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.

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)

Facilitation Notes

First hour, before quiz: Model 1 introduces console input using the `Scanner` class. Mention that console input will appear in HWVaporSales. Model 2 builds on this to talk about command-line interfaces and walks through an example using the familiar Animal Shelter system. In #9, students pair-program a new command, which has the intended side-effect of reminding them how to remove entries from HashMaps.

Second hour: Model 3 demonstrates using GenAI tools to quickly prototype different UML designs and see their strengths/weaknesses more clearly. Model 4 introduces encapsulation (Design Principle 3) and gives teams opportunities to practice applying it, including one in which they use GenAI tools to prototype alternate designs.

Key questions: #3, #9, #12, #13, #19, #20

Source files: *ConsoleInputMain.java*, *Animal.java*, *AnimalShelterMain.java*, *BookMain.java*, *Book.java*, *Kid.java*

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?

Here, the `Scanner` constructor takes an `InputStream` object named `System.in`. This is the input counterpart to the familiar `System.out`, which is the `PrintStream` object we use to print messages to the console.

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?

We can also construct a Scanner that reads from a File or String.

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

The `next()` method reads the next ***complete token***, which is a sequence of characters between ***delimiters*** (by default, whitespace characters, but they could be commas, semicolons, etc.).

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?

`InputMismatchException`

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?

We get a `NumberFormatException` after entering the name. The two calls to `next()` did not consume the newline character, `\n`, so we passed an empty string to `Integer.parseInt`.

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?

A `while` loop processes one command at a time, quitting only when it receives the "quit" command.

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?

The CLI ignores extra leading/trailing whitespace, and can handle any mix of uppercase/lowercase letters. This is accomplished using the `trim()` and `toLowerCase()` methods from the `String` class on the result of `scanner.nextLine()` when reading the next command.

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

It checks the trimmed input String and replaces it with "`UNKNOWN`" if empty.

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.

See solution copy of ***AnimalShelterMain.java***.

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).

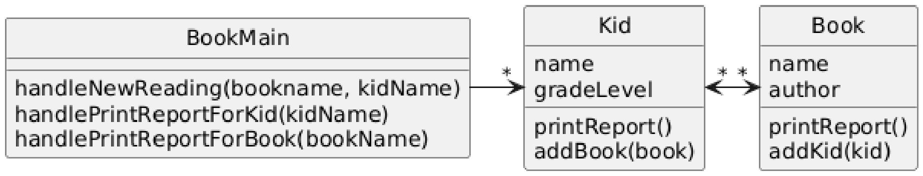

BookMain
handleNewReading(bookname, kidName)
handlePrintReportForKid(kidName)
handlePrintReportForBook(bookName)
*
Kid
name
gradeLevel
printReport()
addBook(book)
* *
Book
name
author
printReport()
addKid(kid)

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?

The `handleNewReading` method creates a new `Book` object (with author "Unknown") every time since `BookMain` can't access existing books directly. The `handlePrintBookNames()` method reveals we may accidentally create multiple books with the same name. The `handlePrintReportForBook` method has to look through every kid's books to try to find the target book, which is very indirect and confusing.

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.

See sample Claude.AI conversation. Prompts should clearly ask for refactoring, give the old design and code, and give the new design. Be sure to mention the designs are in PlantUML.

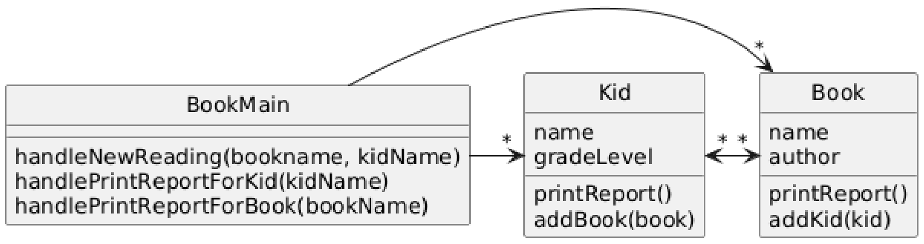

BookMain
handleNewReading(bookname, kidName)
handlePrintReportForKid(kidName)
handlePrintReportForBook(bookName)
Kid
name
gradeLevel
printReport()
addBook(book)
Book
name
author
printReport()
addKid(kid)
*
* *
*

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?

The revised *BookMain.java* should add a field (`ArrayList` or `HashMap`) for storing all `Book` objects. If still using `ArrayList` fields, it may also have new helper methods such as `findBookByName` and `findKidByName`, which loop through the lists to search. All "handle" methods should be simpler except `handlePrintKidNames`, which was fine to begin with.

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?

The code may be hard to read/understand. It may also be difficult to add/modify features.

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?

It would be harder to know what is meant by a method like "add" or "length". And when using/importing a method from the old Math or String class, you would now load all of the other class's content too, more than you might actually need. It would also be harder to find what you're looking for in the Java API docs for the merged class.

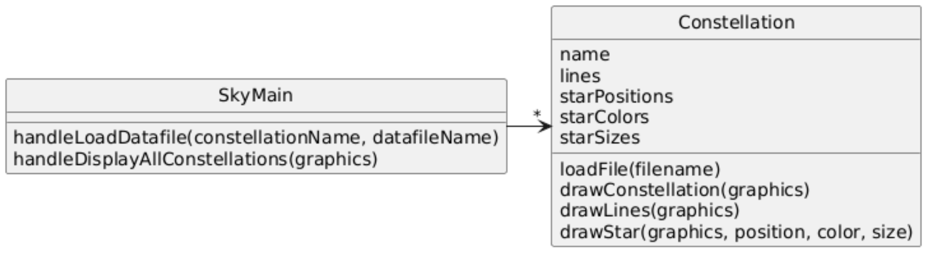
15. In Java, it is possible to put all variables and methods for a system in one big class. Why is this a bad idea?

Possible answers: It will be hard to find variables and methods we are looking for. Lots of unrelated things will be thrown together (e.g., similar-named variables/methods that would normally be in different classes), which can be confusing. The system may be less secure since all methods can access all fields, not just the ones they need to access.

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.

Possible answer: Jobs (and data) should be spread out among right-sized classes, each with one job.

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)

This design violates Design Principle 3a, because the `Constellation` class does everything except maybe some input parsing handled in `SkyMain`.

18. How could we improve this Constellation design?

Create a `Star` class that stores position, color, and size, and has a `drawStar` method. See revised PlantUML. This is an example of a general pattern: Often, we will need to ***extract*** a new class when one gets too complex. (Here, `Star` was extracted from `Constellation`.)

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.

Possible answers: Iceberg, where the tip is the public methods of a class and the underwater portion is the implementation details. A sports team with specialized roles/positions, where each player is like a class with a single job. Players work together, but they don't need to think about every little detail that their teammates are thinking about to perform their roles.

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? `PizzaMain, Order, Pizza`

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

`Pizza`, because it has the needed data. And `Order` already has its own job.

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

Solution A is better, because it distributes functionality by giving `Pizza` a meaningful `getCost()` method. In Solution B, `Pizza` is just a "dumb" or "inert" data holder. (Aside: Moving methods like this from the ***has-a*** class to the ***has'ed*** class is sometimes called ***pushing functionality down***.)

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.

See sample Claude.AI conversation.

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.)

Answers will vary. You should note any fields or methods that the genAI tool added, changed, or removed relative to the provided UML.

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

Answers will vary.

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

Answers will vary.

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

It depends on whether there are other system requirements (perhaps coming soon). Based on the current system description, these classes would just be dumb data holders. But if we anticipate other reqs such as a customer rewards program or varied topping prices, we might want to preemptively add these classes.