Java Part 9: Building a Small Business Application

Everything from the earlier parts comes together in a small application: variables for values, conditions and loops for rules, methods for reuse, collections for many items, classes for things, and exceptions for trouble. Four steps turn a business need into a working program: design, model, calculate, and run and save. In this part, Walid, who owns a small print shop, uses them to build a quotation tool, and each step answers one practical question. The application stays small on purpose, so that every line can be understood.

1. Design

A good program starts on paper. We write the need in two sentences, then underline the nouns and the verbs. The nouns suggest classes, such as Product, QuoteLine and Quote, and the verbs suggest methods, such as add a line, calculate the total and save. We also decide what the user will do: add lines, see the quote and save it. Last, we separate three concerns: the data and rules, the calculations, and the screen. The question for this step is: which things and which actions does the business really have?

Example: A clinic wants an appointment tool. Before anyone opens an editor, the owner writes two sentences about how the clinic works. The nouns are patient, doctor and appointment, and the verbs are book, cancel and list. The three nouns become three classes, and the three verbs become the methods that the classes offer, so the plan exists on paper first.

At work: Walid writes his need in two sentences: a quote has lines, each line has a product and a quantity, and the quote has a total with tax. He underlines the nouns Product, QuoteLine and Quote, and the verbs add, total and save. He decides the user can add a line, show the quote and save it to a file. He plans that only the main class talks to the screen.

2. Model

Classes that mainly carry data can be written as records. A record declares its fields in one line, and Java generates the constructor, the accessor methods, equals and toString for us. The fields of a record are final, so a record is a safe value that cannot be changed. A compact constructor inside the record can check the data and refuse a negative price. Money stays BigDecimal, created from text, as we learned in the second part. Classes that carry rules, like a quote that adds lines, are written as ordinary classes with private fields and clear methods. Records need a reasonably recent version of Java, so we check the version before we rely on them. The question for this step is: what is data, and what carries rules?

Example: A library models a loan as a record with the book, the member and the due date. The record needs no extra code to print a loan or to compare two loans, because Java generates those methods. Once created, a loan cannot be edited by mistake. The rule about renewals lives in a separate class, which keeps the record simple and the rule in one place.

At work: Walid declares `record Product(String name, BigDecimal unitPrice) {}`, a product that cannot change after creation. A QuoteLine record joins a product with a quantity, and its compact constructor rejects a quantity below one. The Quote class keeps a private list of lines and offers a method to add a line. The data is safe, and every rule sits in one place.

3. Calculate

The calculations belong in methods of the model, so that they can be tested and reused. A line total is the unit price multiplied by the quantity, and the quote total is the sum of all line totals, plus tax. With BigDecimal we multiply and add with methods, and we decide the rounding rule once and apply it in one place. A for each loop with a running total is the simplest way to sum the lines. Streams offer a shorter pipeline: we turn

This document is a property of Asaseeyat Academy - © Copy Right Protected.

the list into a stream, map each line to its total, and reduce the totals into one sum. Both ways are correct, and the more readable one is the better one. The question for this step is: which rule should the program calculate, and where does that rule live?

Example: A restaurant sums the prices of the dishes on one table with a stream. The pipeline turns each dish into its price, and then adds all the prices into one sum. A for each loop with a running total visits the same dishes and gives exactly the same result. Both ways are correct, so the team chooses the one that its members read most easily.

At work: Walid adds a method named total to the Quote class. It multiplies each line's unit price by its quantity, adds the tax rate, and sets the scale to two decimals once, at the end. He first writes it with a for each loop, then compares it with a stream pipeline and keeps the loop because his helper reads it faster. A quick check by hand on a quote of three lines confirms the same total as his old spreadsheet.

4. Run and Save

The last layer talks to the user. A Scanner built on System in reads what the user types, and nextLine returns one line of text. A loop shows the menu, reads the choice and uses a switch to call the right method, and it ends when the user chooses to exit. The main method only wires things together: it creates the objects, runs the menu and delegates every decision to the classes. Invalid input, such as text instead of a quantity, is caught with the exceptions of the previous part, and the menu asks again. To save, the program writes the quote to a text file with the Files class, and it reports where the file went. Keeping the screen code apart from the rules means the same classes could later serve a web page or a mobile app. The question for this step is: how does the user reach every action, and what happens when the input is wrong?

Example: A small bakery uses a console menu to record the daily sales. The menu loops until the owner types exit, and a wrong entry shows a message instead of a crash. When the baker types a word where a quantity belongs, the menu simply asks again. At closing time, one option saves the day's sales to a file, so the figures are kept.

At work: Walid writes a menu with four choices: add a line, show the quote, save it and exit. A switch calls the right method for each choice, and a wrong quantity is caught, so the menu asks again. The save option writes the quote as lines of text to a file named after the customer. In one afternoon he prices a real job with the application, and the tax is never forgotten.

Conclusion

To recap, a small application grows from four steps: design on paper, model with records and classes, calculate with exact money, and run through a simple menu that saves its results. Nouns become classes, verbs become methods, and the screen stays apart from the rules. Walid now has a quotation tool that never forgets the tax and saves every quote. Next time, we will prove that such a program works, with automated tests. A good habit is to build the smallest working version first and extend it step by step.

Sources

- Oracle: The Java Tutorials, Classes and Objects – https://docs.oracle.com/javase/tutorial/java/javaOO/index.html
- dev.java: Learn Java (records, collections and streams) – https://dev.java/learn/
- Oracle: The Java Tutorials, Aggregate Operations (streams) – https://docs.oracle.com/javase/tutorial/collections/streams/index.html
- Oracle: The Java Tutorials, Basic I/O (Scanner and files) – https://docs.oracle.com/javase/tutorial/essential/io/index.html
- Java is a trademark of Oracle and/or its affiliates. This video is independent training material and is not affiliated with Oracle.
- Reviewed 29 Oct 2026 (independent fact check), pages opened: docs.oracle.com/en/java/javase/21/language/records.html (records: generated members, final fields, compact constructor); docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Scanner.html (Scanner(InputStream), nextLine); docs.oracle.com/javase/specs/jls/se21/html/jls-14.html#jls-14.14.2 (enhanced for). dev.java records page returned 404 (UNCHECKED).

This document is a property of Asaseeyat Academy - © Copy Right Protected.