H2 Assignment - 2

Implement an enhanced feature to H2 database in terms of data storage and indexing.

Feature :

Implementation of a new CONTACT data type

Overview:

- ☐ The H2 database does not currently support a data type of the Contact format. In this assignment, we have integrated a new data type of the Contact format which takes a 10 digit numeric string as a contact number and stores it as a data type in the back end.

- ☐ The main motive behind choosing this data type is that most often than not, we take contact as an input in a number of forms and it becomes important to guarantee that a contact number entered by a user is a valid contact number indeed.

- ☐ This can be checked by multiple conditions, one of which includes ensuring that the number entered is exactly 10 digits - no more, no less.

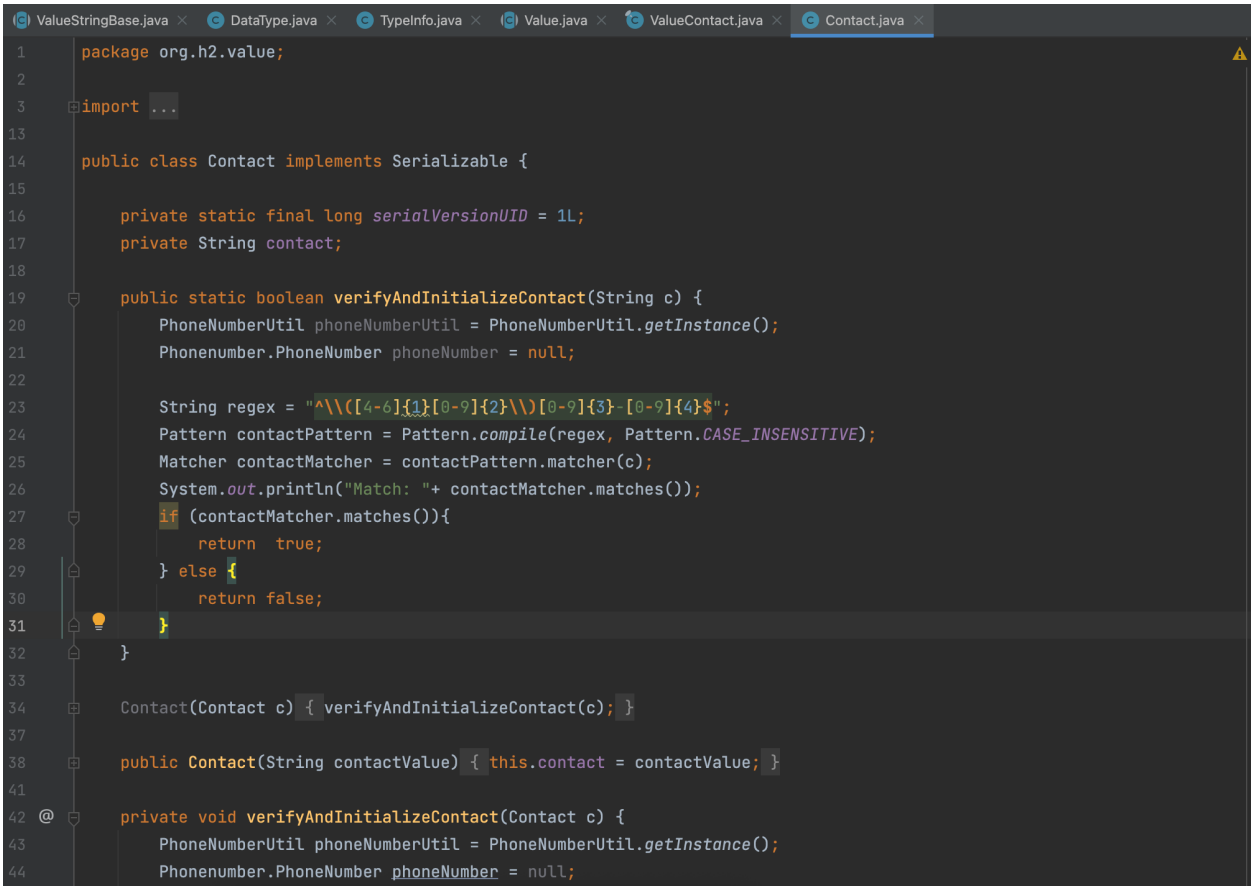
- ☐ Implementing a custom data type for this would avoid any human errors and would make storing and managing contacts in the database much easier and efficient.

- ☐ A series of snapshots are attached below depicting how the custom data type was integrated in the H2 database and what changes were made in the original H2 code.

Implementation of the CONTACT Data Type

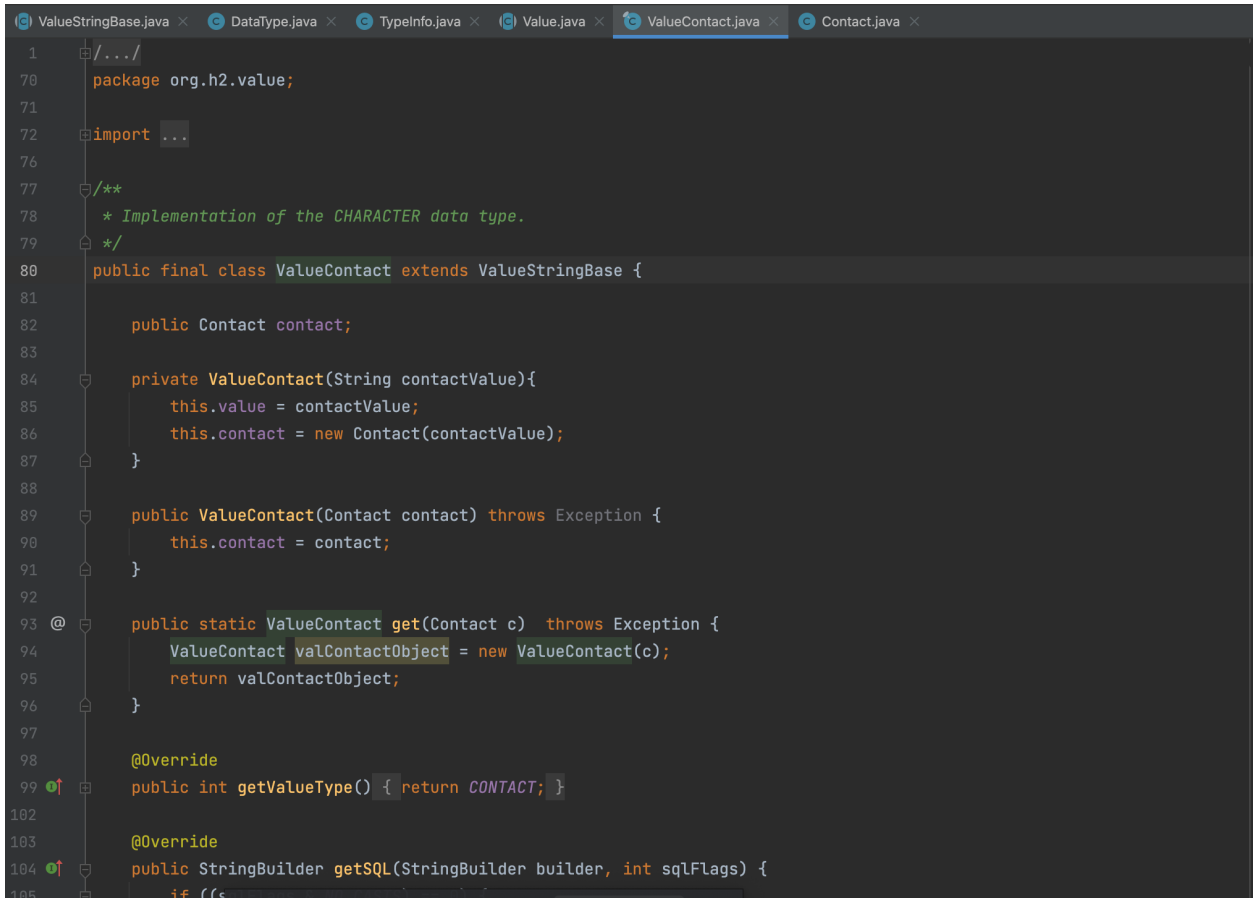
Step 1: **Contact.java** (Path: src/main/org/h2/value/Contact.java) was created.

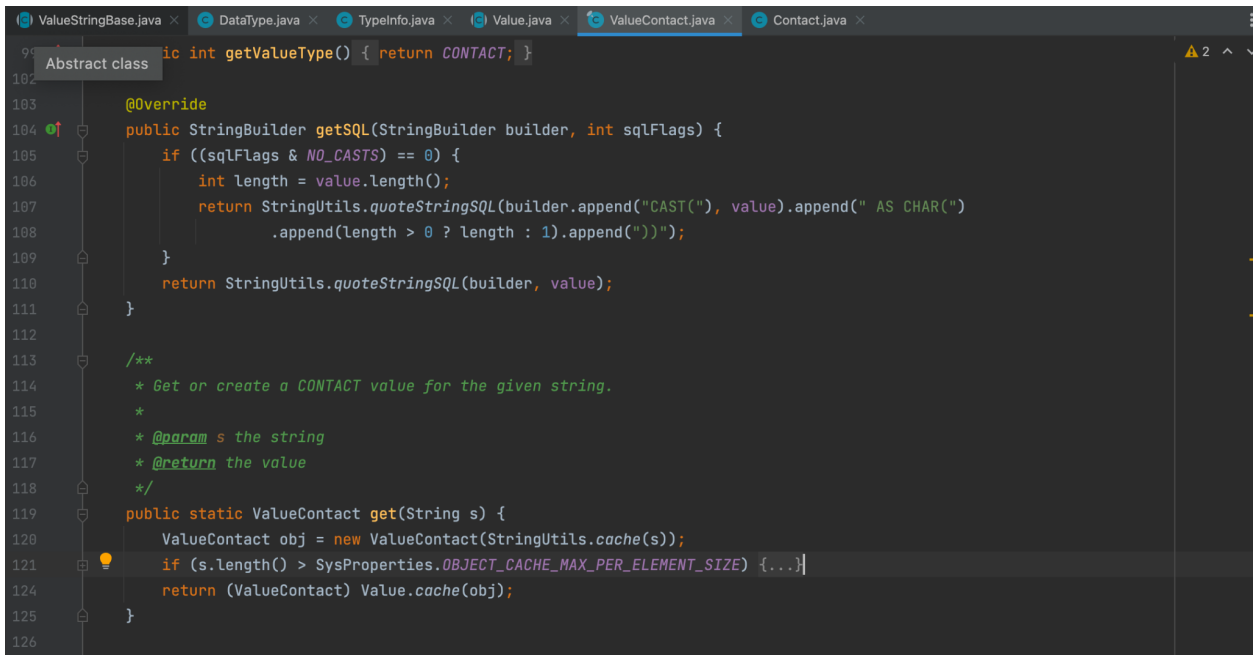
1) The aim of the file is to store the CONTACT Data Type.
2) We serialized this class and added a method to verify whether the entered contact was a valid contact or not using regular expressions (regex).
3) Contact format used in this implementation is of US format.
4) If the entered number passes the regex, we return TRUE, i.e. this data can be stored as a contact data type.
5) Otherwise, it's not possible to store this type of data and we throw an error.



Step 2: **ValueContact.java** (Path: src/main/org/h2/value/ValueContact.java)

1) This file is the implementation of the CONTACT data type.
2) This file extends ValueStringBase which is in layman language responsible for working with String data type.
3) It also implements the overridden methods of ValueStringBase Class.
4) We have defined this class as per the structure of other classes like ValueChar, ValueBoolean, ValueGeometry, etc.

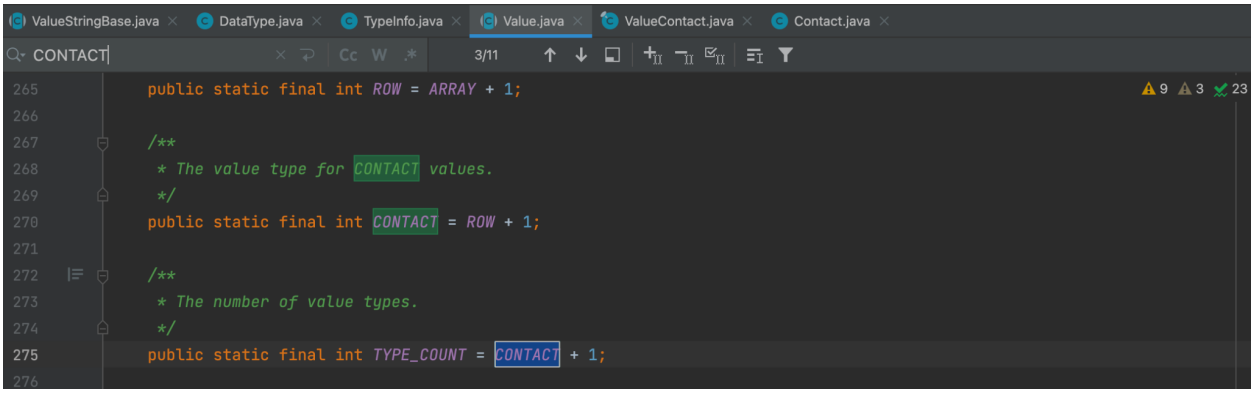

ValueStringBase.java
DataType.java
TypeInfo.java
Value.java
ValueContact.java
Contact.java
/.../
package org.h2.value;
import ...
/**
* Implementation of the CHARACTER data type.
*/
public final class ValueContact extends ValueStringBase {
public Contact contact;
private ValueContact(String contactValue){
this.value = contactValue;
this.contact = new Contact(contactValue);
}
public ValueContact(Contact contact) throws Exception {
this.contact = contact;
}
public static ValueContact get(Contact c) throws Exception {
ValueContact valContactObject = new ValueContact(c);
return valContactObject;
}
@Override
public int getValueType() { return CONTACT; }
@Override
public StringBuilder getSQL(StringBuilder builder, int sqlFlags) {


ValueStringBase.java
DataType.java
TypeInfo.java
Value.java
ValueContact.java
Contact.java
Abstract class
ic int getValueType() { return CONTACT; }
@Override
public StringBuilder getSQL(StringBuilder builder, int sqlFlags) {
if ((sqlFlags & NO_CASTS) == 0) {
int length = value.length();
return StringUtils.quoteStringSQL(builder.append("CAST("), value).append(" AS CHAR(")
.append(length > 0 ? length : 1).append("))");
}
return StringUtils.quoteStringSQL(builder, value);
}
/**
* Get or create a CONTACT value for the given string.
*
* @param s the string
* @return the value
*/
public static ValueContact get(String s) {
ValueContact obj = new ValueContact(StringUtils.cache(s));
if (s.length() > SysProperties.OBJECT_CACHE_MAX_PER_ELEMENT_SIZE) {...}
return (ValueContact) Value.cache(obj);
}

We will now register the CONTACT data type in H2.

Step 3: **Value.java** (Path: src/main/org/h2/value/Value.java)

1) This class is solely responsible for defining all the valid data types in the H2 database and how to work with them like comparing and converting them into the required data types.
2) Below is a screenshot where we define a ValueType for CONTACT data type.



3) Below is a screenshot where we declare all the valid data types in the H2 database which includes CONTACT.

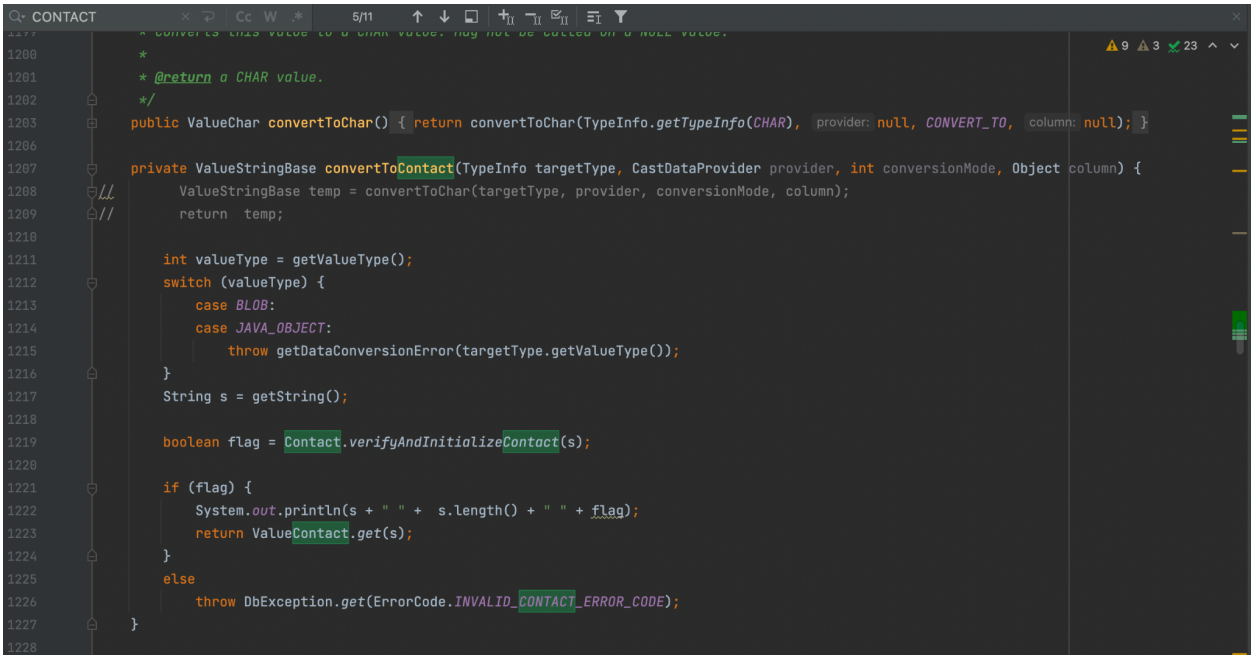
```
359
360     private static final String NAMES[] = {
361             "UNKNOWN",
362             "NULL", //
363             "CHARACTER", "CHARACTER VARYING", "CHARACTER LARGE OBJECT", "VARCHAR_IGNORECASE", //
364             "BINARY", "BINARY VARYING", "BINARY LARGE OBJECT", //
365             "BOOLEAN", //
366             "TINYINT", "SMALLINT", "INTEGER", "BIGINT", //
367             "NUMERIC", "REAL", "DOUBLE PRECISION", "DECFLOAT", //
368             "DATE", "TIME", "TIME WITH TIME ZONE", "TIMESTAMP", "TIMESTAMP WITH TIME ZONE", //
369             "INTERVAL YEAR", "INTERVAL MONTH", //
370             "INTERVAL DAY", "INTERVAL HOUR", "INTERVAL MINUTE", "INTERVAL SECOND", //
371             "INTERVAL YEAR TO MONTH", //
372             "INTERVAL DAY TO HOUR", "INTERVAL DAY TO MINUTE", "INTERVAL DAY TO SECOND", //
373             "INTERVAL HOUR TO MINUTE", "INTERVAL HOUR TO SECOND", "INTERVAL MINUTE TO SECOND", //
374             "JAVA_OBJECT", "ENUM", "GEOMETRY", "JSON", "UUID", //
375             "ARRAY", "ROW", "CONTACT"//
376     };
377
```

4) The following two screenshots demonstrate what the program should execute if it encounters a CONTACT data type. This functionality is defined in our program with the help of convertToContact() function.

```
1100
1101     /**
1102      * Convert a value to the specified type.
1103      *
1104      * @param targetType the type of the returned value
1105      * @param provider the cast information provider
1106      * @param conversionMode conversion mode
1107      * @param column the column (if any), used to improve the error message if conversion fails
1108      * @return the converted value
1109      */
1110     private Value convertTo(TypeInfo targetType, CastDataProvider provider, int conversionMode, Object column) {
1111         int valueType = getValueType(), targetValueType;
1112         if (valueType == NULL
1113                 || valueType == (targetValueType = targetType.getValueType()) && conversionMode == CONVERT_TO
1114                 && targetType.getExtTypeInfo() == null && valueType != CHAR) {
1115             return this;
1116         }
1117         switch (targetValueType) {
1118         case NULL:
```

```
        return convertToRow(targetType, provider, conversionMode, column);
    case CONTACT:
        return convertToContact(targetType, provider, conversionMode, column);
    default:
        throw getDataConversionError(targetValueType);
```

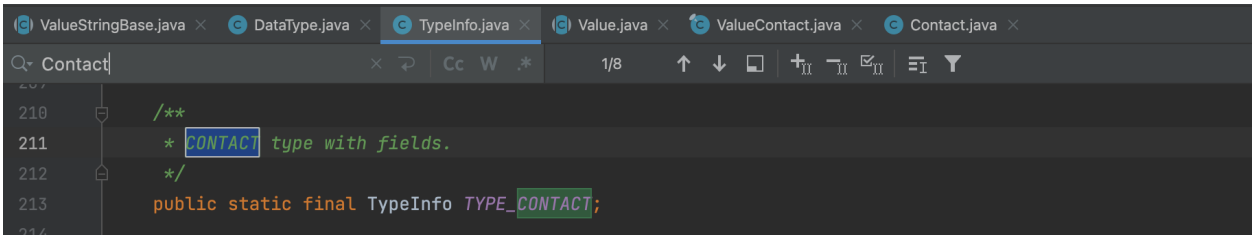
1) Below screenshot shows the implementation of convertToContact() where we also verify whether the entered CONTACT data type is a valid entry or not using verifyAndInitializeContact() defined in our Contact.java class.
2) If it is a valid Contact, we proceed, else, we throw a DBExceptionError with the respective contact error code.



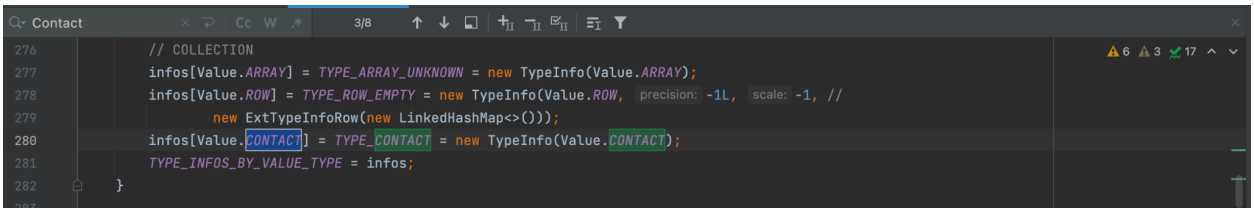
Step 4: **TypeInfo.java** (Path: src/main/org/h2/value/TypeInfo.java)

1) This file holds data types with parameters.

2) Below two screenshots show where we defined a type TYPE_CONTACT for the CONTACT data type and store it.

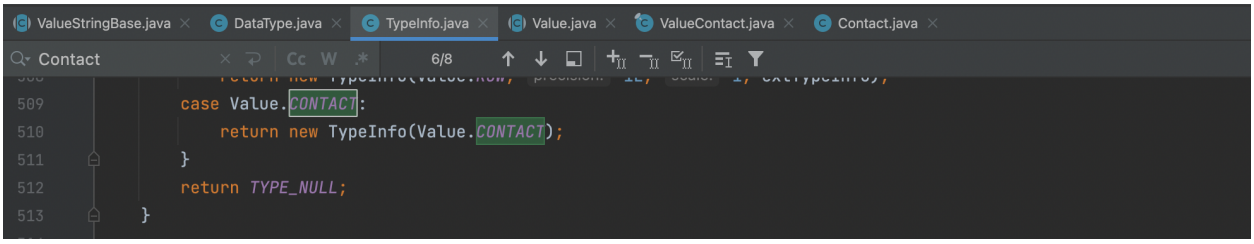


Insert in “static” call in the class:-



3) The function getTypeInfo() returns the data type with respect to the corresponding data type that we encounter.

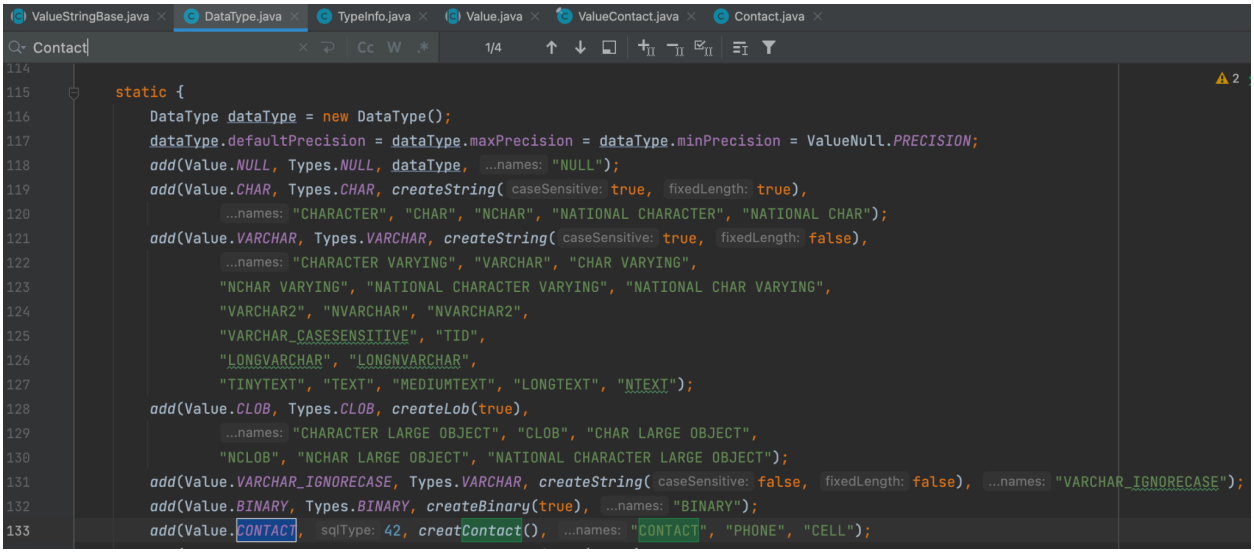
```java
/**
 * Get the data type with parameters object for the given value type and the
 * specified parameters.
 *
 * @param type
 *            the value type
 * @param precision
 *            the precision or {@code -1L} for default
 * @param scale
 *            the scale or {@code -1} for default
 * @param extTypeInfo
 *            the extended type information or null
 * @return the data type with parameters object
 */
public static TypeInfo getTypeInfo(int type, long precision, int scale, ExtTypeInfo extTypeInfo) {
    switch (type) {
    case Value.NULL:
    case Value.BOOLEAN:
```

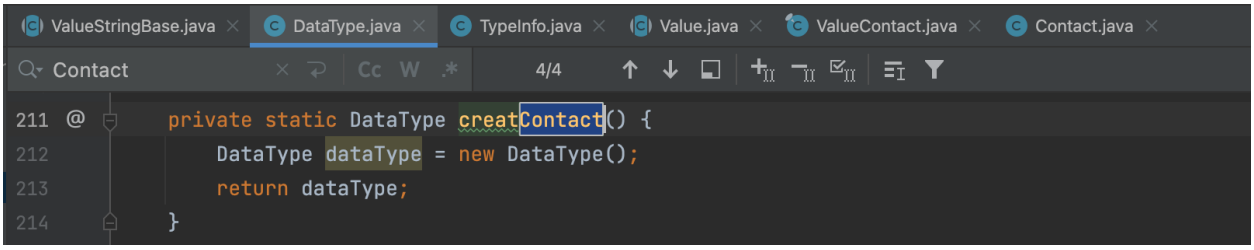


Step 5: **DataType.java** (Path: src/main/org/h2/value/DataType.java)

1) This class contains metadata information of the data types used and handles conversion between Java objects and Values.
2) In the screenshot below, the code line no. 133 is where we are registering the CONTACT data type in our system

3) This data type can then be used using multiple keywords like “CONTACT”, “PHONE” or “CELL” .

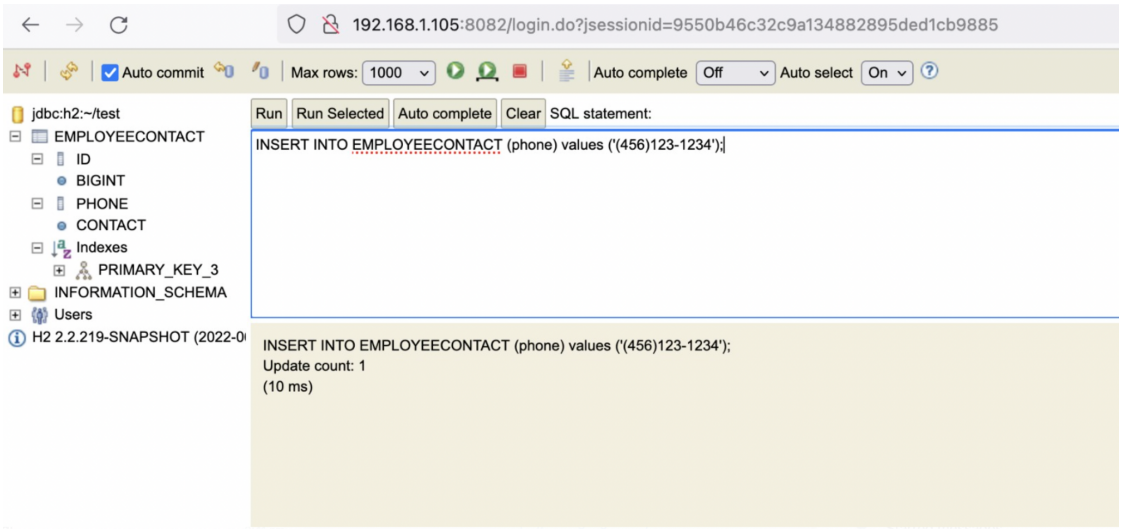




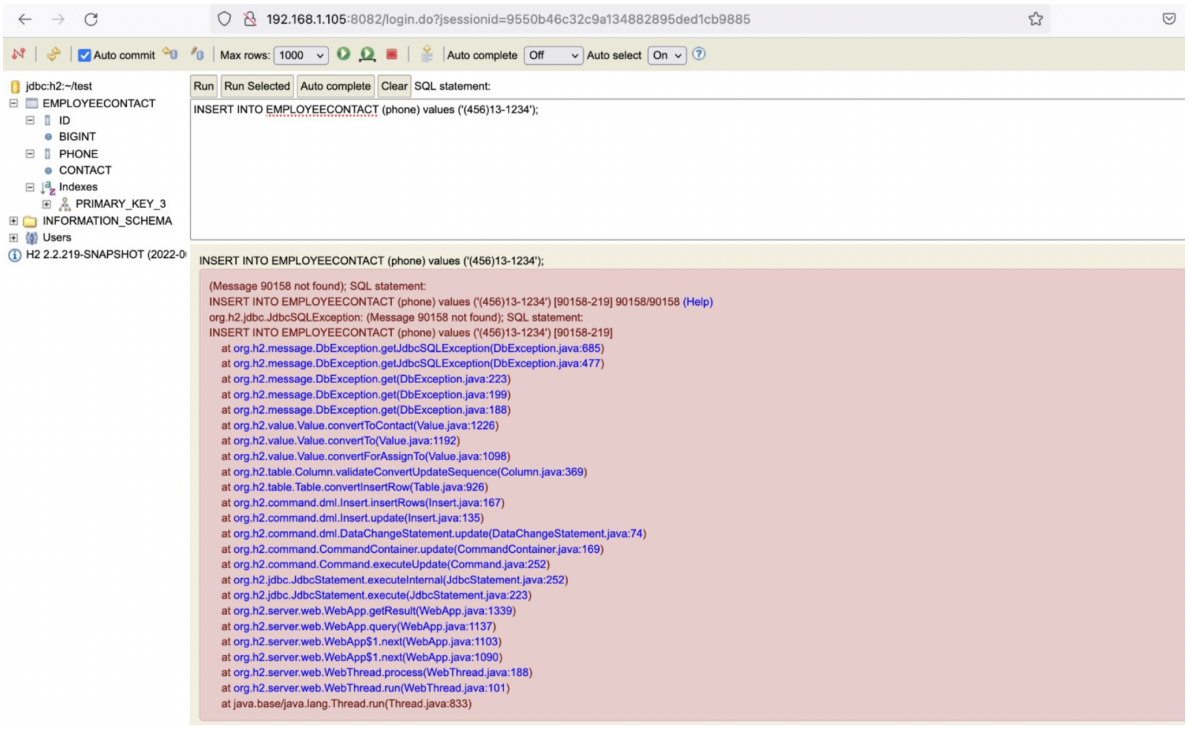
The following are the screenshots of the successful implementation of the custom data type CONTACT in the H2 database :

1) An example of using the CREATE statement in H2:

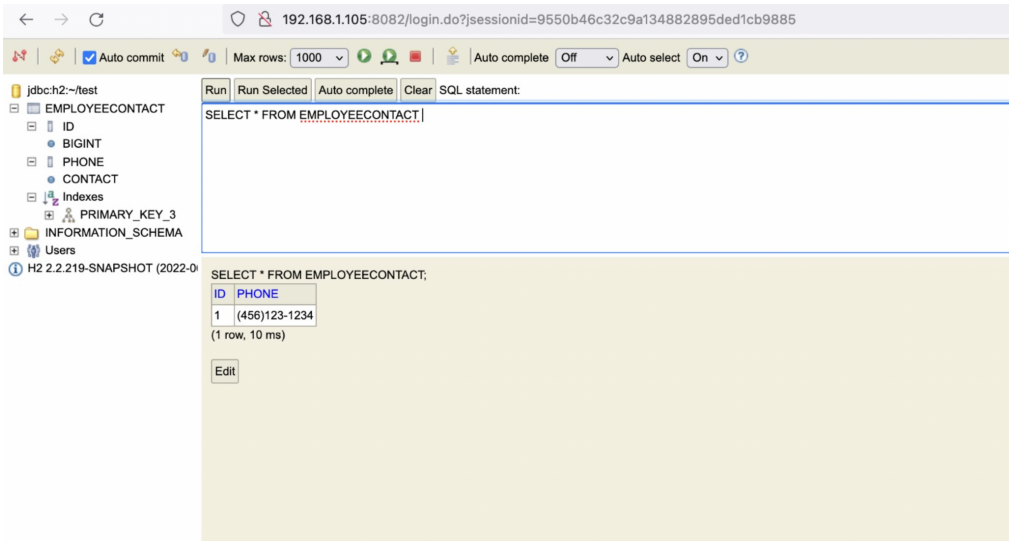
2) An example of inserting the CONTACT data type in H2:



3) An example of the exception handling performed on invalid contact input:



4) A screenshot of the successfully inserted tuple:



Authors

Janhavi Doshi (jd6583)
Kushal Kale (ksk7657)
Nimisha Bhagat (nb2633)
Vinay Jain (vj9898)