**Java Loops**

Loops in programming allow a set of instructions to run multiple times based on a condition. In Java, there are three types of loops, each with its own syntax and method of checking the condition, but fundamentally, they all serve the same purpose.

**Loops in Java**

In Java, there are three types of Loops, which are listed below:

* for loop
* while loop
* do-while loop

**1. for loop**

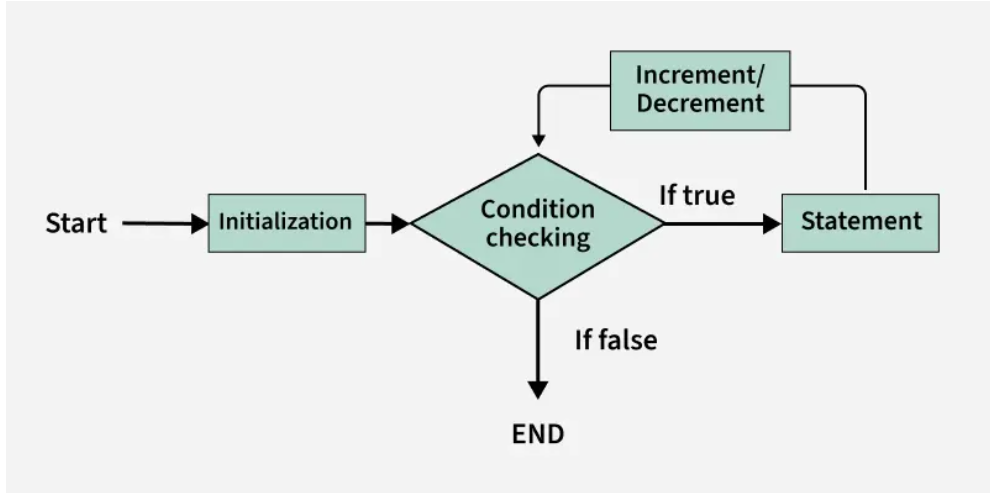
The for loop is used when we know the number of iterations (we know how many times we want to repeat a task). The for statement includes the initialization, condition, and increment/decrement in one line.

**Syntax:**

*for (initialization; condition; increment/decrement) {*

*// code to be executed*

*}*



Flowchart of for -loop

* **Initialization condition**: Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
* **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an Entry Control Loop as the condition is checked prior to the execution of the loop statements.
* **Statement execution**: Once the condition is evaluated to true, the statements in the loop body are executed.
* **Increment/ Decrement**: It is used for updating the variable for next iteration.
* **Loop termination**:When the condition becomes false, the loop terminates marking the end of its life cycle.

**Example:** The below Java program demonstrates a for loop that prints numbers from 0 to 10 in a single line.

// Java program to demonstrates the working of for loop

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

for (int i = 0; i <= 10; i++) {

System.out.print(i + " ");

}

}

}

**Output**

0 1 2 3 4 5 6 7 8 9 10

***Note****: There is another form of the for loop known as* ***Enhanced for loop*** *or (for each loop).*

**Enhanced for loop (for each)**

This loop is used to iterate over arrays or collections.

**Syntax:**

*for (dataType variable : arrayOrCollection) {*

*// code to be executed*

*}*

**Example**: The below Java program demonstrates an Enhanced for loop (for each loop) to iterate through an array and print names.

// Java program to demonstrate

// the working of for each loop

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

String[] names = { "Sweta", "Gudly", "Amiya" };

​

for (String name : names) {

System.out.println("Name: " + name);

}

}

}

**Output**

Name: Sweta

Name: Gudly

Name: Amiya

**2. while Loop**

A while loop is used when we want to check the condition before executing the loop body.

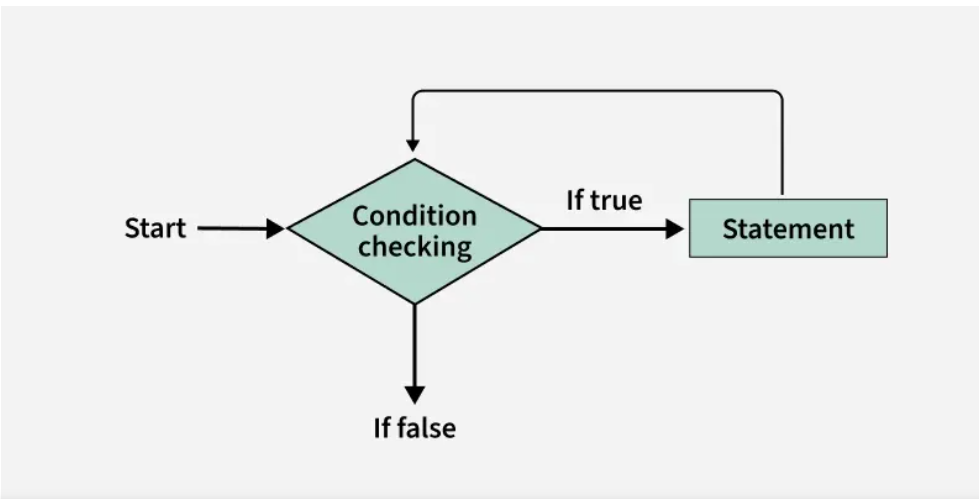
**Syntax:**

*while (condition) {*

*// code to be executed*

*}*

The below image demonstrates the flow chart of a while loop:



Flowchart of while-loop

* While loop starts with the checking of Boolean condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called Entry control loop
* Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
* When the condition becomes false, the loop terminates which marks the end of its life cycle.

**Example:** The below Java program demonstrates a while loop that prints numbers from 0 to 10 in a single line.

// Java program to demonstrates

// the working of while loop

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

int i = 0;

while (i <= 10) {

System.out.print(i + " ");

i++;

}

}

}

**Output**

0 1 2 3 4 5 6 7 8 9 10

**3. do-while Loop**

The do-while loop ensures that the code block executes **at least once**before checking the condition.

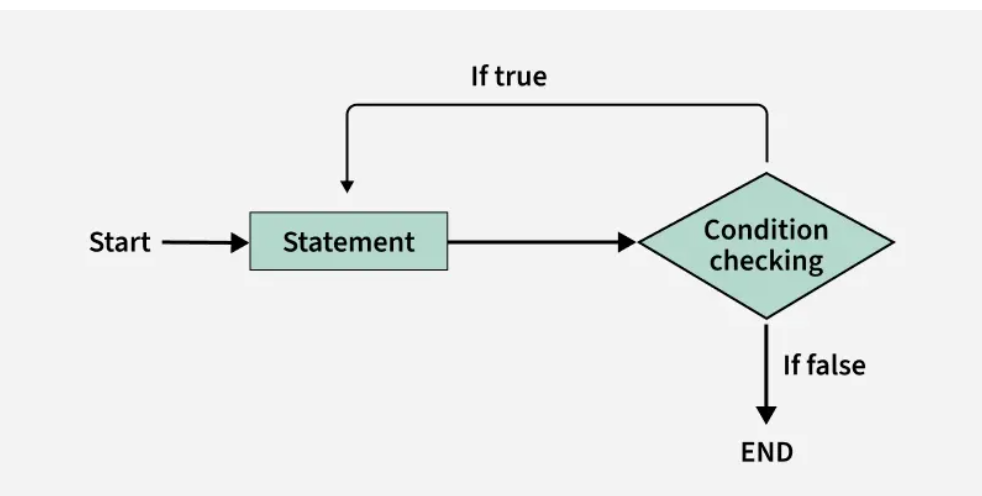
**Syntax:**

*do {*

*// code to be executed*

*} while (condition);*

The below image demonstrates the flow chart of a do-while loop:



Flowchart of do-while loop

* do while loop starts with the execution of the statement. There is no checking of any condition for the first time.
* After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
* When the condition becomes false, the loop terminates which marks the end of its life cycle.
* It is important to note that the do-while loop will execute its statements a tleast once before any condition is checked, and therefore is an example of exit control loop.

**Example**: The below Java program demonstrates a do-while loop that prints numbers from 0 to 10 in a single line.

// Java program to demonstrates

// the working of do-while loop

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

int i = 0;

do {

System.out.print(i + " ");

i++;

} while (i <= 10);

}

}

**Output**

0 1 2 3 4 5 6 7 8 9 10

**Common Loop Mistakes and How to Avoid them**

If loops are not used correctly, they can introduce pitfalls and bugs that affect code performance, readability, and functionality. Below are some common pitfalls of loops:

**1. Infinite Loops**

This is one of the most common mistakes while implementing any sort of looping is that it may not ever exit, that is the loop runs for infinite time. This happens when the condition fails for some reason.

**Types of Infinite Loops:**

* infinite for Loop
* infinite while Loop

**Example:** Here, both the examples demonstrates the infinite loops.

*// Java program to demonstrate*

*// the infinite for loop*

**import** **java.io.\***;

**class** **Geeks** {

**public** **static** void main(String[] args)

{

**for** (int i = 0; i < 5; i--) {

System.out.println(

"This loop will run forever");

}

}

}

**Output:** When you run both the above codes you will get TLE (Time Limit Exceeded) error.

**2. Off-by-One Errors**

Off-by-One Errors are caused when the loop runs one more or one fewer time than you wanted. It basically happens when the loop condition is not set correctly.

**Example**: The below Java program demonstrates an Off-by-One Error, where the loop runs 6 times and we expected it to run 5 times.

// Java Program to demonstrates Off-by-One Errors

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

for (int i = 0; i <= 5; i++) {

System.out.print(i + " ");

}

}

}

**3. Modifying Loop Variables Inside the Loop**

When we change the loop condition (like i) inside the loop, it can cause the loop to skip certain iterations or behave in ways that we did not expected. This might leads to errors or unexpected behavior.

**Example**: The below Java program demonstrates modifying the loop variable inside the loop, which cause the loop to skip certain iterations and behave unexpected.

// Java program demonstrates

// modification in i variable

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

for (int i = 0; i < 5; i++) {

if (i == 2) {

// Modifies the loop variable and skips

// the next iteration

i++;

}

System.out.println(i);

}

}

}

**4. Empty Loop Body**

An empty loop body occurs when a loop is written to iterate but does not perform any operations inside the loop. Running a loop without any useful operations inside it can be confusing.

**Example:** The below Java program demonstrate Empty loop body.

// Java program to demonstrates Empty loop body

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

for (int i = 0; i < 10; i++) {

// Empty body no operations

}

}

}

No output will be generated for this, because the body of the loop is empty.

**Summary Table**

| **Loop Type** | **When to Use** | **Condition Checking** | **Executes At Least Once?** |
| --- | --- | --- | --- |
| **for loop** | When you want exact iterations | Before loop body, It is called Entry-controlled. | no |
| **while loop** | When you need condition check first. | Before loop body, It is called Entry-controlled. | no |
| **do-while loop** | When you need to run at least once | After loop body, It is called Exit-controlled. | yes |
| **for-each loop** | When you process all collection items | Internally handled | no |

# Decision Making in Java (if-else, switch, break, continue, jump)

**Last Updated :**18 Jul, 2025

Decision-making in programming is similar to decision-making in real life. In programming, we also face situations where we want a certain block of code to be executed when some condition is fulfilled.

A programming language uses control statements to control the flow of execution of a program based on certain conditions.. Java provides several control statements to manage program flow, including:

* **Conditional Statements:** if, if-else, nested-if, if-else-if
* **Switch-Case:**For multiple fixed-value checks
* **Jump Statements:** break, continue, return

## Types of Decision-Making Statements

* [if](https://www.geeksforgeeks.org/java-if-statement-with-examples/)
* [if-else](https://www.geeksforgeeks.org/java-if-else-statement-with-examples/)
* [nested-if](https://www.geeksforgeeks.org/nested-if-in-java/)
* if-else-if
* [switch-case](https://www.geeksforgeeks.org/switch-statement-in-java/)
* [jump - break, continue, return](https://www.geeksforgeeks.org/jump-statements-in-java/)

**The table below demonstrates various control flow statements in programming, their use cases, and examples of their syntax.**

| **Statement** | **Use Case** | **Example** |
| --- | --- | --- |
| **if** | Single condition check | if (age >= 18) |
| **if-else** | Two-way decision | if (x > y) {...} else {...} |
| **nested-if** | Multi-level conditions | if (x > 10) { if (y > 5) {...} } |
| **if-else-if** | Multiple conditions | if (marks >= 90) {...} else if (marks >= 80) {...} |
| **switch-case** | Exact value matching | switch (day) { case 1: ... } |
| **break** | Exit loop/switch | break; |
| **continue** | Skip iteration | continue; |
| **return** | Exit method | return result; |

### ****1. Java if Statement****

The**if statement** is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e. if a certain condition is true then a block of statements is executed otherwise not.

**Syntax**:

*if(condition) {*

*// Statements to execute if*

*// condition is true*

*}*

Here, the **condition** after evaluation will be either true or false. if statement accepts boolean values - if the value is true then it will execute the block of statements under it. If we don't use curly braces( {} ), only the next line after the if is considered as part of the if block For example,

*if (condition) // Assume condition is true*

*statement1; // Belongs to the if block*

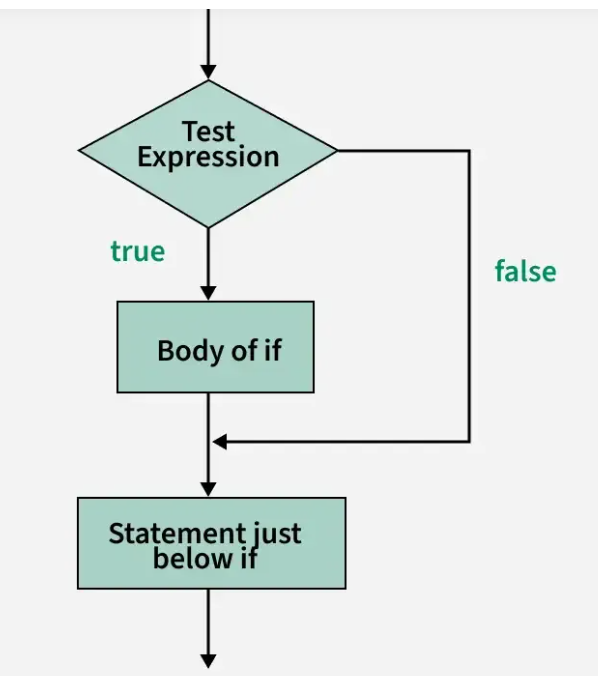
*statement2; // Does NOT belong to the if block*

Here's what happens:

* If the condition is True statement1 executes.
* statement2 runs no matter what because it's not a part of the if block

#### if Statement Execution Flow

The below diagram demonstrates the flow chart of an "if Statement execution flow" in programming.



**Example:**The below Java program demonstrates without curly braces, only the first line after the if statement is part of the if block and the rest code will be execute independently.

// Java program to illustrate

// if statement without curly block

import java.util.\*;

​

class Geeks {

public static void main(String args[])

{

int i = 10;

​

if (i < 15)

// part of if block(immediate one statement

// after if condition)

System.out.println("Inside If block");

​

// always executes as it is outside of if block

System.out.println("10 is less than 15");

​

// This statement will be executed

// as if considers one statement by default again

// below statement is outside of if block

System.out.println("I am Not in if");

}

}

**Output**

Inside If block

10 is less than 15

I am Not in if

### ****2. Java if-else Statement****

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false? Here, comes the "else" statement. We can use the else statement with the if statement to execute a block of code when the condition is false.

**Syntax**:

*if(condition){*

*// Executes this block if*

*// condition is true*

*}else{*

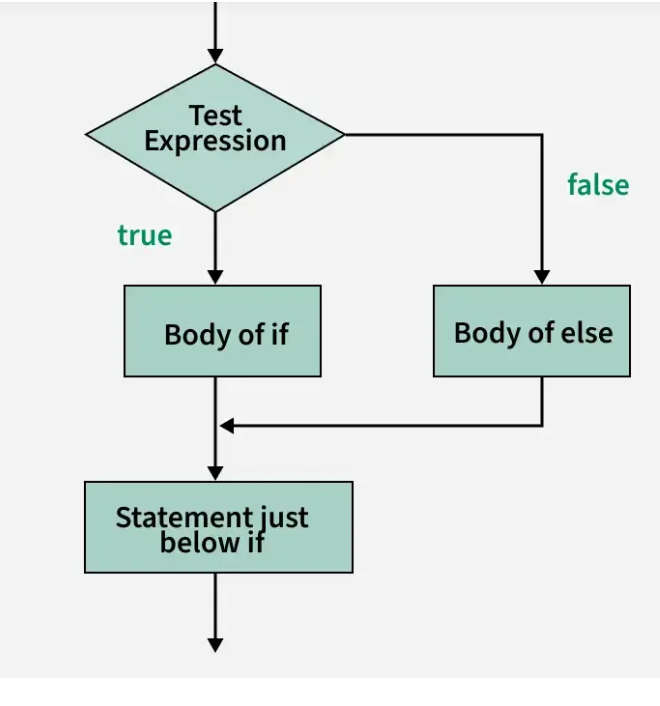
*// Executes this block if*

*// condition is false*

*}*

#### if-else Statement Execution flow

The below diagram demonstrates the flow chart of an "if-else Statement execution flow" in programming



**Example:**The below Java program demonstrates the use of if-else statement to execute different blocks of code based on the condition.

// Java program to demonstrate

// the working of if-else statement

import java.util.\*;

​

class Geeks {

public static void main(String args[])

{

int i = 10;

​

if (i < 15)

System.out.println("i is smaller than 15");

else

System.out.println("i is greater than 15");

}

}

**Output**

i is smaller than 15

### ****3.****Java ****nested-if Statement****

A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

**Syntax:**

*if (condition1) {*

*// Executes when condition1 is true*

*if (condition2)*

*{*

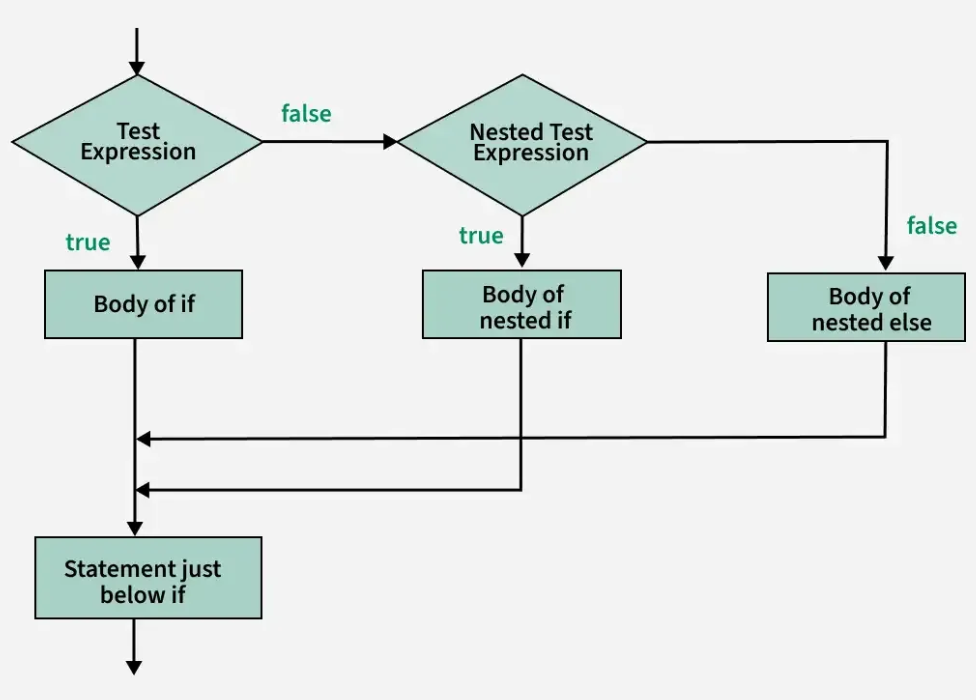
*// Executes when condition2 is true*

*}*

*}*

#### nested-if Statement Execution Flow

The below diagram demonstrates the flow chart of an "nested-if Statement execution flow" in programming.



**Example:**The below Java program demonstrates the use of nested if statements to check multiple conditions.

// Java program to demonstrate the

// working of nested-if statement

import java.util.\*;

​

class Geeks {

public static void main(String args[])

{

int i = 10;

​

if (i == 10 || i < 15) {

// First if statement

if (i < 15)

System.out.println("i is smaller than 15");

​

// Nested - if statement

// Will only be executed if statement above

// it is true

if (i < 12)

System.out.println(

"i is smaller than 12 too");

}

else {

System.out.println("i is greater than 15");

}

}

}

**Output**

i is smaller than 15

i is smaller than 12 too

### ****4. Java if-else-if ladder****

Here, a user can decide among multiple options.The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that 'if' is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. There can be as many as 'else if' blocks associated with one 'if' block but only one 'else' block is allowed with one 'if' block.

**Syntax:**

*if (condition1) {*

*// code to be executed if condition1 is true*

*} else if (condition2) {*

*// code to be executed if condition2 is true*

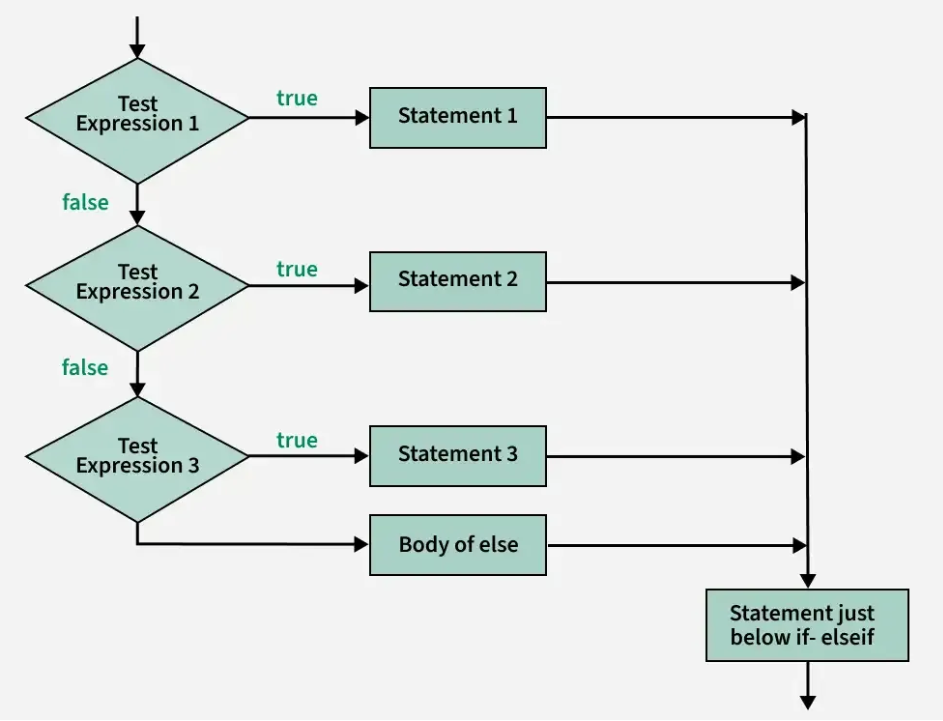
*} else {*

*// code to be executed if all conditions are false*

*}*

#### if-else-if ladder Execution Flow

The below diagram demonstrates the flow chart of an "if-else-if ladder execution flow" in programming



**Example:**This example demonstrates an if-else-if ladder to check multiple conditions and execute the corresponding block of code based on the value of I.

//Java program to demonstrate the

// working of if-else-if ladder

import java.util.\*;

​

class Geeks {

public static void main(String args[])

{

int i = 20;

​

if (i == 10)

System.out.println("i is 10");

else if (i == 15)

System.out.println("i is 15");

else if (i == 20)

System.out.println("i is 20");

else

System.out.println("i is not present");

}

}

**Output**

i is 20

### ****5. Java Switch Case****

The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

**Syntax:**

*switch (expression) {*

*case value1:*

*// code to be executed if expression == value1*

*break;*

*case value2:*

*// code to be executed if expression == value2*

*break;*

*// more cases...*

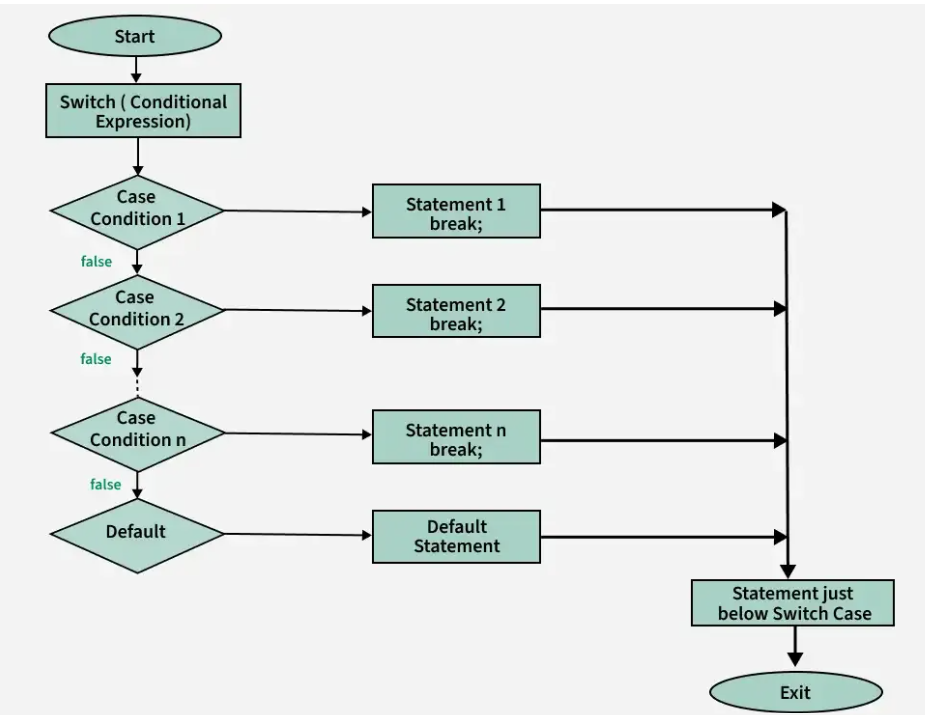
*default:*

*// code to be executed if no cases match*

*}*

#### switch Statements Execution Flow

The below diagram demonstrates the flow chart of a "switch Statements execution flow" in programming.



**Example:**The below Java program demonstrates the use of switch-case statement to evaluate multiple fixed values.

// Java program to demonstrates the

// working of switch statements

import java.io.\*;

​

class Geeks {

public static void main(String[] args)

{

int num = 20;

switch (num) {

case 5:

System.out.println("It is 5");

break;

case 10:

System.out.println("It is 10");

break;

case 15:

System.out.println("It is 15");

break;

case 20:

System.out.println("It is 20");

break;

default:

System.out.println("Not present");

}

}

}

**Output**

It is 20

* The expression can be of type byte, short, int char, or an enumeration. Beginning with JDK7, the expression can also be of type String.
* Duplicate case values are not allowed.
* The default statement is optional.
* The break statement is used inside the switch to terminate a statement sequence.
* The break statements are necessary without the break keyword, statements in switch blocks fall through.
* If the break keyword is omitted, execution will continue to the next case.

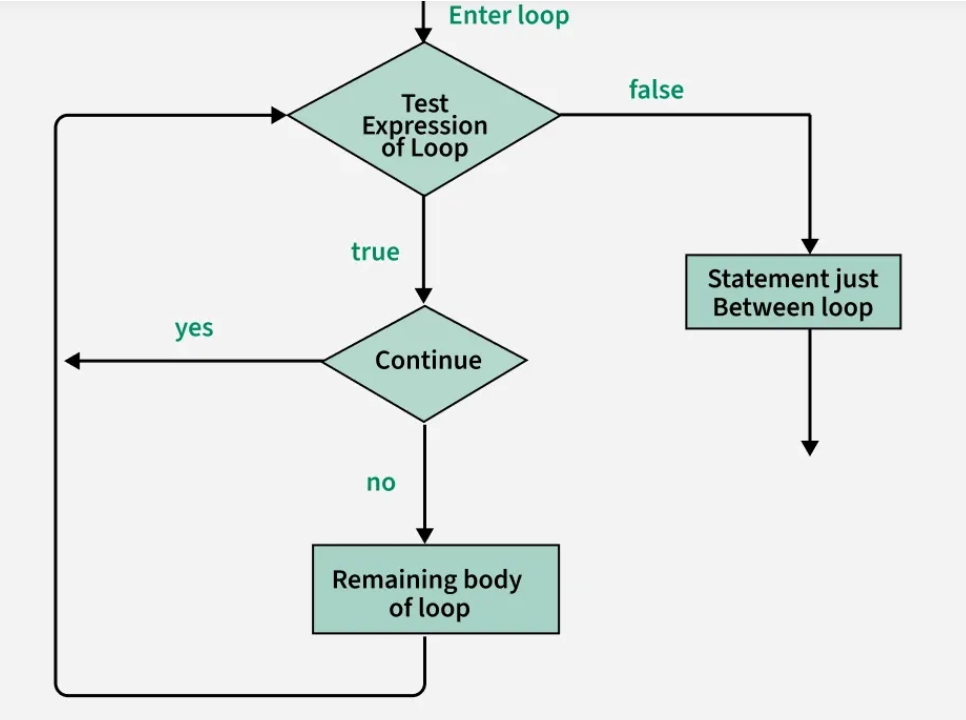
### ****6. jump Statements****

Java supports three jump statements: **break, continue** and **return**. These three statements transfer control to another part of the program.

* **Break:** In Java, a break is majorly used for:   
  + Terminate a sequence in a switch statement (discussed above).
  + To exit a loop.
  + Used as a "civilized" form of goto.
* **Continue:**Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

#### Jump Statements Execution Flow

The below diagram demonstrates the flow chart of a "jump Statements execution flow" in programming.



**Example:**The below Java Program demonstrates how the continue statement skip the current iteration when a condition is true.

// Java program to demonstrates the use of

// continue in an if statement

import java.util.\*;

​

class Geeks {

public static void main(String args[])

{

for (int i = 0; i < 10; i++) {

// If the number is even

// skip and continue

if (i % 2 == 0)

continue;

​

// If number is odd, print it

System.out.print(i + " ");

}

}

}

**Output**

1 3 5 7 9

### ****Return Statement****

The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

**Example:** The below Java program demonstrates how the return statements stop a method and skips the rest of the code.

// Java program to demonstrate the use of return

import java.util.\*;

​

public class Geeks {

public static void main(String args[])

{

boolean t = true;

System.out.println("Before the return.");

​

if (t)

return;

​

// Compiler will bypass every statement

// after return

System.out.println("This won't execute.");

}

}

**Output**

Before the return.

## Comparison of Decision-Making Statements

### if-else vs switch-case

The table below demonstrates the difference between if-else and switch-case.

| **Features** | **if-else** | **switch-case** |
| --- | --- | --- |
| **Use Case** | Suitable for condition-based checks | Best for exact value matching |
| **Readability** | More readable for a few conditions | More readable and efficient for many cases |
| **Performance** | Slower for many checks due to multiple conditions | Faster and optimized for handling many cases |
| **Flexibility** | Supports ranges and complex conditions | Only supports exact matches of values |