

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.

```
ValueStringBase.java × DataType.java × TypeInfo.java × Value.java × ValueContact.java × Contact.java ×
1 package org.h2.value;
2
3 import ...
4
5
6 public class Contact implements Serializable {
7
8     private static final long serialVersionUID = 1L;
9     private String contact;
10
11
12     public static boolean verifyAndInitializeContact(String c) {
13         PhoneNumberUtil phoneNumberUtil = PhoneNumberUtil.getInstance();
14         PhoneNumber phoneNumber = null;
15
16
17         String regex = "^(\\([4-6]\\{1\\})[0-9]\\{2\\})\\([0-9]\\{3\\}-[0-9]\\{4\\}$";
18         Pattern contactPattern = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
19         Matcher contactMatcher = contactPattern.matcher(c);
20         System.out.println("Match: " + contactMatcher.matches());
21
22         if (contactMatcher.matches()){
23             return true;
24         } else {
25             return false;
26         }
27     }
28
29
30     Contact(Contact c) { verifyAndInitializeContact(c); }
31
32
33     public Contact(String contactValue) { this.contact = contactValue; }
34
35
36     @Override
37     private void verifyAndInitializeContact(Contact c) {
38         PhoneNumberUtil phoneNumberUtil = PhoneNumberUtil.getInstance();
39         PhoneNumber phoneNumber = null;
40
41
42 }
```

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.

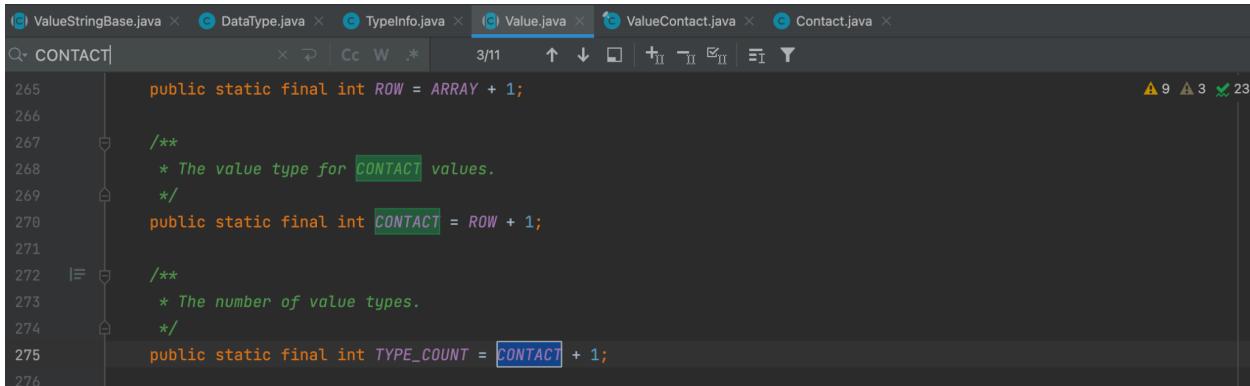
```
ValueStringBase.java × DataType.java × TypeInfo.java × Value.java × ValueContact.java × Contact.java ×
1  .../
70 package org.h2.value;
71
72 import ...
73
74 /**
75  * Implementation of the CHARACTER data type.
76 */
77
78 public final class ValueContact extends ValueStringBase {
79
80     public Contact contact;
81
82     private ValueContact(String contactValue) {
83         this.value = contactValue;
84         this.contact = new Contact(contactValue);
85     }
86
87     public ValueContact(Contact contact) throws Exception {
88         this.contact = contact;
89     }
90
91     @Override
92     public static ValueContact get(Contact c) throws Exception {
93         ValueContact valContactObject = new ValueContact(c);
94         return valContactObject;
95     }
96
97     @Override
98     public int getValueType() { return CONTACT; }
99
100    @Override
101    public StringBuilder getSQL(StringBuilder builder, int sqlFlags) {
102        if ((sqlFlags & NO_CASTS) == 0) {
103            int length = value.length();
104            return StringUtils.quoteStringSQL(builder.append("CAST("), value).append(" AS CHAR(")
105                .append(length > 0 ? length : 1).append("))");
106        }
107        return StringUtils.quoteStringSQL(builder, value);
108    }
109
110    /**
111     * Get or create a CONTACT value for the given string.
112     *
113     * @param s the string
114     * @return the value
115     */
116
117    public static ValueContact get(String s) {
118        ValueContact obj = new ValueContact(StringUtils.cache(s));
119        if (s.length() > SysProperties.OBJECT_CACHE_MAX_PER_ELEMENT_SIZE) {...}
120        return (ValueContact) Value.cache(obj);
121    }
122
123 }
```

```
ValueStringBase.java × DataType.java × TypeInfo.java × Value.java × ValueContact.java × Contact.java ×
99  Abstract class
100
101    @Override
102    public int getValueType() { return CONTACT; }
103
104    @Override
105    public StringBuilder getSQL(StringBuilder builder, int sqlFlags) {
106        if ((sqlFlags & NO_CASTS) == 0) {
107            int length = value.length();
108            return StringUtils.quoteStringSQL(builder.append("CAST("), value).append(" AS CHAR(")
109                .append(length > 0 ? length : 1).append("))");
110        }
111        return StringUtils.quoteStringSQL(builder, value);
112    }
113
114    /**
115     * Get or create a CONTACT value for the given string.
116     *
117     * @param s the string
118     * @return the value
119     */
120
121    public static ValueContact get(String s) {
122        ValueContact obj = new ValueContact(StringUtils.cache(s));
123        if (s.length() > SysProperties.OBJECT_CACHE_MAX_PER_ELEMENT_SIZE) {...}
124        return (ValueContact) Value.cache(obj);
125    }
126 }
```

We will now register the CONTACT data type in H2.

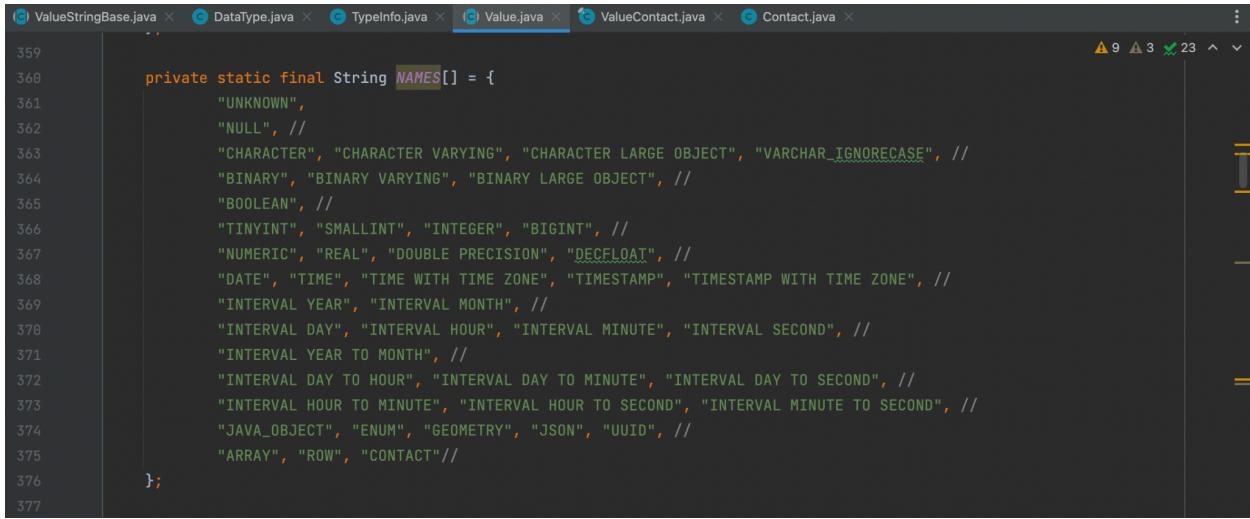
Step 3: **Value.java** (Path: [src/main/org/h2/value/Value.java](#))

- 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.



```
ValueStringBase.java × DataType.java × TypeInfo.java × Value.java × ValueContact.java × Contact.java ×
Q CONTACT| 265     public static final int ROW = ARRAY + 1;
266
267     /**
268      * The value type for CONTACT values.
269      */
270     public static final int CONTACT = ROW + 1;
271
272     /**
273      * The number of value types.
274      */
275     public static final int TYPE_COUNT = CONTACT + 1;
276
```

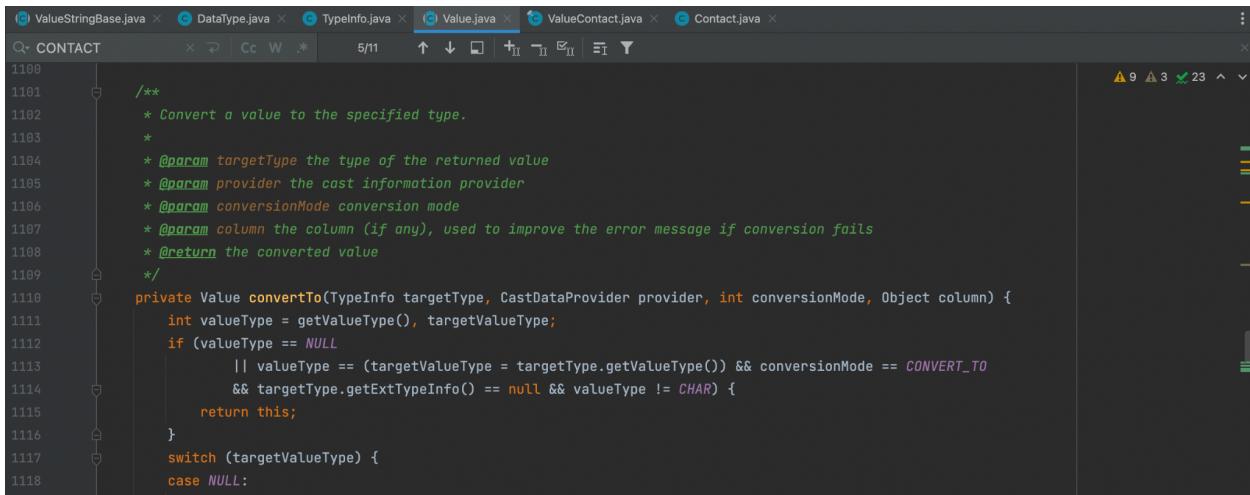
- 3) Below is a screenshot where we declare all the valid data types in the H2 database which includes CONTACT.



A screenshot of a Java code editor showing a list of database data types. The code is in the `Value.java` file, specifically in the `Value` class. The list includes: UNKNOWN, NULL, CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT, VARCHAR_IGNORECASE, BINARY, BINARY VARYING, BINARY LARGE OBJECT, BOOLEAN, TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC, REAL, DOUBLE PRECISION, DECFLOAT, DATE, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, INTERVAL YEAR, INTERVAL MONTH, INTERVAL DAY, INTERVAL HOUR, INTERVAL MINUTE, INTERVAL SECOND, INTERVAL YEAR TO MONTH, INTERVAL DAY TO HOUR, INTERVAL DAY TO MINUTE, INTERVAL DAY TO SECOND, INTERVAL HOUR TO MINUTE, INTERVAL HOUR TO SECOND, INTERVAL MINUTE TO SECOND, JAVA_OBJECT, ENUM, GEOMETRY, JSON, UUID, ARRAY, ROW, and CONTACT. The code is annotated with numerous double slashes (//) indicating comments or specific database dialects.

```
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.



A screenshot of a Java code editor showing the implementation of the `convertToContact()` function. The code is in the `Contact.java` file. The function takes four parameters: `TypeInfo targetType`, `CastDataProvider provider`, `int conversionMode`, and `Object column`. It first checks if the current value type is `NULL` and if the target value type is also `NULL`. If both are `NULL` and the conversion mode is `CONVERT_TO`, it returns the current object. Otherwise, it switches on the target value type. In the case of `NULL`, it handles the conversion logic. The code is annotated with Javadoc comments describing the parameters and the return value.

```
1100
1101     /**
1102      * Convert a value to the specified type.
1103      *
1104      * @param targetType the type of the returned value
1105      * @param provider 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);
```

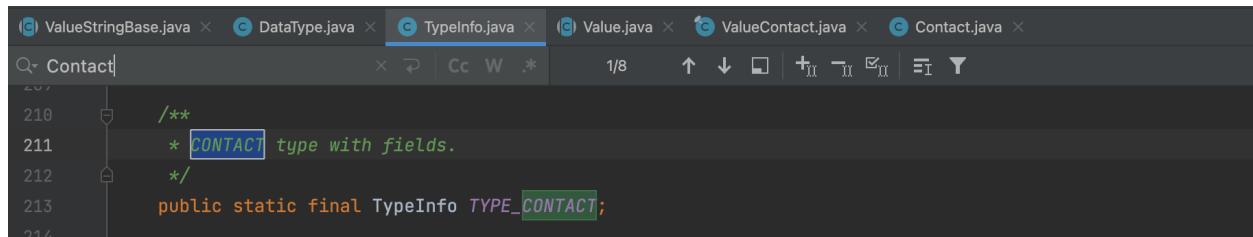
- 1) Below screenshot shows the implementation of `convertToContact()` where we also verify whether the entered CONTACT data type is a valid entry or not using `verifyAndInitializeContact()` defined in our `Contact.java` class.
 - 2) If it is a valid Contact, we proceed, else, we throw a `DBExceptionError`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

```
Q CONTACT x Cc W * 5/11 ↑ ↓ ⌂ ⌃ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌊ ⌋
1200 *
1201 * @return a CHAR value.
1202 */
1203 public ValueChar convertToChar() { return convertToChar(TypeInfo.getTypeInfo(CHAR), provider: null, CONVERT_TO, column: null); }
1204
1205
1206
1207 private ValueStringBase convertToContact(TypeInfo targetType, CastDataProvider provider, int conversionMode, Object column) {
1208     ValueStringBase temp = convertToChar(targetType, provider, conversionMode, column);
1209     return temp;
1210
1211
1212     int valueType = getValueType();
1213     switch (valueType) {
1214         case BLOB:
1215         case JAVA_OBJECT:
1216             throw getDataConversionError(targetType.getValueType());
1217     }
1218     String s = getString();
1219
1220     boolean flag = Contact.verifyAndInitializeContact(s);
1221
1222     if (flag) {
1223         System.out.println(s + " " + s.length() + " " + flag);
1224         return ValueContact.get(s);
1225     }
1226     else
1227         throw DbException.get(ErrorCode.INVALID_CONTACT_ERROR_CODE);
1228 }
```

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

- 1) This file holds data types with parameters.

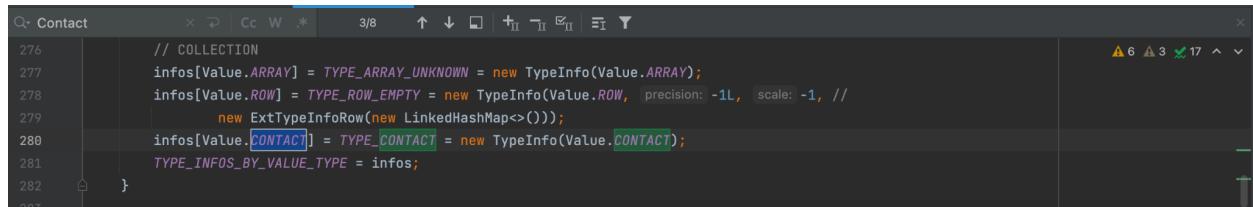
2) Below two screenshots show where we defined a type TYPE_CONTACT for the CONTACT data type and store it.



The screenshot shows a Java code editor with several tabs at the top: ValueStringBase.java, DataType.java, TypeInfo.java (which is the active tab), Value.java, ValueContact.java, and Contact.java. The code in TypeInfo.java defines a static final TypeInfo variable named TYPE_CONTACT, which is annotated with a Javadoc comment specifying it is a CONTACT type with fields.

```
210  /**
211  * CONTACT type with fields.
212  */
213 public static final TypeInfo TYPE_CONTACT;
```

Insert in “static” call in the class:-

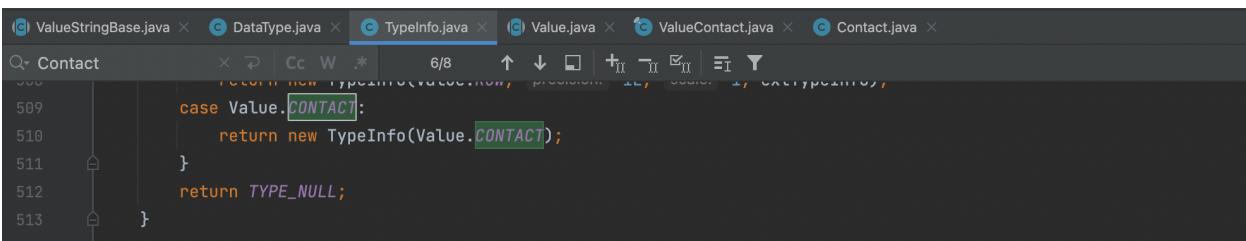


The screenshot shows a Java code editor with the Contact.java tab active. The code is part of a map initialization block. A static call to TYPE_CONTACT is being inserted into the map entry for the CONTACT value type.

```
276 // COLLECTION
277 infos[Value.ARRAY] = TYPE_ARRAY_UNKNOWN = new TypeInfo(Value.ARRAY);
278 infos[Value.ROW] = TYPE_ROW_EMPTY = new TypeInfo(Value.ROW, precision: -1L, scale: -1L, //
279     new ExtTypeInfoRow(new LinkedHashMap<>()));
280 infos[Value.CONTACT] = TYPE_CONTACT = new TypeInfo(Value.CONTACT);
281 TYPE_INFOS_BY_VALUE_TYPE = infos;
282 }
```

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

```
/*
 * 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:
```

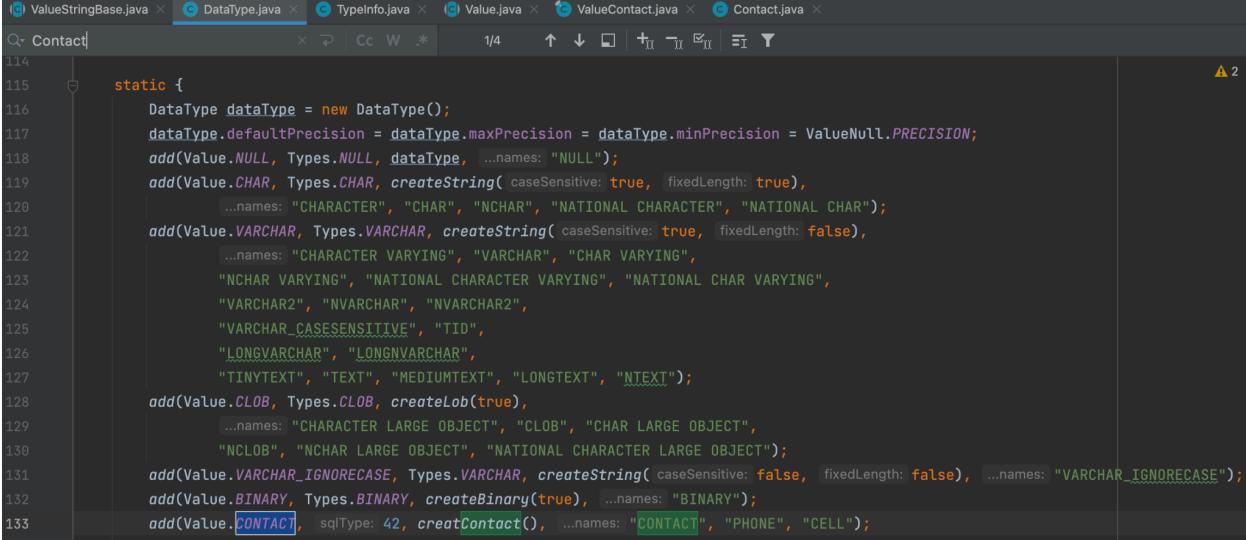


```
case Value.CONTACT:
    return new TypeInfo(Value.CONTACT);
}
return TYPE_NULL;
}
```

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” .



```
114
115     static {
116         DataType dataType = new DataType();
117         dataType.defaultPrecision = dataType.maxPrecision = dataType.minPrecision = ValueNull.PRECISION;
118         add(Value.NULL, Types.NULL, dataType, ...names: "NULL");
119         add(Value.CHAR, Types.CHAR, createString(caseSensitive: true, fixedLength: true),
120             ...names: "CHARACTER", "CHAR", "NCHAR", "NATIONAL CHARACTER", "NATIONAL CHAR");
121         add(Value.VARCHAR, Types.VARCHAR, createString(caseSensitive: true, fixedLength: false),
122             ...names: "CHARACTER VARYING", "VARCHAR", "CHAR VARYING",
123             "NCHAR VARYING", "NATIONAL CHARACTER VARYING", "NATIONAL CHAR VARYING",
124             "VARCHAR2", "NVARCHAR", "NVARCHAR2",
125             "VARCHAR_CASESENSITIVE", "TID",
126             "LONGVARCHAR", "LONGNVARCHAR",
127             "TINYTEXT", "TEXT", "MEDIUMTEXT", "LONGTEXT", "NTEXT");
128         add(Value.CLOB, Types.CLOB, createClob(true),
129             ...names: "CHARACTER LARGE OBJECT", "CLOB", "CHAR LARGE OBJECT",
130             "NCLOB", "NCHAR LARGE OBJECT", "NATIONAL CHARACTER LARGE OBJECT");
131         add(Value.VARCHAR_IGNORECASE, Types.VARCHAR, createString(caseSensitive: false, fixedLength: false),
132             ...names: "VARCHAR_IGNORECASE");
133         add(Value.BINARY, Types.BINARY, createBinary(true), ...names: "BINARY");
134         add(Value.CONTACT, sqlType: 42, createContact(), ...names: "CONTACT", "PHONE", "CELL");
135     }
```



```
211     @
212     private static DataType createContact() {
213         DataType dataType = new DataType();
214         return dataType;
215     }
```

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:

The screenshot shows the H2 Database Browser interface. The left sidebar displays the schema structure with a table named 'EMPLOYEECONTACT' containing columns 'ID' (BIGINT), 'PHONE' (CONTACT), and an index 'PRIMARY_KEY_3'. The main panel shows the SQL statement 'CREATE TABLE EmployeeContact(id IDENTITY PRIMARY KEY, phone CONTACT)' being run. Below the statement, the results show the table was created successfully with an update count of 0 in 16 ms.

```
CREATE TABLE EmployeeContact(id IDENTITY PRIMARY KEY, phone CONTACT);

Update count: 0
(16 ms)
```

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

The screenshot shows the H2 Database Browser interface. The left sidebar displays the schema structure with the same 'EMPLOYEECONTACT' table. The main panel shows the SQL statement 'INSERT INTO EMPLOYEECONTACT (phone) values ('(456)123-1234')' being run. Below the statement, the results show the insert was successful with an update count of 1 in 10 ms.

```
INSERT INTO EMPLOYEECONTACT (phone) values ('(456)123-1234');

Update count: 1
(10 ms)
```

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

The screenshot shows a JDBC interface with the URL `192.168.1.105:8082/login.do?jsessionid=9550b46c32c9a134882895ded1cb9885`. The left sidebar shows the database schema with a table `EMPLOYEECONTACT` having columns `ID`, `PHONE`, and `CONTACT`, and an index `PRIMARY_KEY_3`. The right panel contains a SQL statement window with the following content:

```
INSERT INTO EMPLOYEECONTACT (phone) values ('(456)13-1234');

INSERT INTO EMPLOYEECONTACT (phone) values ('(456)13-1234');

(Message 90158 not found); SQL statement:
INSERT INTO EMPLOYEECONTACT (phone) values ('(456)13-1234') [90158-219] 90158/90158 [Help]
org.h2.jdbc.JdbcSQLException: (Message 90158 not found); SQL statement:
INSERT INTO EMPLOYEECONTACT (phone) values ('(456)13-1234') [90158-219]
    at org.h2.message.DbException.getJdbcSQLException(DbException.java:685)
    at org.h2.message.DbException.getJdbcSQLException(DbException.java:477)
    at org.h2.message.DbException.get(DbException.java:223)
    at org.h2.message.DbException.get(DbException.java:199)
    at org.h2.message.DbException.get(DbException.java:188)
    at org.h2.value.Value.convertToContact(Value.java:1226)
    at org.h2.value.Value.convertTo(Value.java:1192)
    at org.h2.value.Value.convertForAssignTo(Value.java:1098)
    at org.h2.table.Column.validateConvertUpdateSequence(Column.java:369)
    at org.h2.table.Table.convertInsertRow(Table.java:926)
    at org.h2.command.dml.Insert.insertRows(Insert.java:167)
    at org.h2.command.dml.Insert.update(Insert.java:135)
    at org.h2.command.dml.DataChangeStatement.update(DataChangeStatement.java:74)
    at org.h2.command.CommandContainer.update(CommandContainer.java:169)
    at org.h2.command.Command.executeUpdate(Command.java:252)
    at org.h2.jdbc.JdbcStatement.executeInternal(JdbcStatement.java:252)
    at org.h2.jdbc.JdbcStatement.execute(JdbcStatement.java:223)
    at org.h2.server.web.WebApp.getResult(WebApp.java:1339)
    at org.h2.server.web.WebApp.query(WebApp.java:1137)
    at org.h2.server.web.WebApp$1.next(WebApp.java:1103)
    at org.h2.server.web.WebApp$1.next(WebApp.java:1090)
    at org.h2.server.web.WebThread.process(WebThread.java:188)
    at org.h2.server.web.WebThread.run(WebThread.java:101)
    at java.base/java.lang.Thread.run(Thread.java:833)
```

4) A screenshot of the successfully inserted tuple:

The screenshot shows a JDBC interface with the same URL and schema as the previous screenshot. The right panel contains a SQL statement window with the following content:

```
SELECT * FROM EMPLOYEECONTACT;
```

Below the SQL window, the results of the query are displayed in a table:

ID	PHONE
1	(456)123-1234

Text below the table indicates "(1 row, 10 ms)".

Authors

Janhavi Doshi (jd6583)

Kushal Kale (ksk7657)

Nimisha Bhagat (nb2633)

Vinay Jain (vj9898)