la siguiente línea:

```
public class Gato extends Animal implements Mascota {
```

estamos diciendo que Gato es una subclase de Animal y que, además, es una implementación de la **interfaz** Mascota. Fíjate que no es lo mismo la herencia que la implementación.

Observa que los métodos que se indicaban en Mascota únicamente con la cabecera ahora están implementados completamente en Gato. Además, Gato contiene otros métodos que no se indicaban en Mascota como maulla y ronronea.

Los métodos de Gato que implementan métodos especificados en Mascota deben tener la anotación `@Override`.

Como dijimos anteriormente, una **interfaz** puede tener varias implementaciones. A continuación se muestra Perro, otra implementación de Mascota.

```
/**
 * Perro.java
 * Definición de la clase Perro
 *
 * @author Luis José Sánchez
 */
public class Perro extends Animal implements Mascota {

    private String codigo;

    public Perro (Sexo s, String c) {
        super(s);
        this.codigo = c;
    }

    @Override
    public String getCodigo() {
        return this.codigo;
    }

    /**
     * Hace que el Perro emita sonidos.
     */
    @Override
    public void hazRuido() {
        this.ladra();
    }

    /**
     * Hace que el Perro ladre.
     */
    public void ladra() {
        System.out.println("Guau guau");
    }

    /**
     * Hace que el Perro coma.
     * A los Perros les gusta la carne, si le damos otra comida la rechazará.
     *
     * @param comida la comida que se le ofrece al Perro
     */
    @Override
    public void come(String comida) {

        if (comida.equals("carne")) {
```

```

        super.come();
        System.out.println("Hmmm, gracias");
    } else {
        System.out.println("Lo siento, yo solo como carne");
    }
}

/**
 * Pone a pelear el perro contra otro animal.
 * Solo se van a pelear si los dos son perros.
 *
 * @param contrincante es el animal contra el que pelear
 */
@Override
public void peleaCon(Animal contrincante) {
    if (contrincante.getClass().getSimpleName().equals("Perro")) {
        System.out.println("ven aquí que te vas a enterar");
    } else {
        System.out.println("no me gusta pelear");
    }
}
}

```

Por último mostramos el programa que prueba Mascota y sus implementaciones Gato y Perro.

```

/**
 * PruebaMascota.java
 * Programa que prueba la interfaz Mascota
 *
 * @author Luis José Sánchez
 */
public class PruebaMascota {
    public static void main(String[] args) {

        Mascota garfield = new Gato(Sexo.MACHO, "34569");
        Mascota lisa = new Gato(Sexo.HEMBRA, "96059");
        Mascota kuki = new Perro(Sexo.HEMBRA, "234678");
        Mascota ayo = new Perro(Sexo.MACHO, "778950");

        System.out.println(garfield.getCodigo());
        System.out.println(lisa.getCodigo());
        System.out.println(kuki.getCodigo());
        System.out.println(ayo.getCodigo());
        garfield.come("pescado");
        lisa.come("hamburguesa");
    }
}

```

```

    kuki.come("pescado");
    lisa.peleaCon((Gato)garfield);
    ayo.peleaCon((Perro)kuki);
}
}

```

Observa que para crear una mascota que es un gato escribimos lo siguiente:

```
Mascota garfield = new Gato(Sexo.MACHO, "34569");
```

Una **interfaz** no se puede instanciar, por tanto la siguiente línea sería incorrecta:

```
Mascota garfield = new Mascota(Sexo.MACHO, "34569");
```



Interfaces

La interfaz indica “qué” hay que hacer y la implementación específica “cómo” se hace.

Una interfaz puede tener varias implementaciones.

Una interfaz no se puede instanciar.

La implementación puede contener métodos adicionales cuyas cabeceras no están en su interfaz.

9.8 Arrays de objetos

Del mismo modo que se pueden crear arrays de números enteros, decimales o cadenas de caracteres, también es posible crear arrays de objetos.

Vamos a definir la clase `Alumno` para luego crear un array de objetos de esta clase.

```

/**
 * Alumno.java
 * Definición de la clase Alumno
 * @author Luis José Sánchez
 */

public class Alumno {
    private String nombre;
    private double notaMedia = 0.0;

    public String getNombre() {

```



```

        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public double getNotaMedia() {
        return notaMedia;
    }

    public void setNotaMedia(double notaMedia) {
        this.notaMedia = notaMedia;
    }
}

```

A continuación se define un array de cinco alumnos que posteriormente se rellena. Por último se muestran los datos de los alumnos por pantalla.

Observa que la siguiente línea únicamente define la estructura del array pero no crea los objetos:

```
Alumno[] alum = new Alumno[5];
```

Cada objeto concreto se crea de forma individual mediante

```
alum[i] = new Alumno();
```

Aquí tienes el ejemplo completo.

```

/**
 * ArrayDeAlumnosPrincipal.java
 * Programa que prueba un array de la clase Alumno
 * @author Luis José Sánchez
 */

public class ArrayDeAlumnosPrincipal {
    public static void main(String[] args) {

        // Define la estructura, un array de 5 alumnos
        // pero no crea los objetos
        Alumno[] alum = new Alumno[5];

        // Pide los datos de los alumnos //////////////////////////////////////

```

```

    System.out.println("A continuacion debera introducir el nombre y la nota media de 5 al\
umnos.");

    String nombreIntroducido;
    double notaIntroducida;

    for(int i = 0; i < 5; i++) {

        alum[i] = new Alumno();

        System.out.println("Alumno " + i);

        System.out.print("Nombre: ");
        nombreIntroducido = System.console().readLine();
        (alum[i]).setNombre(nombreIntroducido);

        System.out.print("Nota media: ");
        notaIntroducida = Double.parseDouble(System.console().readLine());
        alum[i].setNotaMedia(notaIntroducida);
    } // for i

    // Muestra los datos de los alumnos //////////////////////////////////////

    System.out.println("Los datos introducidos son los siguientes:");

    double sumaDeMedias = 0;

    for(int i = 0; i < 5; i++) {
        System.out.println("Alumno " + i);
        System.out.println("Nombre: " + alum[i].getNombre());
        System.out.println("Nota media: " + alum[i].getNotaMedia());
        System.out.println("-----");

        sumaDeMedias += alum[i].getNotaMedia();
    } // for i

    System.out.println("La media global de la clase es " + sumaDeMedias / 5);
}

```

Veamos ahora un ejemplo algo más complejo. Se trata de una gestión típica - alta, baja, listado y modificación - de una colección de discos. Este tipo de programas se suele denominar CRUD (*Create Read Update Delete*).

Primero se define la clase Disco.

```
/**
 * Disco.java
 * Definición de la clase Disco
 * @author Luis José Sánchez
 */

public class Disco {
    private String codigo = "LIBRE";
    private String autor;
    private String titulo;
    private String genero;
    private int duracion; // duración total en minutos

    public String getCodigo() {
        return codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public String getGenero() {
        return genero;
    }

    public void setGenero(String genero) {
        this.genero = genero;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
}
```

```
public int getDuracion() {
    return duracion;
}

public void setDuracion(int duracion) {
    this.duracion = duracion;
}

public String toString() {
    String cadena = "\n-----";
    cadena += "\nCódigo: " + this.codigo;
    cadena += "\nAutor: " + this.autor;
    cadena += "\nTítulo: " + this.titulo;
    cadena += "\nGénero: " + this.genero;
    cadena += "\nDuración: " + this.duracion;
    cadena += "\n-----";

    return cadena;
}
}
```

A continuación se crea el programa principal. Cada elemento de la colección será un objeto de la clase Disco. Se trata, como verás, de una gestión muy simple pero totalmente funcional.

No se incluye en el manual porque ocuparía varias páginas. El código del programa se puede descargar desde el siguiente enlace: https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/09_POO/ColeccionDeDiscos/ColeccionDeDiscosPrincipal.java

9.9 Ejercicios

9.9.1 Conceptos de POO



Ejercicio 1

¿Cuáles serían los atributos de la clase `PilotoDeFormula1`? ¿Se te ocurren algunas instancias de esta clase?



Ejercicio 2

A continuación tienes una lista en la que están mezcladas varias clases con instancias de esas clases. Para ponerlo un poco más difícil, todos los elementos están escritos en minúscula. Di cuáles son las clases, cuáles las instancias, a qué clase pertenece cada una de estas instancias y cuál es la jerarquía entre las clases: paula, goofy, gardfiel, perro, mineral, caballo, tom, silvestre, pirita, rocinante, milu, snoopy, gato, pluto, animal, javier, bucefalo, pegaso, ayudante_de_santa_claus, cuarzo, laika, persona, pato_lucas.



Ejercicio 3

¿Cuáles serían los atributos de la clase `Vivienda`? ¿Qué subclases se te ocurren?



Ejercicio 4

Piensa en la liga de baloncesto, ¿qué 5 clases se te ocurren para representar 5 elementos distintos que intervengan en la liga?



Ejercicio 5

Haz una lista con los atributos que podría tener la clase `caballo`. A continuación haz una lista con los posibles métodos (acciones asociadas a los caballos).



Ejercicio 6

Lista los atributos de la clase `Alumno` ¿Sería nombre uno de los atributos de la clase? Razona tu respuesta.



Ejercicio 7

¿Cuáles serían los atributos de la clase `Ventana` (de ordenador)? ¿cuáles serían los métodos? Piensa en las propiedades y en el comportamiento de una ventana de cualquier programa.

9.9.2 POO en Java



Ejercicio 1

Implementa la clase `Caballo` vista en un ejercicio anterior. Pruébala creando instancias y aplicándole algunos métodos.



Ejercicio 2

Crea la clase `Vehiculo`, así como las clases `Bicicleta` y `Coche` como subclases de la primera. Para la clase `Vehiculo`, crea los atributos de clase `vehiculosCreados` y `kilometrosTotales`, así como el atributo de instancia `kilometrosRecorridos`. Crea también algún método específico para cada una de las subclases. Prueba las clases creadas mediante un programa con un menú como el que se muestra a continuación:

```
VEHÍCULOS
=====
1. Anda con la bicicleta
2. Haz el caballito con la bicicleta
3. Anda con el coche
4. Quema rueda con el coche
5. Ver kilometraje de la bicicleta
6. Ver kilometraje del coche
7. Ver kilometraje total
8. Salir
Elige una opción (1-8):
```



Ejercicio 3

Crea las clases `Animal`, `Mamifero`, `Ave`, `Gato`, `Perro`, `Canario`, `Pinguino` y `Lagarto`. Crea, al menos, tres métodos específicos de cada clase y redefine el/los método/s cuando sea necesario. Prueba las clases creadas en un programa en el que se instancien objetos y se les apliquen métodos.



Ejercicio 4

Crea la clase Fracción. Los atributos serán numerador y denominador. Y algunos de los métodos pueden ser invierte, simplifica, multiplica, divide, etc.



Ejercicio 5

Crea la clase Pizza con los atributos y métodos necesarios. Sobre cada pizza se necesita saber el tamaño - mediana o familiar - el tipo - margarita, cuatro quesos o funghi - y su estado - pedida o servida. La clase debe almacenar información sobre el número total de pizzas que se han pedido y que se han servido. Siempre que se crea una pizza nueva, su estado es "pedida". El siguiente código del programa principal debe dar la salida que se muestra:

```
public class PedidosPizza {
    public static void main(String[] args) {
        Pizza p1 = new Pizza("margarita", "mediana");
        Pizza p2 = new Pizza("funghi", "familiar");
        p2.sirve();
        Pizza p3 = new Pizza("cuatro quesos", "mediana");
        System.out.println(p1);
        System.out.println(p2);
        System.out.println(p3);
        p2.sirve();
        System.out.println("pedidas: " + Pizza.getTotalPedidas());
        System.out.println("servidas: " + Pizza.getTotalServidas());
    }
}
```

```
pizza margarita mediana, pedida
pizza funghi familiar, servida
pizza cuatro quesos mediana, pedida
esa pizza ya se ha servido
pedidas: 3
servidas: 1
```



Ejercicio 6

Crea la clase Tiempo con los métodos suma y resta. Los objetos de la clase Tiempo son intervalos de tiempo y se crean de la forma `Tiempo t = new Tiempo(1, 20, 30)` donde los parámetros que se le pasan al constructor son las horas, los minutos y los segundos respectivamente. Crea el método `toString` para ver los intervalos de tiempo de la forma 10h 35m 5s. Si se suman por ejemplo 30m 40s y 35m 20s el resultado debería ser 1h 6m 0s. Realiza un programa de prueba para comprobar que la clase funciona bien.



Ejercicio 7

Queremos gestionar la venta de entradas (no numeradas) de **Expocoches Campanillas** que tiene 3 zonas, la sala principal con 1000 entradas disponibles, la zona de compra-venta con 200 entradas disponibles y la zona vip con 25 entradas disponibles. Hay que controlar que existen entradas antes de venderlas.

La clase Zona con sus atributos y métodos se muestra a continuación:

```
/**
 * Definición de la clase Zona
 *
 * @author Luis José Sánchez
 */
public class Zona {

    private int entradasPorVender;

    public Zona(int n){
        entradasPorVender = n;
    }

    public int getEntradasPorVender() {
        return entradasPorVender;
    }

    /**
     * Vende un número de entradas.
     * <p>
     * Comprueba si quedan entradas libres antes de realizar la venta.
     *
     * @param n número de entradas a vender
     */
    public void vender(int n) {

        if (this.entradasPorVender == 0) {
            System.out.println("Lo siento, las entradas para esa zona están agotadas.");
        } else if (this.entradasPorVender < n) {
            System.out.println("Sólo me quedan " + this.entradasPorVender
                               + " entradas para esa zona.");
        }

        if (this.entradasPorVender >= n) {
            entradasPorVender -= n;
            System.out.println("Aquí tiene sus " + n + " entradas, gracias.");
        }
    }
}
```

El menú del programa debe ser el que se muestra a continuación. Cuando elegimos la opción 2, se nos debe preguntar para qué zona queremos las entradas y cuántas queremos. Lógicamente, el programa debe controlar que no se puedan vender más entradas de la cuenta.

1. Mostrar número de entradas libres
2. Vender entradas
3. Salir

9.9.3 Arrays de objetos



Ejercicio 1

Utiliza la clase `Gato` para crear un array de cuatro gatos e introduce los datos de cada uno de ellos mediante un bucle. Muestra a continuación los datos de todos los gatos utilizando también un bucle.



Ejercicio 2

Cambia el programa anterior de tal forma que los datos de los gatos se introduzcan directamente en el código de la forma `gatito[2].setColor("marrón")` o bien mediante el constructor, de la forma `gatito[3] = new Gato("Garfield", "naranja", "macho")`. Muestra a continuación los datos de todos los gatos utilizando un bucle.



Ejercicio 3

Realiza el programa "Colección de discos" por tu cuenta, mirando lo menos posible el ejemplo que se proporciona. Pruébalo primero para ver cómo funciona y luego intenta implementarlo tú mismo.



Ejercicio 4

Modifica el programa "Colección de discos" como se indica a continuación:

- Mejora la opción "Nuevo disco" de tal forma que cuando se llenen todas las posiciones del array, el programa muestre un mensaje de error. No se permitirá introducir los datos de ningún disco hasta que no se borre alguno de la lista.
- Mejora la opción "Borrar" de tal forma que se verifique que el código introducido por el usuario existe.
- Modifica el programa de tal forma que el código del disco sea único, es decir que no se pueda repetir.
- Crea un submenú dentro de "Listado" de tal forma que exista un listado completo, un listado por autor (todos los discos que ha publicado un determinado autor), un listado por género (todos los discos de un género determinado) y un listado de discos cuya duración esté en un rango determinado por el usuario.



Ejercicio 5

Crea el programa GESTISIMAL (GESTIÓN SIMPLIFICADA de Almacén) para llevar el control de los artículos de un almacén. De cada artículo se debe saber el código, la descripción, el precio de compra, el precio de venta y el stock (número de unidades). El menú del programa debe tener, al menos, las siguientes opciones:

1. Listado
2. Alta
3. Baja
4. Modificación
5. Entrada de mercancía
6. Salida de mercancía
7. Salir

La entrada y salida de mercancía supone respectivamente el incremento y decremento de stock de un determinado artículo. Hay que controlar que no se pueda sacar más mercancía de la que hay en el almacén.

10. Colecciones y diccionarios

10.1 Colecciones: la clase `ArrayList`

Una colección en Java es una estructura de datos que permite almacenar muchos valores del mismo tipo; por tanto, conceptualmente es prácticamente igual que un *array*. Según el uso y según si se permiten o no repeticiones, Java dispone de un amplio catálogo de colecciones: `ArrayList` (lista), `ArrayBlockingQueue` (cola), `HashSet` (conjunto), `Stack` (pila), etc. En este manual estudiaremos la colección `ArrayList`.

Un `ArrayList` es una estructura en forma de lista que permite almacenar elementos del mismo tipo (pueden ser incluso objetos); su tamaño va cambiando a medida que se añaden o se eliminan esos elementos.

Nos podemos imaginar un `ArrayList` como un conjunto de celdas o cajoncitos donde se guardan los valores, exactamente igual que un *array* convencional. En la práctica será más fácil trabajar con un `ArrayList`.

En capítulos anteriores hemos podido comprobar la utilidad del *array*; es un recurso imprescindible que cualquier programador debe manejar con soltura. No obstante, el *array* presenta algunos inconvenientes. Uno de ellos es la necesidad de conocer el tamaño exacto en el momento de su creación. Una colección, sin embargo, se crea sin que se tenga que especificar el tamaño; posteriormente se van añadiendo y quitando elementos a medida que se necesitan.

Trabajando con *arrays* es frecuente cometer errores al utilizar los índices; por ejemplo al intentar guardar un elemento en una posición que no existe (índice fuera de rango). Aunque las colecciones permiten el uso de índices, no es necesario indicarlos siempre. Por ejemplo, en una colección del tipo `ArrayList`, cuando hay que añadir el elemento "Amapola", se puede hacer simplemente `flores.add("Amapola")`. Al no especificar índice, el elemento "Amapola" se añadiría justo al final de `flores` independientemente del tamaño y del número de elementos que se hayan introducido ya.

La clase `ArrayList` es muy similar a la clase `Vector`. Ésta última está obsoleta y, por tanto, no se recomienda su uso.

10.1.1 Principales métodos de `ArrayList`

Las operaciones más comunes que se pueden realizar con un objeto de la clase `ArrayList` son las siguientes:

`add(elemento)`

Añade un elemento al final de la lista.

add(indice, elemento)

Inserta un elemento en una posición determinada, desplazando el resto de elementos hacia la derecha.

clear()

Elimina todos los elementos pero no borra la lista.

contains(elemento)

Devuelve `true` si la lista contiene el elemento que se especifica y `false` en caso contrario.

get(indice)

Devuelve el elemento de la posición que se indica entre paréntesis.

indexOf(elemento)

Devuelve la posición de la primera ocurrencia del elemento que se indica entre paréntesis.

isEmpty()

Devuelve `true` si la lista está vacía y `false` en caso de tener algún elemento.

remove(indice)

Elimina el elemento que se encuentra en una posición determinada.

remove(elemento)

Elimina la primera ocurrencia de un elemento.

set(indice, elemento)

Machaca el elemento que se encuentra en una determinada posición con el elemento que se pasa como parámetro.

size()

Devuelve el tamaño (número de elementos) de la lista.

toArray()

Devuelve un *array* con todos y cada uno de los elementos que contiene la lista.

Puedes consultar todos los métodos disponibles en la [documentación oficial de la clase ArrayList](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)¹.

¹<http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>

10.1.2 Definición de un ArrayList e inserción, borrado y modificación de sus elementos

A continuación se muestra un ejemplo en el que se puede ver cómo se declara un ArrayList y cómo se insertan y se extraen elementos.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList01 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        System.out.println("Nº de elementos: " + a.size());

        a.add("rojo");
        a.add("verde");
        a.add("azul");

        System.out.println("Nº de elementos: " + a.size());

        a.add("blanco");

        System.out.println("Nº de elementos: " + a.size());

        System.out.println("El elemento que hay en la posición 0 es " + a.get(0));
        System.out.println("El elemento que hay en la posición 3 es " + a.get(3));
    }
}
```

Observa que al crear un objeto de la clase ArrayList hay que indicar el tipo de dato que se almacenará en las celdas de esa lista. Para ello se utilizan los caracteres < y >. No hay que olvidar los paréntesis del final.



Es necesario importar la clase ArrayList para poder crear objetos de esta clase, para ello debe aparecer al principio del programa la línea `import java.util.ArrayList;`. Algunos IDEs (por ej. Netbeans) insertan esta línea de código de forma automática.

En el siguiente ejemplo se muestra un ArrayList de números enteros.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList011 {
    public static void main(String[] args) {

        ArrayList<Integer> a = new ArrayList<Integer>();

        a.add(18);
        a.add(22);
        a.add(-30);

        System.out.println("Nº de elementos: " + a.size());

        System.out.println("El elemento que hay en la posición 1 es " + a.get(1));
    }
}
```

Se define la estructura de la siguiente manera:

```
ArrayList<Integer> a = new ArrayList<Integer>();
```

Fíjate que no se utiliza el tipo simple `int` sino el wrapper `Integer`. Recuerda que los wrapper son clases que engloban a los tipos simples y les añaden nuevas funcionalidades (p. ej. permiten tratar a las variables numéricas como objetos). El wrapper de `int` es `Integer`, el de `float` es `Float`, el de `double` es `Double`, el de `long` es `Long`, el de `boolean` es `Boolean` y el de `char` es `Character`.

En el siguiente ejemplo podemos ver cómo extraer todos los elementos de una lista a la manera tradicional, con un bucle `for`.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList02 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add("rojo");
        a.add("verde");
        a.add("azul");
        a.add("blanco");
        a.add("amarillo");

        System.out.println("Contenido de la lista: ");

        for(int i=0; i < a.size(); i++) {
            System.out.println(a.get(i));
        }
    }
}
```

Si estás acostumbrado al for clásico, habrás visto que es muy sencillo recorrer todos los elementos del ArrayList. No obstante, al trabajar con colecciones es recomendable usar el for al estilo foreach como se muestra en el siguiente ejemplo. Como puedes ver, la sintaxis es más corta y no se necesita crear un índice para recorrer la estructura.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList03 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();
```

```
a.add("rojo");
a.add("verde");
a.add("azul");
a.add("blanco");
a.add("amarillo");

System.out.println("Contenido de la lista: ");

for(String color: a) {
    System.out.println(color);
}
}
```

Fíjate en estas líneas:

```
for(String color: a) {
    System.out.println(color);
}
```

El significado de este código sería el siguiente:

```
for(String color: a)
```

→ Saca uno a uno todos los elementos de a y ve metiéndolos en una variable de nombre color.

```
System.out.println(color);
```

→ Muestra por pantalla el contenido de la variable color.

Veamos ahora en otro ejemplo cómo eliminar elementos de un `ArrayList`. Se utiliza el método `remove()` y se puede pasar como parámetro el índice del elemento que se quiere eliminar, o bien el valor del elemento. O sea, `a.remove(2)` elimina el elemento que se encuentra en la posición 2 de a mientras que `a.remove("blanco")` elimina el valor "blanco".

Es importante destacar que el `ArrayList` se reestructura de forma automática después del borrado de cualquiera de sus elementos.


```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList04 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add("rojo");
        a.add("verde");
        a.add("azul");
        a.add("blanco");
        a.add("amarillo");
        a.add("blanco");

        System.out.println("Contenido de la lista: ");

        for(String color: a) {
            System.out.println(color);
        }

        if (a.contains("blanco")) {
            System.out.println("El blanco está en la lista de colores");
        }

        a.remove("blanco");

        System.out.println("Contenido de la lista después de quitar la " +
            "primera ocurrencia del color blanco: ");

        for(String color: a) {
            System.out.println(color);
        }

        a.remove(2);
        System.out.println("Contenido de la lista después de quitar el " +
            "elemento de la posición 2: ");

        for(String color: a) {
            System.out.println(color);
        }
    }
}
```

```
    }  
  }  
}
```

A continuación se muestra un ejemplo en el que se “machaca” una posición del `ArrayList`. Al hacer `a.set(2, "turquesa")`, se borra lo que hubiera en la posición 2 y se coloca el valor "turquesa". Sería equivalente a hacer `a[2] = "turquesa"` en caso de que `a` fuese un *array* tradicional en lugar de un `ArrayList`.

```
/**  
 * Ejemplo de uso de la clase ArrayList  
 *  
 * @author Luis José Sánchez  
 */  
  
import java.util.ArrayList;  
  
public class EjemploArrayList05 {  
    public static void main(String[] args) {  
  
        ArrayList<String> a = new ArrayList<String>();  
  
        a.add("rojo");  
        a.add("verde");  
        a.add("azul");  
        a.add("blanco");  
        a.add("amarillo");  
  
        System.out.println("Contenido del vector: ");  
  
        for(String color: a)  
            System.out.println(color);  
  
        a.set(2, "turquesa");  
  
        System.out.println("Contenido del vector después de machacar la posición 2: ");  
  
        for(String color: a) {  
            System.out.println(color);  
        }  
    }  
}
```

El método `add` permite añadir elementos a un `ArrayList` como ya hemos visto. Por ejemplo, `a.add("amarillo")` añade el elemento "amarillo" al final de `a`. Este método se puede utilizar

también con un índice de la forma `a.add(1, "turquesa")`. En este caso, lo que se hace es insertar en la posición indicada. Lo mejor de todo es que el `ArrayList` se reestructura de forma automática desplazando el resto de elementos.

```
/**
 * Ejemplo de uso de la clase ArrayList
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList06 {
    public static void main(String[] args) {

        ArrayList<String> a = new ArrayList<String>();

        a.add("rojo");
        a.add("verde");
        a.add("azul");
        a.add("blanco");
        a.add("amarillo");

        System.out.println("Contenido de la lista: ");

        for(String color: a) {
            System.out.println(color);
        }

        a.add(1, "turquesa");

        System.out.println("Contenido del vector después de insertar en la posición 1: ");

        for(String color: a) {
            System.out.println(color);
        }
    }
}
```

10.1.3 ArrayList de objetos

Una colección `ArrayList` puede contener objetos que son instancias de clases definidas por el programador. Esto es muy útil sobre todo en aplicaciones de gestión para guardar datos de alumnos, productos, libros, etc.

En el siguiente ejemplo, definimos una lista de gatos. En cada celda de la lista se almacenará un objeto de la clase Gato.

```
/**
 * Uso de un ArrayList de objetos
 *
 * @author Luis José Sánchez
 */

import java.util.ArrayList;

public class EjemploArrayList07 {
    public static void main(String[] args) {

        ArrayList<Gato> g = new ArrayList<Gato>();

        g.add(new Gato("Garfield", "naranja", "mestizo"));
        g.add(new Gato("Pepe", "gris", "angora"));
        g.add(new Gato("Mauri", "blanco", "manx"));
        g.add(new Gato("Ulises", "marrón", "persa"));

        System.out.println("\nDatos de los gatos:\n");

        for (Gato gatoAux: g) {
            System.out.println(gatoAux+"\n");
        }
    }
}
```

En el siguiente apartado se muestra la definición de la clase Gato.

10.1.4 Ordenación de un ArrayList

Los elementos de una lista se pueden ordenar con el método sort. El formato es el siguiente:

```
Collections.sort(lista);
```

Observa que sort es un método de clase que está definido en Collections. Para poder utilizar este método es necesario incluir la línea

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

al principio del programa. A continuación se muestra un ejemplo del uso de sort.

```
/**
 * Ordenación de un ArrayList
 *
 * @author Luis José Sánchez
 */

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

public class EjemploArrayList071 {
    public static void main(String[] args) {

        ArrayList<Integer> a = new ArrayList<Integer>();

        a.add(67);
        a.add(78);
        a.add(10);
        a.add(4);

        System.out.println("\nNúmeros en el orden original:");
        for (int numero: a) {
            System.out.println(numero);
        }

        Collections.sort(a);

        System.out.println("\nNúmeros ordenados:");
        for (int numero: a) {
            System.out.println(numero);
        }
    }
}
```

También es posible ordenar una lista de objetos. En este caso es necesario indicar el criterio de ordenación en la definición de la clase. En el programa principal, se utiliza el método `sort` igual que si se tratase de una lista de números o de palabras como se muestra a continuación.

```

/**
 * Ordenación de un ArrayList de objetos
 *
 * @author Luis José Sánchez
 */

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

public class EjemploArrayList08 {
    public static void main(String[] args) {

        ArrayList<Gato> g = new ArrayList<Gato>();

        g.add(new Gato("Garfield", "naranja", "mestizo"));
        g.add(new Gato("Pepe", "gris", "angora"));
        g.add(new Gato("Mauri", "blanco", "manx"));
        g.add(new Gato("Ulises", "marrón", "persa"));
        g.add(new Gato("Adán", "negro", "angora"));

        Collections.sort(g);

        System.out.println("\nDatos de los gatos ordenados por nombre:");

        for (Gato gatoAux: g) {
            System.out.println(gatoAux+"\n");
        }
    }
}

```

Ahora bien, en la definición de la clase Gato hay que indicar de alguna manera cómo se debe realizar la ordenación, ya que Java no sabe de antemano si los gatos se ordenan según el color, el nombre, el peso, etc.

Lo primero que hay que hacer es indicar que los objetos de la clase Gato se pueden comparar unos con otros. Para ello, cambiamos la siguiente línea:

```
public class Gato
```

por esta otra:

```
public class Gato implements Comparable<Gato>
```

Lo siguiente y no menos importante es definir el método `compareTo`. Este método debe devolver un 0 si los elementos que se comparan son iguales, un número negativo si el primer elemento que se

compara es menor que el segundo y un número positivo en caso contrario. Afortunadamente, las clases String, Integer, Double, etc. ya tienen implementado su propio método compareTo así que tenemos hecho lo más difícil. Lo único que deberemos escribir en nuestro código es un compareTo con los atributos que queremos comparar.

En el caso que nos ocupa, si queremos ordenar los gatos por nombre, tendremos que implementar el compareTo de la clase Gato de tal forma que nos devuelva el resultado del compareTo de los nombres de los gatos que estamos comparando, de la siguiente manera:

```
public int compareTo(Gato g) {  
    return (this.nombre).compareTo(g.getNombre());  
}
```

Si en lugar de ordenar por nombre, quisiéramos ordenar por raza, el método compareTo de la clase Gato sería el siguiente:

```
public int compareTo(Gato g) {  
    return (this.raza).compareTo(g.getRaza());  
}
```

A continuación se muestra la definición completa de la clase Gato.

```
/**  
 * Definición de la clase Gato  
 *  
 * @author  
 */  
  
public class Gato implements Comparable<Gato> {  
    private String nombre;  
    private String color;  
    private String raza;  
  
    public Gato(String nombre, String color, String raza) {  
        this.nombre = nombre;  
        this.color = color;  
        this.raza = raza;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public String getRaza() {
```

```
        return raza;
    }

    public String toString() {
        return "Nombre: " + this.nombre + "\nColor: " + this.color + "\nRaza: " + this.raza;
    }

    public int compareTo(Gato g) {
        return (this.nombre).compareTo(g.getNombre());
    }

    public boolean equals(Gato g) {
        return (this.nombre).equals(g.getNombre());
    }
}
```

10.2 Diccionarios: la clase HashMap

Imagina un diccionario inglés-español. Queremos saber qué significa la palabra “*stiff*”. Sabemos que en el diccionario hay muchas entradas y en cada entrada tenemos una palabra en inglés y su correspondiente traducción al español. Buscando por la “s” encontramos que “*stiff*” significa “agujetas”.

Un diccionario en Java funciona exactamente igual. Contiene una serie de elementos que son las entradas que a su vez están formadas por un par (clave, valor). La clave (*key*) permite acceder al valor. No puede haber claves duplicadas. En el ejemplo anterior, la clave sería “*stiff*” y el valor “agujetas”.

Java dispone de varios tipos de diccionarios: HashMap, EnumMap, Hashtable, IdentityHashMap, LinkedHashMap, etc. Nosotros estudiaremos el diccionario HashMap.

10.2.1 Principales métodos de HashMap

Algunos de los métodos más importantes de la clase HasMap son:

get(clave)

Obtiene el valor correspondiente a una clave. Devuelve null si no existe esa clave en el diccionario.

put(clave, valor)

Añade un par (clave, valor) al diccionario. Si ya había un valor para esa clave, se machaca.

keySet()

Devuelve un conjunto (set) con todas las claves.

values()

Devuelve una colección con todos los valores (los valores pueden estar duplicados a diferencia de las claves).

entrySet()

Devuelve una colección con todos los pares (clave, valor).

containsKey(clave)

Devuelve `true` si el diccionario contiene la clave indicada y `false` en caso contrario.

getKey()

Devuelve la clave de la entrada. Se aplica a una sola entrada del diccionario (no al diccionario completo), es decir a una pareja (clave, valor).

Por ejemplo:

```
for (Map.Entry pareja: m.entrySet()) {  
    System.out.println(pareja.getKey());  
}
```

getValue()

Devuelve el contenido de la entrada. Se aplica a una entrada del diccionario (no al diccionario completo), es decir a una pareja (clave, valor).

Por ejemplo:

```
for (Map.Entry pareja: m.entrySet()) {  
    System.out.println(pareja.getValue());  
}
```

10.2.2 Definición de un HashMap e inserción, borrado y modificación de entradas

Al declarar un diccionario hay que indicar los tipos tanto de la clave como del valor. En el siguiente ejemplo definimos el diccionario `m` que tendrá como clave un número entero y una cadena de caracteres como valor. Este diccionario se declara de esta forma:

```
HashMap<Integer, String> m = new HashMap<Integer, String>();
```

No hay que olvidar importar la clase al principio del programa:

```
import java.util.HashMap;
```

Para insertar una entrada en el diccionario se utiliza el método `put` indicando siempre la clave y el valor. Veamos un ejemplo completo.

```
/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.HashMap;

public class EjemploHashMap01 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.println("Los elementos de m son: \n" + m);
    }
}
```

Para extraer valores se utiliza el método `get`. Se proporciona una clave y el diccionario nos devuelve el valor, igual que un diccionario de verdad. Si no existe ninguna entrada con la clave que se indica, se devuelve `null`.

```
/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.HashMap;

public class EjemploHashMap011 {
    public static void main(String[] args) {
```

```

HashMap<Integer, String> m = new HashMap<Integer, String>();

m.put(924, "Amalia Núñez");
m.put(921, "Cindy Nero");
m.put(700, "César Vázquez");
m.put(219, "Víctor Tilla");
m.put(537, "Alan Brito");
m.put(605, "Esteban Quito ");

System.out.println(m.get(921));
System.out.println(m.get(605));
System.out.println(m.get(888));
}
}

```

¿Y si queremos extraer todas las entradas? Tenemos varias opciones. Podemos usar el método `print` directamente sobre el diccionario de la forma `System.out.print(diccionario)` como vimos en un ejemplo anterior; de esta manera se muestran por pantalla todas las entradas encerradas entre llaves. También podemos convertir el diccionario en un `entrySet` (conjunto de entradas) y mostrarlo con `print`; de esta forma se obtiene una salida por pantalla muy parecida a la primera (en lugar de llaves se muestran corchetes). Otra opción es utilizar un `for` para recorrer una a una todas las entradas. En este último caso hay que convertir el diccionario en un `entrySet` ya que no se pueden sacar las entradas directamente del diccionario. Estas dos últimas opciones se ilustran en el siguiente ejemplo.

```

/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.HashMap;
import java.util.Map;

public class EjemploHashMap02 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");
    }
}

```

```

System.out.println("Todas las entradas del diccionario extraídas con entrySet:");
System.out.println(m.entrySet());

System.out.println("\nEntradas del diccionario extraídas una a una:");
for (Map.Entry pareja: m.entrySet()) {
    System.out.println(pareja);
}
}
}

```

A continuación se muestra el uso de los métodos `getKey` y `getValue` que extraen la clave y el valor de una entrada respectivamente.

```

/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.*;

public class EjemploHashMap03 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.println("Código\tNombre\n-----\t-----");

        for (Map.Entry pareja: m.entrySet()) {
            System.out.print(pareja.getKey() + "\t");
            System.out.println(pareja.getValue());
        }
    }
}

```

En el último programa de ejemplo hacemos uso del método `containsKey` que nos servirá para saber si existe o no una determinada clave en un diccionario y del método `get` que, como ya hemos visto, sirve para extraer un valor a partir de su clave.

```
/**
 * Ejemplo de uso de la clase HashMap
 *
 * @author Luis José Sánchez
 */

import java.util.*;

public class EjemploHashMap04 {
    public static void main(String[] args) {

        HashMap<Integer, String> m = new HashMap<Integer, String>();

        m.put(924, "Amalia Núñez");
        m.put(921, "Cindy Nero");
        m.put(700, "César Vázquez");
        m.put(219, "Víctor Tilla");
        m.put(537, "Alan Brito");
        m.put(605, "Esteban Quito ");

        System.out.print("Por favor, introduzca un código: ");
        int codigoIntroducido = Integer.parseInt(System.console().readLine());

        if (m.containsKey(codigoIntroducido)) {
            System.out.print("El código " + codigoIntroducido + " corresponde a ");
            System.out.println(m.get(codigoIntroducido));
        } else {
            System.out.print("El código introducido no existe.");
        }
    }
}
```

10.3 Ejercicios



Ejercicio 1

Crea un `ArrayList` con los nombres de 6 compañeros de clase. A continuación, muestra esos nombres por pantalla. Utiliza para ello un bucle `for` que recorra todo el `ArrayList` sin usar ningún índice.



Ejercicio 2

Realiza un programa que introduzca valores aleatorios (entre 0 y 100) en un `ArrayList` y que luego calcule la suma, la media, el máximo y el mínimo de esos números. El tamaño de la lista también será aleatorio y podrá oscilar entre 10 y 20 elementos ambos inclusive.



Ejercicio 3

Escribe un programa que ordene 10 números enteros introducidos por teclado y almacenados en un objeto de la clase `ArrayList`.



Ejercicio 4

Realiza un programa equivalente al anterior pero en esta ocasión, el programa debe ordenar palabras en lugar de números.



Ejercicio 5

Realiza de nuevo el ejercicio de la colección de discos pero utilizando esta vez una lista para almacenar la información sobre los discos en lugar de un *array* convencional. Comprobarás que el código se simplifica notablemente ¿Cuánto ocupa el programa original hecho con un *array*? ¿Cuánto ocupa este nuevo programa hecho con una lista?



Ejercicio 6

Implementa el control de acceso al área restringida de un programa. Se debe pedir un nombre de usuario y una contraseña. Si el usuario introduce los datos correctamente, el programa dirá “Ha accedido al área restringida”. El usuario tendrá un máximo de 3 oportunidades. Si se agotan las oportunidades el programa dirá “Lo siento, no tiene acceso al área restringida”. Los nombres de usuario con sus correspondientes contraseñas deben estar almacenados en una estructura de la clase `HashMap`.



Ejercicio 7

La máquina *Eurocoin* genera una moneda de curso legal cada vez que se pulsa un botón siguiendo la siguiente pauta: o bien coincide el valor con la moneda anteriormente generada - 1 céntimo, 2 céntimos, 5 céntimos, 10 céntimos, 25 céntimos, 50 céntimos, 1 euro o 2 euros - o bien coincide la posición - cara o cruz. Simula, mediante un programa, la generación de 6 monedas aleatorias siguiendo la pauta correcta. Cada moneda generada debe ser una instancia de la clase *Moneda* y la secuencia se debe ir almacenando en una lista.

Ejemplo:

```
2 céntimos - cara
2 céntimos - cruz
50 céntimos - cruz
1 euro - cruz
1 euro - cara
10 céntimos - cara
```



Ejercicio 8

Realiza un programa que escoja al azar 10 cartas de la baraja española (10 objetos de la clase *Carta*). Emplea un objeto de la clase *ArrayList* para almacenarlas y asegúrate de que no se repite ninguna.



Ejercicio 9

Modifica el programa anterior de tal forma que las cartas se muestren ordenadas. Primero se ordenarán por palo: bastos, copas, espadas, oros. Cuando coincida el palo, se ordenará por número: as, 2, 3, 4, 5, 6, 7, sota, caballo, rey.



Ejercicio 10

Crea un mini-diccionario español-inglés que contenga, al menos, 20 palabras (con su correspondiente traducción). Utiliza un objeto de la clase *HashMap* para almacenar las parejas de palabras. El programa pedirá una palabra en español y dará la correspondiente traducción en inglés.



Ejercicio 11

Realiza un programa que escoja al azar 5 palabras en español del mini-diccionario del ejercicio anterior. El programa irá pidiendo que el usuario teclee la traducción al inglés de cada una de las palabras y comprobará si son correctas. Al final, el programa deberá mostrar cuántas respuestas son válidas y cuántas erróneas.



Ejercicio 12

Escribe un programa que genere una secuencia de 5 cartas de la baraja española y que sume los puntos según el juego de la brisca. El valor de las cartas se debe guardar en una estructura `HashMap` que debe contener parejas (figura, valor), por ejemplo (“caballo”, 3). La secuencia de cartas debe ser una estructura de la clase `ArrayList` que contiene objetos de la clase `Carta`. El valor de las cartas es el siguiente: as → 11, tres → 10, sota → 2, caballo → 3, rey → 4; el resto de cartas no vale nada.

```
Ejemplo:  
as de oros  
cinco de bastos  
caballo de espadas  
sota de copas  
tres de oros  
Tienes 26 puntos
```



Ejercicio 13

Modifica el programa **Gestisimal** realizado anteriormente añadiendo las siguientes mejoras:

- Utiliza una lista en lugar de un *array* para el almacenamiento de los datos.
- Comprueba la existencia del código en el alta, la baja y la modificación de artículos para evitar errores.
- Cambia la opción “Salida de stock” por “Venta”. Esta nueva opción permitirá hacer una venta de varios artículos y emitir la factura correspondiente. Se debe preguntar por los códigos y las cantidades de cada artículo que se quiere comprar. Aplica un 21% de IVA.

11. Ficheros de texto y paso de parámetros por línea de comandos

Mediante un programa en Java se puede acceder al contenido de un fichero grabado en un dispositivo de almacenamiento (por ejemplo en el disco duro) tanto para leer como para escribir (grabar) datos.

La información contenida en las variables, los arrays, los objetos o cualquier otra estructura de datos es volátil, es decir, se pierde cuando se cierra el programa. Los ficheros permiten tener ciertos datos almacenados y disponibles en cualquier momento para cuando nuestro programa los necesite.

Pensemos por ejemplo en un juego. Podríamos utilizar un fichero para guardar el *ranking* de jugadores con las mejores puntuaciones. De esta manera, estos datos no se perderían al salir del juego. De igual forma, en un programa de gestión, puede ser útil tener un fichero de configuración con datos como el número máximo de elementos que se muestran en un listado o los distintos tipos de IVA aplicables a las facturas. Al modificar esta información, quedará almacenada en el fichero correspondiente y no se perderá cuando se cierre el programa.



Cuando un programa se cierra, se pierde la información almacenada en variables, arrays, objetos o cualquier otra estructura. Si queremos conservar ciertos datos, debemos guardarlos en ficheros.

Hay programas que hacen un uso intensivo de datos. Por ejemplo, la aplicación **Séneca** - que, por cierto, está hecha en Java - lleva el control de la matriculación de cientos de miles de alumnos y la gestión de decenas de miles de profesores. En este caso y en otros similares se utiliza un sistema gestor de bases de datos. El acceso a bases de datos mediante Java lo estudiaremos en el [capítulo 13](#).



La creación y uso de ficheros desde un programa en Java se lleva a cabo cuando hay poca información que almacenar o cuando esa información es heterogénea. En los casos en que la información es abundante y homogénea es preferible usar una base de datos relacional (por ejemplo MySQL) en lugar de ficheros.

En este capítulo estudiaremos, además del acceso a ficheros desde un programa en Java, el paso de parámetros desde la línea de comandos. Son cosas diferentes y no hay que utilizarlas juntas necesariamente pero, como podrás comprobar, se combinan muy bien, por eso hemos incluido estos dos recursos de Java en un mismo capítulo.

11.1 Lectura de un fichero de texto

Aunque Java puede manejar también ficheros binarios, vamos a centrarnos exclusivamente en la utilización de ficheros de texto. Los ficheros de texto tienen una gran ventaja, se pueden crear y manipular mediante cualquier editor como por ejemplo **GEdit**.

Siempre que vayamos a usar ficheros, deberemos incluir al principio del programa una o varias líneas para cargar las clases necesarias. Aunque se pueden cargar varias clases en una sola línea usando el asterisco de la siguiente manera:

```
import java.io.*;
```

nosotros no lo haremos así, ya que nuestro código sigue las normas del estándar de Google y estas normas lo prohíben taxativamente. Se importarán las clases usando varias líneas, una línea por cada clase que se importa en el programa; de esta forma:

```
import java.io.BufferedReader;  
import java.io.FileReader;
```

No es necesario saber de memoria los nombres de las clases, el entorno de desarrollo **Netbeans** detecta qué clases hacen falta cargar y añade los `import` de forma automática.

Todas las operaciones que se realicen sobre ficheros deberán estar incluidas en un bloque `try-catch`. Esto nos permitirá mostrar mensajes de error y terminar el programa de una forma ordenada en caso de que se produzca algún fallo - el fichero no existe, no tenemos permiso para acceder a él, etc.

El bloque `try-catch` tiene el siguiente formato:

```
try {  
  
    Operaciones_con_fichero  
  
} catch (tipo_de_error nombre_de_variable) {  
  
    Mensaje_de_error  
  
}
```

Tanto para leer como para escribir utilizamos lo que en programación se llama un “manejador de fichero”. Es algo así como una variable que hace referencia al fichero con el que queremos trabajar.

En nuestro ejemplo, el manejador de fichero se llama `bf` y se crea de la siguiente manera:

```
BufferedReader bf = new BufferedReader(new FileReader("malaga1.txt"));
```

El fichero con el que vamos a trabajar tiene por nombre `malaga1.txt` y contiene información extraída de la Wikipedia sobre la bonita ciudad de Málaga¹. A partir de aquí, siempre que queramos realizar una operación sobre ese archivo, utilizaremos su manejador de fichero asociado, es decir `bf`.

En el ejemplo que se muestra a continuación, el programa lee el contenido del fichero `malaga1.txt` y lo muestra tal cual en pantalla, igual que si hiciéramos en una ventana de terminal `cat malaga1.txt`.

Observa que se va leyendo el fichero línea a línea mediante `bf.readLine()` hasta que se acaban las líneas. Cuando no quedan más líneas por leer se devuelve el valor `null`.

```
/**
 * Ejemplo de uso de la clase File
 * Lectura de un fichero de texto
 *
 * @author Luis José Sánchez
 */

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

class EjemploFichero01 {

    public static void main(String[] args) {

        try {
            BufferedReader bf = new BufferedReader(new FileReader("malaga1.txt"));

            String linea = "";

            while (linea != null) {
                System.out.println(linea);
                linea = bf.readLine();
            }

            bf.close();

        } catch (FileNotFoundException e) { // qué hacer si no se encuentra el fichero
            System.out.println("No se encuentra el fichero malaga.txt");
        } catch (IOException e) { // qué hacer si hay un error en la lectura del fichero
            System.out.println("No se puede leer el fichero malaga.txt");
        }
    }
}
```

¹<http://es.wikipedia.org/wiki/M%C3%A1laga>

```

    }
}
}

```

Es importante “cerrar el fichero” cuando se han realizado todas las operaciones necesarias sobre él. En este ejemplo, esta acción se ha llevado a cabo con la sentencia `bf.close()`.

A continuación se muestra un programa un poco más complejo. Se trata de una aplicación que pide por teclado un nombre de fichero. Previamente en ese fichero (por ejemplo `numeros.txt`) habremos introducido una serie de números, a razón de uno por línea. Se podrían leer también los números si estuvieran separados por comas o espacios aunque sería un poco más complicado (no mucho más). Los números pueden contener decimales ya que se van a leer como `Double`. Cada número que se lee del fichero se va sumando de tal forma que la suma total estará contenida en la variable `suma`; a la par se va llevando la cuenta de los elementos que se van leyendo en la variable `i`. Finalmente, dividiendo la suma total entre el número de elementos obtenemos la media aritmética de los números contenidos en el fichero.

```

/**
 * Ejemplo de uso de la clase File
 * Calcula la media de los números que se encuentran en un fichero de texto
 *
 * @author Luis José Sánchez
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class EjemploFichero08 {

    public static void main(String[] args) {

        System.out.print("Introduzca el nombre del archivo donde se encuentran los números: ");
        String nombreFichero = System.console().readLine();

        try {
            BufferedReader bf = new BufferedReader(new FileReader(nombreFichero));

            String linea = "";
            int i = 0;
            double suma = 0;

            while (linea != null) {
                i++;

```

```
        suma += Double.parseDouble(linea);
        linea = bf.readLine();
    }
    i--;

    bf.close();

    System.out.println("La media es " + suma / (double)i);

} catch (IOException e) {
    System.out.println(e.getMessage());
}
}
```

11.2 Escritura sobre un fichero de texto

La escritura en un fichero de texto es, si cabe, más fácil que la lectura. Solo hay que cambiar `System.out.print("texto")` por `manejador.write("texto")`. Se pueden incluir saltos de línea, tabuladores y espacios igual que al mostrar un mensaje por pantalla.

Es importante ejecutar `close()` después de realizar la escritura; de esta manera nos aseguramos que se graba toda la información en el disco.

Al realizar escrituras en ficheros con Java hay que tener ciertas precauciones. Cuando toca dar este tema en clase es frecuente que a más de un alumno le empiece a ir lento el ordenador, luego se le queda inutilizado y, por último, ni siquiera le arranca ¿qué ha pasado? Pues que ha estado escribiendo datos en ficheros y por alguna razón, su programa se ha metido en un bucle infinito lo que da como resultado cuelgues y ficheros de varios gigabytes de basura. Por eso es muy importante asegurarse bien de que la información que se va a enviar a un fichero es la correcta ¿cómo? muy fácil, enviándola primero a la pantalla.



Primero a pantalla y luego a fichero

Envía primero a la pantalla todo lo que quieras escribir en el fichero. Cuando compruebes que lo que se ve por pantalla es realmente lo que quieres grabar en el fichero, entonces y solo entonces, cambia `System.out.print("texto")` por `manejador.write("texto")`.

A continuación se muestra un programa de ejemplo que crea un fichero de texto y luego escribe en él tres palabras, una por cada línea.

```
/**
 * Ejemplo de uso de la clase File
 * Escritura en un fichero de texto
 *
 * @author Luis José Sánchez
 */

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

class EjemploFichero02 {

    public static void main(String[] args) {

        try {
            BufferedWriter bw = new BufferedWriter(new FileWriter("fruta.txt"));

            bw.write("naranja\n");
            bw.write("mango\n");
            bw.write("chirimoya\n");

            bw.close();

        } catch (IOException ioe) {
            System.out.println("No se ha podido escribir en el fichero");
        }
    }
}
```

11.3 Lectura y escritura combinadas

Las operaciones de lectura y escritura sobre ficheros se pueden combinar de tal forma que haya un flujo de lectura y otro de escritura, uno de lectura y dos de escritura, tres de lectura, etc.

En el ejemplo que presentamos a continuación hay dos flujos de lectura y uno de escritura. Observa que se declaran en total tres manejadores de fichero (dos para lectura y uno para escritura). El programa va leyendo, de forma alterna, una línea de cada fichero - una línea de `fichero1.txt` y otra línea de `fichero2.txt` - mientras queden líneas por leer en alguno de los ficheros; y al mismo tiempo va guardando esas líneas en otro fichero con nombre `mezcla.txt`.

```
/**
 * Ejemplo de uso de la clase File
 * Mezcla de dos ficheros
 *
 * @author Luis José Sánchez
 */

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

class EjemploFichero03 {

    public static void main(String[] args) {

        try {
            BufferedReader bf1 = new BufferedReader(new FileReader("fichero1.txt"));
            BufferedReader bf2 = new BufferedReader(new FileReader("fichero2.txt"));
            BufferedWriter bw = new BufferedWriter(new FileWriter("mezcla.txt"));

            String linea1 = "";
            String linea2 = "";

            while ( (linea1 != null) || (linea2 != null) ) {
                linea1 = bf1.readLine();
                linea2 = bf2.readLine();
                if (linea1 != null) bw.write(linea1 + "\n");
                if (linea2 != null) bw.write(linea2 + "\n");
            }

            bf1.close();
            bf2.close();
            bw.close();

        } catch (IOException ioe) {
            System.out.println("Se ha producido un error de lectura/escritura");
            System.err.println(ioe.getMessage());
        }
    }
}
```

11.4 Otras operaciones sobre ficheros

Además de leer desde o escribir en un fichero, hay otras operaciones relacionadas con los archivos que se pueden realizar desde un programa escrito en Java.

Veremos tan solo un par de ejemplos. Si quieres profundizar más, échale un vistazo a la [documentación oficial de la clase File](http://docs.oracle.com/javase/7/docs/api/java/io/File.html)².

El siguiente ejemplo muestra por pantalla un listado con todos los archivos que contiene un directorio.

```
/**
 * Ejemplo de uso de la clase File
 * Listado de los archivos del directorio actual
 *
 * @author Luis José Sánchez
 */

import java.io.File;

class EjemploFichero04 {

    public static void main(String[] args) {

        File fichero = new File("."); // se indica la ruta entre comillas
                                     // el punto (.) es el directorio actual

        String[] listaArchivos = fichero.list();
        for(int i = 0; i < listaArchivos.length; i++){
            System.out.println(listaArchivos[i]);
        }
    }
}
```

El siguiente programa de ejemplo comprueba si un determinado archivo existe o no mediante `exists()` y, en caso de que exista, lo elimina mediante `delete()`. Si intentáramos borrar un archivo que no existe obtendríamos un error.

²<http://docs.oracle.com/javase/7/docs/api/java/io/File.html>


```
/**
 * Ejemplo de uso de la clase File
 * Comprobación de existencia y borrado de un fichero
 *
 * @author Luis José Sánchez
 */

import java.io.File;

class EjemploFichero05 {

    public static void main(String[] args) {

        System.out.print("Introduzca el nombre del archivo que desea borrar: ");
        String nombreFichero = System.console().readLine();

        File fichero = new File(nombreFichero);

        if (fichero.exists()) {
            fichero.delete();
            System.out.println("El fichero se ha borrado correctamente.");
        } else {
            System.out.println("El fichero " + nombreFichero + " no existe.");
        }
    }
}
```

11.5 Paso de argumentos por línea de comandos

Como dijimos al comienzo de este capítulo, el paso de argumentos por línea de comandos no está directamente relacionado con los ficheros, aunque es muy frecuente combinar estos dos recursos.

Seguro que has usado muchas veces el paso de argumentos por línea de comandos sin saberlo. Por ejemplo, si tecleas el siguiente comando en una ventana de terminal:

```
$ head -5 /etc/bash.bashrc
```

se muestran por pantalla las 5 primeras líneas del fichero `bash.bashrc` que se encuentra en el directorio `/etc`. En este caso, `head` es el nombre del programa que estamos ejecutando y los valores `-5` y `/etc/bash.bashrc` son los argumentos.

Volvamos al comienzo, al primer capítulo. ¿Recuerdas cómo ejecutaste tu primer programa en Java? ¡Qué tiempos aquéllos! El programa `HolaMundo` se ejecutaba así:

```
$ java HolaMundo
```

Para probar los ejemplos que te presentamos más adelante, deberás hacer lo mismo y, además, añadir a continuación y separados por espacios los argumentos que quieres que lea el programa. Vamos a verlo en detalle.

El siguiente programa de ejemplo simplemente muestra por pantalla los argumentos introducidos. Compila el programa y prueba a ejecutar lo siguiente:

```
$ java EjemploArgumentos06 hola que tal 24 1.2 fin
```

con lo que obtendrás el siguiente resultado:

```
Los argumentos introducidos son:
hola
que
tal
24
1.2
fin
```

Los argumentos han pasado al programa en virtud del parámetro que incluimos en la línea del `main`:

```
public static void main(String[] args) {
```

Fíjate en lo que hay entre paréntesis: `String[] args`. Se trata de un array de cadenas de caracteres (`String`) donde cada uno de los elementos será un argumento que se ha pasado como parámetro. Si has ejecutado el ejemplo tal y como se ha indicado, `args[0]` vale “hola”, `args[1]` vale “que”, `args[2]` vale “tal”, `args[3]` vale “24”, `args[4]` vale “1.2” y `args[5]` vale “fin”.

```
/**
 * Paso de argumentos en la línea de comandos
 *
 * @author Luis José Sánchez
 */

class EjemploArgumentos06 {

    public static void main(String[] args) {

        System.out.println("Los argumentos introducidos son: ");
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

Hagamos algo un poco más útil con los argumentos. En el siguiente ejemplo, se suman todos los argumentos que se pasan como parámetros y se muestra por pantalla el resultado de la suma.

Después de compilar, deberás ejecutar desde una ventana de terminal lo siguiente (puedes cambiar los números si quieres):

```
$ java EjemploArgumentos07 10 36 44
```

con lo que obtendrás este resultado:

```
90
```

es decir, la suma de los números que has pasado como parámetros.

```
/**
 * Paso de argumentos en la línea de comandos
 *
 * @author Luis José Sánchez
 */

class EjemploArgumentos07 {

    public static void main(String[] args) {

        int suma = 0;

        for (int i = 0; i < args.length; i++) {
            suma += Integer.parseInt(args[i]);
        }

        System.out.println(suma);
    }
}
```

Observa que el programa convierte en números enteros todos y cada uno de los argumentos mediante el método `Integer.parseInt()` para poder sumarlos.



Los argumentos recogidos por línea de comandos se guardan siempre en un array de `String`. Cuando sea necesario realizar operaciones matemáticas con esos argumentos habría que convertirlos al tipo adecuado mediante `Integer.parseInt()` o `Double.parseDouble()`.

11.6 Combinación de ficheros y paso de argumentos

Llegó el momento de combinar las operaciones con ficheros con el paso de parámetros por línea de comandos. El ejemplo que mostramos a continuación es parecido al que hemos visto en el apartado anterior; calcula la media de una serie de números, pero esta vez esos números se leen desde un fichero y lo que se pasa como parámetro por la línea de comandos es precisamente el nombre del fichero donde están esos números.

Crea un fichero y nómbralo `numeros.txt` e introduce los siguientes números (es importante que estén separados por un salto de línea para que el programa funcione bien):

```
25
100
44
17
6
8
```

A continuación, compila el programa de ejemplo `EjemploFichero09.java` y teclea lo siguiente en la ventana de terminal:

```
$ java EjemploFichero09 numeros.txt
```

Aparecerá el siguiente resultado:

```
La media es 33.333333333333336
```

```
/**
 * Ejemplo de uso de la clase File
 * Calcula la media de los números que se encuentran en un fichero de texto.
 * El nombre del fichero se pasa como argumento en la línea de comandos.
 *
 * @author Luis José Sánchez
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class EjemploFichero09 {

    public static void main(String[] args) {
```

```

    if (args.length != 1) {
        System.out.println("Este programa calcula la media de los números contenidos en un f\
ichero.");
        System.out.println("Uso del programa: java EjemploFichero09 FICHERO");
        System.exit(-1); // sale del programa
    }

    try {
        BufferedReader bf = new BufferedReader(new FileReader(args[0]));

        String linea = "";
        int i = 0;
        double suma = 0;

        while (linea != null) {
            i++;
            suma += Double.parseDouble(linea);
            linea = bf.readLine();
        }
        i--;

        bf.close();

        System.out.println("La media es " + suma/(double)i);

    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}

```

Observa que el programa comprueba el número de argumentos que se pasan por la línea de comandos mediante `if (args.length != 1)` y si este número es distinto de 1, muestra este mensaje:

```

Este programa calcula la media de los números contenidos en un fichero.
Uso del programa: java EjemploFichero09 FICHERO

```

y sale del programa. De esta manera el usuario sabe exactamente cómo hay que utilizar el programa y cómo hay que pasarle la información.

11.7 Procesamiento de archivos de texto

La posibilidad de realizar desde Java operaciones con ficheros abre muchas posibilidades a la hora de procesar archivos: cambiar una palabra por otra, eliminar ciertos caracteres, mover de sitio una

línea o una palabra, borrar espacios o tabulaciones al final de las líneas o cualquier otra cosa que se nos pueda ocurrir.

Cuando se procesa un archivo de texto, los pasos a seguir son los siguientes:

1. Leer una línea del fichero origen mientras quedan líneas por leer.
2. Modificar la línea (normalmente utilizando los métodos que ofrece la clase `String`).
3. Grabar la línea modificada en el fichero destino.
4. Volver al paso 1.

A continuación tienes algunos métodos de la clase `String` que pueden resultar muy útiles para procesar archivos de texto:

- **`charAt(int n)`** Devuelve el carácter que está en la posición `n`-ésima de la cadena. Recuerda que la primera posición es la número 0.
- **`indexOf(String palabra)`** Devuelve un número que indica la posición en la que comienza una palabra determinada.
- **`length()`** Devuelve la longitud de la cadena.
- **`replace(char c1, char c2)`** Devuelve una cadena en la que se han cambiado todas las ocurrencias del carácter `c1` por el carácter `c2`.
- **`substring(int inicio, int fin)`** Devuelve una subcadena.
- **`toLowerCase()`** Convierte todas las letras en minúsculas.
- **`toUpperCase()`** Convierte todas las letras en mayúsculas.

Puedes consultar todos los métodos de la clase `String` en la [documentación oficial](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html)³.

A continuación tienes un ejemplo en el que se usan los métodos descritos anteriormente.

```
/**
 * Ejemplos de uso de String
 *
 * @author Luis José Sánchez
 */
public class EjemplosString {
    public static void main(String[] args) {

        System.out.println("\nEjemplo 1");
        System.out.println("En la posición 2 de \"berengena\" está la letra "
            + "berengena".charAt(2));

        System.out.println("\nEjemplo 2");
```

³<http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

```
String frase = "Hola caracola.";
char[] trozo = new char[10];
trozo[0] = 'z'; trozo[1] = 'z'; trozo[2] = 'z';
frase.getChars(2, 7, trozo, 1);
System.out.print("El array de caracteres vale ");
System.out.println(trozo);

System.out.println("\nEjemplo 3");
System.out.println("La secuencia \"co\" aparece en la frase en la posición "
    + frase.indexOf("co"));

System.out.println("\nEjemplo 4");
System.out.println("La palabra \"murciélago\" tiene "
    + "murciélago".length() + " letras");

System.out.println("\nEjemplo 5");
String frase2 = frase.replace('o', 'u');
System.out.println(frase2);

System.out.println("\nEjemplo 6");
frase2 = frase.substring(3, 10);
System.out.println(frase2);

System.out.println("\nEjemplo 7");
System.out.println(frase.toLowerCase());

System.out.println("\nEjemplo 8");
System.out.println(frase.toUpperCase());
}
}
```

A continuación se muestra un programa que procesa archivos de texto. Lo que hace es cambiar cada tabulador por dos espacios en blanco. Lo hice a propósito cuando estaba escribiendo este manual ya que el [estándar de Google para la codificación en Java](http://google-styleguide.googlecode.com/svn/trunk/javaguide.html)⁴ especifica que la sangría debe ser precisamente de dos espacios. Tenía muchos programas escritos en Java (ejemplos y soluciones a ejercicios) que no cumplían este requisito porque la sangría estaba aplicada con el carácter de tabulación; tenía dos posibilidades, abrir todos y cada uno de los ficheros y cambiarlo a mano, o escribir un programa que lo hiciera. Opté por la segunda opción y aquí tienes el programa ¿verdad que es muy sencillo?

⁴<http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/**
 * Cambia los tabuladores por 2 espacios
 * @author Luis José Sánchez
 */
public class EjemploProcesamiento10 {
    public static void main(String[] args) {

        for (int i = 0; i < args.length; i++) {

            System.out.print("Procesando el archivo " + args[i] + "...");

            try {
                // renombra el fichero añadiendo ".tmp"
                File ficheroOriginal = new File(args[i]);
                File ficheroTemporal = new File(args[i] + ".tmp");
                ficheroOriginal.renameTo(ficheroTemporal);

                // lee los datos del archivo temporal
                BufferedReader bf = new BufferedReader(new FileReader(args[i] + ".tmp"));

                // crea un fichero nuevo con el nombre original
                BufferedWriter bw = new BufferedWriter(new FileWriter(args[i]));

                String linea = "";

                while (linea != null) {
                    linea = bf.readLine();

                    if (linea != null) {
                        // cambia el tabulador por 2 espacios
                        linea = linea.replace("\t", "  ");

                        bw.write(linea + "\n");
                    }
                }

                bf.close();
                bw.close();
            }
        }
    }
}
```



```
// borra el fichero temporal
ficheroTemporal.delete();

} catch (IOException ioe) {
    System.out.println("Se ha producido un error de lectura/escritura");
    System.err.println(ioe.getMessage());
}

System.out.println("hecho");
}
}
```

11.8 Ejercicios



Ejercicio 1

Escribe un programa que guarde en un fichero con nombre `primos.dat` los números primos que hay entre 1 y 500.



Ejercicio 2

Realiza un programa que lea el fichero creado en el ejercicio anterior y que muestre los números por pantalla.



Ejercicio 3

Escribe un programa que guarde en un fichero el contenido de otros dos ficheros, de tal forma que en el fichero resultante aparezcan las líneas de los primeros dos ficheros mezcladas, es decir, la primera línea será del primer fichero, la segunda será del segundo fichero, la tercera será la siguiente del primer fichero, etc.

Los nombres de los dos ficheros origen y el nombre del fichero destino se deben pasar como argumentos en la línea de comandos.

Hay que tener en cuenta que los ficheros de donde se van cogiendo las líneas pueden tener tamaños diferentes.



Ejercicio 4

Realiza un programa que sea capaz de ordenar alfabéticamente las palabras contenidas en un fichero de texto. El nombre del fichero que contiene las palabras se debe pasar como argumento en la línea de comandos. El nombre del fichero resultado debe ser el mismo que el original añadiendo la coletilla `sort`, por ejemplo `palabras_sort.txt`. Suponemos que cada palabra ocupa una línea.



Ejercicio 5

Escribe un programa capaz de quitar los comentarios de un programa de Java. Se utilizaría de la siguiente manera:

```
quita_comentarios PROGRAMA_ORIGINAL PROGRAMA_LIMPIO
```

Por ejemplo:

```
quita_comentarios hola.java holav2.java
```

crea un fichero con nombre `holav2.java` que contiene el código de `hola.java` pero sin los comentarios.



Ejercicio 6

Realiza un programa que diga cuántas ocurrencias de una palabra hay en un fichero. Tanto el nombre del fichero como la palabra se deben pasar como argumentos en la línea de comandos.

12. Aplicaciones web en Java (JSP)

Hasta el momento, los programas realizados funcionan en una austera ventana de terminal. No se trata de algo casual, hemos elegido esta manera de trabajar de forma deliberada, para que el estudiante se centre en resolver el problema y no se distraiga dándole sombra a un botón o escogiendo la mejor imagen de fondo para un listado.

Llegamos, no obstante, a un punto en que algunos programas piden a gritos una apariencia más vistosa.

En este capítulo vamos a aprender cómo utilizar páginas web como interfaz para programas en Java. Usaremos JSP (JavaServer Pages) que nos permitirá mezclar código en Java con código HTML. El código en Java que utilizaremos será muy parecido al que hemos venido utilizando hasta ahora. Cambiará únicamente lo relativo a mostrar información por pantalla (ahora se volcará todo a HTML) y la manera en que se introducen los datos, que se realizará mediante formularios.



El lector deberá tener unos conocimientos básicos de HTML para entender bien y sacar provecho de este capítulo. Una web excelente para aprender HTML es W3Schools¹

El [estándar de programación de Google para el código fuente escrito en Java](#)² no especifica ninguna regla en cuanto a la manera de nombrar los ficheros correspondientes a las aplicaciones JSP. Por tanto vamos a adoptar [las convenciones de Oracle](#)³ en esta materia. Según estas convenciones, los nombres de los ficheros JSP deben estar escritos en *lowerCamelCase*, es decir, la primera letra siempre es minúscula y, en caso de utilizar varias palabras dentro del nombre, éstas van seguidas una de otra empezando cada una por mayúscula. Finalmente se añade la extensión `.jsp`. Por ejemplo, `listadoSocios.jsp` es un nombre correcto, mientras que `ListadoSocios.jsp`, `listado_socios.jsp` o `Listadosocios.jsp` son incorrectos.

12.1 Hola Mundo en JSP

Para desarrollar aplicaciones en JSP utilizaremos el entorno de desarrollo **Netbeans**. Este IDE se puede descargar de forma gratuita en <https://netbeans.org/downloads/>

Sigue los siguientes pasos para crear una aplicación en JSP con **Netbeans**:

- En el menú principal, selecciona **Archivo** y a continuación **Nuevo Proyecto**. Se puede hacer más rápido con la combinación **Control + Mayúscula + N**.

¹<http://w3schools.com>

²<http://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

³<http://www.oracle.com/technetwork/articles/javase/code-convention-138726.html>

- Aparecerá una ventana emergente para elegir el tipo de proyecto. Selecciona **Java Web** en el apartado “Categorías” y **Aplicación Web** en el apartado “Proyectos”. Haz clic en **Siguiente**.
- Ahora hay que darle el nombre al proyecto, lo puedes llamar por ejemplo **Saludo1**. Recuerda no utilizar caracteres especiales ni signos de puntuación. Se creará una carpeta con el mismo nombre que el proyecto que contendrá todos los ficheros necesarios. Haz clic en **Siguiente**.
- En la siguiente ventana tendrás que elegir el servidor, la versión de Java y la ruta. Elige **Glass Fish Server**. En caso de no haber ningún servidor disponible, **Netbeans** permite añadir uno sobre la marcha (se lo descarga y lo instala). Haz clic en **Finalizar**.

A la izquierda, en la ventana de proyectos, debe aparecer el que acabas de crear: **Saludo1**. Navegando por la estructura del proyecto, verás que hay una carpeta con nombre **Web Pages**; en esta carpeta se deberán guardar los archivos correspondientes a la aplicación: archivos jsp, html, css, imágenes, archivos de audio, etc. Para que los archivos estén ordenados, se pueden organizar, a su vez, en subcarpetas dentro de **Web Pages**.

Por defecto, cuando se crea un proyecto JSP, se crea el archivo `index.html`. Elimina este archivo de tu proyecto y luego crea `index.jsp` haciendo clic con el botón derecho en **Web Pages** → **Nuevo** → **JSP**. Al indicar el nombre no hay que poner la extensión.

Ya tienes un “Hola Mundo” en inglés. Si lo quieres en español solo tienes que cambiar “Hello World” por “Hola Mundo”, con lo que la aplicación quedaría como sigue:

```
<%-- saludo1.jsp --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1> ¡Hola Mundo! </h1>
    En esta página se usa únicamente HTML.
  </body>
</html>
```

Para ejecutar la aplicación hay que hacer clic en el botón verde **Ejecutar Proyecto** o pulsar F6. En ese momento, **Netbeans** buscará un fichero con el nombre `index.jsp` y lo lanzará en el navegador.

Observa que el programa del ejemplo se llama `saludo1.jsp` y no `index.jsp`. En este caso, para ejecutar la aplicación hay que hacer clic con el botón derecho encima de `saludo1.jsp` y seleccionar **Ejecutar**.

De momento, solo hemos visto código HTML ¿dónde está Java? Bien, vamos a verlo, pero antes hay entender bien cómo funciona JSP.

Una aplicación JSP consistirá en una o varias páginas web que normalmente contendrá código HTML y que llevarán “insertado” código en Java. Este código en Java puede colocarse en cualquier parte del archivo y se delimita mediante las etiquetas `<% y %>`. Cuando se pulsa **F6** para ejecutar la aplicación sucede lo siguiente:

1. Todo el código (tanto HTML como JAVA) se envía al servidor (Glass Fish o Apache Tomcat por ejemplo).
2. El servidor deja intacto el código HTML y compila y ejecuta el código en Java. Generalmente el código Java genera código HTML mediante las instrucciones `out.print` y `out.println`.
3. Por último, todo el código (ya solo queda HTML) se envía al navegador que es el que muestra la página. Recuerda que el navegador no entiende Java (entiende Javascript que es otro lenguaje diferente) y es imprescindible que todo el código Java haya sido traducido previamente.

12.2 Mezclando Java con HTML

A continuación se muestra un ejemplo que sí tiene Java. Como ves, el código Java está delimitado por los caracteres `<% y %>`.

```
<%-- saludo2.jsp --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <% out.println("<h1> ¡Hola Mundo! </h1>"); %>
  </body>
</html>
```

Mediante Java, se puede generar cualquier contenido HTML, por ejemplo para modificar estilos, crear tablas, mostrar imágenes, etc. Trabajando en consola, lo que se incluye en un `print` es exactamente lo que se muestra por pantalla. Ahora, con JSP, lo que se incluye dentro del `print` es “renderizado” a posteriori por el navegador. En el siguiente ejemplo se puede ver claramente este comportamiento.

```
<%-- saludo3.jsp --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1> ¡Hola Caracola! </h1>
    <% out.print("<b><i>"); %>
    Esta línea se ha puesto en negrita y cursiva mediante Java.
    <% out.print("</i></b>"); %>
  </body>
</html>
```

Es muy útil ver el código fuente que se ha generado una vez que se puede visualizar la aplicación en el navegador. Para ver el código fuente tanto en **Firefox** como en **Chrome**, hay que hacer clic con el botón derecho en la página y seleccionar la opción “Ver código fuente de la página”.

Cuando el contenido que se quiere volcar en HTML es el resultado de una expresión, se pueden utilizar de forma opcional las etiquetas `<%=` y `>%`.

```
<%-- saludo4.jsp --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1> ¡Hola Caracola! </h1>
    <%= "<b><i>" %>
    Esta línea se ha puesto en negrita y cursiva mediante Java.
    <%= "</i></b>" %>
  </body>
</html>
```

En el código Java que se inserta en las aplicaciones JSP se pueden utilizar todos los elementos del lenguaje vistos anteriormente: bucles, sentencias de control, arrays, diccionarios, etc.

En el siguiente ejemplo se utiliza un bucle `for` para mostrar texto en diferentes tamaños, utilizando las etiquetas desde la `<h6>` (cabecera pequeña) hasta la `<h1>` (cabecera más grande).

```

<%-- pruebaVariable1 --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <%
      for (int i = 1; i < 7; i++)
        out.println("<h" + (7-i) + ">" + i + "</h" + (7-i) + ">");
    %>
  </body>
</html>

```

El código en Java se puede insertar en cualquier parte dentro del código HTML, e incluso, se pueden insertar varios trozos como se puede ver en el siguiente ejemplo. Aunque una variable se haya definido en una determinada zona, su ámbito de actuación no se ve reducido a ese fragmento de código sino que se puede utilizar posteriormente en otro lugar de la aplicación.

Observa que en el siguiente ejemplo, la definición e inicialización de la variable `x` se realiza en un bloque de código en Java y que luego esta variable se utiliza en otros dos bloques diferentes.

```

<%-- pruebaVariable2 --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <% int x = 3; %>
    <h<% out.print(x); %>>hola</h<% out.print(x); %>>
  </body>
</html>

```

En ocasiones puede ser útil recabar información del sistema: versión de Java en uso, sistema operativo, nombre de usuario, etc. Estos datos se pueden obtener mediante `System.getProperty(propiedad)`.

```

<!-- muestraInfo.jsp -->
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>Información del entorno de trabajo</h1>
    <%
      out.print("Fabricante de Java: " + System.getProperty("java.vendor"));
      out.print("<br>Url del fabricante: " + System.getProperty("java.vendor.url"));
      out.print("<br>Versión: " + System.getProperty("java.version"));
      out.print("<br>Sistema operativo: " + System.getProperty("os.name"));
      out.print("<br>Usuario: " + System.getProperty("user.name"));
    %>
  </body>
</html>

```

Para mostrar información en una página de forma ordenada es muy frecuente utilizar tablas. Las tablas son especialmente útiles para mostrar listados obtenidos de una base de datos como veremos en el siguiente capítulo.

En el siguiente ejemplo se muestra una tabla con dos columnas; en la primera columna se muestran los números del 0 al 9 y en la segunda, sus correspondientes cuadrados.

```

<!-- tabla.jsp -->
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <body>
    <h1>Ejemplo de tabla</h1>
    <table border="2">
      <tr>
        <td>Número</td><td>Cuadrado</td>
      </tr>
      <%
        for(int i = 0; i < 10; i++) {
          out.println("<tr>");
          out.println("<td>" + i + "</td>");
          out.println("<td>");
          out.println(i * i);
          out.println("</td></tr>");
        }
      %>
    </table>
  </body>
</html>

```



```

    }
    %>
</table>
</body>
</html>

```

12.3 Recogida de datos en JSP

En cualquier aplicación web, la introducción de datos se realiza mediante formularios. Aunque se puede hacer todo en la misma página, de momento vamos a tener dos. La primera página contendrá un formulario y será la encargada de recoger la información y enviarla a una segunda página. Esta última página recibirá los datos, realizará una serie de cálculos u operaciones si procede y, por último, mostrará un resultado.

En el primer ejemplo - un proyecto que llamaremos PasoDeCadena - tenemos una página con nombre `index.jsp` que contiene un formulario HTML. Este formulario contiene una entrada de texto donde el usuario introducirá una cadena de caracteres. Es muy importante darle un valor a la etiqueta `name`. En el caso que nos ocupa tenemos `name="cadenaIntro"`, por tanto el dato que recoge el formulario se llama de esa manera, sería el equivalente al nombre de la variable en Java. No menos importante es indicar la página que recibirá los datos que recoge el formulario. Esta información se indica con la etiqueta `action`, en nuestro ejemplo `action="frase.jsp"` indica que será el fichero `frase.jsp` el que recibirá `cadenaIntro`.

El fichero `index.jsp` únicamente contiene código HTML, por tanto se podría llamar `index.html` y la aplicación funcionaría exactamente igual.

```

<%-- index.jsp (proyecto PasoDeCadena) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Paso de cadena</title>
  </head>
  <body>
    <h1>Pasando una cadena de caracteres</h1>
    <form method="post" action="frase.jsp">
      Introduzca el nombre de una fruta:
      <input type="text" name="cadenaIntro">
      <input type="submit" value="OK">
    </form>
  </body>
</html>

```

Para recoger los datos enviados por un formulario se utiliza `request.getParameter("nombreDeVariable")` donde `nombreDeVariable` es el dato que se envía desde el formulario. En caso de que el dato enviado sea un texto que pueda contener tildes o eñes, hay que especificar la codificación mediante `request.setCharacterEncoding("UTF-8")` antes de recoger el dato.

```
<%-- frase.jsp (proyecto PasoDeCadena) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Paso de cadena</title>
  </head>
  <body>
    <% request.setCharacterEncoding("UTF-8"); %>
    Me gusta mucho comer
    <% out.print(request.getParameter("cadenaIntro")); %>
  </body>
</html>
```

En el siguiente ejemplo, llamado `Incrementa5`, se muestra cómo manipular en JSP un dato numérico. La aplicación recoge un número y luego muestra ese número incrementado en 5 unidades.

```
<%-- index.jsp (proyecto Incrementa5) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <form method="get" action="incrementa5.jsp">
      Introduzca un número (puede tener decimales):
      <input type="text" name="numeroIntro">
      <input type="submit" value="OK">
    </form>
  </body>
</html>
```

Observa en el fichero `incrementa5.jsp` cómo se transforma la cadena de caracteres que se recibe en `numeroIntro` en un dato numérico con el método `Double.parseDouble`; exactamente igual que si estuviéramos leyendo un número desde teclado en la ventana de terminal.

```
<!-- incrementa5.jsp (proyecto Incrementa5) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    El número introducido más cinco es
    <%
      double resultado;
      resultado = Double.parseDouble(request.getParameter("numeroIntro")) + 5;
      out.print(resultado);
    %>
  </body>
</html>
```

El siguiente ejemplo ilustra la recogida y envío de dos variables, x e y. Observa que ahora el formulario contiene dos entradas de texto, una para cada variable.

```
<!-- index.jsp (proyecto Suma) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Suma</title>
  </head>
  <body>
    <h1>Supercalculadora</h1>
    <form method="get" action="resultado.jsp">
      x <input type="text" name="x"/><br>
      y <input type="text" name="y"/><br>
      <input type="submit">
    </form>
  </body>
</html>
```

En el fichero resultado.jsp se reciben las variables recogidas en index.jsp y se suman. Recuerda que por defecto la información enviada por un formulario es una cadena de caracteres. Si queremos sumar los valores introducidos, debemos transformar las cadenas de caracteres a números enteros con el método `Integer.valueOf()`.

```

<%-- resultado.jsp (proyecto Suma) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Suma</title>
  </head>
  <body>
    La suma es
    <%
      int primerNumero = Integer.valueOf(request.getParameter("x"));
      int segundoNumero = Integer.valueOf(request.getParameter("y"));
      out.println(primerNumero + segundoNumero);
    %>
  </body>
</html>

```

El siguiente proyecto de ejemplo es Animales. En la página principal (index.jsp) se puede seleccionar un animal - gato o caracol - y un número. Una vez introducidos los datos, éstos se envían a animales.jsp.

```

<%-- index.jsp (proyecto Animales) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Animales</title>
  </head>
  <body>
    <form method="post" action="animales.jsp">
      Seleccione animal a visualizar
      <select name="animal">
        <option>Gato</option>
        <option>Caracol</option>
      </select>
      <br>
      Número de animales <input type="text" name="numero" size="3">
      <br>
      <input type="submit">
    </form>
  </body>
</html>

```

La página `animales.jsp` nos mostrará una imagen del animal elegido repetida el número de veces que hemos indicado.

```
<!-- animales.jsp (proyecto Animales) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Animales</title>
  </head>
  <body>
    <%
      String nombreAnimal = request.getParameter("animal");
      String nombreImagen;
      if (nombreAnimal.equals("Gato")) {
        nombreImagen = "gato.jpg";
      } else {
        nombreImagen = "caracol.jpg";
      }

      int veces = Integer.parseInt(request.getParameter("numero"));

      for (int i = 0; i < veces; i++) {
        out.print("<img src=\"" + nombreImagen + "\" width=\"20%\">");
      }
    %>
  </body>
</html>
```

12.4 POO en JSP

En las aplicaciones realizadas en JSP se pueden incluir clases definidas por el usuario para posteriormente crear objetos de esas clases. Lo habitual es que el fichero que contiene la definición de la clase se encuentre separado del programa principal.

En un nuevo proyecto con nombre **GatosConClase** vamos a crear la clase `Gato`; para ello haz clic derecho sobre el icono del proyecto, selecciona **Nuevo** y luego selecciona **Clase de Java**. El nombre de la clase será `Gato` y el nombre del paquete será por ejemplo `daw1`.

El fichero `Gato.java` quedaría como el que se muestra a continuación.

```
/* Gato.java (proyecto GatosConClase) */

package daw1;

public class Gato {
    private String nombre;
    private String imagen;

    public Gato(String nombre, String imagen) {
        this.nombre = nombre;
        this.imagen = imagen;
    }

    public String getNombre() {
        return nombre;
    }

    public String getImagen() {
        return imagen;
    }

    @Override
    public String toString() {
        return "<img src='" + imagen + "' width='80'>Hola, soy " + nombre + "<br>";
    }

    public String maulla() {
        return "<img src='" + imagen + "' width='80'>Miauuuuuuu<br>";
    }

    public String come(String comida) {
        return "<img src='" + imagen + "' width='80'>Estoy comiendo " + comida + "<br>";
    }
}
```

Hemos creado una clase Gato muy sencilla que únicamente contiene dos atributos: el nombre del gato y el nombre del fichero de su foto. Como métodos, se han implementado toString(), maulla() y come().

```

<!-- index.jsp (proyecto GatosConClase) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="daw1.Gato"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Gatos con clase</title>
  </head>
  <body>
    <h1>Gatos con clase</h1>
    <hr>
    <%
      Gato g1 = new Gato("Pepe", "gato.jpg");
      Gato g2 = new Gato("Garfield", "garfield.jpg");
      Gato g3 = new Gato("Tom", "tom.png");

      out.println(g1);
      out.println(g2);
      out.println(g3);
      out.println(g1.maula());
      out.println(g2.come("sardinas"));
    %>
  </body>
</html>

```

El código Java del programa principal es casi idéntico al que tendríamos en un programa para consola.

El siguiente ejemplo - GatosConClaseYBocadillos - es una mejora del anterior. Se añaden [estilos](#)⁴ para mostrar lo que dicen los gatos, igual que en un cómic.

```

/* Gato.java (proyecto GatosConClaseYBocadillos) */

package daw1;

public class Gato {
  private String nombre;
  private String imagen;

  public Gato(String nombre, String imagen) {
    this.nombre = nombre;
    this.imagen = imagen;
  }

```

⁴https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/12_JSP/GatosConClaseYBocadillos/estilos.css

```

    }

    public String getNombre() {
        return nombre;
    }

    public String getImagen() {
        return imagen;
    }

    @Override
    public String toString() {
        return "<div class=\"acciongato\"><img src=\"\" + imagen + "\"" width=\"80\"><div class=\"arrow_box\">&nbsp;Hola, soy " + nombre + "&nbsp;</div></div>";
    }

    public String maulla() {
        return "<div class=\"acciongato\"><img src=\"\" + imagen + "\"" width=\"80\"><div class=\"arrow_box\">&nbsp;Miauuuuuuuu&nbsp;</div></div>";
    }

    public String come(String comida) {
        return "<div class=\"acciongato\"><img src=\"\" + imagen + "\"" width=\"80\"><div class=\"arrow_box\">&nbsp;Estoy comiendo " + comida + "&nbsp;</div></div>";
    }
}

```

El único añadido al programa principal es una línea situada en la cabecera de la página que carga la hoja de estilos estilos.css.

```

<%-- index.jsp (proyecto GatosConClaseYBocadillos) --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="daw1.Gato"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Gatos con clase y bocadillos</title>
        <link rel="stylesheet" type="text/css" href="estilos.css" />
    </head>
    <body>
        <h1>Gatos con clase</h1>
        <hr>
        <%
            Gato g1 = new Gato("Pepe", "gato.jpg");

```



```
Gato g2 = new Gato("Garfield", "garfield.jpg");
Gato g3 = new Gato("Tom", "tom.png");

out.println(g1);
out.println(g2);
out.println(g3);
out.println(g1.maula());
out.println(g2.come("sardinas"));
%>
</body>
</html>
```

12.5 Ejercicios



Ejercicio 1

Crea una aplicación web en Java que muestre tu nombre y tus datos personales por pantalla. La página principal debe ser un archivo con la extensión `jsp`. Comprueba que se lanzan bien el servidor y el navegador web. Observa los mensajes que da el servidor. Fíjate en la dirección que aparece en la barra de direcciones del navegador.



Ejercicio 2

Mejora el programa anterior de tal forma que la apariencia de la página web que muestra el navegador luzca más bonita mediante la utilización de CSS (utiliza siempre ficheros independientes para CSS para no mezclarlo con HTML).



Ejercicio 3

Escribe una aplicación que pida tu nombre. A continuación mostrará “Hola” seguido del nombre introducido. El nombre se deberá recoger mediante un formulario.



Ejercicio 4

Realiza una aplicación que calcule la media de tres notas.



Ejercicio 5

Realiza un conversor de euros a pesetas.



Ejercicio 6

Realiza un conversor de pesetas a euros.



Ejercicio 7

Combina las dos aplicaciones anteriores en una sola de tal forma que en la página principal se pueda elegir pasar de euros a pesetas o de pesetas a euros. Adorna la página con alguna foto o dibujo.



Ejercicio 8

Realiza una aplicación que pida un número y que luego muestre la tabla de multiplicar de ese número. El resultado se debe mostrar en una tabla (`<table>` en HTML).



Ejercicio 9

Realiza una aplicación que pinte una pirámide. La altura se pedirá en un formulario. La pirámide estará hecha de bolitas, ladrillos o cualquier otra imagen.



Ejercicio 10

Realiza un cuestionario con 10 preguntas tipo test sobre las asignaturas que se imparten en el curso. Cada pregunta acertada sumará un punto. El programa mostrará al final la calificación obtenida. Pásale el cuestionario a tus compañeros y pídeles que lo hagan para ver qué tal andan de conocimientos en las diferentes asignaturas del curso. Utiliza *radio buttons* en las preguntas del cuestionario.



Ejercicio 11

Escribe una aplicación que genere el calendario de un mes. Se pedirá el nombre del mes, el año, el texto que queremos que aparezca sobre el calendario, el día de la semana en que cae el día 1 y el número de días que tiene el mes.



Ejercicio 12

Mejora la aplicación anterior de tal forma que no haga falta introducir el día de la semana en que cae el día 1 y el número de días que tiene el mes. El programa debe deducir estos datos del mes y el año. Pista: puedes usar la clase `Calendar` (`java.util.Calendar`).



Ejercicio 13

Transforma el test de infidelidad realizado anteriormente para consola en una aplicación web.



Ejercicio 14

Escribe un programa que muestre los n primeros términos de la serie de Fibonacci. El primer término de la serie de Fibonacci es 0, el segundo es 1 y el resto se calcula sumando los dos anteriores, por lo que tendríamos que los términos son 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144... El número n se debe introducir por teclado.



Ejercicio 15

Realiza una aplicación que genere 100 números aleatorios del 1 al 200. Los primos deberán aparecer en un color diferente al resto.



Ejercicio 16

Realiza una aplicación que muestre la tirada aleatoria de tres dados de póker. Utiliza imágenes de dados reales.



Ejercicio 17

Realiza un configurador del interior de un vehículo. El usuario puede elegir, mediante un formulario, el color de la tapicería – blanco, negro o berengena - y el material de las molduras – madera o carbono. Se debe mostrar el interior del coche tal y como lo quiere el usuario.



Ejercicio 18

Crea la aplicación “El Trile”. Deben aparecer tres cubiletes por pantalla y el usuario deberá seleccionar uno de ellos. Para dicha selección se puede usar un formulario con radio-button, una lista desplegable, hacer clic en el cubilete o lo que resulte más fácil. Se levantarán los tres cubiletes y se verá si estaba o no la bolita dentro del que el usuario indicó, así mismo, se mostrará un mensaje diciendo “Lo siento, no has acertado” o “¡Enhorabuena!, has encontrado la bolita”. La probabilidad de encontrar la bolita es de 1/3.



Ejercicio 19

Crea el juego “Apuesta y gana”. El usuario debe introducir inicialmente una cantidad de dinero. A continuación aparecerá por pantalla una imagen de forma aleatoria. Si sale una calavera, el usuario pierde todo su dinero y termina el juego. Si sale medio limón, el usuario pierde la mitad del dinero y puede seguir jugando con esa cantidad o puede dejar de jugar. Si sale el gato chino de la suerte, el usuario multiplica por dos su dinero y puede seguir jugando con esa cantidad o puede dejar de jugar.



Ejercicio 20

Crea una aplicación que dibuje un tablero de ajedrez mediante una tabla HTML generada con bucles usando JSP y que sitúe dentro del tablero un alfil y un caballo en posiciones aleatorias. Las dos figuras no pueden estar colocadas en la misma casilla. Las filas y las columnas del tablero deben estar etiquetadas correctamente.

13. Acceso a bases de datos

En el [capítulo 11](#) vimos cómo almacenar información en ficheros de texto de tal forma que esta información no se pierde cuando el programa se cierra. Normalmente, los ficheros de texto se usan en casos concretos como archivos de configuración o archivos de registro (*log*).

Las bases de datos relacionales entran en juego cuando se necesita almacenar mucha información y, además, esta información es homogénea y se puede representar mediante tablas. Por ejemplo, una aplicación que permita gestionar la matriculación de los alumnos de un centro educativo deberá ser capaz de almacenar los datos de gran cantidad de alumnos (puede que miles). En este caso concreto, cada alumno se representa mediante un **registro** que contiene determinados datos como el número de expediente, el nombre, los apellidos, la nota media, etc. A estos datos se les llama **campos**.

En este capítulo vamos a ver cómo se puede acceder mediante Java a una base de datos relacional **MySQL** para hacer listados, insertar nuevos elementos y borrar elementos existentes.

Los programas de ejemplo que se muestran en este capítulo son aplicaciones JSP, aprovechando lo visto en el [capítulo anterior](#), aunque también se puede acceder a una base de datos desde un programa en Java escrito para la consola.

13.1 Socios de un club de baloncesto

Para ilustrar el acceso a una base de datos mediante Java vamos a utilizar como ejemplo un club de baloncesto. Nuestro club necesita tener almacenada la información de todos los socios. Sobre cada socio se necesita saber su nombre completo, su estatura, su edad y su localidad de residencia. Cada socio tendrá, además, un número de identificación único que será el número de socio.

Nuestra aplicación consistirá en varias páginas JSP que van a permitir ver un listado con los datos de todos los socios, dar de alta un nuevo miembro en el club, y también borrar o modificar los datos de un determinado socio.

Si todavía no tienes instalado en tu equipo el gestor de bases de datos **MySQL**, deberás instalarlo. También es recomendable instalar **PHPMyAdmin** que es una aplicación que permite crear y manejar bases de datos MySQL de una forma muy sencilla, mediante una *interfaz web*.



Instalación de MySQL

```
sudo apt-get install mysql-server  
sudo apt-get install mysql-client
```

Para el usuario `root` de MySQL, deberás establecer la contraseña `root` para que te funcionen correctamente los ejemplos.



Instalación de PHPMysqlAdmin

```
sudo apt-get install phpmyadmin
```

Para acceder a PHPMysqlAdmin debes escribir `http://localhost/phpmyadmin/` en la barra de direcciones del navegador.

Deberás crear la base de datos MySQL a la que llamarás `baloncesto`, que contendrá una tabla de nombre `socio` con los campos `socioID`, `nombre`, `estatura`, `edad` y `localidad` y luego deberás añadir los datos de los socios. Todo esto lo tienes ya preparado en un fichero con nombre `baloncesto.sql` que está en GitHub, en la dirección https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/13_JSP_y_BBDD/baloncesto.sql. Descárgate este fichero, lo utilizaremos en seguida.

A continuación mostramos un fragmento del fichero `baloncesto.sql` para que veas cómo se crea la base de datos, la tabla `socio` y cómo se introducen los datos de muestra.

```
--
-- Base de datos: `baloncesto`
--
CREATE DATABASE IF NOT EXISTS `baloncesto` DEFAULT CHARACTER SET utf8 COLLATE utf8_bin;
USE `baloncesto`;

-- -----

--
-- Estructura de tabla para la tabla `socio`
--

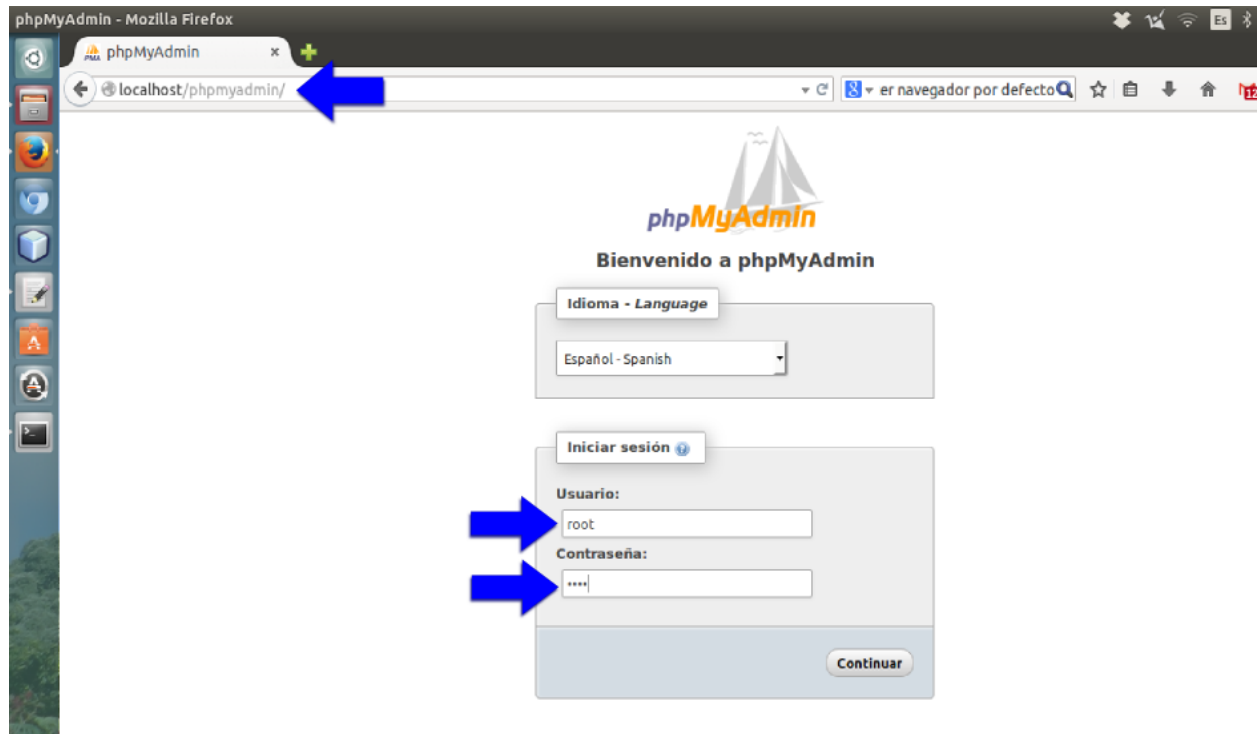
CREATE TABLE IF NOT EXISTS `socio` (
  `socioID` int(11) NOT NULL,
  `nombre` varchar(40) COLLATE utf8_spanish2_ci DEFAULT NULL,
  `estatura` int(11) DEFAULT NULL,
  `edad` int(11) DEFAULT NULL,
  `localidad` varchar(30) COLLATE utf8_spanish2_ci DEFAULT NULL,
  PRIMARY KEY (`socioID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_spanish2_ci;

--
-- Volcado de datos para la tabla `socio`
--

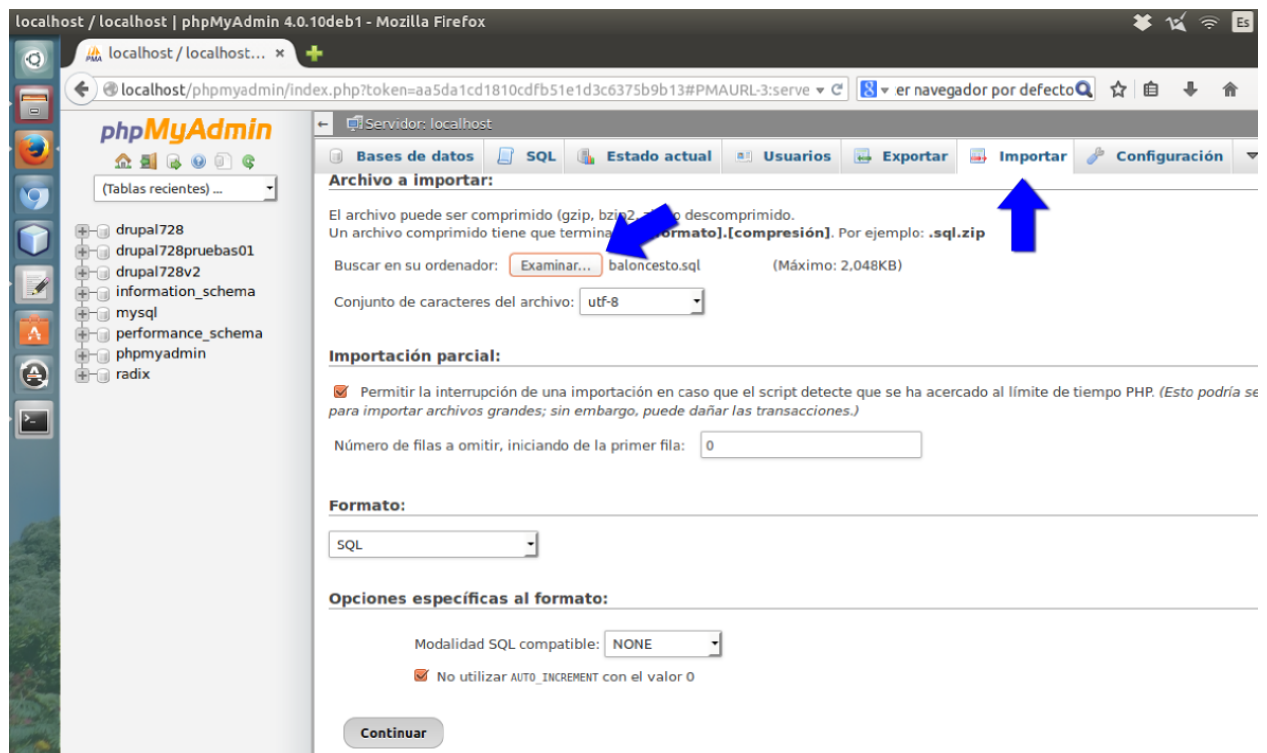
INSERT INTO `socio` (`socioID`, `nombre`, `estatura`, `edad`, `localidad`) VALUES
(1235, 'Bermúdez Espada, Ana María', 186, 46, 'Málaga'),
```

```
(1236, 'Cano Cuenca, Margarita', 161, 48, 'Málaga'),  
...
```

En dos sencillos pasos tendrás preparada y lista para usar la base de datos de ejemplo. Para acceder a PHPMyAdmin debes escribir `http://localhost/phpmyadmin/` en la barra de direcciones de tu navegador. A continuación, teclea el nombre de usuario y la contraseña. Si has instalado MySQL como te hemos indicado, tanto el nombre de usuario como la contraseña deberían ser `root`.



A continuación, haz clic en la pestaña **Importar** y seguidamente en el botón **Examinar**. Selecciona el fichero `baloncesto.sql` que descargaste antes.



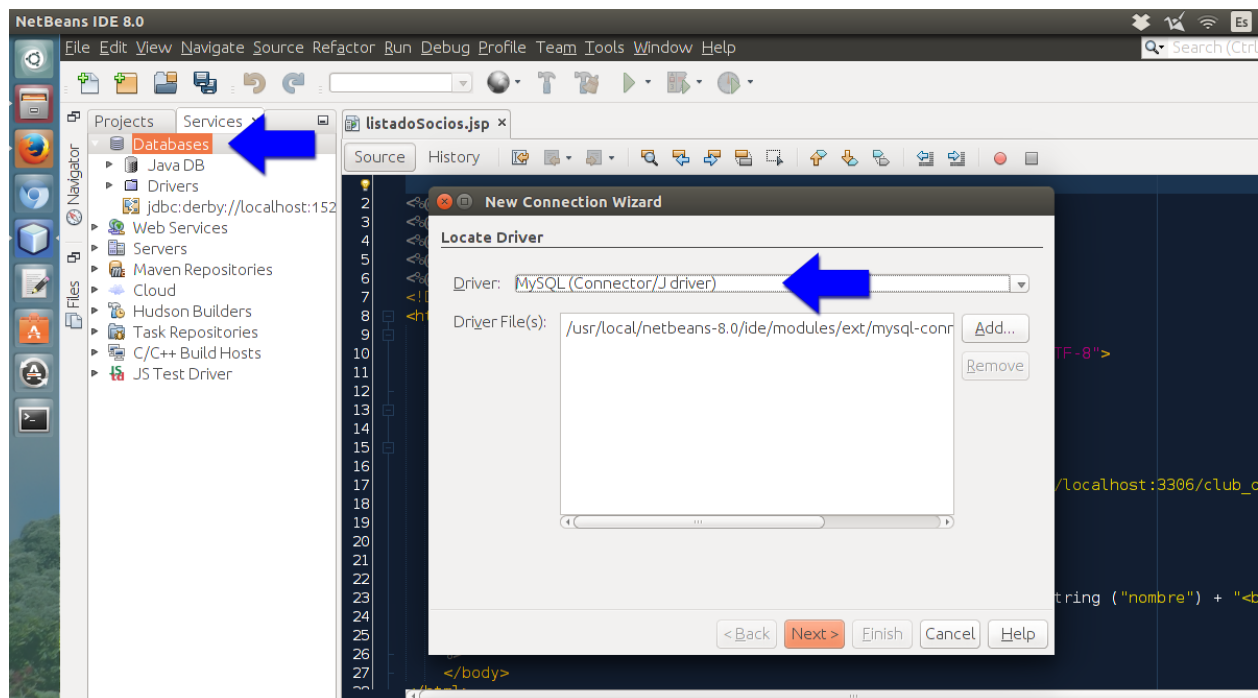
Si has seguido correctamente los pasos, ya tienes la base de datos lista para usar desde un programa escrito en Java.

13.2 Preparación del proyecto de ejemplo

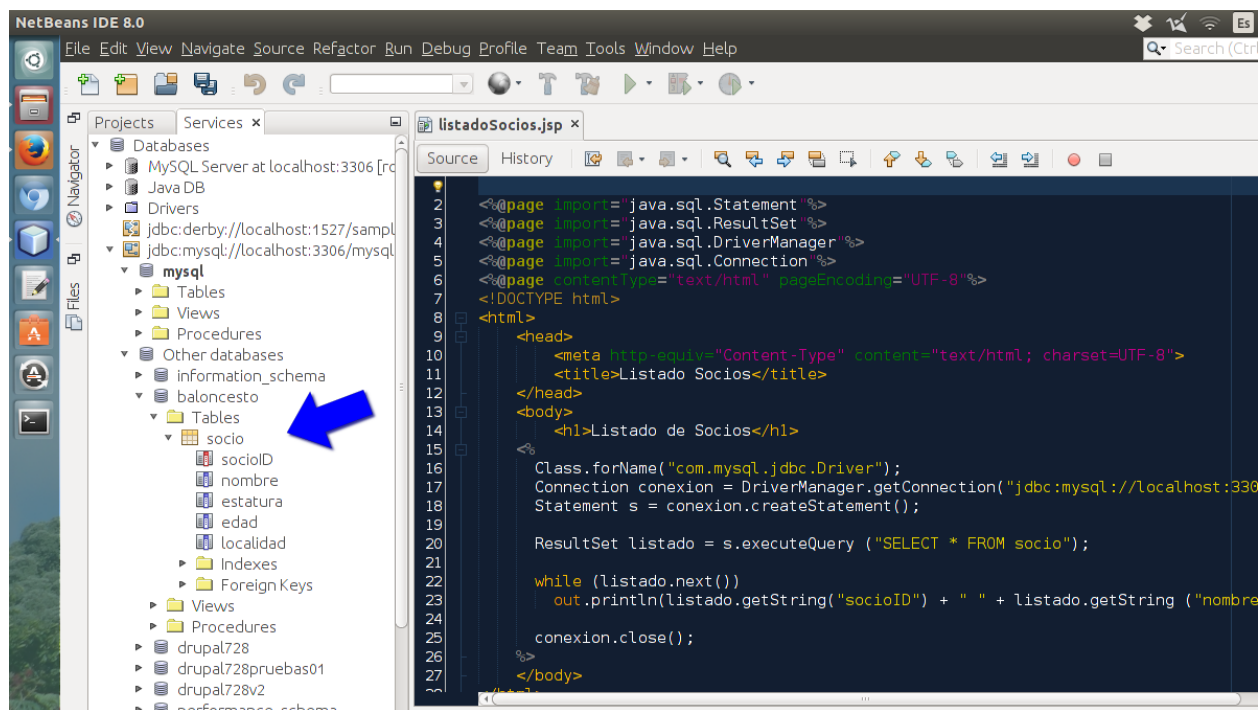
13.2.1 Activar la conexión a MySQL

Abre el entorno Netbeans y crea un nuevo proyecto del tipo **Java Web** - como vimos en el [capítulo 12](#) - y nómbralo **Baloncesto**.

Ahora es necesario activar el servicio de bases de datos. Haz clic en la pestaña **Servicios** (está junto a la pestaña **Proyectos**). Haz clic con el botón derecho en **Bases de datos** y luego selecciona **Nueva conexión**. A continuación aparece una ventana para seleccionar el driver, debes seleccionar **MySQL (Connector/Jdriver)**.



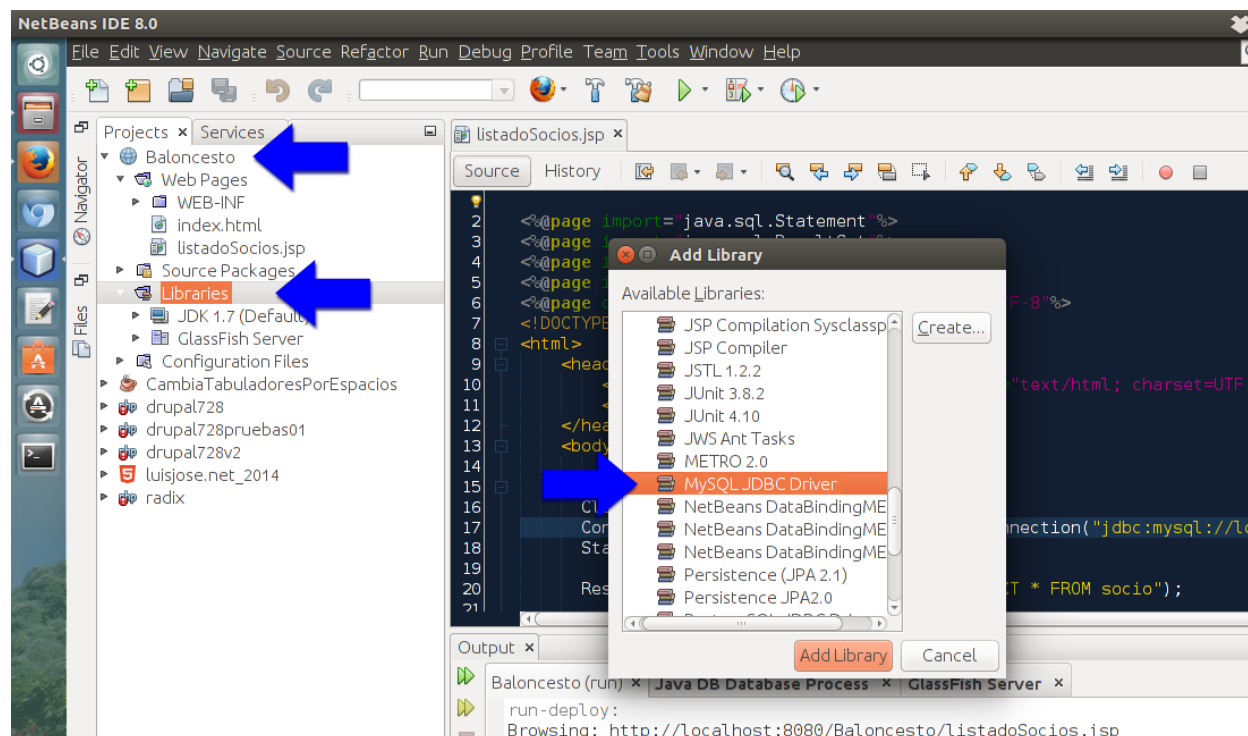
Una vez se haya establecido el servicio, verás las bases de datos disponibles en una estructura de árbol. Puedes ir desplegando hasta encontrar la base de datos baloncesto y la tabla socio, que contiene los campos socioID, nombre, estatura, edad y localidad



Una vez activada la conexión a MySQL, no es necesario realizarla de nuevo para cada proyecto.

13.2.2 Incluir la librería MySQL JDBC

Cada proyecto de aplicación en Java que haga uso de una base de datos MySQL deberá incluir la librería **MySQL JDBC**. Para incluir esta librería, despliega el proyecto **Baloncesto**, haz clic con el botón derecho en **Librerías**, selecciona **Añadir librería** y, por último, selecciona **MySQL JDBC Driver**.



13.3 Listado de socios

El primer ejemplo de acceso a una base de datos mediante Java que veremos será un listado. Si echas un vistazo desde PHPMyAdmin a la tabla socio de la base de datos baloncesto verás que ya contiene información sobre más de 20 socios. Nuestro primer ejemplo será una aplicación en Java que muestre todos esos datos en una página web. Copia el archivo [listadoSocios.jsp](https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/13_JSP_y_BBDD/Baloncesto/listadoSocios.jsp)¹ a la carpeta principal del proyecto (donde está index.html). Para ejecutarlo, haz clic con el botón derecho y selecciona **Ejecutar**.

A continuación tienes el código JSP del listado de socios.

¹https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/13_JSP_y_BBDD/Baloncesto/listadoSocios.jsp

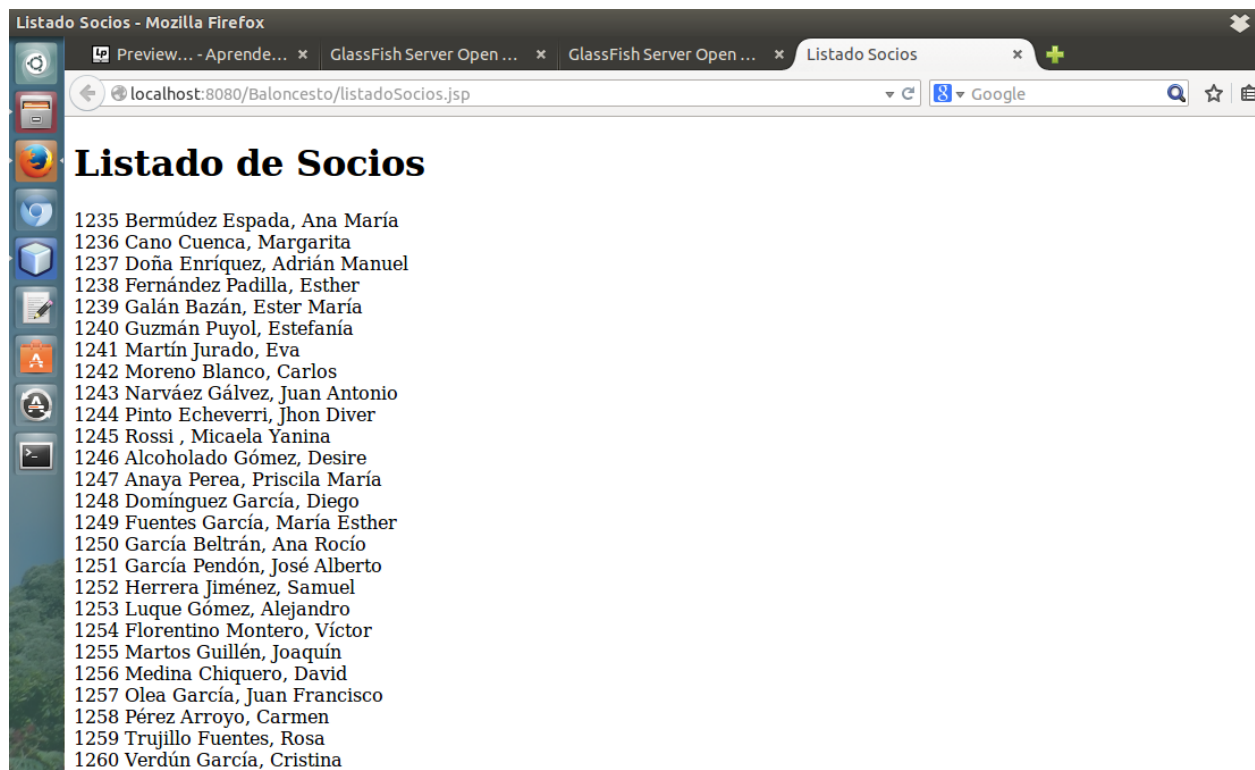
```
<%@page import="java.sql.Statement"%>
<%@page import="java.sql.ResultSet"%>
<%@page import="java.sql.DriverManager"%>
<%@page import="java.sql.Connection"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Listado Socios</title>
  </head>
  <body>
    <h1>Listado de Socios</h1>
    <%
      Class.forName("com.mysql.jdbc.Driver");
      Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/balonce\
sto","root", "root");
      Statement s = conexion.createStatement();

      ResultSet listado = s.executeQuery ("SELECT * FROM socio");

      while (listado.next()) {
        out.println(listado.getString("socioID") + " " + listado.getString("nombre") + "<br>\
");
      }

      conexion.close();
    %>
  </body>
</html>
```

Si todo ha salido bien, al ejecutar el archivo - haciendo clic con el botón derecho sobre el archivo y seleccionando **Ejecutar** - verás por pantalla un listado como el siguiente, con el número y el nombre de cada socio.



Vamos a “diseccionar” el código. Las siguientes tres líneas son obligatorias y deberás copiarlas en todas las páginas JSP que accedan a una base de datos. Lo único que tendrás que cambiar es el nombre de la base de datos - la de nuestro ejemplo se llama baloncesto - según el caso.

```
Class.forName("com.mysql.jdbc.Driver");
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/baloncesto"
, "root", "root");
Statement s = conexion.createStatement();
```

La siguiente línea ejecuta la consulta

```
SELECT * FROM socio
```

sobre la base de datos y guarda el resultado en el objeto listado. Esta consulta extrae todos los datos (*) de la tabla socio. Si quisiéramos obtener esos datos ordenados por nombre, la consulta sería

```
SELECT * FROM socio ORDER BY nombre
```

Aquí está la línea en cuestión:

```
ResultSet listado = s.executeQuery ("SELECT * FROM socio");
```

Una vez extraídos los datos en bruto, hace falta ir sacando cada uno de los registros, es decir, cada una de las líneas. En cada línea se van a mostrar los datos de un socio. Para ello usamos un `while` que va extrayendo líneas mientras quede alguna en el objeto `listado`.

```
while (listado.next()) {
    out.println(listado.getString("socioID") + " " + listado.getString("nombre") + "<br>");
}
```

Observa que el método `getString` toma como parámetro el nombre de un campo. En este ejemplo estamos mostrando únicamente los números de socio con sus correspondientes nombres. Si quisiéramos mostrar también la altura de cada socio, la podemos extraer mediante `listado.getString("altura")`.

13.4 Alta

Para dar de alta un nuevo socio, necesitamos recoger los datos mediante un formulario que puede ser un programa JSP o simplemente una página HTML. Estos datos recogidos por el formulario serán enviados a otra página encargada de grabarlos en la tabla `socio` de la base de datos baloncesto mediante el comando `INSERT` de SQL.

A continuación se muestra el código de `formularioSocio.jsp` que recoge los datos del nuevo socio. Como puedes comprobar es un simple formulario HTML y no contiene código en Java.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h2>Introduzca los datos del nuevo socio:</h2>
    <form method="get" action="grabaSocio.jsp">
      Nº socio <input type="text" name="numero"/></br>
      Nombre <input type="text" name="nombre"/></br>
      Estatura <input type="text" name="estatura"/></br>
      Edad <input type="text" name="edad"/></br>
      Localidad <input type="text" name="localidad"/></br>
      <input type="submit" value="Aceptar">
    </form>
  </body>
</html>
```

El siguiente programa - `grabaSocio.jsp` - se encarga de tomar los datos que envía `formularioSocio.jsp` y grabarlos en la base de datos.

Igual que en los programas del [capítulo 12](#), incluimos esta línea para poder recoger correctamente cadenas de caracteres que contienen tildes, la letra ñ, etc.

```
request.setCharacterEncoding("UTF-8");
```

La variable `insercion` es una cadena de caracteres en la que se va componiendo, a base de ir juntando trozos, la sentencia SQL correspondiente a la inserción.

```
String insercion = "INSERT INTO socio VALUES ("
    + Integer.valueOf(request.getParameter("numero"))
    + ", '" + request.getParameter("nombre")
    + "', " + Integer.valueOf(request.getParameter("estatura"))
    + ", " + Integer.valueOf(request.getParameter("edad"))
    + ", '" + request.getParameter("localidad") + "'");
```

Una vez que se han recogido los datos del formulario y se ha creado la cadena, en la variable `insercion` debería quedar algo como esto:

```
INSERT INTO socio VALUES (6789, "Brito Fino, Alan", 180, 40, "Benalmádena")
```

Por último, se ejecuta la sentencia de inserción mediante `s.execute(insercion)`.

A continuación tienes el código completo de `grabaSocio.jsp`:

```
<%@page import="java.sql.Statement"%>
<%@page import="java.sql.ResultSet"%>
<%@page import="java.sql.DriverManager"%>
<%@page import="java.sql.Connection"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <%
      Class.forName("com.mysql.jdbc.Driver");
      Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/balon\
cesto", "root", "root");
      Statement s = conexion.createStatement();
```

```
request.setCharacterEncoding("UTF-8");
String insercion = "INSERT INTO socio VALUES ("
    + Integer.valueOf(request.getParameter("numero"))
    + ", '" + request.getParameter("nombre")
    + "', " + Integer.valueOf(request.getParameter("estatura"))
    + ", " + Integer.valueOf(request.getParameter("edad"))
    + ", '" + request.getParameter("localidad") + "')";

s.execute(insercion);
conexion.close();
%>
Socio dado de alta.
</body>
</html>
```



Al componer una sentencia SQL concatenando trozos, es frecuente que se produzcan errores porque faltan o sobran comillas, paréntesis, comas, etc. Si se produce un error al ejecutar la sentencia, la manera más fácil de detectarlo es mostrar por pantalla la sentencia. Por ejemplo, si la sentencia es una inserción de datos y la cadena de caracteres que la contiene se llama `insercion`, como en el ejemplo que acabamos de ver, simplemente tendríamos que escribir: `out.println(insercion)`.

13.5 Borrado

El borrado, al igual que el alta, se realiza en dos pasos. En el primer paso, se carga la página `pideNumeroSocio.jsp` que muestra en una tabla todos los socios, cada uno en una fila. Junto a cada socio se coloca un botón **borrar** que al ser pulsado envía el número de socio a la página `borraSocio.jsp` que se encarga de ejecutar la sentencia SQL de borrado.

La tabla con los socios y los botones de borrado quedaría como se muestra a continuación:



Código	Nombre	Estatura	Edad	Localidad	
1235	Bermúdez Espada, Ana María	186	46	Málaga	<input type="button" value="borrar"/>
1236	Cano Cuenca, Margarita	161	48	Málaga	<input type="button" value="borrar"/>
1237	Doña Enríquez, Adrián Manuel	158	31	Málaga	<input type="button" value="borrar"/>
1238	Fernández Padilla, Esther	183	26	Málaga	<input type="button" value="borrar"/>
1239	Galán Bazán, Ester María	184	52	Málaga	<input type="button" value="borrar"/>
1240	Guzmán Puyol, Estefanía	182	30	Málaga	<input type="button" value="borrar"/>
1241	Martín Jurado, Eva	180	44	Málaga	<input type="button" value="borrar"/>
1242	Moreno Blanco, Carlos	191	17	Campanillas	<input type="button" value="borrar"/>
1243	Narváez Gálvez, Juan Antonio	155	22	Campanillas	<input type="button" value="borrar"/>
1244	Pinto Echeverri, Jhon Diver	167	17	Campanillas	<input type="button" value="borrar"/>

A continuación se muestra el código de `pideNumeroSocio.jsp`. El fichero [estilos.css²](https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/13_JSP_y_BBDD/Baloncesto/estilos.css) se encuentra disponible en GitHub, igual que el resto de archivos.

```
<%@page import="java.sql.Statement"%>
<%@page import="java.sql.ResultSet"%>
<%@page import="java.sql.DriverManager"%>
<%@page import="java.sql.Connection"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <link rel="stylesheet" type="text/css" href="estilos.css" />
  </head>
  <body>
    <%
      Class.forName("com.mysql.jdbc.Driver");
      Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/balon\
cesto", "root", "root");
      Statement s = conexion.createStatement();
```

²https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/blob/master/ejemplos/13_JSP_y_BBDD/Baloncesto/estilos.css


```

        ResultSet listado = s.executeQuery ("SELECT * FROM socio");
    %>
    <table>
        <tr><th>Código</th><th>Nombre</th><th>Estatura</th><th>Edad</th><th>Localidad</th></tr>
tr>
    <%
        while (listado.next()) {
            out.println("<tr><td>");
            out.println(listado.getString("socioID") + "</td>");
            out.println("<td>" + listado.getString("nombre") + "</td>");
            out.println("<td>" + listado.getString("estatura") + "</td>");
            out.println("<td>" + listado.getString("edad") + "</td>");
            out.println("<td>" + listado.getString("localidad") + "</td>");
        %>
        <td>
        <form method="get" action="borraSocio.jsp">
            <input type="hidden" name="codigo" value="<%=listado.getString("socioID") %>" />
            <input type="submit" value="borrar">
        </form>
        </td></tr>
    <%
        } // while
        conexion.close();
    %>
    </table>
</body>
</html>

```

A continuación se muestra el código de `borraSocio.jsp`. Como puedes comprobar, la sentencia SQL que borra el registro deseado es un `DELETE` que toma como referencia el número de socio para realizar el borrado.

```

<%@page import="java.sql.ResultSet"%>
<%@page import="java.sql.Statement"%>
<%@page import="java.sql.DriverManager"%>
<%@page import="java.sql.Connection"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <%

```

```

Class.forName("com.mysql.jdbc.Driver");
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/balon\
cesto","root", "root");
Statement s = conexion.createStatement();

s.execute ("DELETE FROM socio WHERE socioID=" + request.getParameter("codigo"));
%>
<script>document.location = "pideNumeroSocio.jsp"</script>
</body>
</html>

```

13.6 CRUD completo con Bootstrap

Los programas que ofrecen la posibilidad de hacer listados y de realizar modificaciones y borrado sobre los registros de una tabla, se denominan CRUD que son las siglas en inglés de *Create*, *Read*, *Update* y *Delete*; en español se diría que es un programa de alta, listado, modificación y borrado. El código fuente completo está disponible en [la carpeta BaloncestoMejorado de nuestro repositorio en GitHub³](#).

Club de Baloncesto						
Nº de socio	Nombre	Estatura	Edad	Localidad		
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<button>+ Añadir</button>	
1235	Bermúdez Espada, Ana María	186	46	Málaga	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1236	Cano Cuenca, Margarita	161	48	Málaga	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1237	Doña Enriquez, Adrián Manuel	158	31	Málaga	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1238	Fernández Padilla, Esther	183	26	Málaga	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1239	Galán Bazán, Ester María	184	52	Málaga	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1241	Martín Jurado, Eva	180	44	Málaga	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1242	Moreno Blanco, Carlos	191	17	Campanillas	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1243	Narváez Gálvez, Juan Antonio	155	22	Campanillas	<button>✎ Modificar</button>	<button>✖ Eliminar</button>
1244	Pinto Echeverri, Jhon Diver	167	17	Campanillas	<button>✎ Modificar</button>	<button>✖ Eliminar</button>

Para dar de alta un nuevo socio, simplemente habría que rellenar los campos del formulario que aparecen en la primera línea y hacer clic en el botón *Añadir*. La aplicación comprueba si el número de socio introducido ya existe en la base de datos y, en tal caso, muestra un mensaje de error; recuerda que el número de socio es único.

³https://github.com/LuisJoseSanchez/aprende-java-con-ejercicios/tree/master/ejemplos/13_JSP_y_BBDD/BaloncestoMejorado

El borrado se lleva a cabo de la misma manera que en el ejemplo del apartado anterior.

El botón *Modificar* nos lleva a otra página que contiene un formulario con todos los datos del socio, donde podemos modificar la información necesaria. A continuación se muestra una captura.



Modificación de socio

Nº de socio: 1235

Nombre: Bermúdez Espada, Ana María

Estatura (en cm): 186 Edad: 40

Localidad: Málaga

✕ Cancelar ✓ Aceptar

Para los estilos - colores, botones, iconos, etc. - hemos utilizado [Bootstrap](http://getbootstrap.com/)⁴. Bootstrap es un **framework** que incluye unos estilos predefinidos para que con solo incluirlo en nuestro proyecto, la aplicación tenga una apariencia bonita y homogénea. Puedes encontrar más información de cómo utilizar este **framework** en [la página oficial de Bootstrap](http://getbootstrap.com/)⁵.

⁴<http://getbootstrap.com/>

⁵<http://getbootstrap.com/>

13.7 Ejercicios





Ejercicio 1

Establece un control de acceso mediante nombre de usuario y contraseña para alguno de los programas de la relación anterior (por ejemplo el que pinta una pirámide). Lo primero que aparecerá por pantalla será un formulario pidiendo el nombre de usuario y la contraseña. Si el usuario y la contraseña son correctos, se podrá acceder al ejercicio; en caso contrario, volverá a aparecer el formulario pidiendo los datos de acceso y no se nos dejará ejecutar la aplicación hasta que iniciemos sesión con un nombre de usuario y contraseña correctos. Los nombres de usuario y contraseñas deben estar almacenados en la tabla de una base de datos.

1

Control de acceso


 Usuario

 Contraseña

ACEPTAR ✓

2

Control de acceso



Acceso permitido a la aplicación.

ACEPTAR ✓

3

Pinta una pirámide

Altura

ACEPTAR ✓

4




Ejercicio 2

Mejora el programa anterior de tal forma que se puedan dar de alta nuevos usuarios para acceder a la aplicación. Si se introduce un nombre de usuario que no sea el administrador (admin) y una contraseña correcta, la aplicación funcionará exactamente igual que el ejercicio anterior. Si se introduce el usuario admin y la contraseña correcta, la aplicación entra en la gestión de usuarios donde se podrán dar de alta nuevos usuarios indicando nombre de usuario y contraseña. No puede haber dos nombres de usuario iguales aunque sí puede haber claves repetidas.


1

Control de acceso

Usuario

 admin


Contraseña

|

ACEPTAR ✓

2

Control de acceso



Tiene acceso al área de gestión de usuarios.

ACEPTAR ✓

3

Gestión de usuarios

Usuario	Contraseña
admin	123456
tux	linux
usuario	usuario
root	toor
usuario2	123

Usuario

Contraseña

AÑADIR USUARIO ✓



Ejercicio 3

Amplía el programa anterior para que se pueda asignar o quitar permiso para ejecutar las aplicaciones de la relación anterior a los distintos usuarios. Por ejemplo, que se pueda especificar que el usuario “jaimito” pueda ejecutar los ejercicios 2, 3 y 5. Para ello, en la base de datos deberá existir una tabla con las parejas (usuario, n° ejercicio). Por ejemplo, si el usuario “jaimito” tiene acceso a los ejercicios 2, 3 y 5; en la tabla correspondiente estarán las parejas (jaimito, 2), (jaimito, 3) y (jaimito, 5). Lo ideal es que la asignación de permisos se haga mediante el marcado de múltiples “checkbox”.

1

Control de acceso

Usuario

admin

Contraseña

.....

ACEPTAR ✓

2

Control de acceso

Tiene acceso al área de gestión de usuarios.

ACEPTAR ✓

3

Gestión de usuarios

Usuario	Contraseña	Permisos
admin	123456	EDITAR
tux	linux	EDITAR
usuario	usuario	EDITAR
root	toor	EDITAR
usuario2	123	EDITAR

Usuario

Contraseña

AÑADIR USUARIO

4

Permisos

Usuario	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
tux	✓	✓	☐	☐	✓	✓	☐	☐	☐	✓	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

✓



Ejercicio 4

Crea una aplicación web que permita hacer listado, alta, baja y modificación sobre la tabla cliente de la base de datos banco. Un cliente tiene su identificador, nombre completo, dirección, teléfono y fecha de nacimiento.

G e s t i b a n k						
Código	Nombre	Dirección	Teléfono	Fecha de nacimiento		
3534534	Cacerolo Tontoñez	Almogía	123456	1963-04-08	EDITAR	BORRAR
456478	Hernán Sánchez	Calle Finlandia, 11	678098867	1972-05-24	EDITAR	BORRAR
45678	Mota	Calle Falsa, 123	555 444333	1980-06-28	EDITAR	BORRAR
555	Luis José	Larios, 10	5555 234233	2013-02-17	EDITAR	BORRAR
65767	Pepito Lupiañez	Alhaurín	867867867	1992-01-10	EDITAR	BORRAR
76859	ignacio	Periquito, 333	555 325476	1995-08-08	EDITAR	BORRAR
789654	Yren	Calle Verdadera, 98	555 98765	1950-06-06	EDITAR	BORRAR
873475933	Maria Sol	Calle Flor	555 123456	1945-01-01	EDITAR	BORRAR
código	nombre	dirección	teléfono	fecha de nacim.	AÑADIR	



Ejercicio 5

Amplía el programa anterior para que se pueda hacer una búsqueda por nombre. El programa buscará la cadena introducida dentro del campo “nombre” y, si hay varias coincidencias, se mostrará una lista de clientes para poder seleccionar uno de ellos y ver todos los datos. Si solo hay una coincidencia, se mostrarán directamente los datos del cliente en cuestión.

1

Gestibank

Código	Nombre	Dirección	Teléfono	Fecha de nacimiento		
3534534	Cacerolo Tontoñez	Almogía	123456	1963-04-08	EDITAR	BORRAR
456478	Hernán Sánchez	Calle Finlandia, 11	678098867	1972-05-24	EDITAR	BORRAR
45678	Mota	Calle Falsa, 123	555 444333	1980-06-28	EDITAR	BORRAR
555	Luis José	Larios, 10	5555 234233	2013-02-17	EDITAR	BORRAR
65767	Pepito Lupiañez	Alhaurín	867867867	1992-01-10	EDITAR	BORRAR
789654	Yren	Calle Verdadera, 98	555 98765	1950-06-06	EDITAR	BORRAR
873475933	Maria Sol	Calle Flor	555 123456	1945-01-01	EDITAR	BORRAR

código

nombre

dirección

teléfono

fecha de nacim.

AÑADIR

nombre

ez

Q BUSCAR

2

Gestibank

Nombre

Cacerolo Tontoñez

DETALLE

Hernán Sánchez

DETALLE

Pepito Lupiañez

DETALLE

3

Hernán Sánchez

Código: 456478

Nombre: Hernán Sánchez

Dirección: Calle Finlandia, 11

Teléfono: 678098867


























Fecha de nacimiento: 1972-05-24

ACEPTAR



Ejercicio 6

Crea una versión web del programa GESTISIMAL (GESTIÓN SIMPLificada de Almacén) para llevar el control de los artículos de un almacén. De cada artículo se debe saber el código, la descripción, el precio de compra, el precio de venta y el stock (número de unidades). La entrada y salida de mercancía supone respectivamente el incremento y decremento de stock de un determinado artículo. Hay que controlar que no se pueda sacar más mercancía de la que hay en el almacén. Aprovecha las opciones que puede ofrecer una interfaz web, por ejemplo para eliminar un artículo ya no será necesario pedir el código sino que se puede mostrar un listado de todos los artículos de tal forma que se puedan borrar un artículo directamente pulsando un botón.

Gestisimal					
Código	Descripción	Precio de compra	Precio de venta	Stock	
2324	MERMELADA DE ALBARICOQUE	1.9	2.3	40	  
34240	TURRÓN BLANDO ALMENDRA	1.75	2.45	250	  
36548	PAN DE MOLDE	0.95	1.8	70	  
4444	CHOCOLATE 150GR	1.9	3.3	5	  
45654	PAN DE MOLDE	1.78	2.14	23	  
6767676	TOFU CON SÉSAMO	1.6	2.85	40	  
67904	CROQUETAS DE SETAS	2.95	3.5	91	  
89567	PATATAS FRITAS	0.37	0.67	90	  
<input type="text" value="código"/>	<input type="text" value="descripción"/>	<input type="text" value="precio de compra"/>	<input type="text" value="precio de venta"/>	<input type="text" value="stock"/>	

Apéndice A. Ejercicios de ampliación



Ejercicio 1: Transcomunicación instrumental mediante ficheros de texto digitales

Se conoce la **transcomunicación instrumental** (TCI) como la técnica de comunicación que utiliza aparatos electrónicos para establecer comunicación con los espíritus. El caso de TCI más conocido es el de la psicofonía.

Para realizar una psicofonía se pone a grabar al aire un dispositivo en un lugar donde previamente hayan ocurrido fenómenos extraños o simplemente se invoca la entidad que queremos que se comunique mediante el ritual adecuado. Posteriormente, se escucha la grabación para comprobar si se ha grabado algo: un mensaje, una palabra, un grito, un lamento...

Los sonidos extraños se graban gracias a lo que se llama en términos parapsicológicos una señal portadora, puede ser el chasquido de una rama, una racha de viento, un golpe. La entidad comunicante aprovecha ese sonido para supuestamente moldearlo y convertirlo en un mensaje inteligible. Muchas de las psicofonías se realizan utilizando como portadora el ruido que produce una radio que no está sintonizada.

El ejercicio que se propone es la realización de un **programa que permita realizar la TCI mediante la creación y el posterior análisis de ficheros de texto**.

En primer lugar, el programa deberá generar la portadora, que será un fichero de texto creado con caracteres producidos de forma aleatoria. Se podrán especificar varios parámetros como el número de líneas o páginas que se quieren generar, si se incluyen letras únicamente o bien letras y números, frecuencia aproximada de espacios, saltos de línea y signos de puntuación, etc. Se obtienen portadoras de calidad - con más probabilidad de contener mensajes - cuando la proporción de las letras que contienen éstas coincide con las del idioma español.

Frecuencia de aparición de letras:

http://es.wikipedia.org/wiki/Frecuencia_de_aparici%C3%B3n_de_letras

Una vez creado el fichero de texto, el programa deberá ser capaz de analizarlo, al menos de una forma superficial, de modo que nos resulte fácil encontrar los posibles mensajes. Para realizar este análisis, el programa irá comparando una a una todas las palabras que aparecen en el texto con las que hay en el diccionario de español. En caso de coincidencia, la palabra o palabras encontradas deben aparecer resaltadas con un color diferente al resto del texto o bien en vídeo inverso. Opcionalmente se pueden mostrar estadísticas con el número total de palabras encontradas, distancia media entre esas palabras, palabras reales por cada mil palabras generadas, etc.

El alumno puede encontrar ficheros con las palabras que tiene el idioma español en formato digital en la siguiente dirección. Se trata de una muestra, hay otros muchos diccionarios disponibles en internet:

<http://lasr.cs.ucla.edu/geoff/ispell-dictionaries.html#Spanish-dicts>



Ejercicio 2: Colección de discos ampliado (con canciones)

Este ejercicio consiste en realizar una mejora del ejercicio del [capítulo 10](#) que permite gestionar una colección de discos. Ahora cada disco contiene canciones. Deberás crear la clase `Cancion` con los atributos y métodos que estimes oportunos. Añade el atributo `canciones` a la clase `Disco`. Este atributo `canciones` debe ser un array de objetos de la clase `Cancion`.

Fíjate bien que `Cancion` NO ES una subclase de `Disco` sino que ahora cada disco puede tener muchas canciones, que están almacenadas en el atributo `canciones`.

Modifica convenientemente el método `toString` para que al mostrarse los datos de un disco, se muestre también la información sobre las canciones que contiene.

La aplicación debe permitir añadir, borrar y modificar las canciones de los discos.