Car File Processor

This is a simple Java Swing desktop application designed to load car data from XML and CSV files, then allow users to process, filter, and sort this data based on various criteria.

Table of Contents

Features

- **Load Data**: Import car data from both CSV and XML files.
- **Process All**: Display all currently loaded car data.
- **Filter Data**: Apply filters based on car Brand, Type, Model, Price, Main Currency, and Release Date.
- **Sort Data**: Sort the displayed car data by various fields (e.g., Brand, Price, Release Date) in ascending or descending order.
- **Dynamic UI**: Filter and Sort panels are hideable/showable for a cleaner interface.
- **Secure XML Parsing**: Implements measures to prevent XML External Entity (XXE) vulnerabilities during XML file loading.
- **Robust Date Parsing**: Handles multiple date formats for consistency.

Project Structure

```
.
├── pom.xml
└── src
    └── main
        └── java
            └── org
                └── blank
                    ├── Main.java
                    ├── factory
                    │   └── Factory.java
                    ├── loader
                    │   └── CarFileLoader.java
                    ├── repository
                    │   ├── CarRepository.java
                    │   └── model
                    │       └── Car.java
                    ├── service
                    │   └── CarService.java
                    └── util
                        └── Utils.java
```

Requirements

- Java Development Kit (JDK) 11 or higher (configured in pom.xml to 11).
- Maven (for building the project and managing dependencies).

Building and Running

From Source (IDE)

1. **Clone the repository** (if applicable) or ensure all Java files are in their respective package folders.
2. **Open the project** in your preferred IDE (IntelliJ IDEA, Eclipse, VS Code with Java extensions).
3. **Import as a Maven project** if prompted by your IDE. This will download all necessary dependencies.
4. **Run Main.java**: Locate the Main class in org.blank.Main and run its main method.

As a Runnable JAR

To create a self-contained executable JAR (an "uber JAR") that includes all

dependencies:

1. **Ensure maven-shade-plugin is configured** in your pom.xml as follows (replace org.blank.Main with your actual main class if it differs):

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.13.0</version>
      <configuration>
        <source>11</source>
        <target>11</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.6.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <transformers>
              <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTra
nsformer">
                <mainClass>org.blank.Main</mainClass>
              </transformer>
              <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTra
nsformer"/>
            </transformers>
            <filters>
              <filter>
                <artifact>*:*</artifact>
```

```
                    <excludes>
                        <exclude>META-INF/*.SF</exclude>
                        <exclude>META-INF/*.DSA</exclude>
                        <exclude>META-INF/*.RSA</exclude>
                    </excludes>
                </filter>
            </filters>
        </configuration>
    </execution>
</executions>
</plugin>
</plugins>
</build>
```

2. Build the project using Maven:
   Open your terminal/command prompt in the project's root directory (where pom.xml is located) and run:
   mvn clean package

   This will create a JAR file (e.g., your-app-name-1.0-SNAPSHOT-shaded.jar) in the target/ directory.
3. Run the JAR file:
   Navigate to the target/ directory in your terminal and execute:
   java -jar your-app-name-1.0-SNAPSHOT-shaded.jar

   (Replace your-app-name-1.0-SNAPSHOT-shaded.jar with the actual file name).

Usage

1. **Load Data**:
   - Click "Load XML" to select an XML file containing car data.
   - Click "Load CSV" to select a CSV file containing car data.
   - After loading both, click "Process File" to merge and display all loaded cars.
2. **Filter Data**:
   - Click the "Filter" button to show/hide the filter panel.
   - Enter desired values in the filter fields (e.g., "Toyota" for Brand, "Sedan" for Type, "2023-01-01" for Release Date).
   - Click "Process with Filters" inside the filter panel to apply the criteria and display filtered results.
3. **Sort Data**:

- Click the "Sort By" button to show/hide the sort panel.
- Select a "Sort Field" from the dropdown (e.g., "Brand", "Price", "Release Date").
- Select an "Order" (Ascending or Descending).
- Click "Apply Sort" to sort the currently displayed data.

Code Documentation

Main.java

The entry point of the application, responsible for setting up the Swing GUI, arranging components using GridBagLayout, and attaching action listeners to buttons.

```
package org.blank;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.GridLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.WindowConstants;
import org.blank.factory.Factory;
import org.blank.service.CarService;

/**
 * Main class for the Car File Processor application.
 * Initializes the Swing GUI, sets up event listeners, and manages the visibility
 * and layout of various application panels.
 */
public class Main {

  /**
   * The main method that starts the Car File Processor application.
   * It sets up the main JFrame, adds panels and buttons for file loading,
   * filtering, and sorting, and attaches action listeners to handle user interactions.
```

```
 *
 * @param args Command line arguments (not used).
 */
public static void main(String[] args) {
  JFrame frame = new JFrame("Car File Processor");
  frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
  frame.setSize(800, 400); // Initial frame size, will be packed later

  JPanel mainPanel = new JPanel();
  mainPanel.setLayout(new GridBagLayout());
  GridBagConstraints gbc = new GridBagConstraints();
  gbc.insets = new Insets(5, 5, 5, 5); // Padding around components

  // Initialize UI components
  JButton loadXml = new JButton("Load XML");
  JButton loadCsv = new JButton("Load CSV");
  JButton processFileButton = new JButton("Process File");
  JButton processWithFiltersButton = new JButton("Process with Filters");
  JButton filterButton = new JButton("Filter");
  JButton sortButton = new JButton("Sort By");

  JTextField xmlPathField = new JTextField(40);
  JTextField csvPathField = new JTextField(40);

  // Initialize and configure the result area (JTextArea)
  // The JTextArea instance is managed by the Factory class.
  Factory.getResultArea().setEditable(false);


  // --- Filter Panel Setup ---
  JPanel filterPanel = new JPanel();
  // 7 label/textfield rows + 1 row for the "Process with Filters" button
  filterPanel.setLayout(new GridLayout(8, 2));
  filterPanel.setVisible(false); // Initially hidden

  JTextField brandFilter = new JTextField();
  JTextField typeFilter = new JTextField();
  JTextField modelFilter = new JTextField();
  JTextField priceFilter = new JTextField();
```

```
    JTextField currencyFilter = new JTextField();
    JTextField releaseDateFilter = new JTextField();
    JTextField additionalPriceFilter = new JTextField(); // Added for completeness based
on Car model

    filterPanel.add(new JLabel("Brand:"));
    filterPanel.add(brandFilter);
    filterPanel.add(new JLabel("Type:"));
    filterPanel.add(typeFilter);
    filterPanel.add(new JLabel("Model:"));
    filterPanel.add(modelFilter);
    filterPanel.add(new JLabel("Price:"));
    filterPanel.add(priceFilter);
    filterPanel.add(new JLabel("Main Currency:"));
    filterPanel.add(currencyFilter);
    filterPanel.add(new JLabel("Release Date (yyyy-MM-dd):"));
    filterPanel.add(releaseDateFilter);
    filterPanel.add(new JLabel("Additional Price:"));
    filterPanel.add(additionalPriceFilter);

    filterPanel.add(new JLabel("")); // Placeholder for left column of button row
    filterPanel.add(processWithFiltersButton); // "Process with Filters" button is part of
this panel


    // --- Sort Panel Setup ---
    JPanel sortPanel = new JPanel();
    // Using GridBagLayout for sortPanel for flexible button placement
    sortPanel.setLayout(new GridBagLayout());
    GridBagConstraints sortGbc = new GridBagConstraints();
    sortGbc.insets = new Insets(2, 5, 2, 5); // Smaller padding for sort elements
    sortPanel.setVisible(false); // Initially hidden

    String[] sortFields = {
      "None", "Brand", "Type", "Model", "Price", "Main Currency", "Release Date",
"Additional Price"
    };
    JComboBox<String> sortFieldComboBox = new JComboBox<>(sortFields);
```

```
String[] sortOrders = {"Ascending", "Descending"};
JComboBox<String> sortOrderComboBox = new JComboBox<>(sortOrders);

JButton applySortButton = new JButton("Apply Sort"); // Button to trigger the sort

// Add components to sortPanel using sortGbc
sortGbc.gridx = 0; sortGbc.gridy = 0;
sortGbc.anchor = GridBagConstraints.WEST;
sortPanel.add(new JLabel("Sort Field:"), sortGbc);

sortGbc.gridx = 1;
sortGbc.fill = GridBagConstraints.HORIZONTAL;
sortGbc.weightx = 0.5;
sortPanel.add(sortFieldComboBox, sortGbc);

sortGbc.gridx = 2; sortGbc.gridy = 0;
sortGbc.anchor = GridBagConstraints.WEST;
sortGbc.fill = GridBagConstraints.NONE;
sortGbc.weightx = 0;
sortPanel.add(new JLabel("Order:"), sortGbc);

sortGbc.gridx = 3;
sortGbc.fill = GridBagConstraints.HORIZONTAL;
sortGbc.weightx = 0.5;
sortPanel.add(sortOrderComboBox, sortGbc);

sortGbc.gridx = 4; sortGbc.gridy = 0;
sortGbc.gridwidth = 1;
sortGbc.anchor = GridBagConstraints.EAST;
sortGbc.fill = GridBagConstraints.NONE;
sortGbc.weightx = 0;
sortPanel.add(applySortButton, sortGbc);


// --- Layout of mainPanel Components (using GridBagLayout) ---

// XML row (gridy=0)
gbc.gridx = 0; gbc.gridy = 0;
gbc.fill = GridBagConstraints.NONE; gbc.anchor = GridBagConstraints.WEST;
```

```
    mainPanel.add(loadXml, gbc);

    gbc.gridx = 1; gbc.anchor = GridBagConstraints.WEST;
    mainPanel.add(new JLabel("XML File Path:"), gbc);

    gbc.gridx = 2; gbc.weightx = 1.0; gbc.fill = GridBagConstraints.HORIZONTAL;
    mainPanel.add(xmlPathField, gbc);

    // CSV row (gridy=1)
    gbc.gridx = 0; gbc.gridy = 1;
    gbc.weightx = 0; gbc.fill = GridBagConstraints.NONE; gbc.anchor =
GridBagConstraints.WEST;
    mainPanel.add(loadCsv, gbc);

    gbc.gridx = 1; gbc.anchor = GridBagConstraints.WEST;
    mainPanel.add(new JLabel("CSV File Path:"), gbc);

    gbc.gridx = 2; gbc.weightx = 1.0; gbc.fill = GridBagConstraints.HORIZONTAL;
    mainPanel.add(csvPathField, gbc);

    // Process File button (gridy=2, spans full width)
    gbc.gridx = 0; gbc.gridy = 2;
    gbc.gridwidth = 3; gbc.weightx = 0; gbc.fill = GridBagConstraints.NONE; gbc.anchor
= GridBagConstraints.WEST;
    mainPanel.add(processFileButton, gbc);

    // Filter button (gridy=3, spans full width)
    gbc.gridx = 0; gbc.gridy = 3;
    gbc.gridwidth = 3; gbc.weightx = 0; gbc.fill = GridBagConstraints.NONE; gbc.anchor
= GridBagConstraints.WEST;
    mainPanel.add(filterButton, gbc);

    // Filter panel (gridy=4, below filter button, spans full width)
    gbc.gridx = 0; gbc.gridy = 4;
    gbc.gridwidth = 3;
    mainPanel.add(filterPanel, gbc);

    // Sort By button (gridy=5, below filter panel, spans full width)
    gbc.gridx = 0; gbc.gridy = 5;
```

```
    gbc.gridwidth = 3; gbc.weightx = 0; gbc.fill = GridBagConstraints.NONE; gbc.anchor
= GridBagConstraints.WEST;
    mainPanel.add(sortButton, gbc);

    // Sort panel (gridy=6, below Sort By button, spans full width)
    gbc.gridx = 0; gbc.gridy = 6;
    gbc.gridwidth = 3;
    mainPanel.add(sortPanel, gbc);

    // Result area (gridy=7, shifted down, spans full width, takes remaining space)
    gbc.gridx = 0; gbc.gridy = 7;
    gbc.gridwidth = 3; gbc.fill = GridBagConstraints.BOTH; gbc.weightx = 1.0;
gbc.weighty = 1.0;
    mainPanel.add(new JScrollPane(Factory.getResultArea()), gbc);

    // Add main panel to frame and finalize
    frame.add(mainPanel);
    frame.setVisible(true);
    frame.pack(); // Pack the frame after all components are added


    // --- Action Listeners for Buttons ---

    /**
     * Listener for the "Load CSV" button.
     * Triggers the file chooser for CSV files and updates the CSV path field.
     */
    loadCsv.addActionListener(e -> CarService.loadCsvFile(csvPathField));

    /**
     * Listener for the "Load XML" button.
     * Triggers the file chooser for XML files and updates the XML path field.
     */
    loadXml.addActionListener(e -> CarService.loadXmlFile(xmlPathField));

    /**
     * Listener for the "Process File" button.
     * Initiates the merging of loaded CSV and XML data and displays all cars.
     */
```

```
    processFileButton.addActionListener(e -> CarService.process());

    /**
     * Listener for the "Process with Filters" button (located inside filterPanel).
     * Gathers filter criteria from input fields and applies them to the car data,
     * then displays the filtered results.
     */
    processWithFiltersButton.addActionListener(
        e ->
            CarService.processWithFilters(
                brandFilter.getText(),
                typeFilter.getText(),
                modelFilter.getText(),
                priceFilter.getText(),
                currencyFilter.getText(),
                releaseDateFilter.getText(),
                additionalPriceFilter.getText())); // Ensure this matches CarService method
signature

    /**
     * Listener for the "Filter" button.
     * Toggles the visibility of the filter panel and re-packs the frame to adjust layout.
     */
    filterButton.addActionListener(
        e -> {
          filterPanel.setVisible(!filterPanel.isVisible());
          frame.pack();
        });

    /**
     * Listener for the "Sort By" button.
     * Toggles the visibility of the sort panel and re-packs the frame to adjust layout.
     */
    sortButton.addActionListener(
        e -> {
          sortPanel.setVisible(!sortPanel.isVisible());
          frame.pack();
        });
```

```
    /**
     * Listener for the "Apply Sort" button (located inside sortPanel).
     * Retrieves the selected sort field and order from the combo boxes and
     * triggers the sorting and display of car data.
     */
    applySortButton.addActionListener(
        e -> {
          String selectedField = (String) sortFieldComboBox.getSelectedItem();
          String selectedOrder = (String) sortOrderComboBox.getSelectedItem();
          CarService.sortAndDisplay(selectedField, selectedOrder);
        });
  }
}
```

Factory.java

A utility class for managing shared UI components, specifically the JTextArea used for displaying results.

```
package org.blank.factory;

import javax.swing.JTextArea;

/**
 * Provides a singleton instance of JTextArea for displaying application results.
 * This ensures that all parts of the application write to the same output area.
 */
public class Factory {

  /**
   * Private constructor to prevent direct instantiation.
   * This class is designed for static utility access.
   */
  private Factory() {}

  private static JTextArea resultArea;

  /**
   * Static initializer block to set up the JTextArea when the class is loaded.
```

```
 */
 static {
  resultArea = new JTextArea(10, 60);
  resultArea.setEditable(false); // Make the text area read-only
 }

 /**
  * Retrieves the singleton instance of the JTextArea used for displaying results.
  *
  * @return The JTextArea instance.
  */
 public static JTextArea getResultArea() {
  return resultArea;
 }
}
```

CarFileLoader.java

Handles the loading and parsing of car data from CSV and XML files, including merging data from both sources.

```
package org.blank.loader;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.logging.Logger;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.blank.repository.CarRepository;
import org.blank.repository.model.Car;
```

```
import org.blank.util.Utils;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource; // For EntityResolver
import org.xml.sax.SAXException; // For XML parsing exceptions

/**
 * Utility class responsible for loading and parsing car data from CSV and XML files.
 * It also handles merging data from both file types into a single list of Car objects.
 */
public class CarFileLoader {

  private static final Logger logger =
Logger.getLogger(CarFileLoader.class.getName());

  /**
   * Private constructor to prevent direct instantiation.
   */
  private CarFileLoader() {}

  /**
   * Processes both CSV and XML files, loads car data from them, merges the data,
   * and adds the final merged list to the CarRepository.
   *
   * @param csvFile The CSV file to load.
   * @param xmlFile The XML file to load.
   * @throws IOException If either file is null or an I/O error occurs during processing.
   */
  public static void process(File csvFile, File xmlFile) throws IOException {
    CarRepository.deleteAllCars(); // Clear existing data before loading new
    if (Objects.isNull(csvFile) || Objects.isNull(xmlFile)) {
      throw new IOException("CSV or XML file is null. Both files must be selected.");
    }
    List<Car> carList = mergeCars(loadCsv(csvFile), loadXml(xmlFile));
    CarRepository.addCars(carList);
    logger.info("Successfully processed and merged data from CSV and XML files.");
  }
```

```
 /**
  * Loads car data from a CSV file.
  *
  * @param file The CSV file to load.
  * @return A list of Car objects parsed from the CSV file.
  */
 public static List<Car> loadCsv(File file) {
  List<Car> cars = new ArrayList<>();
  try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
   // Read and discard the header line to satisfy SonarQube's unused return value
warning.
   String headerLine = reader.readLine();
   logger.info("CSV Header: " + headerLine); // Log the header for debugging/auditing

   String line;
   while ((line = reader.readLine()) != null) {
    String[] fields = line.split(","); // Splits the line by comma

    // Basic validation: ensure at least brand and releaseDate are present
    // Extend this logic to parse all fields based on your CSV structure
    if (fields.length >= 2) {
     String brand = fields[0].trim().replace("\"", "");
     String releaseDateStr = fields[1].trim().replace("\"", ""); // Assuming releaseDate
is second field

     // Parse other potential fields from CSV if they exist in your file format
     String type = fields.length > 2 ? fields[2].trim().replace("\"", "") : null;
     String model = fields.length > 3 ? fields[3].trim().replace("\"", "") : null;
     Double price = null;
     if (fields.length > 4 && !fields[4].trim().isEmpty()) {
      try {
       price = Double.parseDouble(fields[4].trim());
      } catch (NumberFormatException e) {
       logger.warning("Invalid price format in CSV: " + fields[4] + " for line: " + line);
      }
     }
     String currency = fields.length > 5 ? fields[5].trim().replace("\"", "") : null;
     Double additionalPrice = null;
```

```
        if (fields.length > 6 && !fields[6].trim().isEmpty()) {
            try {
                additionalPrice = Double.parseDouble(fields[6].trim());
            } catch (NumberFormatException e) {
                logger.warning("Invalid additional price format in CSV: " + fields[6] + " for
line: " + line);
            }
        }


        LocalDate releaseDate = Utils.parseDateStringSafely(releaseDateStr);

        // Build the Car object
        Car car = Car.builder()
            .brand(brand)
            .type(type)
            .model(model)
            .price(price)
            .currency(currency)
            .releaseDate(releaseDate)
            .additionalPrice(additionalPrice) // Added additionalPrice
            .build();

        cars.add(car);
       } else {
        logger.warning("Skipping malformed CSV line (less than 2 fields): " + line);
       }
     }
    } catch (IOException e) {
     logger.severe("Error reading CSV file: " + file.getAbsolutePath() + " - " +
e.getMessage());
      e.printStackTrace();
    }
    return cars;
  }

  /**
   * Loads car data from an XML file.
   * Implements secure XML parsing to prevent XXE vulnerabilities.
```

```
  *
  * @param file The XML file to load.
  * @return A list of Car objects parsed from the XML file.
  */
 public static List<Car> loadXml(File file) {
  List<Car> cars = new ArrayList<>();
  try {
   Document doc;
   try (FileInputStream fileInputStream = new FileInputStream(file)) {
    DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();

    // --- Secure XML Parsing Configuration to prevent XXE ---
    // Disallow DOCTYPE declarations completely
    dbFactory.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
true);
    // Disable external general entities
    dbFactory.setFeature("http://xml.org/sax/features/external-general-entities",
false);
    // Disable external parameter entities
    dbFactory.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
    // Do not load external DTDs

dbFactory.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd
", false);
    // Disable XInclude processing
    dbFactory.setXIncludeAware(false);

    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
    // Set a custom EntityResolver as a fallback defense to prevent external entity
loading
    dBuilder.setEntityResolver((publicId, systemId) -> {
     logger.warning("Attempted to resolve external entity: Public ID=" + publicId + ",
System ID=" + systemId + ". Blocking.");
     return new InputSource(new java.io.StringReader("")); // Return empty DTD
    });

    doc = dBuilder.parse(fileInputStream);
   }
```

```
    doc.getDocumentElement().normalize(); // Normalize the document structure

    // Get all <car> nodes from the XML
    NodeList carNodes = doc.getElementsByTagName("car");

    for (int i = 0; i < carNodes.getLength(); i++) {
      Node carNode = carNodes.item(i);

      if (carNode.getNodeType() == Node.ELEMENT_NODE) {
        Element carElement = (Element) carNode;

        // Extract car properties
        // Note: brand and releaseDate are expected from CSV for merge,
        // XML provides type, model, price, currency, additional prices
        String type = getTagContent(carElement, "type");
        String model = getTagContent(carElement, "model");

        Element priceElement = (Element)
carElement.getElementsByTagName("price").item(0);
        double price = Double.parseDouble(priceElement.getTextContent());
        String currency = priceElement.getAttribute("currency");

        Car car =
Car.builder().type(type).model(model).price(price).currency(currency).build();

        // Extract additional prices from the <prices> tag
        NodeList priceNodes =
carElement.getElementsByTagName("prices").item(0).getChildNodes();
        for (int j = 0; j < priceNodes.getLength(); j++) {
          Node priceNode = priceNodes.item(j);
          if (priceNode.getNodeType() == Node.ELEMENT_NODE) {
            Element priceDetailElement = (Element) priceNode;
            String priceCurrency = priceDetailElement.getAttribute("currency");
            double priceValue =
Double.parseDouble(priceDetailElement.getTextContent());
            car.addPrice(priceCurrency, priceValue); // Add price to the car's
additionalPrices map
          }
        }
```

```
        cars.add(car);
       }
      }
    } catch (ParserConfigurationException | SAXException | IOException |
NumberFormatException e) {
     logger.severe("Error loading or parsing XML file: " + file.getAbsolutePath() + " - " +
e.getMessage());
     e.printStackTrace();
   }
   return cars;
 }

 /**
  * Helper method to safely get text content from an XML element's child tag.
  * @param parentElement The parent XML element.
  * @param tagName The name of the child tag to get content from.
  * @return The text content of the tag, or null if the tag is not found.
  */
 private static String getTagContent(Element parentElement, String tagName) {
    NodeList nodeList = parentElement.getElementsByTagName(tagName);
    if (nodeList != null && nodeList.getLength() > 0) {
       return nodeList.item(0).getTextContent();
    }
    return null;
 }



 /**
  * Merges two lists of Car objects (typically one from CSV and one from XML).
  * It assumes cars are aligned by index (e.g., first CSV car merges with first XML car).
  * Fields from CSV are preferred if not null; otherwise, XML fields are used.
  * Additional prices are merged into a single map.
  *
  * @param csvCars A list of Car objects loaded from CSV.
  * @param xmlCars A list of Car objects loaded from XML.
  * @return A new list of merged Car objects.
  */
 public static List<Car> mergeCars(List<Car> csvCars, List<Car> xmlCars) {
   List<Car> mergedCars = new ArrayList<>();
```

```
    // Assumes csvCars and xmlCars are conceptually "paired" by index for merging.
    // In a real-world scenario, you might merge based on a unique ID (e.g., VIN).
    int maxSize = Math.min(csvCars.size(), xmlCars.size());
    for (int i = 0; i < maxSize; i++) {
      Car csvCar = csvCars.get(i);
      Car xmlCar = xmlCars.get(i);

      // Use the builder pattern to merge the fields
      Car mergedCar =
          Car.builder()
              // Prefer CSV brand, otherwise XML
              .brand(Objects.nonNull(csvCar.getBrand()) ? csvCar.getBrand() :
xmlCar.getBrand())
              // Prefer CSV type, otherwise XML
              .type(Objects.nonNull(csvCar.getType()) ? csvCar.getType() :
xmlCar.getType())
              // Prefer CSV model, otherwise XML
              .model(Objects.nonNull(csvCar.getModel()) ? csvCar.getModel() :
xmlCar.getModel())
              // Prefer CSV currency, otherwise XML
              .currency(Objects.nonNull(csvCar.getCurrency()) ? csvCar.getCurrency() :
xmlCar.getCurrency())
              // Prefer CSV price, otherwise XML (handling Double.NaN if used as sentinel)
              .price(Objects.nonNull(csvCar.getPrice()) ? csvCar.getPrice() :
xmlCar.getPrice())
              // Prefer CSV releaseDate, otherwise XML
              .releaseDate(
                  Objects.nonNull(csvCar.getReleaseDate())
                      ? csvCar.getReleaseDate()
                      : xmlCar.getReleaseDate())
              // Merge additional prices from both sources
              .additionalPrices(
                  mergeAdditionalPrices(csvCar.getAdditionalPrices(),
xmlCar.getAdditionalPrices()))
              .build();

      mergedCars.add(mergedCar);
    }
```

```
    // If one list is longer, you might choose to add the remaining cars, or ignore them.
    // Current implementation only merges up to the size of the smaller list.

    return mergedCars;
  }

  /**
   * Merges two maps of additional prices. If a currency exists in both,
   * the value from csvPrices is preferred.
   *
   * @param csvPrices A map of additional prices from the CSV source.
   * @param xmlPrices A map of additional prices from the XML source.
   * @return A new map containing merged additional prices.
   */
  private static Map<String, Double> mergeAdditionalPrices(
      Map<String, Double> csvPrices, Map<String, Double> xmlPrices) {
    // Start with CSV prices (or an empty map if CSV prices are null)
    Map<String, Double> mergedPrices = new HashMap<>(Objects.nonNull(csvPrices) ?
csvPrices : new HashMap<>());

    // Merge prices from XML into the map.
    // If a key already exists, keep the existing value (from CSV); otherwise, add the XML
value.
    if (Objects.nonNull(xmlPrices)) {
      for (Map.Entry<String, Double> entry : xmlPrices.entrySet()) {
        mergedPrices.merge(entry.getKey(), entry.getValue(), (v1, v2) -> v1 != null ? v1 :
v2);
      }
    }
    return mergedPrices;
  }
}
```

CarRepository.java

Acts as a simple in-memory data store for Car objects.

```
package org.blank.repository;
```

```
import java.util.ArrayList;
import java.util.Collections; // For unmodifiable list
import java.util.List;
import java.util.Objects;
import org.blank.repository.model.Car;

/**
 * A simple in-memory repository for Car objects.
 * This class provides basic CRUD-like operations (add, get all, delete all)
 * for the Car data within the application.
 */
public class CarRepository {

  // The static list to store all Car objects.
  private static List<Car> cars;

  /**
   * Private constructor to prevent direct instantiation.
   * This class is designed for static utility access.
   */
  private CarRepository() {}

  /**
   * Static initializer block to ensure the cars list is initialized when the class is loaded.
   */
  static {
    cars = new ArrayList<>();
  }

  /**
   * Adds a list of new Car objects to the repository.
   *
   * @param newCars The list of Car objects to add. Must not be null.
   * @throws NullPointerException If newCars is null.
   */
  public static void addCars(List<Car> newCars) {
    Objects.requireNonNull(newCars, "New cars list cannot be null.");
    cars.addAll(newCars);
  }
```

```
  /**
   * Retrieves an unmodifiable list of all Car objects currently stored in the repository.
   *
   * @return An unmodifiable list of Car objects.
   */
  public static List<Car> getAllCars() {
    return Collections.unmodifiableList(cars); // Return unmodifiable list to prevent
external modification
  }

  /**
   * Deletes all Car objects from the repository, effectively clearing the data.
   *
   * @return true if the operation was successful (always true).
   */
  public static boolean deleteAllCars() {
    cars = new ArrayList<>(); // Re-initialize the list to clear it
    return true;
  }

  /**
   * Replaces the current list of cars with a new list.
   * Useful after filtering/sorting when the CarService needs to update the repository's
view.
   * @param newCars The new list of cars to set.
   */
  public static void setAllCars(List<Car> newCars) {
      Objects.requireNonNull(newCars, "New cars list cannot be null.");
      // Create a new ArrayList to ensure it's a deep copy (references to Car objects are
copied,
      // but the list itself is new, preventing external modifications to the repository's
internal list).
      CarRepository.cars = new ArrayList<>(newCars);
  }
}
```

Car.java

A Lombok-enhanced data model class representing a car, with fields for various attributes and a map for additional prices.

```
package org.blank.repository.model;

import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;
import lombok.Builder;
import lombok.Data; // Provides @Getter, @Setter, @ToString, @EqualsAndHashCode, @NoArgsConstructor (with a quirk)
import lombok.NoArgsConstructor; // Explicitly add NoArgsConstructor for frameworks like Jackson/deserialization
import lombok.AllArgsConstructor; // For a constructor with all fields

/**
 * Data model class representing a Car.
 * Uses Lombok annotations for boilerplate code reduction.
 *
 * {@code @Data} provides getters, setters, toString, equals, and hashCode.
 * {@code @Builder} provides a fluent API for constructing Car objects.
 * {@code @NoArgsConstructor} and {@code @AllArgsConstructor} provide constructors.
 */
@Data // Combines @Getter, @Setter, @ToString, @EqualsAndHashCode, and a default @NoArgsConstructor
@Builder // Generates a builder pattern for object creation
@NoArgsConstructor // Creates a no-argument constructor (important for some frameworks)
@AllArgsConstructor // Creates a constructor with all fields
public class Car {

  private String brand;

  private String type;

  private String model;

  // Using 'double' for primitive; consider 'Double' wrapper if price can be null
```

```
  private double price;

  private String currency;

  private LocalDate releaseDate; // Using LocalDate for modern date handling

  // A map to store additional prices with different currencies
  private Map<String, Double> additionalPrices;

  /**
   * Adds an additional price for a specific currency to the car's additional prices map.
   * If the map is null, it will be initialized.
   *
   * @param currency The currency of the additional price (e.g., "EUR", "GBP").
   * @param price The value of the additional price.
   */
  public void addPrice(String currency, double price) {
    if (this.additionalPrices == null) {
       this.additionalPrices = new HashMap<>();
    }
    this.additionalPrices.put(currency, price);
  }
}
```

CarService.java

Contains the business logic for the application, including file loading, data processing, filtering, and sorting.

```
package org.blank.service;

import java.io.File;
import java.io.IOException;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;
```

```
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.JTextField;
import org.apache.commons.lang3.StringUtils;
import org.blank.loader.CarFileLoader;
import org.blank.repository.CarRepository;
import org.blank.repository.model.Car;
import org.blank.util.Utils;

/**
 * Provides the core business logic for the Car File Processor application.
 * Handles file operations, data processing, filtering, and sorting of car data.
 */
public class CarService {

  /**
   * Private constructor to prevent direct instantiation.
   * This class is designed for static utility access.
   */
  private CarService() {}

  // Store the last selected CSV and XML files for processing
  private static File csvFile;
  private static File xmlFile;

  /**
   * Prompts the user to select a CSV file and updates the corresponding JTextField.
   *
   * @param csvPathField The JTextField to display the selected CSV file path.
   */
  public static void loadCsvFile(JTextField csvPathField) {
    try {
      File file = chooseFile("CSV");
      if (file != null) {
        csvPathField.setText(file.getAbsolutePath());
        csvFile = file; // Store the selected file
      }
    } catch (Exception e) {
      JOptionPane.showMessageDialog(null, "Error loading CSV file: " +
```

```
e.getMessage());
    }
  }

  /**
   * Prompts the user to select an XML file and updates the corresponding JTextField.
   *
   * @param xmlPathField The JTextField to display the selected XML file path.
   */
  public static void loadXmlFile(JTextField xmlPathField) {
    try {
      File file = chooseFile("XML");
      if (file != null) {
        xmlPathField.setText(file.getAbsolutePath());
        xmlFile = file; // Store the selected file
      }
    } catch (Exception e) {
      JOptionPane.showMessageDialog(null, "Error loading XML file: " +
e.getMessage());
    }
  }

  /**
   * Triggers the processing (loading and merging) of the selected CSV and XML files.
   * Displays the combined car data in the result area.
   */
  public static void process() {
    try {
      CarFileLoader.process(csvFile, xmlFile); // Load and merge data into CarRepository
      Utils.displayCars(CarRepository.getAllCars()); // Display all cars from the repository
      Utils.displayText("Files processed successfully. Total cars: " +
CarRepository.getAllCars().size());
    } catch (IOException e) {
      Utils.displayText("Error processing files: " + e.getMessage());
      e.printStackTrace();
    }
  }

  /**
```

```
   * Sorts the currently loaded car data based on the specified field and order,
   * then displays the sorted results.
   *
   * @param sortField The field by which to sort (e.g., "Brand", "Price", "Release Date").
   * @param sortOrder The sort order ("Ascending" or "Descending").
   */
  public static void sortAndDisplay(String sortField, String sortOrder) {
    List<Car> currentCars = new ArrayList<>(CarRepository.getAllCars()); // Create a
mutable copy to sort

    if (currentCars.isEmpty()) {
      Utils.displayText("No cars to sort. Please load data first.");
      return;
    }

    if ("None".equals(sortField)) {
      Utils.displayText("No sorting applied (selected 'None'). Displaying current data.");
      Utils.displayCars(currentCars); // Display current data without sorting
      return;
    }

    Comparator<Car> comparator = Utils.CAR_COMPARATORS.get(sortField);

    if (comparator == null) {
      Utils.displayText("Error: No comparator found for sort field: " + sortField);
      Utils.displayCars(currentCars); // Display current data, no sorting
      return;
    }

    if ("Descending".equals(sortOrder)) {
      comparator = comparator.reversed();
    }

    currentCars.sort(comparator); // Sort the mutable copy in place
    Utils.displayCars(currentCars); // Display the sorted list
    Utils.displayText("Cars sorted by " + sortField + " in " + sortOrder + " order. Total
cars: " + currentCars.size());
  }
```

```
  /**
   * Filters the currently loaded car data based on provided criteria and displays the filtered results.
   * Filters are applied cumulatively. Empty/null filter strings mean no filtering on that field.
   *
   * @param brand The brand filter string.
   * @param type The type filter string.
   * @param model The model filter string.
   * @param price The price filter string (will be parsed to Double).
   * @param currency The currency filter string.
   * @param releaseDateString The release date filter string (will be parsed to LocalDate).
   * @param additionalPriceString The additional price filter string (will be parsed to Double).
   */
  public static void processWithFilters(
      String brand,
      String type,
      String model,
      String price,
      String currency,
      String releaseDateString,
      String additionalPriceString) { // Added additionalPriceString parameter

    // Parse numeric inputs, handling NumberFormatException
    Double priceFilter = null;
    if (StringUtils.isNotBlank(price)) {
      try {
        priceFilter = Double.parseDouble(price.trim());
      } catch (NumberFormatException e) {
        Utils.displayText("Invalid price filter value: '" + price + "'. Please enter a valid number.");
        return; // Stop filtering if price input is invalid
      }
    }

    Double additionalPriceFilter = null;
    if (StringUtils.isNotBlank(additionalPriceString)) {
```

```
        try {
            additionalPriceFilter = Double.parseDouble(additionalPriceString.trim());
        } catch (NumberFormatException e) {
            Utils.displayText("Invalid additional price filter value: '" + additionalPriceString +
"'. Please enter a valid number.");
            return; // Stop filtering if additional price input is invalid
        }
    }

    // Parse date input
    LocalDate releaseDateFilter = Utils.parseDateStringSafely(releaseDateString);
    if (StringUtils.isNotBlank(releaseDateString) && Objects.isNull(releaseDateFilter)) {
        Utils.displayText("Invalid release date format: '" + releaseDateString + "'. Please
use yyyy-MM-dd, MM/dd/yyyy, or yyyy,dd,MM.");
        return; // Stop filtering if date format is invalid
    }


    List<Car> carsToFilter = CarRepository.getAllCars(); // Get current cars from
repository

    List<Car> filteredCars =
        carsToFilter.stream()
            // Brand filter: Apply if brand filter string is not blank. Check car's brand for null.
            .filter(e -> StringUtils.isBlank(brand) ||
                    (Objects.nonNull(e.getBrand()) &&
e.getBrand().equalsIgnoreCase(brand.trim())))
            // Type filter: Apply if type filter string is not blank. Check car's type for null.
            .filter(e -> StringUtils.isBlank(type) ||
                    (Objects.nonNull(e.getType()) &&
e.getType().equalsIgnoreCase(type.trim())))
            // Model filter: Apply if model filter string is not blank. Check car's model for
null.
            .filter(e -> StringUtils.isBlank(model) ||
                    (Objects.nonNull(e.getModel()) &&
e.getModel().equalsIgnoreCase(model.trim())))
            // Price filter: Apply if priceFilter (Double) is not null. Check car's price for null
and then equality.
            .filter(e -> Objects.isNull(priceFilter) ||
```

```
                (Objects.nonNull(e.getPrice()) && Objects.equals(e.getPrice(),
priceFilter)))
        // Currency filter: Apply if currency filter string is not blank. Check car's
currency for null.
        .filter(e -> StringUtils.isBlank(currency) ||
                (Objects.nonNull(e.getCurrency()) &&
e.getCurrency().equalsIgnoreCase(currency.trim())))
        // Release Date filter: Apply if releaseDateFilter (LocalDate) is not null. Check
car's releaseDate for null and then equality.
        .filter(e -> Objects.isNull(releaseDateFilter) ||
                (Objects.nonNull(e.getReleaseDate()) &&
e.getReleaseDate().isEqual(releaseDateFilter)))
        // Additional Price filter: Apply if additionalPriceFilter (Double) is not null. Check
car's additionalPrice for null and then equality.
        .filter(e -> Objects.isNull(additionalPriceFilter) ||
                (Objects.nonNull(e.getAdditionalPrice()) &&
Objects.equals(e.getAdditionalPrice(), additionalPriceFilter)))
        .collect(Collectors.toCollection(ArrayList::new)); // Collect results into a new
ArrayList

   CarRepository.setAllCars(filteredCars); // Update the repository with filtered results
   Utils.displayCars(filteredCars); // Display the filtered list
   Utils.displayText("Applied filters. Total cars matching criteria: " + filteredCars.size());
  }

  /**
   * Opens a file chooser dialog to allow the user to select a file.
   *
   * @param fileType The type of file to select (e.g., "CSV", "XML") for dialog title and
filter.
   * @return The selected File object, or null if no file was selected.
   */
  private static File chooseFile(String fileType) {
   JFileChooser fileChooser = new JFileChooser();
   fileChooser.setDialogTitle("Select " + fileType + " file");

   if ("CSV".equals(fileType)) {
    fileChooser.setFileFilter(
        new javax.swing.filechooser.FileNameExtensionFilter("CSV Files", "csv"));
```

```
    } else if ("XML".equals(fileType)) {
      fileChooser.setFileFilter(
          new javax.swing.filechooser.FileNameExtensionFilter("XML Files", "xml"));
    }

    int result = fileChooser.showOpenDialog(null);
    if (result == JFileChooser.APPROVE_OPTION) {
      return fileChooser.getSelectedFile();
    }
    return null;
  }
}
```

Utils.java

A utility class providing helper methods for common tasks such as date parsing, displaying messages, and defining Car comparators.

```
package org.blank.util;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeFormatterBuilder;
import java.time.format.DateTimeParseException;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import org.apache.commons.lang3.StringUtils;
import org.blank.factory.Factory;
import org.blank.repository.model.Car;

/**
 * Utility class providing helper methods for common application tasks.
 * This includes date parsing, displaying messages to the UI, and
 * providing pre-defined comparators for Car objects.
 */
public class Utils {
```

```
 /**
  * Private constructor to prevent direct instantiation.
  * This class is designed for static utility access.
  */
 private Utils() {}

 /**
  * A static map of pre-defined Comparators for Car objects,
  * allowing dynamic sorting based on string field names.
  * Comparators are configured to handle null values by placing them last,
  * and string comparisons are case-insensitive.
  */
 public static final Map<String, Comparator<Car>> CAR_COMPARATORS =
    Map.of(
       "Brand",
          Comparator.comparing(
             Car::getBrand, Comparator.nullsLast(String::compareToIgnoreCase)),
       "Type",
          Comparator.comparing(Car::getType,
Comparator.nullsLast(String::compareToIgnoreCase)),
       "Model",
          Comparator.comparing(
             Car::getModel, Comparator.nullsLast(String::compareToIgnoreCase)),
       "Price", Comparator.comparing(Car::getPrice,
Comparator.nullsLast(Double::compareTo)),
       "Main Currency",
          Comparator.comparing(
             Car::getCurrency, Comparator.nullsLast(String::compareToIgnoreCase)),
       "Release Date",
          Comparator.comparing(
             Car::getReleaseDate, Comparator.nullsLast(LocalDate::compareTo)),
       "Additional Price",
          Comparator.comparing(
             Car::getAdditionalPrice, Comparator.nullsLast(Double::compareTo))
       );

 /**
  * Safely parses a date string into a LocalDate object using multiple predefined
formats.
```

```
   * If the string is blank or does not match any format, it returns null and logs an error.
   *
   * @param dateString The date string to parse.
   * @return A LocalDate object if parsing is successful, otherwise null.
   */
  public static LocalDate parseDateStringSafely(String dateString) {
    // Define multiple date formats to try
    DateTimeFormatter dateFormatter =
        new DateTimeFormatterBuilder()
            .appendOptional(DateTimeFormatter.ofPattern("MM/dd/yyyy"))
            .appendOptional(DateTimeFormatter.ofPattern("yyyy-MM-dd"))
            .appendOptional(DateTimeFormatter.ofPattern("yyyy,dd,MM")) // Assuming
'MM' for month, not 'mm' for minute
            .toFormatter();

    if (StringUtils.isBlank(dateString)) {
      return null;
    }

    try {
      return LocalDate.parse(dateString.trim(), dateFormatter);
    } catch (DateTimeParseException e) {
      displayText("Could not parse date '" + dateString + "'. Invalid format. " +
e.getMessage());
      return null;
    }
  }

  /**
   * Displays a list of Car objects in the application's result area.
   * Each car is represented by its toString() method on a new line.
   *
   * @param cars The list of Car objects to display.
   */
  public static void displayCars(List<Car> cars) {
    StringBuilder sb = new StringBuilder();
    if (Objects.isNull(cars) || cars.isEmpty()) {
        sb.append("No cars to display.");
    } else {
```

```
        for (Car car : cars) {
            sb.append(car).append("\n");
        }
    }
    Factory.getResultArea().setText(sb.toString());
  }

  /**
   * Displays a single text message in the application's result area.
   *
   * @param message The string message to display.
   */
  public static void displayText(String message) {
    Factory.getResultArea().setText(message);
  }
}
```

Dependencies

This project uses Maven for dependency management. The pom.xml includes the following key dependencies:

- **Lombok**: For boilerplate code generation (getters, setters, constructors, builders).
- **Apache Commons Lang3**: For utility methods like StringUtils.isBlank and StringUtils.isNumeric.
- **Jackson**: For potential JSON processing (though not explicitly used for file I/O in the provided code, often used with LocalDate with jackson-datatype-jsr310).
- Standard Java Swing and java.time classes are part of the JDK.

Security Considerations

The CarFileLoader.java includes configurations for DocumentBuilderFactory to mitigate **XXE (XML External Entity) vulnerabilities** during XML parsing. This is achieved by:

- Disallowing DOCTYPE declarations.
- Disabling external general and parameter entities.
- Preventing external DTD loading.
- Setting an EntityResolver as a fallback to block external entity resolution.

This makes the XML parsing more secure against malicious XML inputs.

Future Enhancements

- **Error Handling**: More sophisticated error handling and user feedback for file operations and invalid inputs.
- **Persistence**: Implement saving data back to files (XML/CSV) or a database.
- **UI Improvements**: Enhance the user interface for better usability, such as clearer status messages, progress indicators, or more intuitive filter/sort options.
- **Advanced Filtering/Sorting**: Add more complex filtering (e.g., price ranges, date ranges) and multi-level sorting.
- **Unique Car ID**: Implement a unique identifier for cars (e.g., VIN) to enable more robust merging logic instead of relying on array index.
- **Data Validation**: Implement stricter input validation beyond basic null/blank checks.
- **Unit and Integration Tests**: Add comprehensive tests to ensure all components function correctly and safely.