Singleton

Application à JDBC

I Initialisation dynamique et Design Pattern Singleton

```
package singleton1;


/**
 *  Les méthodes statiques sont souvent utilisés pour assurer l'unicité d'un objet
 *  Design Pattern : singleton
 */

public class Platform {
	private static final String NAME_PER_DEFAULT = "Java 8";

	// l'unique plate-forme
	private static Platform platform = null;

	private String name;

	// Constructeur privé afin qu'un code client n'instancie pas plusieurs fois
	private Platform(String name){
		this.name = name;
	}

	// retourne l'unique instance par défaut
	public static Platform getDefaultPlatform() {
		if (platform == null){
			platform = new Platform(NAME_PER_DEFAULT);
		}
		return platform;
	}

	@Override
	public String toString() {
		return "Platform [name=" + name + "]";
	}
}
```

```
package singleton1;


public class PlatformTest {

	public static final void main(String [] args){
		Platform platform1 = Platform.getDefaultPlatform();
```

```
                System.out.println(platform1);
                Platform platform2 = Platform.getDefaultPlatform();
                System.out.println(platform2);

                if (platform1 == platform2){
                        System.out.println("EGALITE DE
                         REFERENCES ==> UNIQUE INSTANCE!");
                }
                // NE COMPILE PAS : constructeur privé
                // Platform platform2 = new Platform("Java 5");
        }
}
```

II Initialisation statique et Design Pattern Singleton

```
package singleton2;


/**
 *  Les méthodes statiques sont souvent utilisés pour assurer l'unicité d'un objet
 *  Design Pattern : singleton
 */

public class Platform {
        private static final String NAME_PER_DEFAULT = "Java 8";

        // initialisation statique au chargement de la classe
        // ==> une seule instance
        private static Platform platform = new Platform(NAME_PER_DEFAULT);

        private String name;

        // Constructeur privé afin qu'un code client n'instancie pas plusieurs fois
        private Platform(String name){
                this.name = name;
        }

        // retourne l'unique instance par défaut
        public static Platform getDefaultPlatform() {
                return platform;
        }

        @Override
        public String toString() {
                return "Platform [name=" + name + "]";
        }
}
```

III JDBC et Design Pattern Singleton

Les paramètres sous forme d'une interface (on pourrait aussi utiliser un fichier de propriétés).

```
package coffees;


public interface IDatabase {
	public static final String SQL_DRIVER = "com.mysql.jdbc.Driver";
	public static final String SQL_CONNEXION = "jdbc:mysql://localhost/COFFEEBREAK?
user=root&password=";
	public static final String DATABASE_URL = "jdbc:mysql://localhost/COFFEESBREAK";
	public static final String USERNAME = "root";
	public static final String PASSWORD = "";

	// la requête par défaut qui retrouvent toutes les entrées de la table NOM_CAFE
	public static final String DEFAULT_QUERY = "SELECT * FROM NOM_CAFE";
}
```

```
package coffees;


import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

// Classe basée sur le pattern Singleton qui permet :
// 1) la création d'une connexion unique à la base
// 2) la création d'une table unique; afin de ne pas la recréer
public class CreationTable
{
	// unique instance de la connexion
	private static Connection uniqueConnexion = null;
	// Stockage de l'unique instance de cette classe.
	private static CreationTable uniqueTable = null;

	public static String sql_dropTable = null;
	public static String sql_createTable = null;

	public static void loadDriver()
	{
		try
		{
			Class.forName(IDatabase.SQL_DRIVER);
			System.err.println("Succès chargement du driver") ;
		}
		catch(ClassNotFoundException e)
		{
			System.err.println("Erreur de chargement du driver :" + e) ;
		}
	}

	// DESIGN PATTERN SINGLETON
	public static synchronized Connection getConnexion()
	{
		if(uniqueConnexion  == null)
		{
```

```
                System.out.println("DESIGN PATTERN SINGLETON");
                try
                {
            uniqueConnexion = DriverManager.getConnection(IDatabase.SQL_CONNEXION);
                        System.err.println("Succès connexion à la base") ;
                } catch (SQLException ex)
                {
                        // handle any errors
                        System.err.println("SQLException: " + ex.getMessage());
                        System.err.println("SQLState: " + ex.getSQLState());
                        System.err.println("VendorError: " + ex.getErrorCode());
                }
        }
        return uniqueConnexion;
}

// Constructeur en privé (donc inaccessible à l'extérieur de la classe).
// à executer une seule fois
private CreationTable()
{
        try
        {
                Statement stmt = null;
                stmt = uniqueConnexion.createStatement();
                stmt.executeUpdate(sql_dropTable);
                stmt.executeUpdate(sql_createTable);
        }
        catch (SQLException ex)
        {
                // handle any errors
                System.err.println("SQLException: " + ex.getMessage());
        }

}

// DESIGN PATTERN SINGLETON
// Méthode statique qui sert de pseudo-constructeur
// utilisation du mot clef "synchronized" pour le multithread).
public static synchronized CreationTable getTable()
{
        if(uniqueTable == null)
        {
                uniqueTable = new CreationTable();
                System.err.println("Table créee : " + sql_createTable);
        }
        return uniqueTable;
}



public void insertEntree (String sql_insertTable)
{
        try
        {
                Statement stmt = null;
                stmt = uniqueConnexion.createStatement();
                stmt.executeUpdate(sql_insertTable);

                System.err.println(sql_insertTable);
        }
        catch (SQLException ex)
```

```
            {
                  // handle any errors
                  System.err.println("SQLException: " + ex.getMessage());
            }
      }

      public static void closeAll()
      {
            if (uniqueConnexion != null) {
                  try {
                        uniqueConnexion.close();
                  } catch (SQLException sqlEx) { } // ignore
            uniqueConnexion = null;
            }
      }
}
```