Chapter 5: Loops

5.1. Introduction

- **Loops** are fundamental constructs in programming that enable the repeated execution of a sequence of statements.
- This chapter introduces the concept of loops and examines the different types of loops in Java, including the `while`, `do-while`, and `for` loops.

Need for Loops

- Writing the same block of code repeatedly is time-consuming and prone to errors.
- Loops simplify this process by automating repetitive tasks, making the code more efficient and easier to maintain.
- Loops are essential for processing collections of data, such as arrays or lists.

Example: Displaying a message like "Welcome to Java!" a hundred times can be efficiently done using a loop instead of writing the print statement a hundred times.

Types of Loops in Java

- `while` **Loop:** Checks the condition **before** running. Use it when the number of iterations is unknown and depends on a condition.
- `do-while` **Loop:** Runs the loop body **at least once** before checking the condition. Use it when the loop must execute at least once.
- `for` **Loop:** Combines initialization, condition, and update in one line. It is best used when the number of iterations is known.

Key Concepts

- **Loop Body:** The block of statements that are executed repeatedly.
- **Iteration:** A single execution of the loop body.
- **Loop-Continuation Condition:** A Boolean expression that determines whether the loop body will execute.

Example:

```
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}
```

Explain:

- The `while` loop executes the print statement and increments the `count` variable until `count` is no longer less than 100.
- The loop-continuation condition, `count < 100`, determines the number of iterations.
- The `count++` statement increments the `count` variable after each iteration.

Counter-Controlled Loop

- **Counter-controlled loop** is a loop that uses a counter to keep track of how many times it runs. It is useful when you already know how many times the loop should repeat.

Infinite Loops

- **Infinite Loops:** Loops that never stop because the exit condition is never met.
- **Why Avoid Them?** They can freeze or crash your program.
- **Prevention Tips:**
  - Ensure the loop condition eventually evaluates to `false`.
  - Verify your logic to avoid errors such as missing updates or incorrect conditions.

Tip: To avoid infinite loops, carefully test your loop conditions and ensure there is a clear and reliable way to exit the loop.

5.2. The `while` Loop

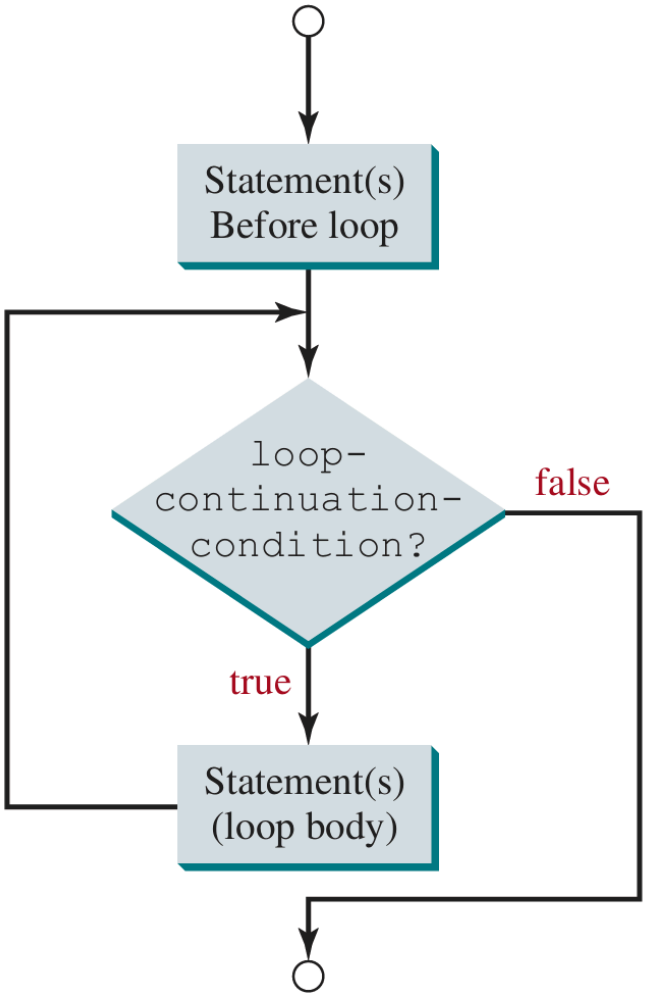
- **The `while` loop** is a fundamental concept in Java. It enables you to repeatedly execute a block of code as long as a specified condition evaluates to `true`.
- This makes it particularly useful for tasks where the number of repetitions depends on a condition rather than being predetermined.

Syntax: The syntax for a `while` loop is:

```
while (loop-continuation-condition) {
    // Loop body
    Statement(s);
}
```

Explain: The loop-continuation condition is evaluated before each iteration. If the condition is true, the loop body is executed.

Flowchart


Statement(s)
Before loop
loop-
continuation-
condition?
false
true
Statement(s)
(loop body)

Execution Process

- **Statements Before the Loop** are the blocks of code executed before the loop starts. These may include variable initialization or other setup tasks.
- **Loop-Continuation Condition** is evaluated first. If it is `true`, the loop body is executed. If it is `false`, the loop terminates.
- **Statements Inside the Loop Body** are the blocks of code executed repeatedly as long as the loop-continuation condition is `true`.

Example: Displaying "Welcome to Java!" a hundred times using a `while` loop:

```
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}
```

Explain: In this example, the loop body prints the message "Welcome to Java!" and increments the `count` variable until the `count` is no longer less than 100.

Counter-Controlled `while` Loop

Example: Displaying numbers from 1 to 10 using a `while` loop:

```
int i = 1;
while (i <= 10) {
    System.out.println(i);
    i++;
}
```

Explain:

- The loop-continuation condition `i <= 10` determines how many times the loop will execute.
- Inside the loop body, the current value of `i` is printed, and then `i` is incremented by 1 after each iteration.

Infinite `while` Loop

Example: Infinite loop:

```
while (true) {
    // Loop body
}
```

Note: This loop will run indefinitely because the loop-continuation condition `true` is always true.

Common Pitfalls

- **Off-By-One Error:** A common mistake where a loop executes either one iteration too many or one iteration too few. This typically occurs due to incorrect initialization, condition, or increment/decrement logic in the loop.
  Example: If the condition is `count <= 100` instead of `count < 100`, the loop will execute 101 times.
- **Infinite Loops:** Always ensure that the loop-continuation condition is designed to eventually evaluate to `false`. This prevents the loop from running indefinitely, which can cause the program to hang or crash.

5.3. Case Study: Guessing Numbers

Practice.

5.4. Loop Design Strategies

- **Designing effective loops** involves understanding the steps that need to be repeated and writing appropriate conditions to terminate the loop.
- Writing correct loops can be challenging for novice programmers, but following a clear strategy can help.

Three steps to help design a loop

Step 1: Identify the Statements to be Repeated

- Determine which statements need to be executed multiple times.

Example: Generating random numbers multiple times.

```
// Statements to be repeated
int number = (int) (Math.random() * 10) + 1;
```

Step 2: Wrap the Statements in a Loop

- Use a `while` loop to repeatedly execute the statements.

Example: Generating random numbers until the sum is greater than 100

```
while (true) {
    // Statements to be repeated
    int number = (int) (Math.random() * 10) + 1;
}
```

Step 3: Add Loop Control

- Code the loop-continuation condition and add statements to control the loop.

Example: Generating random numbers until the sum is greater than 100

```
int sum = 0;
while (sum <= 100) {
    // Generate a random number
    int number = (int) (Math.random() * 10) + 1;
    sum += number;
}
```

Example Multiple Subtraction Quiz

- Generates five subtraction questions and reports the number of correct answers.

Step 1: Identify the Statements to be Repeated:

- Generate two random numbers.
- Prompt the user with a subtraction question.
- Evaluate the user's answer.

```java
// Generate two random numbers
int number1 = (int) (Math.random() * 10);
int number2 = (int) (Math.random() * 10);

// Prompt the user with a subtraction question
System.out.print("What is " + number1 + " - " + number2 + "? ");
int answer = input.nextInt();

// Grade the answer
if (number1 - number2 == answer) {
    System.out.println("You are correct!");
} else {
    System.out.println("Your answer is wrong.");
    System.out.println(number1 + " - " + number2
        + " should be " + (number1 - number2));
}
```

Step 2: Wrap the Statements in a Loop:

- Use a loop to repeat the steps five times.

Step 3: Add a Loop Control Variable and Condition:

- Use a control variable to execute the loop five times.
- This program follows a loop design strategy to ensure correct execution and efficient control of the loop.

```
// Initialize the count of correct answers
int correctCount = 0;

// Repeat the quiz five times
int count = 0;

while (count < 5) {
    // Generate two random numbers
    int number1 = (int) (Math.random() * 10);
    int number2 = (int) (Math.random() * 10);

    // Prompt the user with a subtraction question
    System.out.print("What is " + number1 + " - " + number2 + "? ");
    int answer = input.nextInt();

    // Grade the answer
    if (number1 - number2 == answer) {
        System.out.println("You are correct!");
        correctCount++;
    } else {
        System.out.println("Your answer is wrong.");
        System.out.println(number1 + " - " + number2
            + " should be " + (number1 - number2));
    }

    // Increase the count
    count++;
}
```

5.5. Controlling a Loop with User Confirmation or a Sentinel Value

- **Controlling loops effectively** is crucial in programming to ensure they execute the correct number of times.
- This section explores how to use user confirmation or a sentinel value to control loop execution.

User Confirmation

- You can control a loop by prompting the user for confirmation after each iteration.
- This approach allows the user to decide whether to continue or terminate the loop.

Example: Using user confirmation to control a loop

```
char continueLoop = 'Y';
while (continueLoop == 'Y') {
    // Execute loop body
    ...
    // Prompt user for confirmation
    System.out.print("Enter Y to continue and N to quit: ");
    continueLoop = input.nextLine().charAt(0);
}
```

Note: This allows the user to decide whether to continue or terminate the loop.

Sentinel Value

- A sentinel value is a special value that indicates the termination of a loop.
- This technique is particularly useful when processing a sequence of input values.

Example: Reading input until a sentinel value is entered

```
int data;
int sum = 0;
System.out.print("Enter an integer (the input ends if it is 0): ");
data = input.nextInt();

while (data != 0) {
    sum += data;
    System.out.print("Enter an integer (the input ends if it is 0): ");
    data = input.nextInt();
}

System.out.println("The sum is " + sum);
```

Note: In this example, the sentinel value `0` signals the end of input, and the loop terminates.

Important Points

- **User Confirmation:** Enables dynamic control of loop termination based on user input.
- **Sentinel Value:** Uses a predefined condition to terminate the loop, commonly applied in data processing.

5.6. The `do-while` Loop

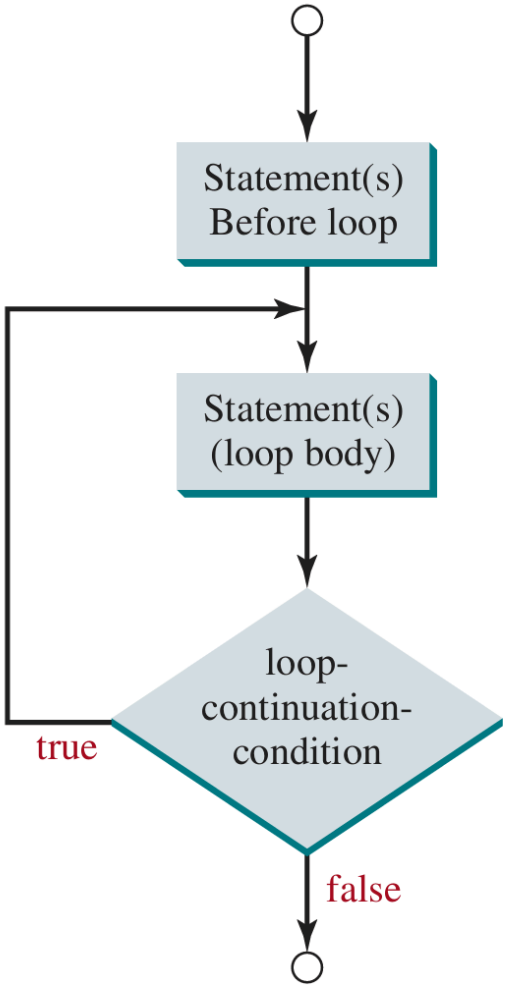
- The `do-while` **loop** is similar to the `while` loop, with one key difference: the loop body is executed at least once before the loop-continuation condition is evaluated.

Syntax: The syntax for a `do-while` loop

```
do {
    // Loop body
} while (loop-continuation-condition);
```

Note: The loop body is executed first, and then the loop-continuation condition is evaluated.

Flowchart:


Statement(s)
Before loop
Statement(s)
(loop body)
loop-
continuation-
condition
true
false

Key Components of `do-while` Loop

- **Statements Before the Loop** are the blocks of code executed before the loop starts.
- **Statements in the Loop Body** are the blocks of code executed repeatedly.
- **Loop-Continuation Condition** is a Boolean expression that determines whether the loop body will be executed.

Example: Basic structure of a `while` loop in Java:

```java
// Statements before the loop
int count = 0;

// Loop-continuation condition
while (count < 10) {
     // Statements in the loop body
     System.out.println("Count is: " + count);
     count++;
}
```

Explain:

- **Statements Before the Loop:** The variable `count` is initialized to `0`.
- **Loop-Continuation Condition:** The loop continues as long as `count < 10`.
- **Statements in the Loop Body:** The current value of `count` is printed, and then `count` is incremented by `1` after each iteration.

`while` vs. `do-while`

Feature	`while` Loop	`do-while` Loop
Condition Check	Before loop body.	After loop body.
Execution	May not execute if condition is false.	Executes at least once.
Use Case	When iterations depend on condition.	When loop must run at least once.

Example: Comparing `while` and `do-while` loops

- `while` loop (may not execute)

```
int i = 5;
while (i < 5) {
    // This will not execute
    System.out.println("Inside while loop");
    i++;
}
```

- `do-while` loop (executes at least once)

```
int i = 5;
do {
    // This will execute once
    System.out.println("Inside do-while loop");
    i++;
} while (i < 5);
```

5.7. The `for` Loop

Introduction

- The `for` **loop** is a concise and efficient construct for writing loops in Java.
- It integrates initialization, condition checking, and increment/decrement into a single line, making it particularly suitable for counter-controlled loops.

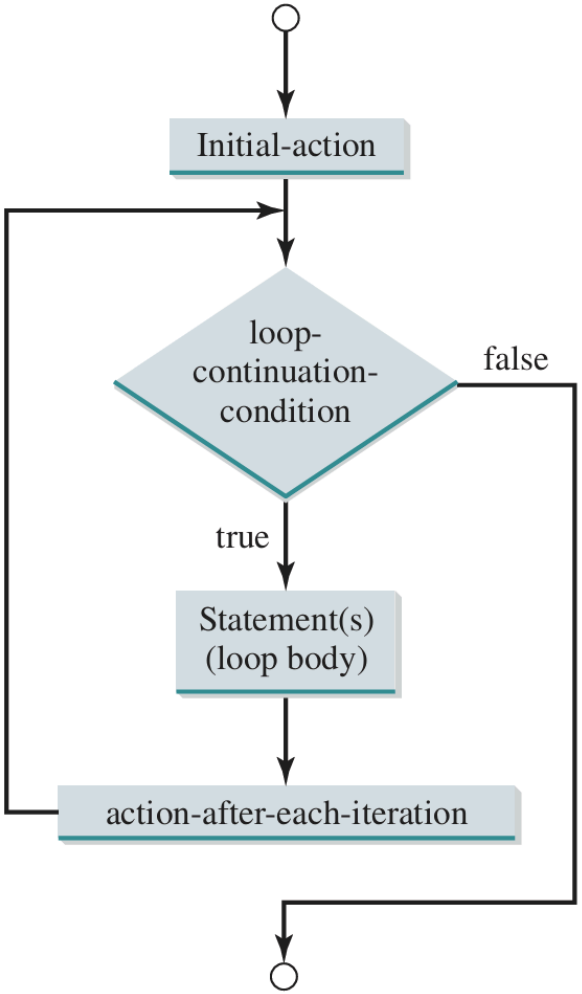
Syntax: The syntax for a `for` loop is:

```
for (initial-action;
    loop-continuation-condition;
    action-after-each-iteration
) {
    // Loop body
}
```

Note:

- The `initial-action` initializes the loop control variable.
- The `loop-continuation-condition` determines whether the loop body executes.
- The `action-after-each-iteration` updates the loop control variable.
- The loop body executes as long as the condition is true.

Flowchart


Initial-action
loop-
continuation-
condition
false
true
Statement(s)
(loop body)
action-after-each-iteration

- **Initial Action:** This is the code that runs before the loop begins. It typically initializes variables or sets up conditions required for the loop to execute.
- **Loop-Continuation Condition:** A Boolean expression that determines whether the loop should continue running. If the condition evaluates to `true`, the loop body executes; otherwise, the loop stops.
- **Statements in the Loop Body:** These are the instructions executed repeatedly as long as the loop-continuation condition remains `true`.
- **Action After Each Iteration:** This is the code that runs after each iteration of the loop body. It often updates variables or performs tasks necessary for the next iteration.

Example: Displaying "Welcome to Java!" a hundred times using a `for` loop:

```
for (int i = 0; i < 100; i++) {
    System.out.println("Welcome to Java!");
}
```

Note:

- The variable `i` starts at 0.
- The loop runs while `i < 100`, printing "Welcome to Java!" and incrementing `i` by 1.
- When `i` reaches 100, the condition becomes false, and the loop terminates.

Advantages

- **Concise and Clear:** Combines initialization, condition checking, and updates in a single line.
- **Reduces Errors:** Minimizes the risk of missing updates, which can lead to infinite loops.

Multiple Control Variables

- A `for` loop can have multiple control variables, separated by commas in the initialization and update sections.
- This feature allows you to use multiple variables to control the loop efficiently.

Example: Using multiple control variables in a `for` loop

```
for (int i = 0, j = 0; i + j < 10; i++, j++) {
    // Loop body
}
```

Infinite Loop

- In a `for` loop, you can create an infinite loop by omitting the initialization, condition, or update parts.

Syntax:

```
for (;;) {
    // Infinite loop
}
```

Note: It's better to use `while (true)` for clarity

Example: Summing numbers from 1 to 10 using a `for` loop:

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    sum += i;
}
System.out.println("The sum is " + sum);
```

5.8. Which Loop to Use?

Scenario	Recommended
Number of iterations is known in advance	`for` loop
Number of iterations is unknown	`while` loop
Loop body must execute at least once	`do-while` loop
Iterating over a collection or array	Enhanced `for` loop
Infinite loop	`while (true)` or `for (;;)`
Loop with multiple control variables	`for` loop
Loop controlled by user confirmation	`do-while` loop
Loop controlled by a sentinel value	`while` loop
Nested iterations (e.g., working with grids)	Nested `for` loops

Pretest vs. Posttest Loops

- **Pretest Loop**: A loop where the condition is evaluated before executing the loop body. Both `while` and `for` loops are pretest loops because the condition is checked prior to the execution of the loop body.
- **Posttest Loop**: A loop where the condition is evaluated after executing the loop body. The `do-while` loop is a posttest loop because the condition is checked after the loop body has been executed.

Expressive Equivalence

- The `while`, `do-while`, and `for` loops are expressively equivalent, meaning you can write a loop in any of these three forms.

General Guidance

Choose the loop type that feels easiest for you to use.

- `for` **Loop:** Best when you know how many times the loop should run, such as printing a message 100 times.
- `while` **Loop:** Best when the number of iterations depends on a condition, such as reading numbers until the user enters 0.
- `do-while` **Loop:** Best when the loop must execute at least once, such as asking for user input.

Common Mistakes

- Avoid placing a semicolon immediately after the `for` statement, as it will prevent the loop from executing correctly.
- Ensure the loop condition eventually evaluates to `false` to avoid infinite loops.

5.9. Nested Loops

- You can place one loop inside another loop.
- This is called a nested loop and is useful for handling more complex tasks step by step.

Definition

- Nested loops consist of an outer loop and one or more inner loops.
- Each time the outer loop repeats, the inner loops are re-entered and start anew.

Multiplication Table

Example: Multiplication Table Program

```
public class MultiplicationTable {
    public static void main(String[] args) {
        System.out.println("Multiplication Table");
        System.out.print("    ");
        for (int j = 1; j <= 9; j++) {
            System.out.print("   " + j);
        }
        System.out.println("\n———————————————");
        for (int i = 1; i <= 9; i++) {
            System.out.print(i + " | ");
            for (int j = 1; j <= 9; j++) {
                System.out.printf("%4d", i * j);
            }
            System.out.println();
        }
    }
}
```

Explain:

- The outer loop `for (int i = 1; i <= 9; i++)` iterates through the rows of the table.
- The inner loop `for (int j = 1; j <= 9; j++)` iterates through the columns of the table.
- The product of `i` and `j` is calculated and printed in a formatted manner.
- The program generates a multiplication table ranging from 1 to 9.

Performance Consideration

- Nested loops can slow down a program because they significantly increase the number of iterations.
- For instance, if you have three nested loops, each running 100 times, the code will execute a total of 1 million iterations.

Practical Tips

- Use nested loops when working with grids, tables, or multi-level data.
- Clearly name each loop's control variable to avoid confusion.
- Limit the number of iterations in nested loops to maintain program efficiency.

5.10. Minimizing Numeric Errors

Introduction

- Minimizing numeric errors in loops is essential to ensure accurate results when working with floating-point numbers.
- This section explores strategies to reduce errors when handling floating-point numbers in loops.

Floating-Point Arithmetic

- Floating-point numbers are represented approximately in computers, which can lead to precision errors.

Example: Adding numbers from 0.01 to 1.0 using a `for` loop. The exact sum should be 50.50, but using the `float` type might result in an imprecise value, such as 50.499985.

```
float sum = 0;
for (float i = 0.01f; i <= 1.0f; i += 0.01f) {
    sum += i;
}
System.out.println("The sum is " + sum);
```

output:

```
The sum is 50.499985
```

Precision Improvement

- Using a double type instead of float might improve precision slightly, but still not perfect.

Example: Using double type:

```
double sum = 0;
for (double i = 0.01; i <= 1.0; i += 0.01) {
    sum += i;
}
System.out.println("The sum is " + sum);
```

Note: Even with double, the result might not be exact (e.g., 49.50000000000003) due to floating-point arithmetic.

Solution Using Integer Count

- Use an integer counter to ensure all numbers are added to the sum precisely.

Example: Using integer count:

```
double sum = 0;
double currentValue = 0.01;
for (int count = 0; count < 100; count++) {
    sum += currentValue;
    currentValue += 0.01;
}
System.out.println("The sum is " + sum);
```

Output:

```
The sum is 50.5
```

Adding in Different Orders

- Adding numbers from smallest to largest is more accurate than adding from largest to smallest.

Example: Adding from largest to smallest:

```
double sum = 0;
double currentValue = 1.0;
for (int count = 0; count < 100; count++) {
    sum += currentValue;
    currentValue -= 0.01;
}
System.out.println("The sum is " + sum);
```

Note: This method might result in less accuracy due to finite-precision arithmetic.

5.11. Case Studies

Practice.

5.12. Keywords `break` and `continue`

The `break` Keyword

- The `break` keyword is used to stop a loop right away.
- It is helpful when you want to end the loop as soon as a certain condition is true.

Example: Using `break` to terminate a loop:

```
while (number < 20) {
    if (sum >= 100) {
        break;
    }
    number++;
    sum += number;
}                                                    .
```

Explain: This code adds numbers from 1 to 20 until the sum reaches 100, using `break` to stop the loop.

The `continue` Keyword

- The `continue` keyword skips the rest of the current loop and moves to the next iteration. It ignores the remaining code in the loop for that iteration.

Example: Using `continue` to skip an iteration:

```
while (number < 20) {
    number++;
    if (number == 10 || number == 11) {
        continue;
    }
    sum += number;
}                                                    .
```

Explain: This code adds numbers from 1 to 20 but skips numbers 10 and 11 using `continue`.

Differences Between `break` and `continue`

- `break` completely stops the loop, while `continue` skips the rest of the current iteration and moves to the next one.
- You can use `continue` in both `while` and `for` loops.
- In `while` and `do-while` loops, `continue` immediately checks the loop condition after it is used.
- In `for` loops, `continue` executes the update step (e.g., `i++`) before checking the loop condition.

Note:

- While `break` and `continue` can simplify coding in certain situations, excessive or improper use can make your program harder to understand and debug.

Example: Using `break` in loops

```
int number = 0;
int sum = 0;

while (number < 20) {
    number++;
    sum += number;

    if (sum >= 100) {
        break;
    }
}

System.out.println("The number is " + number);
System.out.println("The sum is " + sum);
```

Example: Using `continue` in loops

```
int number = 0;
int sum = 0;

while (number < 20) {
    number++;
    if (number == 10 || number == 11) {
        continue;
    }
    sum += number;
}

System.out.println("The sum is " + sum);
```

5.13. Case Study: Checking Palindromes

Practice.

5.14. Case Study: Displaying Prime Numbers

Practice.