Module 06 — Loops

ITP 120 — Java Programming I

Northern Virginia Community College | Randy Michak

Learning Objectives

- Explain what a loop does and why we need them
- Write `while`, `do-while`, and `for` loops in Java
- Distinguish pre-test loops from post-test loops
- Use `break` and `continue` to control loop flow
- Write nested loops and trace their execution step by step
- Recognize and fix common loop mistakes: off-by-one errors and infinite loops
- Apply common loop patterns: accumulator, counter, min/max, input validation
- Read and write an enhanced `for` loop (for-each) with arrays

Key Terms

loop — a block of code that repeats | **iteration** — one pass through a loop body | **pre-test loop** — condition checked before the body runs | **post-test loop** — condition checked after the body runs | **sentinel value** — a special input that signals "stop looping" | **infinite loop** — a loop whose condition never becomes false | **accumulator** — a variable that collects a running total | **nested loop** — a loop inside another loop | **break** — exits a loop immediately | **continue** — skips the rest of the current iteration | **off-by-one error** — looping one too many or one too few times | **enhanced for loop** — a simplified loop that iterates over a collection

1. What Are Loops?

A loop repeats a block of code. Instead of writing the same lines over and over, you write them once and tell Java how many times — or under what conditions — to keep running them.

Here is the difference without and with a loop. Imagine printing "Hello!" five times:

```
// Without a loop — tedious, doesn't scale
System.out.println("Hello!");
System.out.println("Hello!");
System.out.println("Hello!");
System.out.println("Hello!");
System.out.println("Hello!");
```

```
// With a loop — one statement, any number of times
for (int i = 0; i < 5; i++) {
    System.out.println("Hello!");
}
```

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

Want to print it a thousand times? Change one number. The code stays the same size.

Why Loops Matter in Real Programs

- **Processing lists** — read every item in a shopping cart, every student in a grade book
- **Input validation** — keep prompting until the user enters something valid
- **Reading files** — process each line until you reach the end
- **Calculations** — sum values, count occurrences, find a minimum or maximum
- **Retrying operations** — try a network request again if it fails

Java's Three Loop Statements

Loop	When Condition Is Checked	Best Used When...
`while`	Before each iteration (pre-test)	You don't know in advance how many times to loop
`do-while`	After each iteration (post-test)	The body must run at least once (e.g., input validation)
`for`	Before each iteration (pre-test)	You know exactly how many iterations you need

Think of It This Way

A loop is like hitting snooze on your alarm. The condition is "Am I actually awake yet?" If no, sleep nine more minutes and check again. You keep repeating that cycle until the condition is true — then you stop looping and get up. The alarm checks the condition before it lets you sleep (`while`), or it rings once and then decides whether to ring again (`do-while`).

2. The `while` Loop

The `while` loop is the fundamental loop in Java. It checks a condition before each iteration. If the condition is true, the body runs. Then it checks again. This continues until the condition is false.

Syntax

```
while (condition) {
    // body — runs as long as condition is true
}
```

Because the condition is checked *before* the body runs, a `while` loop is called a **pre-test loop**. If the condition is false on the very first check, the body never executes at all.

A Counter-Controlled while Loop

When you use a counter variable to control a `while` loop, you need four things: initialize, check, do work, update.

```
int count = 1;               // 1. Initialize before the loop

while (count <= 5) {         // 2. Check the condition
    System.out.println(count); // 3. Do the work
    count++;                   // 4. Update — or you get an infinite loop!
}
```

```
1
2
3
4
5
```

Trace it mentally: count starts at 1, prints 1, becomes 2. Prints 2, becomes 3. ... Prints 5, becomes 6. Condition `6 <= 5` is false — loop ends.

A Sentinel-Controlled while Loop

When you don't know in advance how many times to loop, use a **sentinel value** — a special input that signals "I'm done." The loop keeps running until it sees that value.

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
System.out.print("Enter a number (-1 to quit): ");
int number = sc.nextInt();

while (number != -1) {                    // -1 is the sentinel
    System.out.println("You entered: " + number);
    System.out.print("Enter a number (-1 to quit): ");
    number = sc.nextInt();
}
System.out.println("Goodbye!");
```

```
Enter a number (-1 to quit): 42
You entered: 42
Enter a number (-1 to quit): 7
You entered: 7
Enter a number (-1 to quit): -1
Goodbye!
```

Notice that the prompt and read appear *twice*: once before the loop (to get the first value) and once at the end of the loop body (to get the next value). This is the standard pattern for sentinel loops.

Infinite Loop Warning

If the loop condition never becomes `false`, the program runs forever. The most common cause is forgetting to update the variable the condition depends on:

```
int x = 1;
while (x < 10) {
    System.out.println(x);
    // BUG: x never changes, so x < 10 is always true — infinite loop!
}
```

In IntelliJ, click the red square in the Run window to stop the program. Then find your missing update statement.

Try It

Write a `while` loop that counts down from 10 to 1, printing each number on its own line. After the loop ends, print "Blastoff!"

3. The `do-while` Loop

The `do-while` loop is almost identical to `while`, with one important difference: it runs the body *first*, then checks the condition. That makes it a **post-test loop** — the body always executes at least once.

Syntax

```
do {
    // body — runs at least once, guaranteed
} while (condition);   // <-- note the semicolon here!
```

The semicolon after the closing parenthesis is required. Java will give you a compile error if you leave it out.

Classic Use Case: Input Validation

You need to show the user a prompt at least once before you can check what they typed. `do-while` handles this perfectly:

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
int age;

do {
    System.out.print("Enter your age (0-120): ");
    age = sc.nextInt();
} while (age < 0 || age > 120);   // keep asking if invalid

System.out.println("Age recorded: " + age);
```

```
Enter your age (0-120): -5
Enter your age (0-120): 999
Enter your age (0-120): 22
Age recorded: 22
```

With a regular `while` loop, you would have to read the first value before the loop and read again inside the loop — duplicating the prompt and read statements. `do-while` keeps it in one place.

do-while vs. while: Side by Side

Feature	`while`	`do-while`
Loop type	Pre-test	Post-test
Condition check	Before each iteration	After each iteration
Minimum iterations	0 — may never run	1 — always runs once
Semicolon after condition?	No	Yes, required
Best for	General looping	Input validation, menus

When to Reach for do-while

Ask yourself: "Does the body have to run at least once before I can even check the condition?" If yes, `do-while` is your loop. Showing a menu is the textbook example — you must display the options before you can tell the user to pick one, and then validate their pick.

Try It

Write a `do-while` loop that displays three menu options (1, 2, 3) and keeps asking until the user enters a valid choice. Print a confirmation once a valid choice is made.

4. The `for` Loop

When you know exactly how many times to loop, the `for` loop is the cleanest choice. It bundles the initialization, condition check, and update all into one header line.

Syntax

```
for (initialization; condition; update) {
    // body
}
```

- **Initialization** — runs exactly once, before the loop starts. Usually declares and sets a counter variable.
- **Condition** — checked before every iteration. Loop runs while it is `true`.
- **Update** — runs after every iteration. Usually increments or decrements the counter.

Execution Order

Step	What Happens	Frequency
1	Initialization runs	Once, at the very start
2	Condition is evaluated	Before every iteration
3	Body executes	Each time condition is true
4	Update runs	After each body execution
5	Go back to step 2	Until condition is false

Counting Up

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

```
1
2
3
4
5
```

Counting Down

```
for (int i = 5; i >= 1; i--) {
    System.out.println(i);
}
System.out.println("Done!");
```

```
5
4
3
2
1
Done!
```

Counting by 2s

```
for (int i = 0; i <= 10; i += 2) {
    System.out.print(i + " ");
}
System.out.println();
```

```
0 2 4 6 8 10
```

When to Use for vs. while

Use `for` when the number of iterations is known before the loop starts. Use `while` when the loop should continue based on something you discover while running. In practice: counting, iterating over arrays → `for`; reading input until valid, processing until a sentinel → `while`.

Don't Modify the Loop Variable Inside the Body

The `for` header's update expression already handles the counter. Changing it inside the body too causes hard-to-find bugs:

```
// BAD — skips every other number
for (int i = 0; i < 10; i++) {
    System.out.println(i);
    i++;   // oops — double increment, skips 1, 3, 5, 7, 9
}
```

Let the header manage the counter. Keep your logic in the body.

Try It

- Print the numbers 1 through 20 on one line, separated by spaces.
- Print every odd number from 1 to 19. (Hint: start at 1, step by 2.)
- Print the 7 times table: "7 × 1 = 7", "7 × 2 = 14", ..., "7 × 10 = 70".

5. Loop Control: `break` and `continue`

Sometimes you need to exit a loop before the condition naturally becomes false, or skip the rest of one iteration and jump straight to the next. `break` and `continue` handle those cases.

break — Exit the Loop Immediately

When Java hits `break`, it exits the loop entirely. Execution jumps to the first statement after the closing brace.

```
for (int i = 1; i <= 10; i++) {
    if (i == 7) {
        System.out.println("Found 7! Stopping early.");
        break;
    }
    System.out.println(i);
}
System.out.println("After the loop.");
```

```
1
2
3
4
5
6
Found 7! Stopping early.
After the loop.
```

A common use: searching an array for a value. Once you find it, `break` so you don't waste time checking the rest.

continue — Skip to the Next Iteration

`continue` skips the remaining statements in the current iteration. In a `for` loop, it jumps to the update step. In a `while` loop, it jumps back to the condition check.

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;     // skip the rest of this iteration for even numbers
    }
    System.out.print(i + " ");
}
System.out.println();
```

```
1 3 5 7 9
```

Use Both Sparingly

`break` and `continue` are legitimate tools, but overusing them makes loops hard to read and reason about. If you're reaching for them frequently, your condition logic probably needs a rewrite instead.

Keyword	Effect	When It Helps
`break`	Exits the loop entirely	Early exit after finding a result; error condition
`continue`	Skips rest of current iteration	Skip invalid or irrelevant values without nesting

> **break Only Exits the Innermost Loop**
>
> In nested loops, `break` exits the loop it is directly inside — not all loops at once. If you need to break out of two levels, you'll need a flag variable or labeled breaks (an advanced feature).

6. Nested Loops

A nested loop is a loop inside another loop. The outer loop drives the rows; the inner loop drives the columns. Every time the outer loop runs once, the inner loop runs all the way through.

How Iteration Counts Multiply

Outer runs 3 times × inner runs 4 times = 12 total executions of the inner body. This multiplying effect matters a lot when you're dealing with large data sets — but for classroom-sized examples it's perfectly fine.

```
for (int outer = 1; outer <= 3; outer++) {
    for (int inner = 1; inner <= 4; inner++) {
        System.out.println("outer=" + outer + ", inner=" + inner);
    }
}
```

```
outer=1, inner=1
outer=1, inner=2
outer=1, inner=3
outer=1, inner=4
outer=2, inner=1
outer=2, inner=2
outer=2, inner=3
outer=2, inner=4
outer=3, inner=1
outer=3, inner=2
outer=3, inner=3
outer=3, inner=4
```

Example: Multiplication Table

```
for (int row = 1; row <= 5; row++) {
    for (int col = 1; col <= 5; col++) {
        System.out.printf("%4d", row * col);
    }
    System.out.println();   // move to next row after each row is complete
}
```

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

Example: Triangle of Stars

The inner loop runs a different number of times on each row — exactly `row` times — because the inner loop's limit is the outer loop variable:

```
for (int row = 1; row <= 5; row++) {
    for (int star = 1; star <= row; star++) {
        System.out.print("*");
    }
    System.out.println();
}
```

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

How to Trace a Nested Loop

Fix the outer variable at its first value. Walk through all inner values. Write down what prints. Then advance the outer variable and repeat. A small table with columns for each loop variable is the fastest way to do this on an exam.

Try It

Write nested loops to print a 4×4 grid where each cell shows the product of its row and column (both starting at 1). Row 2, col 3 should show 6.

7. Common Loop Patterns

These patterns appear constantly in real programs. Learn to recognize and write them without thinking.

Running Total (Accumulator)

Initialize a variable to zero before the loop. Add each new value to it inside the loop.

```
int sum = 0;   // accumulator — starts at 0

for (int i = 1; i <= 10; i++) {
    sum += i;
}

System.out.println("Sum 1 to 10: " + sum);
```

```
Sum 1 to 10: 55
```

Counting Occurrences

Same idea as an accumulator, but you only add 1 when a specific condition is met.

```
int evenCount = 0;

for (int i = 1; i <= 20; i++) {
    if (i % 2 == 0) {
        evenCount++;
    }
}

System.out.println("Even numbers from 1-20: " + evenCount);
```

```
Even numbers from 1-20: 10
```

Finding the Maximum

Initialize max to the first value (or the smallest possible integer). Update it whenever you find something bigger.

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
System.out.print("How many numbers? ");
int n = sc.nextInt();

System.out.print("Enter number 1: ");
int max = sc.nextInt();   // seed max with the first value

for (int i = 2; i <= n; i++) {
    System.out.print("Enter number " + i + ": ");
    int val = sc.nextInt();
    if (val > max) {
        max = val;
    }
}

System.out.println("Maximum: " + max);
```

```
How many numbers? 4
Enter number 1: 12
Enter number 2: 47
Enter number 3: 8
Enter number 4: 31
Maximum: 47
```

Input Validation Loop

Keep asking until the user provides acceptable input. This is the `do-while` pattern from Section 3, but worth calling out explicitly as its own pattern:

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
int choice;

do {
    System.out.println("Choose an option:");
    System.out.println("  1. Start");
    System.out.println("  2. Settings");
    System.out.println("  3. Quit");
    System.out.print("Enter 1-3: ");
    choice = sc.nextInt();
} while (choice < 1 || choice > 3);

System.out.println("You chose: " + choice);
```

```
Choose an option:
1. Start
2. Settings
3. Quit
Enter 1-3: 9
Choose an option:
1. Start
2. Settings
3. Quit
Enter 1-3: 2
You chose: 2
```

8. Common Loop Mistakes

These are the bugs that show up on nearly every intro Java exam. Know what they look like.

Off-by-One Error

Looping one too many or one too few times. Usually a wrong comparison operator (`<` vs. `<=`).

```
// Prints 1 through 4 — MISSING 5
for (int i = 1; i < 5; i++) {
    System.out.print(i + " ");
}
// Output: 1 2 3 4

// Prints 1 through 5 — CORRECT
for (int i = 1; i <= 5; i++) {
    System.out.print(i + " ");
}
// Output: 1 2 3 4 5
```

Always ask: "Should the loop run when `i` equals the boundary value?" If yes, use `<=`. If no, use `<`.

Infinite Loop (Forgot to Update Counter)

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    // BUG: i never changes — loops forever!
}

// Fix: add i++ inside the body
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;   // now the condition eventually becomes false
}
```

Wrong Comparison Operator

```
// BUG: condition is false immediately — loop never runs
for (int i = 10; i < 1; i++) {
    System.out.println(i);
}

// Fix: count down if starting high
for (int i = 10; i >= 1; i--) {
    System.out.println(i);
}
```

Modifying the Loop Variable Inside a for Loop Body

```
// BUG: i is incremented by the header AND by the body
// Skips 1, 3, 5, 7, 9 — prints only 0, 2, 4, 6, 8
for (int i = 0; i < 10; i++) {
    System.out.println(i);
    i++;   // accidental double increment
}
```

Quick Checklist: Loop Mistakes to Avoid

- Did you initialize the counter before the loop?
- Is your comparison operator (<, <=, >, >=) exactly right?
- Does the loop body actually update whatever the condition checks?
- Are you accidentally modifying the loop variable twice?
- Did you test with the boundary values (first and last iteration)?

9. The Enhanced for Loop (for-each)

Java has a fourth loop syntax designed specifically for iterating over arrays and collections. It is called the **enhanced for loop**, or informally the **for-each loop**. You will use it extensively once we cover arrays in a later module — here's a quick preview.

Syntax

```
for (dataType variableName : arrayOrCollection) {
    // use variableName here
}
```

Read the colon as "in". So `for (int score : scores)` means "for each int named score in the scores array."

Example with an Array

```
int[] scores = {85, 92, 78, 96, 88};

for (int score : scores) {
    System.out.println("Score: " + score);
}
```

```
Score: 85
Score: 92
Score: 78
Score: 96
Score: 88
```

Accumulator with for-each

```
int[] scores = {85, 92, 78, 96, 88};
int total = 0;

for (int score : scores) {
    total += score;
}

double average = (double) total / scores.length;
System.out.printf("Average: %.1f%n", average);
```

```
Average: 87.8
```

When to Use for-each vs. Regular for

Situation	Use
Just reading every element in order	Enhanced for (cleaner)
Need the index value (`i`)	Regular for
Need to modify the array while iterating	Regular for
Iterating part of the array	Regular for

for-each Has Limits

The enhanced for loop is read-only from the array's perspective. If you do `score = 100` inside the loop, you're changing the local variable `score`, not the actual array element. Use a regular indexed `for` loop when you need to update array values.

Vocabulary Review

Term	Definition
loop	A control structure that repeats a block of code zero or more times
iteration	One single pass through a loop body
pre-test loop	A loop that checks its condition before executing the body (`while`, `for`)
post-test loop	A loop that executes the body first, then checks the condition (`do-while`)
sentinel value	A special input value that signals "stop looping"
infinite loop	A loop whose condition never becomes false; program runs forever

Term	Definition
accumulator	A variable that builds up a running total across iterations
nested loop	A loop placed inside another loop; inner body runs outer×inner total times
break	Statement that immediately exits the enclosing loop
continue	Statement that skips the rest of the current iteration and moves to the next
off-by-one error	A bug where a loop runs one too many or one too few times
enhanced for loop	The for-each syntax: `for (type item : collection)` — iterates over every element

Module Summary

- **while** — pre-test loop; use when the iteration count isn't known in advance.
- **do-while** — post-test loop; body runs at least once; great for input validation and menus.
- **for** — pre-test loop; packs init, condition, and update in one line; use when you know the count.
- **break** exits the loop; **continue** skips to the next iteration. Use both sparingly.
- **Nested loops** multiply iteration counts. Outer variable controls rows; inner controls columns.
- Common patterns: accumulator, occurrence counter, min/max finder, input validation.
- Common bugs: off-by-one (wrong < vs. <=), infinite loop (forgot to update), accidental double increment.
- **Enhanced for** (for-each) is clean for reading every element; use indexed `for` when you need the index or need to modify elements.

Module 06 Quiz

Name: _____ Date: _____ Score: ____ / 10

Question 1

What type of loop is the `while` loop?

A) Post-test loop

B) Mid-test loop

C) Pre-test loop

D) Enhanced loop

Question 2

What is the output of the following code?

```
int x = 10;
while (x > 7) {
    System.out.print(x + " ");
    x--;
}
```

A) 10 9 8 7

B) 10 9 8

C) 9 8 7

D) 7 8 9 10

Question 3

Which loop type guarantees the loop body executes at least once?

A) `for`

B) `while`

C) Enhanced `for`

D) `do-while`

Question 4

How many times does the body of the inner loop execute in total in this code?

```
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++) {
        System.out.print("*");
    }
}
```

A) 7

B) 9

C) 12

D) 16

Question 5

What does the `break` statement do inside a loop?

A) Restarts the loop from the beginning

B) Pauses the loop for one iteration

C) Skips the rest of the current iteration and continues with the next

D) Exits the loop entirely and continues after the closing brace

Question 6

What is the output of the following code?

```
for (int i = 1; i <= 6; i++) {
    if (i % 3 == 0) {
        continue;
    }
    System.out.print(i + " ");
}
```

A) 1 2 4 5

B) 3 6

C) 1 2 3 4 5 6

D) 1 2 4

Question 7

Which of the following correctly describes an **off-by-one error**?

A) A loop that runs one too many or one too few times due to a wrong boundary

B) Using the wrong variable name inside the loop body

C) Forgetting to declare the loop variable before the loop

D) Using `break` when `continue` was intended

Question 8

What is the output of the following code?

```
int total = 0;
for (int i = 1; i <= 4; i++) {
    total += i;
}
System.out.println(total);
```

A) 4

B) 10

C) 16

D) 6

Question 9

In an enhanced `for` loop (for-each), what happens if you assign a new value to the loop variable inside the body?

A) The corresponding array element is updated

B) A `NullPointerException` is thrown at runtime

C) Only the local loop variable changes; the original array is not modified

D) The compiler produces an error

Question 10

A programmer writes the following loop. What is the most likely bug?

```
int count = 0;
while (count < 5) {
    System.out.println("Count: " + count);
}
```

A) The condition should use `<=` instead of `<`

B) The variable `count` is never incremented, causing an infinite loop

C) `println` should be `print`

D) The loop will only execute once because the condition is checked after the body

✅ Answer Key

Question	Answer	Explanation
1	**C**	A `for` loop has three parts: initialization, condition, and update.
2	**B**	A `while` loop checks its condition before each iteration.
3	**D**	A `do-while` loop always executes at least once before checking the condition.
4	**C**	An infinite loop runs forever because the condition never becomes false.
5	**D**	`break` exits the loop immediately, skipping remaining iterations.
6	**A**	`continue` skips the rest of the current iteration and moves to the next.
7	**A**	A sentinel value signals the end of input in a loop.
8	**B**	Nested loops: the inner loop completes all iterations for each iteration of the outer loop.
9	**C**	The loop counter variable controls how many times the loop runs.
10	**B**	A `for` loop is preferred when the number of iterations is known in advance.