PROGRAMMING IN JAVA

Lecture 4

Control Structures, Arrays, and Strings

Decision Structures, Loops, One- and Two-Dimensional Arrays, and String Processing

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

By the end of this lecture, students should be able to:

- construct decision-making logic using `if`, `if-else`, `else-if` chains, and nested conditions;
- use `switch` statements and modern `switch` expressions appropriately;
- select and apply `for`, `while`, and `do-while` loops;
- control loop execution using `break` and `continue`;
- design nested loops for tables, patterns, and matrix-style processing;
- declare, initialize, access, and traverse one-dimensional arrays;
- process arrays using indexed loops and enhanced `for` loops;
- work with two-dimensional arrays;
- create, compare, and process `String` values correctly;
- avoid common errors such as off-by-one indexing, infinite loops, and incorrect `String` comparison.

2. Program Flow and Control Structures

By default, Java executes statements sequentially from top to bottom. *Control structures* change that flow. They allow a program to make decisions, repeat operations, skip selected statements, and process groups of values systematically.

Category	Java constructs	Purpose
Selection	`if`, `if-else`, `else-if`, `switch`	Choose one path based on conditions
Iteration	`for`, `while`, `do-while`	Repeat statements
Transfer	`break`, `continue`	Alter normal loop/switch flow
Structured data	`arrays`, `String`	Store and process multiple values or text

> A good algorithm normally combines data, expressions, decisions, and repetition. This lecture connects all four elements.

3. Decision Making with `if`

The `if` statement executes a block only when its Boolean condition evaluates to `true`. The condition must have type `boolean`; Java does not treat nonzero integers as true.

Example 1. A simple `if` statement

```
1  public class SimpleIf {
2      public static void main(String[] args) {
3          int temperature = 31;
4
5          if (temperature > 30) {
6              System.out.println("It is a hot day.");
7          }
8
9          System.out.println("Program finished.");
10     }
11 }
```

Expected output

```
It is a hot day.
Program finished.
```

3.1 if-else

An `if-else` statement selects between two alternatives. Exactly one of the two blocks is executed.

Example 2. Even or odd number

```
public class EvenOdd {
    public static void main(String[] args) {
        int number = 17;

        if (number % 2 == 0) {
            System.out.println(number + " is even");
        } else {
            System.out.println(number + " is odd");
        }
    }
}
```

Expected output

```
17 is odd
```

3.2 `else-if` chains

Use an `else-if` chain when several mutually exclusive conditions must be checked in order. Java stops at the first true condition.

Example 3. Converting a numeric mark to a grade

```
public class GradeExample {
    public static void main(String[] args) {
        int score = 86;
        char grade;

        if (score >= 90) {
            grade = 'A';
        } else if (score >= 80) {
            grade = 'B';
        } else if (score >= 70) {
            grade = 'C';
        } else if (score >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }

        System.out.println("Grade = " + grade);
    }
}
```

Expected output

```
Grade = B
```

3.3 Nested `if` statements

An `if` statement can appear inside another `if` statement. Nesting is useful when a second decision is meaningful only after the first condition has been satisfied.

Example 4. Admission rule with a nested decision

```
public class NestedIf {
    public static void main(String[] args) {
        int examScore = 88;
        boolean documentsComplete = true;

        if (examScore >= 75) {
            if (documentsComplete) {
                System.out.println("Eligible for admission.");
            } else {
                System.out.println("Submit the missing documents.");
            }
        } else {
            System.out.println("Exam score is too low.");
        }
    }
}
```

Expected output

```
Eligible for admission.
```

> Prefer clear Boolean expressions and moderate nesting. Deeply nested conditions are difficult to read and are often a sign that the algorithm should be reorganized.

4. Multi-Way Selection with `switch`

A `switch` is convenient when one expression must be compared with several discrete alternatives. Traditional `switch` syntax uses `case` labels and usually requires `break` to prevent fall-through.

Example 5. Traditional `switch` statement

```
public class DayName {
    public static void main(String[] args) {
        int day = 3;
        String name;

        switch (day) {
            case 1:
                name = "Monday";
                break;
            case 2:
                name = "Tuesday";
                break;
            case 3:
                name = "Wednesday";
                break;
            case 4:
                name = "Thursday";
                break;
            case 5:
                name = "Friday";
                break;
            default:
                name = "Weekend or invalid value";
        }

        System.out.println(name);
    }
}
```

Expected output

```
Wednesday
```

Modern Java supports arrow-style `case` labels. This syntax does not fall through automatically and is often clearer.

Example 6. Modern `switch` expression

```java
public class SwitchExpression {
    public static void main(String[] args) {
        int day = 6;

        String type = switch (day) {
            case 1, 2, 3, 4, 5 -> "Weekday";
            case 6, 7 -> "Weekend";
            default -> "Invalid day";
        };

        System.out.println("Day type = " + type);
    }
}
```

Expected output

```
Day type = Weekend
```

Use `if` / `else-if` when...	Use `switch` when...
conditions involve ranges such as `score >= 80`	one expression is matched against fixed alternatives
conditions combine several variables	cases are discrete values
Boolean logic is complex	a menu or category selection is being implemented

5. Repetition with Loops

Loops repeat a statement or block. The correct loop depends on whether the number of repetitions is known in advance and whether the condition should be tested before or after the loop body.

Loop	Condition checked	Typical use
`for`	before each iteration	known or count-controlled repetitions
`while`	before each iteration	condition-controlled repetition
`do-while`	after each iteration	body must run at least once

5.1 The `for` loop

Example 7. Counting with `for`

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("i = " + i);
        }
    }
}
```

Expected output

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

A standard `for` loop contains initialization, a Boolean condition, and an update expression:

```
for (initialization; condition; update) {
    statements
}
```

5.2 The `while` loop

Example 8. Summing integers with `while`

```
public class WhileSum {
    public static void main(String[] args) {
        int n = 5;
        int i = 1;
        int sum = 0;

        while (i <= n) {
            sum += i;
            i++;
        }

        System.out.println("Sum = " + sum);
    }
}
```

Expected output

```
Sum = 15
```

5.3 The `do-while` loop

A `do-while` loop executes its body first and checks the condition afterward. Therefore it always runs at least once.

Example 9. `do-while` executes at least once

```
public class DoWhileExample {
    public static void main(String[] args) {
        int value = 10;

        do {
            System.out.println("Value = " + value);
            value++;
        } while (value < 10);
    }
}
```

Expected output

```
Value = 10
```

5.4 Choosing the correct loop

- Use `for` when a counter or a known number of iterations naturally controls the algorithm.
- Use `while` when repetition depends on a condition that may become false before the first iteration.
- Use `do-while` when the body must execute at least once, such as displaying a menu before checking whether the user wants to continue.

6. break and continue

The `break` statement terminates the nearest loop or traditional `switch`. The `continue` statement skips the remainder of the current loop iteration and moves to the next iteration.

Example 10. Using `break` in a search

```
public class BreakExample {
    public static void main(String[] args) {
        int target = 7;

        for (int i = 1; i <= 10; i++) {
            if (i == target) {
                System.out.println("Found " + target);
                break;
            }
        }
    }
}
```

Expected output

```
Found 7
```

Example 11. Using `continue`

```
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 7; i++) {
            if (i % 2 == 0) {
                continue;
            }
            System.out.print(i + " ");
        }
    }
}
```

Expected output

```
1 3 5 7
```

7. Nested Loops

A loop may contain another loop. The inner loop completes all of its iterations for each iteration of the outer loop. Nested loops are common in tables, grids, matrices, and pattern generation.

Example 12. A multiplication table fragment

```
public class NestedLoops {
    public static void main(String[] args) {
        for (int row = 1; row <= 3; row++) {
            for (int col = 1; col <= 4; col++) {
                System.out.printf("%4d", row * col);
            }
            System.out.println();
        }
    }
}
```

Expected output

```
   1   2   3   4
   2   4   6   8
   3   6   9  12
```

Example 13. Printing a simple pattern

```
public class StarPattern {
    public static void main(String[] args) {
        for (int row = 1; row <= 4; row++) {
            for (int col = 1; col <= row; col++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Expected output

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

8. One-Dimensional Arrays

An *array* stores a fixed number of values of the same type. Arrays are reference types. Each element is accessed by an integer *index*, and Java array indices start at 0.

> For an array with length `n`, valid indices are 0 through `n - 1`.

Operation	Syntax example
Declaration	`int[] numbers;`
Creation	`numbers = new int[5];`
Combined declaration/creation	`int[] numbers = new int[5];`
Initializer	`int[] numbers = {10, 20, 30};`
Access	`numbers[0]`
Length	`numbers.length`

Example 14. Creating and accessing an array

```
public class ArrayBasics {
    public static void main(String[] args) {
        int[] marks = {85, 90, 78, 92, 88};

        System.out.println("First mark = " + marks[0]);
        System.out.println("Last mark = " + marks[marks.length - 1]);
        System.out.println("Number of marks = " + marks.length);
    }
}
```

Expected output

```
First mark = 85
Last mark = 88
Number of marks = 5
```

8.1 Traversing arrays with an indexed `for` loop

Example 15. Array sum and average

```java
public class ArrayAverage {
    public static void main(String[] args) {
        int[] marks = {85, 90, 78, 92, 88};
        int sum = 0;

        for (int i = 0; i < marks.length; i++) {
            sum += marks[i];
        }

        double average = (double) sum / marks.length;
        System.out.println("Sum = " + sum);
        System.out.println("Average = " + average);
    }
}
```

Expected output

```
Sum = 433
Average = 86.6
```

8.2 Enhanced `for` loop

The enhanced `for` loop, often called the *for-each* loop, is convenient when every element should be read in sequence and the index itself is not needed.

Example 16. Finding the maximum value

```
public class ArrayMaximum {
    public static void main(String[] args) {
        int[] values = {12, 7, 35, 19, 24};
        int max = values[0];

        for (int value : values) {
            if (value > max) {
                max = value;
            }
        }

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

Expected output

```
Maximum = 35
```

> Use an indexed loop when you need the position, need to modify specific elements, or need to compare neighboring elements. Use for-each when you only need each value.

9. Basic Array Algorithms

Many programming tasks reduce to a few recurring array patterns: accumulation, counting, searching, selecting a minimum/maximum, and transforming elements.

Example 17. Linear search

```java
public class LinearSearch {
    public static void main(String[] args) {
        int[] data = {14, 22, 9, 31, 18};
        int target = 31;
        int position = -1;

        for (int i = 0; i < data.length; i++) {
            if (data[i] == target) {
                position = i;
                break;
            }
        }

        System.out.println("Index = " + position);
    }
}
```

Expected output

```
Index = 3
```

Example 18. Counting positive, negative, and zero values

```java
public class ArrayCount {
    public static void main(String[] args) {
        int[] values = {-4, 0, 7, 12, -3, 0, 9};
        int positive = 0;
        int negative = 0;
        int zero = 0;

        for (int value : values) {
            if (value > 0) {
                positive++;
            } else if (value < 0) {
                negative++;
            } else {
                zero++;
            }
        }

        System.out.println("Positive = " + positive);
        System.out.println("Negative = " + negative);
        System.out.println("Zero = " + zero);
    }
}
```

Expected output

```
Positive = 3
Negative = 2
Zero = 2
```

10. Two-Dimensional Arrays

A two-dimensional array can represent a table or matrix. It is technically an array whose elements are themselves arrays.

Example 19. Traversing a matrix

```
public class MatrixExample {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Expected output

```
1 2 3
4 5 6
7 8 9
```

Example 20. Sum of the main diagonal

```java
public class DiagonalSum {
    public static void main(String[] args) {
        int[][] matrix = {
            {2, 1, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int sum = 0;
        for (int i = 0; i < matrix.length; i++) {
            sum += matrix[i][i];
        }

        System.out.println("Diagonal sum = " + sum);
    }
}
```

Expected output

```
Diagonal sum = 16
```

11. Strings in Java

`String` is a reference type used to represent text. `String` objects are *immutable*: once a `String` object has been created, its character sequence cannot be changed. Operations that appear to modify a `String` actually produce a new `String` object.

Method	Purpose	Example
`length()`	number of characters	`text.length()`
`charAt(i)`	character at index `i`	`text.charAt(0)`
`substring(a,b)`	part of a string	`text.substring(0, 4)`
`toUpperCase()`	uppercase copy	`text.toUpperCase()`
`toLowerCase()`	lowercase copy	`text.toLowerCase()`
`contains(x)`	tests whether text contains `x`	`text.contains("Java")`
`equals(x)`	content equality	`text.equals(other)`

Example 21. Common `String` operations

```
public class StringOperations {
    public static void main(String[] args) {
        String course = "Programming in Java";

        System.out.println("Length = " + course.length());
        System.out.println("First character = " + course.charAt(0));
        System.out.println("Uppercase = " + course.toUpperCase());
        System.out.println("Contains Java = " + course.contains("Java"));
    }
}
```

Expected output

```
Length = 19

First character = P
Uppercase = PROGRAMMING IN JAVA

Contains Java = true
```

11.1 Comparing Strings: `equals()` versus `==`

For `String` content comparison, use `equals()`. The `==` operator compares whether two references point to the same object, not whether the character sequences are equal.

Example 22. Correct `String` comparison

```
public class StringComparison {
    public static void main(String[] args) {
        String a = new String("Java");
        String b = new String("Java");

        System.out.println("a == b: " + (a == b));
        System.out.println("a.equals(b): " + a.equals(b));
    }
}
```

Expected output

```
a == b: false
a.equals(b): true
```

11.2 Processing characters in a `String`

Example 23. Counting vowels

```java
public class VowelCount {
    public static void main(String[] args) {
        String text = "Programming in Java".toLowerCase();
        int count = 0;

        for (int i = 0; i < text.length(); i++) {
            char ch = text.charAt(i);
            if (ch == 'a' || ch == 'e' || ch == 'i' ||
                    ch == 'o' || ch == 'u') {
                count++;
            }
        }

        System.out.println("Vowels = " + count);
    }
}
```

Expected output

```
Vowels = 6
```

12. Comprehensive Example: Student Score Analysis

The following program combines conditions, loops, arrays, `String` values, accumulation, maximum search, and grade classification in one small application.

Example 24. Integrating the main lecture concepts

```java
public class StudentScoreAnalysis {
    public static void main(String[] args) {
        String student = "Aruzhan";
        int[] scores = {84, 91, 76, 88, 95};

        int sum = 0;
        int max = scores[0];

        for (int score : scores) {
            sum += score;
            if (score > max) {
```

Example 24 (continued)

```
                max = score;
            }
        }

        double average = (double) sum / scores.length;
        char grade;

        if (average >= 90) {
            grade = 'A';
        } else if (average >= 80) {
            grade = 'B';
        } else if (average >= 70) {
            grade = 'C';
        } else if (average >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }

        System.out.println("Student: " + student);
        System.out.println("Average: " + average);
        System.out.println("Highest score: " + max);
        System.out.println("Final grade: " + grade);
    }
}
```

Expected output

```
Student: Aruzhan
Average: 86.8
Highest score: 95
Final grade: B
```

13. Common Errors and Good Practices

Problem	Why it is wrong or risky	Better practice
`if (x = 5)`	assignment is not a boolean expression	use `x == 5` for comparison
`i <= array.length`	last index becomes `array.length` and is invalid	use `i < array.length`
`while` condition never changes	can create an infinite loop	ensure loop state progresses
missing `break` in traditional `switch`	may cause unintended fall-through	use `break` or arrow-style cases
`text1 == text2`	compares `String` references	use `text1.equals(text2)`
deeply nested `if` blocks	hard to read and maintain	simplify conditions or reorganize logic
magic numbers everywhere	meaning is unclear	use named constants when appropriate

> Boundary conditions deserve special attention. Many beginner errors occur at the first or last element of an array, at zero iterations, or exactly at a grading/range threshold.

14. A Practical Algorithm-Design Workflow

Before writing code, students should identify the data, the decision rules, the repetition pattern, and the required result. A simple workflow is:

1. State the input and expected output.
2. Identify variables and data types.
3. Write the decision conditions in plain language.
4. Determine whether repetition is required and select a suitable loop.
5. If several values are related, decide whether an array is appropriate.
6. Write and compile a small version first.
7. Test normal cases, boundary cases, and unusual cases.
8. Check the output and improve readability.

15. Lecture Summary

- `if`, `if-else`, `else-if`, and nested `if` statements implement conditional decisions.
- `switch` is useful for discrete alternatives; modern arrow-style `switch` reduces accidental fall-through.
- `for`, `while`, and `do-while` implement repetition with different control patterns.
- `break` exits a loop or traditional `switch`; `continue` skips the remainder of the current loop iteration.
- nested loops are useful for matrices, tables, and patterns.
- arrays store a fixed number of same-type values and use zero-based indexing.
- `array.length` gives the number of elements; the final valid index is `length - 1`.
- `String` is an immutable reference type; `equals()` should be used for content comparison.
- careful boundary testing is essential for correct control-flow and array code.

16. Review Questions

9. What type must an `if` condition have in Java?
10. What is the difference between `if-else` and an `else-if` chain?
11. When is `switch` preferable to a long `else-if` chain?
12. What is fall-through in a traditional `switch` statement?
13. What are the three control parts of a standard `for` loop?
14. What is the main difference between `while` and `do-while`?
15. What does `break` do inside a loop?
16. What does `continue` do inside a loop?
17. Why can a loop become infinite?
18. What is the first valid index of a Java array?
19. For an array of length `n`, what is the final valid index?
20. When is an enhanced `for` loop convenient?
21. How is a two-dimensional array represented in Java?
22. Why should `String` values normally be compared with `equals()` rather than `==`?
23. What does it mean that `String` is immutable?

17. Programming Exercises

24. Read or assign an integer and determine whether it is positive, negative, or zero.
25. Determine the largest of three integers using conditional statements.
26. Convert a numeric score into a letter grade.
27. Use `switch` to print the name of a month from a number 1–12.
28. Print integers from 1 to 100 and display only values divisible by 5.
29. Calculate $n!$ using a loop.
30. Determine whether an integer is prime.
31. Generate the first n Fibonacci numbers.
32. Create an array of ten integers and calculate its sum and average.
33. Find the minimum and maximum elements of an array.
34. Search an array for a target value and print its index or `-1`.
35. Count positive, negative, and zero elements in an integer array.
36. Reverse the order of an array.
37. Create a 3×3 matrix and calculate the sum of all elements.
38. Calculate the sum of the main diagonal of a square matrix.
39. Count vowels, consonants, digits, and spaces in a `String`.
40. Determine whether a `String` is a palindrome, ignoring letter case.
41. Count how many words are present in a simple sentence.
42. Create a menu-driven calculator using `switch` inside a loop.
43. Create a student-score analyzer that reports average, minimum, maximum, and grade.

18. Homework

Develop a console application called `CourseResultAnalyzer`. The program must use only concepts studied through Lecture 4.

- Store a student name in a `String`.
- Store at least six assessment scores in an `int` array.
- Use a loop to calculate the total and average.
- Find the minimum and maximum scores.
- Count how many scores are at least 50.
- Use an `if/else-if` structure to determine a final letter grade.
- Use a `switch` statement to print a short interpretation of the letter grade.
- Print all results in a clear formatted report.
- Test the program with at least three different score sets and record the outputs.

19. Key Terms

Term	Meaning
control structure	a language construct that changes normal sequential execution
condition	a Boolean expression used to make a decision
iteration	one execution of a loop body
infinite loop	a loop whose termination condition is never reached
`break`	terminates the nearest loop or traditional `switch`
`continue`	skips the rest of the current loop iteration
array	a fixed-size indexed collection of same-type elements
index	integer position used to access an array element
for-each loop	enhanced `for` loop that processes elements sequentially
`String`	an immutable Java reference type representing text
immutability	property of an object whose state cannot be changed after creation

> **Study recommendation:** Run every example, then change the data and predict the new output before executing it. Control flow becomes intuitive only through repeated tracing and testing.