Interesting Source Code Snippets

Retrieval by List

```
/**
 * Retrieves all Products from the Product table
 * @return A List of DisplayItem object's
 * @throws SQLException Error should a product not be found in the table
 */
public List<DisplayItem> getAllProducts() throws SQLException {
    ResultSet resultSet = null;
    ArrayList of products to store all selected products
    List<DisplayItem> products = new ArrayList<>();
    try {
        PreparedStatement pstat = connection.prepareStatement("SELECT productID, productName, description, price, qty, category, image_path FROM Product");
        //Assign resultSet the value of the query
        resultSet = pstat.executeQuery();

        Check if resultSet has a value
        while(resultSet.next()) {
            int id = resultSet.getInt("productID");
            String name = resultSet.getString("productName");
            String desc = resultSet.getString("description");
            double price = resultSet.getDouble("price");
            int qty = resultSet.getInt("qty");
            String image_path = resultSet.getString("image_path");

            DisplayItem product = new DisplayItem(id, name, desc, price, qty, image_path);
            products.add(product);
            }
        }
        catch(SQLException sqlException) {
            System.err.println("Error retrieving all Products from table : " + sqlException.getMessage());
            sqlException.printStackTrace();
        }

    return products;
}
```

Figure 1: getAllProducts() - ProductCrud

Lists are commonly used throughout the system, particularly when performing retrieval operations on the Product table. As seen in Figure 8- it highlights this retrieval process, where:

- The result of the select query is encapsulated within a DisplayItem object.
- Each DisplayItem gets added to a List of DisplayItem's, where each entry in the database is a DisplayItem, and each field is encapsulated within said item.
- The method returns the full list of products.

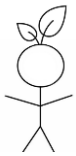
Displaying a List of Objects

```
public void getProducts(List<DisplayItem> products, ProductPanel p) {

        For each Item in the List of Items
    for (Item product : products){
        String image_path = product.getImgPath();
        ImageIcon icon = new ImageIcon(getClass().getResource(image_path));
        Image img = icon.getImage().getScaledInstance(250,  250,  Image.SCALE_SMOOTH);
        JLabel imgLabel = new JLabel(new ImageIcon(img));

        JLabel nameLabel = new JLabel(product.getItemName(), SwingConstants.CENTER);
        JLabel priceLabel = new JLabel("€" + product.getPrice(), SwingConstants.CENTER);
        priceLabel.setFont(PRODUCTFONT);
        nameLabel.setFont(new Font("Arial", Font.BOLD, 24));
        //Place each product into its own container, as we want to display the image,
        //product name and price
        JPanel productPanel = new JPanel(new BorderLayout());
        JPanel infoPanel = new JPanel(new BorderLayout());

            Container just for the text info
        infoPanel.add(nameLabel, BorderLayout.NORTH);
        infoPanel.add(priceLabel, BorderLayout.CENTER);

            Container for the image AND the text
        productPanel.add(imgLabel, BorderLayout.NORTH);
        productPanel.add(infoPanel, BorderLayout.CENTER);

        gridPanel.add(productPanel);
    }
```

Figure 2: getProducts - BrowsePanel

Figure 9 demonstrates how to display the List of DisplayItem's retrieved from Figure 8

- Firstly, a **for each** loop is utilised. In this case, for each Item in the DisplayItem list (DisplayItem extends the abstract class Item), extract the information from the DisplayItem
- Create a new JPanel container for each DisplayItem which holds the extracted information.
- The container makes use of a **BorderLayout,** allowing the individual pieces of product information to be displayed nicely within the container.
- Each container is added to the main JPanel **gridPanel**, which is the container for the **BrowsePanel** content area.



Event Handling and View/Control Interaction

```
    Add event for clicking on the image of a product, so user can visit a detailed screen of that product
imgLabel.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
            //Set the title of the product to that of the currently selected product by calling product.getItemName
        p.setTitle(product.getItemName());
        p.setImage(product.getImgPath());
        p.setDescription(product.getDescription());
        p.setPrice(product.getPrice());
        p.setItem(product);
        Once the title is set, switch the cardLayout to the "Procuct" card, with the new heading matching the produc
        CONTROL.handleSelectedProduct();
    }

});
```

Figure 3: getProducts - BrowsePanel

Figure 10 displays how an event is handled between the user's **mouse** and the **imgLabel** associated with a Product in the **BrowsePanel.**

The method mouseClicked gets triggered when the users clicks the imgLabel. On click the **ProductPanel** p – representing **the selected items detailed panel,** gets assigned all the GUI elements associated with the selected product.

The final piece of code displays the relationship between a view packaged class, which handles any GUI based logic, and a controller packaged class, which handless the more business end logic concerned with the GUI interface. In this instance **CONTROL** represents the final class level variable of type **BrowseControl**. BrowseControl is a custom controller class which contains the necessary functions to operate on the Browse section of the system.

The purpose of diversifying the two packages is to provide an easier platform for expansion in the future, as the abstraction makes it much easier to change information and keeps the system modular.

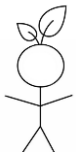
Filtering a Catalogue of Products

```
/**
 * Filters products in the ProductPanel by querying the database based on the value of the filterList selected item
 * @param p The current panel displaying the current state of catalogue
 */
public void handleFilter(ProductPanel p) {
filterBtn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            //CLear grid
            gridPanel.removeAll();
            //Get new products, passing the string of the selected item to the controller argument
            getProducts(CONTROL.filterCatalogue(filterList.getSelectedItem().toString()), p);
        } catch (SQLException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }

        //Revalidate screen
        revalidate();
        repaint();
        }

    });
}
```

Figure 4: handleFilter() - BrowsePanel

Figure 11 details the process of filtering the catalogue of items:

- A **JComboBox** is used to store the filters, which in this case are “Plant” and “Accessory” which are represented as “category” in the Product table.
- When the user selects a filter from the list and clicks the JButton **filterBtn,** a ActionEvent is triggered, calling **removeAll()** method on the gridPanel, removing all components from the container.
- **getProducts()** is then called with the argument **CONTROL.filterCatalogue(….)**
- This returns a list of items retrieved by querying the Product table based on the “category=?” where ? = the selected item from the JComboBox filterList.
- Finally, to dynamically display this new catalogue to the user, the revalidate() and repaint() methods are called.



Hashing Passwords

```java
public class PasswordHasher {

    /**
     * Performs thee SHA-256 hashing algorithm on the users input password
     * @param password The password input by the user
     * @return The hexadecimal of the function, converted to a string
     * @throws Exception thrown if an error occurs when attempting to perform the SHA-256 algorithm
     */
    public static String hashPassword(String password) throws Exception {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hash = digest.digest(password.getBytes(StandardCharsets.UTF_8));

        StringBuilder hexString = new StringBuilder();
        for (byte b : hash) {
            hexString.append(String.format("%02x", b));
        }

        return hexString.toString();
    }
```

Figure 5: Hashing

MessageDigest allows us to make use of the functionality of the SHA-256 algorithm. **Hashing** then follows the following steps:

- Convert the password string into an array of bytes
- The output of that array of bytes gets supplied to the argument **digest,** which finalises the hashing process by performing operations such as padding.
- The result of that function, is then ran through a for each loop, for each byte in the result, convert it back to a string, formatting each byte as a hexadecimal number

Writing to an Error File

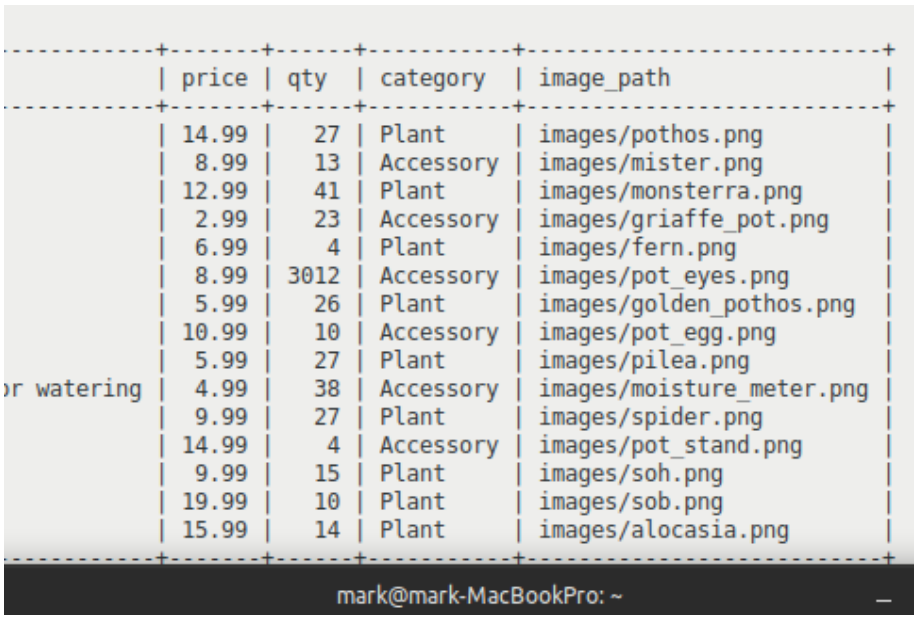
```
            catch (ValidationException e4)
            {
                handleError(e4, "Validation Error");
//              e4.printStackTrace();
            }

        }
    });
}

/**
 * Helper variable used to display error messages to the user via a JOptionPane and write the error to an error file
 * @param errorType The type of error that occured
 * @param e The Exception occured
 */
private void handleError(Exception e, String errorType) {
    JOptionPane.showMessageDialog(CreateAccountPanel.this, e.getMessage(), errorType, JOptionPane.ERROR_MESSAGE);
    GrowingPains.errorWriter.logError(errorType, e.getMessage());
}
```

Figure 6: Error Writing

Figure 13shows a sample throw of a custom Exception **ValidationException**, which is thrown when a field is left blank in the account creation process. When the exception gets caught:

- The handleError method is invoked, with the Exception as an argument and a string describing the type of error.
- **handleError** creates a popup to alert the user of the error
- A static instance of an **ErrorWriter** object located in the GrowingPains class is then called with the logError() method to write to a file which is opened on system load.

Method to Store Images

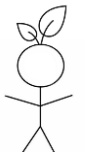


	price	qty	category	image_path
	14.99	27	Plant	images/pothos.png
	8.99	13	Accessory	images/mister.png
	12.99	41	Plant	images/monsterra.png
	2.99	23	Accessory	images/griaffe_pot.png
	6.99	4	Plant	images/fern.png
	8.99	3012	Accessory	images/pot_eyes.png
	5.99	26	Plant	images/golden_pothos.png
	10.99	10	Accessory	images/pot_egg.png
	5.99	27	Plant	images/pilea.png
or watering	4.99	38	Accessory	images/moisture_meter.png
	9.99	27	Plant	images/spider.png
	14.99	4	Accessory	images/pot_stand.png
	9.99	15	Plant	images/soh.png
	19.99	10	Plant	images/sob.png
	15.99	14	Plant	images/alocasia.png

mark@mark-MacBookPro: ~

Figure 7: image_path in Product table

Observing Figure 14, noting the image_path, each product contains the relative path to a .png image, which relates to the plant. The images are located within a folder named **images** within the **view** folder.



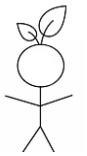
CardLayout as a Central Navigation Tool

```
EDIT ACCOUNT Button
editAccountBtn = createButton("Account","images/account.png");
sideBar.add(editAccountBtn);
editAccountBtn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            checkLoggedIn("Edit Account");
            //customer = login.getLoggedInCustomer(); // Get current customer

            // Create edit panel with current customer
            edit = new EditAccountPanel(customer);
            mainContent.add(edit, "Edit Account");
            cardLayout.show(mainContent, "Edit Account");

            // No need to get updated customer here - it will be updated in the panel
        } catch (UserNotLoggedInException e1) {
            handleError(e1);
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    }
});
```

Figure 8: CardLayout

Figure 15 outlines the relationship of a CardLayout manager being used to switch to a new JPanel.

- The CardLayout works exactly like a deck of cards, where you can call the add() method to add a card to the deck, in this case the card being a JPanel container.
- A second argument is supplied to add(), a String indexing the added JPanel enabling you to easily reference the “card” when calling the show() method.
- The show() method brings the JPanel with the matching index to the foreground.



Driver and main() method

```
package controller;
//GROWING PAINS - A Plant Shop system
//Mark Lambert - C00192497 - Object Oriented Software Development 2 - Year 2 Semester 2

import java.sql.SQLException;
/**
 * Driver for GrowingPains application
 */
public class GrowingMains {

    /**
     * Main
     */
        public static void main(String[] args) {
            try {
                //New instance of GrowingPains application
                GrowingPains g = new GrowingPains();
                //Set GrowingPains to visible
                g.run();
            } catch (SQLException e) {
                e.printStackTrace();
            }

        }
}
```

Figure 9: Driver class

Figure 16 shows the benefits of the Object-Oriented approach to Software Development. The driver class for OPSS contains roughly 10 lines of code excluding comments, allowing for a much more manageable and extensible codebase for future iterations.