```
/** This is Mini-in memory database which acts like hashtable
 * Key will always be String, Value can be one of String,
Integer, Double, ArrayFormat or ObjectFormat
 * Add, Delete, Modify and Undo operations are supported
 */
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.JsonParser;
import org.codehaus.jackson.JsonProcessingException;
import org.codehaus.jackson.annotate.JsonAutoDetect.Visibility;
import org.codehaus.jackson.annotate.JsonMethod;
import org.codehaus.jackson.map.ObjectMapper;

public class Database implements IDatabase {

	protected String commandFile;
	protected String dbSnapshotFile;
	protected IDatabaseData data;
	protected int MAXIMUM_COMMANDS_BEFORE_SAVING = 80;
	protected ICommandProcessor commandProcessor;
	private Hashtable<String, ICursor> databaseObservers;

	private Database() {
		this.commandFile = "commands.txt";
		this.dbSnapshotFile = "dbSnapshot.txt";
		this.data = new DatabaseData();
		this.commandProcessor = new
CommandProcessor(MAXIMUM_COMMANDS_BEFORE_SAVING);
		databaseObservers = new Hashtable<>();
		try {
			recover();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	/** Only one instance of database will be available **/
	private static class SingletonHolder {
		private final static Database INSTANCE = new
Database();
```

```
	}

	public static Database getDatabase() {
		return SingletonHolder.INSTANCE;
	}

	@Override
	public IDatabase put(String key, Integer value) {
		return putHelper(key, value);
	}

	@Override
	public IDatabase put(String key, Double value) {
		return putHelper(key, value);
	}

	@Override
	public IDatabase put(String key, String value) {
		return putHelper(key, value);
	}

	@Override
	public IDatabase put(String key, ArrayFormat value) {
		return putHelper(key, value);
	}

	@Override
	public IDatabase put(String key, ObjectFormat value) {
		return putHelper(key, value);
	}

	private IDatabase putHelper(String key, Object value) {
		if (value == null)
			throw new IllegalArgumentException("Value cannot
be null");
		Boolean addSuccess = this.commandProcessor.commit(new
AddDataCommand(this, key, value));
		if (addSuccess)
			notifyObservers(key);
		return this;
	}

	@Override
	public int getInt(String key) throws
DataTypeMisMatchException {
		if (!this.data().containsKey(key))
			throw new IllegalArgumentException("Key not
found");
		int value;
		try {
			value = (int) this.data().get(key);
		} catch (ClassCastException e) {
```

```
                throw new DataTypeMisMatchException("value is not
of int type");
          }
          return value;
     }

     @Override
     public double getDouble(String key) throws
DataTypeMisMatchException {
          if (!this.data().containsKey(key))
                throw new IllegalArgumentException("Key not
found");
          double value;
          try {
                value = (double) this.data().get(key);
          } catch (ClassCastException e) {
                throw new DataTypeMisMatchException("value is not
of double type");
          }
          return value;
     }

     @Override
     public ArrayFormat getArray(String key) throws
DataTypeMisMatchException {
          if (!this.data().containsKey(key))
                throw new IllegalArgumentException("Key not
found");
          ArrayFormat value = null;
          try {
                value = (ArrayFormat) this.data().get(key);
          } catch (ClassCastException e) {
                throw new DataTypeMisMatchException("value is not
of array type");
          }
          return value;
     }

     @Override
     public String getString(String key) throws
DataTypeMisMatchException {
          if (!this.data().containsKey(key))
                throw new IllegalArgumentException("Key not
found");
          String value;
          try {
                value = (String) this.data().get(key);
          } catch (ClassCastException e) {
                throw new DataTypeMisMatchException("value is not
of string type");
          }
          return value;
```

```
	}

	@Override
	public ObjectFormat getObject(String key) throws
DataTypeMisMatchException {
		if (!this.data().containsKey(key))
			throw new IllegalArgumentException("Key not
found");
		ObjectFormat value = null;
		try {
			value = (ObjectFormat) this.data().get(key);
		} catch (ClassCastException e) {
			throw new DataTypeMisMatchException("value is not
of type object");
		}
		return value;
	}

	@Override
	public Object get(String key) {
		if (!this.data().containsKey(key))
			throw new IllegalArgumentException("Key not
found");
		return this.data().get(key);
	}

	/** This returns the cursor which for the particular key **/
	@Override
	public ICursor getCursor(String key) {
		if (!this.data().containsKey(key))
			throw new IllegalArgumentException("Key not
found");
		Cursor cursor = new Cursor(this, key);
		databaseObservers.put(key, cursor);
		return cursor;
	}

	@Override
	public Object delete(String key) {
		Object valueToDelete = this.data.get(key);
		Boolean deleteSuccess =
this.commandProcessor.commit(new DeleteDataCommand(this, key,
valueToDelete));
		if (deleteSuccess) {
			databaseObservers.remove(key);
		}
		return valueToDelete;
	}

	@Override
	public boolean modify(String key, Integer value) {
```

```
            return modifyValue(key, value);
      }

      @Override
      public boolean modify(String key, Double value) {
            return modifyValue(key, value);
      }

      @Override
      public boolean modify(String key, String value) {
            return modifyValue(key, value);
      }

      @Override
      public boolean modify(String key, ArrayFormat value) {
            return modifyValue(key, value);
      }

      @Override
      public boolean modify(String key, ObjectFormat value) {
            return modifyValue(key, value);
      }

      private boolean modifyValue(String key, Object data) {
            if (!this.data().containsKey(key))
                  throw new IllegalArgumentException("Key not
found");
            if (data == null)
                  throw new IllegalArgumentException("Value cannot
be null");
            Boolean modifySuccess =
this.commandProcessor.commit(new ModifyDataCommand(this, key,
data));
            if (modifySuccess)
                  notifyObservers(key);
            return modifySuccess;
      }

      @Override
      public void undo() {
            if (this.commandProcessor.canUndo()) {
                  this.commandProcessor.undo();
            }
      }

      @Override
      @SuppressWarnings("unchecked")
      public Hashtable<String, Object> data() {
            return (Hashtable<String, Object>) data;
      }

      /**
```

```
       * Returns the transaction object which can be used to
perform operation on
       * the database
       **/
      @Override
      public ITransaction transaction() {
            return (new Transaction(this, this.commandProcessor,
this.data));
      }

      @Override
      public Object snapShot() {
            return new DatabaseMemento(this.dbSnapshotFile,
this.data);
      }

      @Override
      public Object snapShot(File commands, File snapshot) throws
Exception {
            return new DatabaseMemento(commands, snapshot,
this.data);
      }

      @Override
      public void recover() throws Exception {
            File dbSnapshotName = new File(dbSnapshotFile);
            File commandFileName = new File(commandFile);
            if (dbSnapshotName.exists() &&
dbSnapshotName.length() > 0) {
                  ObjectMapper mapper = new ObjectMapper();
                  JsonNode value = null;
                  try {
                        value = mapper.readTree(new
FileInputStream(this.dbSnapshotFile));
                  } catch (IOException e) {
                        e.printStackTrace();
                  }
                  this.data().putAll(JsonNodeHelper(value));
            }
            if (commandFileName.exists() &&
commandFileName.length() > 0) {
                  restoreCommand(commandFileName);
            }
      }

      @Override
      public void recover(File commands, File snapshot) throws
JsonProcessingException, Exception {
            if (snapshot.length() > 0) {
                  ObjectMapper mapper = new ObjectMapper();
                  JsonNode value = null;
                  try {
```

```
                        value = mapper.readTree(new
FileInputStream(snapshot));
                } catch (Exception e) {
                        e.printStackTrace();
                }
                this.data().putAll(JsonNodeHelper(value));
          }
          if (commands.length() > 0)
                restoreCommand(commands);

     }

     /**
      * Helper methods to read the values stored in the
snapShot.txt file Have
      * used Jackson API to read the values
      **/
     private HashMap<String, Object> JsonNodeHelper(JsonNode
value) throws Exception {
          HashMap<String, Object> mapData = new HashMap<String,
Object>();
          Iterator<Entry<String, JsonNode>> fieldsIterator =
value.getFields();
          Map.Entry<String, JsonNode> jsonFields = null;
          while (fieldsIterator.hasNext()) {
                Map.Entry<String, JsonNode> field =
fieldsIterator.next();
                final String key = field.getKey();
                final JsonNode data = field.getValue();
                if (data.isContainerNode()) {
                      Iterator<Map.Entry<String, JsonNode>>
innerfieldsIterator = data.getFields();
                      while (innerfieldsIterator.hasNext()) {
                            jsonFields =
innerfieldsIterator.next();
                      }
                      JsonParser parser = new
ObjectMapper().treeAsTokens(jsonFields.getValue());
                      mapData.put(key, new
ObjectMapper().readTree(parser));
                } else {
                      mapData.put(key, data);
                }
          }
          return mapData;
     }

     /**
      * this reads the operations from commands.txt and performed
based on the
      * commands present in the commands.txt file
      *
```

```
       * Example: add@key@value@String, delete@key@12@Integer
       **/
      private void restoreCommand(File fileName) {
            FileReader fileRead = null;
            FileWriter fileWriter = null;
            String data;
            try {
                  fileRead = new FileReader(fileName);
                  BufferedReader br = new BufferedReader(fileRead);
                  while ((data = br.readLine()) != null) {
                        String[] value = data.split("@");
                        if (value[0].equals("add")) {
                              this.data.put(value[1],
valueTypeConversionHelper(value[2], value[3]));
                        } else if (value[0].equals("delete")) {
                              this.data.remove(value[1]);
                        }
                  }
                  fileWriter = new FileWriter(fileName);
                  fileWriter.write("");
                  fileWriter.flush();
                  fileRead.close();
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }

      /**
       * This converts the string representation of objects into
appropriate java
       * types by using the data present in commands.txt file
       *
       * Example: add@key@value@String
       **/
      private Object valueTypeConversionHelper(String value,
String valueType) {
            if (valueType.equals("Integer"))
                  return Integer.parseInt(value);
            else if (valueType.equals("Double"))
                  return Double.parseDouble(value);
            else if (valueType.equals("String")) {
                  return (String) value;
            } else if (valueType.equals("Array"))
                  return ArrayFormat.fromString(value);
            else
                  return ObjectFormat.fromString(value);
      }

      private void notifyObservers(String key) {
            if (databaseObservers.isEmpty())
                  return;
            ICursor cursor = databaseObservers.get(key);
```

```
            if (cursor != null)
                  cursor.notifyObserver();
      }

      /** Memento pattern used to save the database state to
memento **/
      private class DatabaseMemento {
            private String fileName;
            private IDatabaseData data;

            public DatabaseMemento(String fileName, IDatabaseData
data) {
                  this.fileName = fileName;
                  this.data = data;
                  ObjectMapper mapper = new ObjectMapper();
                  mapper.setVisibility(JsonMethod.FIELD,
Visibility.ANY);
                  try {
                        mapper.writeValue(new File(this.fileName),
this.data);
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
            }

            public DatabaseMemento(File commands, File snapshot,
IDatabaseData data) throws Exception {
                  this.data = data;
                  ObjectMapper mapper = new ObjectMapper();
                  FileWriter fileWriter = null;
                  mapper.setVisibility(JsonMethod.FIELD,
Visibility.ANY);
                  try {
                        mapper.writeValue(snapshot, this.data);
                        fileWriter = new FileWriter(commands,
false);
                        fileWriter.write("");
                        fileWriter.flush();
                        fileWriter.close();
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
            }

      }

}
```