Follow-up Assignment:
A Reflection-Based Class Analyzer with a Swing Interface

October 28, 2025

1. **Reflection CLI Layer**

   - Create a class named `ReflectionCLI`.
   - Inside it, define a `main` method accepting an array of type `String`.
   - Within the `main` method, implement the following steps:
     - Verify whether the number of provided arguments is not equal to 1; if so, display an error message about an incorrect number of parameters and exit the method using `return`.
     - Create a variable named `classFile` of type `File` and initialize it with the first argument from the `args` array.
     - Obtain the class name by calling `getName()` on the `classFile` object and removing the `.class` extension.
     - Retrieve the directory path of the file using `getParentFile()` and store it in a variable named `parentDir` of type `File`.
     - Create a variable `classLoader` of type `URLClassLoader` and initialize it with a new `URLClassLoader` object that takes an array of URLs with a single element – the `parentDir` converted to URI and then to URL.
     - Finally, create a variable `loadedClass` of type `Class<?>` and assign it the result of calling `loadClass` on `classLoader` with `className` as its argument.
   - Test that the application works correctly and present your results to the instructor.
   - Add a private static method named `getParameterString`, which takes a one-dimensional array of `Class<?>` elements named `paramTypes`. Inside this method:
     - If the array length is 0, return an empty string.
     - Create and initialize a `StringBuilder` variable named `params`.
     - Iterate over all elements of `paramTypes`, retrieving the simple name of each type using `getSimpleName()` and appending it to `params`, separated by commas.
     - Return the final string stored in `params`.
   - Create a static method `createFieldRow` that accepts a `Field` argument named `field` and returns an `Object[]` array. Inside, return a new array containing:
     - The string `"Field"`;

  - The field modifiers as a string (use `Modifier.toString(field.getModifiers())`);
  - The field name;
  - The string `"Type:"` concatenated with the simple name of the field type.
- Create a static method `createConstructorRow` that takes a `Constructor<?>` named `constructor` and returns an `Object[]` array containing:
  - The string `"Constructor"`;
  - Constructor modifiers (via `Modifier.toString()`);
  - The simple name of the declaring class (`constructor.getDeclaringClass().getSimpleName()`);
  - The result of calling `getParameterString()` on the constructor's parameter types, enclosed in parentheses.
- Create a static method `createMethodRow` that takes a `Method` named `method` and returns an `Object[]` array containing:
  - The string `"Method"`;
  - Method modifiers as a string;
  - The method name;
  - The return type simple name concatenated with the result of `getParameterString(method.getParameterTypes())` inside parentheses.
- Create a static method `createInterfaceRow` that takes a `Class<?>` named `iface` and returns an `Object[]` array containing:
  - The string `"Interface"`;
  - An empty string;
  - The interface simple name;
  - The full interface name.
- Create a static, generic method `showReflectionElements` that takes three parameters:
  - An array of elements of type `T[]` named `elements`;
  - A `String` named `sectionTitle`;
  - A `Function<T, Object[]>` named `rowCreator`.

  The method should:
  - Check whether `elements` is not null and has a length greater than zero;
  - Print the section title;
  - Iterate through all elements and print the result of `rowCreator.apply(element)`;
  - Print an empty line separator.
- Create a static void method `analyzeClass`, accepting a `Class<?>` argument named `clazz`. Inside it, call `showReflectionElements` four times with:
  - `clazz.getDeclaredFields()`, the title `"FIELDS"` and a reference to a method `createFieldRow`;

    - `clazz.getDeclaredConstructors()`, the title `"CONSTRUCTORS"` and a reference to a method `createConstructorRow`;
    - `clazz.getDeclaredMethods()`, the title `"METHODS"` and a reference to a method `createMethodRow`;
    - `clazz.getInterfaces()`, the title `"INTERFACES"` and a reference to a method `createInterfaceRow`.
- At the end of `main`, call `analyzeClass(loadedClass)`.
- Test that the application works correctly and present your results to the instructor.

2. **Graphical User Interface Layer (UI)**

- Create a class named `ObjectInspector` extending `JFrame`.
- Implement a static `main` method that invokes `SwingUtilities.invokeLater`, inside which an `ObjectInspector` object is created and made visible.
- Declare instance (non-static) fields:
    - `JTable table;`
    - `DefaultTableModel tableModel;`
    - `JLabel statusLabel;`
    - `JTabbedPane tabbedPane;`
- Implement a no-argument constructor that:
    - Sets the window title to `"Object Inspector"`;
    - Sets the default close operation to `EXIT_ON_CLOSE`;
    - Uses `BorderLayout` as the layout manager;
    - Creates a top panel of type `JPanel` with `FlowLayout`;
    - Creates two buttons: `loadButton` ("Select file .class") and `clearButton` ("Clear");
    - Adds both buttons to `topPanel`;
    - Initializes `statusLabel` with text "Select file .class for analysis";
    - Initializes `tabbedPane` as a new `JTabbedPane`;
    - Defines a `String[]` array named `columns` with four labels: "Type", "Modifiers", "Name", "Details";
    - Initializes `tableModel` with a new anonymous subclass of `DefaultTableModel` taking `columns` and 0 rows, and overrides `isCellEditable` to always return `false`;
    - Creates `table` as a new `JTable(tableModel)` and sets its auto-resize mode to `AUTO_RESIZE_ALL_COLUMNS`;
    - Adjusts preferred column widths to 150, 150, 200, and 350;
    - Wraps `table` inside a `JScrollPane` named `scrollPane`;
    - Adds a tab named "All elements" containing `scrollPane` to `tabbedPane`;
    - Adds `topPanel`, `tabbedPane`, and `statusLabel` to the main window (top, center, and bottom positions respectively);

  - Sets the window size to 900 × 600.
- Test that the application works correctly and present your results to the instructor.
- Implement a void, no-argument method `loadClassFile`:
  - Create and initialize a `JFileChooser`;
  - Define a `FileNameExtensionFilter` named `filter` with parameters "Java Class files (.class)" and "class";
  - Set this filter on the file chooser and disable the "accept all" filter;
  - Call `showOpenDialog(this)` and store the result in an `int` variable named `result`;
  - If `result == APPROVE_OPTION`, obtain the selected file and pass it to `analyzeClassFile`.
- Extend `main` by registering an `ActionListener` for `loadButton`, which calls `loadClassFile()`.
- Implement a void, no-argument method `clearTable` that sets `tableModel.setRowCount(0)`.
- Register an `ActionListener` for `clearButton` to call `clearTable()`.
- Implement a method `analyzeClassFile(File selectedFile)` whose logic follows the `main` method of `ReflectionCLI`, performing class loading and reflection analysis.
- Implement `analyzeClass(Class<?> clazz)` similarly to `ReflectionCLI.analyzeClass`, replacing calls to `showReflectionElements` with `addReflectionElements`.
- Implement `addReflectionElements`, taking the same parameters as `showReflectionElements`, but instead of printing results, add the resulting object arrays to `tableModel` using `addRow()`.
- Test that the application works correctly and present your results to the instructor.

3. **Final Report — Observations and Conclusions**

- Prepare a short written report describing your observations and conclusions from this assignment. Include:
  - A short description of how `ReflectionCLI` discovers fields, constructors, methods and interfaces, and how `ObjectInspector` presents them.
  - Notes on class loading from arbitrary `.class` files via `URLClassLoader` (e.g., classpath root, simple name vs. fully-qualified name).
  - Observations about the UI layer: table model design, immutability of cells, column sizing, and how sections are separated.
  - Potential limitations (e.g., handling of nested/anonymous classes, lack of package scanning, missing annotations display) and ideas for extensions.
  - A brief conclusion summarizing what worked well, what was tricky, and how the tool could be improved (e.g., adding modifiers filtering, annotations tab, search by member name).
- Submit the report together with your source code.
- **Do not send `.class` or `.zip` files**