Console Input and Encapsulation

Today, we start with reading user input from the console, or command line. We then examine encapsulation, one of the pillars of object-oriented design.

Manager: Recorder:

Presenter: Reflector:

Content Learning Targets

After completing this activity, you should be able to say:

- I can read various data types from the command line using the built-in `Scanner` class.
- I can explain the difference between the `next()` and `nextLine()` Scanner methods.
- I can add new commands to a command-line interface.
- I can use GenAI tools to rapidly prototype software designs.
- I can explain encapsulation and evaluate how well given UML designs achieve it.

Process Skill Goals

During the activity, you should make progress toward:

- Identifying similarities and differences between syntax for reading different data types from the command line. (Problem Solving)
- Interpreting UML class diagrams. (Information Processing)
- Understanding the context of a problem/system and applying software design principles. (Problem Solving)

Model 1 Console Input using Scanner

Java provides the `Scanner` class for reading user input from the console, a.k.a. command line. Review and run ***src/consoleInput/ConsoleInputMain.java***. Then, use ***ConsoleInputMain.java*** and the Java API docs for the `Scanner` class to answer the following questions.

Questions (15 min) Start time:

1. What type of object does the `Scanner` constructor take as a parameter? In this example, what is the provided argument? To which very familiar Java feature does it connect?

2. Based on the description at the top of the Java API docs for `Scanner`, what are some other options for constructing a `Scanner` object?

3. The example uses the `next()` method to read String inputs. Based on the Java API docs, how does `next()` know when to stop?

4. The example uses `nextInt()` and `nextDouble()` to read integer and floating-point inputs. What type of exception do you get if you try entering an invalid `int`/`double` value?

5. Test out the alternate approaches for reading `age` and `height`. These use static methods from the `Integer` and `Double` wrapper classes to convert the `String` returned by `scanner.nextLine()` into the appropriate type. What happens if we comment out the `scanner.nextLine();` call before reading the age?

Model 2 Command-Line Interface

We can use Java's `Scanner` to implement simple command-line interfaces (CLIs) for our applications. In ***src/animalShelter***, our familiar `AnimalShelterMain` class now has a CLI. Run the program to try out the CLI, then answer the following questions.

Questions (20 min) Start time:

6. How does the CLI allow for an unlimited number of user commands?

7. Try adding extra whitespace at the beginning and end of commands and mixing up the case (i.e., capitalization). What do you notice? What in the code explains your observations?

8. Examine the `handleCommand` method. For the `add animal` command, how does it avoid storing a blank name or species?

9. In pairs, add a new command called ``remove animal'' that:

- prompts the user for an animal ID,
- verifies that an animal with that ID exists,
- prompts the user to confirm the removal with a "y" for yes or "n"/other for no, and
- removes the animal from the shelter records.

When you have finished implementing and testing your remove animal command, compare your solution with that of the other pair in your team.

Model 3 Prototyping Designs (with GenAI)

Sometimes, while we are learning object-oriented design principles, the flaws in a design will not be immediately obvious to us from the UML. In this activity, we see how rapid prototyping with the help of GenAI tools can help us make connections between design flaws in UML and how those flaws manifest in Java code.

Demo (15 min)

Recall the Book Tracker Problem from a previous class.

> A website tracks books and the kids that read them. For each book the system stores the name and author. For each kid the system stores name and grade level. The teacher enters when a kid reads a particular book. It should be possible to print a report on a book that includes all kids who have read a particular book (with their grade level). It should be possible to print a report on a kid that includes the books (with authors) a particular kid has read.

We saw the following design (Fig. 1) does not function correctly (violating principle 1 a/b), because there is no sane way to look up a book for printing a report or associating with a kid (when handling new reading).

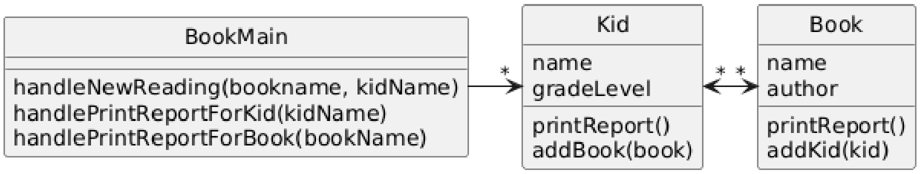

BookMain
handleNewReading(bookname, kidName)
handlePrintReportForKid(kidName)
handlePrintReportForBook(bookName)
*
Kid
name
gradeLevel
printReport()
addBook(book)
* *
Book
name
author
printReport()
addKid(kid)

Figure 1: Bad Design A for the Book Tracker Problem (view PlantUML)

10. The `src/bookTrackerBadA` folder shows the code resulting from Bad Design A. Review the code, and try running it. What issues do you notice?

We saw the following improved design (Fig. 2). To see why this design is better, let's have a GenAI tool refactor our code from Bad Design A to this design.

11. Take notes here on how we prompted the GenAI tool.

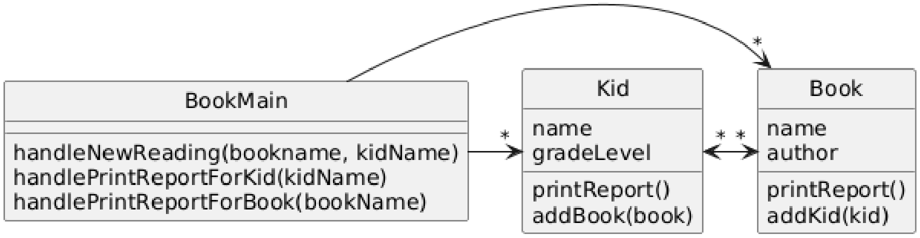

BookMain
handleNewReading(bookname, kidName)
handlePrintReportForKid(kidName)
handlePrintReportForBook(bookName)
Kid
name
gradeLevel
printReport()
addBook(book)
Book
name
author
printReport()
addKid(kid)
*
*
*
*

Figure 2: Improved Design for the Book Tracker Problem (view PlantUML)

12. Compare the ***BookMain.java*** version from Bad Design A with the ***BookMain.java*** version produced by the GenAI tool. What primary differences do you observe?

Model 4 Encapsulation

We end today with one of the pillars of object-oriented software design: ***encapsulation***.

Questions (30 min) Start time:

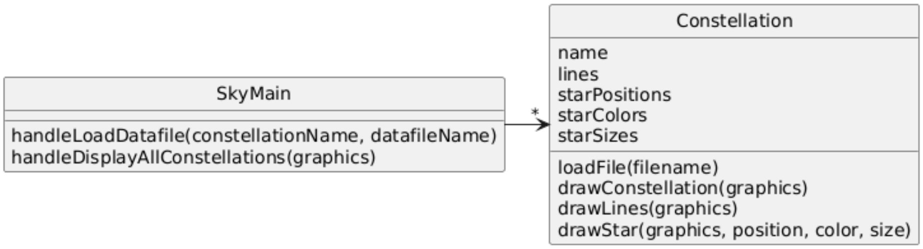
13. If our software system meets all requirements and has no bugs, in what ways could it still be poorly designed?

14. Suppose Java's `Math` and `String` classes were merged into one class, putting methods like `sqrt()` and `substring()` next to each other. What would be some negative consequences of this merger?

15. In Java, it is possible to put all variables and methods for a system in one big class. Why is this a bad idea?

16. Design Principle 3 (OODP 3) states, “Functionality should be distributed efficiently: (a) No class/part should get too large, and (b) Each class should have a single responsibility.” As a team, restate this principle in your own words, using simpler language.

17. **Constellation Design Problem:** A particular program is designed to load constellations from datafiles and draw them on the screen. The datafiles include the names of the constellations and details about star location, size, and color as well as which stars ought to be connected to draw the constellation. Depending on the star data, each star should be drawn differently (e.g., right size, right color).
Apply our first three object-oriented design principles to evaluate the following design.


SkyMain
handleLoadDatafile(constellationName, datafileName)
handleDisplayAllConstellations(graphics)
*
Constellation
name
lines
starPositions
starColors
starSizes
loadFile(filename)
drawConstellation(graphics)
drawLines(graphics)
drawStar(graphics, position, color, size)

(PlantUML source)

18. How could we improve this Constellation design?

Interlude: Defining Encapsulation
Encapsulation refers to the bundling (in a class) of ***related data*** with ***methods that operate on that data***. A well-designed class forms a sort of “capsule” that has a single job and does it well. We can pass around the capsule (as object references), and other classes can use the methods to get things done without needing to know (or interfering with) the private data inside.

Another benefit of encapsulation is that you can often change ***how*** a class works on the inside without needing to change any classes that use that class, as long as the public methods stay the same. For example, if someone comes up with a better way of implementing a HashMap, Java can change the `HashMap` class internals without breaking existing Java code.

19. Come up with a real-world analogy that helps explain the benefits of encapsulation.

20. **Pizza Restaurant Design Problem:** A pizza restaurant needs to calculate the cost of orders and record what pizzas need to be made. An order consists of a number of pizzas, each of which might have toppings, as well as a customer's name and an order date. Each pizza costs $8 with no toppings. The first 2 toppings cost $2 apiece. Each additional topping beyond that costs $1. If a pizza has just peppers, onions, and sausage, that's "The Special" and it costs $12.

a) What classes do you think are needed?

b) Clearly, we need a method to computes a pizza's cost. Where should it go? Why?

c) Compare Solution A and Solution B. Which is a better design, and why?

d) ***For this part and the next one, split into pairs (unless you only have three today). Each pair should choose a different genAI tool.***

Use a GenAI tool to implement both Solution A and Solution B (see Model 3). Prompt it to highlight the differences in the two implementations. Also ask it to evaluate the solutions with respect to OODP 3.

e) Did the GenAI tool faithfully implement each design, or did it change some things without telling you? If something changed, what do you think was the motivation? (Feel free to use follow-up prompts.)

f) Did the genAI tool's assessment of the designs match yours from (c)? Explain.

g) ***Merge your pairs back together and discuss as a full team.*** Which genAI tool gave the better implementation and design comparison? Explain.

h) We could continue extracting classes (e.g., `Customer`, `Topping`). Should we? Explain.