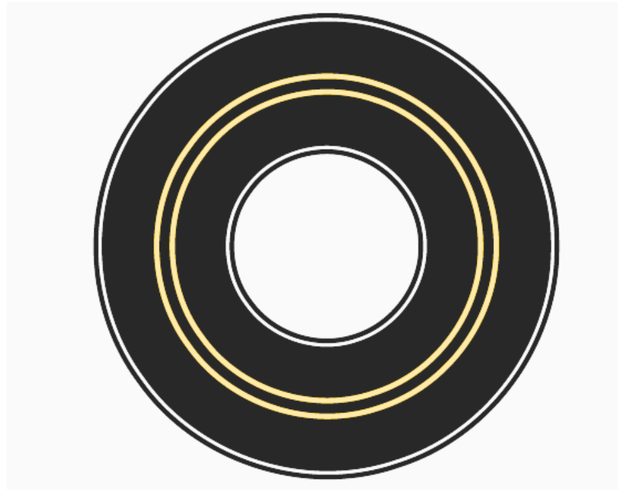
Session 4: Mastering Loops in Programming

Unlock the power of repetition and automation in your code with essential loop structures that every programmer must master.

Before Loops

Loop



Control Statements + Loops

Possibility

Why Loops Matter

Automation

Loops eliminate manual repetition, automating tasks that would otherwise require hundreds of lines of duplicate code.

Efficiency

Write once, execute many times. Loops make your programs faster to write and easier to maintain.

Error Reduction

Repetitive code invites typos and inconsistencies. Loops ensure consistent execution every time.

Today, we'll explore three essential loop types—**for**, **while**, and **do-while**—with hands-on examples that bring these concepts to life.

The For Loop: When You Know the Count

01

Initialization

Set your starting point with a counter variable

02

Condition

Define when the loop should continue running

03

Update

Increment or modify the counter after each iteration

04

Execute

Run your code block for each valid iteration

Syntax: `for (initialization; condition; update) { /* code */ }`

```
int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

The for loop excels when you know exactly how many times to repeat an operation—perfect for counting, iterating through arrays, or processing fixed datasets.

For loop

Making sandwiches for a family: To prepare lunch for the five members of your family, you repeat the steps of making a sandwich a total of five times, once for each person.

The While Loop: Condition-Controlled Repetition

The while loop checks its condition *before* each iteration, making it ideal when the number of repetitions depends on dynamic conditions or user input.

Syntax: `while (condition) { /* code */ }`

```
int main() {
    int i = 1;
    while (i <= 5) {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

Use while loops when you're waiting for a specific state change, processing data until a condition is met, or handling situations where the loop might not run at all.

While loop

Maa ne kaha,

“Jab tak doodh ubal na jaaye,

gas band nahi karni.”

The Do-While Loop: Guaranteed First Run

Execute First

The loop body runs once before any condition check

Then Evaluate

Condition checked after execution—loop continues if true

At Least Once

Guaranteed minimum of one execution regardless of condition

Syntax: `do { /* code */ } while (condition);`

```
int main() {
    int input;
    do {
        printf("Enter a number (0 to exit):\n");
        scanf("%d", &input);
    } while (input != 0);

    return 0;
}
```

Perfect for menu systems, input validation, and any scenario requiring at least one execution before checking continuation conditions.

Do-While loop

Papa bolte hain,

“Jaake main gate ek baar check karke aa.”

Chahe tumhe lagta ho gate band hai,

jaana to padega hi — papa ka rule.

Key Differences at a Glance

For Loop

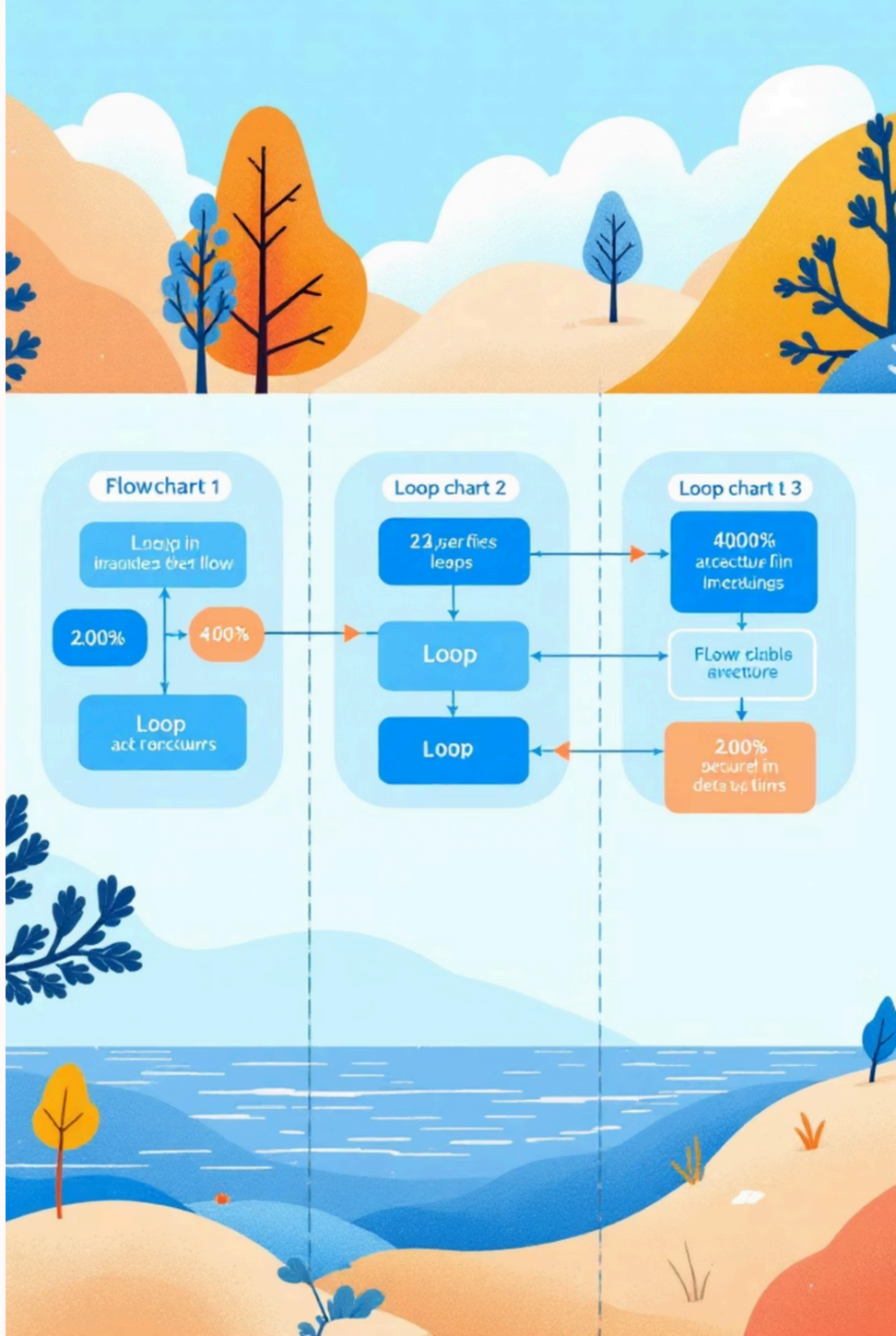
- Best for known iteration counts
- Compact syntax with initialization, condition, and update
- Counter-based repetition

While Loop

- Condition checked **before** execution
- May skip entirely if condition initially false
- Ideal for unknown iteration counts

Do-While Loop

- Condition checked **after** execution
- Guaranteed to run at least once
- Perfect for validation scenarios



Activity 1: Print a Pattern Using Loops

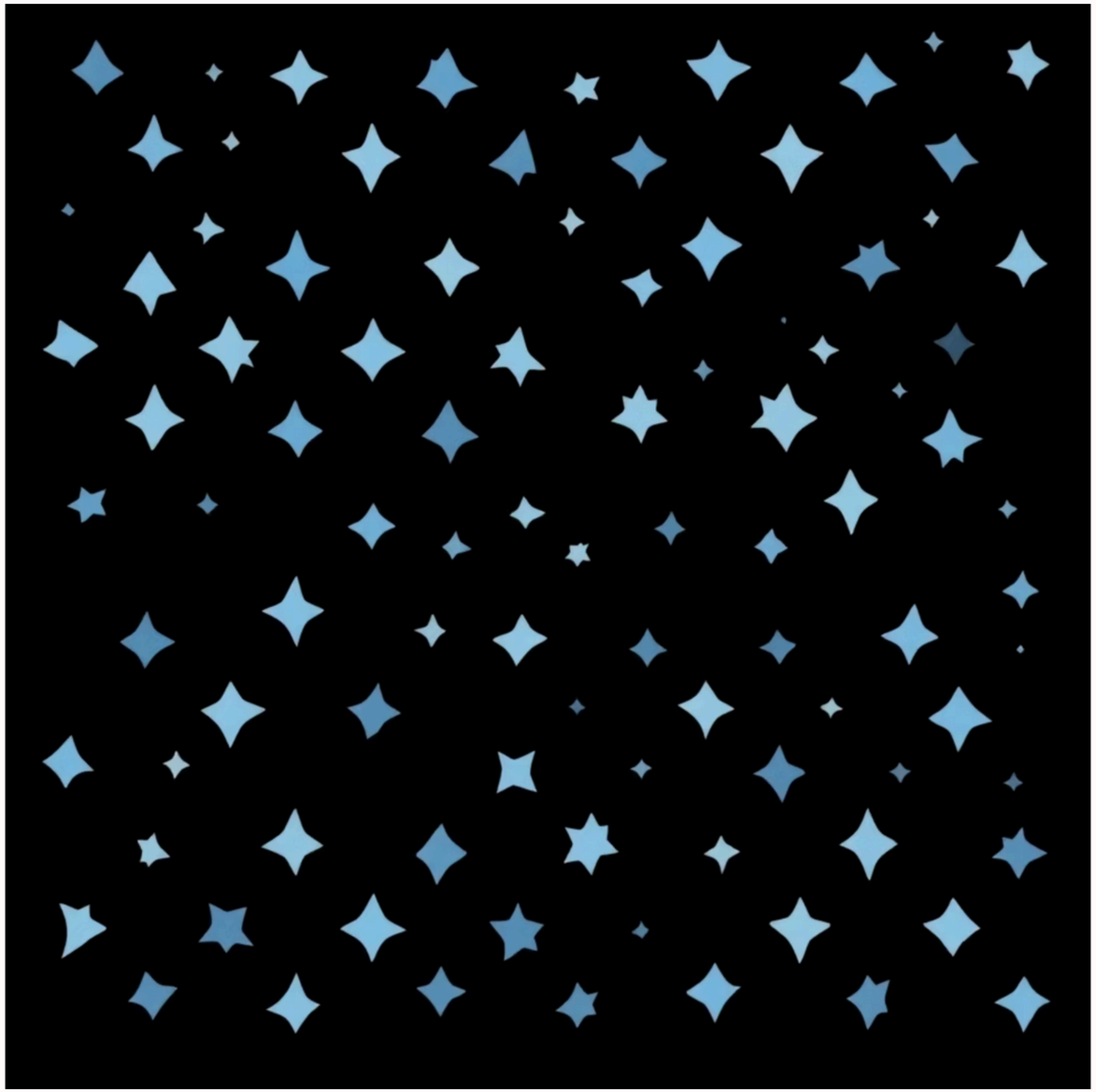
Your Challenge

Use a **for loop** to print a triangle pattern of stars. Each row should have one more star than the previous row.

Expected Output:

```
*
**
***
****
*****
```

Bonus Challenge: Can you recreate this pattern using while and do-while loops? Each approach teaches you different ways to think about iteration!



Hint

Activity 2: Calculate the Sum of Numbers

1

Basic Task

Write a loop to sum all numbers from 1 to N, where N is provided by the user.

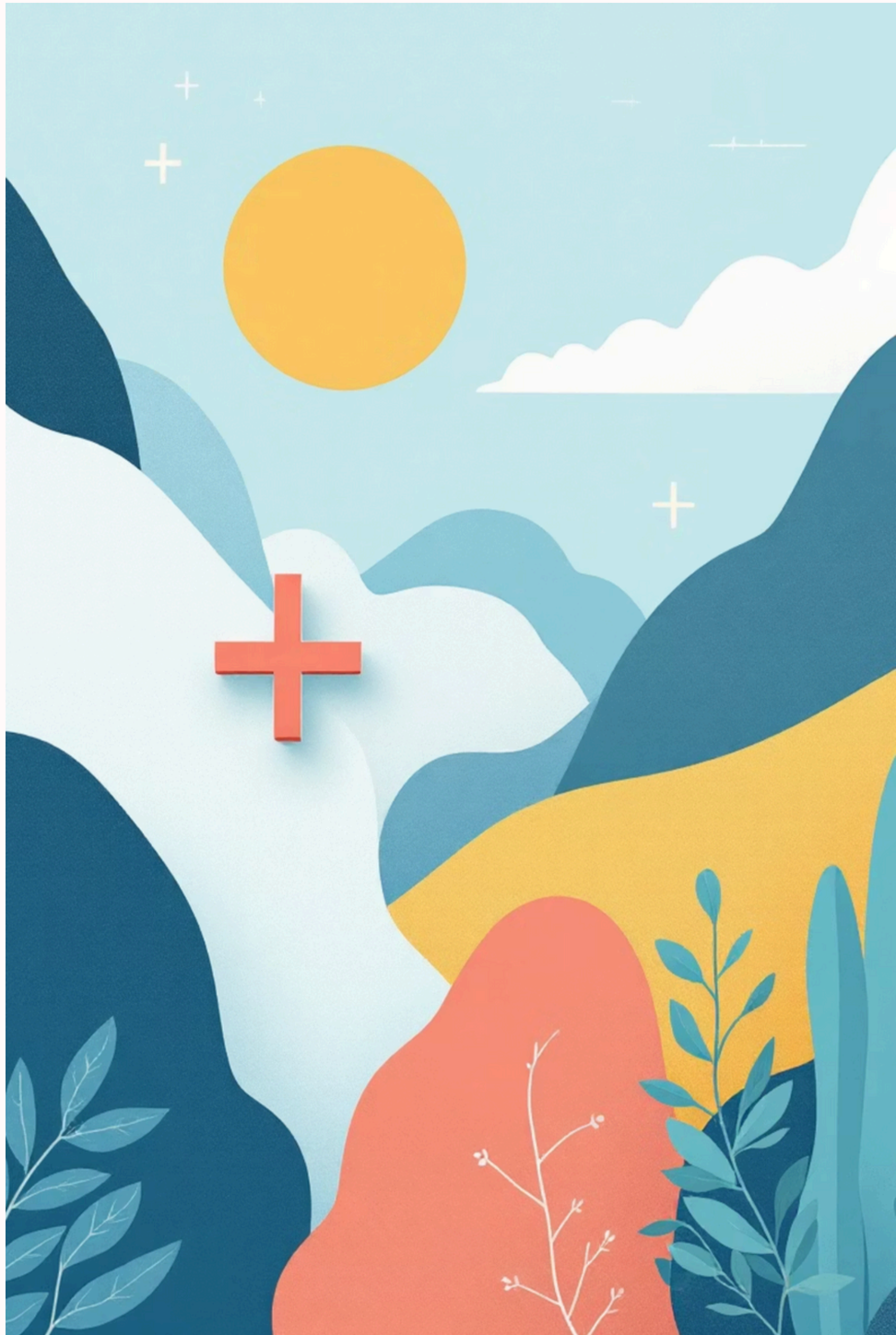
Compare Approaches

Implement the solution using both a for loop and a while loop—notice how each feels different.

Advanced Challenge

Use a do-while loop to continuously accept numbers from the user until they enter 0, then display the total sum.

Learning Tip: This activity demonstrates how the same problem can be solved with different loop types. Understanding when to use each is key to writing elegant code.



Common Pitfalls & Best Practices

Infinite Loops

Always ensure your loop variable updates or condition eventually becomes false. An infinite loop will freeze your program!

```
// BAD: i never changes
while (i < 10) {
    print(i);
}
```

Off-By-One Errors

Check your loop boundaries carefully. Does your loop need i <= 10 or i < 10? This classic mistake causes bugs.

```
// Runs 10 times (0-9)
for (i = 0; i < 10; i++)
```

Break & Continue

Use break to exit a loop early and continue to skip to the next iteration. Powerful tools when used sparingly.

Readability Matters

Keep loops simple and well-commented. Complex nested loops should be refactored into separate functions for clarity.

Summary & Next Steps

Key Takeaways

Loops automate repetition and are fundamental to efficient programming. Master for, while, and do-while to handle any iteration scenario.

Choose Wisely

Each loop type serves distinct purposes—for for counting, while for conditions, do-while for guaranteed execution.

Practice Makes Perfect

Complete the pattern and sum activities to solidify your understanding through hands-on coding.

Coming Up Next

In our next session, we'll explore **nested loops** and advanced control techniques including loop optimization and performance considerations. Get ready to take your loop skills to the next level!

Keep experimenting with loops—they're the gateway to unlocking true programming efficiency and power. 🚀