```
//*************************************************************
// File:   SymbolTable.java
// Author: Procesadores de Lenguajes-University of Zaragoza
// Date:   julio 2021
// Coms:   Atributos públicos para ahorrarnos el uso de getters y setters
//*************************************************************

//la tabla de símbolos será un ArrayList de diccionarios (HashMap<String, Symbol>), manejada como
//una pila: se inserta y accede por la derecha
//Cada nuevo bloque se apilará, guardando los símbolos en el diccionario correspondiente
//El constructor ya genera el primer bloque, vacío inicialmente.

//https://quick-adviser.com/can-a-hashmap-have-multiple-values-for-same-key/
//HashMap doesn’t allow duplicate keys but allows duplicate values. That means A
//single key can’t contain more than 1 value but more than 1 key can contain a single value.
//HashMap allows null key also but only once and multiple null values.
//https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

package lib.symbolTable;

import java.util.*;
import lib.symbolTable.exceptions.SymbolNotFoundException;
import lib.symbolTable.exceptions.AlreadyDefinedSymbolException;

public class SymbolTable {
private final int ST_SIZE = 16; // hasta 16 niveles
private final int HASH_SIZE = 1023; // buckets
private ArrayList<HashMap<String, Symbol>> st;

public int level; // nivel actual
public static int[] direccionPorNivel = new int[16]; // Su indice representa el nivel

// Bloque estático de inicialización
static {
// Inicialización de todas las componentes al valor 3
for (int i = 0; i < direccionPorNivel.length; i++) {
direccionPorNivel[i] = 3; // Todas las direcciones empiezan en 3, debido a los punteros que se guardan a
// otras estructuras
}
}

public SymbolTable() {
st = new ArrayList<HashMap<String, Symbol>>(ST_SIZE);
level = -1; // aún no hay ningún bloque intoducido
insertBlock();
}

// apila un nuevo bloque
public void insertBlock() {
st.add(new HashMap<String, Symbol>(HASH_SIZE));
level++;
}

// elimina un bloque
public void removeBlock() {
st.remove(st.size() - 1);
level--;
}

// Si un símbolo con el mismo nombre está, excepción.
```

```
// Si no, se inserta
public void insertSymbol(Symbol s) throws AlreadyDefinedSymbolException {
HashMap<String, Symbol> currentBlock = st.get(st.size() - 1);
if (currentBlock.containsKey(s.name)) { // ya está
throw new AlreadyDefinedSymbolException();
} else {
s.nivel = level;
if (!(s instanceof SymbolFunction || s instanceof SymbolProcedure)) {
// No se guardan direcciones del id de procedimientos o funciones

// Ids de variables de tipo entero, char, etc...
s.dir = direccionPorNivel[level]; // Asignamos la dirección que le corresponde
direccionPorNivel[level] = direccionPorNivel[level] + 1; // Aumentamos la dirección para la
// siguiente
// variable a almacenar
}
currentBlock.put(s.name, s);
}
}

// Si no está, excepción. Si está, devuelve su referencia
public Symbol getSymbol(String name) throws SymbolNotFoundException {
Symbol result = findSymbol(name);
if (result == null) {
throw new SymbolNotFoundException();
}
return result;
}

// comprueba si está el símbolo
public boolean containsSymbol(String name) {
return findSymbol(name) != null;
}

// para usar en "getSymbol" y "containsSymbol"
private Symbol findSymbol(String name) {
for (int i = st.size() - 1; i >= 0; i--) {
if (st.get(i).containsKey(name)) {
return st.get(i).get(name);
}
}
return null;
}

// devuelve la tabla como un string
public String toString() {
final String linea = "------------------------------------------------\n";
StringBuilder builder = new StringBuilder();
builder.append(linea);
String tabs = "";
for (int i = 0; i < st.size(); i++) {
for (Map.Entry entry : st.get(i).entrySet()) {
// crear secuencia de tabuladores
tabs = new String(new char[i]).replace("\0", "\t");
builder.append(tabs);
builder.append(entry.toString());
builder.append("\n");
}
}
builder.append(linea);
```

```
return builder.toString();
}
}
```