Week 3 Lab: Loops & Methods

1 Objectives

1. Practice solving problems with loops.
2. Practice modularizing code by implementing methods.

2 POGIL Programming

Switch roles this week. **Make a note of each person's name, role, and email.**

As a reminder the roles are:

 **Facilitator:** reads the questions aloud, keeps track of time and makes sure everyone contributes appropriately.

 **Spokesperson:** talks to the instructor and other teams. Compiles and runs programs when applicable.

 **Quality Control:** records all answers & questions, and provides team reflection to team & instructor.

 **Process Analyst:** considers how the team could work and learn more effectively.

You must swap roles every week. By the end of the course, you must have performed each role at least three times. We will provide prompts indicating each individual tasks in the lab assignments.

Note: **Everyone Codes!** You can divide up the work, but each member of the team must code some part of the overall lab solution. Remember, coding is a social activity. The Facilitator ensures that all questions get answered and everyone is able to contribute ideas toward the completion of the assignment. Sharing of ideas and code is encouraged among teammates within the lab. Pull the best code ideas together for your final submission.

Overview and Rationale

In this lab, you will implement arithmetic operators by *reduction to simpler repeated operations*. Specifically, you will:

1. Implement multiplication as repeated addition.
2. Implement exponentiation as repeated multiplication (using your own multiply method).
3. Implement an integer logarithm by repeated exponentiation.
4. Produce tabular output that reports powers and logarithms for a given base.

This lab emphasizes algorithmic decomposition, abstraction barriers, and the disciplined use of loops and method composition.

Learning Outcomes

After completing this lab, you will be able to:

- Decompose a problem into layered abstractions (addition → multiplication → exponentiation → logarithm).
- Implement and test iterative algorithms with `for` and `while` loops.
- Design methods with clear contracts and reuse them to build more complex behavior.
- Produce formatted console output with `System.out.printf`.
- Reason about integer types, ranges, and overflow when doing arithmetic in Java.

Constraints

- Do not use arithmetic shortcuts (e.g., no direct use of `*`, `/`, or `%` in core logic). Use + to accumulate repeated addition in `multiply`. (You may use * in argument progression in `printLogs` as specified below.)
- Do not use the Math library. Do not use `Math.pow`, `Math.log`, or related methods. Implement the logic via loops and the methods you create.
- Use iteration only (no recursion).
- Follow the specified method signatures exactly.
- Assume all inputs are non-negative and bases are ≥ 2 for the print methods. For this lab, handle the simplest well-defined cases and avoid negative numbers.

Tasks

1. Create the project and starter file.
    - Create `Lab3.java` and paste the Starter API skeleton.
    - Add a class-level documentation comment describing the lab and each method's contract.
2. Implement `multiply` using repeated addition.
    - Accumulate the result with each iteration.
    - Do not use `*`, `/`, `%` in the core logic of multiply.
3. Implement `power` using your `multiply` method.
    - Do not use the `Math` library. You may only use the `multiply` method developed in this lab.
4. Implement `log` using repeated calls to `power`.

- Do not use the `Math` library. You may only use the `power` method developed in this lab.

5. Implement `printPowers` to display a table of powers.
   - Use `System.out.printf` to format your results.
   - Use `"%4s %8s %5s%n"` to print the header. (See sample results below.)
6. Implement `printLogs` to display a table of logarithms.
   - Use `System.out.printf` to format your results.
   - Use `"%4s %8s %8s%n"` to print the header. (See sample results below.)
7. In `main`, demonstrate your methods with:
   - `printPowers(2, 8);` and `printLogs(2, 256);`
   - `printPowers(3, 8);` and `printLogs(3, 6561);`

Sample Console Output

Your output must match this spacing:

```
BASE EXPONENT POWER
   2        0     1
   2        1     2
   2        2     4
   2        3     8
   2        4    16
   2        5    32
   2        6    64
   2        7   128
   2        8   256


BASE ARGUMENT EXPONENT
   2        1        0
   2        2        1
   2        4        2
   2        8        3
   2       16        4
   2       32        5
   2       64        6
   2      128        7
   2      256        8
```

Starter API

Create a file named `Lab3.java` with the following structure:

```
public class Lab3 {
    public long multiply( long multiplicand, long multiplier ) {
        /* TODO */
    }
    public long power( int base, int exponent ) {
        /* TODO */
    }
    public long log( int base, int argument ) {
        /* TODO */
    }

    public void printPowers( int base, int maxExponent ) {
        /* TODO */
    }
    public void printLogs( int base, int maxArgument ) {
        /* TODO */
    }

    public static void main(String[] args) {
        /* TODO */
    }
}
```

Note: Return types are `long` to provide a larger numeric range for results while still accepting `int` inputs for convenience. You must preserve these signatures.

Testing Guidance

You must create test methods for the *multiply*, *power*, and *log* methods. Check edge cases such as:

- `multiply(0, k) == 0`.
- `power(b, 0) == 1`.
- `log(b, 1) == 0`.

Critiquing

Criterion	Good	Bad	Ugly
Multiply	Correct, uses loops	Minor errors	Incorrect
Power	Correct, uses multiply	Minor errors	Incorrect
Log	Correct ceiling behavior	Boundary issues	Incorrect
Output	Correct format	Spacing	Missing/incorrect
Design	Clean composition	Some duplication	Poor structure
Documentation	Javadoc present	Inconsistent	None
Testing	Good evidence	Minimal	None

Common Pitfalls and How to Avoid Them

1. **Using forbidden operations**: Do not use `*` in multiply or the `Math` library anywhere. The grader will inspect your code.
2. **Wrong loop bounds**: Avoid *Off-By-One* mistakes.
3. **Type mismatch/overflow**: Use `long` for result. Be aware that even `long` can overflow for large exponents. Keep test ranges modest.
4. **Printing vs. computing**: Keep computation in methods; `printPowers` and `printLogs` only format results calculated using methods developed in this lab.
5. **Log definition confusion**: This lab's log returns the smallest integer e with b^e $\geq$ argument. Do not try to compute floating-point logs.

Reflection

1. What is a *loop invariant*?
2. Explain how the loop invariant in `power` justifies correctness.
3. Describe a situation where a *for-loop* would be more appropriate than a *while-loop*, and vice versa. Can you provide a code snippet for each scenario?
4. Explain how using methods can make a program easier to understand and maintain.
5. Describe what a method signature is in Java. How does it help in method overloading?
6. Consider a problem using nested loops. Explain how the control variables in the outer and inner loops interact with each other. Could you change the order of these loops without affecting the results? Why or why not?

3 What to Submit

Submit your source file(s) (.java) and your answers to the reflection questions (pdf).