do-while Loop

The do-while Loop

Key difference from `while`**:** body executes **at least once** before condition check.

Syntax:

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

Best use case: input validation (user must enter data at least once).

do-while guarantees at least one execution.

Example 1

Task:

Create a program that asks the user to input integers until they enter 0. After entering 0, the program should display the sum of all entered numbers (0 is not included in the sum). Use a do-while loop.

Expected Output:

Enter a number (0 to exit): 5
Enter a number (0 to exit): 3
Enter a number (0 to exit): -2
Enter a number (0 to exit): 0
Sum of entered numbers: 6.

```java
import java.util.Scanner;

public class SumNumbers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int sum = 0;
        int input;

        do {
            System.out.print("Введите число (0 для выхода): ");
            input = scanner.nextInt();
            if (input != 0) {
                sum += input;
            }
        } while (input != 0);

        System.out.println("Сумма введённых чисел: " + sum);
        scanner.close();
    }
}
```

Expected Output:

Enter a number (0 to exit): 5
Enter a number (0 to exit): 3
Enter a number (0 to exit): -2
Enter a number (0 to exit): 0
Sum of entered numbers: 6.

Example 2

Task:
Write a program that asks the user to guess a secret number (7). The program should use a **do-while** loop so that the user is asked at least once, even if they guess correctly on the first try. Keep asking until they guess correctly.

Expected Output:

Guess the secret number (1-10): 4
Wrong. Try again: 8
Wrong. Try again: 7
Correct! Well done.

```java
import java.util.Scanner;

public class DoWhileGuess {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int secret = 7;
        int guess;

        do {
            System.out.print("Guess the secret number (1-10): ");
            guess = scanner.nextInt();

            if (guess != secret) {
                System.out.print("Wrong. ");
            }
        } while (guess != secret);

        System.out.println("Correct! Well done.");
        scanner.close();
    }
}
```

Expected Output:

Guess the secret number (1-10): 4
Wrong. Try again: 8
Wrong. Try again: 7
Correct! Well done.

Exception Handling in Loops

Exception Handling in Loops

Why use exceptions?

- Prevent program crashes on invalid input.
- Handle errors gracefully.

Key components:

- `try` block: code that might throw an exception.
- `catch` block: handles specific exceptions.

Common exception: `InputMismatchException` (invalid data type).

Example 1

Task:

Write a program that asks user to enter an integer and checks if the number is even. If the user enters text (non-integer), the program should catch the exception and output a message.

```
Enter an integer: hello
Invalid input. Please enter a number.
Enter an integer: 4
The number is even: true.
```

Example 1 – common solution

```
import java.util.Scanner;


public class Test {


    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);


        int s = scanner.nextInt();
        System.out.println("number is even: " + (s % 2 == 0));


        scanner.close();
    }
}
```

Exception Handling — Example 1

Single-attempt input with recovery:

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class Test {
   public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            int s = scanner.nextInt();
            System.out.println("number is even: " + (s%2 == 0));
        } catch (InputMismatchException e) {
            System.out.println("Wrong input. Enter a valid integer.");
        }
        scanner.close();
    }
}
```

1.Valid input (6):

`int s = scanner.nextInt(6)` → 6.

Output: 123 OK.

2.Invalid input ("abc"):

`int s = scanner.nextInt("abc")` throws `InputMismatchException`

Catch block executes →

Output: `Wrong input. Please enter a valid integer.`

Example 1

Task:

Write a program that asks user to enter an integer and checks if the number is even. If the user enters text (non-integer), the program should catch the exception **and keep asking until a valid integer is entered.**

```
Enter an integer: hello
Invalid input. Please enter a number.
Enter an integer: 4
The number is even: true.
```

Exception Handling — Example 2

Robust input loop with try-catch**:**

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

class Test {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        boolean flag = false;
        int s = 0;

        do {
            try {
                System.out.println("Enter an integer:");
                s = scanner.nextInt();
                System.out.println("Number is even: " + (s % 2 == 0));
                flag = true;
            } catch (InputMismatchException e) {
                System.out.println("Wrong input. Enter a valid integer.");
                scanner.nextLine(); // clear the buffer from invalid input
            }
        } while (!flag);

        scanner.close();
    }
}
```

Best Practices for Loops and Exception Handling

Rules for robust loops:

1. **Always update condition variables** in `while` loops to avoid infinite loops.
2. **Choose sentinel values carefully** — ensure they don't conflict with valid data.
3. **Use** `do-while` **for guaranteed execution** (e.g., user prompts).
4. **Close resources** (`scanner.close()`) after use.

Exception handling tips:

- Catch specific exceptions (e.g., `NumberFormatException`, `InputMismatchException`).
- Provide clear error messages to guide users.
- Avoid empty `catch` blocks — always handle or log errors.

Common Pitfalls and Solutions

Problem 1: Infinite `while` loop

```
// Pitfall: missing update
String answer = scanner.nextLine();
while (!answer.equals("quit")) {
    // No new input → loop never exits
}
```

Fix: add
`answer = scanner.nextLine();`
inside the loop body.

Problem 2: Unhandled InputMismatchException

```
int num = scanner.nextInt(); // Crashes on non-integer input
```

Fix: wrap in `try-catch`:

```
try {
    int num = scanner.nextInt();
} catch (InputMismatchException e) {
    System.out.println("Invalid input. Enter a number.");
    scanner.nextLine(); // Clear invalid input
}
```

Summary

Key takeaways:

- **`while` loops**: use for dynamic conditions (unknown iterations).
- **`do-while` loops**: guarantee at least one execution (ideal for input validation).
- **Sentinel-controlled loops**: require a unique terminal value.
- **Exception handling**: use `try-catch` to prevent crashes and guide users.

Break and Continue Keywords

The **break** Keyword

Purpose: immediately terminate the **entire loop** when encountered.

Syntax pattern:

```
while (condition) {
    if (someCondition) {
        break; // Exit loop entirely
    }
    // Remaining statements
}
```

Critical behavior:

- Skips all remaining loop iterations.
- Control transfers to the first statement after the loop.

The **continue** Keyword

Purpose: skip the **current iteration** and proceed to the next loop cycle.

Syntax pattern:

```
for (int i = 0; i < 10; i++) {
    if (skipCondition) {
        continue; // Jump to loop update
    }
    // Executed only if not skipped
}
```

Critical behavior:

- In `while/do-while`: condition re-evaluated immediately.
- In `for`: iterator updates first, then condition checked.

Example 1

Task:

Write a Java program that prints numbers from 1 to 10. However:

- Skip printing the number 5 (use **continue**)
- Stop the loop completely when reaching 8 (use **break**)

Expected Output:

```
1
2
3
4
6
7
```

Solution

```
public class BreakContinue {
    public static void main(String[] args) {
        int i = 1;

        while (i <= 10) {
            if (i == 5) {
                i++;
                continue; // Skip number 5
            }
            if (i == 8) {
                break; // Stop the loop at 8
            }
            System.out.println(i);
            i++;
        }
    }
}
```

Expected Output:
1
2
3
4
6
7

Example2

To do: sum numbers 1–20 until sum ≥ 100.

Expected Output:

The number is 14
The sum is 105

Solution

```
public class TestBreak {
    public static void main(String[] args) {
        int sum = 0, number = 0;
        while (number < 20) {
            number++;
            sum += number;
            if (sum >= 100)
                break; // Loop terminates here
        }
        System.out.println("The number is " + number);
        System.out.println("The sum is " + sum);
    }
}
```

Expected Output:

The number is 14
The sum is 105

Example

To do: sum 1–20 excluding 10 and 11. (Full sum (1–20): 210)

```
public class TestContinue {
    public static void main(String[] args) {
        int sum = 0, number = 0;
        while (number < 20) {
            number++;
            if (number == 10 || number == 11)
                continue; // Skip this iteration
            sum += number;
        }
        System.out.println("The sum is " + sum); // Output: 189
    }
}
```

When to Use break

Appropriate use cases:

1. **Search algorithms** (find first match).
2. **Error handling** (abort on critical condition).
3. **User input validation** (exit on “quit”).

Example: Find smallest divisor

To do: Find smallest divisor.

Complete twice – with and without break

```
n = 15
sd =  3
```

```
int factor = 2;
boolean found = false;
while (factor <= n && !found) {
    if (n % factor == 0) found = true;
    else factor++;
}
```

```java
int n = 15;
int factor = 2;
while (factor <= n) {
    if (n % factor == 0)
        break;
    factor++;
}
System.out.println("Smallest divisor: " + factor); // Output: 3
```

When to Avoid break/continue

Red flags:

- Multiple `break` statements in one loop → hard to trace.
- `break` in deeply nested loops → unpredictable flow.
- Using `continue` to skip large code blocks → better to restructure.

Refactoring tip: replace with boolean flags or helper methods when:

- Loop has > 2 `break` points.
- Logic becomes “spaghetti-like”.

break/continue **with** for **Loops**

Special behavior in `for`**:**

`continue` → updates iterator **before** checking condition.

Example: skip even numbers

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0)
        continue;
    System.out.print(i + " "); // Output: 1 3 5 7 9
}
```

Step-by-step:

1. `i=1` → odd → print.
2. `i=2` → even → `continue` → `i++` → `i=3` → print.
3. Repeat until `i=10`.