Permutation.java – Simple Guide (Best-Of v6.3)

Readable walkthrough, Amazon steps analogy, missed-decoding explanation, and real-life uses.

Date: 25 Jan 2026

Focus Decode a digit string into letters (1=A ... 26=Z) by splitting into 1-digit or 2-digit chunks.

Example "2431221" (7 digits)

Key idea Treat chunking like taking steps of size 1 or 2 up a staircase whose height is the string length.

Reject rule Any chunk that is 0, starts with 0 (like "06"), or is above 26 is invalid and that path is discarded early.

1. The principle in plain English

You start at the left of the digit string and repeatedly choose one of two moves:

- Take 1 digit (a 1-step): that forms a number 0–9.
- Take 2 digits (a 2-step): that forms a number 00–99.
- Every time you take a step, you turn that number into a letter using 1=A ... 26=Z.
- If the number is not a valid letter, you reject that route immediately.

Amazon steps analogy (why it helps)

...

All valid step patterns for this example (10 total):

Step pattern	Chunking of "2431221"	Letter result
2-1-2-2	24 | 3 | 12 | 21	XCLU
1-1-1-2-2	2 | 4 | 3 | 12 | 21	BDCLU
2-1-1-1-2	24 | 3 | 1 | 2 | 21	XCABU
2-1-1-2-1	24 | 3 | 1 | 22 | 1	XCAVA
2-1-2-1-1	24 | 3 | 12 | 2 | 1	XCLBA
1-1-1-1-1-2	2 | 4 | 3 | 1 | 2 | 21	BDCABU
1-1-1-1-2-1	2 | 4 | 3 | 1 | 22 | 1	BDCAVA
1-1-1-2-1-1	2 | 4 | 3 | 12 | 2 | 1	BDCLBA
2-1-1-1-1-1	24 | 3 | 1 | 2 | 2 | 1	XCABBA
1-1-1-1-1-1-1	2 | 4 | 3 | 1 | 2 | 2 | 1	BDCABBA

•

2. Walkthrough for "2431221" (the simple way)

Start: You are at the beginning of the string: 2431221.

- At each point you can take 1 digit or 2 digits.
- Two possible starts: If you take 1 digit first, you take "2" which maps to B. If you take 2 digits first, you take "24" which maps to X.
- From there, you keep taking 1-digit or 2-digit steps until you consume all 7 digits.
- Any time you create an invalid chunk (for example 31, 43, or anything above 26), you immediately discard that route.

All valid decodings for this input

There are 10 valid decodings for "2431221":

BDCABBA, BDCABU, BDCAVA, BDCLBA, BDCLU, XCABBA, XCABU, XCAVA, XCLBA, XCLU

3. What your Java code is doing (non-technical view)

Your program follows a sensible idea:

- Build the 1..26 map so you can translate digits into letters.
- Pre-filter possible letters by scanning the string (single digits and adjacent pairs). This reduces random search space.
- Generate candidate letter strings (randomly) and then verify them by converting letters back into digits and comparing to the original string.
- Use a Set to avoid storing duplicates.

This is an explorative / discovery approach: it is especially useful when you want to see "what might work" without writing full recursion immediately.

How your program sets up for "2431221" (step-by-step)

This mirrors the walkthrough we discussed in chat, but ties it directly to your main method:

- A) Build the 1→A ... 26→Z map (mp). This lets you translate digits ↔ letters later.
- B) Compute the letter-count bounds: rMax = 7 (all 1-digit chunks) and rMin = 4 (mostly 2-digit chunks). So the program tries r = 4, 5, 6, 7.
- C) Discover "possible letters" by scanning the string: check each single digit (2,4,3,1) and each adjacent 2-digit pair (24,12,22,21). This produces the letter set {A, B, C, D, L, V, U, X}.
- D) Lock the two start routes: first digit "2" gives B, and first two digits "24" give X. These are your two natural entry points.
- E) Run Encoding for each r (4..7). Each run generates candidates and uses checkMappings() to confirm which candidates match the original digits exactly.

4. Why the fixes were introduced (kept simple)

The fixes are small but important. They do not change your overall approach; they simply make it behave consistently and make the “possible letters” filtering actually take effect:

Index vs letter ...

Even with the fixes, random generation is still not guaranteed to find all decodings in a single run:

•

- Finite attempts: you generate a limited number of candidates for each length, so you may never hit a rare valid string.
  Probability: some correct tails are unlikely by chance unless you generate a lot of candidates.
- Duplicates: a Set prevents repeats (good), but it also means many random cycles can be “wasted” re-producing already-seen strings.

To keep your design intact but avoid missing results, the final code includes a safety net: if random finds fewer decodings than expected, it automatically runs the deterministic solver and reports what was missed.

6. What your original version would miss for "2431221" (and why)

This is the key issue in the original (before the index/letter fix): your program had the right idea, but one small wiring mistake made it miss most valid answers.

What went wrong (in plain English)

- You correctly built a short list of eligible letters from the input string: A, B, C, D, L, V, U, X.
- But the random picker produced a small number like 0..7 (an index into that eligible-letter list).
- Then that index was used as if it were a real letter-key 1..26 in mp (the full alphabet map).
- So instead of selecting from {A,B,C,D,L,V,U,X}, it mostly selected from early mp keys (or even null).

Why that causes missed decodings

- For this input, many correct decodings require letters that come from 2-digit chunks:
- X comes from 24, L comes from 12, V comes from 22, and U comes from 21.
- If the generator cannot reliably produce X/L/V/U, it cannot form most correct outputs - even though checkMappings() would have validated them.

What it could find vs what it would likely miss

- The only valid decoding that uses only the very early letters A..D is: BDCABBA (2|4|3|1|2|2|1).
- The other 9 valid decodings require at least one of X/L/V/U, so the original version would typically miss them:
- BDCLBA, BDCLU, BDCABU, BDCAVA, XCLBA, XCLU, XCABBA, XCABU, XCAVA.

After the index/letter fix, those letters become selectable again. However, the search is still random, so you still do not get a 100% guarantee in one run - which is why the deterministic safety net is useful.

7. Real-life uses of this logic

This pattern is "constrained segmentation + search + verification". It appears in many real systems:

- T9 / keypad text / predictive input: numbers can map to many letter sequences; generate candidates then validate against a dictionary.
- Speech recognition / autocorrect: produce several candidate token sequences, then choose the best by scoring rules.
- Barcode / QR / serial decoding: codes are often concatenated fields; decoders try splits and validate length/checksum rules.
- Reference numbers with check digits: try interpretations and validate with modulo checks.
- Network protocol parsing: parse a byte stream with optional/variable-length fields; backtrack if a constraint fails.

-

  Date/time parsing: ambiguous digit strings can be split differently; rules decide valid interpretations.
- Log / telemetry decoding: fields may be concatenated; systems try splits and reject those outside allowed ranges.
- Corrupted data recovery: if separators are lost, you try plausible segmentations that satisfy constraints.

8. Active line count summary (single primary total)

- ChatGPT: 73 (13.7%)Counting active (non-empty, non-comment) lines in the latest version, including presentation
- Amit: 460 (86.3%)output lines (System.out.*), and excluding only the hard-coded test input line (str = "2431221";):
- Total active lines: 533Amit active lines: 460 ChatGPT active lines: 73 Total active lines: 533 ChatGPT share: 13.7%.

(Note: ChatGPT sections use curly braces consistently for fairness and readability.Active = non-empty, non-comment line; includes System.out.*; excludes only str = "2431221";.)
Note: ChatGPT sections use curly braces consistently for fairness and readability.

Addendum: How the code pieces work together

This addendum is intentionally written in the same simple style as the rest of the guide. It explains how the classes and methods connect, and what your thinking looks like in the structure of the code.

How your classes fit together

- Permutation (main driver / orchestrator): builds the 1-26 to A-Z map (mp), sets the encoded string (str), calculates rMin and rMax, scans the string to build a small set of possible letters, then runs the decoder for each target letter-length r (for "2431221" that is 4, 5, 6, 7).
- It also records the two starting routes: start with the first digit as a 1-digit letter ("2" -> B), or start with the first two digits as a 2-digit letter ("24" -> X). That is why your output naturally splits into a B... family and an X... family.
- Combination + Staircase (your Amazon-steps model): expresses the decoding problem as "take 1-step or 2-step moves that add up to the total digits". Each step pattern is a plan for how to split the digit string into 1-digit and 2-digit chunks (example: 2-1-2-2 means 24 | 3 | 12 | 21).
- In your early exploratory versions this part acted like scaffolding: it helped you reason about the search space and the possible lengths r, even if the random generator was not yet fully driven by every step pattern.
- Encoding (candidate generator + checker): creates candidate letter strings for a chosen r. In your design it works like this: generate candidates (randomly), enforce the correct start route (B-start or X-start), keep unique results in a Set, then verify each candidate by converting letters back into digits and comparing against the original string.
- checkMappings() (the verifier): takes a candidate like "XCLU", converts it back into digits (X->24, C->3, L->12, U->21), forms "2431221", and declares it VALID only if the rebuilt digits match the original exactly. This "round-trip" is your correctness guarantee.

What your mental process looks like (from the code)

- Explorative first, then tighten the rules: you started with broad generation (try things, print a lot, observe), then added clearer boundaries like rMin/rMax, "possible letters only", and the two start routes (B-start and X-start).
- You used a physical analogy to control complexity: the Amazon steps idea (1-step or 2-step) turns a tricky parsing problem into a simple "ways to reach the top" problem. That makes it easier to see why some splits are legal and others are not.
- You built safety rails while exploring: using a Set to avoid duplicates, early rejection of invalid paths, and a final strict equality check (convert back to digits and compare). This is a practical way to explore without losing correctness.
- You validate by round-tripping: generate letters -> convert back -> compare. That is a strong engineering habit because it catches subtle mistakes without needing hand reasoning for every case.
- You care about reducing wasted search: the scan that builds a small possibleLettersMap prevents the generator from picking letters that could never appear for the given digits.

Optional note on the deterministic safety net

The deterministic solver does not replace your design. It is there as a minimal safety net: if the random approach finds fewer decodings than expected, the deterministic method lists what was missed. This keeps

your explorative style, but also prevents false confidence when randomness simply did not stumble onto a valid decoding in a particular run.

Update: console output behaviour in the latest code

This page brings the PDF in line with the current Permutation_unique_valid_manual_invalid(2).java behaviour. No algorithm changes are described here — only what gets printed and when.

What prints by default

- Valid translations print once per unique decoding (duplicates are suppressed).
- Invalid translations are available, but they are controlled by a manual comment/uncomment block inside checkMappings(...) — there is no PRINT_INVALID flag in this version.
- The deterministic solver remains a safety net: it runs only if the random approach finishes with fewer unique decodings than expected, and it reports the missing ones.

Where this happens in code

1) Unique-valid printing uses the existing Set you already maintain (e.g. RANDOM_VALID_DECODINGS) as the gate:

```
if (Permutation.RANDOM_VALID_DECODINGS.add(decoding)) { System.out.println("...valid..."); }
```

2) Invalid printing is a commented block you can toggle:

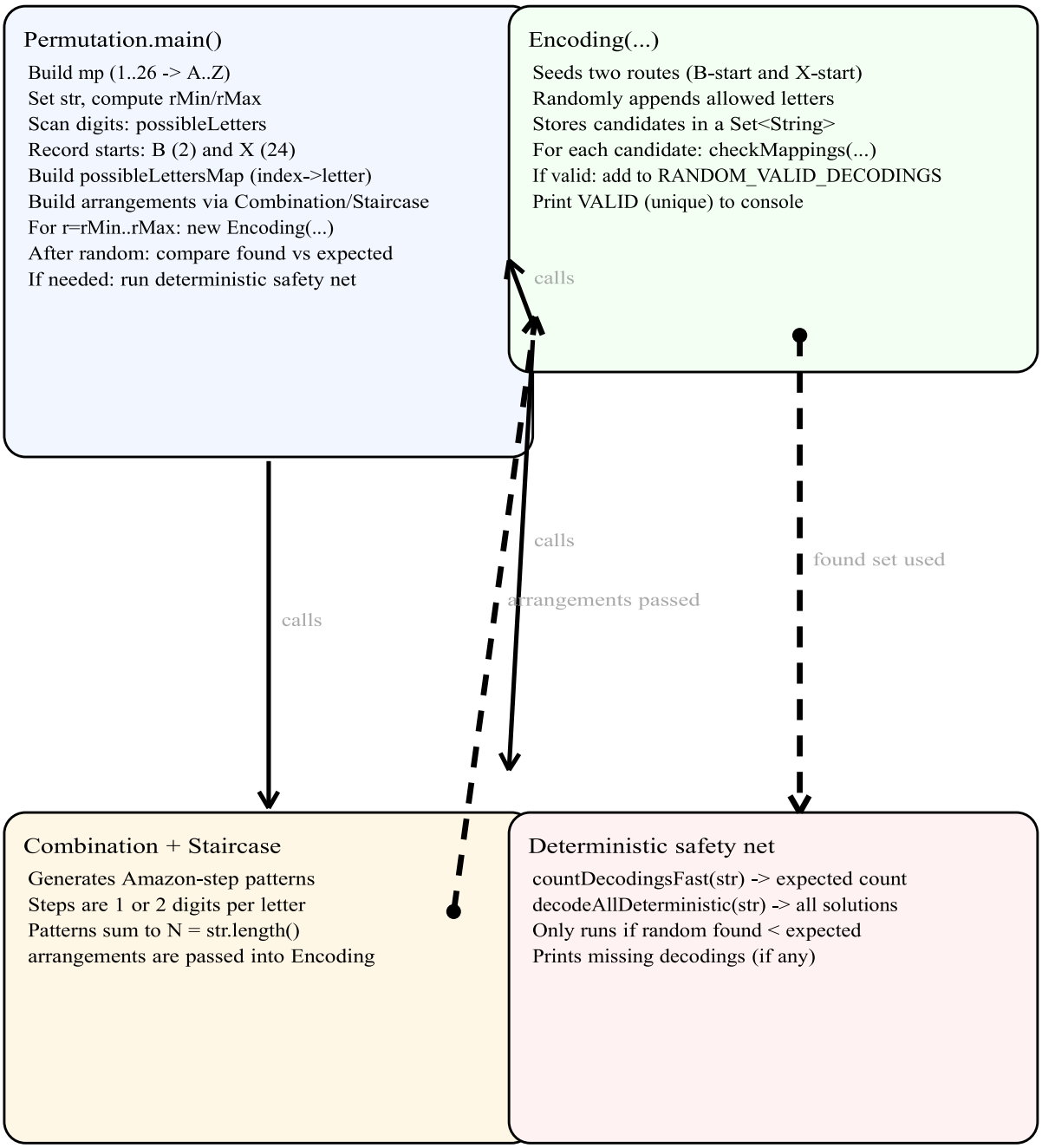
```
// System.out.println("INVALID ..."); // (uncomment if you want invalids)
```

Why this change was made

- It keeps your original design (random exploration) intact, while preventing the console log from exploding with repeated lines.
- It keeps control simple: you can still manually enable invalid printing when you are debugging.

How classes and methods talk together

Solid arrows = direct calls. Dotted arrows = data passed / influences later logic.



How it links right now (in the current code)

- Combination + Staircase is called from main() to build the Amazon-step patterns (arrangements).
- Those arrangements are passed into Encoding(...) (dotted arrow), so the information is available there.
- Encoding(...) still generates candidate letters randomly and verifies them in checkMappings(...);it does not enforce a specific step pattern when building candidates.

Note: main passes arrangements into Encoding; Encoding adds valid results to RANDOM_VALID_DECODINGS; main uses that set to decide whether the deterministic safety net runs.

Console output behaviour (current code)

This section matches the current implementation in Permutation(77).java.

1) Unique valid-only printing (no duplicates)

- When a candidate translation is VALID, the code adds it into a Set first:Permutation.RANDOM_VALID_DECODINGS.add(storedEntry)
- The System.out.println(...) for VALID is guarded by that Set-add, so:
  - the first time a decoding appears: it prints
  - any later repeats: it does not print again

2) Invalid-print debug (manual toggle)

- Invalid candidates are still detected, but the invalid debug prints are meant to becontrolled manually by commenting/uncommenting the System.out.println(...) lines.
- This keeps output files small when you only want the real solutions, but you cantemporarily enable invalid prints for debugging.

CHATGPT ANSWERING ISSUE WITH LETTER SELETION

That change is basically **one core fix** plus **two knock-on edits**, all aimed at making the *random choice* come from the **letters that are actually possible for the user's input** (rather than accidentally picking from A..Z using the wrong key).

Below is the walk-through of what I changed and where it impacts the code.

---

1) What was wrong before (the bug in one sentence)

Your random code was effectively doing:

- generate a random **index** (0..N-1)
- then treat that index as if it were a **1..26 key** into `mp`

That meant the "randomly chosen letter" was not reliably chosen from the set `{A,B,C,D,L,V,U,X}` (for `"2431221"`). It was biased toward early letters or even `null` depending on the exact off-by-one.

---

2) The intended behaviour (what we want)

For a given `str`, you already build a **set of letters that are possible**:

- 1-digit hits: `2→B, 4→D, 3→C, 1→A`
- 2-digit hits (adjacent pairs ≤ 26): `24→X, 12→L, 22→V, 21→U`

So the random generator should pick from:

```
{A, B, C, D, L, V, U, X}
```

---

3) The actual fix (the key change)

A) Main method: build `possibleLettersMap` (index → eligible letter)

This is what makes "random pick by index" meaningful.

Area affected: in `main`, after you populate `possibleLetters`
You then iterate that Set and assign:

```
possibleLettersMap.put(index, iterator.next());
index++;
```

So `possibleLettersMap` becomes something like:

- 0 → A
- 1 → B
- 2 → C
- 3 → D
- 4 → L
- 5 → V
- 6 → U
- 7 → X

(order depends on Set iteration, but contents are correct)

B) Encoding: random selection now uses `possibleLettersMap` (NOT `mp`)

This is the crucial part.

Area affected: inside the big `for (q=0; q<r; q++)` loop in `Encoding(...)`

Old idea (problem):

- you were doing something like `temp1 = rand.nextInt(size) (+1 sometimes)`
- then later `mp.get(temp1)` (wrong map for an index)

New behaviour:

```
temp1   = rand.nextInt(possibleLettersMap.size());  // 0..size-1
mapping = possibleLettersMap.get(temp1);            // actual eligible LETTER
```

Why this matters:

- `possibleLettersMap.size()` matches the number of eligible letters
- keys start at **0**, so **no `+1`**
- `mapping` is now guaranteed to be one of the eligible letters (A/B/C/D/L/V/U/X)

C) Encoding: append uses the chosen eligible letter (`mapping`)

Once `mapping` is an actual letter, appending becomes correct.

Area affected: the parts that used to do:

```
sj.add(mp.get(temp1));
sj1.add(mp.get(temp1));
```

Those are now effectively replaced by:

```
sj.add(mapping);
sj1.add(mapping);
```

So you're not "asking mp for a letter by an index" anymore — you're appending the letter you already chose from the eligible set.

---

4) What this changes for `"2431221"` in practice

Before the fix, your random generator struggled to produce letters like **X, L, V, U** reliably, because it wasn't truly selecting from `{A,B,C,D,L,V,U,X}`.

After the fix:

- X/L/V/U are **equally selectable** (because they're in `possibleLettersMap`)
- so decodings such as **XCLU**, **XCAVA**, **BDCABU**, etc. become realistically discoverable by the random run.

---

5) Summary: exact "areas affected"

1. **`main()`**
    - the loop that scans digits/pairs to populate `possibleLetters`
    - the conversion of `possibleLetters` into `possibleLettersMap`
2. **`Encoding(...)`**
    - the random selection lines:
        - `temp1 = rand.nextInt(possibleLettersMap.size());`
        - `mapping = possibleLettersMap.get(temp1);`
    - the append lines:
        - `sj.add(mapping);`
        - `sj1.add(mapping);`
    - and the older `mp.get(temp1)` append lines are no longer used.

CHATGPT ANSWERING DUPLICATE CONCEPT

They duplicate for a simple reason: **your generator is random and keeps running**, so it can "rediscover" the *same* valid decoding many times.

What's happening in your version:

- You generate lots of candidate letter strings (often millions of attempts).
- For each candidate, `checkMappings()` converts letters → digits and compares with the original `"2431221"`.
- If it matches, you print **immediately**.

But there's nothing stopping the random process from producing the same candidate again later, e.g.:

- `XCLU` might be found at cycle 50,000
- then again at cycle 900,000
- then again at cycle 5,000,000

Every time it passes validation, your code prints it again — so you see duplicates.

Why some duplicates are more common than others

Some valid decodings have a **higher probability** of being generated by your random method (depending on how you seed the start routes and how many "free choices" remain), so they get rediscovered more often. In your sample:

- `BDCLU`, `XCLU`, `XCAVA`, etc. appear multiple times because they're easier to hit again in the random space.
- Another valid one might appear only once because it's rarer in the random search space.

The key point

Duplicates don't mean your decoding is wrong — they just mean:

✅ **Your validation is correct**
✅ **Your random generator keeps sampling**
❌ **You're printing per-hit, not per-unique-hit**