Object-Oriented Programming

Input / Output
Databases (optional)

Computer Science and Technology
United International College

Outline

- Simple Input / Output
- File Input / Output
- Objects Input / Output
- Object Serialization
- Databases (optional)

Input / Output

- Warning: Java has many different ways to do input and output.
  - **`FileReader`**, **`FileWriter`**, **`FileOutputStream`**, **`FileInputStream`**, **`PrintWriter`**, **`PrintStream`**, etc.!
  - So we will only cover the basic ways to do I/O.
  - For more: Basic I/O Tutorial (Oracle web site)
- We will cover only two types of I/O:
  - From keyboard / to screen.
  - From file / to file.
  - Not covered: I/O with network, printer, video camera, etc.

Input / Output

- In Java, an input / output stream is an object that represent a source of input data or a destination of output data.
- Java programs have three streams that are always available by default:
  - `System.in` is the standard input stream, which usually represents the computer's keyboard.
  - `System.out` is the standard output stream, which usually represents the computer's screen.
  - `System.err` is the standard error stream, which usually represents the computer's screen and is only used to output error messages (independently of `System.out`, so the two might get mixed on the screen!)

Screen Output

```
public class Test {
    public static void main(String[] args) {
        System.out.print("hello ");
        System.out.println("world");
        // %n is a better version of \n which works correctly on all
        // computers (Windows, Mac OS, etc.) and with files too.
        System.out.printf("hello world%n");
        System.out.printf("hello %-4d world %10.4f%n", 2, 3.4);
        System.err.print("hello ");
        System.err.println("world");
        System.err.printf("hello world%n");
        System.err.printf("hello %-4d world %10.4f%n", 2, 3.4);
    }
}
```

Keyboard Input

- The easiest way to use `System.in` is through a `Scanner` object that reads bytes of data from `System.in` and transforms the bytes into Java values.
- The `Scanner` class provides many method to read various kinds of inputs: `nextLine`, `nextInt`, `nextDouble`, `nextBoolean`, etc.
  - An `InputMismatchException` will be thrown if the user types the wrong thing!
- You can check whether more input is available by using the scanner's `hasNext` method, which returns `true` as long as there is more data to be read.
  - Type `Control-z` (Windows) or `Control-d` (Mac OS, Unix) to manually indicate the end of the input (EOF).

Keyboard Input

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Type some text: ");
        while(s.hasNext()) { // Block until something typed or EOF.
            String str = s.nextLine(); // Read one line, discard %n.
            System.out.println("The text is: " + str);
            System.out.print("Type some text: ");
        }
        System.out.println(); // Go to next line after end of input.
        System.out.println("No more input.");
    }
}
```

File Output

- The easiest way to write to a text file is to create a `PrintWriter` output stream object for the file.
  - If the file does not yet exist, it is created (in the folder for the current Eclipse project).
  - If the file already exists, the old content of the file is deleted.
  - A `FileNotFoundException` is thrown if it is not possible to create or open the file (hard disk full, etc.)
- You can then use the usual `print` / `println` methods to write to the file through the stream.
- Remember to close the file at the end to guarantee that all your data has been saved on disk.

File Output

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
public class Test {
    public static void main(String[] args) {
        try {
            PrintWriter out = new PrintWriter("mydata.txt");
            out.print("hello");
            out.println(" world");
            out.printf("hello world%n");
            out.printf("hello %-4d world %10.4f%n", 2, 3.4);
            out.close(); // Do not forget this!
        } catch(FileNotFoundException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

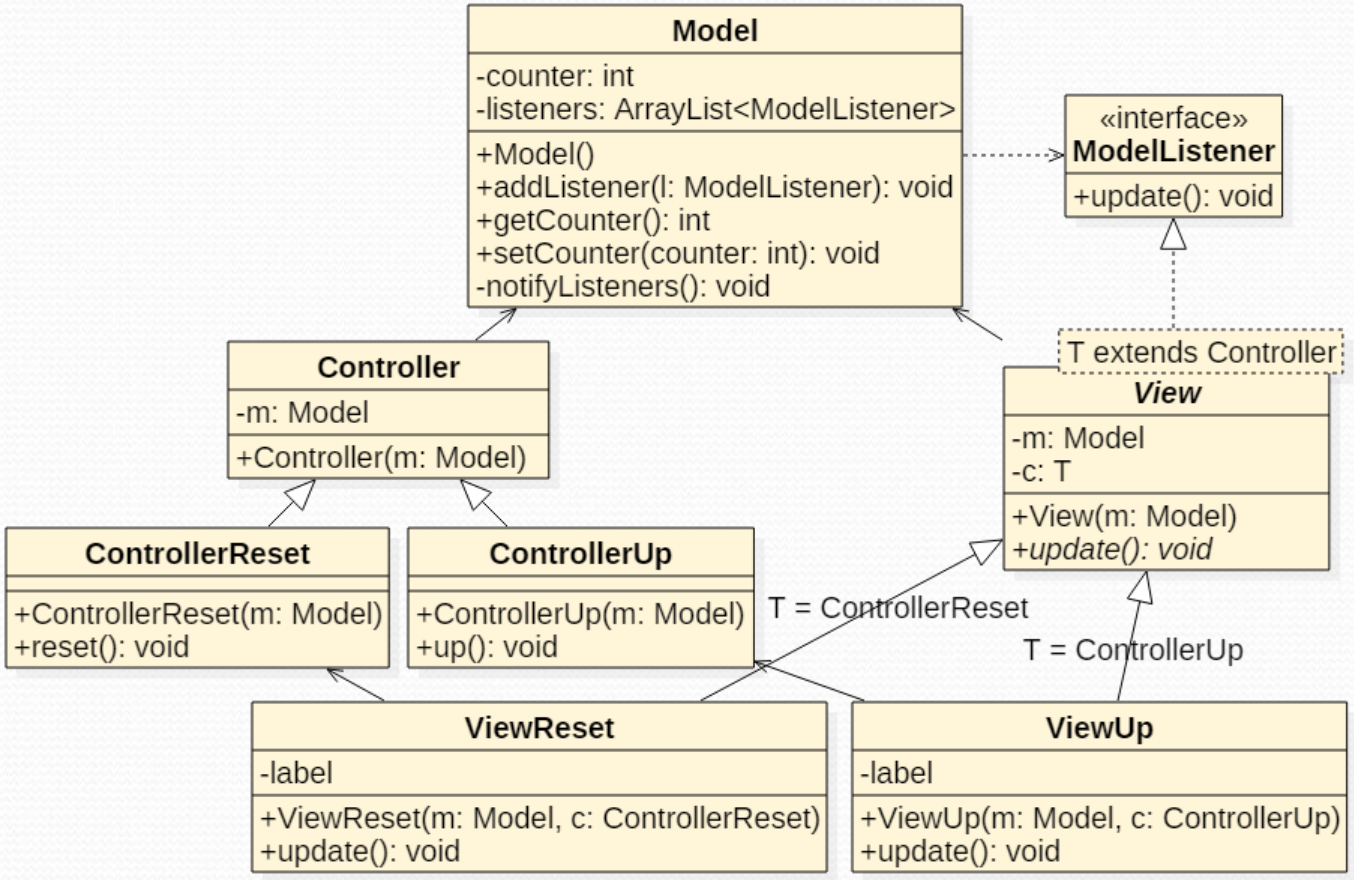
File Input

- The easiest way to read from a text file is to create a `File` object representing the file, and then read bytes from the file object and transform them into Java values using a `Scanner` object.
  - A `FileNotFoundException` is thrown if the file does not already exist.
- You can then use the usual `nextLine`, `nextInt`, etc., methods to read from the file through the scanner.
- The method `hasNext` returns `true` until you reach the end of the file (EOF), then it returns `false`.
- Remember to close the file at the end.

File Input

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        File f = new File("mydata.txt");
        try {
            Scanner s = new Scanner(f);
            while(s.hasNext()) { // True until EOF.
                String str = s.nextLine(); // Read one line, discard %n.
                System.out.println("The file text is: " + str);
            }
            s.close(); // Do not forget this!
        } catch(FileNotFoundException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

Example: MVC GUI




```
public interface ModelListener {
    public void update();
}
```

Example: MVC GUI

```
import java.util.ArrayList;
public class Model {
    private int counter;
    private ArrayList<ModelListener> listeners;

    public Model() {
        counter = 0;
        listeners = new ArrayList<ModelListener>();
    }
    public void addListener(ModelListener l) {
        listeners.add(l);
    }
    public int getCounter() { return counter; }
    public void setCounter(int counter) {
        this.counter = counter;
        notifyListeners(); // Counter changed so notify the listeners.
    }
    private void notifyListeners() {
        for(ModelListener l: listeners) {
            l.update(); // Tell the listener that something changed.
        }
    }
}
```

Example: MVC GUI

```
import javax.swing.JFrame;
public abstract class View<T extends Controller> extends JFrame
implements ModelListener {
    protected Model m;
    protected T c;
    public View(Model m, T c) {
        this.m = m;
        this.c = c;
        m.addListener(this);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    @Override
    public abstract void update();
}

public class Controller {
    protected Model m;
    public Controller(Model m) {
        this.m = m;
    }
}
```

Example: MVC GUI

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JLabel;
public class ViewUp extends View<ControllerUp> {
    private JLabel label;

    public ViewUp(Model m, ControllerUp c) {
        super(m, c);

        this.setTitle("View Up");
        this.setSize(150, 150);
        this.setLayout(new GridLayout(2, 1));

        label = new JLabel();
        update();
        this.add(label);
        ...
```

Example: MVC GUI

```
        ...
        JButton buttonUp = new JButton("Up");
        buttonUp.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                c.up();
            }
        });
        this.add(buttonUp);
        this.setVisible(true);
    }
    @Override
    public void update() {
        label.setText("counter is: " + m.getCounter());
    }
}
```

Example: MVC GUI

```
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JLabel;
public class ViewReset extends View<ControllerReset> {
    private JLabel label;

    public ViewReset(Model m, ControllerReset c) {
        super(m, c);

        this.setTitle("View Reset");
        this.setSize(150, 150);
        this.setLayout(new GridLayout(2, 1));

        label = new JLabel();
        update();
        this.add(label);
        ...
```

Example: MVC GUI

```
        ...
        JButton buttonReset = new JButton("Reset");
        buttonReset.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                c.reset();
            }
        });
        this.add(buttonReset);
        this.setVisible(true);
    }
    @Override
    public void update() {
        label.setText("counter is: " + m.getCounter());
    }
}
```

Example: MVC GUI

```
public class ControllerUp extends Controller {
    public ControllerUp(Model m) {
        super(m);
    }
    public void up() {
        m.setCounter(m.getCounter() + 1);
    }
}

public class ControllerReset extends Controller {
    public ControllerReset(Model m) {
        super(m);
    }
    public void reset() {
        m.setCounter(0);
    }
}
```

Example: MVC GUI

```
public class Test {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                Model m = new Model(); // Single shared model.
                ControllerUp c1 = new ControllerUp(m);
                ViewUp v1 = new ViewUp(m , c1);
                ControllerUp c2 = new ControllerUp(m);
                ViewUp v2 = new ViewUp(m , c2);
                ControllerReset c3 = new ControllerReset(m);
                ViewReset v3 = new ViewReset(m, c3);
            }
        });
    }
}
```

Example: MVC GUI

- If your software correctly follows the MVC design pattern then all the data is guaranteed to be only in the `Model` class so this is the only class that needs to change if we want to store the counter in a file!
- Then the software will remember the counter from one execution to the next.
- We just need to create the file in the `Model` constructor if it does not exist yet.
- And read / write the file in the `getCounter` / `setCounter` methods.
- Then we can remove the `counter` instance variable.

Example: MVC GUI

```
import ...
public class Model {
    private File file; // No counter anymore.
    private ArrayList<ModelListener> listeners;

    public Model() {
        file = new File("mycounter.txt");
        if(!file.exists()) { // then create it:
            try {
                PrintWriter out = new PrintWriter(file);
                out.println(0); // Initialize counter in new file.
                out.close();
            } catch(FileNotFoundException e) {
                System.err.println(e.getMessage());
                System.exit(1); // End immediately.
            }
        }
        listeners = new ArrayList<ModelListener>();
    }
    ...
```

Example: MVC GUI

```
public int getCounter() {
    try {
        Scanner s = new Scanner(file);
        int counter = s.nextInt();
        s.close();
        return counter;
    } catch(FileNotFoundException ex) {
        System.err.println(ex.getMessage());
        System.exit(1); // End immediately.
        return 0; // To make the compiler happy.
    }
}
public void setCounter(int counter) {
    try {
        PrintWriter out = new PrintWriter(file);
        out.println(counter); // Update file.
        out.close();
    } catch(FileNotFoundException e) {
        System.err.println(e.getMessage());
        System.exit(1); // End immediately.
    }
    notifyListeners(); // Counter changed so notify the listeners.
}
```

Example: MVC GUI

- Reading / writing the file on every read / write of the counter is not very efficient though.
- It is better to:
  - read the file once when the software starts and the model is created (the constructor of `Model` is called);
  - write the file once when the software stops.
- Problem: how to execute some code when the software stops?
- One solution: have "save" and "quit" buttons in a view and check that the user saved the data before quitting.

Example: MVC GUI

Another solution:

- Change the `View` superclass so that clicking on the "close" button in the title of the frame only hides the frame instead of terminating the software.
- Add a "window closing" event handler to `View` that calls the `shutdown` method of the `Controller`.
  - Remember that, in the MVC design pattern, it is the controller which decides the meaning of user actions.
- The `shutdown` method of the `Controller` then:
  - tells the model to save its data into the file using the model's new `saveData` method;
  - manually terminates the software using `System.exit(0)`.

Example: MVC GUI

```
import ...
public class Model {
    private int counter;
    private File file;
    private ArrayList<ModelListener> listeners;

    public Model() {
        file = new File("mycounter.txt");
        if(file.exists()) { // then read the counter once:
            try {
                Scanner s = new Scanner(file);
                counter = s.nextInt();
                s.close();
            } catch(FileNotFoundException ex) {
                System.err.println(ex.getMessage());
                System.exit(1); // End immediately.
            }
        } else {
            counter = 0;
        }
        listeners = new ArrayList<ModelListener>();
    }
```

Example: MVC GUI

```
    public void saveData() { // Write the counter once.
        try {
            PrintWriter out = new PrintWriter(file);
            out.println(counter); // Update file.
            out.close();
        } catch(FileNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(1); // End immediately.
        }
    }
    public void addListener(ModelListener l) {
        listeners.add(l);
    }
    public int getCounter() { // No file access.
        return counter;
    }
    public void setCounter(int counter) { // No file access.
        this.counter = counter;
        notifyListeners(); // Counter changed so notify the listeners.
    }
    private void notifyListeners() {
        ...
    }
}
```

Example: MVC GUI

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
public abstract class View<T extends Controller> extends JFrame
implements ModelListener {
    protected Model m;
    protected T c;

    public View(Model m, T c) {
        this.m = m;
        this.c = c;
        m.addListener(this);
        this.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
        this.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                c.shutdown(); // Controller decides meaning.
            }
        });
    }
    @Override
    public abstract void update();
}
```

Example: MVC GUI

```
public class Controller {
    protected Model m;

    public Controller(Model m) {
        this.m = m;
    }
    protected void shutdown() {
        m.saveData();    // Tell the model to save its data.
        System.exit(0); // Normal exit.
    }
}
```

Objects Input / Output

If you use **println** or some other method to write an object to the screen or to a file, then Java will only print the type and address of the object:

```
public class Circle {
    private int x;
    private int y;
    private double radius;
    public Circle(int x, int y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    public static void main(String[] args) {
        Circle c = new Circle(10, 20, 30.0);
        System.out.println(c);
    }
}
```

→ **Circle@70dea4e**

Objects Input / Output

We can override the **`toString`** method inherited from the **`Object`** class and then things work nicely:

```
public class Circle {
    private int x;
    private int y;
    private double radius;
    public Circle(int x, int y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    @Override
    public String toString() {
        return "(Circle " + x + " " + y + " " + radius + ")";
    }
    public static void main(String[] args) {
        Circle c = new Circle(10, 20, 30.0);
        System.out.println(c);
    }
}
```

→ `(Circle 10 20 30.0)`

Objects Input / Output

- Unfortunately this only works nicely for writing, not for reading!
- For reading, we need to transform the string "`(Circle 10 20 30.0)`" back into a `Circle` object.
  - This requires reading each piece of data (the type, `x`, `y`, and `radius`) from a file into local variables using the appropriate `next` / `nextInt` / `nextDouble` / … methods of the `Scanner` class, then creating the `Circle` object using the `Circle`'s constructor.
  - It must be done in a `static` method of the `Circle` class because we must be able to call the method even though we do not have any `Circle` object yet (which is the result of the method).

Objects Input / Output

- If the file can contain different kinds of objects (circles, squares, dots, etc.) things become more complicated.
- If the objects can be nested `(Circle (Point 10 20) 30.0)` then things become even more complicated.
- We might then need to write a full-scale parser (see the "Compiler Construction" course).

Object Serialization

Fortunately Java provides an easy solution: serialization.

- Just specify that your class implements the (empty) `Serializable` Java interface.
- You can then easily read / write objects of the class into a binary file using the `readObject` / `writeObject` methods!
- The result of the `readObject` method is of type `Object` so a downcast is necessary.

Object Serialization

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Circle implements Serializable {
    private int x;
    private int y;
    private double radius;

    public Circle(int x, int y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    // toString removed.

    ...
```

Object Serialization

```
public static void main(String[] args) {
    Circle c1 = new Circle(10, 20, 30.0);

    try {
        FileOutputStream fo = new FileOutputStream("mydata.bin");
        ObjectOutputStream out = new ObjectOutputStream(fo);
        out.writeObject(c1);
        out.close();
        fo.close();
    } catch(IOException e) {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    ...
```

Object Serialization

```
        try {
            FileInputStream fi = new FileInputStream("mydata.bin");
            ObjectInputStream in = new ObjectInputStream(fi);
            Circle c2 = (Circle)in.readObject(); // Cast is mandatory!
            in.close();
            fi.close();
            System.out.println("c2 radius: " + c2.radius);
        } catch(IOException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        } catch(ClassNotFoundException e) {
            // This only happens if the Circle class does not exist
            // in the Java software used to read the binary file.
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

Object Serialization

- If an object depends on other objects (for example, a `Circle` object uses a `Point` object for its center) then Java automatically reads / writes all the other objects too, recursively!
- `ArrayList` and many other Java classes also implement `Serializable`, so no need to worry about that either!

Life is beautiful!

- Note: serialization is also very useful to transmit objects between Java programs over a network.

Databases (optional)

- Instead of saving the data into a text file or a binary file, we can save the data into a database.
  - For convenience in this course we use a database running on the same computer as the Java software.
  - The database can equally be on a remote computer accessed through the network (with a password).
- The Java Database Connectivity (JDBC) API is mostly independent of which database software you use, except that:
  - You need to use the correct Java database "connector" software for the specific database you are using.
  - You need to specify the correct `"jdbc:...://..."` URL for your specific database when connecting to it.

Databases

How to set up things to make it work:

1. Download and install XAMPP.
   - Includes the **MariaDB** database (same as **MySQL**) plus the **Apache** / **phpMyAdmin** DBMS web system.
2. Start **XAMPP**, then **Apache** and **MariaDB** / **MySQL**.
3. Connect to the DBMS with a web browser using the http://localhost/phpmyadmin/ web link.
4. Use **phpMyAdmin** to create a database and tables.
   - For example: `CREATE DATABASE java;`
   - Then (in the **java** database):

```
CREATE TABLE data (ID INT, COUNTER INT, PRIMARY KEY(ID));
```

Databases

5. Download the MariaDB Connector/J.
   - This is the Java software that will make the connection between your Java code and the DB through the network.
6. Use the **`New`** → **`Folder`** project menu to add a new **`lib`** folder in your Eclipse project.
7. Drag the **`MariaDB Connector/J`** file into the **`lib`** folder.
   - Select "copy files" when asked, otherwise your project will only contain a link to the file, not the file itself, and your code will fail if your send it to someone else later.
8. Use the **`Build Path`** → **`Add to Build Path`** menu to make the **`MariaDB Connector/J`** software available to your own Java software.

Databases

To use the database in your Java code:

1. Open a **`Connection`** to the DB using the **`DriverManager.`*`getConnection`*** method:
   - Use **`"jdbc:mariadb://hostname/DBname"`** for the URL.
   - Default user name is **`"root"`** and default password is empty.
2. Create a **`Statement`** object from the **`Connection`** object to represent an SQL statement.
3. Use the **`executeQuery`** method of the **`Statement`** object to make a query. The result is a **`ResultSet`** object that contains all the rows for the query's result.

Databases

4. Process the **`ResultSet`** object as required:
   - Use the **`next`** method to move from row to row in the result (use this method as the test of a **`while`** loop if there are many rows to process).
   - Use the **`getInt(columnIndex)`** / **`getString(…)`** / **`getDouble(…)`** / etc. methods to access the values in the current row at the given column index.
5. To execute **`INSERT`** or **`UPDATE`** or **`DELETE`** SQL commands to change the content of the DB, use the **`executeUpdate`** method of the **`Statement`** object instead of the **`executeQuery`** method.

Databases

Important: make sure you always close:

- the `ResultSet` object;
- the `Statement` object;
- the DB `Connection` object;

otherwise strange things might start to happen (too many simultaneous DB connections, maximum number of DB "cursors" reached, etc.)

Java's automatic memory management system (garbage collector) only manages the Java objects, it cannot automatically manage for you the corresponding resources allocated inside the DB itself! So you must do it yourself!

Databases

The easiest way to do this is to use a Java "try-with-resources" statement:

```
try (
        ... // Open resources here!
    ) {
        ... // Use resources here.
} catch(...) { // Optional.
        ...
} // Resources are automatically closed here.
```

Any resource created inside the **()** parentheses is then guaranteed to be automatically closed and freed at the end of the whole statement, even if there is an exception.

This is similar to using a **finally** statement at the end of the **try** to manually close all resources, but it is done for you automatically by the JVM so it is much easier to use.

Example: MVC GUI

```
import ...
public class Model {
    private int counter;
    private ArrayList<ModelListener> listeners;
    public Model() {
        try (// Connect to DB using URL and user name and empty password.
            Connection conn = DriverManager.getConnection(
                                "jdbc:mariadb://localhost/java", "root", "");
            // A Statement object represents an SQL command.
            Statement stmt = conn.createStatement();
            // A ResultSet object represents the command's result.
            ResultSet rs = stmt.executeQuery("SELECT COUNTER FROM data;");
            ) {
            if(rs.next()) { // Result non-empty => counter is already in DB.
                // Read counter from first column of query result:
                counter = rs.getInt(1);
            } else { // DB is empty.
                counter = 0;
                // Counter is initialized to 0 in row 1 of DB:
                stmt.executeUpdate("INSERT INTO data VALUES (1, 0)");
            }
        } catch(SQLException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        } // rs, stmt, and conn are closed automatically.
        listeners = new ArrayList<ModelListener>();
    }
```

Example: MVC GUI

```
    public void saveData() {
        try (// Connect to DB using URL and user name and empty password.
            Connection conn = DriverManager.getConnection(
                                    "jdbc:mariadb://localhost/java", "root", "");
            // A Statement object represents an SQL command.
            Statement stmt = conn.createStatement();
            ) {
            // Update counter in DB.
            stmt.executeUpdate("UPDATE data SET COUNTER = "
                                    + counter + " WHERE ID = 1;");
        } catch(SQLException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        } // stmt and conn are closed automatically.
    }
    ...
}
```

Everything else remains the same because all the data is in the **`Model`** class only!

Databases

Hints to make your life easier when using a database:

- Print all exceptions to `System.err` so you can easily see if there is any DB connection problem.
- Manually test all your SQL statements using `phpMyAdmin` before using them in your Java code.
- Remember to close all your resources (always use a "try-with-resources" statement).
- When using a remote DB through the network, make sure all the connection information is correct (host name, DB name, user name, password).

More information about databases (Oracle web site).

Summary

- Simple Input / Output
- File Input / Output
- Objects Input / Output
- Object Serialization
- Databases