Analysis Report: Java Code `CsvToJsonConverter`

Saturday 5th April, 2025

Contents

1 Introduction

This report documents the Java code within the `converter` package, specifically the `CsvToJsonConverter` class. Its objective is to explain in detail the purpose of the code, its functionality, and provide a higher-level overview of its implementation. The code is designed to read data from a CSV file (`HCLEcatalog.csv`), interpret it according to specific business logic, transform it into a structured data model comprising specific entities (`Item`, `Document`, and related types) and relationships, and finally serialize this model into a **JSON Lines** file (`data.json`). This format, where each line is a valid JSON object representing a node or a relationship, is specifically intended for bulk import into graph databases like Neo4j. It incorporates logic to handle missing or invalid data by assigning the string literal `"N/A"` to *specific* fields under defined conditions, while other invalid fields might be omitted from the final JSON output.

2 Purpose of the Code

The `CsvToJsonConverter` code has been developed with the primary purpose of converting a data catalog stored in a tabular CSV (Comma Separated Values) format into a **JSON Lines** format suitable for graph database ingestion, particularly Neo4j.

More specifically, the code is designed to:

- Read data from a specific CSV file (`HCLEcatalog.csv`).
- Interpret each CSV row as representing a primary entity, classifying it definitively as either an `Item` or a `Document` based on the presence of specific fields (`ToC`, `Extent`, `SerialNum`, `BibCit`).
- Extract and map data from CSV columns to the fields of well-defined Java objects, representing Items, Documents, Persons, Organizations, Categories, and Materials.
- Apply data validation logic: If certain fields required for `Document` or `Item` identification are invalid or missing (null/empty/"None"), assign the literal string `"N/A"` to specific target fields (`toc`, `extent`, `serialNum`, `bibCit`, `created` for Documents; `partNum` for Items). Other fields (like common fields `description`, `wherMade`, or `Item.conditionNts`) are populated only if valid, otherwise the Java object field remains `null` and is omitted from the JSON output via Jackson's `NON_NULL` setting.
- Identify and create related entities such as persons (`Person`), organizations (`Organization`), categories (`Category`), and materials (`Material`) from data in other columns (e.g., `Creator`, `SubjectTop`, `Material`).
- Establish meaningful relationships between these entities (e.g., an `Item` "belongsTo" a `Collection`, a `Document` is "developed by" a `Person`, an `Item` is "madeOf" a `Material`). Note that relationship identifiers involving `Items` incorporate the `partNum`, which might be `"N/A"`. Relationships are represented as distinct JSON objects in the JSON Lines output.
- Ensure the uniqueness of the extracted entities (avoiding duplicates) by using a `Map` (`entityToIdentityMap`) keyed by the entity objects (relying on their `equals()` and `hashCode()` methods). Each unique entity is assigned a unique integer ID.
- Create an initial root node representing the collection (`label: "Collection", name: "HCLE"`) before processing the CSV data.

- Generate a log file (`parsing.log`) that tracks the processing status and reports any errors or warnings for each CSV row.
- Produce a final JSON Lines file (`data.json`), where each line is a self-contained JSON object representing either a unique node (`jtype: "node"`) or a relationship (`jtype: "relationship"`) in a structure optimized for Neo4j import. The JSON is formatted with spaces after colons and commas. Fields explicitly assigned the value `"N/A"` are included as strings in the output.

In summary, the code acts as a bridge between a tabular CSV format and a graph-database-ready JSON Lines format, applying domain-specific classification logic (Item vs. Document) and specific rules for handling missing/invalid data (assigning `"N/A"` to certain fields, omitting others).

3 Main Functionality (What it Does)

The code executes a series of logical steps to achieve its purpose:

1. **CSV Reading and Parsing:**
   - Opens the specified CSV file (`HCLEcatalog.csv`) using UTF-8 encoding.
   - Utilizes the Apache Commons CSV library to interpret the file.
   - Recognizes the first row as the header, ignoring case in column names (`withHeader().withIgnoreHeaderCase()`).
   - Trims leading and trailing whitespace from each read value (`withTrim()`).
   - Iterates over each record (row) in the CSV file, excluding the header.
2. **Initial Node Creation:**
   - Before processing CSV rows, creates and writes a JSON Line object for the root collection node (`label: "Collection", name: "HCLE", identity: 0`).
   - Initializes the identity counter for subsequent nodes/relationships.
3. **Row Validation and Data Extraction:**
   - For each row, extracts the value from the `IdNum` column. If it is missing or effectively empty (`isNullOrEmpty`), the row is skipped, and an error is logged.
   - Extracts values from all other columns defined in the header and stores them in a map (`Map<String, String>`) for easy access. Values are stored as extracted (null if missing).
4. **Determining the Main Entity Type (Document or Item):**
   - Checks if any of the Document-specific fields (`ToC`, `Extent`, `SerialNum`, `BibCit`) have a non-null/non-"None" value in the row data (`!isNullOrNone`).
   - If any of these fields are present and valid, the entity type is determined to be `Document`. Otherwise, it is classified as `Item`.
5. **Creating the Main Entity Object (Document or Item):**
   - Instantiates a Java object of the determined class (`Document` or `Item`).

- Populates the object's fields with values extracted from the corresponding CSV columns, applying specific logic for handling invalid/missing values:
  - **For Documents:**
    - Sets `toC`, `extent`, `serialNum`, `bibCit` fields to the string `"N/A"` if the corresponding CSV value is null, empty, or "None" (`isNullOrNone`). Otherwise, uses the valid CSV value.
    - Sets `created` field: Uses the value from `Created`, falls back to `DateCR` if the former is invalid, then sets the field to `"N/A"` if the final resulting value is null, empty, or "None". Otherwise, uses the valid date string.
    - Sets `copyrighted` field: Converts 'y' to `true`, 'n' or '0' to `false`. If the value is missing, invalid, or "None", the field remains `null` and is **omitted** from the JSON output (no "N/A" applied).
  - **For Items:**
    - Sets `partNum` field: Sets to the string `"N/A"` if the corresponding CSV value is null, empty, "None", or consists of only a single digit (`matches("^\d$")`). Otherwise, uses the valid CSV value.
    - Sets `conditionNts` field only if the corresponding CSV value is valid (not null, empty, or "None"). If invalid, the field remains `null` and is **omitted** from the JSON output.
  - **For Common Fields (Description, DescComment, WherMade):** Populates these fields on the `Document` or `Item` object only if the corresponding CSV value is valid (not null, empty, or "None"). If invalid, the fields remain `null` and are **omitted** from the JSON output.

6. **Identifying and Creating Related Entities:**
   - **Material:** If the `Material` column contains 'papr', 'digi', or 'mix', creates a `Material` object ("paper", "digital", "mix"). Unrecognized non-empty values are logged as warnings.
   - **Category:** If the `SubjectTop` column has a valid value (`!isNullOrNone`), creates a `Category` object.
   - **Person/Organization (from Creator, Contributor, AddlAuth):** Uses regex and helper methods (`parsePerson`, `processContributorField`) to attempt parsing strings as `Person` objects. If parsing fails or the string doesn't match the person pattern, it's treated as an `Organization`. Handles lists of names separated by delimiters.
7. **Uniqueness Management and ID Assignment:**
   - Uses a **`HashMap<BaseEntity, Integer> entityToIdentityMap`** to track unique entities already processed and assigned an ID, relying on the `equals()` and `hashCode()` methods in the Java object classes.
   - Uses an `AtomicInteger identityCounter` to assign sequential, unique integer IDs (`identity`) to each new, unique entity.
   - The `getOrCreateEntityIdentity` method manages this process, ensuring each unique entity is written as a node object exactly once.
8. **Relationship Creation:**

- `belongsTo` (Entity -> Collection): Created for every valid `Item` or `Document`, linking to the "HCLE" node. The object identifier includes the title and, for Items, the potentially `"N/A" partNum`. The original `IdNum` is stored as a property.
- `madeOf` (Item -> Material): Created only for `Items` with an identified `Material`.
- `describe` (Category -> Document): Created **only** if the main entity is a `Document` and a `Category` was identified.
- **Creator Relationships** (`developed` / `produced`): Created **only** if the main entity is a `Document` and a `Creator` (Person/Organization) was identified.
- **Contributor/AddlAuth Relationships** (`developed/produced/collaborated`): Created for both `Items` and `Documents` if contributors/authors are identified via `processContributorField`.
- Relationships are written as distinct JSON Line objects using the `writeRelationship` method.

9. **Logging:**
   - Logs processing status, errors (missing IdNum/Title), and warnings (unrecognized Material code, failed person parsing) to `output/parsing.log`.
   - Logs a summary at the end with node and relationship counts by type.

10. **JSON Output Generation:**
    - Writes output to `output/data.json` in **JSON Lines** format.
    - Each line represents a single node (`jtype: "node"`) or relationship (`jtype: "relationship"`) object.
    - Uses Jackson's `ObjectMapper` (configured with `JsonInclude.Include.NON_NULL`) for serialization, omitting fields that were left `null` in the Java objects.
    - Fields explicitly assigned the string `"N/A"` in the Java objects (`toC`, `extent`, `serialNum`, `bibCit`, `created`, `partNum` under specific conditions) are included in the JSON output as string values.
    - Uses `formatJsonString` to add spaces after colons and commas for readability.

11. **Error Handling:**
    - Uses try-with-resources and standard exception handling (`IOException`, `Exception`, `UncheckedIOException`) for robust file processing and error reporting.

4 Implementation Overview (How it Works)

This section provides a high-level view of how the code achieves its functionality.

- **External Libraries:** The implementation relies on Apache Commons CSV for parsing the CSV file and Jackson Databind for converting Java objects to JSON strings.
- **Java Object-Based Structure:** The data model uses Java object classes (`Item`, `Document`, `Person`, etc.). `Artifact` serves as a base class. Specific fields in `Item` and `Document` (`toC`, `extent`, `serialNum`, `bibCit`, `created`, `partNum`) are defined to potentially hold the string `"N/A"`. Other fields might be omitted if null via Jackson's `@JsonInclude(JsonInclude.Include.NON_NULL)` annotation. The `equals()` and `hashCode()` methods in these Java classes are crucial for ensuring entity uniqueness via the `entityToIdentityMap`.

- **Main Class and Control Flow:** The `CsvToJsonConverter` class orchestrates the process. It writes the initial "HCLE" node, then loops through CSV records. Entity type determination is binary (Document or Item). Field population logic includes specific checks for assigning `"N/A"` based on the rules defined in Section 3, while other invalid fields are set to `null` for omission.
- **Uniqueness and ID Management:** A `HashMap<BaseEntity, Integer> entityToIdentityMap` and an `AtomicInteger identityCounter` manage unique entity IDs for the JSON Lines output. The `getOrCreateEntityIdentity` method ensures each unique entity is written as a node object exactly once.
- **JSON Lines Output:** The code generates JSON Lines output suitable for Neo4j import. Nodes (`jtype: "node"`) and relationships (`jtype: "relationship"`) are written as separate JSON objects on each line using `writeNode` and `writeRelationship`.
- **JSON Formatting:** The `formatJsonString` method adds spaces after colons/commas to the serialized JSON strings for readability.
- **Conditional Logic and Regex:** Entity classification is based on specific fields. Regex is used for person name parsing and `Item.partNum` validation (`^\d$`).
- **Use of Standard Collections:** Uses `Map<String, String>` for row data and `Map<BaseEntity, Integer>` for uniqueness/ID management.
- **File and Resource Management:** Try-with-resources and UTF-8 encoding are employed for proper resource handling and character support.

5 Definition of Entities and Relationships

The following main data structures (Java classes) were defined for the data model, which generates JSON Lines output based on these structures:

Java Object Definitions

- `BaseEntity`: Abstract base class providing `entityType` and `getNodeLabel()` (`@JsonIgnore`).
- `Artifact`: Base class for Item and Document, providing common fields (`title`, `description`, etc.). Annotated `@JsonInclude(Include.NON_NULL)`.
- `Item`: Subclass of `Artifact`. Represents a physical object. `partNum` field **can hold the string "N/A"**. `conditionNts` is omitted if null. `entityType` is "Item". Uses `title` and `partNum` for `equals/hashCode`.
- `Document`: Subclass of `Artifact`. Represents a document. `toC`, `extent`, `serialNum`, `bibCit`, `created` fields **can hold the string "N/A"**. `copyrighted (Boolean)` field is omitted if null. `entityType` is "Document". Uses `title` for `equals/hashCode`.
- `Person`: Represents a person (`name`, `surname`). `entityType` is "Agent:Person". Uses `name`, `surname` for `equals/hashCode`.
- `Organization`: Represents an organization (`name`). `entityType` is "Agent:Organization". Uses `name` for `equals/hashCode`.

- `Category`: Represents a category (`name`). `entityType` is "ContentDescription:Category". Uses `name` for `equals/hashCode`.
- `Material`: Represents material (`name`). `entityType` is "Material". Uses `name` for `equals/hashCode`.

Jackson annotations (`@JsonInclude`, `@JsonProperty`, `@JsonIgnore`) control JSON serialization, including the omission of null fields (unless explicitly assigned "N/A") and JSON field naming. All Java classes extending `BaseEntity` effectively inherit or have the `@JsonInclude(Include.NON_NULL)` setting applied by Jackson during serialization.

JSON Lines Output Structure

The code generates a `data.json` file where each line is a separate JSON object representing either a node or a relationship for Neo4j import.

- **Node Object Structure Example:**

```
    {"jtype": "node", "identity": 1, "label": "Document", "
       properties": {"title": "...", "toc": "N/A", "created": "..."
       }}
```

  (Note: `extent`, `serialNum`, etc., might be omitted if null, or present with value `"N/A"`)

  - `jtype` is always "node".
  - `identity` is a unique integer.
  - `label` is the Neo4j node label (e.g., "Item", "Document", "Person").
  - `properties` contains the relevant fields. Fields assigned `"N/A"` appear as strings. Fields left `null` in Java are omitted. All values are converted to strings.

- **Relationship Object Structure Example:**

```
    {"jtype": "relationship", "subject": 1, "object": 0, "name": "
       belongsTo", "properties": {"originalIdNum": "1288"}}
```

  - `jtype` is always "relationship".
  - `subject` is the source node ID.
  - `object` is the target node ID.
  - `name` is the relationship type.
  - `properties` contains relationship properties (as strings). Empty `{}` if none.

The generated JSON is formatted using `formatJsonString` to include spaces after colons and commas.

6 Proposed Ontology Modifications

To effectively represent the data converted by the `CsvToJsonConverter` and capture the nuances present in the source CSV, the following modifications to the target ontology are proposed:

- Add an entity `General:Material` with an attribute `Name` (expandable if needed).
- Add a relationship `General:MadeOf` between `Item` and `Material`.

- Add an attribute `General:Item:conditionNotes`.
- Add an attribute `General:Document:ToC`.

These additions allow for the explicit modeling of material composition, condition details for items, and table of contents information for documents, as extracted by the converter.

7 CSV Field Mapping and Relationships

This section details how the source CSV fields are mapped to the proposed ontology attributes and how relationships are established by the converter.

7.1 Field Mapping (CSV → Ontology)

The mapping of CSV fields to ontology attributes is defined as follows:

- `Title` → `General:Document:Title` or `General:Item/Artifact:Name`
- `Description` → `description` of the entity (for both `General:Document` and `General:Item/Artifact`).
- `DescComment` → `Notes` (for both `General:Document` and `General:Item/Artifact`).
- `SubjectTop` → `name` of `General:ContentDescription:Category`.
- `DonorNotes` → Ignored.
- `WherMade` →
  - `General:Item:madeIn` if Item
  - `General:Document:Place` if General:Document
  - Not applicable if General:Artifact (Note: The code classifies as Item or Document, so this case might not occur directly from the primary entity logic).
- `Color` → Foreign key to an external table; unusable without the original table. (Ignored by converter).
- `Material` → `General:Material:name` (after reformatting, e.g., papr → paper).
- `ItemCondition` → Foreign key to an external table; unusable without the original table. (Ignored by converter).
- `ConditionNts` → `General:Item:conditionNotes`.
- `Fragility` → Foreign key to an external table; unusable without the original table. (Ignored by converter).
- `Functional` → Ignored.
- `Power` → Ignored.
- `ToC` → `ToC` of the Document (`General:Document:ToC`). Set to "N/A" if invalid.
- `Creator` → `General:Agent:Person` or `Organization` (regex-based).
- `Publisher` → `General:Agent:Organization` (`name`). (Note: Converter logic currently processes this via `processContributorField`, potentially creating `Person` or `Organization`).

- Contributor → General:Agent:Person or Organization.
- AddlAuth → General:Agent:Person or Organization.
- Created → General:Document:date. Set to "N/A" if invalid (after checking DateCR).
- Copyrighted → Boolean (False for 0 or n, True for y; otherwise, empty/omitted). Maps to General:Document:useRights. (Note: Converter stores as Boolean, omits if null).
- DateCR → General:Document:date when Created is unknown/invalid.

Fields beyond this point in the CSV are either predominantly empty or deemed of limited interest for this conversion process. Note that the converter logic assigns "N/A" to Extent, SerialNum, and BibCit for Documents if the source is invalid, and to PartNum for Items under specific invalid conditions. These correspond conceptually to ontology attributes but the "N/A" value signifies missing/invalid source data.

7.2 Relationships

The relationships established between entities by the converter are as follows:

- IdNum is stored as an attribute (originalIdNum) of the **belongsTo** relationship connecting each Item or Document to the root HCLE Collection node.
- **describe** between ContentDescription:Category and Document.
- **madeOf** between Item and Material.
- **developed** between Person and Document (derived from Creator, Contributor, or AddlAuth if identified as a Person).
- **produced** between Organization and Document (derived from Creator, Contributor, or AddlAuth if identified as an Organization).
- **collaborated** between Person or Organization and Item or Document (derived from Contributor or AddlAuth fields). The specific relationship type (developed, produced, collaborated) depends on the source field and the identified agent type.

8 Estimation of the errors

A qualitative assessment of the data quality within the generated JSON file (data.json) was performed as part of the development process to identify potential error patterns or inconsistencies arising from the source data or conversion logic.
This assessment involved manual inspection and pattern analysis of the node and relationship objects within the JSON data.

8.1 Creators and contributors

Analysis of the classification logic for creators and contributors indicated potential errors in distinguishing between Person and Organization entities based solely on name patterns present in the source CSV. The following resume summarizes the estimated classification accuracy based on tests run during development:
This suggests potential limitations in relying solely on name format (via regex) to distinguish between people and organizations in the source data fields (Creator, Contributor, AddlAuth), likely originating from inconsistencies in the source CSV.

Classified as ↓ / Actual →	Person	Organization
Person	**n** (Correct)	**13** (Incorrect)
Organization	**71** (Incorrect)	**m** (Correct)

Table 1: Confusion matrix for People/Organizations classification. Placeholders n, m represent counts estimated during testing.

8.2 Summary

Based on the development process and analysis of the generated `data.json` file:

- **Entity Misclassification (People/Orgs):** Testing estimated approximately 84 instances of potential misclassification when distinguishing between Persons and Organizations based on name patterns.
- **Placeholder/Incomplete Data:** The design includes assigning the string `"N/A"` to specific fields (`toc`, `extent`, `serialNum`, `bibCit`, `created`, `partNum`) when source data is invalid according to defined rules. Other fields with invalid source data are omitted entirely from the JSON. This selective approach reflects the completeness and validity of the source data for those specific fields, affecting dozens to hundreds of records.
- **Formatting Inconsistencies:** While the code includes logic (`formatJsonString`) to standardize JSON formatting per line, potential inconsistencies might still exist within field values inherited directly from the source data (e.g., variations in date formats if not fully normalized by the 'created' field logic, or inconsistencies in free-text fields).