

CSCI-621 DBSI

H2 ASSIGNMENT_2: ENHANCED FEATURE

TRISHA P. MALHOTRA
VAMSI CHANDU MANNE
MAYANK PANDEY

INTRODUCTION

- We use H2 Database's open source code to implement an additional feature that enhances its data storage capability and also its indexing capacities. In this homework, we have demonstrated the use of a new datatype called PHONE_NUMBER.
- This is an addition to the current datatypes available for use in H2 database system. Phone_Number datatype is used to store numerical contact numbers in the database for all users.
- The validity of the phone numbers is checked and then they are saved in the memory.
- Syntax for a correct phone_number datatype entry is : '1234567890'.

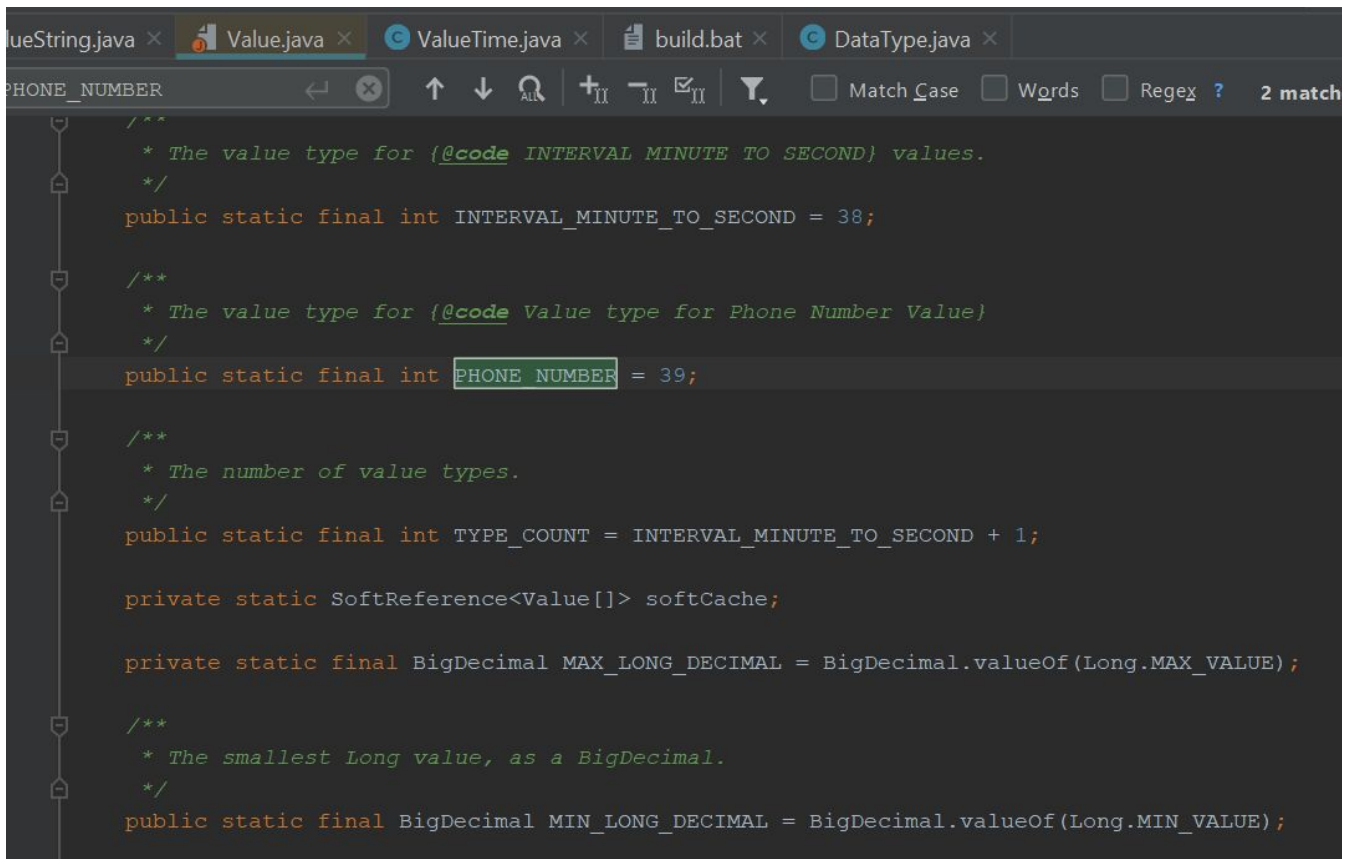
ENHANCED FEATURE

- The current functionalities of H2 do not have a datatype specific to phone numbers. There are various data types available in H2 database like INT, ARRAY, BOOLEAN, BLOB, CLOB etc. For this assignment, we have introduced a new data type "phone number". Along with this we also provide validation functionality of the phone number.
- The reason behind implementing this data type is that many times users enter the wrong number by mistake or incorrect and then it becomes impossible to contact them. This functionality restricts the user to enter their phone number in one format and also validates if the entered number is in the correct format. The validation checks that there are 10 numeric digits in the phone_number.
- For the task of adding a new data type, the group modified two files 'Value.java', 'DataType.java' and created a new class: 'ValuePhoneNumber.java'. These changes were done to incorporate the enhanced feature of phone number data type and its validation. The screenshots below show the code changes done to implement the feature.

IMPLEMENTATION

List of files modified:

- **Value.java**
 - Path: 'h2database-master\h2\src\main\org\h2\value\Value.java'
 - This class provides all conversion and comparison methods.
 - As seen in figure 1, we added a new datatype in this file that contains a list of index values for every datatype in H2.
 - **INDEXING:** Each datatype is given unique index value. Thus ensures that other datatypes do not need to access the memory location allocated to Phone_number datatype. The index value fixed for our datatype is 39. This specifies a unique index value that will be given to the database parser when needed.



```
ValueString.java x Value.java x ValueTime.java x build.bat x DataType.java x
PHONE_NUMBER
/**
 * The value type for {@code INTERVAL MINUTE TO SECOND} values.
 */
public static final int INTERVAL_MINUTE_TO_SECOND = 38;

/**
 * The value type for {@code Value type for Phone Number Value}
 */
public static final int PHONE_NUMBER = 39;

/**
 * The number of value types.
 */
public static final int TYPE_COUNT = INTERVAL_MINUTE_TO_SECOND + 1;

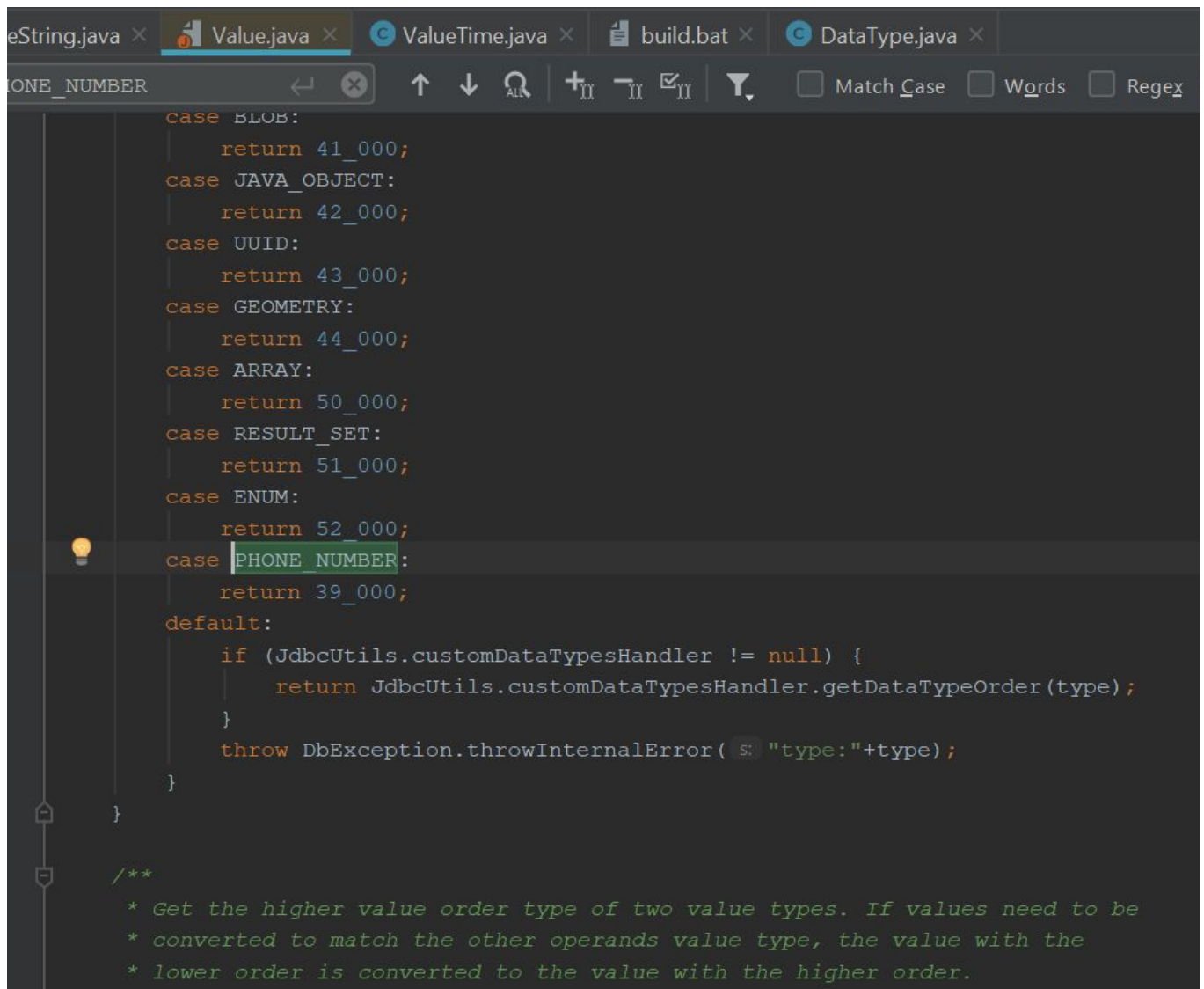
private static SoftReference<Value[]> softCache;

private static final BigDecimal MAX_LONG_DECIMAL = BigDecimal.valueOf(Long.MAX_VALUE);

/**
 * The smallest Long value, as a BigDecimal.
 */
public static final BigDecimal MIN_LONG_DECIMAL = BigDecimal.valueOf(Long.MIN_VALUE);
```

Figure 1: Value.java

- Once the DataType is selected by the user, corresponding datatype value will be returned based switch/ case statements. This file Value.java will return the index number of the concerned datatype which is set as 39 for our new datatype Phone_Number.
- We can see there are other datatypes with different and unique index values set for them. Thus if there is any new datatype that needs to be added later, it cannot have the index value of 39.



```
ONE_NUMBER
case BLOB:
    return 41_000;
case JAVA_OBJECT:
    return 42_000;
case UUID:
    return 43_000;
case GEOMETRY:
    return 44_000;
case ARRAY:
    return 50_000;
case RESULT_SET:
    return 51_000;
case ENUM:
    return 52_000;
case PHONE_NUMBER:
    return 39_000;
default:
    if (JdbcUtils.customDataTypesHandler != null) {
        return JdbcUtils.customDataTypesHandler.getDataTypeOrder(type);
    }
    throw DbException.throwInternalError("type:" + type);
}

/**
 * Get the higher value order type of two value types. If values need to be
 * converted to match the other operands value type, the value with the
 * lower order is converted to the value with the higher order.
```

Figure 2: Return index value

- **ValuePhoneNumber.java**

- Path:
`'h2database-master\h2\src\main\org\h2\value\ValuePhoneNumber.java'`
- This file contains the core implementation of ValuePhoneNumber datatype. As seen in figure 3, this is our main implementation. A new class file was developed to manipulate new entries of datatype phone_number.
- We have defined our new class as per the structure of other available classes for currently supported datatypes. The class `'ValuePhoneNumbe.java'`, makes use of the same structure and configurations to make the integration seamless with the current code of H2 database system.
- All the functions shown in figure3 are key functions like getSQL,getString and equals which are all provided to check and verify different parameters when fetching the phone number that is provided by the user in the server.
- Then there are other methods which provided for basic comparisons between hash index and parameter index.

```

package org.h2.value;

import ...

public class ValuePhoneNumber extends Value {

    public static final ValuePhoneNumber EMPTY = new ValuePhoneNumber( phoneNumberValue: "" );

    public static final int PRECISION = 10;

    private final String phoneNumberValue;

    private ValuePhoneNumber(String phoneNumberValue) {
        if(isValid(phoneNumberValue))
            this.phoneNumberValue = phoneNumberValue;
        else{
            throw DbException.get(ErrorCode.UNKNOWN_DATA_TYPE_1);
        }
    }

    @Override
    public String getSQL() { return StringUtils.quoteStringSQL(phoneNumberValue); }

    @Override
    public int getType() { return Value.PHONENUMBER; }

    @Override
    public long getPrecision() { return PRECISION; }

    @Override
    public int getDisplaySize() { return PRECISION; }

    @Override
    public String getString() { return phoneNumberValue.toString(); }

    @Override
    public Object getObject() { return EMPTY; }

    @Override
    public void set(PreparedStatement prep, int parameterIndex) throws SQLException {
        prep.setString(parameterIndex, phoneNumberValue);
    }
}

```

Figure 3: ValuePhoneNumber.java

- Figure 4 below, shows that the validation of the entered phone number is carried out in the function 'isValid' in the class 'ValuePhoneNumber.java'.
- This method takes input parameter of a String value.
- We know phone number can be in different formats:
 - (123)-(456)-(7890)

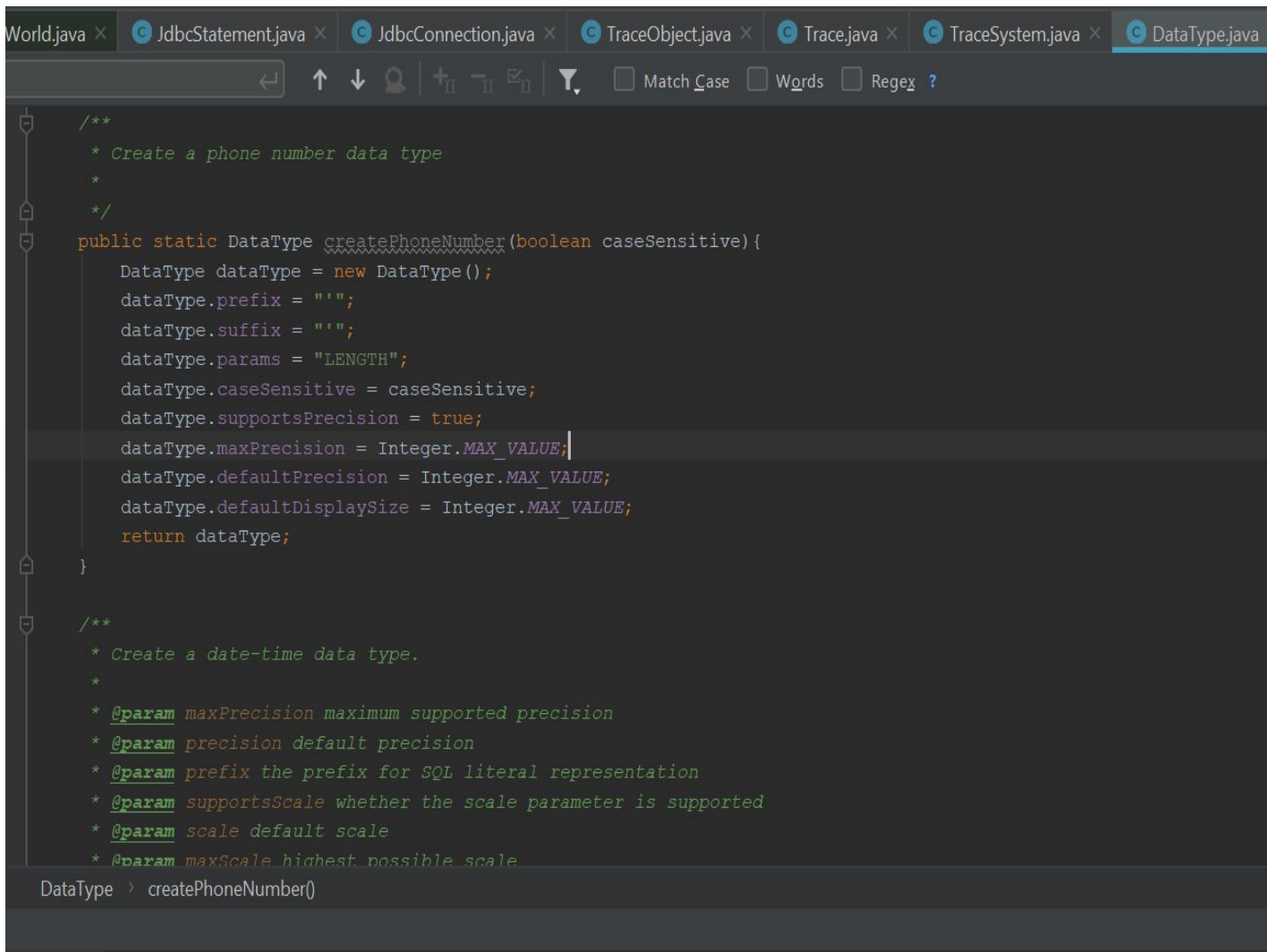
- (+1)-(123)-(456)-(7890)
 - (+1)-1234567890
 - (123)-456-7890
 - 1234567890
- We have limited the format to be only of consecutive 10 numeric values, without any special characters.
 - Of course the validation also checks if any alphabets or other symbols have been given in input by mistake. Thus, expected and approved format of the datatype Phone_Number is '1234567890'.
 - We use regex to perform this check. The function returns a boolean value to approve or disapprove the input's validity.

```
public boolean equals(Object other) {  
    return other instanceof ValuePhoneNumber  
        && phoneNumberValue.equals(((ValuePhoneNumber) other).phoneNumberValue);  
}  
  
@Override  
public int compareTypeSafe(Value v, CompareMode mode) {  
    // TODO Auto-generated method stub  
    return 0;  
}  
  
private boolean isValid(String number){  
    String regex = "d{10}";  
    Pattern pr = Pattern.compile(regex);  
    Matcher match = pr.matcher(regex);  
    return match.matches();  
}  
  
}
```

Figure 4: isValid function

- **DataType.java**

- path: 'h2database-master\h2\src\main\org\h2\value\DataType.java'
- As observed in Figure 5, we have changed an existing file to accommodate a new function for Phone_Number datatypes. Our method 'createPhoneNumber' takes in a boolean value called caseSensitive.
- In this method we define the specifics for writing the data to the database. We set the Display size, Prefix, suffix, precision details and return the datatype back.



```
/**
 * Create a phone number data type
 *
 */
public static DataType createPhoneNumber(boolean caseSensitive){
    DataType dataType = new DataType();
    dataType.prefix = "";
    dataType.suffix = "";
    dataType.params = "LENGTH";
    dataType.caseSensitive = caseSensitive;
    dataType.supportsPrecision = true;
    dataType.maxPrecision = Integer.MAX_VALUE;
    dataType.defaultPrecision = Integer.MAX_VALUE;
    dataType.defaultDisplaySize = Integer.MAX_VALUE;
    return dataType;
}

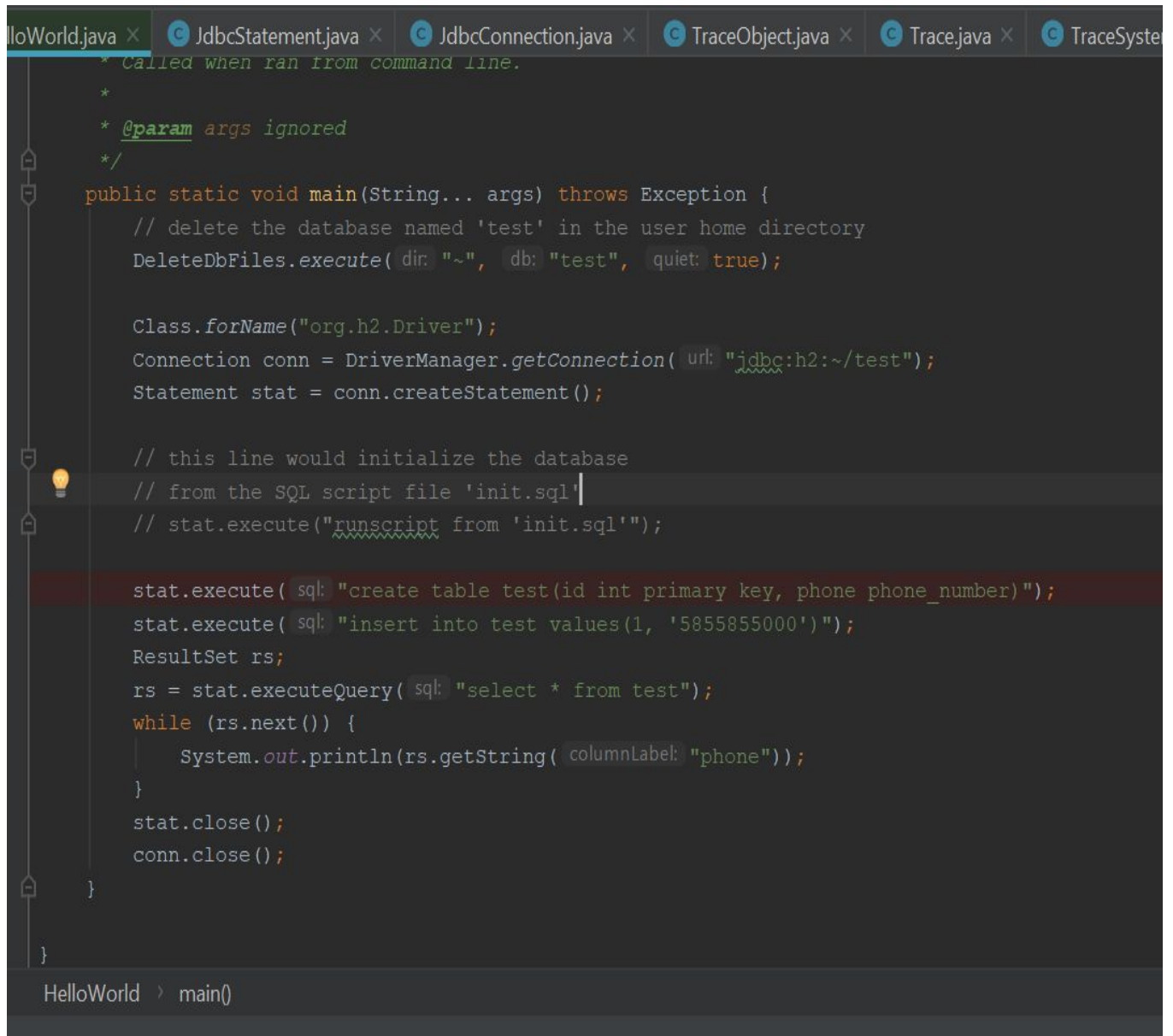
/**
 * Create a date-time data type.
 *
 * @param maxPrecision maximum supported precision
 * @param precision default precision
 * @param prefix the prefix for SQL literal representation
 * @param supportsScale whether the scale parameter is supported
 * @param scale default scale
 * @param maxScale highest possible scale
 */
DataType > createPhoneNumber()
```

Figure 5: DataType.java

- **build.bat**
 - Path: 'h2database-master\h2\build.bat'
 - This file provides custom System Properties to require building the database application. Since we are adding new data type, we need to set its customDataTypeHandler as a System Property.
 - Prior to build an application, code accepts all system properties in the form of VarArgs.

Testing

- We used the file 'h2database-master\h2\src\test\org\h2\samples\HelloWorld.java' for our testing purposes.
- As seen in figure 5, we test the working of new datatype by a slight modification in the HelloWorld.java in test folder.
- Here we test it by creating a table called test and with columns id and phone_number.
- Using the following statement: "create table test(id int primary key, phone_number) " given as string to sql JDBC , we then try to insert values by querying "insert into test values(1, '5855855000')".



```
loWorld.java x JdbcStatement.java x JdbcConnection.java x TraceObject.java x Trace.java x TraceSystem.java x
* Called when ran from command line.
*
* @param args ignored
*/
public static void main(String... args) throws Exception {
    // delete the database named 'test' in the user home directory
    DeleteDbFiles.execute( dir: "~", db: "test", quiet: true);

    Class.forName("org.h2.Driver");
    Connection conn = DriverManager.getConnection( url: "jdbc:h2:~/test");
    Statement stat = conn.createStatement();

    // this line would initialize the database
    // from the SQL script file 'init.sql'
    // stat.execute("runscript from 'init.sql'");

    stat.execute( sql: "create table test(id int primary key, phone phone_number)");
    stat.execute( sql: "insert into test values(1, '5855855000')");
    ResultSet rs;
    rs = stat.executeQuery( sql: "select * from test");
    while (rs.next()) {
        System.out.println(rs.getString( columnLabel: "phone"));
    }
    stat.close();
    conn.close();
}
}

HelloWorld > main()
```

Figure 6: Testing

All the functions created for PhoneNumber data type is used to either compare or store/retrieve that provided phone number from H2 database. In conclusion, we created a new feature called PhoneNumber as it is a function that is most commonly used while providing user credentials. All the above seen snippets of code are compiled and executed. Creating this feature just made it easier for user to store user contact details in H2 database.

-----X-----