13/10/25

# **CastleBlack Daily Mini-Coding Exercises (Task #1 → #80)**

# INDEX

[CastleBlack Daily Mini-Coding Exercises (Task #1 → #80) 1](#_Toc1256080209)

[INDEX 1](#_Toc887702115)

[Week 1: JSON Mastery (Oct 13–19) 2](#_Toc1951732029)

[Task1 — Parse a Simple JSON in Java 4](#_Toc2121009046)

[Concept 4](#_Toc1319249055)

[JSON Input 4](#_Toc1593701334)

[Step-by-Step Logic 5](#_Toc93590005)

[Extra Challenge (Optional) 5](#_Toc47753218)

[ESCAPE CHARACTER 6](#_Toc74904054)

[What’s Actually Happening 6](#_Toc107720386)

[Clean version (if reading from file or multi-line) 6](#_Toc48225351)

[What is ObjectMapper? 8](#_Toc832735063)

[You can think of it like this: 8](#_Toc1263820795)

[1. readTree() – Parse JSON into a Tree (JsonNode) 9](#_Toc960812032)

[2. readValue() – Parse JSON into a Java Object 10](#_Toc1020264529)

[3. writeValueAsString() – Convert Java object → JSON string 11](#_Toc373456940)

[Summary Table 11](#_Toc1761472544)

[In Your Code Snippet: 11](#_Toc1522423944)

[What is Classpath? 11](#_Toc1286364448)

[How Classpath Works 12](#_Toc385712789)

[Default Classpath 12](#_Toc586606537)

[Setting Classpath 13](#_Toc739377530)

[a) Using -cp or -classpath with java or javac 13](#_Toc276759726)

[b) Using CLASSPATH environment variable 13](#_Toc1078898381)

[Examples 14](#_Toc532889444)

[⃣ Common Mistakes 14](#_Toc1807707861)

[Breakdown of build.gradle 15](#_Toc259208334)

[Dependencies Section 15](#_Toc386708121)

[1. Jackson Databind 16](#_Toc1965127903)

[2. Log4j2 16](#_Toc2024468828)

[3. Lombok 17](#_Toc1859940842)

[4. JUnit 5 17](#_Toc2033441107)

# Week 1: JSON Mastery (Oct 13–19)

|  |  |  |  |
| --- | --- | --- | --- |
| **Day** | **Task #** | **Exercise** | **Expected Output** |
| Mon, Oct 13 | #task1 | Parse simple JSON: {"name":"MouldA","material":"Steel"}. Print name & material. | Console: MouldA, Steel |
|  | #task2 | Add missing attribute check: weight. Print "N/A" if not present. | Console: N/A |
| Tue, Oct 14 | #task3 | Parse nested array JSON: {"data":[{"SKU":"123","PRT":"A1"},{"SKU":"124","PRT":"B2"}]}. Print all SKUs + PRTs. | Console: 123:A1, 124:B2 |
|  | #task4 | Fetch attribute with Optional, handle nulls safely. | Console: value or "N/A" |
| Wed, Oct 15 | #task5 | Update first element’s SKU to 999. Print updated JSON. | Console: updated JSON string |
|  | #task6 | Add new element to array: {"SKU":"555","PRT":"C3"}. | Console: JSON array with new element |
| Thu, Oct 16 | #task7 | Remove an element from JSON array based on PRT value "B2". | Console: array without B2 |
|  | #task8 | Read JSON from file using ClassPathResource. | Print JSON content |
| Fri, Oct 17 | #task9 | Combine parsing, updating, adding/removing into one class. | Console shows all operations |
|  | #task10 | Edit & resend simulation: update a failed attribute, print before/after. | Console: before/after JSON |
| Sat, Oct 18 | #task11 | Loop over multiple JSON files and merge SKUs into a single list. | Print merged SKUs |
| Sun, Oct 19 | #task12 | Bonus: Validate JSON keys exist against mandatory list [SKU, PRT, Material]. | Console warnings for missing |

# Task1 — Parse a Simple JSON in Java

**Goal:** Learn to read and extract values from a JSON object using Jackson (most commonly used JSON library in Java).

### **Concept**

JSON (JavaScript Object Notation) is a lightweight data-interchange format.  
 In Java, we commonly use **Jackson** (com.fasterxml.jackson.databind.ObjectMapper) to parse or generate JSON.

Jackson converts:

* JSON → Java objects (Deserialization)
* Java objects → JSON (Serialization)

### **JSON Input**

We’ll use this simple JSON:

{

"name": "MouldA",

"material": "Steel"

}

Output

Name: MouldA

Material: Steel

## Step-by-Step Logic

|  |  |  |
| --- | --- | --- |
| **Step** | **What happens** | **Why** |
| 1 | ObjectMapper created | It’s your JSON interpreter |
| 2 | JSON string defined | Your input data |
| 3 | mapper.readTree(json) | Parses the string into a tree structure (JsonNode) |
| 4 | .get("name").asText() | Fetches the text value of the name key |
| 5 | Printed on console | Confirms successful parsing |

### **Extra Challenge (Optional)**

1. Change "material":"Steel" to "weight":10 → see what happens if "material" is missing.
2. Try to fetch a missing field using .get("weight") — it’ll give null, not "N/A".  
    (That’s what we’ll handle in **#task2** tomorrow.)

## ESCAPE CHARACTER

That mysterious \ is not some Matrix glitch — it’s just **an escape character** used in Java **strings**.

### **What’s Actually Happening**

In Java, when you write:

String json = "{\"name\":\"MouldA\",\"material\":\"Steel\"}";

The \ is used **to escape the double quotes** inside the string — because Java itself uses double quotes to mark string boundaries.

So:

"{\"name\":\"MouldA\"}"

…is how Java understands this literal string:

{"name":"MouldA"}

### **Clean version (if reading from file or multi-line)**

If this annoys you, you can:

String json = """  
{  
 "name": "MouldA",  
 "material": "Steel"  
}  
""";

That’s a **text block** (since Java 15), and no need to escape quotes anymore.  
 Much cleaner for testing or static JSON templates.

So in short:

* \" = quote inside string
* [\\](file:///\\) = literal backslash
* \n = newline
* \t = tab

They’re all **escape sequences**.

Common Escape Sequences in Java:

|  |  |  |  |
| --- | --- | --- | --- |
| **Escape** | **Meaning** | **Example** | **Output** |
| \" | Double quote inside string | "He said, \"Hi\"" | He said, "Hi" |
| [\\](file:///\\) | A single backslash | "C:\\Users\\Tony" | C:\Users\Tony |
| \n | Newline | "Hello\nWorld" | HelloWorld |
| \t | Tab (4 spaces visually) | "Name:\tTony" | Name: Tony |
| \' | Single quote | 'I\'m Stark' | I'm Stark |
| \r | Carriage return | "Hello\rWorld" | (Depends on system, rarely used now) |
| \b | Backspace | "ABC\bD" | ABD (B erased visually) |

# What is ObjectMapper?

ObjectMapper is the **main class** of the **Jackson library** — your go-to tool for handling JSON in Java.  
 Think of it as a **translator** between JSON and Java objects.

It can:

1. **Read JSON** → Convert it into a Java object or a JsonNode.
2. **Write JSON** → Convert a Java object into a JSON string.

### **You can think of it like this:**

ObjectMapper mapper = new ObjectMapper();

This line says:

“Create a Jackson translator who understands JSON grammar.”

### 1. readTree() – Parse JSON into a Tree (JsonNode)

JsonNode root = mapper.readTree(jsonString);

This creates a **tree-like structure** (like a JSON DOM).  
 You can then navigate through it:

root.get("name").asText();

So if your JSON was:

{  
 "name": "MouldA",  
 "material": "Steel"  
}

Then root.get("name") points to "MouldA".

Use this when you **don’t have a predefined Java class** and just want to “walk” through the JSON.

### 2. readValue() – Parse JSON into a Java Object

Mould mould = mapper.readValue(jsonString, Mould.class);

If you have a Java POJO like:

class Mould {  
 private String name;  
 private String material;  
  
 // Getters and Setters  
}

Then readValue maps the JSON keys directly to Java fields.

Use this when you **know the structure of JSON** and want to work with Java objects.

### **3**. writeValueAsString() – Convert Java object → JSON string

String json = mapper.writeValueAsString(mould);

That’ll give you:

{"name":"MouldA","material":"Steel"}

Use this when sending JSON in an API response or file.

### Summary Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Input** | **Output** | **Use When** |
| readTree() | JSON string | JsonNode | You want to **explore** JSON dynamically |
| readValue() | JSON string + class type | Java object | You know the structure |
| writeValueAsString() | Java object | JSON string | You need JSON output |

### **In Your Code Snippet:**

JsonNode mouldProcurementDataObjectNode = new ObjectMapper().readTree(mouldProcurementData);

It means:

“Hey Jackson, take this JSON text I just fetched, and convert it into a JsonNode tree so I can read values using .get().”

What is Classpath?

The **classpath** is the **path that the Java Virtual Machine (JVM) and Java compiler use to look for classes and packages** needed for your Java program.

Think of it like this: your program says, new MyClass(). JVM now needs to find MyClass.class. The **classpath tells it where to look**—in directories, JAR files, or both.

### How Classpath Works

* It can point to:
  + **Directories** containing compiled .class files.
  + **JAR files** (Java archives containing multiple classes).
  + **ZIP files** (less common nowadays, similar to JAR).
* If a class is not found in the classpath, you get a **ClassNotFoundException** or **NoClassDefFoundError**.

### Default Classpath

If you **don’t specify a classpath**, Java defaults it to the **current working directory (.)**.

java MyProgram

This looks for classes in the current folder.

### Setting Classpath

There are **two ways**:

### a) Using -cp or -classpath with java or javac

javac -cp "lib/mylib.jar;." MyProgram.java # Windows  
javac -cp "lib/mylib.jar:." MyProgram.java # Linux/Mac  
java -cp "lib/mylib.jar;." MyProgram # Windows

### b) Using CLASSPATH environment variable

# Windows  
set CLASSPATH=C:\myproject\lib\mylib.jar;.  
  
# Linux/Mac  
export CLASSPATH=/home/user/myproject/lib/mylib.jar:.

**Tip:** It's generally recommended to use -cp instead of environment variables to avoid confusion.

### **Examples**

Suppose your project structure is:

project/  
 ├─ bin/  
 │ └─ MyProgram.class  
 └─ lib/  
 └─ utils.jar

To run MyProgram:

java -cp "bin;lib/utils.jar" MyProgram # Windows  
java -cp "bin:lib/utils.jar" MyProgram # Linux/Mac

This tells Java to look in bin for your classes and utils.jar for library classes.

### **⃣** Common Mistakes

1. Forgetting to include your compiled classes in the classpath.
2. Using ; vs : incorrectly (Windows vs Linux).
3. Relying on default classpath when running from a different folder.

## **Breakdown of build.gradle**

plugins {  
 id 'java'  
}

✅ Enables Java build support.  
 This gives you compileJava, test, jar, etc. tasks automatically.

group = 'org.stark'  
version = '1.0-SNAPSHOT'

✅ Declares your project’s group and version metadata (used in JAR naming).  
 Your future JAR will be:

org.stark-1.0-SNAPSHOT.jar

repositories {  
 mavenCentral()  
}

✅ Tells Gradle where to fetch dependencies from — mavenCentral() is the standard global repository.

Dependencies Section

#### 1. Jackson Databind

implementation group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: '2.0.1'

Brings in the ObjectMapper, JsonNode, and everything you need for JSON parsing.  
 **But** — 2.0.1 is *very old* (from 2012).  
 You should bump it up to something recent (like 2.17.2 or 2.16.0).  
 Otherwise, you might run into bugs or missing features.

**Recommended:**

implementation 'com.fasterxml.jackson.core:jackson-databind:2.17.2'

### 2. Log4j2

implementation 'org.apache.logging.log4j:log4j-api:2.21.0'  
runtimeOnly 'org.apache.logging.log4j:log4j-core:2.21.0'

Perfect for when we move into logging — matches your “CastleBlack monitoring” theme.  
 (We’ll use it around #task4 or #task5 when we start logging JSON parsing results.)

#### 3. Lombok

compileOnly("org.projectlombok:lombok:1.18.42")  
annotationProcessor("org.projectlombok:lombok:1.18.42")

Adds support for annotations like @Getter, @Setter, @ToString.  
 You’re future-proofing already.

#### 4. JUnit 5

testImplementation platform('org.junit:junit-bom:5.10.0')  
testImplementation 'org.junit.jupiter:junit-jupiter'

Sets up JUnit 5 for testing.  
 You’ll use this later when you start testing your JSON parsing methods.

### 

test {  
 useJUnitPlatform()  
}

Ensures Gradle uses JUnit 5 instead of the legacy engine.

# LOGGER

### **The Problem**

This line:

Logger log = new Logger(Main.class);

is **invalid**, because Logger in Log4j2 is an **interface**, not a concrete class.  
 You can’t create it with new.  
 Instead, you **ask LogManager to give you one**.

### **2️⃣ The Correct Way**

Use **LogManager.getLogger()** to obtain a logger instance:

package org.stark;  
  
import org.apache.logging.log4j.LogManager;  
import org.apache.logging.log4j.Logger;  
  
public class Main {  
  
 private static final Logger log = LogManager.getLogger(Main.class);  
  
 public static void main(String[] args) {  
 log.info("System initialized successfully.");  
 log.warn("Low battery detected... oh wait, that’s just my sarcasm level.");  
 log.error("Critical failure — Jarvis refused to run on Windows XP.");  
 }  
}

### **3️⃣ Why This Works**

* LogManager.getLogger(Main.class) returns a **logger object** specifically tied to your class.
* Behind the scenes, Log4j2 uses its **factory pattern** to:
  + Find or create the logger.
  + Bind it with configuration from log4j2.xml or log4j2.properties.

### **4️⃣ Typical Setup**

Add this to your build.gradle (if you haven’t already):

dependencies {  
 implementation 'org.apache.logging.log4j:log4j-api:2.24.0'  
 implementation 'org.apache.logging.log4j:log4j-core:2.24.0'  
}

And create a simple log4j2.xml in your src/main/resources folder:

<?xml version="1.0" encoding="UTF-8"?>  
<Configuration status="INFO">  
 <Appenders>  
 <Console name="Console" target="SYSTEM\_OUT">  
 <PatternLayout pattern="%d{HH:mm:ss} | %-5level | %c{1} - %msg%n" />  
 </Console>  
 </Appenders>  
 <Loggers>  
 <Root level="info">  
 <AppenderRef ref="Console" />  
 </Root>  
 </Loggers>  
</Configuration>

Now, when you run your program:

12:30:45 | INFO | Main - System initialized successfully.  
12:30:45 | WARN | Main - Low battery detected... oh wait, that’s just my sarcasm level.  
12:30:45 | ERROR | Main - Critical failure — Jarvis refused to run on Windows XP.

Would you like me to show **how Log4j2 internally maps LogManager.getLogger() → configuration → actual logging output** (i.e., its internal flow)? It’s worth knowing if you’re learning logging in depth.

##### **You said:**

yes

###### **ChatGPT said:**

Excellent, Mr. Stark — let’s lift the hood on **how Log4j2 actually works internally** when you call:

Logger log = LogManager.getLogger(Main.class);  
log.info("System initialized.");

We’ll walk through this **step-by-step**, like tracing a signal from your suit’s sensor to J.A.R.V.I.S.’ neural core.

## **⚙️ 1️⃣ The Flow Overview**

Your Code  
 ↓  
LogManager  
 ↓  
LoggerContext  
 ↓  
Configuration (log4j2.xml / log4j2.properties)  
 ↓  
Appenders (Console, File, RollingFile, etc.)  
 ↓  
Layout (Formatting pattern)  
 ↓  
Output (Console, Log file, Socket, etc.)

Let’s dissect each layer.

## **🧠 2️⃣ Step-by-Step Internal Flow**

### **Step 1: Your Call**

You write:

Logger log = LogManager.getLogger(Main.class);

* LogManager is a **factory class**.
* It calls LoggerContext.getLogger() to either:
  + Retrieve an existing Logger instance (from cache), or
  + Create a new one if none exists.

⚡ Log4j2 maintains a *singleton-like registry* of loggers, so multiple calls for the same class return the same object.

### **Step 2: LoggerContext**

Think of **LoggerContext** as the **brain** managing all loggers in your application.  
 It holds:

* Configuration (loaded from XML, JSON, YAML, or properties)
* Logger hierarchy (like root → org.stark → org.stark.service)

It decides **how to configure each Logger**—its level (INFO, DEBUG, ERROR), and what appenders (destinations) it will use.

### **Step 3: Configuration Loading**

When your app starts:

* Log4j2 **automatically searches** your classpath for configuration files in this order:
  + log4j2-test.xml
  + log4j2.xml
  + log4j2-test.properties
  + log4j2.properties

If none are found, it loads a **default configuration** (console appender, level = ERROR).

So even if you don’t define a config, it will still print errors to the console.

### **Step 4: Logger Object is Returned**

LogManager now returns a fully configured Logger object to your variable:

private static final Logger log = LogManager.getLogger(Main.class);

This log instance is lightweight — it delegates all the heavy lifting to Log4j2’s asynchronous background threads (depending on config).

### **Step 5: Logging the Message**

When you call:

log.info("System initialized.");

Internally, this happens:

1. **Check log level**:  
    Log4j2 checks if INFO is enabled for this logger.  
    If the current level is higher (say ERROR), it skips everything (performance optimization).
2. **Create a LogEvent**:  
    It wraps your message, timestamp, thread info, class name, etc. in a LogEvent object.
3. **Pass LogEvent to Appenders**:  
    The event travels through configured appenders like:
   1. ConsoleAppender
   2. FileAppender
   3. RollingFileAppender
   4. SocketAppender
   5. AsyncAppender
4. **Layout Formatting**:  
    Each appender uses a Layout to format your message, typically:

%d{HH:mm:ss} | %-5level | %c{1} - %msg%n

1. **Output Destination**:  
    Finally, the formatted log line is written to:
   1. Console (System.out)
   2. File (like application.log)
   3. Or even a remote server (for distributed logging)

### **Step 6: Asynchronous Logging (Optional but Awesome)**

Log4j2 has an **AsyncLogger** mechanism using the **LMAX Disruptor** library (ultra-fast ring buffer).  
 It can process logs on a background thread — no blocking your main thread.  
 This can give **10x+ performance boost** in production.

You can enable it in your config:

<Configuration status="INFO" packages="">  
 <Appenders>  
 <Console name="Console" target="SYSTEM\_OUT">  
 <PatternLayout pattern="%d{HH:mm:ss} | %-5p | %c{1} - %m%n" />  
 </Console>  
 </Appenders>  
 <Loggers>  
 <Root level="info">  
 <AppenderRef ref="Console"/>  
 </Root>  
 <AsyncLogger name="org.stark" level="debug" />  
 </Loggers>  
</Configuration>

## **🪄 7️⃣ Internal Design Pattern Summary**

|  |  |
| --- | --- |
| **Pattern** | **Role** |
| **Factory Pattern** | LogManager creates and manages loggers |
| **Singleton Pattern** | LoggerContext and configuration are shared globally |
| **Observer Pattern** | Appenders observe log events |
| **Builder Pattern** | Used for configuration objects (Log4j2 2.x design) |
| **Asynchronous Queues** | Using Disruptor for high-speed non-blocking logging |

### **✅ In a Nutshell:**

* LogManager → Factory that gives you a logger
* LoggerContext → Manages all loggers + configuration
* Configuration → Defines where and how to log
* Appenders → Send logs to destinations
* Layouts → Format the output
* AsyncLogger → Sends logs via fast async buffer