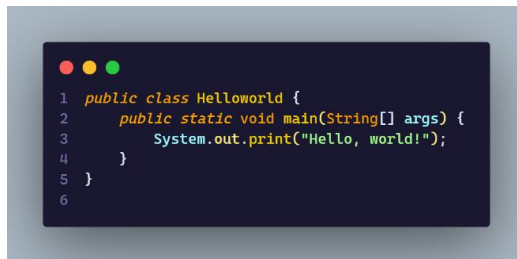


Name: VEROAUGSTAN MAC-ANTHONIO CHUKWUDUBEM

Matric Number: 2023/253016

Lecturer: MRS. AMINAT ATANDA

1. Write a Java program to print "Hello, World!"

A screenshot of a Java IDE with a dark background. It shows a simple Java class named 'Helloworld' with a 'main' method that prints 'Hello, world!'. The code is as follows:

```
1 public class Helloworld {
2     public static void main(String[] args) {
3         System.out.print("Hello, world!");
4     }
5 }
6
```

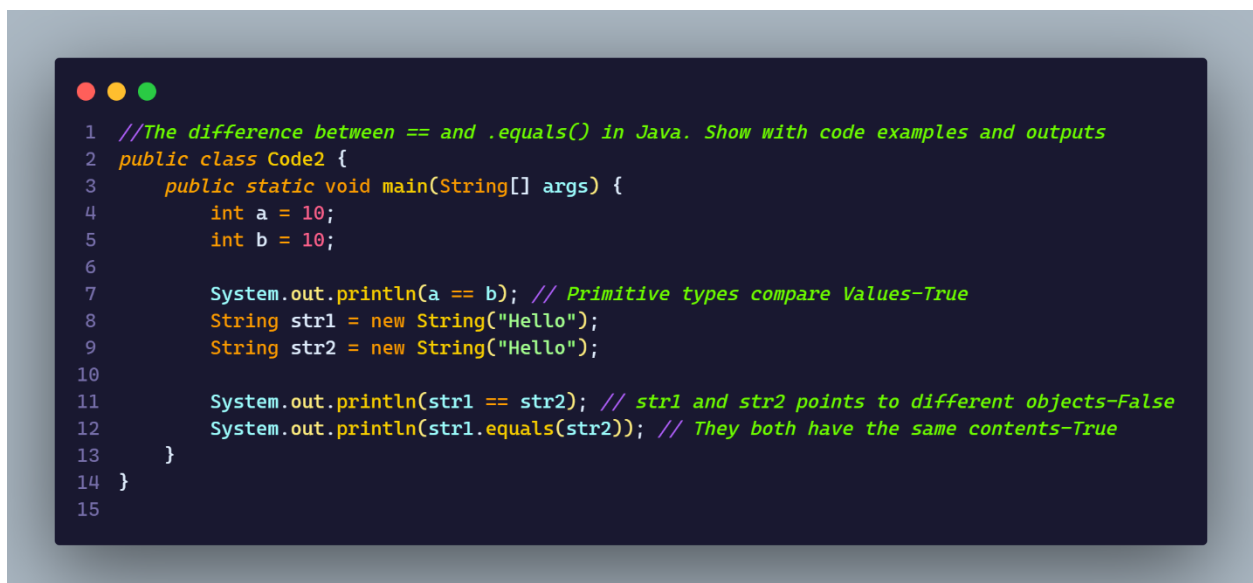
2. Explain the difference between == and .equals () in Java. Show with code examples and outputs

In Java, == and .equals () are both used to compare objects, but they are used in different contexts and behave differently.

The == operator is used to compare **references** (memory addresses) of objects. It checks if two references point to the exact same object in memory. For primitive types (like int, char, double, etc.), == compares the actual values.

The .equals() is method is used to compare **the content** of two objects (for class types) to see if they are logically equivalent.

Example: Comparing Primitives and References

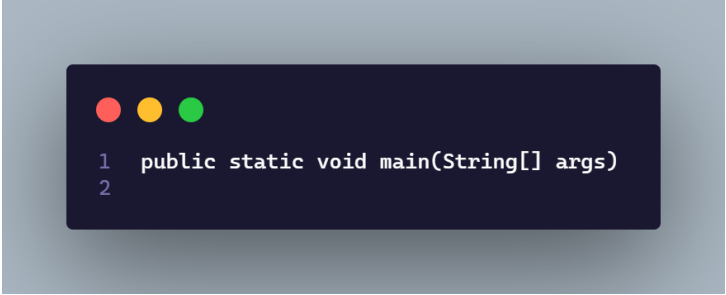
A screenshot of a Java IDE with a dark background. It shows a Java class named 'Code2' with a 'main' method. The code demonstrates comparing primitive types (int) and reference types (String) using == and .equals(). The code is as follows:

```
1 //The difference between == and .equals() in Java. Show with code examples and outputs
2 public class Code2 {
3     public static void main(String[] args) {
4         int a = 10;
5         int b = 10;
6
7         System.out.println(a == b); // Primitive types compare Values-True
8         String str1 = new String("Hello");
9         String str2 = new String("Hello");
10
11         System.out.println(str1 == str2); // str1 and str2 points to different objects-False
12         System.out.println(str1.equals(str2)); // They both have the same contents-True
13     }
14 }
15
```

3. What is the use of the main method in Java?

The `main` method in Java is the **entry point** of any standalone Java application — it's where the program starts running.

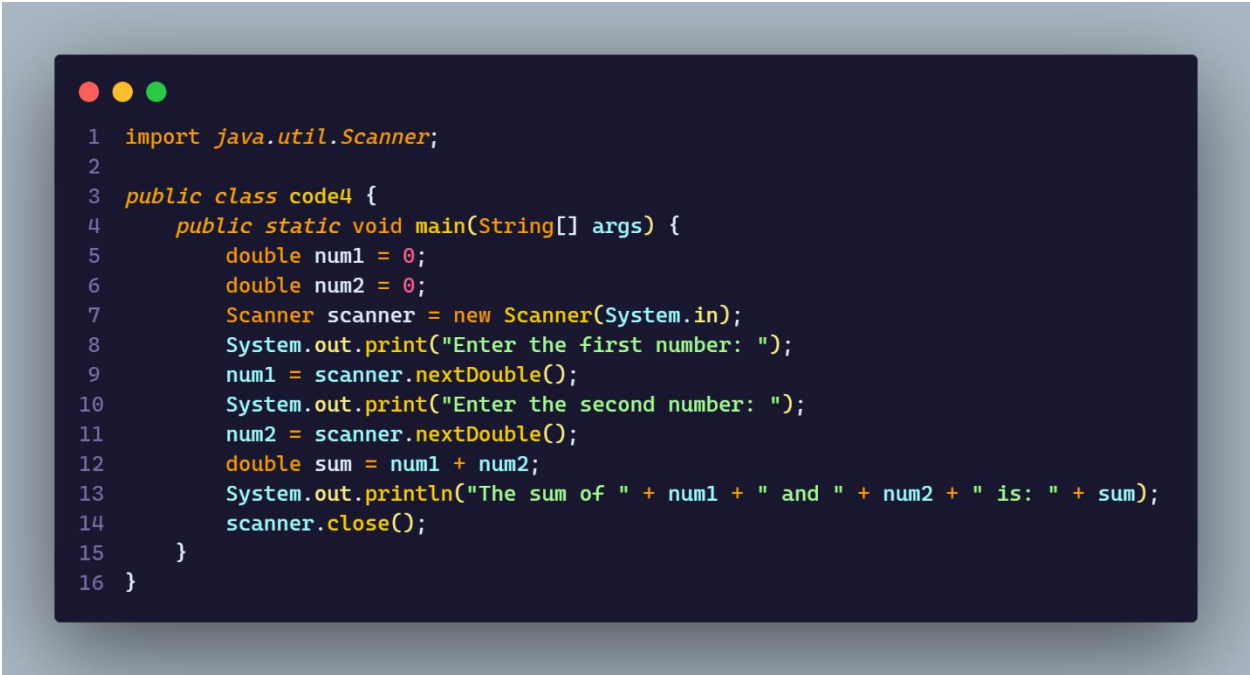
Its standard form is:



```
1 public static void main(String[] args)
2
```

- **public:** It must be accessible from **outside the class** so the Java Virtual Machine (JVM) can call it.
- **static:** No need to create an object of the class to run the method — the JVM can call it **without instantiating the class**.
- **void:** It doesn't return anything.
- **main:** The name the JVM looks for to start the program.
- **String[] args:** A parameter to receive **command-line arguments**, if any.

4. Write a Java program to add two numbers entered by the user.



```
1 import java.util.Scanner;
2
3 public class code4 {
4     public static void main(String[] args) {
5         double num1 = 0;
6         double num2 = 0;
7         Scanner scanner = new Scanner(System.in);
8         System.out.print("Enter the first number: ");
9         num1 = scanner.nextDouble();
10        System.out.print("Enter the second number: ");
11        num2 = scanner.nextDouble();
12        double sum = num1 + num2;
13        System.out.println("The sum of " + num1 + " and " + num2 + " is: " + sum);
14        scanner.close();
15    }
16 }
```

Output:

```
PS C:\Users\Admin\COS 261 Assignment\Codes> cd "c:\Users
Enter the first number: 7
Enter the second number: 9
The sum of 7.0 and 9.0 is: 16.0
```

5. What is the difference between int, Integer, and String?

Int is a primitive data type that has a **Size**: 32-bit signed integer, it holds simple numeric values.

```
1 int x = 10;
```

Integer is a **Wrapper Class of type "Object"** part of java.lang package. It Wraps a primitive int in an object and is useful when working with collections (like ArrayList), or when null values are needed. It **Provides methods** like Integer.parseInt("123"), .compareTo(), .toString(), etc.

```
1 Integer y = Integer.valueOf(10);
```

A String is a class for text. It represents sequences of characters like "Hello" or "123".

```
1 String name = "Alice";
2
```

Control Structures

6. Write a program to check if a number is even or odd:

```
1  import java.util.Scanner;
2
3  public class Code6 {
4      public static void main(String[] args) {
5
6          Scanner myInput = new Scanner(System.in);
7          System.out.println("Welcome to the Even/Odd Checker program");
8
9          boolean keepRunning = true; // Control variable to keep the loop running
10
11         // Start of the loop to check numbers repeatedly
12         while (keepRunning) {
13             // Prompt user to enter a number
14             System.out.print("Enter a number: ");
15             int number = myInput.nextInt(); // Read an integer from user input
16
17             // Check if the number is even or odd using modulus operator
18             if (number % 2 == 0) {
19                 System.out.println(number + " is an even number.");
20             } else {
21                 System.out.println(number + " is an odd number.");
22             }
23
24             // Ask user if they want to check another number
25             System.out.print("Would you like to check another number? (yes/no): ");
26             String response = myInput.next().toLowerCase(); // Read response and convert to lowercase
27
28             // Exit the loop if the user answers "no"
29             if (response.equals("no")) {
30                 keepRunning = false;
31             }
32         }
33
34         // Close the Scanner to prevent resource leak
35         myInput.close();
36
37         // Final message to the user after exiting the loop
38         System.out.println("Thank you for using the Even/Odd Checker program!");
39     }
40 }
41
```

Output:

```
PS C:\Users\Admin\COS 261 Assignment\Codes> cd "c:\Users\
Welcome to the Even/Odd Checker program
Enter a number: 70
70 is an even number.
Would you like to check another number? (yes/no): yes
Enter a number: 45
45 is an odd number.
Would you like to check another number? (yes/no): no
Thank you for using the Even/Odd Checker program!
PS C:\Users\Admin\COS 261 Assignment\Codes>
```

7. Write a program to find the largest among three numbers

```
1  import java.util.Scanner;
2
3  public class code7 {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6
7          // Ask the user to enter three numbers
8          System.out.print("Enter the first number: ");
9          int num1 = input.nextInt();
10
11         System.out.print("Enter the second number: ");
12         int num2 = input.nextInt();
13
14         System.out.print("Enter the third number: ");
15         int num3 = input.nextInt();
16
17         int largest;
18
19         // Compare the numbers using if-else
20         if (num1 >= num2 && num1 >= num3) {
21             largest = num1;
22         } else if (num2 >= num1 && num2 >= num3) {
23             largest = num2;
24         } else {
25             largest = num3;
26         }
27
28         // Display the result
29         System.out.println("The largest number is: " + largest);
30
31         input.close(); // Close the Scanner
32     }
33 }
34
```

Output:

```
Enter the first number: 20
Enter the second number: 80
Enter the third number: 90
The largest number is: 90
```

8. Explain the difference between while, for, and do-while loops in Java.

while Loop:

The while loop in Java is used when you want to repeat a block of code **an unknown number of times**, but **only while a specific condition remains true**. The condition is checked **before** each iteration, so if it's false right at the start, the loop body won't run even once. This makes while loops useful when you can't determine in advance how many times you need to loop—for example, waiting for a specific **user** input or external condition to be met. Since it checks the condition first, the loop has the potential to never execute at all.

```
1 int i = 0;
2 while (i < 5) {
3     System.out.println(i);
4     i++;
5 }
```

For Loop:

The `for` loop is typically used when you **know how many times you want to run a block of code**. It combines loop initialization, condition checking, and iteration into a single line, making it concise and easy to manage. Like the `while` loop, the `for` loop also checks its condition **before each iteration**, so if the condition is false initially, the loop body will not execute. This type of loop is commonly used when counting, such as iterating over a range of numbers or processing items in an array with a fixed size.

```
1 for (int i = 0; i < 5; i++) {
2     System.out.println(i);
3 }
```

Do-while Loop:

The do-while loop differs from the other two in that it **guarantees the loop body will run at least once**, regardless of the condition. This is because the condition is checked **after** the code inside the loop has executed. It's particularly useful in scenarios where the loop logic needs to happen first—such as displaying a menu or asking for user input—before determining whether to continue. Even if the condition is false from the beginning, the loop body will execute once before it stops.

```
1 int i = 0;
2     do {
3         System.out.println(i);
4         i++;
5     } while (i < 5);
```

9. Write a Java program to print the multiplication table of any number.

```
1
2 import java.util.Scanner;
3
4 public class code9 {
5     public static void main(String[] args) {
6         Scanner input = new Scanner(System.in);
7
8         // Ask the user to enter a number
9         System.out.print("Enter a number to print its multiplication table: ");
10        int number = input.nextInt();
11
12        // Print multiplication table from 1 to 10
13        System.out.println("Multiplication table for " + number + ":");
14        for (int i = 1; i <= 12; i++) {
15            System.out.println(number + " x " + i + " = " + (number * i));
16        }
17
18        input.close(); // Close the scanner
19    }
20 }
21
```

Output:

```
Enter a number to print its multiplication table: 8
Multiplication table for 8:
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
8 x 10 = 80
8 x 11 = 88
8 x 12 = 96
```

Intermediate-Level Questions:

10. Explain the four pillars of OOP in Java.

A. Encapsulation

Encapsulation is the concept of **bundling data (fields) and methods (functions) that operate on that data into a single unit**, usually a class. It also involves restricting direct access to some of the object's components, typically by marking fields as `private` and providing access through `public` getters and setters. This helps protect the internal state of an object from unintended interference and enforces a controlled interface for interaction.

B. Abstraction

Abstraction is the process of **hiding complex implementation details and showing only the essential features** of an object. It allows you to focus on what an object does rather than how it does it. In Java, abstraction is achieved using **abstract classes** and **interfaces**. This helps simplify code and reduce complexity by focusing on high-level operations.

C. Inheritance

Inheritance allows one class (child or subclass) to **inherit the fields and methods** of another class (parent or superclass). This promotes **code reuse** and establishes a **hierarchical relationship** between classes. A subclass can extend or override the behavior of its superclass to provide more specific functionality.

D. Polymorphism

Polymorphism means "many forms." It allows objects to be treated as instances of their parent class rather than their actual class. Java supports **compile-time (method overloading)** and **runtime polymorphism (method overriding)**. This enables the same interface or method name to behave differently based on the object that is calling it.

11. Create a class Student with properties name, matricNo, and score, and add methods to display the student's info.

```
1  public class Student {
2      // Properties
3      private String name;
4      private String matricNo;
5      private double score;
6
7      // Constructor
8      public Student(String name, String matricNo, double score) {
9          this.name = name;
10         this.matricNo = matricNo;
11         this.score = score;
12     }
13
14     // Method to display student information
15     public void displayInfo() {
16         System.out.println("Student Name: " + name);
17         System.out.println("Matric Number: " + matricNo);
18         System.out.println("Score: " + score);
19     }
20
21     // Main method to test the Student class
22     public static void main(String[] args) {
23         Student student1 = new Student("Alice Johnson", "MAT2023001", 87.5);
24         student1.displayInfo();
25     }
26 }
27
```

Test Output:

```
Student Name: Alice Johnson
Matric Number: MAT2023001
Score: 87.5
```

12. What is method overloading? Give a code example.

Method overloading in Java is a feature that allows a class to have more than one method with the same name, but with different **parameter lists** (different number or type of parameters). It increases the readability of the program and allows flexibility in method usage.

Example:

```
1  public class Calculator {
2
3      // Method to add two integers
4      public int add(int a, int b) {
5          return a + b;
6      }
7
8      // Overloaded method to add three integers
9      public int add(int a, int b, int c) {
10         return a + b + c;
11     }
12
13     // Overloaded method to add two doubles
14     public double add(double a, double b) {
15         return a + b;
16     }
17
18     public static void main(String[] args) {
19         Calculator calc = new Calculator();
20
21         System.out.println("Sum of 2 integers: " + calc.add(5, 10)); // Uses int add(int, int)
22         System.out.println("Sum of 3 integers: " + calc.add(5, 10, 15)); // Uses int add(int, int, int)
23         System.out.println("Sum of 2 doubles: " + calc.add(5.5, 10.2)); // Uses double add(double, double)
24     }
25 }
26
```

Output:

```
Sum of 2 integers: 15
Sum of 3 integers: 30
Sum of 2 doubles: 15.7
```

13. What is inheritance? Create a base class Person and a subclass Teacher.

Inheritance in Java is a mechanism that allows one class (called a **subclass** or **child class**) to inherit fields and methods from another class (called a **superclass** or **parent class**). This promotes code reusability and establishes a natural hierarchy between classes.

```

1  // Base class
2  public class Person {
3      protected String name;
4      protected int age;
5
6      // Constructor
7      public Person(String name, int age) {
8          this.name = name;
9          this.age = age;
10     }
11
12     // Method to display person info
13     public void displayInfo() {
14         System.out.println("Name: " + name);
15         System.out.println("Age: " + age);
16     }
17 }
18
19 // Subclass
20 public class Teacher extends Person {
21     private String subject;
22
23     // Constructor
24     public Teacher(String name, int age, String subject) {
25         super(name, age); // Call the constructor of the superclass
26         this.subject = subject;
27     }
28
29     // Override or extend functionality
30     @Override
31     public void displayInfo() {
32         super.displayInfo(); // Call the method from the superclass
33         System.out.println("Subject: " + subject);
34     }
35
36     // Main method to test
37     public static void main(String[] args) {
38         Teacher t = new Teacher("Mr. Smith", 40, "Mathematics");
39         t.displayInfo();
40     }
41 }
42

```

14. What does it mean to write “clean code”? Give 3 practices that make code clean and maintainable.

"Clean code" refers to code that is **easy to read, understand, and maintain**. It's not just about making the code work — it's about writing it in a way that others (and your future self) can easily work with. Clean code reduces bugs, improves collaboration, and scales better over time.

Three practices that make code clean and maintainable

A. Use Meaningful and Descriptive Names:

Variable, method, and class names should clearly indicate their purpose.

B. Keep Methods Short and Focused:

Each method should do **one thing only**, and do it well.

C. Avoid Code Duplication:

Duplicated logic is harder to update and prone to inconsistency.

Extract common code into reusable methods or classes.

15. Why should you avoid writing very long methods in Java programs?

They make your code **harder to understand, test, debug, and maintain**. Long methods often try to do too much, violating the **Single Responsibility Principle**, which is a core part of clean, modular code.

Best Practice is to break long methods into **smaller, well-named helper methods**, each doing a clearly defined task.

16. What naming conventions should be followed in Java for: Classes, Variables, Methods?

A. Class Names

- **Convention:** Use **PascalCase**, where each word starts with a capital letter and there are no underscores.
- **Purpose:** Class names represent objects or entities, so they should be nouns and descriptive.
- **Examples:**
 - Student
 - BankAccount
 - EmployeeData

B. Variable Names

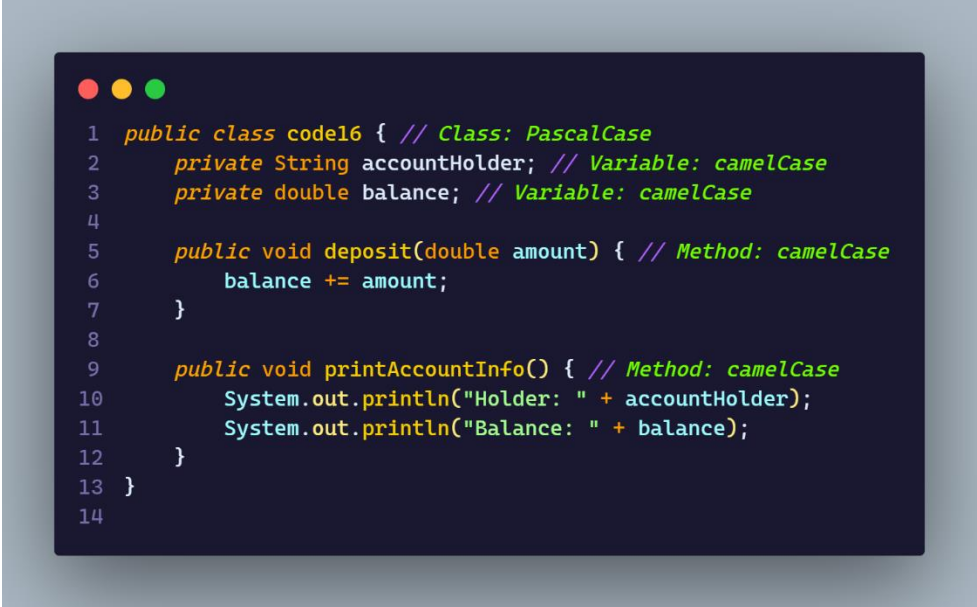
- **Convention:** Use **camelCase**, where the first word is lowercase and each subsequent word starts with a capital letter.
- **Purpose:** Variables hold data, so their names should clearly indicate what the data represents.
- **Examples:**
 - firstName
 - totalAmount
 - isValidUser

C. Method Names

- **Convention:** Also use **camelCase**, just like variables.

- **Purpose:** Method names typically describe an action, so they should start with a verb and be meaningful.
 - **Examples:**
 - calculateSalary()
 - printDetails()
 - getUserData()
-

Code Example:



```
1 public class code16 { // Class: PascalCase
2     private String accountHolder; // Variable: camelCase
3     private double balance; // Variable: camelCase
4
5     public void deposit(double amount) { // Method: camelCase
6         balance += amount;
7     }
8
9     public void printAccountInfo() { // Method: camelCase
10        System.out.println("Holder: " + accountHolder);
11        System.out.println("Balance: " + balance);
12    }
13 }
14
```

17. What is the importance of breaking your Java program into methods?

Breaking your Java program into **methods** is important for several key reasons — it leads to code that is **easier to read, manage, debug, reuse, and scale**. Instead of writing one long block of code, organizing functionality into separate methods improves structure and quality.

Key Benefits of Using Methods in Java:

A. Improves Readability

- Code is easier to follow when each method has a clear, focused purpose.
 - A well-named method like calculateGrade() instantly tells you what it does — no need to read every line.
-

B. Encourages Reusability

- You can reuse the same method in multiple parts of your program instead of rewriting code.
 - Example: A method `validateInput()` can be used for both login and registration.
-

C. Simplifies Debugging and Testing

- Smaller methods isolate functionality, making bugs easier to locate and fix.
 - Unit testing is much easier with well-scoped, individual methods.
-

D. Enforces the Single Responsibility Principle

- Each method should do one thing well. This keeps logic clean and avoids “spaghetti code.”
-

E. Improves Maintainability

- If changes are needed, you can often update a single method without touching the entire program.
-

F. Makes Collaboration Easier

- In team environments, developers can work on different methods without interfering with each other’s work.
-

18. Explain the concept of DRY (Don’t Repeat Yourself) with a Java code example

DRY (Don’t Repeat Yourself) is a principle in software development aimed at reducing repetition in code. The idea is that every piece of knowledge or logic should be represented **only once** within a system. Repeating code can lead to errors, increased maintenance costs, and harder-to-understand code. Instead, we should extract common functionality into methods, functions, or classes, and reuse them as needed.

EXAMPLE OF CODE WITHOUT DRY:

```
1 package code17;
2
3 public class Calculator {
4
5     public static void main(String[] args) {
6         int num1 = 5, num2 = 3;
7
8         // Repeated code for adding numbers
9         int sum1 = num1 + num2;
10        System.out.println("Sum1: " + sum1);
11
12        num1 = 7;
13        num2 = 8;
14
15        // Repeated code for adding numbers
16        int sum2 = num1 + num2;
17        System.out.println("Sum2: " + sum2);
18
19        num1 = 9;
20        num2 = 4;
21
22        // Repeated code for adding numbers
23        int sum3 = num1 + num2;
24        System.out.println("Sum3: " + sum3);
25    }
26 }
27
```

EXAMPLE OF CODE WITH DRY:

```

1  package code17;
2
3  public class calc {
4
5      // DRY: Create a method to calculate sum
6      public static int add(int num1, int num2) {
7          return num1 + num2;
8      }
9
10     public static void main(String[] args) {
11         int num1 = 5, num2 = 3;
12
13         // Reuse the method to calculate the sum
14         System.out.println("Sum1: " + add(num1, num2));
15
16         num1 = 7;
17         num2 = 8;
18
19         // Reuse the method to calculate the sum
20         System.out.println("Sum2: " + add(num1, num2));
21
22         num1 = 9;
23         num2 = 4;
24
25         // Reuse the method to calculate the sum
26         System.out.println("Sum3: " + add(num1, num2));
27     }
28 }
29

```

19. What are the benefits of using classes and objects instead of writing all logic in the main methods?

In Java programming, relying solely on the main method to handle all logic quickly leads to disorganized, repetitive, and difficult-to-maintain code. Instead, using **classes and objects** — the foundation of object-oriented programming (OOP) — brings structure and clarity to your programs. A class is a blueprint that defines properties (fields) and behaviors (methods), while an object is a specific instance of that class. This approach mirrors real-world entities, making your code more intuitive and easier to manage as it grows in complexity.

One of the main benefits of using classes and objects is **encapsulation** — grouping related data and behaviors into self-contained units. This not only keeps your code organized but also promotes **reusability**, as you can create multiple objects from the same class without duplicating code. By splitting logic across classes, programs become easier to read, test, debug, and scale. It also enables the use of powerful OOP principles like **inheritance**, **polymorphism**, and **abstraction**, which further enhance flexibility and maintainability. In contrast, cramming all functionality into the main method leads to monolithic code that's error-prone and hard to extend. Embracing classes and objects is essential for writing clean, modular, and professional Java applications.

20. Why is Testing Important During Program Development?

Testing is crucial in software development because it ensures that your program behaves as expected. It helps you:

- **Catch bugs early** before they become harder (and more expensive) to fix.
- **Verify correctness** of your code, especially for critical logic.
- **Improve code quality** by forcing you to consider edge cases and handle unexpected input.
- **Boost confidence** in making future changes without breaking existing functionality.
- **Document behavior** — good tests describe what the code is supposed to do.

In short, testing is not just about finding bugs; it's about **building reliable and maintainable software**.

21. Difference Between Syntax Error, Runtime Error, and Logic Error

A. Syntax Error

- Occurs when your code violates the rules of the programming language.
- **Caught by the compiler** before the program runs.
- Example: Missing semicolon, unmatched brackets, wrong keywords.

B. Runtime Error

- Happens **during program execution** and causes it to crash or behave unexpectedly.
- Example: Dividing by zero, accessing null objects, or array index out of bounds.

C. Logic Error

- The program runs, but the **output is incorrect** because of flawed logic.
- These are the **hardest to catch**, and testing helps detect them.

22. How would you test a method that calculates the average of five numbers?

Assume the Method Looks Like This:

```
public class Calculator {  
    public static double calculateAverage(int a, int b, int c, int d, int e) {  
        return (a + b + c + d + e) / 5.0;  
    }  
}
```

You would need to write test cases for

- Normal inputs
- All Zeroes
- All Negative numbers
- All same values
- Mixed values
- Large numbers

In summary:

To test this method, supply a variety of inputs — positive, negative, zeros, mixed values — and check that the output is correct. You can do this manually with print statements or, ideally, by writing automated unit tests.

23. Why should Java developers write comments in their code?

Java developers should write comments in their code to:

1. Improve code readability – Comments help others (and your future self) understand what the code does.
 2. Explain complex logic – When logic is non-obvious, comments clarify the reasoning behind it.
 3. Facilitate maintenance – Clear documentation helps developers quickly locate and fix bugs or update code.
 4. Aid collaboration – In team environments, comments ensure everyone understands the code's purpose and functionality.
31. Build a command-line application that keeps track of student grades and allows adding, updating, and viewing records.

24. What are JavaDoc comments and how are they different from regular comments?

JavaDoc comments are special multi-line comments used to generate documentation for Java classes, methods, constructors, and fields. They start with `/**` and end with `*/`. Tools like javadoc can read these comments and produce HTML documentation automatically.

JavaDoc comments support special tags like:

- `@param` – describes method parameters
- `@return` – describes return value
- `@throws` – documents exceptions the method may throw
- `@see`, `@author`, `@version`, etc.

Regular comments don't support any tags and aren't used for generating documentation.

26. What is version control and why is it important in team projects?

Version control is a system that records changes to files over time so that you can:

- Track the history of changes
- Collaborate with others without overwriting each other's work
- Revert to previous versions if something breaks
- The most popular version control system is Git.

Why It's Important in Team Projects?

- Collaboration: Multiple developers can work on the same project simultaneously without conflict.
- Backup: Code history is preserved and can be restored if needed.
- Tracking Changes: You can see who made what change, when, and why (via commit messages).
- Branching and Merging: Teams can work on features independently and merge them safely into the main codebase.
- Accountability and Review: Makes peer review and auditing easier.

27. How would you explain the concept of “code refactoring” to a junior developer?

Code refactoring is the process of restructuring existing code without changing its behavior. The goal is to improve code readability, maintainability, and performance

How to Explain It to a Junior Developer:

“Imagine you wrote a story, and it works, but it’s a bit messy—some sentences are too long, there’s repetition, and the layout is hard to follow. Refactoring is like rewriting that story to make it cleaner and easier to understand, without changing what the story says. In code, that means making it simpler, more organized, and less error-prone—without changing how it runs.”

28. What tools can Java developers use to collaborate on large projects? Attach

screenshots of 3 examples.

Java developers can use a variety of tools to collaborate effectively on large projects. Here are some of the most commonly used:

A. Version Control Systems

- Git (with platforms like GitHub, GitLab, or Bitbucket):

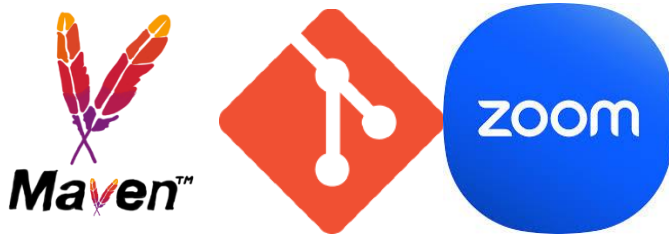
Enables collaborative coding, version tracking, and pull/merge requests.

B. Build Tools.

- Maven: Manages dependencies and project builds.
- Gradle: A powerful, flexible build system faster than Maven in some cases.

C. Communication Tools

- Slack, Microsoft Teams, Zoom: For team discussions, meetings, and quick collaboration.



29. Mention 5 best practices you follow when developing a Java program.

A. Follow Naming Conventions

Use meaningful names (camelCase for variables/methods, PascalCase for classes) to enhance clarity.

B. Write Modular Code

Break large programs into classes and methods with single responsibilities to improve maintainability.

C. Use Proper Exception Handling

Avoid empty catch blocks; handle or log exceptions meaningfully.

D. Comment and Document

Use JavaDoc for public APIs and inline comments where logic is non-obvious.

E. Avoid Magic Numbers and Hardcoding

F. Use constants (final variables) and configuration files to improve flexibility and understanding.

30. What is code readability, and why is it more important than “smart” code?

Code readability refers to how easily another developer (or your future self) can understand your code. It includes clear naming, consistent formatting, logical structure, and meaningful comments.

- Easier Maintenance: Readable code is simpler to debug, update, or extend.
- Team Collaboration: Other developers can understand and contribute quickly.
- Long-Term Value: Readable code is sustainable and lowers the learning curve.

"Smart" Code Can Be Confusing: Clever one-liners or over-optimized code might save lines but often hide intent.

31. Build a command-line application that keeps track of student grades and allow adding, updating, and viewing records.

```
1 package code31;
2
3 public class Student {
4     private String name;
5     private String matricNo;
6     private double score;
7
8     public Student(String name, String matricNo, double score) {
9         this.name = name;
10        this.matricNo = matricNo;
11        this.score = score;
12    }
13
14    public String getMatricNo() {
15        return matricNo;
16    }
17
18    public void setScore(double score) {
19        this.score = score;
20    }
21
22    public void displayInfo() {
23        System.out.println("Name: " + name);
24        System.out.println("Matric No: " + matricNo);
25        System.out.println("Score: " + score);
26        System.out.println("-----");
27    }
28 }
29
```

32. Write a program that simulates a basic ATM system (check balance, deposit, withdraw)

```

1
2 import java.util.Scanner;
3
4 public class ATM {
5     // Initial account balance
6     private static double balance = 1000.00;
7
8     public static void main(String[] args) {
9         Scanner scanner = new Scanner(System.in);
10        int choice;
11
12        System.out.println("Welcome to the ATM!");
13
14        // Menu loop
15        do {
16            // Display options
17            System.out.println("\nChoose an option:");
18            System.out.println("1. Check Balance");
19            System.out.println("2. Deposit Money");
20            System.out.println("3. Withdraw Money");
21            System.out.println("4. Exit");
22            System.out.print("Enter your choice: ");
23
24            choice = scanner.nextInt();
25
26            // Handle user selection
27            switch (choice) {
28                case 1:
29                    checkBalance();
30                    break;
31                case 2:
32                    deposit(scanner);
33                    break;
34                case 3:
35                    withdraw(scanner);
36                    break;
37                case 4:
38                    System.out.println("Thank you for using the ATM. Goodbye!");
39                    break;
40                default:
41                    System.out.println("Invalid choice. Please try again.");
42            }
43
44            } while (choice != 4); // Repeat until user exits
45
46        scanner.close();
47    }
48
49    // Display current balance
50    private static void checkBalance() {
51        System.out.printf("Your current balance is: $%.2f\n", balance);
52    }
53
54    // Handle deposit transaction
55    private static void deposit(Scanner scanner) {
56        System.out.print("Enter amount to deposit: $");
57        double amount = scanner.nextDouble();
58        if (amount > 0) {
59            balance += amount; // Update balance
60            System.out.printf("Deposit successful! New balance: $%.2f\n", balance);
61        } else {
62            System.out.println("Invalid amount. Deposit must be greater than zero.");
63        }
64    }
65
66    // Handle withdrawal transaction
67    private static void withdraw(Scanner scanner) {
68        System.out.print("Enter amount to withdraw: $");
69        double amount = scanner.nextDouble();
70        if (amount > 0 && amount <= balance) {
71            balance -= amount; // Deduct from balance
72            System.out.printf("Withdrawal successful! New balance: $%.2f\n", balance);
73        } else if (amount > balance) {
74            System.out.println("Insufficient funds.");
75        } else {
76            System.out.println("Invalid amount. Withdrawal must be greater than zero.");
77        }
78    }
79 }
80

```