Core Java

Dr.K.Sathiyamurthy

JAVA I/O

Core Concepts

- **Streams:** Java treats input and output as a continuous flow of data called streams. There are two primary types:
    - **Byte Streams:** Handle raw data (like images, audio, etc.) as bytes.
    - **Character Streams:** Work with text data as a sequence of characters.
- **Standard Streams:**
    - **System.in** (InputStream): The standard input stream, typically connected to the keyboard for user input.
    - **System.out** (PrintStream): The standard output stream, commonly connected to your console or terminal window for displaying results.
    - **System.err** (PrintStream): The standard error stream, reserved for printing error messages.

JAVA I/O

Common Input/Output Classes

1.Scanner (for user input)

1. Versatile way to read various data types (numbers, strings, etc.) from the user.

2.BufferedReader (for efficient input)

1. Improves performance by reading text in chunks, often used with System.in or file reading.

3.InputStream/OutputStream (for byte data)

1. Base classes for byte-oriented I/O, like reading from or writing to files.

4.Reader/Writer (for character data)

1. Base classes for character-based I/O, like reading or writing text files.

JAVA I/O

Input streams and output streams have a wide range of applications in Java programming. Here are some of the key areas where they are used:

1. File Input/Output

•**Reading data from files:** Used to read the contents of various file types:

- Text files (InputStreamReader, FileReader)
- Images (ImageInputStream, specialized libraries)
- Configuration files (FileInputStream)
- Any binary data (FileInputStream)

•**Writing data to files:** Used to save various data into files:

- Saving text output to files (OutputStreamWriter, FileWriter)
- Logging information
- Saving objects (ObjectOutputStream)

2. Network Communication

•**Retrieving data from web servers:** Used to fetch webpages, download files, or interact with web services (URLConnection, specialized libraries for HTTP).

•**Sending data to web servers:** Used to upload data or send requests to web services (URLConnection).

•**Socket communication:** Essential for low-level network communication between applications (Socket, ServerSocket).

JAVA I/O

3. Data Processing

•**Filtering and transforming data:** Input and output streams can be chained together with various filter streams to modify data on the fly:

- Compression/Decompression (ZipInputStream, GZIPOutputStream)
- Encryption/Decryption (CipherInputStream, CipherOutputStream)
- Data conversion between formats

•**Piping:** Output from one process can be connected as the input to another process (PipedInputStream, PipedOutputStream).

4. User Input/Output

•**Standard input (System.in):** Used to read user input from the keyboard (often combined with Scanner or BufferedReader).

•**Standard output (System.out) and error (System.err):** Used to display text in the console or terminal and print error messages.

5. Other Uses

•**In-memory data structures:** Input/output streams work with memory-based data structures like ByteArrays (ByteArrayInputStream, ByteArrayOutputStream).

•**Resource Loading:** Used to load resources like images or configuration files embedded within your Java application (Class.getResourceAsStream()).

Lambda expressions

- ❖ Lambda expressions are a feature introduced in Java 8 -provides a clear and concise way to represent one method interface using an expression.
- ❖ Used primarily to define inline implementation of a functional interface, an interface with a single method only.
- ❖ Lambda expressions bring a new, simplified syntax to Java that enhances readability and reduces the amount of boilerplate code required to implement functional programming concepts.

Syntax

The basic syntax of a lambda expression is:

```
(argument-list) -> { body }
```

- **argument-list**: This can be a single argument or multiple arguments enclosed in parentheses. For a single argument, parentheses can be omitted.
- **->**: The arrow token signifies that the expression on the left side is being mapped to the block on the right side.
- **body**: This contains expressions and statements to define the lambda's functionality. If the body consists of a single expression or statement, curly braces can be omitted.

```
// Traditional anonymous inner class
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});

// Lambda expression
button.addActionListener(e -> System.out.println("Button clicked!"));
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .filter(n -> n % 2 == 0) // Filter even numbers
                 .mapToInt(n -> n)        // Convert to Integer for sum
                 .sum();                  // Calculate the sum
```

```
public static <T> void sort(List<T> list, Comparator<T> comparator) {

 Collections.sort(list, comparator);

}

<String> names = Arrays.asList("Bob", "Alice", "Charlie");

sort(names, (s1, s2) -> s1.compareToIgnoreCase(s2)); // Case-insensitive sort
```

TDD – TEST DRIVEN DEVELOPMENT

The core principles of **Test-Driven Development (TDD)** revolve around a set of practices and mindsets that prioritize testing, code quality, and iterative improvement. These principles guide developers in creating robust, maintainable, and error-free code. Here are the fundamental principles of TDD:

1. Write Tests First

Before writing any implementation code, you start by writing tests for the smallest unit of functionality. This is a departure from traditional coding practices where tests are written after the code. The test-first approach ensures that development is focused on requirements and that each piece of code has a corresponding test case right from the start.

Red-Green-Refactor Cycle

TDD is structured around a simple, repeatable cycle:

- **Red**: Write a test for a new function or improvement. Run the test, which should fail because the functionality isn't implemented yet. This step ensures that the test is valid and that it will genuinely verify the functionality once it's implemented.
- **Green**: Write the minimum amount of code needed to make the test pass. This encourages simplicity and the implementation of only what is necessary.
- **Refactor**: Clean up the new code without changing its behavior. This includes removing duplication, improving names, and simplifying complex logic. The tests should pass after refactoring, confirming that no functionality was altered.

Tests as Specifications

In TDD, tests serve as a form of live documentation. They specify what the code is supposed to do, thus acting as detailed requirements. This approach ensures that the codebase is always accompanied by up-to-date documentation on how each part is intended to work.

4. Small, Incremental Steps

TDD encourages development in small, manageable increments. This minimizes the risk of introducing errors and makes it easier to isolate and fix bugs. It also keeps the system in a potentially shippable state after each increment.

5. Continuous Feedback

The TDD cycle provides continuous feedback about the code's functionality and design. Developers receive immediate information on whether the new code works as expected and whether it has introduced regressions in existing functionality.

6. Refactoring with Confidence

Since every change is made in small steps and immediately tested, developers can refactor code with confidence. They can improve the design and structure of the code without fearing that they'll break existing functionality.

7. Emphasis on Clean Code

TDD inherently promotes writing clean, maintainable code. By refactoring with each cycle, the codebase evolves into a well-organized system with reduced duplication and increased clarity.

8. Developer Ownership

TDD gives developers a greater sense of ownership and responsibility for the quality of their code. By writing tests and seeing them pass, developers can take pride in their work and be assured of its quality.

These core principles of TDD not only guide developers in creating high-quality software but also foster a culture of continuous improvement and attention to detail.

JUNIT5

What is JUnit 5?

JUnit 5 is the fifth major release of the JUnit framework, a widely used unit testing framework in the Java programming world. It represents a significant evolution from its predecessors, introducing several new features and improvements to make testing more powerful and easier to use. JUnit 5 consists of three main components:

1. **JUnit Platform**: Serves as the foundation for launching testing frameworks on the JVM. It defines the TestEngine API for developing testing frameworks that run on the platform.
2. **JUnit Jupiter**: Provides the new programming model and extension model for writing tests and extensions in JUnit 5. It includes new annotations, assertions, and assumptions that enhance the ease and flexibility of writing tests.
3. **JUnit Vintage**: Provides a TestEngine for running tests written in the JUnit 3 and JUnit 4 formats on the JUnit 5 platform, ensuring backward compatibility.

Why is JUnit 5 Linked with TDD?

JUnit 5 is closely linked with Test-Driven Development (TDD) for several reasons:

1. **Facilitates the Red-Green-Refactor Cycle**: JUnit 5's features and design align perfectly with the TDD methodology. It allows developers to write tests first (Red), quickly implement the minimal code to pass the tests (Green), and then refactor as needed while ensuring that the tests still pass (Refactor).
2. **Enhanced Feedback Loop**: JUnit 5 provides immediate feedback on the tests' pass/fail status, which is crucial for the TDD process. This feedback loop helps developers to quickly adjust their code or tests, ensuring that the development is moving in the right direction.

3. Promotes Testability and Clean Code: Writing tests in JUnit 5 encourages developers to write testable code, which often results in cleaner, more modular designs. This is a natural outcome of following TDD principles, where the need to write tests upfront influences the architecture and design of the code.

4. Advanced Features for Complex Scenarios: JUnit 5 introduces advanced features like parameterized tests, nested tests, dynamic tests, and more. These features are particularly useful in TDD for covering a wide range of input conditions and complex testing scenarios, allowing for more thorough testing and higher code quality.

5. Integration with Development Environments: JUnit 5 integrates seamlessly with popular IDEs and build tools, making it an integral part of the development process. This integration supports TDD by allowing developers to easily write, run, and debug tests as part of their normal workflow.

In summary, JUnit 5 is an essential tool for implementing TDD in Java projects. Its design and features support the core principles of TDD, helping developers to write high-quality, well-tested software.

Maven

Maven is a powerful project management and comprehension tool used primarily for Java projects. It simplifies the build process like compiling code, packaging binaries, and managing documentation and dependencies. Maven uses an XML file (pom.xml) to describe the software project being built, its dependencies on other external modules and components, the build order, directories, and required plugins.

Key Features of Maven:

1. **Project Object Model (POM)**: At its core, Maven uses a declarative approach to project configuration, centralized around the `pom.xml` file. This file includes project details, configuration settings, and dependencies.
2. **Dependency Management**: Maven automates the process of downloading libraries (dependencies) required for a project from a central repository and including them in the build path. It also manages transitive dependencies (dependencies of dependencies), ensuring that all needed jars are available without having to manually gather and include them.
3. **Convention Over Configuration**: Maven uses a standard directory layout and a default build lifecycle to minimize configuration details, making projects easy to understand and build with minimal additional information.

4. Build Lifecycle: Maven defines a clear lifecycle for building software. It includes phases like compile, test, package, install, and deploy, which can be executed in order. This standard lifecycle facilitates automation and consistency across projects.

5. Plugins and Goals: Maven functionality can be extended with plugins. Each plugin can have one or more goals (tasks) it can perform. For example, compiling code, creating a JAR file, or creating Javadoc documentation can be plugin goals.

6. Repositories: Maven interacts with two types of repositories - local and remote. The local repository is a directory on the developer's machine where all the project dependencies are stored. Remote repositories are accessed over the internet to download dependencies that are not present in the local repository.

7. Multi-module Projects: Maven supports multi-module projects, allowing large projects to be broken down into smaller, manageable modules that can be built, tested, and deployed independently or together as a whole.

8. Project Templates (Archetypes): Maven provides project templates (archetypes) for generating new projects. This feature helps in setting up a new project quickly, following best practices and suitable structure as per the project needs.