**Nested For Loop in Java | Pattern Example**

**Nested for loop in Java** means one for statement inside another for statement. In other words, a [for loop](https://www.scientecheasy.com/2021/05/for-loop-in-java.html/) nested inside another for loop is called nested for loops.

A nested for loops consists of an outer for loop and one or more inner for loops. Each time the outer for loop repeats, the inner for loop re-enters and starts a new execution.

That is, each time the control will enter inside the inner for loop when the outer for loop repeats.

We can put many loops inside a loop. But, it is advice that you do not go beyond three levels of nested loops, as it will make the program look clumsy.

**Syntax of Nested for loop in Java**

The general syntax for using nested for loop in Java is as follows:

// Outer for loop.

for ( initialization; test-condition; increment )

{

// Inner for loop.

for ( initialization; test-condition; increment )

{

// statement of inner loop

}

// statement of outer loop

}

Let’s understand it with the help of an example.

for(int i = 1; i <= 3; i++){

statement1; // This statement will execute 3 times by changing i values from 1 to 3.

}

for(int j = 1; j <= 4; j++){

statement2; // This statement will execute 4 times by changing j value from 1 to 4.

}

If we write first for loop inside the second for loop, it will look like this:

for(int i = 1; i <= 3; i++)

{

statement1; // will execute 3 times.

for(int j = 1; j <= 4; j++)

{

statement2; // will execute 12 (3 \* 4) times.

}

}

a) In this example, when i = 1, execution will start from the outer for loop and statement1 will execute once.

b) Now, the control of execution enters inside the inner for loop. Since we initially set the control variable j to 1, statement2 will execute once.

c) After this execution, the value of j will be 2 because of increment by 1. Then, statement2 will execute once again.

d) Like this, the inner for loop will execute 4 times with changing j values from 1 to 4. This means that statement2 will execute 4 times in the first execution of outer for loop.

e) When the execution of inner for loop completes, the control of execution goes to the outer for loop. Now, the value of i will be 2 because of increment by 1.

This time, the control of execution again enters inside the inner for loop and statement2 will execute 4 times.

f) Then, the control again goes to outer for loop and the value of i will set to 3. Again, the inner for loop will execute 4 times.

This means that the values of i and j will change as:

* When i = 1,  j = 1, 2, 3, 4
* i = 2,  j = 1, 2, 3, 4
* i = 3,  j = 1, 2, 3, 4

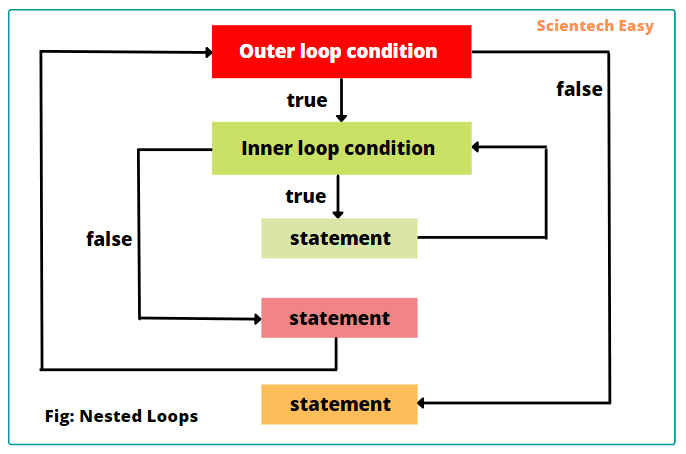
In the above sequence, the outer for loop will execute total 3 times. Hence, statement1 inside the loop body will also execute 3 times.

Since the inner for loop will execute 4 times for each i value therefore, statements2 inside the loop body will execute 12 times.

In the same way, we can also nest a while loop or a do-while loop inside a for loop and vice versa. These are called nested loops in Java.

**Flowchart Diagram for Nested Loops in Java**

The basic flowchart diagram for nested loops in Java has shown in the below figure.

[](https://www.scientecheasy.com/2021/05/nested-for-loop-in-java.html/)

Thus, we can also nest a while loop or a do-while loop inside a for loop and vice versa. These are called nested loops in Java.

The general syntax to nest while loop inside a for loop is as:

for(initialization; test-condition; increment/decrement)

{

statements;

while(conditional expression)

{

statements;

}

}

**Nested For Loops Example Programs**

Let’s take an example program in which we will display values of the inner for loop for each outer iteration, as well as outer for loop.

**Program code 1:**

public class NestedForLoopEx {

public static void main(String[] args)

{

// Outer for loop.

for(int i = 1; i <= 3; i++)

{

System.out.print(i+ "\n"); // will execute 3 times.

// Inner for loop.

for(int j = 1; j <= 4; j++){

System.out.print(j+ " "); // will execute 12 (3 \* 4) times.

}

System.out.println(" ");

}

}

}

Output:

1

1 2 3 4

2

1 2 3 4

3

1 2 3 4

As you observe in the output, during each iteration of outer for loop, the statement inside the inner for loop executes 4 times. Thus, outer for loop executes three times and inner for loop executes total four times.

2. Let’s take an example in which we will display tables for 1 and 2 using nested for loops.

**Program code 2:**

public class Tables {

public static void main(String[] args)

{

System.out.println("Displaying Tables: ");

// Outer for loop.

for(int i = 1; i <= 3; i++)

{

// Inner for loop.

for(int j = 1; j <= 10; j++) {

System.out.println(i+ " \* " +j+" = "+ (i\*j));

}

System.out.println(" ");

}

}

}

Output:

Displaying Tables:

1 \* 1 = 1

1 \* 2 = 2

1 \* 3 = 3

1 \* 4 = 4

1 \* 5 = 5

1 \* 6 = 6

1 \* 7 = 7

1 \* 8 = 8

1 \* 9 = 9

1 \* 10 = 10

2 \* 1 = 2

2 \* 2 = 4

2 \* 3 = 6

2 \* 4 = 8

2 \* 5 = 10

2 \* 6 = 12

2 \* 7 = 14

2 \* 8 = 16

2 \* 9 = 18

2 \* 10 = 20

**Java Pattern Examples using Nested For Loop**

1. Let’s take an example program in which we will display a triangle of \* using nested for loop.

**Program code 3:**

public class PatternEx {

public static void main(String[] args)

{

System.out.println("Displaying a right triangle of \*: ");

// Outer for loop.

for(int i = 1; i <= 5; i++)

{

// Inner for loop.

for(int j = 1; j <= i; j++) {

System.out.print("\*"+ " ");

}

System.out.println(" ");

}

}

}

Output:

Displaying a right triangle of \*:

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

2. Let’s take an example program to display a triangle of numbers.

**Program code 4:**

public class PatternEx {

public static void main(String[] args)

{

System.out.println("Displaying a right triangle pattern of numbers: ");

// Outer for loop.

for(int i = 1; i <= 5; i++)

{

// Inner for loop.

for(int j = 1; j <= i; j++) {

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

}

System.out.println(" ");

}

}

}

Output:

Displaying a right triangle pattern of numbers:

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

**Program code 5:**

public class PatternEx {

public static void main(String[] args)

{

int k = 1;

System.out.println("Displaying right triangle pattern of numbers: ");

// Outer for loop.

for(int i = 1; i <= 5; i++)

{

// Inner for loop.

for(int j = 1; j <= i; j++) {

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

}

System.out.println(" ");

k++;

}

}

}

Output:

Displaying a right triangle pattern of numbers:

1

2 2

3 3 3

4 4 4 4

5 5 5 5 5

3. Let’s take an example program to the display the following pattern using Java nested for loops.

**Program code 6:**

public class PatternEx {

public static void main(String[] args)

{

System.out.println("Displaying pattern of numbers: ");

// Outer for loop.

for(int i = 5; i >= 1; i--)

{

// Inner for loop.

for(int j = 1; j <= i; j++) {

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

}

System.out.println(" ");

}

}

}

Output:

Displaying pattern of numbers:

1 2 3 4 5

1 2 3 4

1 2 3

1 2

1

**Alphabet Pattern using Nested For Loops**

4. Let’s take example programs to print the following alphabet pattern using Java nested for loops.

**Program code 7:**

public class PatternEx {

public static void main(String[] args)

{

System.out.println("Displaying alphabet pattern: ");

// Outer for loop.

for(int i = 65; i <= 69; i++)

{

// Inner for loop.

for(int j = 65; j <= i; j++) {

char ch = (char)j;

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

}

System.out.println(" ");

}

}

}

Output:

Displaying alphabet pattern:

A

A B

A B C

A B C D

A B C D E

**Program code 8:**

public class PatternEx {

public static void main(String[] args)

{

System.out.println("Displaying alphabet pattern: ");

// Outer for loop.

for(int i = 69; i >= 65; i--)

{

// Inner for loop.

for(int j = 65; j <= i; j++) {

char ch = (char)j;

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

}

System.out.println(" ");

}

}

}

Output:

Displaying alphabet pattern:

A B C D E

A B C D

A B C

A B

A

**More Example Patterns for Best Practices**

**Program code 9:**

public class PatternEx {

public static void main(String[] args)

{

int k = 65;

System.out.println("Displaying alphabet pattern: ");

// Outer for loop.

for(int i = 65; i <= 69; i += 2)

{

// Inner for loop.

for(int j = 69; j >= 65; j--)

{

if(j > i)

System.out.print(" ");

else

System.out.format("%c ", k++);

}

System.out.println(" ");

}

}

}

Output:

Displaying alphabet pattern:

A

B C D

E F G H I

**Program code 10:**

public class PatternEx {

public static void main(String[] args)

{

int i, j, k = 1;

for(i = 1; i <= 5; i += 2)

{

for(j = 5; j >= 1; j--)

{

if(j > i)

System.out.print(" ");

else

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

}

System.out.println();

}

}

}

Output:

1

2 3 4

5 6 7 8 9

**Advantages of Nested For Loops**

There are the following advantages of using nested for loops. They are as:

* Nested for loops are suitable for handling multidimensional arrays and matrices, simplifying data processing.
* They help us to solve complex tasks in which we need multiple levels of iteration.
* We commonly used them for generating various patterns and shapes in the output.

**Limitations of Nested For Loops**

Although nested for loops are powerful, but they also have some limitations:

* Excessive nesting can impact the performance of code as the number of iterations increases significantly.
* Excessive nesting can make code harder to read, understand, and maintain.

**Tips for Using Nested For Loops Efficiently**

To use of nested for loops efficiently, you should keep the following points in mind.

* Keep it simple.
* Avoid excessive levels of nesting. Stick to two levels whenever possible.
* Optimize loop conditions to minimize unnecessary iterations.
* Keep proper indentation to enhance code readability.
* Add proper comments to explain the aim of nested loops, especially when working with complex tasks.
* Evaluate your code thoroughly and debug any issues that arise during the execution of the program.

**For Each Loop in Java | Enhanced For Loop**

The **for each loop in Java** (also referred to as enhanced for loop) is an extended feature of Java language that was introduced with the J2SE 5.0 release.

This feature is specially designed to retrieve elements of the array efficiently rather than using array indexes.

Java for-each loop can also be used to retrieve elements of a collection. A collection represents a group of elements as integer values, strings, or objects.

Enhanced for loop repeatedly executes a group of statements for each element of the collection. It can execute as many times as a number of elements in the collection.

**Syntax of Java For Each Loop**

The general syntax for using for each loop in Java is as follows:

for(data\_type identifier : expression)

{

// Code block to be executed for each element.

}

In the above syntax,

* data\_type represents the data type of elements in the collection or object used;
* Identifier specifies the name of an iteration variable that receives elements from a collection, one at a time, from beginning to end;
* expression is an object of java.lang.Iterable interface or an array.

**Note:** Unlike some other programming languages, such as C#, that implement a for-each loop by using **foreach** keyword.

But Java language added the for each capability by enhancing the for statement. The advantage of enhancing is that no new keyword is needed, and no preexisting code is broken.

Therefore, the for each loop is also known as **enhanced for loop in Java**. Professional programmers generally use enhanced for loop in the industry.

**For Each Loop Examples**

Let’s understand the above syntax with the help of an example. Consider the following examples below:

**Example 1:**

// Create an array of three elements.

int numarr[] = {10, 20, 30};

// Applying for loop to iterate over elements.

for(int i = 0; i <= 3; i++)

{

if(numarr[i] > 5 && numarr[i] < 40)

System.out.println("Selected value: " +numarr[i]);

}

// This code is equivalent to the following code:

int numarr[] = {10, 20, 30};

// Applying for each loop to iterate over elements.

for(int i : numarr)

{

if(i > 5 && i < 40)

System.out.println("Selected value: " +i);

}

In this example, we have created an array of three elements and store in a variable named numarr. Then, we have used for each loop to iterate over each element in the “numarr” array.

According to the syntax of for each loop, int represents the data type, i represents the name of temporary variable used for storing each element during iteration, and numarr is an iterable object being traversed.

The enhanced for loop iterates over each element in the array and prints the result. Thus, we can use Java for each loop to track elements of an array efficiently.

In the same style, we can also use for each loop to track elements of the collection, as follows:

**Example 2:**

// Creating an object of ArrayList of type String.

ArrayList<String> cities = new ArrayList<String>();

// Adding elements in the array list.

cities.add("Delhi");

cities.add("Mumbai");

cities.add("Dhanbad");

cities.add("Kolkata");

// Iterating array list using enhanced for loop.

for(String city : cities)

{

System.out.println(city);

}

In this example, we have created an ArrayList named “cities” of type string. We have added four string elements to the list. Then, we have used the for-each loop to iterate through each element in the “cities” list. If we compare the syntax of for each loop with the above code, we will get the following information:

* string represents the data type of elements,
* city represents the name of iteration variable used to store each element during the iteration,
* cities represents an iterable object (ArrayList) being traversed.

The enhanced for loop iterates through each element in the array list and prints its value using System.out.println(city). The output will display the name of cities Delhi, Mumbai, Dhanbad, and Kolkata on separate lines.

**Java For Each Loop Example Programs**

1. Let’s take an example program where we retrieve elements of an array one by one using for-each loop and display it on the console.

**Program code 1:**

package javaProgram;

public class Test {

public static void main(String[] args)

{

// Create an array of 5 elements.

String cities[] = {"Dhanbad", "Mumbai", "Delhi", "New York", "Godda" };

// Using for each loop to retrieve elements (i.e. cities) from an array.

System.out.println("Name of cities:");

for(String city : cities)

{

System.out.println(city); // city represents each element of an array.

}

}

}

Output:

Name of cities:

Dhanbad

Mumbai

Delhi

New York

Godda

2. Let’s take another example program where we will calculate the sum of the first 10 numbers using enhanced for loop and display it on the console.

**Program code 2:**

package javaProgram;

public class Test {

public static void main(String[] args)

{

// Declare an array of 10 elements.

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

// Iterating elements using for each loop to display and sum the values.

for(int x : nums) {

sum += x;

}

System.out.println("Sum: " + sum);

}

}

Output:

Sum: 55

3. Suppose in the above program, we want to calculate only the sum of five numbers out of 10 numbers. Here, we will use [*break statement*](https://www.scientecheasy.com/2021/05/break-statement-in-java.html/). Let’s write the code for it.

**Program code 3:**

package javaProgram;

public class Test {

public static void main(String[] args)

{

// Declare an array of 10 elements.

int nums[ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

// Iterating elements using for-each loop to display and sum the values.

for(int x : nums) {

sum += x;

if(x == 5) break; // stop the loop when x is equal to 5.

}

System.out.println("Sum of first 5 elements: " + sum);

}

}

Output:

Sum of first 5 elements: 15

As you can observe in the example program, the for-each loop stops after the fifth element has been obtained.

**Iterating elements of Multidimensional Array**

We can also use the Java enhanced for loop on multidimensional arrays. In Java, multidimensional arrays consist of arrays of arrays. Let’s understand the code with the help of an example program. Look at the source code.

**Program code 4:**

package javaProgram;

public class Test {

public static void main(String[] args)

{

int sum = 1;

// Create an object of 2D array.

int nums[ ][ ] = new int[3][5];

for(int i = 1; i < 3; i++)

{

for(int j = 1; j < 5; j++)

{

nums[i][j] = i \* j;

// Using for each loop to display and sum the values.

for(int x[ ] : nums)

{

for(int y : x)

{

sum += y;

}

}

}

}

System.out.println("Sum: " + sum);

}

}

Output:

Sum: 101

5. Let’s take an example in which we will search for an element from an array. If the element is found, it will stop. Look at the source code to understand better.

**Program code 5:**

package javaProgram;

public class SearchTest {

public static void main(String[] args)

{

// Creating an array object of 8 elements.

int nums[] = { 1, 8, 3, 7, 5, 6, 10, 4 };

int val = 10;

boolean found = false;

// Searching for an element value 10 from an array using Java for each loop.

for(int x : nums) {

if(x == val) {

found = true;

break;

}

}

if(found)

System.out.println("Value found!");

}

}

Output:

Value found!

**Advantages of For-Each Loop Over Traditional For Loop**

The for-each loop provides several advantages over the traditional for loop in Java:

* The syntax of the for-each loop is very simple and concise, resulting in more concise code compared to traditional loops.
* The for-each loop enhances code readability as compared to the traditional for loop in Java.
* As there is no explicit use of indices, the for-each loop helps avoid common errors, such as “IndexOutOfBoundException”.
* With the for each loop, we can easily iterate over elements in arrays and collections without the need to manage indices explicitly.
* The enhanced for loop makes the code more concise and easier to understand.

**Limitations of For Each Loop (or Enhanced For Loop)**

Although the for-each loop provides several significant advantages, but it does have some limitations and potential pitfalls:

* In Java, the for each loop is designed for read-only access to elements. If we try to modify it during the iteration, we may get to runtime exceptions.
* The for each loop is not suitable when we need to access elements by index.
* It may not be suitable in certain situations, such as looping backward or skipping certain elements.

**When to Use For Each Loop in Java**

Java for each loop finds extensive use in scenarios where we need to iterate over all elements of an array or collection sequentially. Some common use cases include:

* Calculating the sum, average, or other aggregate values of an array of numbers.
* Searching for specific elements in a collection.
* Printing or displaying elements to the user.
* Copying elements from one collection to another.
* Filtering elements based on certain criteria.
* Use for each loop with custom objects if the collection contains instances of the custom class.
* Use the enhanced for loop if you need the guarantee order of elements in the collection, like list.

**Switch Statement in Java | Syntax, Example**

A **switch statement in Java** is a conditional control statement (or multi-way decision statement) that executes one statement from multiple conditions.

It uses the result of an expression to evaluate which statements to execute. It is an alternative to [if-else-if ladder statement](https://www.scientecheasy.com/2021/04/if-else-in-java.html/).

In other words, a switch statement executes statements based on the value of a variable or an expression, against a list of case values. If a match is found, a block of statements corresponding to that case is executed.

**Syntax to Switch Statement in Java**

The general syntax of using the switch statement in Java is as follows:

switch(integer expression)

{

case value-1:

// statement sequence

break;

case value-2:

// statement sequence

break;

. . . . .

. . . . .

case value-n:

// statement sequence

break;

default:

// default statement sequence

}

statement-x;

a) In the switch syntax, an expression must be of data types byte, short, int, char, long, or String. From Java 7 onwards, we can use string in the switch statement.

We can also use four wrapper classes such as Byte, Short, Integer, and Long in the switch statement.

b) value-1, value-2, value-3 . . . . are constants or literals. These constants are called case labels (or case clauses). Each of the values specified in the case labels must be of a type compatible with the expression. Duplicate case values are not permitted.

**Note:** Case labels cannot be variables, float values, or boolean expressions. The case labels value-1, value-2, . . . should be distinct.

c) A switch can have multiple case clauses depending on requirements and must be unique within a switch statement. The case labels must end with a colon (:).

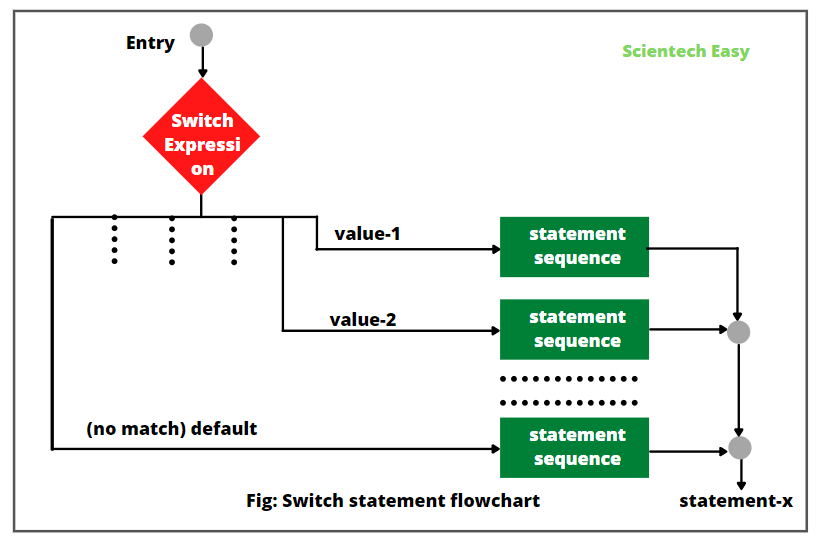
d) The statement sequence is a list of statements and may contain zero or more statements.

e) After each case, a break statement is necessary inside the switch block to come out switch block.

f) The default statement sequence is optional and is executed when none of the previous cases are matched.

**How does switch statement work in Java?**

The flowchart of Java switch statement for the selection process is shown in the below figure.

[](https://www.scientecheasy.com/2021/05/switch-statement-in-java.html/)

Java switch statement works like this:

a) When the switch statement is executed, the value of the expression is successively compared with each case value like value-1, value2-, . . .

b) If a match is found, the statement sequence following that case label is executed.

c) If the value of expression does not equal to value-1, value-2, . . . . then none of the statement sequences will be executed.

d) In this case, the default clause is executed, and then default statements are executed. However, the default statement is optional. If no case is matched and no default is present, then no further execution takes place.

e) The break statement is used inside the switch block to terminate a statement sequence. It is optional. It immediately ends the switch  
statement.

f) When a break statement is executed, the control of execution is moved to the next statement-x after the switch statement (skipping all the rest of case statements).

**Switch Statement Example Programs**

1. Let’s take a very simple example program based on the switch statement where we will execute a particular task depending on city value. We will take a variable city, that is initialized to d.

Depending on the city value, Mumbai is displayed when the value is m; Dhanbad is displayed when the value is d; and so on.

If city value is neither of the specified values: m, d, c, or r, then none of the statements will be executed. In this case, the default statement will execute and it will display no city. Look at the source code to understand better.

**Program code 1:**

package javaProgram;

public class SwitchTest {

public static void main(String[] args)

{

// Declaring a variable for switch expression.

char city = 'd'; // Here, city is set to 'd'.

// Switch expression with char value.

switch(city) {

// Case statements.

case 'm':

System.out.println("Mumbai");

break;

case 'd':

System.out.println("Dhanbad");

break;

case 'c':

System.out.println("Chennai");

break;

case 'r':

System.out.println("Ranchi");

break;

// Default case statement.

default:

System.out.println("No city");

}

}}

Output:

Dhanbad

In this example, since the city value is set to d, it will display “Dhanbad” as output and then executes break statement which terminates the switch block.

**Program code 2:**

package javaProgram;

public class SwitchTest2 {

public static void main(String[] args)

{

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

switch(i) {

case 0:

System.out.println("Zero");

break;

case 1:

System.out.println("One");

break;

case 2:

System.out.println("Two");

break;

case 3:

System.out.println("Three");

break;

default:

System.out.println("i is greater than 3");

}

}

}

Output:

Zero

One

Two

Three

i is greater than 3

2. Let’s understand an example based on if and switch statements where we will take total marks of five subjects from students as input. Then, we will calculate the percentage of total marks, and display Grade based on the percentage. Look at the following program source code to understand better.

**Program code 3:**

package javaProgram;

import java.util.Scanner;

public class GradeApplication {

public static void main(String[] args)

{

int per = 0;

// Create an object of Scanner class to take an input.

Scanner sc = new Scanner(System.in);

System.out.println("Enter your total marks of five subjects:");

int total = sc.nextInt();

if(total <= 500) {

per = total/5;

System.out.println("Your percentage: " +per);

} else {

System.out.println("Invalid marks");

}

int i = per/10;

switch(i) {

case 9:

System.out.println("Grade A+");

break;

case 8:

System.out.println("Grade A");

break;

case 7:

System.out.println("Grade B+");

break;

case 6:

System.out.println("Grade B");

break;

case 5:

System.out.println("Grade C");

break;

case 4:

System.out.println("Grade D");

break;

case 3:

System.out.println("Fail");

break;

}

}}

Output:

Enter your total marks of five subjects:

450

Your percentage: 90

Grade A+

4. Let’s take an example program where we will not use break statements. It means that it will execute all statements after the first match.

**Program code 4:**

package javaProgram;

public class SwitchTest3 {

public static void main(String[] args)

{

int num = 20;

switch(num) {

// Case statement without break statements.

case 10: System.out.println("Ten");

case 20: System.out.println("Twenty");

case 30: System.out.println("Thirty");

case 40: System.out.println("Forty");

default: System.out.println("Default statement");

}

}

}

Output:

Twenty

Thirty

Forty

Default statement

5. Let’s take an example program where we will use strings in the switch expression. The case labels must be string literal. Look at the source code below.

**Program code 5: Switch statement with string.**

package javaProgram;

public class SwitchStringTest {

public static void main(String[] args)

{

// Declaring a variable of string type.

String gameLevel = "Intermediate";

int level = 0;

// Using string in switch statement.

switch(gameLevel) {

// Using string literal in switch case.

case "Beginner": level = 1;

break;

case "Intermediate": level = 2;

break;

case "Expert": level = 3;

break;

default: level = 0;

break;

}

System.out.println("Your game level is: " +level);

}

}

Output:

Your game level is: 2

**Switch Statement Examples for Best Practices**

6. Let’s take an example in which we will display the name of day in the week using Java switch statement.

**Program code 6:**

package javaProgram;

public class DayChecker {

public static void main(String[] args)

{

int dayOfWeek = 3;

String dayName;

switch (dayOfWeek) {

case 1:

dayName = "Sunday";

break;

case 2:

dayName = "Monday";

break;

case 3:

dayName = "Tuesday";

break;

case 4:

dayName = "Wednesday";

break;

case 5:

dayName = "Thursday";

break;

case 6:

dayName = "Friday";

break;

case 7:

dayName = "Saturday";

break;

default:

dayName = "Invalid day";

}

System.out.println("Today is " + dayName);

}

}

Output:

Today is Tuesday

7. Let’s take an example where we will display the name of season occurring during the year using switch statement. In this example, we will understand how to handle multiple cases.

**Program code 7:**

package javaProgram;

public class SeasonChecker {

public static void main(String[] args)

{

int month = 7;

String season;

switch(month) {

case 12:

case 1:

case 2:

season = "Winter";

break;

case 3:

case 4:

case 5:

season = "Summer";

break;

case 6:

case 7:

case 8:

case 9:

season = "Mansoon";

break;

case 10:

case 11:

season = "Autumn";

break;

default:

season = "Unknown season";

}

System.out.println("The current season is " + season);

}

}

Output:

The current season is Mansoon

8. Let’s take an example in which we will take a grade as an input from the user and display a feedback message based on the grade.

**Program code 8:**

package javaProgram;

import java.util.Scanner;

public class GradeChecker {

public static void main(String[] args)

{

// Creating a scanner object.

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your grade (A, B, C, D, or F): ");

char grade = scanner.next().charAt(0);

switch (grade) {

case 'A':

System.out.println("Excellent! You've got an A.");

break;

case 'B':

System.out.println("Good job! You've got a B.");

break;

case 'C':

System.out.println("Well done. You've got a C.");

break;

case 'D':

System.out.println("You passed, but there's room for improvement. You've got a D.");

break;

case 'F':

System.out.println("Unfortunately, you've failed. You've got an F.");

break;

default:

System.out.println("Invalid grade entered.")

}

scanner.close(); // closing scanner.

}

}

Output:

Enter your grade (A, B, C, D, or F): B

Good job! You've got a B.

**Nested Switch Statements in Java**

Like if statements, we can also nest switch statements. That is, a switch can also be a part of the statement sequence of an outer switch. It is called nested switch.

When a switch statement is placed within a case statement or default statement of another switch statement, it is called **nested switch in Java**.

The purpose of using nested switch statement in a program is to tell Java to make another decision after first decision.

Since a switch statement defines its own block, no conflicts occur between the case labels in the inner switch and those in the outer switch.

9. Consider the following example program to understand nested switch statement.

**Program code 9:**

package javaProgram;

public class NestedSwitchEx {

public static void main(String[] args)

{

// E - ECE, C - CS, I - Information Technology.

char branch = 'E';

int collegeYear = 3;

switch(branch) {

case 'E':

switch(collegeYear) // Inner switch

{

case 3:

System.out.println("Microcontroller, Power Electronics, Analog circuit, Digital circuit");

break;

}

break;

case 'C':

switch(collegeYear) // Inner switch.

{

case 3:

System.out.println("Java, Python, Data structure");

break;

}

break;

case 'I':

switch(collegeYear) {

case 3:

System.out.println("EDC, Java, Data structure, Microprocessor");

break;

}

break;

default:

System.out.println("Invalid selection");

break;

}

}

}

Output:

Microcontroller, Power Electronics, Analog circuit, Digital circuit

As you can see in this program, we can nest a switch statement inside another switch statement in Java.

10. Let’s take an example in which we will use the wrapper class in switch statement.

**Program code 10:**

package javaProgram;

public class WrapperSwitchCaseEx {

public static void main(String[] args)

{

Integer age = 18;

switch(age) {

case 16:

System.out.println("You are not eligible for vote");

break;

case 18:

System.out.println("You are eligible for vote");

break;

default:

System.out.println("Please, enter valid age");

}

}

}

Output:

You are eligible for vote

**When to Use Switch Statement in Java**

A switch statement can be used to select one block out of multiple blocks of statement. That is, we can use switch statement to execute one statement from multiple conditions. It is a better choice than Java if-else-if ladder.

Some common scenarios to use switch statement in Java are as:

* We can use the switch statement for menu selection. In menu-driven applications, we can allow users to make choices by selecting corresponding characters or numbers.
* We can use switch statement for mapping constant values to corresponding actions or behaviors. It will offer a well and concise way to handle such mapping in the code.
* A switch statement is an excellent choice to implement state machines. Each case in the switch statement represents a different state, and the switch logic controls transitions between states.
* In programs that deal with dates and time, we can use the switch statements to convert numeric representations of days (e.g., 1 for Sunday, 2 for Monday) to their corresponding names.
* You can use the switch statements in calculator programs to handle different mathematical operations based on user inputs.

When you need to execute one statement out of two alternatives, use if-else statement.

**Advantages of Using Switch Statements**

Switch statements provide several advantages, making them a precious decision-making statement in a Java programming. They are as:

* Switch statement is easy to read and understand, especially when dealing with multiple options.
* With the switch statement, we can easily understand the logic behind different cases.
* The main advantage of using switch statement in Java program is that it provides a well and clear code structure by representing different cases in a straightforward manner.
* Because of their direct mapping of values, switch statements can be faster than equivalent if-else-if ladder statements.
* Switch statement reduce code duplication by mapping similar logic into separate cases.
* It makes the code debugging easy.

# Break Statement in Java | Example Program

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

For example, suppose there is a list of 100 names and we need to search a particular name from the list. When the search completes and the desired name found, the program loop written for searching must terminate.

For this purpose, Java allows a jump from one statement to the end or beginning of a loop as well as jump out of a loop. So, let’s understand all three types of jump statements one by one.

## Break Statement in Java

A break statement in Java is used to break a loop or switch statement. When a break statement encounters inside a loop statement, the loop immediately ends at a specified condition.

As a result, the program control continues the execution of the next statement immediately following the loop body.

Similarly, when a break statement executes inside the switch block, it terminates ‘case’ inside the switch block and stops the execution of more cases inside the switch.

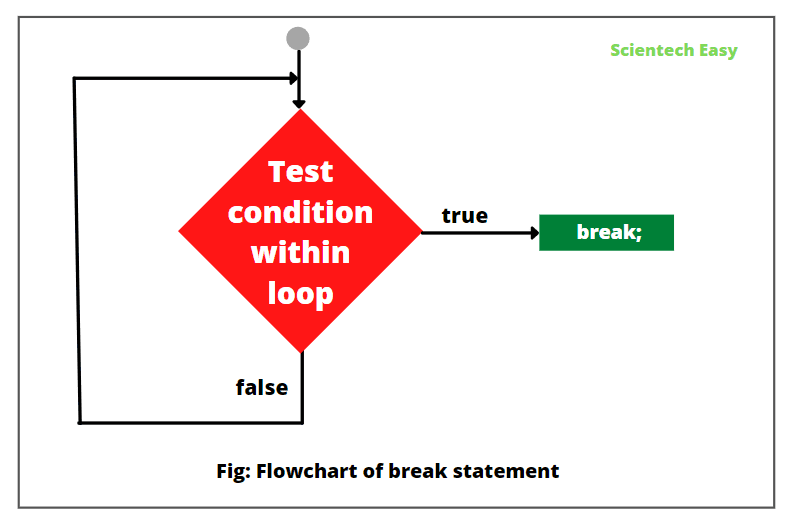
When the loops are nested, the break statement breaks only the inner loop. We can use Java break statement in all types of loops such as for loop, while loop, and do-while loop.

The general syntax for break statement is as follows:

// Jump statement

break;

The flowchart of break statement in java has shown in the below figure.

[](https://www.scientecheasy.com/2021/05/break-statement-in-java.html/)

You can also use the break statement with a label reference to break out of any code block. The general syntax to use break statement with label is as:

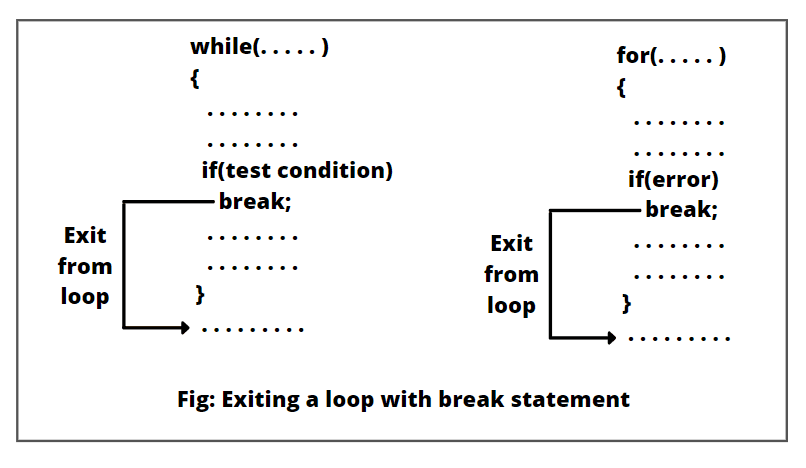
break labelName;

Without a label, you can only use break inside a loop or a switch.

## Use of Break Statement

We can use a break statement in Java in three ways that are as:

1. We can use a break inside the loop to come out of it.

[](https://www.scientecheasy.com/2021/05/break-statement-in-java.html/)

2. We can use it inside the switch block to come out of switch block.

3. Break can also be used inside the nested blocks to go to the end of block.

## Break Statement Example Programs

1. Let’s take a simple example program where we will use a break statement inside the for loop for breaking the loop. Look at the program source code.

**Program code 1:**

package javaProgram;

public class BreakExample1 {

public static void main(String[] args)

{

// Using for loop.

for(int i = 1; i <= 10; i++)

{

if(i == 5) break; // Here, the break statement is breaking a loop.

System.out.println("I: " +i);

}

}

}

Output:

I: 1

I: 2

I: 3

I: 4

2. Let’s take another example program where we will use break statement inside the inner for loop.

**Program code 2:**

package javaProgram;

public class BreakExample2 {

public static void main(String[] args)

{

// Outer for loop.

for(int i = 1; i <= 3; i++)

{

// Inner for loop.

for(int j = 0; j <= 3; j++) {

if(i == 2 && j == 2)

break; // Using break statement inside for loop.

System.out.println(i+" "+j);

}

}

}

}

Output:

1 0

1 1

1 2

1 3

2 0

2 1

3 0

3 1

3 2

3 3

3. Let’s take an example program where we will use the break statement with labeled for loop. We can use break with the label. This feature was added since JDK 1.5 version.

**Program code 3:**

package javaProgram;

public class BreakExample3 {

public static void main(String[] args)

{

// Outer for loop.

for(int i = 1; i <= 3; i++)

{

bb:

// Inner for loop.

for(int j = 1; j <= 3; j++){

if(i == 2 && j == 2)

break bb; // Using break statement with label.

System.out.println(i+" "+j);

}

}

}}

Output:

1 2

1 3

2 1

3 1

3 2

3 3

4. Let’s take a simple example program where we will use the break statement inside while loop.

**Program code 4:**

package javaProgram;

public class BreakWhileEx {

public static void main(String[] args)

{

// while loop.

int i = 1;

while(i <= 10) {

if(i == 5) {

i++;

break; // Breaking the loop.

}

System.out.println(i);

i++;

}

}

}

Output:

1

2

3

4

5. Let’s take an example in which we will use the break statement inside the do-while loop.

**Program code 5:**

package javaProgram;

public class BreakDoWhileEx {

public static void main(String[] args)

{

// Initialization.

int i = 1;

// do-while loop.

do

{

if(i == 5) {

i++;

break;

}

System.out.println(i);

i++;

} while(i <= 10);

}

}

Output:

1

2

3

4

6. Let’s take an example in which we will search an element within an array using break statement.

**Program code 6:**

package javaProgram;

import java.util.Scanner;

public class SearchingEx {

public static void main(String[] args)

{

// Create a Scanner object to read user input.

Scanner scanner = new Scanner(System.in);

// Initializing a variable to keep track of the loop.

boolean found = false;

// Creating an array of five numbers.

int[] numbers = { 10, 20, 30, 40, 50 };

System.out.print("Enter a number to search for: ");

int target = scanner.nextInt();

// Using a for loop to search for the target number

for (int number : numbers)

{

if (number == target)

{

// If the target number is found, set found to true and break out of the loop

found = true;

break;

}

}

// Check if the number was found or not

if (found) {

System.out.println("Number " + target + " found in the array.");

} else {

System.out.println("Number " + target + " not found in the array.");

}

// Close the Scanner object.

scanner.close();

}

}

Output:

Enter a number to search for: 30

Number 30 found in the array.

In this example, we have taken a number as an input from the user and searches for a target number in an array. In the program, we have used the for loop to iterate through the array elements. If the target number found in the array, the loop is terminated using the break statement, and the program displays a message showing that the number was found. If the target number is not found, the loop completes naturally, and a different message is displayed.

# Continue Statement in Java | Use, Example

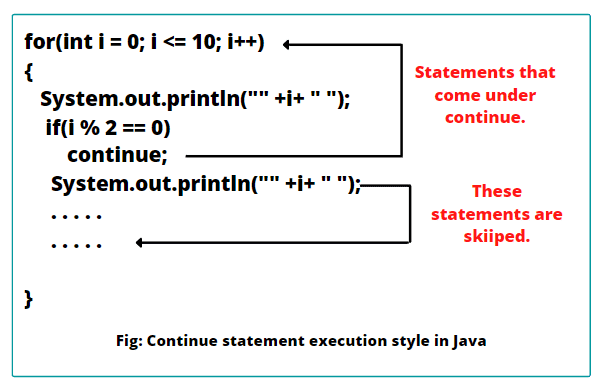
**Continue statement in Java** is another similar statement, like [break statement](https://www.scientecheasy.com/2021/05/break-statement-in-java.html/) that is used inside a loop to repeat the next iteration of the loop.

In other words, the continue statement stops the current iteration of loop and begins a new iteration of loop.

When the continue statement encounters, subsequent statements in the loop skips (i.e. not executed) at a specified condition, and the control of execution continues with the next repetition of the loop.

Continue statement in Java do not break out of the loop entirely. It just jumps back to the beginning of the loop by skipping the rest of code in the loop body for the next iteration.

Look at the below figure to understand better.

[](https://www.scientecheasy.com/2021/05/continue-statement-in-java.html/)

## Syntax of Continue Statement in Java

The general syntax of using the continue statement in Java is as follows:

Syntax:

continue;

Here, continue is a keyword. We can also use continue statement with label to interrupt the loop. The syntax is as:

continue labelname;

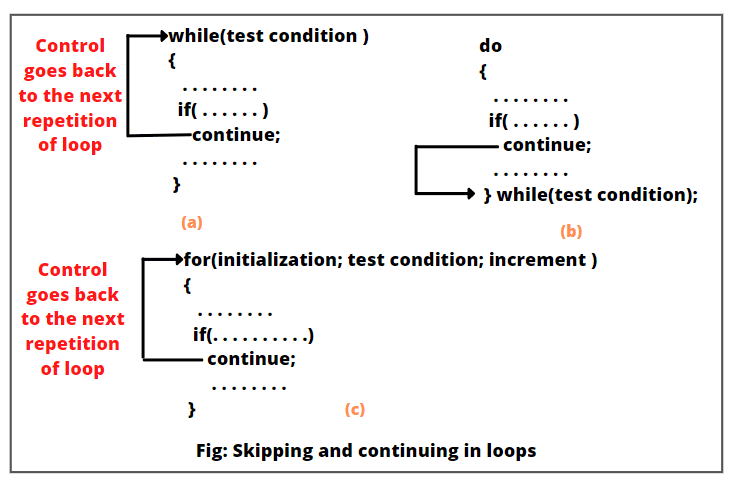
Just like the break statement, we write continue statement inside the statement block that the loop executes, preceded by a conditional test. The “continue” statement is only valid within the body of loop. If we use outside the loop, it will result in a compilation error.

## When to Use Continue Statement in Java?

Java continue statement is another jump statement used in loop control structure that is used when we need to jump to the next iteration of the loop immediately.

We can use continue statement in all the types of loops like for loop, while loop, and do-while loop. In the case of inner loop, it continues with inner loop only.

The use of continue statement in loops has shown in the below figure.

[](https://www.scientecheasy.com/2021/05/continue-statement-in-java.html/)From figure (a) and (b), in the case of while and do-while loops, continue statement causes the control of execution to go directly to the test condition and then to continue to the repetition process. But in the case of for loop, the increment section of the loop executes first and then the test condition evaluates.

Let’s understand the use of continue statement in Java with the help of an example program. Suppose we want to display numbers in descending order from 1 to 10. For this, let’s write a program using for loop. Look at the source code.

**Program code 1:**

package javaProgram;

public class ContinueUse {

public static void main(String[] args)

{

for(int i = 10; i >= 1; i--)

{

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

}

}

}

Output:

10 9 8 7 6 5 4 3 2 1

In this example, the value of “i” begins with 10 and decrements by 1 until the value of “i” is greater or equal to 1. Now, we will add continue statement in this example program like this:

package javaProgram;

public class ContinueUse {

public static void main(String[] args)

{

for(int i = 10; i >= 1; i--)

{

if(i > 5)

continue; // It will skip the rest statement and go back in the for loop.

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

}

}

}

In this example, when i > 5, we are redirecting the flow of execution back to the next iteration of loop. During the change of the value of i from 10 to 6, the continue statement will execute and subsequent statements will not execute. Therefore, the value of i from 10 to 6 will not display and the output will be like this:

Output:

5 4 3 2 1

Let us consider another simple example where we will display all even numbers between 1 and 10 using a for loop. With the “continue” statement, we can easily achieve it by skipping odd numbers. Look at the source code.

**Program code 2:**

package javaProgram;

public class ContinueEx {

public static void main(String[] args)

{

for (int i = 1; i <= 10; i++)

{

if (i % 2 != 0) {

continue;

}

System.out.println(i);

}

}

}

Output:

2 4 6 8 10

In this example, we have used a for loop to iterate over 1 to 10 numbers (inclusive). We have initialized the loop variable “i” to 1, and the loop continues as long as “i” is less than or equal to 10. After each iteration, the loop variable “i” is incremented by 1.

Within the loop body, there is an if statement that checks whether the current value of “i” is an odd number or not. The conditional expression i % 2 != 0 checks if the remainder of i divided by 2 is not equal to 0, which tells that “i” is an odd number.

If the conditional expression evaluates to true (meaning “i” is an odd number), the continue statement will execute. The continue statement causes the loop to immediately jump to the next iteration, skipping any statement that comes after it within the body of loop.

On the other hand, if the conditional expression evaluates to false (meaning “i” is an even number), the continue statement is not executed, and the program proceeds with the System.out.println(i) statement.

The System.out.println(i) statement prints the current value of “i” to the console. However, since the continue statement is skipping odd numbers, only even numbers between 1 and 10 (inclusive) will display. So, the output of the code will be: 2 4 6 8 10.

### Realtime Use of “Continue” Statement

(1) **Skipping Odd Numbers:**

The above example exhibits how can we use the “continue” statement to skip elements that don’t meet specific criteria, such as odd numbers in this case.

(2) **Skipping Unwanted Elements in Collections:**

While working with collections, we may often need to skip certain elements based on specific conditions. We can use the continue statement to achieve it efficiently.

## Continue Statement Example Program

Let’s take some important example programs based on the continue statement in Java.

**Program code 2: Use of continue statement within**[***for loop***](https://www.scientecheasy.com/2021/05/for-loop-in-java.html/)

package javaProgram;

public class ContinueEx {

public static void main(String[] args)

{

for(int i = 10; i >= 1; i--)

{

if(i == 5)

continue;

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

}

}

}

Output:

10 9 8 7 6 4 3 2 1

As you can observe in the output, 5 is not displayed on the console because the loop is continued when it is equal to 5.

**Program code 3: Use of continue statement within**[***while loop***](https://www.scientecheasy.com/2021/05/while-loop-in-java.html/)

package javaProgram;

public class ContinueUse {

public static void main(String[] args)

{

// while loop.

int x = 1; // Initialization.

while(x <= 10) {

if(x == 5) {

x++;

continue;

}

System.out.println(x);

x++;

}

}

}

Output:

1 2 3 4 6 7 8 9 10

As you can see in the output, the number 5 is skipped.

**Program code 4: Continue statement in**[***do-while loop***](https://www.scientecheasy.com/2021/05/do-while-loop-in-java.html/)

package javaProgram;

public class ContinueUse {

public static void main(String[] args)

{

// do-while loop.

int x = 1; // Declaration and Initialization of variable.

do {

if(x == 5) {

x++;

continue;

}

System.out.println(x);

x++;

} while(x <= 10);

}

}

Output:

1 2 3 4 6 7 8 9 10

**Program code 5: Continue statement inside inner loop**

package javaProgram;

public class ContinueUse {

public static void main(String[] args)

{

// Outer loop.

for(int i = 1; i<= 3; i++)

{

// Inner loop.

for(int j = 1; j <= 3; j++)

{

if(i == 2 && j == 3)

continue; // continue statement inside inner loop.

System.out.println(i+ " " +j);

}

}

}

}

Output:

1 1

1 2

1 3

2 1

2 2

3 1

3 2

3 3

### More Examples on Continue Statement

**Program code 6:**

public class ContinueEx {

public static void main(String[] args)

{

int i;

for(i = 0; i <= 6; i++)

{

if(i < 2) continue;

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

if(i < 4) continue;

System.out.println(10 \* i+ " ");

}

}

}

Output:

2

3

4

40

5

50

6

60

In this example program, we have created a loop that iterates the value of variable i from 0 to 6. If i is less than 2, we do nothing. Once i is 3 or higher, we print its value.

Once i is greater than 5, we print 10 \* i. So, the above program will print the values as shown in the output.

**Program code 7: Continue statement with label**

public class ContinueEx {

public static void main(String[] args)

{

int i, j;

outerloop:

for(i = 0; i < 3; i++)

{

for(j = 0; j < 3; j++)

{

if((i == 2) && (j == 2))

{

continue outerloop; // skips when both expressions are true.

} else {

System.out.println(i+ ", " +j+ " ");

}

}

}

}

}

Output:

0, 0

0, 1

0, 2

1, 0

1, 1

1, 2

2, 0

2, 1

**Labelled Loop in Java with Example**

In Java, we can give a label to a [*loop*](https://www.scientecheasy.com/2021/04/loops-in-java.html/). When we place a label before any loop, it is called **labelled loop in Java**.

A label is a valid variable name (or identifier) in Java that represents the name of the loop to where the control of execution should jump.

The syntax for creating a labeled loop is straightforward. To label a loop, place the label (identifier) before the loop with a colon at the end. The general syntax to give a label to loops is as:

labelname:

for(initialization; test-condition; incr/decr) {

// code to be executed.

}

or,

labelname: for(initialization; test-condition; incr/decr) {

// code to be executed.

}

**Example 1:**

Loop1: for(initialization; test-condition; incr/decr) {

// code to be executed.

}

**Example 2:**

int i = 1;

Loop2: while(i <= 3) {

// code to be executed.

}

We can also use the continue and break statements with a label like this:

Syntax:

continue labelName: // It is called labelled continue statement.

Similarly,

break labelName: // It is called labelled break statement.

In the above syntax, labelName represents the name of loop. We mainly use the labelled break statement or labelled continue statement when we want to jump outside a nested loop or to continue a loop that is outside the current one.

**Example 3:**

Let us take a very simple example where we will find all even numbers in a two-dimensional array using labeled loops. The array contains three rows, having each row has four elements. We will use a labeled loop to traverse through the array and identify the even numbers.

**Program code:**

package javaProgram;

public class LabeledLoopExample {

public static void main(String[] args)

{

// Creating a two-dimensional array of elements.

int[][] numbers = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 12}

};

// Labeling the outer loop as 'outerLoop'

outerLoop: for(int i = 0; i < numbers.length; i++)

{

// Labeling the inner loop as 'innerLoop'

innerLoop: for(int j = 0; j < numbers[i].length; j++)

{

if(numbers[i][j] % 2 == 0) {

System.out.println(numbers[i][j]);

}

}

}

}

}

Output:

2

4

6

8

10

12

In this example, we have used two nested for loops. We have labeled the outer loop with a name “outerLoop” and the inner loop with a name “innerLoop”. The outer loop iterates over each row of the two-dimensional array, and the inner loop iterates over each element within that row.

Within the inner loop, we have checked the number is even or not. If the current number divides by 2 and the remainder is 0, then it means that the number of even. We have displayed the result on the console.

**Java Labelled Loop Example Program**

1. Let’s take an example program based on labelled loop in Java where we will use a for loop inside another for loop that displays i and j values. It means that we are using nested for loops.

In this program, we will use a label (name) to represent the outer for loop as outer and labelled continue statement. Look at the source code to understand better.

**Program code 1:**

package javaProgram;

public class LabelledLoopEx {

public static void main(String[] args)

{

// Outer loop.

outer: for(int i = 1; i < 5; i++)

{

System.out.println(i);

// Inner loop.

for(int j = 1; j < 3; j++)

{

System.out.println(j);

if(i == j)

continue outer;

}

}

}

}

Output:

1 1 2 1 2 3 1 2 4 1 2

Let’s understand how the flow of execution of this program is working:

a) When i = 1, it displays a value 1. Then, enter inside the inner for loop.

b) Since j is initially set to a value 1, it will print 1. Now, the continue statement will terminate inner loop when i == j and continues with the next iteration of outer loop (counting i).

c) In the next iteration of outer for loop, the value of i will be incremented by 1 and it will display 2 on the console. Then, the control of execution enters inside the inner for loop.

d) Since the value of i is 2, therefore, i is not equal j in the inner for loop iteration. Hence, when i = 2, the values of j will change from 1 to 2 and print 1, 2 on the console.

e) After the complete iteration of inner for loop, the control of execution again goes back to the outer for loop for the next iteration.  
This process continues until the value of i is greater or equal to 5.

2. Let’s take another example program based on labelled loop where we will use statement to come out of nested loops. Look at the source code and its explanation to understand it better.

**Program code 2:**

package javaProgram;

public class LabelledLoopEx2 {

public static void main(String[] args)

{

// Outer loop.

outer: for(int i = 1; i < 3; i++)

{

System.out.println("i: " +i);

// Inner loop.

int j = 1;

while(j < 3)

{

System.out.println("j: " +j);

int x = i + j;

if(x > 2)

break outer;

j++;

}

}

System.out.println("Jumping out of both labelled loops");

}

}

Output:

i: 1

j: 1

j: 2

Jumping out of both labelled loops

As you can observe in this program, the label outer labels the outer loop, and the labelled break statement terminates the flow of execution to come out of both loops.

3. Let’s take an example program where we will use both continue and break statements and produce the following output.

**Program code 3:**

package javaProgram;

public class ContinueUse {

public static void main(String[] args)

{

// Outer loop.

outer: for(int i = 1; i < 100; i++)

{

System.out.println(" ");

if(i >= 10) break;

for(int j = 1; j < 100; j++)

{

System.out.printf(" \* ");

if(j == i)

continue outer;

}

}

System.out.println("Termination by break statement");

}

}

Output:

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

\* \* \* \* \* \*

\* \* \* \* \* \* \*

\* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \*

Termination by break statement

4. Let’s take an example in which we will break out multiple nested loops simultaneously by using labeled loops. Dealing with complex data structures or algorithms becomes much easier with the help of this feature.

**Program code 4:**

package javaProgram;

public class LabeledLoopExample {

public static void main(String[] args)

{

outerLoop: for (int i = 0; i < 3; i++)

{

innerLoop: for (int j = 0; j < 3; j++)

{

if (i == 1 && j == 1)

{

continue outerLoop; // This will skip the rest of the inner loop

}

System.out.println("i: " + i + ", j: " + j);

}

}

}

}

Output:

i: 0, j: 0

i: 0, j: 1

i: 0, j: 2

i: 1, j: 0

i: 2, j: 0

i: 2, j: 1

i: 2, j: 2

In this example, we have used two nested loops, an outer loop labeled as outerLoop, and an inner loop labeled as innerLoop. The outer for loop will iterate over each value of i from 0 to 2. Similarly, the inner for loop will also iterate over each value of j from 0 to 2.

Inside the inner loop, we have checked a conditional expression i == 1 and j == 1 using if statement. If this conditional expression is true, the continue outerLoop; statement will execute, which immediately jumps to the next iteration of the outer loop, skipping the rest of the inner loop for that particular value of i and j.

As you can observe, the execution of inner for loop is skipped for i == 1 and j == 1, resulting in only six iterations instead of nine. This example exhibits how labeled loops we can use with continue to control the flow of execution within nested loops effectively.

**Advantages of Labeled Loop in Java**

There are the following advantages of using labeled loops in Java program. They are:

* Labeled loops enhance the level of control over loop execution, making complex scenarios more manageable.
* Properly labeled loops can make the code more readable and self-explanatory.
* Labeled loops enable us to break out multiple nested loops simultaneously.

**Best Practices for Using Labeled Loops**

Labeled loops is a powerful provided by Java. Here, we have mentioned some key points that you should keep in mind for the best practices:

* Give descriptive and meaningful labels that describe the purpose of the loop.
* Avoid excessive nesting to keep the code clean and understandable.
* Always write clear and well-defined code within the labeled loops so that it should be easy to understand.
* Ensure that the label used with the break or continue statement corresponds to the correct loop.
* Be careful to avoid accidentally creating infinite loops.

**Labeled Loops vs Unlabeled Loops**

In comparison to unlabeled loops, labeled loops add an extra layer of control and can handle more complex situations. However, in simple scenarios, using labeled loops might introduce unnecessary complexity.