Rohan Muppa
Ms. Nishiwaki
IB Computer Science SL
March 20, 2023

Criterion C:

Techniques:

- UI Framework
    - JavaFX GridPanes and Labels
    - Data validation through Try and Catch blocks
- Saving and loading plant data
    - File I/O
    - Encapsulation
- Garden Algorithms
    - Linear search
    - String builder
- Data structures
    - HashMaps
    - Lists
        - ObservableLists
        - Arraylists

UI Framework:

- GreenGarden uses JavaFX to build its GUI with UI components like buttons, text fields, and labels. Users can input their garden details such as location, sun exposure, maintenance level, budget, and plant types via text fields and dropdown menus.
- Gridpanes are used to create a structured layout for the UI components in GreenGarden. Labels are used to provide descriptive text for the UI components in GreenGarden. This is important because it helps users to understand what each component is for and how to interact with it.
- The input is verified for proper format and range, and a try-catch block is used to handle potential NumberFormatException. It uses the showAlert class to display an error message to the user if garden size, location, or budget is not a positive integer. (Appendix A, Full JavaFx Programming Course)

Figure 1: UI with invalid inputs and “Please enter valid inputs” alert

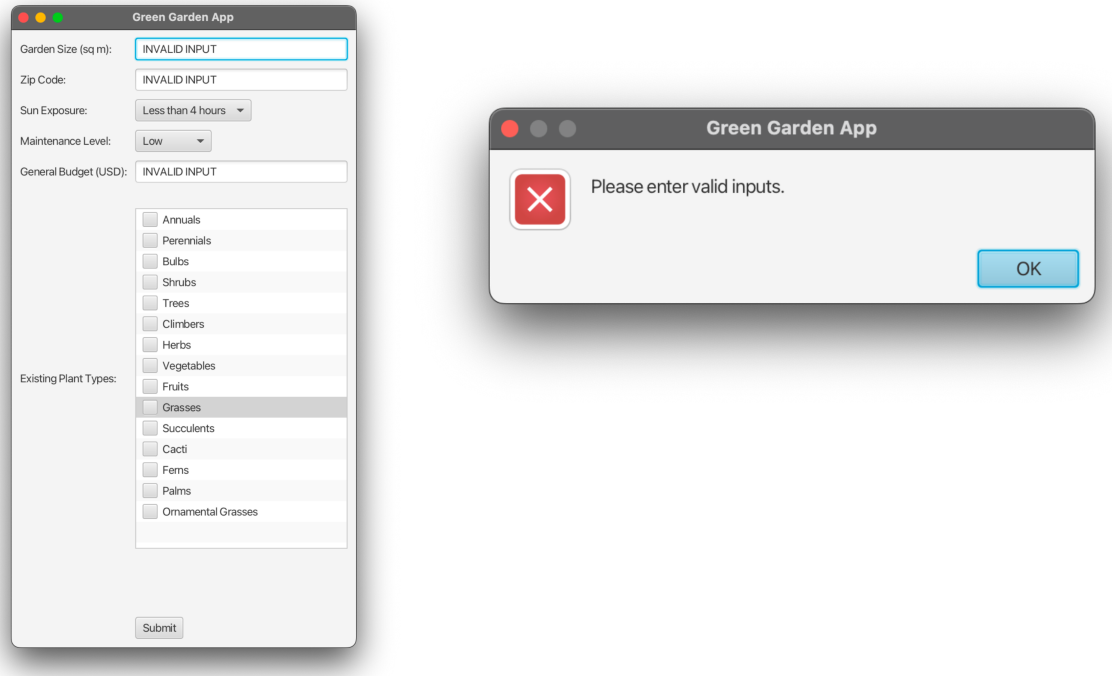


Figure 2: Try and Catch block for input validation

A try-catch block is a Java statement that allows one to define and run a piece of code but, if an error occurs, allows one to run another piece of code, usually telling the user to enter valid inputs.

The submit button below uses a try-catch block to validate user inputs entered with UI elements. If an exception is thrown while parsing any of the inputs, the catch block displays an error message to the user. The try-catch block ensures that GreenGarden can handle invalid inputs without crashing.

```
submitButton.setOnAction(event -> {
            try {
                // Parse user inputs
                int size = Integer.parseInt(sizeTextField.getText());
                int zipCode = Integer.parseInt(zipCodeTextField.getText());
                int sunExposure = sunExposureChoiceBox.getSelectionModel().getSelectedIndex() + 1;
                int maintenanceLevel = maintenanceChoiceBox.getSelectionModel().getSelectedIndex() + 1;

                int budget = Integer.parseInt(budgetTextField.getText());

                ChoiceBox<String> maintenanceLevelChoiceBox = new ChoiceBox<>();
                maintenanceLevelChoiceBox.getItems().addAll( ...es: "Low", "Moderate", "High");
                maintenanceLevelChoiceBox.setValue("Low");

                // Get selected plant types
                ObservableList<String> existingPlantTypes = FXCollections.observableArrayList();
                for (int i = 0; i < typeSelected.length; i++) {
                    if (typeSelected[i].get()) {
                        existingPlantTypes.add(plantTypes.get(i));
                    }
                }


                // create new garden object
                Garden garden = new Garden(size, zipCode, sunExposure, budget, existingPlantTypes, maintenanceLevel);

                List<Plant> selectedplants = garden.getNewPlants();

                // generate report
                String report = garden.generateReport();
                String tips = garden.getGardenRecommendations();



                // show report
                Alert alert = new Alert(Alert.AlertType.INFORMATION);
                alert.setTitle("Green Garden App");
                alert.setHeaderText(null);
                alert.setContentText(report);
                alert.showAndWait();

                alert.setTitle("Green Garden App");
                alert.setHeaderText(null);
                alert.setContentText(garden.createMaintenanceSchedule(maintenanceLevel));
                alert.showAndWait();

                alert.setTitle("Green Garden App");
                alert.setHeaderText(null);
                alert.setContentText(garden.listPlantNames());
                alert.showAndWait();

            } catch (NumberFormatException e) {
                showAlert( s: "Please enter valid inputs.", Alert.AlertType.ERROR);
            }
        });
```

Figure 3: showAlert Method

```
private void showAlert(String s, Alert.AlertType error) {
    Alert alert = new Alert(error);
    alert.setTitle("Green Garden App");
    alert.setHeaderText(null);
    alert.setContentText(s);
    alert.showAndWait();
}
```

File I/O:

For the data storage method, a CSV file is used. The plant object's ID, name, water and sunshine requirements, price, and type are all details. This is useful to my program because it allows the **selectPlants()** method to create a list of plants to filter through.

Figure 4: Implementation of readPlantsFromCSVFile in selectPlants() method

```
List<Plant> plantsToFilter = readPlantsFromCsvFile( path: "/Users/Rohan/Desktop/GreenGarden/Plants.csv"); // read plants from CSV file
```

Figure 5: readPlantsFromCSVFile() method

```
private List<Plant> readPlantsFromCsvFile(String path) {
    List<Plant> plants = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        String line;
        while ((line = br.readLine()) != null) {
            String[] data = line.split( regex: ",");
            String name = data[1];
            int sunNeeded = Integer.parseInt(data[2]);
            int waterNeeded = Integer.parseInt(data[3]);
            double cost = Double.parseDouble(data[4]);
            int zipcode = Integer.parseInt(data[5]);
            String type = data[6];
            Plant plant = new Plant(name, sunNeeded, waterNeeded, cost, zipcode, type);
            plants.add(plant);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return plants;
}
```

The CSV file is read using the openCSV module, and the fields are divided using the **Buffered()** function. The data from each line is then utilized to construct a new Plant object and add it to a list of all accessible plants. By doing so, I can refer to Plant information through **encapsulation** (Appendix A, What Is Encapsulation in Java and How to Implement It).

The **openCSV module** is used to read the CSV file, and the **Buffered()** function is then used to separate the fields. A new Plant object is then made using the details from each line and added to a list of all the available plants. This file input makes it easy to parse large quantities of data when using only a single object, as opposed to things such as SQL that are more beneficial to more complex datasets. Unlike approaches like using SQL, this file input makes it simple to parse enormous amounts of data using only one object. Furthermore, the CSV format is easy to

work with and can be edited using a variety of tools, making it accessible to both developers and end-users (Appendix A, Reading a CSV File in Java Using OpenCSV).

Garden Algorithms:

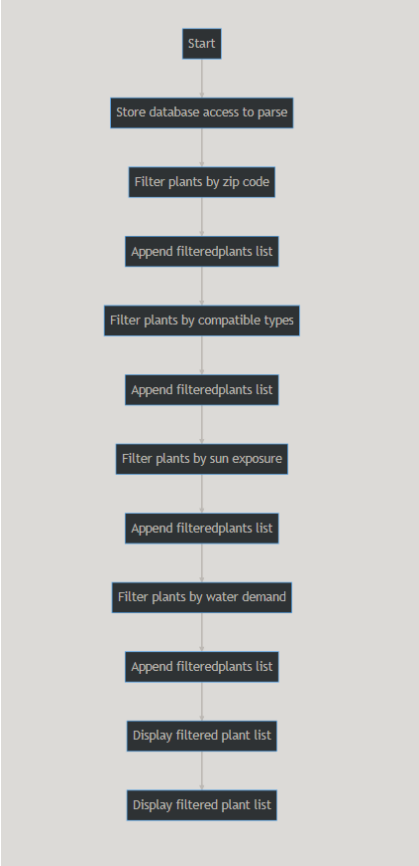
Linear Search:

- **Plant search algorithm:** Linear search is used to loop through each plant in the database. The plants were then appended to a class variable list, newPlants. A global variable was used because a local variable would be inaccessible outside the function and ArrayLists had trouble returning non-null values (Appendix A, Linear Search in Java - Scaler Topics).

Figure 6: selectPlants() method

```
public void selectPlants(int sunExposure, int waterNeeded, int budget) {
    double totalCost = 0.0;
    List<Plant> plantsToFilter = readPlantsFromCsvFile( path: "/Users/Rohan/Desktop/GreenGarden/Plants.csv"); // read plants from CSV file

    for (int i = 0; i < plantsToFilter.size(); i++) {
        Plant plant = plantsToFilter.get(i);
        // Filter by nearby zip codes
        if (Math.abs(plant.getZipcode() - this.zipCode) > 500) {
            continue;
        }

        // Filter by compatibility with existing plants
        boolean isCompatible = true;
        for (int j = 0; j < existingPlantTypes.size(); i++) {
            if (!isCompatibleWith(plant, existingPlantTypes, plantsToFilter)) {
                isCompatible = false;
                break;
            }
        }

        // Filter by sun exposure
        int plantSunExposure = plant.getSunExposure();
        if (sunExposure >= plantSunExposure){
            continue;
        }

        // Filter by maintenance needed
        if (waterNeeded <= plant.getWaterDemand()){
            continue;
        }


        //Calculate totalCost and stop when budget is exceeded
        totalCost += plant.getCost();
        newPlants.add(plant);
        if (totalCost > budget) {
            newPlants.remove(plant);
            totalCost = totalCost - plant.getCost();
        }
    }
}
```

Figure 7: selectPlants() flowchart



StringBuilder:

- StringBuilder objects are mutable, variable-length arrays of characters that can be modified through method invocations, allowing their length and content to change dynamically during runtime (Appendix A, StringBuilder (Java Platform SE 7))

Figure 8: StringBuilder in generateReport()

```
1 usage
public String generateReport() {
    selectPlants(sunExposure,maintenanceLevel,budget);
    WaterCalculator calculator = new WaterCalculator(sunExposure, size);

    double costOfPlants = getTotalCostOfPlants();
    double co2Sequestered = getCO2Sequestered();
    double waterSaved = calculator.calculateWaterSavingsGallons();
    double waterSavingsPercent = calculator.calculateWaterSavingsPercent();
    StringBuilder reportBuilder = new StringBuilder();
    reportBuilder.append("Environmental Report:\n\n");
    reportBuilder.append("- Total Cost of Plants: $" + String.format("%.2f", costOfPlants) + "\n");
    reportBuilder.append("- CO2 Sequestered: " + String.format("%.2f", co2Sequestered) + " kg\n");
    reportBuilder.append("- Water Saved: " + String.format("%.2f gallons\n", waterSaved));
    reportBuilder.append(String.format("- Water Savings: %.2f%%\n", waterSavingsPercent));
    reportBuilder.append(getGardenRecommendations());
    return reportBuilder.toString();
}
```

- generateReport() uses the StringBuilder to display the outputs. The outputs are returned from other methods that also return a string of a StringBuilder.

Figure 9: StringBuilder in getGardenRecommendations()

```
public String getGardenRecommendations() {
    StringBuilder sb = new StringBuilder();

    Map<String, Integer> plantCounts = new HashMap<>();

    // Count existing plants in garden
    for (int i = 0; i < existingPlantTypes.size(); i++) {
            String plantType = existingPlantTypes.get(i);
            plantCounts.put(plantType, plantCounts.getOrDefault(plantType, defaultValue: 0) + 1);
        }

    // Count new plants to be added
    for (int i = 0; i < newPlants.size(); i++) {
        Plant plant = newPlants.get(i);
        if (plant != null) {
            String plantType = plant.getType();
            plantCounts.put(plantType, plantCounts.getOrDefault(plantType, defaultValue: 0) + 1);
        }
    }

    sb.append("\n\nBased on the types of plants in your garden, we recommend:\n");
    if (plantCounts.getOrDefault( key: "Ferns", defaultValue: 0) > 12) {
        // Recommend using rainwater to water ferns as they prefer soft water
        sb.append("- Collecting rainwater to water your ferns as they prefer soft water.\n");
    }
    if (plantCounts.getOrDefault( key: "Palms", defaultValue: 0) > 3) {
        // Recommend using a slow-release fertilizer for palm trees to avoid overfertilization
        sb.append("- Using a slow-release fertilizer for your palm trees to avoid overfertilization.\n");
    }
    if (plantCounts.getOrDefault( key: "Grasses", defaultValue: 0) > 10) {
        // Recommend using organic fertilizer for ornamental grasses to avoid chemical buildup
        sb.append("- Using organic fertilizer for your ornamental grasses to avoid chemical buildup.\n");
    }
    if (plantCounts.getOrDefault( key: "Climbers", defaultValue: 0) > 5) {
        // Recommend using trellises to support climbers and prevent damage to walls and fences
        sb.append("- Using trellises to support your climbers and prevent damage to walls and fences.\n");
    }
    if (plantCounts.getOrDefault( key: "Fruits", defaultValue: 0) > 4) {
        // Recommend using natural pest control methods for fruit trees to avoid harmful chemicals
        sb.append("- Using natural pest control methods for your fruit trees to avoid harmful chemicals.\n");
    }
    if (plantCounts.getOrDefault( key: "Annuals", defaultValue: 0) > 20) {
        // Recommend using organic fertilizer for annuals to avoid chemical buildup
        sb.append("- Using organic fertilizer for your annuals to avoid chemical buildup.\n");
    }
    if (plantCounts.getOrDefault( key: "Perennials", defaultValue: 0) > 15) {
        // Recommend using companion planting to improve soil health and reduce pest problems for perennials
        sb.append("- Using companion planting to improve soil health and reduce pest problems for your perennials.\n");
    }
    if (plantCounts.getOrDefault( key: "Bulbs", defaultValue: 0) > 25) {
        // Recommend using bone meal as a natural fertilizer for bulbs
        sb.append("- Using bone meal as a natural fertilizer for your bulbs.\n");
    }
    if (plantCounts.getOrDefault( key: "Ornamental Grasses", defaultValue: 0) > 8) {
        // Recommend trimming ornamental grasses to a height of 2-3 feet in late winter or early spring
        sb.append("- Trimming your ornamental grasses to a height of 2-3 feet in late winter or early spring.\n");
    }
```

```
    // Recommend sustainable practices based on plant types
    if (plantCounts.containsKey("Vegetables") || plantCounts.getOrDefault( key: "Herbs",  defaultValue: 0) > 18) {
        // Recommend starting a compost bin to reduce waste and improve soil health
        sb.append("- Starting a compost bin to generate organic fertilizer for your vegetable and herb plants.\n");
    }
    else if (plantCounts.containsKey("Trees") || plantCounts.containsKey("Shrubs")) {
        // Recommend using mulch to conserve water and reduce weed growth around trees and shrubs
        sb.append("- Using mulch around your trees and shrubs to conserve water and reduce weed growth.\n");
    }
    else if (plantCounts.containsKey("Succulents") || plantCounts.containsKey("Cacti")) {
        // Recommend using a drip irrigation system to conserve water and prevent overwatering of succulents and cacti
        sb.append("- Using a drip irrigation system to water your succulents or cacti to prevent overwatering.\n");
    }
    return sb.toString();
}
```

Data Structures:

HashMaps

- **HashMaps** are data structures used to store key-value pairs for efficient retrieval. A value stored in a hash map is retrieved using the key under which it was stored. This data structure is used in the **Garden** class and is used to determine the compatibility between the existing plant types and the new plant types (of recommended plants).

- The **HashMap** is created with the plant types as keys and their compatible plant types (of the recommended plants) as values. The **isCompatibleWith()** function takes in the new plant, a list of existing plants, and a list of filtered plants as inputs. It then looks up the compatibility of the new plant type with the existing plant types using the **HashMap**. If the new plant type is compatible with any of the existing plant types, the function returns true, indicating that the plant is compatible.

Figure 10: Implementation of isCompatible() method

```
public boolean isCompatibleWith(Plant plant, ObservableList<String> existingPlants, List<Plant> filteredPlants) {
    //Shows the compatibility between the existing plant types (more general) to the specific plant types of the new plants
    HashMap<String, List<String>> compatibility = new HashMap<>();
    compatibility.put("Annuals", new ArrayList<>(List.of("Petunia", "Marigold", "Zinnia")));
    compatibility.put("Perennials", new ArrayList<>(List.of("Hosta", "Lavender", "Daylily")));
    compatibility.put("Bulbs", new ArrayList<>(List.of("Daffodil", "Tulip", "Crocus")));
    compatibility.put("Shrubs", new ArrayList<>(List.of("Azalea", "Lilac", "Rhododendron")));
    compatibility.put("Trees", new ArrayList<>(List.of("Maple", "Oak", "Birch")));
    compatibility.put("Climbers", new ArrayList<>(List.of("Clematis", "Ivy", "Honeysuckle")));
    compatibility.put("Herbs", new ArrayList<>(List.of("Basil", "Rosemary", "Thyme")));
    compatibility.put("Vegetables", new ArrayList<>(List.of("Tomato", "Cucumber", "Pepper")));
    compatibility.put("Fruits", new ArrayList<>(List.of("Strawberry", "Blueberry", "Raspberry")));
    compatibility.put("Grasses", new ArrayList<>(List.of("Bamboo", "Ornamental grass", "Pampas grass")));
    compatibility.put("Succulents", new ArrayList<>(List.of("Aloe vera", "Jade plant", "Cactus")));
    compatibility.put("Cacti", new ArrayList<>(List.of("Barrel cactus", "Christmas cactus", "Prickly pear")));
    compatibility.put("Ferns", new ArrayList<>(List.of("Boston fern", "Maidenhair fern", "Bird's nest fern")));
    compatibility.put("Palms", new ArrayList<>(List.of("Areca palm", "Date palm", "Coconut palm")));
    compatibility.put("Ornamental Grasses", new ArrayList<>(List.of("Japanese blood grass", "Pampas grass", "Fountain grass")));

    String newPlantType = plant.getType();

    // Check compatibility with existing plants
    for (int i = 0; i < existingPlants.size(); i++) {
        String existingPlantType = existingPlants.get(i);
        if (compatibility.containsKey(existingPlantType)) {
            List<String> compatibleTypes = compatibility.get(existingPlantType);
            if (!compatibleTypes.contains(newPlantType)) {
                return false;
            }
        }
    }
```

```
    // Check compatibility with other plants in filteredPlants list
    for (int i = 0; i < filteredPlants.size(); i++) {
        String filteredPlantType = filteredPlants.get(i).getType();
        if (compatibility.containsKey(filteredPlantType)) {
            List<String> compatibleTypes = compatibility.get(filteredPlantType);
            if (!compatibleTypes.contains(newPlantType)) {
                return false;
            }
        }
    }

    return true;
}
```

- Inside this method, a new **HashMap** called **plantCounts** is created to keep track of the number of plants of each type in the user's garden

- A loop then iterates over each plant in the user's garden. For each plant, its type is extracted and the corresponding count in the **plantCounts HashMap** is incremented by 1 using the put method. If the plant type is not already in the **plantCounts HashMap**, a default value of 0 is used instead.

- Finally, the **plantCounts HashMap** is used to generate recommendations for new plants based on the existing plant types. The method checks each plant in the **newPlants** list and provides textual recommendations based on the type of plants in the garden. The recommendations are then added to the **StringBuilder** as a string and returned (Appendix A, HashMap in Java with Examples).

Lists:

- In the **Main** and **Garden** classes, lists are used to store lists of the **Plant** object. These lists are passed as parameters in various methods. Also stored in lists were the Strings of the various types of plants. For example, in the Garden class, the existing plant types are stored in an ObservableList<String>.
    - For example, the **existingPlants** list stores the plants that are currently in the garden and is passed as a parameter to the **isCompatibleWith()** method in order to determine if a new plant is compatible with the existing plants
- The lists in **newPlants** are implemented as ArrayLists because of their dynamic nature, since their sizes can change with user activity as they are resizable (Appendix A, ArrayList (Java Platform SE 8)).

Figure 11: newPlants ArrayList of Plant

```
public ArrayList<Plant> newPlants = new ArrayList<>();
```

- An **ObservableList**, a special type of list in JavaFX that allows other objects to observe changes to the list and respond accordingly, is used in the Garden class to keep track of

the type plants that the user inputted. The Garden class in GreenGarden uses an ObservableList to track user-inputted plant types, allowing the app to dynamically update the UI and pass the list to other methods. If a new plant type is added, the ObservableList automatically updates the UI, whereas an ArrayList requires manual updates, as there are no listeners. **newPlants** does not require updating the UI so it uses a simple ArrayList (Appendix A, ObservableList (JavaFX 8)).

Figure 12: existingPlantTypes ObservableList of String

Word Count: 1177