



# Práctica 4

Sudoku

Resolución

Variables de instancia

Constructores

Métodos auxiliares

Implementados en la plantilla

Implementados por mí

Métodos sobrescritos

Antonio J. Galán Herrera

Ingeniería Informática (A)

Nota alcanzada en SIETTE: 100 / 100.

## Sudoku

Un sudoku es un juego que consiste en rellenar una tabla de 9x9 celda y 9 subtableros de 3x3, con números del 1 al 9, cumpliéndose las siguientes reglas:

- No puede repetirse un número en la fila donde esté colocado.
- No puede repetirse un número en la columna donde esté colocado.
- No puede repetirse un número en el subtablero donde esté colocado.

	6	5					4	
9			7		6	8		
	1			4			7	6
2	8			9	7	5		
3			2		1	4		
	2		1					
6								
1		4	6		8			5

7	6	5	3	8	9	1	4	2
9	4	2	7	1	6	8	5	3
8	1	3	5	4	2	9	7	6
2	8	6	4	9	7	5	3	1
4	5	1	8	6	3	7	2	9
3	9	7	2	5	1	4	6	8
5	2	9	1	3	4	6	8	7
6	7	8	9	2	5	3	1	4
1	3	4	6	7	8	2	9	5

## Resolución



Todos los métodos incluidos en la memoria cuentan con `javadocs` en el fichero de código, pero estos no se han añadido aquí para que sea más legible y ligero.

Si se necesita un contexto más amplio, recomiendo leer directamente el fichero, ya que está comentado.


```
public static void main(String[] args) {
    String configuracion = ".4....36263.941...5.7.3.....9.3751..3.48.....17..62...716.9..2...96.....312..9.";
    TableroSudoku sudoku = new TableroSudoku(configuracion);
    List<TableroSudoku> soluciones = sudoku.resolverTodos();


    // Usar este código si se quiere debuguear
    System.out.println("Sudoku planteado:\n" + sudoku.tablero());
    System.out.println("Se encontraron " + soluciones.size() + " soluciones:");


    for (TableroSudoku solucion : soluciones) {
        System.out.println(solucion.tablero());
    }

    // Usar este código si se quiere corregir usando SIETTE
    // System.out.println(sudoku);
    // System.out.println(soluciones.size());
    //
    // for (TableroSudoku solucion : soluciones) {
```

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

 Este método ya venía definido en los archivos necesarios para realizar la práctica.

 Sin embargo, se ha modificado para hacer uso del método `tablero()` que se describirá más adelante.

 El código original se ha mantenido comentado y ofrece la salida reconocida por el corrector SIETTE; por lo que **para hacer uso de esa herramienta, se debe comentar el código modificado y usar el original.**


## Variables de instancia

```
// Datos del estado del tablero  
protected static final int MAXVALOR = 9; // Valor máximo que puede tener una celda  
protected static final int FILAS    = 9; // Número de filas del tablero  
protected static final int COLUMNAS = 9; // Número de columnas del tablero  
protected static final int raizF    = 3; // Número de filas de un subtablero  
protected static final int raizC    = 3; // Número de columnas de un subtablero  
  
// Contenido del tablero  
protected int[][] celdas;
```

Estas variables definen el tipo de sudoku con el que se va a trabajar en la práctica, indicado tanto en la descripción del apartado anterior como en el enunciado:

- Los números a colocar son del 1 a 9, por lo que el máximo valor en una celda es 9.
- El tablero mide 9x9, por lo que el número de filas y columnas es 9 cada uno.
  - Por eso, los subtableros medirán 3x3, siendo las filas y columnas de estos 3 cada uno.
- Por último, se establece que el contenido del tablero está definido en un array de celdas.

## Constructores

 Estos métodos ya venían definidos en los archivos necesarios para realizar la práctica.

```
public TableroSudoku() {  
    celdas = new int[FILAS][COLUMNAS];  
}  
  
public TableroSudoku(TableroSudoku tablero) {  
    TableroSudoku otro = (TableroSudoku) tablero.clone();  
    this.celdas = otro.celdas;  
}  
  
public TableroSudoku(String configuracion) {  
    this();  
  
    if(configuracion.length() == FILAS*COLUMNAS) {  
        for(int f = 0; f < FILAS; f++) {  
            for (int c = 0; c < COLUMNAS; c++) {  
                char ch = configuracion.charAt(f*FILAS + c);  
                celdas[f][c] = (Character.isDigit(ch) ? Integer.parseInt(Character.toString(ch)) : 0);  
            }  
        }  
    }  
    } else {
```

```
        throw new RuntimeException("Construcción de sudoku no válida.");
    }
}
```

El **primero** crea un sudoku vacío, un array bidimensional 9x9 cuyas celdas contienen 0.

El **segundo** copia un tablero y lo almacena como un objeto diferente (esto se usará más adelante en el proceso de resolución, para conservar los estados).

El **tercero** crea un sudoku vacío e inmediatamente lo rellena en base a una configuración presentada como una cadena de caracteres.

## Métodos auxiliares

### Implementados en la plantilla



Estos métodos ya venían definidos en los archivos necesarios para realizar la práctica.

```
public List<TableroSudoku> resolverTodos() {
    List<TableroSudoku> soluciones = new LinkedList<>();
    resolverTodos(soluciones, 0, 0);

    return soluciones;
}
```

Como se indica en la práctica, se buscarán **todas las soluciones posibles** a los sudokus planteados **aplicando vuelta atrás**.

Por tanto, se empezará la ejecución desde la primera celda ( `celdas[0][0]` ) intentando colocar un valor en ella, y se irá avanzando de izquierda a derecha en la fila, y de arriba a abajo en las columnas.

```
protected boolean estaLibre(int fila, int columna) {
    return celdas[fila][columna] == 0;
}
```



Indica si una celda está vacía (según el enunciado, cuando esta **contiene un 0**).

```
protected int numeroDeLibres() {
    int libres = 0;

    // Recorrer el tablero contando las celdas con 0.
    for (int f = 0; f < FILAS; f++) {
        for (int c = 0; c < COLUMNAS; c++) {
            if (estaLibre(f, c)) {
                libres++;
            }
        }
    }

    return libres;
}
```



Indica el número de celdas vacías en un tablero.

```
protected int numeroDeFijos() {
    return FILAS*COLUMNAS - numeroDeLibres();
}
```

```
}
```



Indica el número de celdas con un valor en el tablero.

### Implementados por mí

```
protected void resolverTodos(List<TableroSudoku> soluciones, int fila, int columna) {
    // Comprueba si se encontró una solución.
    if (numeroDeLibres() == 0) {
        // Se ha encontrado una solución y se añade a la lista.
        soluciones.add(new TableroSudoku(this));

    } else {
        // No se ha encontrado una solución.
        int sigFil = fila;
        int sigCol = (columna + 1) % COLUMNAS;

        // Se comprueba si se ha llegado al final de la fila.
        if (sigCol == 0) {
            sigFil++;
        }

        // Se comprueba si se ha llegado al final del tablero.
        if (!estaLibre(fila, columna)){
            resolverTodos(soluciones, sigFil, sigCol);

        } else {
            // Se intenta introducir un valor en la celda.
            for (int valor = 1; valor <= MAXVALOR; valor++) {
                if (sePuedePonerEn(fila, columna, valor)) {
                    celdas[fila][columna] = valor;
                    resolverTodos(soluciones, sigFil, sigCol);
                    celdas[fila][columna] = 0;
                }
            }
        }
    }
}
```



Este método recursivo se encarga de **colocar valores en las celdas libres hasta obtener un tablero completamente lleno**, considerándose válido.



Cabe destacar que **las celdas solo se llenan si es posible introducir un valor**: por lo que si **un tablero no tiene celdas libres**, esto quiere decir que todos los valores están bien colocados y por tanto, **es una solución**.

Si **el tablero no tiene ninguna celda libre**, entonces se almacena **una solución** usando el constructor descrito en el apartado Constructores, que almacena una copia de ese tablero.



Si se usara `soluciones.add(this)` no almacenaría el estado correctamente, ya que se modificaría el tablero en las siguientes iteraciones de *vuelta atrás*.

Si **el tablero tiene celdas libres**, entonces se calculan la fila y columna de la **siguiente iteración**; acto seguido **se comprueba si la celda actual está libre**.

Si **está ocupada**, se ejecuta el método para la **siguiente celda**.

En **caso contrario**, se generan nuevos tableros para todos los posibles valores que pueden colocarse en esa celda y se ejecuta el método para la **siguiente**.



Aquí se vacía la celda tras la llamada al método, porque si no la condición `sePuedePonerEn()` sería `false` tras la primera posibilidad y no se generarían todas las soluciones.

```
protected boolean estaEnFila(int fila, int valor) {
    // Recorre toda la fila buscando el valor
    for (int c = 0; c < COLUMNAS; c++) {
        if (celdas[fila][c] == valor) {
            return true;
        }
    }

    return false;
}
```



Indica si **un valor ya pertenece a la fila**, recorriendo todas las celdas de dicha fila.

```
protected boolean estaEnColumna(int columna, int valor) {
    // Recorre toda la columna buscando el valor
    for (int f = 0; f < FILAS; f++) {
        if (celdas[f][columna] == valor) {
            return true;
        }
    }

    return false;
}
```



Indica si **un valor ya pertenece a la columna**, recorriendo todas las celdas de dicha columna.

```
protected boolean estaEnSubtablero(int fila, int columna, int valor) {
    int subFil = fila/raizF;
    int subCol = columna/raizC;

    // Recorre el subtablero buscando el valor.
    for (int f = 0; f < raizF; f++) {
        for (int c = 0; c < raizC; c++) {
            if (celdas[subFil*raizF + f][subCol*raizC + c] == valor) {
                return true;
            }
        }
    }

    return false;
}
```



Indica **si un valor pertenece al subtablero** al que pertenece las coordenadas indicadas.

Las coordenadas vienen indicadas por la fila y columna recibidas por parámetro, y las celdas pertenecientes al subtablero se calculan usando las variables de instancia `raizF` y `raizC`. No obstante, para esta implementación dichas variables **siempre valen 3**.

- Al usar la división entera, el valor se redondea a su parte entera en todas las ocasiones. Esto permite identificar los valores de las filas y columnas ( `subFil` y `subCol` ) de la siguiente forma:

$$\begin{array}{ccc} \underbrace{0 \quad 1 \quad 2}_0 & \underbrace{3 \quad 4 \quad 5}_1 & \underbrace{6 \quad 7 \quad 8}_2 \end{array}$$

- Siendo en los bucles las variables `f` y `c`, dichos valores. Es decir:  $f, c \in \{0, 1, 2\}$ .

- Por último, las posiciones de las celdas se calculan multiplicando la coordenada obtenida en el primer paso por 3, y añadiéndole el valor de **f** o **c** para que quede encerrada en el subtablero correspondiente.

```
// Ejemplo para 'celdas[1][3]'
int subFil = 0; porque 0/3 = 0
int subCol = 1; porque 3/3 = 0

// La celda se calcula como 'celdas[subFil * 3 + f][subCol * 3 + c]',
// donde f,c pertenecen a {0, 1, 2} siempre.
celdas[0 * 3 + (0)][1 * 3 + (0)] = celdas[0][3] // Esquina sup-izq del subtablero
celdas[0 * 3 + (0)][1 * 3 + (1)] = celdas[0][4]
. . . . . = . . . . .

celdas[0 * 3 + (2)][1 * 3 + (1)] = celdas[2][4]
celdas[0 * 3 + (2)][1 * 3 + (2)] = celdas[2][5] // Esquina inf-der del subtablero
```

```
protected boolean sePuedePonerEn(int fila, int columna, int valor) {
    return estaLibre(fila, columna)
        && !(estaEnFila(fila, valor)
            || estaEnColumna(columna, valor)
            || estaEnSubtablero(fila, columna, valor));
}
```



Indica si **un valor puede colocarse en una celda**, atendiendo a las restricciones definidas por los métodos anteriores.

Un valor puede colocarse si:

- La celda está libre.
- No está el valor en la fila, ni está el valor en la columna, ni está el valor en el subtablero.

```
public String tablero() {
    StringBuilder tablero = new StringBuilder();

    for (int f = 0; f < FILAS; f++) {
        for (int c = 0; c < COLUMNAS; c++) {

            // Mostrar el valor de una celda
            if (celdas[f][c] != 0) {
                tablero.append(celdas[f][c]);
            } else {
                tablero.append(" "); // Opción por defecto de celda vacía
                // tablero.append("."); // Opción alternativa de celda vacía
            }

            tablero.append(" "); // Espacio entre columnas

            // Separación de vertical de subtableros
            if (c == 2 || c == 5) {
                tablero.append("| ");
            }
        }

        // Separación horizontal de subtableros
        if (f == 2 || f == 5) {
            tablero.append("\n-----+-----+-----");
        }

        tablero.append("\n"); // Salto de línea entre filas
    }

    return tablero.toString();
}
```



Este método se creó para mejorar el proceso de debug, generando una representación más visual de un tablero de sudoku.



Este es el método al que hace referencia la modificación de `main()` mencionada al inicio.

No influye en nada respecto al desarrollo de la práctica, la única función que tiene es la de representar los tableros de la siguiente forma:

```
Sudoku sin resolver:
 4   |   | 3 6 2
6 3   | 9 4 1 |
5  7 |   3   |
-----+-----+-----
  9   | 3 7 5 | 1
3   4 | 8     |
1 7   |   6 2 |
-----+-----+-----
7 1 6 |   9   | 2
    9 | 6     |
    3 | 1 2   | 9
```

```
Sudoku resuelto:
9 4 1 | 7 5 8 | 3 6 2
6 3 2 | 9 4 1 | 5 8 7
5 8 7 | 2 3 6 | 4 1 9
-----+-----+-----
2 9 8 | 3 7 5 | 1 4 6
3 6 4 | 8 1 9 | 2 7 5
1 7 5 | 4 6 2 | 9 3 8
-----+-----+-----
7 1 6 | 5 9 4 | 8 2 3
4 2 9 | 6 8 3 | 7 5 1
8 5 3 | 1 2 7 | 6 9 4
```

## Métodos sobrescritos



Estos métodos ya venían definidos en los archivos necesarios para realizar la práctica.

```
@Override
public Object clone() {
    TableroSudoku clon;

    try {
        clon = (TableroSudoku) super.clone();
        clon.celdas = new int[FILAS][COLUMNAS];

        for(int i = 0; i < celdas.length; i++) {
            System.arraycopy(celdas[i], 0, clon.celdas[i], 0, celdas[i].length);
        }

    } catch (CloneNotSupportedException e) {
        clon = null;
    }

    return clon;
}
```



### Realiza una copia (clon) de un tablero.

Una copia de un sudoku es definida como otro sudoku donde el contenido de sus celdas es el mismo que el del sudoku original.

```
@Override
public boolean equals(Object objeto) {
    // Comprobar que el objeto recibido es un TableroSudoku
    if (objeto instanceof TableroSudoku) {
        TableroSudoku otro = (TableroSudoku) objeto; // Esto no se puede suprimir por cula de SIETTE

        // Comprobar que las filas de ambos tableros son iguales
        for(int f = 0; f < FILAS; f++) {
            if (!Arrays.equals(this.celdas[f], otro.celdas[f])) {
```

```
        return false;
    }
}

return true;
} else {
    return false;
}
}
```



### Verifica que un sudoku es igual que el sudoku actual.

Dos sudokus son iguales si el contenido de sus celdas son iguales.

```
@Override
public String toString() {
    StringBuilder s = new StringBuilder();

    for(int f = 0; f < FILAS; f++) {
        for(int c = 0; c < COLUMNAS; c++) {
            s.append(celdas[f][c] == 0 ? "." : String.format("%d", celdas[f][c]));
        }
    }

    return s.toString();
}
```



### Representa el estado del tablero de un sudoku como una cadena de caracteres.

Una representación de un sudoku se define como indica el enunciado: una lista de caracteres que representa el contenido de cada celda de un sudoku.

- Colocando el `valor` de la celda o `.` si la celda está vacía.
- Recorriendo el tablero de izquierda a derecha y de arriba a abajo.



Esta representación actúa como configuración del sudoku, por lo que puede usarse un constructor para crear un tablero que contenga dicha información.



Como esta representación no es muy natural, se creó el método `tablero()` que hace lo mismo, pero muestra el tablero como una tabla.