

## JAVA / LESSON 29 OF 40

# Save Your World

Use UTF-8 to save data and restart the program

This chapter uses JDK 21 and does not require the preview function. | 100-120 minutes

## Create scoreboard archive WorldSave

When the program exits, the scores in the memory will also disappear. Give it a Chinese-readable text archive, clearly cover and append, check the damaged data line by line, and use round-trip testing to prove that the storage is reliable.

Prerequisites: Map, string splitting and exception handling. Today, the diamonds in the memory will no longer fall into the lava when the program is closed.

Before you start: It is recommended to complete Lessons 26 and 28 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=29&lang=en>

Long code lines may wrap at the > margin marker. The marker is not part of the code.

## The relative path starts from the program working directory

| Element       | Explanation                                 |
|---------------|---------------------------------------------|
| Path          | Describe location, do not create file       |
| Relative path | Determined by the startup working directory |
| Absolute path | Help find the real archive                  |

## In this lesson

01. Before starting: predict first, then let the program answer
02. Path points to the archive location
03. UTF 8 save Chinese
04. Cover append and directory
05. Each record occupies one line
06. Parse line by line and reject bad data
07. Missing files and IOException
08. Round trip test verification archive
09. Challenge Blueprint Scoreboard Archive
10. World Archive Program
11. Expansion tasks and clearance review
12. Own acceptance record

01 / 12

## Before starting: predict first, then let the program answer

This chapter uses JDK 21 and does not require the preview function. The story is borrowed from the block world, and the example runs on the ordinary Java console; if the values and rules are course models, they will be clearly stated.

Each common example is a complete independent file, saved according to the file name marked on the box. Place each example in a different folder to avoid duplicate names of auxiliary classes. When the complete project is divided into sections, they are connected sequentially in the same file.

**Terminal command: replace the Chinese placeholder with the actual file name and class name**

```
javac -encoding UTF-8 ExampleFile.java
java ExampleClass
```

- Prediction: Write the expected output first, or draw out the changes in references and data.
- Verification: Check line by line after running, and then only change one condition.
- Explanation: Explain whether the results come from Java rules, course contracts, or input data.

"Intentional error" is used to recognize bugs and cannot be mixed with ordinary examples. Use a special temporary directory for file experiments; do not use real archives to test coverage and deletion.

02 / 12

## Path points to the archive location

Path only describes the location and does not mean that the file already exists. The actual location of a relative path depends on where the program is started from.

**The relative path starts from the program working directory**

| Element       | Explanation                                 |
|---------------|---------------------------------------------|
| Path          | Describe location, do not create file       |
| Relative path | Determined by the startup working directory |
| Absolute path | Help find the real archive                  |

When saving is completed, the absolute path is displayed so that the player can find the file. Path issues are often just people looking for things in different boxes.

### L29Example01.java · Complete independent example

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
public class L29Example01 {
    public static void main(String[] args) {
        Path path = Path.of("saves", "lesson29-scores.txt");
        System.out.println(path.getFileName());
        System.out.println(path.toAbsolutePath().isAbsolute());
    }
}
```

### Output

```
lesson29-scores.txt
true
```

Print `path.toAbsolutePath()` yourself and copy the real address into the record. The check here only lists file names and true and false values that have nothing to do with the computer; `Path.of` does not create directories or files.

03 / 12

## UTF 8 save Chinese

Encoding is the rules between characters and bytes. Use the same encoding for saving and loading, so that Chinese names will not turn into question mark boxes.

### Writing and reading use the same character set

| Element             | Explanation                                          |
|---------------------|------------------------------------------------------|
| Character           | For example, pony=12                                 |
| UTF-8               | Writing and reading are explicitly the same encoding |
| Result verification | Chinese content remains consistent                   |

Do not rely on default encodings that may differ from computer to computer. Course archives are uniformly written in UTF-8.

## L29Example02.java · Complete independent example

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
public class L29Example02 {
    public static void main(String[] args) throws IOException {
        Path folder = Files.createTempDirectory("java-utf8-");
        Path file = folder.resolve("scores.txt");
        try {
            Files.writeString(file, "pony=12\n", StandardCharsets.UTF_8);
            System.out.print(Files.readString(file, StandardCharsets.UTF_8));
        } finally { Files.deleteIfExists(file); Files.deleteIfExists(folder); }
    }
}
```

### Output

```
pony=12
```

04 / 12

## Cover append and directory

The complete scoreboard is usually saved each time it is overwritten, and chat logs may be appended. Making the wrong selection can cause old records to be duplicated or new records to disappear.

### Default writing will overwrite old content

| Element           | Explanation                                  |
|-------------------|----------------------------------------------|
| Overwrite         | File contains only new content               |
| Append            | Keep the original text and add it            |
| Table of Contents | createDirectories creates parent directories |

Before writing a file each time, tell whether to overwrite or append. Don't wait for players to discover that the archive is 100,000 lines long.

## Acceptance comparison table

| Task                                | Mode              | Result                                   |
|-------------------------------------|-------------------|------------------------------------------|
| Complete archive                    | Overwrite         | The file only contains the latest status |
| Event Log                           | Append            | New content is placed at the end         |
| Create directory for the first time | createDirectories | Existing directories are also safe       |

## L29Example03.java · Complete independent example

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
public class L29Example03 {
    public static void main(String[] args) throws IOException {
        Path folder = Files.createTempDirectory("java-write-");
        Path file = folder.resolve("log.txt");
        try {
            Files.writeString(file, "old\n", StandardCharsets.UTF_8);
            Files.writeString(file, "New\n", StandardCharsets.UTF_8);
            Files.writeString(file, "Append\n", StandardCharsets.UTF_8,
                StandardOpenOption.APPEND);
            System.out.print(Files.readString(file, StandardCharsets.UTF_8));
        } finally { Files.deleteIfExists(file); Files.deleteIfExists(folder); }
    }
}
```

## Output

```
New
Append
```

Default writing creates the file or truncates the old file. APPEND only appends, this file already exists; to support first creation, pass CREATE and APPEND at the same time. Only do coverage experiments in dedicated test directories.

05 / 12

## Each record occupies one line

Text archiving begins with a human-readable format. One record per line, the left side of the equal sign is the player name, and the right side is the integer score.

## Simple format that can be inspected by both humans and programs

| Element         | Explanation                                        |
|-----------------|----------------------------------------------------|
| Name            | Not empty, does not contain equal sign and newline |
| Score           | Non-negative int integer                           |
| One line record | name=score                                         |

The archive name does not contain equal signs, CR, LF or leading and trailing spaces, and the spaces in the middle can be retained. The fraction is a non-negative int; the name and number still need to be verified after splitting into two parts.

### L29Example04.java · Complete independent example

```
public class L29Example04 {
    static boolean validName(String name) {
        return name != null && !name.isBlank() && name.equals(name.strip())
            && !name.contains("=") && !name.contains("\n") && !name.contains("\r");
    }
    public static void main(String[] args) {
        System.out.println(validName("pony"));
        System.out.println(validName("A=B"));
        System.out.println(validName(" Alex "));
        String[] parts = "pony=12".split("=", 2);
        System.out.println(parts[0]);
        System.out.println(Integer.parseInt(parts[1]));
    }
}
```

### Output

```
true
false
false
pony
12
```

This format allows spaces in the middle of the name, but rejects leading and trailing blanks, empty names, equal signs, CR and LF. The fraction is an integer from 0 to the int upper limit. The 2 of split limits the number of splits and will not check for you whether there is still an equal sign in the second paragraph.

06 / 12

## Parse line by line and reject bad data

The loader cannot pretend that the file is always perfect. Each line is checked for delimiters, names and numbers.

**split limit 2 retain the remaining content on the right**

| Element        | Explanation                                 |
|----------------|---------------------------------------------|
| Format check   | Separators and legal names                  |
| Digital Check  | Analysis, non-negative, integer upper limit |
| Duplicate name | The first legal record is retained          |

One line of damage doesn't have to give up the whole world. Record line numbers and skip bad lines, other valid players can still load.

For duplicate names, the first legal record is retained and the line number is reported. In order to focus on the parsing rules in this chapter, the reading and writing methods are placed in the same complete file; copy the entire code segment, not just main.

**L29Example05.java · Paragraph 1/3, sequentially connected to the same file**

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
public class L29Example05 {
    static boolean validName(String name) {
        return name != null && !name.isBlank() && name.equals(name.strip())
            && !name.contains("=") && !name.contains("\n") && !name.contains("\r");
    }
    static void save(Path path, Map<String, Integer> scores) throws IOException {
        if (path == null || scores == null)
            throw new IllegalArgumentException("Paths and scoreboards cannot be null");
        List<String> lines = new ArrayList<>();
        for (var entry : scores.entrySet()) {
            if (!validName(entry.getKey()) || entry.getValue() == null
                || entry.getValue() < 0)
                throw new IllegalArgumentException("The archive record is illegal");
        }
        for (var entry : new TreeMap<>(scores).entrySet())
            lines.add(entry.getKey() + "=" + entry.getValue());
        Path parent = path.getParent();
        if (parent != null) Files.createDirectories(parent);
        Files.write(path, lines, StandardCharsets.UTF_8);
    }
    static Map<String, Integer> load(Path path) throws IOException {
        if (path == null) throw new IllegalArgumentException("Path cannot be null");
        List<String> lines;
        try { lines = Files.readAllLines(path, StandardCharsets.UTF_8); }
    }
}
```

**L29Example05.java · Section 2/3, sequentially connected to the same file**

```
        catch (NoSuchFileException missing) { return new HashMap<>(); }
        Map<String, Integer> scores = new HashMap<>();
        int lineNumber = 0;
        for (String line : lines) {
            lineNumber++;
            String[] parts = line.split("=", 2);
            if (parts.length != 2 || !validName(parts[0])) {
                System.out.println("Skip bad lines=" + lineNumber);
                continue;
            }
            try {
                int score = Integer.parseInt(parts[1].strip());
                if (score < 0) throw new NumberFormatException("Negative fraction");
                if (scores.containsKey(parts[0])) {
                    System.out.println("Skip duplicate names=" + lineNumber);
                    continue;
                }
                scores.put(parts[0], score);
            } catch (NumberFormatException error) {
                System.out.println("Skip bad scores=" + lineNumber);
            }
        }
        return scores;
    }

    public static void main(String[] args) throws IOException {
        Path folder = Files.createTempDirectory("java-load-");
        Path file = folder.resolve("scores.txt");
        try {
```

**L29Example05.java · Section 3/3, sequentially connected to the same file**

```
            Files.write(file, List.of("Alex=12", "bad line", "Mia=Many",
                "Alex=99", "pony=8", "Zero=-1"), StandardCharsets.UTF_8);
            Map<String, Integer> loaded = load(file);
            System.out.println(loaded.get("Alex"));
            System.out.println(loaded.get("pony"));
            System.out.println(loaded.size());
        } finally { Files.deleteIfExists(file); Files.deleteIfExists(folder); }
    }
}
```

**Output**

```
Skip bad lines=2
Skip bad scores=3
Skip duplicate names=4
Skip bad scores=6
12
8
2
```

07 / 12

## Missing files and IOException

The archive not existing may only be running for the first time and is not necessarily an error. Insufficient permissions or disk failure are the reasons why reading and writing cannot be completed.

### First startup returns empty scoreboard

| Element           | Explanation                             |
|-------------------|-----------------------------------------|
| File missing      | Return empty scoreboard                 |
| Other IOException | Continue to spread and explain the path |
| Bad deeds         | Skip after reporting bank number        |

IOException If it cannot be recovered, the path and reason are displayed. Don't return empty data but tell the player the load was successful.

### L29Example06.java · Section 1/2, sequentially connected to the same file

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
public class L29Example06 {
    static boolean validName(String name) {
        return name != null && !name.isBlank() && name.equals(name.strip())
            && !name.contains("=") && !name.contains("\n") && !name.contains("\r");
    }
    static void save(Path path, Map<String, Integer> scores) throws IOException {
        if (path == null || scores == null)
            throw new IllegalArgumentException("Paths and scoreboards cannot be null");
        List<String> lines = new ArrayList<>();
        for (var entry : scores.entrySet()) {
            if (!validName(entry.getKey()) || entry.getValue() == null
                || entry.getValue() < 0)
                throw new IllegalArgumentException("The archive record is illegal");
        }
        for (var entry : new TreeMap<>(scores).entrySet())
            lines.add(entry.getKey() + "=" + entry.getValue());
        Path parent = path.getParent();
        if (parent != null) Files.createDirectories(parent);
        Files.write(path, lines, StandardCharsets.UTF_8);
    }
    static Map<String, Integer> load(Path path) throws IOException {
        if (path == null) throw new IllegalArgumentException("Path cannot be null");
        List<String> lines;
        try { lines = Files.readAllLines(path, StandardCharsets.UTF_8); }
    }
```

## L29Example06.java · Paragraph 2/2, sequentially connected to the same file

```
catch (NoSuchFileException missing) { return new HashMap<>(); }
Map<String, Integer> scores = new HashMap<>();
int lineNumber = 0;
for (String line : lines) {
    lineNumber++;
    String[] parts = line.split("=", 2);
    if (parts.length != 2 || !validName(parts[0])) {
        System.out.println("Skip bad lines=" + lineNumber);
        continue;
    }
    try {
        int score = Integer.parseInt(parts[1].strip());
        if (score < 0) throw new NumberFormatException("Negative fraction");
        if (scores.containsKey(parts[0])) {
            System.out.println("Skip duplicate names=" + lineNumber);
            continue;
        }
        scores.put(parts[0], score);
    } catch (NumberFormatException error) {
        System.out.println("Skip bad scores=" + lineNumber);
    }
}
return scores;
}

public static void main(String[] args) throws IOException {
    Path folder = Files.createTempDirectory("java-missing-");
    try { System.out.println(load(folder.resolve("missing.txt")).size()); }
    finally { Files.deleteIfExists(folder); }
}
```

### Output

0

Only `NoSuchFileException` is considered for missing archives on first launch. Permissions, directories, files, encoding corruption, and other `IOExceptions` will propagate upward; they cannot be disguised as an empty world that is successfully loaded.

08 / 12

## Round trip test verification archive

The most important test of file functionality is the round trip: prepare data, save, load, and compare the contents.

## Round trip test: Compare again after walking two sections

| Step                  | Explanation                   |
|-----------------------|-------------------------------|
| 1. Before saving      | Prepare Map content           |
| 2. Round trip reading | Save and then load            |
| 3. Compare            | The content is equal to pass. |

The automatic test uses a special temporary directory. After the test is completed, the test files are deleted and do not touch the player's real world.

## Acceptance comparison table

| Scene     | Operation                               | Expected                |
|-----------|-----------------------------------------|-------------------------|
| Normal    | Save two players and load again         | Map content is the same |
| Border    | Save empty Map                          | Loading is empty        |
| Character | Player name contains Chinese characters | UTF-8 remains unchanged |
| Error     | Bad number row                          | Prompt and skip         |
| missing   | Reading file that does not exist        | Return empty Map        |

**L29Example07.java · Paragraph 1/3, sequentially connected to the same file**

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
public class L29Example07 {
    static boolean validName(String name) {
        return name != null && !name.isBlank() && name.equals(name.strip())
            && !name.contains("=") && !name.contains("\n") && !name.contains("\r");
    }
    static void save(Path path, Map<String, Integer> scores) throws IOException {
        if (path == null || scores == null)
            throw new IllegalArgumentException("Paths and scoreboards cannot be null");
        List<String> lines = new ArrayList<>();
        for (var entry : scores.entrySet()) {
            if (!validName(entry.getKey()) || entry.getValue() == null
                || entry.getValue() < 0)
                throw new IllegalArgumentException("The archive record is illegal");
        }
        for (var entry : new TreeMap<>(scores).entrySet())
            lines.add(entry.getKey() + "=" + entry.getValue());
        Path parent = path.getParent();
        if (parent != null) Files.createDirectories(parent);
        Files.write(path, lines, StandardCharsets.UTF_8);
    }
    static Map<String, Integer> load(Path path) throws IOException {
        if (path == null) throw new IllegalArgumentException("Path cannot be null");
        List<String> lines;
        try { lines = Files.readAllLines(path, StandardCharsets.UTF_8); }
    }
}
```

**L29Example07.java · Paragraph 2/3, sequentially connected to the same file**

```
        catch (NoSuchFileException missing) { return new HashMap<>(); }
        Map<String, Integer> scores = new HashMap<>();
        int lineNumber = 0;
        for (String line : lines) {
            lineNumber++;
            String[] parts = line.split("=", 2);
            if (parts.length != 2 || !validName(parts[0])) {
                System.out.println("Skip bad lines=" + lineNumber);
                continue;
            }
            try {
                int score = Integer.parseInt(parts[1].strip());
                if (score < 0) throw new NumberFormatException("Negative fraction");
                if (scores.containsKey(parts[0])) {
                    System.out.println("Skip duplicate names=" + lineNumber);
                    continue;
                }
                scores.put(parts[0], score);
            } catch (NumberFormatException error) {
                System.out.println("Skip bad scores=" + lineNumber);
            }
        }
        return scores;
    }

    public static void main(String[] args) throws IOException {
        Path folder = Files.createTempDirectory("java-roundtrip-");
        Path file = folder.resolve("scores.txt");
        Map<String, Integer> original = Map.of("pony", 12, "Alex", 0);
```

**L29Example07.java · Paragraph 3/3, sequentially connected to the same file**

```
        try {
            save(file, original);
            System.out.println(original.equals(load(file)));
            save(file, Map.of());
            System.out.println(load(file).isEmpty());
        } finally { Files.deleteIfExists(file); Files.deleteIfExists(folder); }
    }
}
```

**Output**

```
true
true
```

09 / 12

## Challenge Blueprint Scoreboard Archive

The challenge program is called WorldSave. It saves the Map as UTF-8 text, loads it, and validates the two players and total count.

### Sort by key to save and load safely line by line

| Element             | Explanation                 |
|---------------------|-----------------------------|
| Stable preservation | TreeMap sort by name        |
| Safe entrance       | Verify all records first    |
| Already archived    | The main program only loads |

The save method explicitly overwrites the file; the main program creates the demo file for the first time, and only loads existing files subsequently. Sorting by key is easy to view and does not mean that backup or atomic writes have been implemented.

### Acceptance sequence

- Save Alex=12 and Steve=8.
- The file uses UTF-8 and has one line per person.
- After loading, query by name to get the same score.
- Missing files return an empty Map.
- Print a prompt for bad lines and continue reading.

10 / 12

## World Archive Program

Create a new WorldSave.java, compile and run using Java 21.

## Complete save and load process

| Element         | Explanation                    |
|-----------------|--------------------------------|
| First run       | Create demo file               |
| Run again       | Do not overwrite existing data |
| Failure display | Path and exception type        |

Manually add bad line and Mia=many at the end of the file, and then load it; two bad lines should be prompted, and Alex and Steve still exist.

## WorldSave.java · Paragraph 1/3, sequentially connected to the same file

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
public class WorldSave {
    static boolean validName(String name) {
        return name != null && !name.isBlank() && name.equals(name.strip())
            && !name.contains("=") && !name.contains("\n") && !name.contains("\r");
    }
    static void save(Path path, Map<String, Integer> scores) throws IOException {
        if (path == null || scores == null)
            throw new IllegalArgumentException("Paths and scoreboards cannot be null");
        List<String> lines = new ArrayList<>();
        for (var entry : scores.entrySet()) {
            if (!validName(entry.getKey()) || entry.getValue() == null
                || entry.getValue() < 0)
                throw new IllegalArgumentException("The archive record is illegal");
        }
        for (var entry : new TreeMap<>(scores).entrySet())
            lines.add(entry.getKey() + "=" + entry.getValue());
        Path parent = path.getParent();
        if (parent != null) Files.createDirectories(parent);
        Files.write(path, lines, StandardCharsets.UTF_8);
    }
    static Map<String, Integer> load(Path path) throws IOException {
        if (path == null) throw new IllegalArgumentException("Path cannot be null");
        List<String> lines;
        try { lines = Files.readAllLines(path, StandardCharsets.UTF_8); }
    }
```

## WorldSave.java · Paragraph 2/3, sequentially connected to the same file

```
catch (NoSuchFileException missing) { return new HashMap<>(); }
Map<String, Integer> scores = new HashMap<>();
int lineNumber = 0;
for (String line : lines) {
    lineNumber++;
    String[] parts = line.split("=", 2);
    if (parts.length != 2 || !validName(parts[0])) {
        System.out.println("Skip bad lines=" + lineNumber);
        continue;
    }
    try {
        int score = Integer.parseInt(parts[1].strip());
        if (score < 0) throw new NumberFormatException("Negative fraction");
        if (scores.containsKey(parts[0])) {
            System.out.println("Skip duplicate names=" + lineNumber);
            continue;
        }
        scores.put(parts[0], score);
    } catch (NumberFormatException error) {
        System.out.println("Skip bad scores=" + lineNumber);
    }
}
return scores;
}

public static void main(String[] args) {
    Path path = args.length == 0
        ? Path.of("saves", "lesson29-scores.txt") : Path.of(args[0]);
    try {
```

## WorldSave.java · Paragraph 3/3, sequentially connected to the same file

```
        if (Files.notExists(path)) {
            save(path, Map.of("Alex", 12, "Steve", 8));
            System.out.println("First run: Demonstration archive created");
        } else {
            System.out.println("Already archived: only loaded this time");
        }
        Map<String, Integer> loaded = load(path);
        System.out.println("File=" + path.getFileName());
        System.out.println("Alex=" + loaded.get("Alex"));
        System.out.println("Steve=" + loaded.get("Steve"));
        System.out.println("Number of players=" + loaded.size());
    } catch (IOException error) {
        System.out.println("Failed to read and write: " + path.toAbsolutePath());
        System.out.println(error.getClass().getSimpleName());
    }
}
}
```

## Output

```
Already archived: only loaded this time
File=lesson29-scores.txt
Alex=12
Steve=8
Number of players=2
```

The above is the output of the first run in the dedicated directory. Run it again and change the first line to "Existing archive: only load this time"; the program will not overwrite the manually modified data. save itself explicitly performs an overwrite, not an automatic backup, atomic write, or multiplayer server save system.

11 / 12

## Expansion tasks and clearance review

The job must indicate the actual path to the file, encoding, coverage or append method, and submit a round trip test.

### The third-level task continues to expand the archive system

| Element                      | Explanation                            |
|------------------------------|----------------------------------------|
| Encoding evidence            | Explicit UTF-8                         |
| Path evidence                | Record absolute address                |
| Reading and writing evidence | Round trip testing and bad row testing |

Lesson 30: Send mining records to the data pipeline, and use filter, map, sorted and sum to generate statistical briefings.

### Third gear operation

- Wooden Pickaxe: Save a Chinese notice as UTF-8, then read and display it.
- Iron Pickaxe: Save multiple player scores, skip bad lines when loading.
- Diamond Pickaxe: Add the version number in the first line. If the version is not supported, it will refuse to load and give a prompt.

## Acceptance comparison table

| Self-evaluation             | Can                      | Still need to practice   |
|-----------------------------|--------------------------|--------------------------|
| I will use Path and Files   | <input type="checkbox"/> | <input type="checkbox"/> |
| I can explicitly use UTF-8  | <input type="checkbox"/> | <input type="checkbox"/> |
| I will do a round trip test | <input type="checkbox"/> | <input type="checkbox"/> |

12 / 12

## Own acceptance record

Beautiful output is the starting point, and being able to explain boundaries is considered mastery. Select one case each from normal, borderline and error, write down your expectations before running, and leave evidence after running.

### Leave a basis for judgment

| Category | The operation you selected                   | Check key points                                                       |
|----------|----------------------------------------------|------------------------------------------------------------------------|
| Normal   | A common input or a successful action        | Is the result consistent line by line?                                 |
| Border   | Range endpoints and adjacent values          | Is there one more step or one less thing?                              |
| Error    | Illegal quantity, missing object or bad data | Whether it is explicitly rejected and whether the status is maintained |

- Bug records: triggering conditions, actual phenomena, causes, and minimum repairs.
- Illustrated record: describe the starting point, end point and meaning of each arrow.
- Adaptation record: only change one rule, and modify expectations and tests at the same time.

Technical check: Java SE 21 language specification and standard library. See the language specification for class, interface, type and expression rules; see the corresponding API for the behavior of collections, Random, Files and Stream.

<https://docs.oracle.com/javase/specs/jls/se21/html/index.html>

<https://docs.oracle.com/en/java/javase/21/docs/api/index.html>