INTRODUCTION TO JAVA PROGRAMMING

By

Ahn Nuzen

Acknowledgments

The author extends his genuine thanks to the dedicated faculty of the Grossmont College Computer Science Department for their invaluable support throughout the development of this ZBook.

Their thoughtful reviews, constructive feedback, and encouragement through many rounds of revision significantly strengthened the quality and clarity of this work.

While this ZBook was not directly funded by the Zero Textbook Cost (ZTC) Acceleration Grant, its existence is a direct result of the program's impact. The skills and insights gained from authoring several ZTC-funded books enabled the expeditious creation of this Java version. The author wishes to express his appreciation to the ZTC team at Grossmont College for their guidance, support, and commitment to accessible education.

Ahn Nuzen
Grossmont College, Fall 2025

Table of Contents

Module 1: Getting Started with Java.

Java is a robust, high-level, object-oriented programming language developed by Sun Microsystems in the mid-1990s. The primary objective behind Java's creation was to develop a language that could operate across a wide range of devices and platforms without requiring recompilation. This vision led to the formulation of Java's core philosophy: **"Write Once, Run Anywhere."**

This principle is made possible through the Java Virtual Machine (JVM), which allows Java programs to be compiled into bytecode and executed consistently across various operating systems. In this module, we will examine the origins, design goals, and foundational features of the Java programming language, setting the stage for more advanced topics in the modules that follow.

With its portability, strong security features, and comprehensive standard library, Java rapidly gained popularity. Today, Java is widely used in enterprise environments, Android app development, web applications, and large-scale distributed systems

Learning Objectives

By the end of this Module, learners will:

- Understand the origin and purpose of Java.
- Successfully install the Java Development Kit (JDK) and a modern IDE.
- Write and run their first Java program.
- Declare and use variables of common Java data types.

1.1 History of Java

Java was developed in the mid-1990s by James Gosling and his team at Sun Microsystems. Originally designed for interactive television, it soon evolved into a robust platform for general-purpose computing.

Key milestones:

- 1995: Java 1.0 was released with the promise "Write Once, Run Anywhere" (WORA).
- 2006: Java was open-sourced and managed by the OpenJDK project.

- 2010–Present: Oracle acquired Sun; Java now follows a regular six-month release cycle.

Why Java is still relevant today:

- Platform independence (runs on Windows, Mac, Linux)
- Strongly typed and object-oriented
- Widely used in Android development, web servers, finance, and education

1.2. Installing JDK & IDE

To write and run Java code, learners need the Java Development Kit JDK (not just the Java Runtime Environment JRE).

The next step is setting up the development environment. Follow the instructions provided in the resources guide to download the Java Development Kit (JDK) from the official Oracle website or an open-source alternative like OpenJDK. The JDK includes the Java compiler, runtime, and other essential tools.

There are several modern Integrated Development Environment (IDE) such as IntelliJ IDEA, Eclipse, or Visual Studio Code. The class covers how to configure the IDE to recognize the JDK, ensuring that the environment is ready for writing and running Java code. Learners practice navigating the IDE interface, creating new projects, and understanding basic features like code completion and error highlighting.

IDE (Integrated Development Environment)

Beginners are encouraged to use a modern IDE:

- VS Code with Java extensions – Beginner-friendly with autocomplete and real-time error feedback.
- Alternatives: Eclipse, NetBeans, and IntelliJ.

Setup Tasks:

1. Install JDK
2. Install VSCode
3. Create a “HelloWorld” project with a single Java file

For an example see Module 1 : sources c) How to set up Java in Visual Studio Code video

1.3. Hello World Program

The classic first program, with the environment ready, we can write the first Java program—the classic "Hello, World!" example. The Java Class HelloWord has the structure of a basic Java program, including the importance of the main method as the entry point for execution. By convention we capitalize the first letter of the class name, the significance of the public and static keywords, and how the System.out.println statement is used to display output on the console, will be discussed in greater detail in the following lessons.

First Java class, **H**elloWorld.java:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Explanation of components:

- `public class HelloWorld`: Declares a class.
- `public static void main(...)`: The entry point of any Java application.
- `System.out.println(...)`: Outputs text to the console.

Exercise: Modify the program to print your name and favorite quote.

Common beginner errors:

- Mismatched braces `{}` or parentheses `()`
- Misspelling `System.out.println`
- Saving the file with a wrong filename (must match the class name)
- Java is case sensitive so .Java is not recognize as a Java class but .java is.

1.4. Variables & Data Types

Java is a statically typed language, which means that every variable must be declared with a specific data type before it can be used. This approach helps catch errors early, as the compiler checks that the correct type of data is being assigned to each variable. By requiring explicit type declarations, Java ensures code is clear, predictable, and easier to maintain.

Java provides several built-in primitive data types that represent simple values. These types are the basic building blocks for storing data in Java programs. Below is an overview of the most commonly used primitive data types, along with examples and typical use cases:

1.4.1 Primitive Data Types:

Type	*Example*	*Use Case*
int	`int age = 18;`	Whole numbers
double	`double temp = 98.6;`	Decimal numbers
boolean	`boolean isOn = true;`	True/false logic
char	`char grade = 'A';`	Single characters
String	`name = "Hello"`	Class String

```
String name = "Alice";
System.out.println("Hello " + name);
```

1.4.2 Java Naming Convention for Data Types

In Java, the names of primitive data types—such as `int`, `double`, `char`, and `boolean`—are always written in lowercase. These are built-in types and do not represent objects or classes.

In contrast, when you use a class type—such as `String`, `Integer`, or any custom class—the name always starts with an uppercase letter. For example, `String` is a class, so its name is capitalized.

1.4.3 CamelCase

CamelCase is a naming convention widely used in Java and many other programming languages to make variable, method, and class names more readable by removing spaces and using capital letters to indicate word boundaries.

In Java specifically, there are two main types of CamelCase:

1.4.3.1 lowerCamelCase (camelCase):

- The first word starts with a lowercase letter.
- Each subsequent word starts with a capital letter.
- Used for variables and methods (e.g., firstName, calculateTotalAmount)

1.4.3.2 UpperCamelCase (PascalCase):

- Used for class names (e.g., StudentRecord, BankAccount)

In this book we will use UpperCamelCase for classes, and lowerCamelCase for everything else.

```
String userFirstName;     // lowerCamelCase (variable)
int totalItems;           // lowerCamelCase (variable)
void printReport() { ... } // lowerCamelCase (method)
class BankAccount { ... }  // UpperCamelCase (class)
```

1.5 Java variables, assignment, and assignment operator precedence

In Java, assignment operators are used to store values in variables. The most basic is the = operator, but there are compound assignment operators for arithmetic and bitwise operations as well.

Assignment is always from *right to left*: the right-hand side expression is evaluated and that result is stored in the left-hand side variable

1.5a PEMDAS

PEMDAS is an acronym that helps remember the correct order to perform operations in a math expression. The rules ensure everyone solves expressions the same way, avoiding mistakes and confusion.

The PEMDAS order is:

1. Parentheses: Calculate everything inside parentheses (like ( ))—and also consider brackets [ ] and braces { }—first.
2. Exponents: Solve powers and roots, such as x^2, next.
3. Multiplication and Division: Perform these operations *from left to right* as they appear in the expression. They are equal in priority—do not always multiply before dividing.
4. Addition and Subtraction: Finally, do these operations *from left to right* as they occur, since they also have equal priority

Example:
Evaluate: $8+2\times(3^2-1)$

Step-by-step with PEMDAS:

- **Parentheses:** $3^2-1=9-1=8$
- **Exponents:** already handled the previous step.
- **Multiplication:** $2\times8=16$
- **Addition:** $8+16=24$

So, the answer is **24**.

Key tips:

- For multiplication/division and addition/subtraction, go *left to right* within the group, not strictly the letter order in PEMDAS

1.5b Simple Assignment

```
int x = 10;    // Assigns value 10 to variable x
String name = "Java"; // Assigns "Java" to variable name
char initial = 'J'; // Assigns 'J' to variable initial
double pi = 3.14; // Assigns 3.14 to variable pi
float feet = 5.9f; // Assigns 5.9 to variable feet
boolean isJavaFun = true; // Assigns true to variable isJavaFun
String greeting = "Welcome to Java programming!"; // Assigns a greeting message
greeting = greeting + " Enjoy coding!"; // Concatenates a message to greeting
double average = (x + pi) / 2; // Calculates the average of x and pi
boolean isEven = x % 2 == 0; // Checks if x is even
```

1.5c Compound Assignment Operators

Compound assignments combine arithmetic operations with assignment, saving you from repeating the variable:

Operator	Example	Equivalent To	Operation
+=	x += y;	x = x + y;	Addition
-=	x -= y;	x = x - y;	Subtraction
***=**	x *= y;	x = x * y;	Multiplication
/=	x /= y;	x = x / y;	Division
%=	x %= y;	x = x % y;	Modulo

```
int x = 5;
x += 3;    // Same as: x = x + 3;   Now x is 8
x -= 2;    // Same as: x = x - 2;   Now x is 6
```

```
x *= 4;    // Same as: x = x * 4;   Now x is 24
x /= 6;    // Same as: x = x / 6;   Now x is 4
x %= 3;    // Same as: x = x % 3;   Now x is 1


String greeting = "Hello";
greeting += " World"; // Now greeting is "Hello World"
```

Module 1: Learning Materials & References

a) Java Basics and Data Types
b) Java – Data Types - video
c) How to set up Java in Visual Studio Code video
d) Getting Started with Java in VS Code
e) Java Style Guide
f) Java – Variables - video
g) Java – Operators - video
h) Java JOptionPane

Part A: Quiz for Module 1

1.1a What is the main philosophy behind Java's design?

A) Write Once, Run Anywhere
B) Write Everywhere, Run Nowhere
C) Write Once, Compile Everywhere
D) Write Nowhere, Run Everywhere

1.2a Which of the following is NOT a valid Java data type?

A) int
B) double
C) boolean
D) string (lowercase)

1.3a What is the purpose of the JDK?

A) To play Java games
B) To develop and compile Java programs

C) To browse the internet
D) To create digital art

1.4a What does the following Java code do?

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

A) Displays "Hello, Java!" on the screen
B) Declares an integer variable
C) Calculates a sum
D) Draws a circle

1.5a Which of the following class names correctly follows the standard Java naming convention?
A) myClass
B) MyClass
C) My_Class
D) myclass

1.6a Which naming convention is typically used for Java variable names?

A) SCREAMING_SNAKE_CASE
B) lowerCamelCase
C) UPPERCamelCase
D) kebab-case

1.7a Which of these is the correct way to declare and assign a decimal variable in Java?

A) float distance = 5.25;
B) float distance = 5.25f;
C) float distance := 5.25;
D) float distance == 5.25;

1.8a What is a correct declaration and assignment of a boolean variable in Java?

A) boolean isOpen = True;
B) boolean isOpen = yes;

C) boolean isOpen = "true";
D) boolean isOpen = false;

1.9a How do you declare and assign a char variable the value of the letter Z in Java?

A) char letter = 'z';
B) char letter = 'Z';
C) char letter ="Z";
D) char letter = Z;

1.10a Which of the following is the correct way to concatenate two strings in Java?

A) String result = 'Hello' + 'World'
B) String result = concat("Hello", "World");
C) String result = "Hello" + "World";
D) String result = String.add("Hello", "World");

Part B: Reflection

1.1b How did Java's 'Write Once, Run Anywhere' design shape the programming landscape?

1.2b Research the web and find similarities and differences between Java and Python.

Part C: Lab Assignments

1.1c HelloClassmate.java

Create a Java class HelloClassmate.java that declares, age as an int, favoriteFood, name as String, and outputs the following (replace Alice with your classmate's first name, and favorite food):

Sample output: `Hello, my name is Alice. I am 20 years old, and I love sushi!`

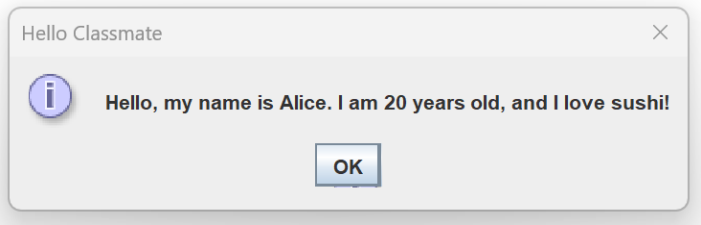
1.2c HelloClassmateDialog.java

Copy HelloClassmate.java to HelloClassmateDialog.java, and Repeat 1.1c exercise but replace console output with JOptionPane dialog. The JOptionaPane is a class in Java's Swing library used to create standard dialog boxes.

Syntax for JOptionaPane:

```
String result = JOptionPane.showInputDialog(null,Message,Title,JOptionPane.QUESTION_MESSAGE)
JOptionPane.showMessageDialog(null,Message,Title,JOptionPane.INFORMATION_MESSAGE);
```
Don't forget to import javax.swing.JOptionPane;
For more information on JOptionPane see module 1: references.

Sample output:



1.3c: MyProfile.java

Create a new Java class MyProfile.java that declares variables to store your name, favoriteFood as String, age (as an integer), your height (as a double), and your first initial (as a character). Output these values to the console.

Sample output:

```
--- My Profile ----

Name: Alice
Age: 20
Height: 5.8
Initial: J
Favorite Food: Sushi

--------------------
```

1.4c: MyProfileDialog.java

Copy MyProfile.java to MyProfileDialog.java and repeat 1.3c exercise but replace console output with JOptionPane dialog

For more information on JOptionPane see module 1: references.

Sample output:

Module 2: Java Control Flow

Control flow refers to the order in which statements and instructions are executed in a Java program. Normally, code runs from top to bottom, but control flow statements allow the program to make decisions and take different paths depending on specific conditions. This makes programs dynamic and responsive, rather than following a fixed sequence every time

Learning Objectives

By the end of this Module, readers will be able to:

- Understand and implement decision-making using if, else if, and else.
- Use switch statements for multiple-case conditions.
- Apply relational (>, <, ==) and logical operators (&&, ||, !) in Java.
- Debug and trace code with branching logic.
- Basic understanding of Pseudocode

2.1. What is Control Flow?

Control flow allows a program to make decisions and choose different paths based on conditions. Instead of executing linearly from top to bottom, programs use branches to handle various scenarios dynamically.

Java offers two main forms of control flow:

- Conditional Statements (if, if-else, switch)
- Logical Operators for combining conditions

2.2. If, Else If, and Else Statements

These are used to run code only when certain conditions are true.

Syntax:

```
boolean condition = true;
boolean anotherCondition = false;

if (condition) {
    // code if true
```

```
        } else if (anotherCondition) {
            // code if previous false, this true
        } else {
            // code if all above are false
        }

    }
```

Examples:

```
    if (score >= 90) {
        System.out.println("Grade: A");
    } else if (score >= 80) {
        System.out.println("Grade: B");
    } else {
        System.out.println("Grade: C or lower");
    }



    age = scanner.nextInt();

    if (age < 18) {
        System.out.println("You are a minor.");
    } else {
        System.out.println("You are an adult.");
    }
```

Notes:

- Java uses boolean expressions: score >= 90 evaluates to true or false.
- Curly braces {} are optional for single statements but always recommended for clarity.

2.3. Comparison and Logical Operators

These logical operators are key to building complex conditions for the control flow. Relational operators (>, <, ==, !=, >=, <=) compare values, while logical operators (&&, ||, !) combine or inverted conditions.

For example:

```
int score = 85;
```

```java
if (score >= 90 && score <= 100) {
    System.out.println("Grade: A");
} else if (score >= 80 && score < 90) {
    System.out.println("Grade: B");
}
```

2.3.1 Comparison Operators:

Operator	*Description*	*Example*
==	Equal to	`a == b`
!=	Not equal to	`a != b`
>	Greater than	`a > b`
<	Less than	`a < b`
>=	Greater than or equal to	`a >= b`
<=	Less than or equal to	`a <= b`

2.3.2 Logical Operators:

Operator	*Meaning*	*Example*
&&	AND	`a > 5 && b < 10`
!	NOT	`!(a > 5)`

Apply Relational and Logical Operators
Relational operators (>, <, ==, !=, >=, <=) compare values, while logical operators (&&, ||, !) combine or inverted conditions. For example:

Mini Lab: Write a Java Code snippet that checks whether a number is positive, negative, or zero.

2.4. Switch Statement

Used as an alternative to multiple if-else conditions when checking a single variable.

```
int option = 2;
    switch (option) {
    case 1:
        System.out.println("Play Game");
        break;

    case 2:
        System.out.println("Load Game");
        break;

    case 3:
        System.out.println("Quit");
        break;

    default:
        System.out.println("Invalid Option");
    }
```

A more modern JDK 17+ switch statement:

```
int option2 = 2;

switch (option2) {
case 1 -> System.out.println("Play Game");
case 2 -> System.out.println("Load Game");
case 3 -> System.out.println("Quit");
default -> System.out.println("Invalid Option");
}
```

Notes:

- break prevents "fall-through" (executing all cases after a match).
- switch works with primitive, classes, and Enums.

Common Pitfall: Forgetting break causes unintended execution of multiple cases. That is why the new Switch statement is preferred.

2.5 Pseudocode

Pseudocode is a plain-language description of the steps in an algorithm. It uses simple, human-readable statements to outline logic, often without strict syntax rules. Pseudocode helps us think through the logic before getting bogged down with Java syntax details. We can think of pseudocode as "programming" thinking about the how at a higher abstract level than "coding" which is the detailed rules and syntax of the language.

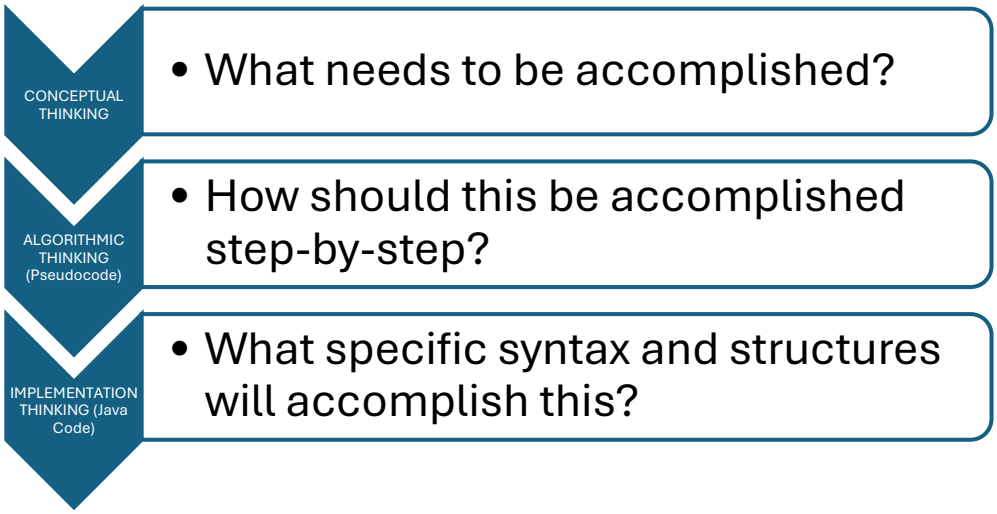
Why Pseudocode Matters in Java Development

Pseudocode helps us think through the logic before getting bogged down with Java syntax details. This separation of concerns allows developers to:

- Focus on problem-solving first - Work out the logical steps without worrying about semicolons, brackets, or method signatures
- Communicate ideas clearly - Share algorithmic thinking with team members regardless of their programming language expertise
- Catch logical errors early - Identify flawed reasoning before investing time in implementation
- Plan complex solutions - Break down complicated problems into manageable, sequential steps

The Thinking Hierarchy

PROBLEM SOLVING LEVELS:



Example Pseudocode:

IF age is greater than or equal to 18

  PRINT "You are an adult."

ELSE

  PRINT "You are a minor."

To convert the above pseudocode into Java, we need to identify the conditional structures similar in Java, and in this case it is the if-else structure.

```
int age = 20; // Example variable

if (age >= 18) {
    System.out.println("You are an adult.");
} else {
    System.out.println("You are a minor.");
    }
```

2.6 The Translation Process

Problem: Find the average grade of students above a certain threshold. A example that you will encounter in module 5:

Step 1 - Conceptual Thinking: *"I need to filter students by grade, then calculate an average of the filtered results."*

Step 2 - Pseudocode (Algorithmic Thinking):

1. READ threshold value from user
2. CREATE empty list for qualifying grades
3. FOR each student in class:
  4. IF student's grade >= threshold:
    5. ADD grade to qualifying list
6. IF qualifying list is not empty:
  7. CALCULATE sum of qualifying grades
  8. CALCULATE average = sum / count
  9. DISPLAY average
10. ELSE:
  11. DISPLAY "No students meet criteria"

Module 2 : Learning Materials & References

a) Java if statements video
b) Ternary Operator – shortcut of – if-else video
c) Java nested if statements video
d) Java Logical Operators with Examples

Part A: Quiz for Module 2

2.1a Which of the following is NOT a type of conditional statement in Java?

A) if
B) switch
C) while
D) else if

2.2a What is the purpose of the else statement in Java?

A) To start a new loop
B) To execute code when the preceding if condition is false
C) To define a new variable
D) To exit a method

2.3a Which operator can be used to combine multiple conditions so that both must be true for the overall condition to be true?

A) ||
B) &&
C) !
D) ==

2.4a What is the correct syntax for a switch statement in Java?

A)

```
switch (variable) {
   case value1 -> // statements
   case value2 -> // statements
}
```

B)

```
switch (variable) {
   if value1: // statements
   if value2: // statements
}
```

C)

```
switch (variable) {
   value1: // statements
   value2: // statements
}
```

D)

```
switch (variable) {
   then value1: // statements
   then value2: // statements
}
```

2.5a What does the break statement do in a switch block?

A) Ends the program
B) Skips the next case and continues
C) Exits the switch block
D) Repeats the current case

2.6a What is the output of the following Java code?

```
int a = 10;
int b = 20;
System.out.println(a < b && a != b);
```

A) true
B) false
C) 1
D) 0

2.7a What is the main benefit of writing pseudocode before Java code?
A) It makes your Java code run faster
B) It helps you think through the steps without worrying about Java syntax
C) It prevents all errors in your program
D) It writes the Java code for you automatically

2.8a Which is NOT a benefit of using pseudocode?
A) Helps you plan your program steps
B) Makes it easier to find mistakes in your logic
C) Fixes all bugs in your Java code
D) Helps team members understand your ideas

Part B: Reflection Prompt

2.1b In which real-world applications would the pseudocode translation process be most beneficial?

2.2b How might critical systems like vending machines or heart rate monitors fail without proper algorithmic planning?

Part C:Lab Assignments:

2.1c StudentCheck Java Class:

Create a Java application StudentCheck, that declares a boolean variable to check if you are a student. Use an if-else statement to print "I am a student." if the variable is true, or "I am not a student." if the variable is false.

2.2c StudentCheckDialog Java Class:

Repeat 2.1c but replacing console dialogs with JOptionPanel Dialogs. It is a class in Java's Swing library used to easily create standard dialog boxes.

Syntax for JOptionaPane:

```
String result = JOptionPane.showInputDialog(null,Message,Title,JOptionPane.QUESTION_MESSAGE)
JOptionPane.showMessageDialog(null,Message,Title,JOptionPane.INFORMATION_MESSAGE);
```
Don't forget to import javax.swing.JOptionPane;

For more information on JOptionPanel see module 1: references.

2.3c FoodMenu.java Class

Create a Java application FoodMenu, that reads the user input using Scanner, and provides feedback based on their selection. This exercise introduces basic input handling, control flow with if-else-if and nested if statements. Make sure to check for invalid options. You need to declare all food items, prices, using variables of appropriate type.

```
Sample output:

=== FOOD MENU ===
1. Pizza - $8.99
2. Burger - $6.5
3. Salad - $5.25
================
Enter the number of the item you want (1-3): 2
You ordered Burger! Enjoy your meal!

=== FOOD MENU ===
1. Pizza - $8.99
2. Burger - $6.5
3. Salad - $5.25
================
Enter the number of the item you want (1-3): 4
Invalid choice. Please select a number between 1 and 3.
```

2.4c FoodMenuS.java Class

Copy FoodMenu.java to FoodMenuS.java and repeat exercise 2.2c but replacing if-else-if block with new switch statements from JDK17+.

Note:

To accept user input in Java, use the **Scanner** class.

First, import the Scanner class at the top of your file:

```
import java.util.Scanner;
```

Then, create a Scanner object to read input from the keyboard:

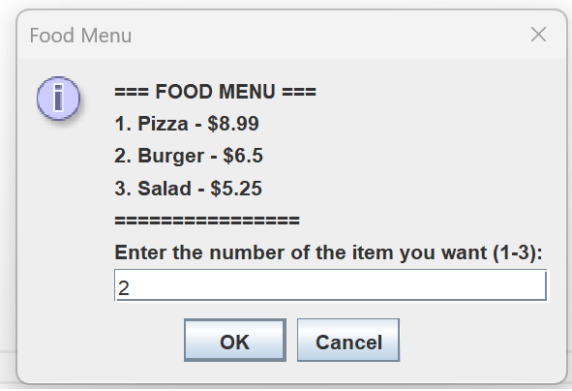
```
Scanner keyboard = new Scanner(System.in);
```

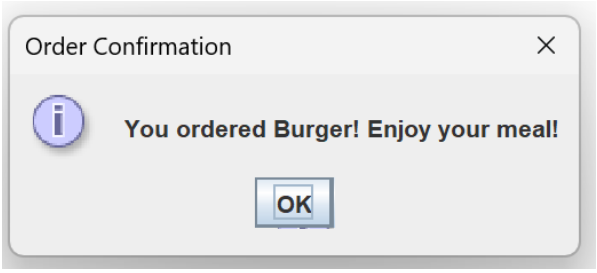
To get an integer input from the user, use the following line:

```
int choice = keyboard.nextInt();
```

2.5c FoodMenuDiaglog.java Class

Copy FoodMenuS.java to FoodMenuDialog.java and repeat exercise 2.4c but replace console input/output with JOptionPane dialogs.





2.6c LetterGrade.java class

Convert the following pseudocode into a working Java application called LetterGrade.

```
Pseudo code for a letter grade system based on user input.
Ask the user to enter a score.
Divide the score by 10. For example:
    If the score is 95 → 95 ÷ 10 = 9
    If the score is 72 → 72 ÷ 10 = 7
Use that result to match a case:
    If the result is 10 or 9 → print A
    If it's 8 → print B
    If it's 7 → print C
    If it's 6 → print D
    Any other number → print F
```

Module 3: Loops and Iteration

Loops are fundamental programming constructs that allow a block of code to be executed repeatedly as long as a specified condition remains true. They are essential for automating repetitive tasks, making code more efficient, and reducing redundancy. Instead of writing the same instruction multiple times, loops enable you to "wrap" the instruction inside a loop structure and let the program handle the repetition automatically

Learning Objectives

By the end of this Module, learners will be able to:

- Understand the purpose and structure of loops.
- Implement while, do-while, and for loops effectively.
- Use break and continue statements to control loop execution.
- Trace and debug iteration logic in common programming problems.

3.1. Why Use Loops?

Loops allow a set of instructions to be executed repeatedly until a condition is no longer true. Common scenarios where loops are invaluable include:

- Counting or aggregating values: For example, summing numbers from 1 to 100 or counting the number of times a condition is met.
- Searching and processing input: Such as reading through a list of user inputs or searching for a specific value in an array.
- Automating repeated actions: Like printing a menu until the user chooses to exit or performing a calculation for each element in a data set.

Instead of writing the same statement over and over, we "wrap" it inside a loop.

3.2. Types of Loops in Java

A loop consists of three main components:

1. Initialization: Setting up a starting value for the loop variable.
2. Condition: The test that determines whether the loop continues.
3. Update: Modifying the loop variable so the condition will eventually become false, preventing infinite loops.

Java provides three primary loop structures: while, do-while, and for loops. Each has its own syntax and use cases, but all serve the purpose of repeating code under certain conditions.

3.2.1 While Loop

The `while` loop checks the condition before each iteration. If the condition is true, the loop body is executed. This continues until the condition becomes false. This loop is best used when the number of iterations is not known in advance

Syntax:

```
while (condition) {
// Code to repeat
}
```

Example:

```
int count = 1;
while (count <= 5) {
    System.out.println("Count: " + count);
    count++;
}
```

Warning:

If the condition never becomes false, the loop runs infinitely.

3.2.2 Do-While Loop

The `do-while` loop checks the condition **after** executing the loop body, so it always runs **at least once**. This loop is useful when you want to ensure the loop body runs at least once, such as in menu-driven applications

Syntax:

```
do {
    // Code to run
} while (condition);
```

Example:

```
int input;
Scanner keyboard = new Scanner(System.in);
System.out.println("Enter numbers (0 to exit):");
```

```
do {
    input = keyboard.nextInt();
    System.out.println("You entered: " + input);
} while (input != 0);
keyboard.close();
```

3.2.3 For Loop

The for loop is ideal when the number of iterations is known beforehand. It combines initialization, condition, and update in one line for clarity and brevity.

Syntax:
```
for (initialization; condition; update) {
    // Code to repeat
}
```

Example:

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

Use Case:

Iterating over ranges, arrays, and when loop variables need to be declared and modified within the loop structure.

3.3. Controlling Loops with `break` and `continue`

The break statement immediately exits the current loop, regardless of the loop condition. It is often used to stop a loop early when a certain condition is met.

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) break;
    System.out.println(i);
}
```

Stops when i reaches 5.

Continue Skips the current iteration and jumps to the **next one**.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
```

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

Skips `i == 3`, prints 1, 2, 4, 5.

3.4. Loop Design Patterns

- Counting Up or Down
- Sentinel Loops: Run until a specific input is detected.
- Nested Loops: A loop inside another loop (used in matrix traversal, printing patterns, etc.)

Nested Example:

```
for (int row = 1; row <= 3; row++) {
    for (int col = 1; col <= 5; col++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

Module 3 : Learning Materials & References

a) Loops in Java
b) While loops in Java
c) Loop Structures and Control Statements
d) Modulo Arithmetic or Remainder

Part A: Quiz for Module 3

3.1a What is the output of the following code?

```
int x = 3;
while (x > 0) {
    System.out.print(x + " ");
    x--;
}
```

A) 3 2 1

B) 3 2 1 0

C) 3 2

D) Infinite loop

3.2a What does this code output?

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

A) 0 1 2 3 4

B) 0 1 2 3

C) 1 2 3 4

D) Infinite loop

3.3a What is the output of this code?

```
int y = 5;
do {
    System.out.print(y + " ");
    y--;
} while (y > 3);
```

A) 5 4

B) 5 4 3

C) 5

D) Infinite loop

3.4a You need to prompt a user for input at least once and repeat until valid input is received. Which loop is most appropriate?

A) for loop
B) while loop
C) do-while loop
D) Nested if statements

3.5a You need to print numbers from 1 to 10. Which loop is best suited?

A) for loop
B) while loop
C) do-while loop
D) All are equally suitable

3.6a A loop must run only if a condition is true initially and stop when it becomes false. Which loop should you use?

A) for loop
B) while loop
C) do-while loop
D) switch statement

3.7a You want to generate a random integer between 5 (inclusive) and 20 (inclusive). Which code snippet accomplishes this using the Random class?

A)
```
Random rand = new Random();
int num = rand.nextInt(20);
```

B)
```
Random rand = new Random();
int num = rand.nextInt(16) + 6;
```

C)
```
Random rand = new Random();
int num = rand.nextInt(5, 21);
```

D)

```
Random rand = new Random();
int num = rand.nextInt(21) + 5;
```

Part B: Reflection Prompt

3.1b Where in real life do we repeat tasks in cycles, and how can that be modeled in code with loops?

Part C:Lab Assignments

3.1c UpperAZ.java class

Create an UpperAZ java class that outputs only 26 uppercase letters of the English alphabet, and only 7 letters per line.

Hints: use modulo arithmetic with 7

Sample output:

```
Upper case A-Z, 7 per line:

A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
```

3.2c Multiplication.java Class

Use nested loops to output multiplication tables from 1 to 10.

Sample Output:

1 x 1 = 1
1 x 2 = 2
...
10 x 10 = 100

3.3c. RandomNumbers.java Class

Allow the user to generate a random number, until a ‘q’ or ‘Q’ entered to stop the program

Sample output:

```
Generate random numbers. Enter 'q' or 'Q' to quit.
Random number: -1136909465
Press Enter to generate another, or 'q'/'Q' to quit:
Random number: 931268184
Press Enter to generate another, or 'q'/'Q' to quit:
Random number: -890670818
Press Enter to generate another, or 'q'/'Q' to quit: q
Program stopped. Goodbye!
```

Notes:

To generate a random number in Java, you have several options, but for this class we will use **Random class**.

The Random class from the java.util package provides methods to generate random numbers of different types.

Steps:

1. Import the Random class:
   ```
   import java.util.Random;
   ```
2. Create a random object
   ```
   Random random = new Random();
   ```
3. Generate a random integer:
   ```
   int randomNumber;
   randomNumber = random.nextInt();
   ```
4. To generate a random integer within a specific range (e.g., 1 to 99):
   ```
   randomNumber = random.nextInt(1,99);
   ```

3.4c. GuessingGame.java Class

Create Java program for a guessing game where the program generates a random number, and the user tries to guess it. The program loops until the user guesses correctly, providing Welcome to the Guessing Game!

```
I have chosen a number between 1 and 100. Try to guess it!
Enter your guess: 8
Too low. Try again!
```

```
Enter your guess: 90
Too high. Try again!
Enter your guess: 45
Too low. Try again!
Enter your guess: 50
Too low. Try again!
Enter your guess: 55
Too low. Try again!
Enter your guess: 70
Too high. Try again!
Enter your guess: 60
Too low. Try again!
Enter your guess: 65
Too low. Try again!
Enter your guess: 68
Too low. Try again!
Enter your guess: 69
Correct! You guessed the number: 69hints ("Too high" or "Too low") along the
way.”).
```

Module 4: Arrays and Strings in Java

Arrays and strings are fundamental data structures in Java, essential for organizing and manipulating collections of data. This module introduces you to one-dimensional (1D) arrays, basic two-dimensional (2D) arrays, and the powerful String class with its methods.

Learning Objectives

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

- Understand what arrays are and how they store multiple values of the same type.
- Declare, initialize, and access elements in arrays.
- Work with common array operations (traversal, search, update).
- Understand strings as sequences of characters in Java.
- Perform common string operations using Java's String class.

In Java, an array is a data structure used to store multiple values of the same data type in a single variable. Think of it as a row of boxes, where each box holds a value and has a unique index.

4.1 1D Arrays (One-Dimensional Arrays)

A one-dimensional array in Java is a linear collection of elements of the same type, stored in contiguous memory locations. Arrays allow you to group multiple values into a single variable, making data management more efficient and your code more organized

Declaring and Initializing 1D Arrays

You can declare an array by specifying its type and using square brackets. For example:

```
int[] numbers; // declares an array of integers
String[] names; // declares an array of strings
```

To initialize an array, you can use an array initializer:

```
int[] numbers = {10, 20, 30, 40};
String[] names = {"Alice", "Bob", "Charlie"};
```

Alternatively, you can use the `new` keyword to allocate memory:

```
int[] numbers = new int[5]; // array of 5 integers, initialized to 0
String[] names = new String[3]; // array of 3 strings, initialized to null
```

Numeric arrays are initialized to zero, boolean arrays to `false`, and object arrays (like `String`) to `null`

4.1.1 Accessing and Modifying Array Elements

Array elements are accessed using zero-based indexing:

```
int firstNumber = numbers[0]; // gets the first element
names[1] = "Bobby"; // changes the second element
```

You can find the length of an array using the `length` property:

```
int arrayLength = numbers.length; // returns 5 for the above example
```

5.1.2 Iterating Through Arrays

Loops are commonly used to process array elements:

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

Or we can use the enhanced for-loop:

```
for (var name : names) {
    System.out.println(name);
}
```

The enhanced for-loop syntax is concise and declarative. It can be read as "for each name in the collection of names." The above enhance for-loop declares a new variable `name` that will hold an element of name from the collection of `names` for each pass of the loop. The usage of `var` instructs the compile to match the compatible type with the type of elements stored in the collection.

4.2 Basic 2D Arrays (Two-Dimensional Arrays)

A **two-dimensional array** is essentially an array of arrays, often used to represent matrices or grids

4.2.1 Declaring and Initializing 2D Arrays

You declare a 2D array using two sets of square brackets:

```
int[][] matrix;
```

Alternatively, you can use the `new` keyword to allocate memory:

```
int[][] matrix = new int[3][3]; // 3 rows, 3 columns
```

You can initialize a 2D integer array in Java using an array initializer, which is a block of code within curly braces `{}` used at the time of declaration. This is sometimes referred to as an anonymous block initialization because you don't explicitly call `new int[][]` with dimensions. Anonymous initialization can be done as follows:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Another common method to initialize 2D array in Java is the nested for loop. A nested for loop is a loop placed inside the body of another loop. The inner loop is executed completely for each iteration of the outer loop.

```
    // Outer loop for iterating through rows
    for (int i = 0; i < rows; i++) {
        // Inner loop for iterating through columns of the current row
        for (int j = 0; j < cols; j++) {
            // Assign a value to the element at matrix[i][j]
            matrix[i][j] = /* some value */;
        }
    }
```

4.2.2 Accessing Elements in 2D Arrays

Elements are accessed using two indices:

```
int value = matrix[1][2]; // gets the element in the second row, third column
```

4.2.3 Iterating Through 2D Arrays

Nested loops are used to traverse 2D arrays:

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

Summary Table: 1D vs 2D Arrays

Feature	1D Array	2D Array
Declaration	`int[] arr;`	`int[][] matrix;`
Initialization	`{1, 2, 3}`	`{{1,2},{3,4}}`
Access	`arr`	`matrix[1][2]`
Length	`arr.length`	`matrix.length`
Element Type	Any type	Any type

4.3 Java String

In Java, a String is **not** a primitive type like int or double. It's a **class** — part of the java.lang package — designed to hold sequences of characters, such as "Hello" or "123abc!".

Even though it's a class, Java lets you create strings using a simple and intuitive syntax:

4.3.1 Creating Strings

Strings can be created directly or using the new keyword:

```
String greeting = "Hello";
String name = new String("Alice");
```

Internally, a `String` is an **immutable** sequence of Unicode characters — meaning once a string object is created, it cannot be changed.

4.3.2 Why Strings Are Immutable

Immutability ensures that `String` objects are inherently safe for sharing in multi-threaded environments, as their state cannot be altered once created, eliminating the need for synchronization when accessed by multiple threads. This property also enables Java to optimize memory usage by storing string literals in a shared string pool—only one copy of each unique literal is kept, which improves efficiency. Furthermore, because a `String`'s value never changes, it is highly reliable as a key in hash-based collections such as `HashMap`, since its hash code remains consistent throughout its lifetime.

4.3.3 Key String Methods

Java provides many built-in methods for string manipulation. Here are some of the most common ones:

Method	Description	Example
`length()`	Number of characters	`"abc".length()` ⟶ `3`
`charAt(i)`	Character at index `i`	`"abc".charAt(1)` ⟶ `'b'`
`substring(start, end)`	Extract substring	`"hello".substring(1, 3)` ⟶ `"el"`
`equals()`	Check equality	`"hi".equals("hi")` ⟶ `true`
`toUpperCase()`	Uppercase string	`"java".toUpperCase()` ⟶ `"JAVA"`
`trim()`	Removes spaces	`" abc ".trim()` ⟶ `"abc"`
`split(",")`	Split string	`"a,b,c".split(",")` ⟶ `["a","b","c"]`
`replace("a","@")`	Replace characters	`"Java".replace("a","@")` ⟶ `"J@v@"`

Example Usage:

```
String text = "  Java Programming  ";
System.out.println(text.length()); // 19
System.out.println(text.trim().length()); // 16
System.out.println(text.toUpperCase()); // "  JAVA PROGRAMMING  "
System.out.println(text.contains("Java")); // true
String[] words = text.trim().split(" "); // splits into ["Java", "Programming"]
```

4.4 Sorting Algorithms

Sorting in computer science refers to the process of arranging elements in a list or array in a specific order, most commonly in numerical or alphabetical (lexicographical) order, either from smallest to largest (ascending) or largest to smallest (descending). The main goal of a

sorting algorithm is to transform an unsorted collection into a sorted one, where each element is compared and placed in its correct position according to the desired order.

Sorting is fundamental in computer science because many other algorithms (like searching or merging) work much more efficiently when the input data is sorted. Sorting is also useful for organizing data so it is easier to read or process or ask questions, such as which item is the most expensive or conversely which item is the least expensive.

There are many different sorting algorithms, such as bubble sort, selection sort, insertion sort, merge sort, and quicksort. Each algorithm has its own way of comparing and rearranging elements, and they differ in terms of speed, memory usage, and complexity. In your introductory Java class, you will learn some of these basic sorting methods to understand how computers organize and manage data.

Bubble Sort

Bubble sort is one of the simplest sorting algorithms, often used as an introductory example in programming courses. Its name comes from the way smaller or larger elements "bubble" to the top or end of the list as the algorithm progresses

This method is easy to understand and visualize, making it an excellent starting point for learning about sorting. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until no more swaps are needed, which means the array is sorted.

In bubble sort, we use nested Loops or Loop within a Loop

1. Outer loop → controls the number of passes.
2. Inner loop → compares adjacent elements and performs swaps.
3. With each pass, the unsorted portion gets smaller.

Example:

```
public static void bubbleSort(int[] array) {
int n = array.length;
boolean swapped;
for (int i = 0; i < n-1; i++) {
    swapped = false;
    for (int j = 0; j < n-i-1; j++) {
        if (array[j] > array[j+1]) {
            int temp = array[j];
            array[j] = array[j+1];
            array[j+1] = temp;
            swapped = true;
            }
        }
    if (!swapped) break; // no swaps means array is sorted
    }
}
```

Given an array of [5, 1, 4, 2]

1. **Pass 1**: [1, 4, 2, 5] (5 bubbles to the end).
2. **Pass 2**: [1, 2, 4, 5] (4 moves to its place).
3. **Pass 3**: No swaps → sorted.

4.5.1 Comparison of Different Sort Methods

Algorithm	Best Case	Average/Worst Case	Space	Stability	Use Case
Bubble Sort	O(n)	$O(n^2)$	O(1)	Stable	Small datasets, education
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	Unstable	Small datasets
Insertion Sort	O(n)	$O(n^2)$	O(1)	Stable	Nearly sorted data

Module 4: Learning Materials & References

1. Arrays in Java
2. String in Java
3. Arrays in Java tutorial video
4. Java – Strings - video
5. Java printf() more control on output video
6. Bubble Sort video

Part A: Quiz

4.1a How does bubble sort swap elements in an array?

A) It uses a temporary variable to help switch two elements' places
B) It moves elements to a new array
C) It uses special swapping commands that don't need extra variables
D) It swaps elements only at the start and end of the array

4.2a Which is the correct way to initialize a 1D integer array with values 1, 2, 3?
A) int[] arr = new int[]{1, 2, 3};
B) int arr = {1, 2, 3};
C) int[] arr = new int[3]{1, 2, 3};
D) int[] arr = (1, 2, 3);

4.3a What is the correct syntax for iterating over a String[] names array using an enhanced for loop?
A) for (int i = 0; i < names.length; i++)
B) for (String name : names)
C) for (names => String name)
D) for (String name in names)

4.4a Which code initializes a 2D array with rows {1, 2} and {3, 4}?
A) int[][] matrix = {{1, 2}, {3, 4}};
B) int matrix[][] = new int[2][2];
C) int[][] matrix = [1][2][3,[4];
D) int[][] matrix = {1, 2, 3, 4};

4.5a What does "Hello".substring(1, 4) return?
A) "ell"
B) "Hel"
C) "ello"
D) "ell"

4.6a Given int[][] grid = {{1, 2}, {3, 4}};, what is grid[1]?
A) 1
B) 2
C) 3
D) 4

4.7a What is the difference between arr.length and str.length()?
A) arr.length is a method, str.length() is a field.
B) arr.length is a field, str.length() is a method.
C) Both are methods.
D) Both are fields.

4.8a Which method would you use to compare two strings for equality, regardless of their case?

A) equals()
B) compareTo()
C) equalsIgnoreCase()
D) contains()

4.9a Which code snippet correctly checks if the string message contains the word "hello" (it could be part of another word), regardless of case?

A. message.toLowerCase().contains("hello")
B. message.equals("hello")
C. message.indexOf("hello")
D. message.substring(5)

4.10a Which of the following is an advantage of bubble sort?

A) It is highly efficient for large datasets
B) It is easy to understand and simple to implement
C) It requires additional memory proportional to the input size
D) It always requires fewer comparisons than other sorting algorithms

4.11a What happens if the end index in substring(int beginIndex, int endIndex) is greater than the string length?

A) The method returns the substring from beginIndex to the end of the string
B) The method throws a StringIndexOutOfBoundsException
C) The method returns an empty string
D) The method returns the whole string

4.12a How do you swap two elements in an array during bubble sort in Java?
A)

```
array[j] = array[j+1];
array[j+1] = array[j];
```

B)

```
int temp = array[j];
array[j] = array[j+1];
array[j+1] = temp;
```

C)

```
array[j+1] = array[j];
array[j] = array[j+1];
```

D)

```
array[j] = temp;
temp = array[j+1];
array[j+1] = array[j];
array[j] = temp;
```

Part B: Reflection Prompt

4.1b What debugging strategies would you use to verify your array operations are working correctly?

4.2b How would you design the assignment 2.c? Would you choose to use a 2D array only, or would you store everything in a mix of 1D and 2D array? explain your decision.

4.3b What is the main idea behind the bubble sort algorithm?

Part C:Lab Assignments

All assignments must conform to java best practices, including headers, name, class, and using javadoc to document the Java application, modular programming with methods.

4.1c TextAnalyzer.java Class

A simple Java program that analyzes a sentence by counting the number of words, vowels, consonants, and total letters. This program prompts the user to enter a sentence and then performs the analysis.

```
=== Text Analyzer ===
Enter a sentence: Hello World! This is a test.

--- Analysis Results ---
Sentence: Hello World! This is a test.
Word count: 6
Vowel count: 7
Consonant count: 11
Total letters: 18
```

4.2c StudentGrades.java Class

```
This exercise demonstrates the usage of 1D and 2D Arrays to manage class data.

 Data Structure Requirements:
  1. Use a 1D array to store student names
  2. Use a 1D array to store corresponding student grades
  3. Use a 2D array to combine names and grades for a class of 5 students

 The Class will includes methods to:
  1 Print the entire class grades.
  2 Calculate the average grade.
  3 Find the highest grade.

       Sample output

       Intro to Java
       ----------------
       Name       Grade
       ----------------
       Alice      85
       Bob        92
       Maria      78
       Ali        90
       Antoine    88

       Class Statistics:
       Total Students: 5
       Average Grade: 86.60
       Highest Grade: 92

       ----------------
```

4.3c CharSort.java Class

Overloading BubbleSort Method in 4.4 example to sort an array of characters.

Example:

```
char[] char_array = {'z', 'B', 'C', '1', '5'};
bubbleSort(char_array);
```

Sample output:

Original char array: [z, B, C, 1, 5]
Sorted char array: [1, 5, B, C, z]

Module 5: Methods and Modularity

As programs grow larger and more complex, writing all the code in a single block becomes unwieldy and difficult to maintain. Methods (sometimes called functions in other languages) allow you to break down your code into manageable, reusable pieces. This module introduces you to the power of methods and modularity in Java, helping you write cleaner, more organized, and more efficient code.

5.1. Defining and Calling Methods

Methods are blocks of code that perform a specific task and can be called (invoked) from other parts of your program. Defining a method involves specifying its signature—which includes the method's name, return type, and parameters—along with its body, which contains the executable code.

Example:

```
public class Example extends Object {
    /**
     * Method greet
     * This method prints a greeting message to the console.
     * It does not take any parameters and does not return any value.
     */
    public static void greet() { // method's signature
        System.out.println("Hello, world!");
    }
    /**
     * Main method
     * This is the entry point of the program.
     * It calls the greet method to display a message.
     * @param args not used
     */
    public static void main(String[] args) {
        greet(); // Method call
    }
}
```

In the above example, `greet()` is a method that prints a greeting. It is called from the main method.

5.2. Methods Parameters & Return Values

Parameters allow you to pass data into a method, while return values let a method send data back to the code that invoked or called it.

Example:

```
public class Example extends Object {
    /**
     * Method with parameters and return value
     * This method takes two integers as parameters and returns their sum.
     *
     * @param a First integer
     * @param b Second integer
     * @return The sum of a and b
     */
    public static int add(int a, int b) {
        return a + b;
    }
    /**
     * Main method to demonstrate the add method
     * This method calls the add method with two integers and prints the result.
     * @param args not used
     */
    public static void main(String[] args) {
        int result = add(3, 5); // result = 8
        System.out.println(result);
    }
}
```

This method takes two integers, adds them, and returns the result.

5.2.1 Parameters & Return Values

Parameters allow you to pass data into a method, while return values let a method send data back to the code that invoked or called it.

Example:

```
public class Example extends Object {
    /**
     * Method with parameters and return value
     * This method takes two integers as parameters and returns their sum.
     * @param a First integer
     * @param b Second integer
     * @return The sum of a and b
     */
    public static int add(int a, int b) {
        return a + b;
    }

    /**
     * Method with no parameters and no return value
     * @param args not used
     * This is the main method that serves as the entry point of the program.
     */
    public static void main(String[] args) {
        int result = add(3, 5); // result = 8
        System.out.println(result);
    }
}
```

5.3 Method Overloading

Method overloading is a feature that allows you to have multiple methods with the same name, in this example, **add** but with different parameter types. The first add method accepts two integers' parameters, the second add method accepts two double parameters. We are overloading the name add so that we can send different parameter types.

Example:

```
public class Example extends Object {
    /**
     * Method with parameters and return value
     * This method takes two integers as parameters and returns their sum.
     * @param a First integer
     * @param b Second integer
     * @return The sum of a and b
     */
    public static int add(int a, int b) {
        return a + b;
    }
    /**
     * Method with parameters and return value
     * This method takes two doubles as parameters and returns their sum.
     * @param a First double
     * @param b Second double
     * @return The sum of a and b
     */
    public static double add(double a, double b) {
        return a + b;
    }
    /**
     * Method with no parameters and no return value
     * @param args not used
     * This is the main method that serves as the entry point of the
program.
     */
    public static void main(String[] args) {
        int result = add(3, 5); // result = 8
        double resultDouble = add(3.5, 2.5); // doubleResult = 6.0
        System.out.println(result);
        System.out.println(resultDouble);
    }
}
```

5.5 Javadoc

So far, Module 4 has introduced the tools needed to write large Java applications in a modular, reusable, and maintainable way using well-structured methods. In addition to methods, another critical component of the JDK that supports the long-term maintenance of Java programs is **Javadoc**.

Javadoc is vital for maintaining Java code because it offers a structured and standardized approach to documenting the purpose, behavior, and usage of your code. This documentation enhances the clarity and modularity provided by well-designed methods, ensuring that your codebase remains understandable, scalable, and easy to manage throughout its lifecycle.

Javadoc-generated **HTML** so that every component of a Java class—fields, constructors, methods, inner classes, and inherited members—has **clickable** links for easy navigation, similar to Oracle's official String class documentation in the example below. Javadoc automatically creates hyperlinks for all variables, methods, etc., building summary tables with anchor links, navigation via tables or sidebars, thereby improving readability and discoverability.

Live example Javadoc for String:
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/String.html

5.5.1 Standardized and Structured Documentation

Javadoc comments follow a specific format, using special tags and syntax that are recognized by both humans and tools. This standardization ensures that documentation is consistent across projects and teams, making it easier for anyone to understand and contribute to the codebase

Unlike simple inline comments, Javadoc is designed to be comprehensive, covering not just what the code does, but also how and why it does it, including parameters, return values, exceptions, and usage examples.

In the example below we use /** */ as Javadoc tags, not comment /* */.

The Javadoc header begin with a description of the purpose of the method add, what are the parameter types and purposes of the parameters, it also includes a tag @return which indicates what will be return to the caller from this method.

```java
/**
 * Method with parameters and return value
 * This method takes two integers as parameters and returns their sum.
 * @param a First integer
 * @param b Second integer
 * @return The sum of a and b
 */
public static int add(int a, int b) {
    return a + b;
}
```

Using the Javadoc tool, we can generate HTML documentation as part of the build process, making it easily available for both reference and distribution. Below is a command line example on how to generate the HTML document for the Example.java class.

```
javadoc -d docs -private -author Example.java
```

Command Breakdown

Component	Explanation
`javadoc`	The Java tool that generates API documentation in HTML format from Java source comments (those written in `/** ... */` format).
`-d docs`	Specifies the **output directory** for the generated documentation. In this case, it will create (or use) a folder named `docs`.
`-private`	Includes **private class members** (fields, methods, constructors) in the generated documentation. By default, Javadoc only includes `public` and `protected` members.
`-author`	Includes the `@author` tag (if present) in the documentation.
`Example.java`	The **Java source file** to generate documentation from.

Module 5: Learning Materials & References

a) Methods in Java
b) Method Definitions and Overloading
c) How to write Java Doc
d) What is Javadoc and how to use it?

Part A: Quiz

5.1a What is the correct way to declare a method that does not return any value?

A) public int myMethod()
B) public void myMethod()
C) public String myMethod()
D) public myMethod()

5.2a Which of the following best describes the purpose of method overloading?

A) To allow methods with the same name but different parameter lists
B) To allow methods with the same name and parameter list but different return types
C) To allow methods with only the same name
D) To allow methods with only the same return type

5.3a Which of the following is a valid Javadoc comment for a method?

A) // Adds two numbers
B) /* Adds two numbers */
C) /** Adds two numbers */
D) # Adds two numbers

5.4a What is the method signature in the following declaration?

```
public static double calculate(int x, double y)
```

A) calculate
B) public static double calculate(int x, double y)
C) calculate(int x, double y)
D) double calculate(int x, double y)

5.5a What does the following Javadoc tag describe in a method?

A) The return value
B) The method name
C) The parameter x
D) The method description

5.6a Which statement is true about Java methods?

A) All methods must return a value
B) Methods can have zero or more parameters
C) Method names must start with a capital letter
D) Methods cannot be called from other methods

5.7a Why is Javadoc important for maintaining modular and maintainable code?

A) It automatically fixes bugs
B) It provides standardized, readable documentation that explains how to use methods
C) It replaces the need for comments
D) It compiles code faster

Part B: Reflection Prompt

5.1b Why is it important to write code that is easy to understand and update?

5.2b How can good comments and documentation help reduce the cost of fixing or changing software later?

Part C:Lab Assignments

All assignments must conform to java best practices, including headers, name, class, and using javadoc to document the Java application, modular programming with methods.

5.1c Metric2English.java Class

Metric to English Conversions, write a Java program with two methods:
1/ celsiusToFahrenheit which converts a temperature from Celsius to Fahrenheit.
2/ kmToMiles which Converts a distance from kilometers to miles.
In the main method, call both methods with sample values and print the results.
3/ Metric2English must use best practices Javadoc
Sample output:

```
Welcome to the Metric to English Converter!
    Iteration: 0
30.0°C is 86.0°F
100.0 km is 62 miles
    Iteration: 1
35.0°C is 95.0°F
105.0 km is 65 miles
    Iteration: 2
40.0°C is 104.0°F
```

110.0 km is 68 miles
    Iteration: 3
45.0°C is 113.0°F
115.0 km is 71 miles
    Iteration: 4
50.0°C is 122.0°F
120.0 km is 75 miles

Use Math.round() to round miles.

5.2c Calculator.java Class

Write a Java program that performs basic arithmetic operations (addition, subtraction, multiplication, division) using separate methods for each operation.

Sample output:

```
Welcome to the Calculator Program!
Enter first number: 11
Enter second number: -4

Addition: 7
Subtraction: 15
Multiplication: -44
Division: -2

Thank you for using the Calculator Program!
```

Module 6: Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects rather than functions and logic. Java is a class-based, object-oriented language, making it essential to understand the core concepts of classes, objects, and their interactions.

6.1 Classes and Objects

- **Classes** are blueprints or templates for creating objects. They define the attributes (fields) and behaviors (methods) that the objects created from the class will have
- **Objects** are instances of a class. When you create an object, you instantiate a class, resulting in a unique entity with its own set of attributes and behaviors

Example:

- A Student class might define attributes like name, studentID, major, etc...
- An object of Student could be Maria, with specific values for each attribute.

6.1 Fields, Constructors, and this Keyword

- Fields (also called member variables or instance variables) are variables defined within a class and are accessible to all methods in the class. They store the state of an object
- Constructors are special methods used to initialize objects. A default constructor is called automatically when an object is created. It has the same name as the class and no return type
- Default Constructor: Takes no arguments and is provided by Java if none is defined.
- Non-default or Parameterized Constructor: Takes arguments to initialize fields with specific values
- The **this** Keyword refers to the current object. It is used to distinguish between class fields and parameters or local variables with the same name

Example:

```
// Non-default constructor
public Student(String name, int studentId, String major) {
    this.name = name;
    this.studentId = studentId;
    this.major = major;
}
```

6.2 Class Methods: Defining Object Behavior

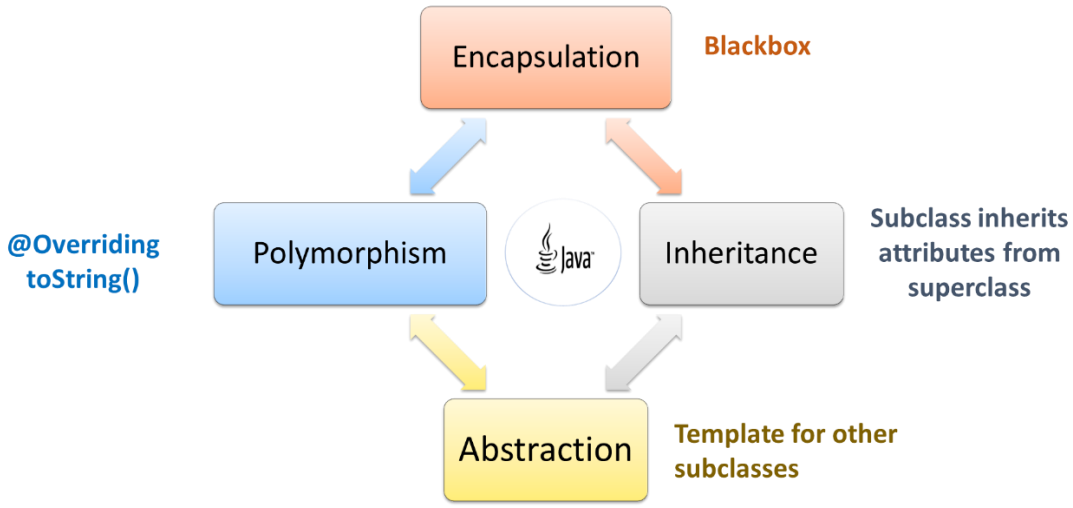
by calling each other's methods, passing messages, or accessing each other's fields.

Example:

- A `StudentRegistry` class may store a list of `Student` objects and provide methods to add, remove, or print students.
- Each `Student` object can interact with the registry by being added to it or by providing its information when requested.

6.3 The Four Pillars of Object-Oriented Programming (OOP)

The four fundamental principles, or "pillars," of Object-Oriented Programming (OOP) are the concepts that essential for designing robust, scalable, and maintainable software.



Object-Oriented Programming is a paradigm based on the concept of "objects," which can contain data in the form of fields (often known as attributes or properties) and code in the form of procedures (often known as methods). The four main pillars that support this paradigm are Encapsulation, Abstraction, Inheritance, and Polymorphism.

6.3.1 Encapsulation – Protect the data

Encapsulation is the practice of bundling an object's data (attributes) and the methods (functions) that operate on that data into a single, self-contained unit called a class.

Crucially, it involves restricting direct access to some of an object's components, a concept also known as data hiding.

An object should control its own state. The internal data of an object is kept private to prevent external code from accidentally or maliciously corrupting it. To read or modify this data, external code must use the object's public methods (a well-defined public interface).

Analogy: A Car Engine
Think of a car's engine. It's a complex system of pistons, valves, and spark plugs (the data and internal methods). The engine is enclosed in a metal casing (the class). As a driver, you don't interact with these parts directly. Instead, you use a simple public interface: the ignition key and the accelerator pedal (the public methods). This prevents you from messing up the engine's delicate internal state while still allowing you to use its functionality.

Implementation in Java:

- Declare the class's instance variables as private.
- Provide public methods, known as **getters** (to read data) and **setters** (to modify data), which act as the gatekeepers to the private data.

```
public class Student extends Object{

    /* Every class in Java implicitly subclass of Object, so this is optional
    but it's good practice to show that you understand inheritance.*/


    // Private Fields (instance variables)
    private String name;
    private int studentId;
    private String major;

    // Getter methods Public
    public String getName() {
        return name;
    }
    // Setter methods public
    public void setName(String name) {
        this.name = name;
    }
```

Why it's useful:

- Prevents external interference and misuse.
- Controls how data is accessed and modified.
- Increases code maintainability and flexibility.

6.3.2 Abstraction – Hide the complexity

Abstraction means hiding complex implementation details and exposing only the essential, high-level functionality to the user. It focuses on *what* an object does rather than *how* it does it.

Simplify complexity. Users of a class don't need to know the intricate details of its inner workings. They only need to understand the public interface that allows them to interact with it. Encapsulation is the mechanism that *enables* abstraction.

Analogy: A Television Remote

When you use a TV remote, you interact with simple buttons like "Power," "Volume Up," and "Channel Down." You know what they do. You don't need to know the complex details of infrared signals, frequency modulation, or the TV's internal circuitry. The complexity is abstracted away, leaving you with a simple interface.

Implementation in Java:

- Using **abstract classes** and **interfaces**. These define a "contract" of methods that a class must implement, but they don't provide the implementation itself.

Java Example:

```
// 1. The interface defines WHAT a vehicle can do, not HOW.
interface Vehicle {
    void startEngine();
    void stopEngine();
}

// 2. The implementation is hidden inside the concrete class.
class Car implements Vehicle {
    @Override
    public void startEngine() {
        // Complex logic: check fuel, engage spark plugs, etc.
        System.out.println("Car engine started.");
    }
    // ... other methods
```

}

Why it's useful:

- Simplicity: Makes systems easier to think about and use.
- Decoupling: Isolates the impact of changes. You can completely change how a method works internally without affecting any code that calls it.
- Focus: Allows programmers to focus on a higher level of design without getting bogged down in details.

6.3.3 Inheritance – Reuse code

Inheritance is a mechanism that allows a new class (the subclass or child class) to acquire the properties (fields) and behaviors (methods) of an existing class (the superclass or parent class).

Promote code reuse and establish a logical "is-a" relationship. For example, a Dog *is an* Animal. A Car *is a* Vehicle. The child class inherits the common attributes and methods and can add its own unique ones or modify the inherited ones.

Analogy: A Family

A child inherits traits like eye color and last name from their parents. They share these common attributes but can also have their own unique skills and characteristics, like being a great musician or artist.

Implementation in Java:

- Using the extends keyword.

Inheritance allows one class to **inherit fields and methods** from another class, promoting code reuse and a hierarchical relationship.

Java uses the `extends` keyword for inheritance.

```
/** Person class representing a person
 * This class is a simple example of inheritance in Java.
 * It can be extended by other classes, such as Student.
*/
class Person extends Object {
    /* Every class in Java implicitly subclass of Object, so this is optional
```

```
    but it's good practice to show that you understand inheritance.*/
    String name;
    int age;
    /**
     * Constructor for the Person class.
     * This constructor initializes the properties of the Person class.
     * @param name
     * @param age
     */
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

}
/**
 * The Student class extends Person, inheriting its properties and methods.
*/
public class Student extends Person{
/**
 * It inherits from the Person class, which means it has access to the properties
and methods of that class.
 * It can also have its own properties and methods.
 */
    String studentId;
    String major;

    /**
     * Constructor for the Student class.
     * This constructor initializes the properties of the Student class.
     * @param name
     * @param age
     * @param studentId
     * @param major
     */
    public Student(String name, int age, String studentId, String major) {
        super(name, age);
        this.studentId = studentId;
        this.major = major;
    }
```

```
}
```

Why it's useful:

- Reduces redundancy.
- Encourages reuse of existing code.
- Allows for logical class hierarchies.

6.3.4 Polymorphism – Many forms

Polymorphism, which means "many forms," is the ability of an object, method, or operator to take on multiple forms. In programming, it means a single interface (like a method signature) can be used for a general class of actions.

One name, many actions. The specific action that gets executed depends on the exact nature of the object. This allows you to write flexible, generic code that can work with objects of different types.

Analogy: A Smartphone's "Open" Command

Imagine your phone's operating system has an "open" command.

- If you use "open" on a photo file, it launches the gallery app.
- If you use "open" on a web link, it launches the browser.
- If you use "open" on a contact file, it opens the contacts app.

The action is the same ("open"), but the result is different based on the type of object you are opening.

Implementation in Java:

- Method Overriding (Runtime Polymorphism): A subclass provides its own specific implementation of a method that is already defined in its parent class. This is the most common form of polymorphism.

- Method Overloading (Compile-time Polymorphism): A class has multiple methods with the same name but different parameters (e.g., add(int a, int b) and add(double a, double b)).

```
/* Every class in Java implicitly subclass of Object, so this is optional
    but it's good practice to show that you understand inheritance.*/
class Animal extends Object {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow");
    }
}
```

Why it's useful:

- Flexibility & Extensibility: Allows you to treat objects of different types in a uniform way. You can add new classes without changing the code that uses them.
- Decoupling: Eliminates the need for long if-else or switch statements that check the type of an object.

The Four Pillar

Pillar	Purpose	Example Keyword(s)
Encapsulation	Hide internal data	`private`, `getter`, `setter`
Abstraction	Hide complex implementation	`interface`, `abstract`
Inheritance	Reuse code across classes	`extends`
Polymorphism	Use one interface for many forms	`override`, `overload`

6.4 Student Class:

A typical Java class that manifests 3 of 4 OOP pillars: Encapsulation, , Inheritance, and Polymorphism

```java
/**
 * The Student class extends Person, inheriting its properties and methods.
*/
public class Student extends Person{
/**
 * It inherits from the Person class, which means it has access to
 * the properties and methods of that class.
 * It can also have its own properties and methods.
 */
    private String studentId;
    private String major;

    /**
     * Non-default Constructor for the Student class.
     * This constructor initializes the properties of the Student class.
     * @param name
     * @param age
     * @param studentId
     * @param major
     */
    public Student(String name, int age, String studentId, String major) {
        super(name, age);
        this.studentId = studentId;
        this.major = major;
    }
    /** default constructor
     * This constructor initializes the properties of the Student class
       with default values.
     */
    public Student() {
        super("Unknown", 0);
        this.studentId = "Unknown";
        this.major = "Undeclared";
    }
    public String getStudentId() {
        return studentId;
    }
    public String getMajor() {
        return major;
    }
    public void setStudentId(String studentId) {
```

```
        this.studentId = studentId;
    }
    public void setMajor(String major) {
        this.major = major;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + ", getStudentId()=" +
                                                getStudentId() + ", getMajor()="
                                                + getMajor() + "]";
    }
}
```

6.4.1 Instantiate Objects from Student class

Instantiating an object means creating a specific, usable instance of a class. In Java, this is done using the new keyword, which allocates memory for the object and calls the class constructor to initialize its state

Step-by-Step Process

1. Class Definition
    - The Student class acts as a blueprint, defining what attributes (fields) and behaviors (methods) a student object will have.
2. Object Instantiation
    - To create a Student object, use the new keyword followed by the constructor of the Student class.
3. What Happens in Memory?
    - Memory Allocation: The `new` operator allocates memory for the new `Student` object.
    - Constructor Call: The constructor is called to set the initial values for the object's fields (e.g., `name`, `studentId`, `major`).
    - Reference Assignment: The variable (`alice`) is assigned the memory address of the new object, allowing you to access its fields and methods

Example:

```
public static void main(String[] args) {
    // Create a student using the non-default constructor
```

```
    Student alice = new Student("Alice", 12345, "Computer Science");
    System.out.println(alice); //using the toString method implicitly

    // Create a student using the default constructor
    Student bob = new Student();
    bob.setName("Bob");
    bob.setStudentId(67890);
    bob.setMajor("Mathematics");
    System.out.println(bob);
}
```

Step	Description
Class Definition	Blueprint for what a student is (fields, methods)
Object Instantiation	Use `new` and constructor to create a specific student object
Memory Allocation	Allocates memory for the new object
Constructor Call	Initializes the object's fields
Reference Assignment	Assigns the object's memory address to a variable for access
Multiple Objects	Can create many student objects, each with unique data

To instantiate an object from the `Student` class, use the `new` keyword and the appropriate constructor. This creates a new student in memory, initializes its fields, and allows you to work with it in your program.

6.5.1 Fruit.java Class

```
/**
 * The Fruit class with pH levels.
 * <pre>
 * Name: Your Name
 * Course: CSIS 293
 * Professor: Ahn Nuzen
 * </pre>
 */
class Fruit extends Object {
    /* Every class in Java implicitly subclass of Object, so this is optional
    but it's good practice to show that you understand inheritance.*/
    private String name;
    private String color;
```

```
private double weight;
private double phLevel;

public Fruit(String name, String color, double weight, double phLevel) {
    this.name = name;
    this.color = color;
    this.weight = weight;
    this.phLevel = phLevel;
}

// Getters to enforce encapsulation
public String getName() { return name;}
public String getColor() { return color;}
public double getWeight() { return weight;}
public double getPhLevel() { return phLevel;}

// Setters to enforce encapsulation
public void setName(String name) { this.name = name;}
public void setColor(String color) { this.color = color;}
public void setWeight(double weight) {
    if (weight > 0) {
        this.weight = weight;
    }
}

public void setPhLevel(double phLevel) {
    if (phLevel >= 0 && phLevel <= 14) {
        this.phLevel = phLevel;
    }
}

public String getAcidityLevel() {
    if (phLevel < 7) {
        return "Acidic";
    } else if (phLevel == 7) {
        return "Neutral";
    } else {
        return "Basic";
    }
}
```

```
    // Implement Polymorphism
    @Override
    public String toString() {
        return "Fruit{name='" + name + "', color='" + color + "', weight=" + weight
               + "g, pH=" + phLevel + " (" + getAcidityLevel() + ")}";
    }
}
```

6.5. DemoFruit Class

It's good programing practice to have a separate class to test another class, the test class is usually called DemoClassname, or TestClassname. In this example, we will create a DemoFruit.java class to test the Fruit.java Class.

```
/**
 * Demonstration of the Fruit class with pH levels.
 * <pre>
 * Name: Your Name
 * Course: CSIS 293
 * Professor: Ahn Nuzen
 * </pre>
 */

/* Every class in Java implicitly subclass of Object, so this is optional
    but it's good practice to show that you understand inheritance.*/

public class DemoFruit extends Object {
    /**
     * Main method to demonstrate the Fruit class functionality.
     * It creates instances of Fruit, tests getters and setters,
     * @param args not used.
     */
    public static void main(String[] args) {
        // Create instances with pH levels
        Fruit apple = new Fruit("Apple", "Red",150.0, 3.3);
        Fruit banana = new Fruit("Banana","Yellow",120.0, 5.9);

        System.out.println("=== Initial Fruit Information ===");
        System.out.println(apple);
        System.out.println(banana);
```

```
        System.out.println("\n=== Testing Getters ===");
        System.out.println("Fruit name: " + apple.getName());
        System.out.println("Fruit weight: " + apple.getWeight());
        System.out.println("Fruit name: " + banana.getName());
        System.out.println("Fruit phLevel: " + banana.getPhLevel());

        System.out.println("\n=== Testing Setters ===");
        apple.setWeight(200);
        apple.setPhLevel(3.1);

        banana.setWeight(118);
        banana.setPhLevel(5.1);

        System.out.println("After modifications:");
        System.out.println(apple);
        System.out.println(banana);

        System.out.println("\n=== pH Level Analysis ===");
        Fruit[] fruits = {apple, banana};
        for (Fruit fruit : fruits) {
            System.out.println(fruit.getName() + " - pH: " + fruit.getPhLevel() +
                               " (" + fruit.getAcidityLevel() + ")")}
        System.out.println("\n=== Testing validation in setters ===");
        System.out.println("Trying to set invalid pH (15) for apple...");
        apple.setPhLevel(15); // Should not change pH
        System.out.println("Apple pH after invalid attempt: " + apple.getPhLevel());

        System.out.println("Trying to set negative weight (-50) for banana...");
        banana.setWeight(-50.0); // Should not change weight
        System.out.println("Banana weight after invalid attempt: " + banana.getWeight());

        System.out.println("\n=== Polymorphism with remaining fruits ===");
        for (Fruit fruit : fruits) {
            System.out.println("Processing: " + fruit);
        }
    }
}
```

Module 6: Learning Materials & References

a) Java OOP
b) Java Objects (OOP) Video
c) Java Constructors Video

Part A: Quiz

1. What is the purpose of the Polygon class in the inheritance example?

A) To create circles
B) To serve as a blueprint for all shape classes
C) To calculate the perimeter of any shape
D) To print the area of a shape

2. Which of the following is NOT one of the four pillars of OOP?

A) Abstraction
B) Encapsulation
C) Inheritance
D) Iteration

3. How is encapsulation demonstrated in the example?

A) By using public instance variables
B) By using private/protected instance variables and public methods (getters/setters)
C) By using static variables
D) By using comments

4. What is the purpose of a default constructor in Java?

A) To initialize all instance variables to specific values
B) To allow object creation with default values
C) To destroy objects
D) To print object details

5. Which of the following is an example of abstraction in OOP?

A) Creating a class with private fields and public methods
B) Showing only the necessary features and hiding implementation details
C) Inheriting properties from another class
D) Allowing methods to perform differently based on the object

6. What is polymorphism in the context of OOP?

A) The ability of an object to take many forms
B) The process of wrapping data and methods into a single unit
C) The use of getters and setters
D) The creation of instance variables

7. What is the main benefit of using getters and setters in Java?

A) To increase code length
B) To control access to instance variables and enforce encapsulation
C) To replace constructors
D) To print object state

8. What would a typical toString() method do in a Java class?

A) Convert the object to an integer
B) Return a string representation of the object
C) Set instance variables to default values
D) Print all private variables

9. Which access modifier would you use for an instance variable if you want it to be accessible only within its own class?

A) public
B) private
C) protected
D) default (package-private)

10. In the inheritance example, how is inheritance demonstrated?

A) By using the extends keyword to inherit from a superclass
B) By using static methods
C) By using private constructors
D) By using interfaces

Part B: Reflection Prompt

6.1b If you were designing a program for a zoo, what might be your base class? What would the subclasses be?

6.2b Think of a real-world example where inheritance makes sense. How would you model it in Java?

Part C:Lab Assignments

All assignments must conform to java best practices, including headers, name, class, and using javadoc to document the Java application, modular programming with methods.

6.1c Book.java Class

Create a `Book` class with the following:
Fields: `title`, `author`, `yearPublished`
Constructor to initialize the fields
A `toString()` method to print the book info
Then, in your `main()` method:
Create two `Book` objects.
Call `print Details()` for each.

6.2c Person.java Class

Create a Person class with a **private** name field.

- Add a getter and setter for name.
- Create a subclass Employee that adds employeeId and a method to print employees' information.

Demonstrate the use of:

- the setter to assign a name
- a method in the subclass accessing the parent's field via the getter

Module 7: Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherit the properties (fields) and behaviors (methods) of another class. This creates a parent-child relationship between classes, promoting code reusability and logical organization.

Inheritance promotes code reuse in Java by enabling a subclass (child class) to inherit attributes and methods from a superclass (parent/base class) that contains common properties and behaviors. Rather than duplicating code across multiple classes, the shared attributes or futures are defined once in the superclass. Subclasses then automatically gain access to these inherited fields and methods and can also add their own unique features or override inherited ones as needed.

For example, if you have a superclass called Animal with methods like eat() and sleep(), subclasses such as Dog and Cat can reuse these methods without rewriting them. The subclasses can also add new methods or override inherited ones to provide specialized behavior

This approach reduces redundancy, makes your code easier to maintain, and allows you to build class hierarchies that model real-world relationships more naturally

Inheritance ensures that shared logic is written once and reused across related classes, streamlining development and improving code organization.

7.1 Inheritance in Java

How inheritance works in Java:

- **Superclass (Parent/Base Class):** The class whose properties and methods are inherited.
- **Subclass (Child/Derived Class):** The class that inherits from the superclass.

To implement inheritance in Java, you use the `extends` keyword.

Key Benefits:

- Code Reusability: Reuse fields and methods from existing classes, reducing code duplication.
- Extensibility: Easily extend existing classes to add new features.

- Logical Hierarchy: Organize classes in a hierarchical manner, making code easier to manage and understand

Important Notes:

- Access Modifiers: Fields and methods marked as private in the superclass are not accessible directly by the subclass. Use protected or public for inheritance.
- Constructors: Constructors are not inherited, but they can be called from the subclass using the super() keyword.
- Types of Inheritance: Java supports single, multilevel, and hierarchical inheritance, but not multiple inheritance (a class cannot extend more than one class directly)

Inheritance is widely used to model relationships where one class is a specialized version of another (e.g., Dog is an Animal), making it easier to maintain and extend code

Java's transitive inheritance means that when you build complex class hierarchies, a subclass inherits not only from its immediate superclass but also, indirectly, from all super classes up the inheritance chain—all the way to the root class, `Object`. This has several important implications for the structure and behavior of your class hierarchies:

- Automatic Inheritance of Methods and Fields:
  Subclasses inherit all accessible (public, protected, and—if in the same package—package-private) fields and methods from every class up the hierarchy, not just their direct parent. This allows you to define common behaviors and attributes once at a high level and have all descendants inherit them
- Polymorphism and Flexibility:
  Because every class in the hierarchy is also an instance of its superclasses (all the way up to `Object`), you can treat any subclass instance as an instance of any superclass in the chain. This enables polymorphism, where code written to work with a superclass can automatically work with any subclass
- All Classes Are Objects:
  Since all classes ultimately inherit from `Object`, they inherit basic methods like `toString()`, `equals()`, and `hashCode()`. This provides a consistent interface for all Java objects, but also means that these methods may need to be overridden in subclasses to provide meaningful behavior

In summary, Java's transitive inheritance allows for powerful code reuse and flexible hierarchies, but it also means that changes and design decisions at the top of the hierarchy can ripple through many classes, affecting both structure and functionality throughout your program.

7.2. Fruit.java a Super Class

The below java code defines a Fruit superclass that serves as a foundation for inheritance in Java. Here are some of its key components and design principles:

7.2.1 Class Declaration and Inheritance:

The class explicitly extends Object, which is redundant since every Java class implicitly inherits from Object. However, as the comment notes, this demonstrates understanding of Java's inheritance hierarchy where Object is the root of all classes.

```
class Fruit extends Object {
```

7.2.2 Instance Variables (Attributes):

All attributes are declared `private`, implementing **encapsulation** - a core OOP principle. This means these variables cannot be accessed directly from outside the class, protecting the internal state.

```
    private String name;
    private String color;
    private double weight;
    private double phLevel;
```

7.2.3 Constructor:

The constructor initializes all four attributes when a Fruit object is created. It uses this. to distinguish between parameters and instance variables.

```
public Fruit(String name, String color, double weight, double phLevel) {
    this.name = name;
    this.color = color;
    this.weight = weight;
    this.phLevel = phLevel;
}
```

7.2.4 Getter Methods (Accessors)

These provide controlled read access to private attributes, maintaining encapsulation while allowing external code to retrieve values.

```
// Getters to enforce encapsulation
public String getName() { return name;}
public String getColor() { return color;}
public double getWeight() { return weight;}
public double getPhLevel() { return phLevel;}
```

7.2.5 Setter Methods (Mutators)

The setters include **validation logic**:
Weight must be positive
pH level must be between 0-14 (valid pH range)
This prevents invalid data from corrupting the object's state.

```
public void setWeight(double weight) {
    if (weight > 0) {
        this.weight = weight;
    }
}
public void setPhLevel(double phLevel) {
    if (phLevel >= 0 && phLevel <= 14) {
        this.phLevel = phLevel;
    }
}
```

7.2.6 Business Logic Method

This method provides domain-specific functionality, converting numeric pH values into meaningful acidity classifications.

```
public String getAcidityLevel() {
    if (phLevel < 7) {
        return "Acidic";
    } else if (phLevel == 7) {
        return "Neutral";
    } else {
        return "Basic";
    }
}
```

7.2.7 Polymorphism Implementation

```
    // to implement polymorphism
    @Override
    public String toString() {
              return "Fruit{name='" + name + "', color='" + color + "', weight=" + weight +
              "g, pH=" + phLevel + " (" + getAcidityLevel() + ")}";
    }
```

The `@Override` annotation indicates this method overrides the `toString()` method inherited from `Object`. This enables polymorphism - different fruit types can have their own string representations while maintaining a consistent interface.

7.2.8 Design as a Superclass

This Fruit class is a good example of well-designed as a superclass because it:

1. Provides common attributes that all fruits share (name, color, weight, pH)
2. Implements shared behavior (acidity classification)
3. Uses protected access where appropriate for inheritance
4. Follows encapsulation principles with private attributes and public methods
5. Includes validation to maintain data integrity

Subclasses like Apple, Orange, or Banana can extend this class and inherit all these features while adding their own specific attributes and methods.

Below is the full java code of the superclass Fruit.

```
/**
 * The Superclass Fruit class with pH levels.
 * <pre>
 * Name: Your Name
 * Course: CSIS 293
 * Professor: Ahn Nuzen
 * </pre>
 */
class Fruit extends Object {
    /* every class in Java is implicitly a subclass of Object, so this is optional but it's
good practice to show that you understand inheritance.*/

    private String name;
    private String color;
    private double weight;
    private double phLevel;
```

```
public Fruit(String name, String color, double weight, double phLevel) {
    this.name = name;
    this.color = color;
    this.weight = weight;
    this.phLevel = phLevel;
}

// Getters to enforce encapsulation
public String getName() { return name;}
public String getColor() { return color;}
public double getWeight() { return weight;}
public double getPhLevel() { return phLevel;}

// Setters to enforce encapsulation
public void setName(String name) { this.name = name;}
public void setColor(String color) { this.color = color;}
public void setWeight(double weight) {
    if (weight > 0) {
        this.weight = weight;
    }
}

public void setPhLevel(double phLevel) {
    if (phLevel >= 0 && phLevel <= 14) {
        this.phLevel = phLevel;
    }
}

public String getAcidityLevel() {
    if (phLevel < 7) {
        return "Acidic";
    } else if (phLevel == 7) {
        return "Neutral";
    } else {
        return "Basic";
    }
}
// to implement polymorphism
@Override
public String toString() {
```

```
        return "Fruit{name='" + name + "', color='" + color + "', weight=" + weight +
        "g, pH=" + phLevel + " (" + getAcidityLevel() + ")}";
    }
}
```

7.3. Apple.java a Subclass of Fruit.java Class

This code demonstrates how to create a subclass in Java inheritance. The Apple class extends the Fruit superclass, inheriting its functionality while adding apple-specific features. Here are some of its key components and design principles:

7.3.1 Class Declaration and Inheritance

```
class Apple extends Fruit {
```

The extends keyword establishes that Apple is a subclass of Fruit. This means Apple automatically inherits all non-private attributes and methods from the Fruit class.

7.3.2 Additional Attributes

The subclass adds two new private attributes specific to apples:

- variety: Stores the apple type (e.g., "Granny Smith", "Red Delicious")
- isOrganic: Boolean flag indicating if the apple is organically grown

These complement the inherited attributes (name, color, weight, phLevel) from the superclass.

7.3.3 Constructor with Super Call

Key points about the constructor:

a) super() call: The first line calls the parent class constructor, passing "Apple" as the name along with the other inherited parameters
b) Parameter handling: Notice that name isn't a parameter since all Apple objects will have "Apple" as their name
c) Subclass initialization: After calling super(), it initializes the Apple-specific attributes
d) The super() call must be the first statement in a subclass constructor.

```
public Apple(String color, double weight, double phLevel, String variety, boolean isOrganic) {
    super("Apple", color, weight, phLevel);
    this.variety = variety;
    this.isOrganic = isOrganic;
}
```

7.3.4 Getter and Setter Methods

These provide encapsulated access to the Apple-specific attributes, following the same pattern as the superclass.

7.3.5 Method Overriding

This demonstrates polymorphism:

- Overrides the toString() method from the superclass
- Uses inherited getter methods (getColor(), getWeight(), etc.) to access superclass data
- Calls inherited method (getAcidityLevel()) for functionality
- Provides Apple-specific string representation while maintaining the same method signature

```
// to implement polymorphism
@Override
public String toString() {
    return "Apple{variety='" + variety + "', organic=" + isOrganic + ", color='" +
           getColor() +
           "', weight=" + getWeight() + "g, pH=" + getPhLevel() +
            " (" + getAcidityLevel() + ")}";
}
```

7.3.6 Apple Class Specific Behavior

This method is unique to the Apple class and wouldn't be available in the generic Fruit superclass or other fruit subclasses.

```
public void makeJuice() {
    System.out.println("Making apple juice from " + variety + " apple");
}
```

7.3.7 What Apple Inherits

From the Fruit superclass, Apple automatically gets:

a) All public and protected methods (getName(), getColor(), getWeight(), getPhLevel(), getAcidityLevel(), setters)
b) Access to inherited functionality without rewriting code
c) The ability to be treated as a Fruit object (polymorphism)

7.3.8 Complete java code of Apple.java subclass:

```
/**
 * The Apple class subclass of Fruit.
 * <pre>
 * Name: Your Name
 * Course: CSIS 293
 * Professor: Ahn Nuzen
 * </pre>
 */
class Apple extends Fruit {
    private String variety;
    private boolean isOrganic;

    public Apple(String color, double weight, double phLevel, String variety, boolean isOrganic) {
        super("Apple", color, weight, phLevel);
        this.variety = variety;
        this.isOrganic = isOrganic;
    }

    // Getters to enforce encapsulation
    public String getVariety() {
        return variety;
    }

    public boolean isOrganic() {
        return isOrganic;
    }

    // Setters to enforce encapsulation
    public void setVariety(String variety) {
        this.variety = variety;
    }

    public void setOrganic(boolean organic) {
        this.isOrganic = organic;
    }

    // to implement polymorphism
    @Override
    public String toString() {
        return "Apple{variety='" + variety + "', organic=" + isOrganic + ", color='" +
```

```
                getColor() +
                "', weight=" + getWeight() + "g, pH=" + getPhLevel() +
                 " (" + getAcidityLevel() + ")}";
    }

    public void makeJuice() {
        System.out.println("Making apple juice from " + variety + " apple");
    }
}
```

7.4 Banana.java a Subclass of Fruit.java Class

Banan.java is another subclass implementation, demonstrating how different fruit types can extend the same Fruit superclass while having their own unique characteristics. Banana subclass follows the same design patterns as Apple with class declaration and inheritance using extends Fruits, Constructor calling super class Fruit to pass on the information, Getters, and Setters to enforce encapsulation, and Method overriding.

7.4.1 Constructor Design Differences

```
        public Banana(double weight, double phLevel, String ripeness, double potassiumContent)
        {
                super("Banana", "Yellow", weight, phLevel);
                this.ripeness = ripeness;
                this.potassiumContent = potassiumContent;
            }
```

Notable design choices:

- Hardcoded values: The constructor automatically sets name to "Banana" and color to "Yellow"
- Simplified parameters: Unlike Apple, the constructor doesn't take color as a parameter since bananas are typically yellow
- Focused parameters: Only asks for the variable attributes (weight, pH, ripeness, potassium content)
- This shows how subclasses can make different assumptions about their inherited properties based on real-world characteristics.

7.4.2 Complete java code of Banana.java subclass

```
/**
 * The Banana class subclass of Fruit.
 * <pre>
 * Name: Your Name
 * Course: CSIS 293
 * Professor: Ahn Nuzen
 * </pre>
 */
class Banana extends Fruit {
    private String ripeness;
    private double potassiumContent;

    public Banana(double weight, double phLevel, String ripeness, double potassiumContent) {
        super("Banana", "Yellow", weight, phLevel);
        this.ripeness = ripeness;
        this.potassiumContent = potassiumContent;
    }

    // Getters to enforce encapsulation
    public String getRipeness() {
        return ripeness;
    }

    public double getPotassiumContent() {
        return potassiumContent;
    }

    // Setters to enforce encapsulation
    public void setRipeness(String ripeness) {
        this.ripeness = ripeness;
    }

    public void setPotassiumContent(double potassiumContent) {
        if (potassiumContent >= 0) {
            this.potassiumContent = potassiumContent;
        }
    }

    // to implement polymorphism
```

```
    @Override
    public String toString() {
        return "Banana{ripeness='" + ripeness + "', potassium=" + potassiumContent +
               "mg, color='" + getColor() +
               "', weight=" + getWeight() + "g, pH=" + getPhLevel() + " (" + getAcidityLevel()
                       + ")}";
    }

    public void makeSmootie() {
        System.out.println("Blending " + ripeness + " banana for smoothie");
    }
}
```

7.5 Comparison with Apple Class

Below table lists the similarities, differences between Banana and Apple subclasses:

Aspect	Apple		Banana
Constructor parameters	Takes color as input		Hardcodes color as "Yellow"
Unique attributes	variety, isOrganic		ripeness, potassiumContent
Unique method	makeJuice()		makeSmootie()
Validation	None in setters		Validates potassium ≥ 0

7.6 Inheritance Benefits Demonstrated

The Fruit class hierarchy provides a comprehensive illustration of inheritance advantages in object-oriented programming. Here's an enhanced analysis of the key benefits:

Code Reusability and DRY Principle (Do not repeat yourself)

Instead of duplicating common fruit properties across multiple classes, inheritance allows shared functionality to be written once in the superclass:

Code reuse: Both `Apple` and `Banana` inherit:

1. Basic fruit properties (name, color, weight, pH)
2. Acidity classification logic
3. Getter methods for inherited attributes
4. Basic validation for weight and pH

7.7 DemoFruit.java a Test Class

This java application program demonstrates Java inheritance, polymorphism, and object-oriented programming principles using the `Fruit` class hierarchy.

This class by convention is called **driver class** (main class) or java application that tests the functionality of the `Fruit`, `Apple`, and `Banana` classes. It's designed to demonstrate all aspects of the inheritance hierarchy.

7.7a Object Creation and Initialization

```
Apple apple = new Apple("Red", 150.0, 3.3, "Gala", true);
Banana banana = new Banana(120.0, 5.9, "ripe", 358.0);
```

Creates instances of both subclasses with initial values:

- **Apple:** Red Gala apple, 150g, pH 3.3 (acidic), organic
- **Banana:** 120g, pH 5.9 (slightly acidic), ripe, 358mg potassium

7.7.b Demonstrates polymorphism:

When println() calls toString(), it automatically uses the overridden version from each subclass, showing different formatted output for each fruit type.

```
System.out.println("\n=== pH Level Analysis ===");
Fruit[] fruits = {apple, banana};
for (Fruit fruit : fruits) {
    System.out.println(fruit.getName() + " - pH: " + fruit.getPhLevel() +
                       " (" + fruit.getAcidityLevel() + ")");
}
```

7.7.c Testing Setter Methods and Validation

Modifies object states using setter methods, then displays the updated objects to verify changes took effect.

```
System.out.println("\n=== Testing validation in setters ===");
System.out.println("Trying to set invalid pH (15.0) for apple...");
apple.setPhLevel(15.0); // Should not change pH
System.out.println("Apple pH after invalid attempt: " + apple.getPhLevel());
```

7.7.d Enhanced Loop Array Processing

- Creates an array of `Fruit` references containing different subclass objects
- Processes them uniformly using inherited methods

- Shows how different object types can be treated as their common superclass

```
System.out.println("\n=== Polymorphism with remaining fruits ===");
for (Fruit fruit : fruits) {
    System.out.println("Processing: " + fruit);
}
```

7.7.e Complete java code of DemoFruit.java Driver Class

```
/**
 * Super Clas Fruit class with pH levels.
 * <pre>
 * Name: Your Name
 * Course: CSIS 293
 * Professor: Ahn Nuzen
 * </pre>
 */

public class DemoFruit extends Object {
    /**
     * Main method to demonstrate the Fruit class functionality.
     * It creates instances of Fruit, tests getters and setters,
     * @param args not used.
     */
    public static void main(String[] args) {
        // Create instances with pH levels
        Apple apple = new Apple("Red", 150.0, 3.3, "Gala", true);
        Banana banana = new Banana(120.0, 5.9, "ripe", 358.0);

        System.out.println("=== Initial Fruit Information ===");
        System.out.println(apple);
        System.out.println(banana);

        System.out.println("\n=== Testing Getters ===");
        System.out.println("Apple variety: " + apple.getVariety());
        System.out.println("Apple is organic: " + apple.isOrganic());
        System.out.println("Banana ripeness: " + banana.getRipeness());
        System.out.println("Banana potassium: " + banana.getPotassiumContent() + "mg");

        System.out.println("\n=== Testing Setters ===");
        apple.setVariety("Honeycrisp");
```

```
        apple.setWeight(175.0);
        apple.setPhLevel(3.1);

        banana.setRipeness("overripe");
        banana.setPotassiumContent(400.0);

        System.out.println("After modifications:");
        System.out.println(apple);
        System.out.println(banana);

        System.out.println("\n=== pH Level Analysis ===");
        Fruit[] fruits = {apple, banana};
        for (Fruit fruit : fruits) {
            System.out.println(fruit.getName() + " - pH: " + fruit.getPhLevel() +
                               " (" + fruit.getAcidityLevel() + ")");
        }

        System.out.println("\n=== Subclass-specific methods ===");
        apple.makeJuice();
        banana.makeSmootie();

        System.out.println("\n=== Testing validation in setters ===");
        System.out.println("Trying to set invalid pH (15.0) for apple...");
        apple.setPhLevel(15.0); // Should not change pH
        System.out.println("Apple pH after invalid attempt: " + apple.getPhLevel());

        System.out.println("Trying to set negative weight (-50) for banana...");
        banana.setWeight(-50.0); // Should not change weight
        System.out.println("Banana weight after invalid attempt: " + banana.getWeight());

        System.out.println("\n=== Polymorphism with remaining fruits ===");
        for (Fruit fruit : fruits) {
            System.out.println("Processing: " + fruit);
        }
    }
}
```

7.8 Visualizing Inheritance with UML

UML stands for Unified Modeling Language. It is a visual modeling language used to represent the structure, behavior, and interactions of software systems, especially in object-oriented programming like Java.

UML is not a programming language — it's a standard way to visually document:

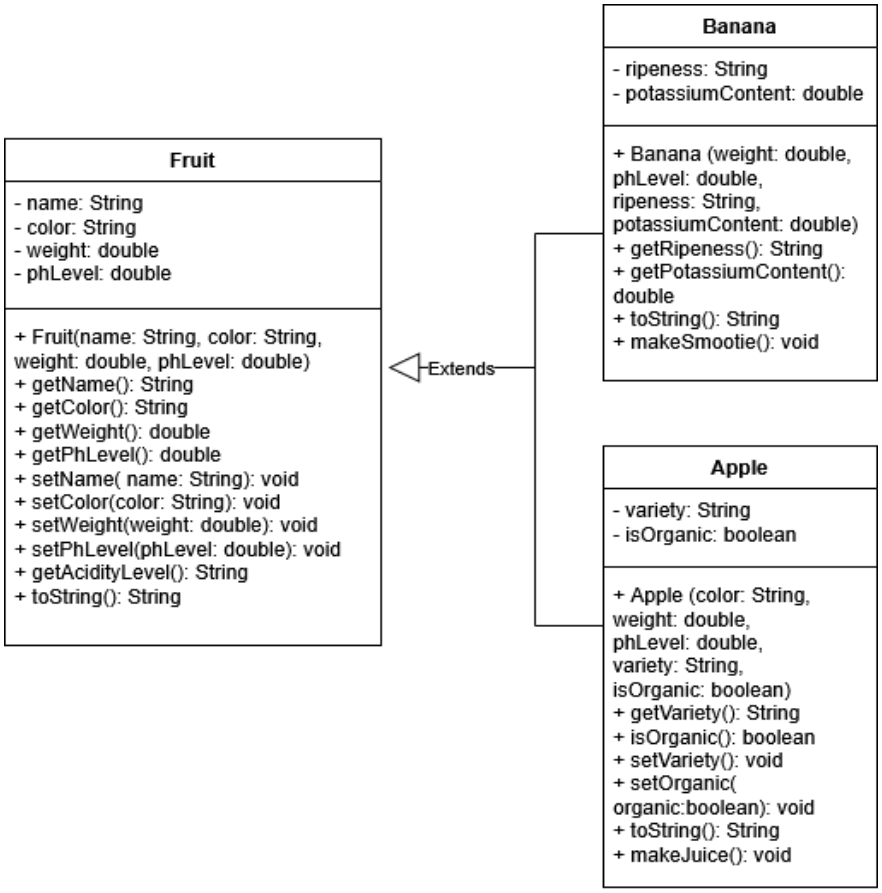
- Classes and their relationships
- Objects and their interactions
- System workflows and architecture

7.8.1 UML Applications:

a) Visualizes System Design
   UML helps developers and stakeholders see the architecture before coding begins.
b) Clarifies Object-Oriented Concepts
   It shows how classes, objects, inheritance, and interfaces relate — perfect for understanding OOP.
c) Improves Communication
   UML diagrams serve as a common language among:
   - Developers
   - Designers
   - Project managers
   - Clients
d) Assists in Planning and Maintenance
   - Helps new developers understand existing code.
   - It makes refactoring and debugging easier.
e) Supports Agile and Waterfall Development
   You can use UML for both formal documentation or lightweight sketching, depending on your process.

UML of the Fruit Superclass:

The UML diagram bellow illustrates the inheritance hierarchy, encapsulation principles, and the relationship between the superclass Fruit and its subclasses Apple and Banana in the previous example.



UML Diagram explanations:

- Rectangle boxes represent classes
- Top section contains the class name
- Middle section contains attributes (fields) with visibility modifiers:
  - – means private
- + means public
- # means protected
- Bottom section contains methods with visibility modifiers
- Triangle arrow (△) pointing upward represents inheritance (extends)
- Solid line connects subclasses to superclass

Key UML Elements Shown:

1. Inheritance relationship: Apple and Banana both extend Fruit
2. Encapsulation: All instance variables are private (-)
3. Public interface: All getters, setters, and methods are public (+)
4. Method overriding: toString() appears in all classes (overridden in subclasses)
5. Subclass-specific methods: makeJuice() in Apple, makeSmootie() in Banana
6. Constructor parameters: Shown with parameter types, non-default

7.9 Abstract Class & Interfaces

Java abstract classes and interfaces are both ways to achieve abstraction - hiding implementation details while showing only essential features.

An abstract class is a class that cannot be instantiated (you can't create objects from it directly). It's like a blueprint that other classes must extend and complete.

Key features:

- Declared with the abstract keyword
- Can have both abstract methods (no implementation) and concrete methods (with implementation)
- Can have instance variables, constructors, and static methods
- A class can only extend one abstract class

7.9.1 Language.java Abstract Class

This code illustrates a real-world example of abstract class of Language. It is abstract because conjugation works differently across languages. For example:

- English: "I walk, you walk, he/she walks"
- Spanish: "Yo camino, tú caminas, él/ella camina"
- French: "Je marche, tu marches, il/elle marche"

Since each language has its own rules, the abstract class says, "every language MUST have conjugation, but I can't tell you exactly how to do it."

Why Conjugation () is abstract:

- Every language conjugates verbs differently
- The parent class can't provide a "one-size-fits-all" implementation
- Forces each language subclass to provide its own specific rules, or implement the Conjugation method!

Why Capitalization() is concrete:

- Most languages capitalize words similarly (first letter uppercase)
- No need to force each subclass to rewrite this common functionality

```
abstract class Language extends Object {
    // This method MUST be implemented by every subclass
    abstract void Conjugation();

    // This method is already implemented - all languages can use it
    void Capitalization() {
        System.out.println("This is a regular method, most languages have a similar way to
                                                                    capitalize words.");
    }
}
```

7.9.2 Spanish.java a Subclass of Language

The Spanish is a subclass of Language, as such, it **must** implement Conjugation() because Conjugation() is just an method without body. A concrete class can't instantiate from a Abstract class without implementing the Abstract method.

```
class Spanish extends Language {
    void Conjugation() {
        System.out.println("Spanish Implementation of method Conjugation");
    }
    public static void main(String[] args) {
        Spanish spanish = new Spanish();
        spanish.Conjugation(); // Calls the Spanish implementation of Conjugation
        spanish.Capitalization(); // Calls the inherited method from Language
    }
```

7.9.3 Why use Abstract class?

The Abstract class is useful because:

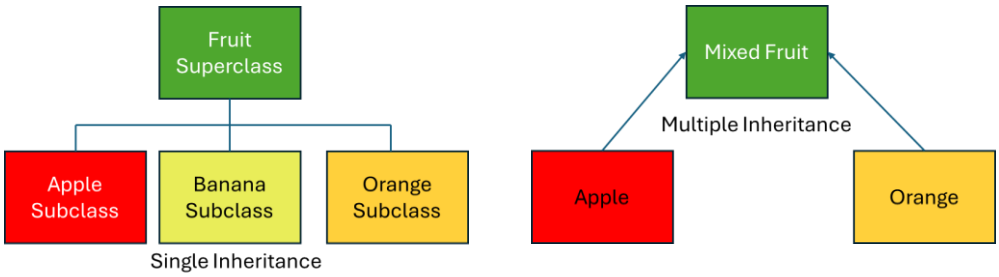
1. **Consistency:** Every language class is guaranteed to have conjugation functionality
2. **Flexibility:** Each language implements conjugation according to its own grammar rules
3. **Code Reuse:** Common functionality like capitalization is inherited, not duplicated
4. **Extensibility:** Adding new languages (French, German, etc.) is easy - just extend Language and implement Conjugation()

The abstract class essentially creates a "contract" that says "if you want to be a Language in this system, you must be able to conjugate verbs, but you can do it however your language requires."

7.9.4 Java Interfaces

In Java, while a class cannot directly inherit from multiple superclasses (a concept known as multiple inheritance of implementation), Java does offer several other forms of inheritance:

- Single Inheritance: A class inherits from only one direct superclass.
- Multilevel Inheritance: A class inherits from a superclass, which in turn inherits from another superclass, forming a chain.
- Hierarchical Inheritance: Multiple subclasses inherit from a single superclass



The restriction against multiple inheritance of implementation in Java is a design choice aimed at preventing complex issues like the "diamond problem," where a class inherits ambiguous methods from multiple paths in the inheritance hierarchy.

However, to achieve the benefits of multiple inheritance in terms of behavior or type, Java provides **interfaces**. An interface is a blueprint of a class. It can have abstract methods and constants. A class can implement multiple interfaces, thereby inheriting the abstract methods defined in each interface. This allows a class to declare that it will provide the functionality specified by multiple distinct contracts.

7.9.5 Interface & Abstract

When designing complex Java software, developers often combine abstract classes and interfaces to create flexible, scalable, and maintainable applications. Abstract classes are used to provide a shared base implementation and to capture common behavior among closely related classes, while interfaces define clear contracts that can be implemented by any class, regardless of its inheritance hierarchy. This dual approach allows for both code reuse (via abstract classes) and multiple inheritance of type (via interfaces), enabling robust object-oriented design patterns that are adaptable to evolving requirements. By leveraging the strengths of both constructs, Java applications achieve a balance between structure and flexibility, making them easier to extend, test, and maintain.

Below is an example of the Translatable interface:

```
// Interfaces define capabilities that languages might have

interface Translatable {
    String translate(String text, String targetLanguage);
}
```

What it means: "Any class that implements this interface PROMISES it can translate text from one language to another."

```
interface Speakable {
    void speak(String text);
    void setAccent(String accent);
}
```

What it means: "Any class that implements this interface PROMISES it can be spoken aloud and can have different accents."

```
interface Writable {
    void write(String text);
    String getWritingSystem(); // Latin, Cyrillic, Arabic, etc.
}
```

What it means: "Any class that implements this interface PROMISES it has a writing system and can write text.

7.9.6 The Abstract Class - Common Foundation

Once we have the interfaces “contracts” declared, we can use a common abstract class like Language as the common Foundation.

What this abstract class provides:

- **Common data:** Every language has a name and region
- **Common behavior:** All languages can capitalize words the same way
- **Required behavior:** All languages MUST implement conjugation (but each does it differently)

```java
// Abstract class for common language features
abstract class Language {
    protected String name;
    protected String region;

    public Language(String name, String region) {
        this.name = name;
        this.region = region;
    }

    // Abstract method - each language conjugates differently
    abstract void conjugate(String verb, String person);

    // Concrete method - most languages capitalize similarly
    void capitalize(String word) {
       System.out.println("Capitalizing: " + word.substring(0,1).toUpperCase() + word.substring(1));
    }
}
```

7.9.7 The concrete Implementation – Spanish Class

```java
// Concrete language classes implementing multiple interfaces
class Spanish extends Language implements Translatable, Speakable, Writable, GrammarRules {
```

This line means: **"Spanish IS-A Language AND CAN-DO translation, speaking, writing, and has grammar rules."**

7.9.8 DemoLanguage.java Class

The DemoLanguage class demonstrates **Multiple** Inheritance of Capabilities & Flexibility:

- Spanish can translate AND speak AND write AND has grammar rules,
- each interface adds a specific set of abilities
- Not every language needs every capability
- A dead language might implement Writable and GrammarRules but not Speakable

```
public class DemoLanguage extends  Object{
    public static void main(String[] args) {
        Spanish spanish = new Spanish();

        // Using inherited abstract class methods
        spanish.conjugate("hablar", "yo");      // "Spanish conjugation: yo hablara/e/o"
        spanish.capitalize("hola");             // "Capitalizing: Hola"

        // Using interface methods
        spanish.speak("Hola mundo");   // "Speaking in Spanish with neutral accent: Hola mundo"
        spanish.setAccent("Mexican");
        spanish.speak("Hola mundo");   // "Speaking in Spanish with Mexican accent: Hola mundo"

        System.out.println(spanish.translate("Hello", "French"));
                                       // "Translating 'Hello' from Spanish to French"
        spanish.write("Buenos días");       // "Writing in Spanish: Buenos días"

        System.out.println("Has gender: " + spanish.hasGender());        // "Has gender: true"
        System.out.println("Tenses: " + spanish.getVerbTenses());          // "Tenses: 14"
        System.out.println("Word order: " + spanish.getWordOrder());       // "Word order: SVO"
    }
}
```

7.9.9 Comparable Interface

The Java Comparable interface allows a class to define how its objects should be compared and sorted, establishing what is known as the class's "natural ordering". This is vital for sorting custom objects in collections or arrays using Java's built-in sort mechanisms, such as Arrays.sort() or Collections.sort().

The Comparable interface belongs to the java.lang package and contains a single method, compareTo(T obj), which compares the calling object with the specified object. The method returns:

- A negative integer if the calling object is less than the specified object.
- Zero if both are considered equal.
- A positive integer if the calling object is greater than the specified object.

Why Do We Need Comparable?

- Implementing Comparable allows objects to be sorted according to a natural order defined by the class developer (for example, alphabetically by name or numerically by ID).
- Java's built-in sorting methods can automatically sort objects that implement Comparable, eliminating the need for custom sorting code each time.
- It is especially useful for collections like lists or arrays where ordering is desirable, making data management and retrieval more efficient.

Key Points

- Only one natural ordering can be specified by Comparable in a class.
- Common Java classes like String and Integer already implement Comparable, which is why they can be easily sorted.
- For different sorting logic (not just natural order), use the separate Comparator interface.

In summary, the Comparable interface provides a standard way for objects to be compared and sorted, which is fundamental for working with collections in Java.

Module 7: Learning Materials & References

a) Java OOP Inheritance
b) Inheritance in Java
c) Using Java Abstract Classes
d) Abstract Class vs Interface

Part A: Quiz

7.1a Which statement about Java inheritance is TRUE?

A) A subclass can inherit from multiple parent classes using the extends keyword
B) Private members of a parent class are directly accessible in the subclass
C) A subclass inherits all non-private members from its parent class
D) The super() call must always be the last statement in a constructor

7.2a In UML class diagrams, inheritance is represented by:

A) A solid line with a filled diamond at the parent class end
B) A solid line with an open triangle (arrow) pointing to the parent class
C) A dashed line with an open triangle pointing to the parent class
D) A solid line with no special symbols

7.3a What happens when a subclass overrides a parent class method in Java?

A) The parent class method is completely deleted from memory
B) Both methods exist, and Java randomly chooses which one to call
C) The subclass method hides the parent method, but the parent method can still be called using super
D) A compilation error occurs because methods cannot have the same name

7.4a Which of the following is NOT a primary benefit of inheritance in object-oriented programming?

A) Code reusability - common functionality can be written once in the parent class
B) Polymorphism - objects of different subclasses can be treated uniformly
C) Faster program execution - inheritance always makes programs run faster
D) Maintainability - changes to common functionality only need to be made in one place

7.5a What must be added to the Car class for this code to compile successfully?

```
abstract class Vehicle {
    abstract void start();
    void stop() { System.out.println("Vehicle
stopped"); }
}
class Car extends Vehicle {
    // Missing implementation
}
```

A) A constructor that calls `super()`
B) An override of the `stop()` method
C) Nothing - the code will compile as is
D) An implementation of the `start()` method

7.6a What must be added to the Circle class for this code to compile successfully?

```
abstract class Shape {
    abstract void draw();
    abstract void color();
}
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing circle");
    }
    // Missing something
}
```

A) It needs a constructor
B) The draw() method should be public
C) It's missing the color() method
D) It cannot extend an abstract class

7.7a What will happen when you try to compile this code?

```
abstract class Animal {
    abstract void eat();
    abstract void sleep();
}
abstract class Bird extends Animal {
    void eat() {
        System.out.println("Bird eating seeds");
    }
    abstract void fly();
}
class Eagle extends Bird {
    void fly() {
        System.out.println("Eagle soaring high");
    }
}
```

A) Compilation error - Eagle is missing required method implementations

B) Compilation error - Eagle cannot extend Bird

C) Compilation error - Bird cannot extend Animal

D) Code compiles successfully

7.8a What's the problem with the Motorcycle class?

```
abstract class Vehicle {
    protected String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }
    abstract void start();
    void displayBrand() {
        System.out.println("Brand: " + brand);
    }
}
class Motorcycle extends Vehicle {
    void start() {
        System.out.println("Motorcycle started with kick");
    }
}
```

A) It cannot override the abstract `start()` method

B) It cannot access the `brand` field from Vehicle

C) Abstract classes cannot have constructors

D) It's missing a constructor that calls the parent constructor

7.9a Which statement about this code is TRUE?

```
abstract class Animal {
    String name;
    abstract void makeSound();
    void breathe() {
        System.out.println("Breathing");
    }
}

interface Swimmable {
    void swim();
}

interface Climbable {
    void climb();
}

class Monkey extends Animal implements Swimmable,
Climbable {
    void makeSound() {
        System.out.println("Ooh ooh ah ah");
    }
```

A) This code won't compile because a class cannot implement multiple interfaces

B) This code won't compile because interfaces cannot be implemented along with inheritance

C) This code compiles successfully and demonstrates multiple inheritance of behavior

D) The Monkey class should be declared abstract since it extends an abstract class

```
    public void swim() {
        System.out.println("Monkey swimming");
    }

    public void climb() {
        System.out.println("Monkey climbing trees");
    }
}
```

7.10a What must be added to make the Helicopter class compile successfully?

```
abstract class Vehicle {
    protected String brand;
    abstract void start();
    void stop() {
        System.out.println("Vehicle stopped");
    }
}
interface Flyable {
    void fly();
    void land();
}
class Helicopter extends Vehicle implements Flyable {
    public Helicopter(String brand) {
        super(brand);
    }
    void start() {
        System.out.println("Helicopter engine started");
    }
    public void fly() {
        System.out.println("Helicopter flying");
    }
    // Missing something
}
```

A) Override the stop() method from Vehicle

B) Add a default constructor

C) Make the start() method public

D) Implement the land() method from Flyable interface

Part B: Reflection Prompt

Short answers:

7.1b How would you decide what should be an abstract class versus an interface in this system?

7.2b Does Java support multiple inheritance? Explain briefly.

Part C: Lab Assignments

7.1c: Vehicle.java class

Create a class hierarchy like below:

Vehicle

- Boat
- Truck
- Jet

Vehicle should have a method move(), start(), stop().
Boat, Truck, Jet should override these methods with specific messages.
DemoVehicles.java Application:

1. Create an array of Vehicles type Boat, Jet, Truck
2. Call move(), start(), stop for each.

Module 8 Collections and Generic

Java Collections and Generics are foundational concepts in Java programming, designed to provide type safety, reduce code duplication, and improve code clarity and maintainability. Here's a detailed narration of how generics are used in the Java Collections Framework.

8.1. Introduction to Collections Framework

The Java Collections Framework (JCF) is a unified architecture for storing and manipulating groups of objects. It is part of the java.util package and provides both interfaces and concrete implementations (classes) to handle common data structures efficiently.

Before the Collections Framework was introduced, Java developers had to rely on arrays or custom data structures. Arrays are fixed in size and lack flexibility, while custom structures were error-prone and inconsistent.

The JCF addresses these limitations by providing:

- Standardization: A common interface for all collections simplifies code reuse and maintenance.
- Efficiency: Well-tested and optimized implementations of data structures like lists, sets, and maps.
- Interoperability: Algorithms (like sorting and searching) can operate on various collection types in a uniform way.
- Polymorphism: Programs can be written using interfaces rather than concrete classes, allowing flexibility in choosing or changing implementations.
- Thread Safety: Some implementations offer synchronized (thread-safe) versions.

8.2 What Are Generics?

Generics allow classes, interfaces, and methods to operate on types specified by parameters, rather than specific types. This means you can write a single class or method that works with many different types, while still ensuring type safety at compile time

Before generics, Java collections could hold any type of object. For example, you could add both a String and an Integer to the same ArrayList, which often led to runtime errors when retrieving and casting objects. Generics solve this problem by allowing you to specify the type of objects a collection can contain.

Generics enforce type safety at compile time. If you try to add an object of the wrong type to a collection, the compiler will generate an error. This prevents ClassCastException at runtime, which would otherwise occur if you accidentally cast an object to the wrong type

8.2.a Common Generic Collections

The Java Collections Framework provides several generic interfaces and classes, such as:

- **List:** Ordered collection that allows duplicates (e.g., ArrayList<E>, LinkedList<E>)
- **Set:** Unordered collection that does not allow duplicates (e.g., HashSet<E>, TreeSet<E>)
- **Map:** Key-value pairs (e.g., HashMap<K,V>, TreeMap<K,V>)

Example:

```
List<String> list = new ArrayList<>();
list.add("S");
list.add("B");

// Sort the list in ascending (natural) order
Collections.sort(list);

// Sort the list in descending order Large - Small
Collections.sort(list, Collections.reverseOrder());

Set<Integer> set = new HashSet<>();
set.add(1);
set.add(1); // Duplicate ignored
set.add(11);
for (var item : set) {
      System.out.println(item);
  }
Map<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");

if (map.containsKey("Three")) {
      System.out.println("Key 'Three' exists in the map.");
}
```

8.2.b Wildcard Generics

What is **?** in Java generics? The **?** is called wildcard. It stands for "some unknown type".

There are 3 common wildcards:

Syntax	Meaning	Can	Cannot	Example
<?>	Unbounded wildcard: any type	`Read as Object`	`Write (except null)`	`List<?> list`
<? extends T>	Upper bounded wildcard: any subclass of `T` (including `T`)	`Read as T`	`Write(except null)`	`List<? extends Number>` (can hold `Integer`, `Double`, etc.)
<? super T>	Lower bounded wildcard: any superclass of `T` (including `T`)	`Write T or subclasses`	`Read as Object`	`List<? super Integer>` (can hold `Integer`, `Number`, `Object`)

The key takeaways from this table is the **PECS Principle**: "Producer Extends, Consumer Super"

8.2.1a Unbounded wildcard example:

What `List<?> list` means: A list of unknown type or a list of Object. The below method will accept (List<String>, List <Integer>, etc.), essentially <?> will accept any Object.

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

8.2.1b Upper Bounded Wildcard example:

What `List<? extends Number> list` means: "A list of some unknown type that extends Number" The ? is a wildcard that represents any type that is Number or a subclass of Number. The below method will accept List<Integer>, List<Double>, List<Float>, etc. Use extends when the collection is a **producer** (reading from it, but write to i)

```
public double sumList(List<? extends Number> list) {
    double sum = 0.0;
```

```
    for (Number num : list) {
        sum += num.doubleValue();
    }
    return sum;
}
```

8.2.1c Lower Bounded Wildcard example:

What `List<? super Integer>` means: "A list of some type that is Integer or a supertype of Integer", the method below will accept <List Integer>, List<Number>, List<Object>.

```
public void addIntegers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
}
```

Why Use Wildcards?

- For **flexibility** in APIs: accept broader or more generic inputs.
- For **readability and safety**: enforce what types can be read or written.

Just remember that **PECS Principle**: "Producer Extends, Consumer Super"

8.3 Generic Classes and Methods

In addition to generic methods, classes can also be defined as generic. A generic class includes a type parameter, typically written as `<T>` by convention, which serves as a placeholder for a specific type specified at instantiation. However, different type parameter names are often used for clarity based on context—for example, `<E>` for elements in a collection, and `<K>` and `<V>` to represent keys and values in a map or hash map.

Example of a Generic Class:

```
class Box<T> {
    private T content;
    public void set(T content) { this.content = content; }
    public T get() { return content; }
}

/** Box data type in this instance is of class type Integer*/
Box<Integer> intBox = new Box<>();
```

```
intBox.set(10);
int value = intBox.get(); // No casting needed

/** Box data type in this instance is of class type String*/
Box<String> strBox = new Box<>();
strBox.set("Hello");

/** Box data type in this instance is of class type Double*/
Box<Double> doubleBox = new Box<>();
doubleBox.set(3.14);
```

Generic methods are methods that introduce their own type parameters. This allows the method to operate on any type, or on types that satisfy certain constraints, rather than being limited to specific types. Generic methods provide flexibility and type safety by letting you write a single method that can be called with arguments of different types

Example of a Generic Methods:

```
public <T> void Print(T data) {
    System.out.println("Data: " + data);
}
// Print size of any list, regardless of element type
public void printSize(List<?> list) {
    System.out.println("List size: " + list.size());
}

// Usage:

printSize(Arrays.asList("a", "b"));     // Type inferred as List<String>
printSize(Arrays.asList(1, 2, 3.14));   // Type inferred as List<Number>
printSize(Arrays.asList(1, 2, 3));      // Type inferred as List<Integer>
printSize(Arrays.asList(new Person()));      // Type inferred as List<Person>

Print("Hello");       // Type inferred as String
Print(123);           // Type inferred as Integer
Print(3.14);          // Type inferred as Double
```

Module 8: Learning Materials & References

a) https://www.protechtraining.com/bookshelf/java_fundamentals_tutorial/generics_collections
b) https://docs.oracle.com/javase/tutorial/java/generics/methods.html
c) https://www.tutorialspoint.com/java/java_generics.htm
d) https://www.tutorialspoint.com/java/java_collections.htm

Part A: Quiz

8.1a What is the primary purpose of Java generics?

A) To improve runtime performance by avoiding boxing/unboxing
B) To provide type safety at compile time and eliminate the need for casting
C) To enable multiple inheritance in Java classes
D) To automatically manage memory allocation for collections

8.2a Which wildcard should you use when you need to READ items from a collection but NOT add new items?

A) <? super T> - lower bounded wildcard
B) <?> - unbounded wildcard
C) <? extends T> - upper bounded wildcard
D) <T> - type parameter

8.3a What is the main difference between ArrayList<T> and LinkedList<T> in terms of use cases?

A) ArrayList is thread-safe while LinkedList is not
B) ArrayList is better for frequent insertions/deletions in the middle, LinkedList is better for random access
C) ArrayList provides O(1) random access, LinkedList is better for frequent insertions/deletions
D) ArrayList only works with primitive types, LinkedList works with objects

8.4a When would you use a Set<T> collection instead of a List<T>?

A) When you need to maintain insertion order of elements
B) When you need to allow duplicate elements in the collection
C) When you need to ensure uniqueness of elements and don't care about order
D) When you need indexed access to elements by position

8.5a What will be the output of the following code?

```
List<? extends Number> numbers = Arrays.asList(1, 2.5, 3L);
numbers.add(4);
System.out.println(numbers.size());
```

A) 4
B) 3
C) Compilation error
D) Runtime exception

8.6a What will be the output of the following code?

```
public class GenericExample {
    public static <T> void swap(List<T> list, int i, int j) {
        T temp = list.get(i);
        list.set(i, list.get(j));
        list.set(j, temp);
    }

    public static void main(String[] args) {
        List<String> words = new ArrayList<>();
        words.add("Apple");
        words.add("Banana");
        words.add("Cherry");

        swap(words, 0, 2);
        System.out.println(words.get(1));
    }
}
```

A) Apple
B) Banana
C) Cherry
D) Compilation error

8.7a Which of the following statements about this code is correct?

```java
Map<String, List<Integer>> map = new HashMap<>();
map.put("evens", Arrays.asList(2, 4, 6));
List<? super Integer> list = map.get("evens");
```

A) Compilation error - cannot assign List<Integer> to List<? super Integer>
B) Code compiles and runs successfully
C) Runtime exception will occur
D) The wildcard is unnecessary here

8.8a Which statement about this code is true?

```java
List<Integer> integers = Arrays.asList(1, 2, 3);
List<Number> numbers = integers;  // Line A
numbers.add(4.5);                 // Line B
```

A) Code compiles and runs successfully
B) Compilation error at Line A
C) Compilation error at Line B
D) Runtime exception at Line B

8.9a What is the result of this generic method called?

```java
public class CollectionUtils {
    public static <T extends Comparable<T>> boolean isSorted(List<T> list) {
        for (int i = 0; i < list.size() - 1; i++) {
            if (list.get(i).compareTo(list.get(i + 1)) > 0) {
                return false;
            }
        }
        return true;
    }
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 3, 2, 4, 5);
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        System.out.println(isSorted(numbers) + " " + isSorted(names));
    }
}
```

A) true true
B) false true
C) true false
D) false false

8.10a What will be the output of the following code?

```
public class SortExample {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>();
        words.add("banana");
        words.add("Apple");
        words.add("cherry");
        words.add("Date");

        Collections.sort(words);
        System.out.println(words.get(0) + " " + words.get(3));
    }
}
```

A) Apple Date
B) banana cherry
C) Apple banana
D) Date cherry

8.11a What will be the output of the following code?

```
public class CustomSort {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(15, 3, 9, 1, 12);

        Collections.sort(numbers, new Comparator<Integer>() {
            @Override
            public int compare(Integer a, Integer b) {
                return b - a;
            }
        });

        System.out.println(numbers.get(0) + " " +
numbers.get(numbers.size()-1));
    }
}
```

A) 1 15
B) 15 1
C) 3 12
D) Compilation error

Part B: Reflection Prompt

8.1b Explain One Benefit of Using Java Generics, and one drawback.

8.2b What Java Would Look Like Without Collections?

Part C: Lab Assignments

8.1.c Create Java method filterShortWords

Write a Java method called filterShortWords that takes a List<T> and returns a new List<T> containing only items with 3 or fewer. Use generics and collections.

8.2.c Create Java method countFrequencies

Write a Java method called countFrequencies that take a List<T> items and return a frequency map (how many times a particular word appears in the list).

Module 9 Recursion

Recursion is a programming technique where a method or function calls itself to solve a problem. This approach is often compared to Russian dolls nested within each other, mirrors reflecting into infinity, or fractals that repeat the same pattern at every scale. Unlike traditional loops (iteration), recursion solves complex problems by breaking them down into smaller, self-similar sub problems.

One of the main reasons recursion is so useful is its ability to provide elegant and concise solutions for certain types of problems. Many mathematical concepts, such as calculating factorials or generating Fibonacci numbers, are naturally defined in recursive terms. Additionally, recursion simplifies the code needed to work with complex data structures like trees and graphs, making it easier to write and understand algorithms that traverse or manipulate these structures.

A recursive method typically consists of two key components: the base case and the recursive step. The base case is a condition that stops the recursion, preventing it from continuing indefinitely and causing a stack overflow error. The recursive step is where the method calls itself, usually with a modified input that brings it closer to the base case. By carefully designing both components, recursion can be a powerful and efficient tool in a programmer's toolkit.

Example of Recursion in Java:

```
/** Define a class to contain our recursive example method. */
public class RecursiveSum extends Object {
    /**
     * Calculates the sum of all integers from 1 up to 'n' using recursion.
     * @param n The upper limit of the sum. Must be a non-negative integer.
     * @return The sum of integers from 1 to n.
     */
    public static int sumUpToN(int n) {
        // --- Base Case ---
        // If n is 0, the sum is 0. This is the stopping condition for the recursion.
        if (n == 0) {
            return 0;
        }

        // --- Recursive Step ---
        return n + sumUpToN(n - 1);
    }
```

```
    // Main method to demonstrate the recursive function.
    public static void main(String[] args) {
        // Test cases for the recursive sumUpToN method.
        int number1 = 5;
        System.out.println("Sum up to " + number1 + ": " + sumUpToN(number1));
        // Expected: 15 (5+4+3+2+1)

        int number2 = 10;
        System.out.println("Sum up to " + number2 + ": " + sumUpToN(number2)); // Expected: 55

    }
}
```

Explanations:

sumUpToN(int n) Method:

1. **Base Case (if (n == 0)):** This is the crucial stopping condition for the recursion. When `n` becomes 0, the method returns `0`, preventing an infinite loop. Without a base case, the method would keep calling itself, leading to a StackOverflowError.
2. **Recursive Step (return n + sumUpToN(n - 1);):** This is where the magic of recursion happens. The method returns the current value of `n` plus the result of calling `sumUpToN` with `n-1`. This breaks down the problem into smaller, identical subproblems until the base case is hit, and then the results are combined back up the call stack.

main(String[] args) Method:

3. This is the entry point of the program. It serves to test the `sumUpToN` method with various integer inputs (5, 10) and prints their respective sums to the console, demonstrating how the recursive function works.
4. What would happen if a negative input were given without proper validation, leading?

9.2 Definition: What is Recursion?

- Definition: Recursion is a method (or function) that calls itself.
- Analogy: Like Russian dolls nested inside each other, mirrors reflecting mirrors, or fractals repeating patterns.
- Contrast: Unlike iteration (loops), recursion solves problems by breaking them into smaller, similar sub problems.

The factorial function is a classic example used to illustrate recursion in programming. In mathematics, the factorial of a number **n** as **n!** or

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

For example:

1) $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
2) $3! = 3 \times 2 \times 1 = 6$
3) $1! = 1$
4) 0! = 1 by definition

9.2a Recursive Definition of factorial

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1), & if\ n > 0 \end{cases}$$

```
long factorial (int n) {
  if (n == 0 || n == 1) {b
      return 1; // base case
  } else
      return n * factorial (n - 1); // recursive step
}
```

For example, calling factorial (3) leads to the following sequence:

- factorial (3) returns 3×factorial (2)3×factorial (2)
- factorial (2) returns 2×factorial (1)2×factorial (1)
- factorial(1) returns 1 (base case)
  The results are then combined: 2×1=22×1=2, then 3×2=63×2=6,
  so factorial(3) returns 6

9.2b. Why is Recursion Useful?

Recursive solutions are often Elegant and concise for problems that can be broken down into smaller, self-similar sub problems. Often mirrors mathematical definitions (e.g., factorial, Fibonacci). It is often simplifying code for certain data structures (e.g., trees, graphs).

9.2c. Key Components of a Recursive Method

1. Base Case: The condition that stops the recursion; essential to prevent infinite loops (StackOverflowError).

2. Recursive Step: The part of the method that calls itself, typically with a modified input that moves closer to the base case.

Module 9: Learning Materials & References

a) Java Recursion
b) Everything about Java Recursion
c) Recursion in Java Full Tutorial Video

Part A: Quiz

9.1a. What is the base case in a recursive function?

A) The part of the code that prints the result
B) The condition where the function calls itself
C) The condition that ends the recursive calls
D) The first value passed to the function

9.2a. Which of the following is true about recursion?

A) Recursion uses less memory than iteration
B) Every recursive function must have a base case
C) Recursion is always faster than loops
D) Recursive functions do not use the call stack

9.3b What does the following method compute?

```
int compute(int n) {
     if (n == 1) return 1;
     return n * compute(n - 1);
 }

  System.out.println(compute(4));
```

A) 10
B) 24
C) 16
D) 5

9.4b What happens if a recursive method lacks a proper base case?

A) The program will run infinitely
B) The compiler will throw an error

C) The function will execute only once
D) It will automatically stop after 10 calls

9.5b Recursive function calls are stored in the _____, which allows the program to return to the previous state after each call finishes.

A) Heap
B) Pool
C) Stack
D) Frame

9.6b. A recursive method typically includes two parts: the base case and the _____ case.

A) Primary
B) Secondary
C) Base
D) Recursive

Part B: Reflection Prompt

9.1b When solving a problem, how do you decide whether to use recursion or iteration? Give an example.

9.2b Can every recursive function be rewritten iteratively? What are the trade-offs when doing so?

9.3b How does memory usage differ between recursion and iteration?

Part C: Lab Assignments

9.1c Implement Factorial Both Ways

Write **two methods** in Java: calculate the sum of numbers from 1 to n

Recursive method.
Iterative method.

Method signatures:

```
long sumRecursive(int n)
long sumIterative(int n)
```

9.2c No base case

Rewrite 9.1c recursive with the base case. Invoke the method with the following parameter, `sumRecursive(Integer.MAX_VALUE)` and observe the results, then write up your observations.

Module 10 Java Exceptions

10.1 What is a Java Exception?

A Java exception is an event that occurs during the execution of a program which disrupts the normal flow of instructions. When an error or an unexpected event occurs—such as dividing by zero, accessing an invalid array index, or opening a non-existent file—Java generates an exception object and throws it. If not handled, the program will terminate abruptly, and the code after the exception will not execute.

Common examples of exceptions in Java include:

- `ArithmeticException` (e.g., division by zero)
- `ArrayIndexOutOfBoundsException`
- `NullPointerException`
- `FileNotFoundException`

10.2 Try-Catch

Java provides structured exception handling using the `try` and `catch` blocks:

Syntax

```
try {
    // Code that may cause an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

Example: Handling Array Index Exception

The following code prevents the program from crashing by catching and handling the exception

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]); // This causes an exception
        } catch (Exception e) {
            System.out.println("Something went wrong.");
        }
    }
}
```

Output: `Something went wrong.`

10.3 The finally Block

Optional `finally` block can be included that will run regardless of whether an exception is caught:

```
try {
    // risky code
} catch (Exception e) {
    // handle exception
} finally {
    // always executed
}
```

10.4 Throwing and Catching Multiple Exceptions

Instead of waiting for the exception to occur, a program throws an exception. A single method can declare that it might throw more than one exception type using the `throws` keyword:

In the code fragment below method readFile() can expected to throw two exceptions, IO, and SQL.

```
public void readFile() throws IOException, SQLException {
    // code that might throw either exception
}
```

However, a method can only throw **one exception instance at a time**. To encapsulate multiple exceptions, use custom aggregation or the `addSuppressed()` method on `Throwable`.

Similar to throw exceptions, Java allows a single `catch` block to handle multiple exception types. The code fragment below can catch IOException OR SQL Exception:

```
try {
    // code that may throw multiple exceptions
} catch (IOException | SQLException ex) {
    // handle both exception types
}
```

This approach reduces code duplication when multiple exception types require similar handling.

10.5 Tracing Exceptions Through the Call Stack

When an exception is thrown, Java generates a **stack trace**. This is a sequence of method calls that shows how the program arrived at the point where the exception occurred. The stack trace is invaluable for debugging.

Example Call Stack Output:

```
Exception in thread "main" java.lang.RuntimeException: A test exception
  at Example.methodB(Example.java:13)
  at Example.methodA(Example.java:9)
  at Example.main(Example.java:5)
```

The stack trace lists the most recent method first, and each line shows the file and line number. You can print the stack trace in Java using:

```
catch (Exception e) {
    e.printStackTrace();
}
```

This helps you trace the flow of execution that led to the error.

10.6 Creating Your Own Exception Classes

Custom exceptions can make your code clearer by signaling error conditions unique to your application.

Steps to Create a Custom Exception

1. Extend the Exception class (for checked exceptions) or RuntimeException (for unchecked exceptions)
2. Provide Constructors:
    - One that accepts a message.
    - Optionally, another that accepts a message and cause.

Example: Custom Checked Exception

```
public class IncorrectFileNameException extends Exception {
    public IncorrectFileNameException(String errorMessage) {
        super(errorMessage);
    }
    public IncorrectFileNameException(String errorMessage, Throwable err) {
        super(errorMessage, err);
    }
}
```

Usage:

```
public void loadFile(String fileName) throws IncorrectFileNameException {
    if (!isValid(fileName)) {
        throw new IncorrectFileNameException("Invalid file name: " +
fileName);
    }
    // Continue loading
}
```

This makes your application's error handling highly specific and meaningful.

By understanding and using exception handling, you create robust, error-tolerant Java applications that behave predictably even in exceptional situations.

Module 10: Learning Materials & References

a) Exception Handling in Java Tutorial Video
b) What Is an Exception?
c) Lesson: Exceptions
d) Java Exception Handling
e) Java Exceptions - Try...Catch
f) Java try...catch
g) Java Try Catch Block

Part A: Quiz

10.1a What happens when an exception is thrown but not caught in a Java program?

A) The program continues executing the next line after the exception
B) The program terminates and prints a stack trace
C) The exception is automatically converted to a warning
D) The JVM ignores the exception and logs it silently

10.2a Which of the following statements about checked exceptions is TRUE?

A) Checked exceptions extend RuntimeException
B) Checked exceptions can only be thrown in main() methods
C) Checked exceptions must be either caught or declared in the method signature
D) Checked exceptions are automatically handled by the JVM

10.3a What is the output of the following code?

```
try {
    int x = 10 / 0;
    System.out.println("A");
} catch (ArithmeticException e) {
    System.out.println("B");
} finally {
    System.out.println("C");
}
System.out.println("D");
```

A) B C D
B) A C D
C) A B C D
D) B C

10.4a Which exception type should you extend to create a custom unchecked exception?

A) Exception
B) Throwable
C) RuntimeException
D) Error

10.5a In a try-catch block with multiple catch statements, what determines which catch block executes?

A) The catch block that appears first in the code
B) The catch block with the most specific exception type that matches the thrown exception
C) The catch block that appears last in the code
D) All catch blocks execute in sequence

Part C: Lab Assignments

10.1c Bank Account Exception Handling

Write a Java program that implements a BankAccount class with the following requirements:

1. Create a custom checked exception called InsufficientBalanceException
2. The BankAccount class should have:
    - Private fields: accountNumber (String) and balance (double)
    - Constructor that initializes both fields
    - Method withdraw(double amount) that throws InsufficientBalanceException if withdrawal amount exceeds balance
    - Method deposit(double amount) that throws IllegalArgumentException if amount is negative
    - Getter methods for both fields
3. In the main method, demonstrate handling both types of exceptions with appropriate error messages.

10.2c: File Processing with Exception Chaining

Write a Java program that reads integers from a file and calculates their average. The program should:

1. Create a custom exception class called DataProcessingException that can wrap other exceptions
2. Create a method processFile(String filename) that:
    - Reads integers from a file (one per line)
    - Returns the average as a double
    - Handles FileNotFoundException, NumberFormatException, and IOException
    - Wraps any caught exceptions in your custom DataProcessingException
3. In the main method:
    - Call processFile() with both a valid file and an invalid filename
    - Handle the DataProcessingException and print both the custom message and the original cause
4. Create a test file with some integers and one invalid entry to demonstrate the exception handling.

5. Include a finally block that prints "File processing completed" regardless of success or failure.

numbers.txt (valid file):	invalid_numbers.txt (file with invalid data):
10	10
20	20
30	abc
40	40
50	50

Sample Output

```
=== Testing with valid file ===
Successfully processed 5 numbers from numbers.txt
File processing completed
Average: 30.0
=== Testing with invalid file ===
File processing completed
Processing failed: File not found: nonexistent.txt
Root cause: FileNotFoundException - nonexistent.txt (No such file or directory)

=== Testing with file containing invalid data ===
File processing completed
Processing failed: Invalid number format at line 3: 'abc'
Root cause: NumberFormatException - For input string: "abc"
```

Module 11 Input/output

Java provides robust APIs for reading and writing files, making it possible to exchange data between your programs and the outside world in a variety of formats. File I/O (Input/Output) is a key skill for any developer, enabling persistent storage, report generation, and data analysis tasks

11.1 Path & Files Classes

Java's modern approach to file management utilizes the Path and Files classes from the java.nio.file package, introduced in Java 7.

Path

- Represents: The location of a file or directory in the file system.
- Creation:
  ```
  Path path = Paths.get("C:/example/data.txt");
  ```
- Features: Manipulate file and directory paths, resolve relative and absolute paths, traverse directory trees.

Files

- Features: Contains static methods for file and directory operations such as creation, deletion, reading, and writing.

  ```
  Files.createFile(path);
  Files.deleteIfExists(path);
  Files.readAllLines(path);
  ```

Both APIs are used for file/directory management, but java.nio.file.Path and Files are preferred for newer applications due to their versatility and performance.

11.2. Streams and Buffers

In Java, I/O is based on the concept of streams, which are sequences of data. Think of them as a pipe connecting your program to a data source (input) or a data destination (output).

11.2a Streams

A stream is a continuous flow of data from a source to a destination. There are two primary categories of streams:

- Byte Streams: Deal with raw binary data. Examples: InputStream, OutputStream, FileInputStream, FileOutputStream.
- Character Streams: Deal with text data (handling character encoding). Examples: Reader, Writer, FileReader, FileWriter.

Example: Reading With a Byte Stream

```
FileInputStream input = new FileInputStream("data.bin");
int data = input.read();
input.close();
```

11.2b Buffers

Buffers increase efficiency by storing input or output data in memory before processing it. Buffering is the practice of wrapping a stream with a buffer. Instead of reading or writing one byte/character at a time, data is read/written in large chunks into a temporary memory buffer. This significantly reduces the number of interactions with the underlying operating system and boosts performance.

Buffered Streams (BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter): Help reduce the number of direct I/O operations and improve performance.

Example:

BufferedReader reader = **new** BufferedReader(**new** FileReader("input.txt"));

```
String line = reader.readLine();
reader.close();
```

Advantage: Fewer system calls and faster I/O operations, especially for large files.

11.3 Java's IO Classes Overview

Java's IO API is centered on several abstract base classes and their concrete implementations for different data and use cases:

Type	Abstract Base Class	Examples (Common Classes)	Use Case
Byte Input	InputStream	FileInputStream, BufferedInputStream	Binary data (images, files)
Byte Output	OutputStream	FileOutputStream, BufferedOutputStream	Writing binary data
Character Input	Reader	FileReader, BufferedReader	Text files (human-readable data)
Character Output	Writer	FileWriter, BufferedWriter	Writing text data

Common IO Methods for different tasks:

1. PrintWriter: Easily write formatted text.
2. DataInputStream/DataOutputStream: For reading and writing primitive data types.
3. RandomAccessFile: Read and write file contents at arbitrary locations (not just sequentially).

11.4 Sequential Data Files

Sequential files store data records one after another. They are read or written from beginning to end.

11.4a Writing Sequential Data

- Open a file for writing (creates the file if it doesn't exist).
- Use PrintWriter, BufferedWriter, or FileWriter for writing text.
- Close the file after writing.

Example:

```
PrintWriter out = new PrintWriter(new FileWriter("output.txt"));
out.println("Alice,25");
out.println("Bob,30");
out.close();
```

11.4b Reading Sequential Data

Use Scanner, BufferedReader, or FileReader.

Example:

```
Scanner reader = new Scanner(new File("output.txt"));
while (reader.hasNextLine()) {
    String line = reader.nextLine();
    // Process line
}
reader.close();
```

This approach is ideal when all records must be processed, and file order is important.

Module 11: Learning Materials & References

a) Java File Input/Output - It's Way Easier Than You Think
b) Java IO - Input/Output in Java with Examples
c) Java Files - java.nio.file.Files Class
d) https://bimstudies.com/docs/object-oriented-programming-with-java/essential-java-classes/i-o-classes-and-interfaces/
e) Java: Iterating Streams Using Buffers
f) Lesson: Basic I/O

Part A: Quiz

11.1a What is the primary advantage of using buffered streams (BufferedReader/BufferedWriter) over regular streams (FileReader/FileWriter)?

A) Buffered streams can read binary data while regular streams cannot
B) Buffered streams automatically handle character encoding
C) Buffered streams reduce the number of system calls by reading/writing data in chunks
D) Buffered streams can only be used with text files

11.2a Which of the following is the correct way to create a BufferedReader for reading from a file?
A) BufferedReader br = new BufferedReader("filename.txt");
B) BufferedReader br = new BufferedReader(new FileReader("filename.txt"));
C) BufferedReader br = new FileReader(new BufferedReader("filename.txt"));
D) BufferedReader br = new BufferedReader(new File("filename.txt"));

11.3a What happens when you call flush() on a BufferedWriter?

A) It forces any buffered data to be written to the underlying stream immediately
B) It closes the stream permanently
C) It clears the buffer and discards any unwritten data
D) It resets the buffer size to its default value

11.4a Which of the following code snippets correctly demonstrates reading a file line by line using BufferedReader with proper resource management?

A.
```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
```
B.
```
try (FileReader fr = new FileReader("file.txt")) {
    BufferedReader br = new BufferedReader(fr);
    String line = br.readLine();
    System.out.println(line);
}
```
C.
```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
try {
    String line = br.readLine();
    System.out.println(line);
    } finally {
        br.close();
    }
```
D.
```
try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

Part C: Lab Assignments

11.1c Student Grade Processor

Write a Java program that processes student grades from a text file and generates a summary report. The program should:

1. Read student data from a file called grades.txt where each line contains: StudentName,Math,English,Science (comma-separated values)
2. Create a Student class with:
    a. Name and three subject grades
    b. Method to calculate average grade
    c. Method to determine letter grade (A: 90-100, B: 80-89, C: 70-79, D: 60-69, F: below 60)
3. Use BufferedReader to read the file and BufferedWriter to write results to report.txt
4. The output file should contain:
    - Each student's name, average, and letter grade
    - Class statistics (total students, class average, highest/lowest average)
5. Handle file I/O exceptions appropriately and use try-with-resources for proper resource management.

Sample Input File (grades.txt):

John Smith,85,92,78
Jane Doe,95,88,91
Bob Johnson,72,68,75
Alice Brown,88,94,86

11.2c Log File Analyzer

Write a Java program that analyzes web server log entries and creates a filtered output. The program should:

1. Read from a log file called server.log where each line contains: [TIMESTAMP] LEVEL: MESSAGE Example: [2024-01-15 14:30:25] ERROR: Database connection failed
2. Create an enum LogLevel with values: INFO, WARNING, ERROR, DEBUG
3. Use BufferedReader to read the log file and BufferedWriter to create two output files:

    - errors.log: Contains only ERROR level entries
    - summary.txt: Contains statistics about log entries
4. The summary file should include:
    - Total number of entries for each log level
    - Most recent error message
    - Time range of the log entries (first and last timestamps)
5. Use proper exception handling and ensure all resources are properly closed.
6. Include a method parseLogEntry(String line) that extracts timestamp, level, and message from each line.

Sample Input File (server.log):

[2024-01-15 10:00:00] INFO: Server started successfully
[2024-01-15 10:05:12] WARNING: High memory usage detected
[2024-01-15 10:10:30] ERROR: Database connection failed
[2024-01-15 10:15:45] INFO: User login successful
[2024-01-15 10:20:18] ERROR: File not found: config.xml
[2024-01-15 10:25:33] DEBUG: Processing request ID: 12345

Module 12 Java Swing

This module covers the fundamentals of creating graphical user interfaces (GUIs) in Java using the Swing library. We'll explore core components, layout management, and how to make your applications interactive through event handling.

Java Swing is a GUI (Graphical User Interface) toolkit in Java that allows developers to create desktop applications with interactive user interfaces. Swing is part of the Java Foundation Classes (JFC) and is built on top of the older Abstract Window Toolkit (AWT). It provides rich components like windows, buttons, text boxes, menus, tables, trees, and more. Fig-12.1 illustrates different components of Swing such drop box, check boxes etc...

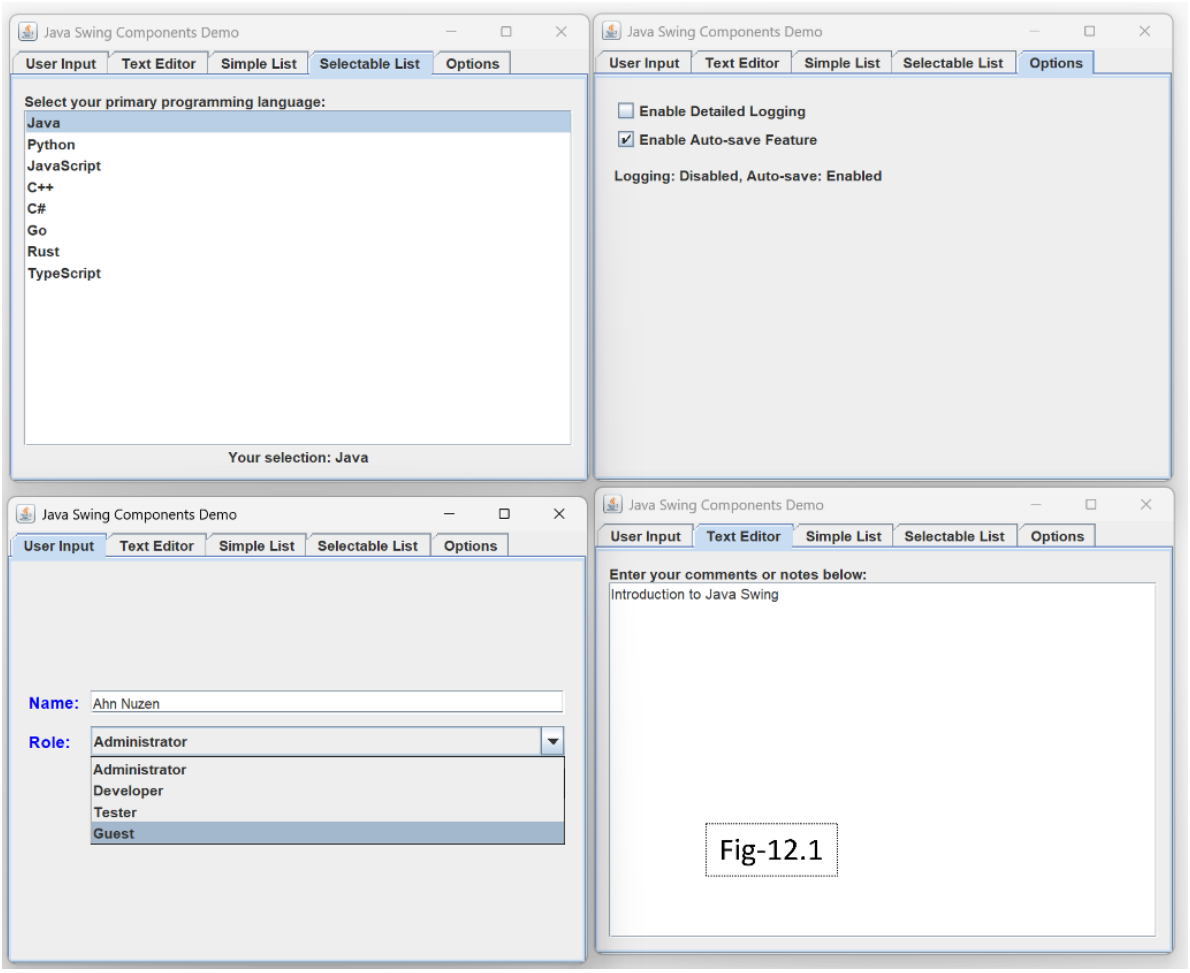


Fig-12.1

12.1 JFrame Class

At the heart of any Swing application is a top-level container, most commonly a JFrame. Think of it as the main window of your application. Components like labels, buttons, and text fields are placed inside this frame.

The JFrame class creates this main window. Key methods include:

1. setTitle(String title): Sets the text in the window's title bar.
2. setSize(int width, int height): Sets the window's dimensions in pixels.
3. setDefaultCloseOperation(int operation): Defines what happens when the user closes the window. JFrame.EXIT_ON_CLOSE is commonly used to terminate the application.
4. setVisible(true): Makes the window appear on the screen.

The JLabel class is one of the simplest Swing components. Its purpose is to display a single line of read-only text, an image, or both. You can create a label with text using new JLabel("Your text here"). The components are not visible until they added to the JFrame with the add() method.

In the example below, the SimpleWindow.java class creates a basic Swing application, using JFrame. The key part is extends JFrame, which means SimpleWindow inherits all the properties and behaviors of a JFrame. In essence, SimpleWindow *is* a specialized type of JFrame class.

The Constructor: public SimpleWindow()

This is where the window is configured, and its contents are added.

- setTitle("My First GUI");: Sets the text that appears in the window's title bar.
- setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);: This is a crucial line. It tells the program to terminate completely when the user clicks the 'X' (close) button on the window. Without it, the window would close, but the application would keep running in the background.
- setSize(300, 200);: Sets the initial size of the window to be 300 pixels wide and 200 pixels tall.
- JLabel label = new JLabel("Hello, Swing!");: This creates an instance of a JLabel component that will display the text "Hello, Swing!".
- add(label);: This adds the label component to the content area of the JFrame.

- setLocationRelativeTo(null);: This automatically centers the window on the computer screen.
- setVisible(true);: This makes the window, and all its components actually appear on the screen. By default, frames are not visible.

The main Method

- public static void main(String[] args) { ... }: This is the standard entry point for any Java application.
- new SimpleWindow();: This line creates a new instance of our SimpleWindow class, which runs the constructor and displays the GUI we've defined.

Example: Creating a SimpleWindow.java Class

```
import javax.swing.JFrame;
import javax.swing.JLabel;

/**
 * This is a simple Swing application
 * SimpleWindow is a subclass of JFrame that creates a basic GUI window.
 *
*/
public class SimpleWindow extends JFrame {
    /**
     * Constructor for SimpleWindow
     * This sets up the window with a title, default close operation, size, and a label.
     * It also centers the window on the screen and makes it visible.
     */
    public SimpleWindow() {
        setTitle("My First GUI");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 200);

        JLabel label = new JLabel("Hello, Swing!");
        add(label);

        // Add this line to center the frame
        setLocationRelativeTo(null);
        // Make the window visible
        setVisible(true);
    }
```

```
    public static void main(String[] args) {
        new SimpleWindow();
    }

}
```

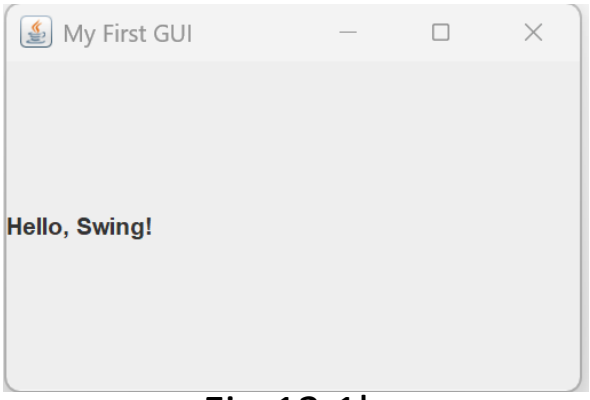


Fig-12.1b

12.2 Layout Managers

Without instructions, Swing wouldn't know how to arrange the components you add to a frame. A Layout Manager is an object that controls the positioning and sizing of components within a container. You set the layout manager for a frame using the setLayout() method.

Common Layout Managers:

- FlowLayout: The default for JPanel. It lays out components in a single row, one after another, much like words on a page.
- BorderLayout: The default for JFrame. It divides the container into five regions: NORTH, SOUTH, EAST, WEST, and CENTER. You add components specifying a region, e.g., frame.add(component, BorderLayout.NORTH).
- GridLayout: Arranges components in a rectangular grid of cells with equal size.

Example: Using BorderLayout.java Class

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.BorderLayout;
```

```
/**
 * This example demonstrates the use of BorderLayout in a JFrame
 *
 */

public class LayoutExample extends JFrame {
    /**
     * Constructor for LayoutExample
     * This sets up the frame with a title, default close operation, and size.
     * It also adds buttons to different regions of the BorderLayout.
     */
    public LayoutExample() {
        setTitle("BorderLayout Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 300);

        // Set the layout manager for the frame
        setLayout(new BorderLayout());

        // Add buttons to different regions
        add(new JButton("North"), BorderLayout.NORTH);
        add(new JButton("South"), BorderLayout.SOUTH);
        add(new JButton("East"), BorderLayout.EAST);
        add(new JButton("West"), BorderLayout.WEST);
        add(new JButton("Center"), BorderLayout.CENTER);

        // Add this line to center the frame
        setLocationRelativeTo(null);

        setVisible(true);
    }
    public static void main(String[] args) {
        // Run the GUI code on the Event Dispatch Thread
        new LayoutExample();


    }

}
```

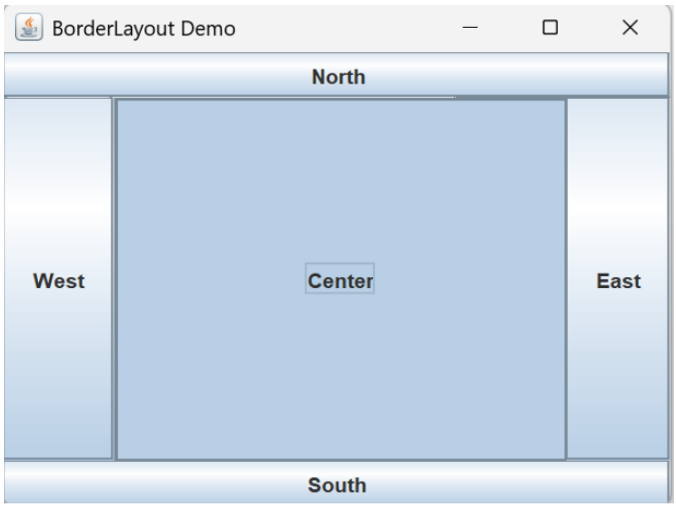


Fig-12.1

12.3 Event-Driven Programming & Event Listeners

GUI applications are event-driven. This means the program doesn't follow a predefined path; instead, it waits for and responds to events, which are typically user actions like clicking a button or moving the mouse.

The Java event model consists of three parts:

1. Event Source: The GUI component that generates an event (e.g., a JButton).
2. Event Object: An object that contains information about the event that occurred (e.g., an ActionEvent for a button click).
3. Event Listener: An object that "listens" for events and contains the code to handle them.

An Event Listener is an interface that you implement. To handle an event, you must:

1. Create a class that implements the appropriate listener interface (e.g., ActionListener for button clicks).
2. Override the method(s) defined in that interface (e.g., actionPerformed(ActionEvent e)).
3. Register an instance of your listener class with the event source component (e.g., button.addActionListener(yourListenerObject)).

In the example below, the EventExample.java class creates a basic Swing application, as a subclass of JFrame, additionally it also implements the interface ActionListener to handle the event when the user clicks the button. Let review the keys methods and class definition.

Class EventExample definition:

- public class EventExample **extends** JFrame **implements** ActionListener {
    - extends JFrame: This means the EventExample class *is* a subclass of JFrame. It inherits all the functionality of a swing window.
    - implements ActionListener: This tells the Java compiler that this class will provide a method to handle ActionEvents. The required method is actionPerformed().

The Constructor: public EventExample()

This method sets up the entire GUI similar to previous example.

- JButton button = new JButton("Click Me!");: Creates a button component with the text "Click Me!".
- button.addActionListener(this);: This is the core of the event handling.
    - button is the event source (the component that generates the event).
    - addActionListener(...) is the registration method. It tells the button which object to notify when it's clicked.
    - **this** refers to the current EventExample object. Since EventExample implements ActionListener, it is a valid event listener.

The actionPerformed Method

This is the method that the ActionListener interface requires. It is automatically called by Swing whenever the button (the registered source) is clicked.

- @Override: An annotation that tells the compiler we are intentionally overriding a method from a superclass or interface. It helps prevent typos.

- public void actionPerformed(ActionEvent e): The method signature. The ActionEvent e object is passed in by Swing and contains details about the event, such as which component triggered it.
- if (e.getSource() instanceof JButton): This is good practice. e.getSource() returns the object that originated the event. This if statement checks if that object was a JButton. This is useful if you have one listener handling events from multiple components (e.g., several buttons, menu items, etc.).
- JButton sourceButton = (JButton) e.getSource();: Casts the event source back to a JButton so we can call button-specific methods on it.
- label.setText("Button was clicked!"); and label.setForeground(Color.blue);: This is the response to the event. It changes the text and color of the JLabel.
- sourceButton.setText("Clicked!"); and sourceButton.setBackground(...): This demonstrates that you can also modify the source component itself in response to the event.

Example: Handling a Button Click

This example creates a button and a label. Clicking the button changes the label's text.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
/**
 * This example demonstrates basic event handling in Java Swing.
 * It creates a simple GUI with a button and a label.
 * When the button is clicked, the label updates to show a message.
 */

public class EventExample extends JFrame implements ActionListener {
    private JLabel label;
    /**
     * Constructor for EventExample
     * This sets up the frame with a title, default close operation, size, and a
button.
     * It also adds an ActionListener to the button to handle click events.
     */
```

```
    public EventExample() {
        setTitle("Event Handling");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 150);
        setLayout(new FlowLayout());

        JButton button = new JButton("Click Me!");
        // Register 'this' class as the listener for the button
        button.addActionListener(this);

        label = new JLabel("Waiting for a click...");

        add(button);
        add(label);
        button.setForeground(Color.white);
        //  make font larger
        button.setFont(new Font("Arial", Font.BOLD, 16));
        // Change the background color of the button
        button.setBackground(Color.gray);
        // Add this line to center the frame
        setLocationRelativeTo(null);
        // Make the window visible
        setVisible(true);
    }
    /**
     * This method is called when the button is clicked.
     * It updates the label to show a message indicating that the button was
clicked.
     */
    @Override
    public void actionPerformed(ActionEvent e) {
        /**
         * Since any event from any component will call this method
         * Check if the source of the event is the button
         * This ensures that we only respond to clicks on the button
         * If you have multiple buttons, you can use getActionCommand() or check the
source
         */
        if (e.getSource() instanceof JButton) {
            JButton sourceButton = (JButton) e.getSource();
            label.setText("Button was clicked!");
```

```
            label.setForeground(Color.blue);
            sourceButton.setText("Clicked!");
            sourceButton.setBackground(Color.darkGray);
        }
    }

    public static void main(String[] args) {
        new EventExample();
    }
}
```

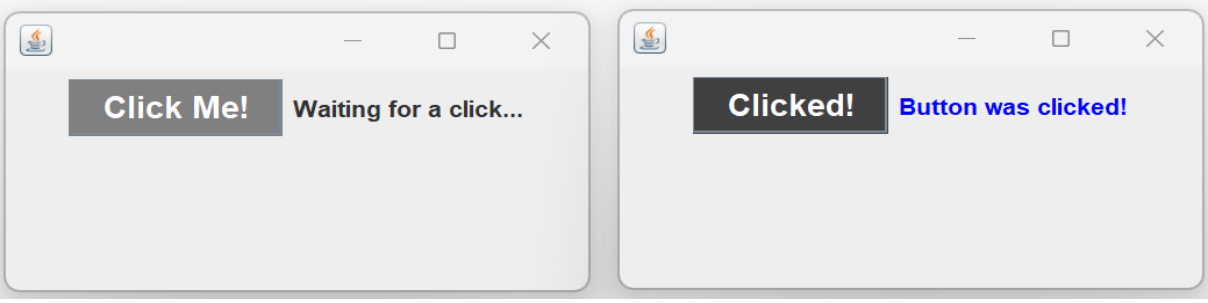


Fig-12.3

12.4 JSpinner Class

A JSpinner is a single-line input field that lets a user select a number or an object value from an ordered sequence using small up and down arrow buttons. It's a more constrained input method than a text field.

A JSpinner relies on a model to define its sequence of values. The most common is SpinnerNumberModel, which defines a sequence of numbers.

- SpinnerNumberModel(initialValue, minimum, maximum, stepSize)

To get the current value from a spinner, you use the getValue() method. To listen for changes in its value, you add a ChangeListener, which requires implementing the stateChanged(ChangeEvent e) method.

Example: Using a JSpinner

```
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
```

```
import java.awt.*;
/**
 * This example demonstrates the use of JSpinner in a Java Swing application.
 * It creates a simple GUI with a spinner that allows the user to select a number
 */
public class SpinnerExample extends JFrame implements ChangeListener {
    private JSpinner spinner;
    private JLabel label;
    /**
     * Constructor for SpinnerExample
     * This sets up the frame with a title, default close operation, size, and a
spinner.
     * It also adds a ChangeListener to the spinner to handle value changes.
     */

    public SpinnerExample() {
        setTitle("JSpinner Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 150);
        setLayout(new FlowLayout());

        // Create a model for numbers from 0 to 100, starting at 50, with a step of
5
        SpinnerNumberModel spinnerModel = new SpinnerNumberModel(50, 0, 100, 5);
        spinner = new JSpinner(spinnerModel);

        // Add a listener to detect when the spinner's value changes
        spinner.addChangeListener(this);

        label = new JLabel("Current value: 50");

        add(new JLabel("Select a value:"));
        add(spinner);
        add(label);

        // Center the frame on the screen
        setLocationRelativeTo(null);

        setVisible(true);
    }
```

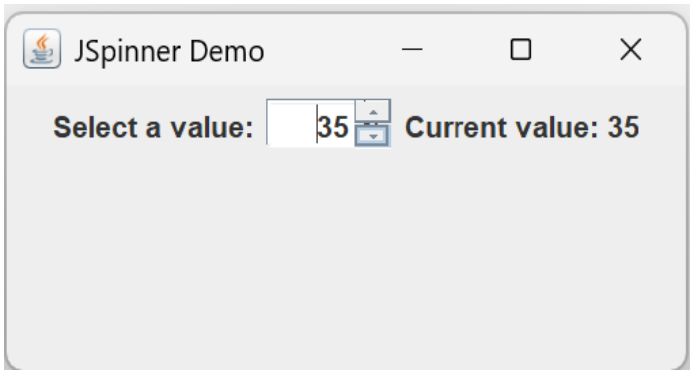


Fig-12.4

Module 11: Learning Materials & References

a) Java Swing Tutorial
b) Package javax.swing
c) Java GUI Windows (Swing Tutorial) Video

Part A: Quiz

12.1a Which of the following is the default layout manager for a JFrame's content pane?

a) BorderLayout
b) GridLayout
c) FlowLayout
d) BoxLayout

12.2a To handle the event generated by clicking a JButton, which listener interface should you implement?

a) MouseListener
b) ItemListener
c) WindowListener
d) ActionListener

12.3a What is the most common top-level container used to create the main window of a Java Swing application?

a) JPanel
b) JDialog
c) JFrame
d) JApplet

12.4a Which Swing component is specifically designed to display a single line of non-editable text or an image?

a) JTextField
b) JLabel
c) JTextArea
d) JEditorPane

12.5a In the Java Swing event model, what is the primary role of an ActionListener?

a) To define the specific method (actionPerformed) that will execute when a component's action event occurs.
b) To create the visual component, like a button, that the user interacts with.
c) To hold information about an event that has already happened, such as the time it occurred.
d) To manage the layout and positioning of components within a window.

12.6a Before an event listener object can respond to an event from a component (like a JButton), what crucial step must occur in the code?

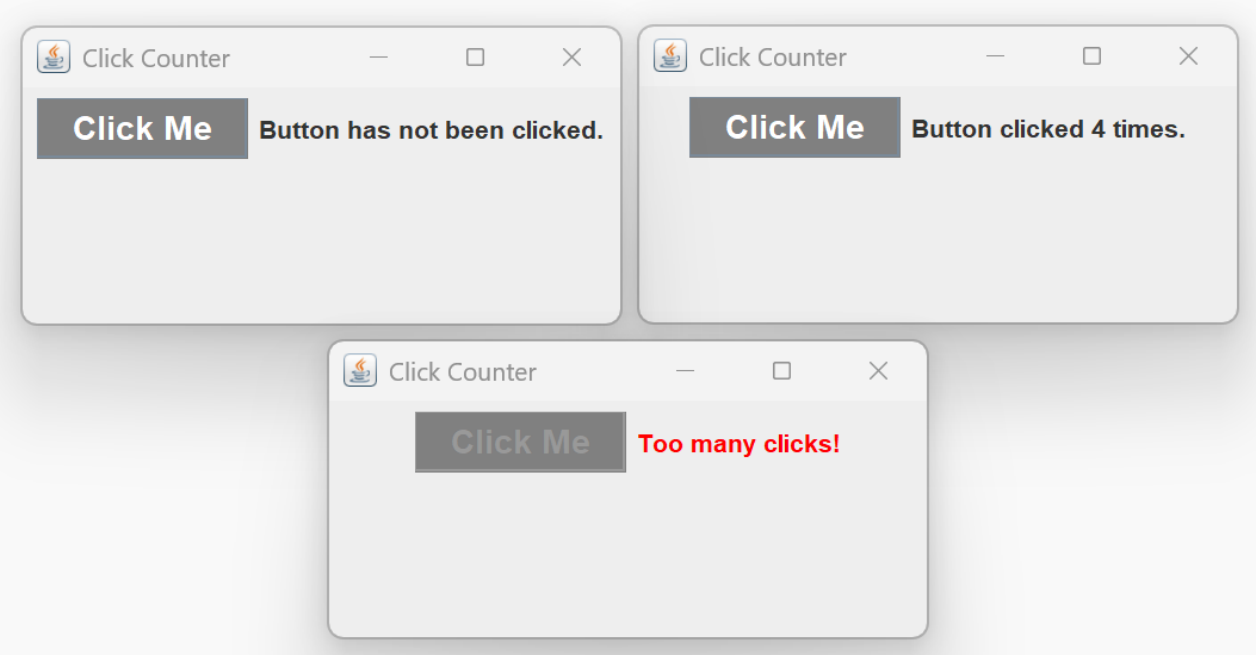
a) The listener must be made visible using the setVisible(true) method.
b) The component must be added to the listener.
c) The listener must be registered with the component (the event source) using an add...Listener() method.
d) The component must be disabled and then re-enabled to activate the event system.

Part C: Lab Assignments

12.1c Simple Click Counter

Write a Java Swing application that displays a button "Click Me". Each time the button is clicked, the label's text must be updated to show the current click count.
For example, after the first click, it should say "Button clicked 1 time." After the second, "Button clicked 2 times.", and so on. Once the user clicked the buttons more than 5 times, you should disabled the button.

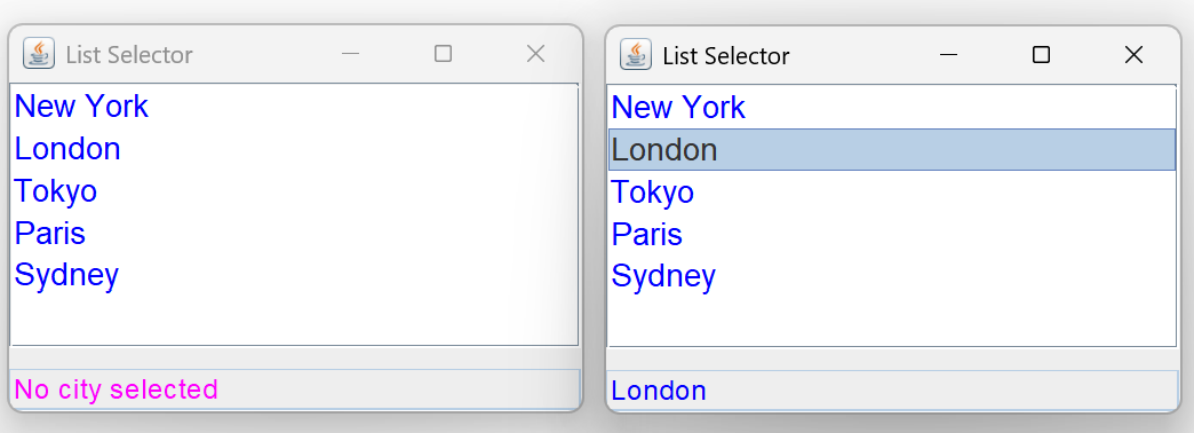
Hint: You will need to implement the ActionListener interface to handle the button click event. Keep a counter variable as a class member to track the number of clicks.



12.2c JList Selection to JTextField

Write a Java Swing application that displays a list of items (JList) and a text field (JTextField).

1. Populate the JList with a predefined array of strings (e.g., city names: "New York", "London", "Tokyo", "Paris").
2. The JTextField should initially be empty and non-editable.
3. When a user selects an item from the JList, the selected item's text should immediately appear in the JTextField.



Hints:

a) You will need to implement the ListSelectionListener interface and its valueChanged method.
b) Add the listener to the JList's selection model.
c) To prevent the event from firing twice for a single click, check if (!e.getValueIsAdjusting()) inside the valueChanged method.
d) use a JScrollPane for the list
e) For status line use BorderLayout.SOUTH

Appendix A

Java's Best Practices

Why is adherence to Java best practices essential for software development?

1. Adhering to Java best practices is essential for producing robust, readable, and maintainable software. Following established guidelines—such as those from Google's Java Style Guide or Oracle's documentation— consistent compliance promotes code clarity and understanding.
2. Documentation with javadoc string /** */, using javadoc string is recommended for documenting Java code. This comprehensive documentation enhances code clarity and understanding.
3. Follow Variable and Class Naming Conventions, outlined in Module 1 i.e., using lowerCarmelCase for variables and UpperCamelCase for class variables.
4. Enforcing OOP Encapsulation by using setters and encapsulation. This practice helps manage access visibilities to class attributes.
5. Every class should have a default toString() method, this method provides a string representation of the object when using System.out.println() , which is useful for debugging and displaying information about the object. Well-documented classes and methods improve code clarity and make it easier for others (and your future self) to understand intent, usage, and expected behavior.
6. Consistent Naming Conventions, Variable and class names should follow established conventions: lowerCarmelCase for variables and methods, UpperCamelCase for classes. Logical, predictable names make code more intuitive, reducing the cognitive load for readers and collaborators.
7. Proper use of encapsulation—defining private fields and providing controlled access via setters and getters—protects class invariants and abstracts implementation details. This not only secures internal state but also enables safer code evolution.
8. Implementing a Default toString() Method, providing a custom toString() in every class ensures meaningful output during debugging and logging. When System.out.println() is called, the object's essential state is clearly represented. This practice aids both debugging and code reviews.

1.0 How should class & variables be named according to best practices?

See Module 1.4.2 Naming Conventions.

Example

```
/**
 * programming Exercise Product.java class
 * Student Name: Your Name
 * Course Name: CSIS-293
 * Professor: A Nuzen
 * Java Class Name: Product
 * Purpose: declares object Product and its methods
 */
public class Product extends Object {
   /** even though Object is the default superclass
    * it is good practice to explicitly extend it */

   /** Product attributes */
   private int itemNumber;// lowerCarmelCase
   private double price;  //  lower case for the first word

   /** non-default constructor initialize object with parameters */
   Product(int itemNumber, double price){
      /** this refers to the current object
       *  this.itemNumber and this.price refer to the instance variables
       *  itemNumber and price refer to the parameters
       *  use this instead of using different names for the parameters
       */
      this.itemNumber = itemNumber;
      this.price = price;
   }

   /** default constructor initialize object with default values
    * itemNumber = 0, price = 0.0 */
   Product(){
      this(0, 0.0);
   }
```

```
   /** setters and getters */
   public int getItemNumber() { return itemNumber;}
   public double getPrice(){ return price;}
   public void setPrice(double price){ this.price = price;}

   @Override
   /** return string representing the object Product */
   public String toString() {
      return String.format("%d\t$%.2f",itemNumber,price);
   }
}
```

3.11 Constants in Java

```
    /** example of constants in Java
     * Constants are typically declared as final
     * to prevent modification after initialization
     * and are usually in uppercase letters
    */
    final String STUDENT_NAME = "Your Name";
    final String JAVA_CLASS_NAME = "Product";
    public final double PI = 3.14159;
    static final int MAX_USERS = 1000;

    public static final double PI = 3.14159;
    public static final int MAX_USERS = 1000;

       /** Utility class for HTTP status codes */
       class StatusUtils extends Object{
           /** HTTP status codes as constants */
           final int STATUS_OK = 200;
           final int STATUS_ERROR = 500;

           public String getStatusMessage(int statusCode) {
               if (statusCode == STATUS_OK) {
                   return "Success";
               } else if (statusCode == STATUS_ERROR) {
                   return "Error";
               } else {
                   return "Unknown Status";
               }
```

```
        }
    }
```

3.12 Variables & Data Structures with Implicit Declarations

```
    /** examples of implicit declaration of constants */
    /** array of object products with initial values */
    Product[] products = {
        new Product(101, 19.99),
        new Product(102, 29.99),
        new Product(103, 39.99)
    };

  /** array of product prices */
    double[] prices = {19.99, 29.99, 39.99};

    /** multidimensional array of maximum values */
    int[][] maxValues    = {{1, 2}, {3, 4}};

    /** array of product names */
    String[] productNames = {"Widget", "Gadget", "Doodad"};
    int[] numbers = {10, 20, 30, 40};

    for (double price : prices) {
        System.out.println(price);
    }
    for (String name : productNames) {
        System.out.println(name);
    }
    for (int[] row : maxValues) {
        for (int value : row) {
            System.out.println(value);
        }
    }
```

3.13 Why is it important for every class to have a default toString() method?

Every class should have a default toString() method because it provides a readable string representation of an object instance. This method is automatically called when you use System.out.println() on an instance of the class. It is invaluable for debugging and for easily displaying information about the object's state.

```
@Override
/** return string representing the object Product */
public String toString() {
   return String.format("%d\t$%.2f",itemNumber,price);
}
```

3.14 Generate JavaDoc HTML

The javadoc command is used to generate HTML-formatted API documentation from Java source files that contain special Javadoc comments (/** ... */). It produces a set of linked HTML pages describing the classes, methods, fields, and packages.

Basic Usage of javadoc Command to Generate HTML

javadoc -d <output-directory> -sourcepath <source-directories> <packages-or-source-files>

- -d <output-directory> specifies the destination directory where the generated HTML files will be saved.
- -sourcepath <source-directories> tells the tool where the Java source files are located (can be multiple directories separated by colon : on Unix/macOS or semicolon ; on Windows).
- <packages-or-source-files> is the list of package names or individual .java files you want to document.

Example:

javadoc **-author -private -d docs** Product.java

Include private variables and authors of HelloWorld.java class in the current directory. The output directory is docs.

javadoc -d C:\project\docs -sourcepath C:\project\src Product.java