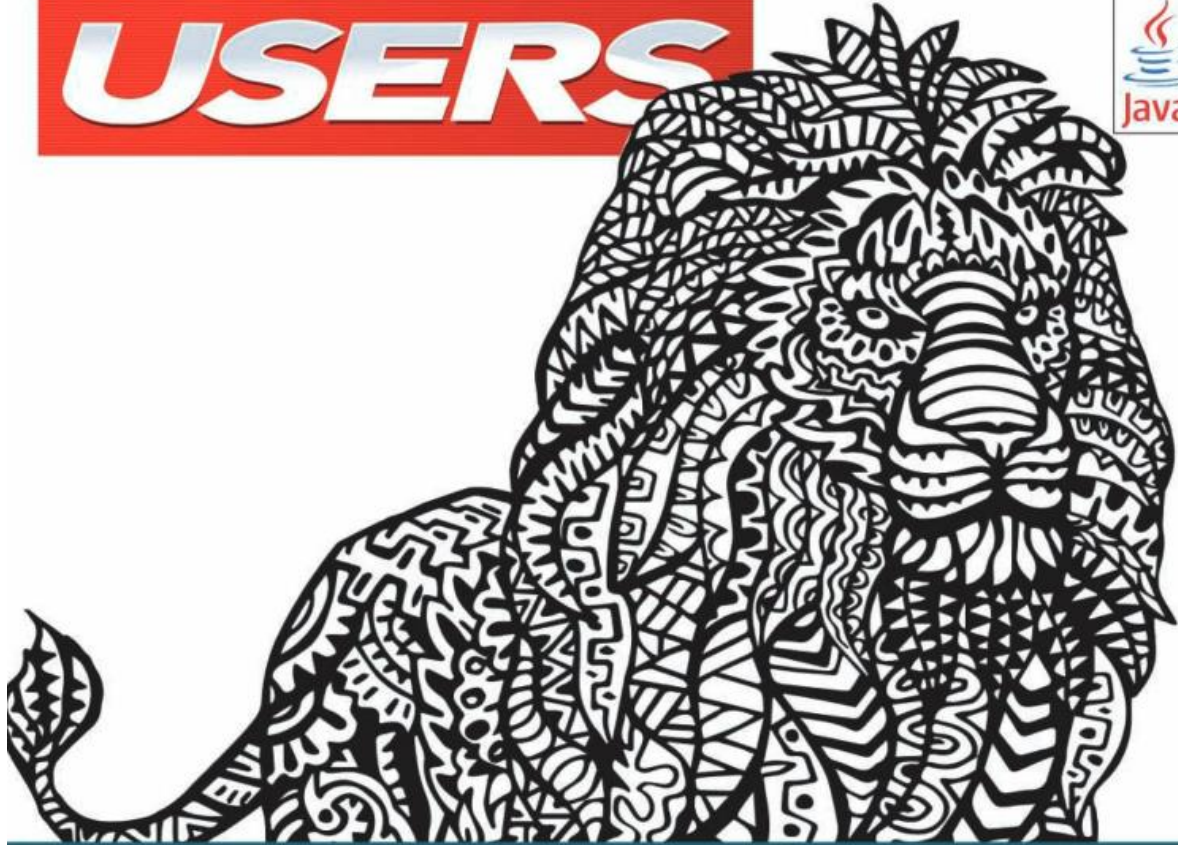


USERS



Programación en

Java™

Vol. IV

**Estructuras dinámicas
Bases de datos - Android Studio
Búsqueda y ordenamiento**

Programación en

Java™

Vol. IV

**Estructuras dinámicas - Bases de datos
Android Studio - Búsqueda y ordenamiento**

USERS

Título: Programación en Java IV / **Autor:** Carlos Arroyo Díaz

Coordinador editorial: Miguel Lederkremer / **Edición:** Claudio Peña

Maquetado: Marina Mozzetti / **Colección:** USERS ebooks - LPCU293

Copyright © MMXIX. Es una publicación de Six Ediciones. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio actual o futuro, sin el permiso previo y por escrito de Six Ediciones. Su infracción está penada por las leyes 11723 y 25446. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen y/o analizan. Todas las marcas mencionadas en este libro son propiedad exclusiva de sus respectivos dueños. Libro de edición argentina.

Arroyo Díaz, Carlos

Programación en Java : Estructuras dinámicas. Bases de datos. Android studio. Búsqueda y ordenamiento / Carlos Arroyo Díaz. - 1a ed. - Ciudad Autónoma de Buenos Aires : Six Ediciones, 2019.

Libro digital, PDF - (Programación en JAVA ; 4)

Archivo Digital: online

ISBN 978-987-4958-17-4

1. Lenguajes de Programación. I. Título.
CDD 005.4

USERS

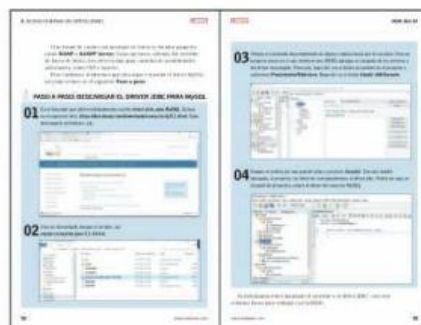
Programación en

Java™

Vol. IV

ACERCA DE ESTE CURSO

Java es uno de los lenguajes más robustos y populares en la actualidad, existe hace más de 20 años y ha sabido dar los giros adecuados para mantenerse vigente. Este curso de Programación en Java nos enseña, desde cero, todo lo que necesitamos para aprender a programar y, mediante ejemplos prácticos, actividades y guías paso a paso, nos presenta desde las nociones básicas de la sintaxis y codificación en Java hasta conceptos avanzados como el acceso a bases de datos y la programación para móviles.



ACERCA DE ESTE VOLUMEN

En este volumen se enseña el uso de las estructuras de datos dinámicas, el acceso a bases de datos y la programación Java para Android.



► SOBRE EL AUTOR

Carlos Arroyo Díaz es programador, escritor especializado en tecnologías y docente. Se desempeña como profesor de Informática General, Java y Desarrollo Web. También ha trabajado como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires.

► RedUSERS

En nuestro sitio podrá encontrar noticias relacionadas y participar de la comunidad de tecnología más importante de América Latina.

► RedUSERS PREMIUM

RedUSERS PREMIUM la biblioteca digital de USERS. Accederás a cientos de publicaciones: Informes; eBooks; Guías; Revistas; Cursos. Todo el contenido está disponible online - offline y para cualquier dispositivo. Publicamos, al menos, una novedad cada 7 días





RedUSERS **PREMIUM**

- ✓ Cientos de publicaciones USERS por una mínima cuota mensual.
- ✓ Siempre, donde vayas. On Line - Off Line. En cualquier dispositivo.
- ✓ Al menos 1 novedad semanal. Son 680 publicaciones y sumando...!!
- ✓ Incluye: eBooks - Informes USERS - Guías USERS - Revistas USERS y Power – CURSOS

SUSCRÍBETE



usershop.redusers.com



+54-11-4110-8700



usershop@redusers.com

ACERCA DE ESTE CURSO

Java es un lenguaje de programación que sigue afianzándose como un estándar de la web y, por eso, año tras año, aparece en el tope de las búsquedas laborales de programadores.

Es por esto que hemos creado este curso de **Programación en Java**, donde encontrarán todo lo necesario para iniciarse o profundizar sus conocimientos en este lenguaje de programación.

El curso está organizado en cuatro volúmenes, orientados tanto a quien recién se inicia en este lenguaje, como a quien ya está involucrado y enamorado de Java.

En el **primer volumen** se realiza una revisión de las características de este lenguaje, también se entregan las indicaciones para instalar el entorno de desarrollo y, posteriormente, se analizan los elementos básicos de la sintaxis y el uso básico de las estructuras de control.

En el **segundo volumen** se presentan las clases en Java, se realiza una introducción a los conceptos asociados a la Programación Orientada a Objetos y también se profundiza en el uso de la herencia, colaboración entre clases y polimorfismo.

El **tercer volumen** contiene información sobre el uso de las clases abstractas e interfaces, el manejo de excepciones y la recursividad.

Finalmente, en el **cuarto volumen** se enseña el uso de las estructuras de datos dinámicas, el acceso a bases de datos y la programación Java para Android.

Sabemos que aprender todo lo necesario para programar en Java en tan solo cuatro volúmenes es un tremendo desafío, pero conforme vamos avanzando, el camino se va allanando y las ideas se tornan más claras.

¡Suerte en el aprendizaje!

SUMARIO DEL VOLUMEN IV

01 INTRODUCCIÓN / 6

Nodo / Listas / Pila / Colas /

ALGORITMOS DE BÚSQUEDA / 16

Búsqueda Lineal / Búsqueda Binaria

ALGORITMOS DE ORDENAMIENTO / 25

Ordenamiento por selección / Ordenamiento por inserción /
Ordenamiento por Combinación

ÁRBOLES / 36

02 BASES DE DATOS / 44

CONEXIÓN A UNA BBDD / 45

Establecer la conexión a la base de datos /
Crear el Objeto Statement / Ejecutar una sentencia SQL

INSTALACIÓN DE UN SERVIDOR DE BBDD / 46

Instalar un Servidor de MySQL / Descargar el DriverJDBC para MySQL

CONEXIÓN A MySQL / 56

Ap INTRODUCCIÓN AL MUNDO MOBILE / 62

Ventajas de usar y desarrollar para Android

ANDROID STUDIO / 63

Instalar Android Studio / Crear un Proyecto en Android

ESTRUCTURA BÁSICA DE UN PROYECTO / 71

Exploración de un Proyecto / AndroidManifest /
Componentes de una aplicación Android

ACTIVIDADES BÁSICAS / 76

Ciclo de Vida / XML

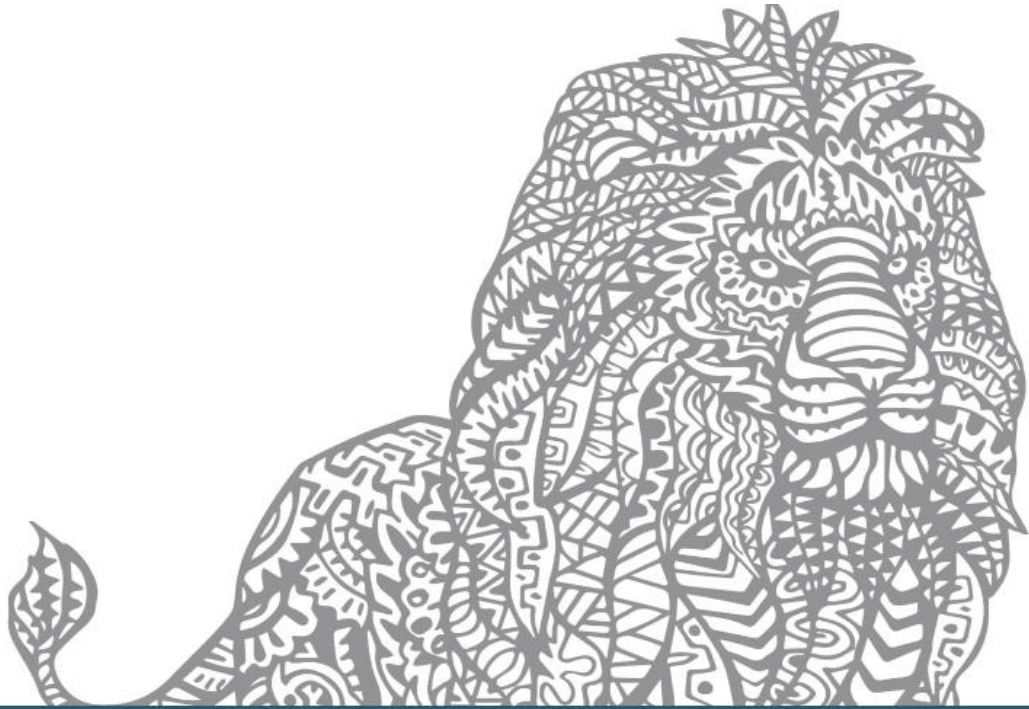
INTERACCIÓN ENTRE JAVA Y ANDROID / 79

Cómo usar el Archivo R

PRIMERAS APPS / 82

Uso del emulador de Android Studio





Estructuras de Datos Dinámicas

En capítulos anteriores, trabajamos con estructuras de datos estáticas, como los arreglos; pero existe otro tipo de soluciones que resultan eficientes para el tratamiento de la información o la buena administración de la memoria. En este capítulo explicaremos qué son las estructuras de datos dinámicas y veremos cómo utilizarlas. También entenderemos cómo funcionan los árboles en una estructura de búsqueda avanzada.



01

INTRODUCCIÓN

Trabajar con matrices y vectores a la manera de estructuras de datos es muy común en programación, pero existen escenarios en donde no es suficiente, por lo que se hace necesario utilizar **estructuras dinámicas**.

Dentro de las estructuras estáticas sobresalen los arrays, en cambio, un vector, que proviene de la clase **Vector**, representa al grupo de las estructuras dinámicas porque nos permite almacenar tantos elementos como queramos, siempre y cuando haya memoria disponible. En un vector podemos insertar o eliminar elementos a favor o en detrimento de la memoria. Sin embargo, en el sentido estricto del concepto un vector no es una estructura de datos.

La clase **Vector** encapsula y mantiene una estructura de datos interna para así almacenar información.

En definitiva, una estructura dinámica permite que almacenemos una cantidad variable de datos, los que pueden aumentar o decrecer en relación directa con la memoria que ocupen.

Este tipo de estructuras se generan encadenando unidades de información conocidas como **nodos**. Un conjunto de nodos enlazados entre sí corresponden a una estructura dinámica, a esto hay que agregarle un conjunto de operaciones que están asociadas, las que permiten manipular y acceder a los nodos y a su información.

Nodo

Se trata de la unión de un **dato** más una **referencia** (o dirección de memoria) a otro nodo.

El dato (o información) puede ser cualquier tipo de dato simple, estructura de datos e, incluso, un objeto. La dirección al siguiente nodo es un **puntero**. La forma de representar un nodo es la siguiente:

```
public class Nodo<T>{  
    private T dato;  
    private Nodo<T> ref;  
    instrucciones...  
    //getters y setters  
}
```

En una instancia del **Nodo<T>** se puede almacenar una unidad de información y, desde luego, la dirección a otro nodo. Cuando tenemos una cantidad de este tipo de bloques, nos encontramos con las denominadas **listas enlazadas**.



Figura 1. El gráfico nos muestra la parte básica de la estructura de datos, un nodo.

Listas

Una **lista** es un conjunto de nodos, este cuenta con dos partes, los **datos** y el **apuntador** al siguiente nodo. El acceso a la lista siempre será a partir del primer nodo, de tal forma que hay que mantenerlo referenciado.

En el caso de una lista enlazada, nos referimos a la estructura dinámica básica que funciona como base para otras estructuras.

No olvidemos que las estructuras dinámicas se definen como un conjunto de nodos enlazados, sumados a un conjunto de operaciones, entre las que podemos enumerar: **agregarAlFinal**, **agregarAlPrincipio**, **buscar** y **eliminar**.

Veamos un ejemplo para entender cómo se implementan las listas y las operaciones:



Tipos de estructuras

Son el soporte fundamental de un conjunto grande de algoritmos, y la elección de una estructura adecuada facilita enormemente la producción de programas. Muy por el contrario, la mala elección de una buena estructura de datos nos traerá programas muy complejos. La capacidad que tienen este tipo de estructuras para incorporar o deshacerse de elementos a medida que sea necesario determinará el tipo de ellas que estamos usando: **estáticas** o **dinámicas**.

```
package estructurasdinamicas;
import java.util.ArrayList;
import java.util.List;
public class Listas {
    public static void main(String[] args) {

        List<String> paisesLista = new ArrayList<String>();
        paisesLista.add("Argentina");
        paisesLista.add("Brasil");
        paisesLista.add("Chile");
        paisesLista.add("Paraguay");
        paisesLista.add("Uruguay");

        paisesLista.size();
        paisesLista.get(0);
        paisesLista.remove(0);
        paisesLista.remove("Paraguay");

        System.out.println("Los países que asistieron son: " + paisesLista);
    }
}
```

Como sabemos, las listas pueden guardar cualquier tipo de datos, aquí guardamos del tipo **String**.

Luego, con el método **add**, hemos agregado 5 valores. También es posible realizar algunas operaciones en nuestro listado:

- Para obtener la cantidad de la lista usamos el método **size**:

```
paisesLista.size();
```

- Si queremos obtener un elemento específico, usamos el método **get** y como argumento colocamos el índice de la lista:

```
paisesLista.get(0);
```

- Para eliminar un elemento de la lista usamos **remove**:

```
paísesLista.remove(0);  
paísesLista.remove("Paraguay");
```

Luego, si imprimimos la lista, debería darnos lo siguiente:

Los países que asistieron son [Brasil, Chile, Uruguay]

- Para imprimir a los países de una manera individual, podemos recurrir a un **for**:

```
for (int i=0; i<=paísesLista.size()-1; i++) {  
    System.out.println(paísesLista.get(i));  
}
```

- Usamos un **iterador** para recorrer la lista e imprimir todos sus valores, antes debemos importar para este caso:

```
import java.util.Iterator;
```

Luego escribimos:

```
Iterator i = paísesLista.iterator();  
while(i.hasNext()) {  
    System.out.println(i.next());  
}
```

- Para eliminar todos los elementos del listado debemos usar:

```
paísesLista.clear();
```

- Para saber si nuestro listado contiene algún elemento:

```
paísesLista.isEmpty();
```

Devolverá **true** o **false**.

- Para conocer si nuestra lista contiene, por ejemplo, el nombre de **Brasil**, utilizamos:

```
paísesLista.contains("Brasil");
```

Esto también nos devolverá un **true** o un **false**.

- Para modificar algún dato de nuestra lista, por ejemplo en el índice **2** que contiene el nombre de **Chile**, utilizamos el siguiente método:

```
paísesLista.set(2, "Bolivia");
```

Entonces con esto cambiamos **Chile** por **Bolivia**.

- Para extraer una lista que contenga los nombres entre un índice y otro debemos utilizar:

```
paísesLista.subList(0, 2);
```

Si queremos imprimir el código anterior:

```
System.out.println(paísesLista.subList(0, 2));
```

El resultado será:

```
[Argentina, Brasil]
```

Notemos que se transforman los elementos desde el índice inicial (**0**) hasta el elemento anterior al índice final (**2**), despreciando el elemento que se encuentra en el índice final, por eso no da **Argentina** y **Chile**.

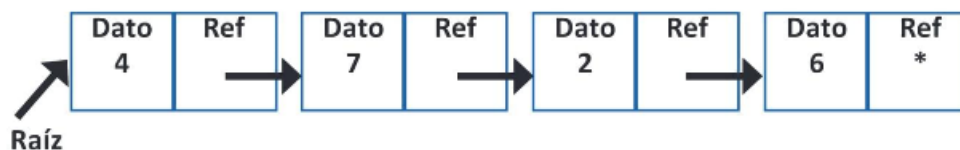


Figura 2. El gráfico nos muestra una lista enlazada, en la que podemos observar los nodos y sus referencias.

Pila

Podemos decir que una **pila** o **stack** es una versión restringida de una lista enlazada. Para entender un poco más este concepto, debemos saber que las pilas utilizan una tecnología **UEPS**, es decir, último en entrar, primero en salir o en inglés **LIFO** (last in/first out).

Como ejemplo podemos reproducir discos de música y los vamos colocando en forma apilada, de tal manera que el último que colocamos es el primero que escucharemos. Por eso se dice que podemos agregar o eliminar elementos solo de la parte superior de la pila.

Existen varias maneras de implementar una pila, ya sea a través de un arreglo, que debe ser fijo; también por medio de un vector o simplemente creando una lista enlazada en donde cada elemento de la pila forma un nodo de la lista.

A esto lo que llamamos una **representación dinámica** y no existe limitación en su tamaño, excepto la memoria.

Una pila puede estar **vacía** (cero elementos) o **llena** (en la representación con un arreglo, si se ha llegado al último elemento). Si intentamos extraer un elemento de una pila vacía, se producirá una excepción, debido a que esa operación no es posible; esta situación se denomina **desbordamiento negativo** o **underflow**. Por el contrario, si intentamos poner un elemento en una pila llena, producirá un efecto contrario de desbordamiento u **overflow**.

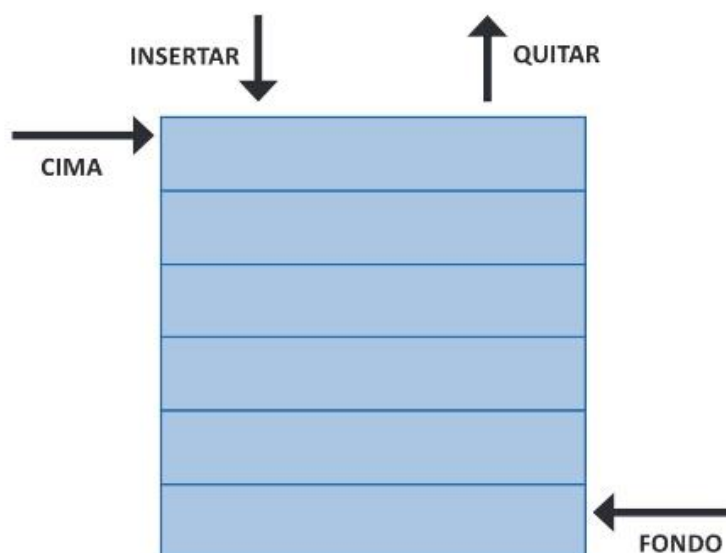


Figura 3. Aquí vemos cómo opera básicamente una pila.

Para insertar información en una pila, utilizaremos el método **push** y, para eliminar, el método **pop**.

El método **push** se insertará al frente de la pila de tal forma que empujará todos los elementos, por eso utilizaremos este método cada vez que necesitemos insertar al inicio de la clase. En cambio **pop** será el encargado de extraer el elemento que necesitemos eliminar de la lista.

Veamos un ejemplo en el que implementaremos las operaciones básicas, como **insertar** y **extraer** los datos de una pila de tipo entero:

```
import java.util.Stack;
public class Principal {
    public static void main(String[] args) {

        Stack<Libro> pila = new Stack<Libro>();

        Libro 1 = new Libro("El Quijote de la Mancha", "Cervantes");
        Libro 2 = new Libro("La Divina Comedia", "Dante");
        Libro 3 = new Libro("Macbeth", "Shakespeare");

        pila.push(1); // agrega un libro a la pila
        pila.push(2);
        pila.push(3);

        System.out.println(pila.peek().getTitulo()); // el último elemento adicionado

        while (!pila.isEmpty()){ // muestra la pila completa
            System.out.println(pila.pop().getTitulo()); //extrae un elemento de la pila
        }

    }
}
```

Luego, una clase llamada **Libro** que contendrá los **getter** y **setter** de los atributos tanto de los títulos como de los autores:

```
public class Libro {  
    private String titulo;  
    private String autor;  
  
    public Libro() {  
        this.titulo = "";  
        this.autor = "";  
    }  
  
    public Libro(String titulo, String autor) {  
        this.titulo = titulo;  
        this.autor = autor;  
    }  
  
    public String getTitulo() {  
        return titulo;  
    }  
  
    public void setTitulo(String titulo) {  
        this.titulo = titulo;  
    }  
  
    public String getAutor() {  
        return autor;  
    }  
  
    public void setAutor(String autor) {  
        this.autor = autor;  
    }  
}
```

Colas

Las **colas** o **queue** son conocidas como estructuras **FIFO** (first-in/first-out, es decir, 'primero en entrar', 'primero en salir'), debido al orden de inserción y de extracción de elementos de la cola. Hay que destacar que tienen numerosas aplicaciones: colas de mensajes, colas de impresión, colas de prioridades, etcétera.

Veamos un ejemplo en el que implementaremos este algoritmo. En principio tenemos la clase **Principal** que contiene los elementos que van a producir la cola:

```
import java.util.PriorityQueue;
import java.util.Queue;

public class Colas {
    public static void main(String[] args) {
        Queue<Persona> cola=new PriorityQueue<Persona>();

        cola.add(new Persona("Juan Solis", 1));
        cola.add(new Persona("Rosa Agrelo", 3));
        cola.add(new Persona("Manuel Loiacono", 2));
        cola.add(new Persona("Pedro Marat", 1));

        while (!cola.isEmpty()) {
            Persona a = cola.remove();
            System.out.println(a.getNombre() + " " + a.getTipo());
        }
    }
}
```

Luego tenemos la clase **Persona**, que contendrá los **getter** y **setter**:

```
public class Persona implements Comparable<Persona>{
    private String nombre;
    private int tipo; // 1 normal, 2 tercera edad, 3 embarazada

    public Persona(String nombre, int tipo) {
```

```
        this.nombre = nombre;
        this.tipo = tipo;
    }

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public int getTipo() {
        return tipo;
    }
    public void setTipo(int tipo) {
        this.tipo = tipo;
    }

    @Override
    public int compareTo(Persona o) {
        if (tipo < o.getTipo()) {
            return 1;
        } else if (tipo > o.getTipo()) {
            return -1;
        } else {
            return 0;
        }
    }
}
```

Persona implementa de la interfaz **Comparable**, y también declaramos **tipo**, que implementa prioridades en las personas.

ALGORITMOS DE BÚSQUEDA

Una **búsqueda de datos** implica que vamos a determinar si un valor, al que llamaremos **clave de búsqueda**, se encuentra presente. En el caso de que sea positivo, podremos también saber su ubicación.

Dentro de los algoritmos que se encuentran en este orden, encontramos la **búsqueda lineal simple** y la **búsqueda binaria**.

Búsqueda lineal

La búsqueda lineal es la más sencilla e intuitiva de implementar. El algoritmo busca en forma secuencial un elemento, evalúa si este elemento es igual al primero, segundo, tercero, y así sucesivamente hasta compararse con el buscado; al encontrarlo devuelve el índice. Es un algoritmo de complejidad **O(n)**.

Como ejemplo tenemos los siguientes valores en un arreglo:

33 22 55 2 47 3 16 24 19 5 87

Creamos un programa que busque el número **19**, si usamos este algoritmo de búsqueda, comprobará el primer elemento del arreglo (**33**) y verificará si coincide con la clave de búsqueda. Si es de esta manera, va a evaluar el siguiente elemento (**22**), y así sucesivamente hasta encontrarlo; si estuviera el elemento que estamos buscando, devolverá el índice **8**.

Si no hubiera coincidido (si hubiéramos puesto **29**), el programa debería devolvernos un **valor centinela** (por ejemplo, **-1**).

Veamos un ejemplo en el que tendremos dos clases, una llamada **ArregloLineal**, que contendrá un arreglo de enteros aleatorios, y otra clase **BusquedaLineal** que buscará un elemento del arreglo. Veamos el código de la clase **ArregloLineal**:

```
package estructurasdinamicas.Busquedas;  
import java.util.Random;  
import java.util.Arrays;  
public class ArregloLineal {  
    private int[] datos;
```

```
private static final Random generador = new Random();
    public ArregloLineal(int tamaño){
        datos = new int[tamaño];
        for (int i=0;i<tamaño;i++)
            datos[i]=10+generador.nextInt(90);
    }
```

Se importa la clase **Random**, que realiza números aleatorios dada una lista o array. Luego se crea un arreglo y se rellena con números aleatorios en el rango **10** a **99**.

```
public int busquedaLineal(int claveBusqueda){
    for(int indice=0;
    indice<datos.length;indice++)
        if (datos[indice]==claveBusqueda)
            return indice;
    return -1;
}
```

El método **busquedaLineal** se encarga de hacer la búsqueda del elemento. Itera de forma secuencial por el arreglo mediante un **for**, y mediante un **if** compara el índice y verifica si está; en caso contrario, retorna **-1**.

```
@Override
public String toString(){
    return Arrays.toString(datos);
}
}
```

Luego el código de la clase **BusquedaLineal** será el siguiente:

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

```
public static void main(String[] args ){
    Scanner entrada = new Scanner( System.in );
    int enteroBusqueda; //clave de búsqueda
    int posicion; //ubicación de la clave
```

En el siguiente bloque se crea el array y se muestra en pantalla:

```
ArregloLineal arregloBusqueda=new ArregloLineal(10);
System.out.println(arregloBusqueda );
```

Ahora obtendremos la entrada del usuario. Y se leerá el primer entero:

```
System.out.print("Escriba un valor entero (-1 para terminar):
");
enteroBusqueda = entrada.nextInt();
```

Finalmente se recibirá en forma iterativa (con un **while**) un entero como entrada. Un **if-else** determinará si el entero se encontró y en qué posición. Cuando se ingrese **-1**, el ciclo se dará por finalizado.

```
while(enteroBusqueda != -1){
    // realiza una búsqueda lineal
    posicion=arregloBusqueda.busquedaLineal(enteroBusqueda);
```



Rendimiento y eficacia

Siempre estaremos atentos a los algoritmos que cumplen con la regla de rendimiento y eficacia, puesto que algunos son bastante simples de implementar y de entender, pero no siempre son los más eficaces. Entonces, se presenta un efecto colateral con este tipo de algoritmos, pues su sencillez no los hace mejorar el tiempo de ejecución y performance.

```
        if(posicion== -1)
            System.out.println("El entero "+enteroBusqueda+" no se
encontró.\n");
        else
            System.out.println("El entero "+enteroBusqueda+" se encontró en
la posición "+ posicion + ".\n");
            System.out.print("Escriba un valor entero (-1 para terminar):
");
            enteroBusqueda=entrada.nextInt(); //lee el siguiente entero
del usuario
        }
    }
}
```

Podemos concluir que, en este tipo de algoritmos, se invierte un gran esfuerzo para completar la búsqueda. Una de las maneras para describir este esfuerzo es mediante la notación **Big O**, que nos indica el tiempo de ejecución para el peor de los casos; esto está sujeto a la cantidad de elementos que tenga el arreglo.

Búsqueda binaria

El algoritmo de la búsqueda binaria es más eficiente. La búsqueda binaria, al igual que el algoritmo de búsqueda **quicksort**, utiliza la técnica **divide y vencerás**.

Una de las exigencias requeridas antes de ejecutar este tipo de búsquedas es que el conjunto de elementos debe estar **ordenado**. Luego, este algoritmo precisa de una serie de datos de los que se toma para lograr su propósito: el elemento que está al inicio, el del final y el del medio; además del tamaño del arreglo y del elemento por encontrar. La sintaxis de este algoritmo es:

```
int inicio =0;
int medio = 0;
int ultimo = arrayList -1;
medio = (inicio + ultimo)/2;
```

Analicemos cómo funciona el algoritmo de búsqueda binaria de la lista: [1, 4, 6, 9, 11, 13, 17, 19, 21, 23, 24, 25], en el que se pretende buscar el número 18.

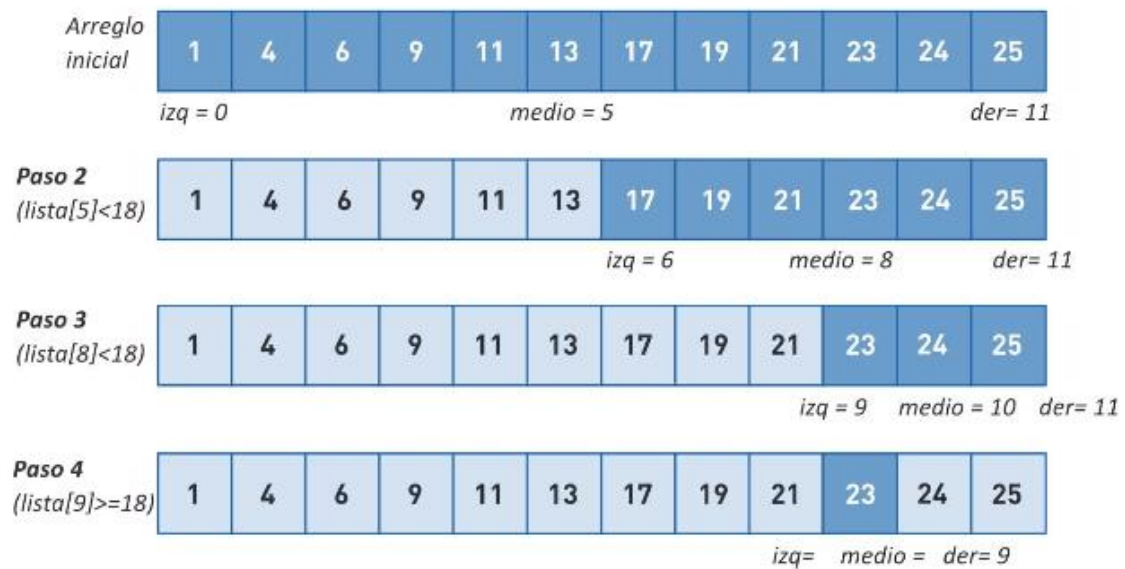


Figura 4. En el gráfico, vemos qué pasa cuando se busca el valor 18.

- Consideramos la lista completa como segmento inicial de búsqueda. Analizamos el punto medio del segmento (el valor central); si es el valor buscado, devolvemos el índice del punto medio.
- Si el valor central es mayor al buscado, podemos descartar el segmento que está desde el punto medio hacia la derecha. En caso de que el valor central sea menor al buscado, descartamos el segmento que está desde el punto medio hacia la izquierda.
- Una vez descartado el segmento que no nos interesa, volvemos a analizar el segmento restante de la misma forma.
- Si en algún momento el segmento por analizar tiene longitud 0 o negativa, significa que el valor buscado no se encuentra en la lista.

Para señalar la porción del segmento que se está analizando a cada paso, utilizamos dos variables (**izq** y **der**) que contienen la posición de inicio y la posición de fin del segmento que se está considerando. De la misma manera, usaremos la variable **medio** para contener la posición del punto medio del segmento. En el ejemplo, verificamos que el número **18** no se encuentra en la lista, por lo tanto, dará **-1**.

Si comparamos con el algoritmo de búsqueda lineal, notaremos que este hubiera requerido varias comparaciones más. Ahora veamos un ejemplo en el que verificamos la implementación de este algoritmo en Java.

Primero lo analizaremos desmenuzando el código de la clase **ArregloBinario**. En la primera parte, se han importado las clase **Random** (como en el ejemplo anterior) y la clase **Arrays**. Se crea un objeto **generador** de la clase **Random**. Luego se crea un arreglo de un tamaño dado, que se llena con enteros aleatorios. Asimismo, se crea espacio para el arreglo con **datos**, que se llenará con enteros aleatorios en el rango **10** a **99**, gracias al **for**. Finalmente, el método **sort** producirá el ordenamiento del array:

```
package estructurasdinamicas.Binarias;
/**@author CharlyRed
 */
import java.util.Random;
import java.util.Arrays;
public class ArregloBinario{
    private int[] datos;
    private static final Random generador = new Random();

    public ArregloBinario(int tamaño){
        datos = new int[tamaño];
        for ( int i=0;i<tamaño;i++)
            datos[i] = 10 + generador.nextInt(90);

        Arrays.sort(datos);
    }
}
```

El método **busquedaBinaria** es el que realiza la búsqueda del valor requerido en el array (previamente ordenado); realiza la ejecución del

algoritmo binario, va por el extremo izquierdo del área de búsqueda, el extremo superior y el medio. Devolverá **-1** si no lo encuentra:

```
public int busquedaBinaria(int elementoBusqueda){
    int inferior=0;
    int superior=datos.length-1;
    int medio=(inferior+superior+1)/2;
    int ubicacion=-1;
```

Con un ciclo **do-while** buscaremos el elemento; luego se imprimen los demás elementos del array.

Un **for** imprimirá espacios para la alineación de los elementos y, con un símbolo de asterisco (*), marcaremos el elemento del medio.

Luego recrearemos un **if-else**, que determinará la eliminación de la mitad izquierda o la derecha de nuestro arreglo.

Después, esta sentencia: **medio = (inferior + superior + 1) / 2** recalculará el nuevo elemento medio de los valores nuevos que resultaron restantes.

Para terminar, **ubicacion** es la clave de búsqueda:

```
do {
    System.out.print(elementosRestantes(inferior,
superior));

    for(int i = 0; i<medio; i++)
        System.out.print(" ");
    System.out.println(" * ");
```



StringBuilder y StringBuffer

Java provee soporte especial para la concatenación de strings con las clases **StringBuilder** y **StringBuffer**. Un objeto **StringBuilder** es una secuencia de caracteres mutable; su contenido y su capacidad pueden cambiar en cualquier momento. Además, a diferencia de los strings, los **builders** cuentan con una capacidad, la cantidad de espacios de caracteres asignados. Esta es siempre mayor o igual que la longitud y se expande automáticamente para acomodarse a más caracteres.

```

        if (elementoBusqueda==datos[medio])
            ubicacion=medio;

        else if (elementoBusqueda<datos[medio])
            superior = medio - 1;
        else
            inferior=medio+1;
        medio=(inferior+superior+1)/2;      }while((inferior<=superior)&&(ubicacion== -1));

        return ubicacion;
    }

```

Ahora le toca el turno al método **elementosRestantes**, que se encarga de imprimir algunos de los valores del array:

```

public String elementosRestantes(int inferior,int superior){
    StringBuilder temporal = new StringBuilder();

    // imprime espacios para alineación
    for(int i=0; i<inferior; i++)
        temporal.append("  ");

    // imprime los elementos que quedan en el arreglo
    for(int i=inferior;i<=superior;i++)
        temporal.append(datos[i] + " ");

    temporal.append("\n");
    return temporal.toString();
}

//método para imprimir los valores en el arreglo
@Override
public String toString(){
    return elementosRestantes(0,datos.length-1);
}
}

```

En cuanto a los códigos de la clase ejecutora, **BusquedaBinaria**, creamos la clave de búsqueda **enteroABuscar** y la ubicación de la clave de búsqueda (**posicion**):

```
package estructurasdinamicas.Binarias;
import java.util.Scanner;
public class BusquedaBinaria{
    public static void main( String[] args ){
        Scanner entrada = new Scanner(System.in);
        int enteroABuscar;
        int posicion;

    }
}
}
```

Creamos el arreglo que se mostrará en pantalla:

```
ArregloBinario arregloBusqueda = new ArregloBinario(15);
System.out.println(arregloBusqueda);
System.out.print("Escriba un valor entero (-1 para salir): ");
enteroABuscar = entrada.nextInt();      System.out.println();
```

Con un **while** recibirá una entrada repetida y usará la búsqueda binaria para tratar de encontrar el entero:

```
while(enteroABuscar != -1{      posicion=arregloBusqueda.
busquedaBinaria(enteroABuscar);
```

Mediante una secuencia **if**, tendremos un valor de retorno (-1), que nos indicará que no se encontró el entero:


```
        if (posicion==-1)
            System.out.println("El entero " + enteroABuscar + " no se
encontró.\n");
        else
            System.out.println("El entero " + enteroABuscar + " se
encontró en la posición " + posicion + ".\n");

        System.out.print("Escriba un valor entero (-1 para salir): ");
        enteroABuscar = entrada.nextInt();
        System.out.println();
    }
```

¿Cómo se mide la **eficiencia** de este algoritmo? Como siempre nos ubicamos en el peor de los casos: el proceso de búsqueda en un arreglo ordenado de más de mil elementos va a requerir solo 10 comparaciones, puesto que dividiremos este número por 2 y así de forma consecutiva, y después de cada comparación eliminamos la mitad que no necesitamos. De allí el nombre de **búsqueda binaria**.

El máximo número de comparaciones necesarias para la búsqueda binaria es el exponente de la primera potencia de 2 mayor que el número de elementos en el arreglo y se representa como $\log_2 n$.

Por eso a este algoritmo también se le conoce como **tiempo de ejecución logarítmico**.

ALGORITMOS DE ORDENAMIENTO

El **ordenamiento de datos** responde a un factor específico de orden, como **ascendente** o **descendente** y corresponde a una de las características más usadas en la computación.

Podemos citar cientos de ejemplos por la que usamos en distintas áreas el ordenamiento: los alumnos de un colegio por orden alfabético; los equipos de fútbol de acuerdo a los puntajes obtenidos (de mayor a menor) en un campeonato; los participantes de una carrera de 100 metros planos por el tiempo logrado, de menor a mayor, etcétera.

Un punto importante es que el resultado final (los datos ordenados) será el mismo, no importa qué algoritmo hayamos utilizado. El tema es que debemos construir un algoritmo que sea capaz de realizarlo en el menor tiempo posible (cosa parecida a los algoritmos de búsqueda) y un uso efectivo de la memoria. Trataremos tres tipos de algoritmos: por selección, por inserción y por combinación.

Los dos primeros se caracterizan por ser simples de programar, pero a la vez ineficientes. El último es más difícil de programar, pero más rápido en entregar los resultados.

Ordenamiento por selección

Este se convierte en un tipo de **ordenamiento simple**, pero como ya hemos dicho, resulta ineficiente.

Funciona de manera **iterativa**: la primera vez selecciona el elemento más pequeño del arreglo y lo intercambia con el primer elemento. Luego, en la segunda, iteración elige el elemento más pequeño de los restantes y lo intercambia con el segundo elemento. Esta tarea continúa de tal forma que, en la última iteración, se selecciona el segundo elemento más grande y lo intercambia con el índice del segundo al último, en consecuencia, deja el elemento más grande en el último índice.

Supongamos que tenemos los siguientes elementos de una lista:

23 44 99 54 12 63 14 8.

Un programa que intente ordenarlos bajo este algoritmo debería determinar el elemento más pequeño (8) que está en el índice 7, entonces el programa intercambia el 8 por el 23 (de índice 0), con lo que quedaría:

8 44 99 54 12 63 14 23

Luego el programa determina cuál es el valor más pequeño, pero esta vez solo compara los valores restantes (no incluye el 8), en este caso tenemos el 12 al que intercambia por el siguiente de la lista de índice 1, es decir el 44; así la lista quedaría de la siguiente forma:

8 12 99 54 44 63 14 23

Si entendemos la forma cómo funciona este algoritmo, esto debería seguir:

```
8 12 14 54 44 63 99 23
8 12 14 23 44 63 99 54
8 12 14 23 44 54 99 63
8 12 14 23 44 54 63 99
```

Y así ya tenemos ordenado nuestro arreglo.

Veamos cómo podemos implementar este tipo de ordenamiento en Java, analicemos la clase **OrdenamientoSeleccion**:

```
package Ordenamiento.porSeleccion;
/**@author CharlyRed
 */
import java.util.Arrays;
import java.util.Random;
public class OrdenamientoSeleccion{
    private int[] datos; // arreglo de valores
    private static final Random generador = new Random();

}
```

Este método creará un arreglo con un tamaño determinado para luego llenarlo con enteros aleatorios:

```
public OrdenamientoSeleccion(int tamaño){
    datos = new int[tamaño];
    for (int i=0; i<tamaño; i++)
        datos[i]=10+generador.nextInt(90);
}
```

El método **ordenar** empieza a hacer este tipo de ordenamiento. Creamos un índice para el elemento más pequeño. Luego, dentro del **for**, iteraremos a través de **datos.length -1** elementos. Después, por medio de otro **for**, itera para buscar el índice del elemento más pequeño:


```

public void ordenar(){
    int masPequeño;
    for(int i=0; i<datos.length-1; i++){
        masPequeño=i; // primer índice del resto del arreglo
        for(int indice=i+1; indice<datos.length; indice++){
            if (datos[indice]<datos[masPequeño])
                masPequeño=indice;

            intercambiar(i,masPequeño); //intercambia el elemento más
pequeño en la posición
            imprimirPasada(i+1,masPequeño); //imprime la pasada del al-
goritmo
        }
    }
}

```

El siguiente método es auxiliar y va a intercambiar los valores de dos elementos. Ahora creamos la variable **temporal** que almacena **primero** en **temporal**. Después sustituimos **primero** con **segundo** y, finalmente, colocamos **temporal** en **segundo**:

```

public void intercambiar(int primero, int segundo){
    int temporal = datos[primero];
    datos[primero] = datos[segundo];
    datos[segundo]=temporal;
}

```

Ahora este método, **imprimirPasada**, va a imprimir una pasada. El **for** nos servirá para imprimir elementos hasta el elemento seleccionado. El signo asterisco (*) va a marcar el intercambio.

El siguiente **for** termina de imprimir el arreglo en la pantalla, y el último **for** nos indicará la cantidad del arreglo que está almacenada:

```

public void imprimirPasada(int pasada, int indice){
    System.out.print(String.format("después de pasada %2d: ",
pasada));

    for(int i=0; i<indice; i++)
        System.out.print(datos[i] + " ");
}

```

```

        System.out.print(datos[indice] + "* ");

        for (int i=indice+1; i<datos.length; i++)
            System.out.print(datos[ i ] + " ");
        System.out.print("\n");

        for(int j = 0; j < pasada; j++)
            System.out.print("-- ");
        System.out.println( "\n" );    }

```

Ahora veremos los códigos de la clase **TestOrdenamientoSeleccion**.

Primero creamos un objeto que realizará el ordenamiento y respeta la siguiente lógica (verificar los comentarios):

```

public class PruebaOrdenamientoSeleccion{
    public static void main( String[] args ){
        OrdenamientoSeleccion arregloOrden = new OrdenamientoSeleccion(10);

        System.out.println("Arreglo desordenado:");
        System.out.println(arregloOrden + "\n"); //imprime arreglo desordenado

        arregloOrden.ordenar(); //ordena el arreglo
        System.out.println("Arreglo ordenado:");
        System.out.println(arregloOrden); //imprime el arreglo ordenado
    }
}

```

Para terminar, este último método nos imprimirá los valores del arreglo:

```

@Override
    public String toString(){
        return Arrays.toString(datos);
    }

```

En cuanto a la eficiencia del ordenamiento por selección, este se ejecuta en un tiempo igual a $O(n^2)$.

Ordenamiento por inserción

El **ordenamiento por inserción** es otro algoritmo de ordenamiento simple, pero ineficiente. En la primera iteración de este algoritmo, se toma el segundo elemento en el arreglo y, si es menor que el primero, se intercambian. En la segunda iteración, se analiza el tercer elemento y se inserta en la posición correcta con respecto a los primeros dos elementos, de manera que los tres estén ordenados. En la i -ésima iteración de este algoritmo, los primeros i elementos en el arreglo original estarán ordenados. Para que veamos la eficiencia de este tipo de ordenamiento, utilizaremos la misma serie que usamos para el ejemplo anterior:

23 44 99 54 12 63 14 8

Un programa que implemente el algoritmo de ordenamiento por inserción analizará al principio los primeros dos elementos del arreglo, **23** y **44**. Como estos ya se encuentran ordenados, el programa continúa; en caso contrario, el programa los intercambiaría.

En la siguiente iteración, el programa analiza el tercer valor, **99**. Este valor es mayor que **44**, por lo que el programa continúa.

El siguiente número es **54**; este, al ser menor que **99**, se intercambia y se inserta antes que él. Luego compara con el **44**, pero, al ser mayor que él, ya no sigue con el intercambio. Tenemos:

23 44 54 99 12 63 14 8

Ahora viene el número **12**, al que comparamos con el **99**. Al ser menor, se intercambia con el **99** y sigue comparando; al ser menor que el **54**, se coloca a su izquierda; luego en **44**, sucede lo mismo y, finalmente, se compara con el **23** y también se intercambia. Esto quedaría:

12 23 44 54 99 63 14 8

Este mismo comportamiento lo vamos a continuar hasta terminar con el ordenamiento.

El algoritmo de ordenamiento por inserción también se ejecuta en un tiempo igual a $O(n^2)$, al igual que el ordenamiento por selección.

Ordenamiento por combinación

El **ordenamiento por combinación** es un algoritmo de ordenamiento eficiente, pero resulta más complejo que sus predecesores. La forma en que funciona este algoritmo lo va dividir en **dos subarreglos** de igual tamaño, ordena cada subarreglo y después los combina en un arreglo más grande. Con un número impar de elementos, el algoritmo crea los dos subarreglos de tal forma que uno tenga más elementos que el otro.

La implementación del ordenamiento por combinación en este ejemplo es recursiva. El caso base es un arreglo con un elemento que, desde luego, está ordenado, por lo que el ordenamiento por combinación regresa de inmediato en este caso. El paso recursivo divide el arreglo en dos piezas de un tamaño aproximadamente igual, las ordena en forma recursiva y, después, combina los dos arreglos ordenados en un arreglo ordenado de mayor tamaño.

Supongamos que tenemos el algoritmo que ya ha combinado arreglos más pequeños para crear los arreglos ordenados

A: **23 44 54 99** y B: **8 12 14 63**

El ordenamiento por combinación combina estos dos arreglos en un arreglo ordenado de mayor tamaño.

El elemento más pequeño en A es **23** (que se encuentra en el índice cero de A). El elemento más pequeño en B es **8** (que se encuentra en el índice cero de B). Para poder determinar el elemento más pequeño en el arreglo más grande, el algoritmo compara **4** y **5**. El valor de B es más pequeño, por lo que el **8** se convierte en el primer elemento del arreglo combinado. El algoritmo continúa y compara **44** (el segundo elemento en A) con **8** (el primer elemento en B). El valor de B es más pequeño, por lo que **8** se convierte en el segundo elemento del arreglo más grande. El algoritmo continúa comparando **44** con **12**, en donde **12** se convierte en el tercer elemento del arreglo, y así sucesivamente.

Ahora implementaremos un código para Java así veremos la forma de trabajar este algoritmo. Siguiendo la misma forma de trabajar los algoritmos anteriores crearemos dos clases, una de ellas creará un array aleatorio de **10** a **99**. Será la clase **OrdenamientoCombinacion**, cuyo código es:


```
package ordenamientocombinacion;
import java.util.Random;
public class OrdenamientoCombinacion{
    private int[] datos; // arreglo de valores
    private static Random generador = new Random();
}
```

Este método crea un arreglo de un tamaño dado y lo llena con enteros aleatorios:

```
public OrdenamientoCombinacion(int tamaño){
    datos=new int[tamaño];
    for(int i=0; i<tamaño; i++)
        datos[i]=10 + generador.nextInt(90);
}
```

Este método llama al método de **división recursiva** para comenzar el ordenamiento por combinación, de tal forma que va a dividir todo el arreglo:

```
public void ordenar(){
    ordenarArreglo(0, datos.length - 1);
}
```

Este método divide el arreglo, ordena los subarreglos y los combina en un arreglo ordenado. Luego evaluará el caso base si el tamaño del arreglo es igual a 1 e imprime el paso de la división.

En **ordenarArreglo** divide el arreglo a la mitad, luego ordena cada mitad a través de llamadas recursivas:

```
private void ordenarArreglo(int inferior, int superior){
    if((superior - inferior)>=1){ // si no es el caso base
        int medio1=(inferior + superior)/2; //calcula el elemento me-
dio del arreglo
        int medio2 = medio1 + 1; //calcula el siguiente elemento ar-
riba

        System.out.println( "division:  " + subarreglo(inferior,
```

```

superior));
        System.out.println( "          " + subarreglo(inferior,
mediol));
        System.out.println( "          " + subarreglo(medio2, su-
perior));
        System.out.println();

        ordenarArreglo(inferior, mediol); // primera mitad del arreglo
        ordenarArreglo(medio2, superior); // segunda mitad del arreglo

        combinar(inferior,mediol,medio2,superior);
    }
}

```

Este método combina dos subarreglos ordenados en un subarreglo ordenado. Luego se imprimen en pantalla los dos subarreglos antes de combinarlos. Por medio de un **while** se combinan los arreglos hasta llegar al final de uno de ellos:

```

    private void combinar(int izquierdo, int mediol, int medio2, int
derecho) {
        int indiceIzq = izquierdo; //índice en sub arreglo izquierdo
        int indiceDer = medio2; //índice en sub arreglo derecho
        int indiceCombinado = izquierdo; //índice en arreglo de tra-
bajo temporal
        int[] combinado = new int[datos.length]; // arreglo de trabajo

        System.out.println("combinacion:" + subarreglo(izquierdo, me-
diol));
        System.out.println("          " + subarreglo(medio2,
derecho));

        while(indiceIzq<=mediol&&indiceDer<=derecho){
            if(datos[indiceIzq]<=datos[indiceDer])                combinado[
indiceCombinado++]=datos[indiceIzq++];
            else                combinado[indiceCombinado++]=datos[indiceD
er++];
        }
    }

```

```

        // si el arreglo izquierdo está vacío
        if (indiceIzq==medio2)
            // copia el resto del arreglo derecho
            while(indiceDer <= derecho)        combinado[indiceComb
inado++]=datos[indiceDer++];
        else // el arreglo derecho está vacío
            // copia el resto del arreglo izquierdo
            while(indiceIzq<=medio1)        combinado[indiceCombina
do++]=datos[indiceIzq++];

        for(int i=izquierdo; i<=derecho; i++)
            datos[i] = combinado[i];

        // imprime en pantalla el arreglo combinado
        System.out.println("          " + subarreglo(izquierdo,
derecho));
        System.out.println();
    }

```

El **for** copia los valores de vuelta al arreglo original.

El siguiente método imprime en pantalla ciertos valores en el arreglo:

```

public String subarreglo(int inferior, int superior){
    StringBuilder temporal = new StringBuilder();

    // imprime en pantalla espacios para la alineación
    for (int i=0; i<inferior; i++)
        temporal.append("  ");

    // imprime en pantalla el resto de los elementos en el arre-
glo
    for(int i=inferior; i<=superior; i++)
        temporal.append(" " + datos[i]);
    return temporal.toString();
}

```

Para terminar, este método imprime los valores en el arreglo:

```
public String toString(){
    return subarreglo(0, datos.length - 1);
}
```

Y, ahora, el código de la clase que ejecutará el ordenamiento:

```
package ordenamientocombinacion;
public class PruebaOrdenamientoCombinacion{
    public static void main(String[] args){

        OrdenamientoCombinacion arregloOrden=new Ordenamien-
toCombinacion(10);

        // imprime el arreglo desordenado
        System.out.println("Desordenado:" + arregloOrden +
"\n");

        arregloOrden.ordenar(); //ordena el arreglo

        //imprime el arreglo ordenado
        System.out.println("Ordenado:   "+arregloOrden);
    }
}
```

Consideremos la primera llamada (no recursiva) al método **ordenarArreglo**. Esto produce dos llamadas recursivas al método **ordenarArreglo** con subarreglos; cada uno de ellos tiene un tamaño aproximado de la mitad del arreglo original y una sola llamada al método combinar. Esta llamada a combinar requiere **n-1** comparaciones para llenar el arreglo original, que es **O(n)**. Recordemos que se puede elegir cada elemento en el arreglo mediante la comparación de un elemento de cada uno de los subarreglos. Las dos llamadas al método **ordenarArreglo** producen cuatro llamadas recursivas más al método **ordenarArreglo**, cada una con un subarreglo de un tamaño aproximado a una cuarta parte del tamaño del arreglo original, junto con dos llamadas al método combinar. Estas dos llamadas al método combinar requieren, **n/2 -1** comparaciones para un número total de **O(n)** comparaciones.

Este proceso continúa, y cada llamada a **ordenarArreglo** genera dos llamadas adicionales a **ordenarArreglo** y una llamada a combinar, hasta que el algoritmo divide el arreglo en subarreglos de un elemento. En cada nivel, se requieren **$O(n)$** comparaciones para combinar los subarreglos. Cada nivel divide el tamaño de los arreglos a la mitad, por lo que al duplicar el tamaño del arreglo se requiere un nivel más. Si se cuadruplica el tamaño del arreglo, se requieren dos niveles más. Este patrón es logarítmico, y produce **$\log_2 n$** niveles. Esto resulta en una eficiencia total de **$O(n \log n)$** .

ÁRBOLES

En una estructura de datos, los **árboles binarios** o tree set son elementos que apuntan a un lado izquierdo o derecho, puesto que no pueden apuntar a más de dos nodos. Su objetivo principal es efectuar grandes búsquedas, por la facilidad del recorrido, ya que lo realiza por **nodo** en vez de hacerlo de forma lineal como vimos antes.

Existen varias técnicas para hacer el recorrido a través de un árbol, entre ellas la de **pre orden**, **post orden** e **in orden**.

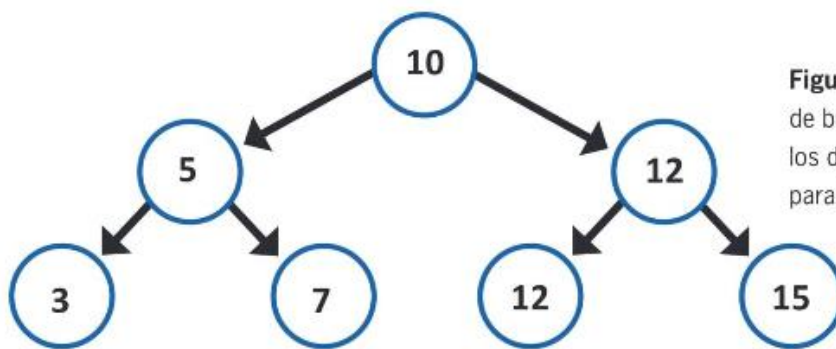


Figura 5. Este sistema de búsqueda mostrará los distintos métodos para hacer recorridos.

- **Recorrido pre orden:** este recorre el nodo raíz, luego pasa hacia el nodo izquierdo y, al final, el nodo derecho. Según el esquema de la **Figura 5** el recorrido debería ser:

10 - 5 - 3 - 7 - 12 - 11 - 15

- **Recorrido post orden:** Hace el recorrido en el lado izquierdo, luego se pasa al lado derecho y por último el nodo raíz. Según el esquema de la **Figura 5** el recorrido debería ser:

3 - 7 - 5 - 11 - 15 - 12 - 10

- **Recorrido in orden:** este recorrido empieza en el nodo izquierdo, continúa en el nodo raíz y termina en el nodo de la derecha. Siguiendo el esquema sería:

3 - 5 - 7 - 10 - 11 - 12 - 15

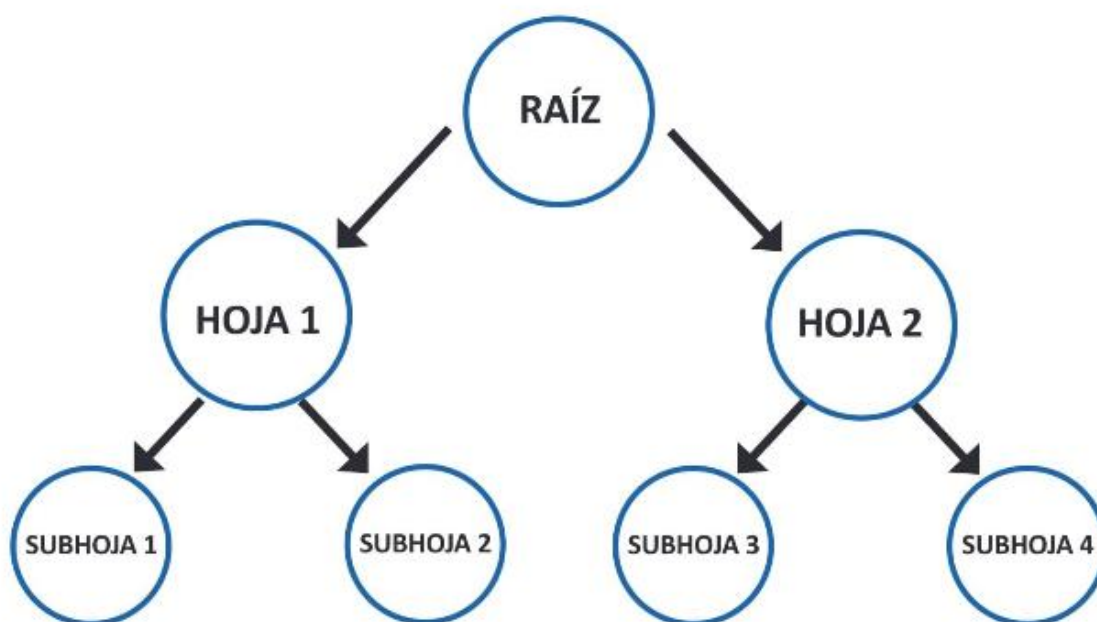


Figura 6. Gráfico del funcionamiento de los árboles. Podemos ver cómo raíz es padre de hoja 1 y hoja 2, y estas a su vez se convierten en las raíces de las subhojas.

Veamos un ejercicio que muestra cómo funcionan los tres recorridos de los árboles. Tendremos dos clases: una llamada **ArbolBinario**, que tendrá todos los recorridos, y la clase **Principal**, que llamará a estas, más las impresiones.

Analizaremos, en primer lugar, la clase **ArbolBinario**, esta tendrá **Nodo** como **raiz**, y un constructor en donde **raiz** tendrá por defecto un **null**.

```
package arboles;
public class ArbolBinario{
    Nodo raiz;
    public ArbolBinario(){
        raiz=null;
    }
}
```

En el siguiente método **insertar** tendremos un objeto de la clase **Nodo** que será, precisamente, el encargado de ir insertando nuevas hojas y subhojas. Esto lo lograremos a través de un **if – else**:

```
public void insertar(int info){
    Nodo nuevo;
    nuevo = new Nodo();
    nuevo.info = info;
    nuevo.izq = null;
    nuevo.der = null;
    if (raiz == null)
        raiz = nuevo;
    else {
        Nodo anterior = null, reco;
        reco = raiz;
        while (reco != null){
            anterior = reco;
            if (info < reco.info)
                reco = reco.izq;
            else
                reco = reco.der;
        }
        if (info < anterior.info)
            anterior.izq = nuevo;
        else
            anterior.der = nuevo;
    }
}
```

Los siguientes métodos harán la impresión de los distintos recorridos:

```
private void imprimirPre(Nodo reco){
    if (reco != null){
        System.out.print(reco.info + " ");
        imprimirPre (reco.izq);
        imprimirPre (reco.der);
    }
}

public void imprimirPre(){
    imprimirPre (raiz);
    System.out.println();
}

private void imprimirIn(Nodo reco){
    if (reco != null){
        imprimirIn(reco.izq);
        System.out.print(reco.info + " ");
        imprimirIn(reco.der);
    }
}

public void imprimirIn(){
    imprimirIn(raiz);
    System.out.println();
}

private void imprimirPost(Nodo reco){
    if (reco != null){
        imprimirPost(reco.izq);
        imprimirPost(reco.der);
        System.out.print(reco.info + " ");
    }
}

public void imprimirPost (){
    imprimirPost(raiz);
    System.out.println();
}
```

Finalmente, tenemos la clase **Nodo**, que se encargará de apuntar a dos lugares, izquierdo y derecho:

```
class Nodo{
    int info;
    Nodo izq, der;
}
```

Luego, la clase **Principal** que se encargará de crear el objeto de clase e imprimir la información:

```
package arboles;
public class Principal {
    public static void main(String[] args) {
        ArbolBinario ab = new ArbolBinario();
        ab.insertar(7);
        ab.insertar(2);
        ab.insertar(3);
        ab.insertar(1);
        ab.insertar(8);
        ab.insertar(6);
        ab.insertar(10);
        ab.insertar(9);
        ab.insertar(4);
        ab.insertar(5);

        System.out.println("Impresión pre orden");
        ab.imprimirPre();
        System.out.println("Impresión in orden");
        ab.imprimirIn();
        System.out.println("Impresión post orden");
        ab.imprimirPost();
    }

    public static void imprimir(String numero){
        System.out.println(numero);
    }
}
```

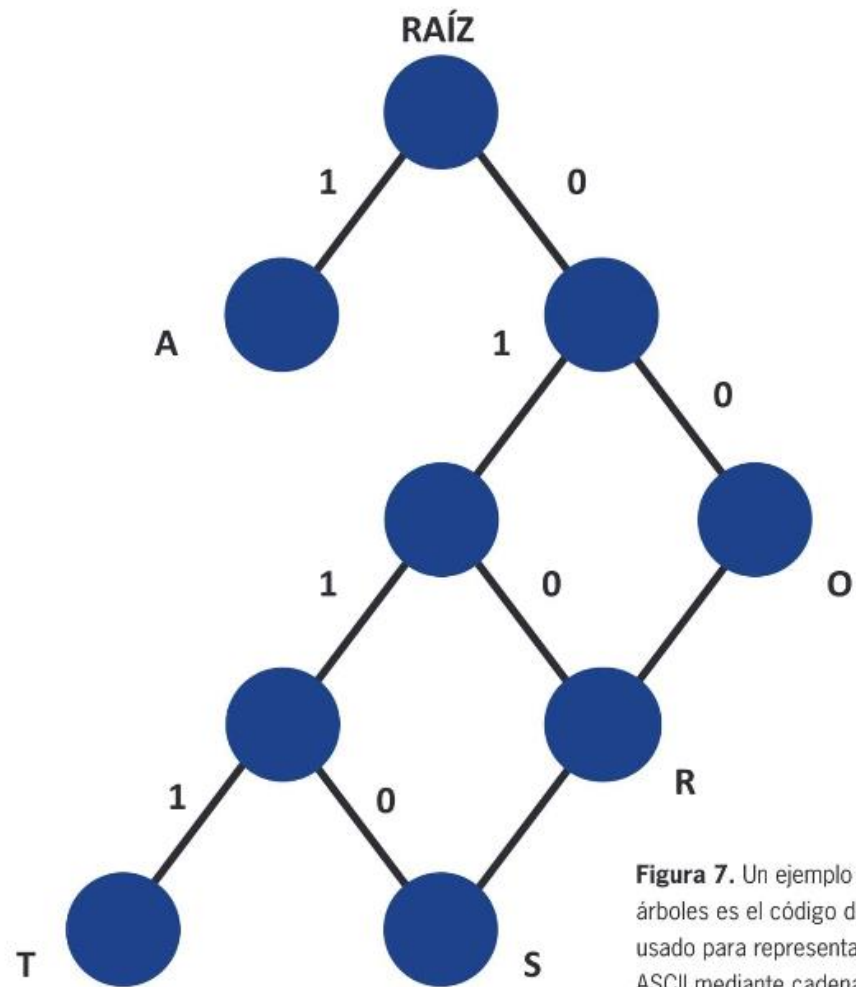



Figura 7. Un ejemplo de uso de árboles es el código de Huffman usado para representar los códigos ASCII mediante cadenas de bits.



RESUMEN CAPÍTULO 01

En este capítulo profundizamos los ordenamientos y las búsquedas de una manera más compleja, a través de algoritmos sofisticados. Vimos estructuras dinámicas, como las pilas, colas y listas enlazadas. Luego revisamos diferentes algoritmos de búsquedas, tanto lineales como binarias, pasando por los ordenamientos simples de codificar, y aquellos más complejos. Finalmente, revisamos un tipo de búsquedas más avanzadas, los árboles, los que son utilizados en la actualidad.

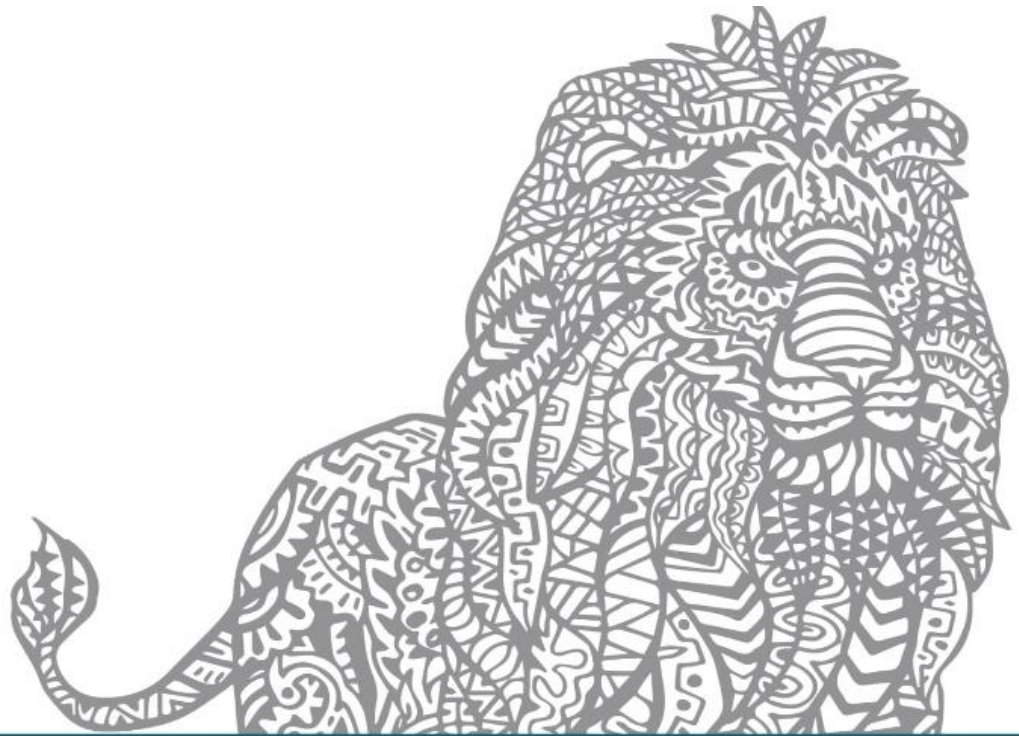
Actividades 01

Test de Autoevaluación

1. ¿Qué diferencia existe entre las estructuras estáticas y las dinámicas?
2. ¿Que son las listas enlazadas?
3. ¿Para qué sirven las pilas?
4. ¿Que son las colas?
5. ¿Qué algoritmos de búsqueda conoce? ¿Qué diferencia existe entre ellos?
6. ¿Qué tipo de algoritmos de ordenamiento de datos conoce?
7. ¿Por qué es tan efectivo el algoritmo de combinación?
8. ¿Qué son los árboles?
9. ¿Cuál es la función de los árboles?
10. ¿Qué tipo de recorridos presentan los árboles?

Ejercicios prácticos

1. Utilice un grupo de alumnos para enlistar sus notas y promedios.
2. Implemente una pila que soporte el ejercicio anterior.
3. Ordene a los alumnos alfabéticamente con alguno de los métodos de ordenamiento estudiados, luego ordénelos por el promedio (del mayor al menor).
4. Ingrese una cierta cantidad de elementos y, por medio de alguno de los recorridos ya estudiados, utilice el algoritmo de búsqueda de árboles.



Acceso a bases de datos (JDBC)

Trabajar con bases de datos es un tema importante a la hora de programar. Esto se relaciona con la persistencia de datos y la posibilidad de trasladar esa información a una plataforma que la administre para que podamos acceder a ella cuando lo necesitemos. En este capítulo, revisaremos los aspectos fundamentales del tratamiento de las BBDD en Java, a lo que llamamos JDBC o *Java Data Base Connectivity*.



02

BASES DE DATOS

Antes de establecer una conexión, debemos entender qué es una **base de datos del tipo relacional**, que es la que manejaremos en este capítulo. Se trata de un conjunto de datos persistentes que están ordenados de acuerdo a un estamento de **tablas**, entre otros elementos; estas a su vez contienen una disposición de **filas** y **columnas**. Las primeras incluyen los **registros** de la base, y las segundas constituyen los **campos** (o encabezamientos), por ejemplo, **IDEmpleado**, **Nombre**, **Apellido**, **DNI**, **fechaIngreso**, etcétera.

JDBC es un driver que se conecta desde nuestra aplicación hasta un gestor de una BBDD, como **mySQL**, **Oracle**, **SQL Server**, **Paradox**, etcétera. Una vez que se conecta, podremos manipular los datos que la base contenga.

Una de las características del driver de JDBC es que lo proporciona el sistema gestor de la BBDD y, por lo tanto, es distinto en cada gestor.

Para poder establecer la conexión a la BBDD, necesitaremos dos paquetes: **Java.sql** y **Javax.sql**. Estos poseen una serie de clases e interfaces con sus correspondientes métodos, lo que nos permitirá realizar la conexión de la información requerida y manipularla.

Los pasos por seguir desde que conectamos a una BBDD hasta que conseguimos acceder o modificar los datos son los siguientes:

Establecer la conexión con la BBDD.
Instanciar un objeto del tipo **Statement**.
Ejecutar una sentencia SQL.
Esta sentencia devuelve un objeto del tipo **ResultSet** del que debemos leer para poder acceder a la información de la BBDD.

Para hacer todo esto, es necesario conocer el lenguaje básico del **lenguaje SQL**; sin embargo, para los efectos deseados daremos en este capítulo los pasos necesarios para poder entenderlo y trabajarlo. Además, conoceremos algunas de las sentencias elementales de ese lenguaje, entre las que se destacan **insert**, **update**, **delete** o **select**, que cumplen con las cuatro acciones básicas de las BBDD.

CONEXIÓN A UNA BBDD

Ahora que sabemos qué es una BBDD, debemos entender cómo se establece la conexión hacia ellas. A continuación, veremos de qué manera podemos realizar cada uno de estos procesos.

Establecer la conexión a la base de datos

Esto se realiza mediante el uso de las clases correspondientes y la instanciación de un objeto del tipo conexión. A la hora de crear este objeto, vamos a utilizar uno del tipo **String**, que almacena la cadena de conexión propiamente dicha. Esto dependerá de la BBDD a la que vamos a conectarnos, en este caso utilizaremos una del tipo Open Source: **mysql**.

Primero verifiquemos qué elementos se encuentran involucrados cuando hay una conexión:

```
jdbc:mysql://localhost:9999/accesoPedidos  
driver || protocolo || detalles de la conexión del driver
```

En los detalles de la conexión, hemos usado una **conexión local** que es la que utilizaremos para hacer nuestras pruebas, sin embargo, en el uso real, dicha conexión se efectúa a través de una IP o la URL del servidor en donde esté alojada la BBDD.

Además de este tipo de información, para realizar la conexión a la base de datos precisaremos dos elementos muy importantes y necesarios: el **usuario** y la **contraseña**.



Bases relacionales

Una base de datos relacional permite establecer interconexiones (relaciones) entre los datos (que están guardados en tablas), y a través de dichas conexiones, relacionar los datos de ambas tablas. Tras ser postuladas sus bases en 1970 por **Edgar Frank Codd**, de los laboratorios IBM en San José (California), este tipo de bases de datos no tardaron en consolidarse como un nuevo paradigma.

Debemos tener en cuenta que las conexiones pueden lanzar excepciones por diversos motivos. Por esta razón el proceso de conexión debería estar dentro de un bloque **try – catch**, para así capturar la posible excepción que pudiera presentarse.

Crear el objeto Statement

Este es el segundo paso, crear un objeto del tipo **Statement**. Con el objeto conexión creado, vamos a utilizar un método llamado **createStatement**, que, aplicado sobre el objeto mencionado, nos devolverá un objeto del tipo **Statement**.

Ejecutar una sentencia SQL

Una vez creado el objeto **Statement**, estamos habilitados para utilizar y ejecutar las sentencias SQL, esto lo haremos a través de un método llamado **executeQuery**.

Este método a su vez nos devuelve un **resultSet**, un objeto donde se almacena la información que nos devuelve la sentencia SQL, es decir, allí se almacena qué vamos a consultar a la BBDD y acceder a cualquiera de sus tablas.

INSTALACIÓN DE UN SERVIDOR DE BBDD

Antes de ejecutar sentencias y de utilizar nuestras BBDD, necesitamos instalar un servidor para el gestor de la BBDD, en este caso lo haremos a través de MySQL. Primero debemos instalar el **driver jdbc** para BBDD en MySQL; una vez logrado este paso, agregaremos al **classpath** del proyecto.



PASO A PASO: INSTALAR UN SERVIDOR DE MySQL

01 Inicie una sesión de **Administrador** en el sistema operativo como Administrador. Luego descargue **MySQL Server Community Edition**. La dirección es la siguiente: <https://dev.mysql.com/downloads/mysql/5.5.html>.



02 Antes de descargar MySQL debe verificar en su computadora qué tipo de sistema operativo tiene instalado, es decir, si es de 32 o 64 bits.





PASO A PASO: INSTALAR UN SERVIDOR DE MySQL

- 01** Inicie una sesión de **Administrador** en el sistema operativo como Administrador. Luego descargue **MySQL Server Community Edition**. La dirección es la siguiente: <https://dev.mysql.com/downloads/mysql/5.5.html>.



- 02** Antes de descargar MySQL debe verificar en su computadora qué tipo de sistema operativo tiene instalado, es decir, si es de 32 o 64 bits.



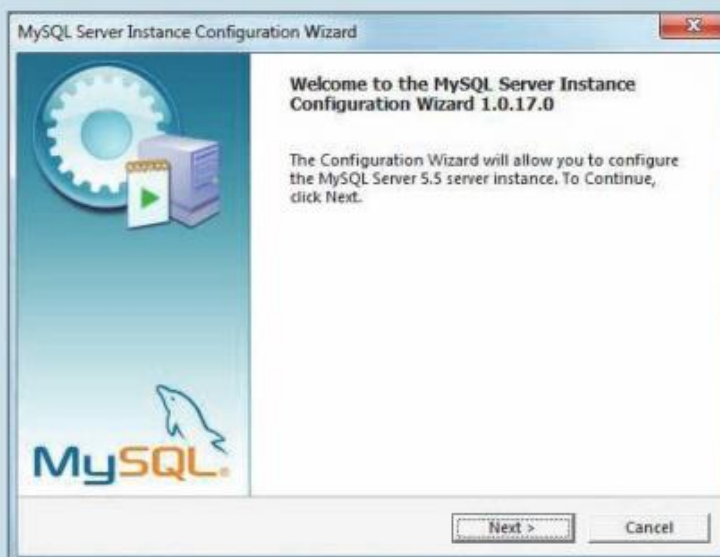
- 03** Una vez descargado el archivo **mysql-5.5.57-winx64.msi**, proceda a instalar el software haciendo doble clic en el archivo. Recuerde que lo debe hacer como **Administrador**.



- 04** Acepte los términos de la licencia y presione **Siguiente**. Elija la opción **Complete** y presione sobre **Install**.



- 05** Una vez terminada la instalación, haga clic sobre **Finish**. Ahora es el momento de configurar el servidor MySQL; aparecerá un asistente que lo guiará en el proceso, presione **Next**.



- 06** En la siguiente pantalla, podrá elegir si quiere configurar en forma detallada o estándar. Elija la primera opción; el asistente empezará a configurar el servidor.



07 En la siguiente pantalla podrá elegir el tipo de instalación que desea. En este caso, indique que se utilizará una máquina de desarrollo: **Developer Machine**.



08 Ahora preguntará por el tipo de base de datos con la que trabajará; para el propósito de nuestros ejemplos, la primera opción estará bien.



09 En la siguiente pantalla, podrá seleccionar en qué partición de la PC instalará los archivos del programa. Puede dejar la opción indicada en forma predeterminada.



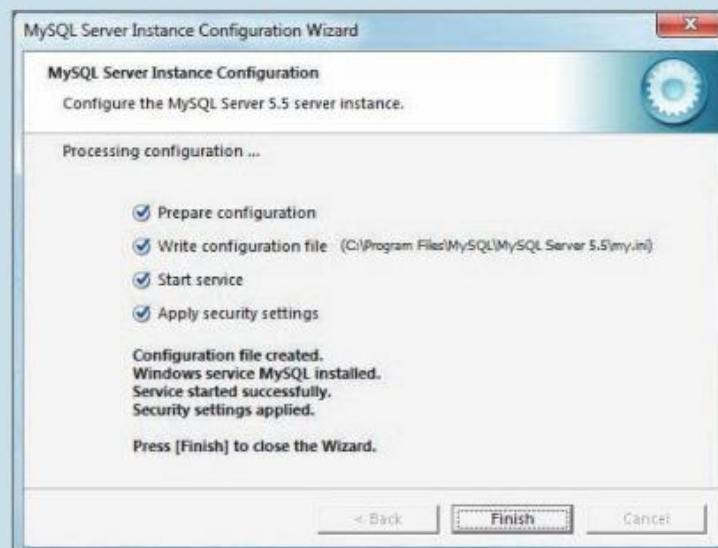
10 Ahora debe elegir la cantidad de conexiones máximas que podrá tener, en esta ocasión optaremos por **Manual Settings**. Luego, coloque el puerto por el que se conectará, elija el **3306**.



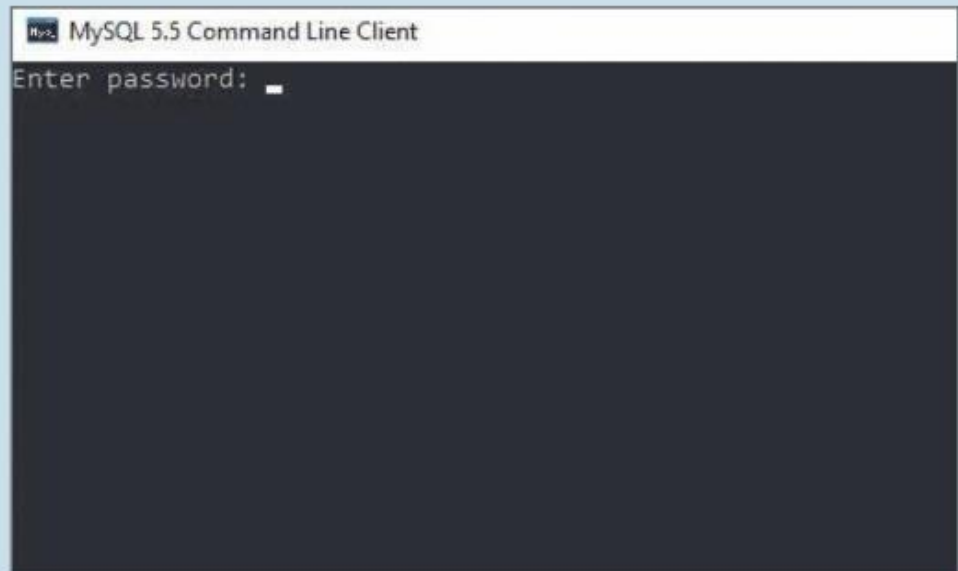
- 11** La siguiente pantalla es para los caracteres, elija la opción **Standard**. Luego deberá indicar la tecnología de servidor con la que trabajará, en este caso elija **MySQL**; posteriormente indique una contraseña para acceder a la BBDD.



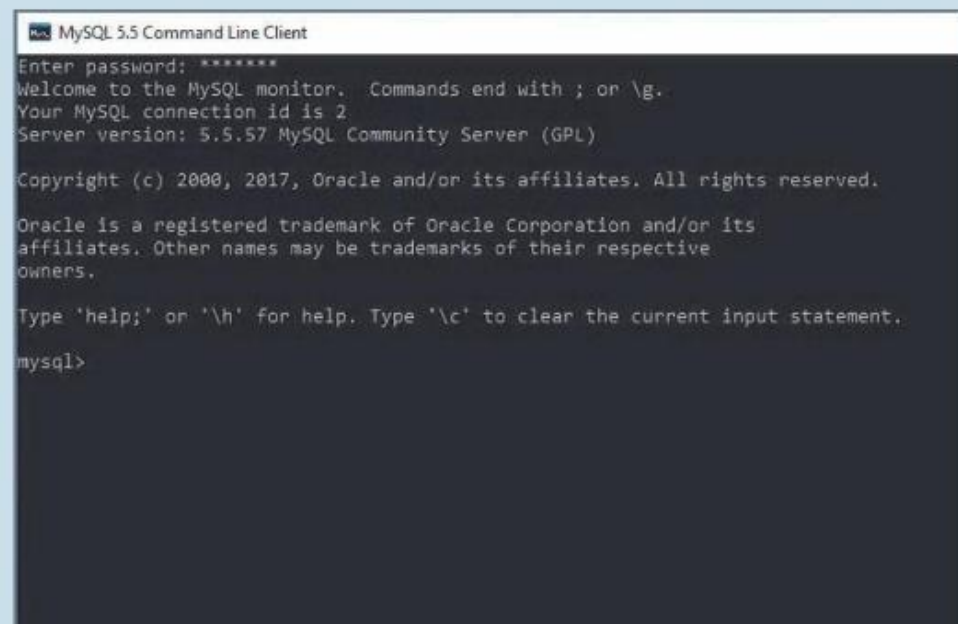
- 12** Ha llegado al final de la configuración, presione **Execute**. Una vez terminada la configuración, podrá ver un resumen de lo realizado. Haga clic en **Finish**.



- 13** Para probar si todo resultó bien, puede lanzar el servidor desde el menú **Inicio** del sistema operativo, para eso, ingrese el password que indicó durante la instalación.



- 14** Una vez que acceda al servidor, verá este mensaje de bienvenida directamente en la consola de comandos.



Otra forma de contar con una base de datos es instalar paquetes como **WAMP** o **XAMPP Server**. Estas opciones, además del servidor de bases de datos, nos ofrecen una gran cantidad de posibilidades adicionales, como PHP y Apache.

Para continuar, tendremos que descargar e instalar el driver MySQL, tal como vemos en el siguiente **Paso a paso**:

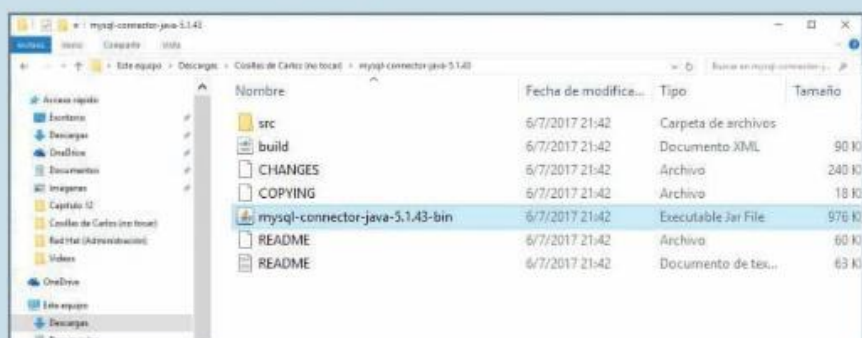


PASO A PASO: DESCARGAR EL DRIVER JDBC PARA MySQL

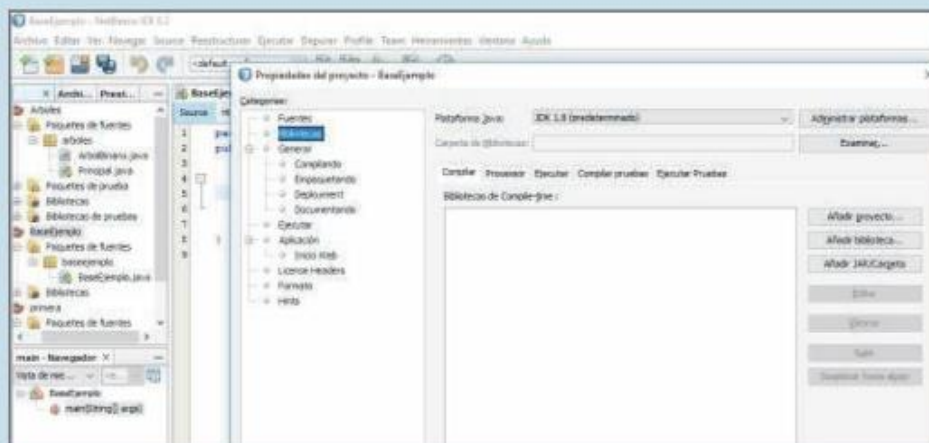
01 En el buscador que utiliza habitualmente escriba **driver jdbc para MySQL**. Enlace en el siguiente sitio: <https://dev.mysql.com/downloads/connector/j/5.1.html>. Debe descargarlo en formato .zip.



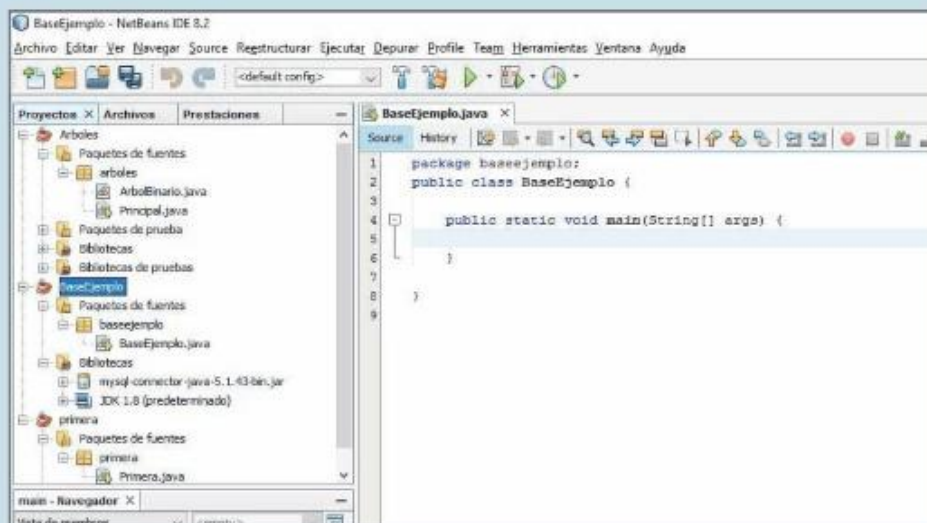
02 Una vez descargado, busque el archivo .jar: **mysql-connector-java-5.1.43-bin**.



- 03** Ubique el contenido descomprimido en alguna carpeta hasta que lo necesite. Cree un proyecto nuevo en el que involucre una BBDD, agregue al classpath de los archivos y del driver descargado. Para esto, haga clic con el botón secundario en el proyecto y seleccione **Propiedades/Biblioteca**. Haga clic en el botón **Añadir JAR/Carpeta**.



- 04** Busque el archivo.jar que guardó antes y presione **Aceptar**. Con esto tendrá agregado, al proyecto, las librerías correspondientes al driver jdbc. Podrá ver que, en el panel de proyectos, estará el driver del conector MySQL.



Ya deberíamos tener instalado el servidor y el driver JDBC; con esto estamos listos para trabajar con la BBDD.

CONEXIÓN A MYSQL

Para conectarnos a una BBDD, es necesario conocer cómo se crea una base de datos, en nuestro caso en MySQL. Sin embargo, por el alcance de este libro, no desarrollaremos esta parte, ya que tenemos una BBDD preparada. Nosotros aprenderemos cómo acceder a ella y realizar las acciones pertinentes.

Crearemos dentro del proyecto que hicimos en la sección anterior un package al que llamaremos **conectaBBDD**. Dentro de este paquete, crearemos una clase a la que llamaremos **ConectaBasePruebas**. La BBDD se llama **BasePruebas**.

Recordemos que nos va a mostrar una excepción, por lo tanto, se requiere la creación de un bloque try-catch:

```
package conectaibdd;  
/**  
 * @author CharlyRed  
 */  
public class ConectaBasePruebas {  
    public static void main(String[] args) {  
        try{  
  
        }catch(Exception e){  
  
        }  
    }  
}
```

Luego necesitaremos colocar un objeto del tipo **Connection** como vimos en apartados anteriores. Y desde luego, la clase **DriverManager**, que tiene un método llamado **getConnection** el cual nos pedirá como parámetros la dirección url, el usuario y un password.

Debemos saber que todas las BBDD de MySQL tienen un usuario y una contraseña, que, si no hemos producido ningún cambio, deberían ser: para el usuario, **root** y para la contraseña, vacía ("").

Como dicho método nos devuelve un objeto del tipo **Connection**, para poder utilizarlo debemos crear una variable del mismo tipo (**Connection**) y, como pertenece a un paquete del tipo **sql**, antes debemos importarlo:

```
import java.sql.*;
```

Ahora el código continuaría de la siguiente manera (dentro del try):

```
//1. crear la conexión
        Connection miConexion = DriverManager.
getConnection("jdbc:mysql://localhost:3306/BasePrueba","root","");
```

El siguiente paso consiste en crear un objeto del tipo **Statement**, que posteriormente nos permitirá ejecutar una consulta SQL. Si accedemos a la API, podremos verificar que dentro de la interfaz **Connection** tiene un método **createStatement**. Este sirve para enviar consultas SQL a la BBDD, por lo tanto, devolverá un objeto del tipo **Statement**. Lo haremos de la siguiente manera:

```
//2. Crear el objeto Statement
        Statement miStatement = miConexion.createStatement();
```

Creamos un objeto al que llamamos **miStatement**. Luego lo igualamos al objeto creado en el paso anterior (**miConexion**) e invocamos al método **createStatement**.

Como tercer paso debemos ejecutar la instrucción SQL, y para ello necesitamos el objeto del tipo **Statement**. Debemos ir a la API para ver si existe un método que haga este tipo de operaciones, y encontramos que está el **executeQuery**, que nos permitirá ejecutar sentencias SQL y devolverá un objeto del tipo **ResultSet**. Y es lo que exactamente haremos:

```
//3. Ejecutar SQL
        ResultSet miResultSet = miStatement.
executeQuery("SELECT * FROM PRODUCTOS");
```

Dentro de los parámetros hemos colocado sentencias exclusivamente del lenguaje SQL (al que no tiene alcance el presente libro), y significa que de la tabla **Productos** vamos a hacer una consulta.

Dentro del objeto **ResultSet** tendremos lo que nos devuelva esa consulta. Este objeto tiene forma de una tabla virtual. En otros lenguajes es denominado el **RecordSet**.

Finalmente, debemos movernos dentro de esta tabla virtual y lo hacemos a través de alguno de los métodos existentes de la interfaz **ResultSet**. Cuando hay que hacer movimientos a través de la tabla, debería haber un cursor que se traslade por ella y realice un recorrido, por ejemplo, ir al comienzo, al final, al siguiente, etcétera. Vayamos al código y verifiquemos:

```
//4. Recorrer el ResultSet
while(miResultSet.next()){
    System.out.println(miResultSet.
getString("NOMBREARTICULO") + " " + miResultSet.
getString("CODIGOARTICULO")+
        miResultSet.getString("PRECIO"));
}
```

En este paso, con un **while** verificamos el contenido dentro de la tabla y recorremos los distintos campos (nombre del artículo, precio, etcétera). Como estamos trabajando con los bloques **try-catch**, ya tenemos todo lo que esperaríamos que hiciera cuando intentemos (try) realizar una consulta a la BBDD; sin embargo, en el bloque **catch** debemos hacer lo siguiente:

```
catch(Exception e){
    System.out.println("No hay acceso a la Base de Datos
en este momento");
    e.printStackTrace();
}
```

Con este mensaje y el trazado de qué se trata, podremos asegurarnos de que el usuario se quede con la incertidumbre de qué sucede con la conexión.

Cuando ejecutemos el programa, verificaremos los resultados de la consulta, y saldrá una lectura completa y correcta. En caso contrario, debería aparecernos el mensaje que colocamos en el bloque **catch**.

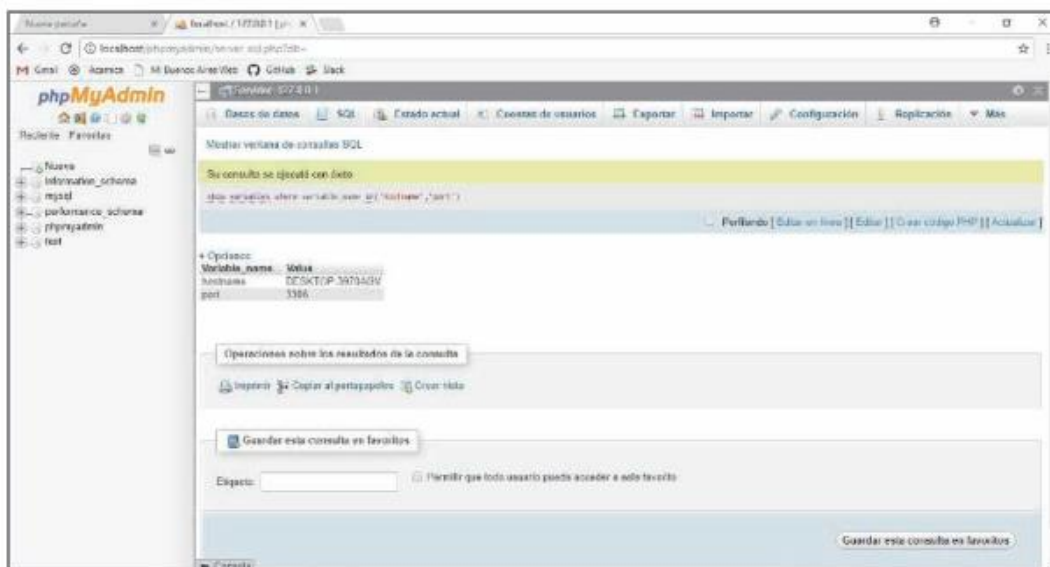


Figura 1. En la página local de phpmyAdmin, podemos verificar, mediante algunos comandos, el host name y el puerto por el que estamos accediendo a nuestra BBDD.



RESUMEN CAPÍTULO 02

Las bases de datos son muy utilizadas en informática general, y en este libro no podríamos quedar exentos de abordar estos conceptos. En este capítulo recorrimos los cuatro pasos fundamentales para poder conectarnos a una BBDD: cómo establecer la conexión, crear el objeto Statement, ejecutar una sentencia SQL y recorrer el ResultSet. También aprendimos a instalar un servidor para conectarnos a una BBDD. Y, finalmente, realizamos una conexión a través de un programa en Java para conectarnos a una BBDD existente en MySQL.

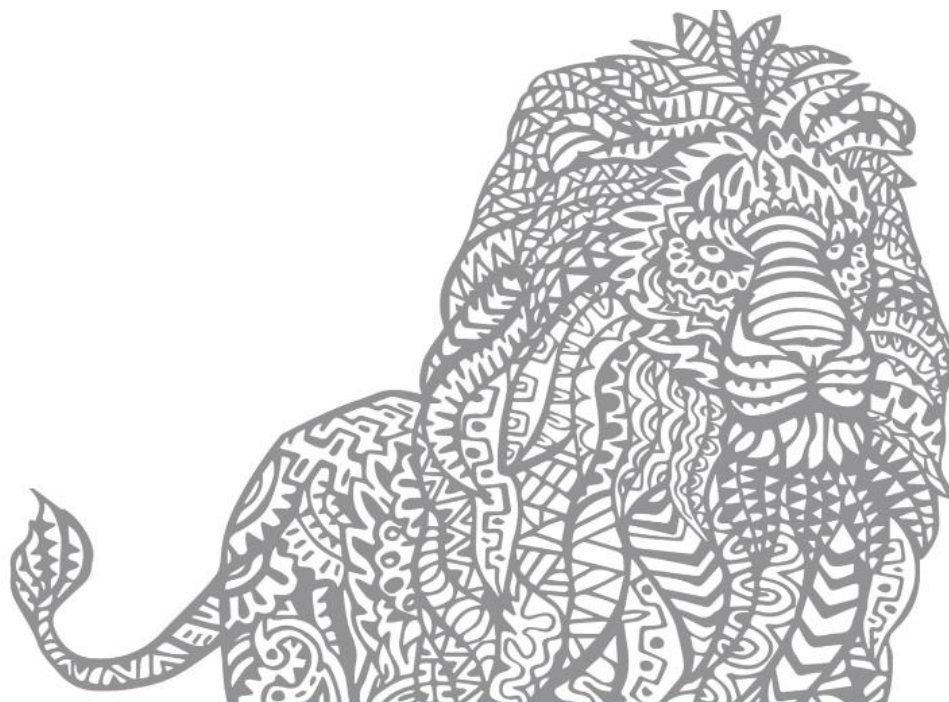
Actividades 02

Test de Autoevaluación

1. ¿Qué es una base de datos?
2. ¿Qué es un JDBC?
3. Nombre los cuatro pasos principales para establecer una conexión a una BBDD.
4. ¿Qué paquetes utiliza Java para conexiones a BBDD SQL?
5. Nombre las sentencias básicas del lenguaje SQL.
6. ¿Cómo se establece la conexión a una BBDD? ¿Qué elementos se encuentran involucrados en ella?
7. ¿Para qué sirve el objeto Statement?
8. ¿Qué función cumple el resultSet?
9. ¿Qué otra manera existe para conectarnos a una BBDD?
10. ¿Cómo hacemos para conectarnos a una BBDD a través de un programa en Java?

Ejercicios prácticos

1. Realice una conexión a una BBDD utilizando un servidor Apache, puede usar XAMPP o WAMP.
2. Cree una base de datos simple, que contenga una tabla de alumnos con algunos campos.
3. Escriba el código para conectarse a la BBDD creada y traiga alguna información de ella, como el nombre y la edad de un alumno.



Java para Android

En este apéndice veremos los aspectos básicos necesarios para programar en Android, por supuesto, trabajando con el lenguaje de programación Java. Entre otros temas, realizaremos una introducción al mundo mobile y revisaremos el uso de Android Studio.



Ap

INTRODUCCIÓN AL MUNDO MOBILE

El **mundo mobile** es una de las tecnologías IT que más ha evolucionado en los últimos años de una forma exponencial de la mano de Apple por un lado y, desde no hace mucho (casi 9 años), por parte de Google, a través de Android.

Android está basado en el **kernel de Linux** bajo la licencia Apache, es decir, al igual que en Debian o Red Hat, entre otros, también es posible descargar su código fuente y trabajar en él.

La adopción de Android ofrece una ventaja significativa a los desarrolladores, pues presenta un enfoque unificado para el desarrollo de aplicaciones (apps), o sea que solo debemos desarrollar para un entorno y usarlo en muchos dispositivos, por ejemplo, móviles, tabletas, Smart TV, etcétera.

Ventajas de usar y desarrollar para Android

Entre las ventajas que vale la pena mencionar, encontramos una gran cantidad de **bibliotecas** disponibles para el desarrollo, la mayoría en C y C++ (predecesores de Java).

También es importante que los **componentes** se pueden sustituir con facilidad por otros, siempre se busca la **portabilidad** y la **reutilización** de las aplicaciones. Por otra parte, cuenta con una máquina virtual propia, **Dalvik**, que se encarga de interpretar y ejecutar los códigos hechos en Java, permite la representación en gráficos tanto en 2D como en 3D, y posibilita el acceso a la **base de datos** y su utilización.

También soporta una importante cantidad de formatos multimedia; gracias a su versatilidad, podemos controlar numerosos elementos de hardware, mediante Bluetooth, Wi-Fi, cámara fotográfica o video, GPS, y todo lo que un dispositivo móvil tenga incorporado. Podemos acceder a un SDK gratuito como entorno de desarrollo; además nos ofrece plugin para entornos de desarrollo más populares, como **Android Studio** o **Eclipse**, así como los emuladores para probar nuestras aplicaciones antes de ser subidas a **Google Play**.

Las bibliotecas proporcionan a Android la mayor parte de las capacidades con las que cuenta. Podemos mencionar algunas de ellas:

- ▶ Librería **libc**: incluye las cabeceras y las funciones según el estándar de C.
- ▶ Librería **Surface Manager**: se encarga de colocar los elementos de navegación en la pantalla.
- ▶ **OpenGL/SL y SGL**: para la parte gráfica tanto en 2D como en 3D.
- ▶ Librería **Media Libraries**: proporciona los códecs para el contenido multimedia.
- ▶ **FreeType**: para las diferentes fuentes.
- ▶ Librería **SSL**: para realizar comunicaciones seguras.
- ▶ Librería **SQLite**: para la creación y gestión de bases de datos.
- ▶ Librería **WebKit**: proporciona el motor para las apps de tipo navegador.

ANDROID STUDIO

En este capítulo haremos énfasis en el entorno en que desarrollaremos para Android: **Android Studio**, el que podemos descargar desde su sitio oficial <https://developer.android.com/studio/index.html>.



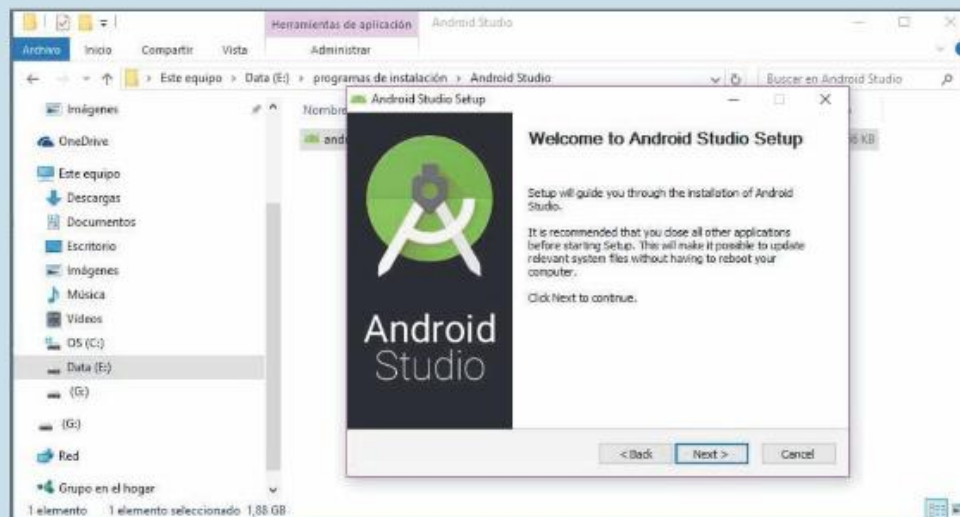
Figura 1.
Sitio oficial de
descarga del
IDE Android
Studio.

Una vez descargado, procedemos con el proceso de instalación, tal como vemos en las siguientes instrucciones.



PASO A PASO: INSTALAR ANDROID STUDIO

01 Proceda con la instalación de Android Studio, haciendo doble clic en el archivo .EXE que descargó desde el sitio oficial del IDE. Siga cada una de las indicaciones del asistente de instalación.



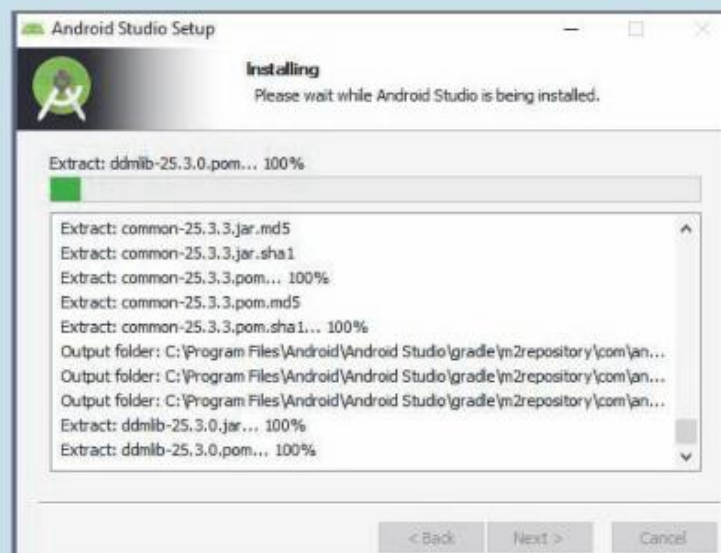
Fragmentación en Android

La **fragmentación** es uno de los problemas que más se le ha criticado a Android, esto quiere decir que para cada versión del sistema existen muchas versiones menores, cosa que no ocurre con Apple. Entre las ventajas de esto, encontramos que permite a los usuarios una mayor variedad a la hora de elegir dispositivos. En cuanto a las desventajas relacionadas, ocurre que dificulta el trabajo a los desarrolladores pues el diseño debe cambiar según el dispositivo y la versión.

- 02** El asistente le advertirá del espacio en disco que va ser requerido, luego deberá aceptar el contrato de uso del software. Más adelante elija la carpeta en donde será instalado el IDE y presione **Next**.



- 03** Elija los componentes que serán instalados y también la ubicación del IDE en el menú **Inicio** de Windows. Haga clic en **Install** y espere mientras la instalación se completa. Puede presionar **Show details** para observar el proceso de extracción de los archivos involucrados.



04 Al cabo de algunos minutos verá que la instalación se ha realizado de manera correcta, presione **Next**. Luego de ver el resumen de la instalación, haga clic sobre **Finish**.

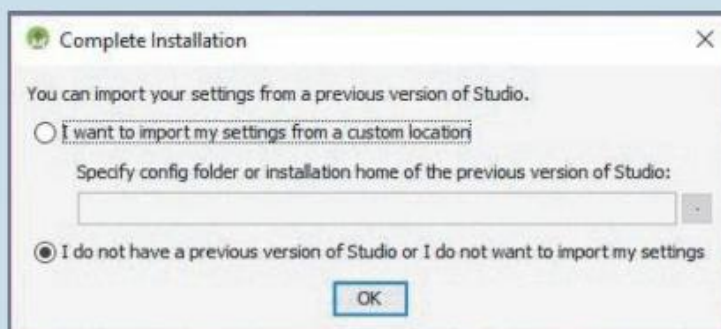


Una vez instalado, el IDE se ejecutará por primera vez y podremos crear nuestro primer proyecto en Android Studio. Para lograrlo, debemos seguir las instrucciones que se detallan en el siguiente **Paso a paso**:

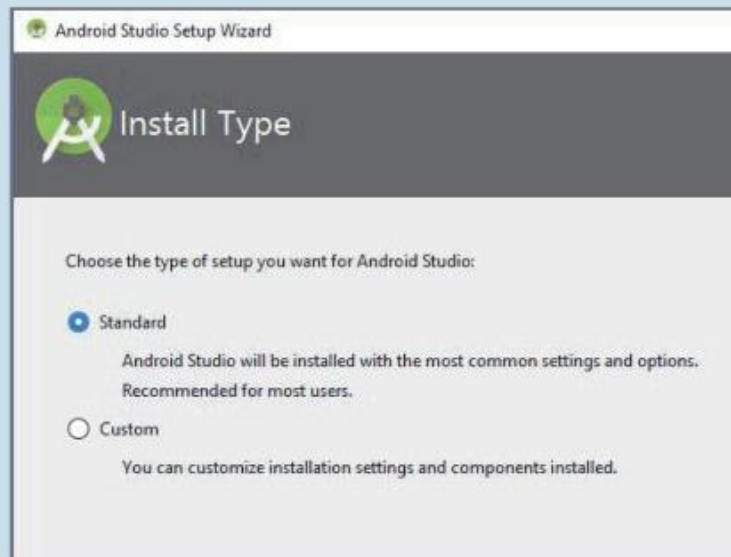


PASO A PASO: CREAR UN PROYECTO EN ANDROID

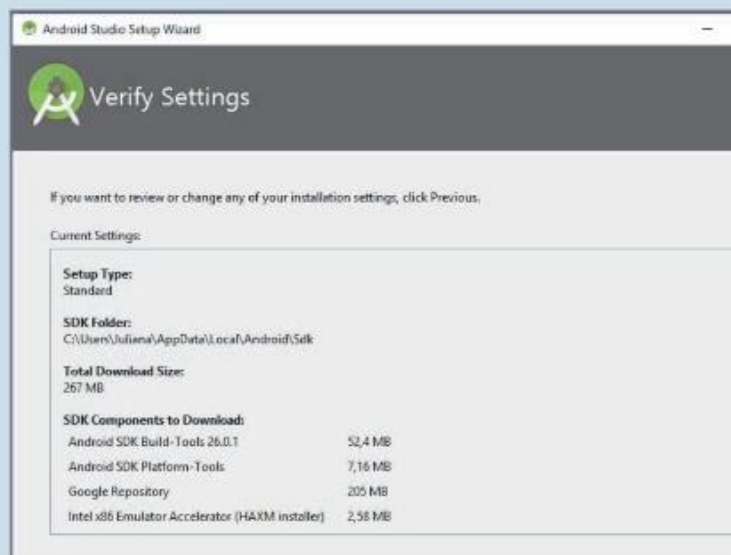
01 Al iniciar, el IDE preguntará si desea importar alguna de las versiones previas de configuración, indique que no tiene nada que importar.



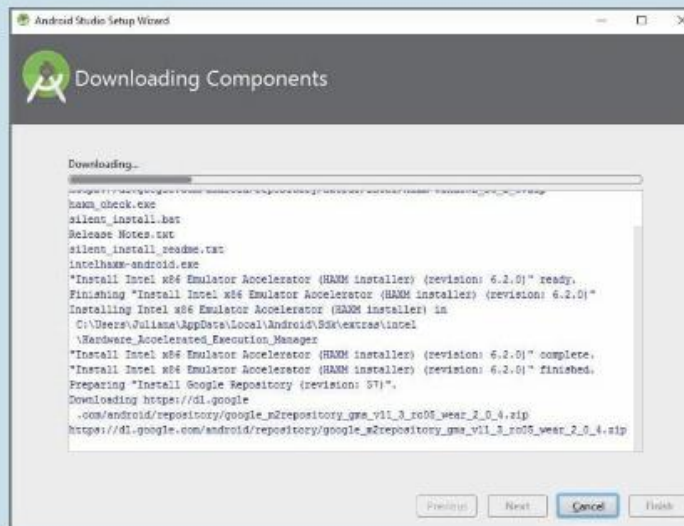
02 Luego verá un mensaje de bienvenida; por supuesto, les dará siempre una opción a los usuarios estándares y otra, a los usuarios avanzados. En el segundo caso puede personalizar sus opciones, pero en esta ocasión opte por la opción **Standard**.



03 En la siguiente pantalla verá un resumen de todo lo que ha sido configurado, haga clic sobre el botón **Finish**.



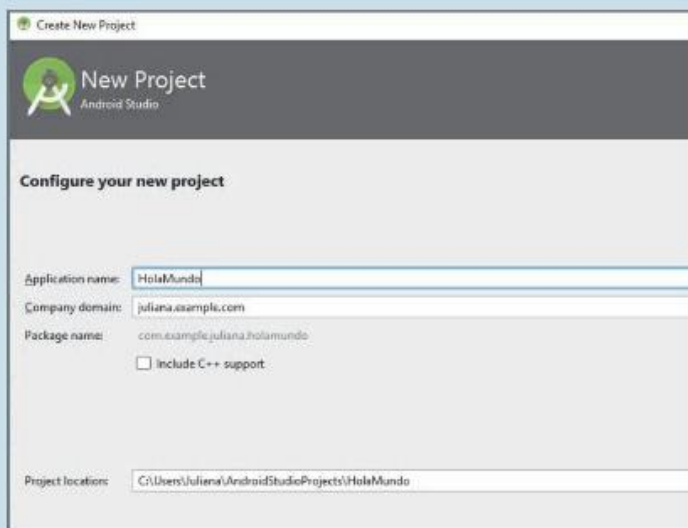
04 Android Studio descargará los últimos componentes necesarios para su utilización. Puede hacer clic en **Show details** y ver qué elementos está cargando.



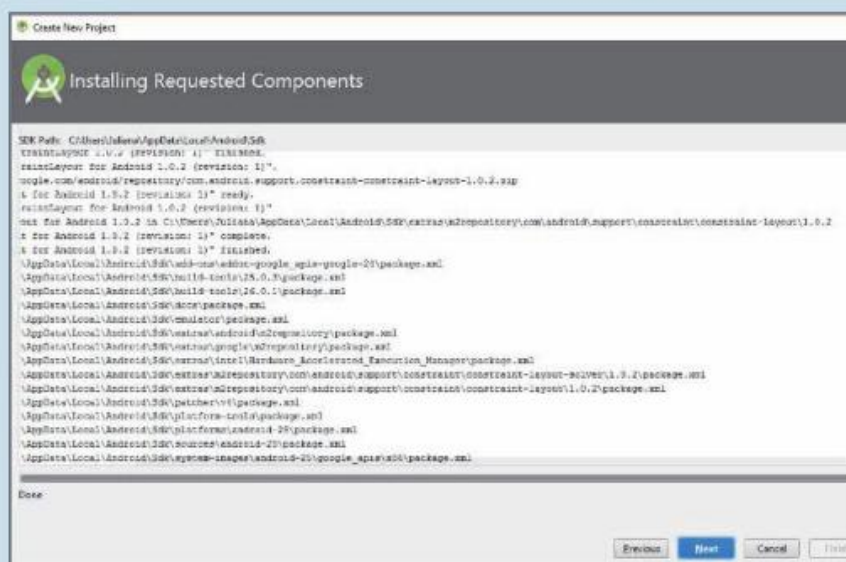
05 Ahora se mostrará un resumen de lo instalado, haga clic en **Finish**. Verá varias opciones que le permitirán iniciar un nuevo proyecto, abrir alguno en el que ya esté trabajando, importar de Eclipse u otra IDE, o bien acceder a algunos ejemplos.



06 Para este ejemplo elija la primera opción (**Start a New Android Project**), pues realizará su primer proyecto. Indique el nombre que ha elegido.



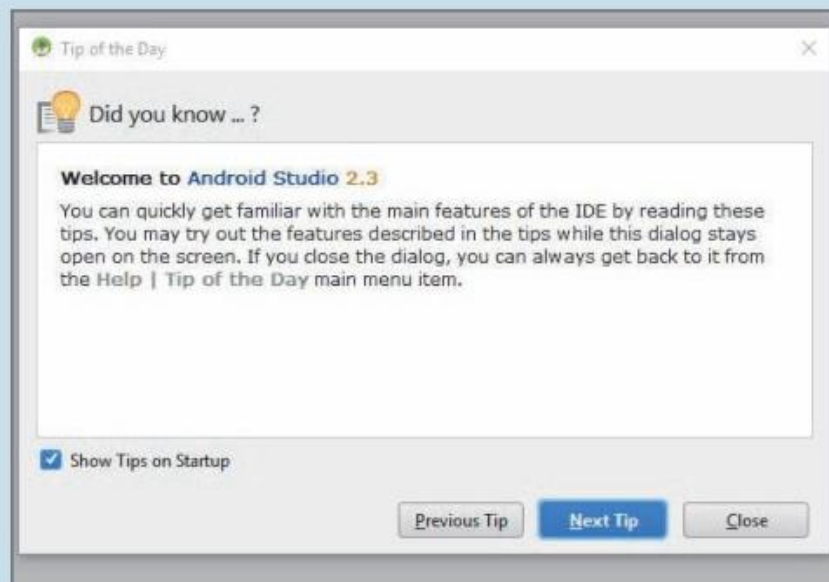
07 En este punto deberá indicar en qué tipo de app trabajará, por ahora deje la opción predeterminada. Luego de esto verá un resumen de lo configurado.



08 Necesitará indicar un fondo para la aplicación, esta vez elija blanco, así podrá diseñarlo a su gusto. Indique un nombre para la actividad y espere a que el IDE realice los cambios necesarios de configuración.



09 Luego de esto ya ha completado la configuración inicial de Android Studio y ha logrado crear su primer proyecto.



ESTRUCTURA BÁSICA DE UN PROYECTO

Cuando estemos ejecutando Android Studio, debemos tener paciencia a la hora de empezar a trabajar, pues se trata de un programa bastante pesado por lo que es recomendable tener, al menos, 8 GB de memoria disponible.

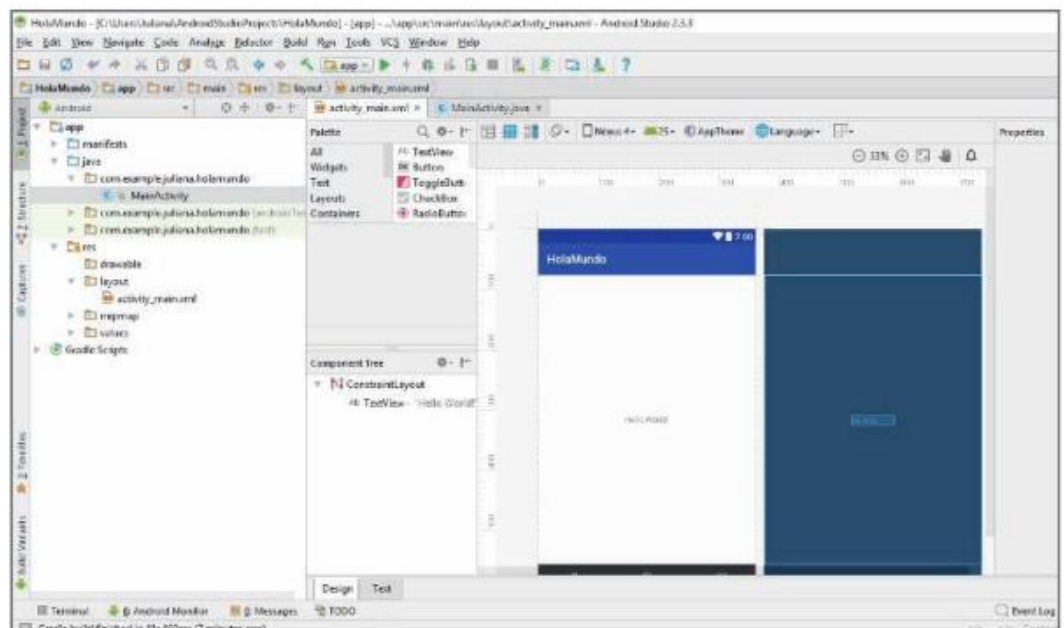


Figura 2. Vista inicial del entorno de trabajo de nuestro IDE.

Para empezar a comprender cómo se construye una aplicación Android, vamos a dar un vistazo a la estructura general de un proyecto.

Cuando creamos un nuevo proyecto Android en Android Studio, se genera automáticamente la estructura de carpetas necesaria para poder generar luego la aplicación. Esta estructura será común a cualquier aplicación, en forma independiente de su tamaño y su complejidad.

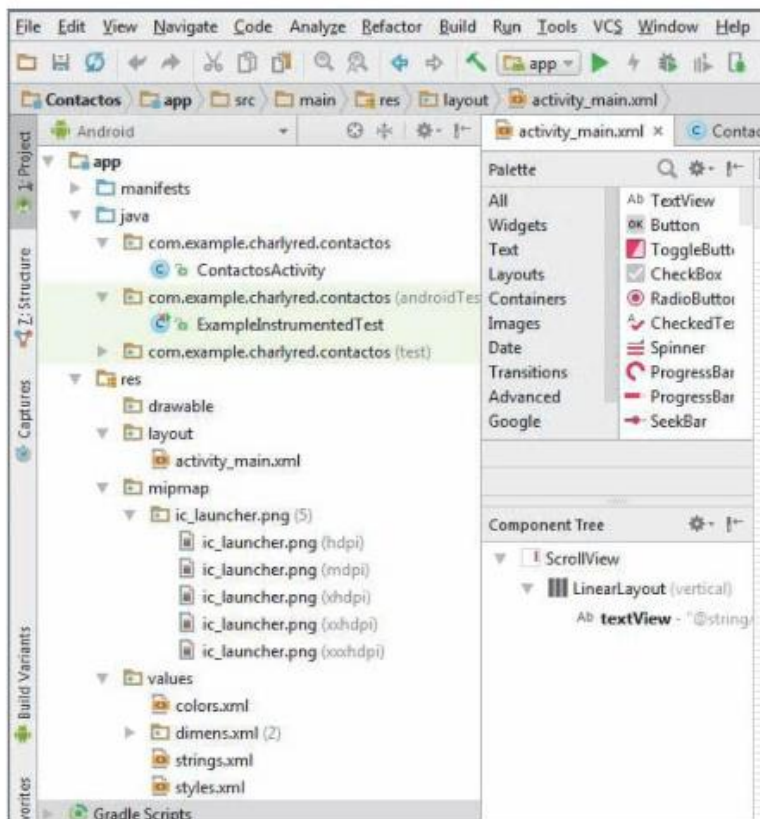


Figura 3. En la imagen podemos ver los elementos creados inicialmente para un nuevo proyecto Android.

Exploración de un proyecto

Cuando analizamos de cerca la estructura de carpetas que se nos presentan en un proyecto, encontramos las siguientes:

/src

Contiene todos los archivos fuente de Java.

/res

Presenta las carpetas e interfaces, los recursos necesarios para el proyecto, como imágenes, videos, cadena de texto, etcétera. Si lo desplegamos, aparecerán otras subcarpetas:

drawable: para las imágenes. Hay ldpi, mdpi, hdpi, esto depende de la resolución del dispositivo.

layout: contendrá los archivos para la definición de las diferentes pantallas de la interfaz gráfica.

anim: para las animaciones.

menu: para los menús de la aplicación.

values: para los strings, styles, colors.

xml: contiene los archivos XML de la aplicación.

raw: para los recursos adicionales y que no estén incluidos en la estructura original, por ejemplo, sonidos.

AndroidManifest

Contiene la definición en XML de los aspectos principales de la aplicación, como por ejemplo, su identificación (nombre, versión, icono, etcétera), sus componentes (pantallas, mensajes), o los permisos necesarios para su ejecución. Veamos la estructura del archivo **manifest.xml**:

```
<?xml version="1.0" encoding="utf-8"?>

<manifest>
    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity>
```

```

        <intent-filter>
            <action />
            <category />
            <data />
        </intent-filter>
        <meta-data />
    </activity>

    <activity-alias>
        <intent-filter> . . . </intent-filter>
        <meta-data />
    </activity-alias>

    <service>
        <intent-filter> . . . </intent-filter>
        <meta-data/>
    </service>

    <receiver>
        <intent-filter> . . . </intent-filter>
        <meta-data />
    </receiver>

    <provider>
        <grant-uri-permission />
        <meta-data />
        <path-permission />
    </provider>

    <uses-library />

</application>

</manifest>

```

Este tipo de archivo está basado en etiquetas, como HTML, aunque en rigor a la verdad difiere mucho del famoso lenguaje de hipertexto.

Componentes de una aplicación Android

En Java o .NET estamos acostumbrados a manejar conceptos como **ventana, control, eventos** o **servicios**, como los elementos básicos en la construcción de una aplicación. Pues bien, en Android vamos a disponer de esos mismos elementos básicos, aunque con un pequeño cambio en la terminología y el enfoque. Repasemos los componentes principales que pueden formar parte de una aplicación Android, veamos:

ACTIVITY

Las actividades van a representar el componente principal de la interfaz gráfica de una aplicación Android. Se puede pensar en una actividad como el elemento análogo a una ventana en cualquier otro entorno visual.

VIEW

Estos objetos son los componentes básicos con los que se construye la interfaz gráfica de la aplicación, análoga por ejemplo a los controles de Java. Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.

SERVICE

Son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto, son exactamente iguales a los servicios presentes en cualquier otro sistema operativo. Los servicios pueden realizar cualquier tipo de acciones, por ejemplo, actualizar datos, lanzar notificaciones o, incluso, mostrar elementos visuales (por ejemplo, las actividades) si se necesita en algún momento la interacción con el usuario.

CONTENT PROVIDER

Nos servirá para compartir datos entre aplicaciones. Mediante estos componentes es posible compartir determinados datos de nuestra aplicación sin mostrar detalles sobre su almacenamiento interno, su estructura o su implementación. De la misma forma, nuestra aplicación podrá acceder a los datos de otra a través de los content provider que se hayan definido.

BROADCAST PROVIDER

Este componente está destinado a detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema (por ejemplo: "Batería baja", "SMS recibido", "Tarjeta SD insertada", etcétera) o por otras aplicaciones ya que todas las aplicaciones pueden generar mensajes (intents) broadcast, es decir, no dirigidos a una aplicación concreta, sino a cualquiera que desee escucharlo.

WIDGETS

Son elementos visuales, normalmente interactivos, que pueden mostrarse en la pantalla principal (home screen) del dispositivo Android y recibir actualizaciones periódicas. Ofrecen información de la aplicación al usuario directamente sobre la pantalla principal.

INTENT

Es el elemento básico de comunicación entre los distintos componentes Android que hemos descrito antes. Básicamente son los mensajes o peticiones que se envían entre los distintos componentes de una aplicación o entre diferentes aplicaciones. Mediante un intent se puede mostrar una actividad desde cualquier otra, iniciar un servicio, enviar un mensaje broadcast, iniciar otra aplicación, etcétera.

ACTIVIDADES BÁSICAS

En todo proyecto de Android es necesario saber que existe lo que se denomina **Activity**, una ventana que contiene la interfaz del usuario –o pantalla– para su aplicación, y los usuarios pueden interactuar directamente a través de ellas. Estas son los componentes más habituales de las aplicaciones para Android. La mayoría de las aplicaciones permiten la ejecución de varias acciones a través de una o más pantallas. Android admite controlar por completo el ciclo de vida de los componentes **Activity**.

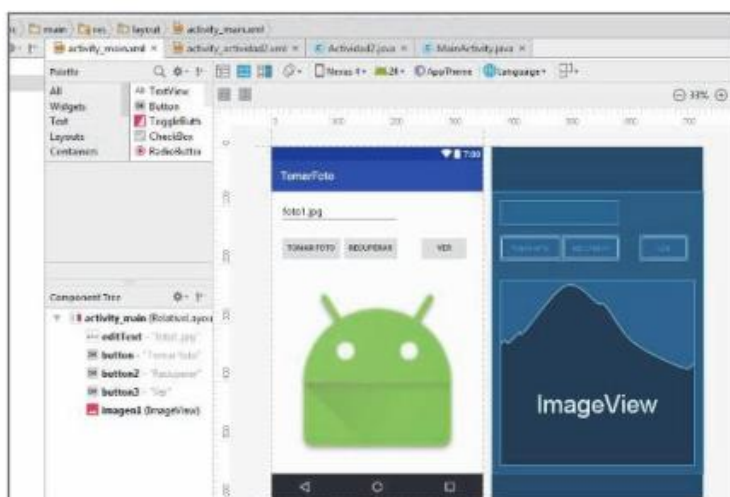


Figura 4. Vista de una Activity, en cuya parte derecha está un nuevo componente en Android Studio.

Ciclo de vida

En Android, cada aplicación se ejecuta en su propio **proceso**, y esto nos aportará beneficios en cuestiones básicas, como seguridad, gestión de memoria o la ocupación de la CPU del dispositivo. Android se ocupa de lanzar y parar todos estos procesos, gestionar su ejecución, y decidir qué hacer en función de los recursos disponibles y de las órdenes dadas por el usuario.

El usuario no tiene conocimiento de este comportamiento, solo es consciente de que, mediante un simple clic, pasa a otra aplicación y vuelve a una anterior que ya tenía abierta. No se preocupa de cuál es la aplicación que se encuentra activa o cuánta memoria está consumiendo, ni siquiera de si existen recursos suficientes para abrir una siguiente aplicación.

Cada uno de los **componentes básicos** de Android tiene un ciclo de vida bien definido; esto implica que el desarrollador puede controlar cada momento en qué estado se encuentra dicho componente, y así programar las acciones que más convienen.

El componente **Activity**, probablemente el más importante, tiene un ciclo de vida que a continuación describiremos:

- ▶ **onCreate()**: indica el principio de la actividad.
- ▶ **onDestroy()**: indica el fin de la actividad.
- ▶ **onStart()** – **onStop()**: representan la parte visible del ciclo de vida.
- ▶ **onResume()** – **onPause()**: delimitan la parte útil del ciclo de vida. Esta parte no solo es visible, sino que tiene un foco de acción, y el usuario puede interactuar con ella.

En cuanto al orden de los eventos que suceden cuando se lanza una aplicación de Android es la siguiente:

1

Al ejecutarse por vez primera la aplicación:

- Constructor
- Evento onStart
- Evento onResume
- Evento surfaceCreated
- Evento surfaceChanged

2

En el momento en que se pulsa la tecla inicio:

- onPause
- surfaceDestroyed
- onStop

3

Al retomar la aplicación:

- onRestart
- onStart
- onResume
- surfaceCreated
- surfaceChanged

4

En el momento de pulsar la tecla ESC:

- onPause
- onStop
- surfaceDestroyed

XML

Es un subconjunto de **SGML** (Standart Generalized Mark-up Language), aunque en rigor no es propiamente un lenguaje de marcado como sí lo es HTML, y podríamos considerarlo como un metalenguaje.

Las **Activities** llevan su extensión, lo que indica que, dentro de ellas, tendremos código en XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_actividad2"
    android:layout_width="match_parent"
```

```
        android:layout_height="match_parent"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context="com.example.charlyred.tomarfoto.Activi-
dad2">

        <ListView
            android:layout_width="match_parent"
            android:layout_height="213dp"
            android:id="@+id/listView1"
            android:layout_weight="1"/>

        <ImageView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:srcCompat="@mipmap/ic_launcher"
            android:id="@+id/imageView"
            android:layout_weight="1" />
    </LinearLayout>
```

En la codificación verificamos que existen etiquetas, como **ImageView**, **LinearLayout** o **ListView**, y dentro de ellas se ingresarán las características o los atributos propios de las componentes, como por ejemplo: ancho, si va a abarcar todo el ancho o parte del alto, un id, etcétera.

INTERACCIÓN ENTRE JAVA Y ANDROID

Para que exista una interconexión entre los archivos nativos de Android y Java, deberá existir una especie de dualidad, casi mágica, esto es el archivo **R.java**. Entendamos que los recursos en Android están por todos lados y, cada vez que generemos cualquier tipo de archivo, estos van a

estar presentes, directa o indirectamente. Pero un archivo de recursos puede contener dentro una cantidad de recursos. Recordemos que se encuentran dentro de la carpeta **/res**.

Entonces, el archivo **R** se genera automáticamente y, no importa si lo borramos, este se volverá a regenerar. En sí, el archivo es bastante pequeño, pero lo suficientemente importante como para darnos cuenta de que es la palanca que mueve todo este enjambre de archivos.

Si analizamos su código, veremos que se trata de una clase llena de variables estáticas en las que se identifica cada tipo de recurso. Como podemos ver, la cadena de **holaMundo** está en la sección de **Strings**, y el id **texto1**, en la sección de **id**. Pero ¿para qué sirve esta clase exactamente? Usamos archivos XML para definir varias estructuras (cadenas de texto, layouts, estilos, etcétera).

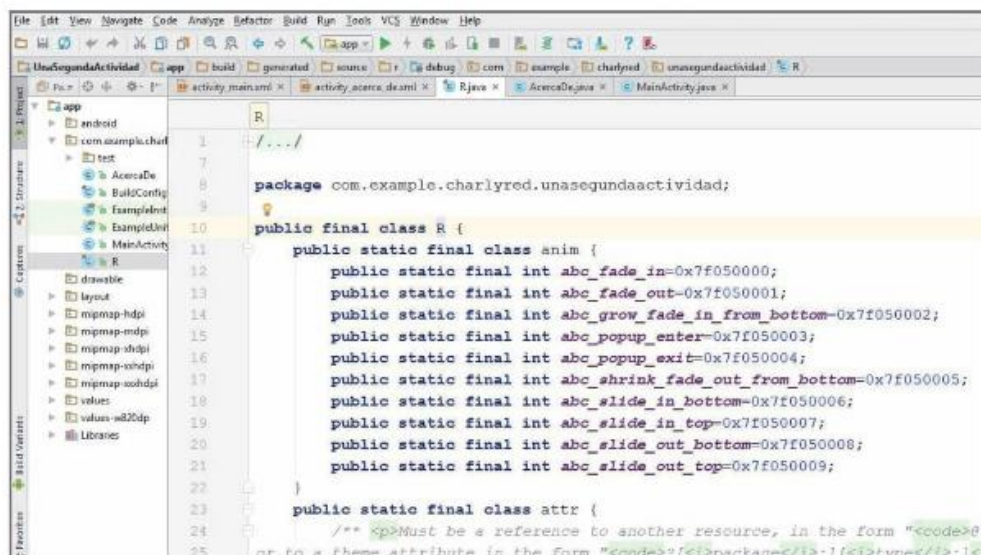


Figura 5. El archivo **R**, un nexo entre la parte visual y Java.

Cómo usar el archivo R

Vamos a ver un sencillo bloque de código en el cual usamos el archivo **R** para localizar el **TextView** de la aplicación y cambiar su contenido. Para ello modificaremos el archivo de recursos de **strings.xml** y añadiremos una nueva cadena **“hola_mundo”**. Para mostrar un mensaje en la consola, escribimos el siguiente código:

El código fundamental son estas dos líneas

```
TextView textol= (TextView) findViewById(R.id.textol);
textol.setText(R.string.hola_mundo);
```

El método **findViewById** nos localiza el **TextView** usando el archivo **R** y el bloque de identificadores; luego, usamos el archivo **R** y su acceso a las cadenas para cambiar el contenido.

El programa dará como resultado **hola mundo**.

PRIMERAS APPS

Diseñaremos lo que será la pantalla principal. Ya Android Studio nos genera una de forma predeterminada. Pero ¿dónde y cómo se define cada pantalla de la aplicación? En principio, el diseño y la lógica de una pantalla están separados en dos archivos distintos. Por un lado, en el archivo **/res/layout/activity_main.xml** tendremos el diseño puramente visual de la pantalla, definido como archivo **XML** y, por otro lado, en el archivo **/app/java/MainActivity.java**, encontraremos el código en lenguaje Java, que determina la lógica de la pantalla. Verifiquemos el archivo XML:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
```

```
tools:context="com.example.charlyred.unasegundaactividad.Main-
Activity">

    <TextView
        android:text="Ingrese la Clave"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginLeft="11dp"
        android:layout_marginStart="11dp"
        android:id="@+id/textView3" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="textPassword"
        android:ems="10"
        android:layout_below="@+id/textView3"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="29dp"
        android:id="@+id/editText" />

    <Button
        android:text="VERIFICAR"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button"
        android:onClick="verificar"
        android:layout_below="@+id/editText"
        android:layout_alignRight="@+id/editText"
        android:layout_alignEnd="@+id/editText"
        android:layout_marginRight="20dp"
        android:layout_marginEnd="20dp"
        android:layout_marginTop="69dp" />

</RelativeLayout>
```

En este XML se definen los elementos visuales que componen la interfaz de nuestra pantalla principal, y se especifican todas sus propiedades.

Encontramos el **RelativeLayout**. Los **layout** son elementos no visibles que determinan cómo se van a distribuir en el espacio los controles que incluyamos en su interior. Los programadores de Java, y más concretamente de **Swing**, conocerán este concepto perfectamente. En este caso, un **RelativeLayout** distribuirá los controles uno tras otro y en la orientación que indique su propiedad **android:orientation**.

Dentro del layout, incluiremos tres controles: una etiqueta (**TextView**), un cuadro de texto (**EditText**) y un botón (**Button**).

Con esto ya tenemos definida la presentación visual de nuestra ventana principal de la aplicación. De igual forma definiremos la interfaz de la segunda pantalla, creando un nuevo archivo llamado **activity_acerca_de.xml**, y añadiremos algunas etiquetas (**TextView**) para mostrar el mensaje personalizado al usuario, y un botón **salir**. Veamos cómo quedaría nuestra segunda pantalla:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_acerca_de"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.charlyred.unasegundaactividad.Acer-
caDe">

    <TextView
        android:text="Developer:"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
```

```

        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginLeft="17dp"
        android:layout_marginStart="17dp"
        android:layout_marginTop="57dp"
        android:id="@+id/textView"
        android:textColorLink="?android:attr/colorPressedHigh-
light"

        android:textColor="@android:color/holo_blue_dark"
        android:textSize="24sp" />

<TextView
    android:text="Carlos A. Arroyo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/textView"
    android:layout_toRightOf="@+id/textView"
    android:layout_toEndOf="@+id/textView"
    android:layout_marginTop="36dp"
    android:id="@+id/textView2"

    android:textSize="18sp"
    android:textStyle="italic"
    android:textColor="@android:color/holo_green_dark" />

<Button
    android:text="Salir"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/textView2"
    android:layout_alignLeft="@+id/textView2"
    android:layout_alignStart="@+id/textView2"
    android:layout_marginTop="114dp"
    android:id="@+id/button2"
    android:onClick="salir" />
</RelativeLayout>

```

Una vez definida la interfaz de las pantallas de la aplicación, deberemos implementar su lógica. Como ya hemos dicho antes, la lógica de la aplicación se definirá en archivos del tipo Java independientes. Para la pantalla principal, ya tenemos creado un archivo por defecto llamado **MainActivity.java**.

```
package com.example.charlyred.project001;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity{

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

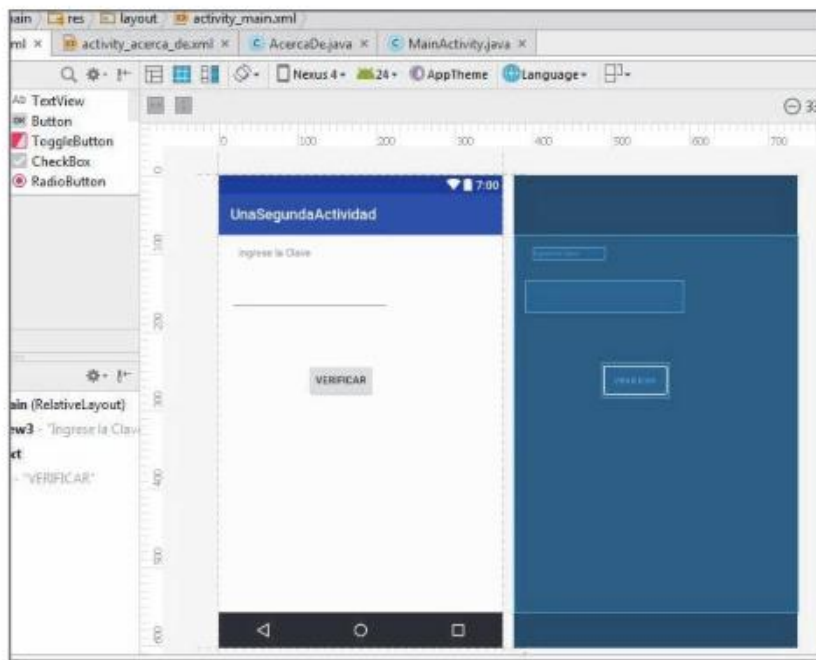


Figura 6. Al seleccionar este archivo, Android Studio nos permite visualizar el contenido en Design o Text.

Android Studio ya insertó un control de tipo **RelativeLayout** que permite ingresar controles visuales alineados a los bordes y a otros controles que haya en la ventana.

Ya veremos que podemos modificar todo este archivo para que se adapte a la aplicación que queremos desarrollar.

Antes de probar la aplicación en el emulador de un dispositivo Android, procederemos a hacer un pequeño cambio a la interfaz que aparece en el celular: borraremos la **label** que dice **Hello World** (simplemente seleccionando con el mouse dicho elemento y presionando la tecla **delete**); de la **Palette** arrastraremos un **Button** al centro del celular y, en la ventana **Properties**, con el **Button** seleccionado, cambiaremos la propiedad **text** por la cadena **Verificar**. Además agregaremos una caja de texto para poder ingresar una contraseña, y una etiqueta que contenga **Ingrese la clave**.

El código en Java que contendrá este archivo será:

```
package com.example.charlyred.unasegundaactividad;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.AutoCompleteTextView;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
    private EditText et1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        et1 = (EditText)findViewById(R.id.editText);
    }
    public void verificar(View v){
        String clave = et1.getText().toString();
```

```

        if(clave.equals("redusers2017")){
            Intent i = new Intent(this,AcercaDe.class);
            startActivity(i);
        }else{
            Toast notificacion = Toast.makeText(this,"Clave
Incorrecta",Toast.LENGTH_LONG);
            notificacion.show();
        }
    }
}

```

El otro archivo de Java, **AcercaDe**, solo contendrá la siguiente codificación:

```

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;

public class AcercaDe extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_acerca_de);
    }
    public void salir(View v){

        finish();
    }
}

```

Donde el método **salir** hará que se cierre la pantalla.

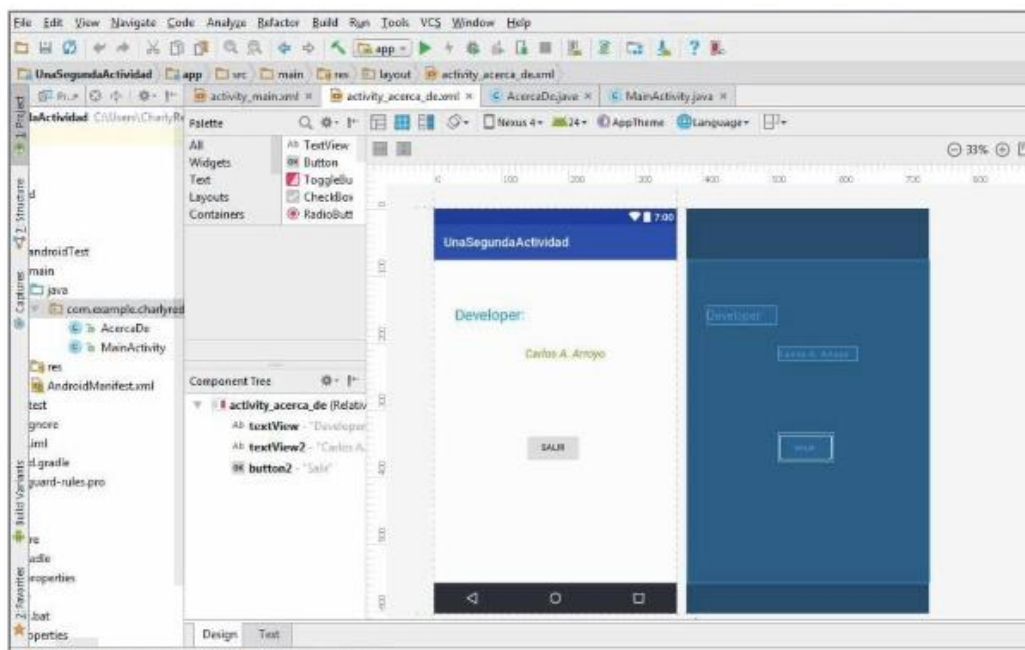


Figura 7. Vista de la segunda actividad. Observamos que contiene algunas etiquetas y un botón que hará que se salga de esta pantalla.

Hasta acá la pequeña aplicación debería funcionar, puesto que no hemos detectado error alguno. Debemos ejecutarla, pero la primera vez necesitamos de un emulador que, a continuación, configuraremos.

Uso del emulador de Android Studio

Para ejecutar la aplicación, presionamos el triángulo verde o seleccionamos del menú de opciones **Run/ Run app** y en este diálogo procedemos a dejar seleccionado el emulador por defecto que aparece (**Nexus 5X** por ejemplo) y presionamos el botón **OK**.

En caso de no tener un emulador como la vez primera que lo usamos, debemos seguir el asistente luego de presionado el botón **Create New virtual Device**.

Luego de un momento (el lanzamiento del emulador puede llevar más de un minuto), aparecerá el emulador de Android en pantalla. Es importante tener en cuenta que, una vez que el emulador ha arrancado, si seguimos probando nuestra aplicación no lo debemos cerrar cada vez que hacemos cambios o codificamos otras aplicaciones, sino

que volvemos a ejecutar la aplicación con los cambios y, al estar el emulador corriendo, el tiempo que tarda hasta que aparece nuestro programa en el emulador es muy reducido.

Cuando terminó de cargarse, el emulador debe aparecer en nuestra aplicación ejecutándose.

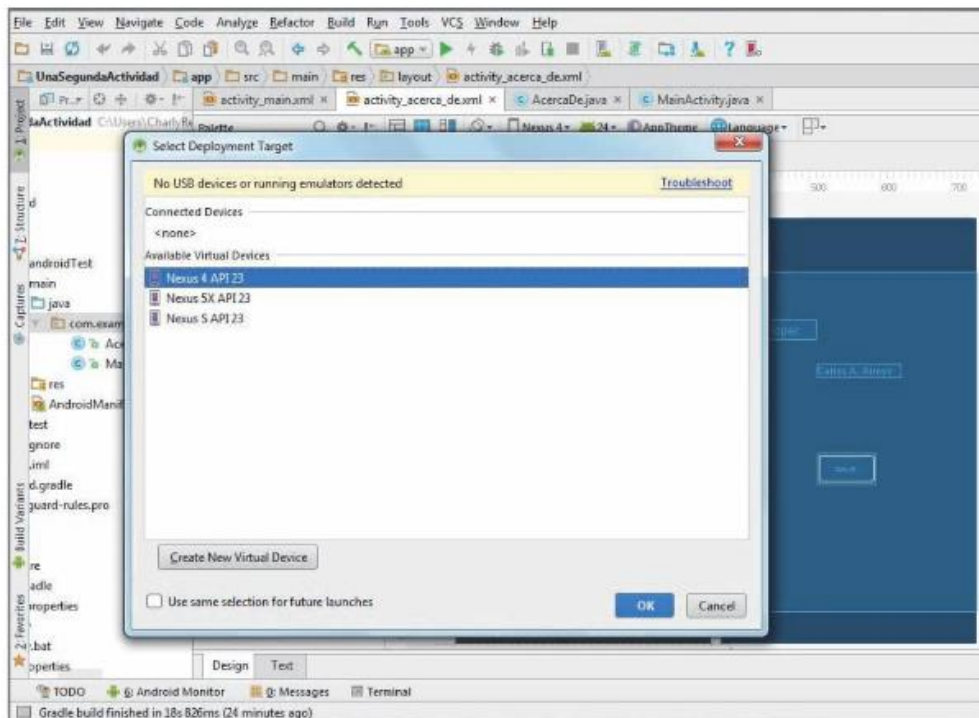


Figura 8. En la figura se muestra un emulador dentro de la lista que podemos trabajar.

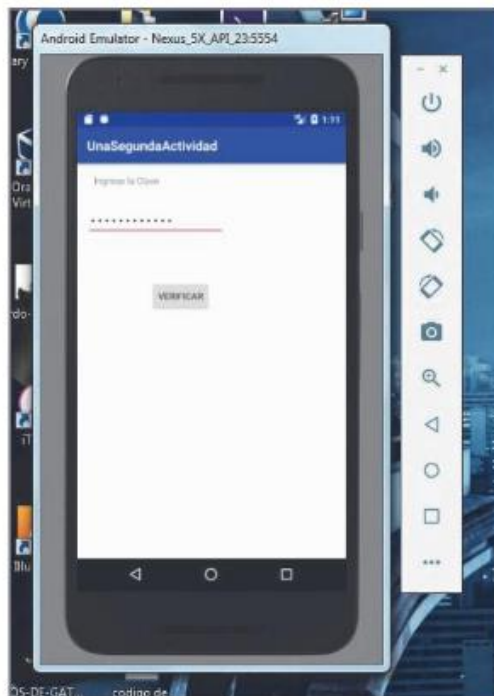


Figura 9. Vista de cómo se visualiza la aplicación en el emulador.



RESUMEN CAPÍTULO AP

En este capítulo profundizamos en Android y el mundo mobile, instalamos el IDE que nos permitirá nuestras primeras apps y conocimos la estructura de un proyecto en Android. Entendimos la dualidad que existe entre un archivo XML y uno de Java, y cómo de alguna manera se concatenan y coexisten para poder crear una aplicación para Android. Finalmente ejecutamos nuestra aplicación no sin antes haber creado un emulador para verificar que funciona tal como las apps de los móviles.

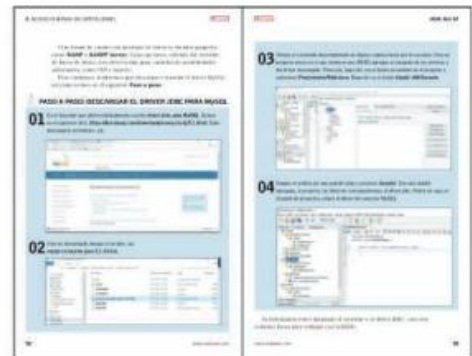
USERS

Programación en Java™

Vol. IV

ACERCA DE ESTE CURSO

Java es uno de los lenguajes más robustos y populares en la actualidad, existe hace más de 20 años y ha sabido dar los giros adecuados para mantenerse vigente. Este curso de Programación en Java nos enseña, desde cero, todo lo que necesitamos para aprender a programar y, mediante ejemplos prácticos, actividades y guías paso a paso, nos presenta desde las nociones básicas de la sintaxis y codificación en Java hasta conceptos avanzados como el acceso a bases de datos y la programación para móviles.



ACERCA DE ESTE VOLUMEN

En este volumen se enseña el uso de las estructuras de datos dinámicas, el acceso a bases de datos y la programación Java para Android.



► SOBRE EL AUTOR

Carlos Arroyo Díaz es programador, escritor especializado en tecnologías y docente. Se desempeña como profesor de Informática General, Java y Desarrollo Web. También ha trabajado como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires.

► RedUSERS

En nuestro sitio podrá encontrar noticias relacionadas y participar de la comunidad de tecnología más importante de América Latina.

► RedUSERS PREMIUM

RedUSERS PREMIUM la biblioteca digital de USERS. Accederás a cientos de publicaciones: Informes; eBooks; Guías; Revistas; Cursos. Todo el contenido está disponible online - offline y para cualquier dispositivo. Publicamos, al menos, una novedad cada 7 días

