

Java if...else Statement

In computer programming, we use the if statement to control the flow of the program. For example, if a certain condition is met, then run a specific block of code. Otherwise, run another code.

For example assigning grades (A, B, C) based on percentage obtained by a student.

- if the percentage is above **90**, assign grade **A**
- if the percentage is above **75**, assign grade **B**
- if the percentage is above **65**, assign grade **C**

There are three forms of `if...else` statements in Java.

1. if statement
2. if...else statement
3. if...else if...else statement
4. Nested if...else statement

1. Java if (if-then) Statement

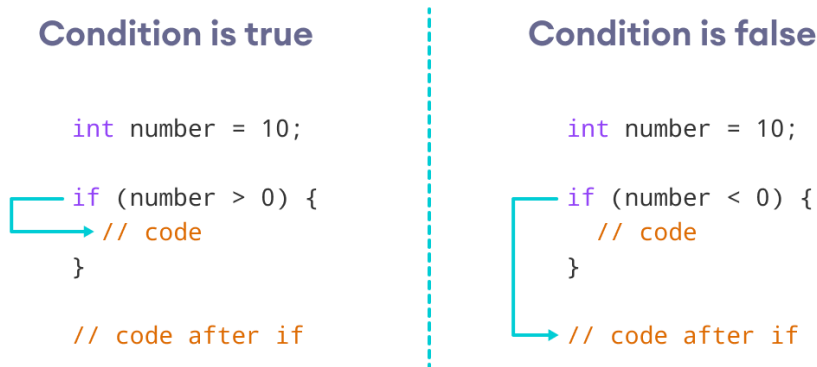
The syntax of a **if-then** statement:

```
if (condition) {  
    // statements  
}
```

Here, *condition* is a boolean expression. It returns either true or false.

- if *condition* evaluates to true, statements inside the body of if are executed
- if *condition* evaluates to false, statements inside the body of if are skipped

How if statement works?



Working of Java if statement

Example 1: Java if Statement

```
class IfStatement {
    public static void main(String[] args) {

        int number = 10;

        // checks if number is greater than 0
        if (number > 0) {
            System.out.println("The number is positive.");
        }

        System.out.println("Statement outside if block");
    }
}
```

Output

```
The number is positive.
Statement outside if block
```

In the above example, we have created a variable named *number*. Notice the test condition,

```
number > 0
```

Here, the condition is checking if *number* is greater than 0. Since *number* is greater than 0, the condition evaluates true.

If we change the variable to a negative integer. Let's say -5.

```
int number = -5;
```

Now, when we run the program, the output will be:

```
Statement outside if block
```

This is because the value of *number* is less than **0**. Hence, the condition evaluates to false. And, the body of if block is skipped.

We can also use Java Strings as the test condition.

Example 2: Java if with String

```
class Main {
    public static void main(String[] args) {
        // create a string variable
        String language = "Java";

        // if statement
        if (language == "Java") {
            System.out.println("Best Programming Language");
        }
    }
}
```

Output

```
Best Programming Language
```

In the above example, we are comparing two strings in the if block.

2. Java if...else (if-then-else) Statement

The if statement executes a certain section of code if the test expression is evaluated to true. However, if the test expression is evaluated to false, it does nothing.

In this case, we can use an optional else block. Statements inside the body of else block are executed if the test expression is evaluated to false. This is known as the **if...else** statement in Java.

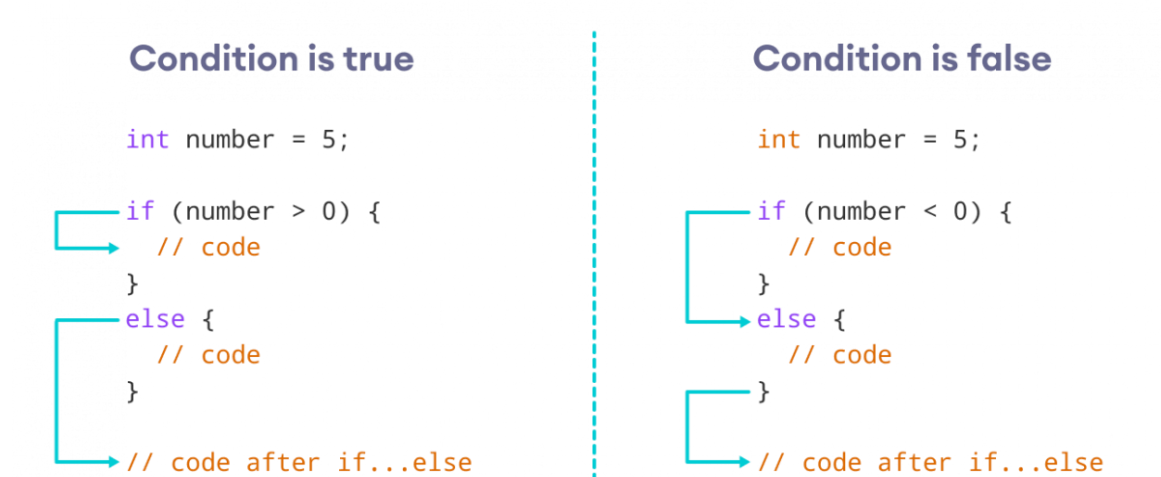
The syntax of the **if...else** statement is:

```
if (condition) {
    // codes in if block
}
```

```
else {
    // codes in else block
}
```

Here, the program will do one task (codes inside if block) if the condition is true and another task (codes inside else block) if the condition is false.

How the if...else statement works?



Working of Java if-else statements

Example 3: Java if...else Statement

```
class Main {
    public static void main(String[] args) {
        int number = 10;

        // checks if number is greater than 0
        if (number > 0) {
            System.out.println("The number is positive.");
        }

        // execute this block
        // if number is not greater than 0
        else {
            System.out.println("The number is not positive.");
        }

        System.out.println("Statement outside if...else block");
    }
}
```

Output

```
The number is positive.  
Statement outside if...else block
```

In the above example, we have a variable named *number*. Here, the test expression `number > 0` checks if *number* is greater than 0.

Since the value of the *number* is *10*, the test expression evaluates to true. Hence code inside the body of if is executed.

Now, change the value of the *number* to a negative integer. Let's say -5.

```
int number = -5;
```

If we run the program with the new value of *number*, the output will be:

```
The number is not positive.  
Statement outside if...else block
```

Here, the value of *number* is -5. So the test expression evaluates to false. Hence code inside the body of else is executed.

3. Java if...else...if Statement

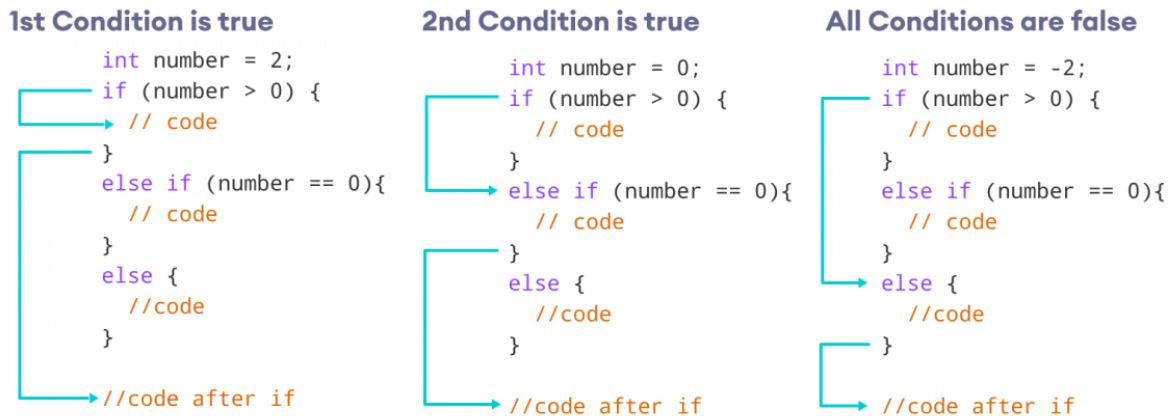
In Java, we have an **if...else...if** ladder, that can be used to execute one block of code among multiple other blocks.

```
if (condition1) {  
    // codes  
}  
else if(condition2) {  
    // codes  
}  
else if (condition3) {  
    // codes  
}  
.  
.  
else {  
    // codes  
}
```

Here, if statements are executed from the top towards the bottom. When the test condition is true, codes inside the body of that if block is executed. And, program control jumps outside the if...else...if ladder.

If all test expressions are false, codes inside the body of else are executed.

How the if...else...if ladder works?



Working of if...else...if ladder

Example 4: Java if...else...if Statement

```
class Main {
    public static void main(String[] args) {

        int number = 0;

        // checks if number is greater than 0
        if (number > 0) {
            System.out.println("The number is positive.");
        }

        // checks if number is less than 0
        else if (number < 0) {
            System.out.println("The number is negative.");
        }

        // if both condition is false
        else {
            System.out.println("The number is 0.");
        }
    }
}
```

Output

The number is 0.

In the above example, we are checking whether *number* is **positive**, **negative**, or **zero**. Here, we have two condition expressions:

- `number > 0` - checks if *number* is greater than 0
- `number < 0` - checks if *number* is less than 0

Here, the value of *number* is 0. So both the conditions evaluate to false. Hence the statement inside the body of else is executed.

Note: Java provides a special operator called **ternary operator**, which is a kind of shorthand notation of **if...else...if** statement.

4. Java Nested if..else Statement

In Java, it is also possible to use if..else statements inside an if...else statement. It's called the nested if...else statement.

Here's a program to find the largest of 3 numbers using the nested if...else statement.

Example 5: Nested if...else Statement

```
class Main {
    public static void main(String[] args) {

        // declaring double type variables
        Double n1 = -1.0, n2 = 4.5, n3 = -5.3, largest;

        // checks if n1 is greater than or equal to n2
        if (n1 >= n2) {

            // if...else statement inside the if block
            // checks if n1 is greater than or equal to n3
            if (n1 >= n3) {
                largest = n1;
            }

            else {
                largest = n3;
            }
        } else {
```

```

        // if..else statement inside else block
        // checks if n2 is greater than or equal to n3
        if (n2 >= n3) {
            largest = n2;
        }

        else {
            largest = n3;
        }
    }

    System.out.println("Largest Number: " + largest);
}
}

```

Output:

```
Largest Number: 4.5
```

In the above programs, we have assigned the value of variables ourselves to make this easier.

However, in real-world applications, these values may come from user input data, log files, form submission, etc.

Java switch Statement

The switch statement allows us to execute a block of code among many alternatives.

The syntax of the switch statement in Java is:

```

switch (expression) {

    case value1:
        // code
        break;

    case value2:
        // code
        break;

    ...

    default:
        // default statements
}

```



```
}
```

How does the switch-case statement work?

The expression is evaluated once and compared with the values of each case.

- If expression matches with value1, the code of case value1 are executed. Similarly, the code of case value2 is executed if expression matches with value2.
- If there is no match, the code of the default case is executed.

Note: The working of the switch-case statement is similar to the Java if...else...if ladder. However, the syntax of the switch statement is cleaner and much easier to read and write.

Example: Java switch Statement

```
// Java Program to check the size
// using the switch...case statement

class Main {
    public static void main(String[] args) {

        int number = 44;
        String size;

        // switch statement to check size
        switch (number) {

            case 29:
                size = "Small";
                break;

            case 42:
                size = "Medium";
                break;

            // match the value of week
            case 44:
                size = "Large";
                break;

            case 48:
                size = "Extra Large";
                break;

            default:
                size = "Unknown";
```

```
        break;
    }
    System.out.println("Size: " + size);
}
}
```

Output:

Size: Large

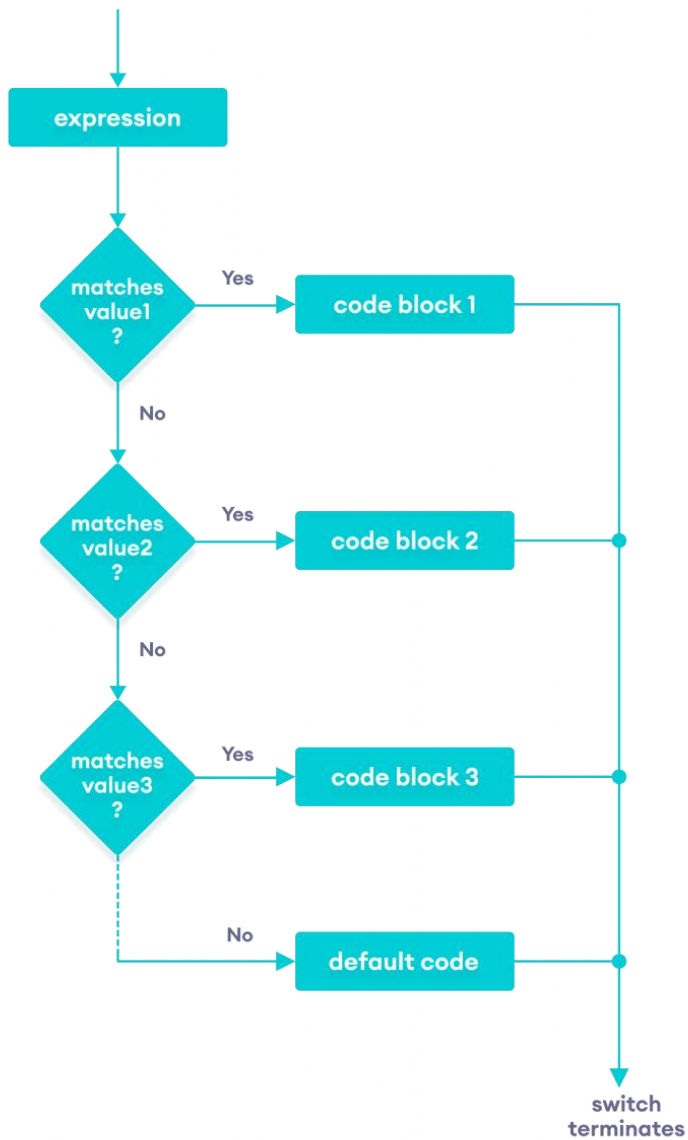
In the above example, we have used the switch statement to find the size. Here, we have a variable *number*. The variable is compared with the value of each case statement.

Since the value matches with **44**, the code of case 44 is executed.

```
size = "Large";
break;
```

Here, the *size* variable is assigned with the value Large.

Flowchart of switch Statement



Flow chart of the Java switch statement

break statement in Java switch...case

Notice that we have been using break in each case block.

```
...
case 29:
    size = "Small";
    break;
...
```

The break statement is used to terminate the **switch-case** statement. If break is not used, all the cases after the matching case are also executed. For example,

```
class Main {
    public static void main(String[] args) {

        int expression = 2;

        // switch statement to check size
        switch (expression) {
            case 1:
                System.out.println("Case 1");

                // matching case
            case 2:
                System.out.println("Case 2");

            case 3:
                System.out.println("Case 3");

            default:
                System.out.println("Default case");
        }
    }
}
```

Output

```
Case 2
Case 3
Default case
```

In the above example, *expression* matches with case 2. Here, we haven't used the break statement after each case.

Hence, all the cases after case 2 are also executed.

This is why the break statement is needed to terminate the **switch-case** statement after the matching case.

default case in Java switch-case

The switch statement also includes an optional default case. It is executed when the expression doesn't match any of the cases. For example,

```
class Main {
    public static void main(String[] args) {

        int expression = 9;

        switch(expression) {

            case 2:
                System.out.println("Small Size");
                break;

            case 3:
                System.out.println("Large Size");
                break;

            // default case
            default:
                System.out.println("Unknown Size");
        }
    }
}
```

Output

Unknown Size

In the above example, we have created a **switch-case** statement. Here, the value of *expression* doesn't match with any of the cases.

Hence, the code inside the **default case** is executed.

```
default:
    System.out.println("Unknown Size);
```

Note: The Java switch statement only works with:

- **Primitive data types:** byte, short, char, and int
- **Enumerated types**
- **String Class**
- **Wrapper Classes:** Character, Byte, Short, and Integer.

Java for Loop

In computer programming, loops are used to repeat a block of code. For example, if you want to show a message 100 times, then rather than typing the same code 100 times, you can use a loop.

In Java, there are three types of loops.

- for loop
- while loop
- do...while loop

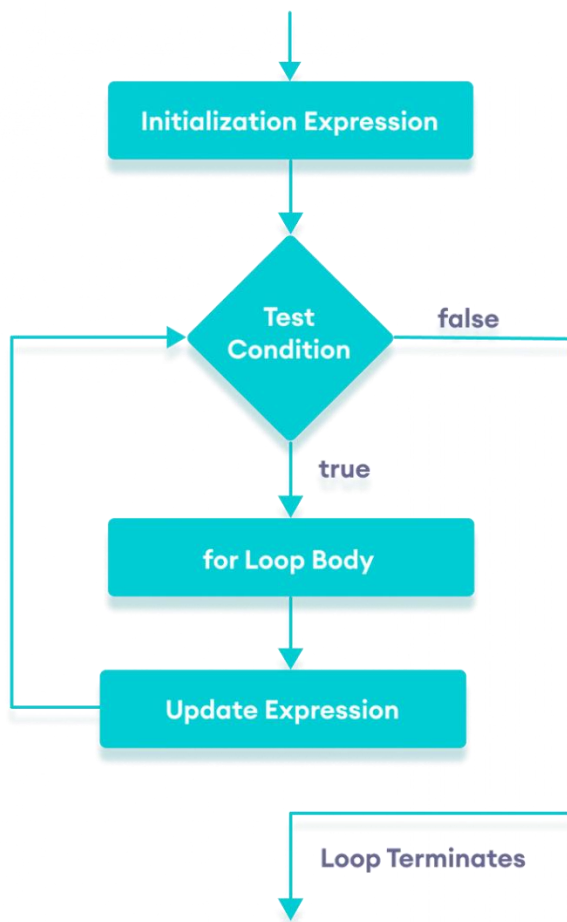
Java for Loop

Java for loop is used to run a block of code for a certain number of times. The syntax of for loop is:

```
for (initialExpression; testExpression; updateExpression) {  
    // body of the loop  
}
```

Here,

1. The **initialExpression** initializes and/or declares variables and executes only once.
2. The **condition** is evaluated. If the **condition** is true, the body of the for loop is executed.
3. The **updateExpression** updates the value of **initialExpression**.
4. The **condition** is evaluated again. The process continues until the **condition** is false.



Flowchart of Java for loop

Example 1: Display a Text Five Times

```
// Program to print a text 5 times

class Main {
    public static void main(String[] args) {

        int n = 5;
        // for loop
        for (int i = 1; i <= n; ++i) {
            System.out.println("Java is fun");
        }
    }
}
```

Output

```
Java is fun
Java is fun
```

```
Java is fun
Java is fun
Java is fun
```

Here is how this program works.

Iteration Variable Condition: $i \leq n$			Action
1st	$i = 1$ $n = 5$	true	Java is fun is printed. i is increased to 2.
2nd	$i = 2$ $n = 5$	true	Java is fun is printed. i is increased to 3.
3rd	$i = 3$ $n = 5$	true	Java is fun is printed. i is increased to 4.
4th	$i = 4$ $n = 5$	true	Java is fun is printed. i is increased to 5.
5th	$i = 5$ $n = 5$	true	Java is fun is printed. i is increased to 6.
6th	$i = 6$ $n = 5$	false	The loop is terminated.

Example 2: Display numbers from 1 to 5

```
// Program to print numbers from 1 to 5

class Main {
    public static void main(String[] args) {

        int n = 5;
        // for loop
        for (int i = 1; i <= n; ++i) {
            System.out.println(i);
        }
    }
}
```

Output

```
1
2
3
```


4
5

Here is how the program works.

Iteration Variable Condition: $i \leq n$			Action
1st	$i = 1$ $n = 5$	true	1 is printed. i is increased to 2.
2nd	$i = 2$ $n = 5$	true	2 is printed. i is increased to 3.
3rd	$i = 3$ $n = 5$	true	3 is printed. i is increased to 4.
4th	$i = 4$ $n = 5$	true	4 is printed. i is increased to 5.
5th	$i = 5$ $n = 5$	true	5 is printed. i is increased to 6.
6th	$i = 6$ $n = 5$	false	The loop is terminated.

Example 3: Display Sum of n Natural Numbers

// Program to find the sum of natural numbers from 1 to 1000.

```
class Main {
    public static void main(String[] args) {

        int sum = 0;
        int n = 1000;

        // for loop
        for (int i = 1; i <= n; ++i) {
            // body inside for loop
            sum += i;      // sum = sum + i
        }

        System.out.println("Sum = " + sum);
    }
}
```

Output:

Sum = 500500

Here, the value of *sum* is **0** initially. Then, the for loop is iterated from $i = 1$ to 1000. In each iteration, i is added to *sum* and its value is increased by **1**.

When i becomes **1001**, the test condition is false and *sum* will be equal to $0 + 1 + 2 + \dots + 1000$.

The above program to add the sum of natural numbers can also be written as

```
// Program to find the sum of natural numbers from 1 to 1000.
```

```
class Main {
    public static void main(String[] args) {

        int sum = 0;
        int n = 1000;

        // for loop
        for (int i = n; i >= 1; --i) {
            // body inside for loop
            sum += i;      // sum = sum + i
        }

        System.out.println("Sum = " + sum);
    }
}
```

Java for-each Loop

The Java for loop has an alternative syntax that makes it easy to iterate through arrays and collections. For example,

```
// print array elements
```

```
class Main {
    public static void main(String[] args) {

        // create an array
        int[] numbers = {3, 7, 5, -5};

        // iterating through the array
        for (int number: numbers) {
            System.out.println(number);
        }
    }
}
```

Output

```
3
7
5
-5
```

Here, we have used the **for-each loop** to print each element of the *numbers* array one by one.

In the first iteration of the loop, *number* will be 3, *number* will be 7 in second iteration and so on.

Java Infinite for Loop

If we set the **test expression** in such a way that it never evaluates to false, the for loop will run forever. This is called infinite for loop. For example,

```
// Infinite for Loop

class Infinite {
    public static void main(String[] args) {

        int sum = 0;

        for (int i = 1; i <= 10; --i) {
            System.out.println("Hello");
        }
    }
}
```

Here, the test expression ,*i <= 10*, is never false and Hello is printed repeatedly until the memory runs out.

Java for-each Loop

In Java, the **for-each** loop is used to iterate through elements of arrays and collections (like ArrayList). It is also known as the enhanced for loop.

for-each Loop Sytnax

The syntax of the Java **for-each** loop is:

```
for(dataType item : array) {  
    ...  
}
```

Here,

- **array** - an array or a collection
- **item** - each item of array/collection is assigned to this variable
- **dataType** - the data type of the array/collection

Example 1: Print Array Elements

```
// print array elements  
  
class Main {  
    public static void main(String[] args) {  
  
        // create an array  
        int[] numbers = {3, 9, 5, -5};  
  
        // for each loop  
        for (int number: numbers) {  
            System.out.println(number);  
        }  
    }  
}
```

Output

```
3  
9  
5  
-5
```

Here, we have used the **for-each loop** to print each element of the *numbers* array one by one.

- In the first iteration, *item* will be 3.
- In the second iteration, *item* will be 9.
- In the third iteration, *item* will be 5.
- In the fourth iteration, *item* will be -5.

Example 2: Sum of Array Elements

```
// Calculate the sum of all elements of an array

class Main {
    public static void main(String[] args) {

        // an array of numbers
        int[] numbers = {3, 4, 5, -5, 0, 12};
        int sum = 0;

        // iterating through each element of the array
        for (int number: numbers) {
            sum += number;
        }

        System.out.println("Sum = " + sum);
    }
}
```

Output:

Sum = 19

In the above program, the execution of the for each loop looks as:

Iteration	Variables
1	<i>number</i> = 3 <i>sum</i> = 0 + 3 = 3
2	<i>number</i> = 4 <i>sum</i> = 3 + 4 = 7
3	<i>number</i> = 5 <i>sum</i> = 7 + 5 = 12
4	<i>number</i> = -5 <i>sum</i> = 12 + (-5) = 7

```
5      number = 0
      sum = 7 + 0 = 7

6      number = 12
      sum = 7 + 12 = 19
```

As we can see, we have added each element of the *numbers* array to the *sum* variable in each iteration of the loop.

for loop Vs for-each loop

Let's see how a for-each loop is different from a regular Java for loop.

1. Using for loop

```
class Main {
    public static void main(String[] args) {

        char[] vowels = {'a', 'e', 'i', 'o', 'u'};

        // iterating through an array using a for loop
        for (int i = 0; i < vowels.length; ++ i) {
            System.out.println(vowels[i]);
        }
    }
}
```

Output:

```
a
e
i
o
u
```

2. Using for-each Loop

```
class Main {
    public static void main(String[] args) {

        char[] vowels = {'a', 'e', 'i', 'o', 'u'};

        // iterating through an array using the for-each loop
        for (char item: vowels) {
            System.out.println(item);
        }
    }
}
```

```
}
```

Output:

```
a  
e  
i  
o  
u
```

Here, the output of both programs is the same. However, the **for-each** loop is easier to write and understand.

This is why the **for-each** loop is preferred over the **for** loop when working with arrays and collections.

Java while and do...while Loop

In computer programming, loops are used to repeat a block of code. For example, if you want to show a message 100 times, then you can use a loop. It's just a simple example; you can achieve much more with loops.

In the previous tutorial, you learned about Java for loop. Here, you are going to learn about while and do...while loops.

Java while loop

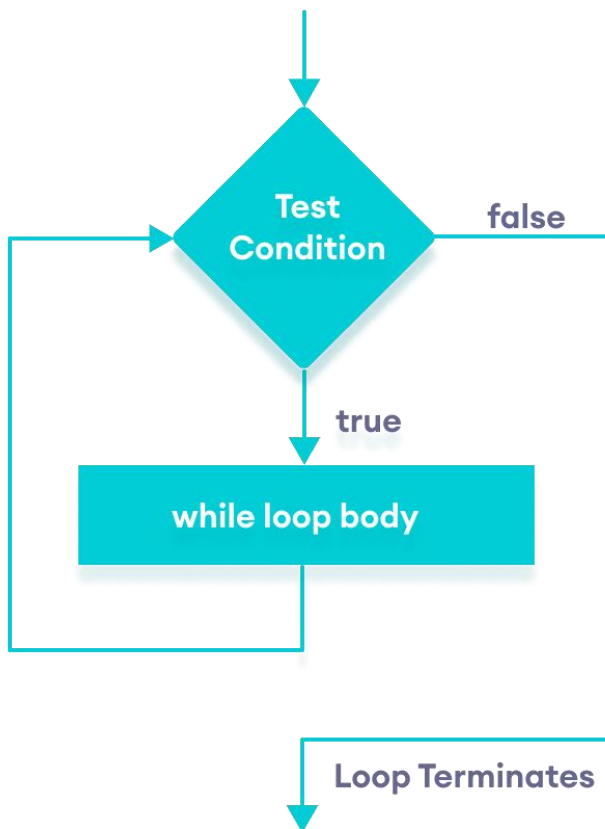
Java while loop is used to run a specific code until a certain condition is met. The syntax of the while loop is:

```
while (testExpression) {  
    // body of loop  
}
```

Here,

1. A while loop evaluates the **testExpression** inside the parenthesis ().
2. If the **testExpression** evaluates to true, the code inside the while loop is executed.
3. The **testExpression** is evaluated again.
4. This process continues until the **testExpression** is false.
5. When the **testExpression** evaluates to false, the loop stops.

Flowchart of while loop



Flowchart of Java while loop

Example 1: Display Numbers from 1 to 5

// Program to display numbers from 1 to 5

```
class Main {
    public static void main(String[] args) {

        // declare variables
        int i = 1, n = 5;

        // while loop from 1 to 5
        while(i <= n) {
            System.out.println(i);
            i++;
        }
    }
}
```


Output

1
2
3
4
5

Here is how this program works.

Iteration	Variable	Condition: $i \leq n$	Action
1st	$i = 1$ $n = 5$	true	1 is printed. i is increased to 2.
2nd	$i = 2$ $n = 5$	true	2 is printed. i is increased to 3.
3rd	$i = 3$ $n = 5$	true	3 is printed. i is increased to 4.
4th	$i = 4$ $n = 5$	true	4 is printed. i is increased to 5.
5th	$i = 5$ $n = 5$	true	5 is printed. i is increased to 6.
6th	$i = 6$ $n = 5$	false	The loop is terminated

Example 2: Sum of Positive Numbers Only

```
// Java program to find the sum of positive numbers
import java.util.Scanner;
```

```
class Main {
    public static void main(String[] args) {

        int sum = 0;

        // create an object of Scanner class
        Scanner input = new Scanner(System.in);

        // take integer input from the user
        System.out.println("Enter a number");
        int number = input.nextInt();

        // while loop continues
        // until entered number is positive
```

```

while (number >= 0) {
    // add only positive numbers
    sum += number;

    System.out.println("Enter a number");
    number = input.nextInt();
}

System.out.println("Sum = " + sum);
input.close();
}
}

```

Output

```

Enter a number
25
Enter a number
9
Enter a number
5
Enter a number
-3
Sum = 39

```

In the above program, we have used the Scanner class to take input from the user. Here, nextInt() takes integer input from the user.

The while loop continues until the user enters a negative number. During each iteration, the number entered by the user is added to the sum variable.

When the user enters a negative number, the loop terminates. Finally, the total sum is displayed.

Java do...while loop

The do...while loop is similar to while loop. However, the body of do...while loop is executed once before the test expression is checked. For example,

```

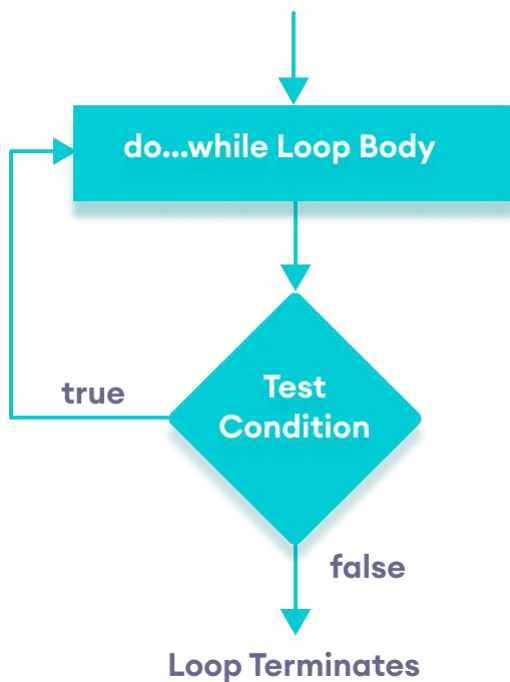
do {
    // body of loop
} while(textExpression)

```

Here,

1. The body of the loop is executed at first. Then the **textExpression** is evaluated.
2. If the **textExpression** evaluates to true, the body of the loop inside the do statement is executed again.
3. The **textExpression** is evaluated once again.
4. If the **textExpression** evaluates to true, the body of the loop inside the do statement is executed again.
5. This process continues until the **textExpression** evaluates to false. Then the loop stops.

Flowchart of do...while loop



Flowchart of Java do while loop

Let's see the working of do...while loop.

Example 3: Display Numbers from 1 to 5

```
// Java Program to display numbers from 1 to 5
```

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

```
// Program to find the sum of natural numbers from 1 to 100.
```

```
class Main {
```

```

public static void main(String[] args) {

    int i = 1, n = 5;

    // do...while loop from 1 to 5
    do {
        System.out.println(i);
        i++;
    } while(i <= n);
}

```

Output

```

1
2
3
4
5

```

Here is how this program works.

Iteration	Variable	Condition: $i \leq n$	Action
	$i = 1$ $n = 5$	not checked	1 is printed. i is increased to 2 .
1st	$i = 2$ $n = 5$	true	2 is printed. i is increased to 3 .
2nd	$i = 3$ $n = 5$	true	3 is printed. i is increased to 4 .
3rd	$i = 4$ $n = 5$	true	4 is printed. i is increased to 5 .
4th	$i = 5$ $n = 5$	true	5 is printed. i is increased to 6 .
5th	$i = 6$ $n = 5$	false	The loop is terminated

Example 4: Sum of Positive Numbers

```

// Java program to find the sum of positive numbers
import java.util.Scanner;

```

```

class Main {
    public static void main(String[] args) {

```

```

    int sum = 0;
    int number = 0;

    // create an object of Scanner class
    Scanner input = new Scanner(System.in);

    // do...while loop continues
    // until entered number is positive
    do {
        // add only positive numbers
        sum += number;
        System.out.println("Enter a number");
        number = input.nextInt();
    } while(number >= 0);

    System.out.println("Sum = " + sum);
    input.close();
}
}

```

Output 1

```

Enter a number
25
Enter a number
9
Enter a number
5
Enter a number
-3
Sum = 39

```

Here, the user enters a positive number, that number is added to the *sum* variable. And this process continues until the number is negative. When the number is negative, the loop terminates and displays the sum without adding the negative number.

Output 2

```

Enter a number
-8
Sum is 0

```

Here, the user enters a negative number. The test condition will be false but the code inside of the loop executes once.

Infinite while loop

If **the condition** of a loop is always true, the loop runs for infinite times (until the memory is full). For example,

```
// infinite while loop
while(true){
    // body of loop
}
```

Here is an example of an infinite do...while loop.

```
// infinite do...while loop
int count = 1;
do {
    // body of loop
} while(count == 1)
```

In the above programs, the **textExpression** is always true. Hence, the loop body will run for infinite times.

for and while loops

The for loop is used when the number of iterations is known. For example,

```
for (let i = 1; i <=5; ++i) {
    // body of loop
}
```

And while and do...while loops are generally used when the number of iterations is unknown. For example,

```
while (condition) {
    // body of loop
}
```

Java break Statement

While working with loops, it is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

In such cases, break and continue statements are used.

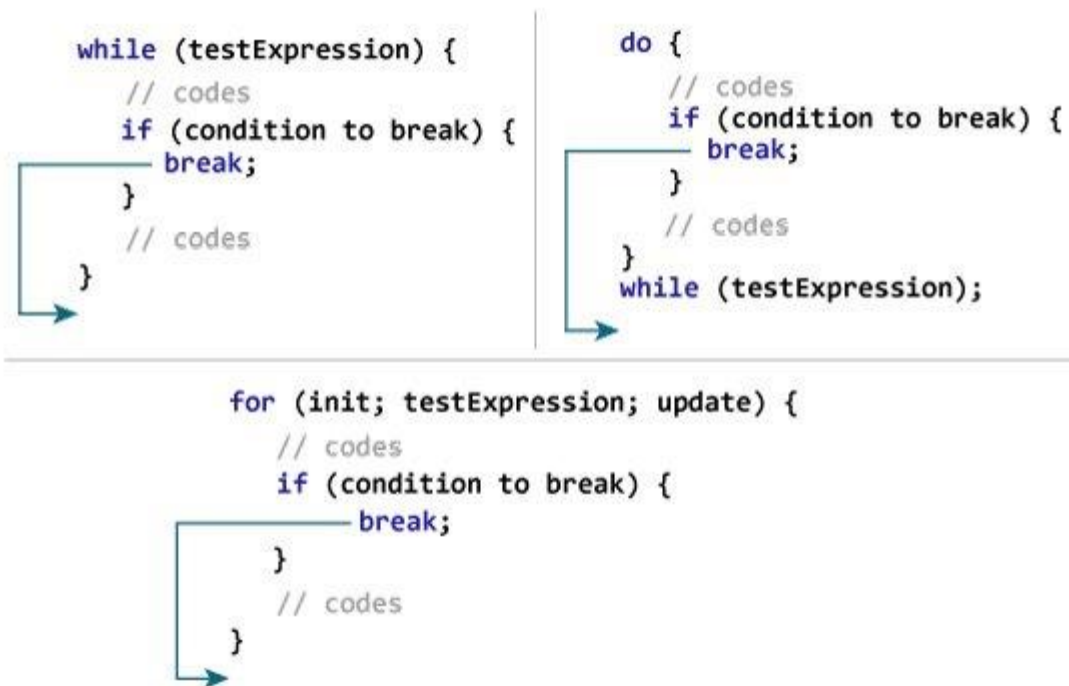
The break statement in Java terminates the loop immediately, and the control of the program moves to the next statement following the loop.

It is almost always used with decision-making statements (Java if...else Statement).

Here is the syntax of the break statement in Java:

```
break;
```

How break statement works?



Working of Java break Statement

Example 1: Java break statement

```
class Test {  
    public static void main(String[] args) {  
  
        // for loop
```

```

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

            // if the value of i is 5 the loop terminates
            if (i == 5) {
                break;
            }
            System.out.println(i);
        }
    }
}

```

Output:

```

1
2
3
4

```

In the above program, we are using the for loop to print the value of *i* in each iteration.

```

if (i == 5) {
    break;
}

```

This means when the value of *i* is equal to 5, the loop terminates. Hence we get the output with values less than 5 only.

Example 2: Java break statement

The program below calculates the sum of numbers entered by the user until user enters a negative number.

To take input from the user, we have used the Scanner object.

```

import java.util.Scanner;

class UserInputSum {
    public static void main(String[] args) {

        Double number, sum = 0.0;

        // create an object of Scanner
        Scanner input = new Scanner(System.in);

        while (true) {
            System.out.print("Enter a number: ");

```



```

        // takes double input from user
        number = input.nextDouble();

        // if number is negative the loop terminates
        if (number < 0.0) {
            break;
        }

        sum += number;
    }
    System.out.println("Sum = " + sum);
}
}

```

Output:

```

Enter a number: 3.2
Enter a number: 5
Enter a number: 2.3
Enter a number: 0
Enter a number: -4.5
Sum = 10.5

```

In the above program, the test expression of the while loop is always true. Here, notice the line,

```

if (number < 0.0) {
    break;
}

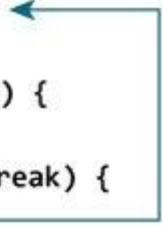
```

This means when the user input negative numbers, the while loop is terminated.

Java break and Nested Loop

In the case of nested loops, the break statement terminates the innermost loop.

```
while (testExpression) {  
    // codes  
    while (testExpression) {  
        // codes  
        if (condition to break) {  
            break;  
        }  
        // codes  
    }  
    // codes  
}
```



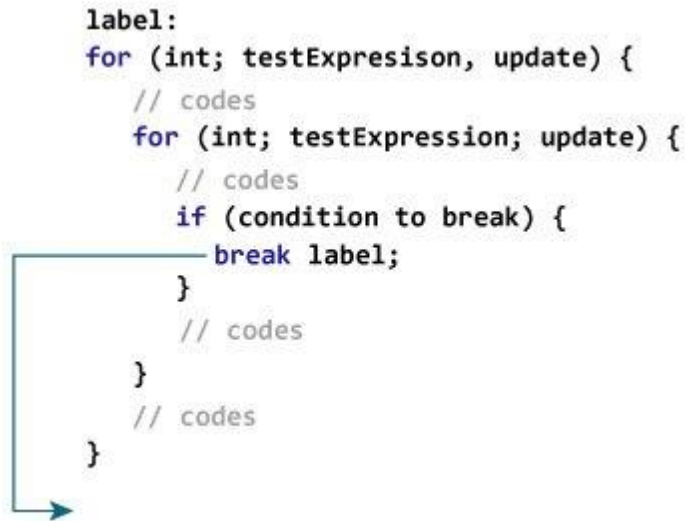
Working of break Statement with Nested Loops

Here, the break statement terminates the innermost while loop, and control jumps to the outer loop.

Labeled break Statement

Till now, we have used the unlabeled break statement. It terminates the innermost loop and switch statement. However, there is another form of break statement in Java known as the labeled break.

We can use the labeled break statement to terminate the outermost loop as well.



Working of the labeled break statement in Java

As you can see in the above image, we have used the *label* identifier to specify the outer loop. Now, notice how the break statement is used (break label;).

Here, the break statement is terminating the labeled statement (i.e. outer loop). Then, the control of the program jumps to the statement after the labeled statement.

Here's another example:

```

while (testExpression) {
    // codes
    second:
    while (testExpression) {
        // codes
        while(testExpression) {
            // codes
            break second;
        }
    }
    // control jumps here
}

```

In the above example, when the statement break second; is executed, the while loop labeled as *second* is terminated. And, the control of the program moves to the statement after the second while loop.

Example 3: labeled break Statement

```

class LabeledBreak {
    public static void main(String[] args) {

```

```

// the for loop is labeled as first
first:
for( int i = 1; i < 5; i++) {

    // the for loop is labeled as second
    second:
    for(int j = 1; j < 3; j ++ ) {
        System.out.println("i = " + i + "; j = " +j);

        // the break statement breaks the first for loop
        if ( i == 2)
            break first;
    }
}
}

```

Output:

```

i = 1; j = 1
i = 1; j = 2
i = 2; j = 1

```

In the above example, the labeled break statement is used to terminate the loop labeled as first. That is,

```

first:
for(int i = 1; i < 5; i++) {...}

```

Here, if we change the statement break first; to break second; the program will behave differently. In this case, for loop labeled as second will be terminated. For example,

```

class LabeledBreak {
    public static void main(String[] args) {

        // the for loop is labeled as first
        first:
        for( int i = 1; i < 5; i++) {

            // the for loop is labeled as second
            second:
            for(int j = 1; j < 3; j ++ ) {

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

                // the break statement terminates the loop labeled as
                second
                if ( i == 2)

```

```

        break second;
    }
}
}

```

Output:

```

i = 1; j = 1
i = 1; j = 2
i = 2; j = 1
i = 3; j = 1
i = 3; j = 2
i = 4; j = 1
i = 4; j = 2

```

Note: The break statement is also used to terminate cases inside the switch statement.

Java continue Statement

While working with loops, sometimes you might want to skip some statements or terminate the loop. In such cases, break and continue statements are used.

Java continue

The continue statement skips the current iteration of a loop (for, while, do...while, etc).

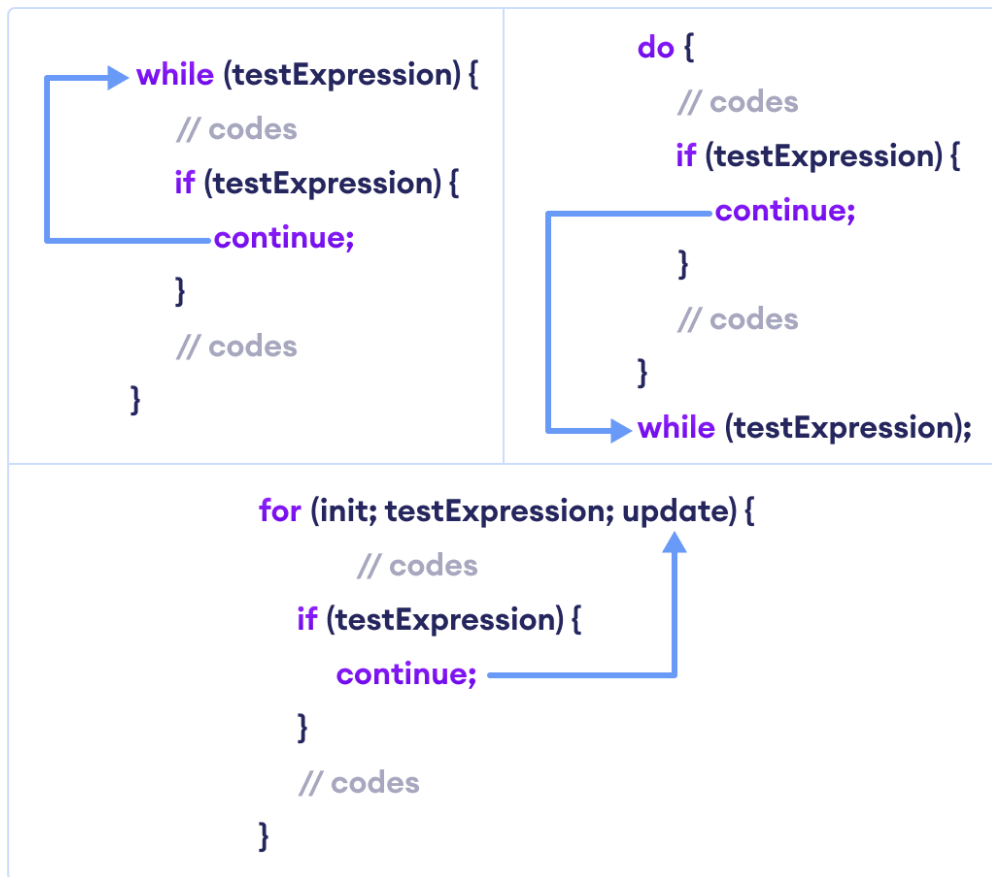
After the continue statement, the program moves to the end of the loop. And, test expression is evaluated (update statement is evaluated in case of the for loop).

Here's the syntax of the continue statement.

```
continue;
```

Note: The continue statement is almost always used in decision-making statements (if...else Statement).

Working of Java continue statement



Working of Java continue Statement

Example 1: Java continue statement

```
class Main {  
    public static void main(String[] args) {  
  
        // for loop  
        for (int i = 1; i <= 10; ++i) {  
  
            // if value of i is between 4 and 9  
            // continue is executed  
            if (i > 4 && i < 9) {  
                continue;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

Output:

```
1
2
3
4
9
10
```

In the above program, we are using the for loop to print the value of i in each iteration. To know how for loop works, visit [Java for loop](#). Notice the statement,

```
if (i > 5 && i < 9) {
    continue;
}
```

Here, the continue statement is executed when the value of i becomes more than 4 and less than 9.

It then skips the print statement inside the loop. Hence we get the output with values 5, 6, 7, and 8 skipped.

Example 2: Compute the sum of 5 positive numbers

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {

        Double number, sum = 0.0;
        // create an object of Scanner
        Scanner input = new Scanner(System.in);

        for (int i = 1; i < 6; ++i) {
            System.out.print("Enter number " + i + " : ");
            // takes input from the user
            number = input.nextDouble();

            // if number is negative
            // continue statement is executed
            if (number <= 0.0) {
                continue;
            }

            sum += number;
        }
        System.out.println("Sum = " + sum);
        input.close();
    }
}
```

```
}
```

Output:

```
Enter number 1: 2.2
Enter number 2: 5.6
Enter number 3: 0
Enter number 4: -2.4
Enter number 5: -3
Sum = 7.8
```

In the above example, we have used the for loop to print the sum of 5 positive numbers. Notice the line,

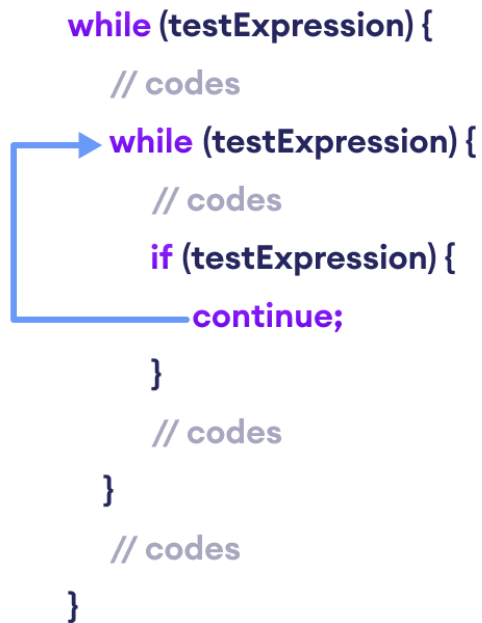
```
if (number < 0.0) {
    continue;
}
```

Here, when the user enters a negative number, the continue statement is executed. This skips the current iteration of the loop and takes the program control to the update expression of the loop.

Note: To take input from the user, we have used the Scanner object.

Java continue with Nested Loop

In the case of nested loops in Java, the continue statement skips the current iteration of the innermost loop.



Working of Java continue statement with Nested Loops

Example 3: continue with Nested Loop

```
class Main {  
    public static void main(String[] args) {  
  
        int i = 1, j = 1;  
  
        // outer loop  
        while (i <= 3) {  
  
            System.out.println("Outer Loop: " + i);  
  
            // inner loop  
            while(j <= 3) {  
  
                if(j == 2) {  
                    j++;  
                    continue;  
                }  
  
            }  
  
        }  
    }  
}
```

```

        System.out.println("Inner Loop: " + j);
        j++;
    }
    i++;
}
}

```

Output

```

Outer Loop: 1
Inner Loop: 1
Inner Loop: 3
Outer Loop: 2
Outer Loop: 3

```

In the above example, we have used the nested while loop. Note that we have used the continue statement inside the inner loop.

```

if(j == 2) {
    j++;
    continue;
}

```

Here, when the value of *j* is **2**, the value of *j* is increased and the continue statement is executed.

This skips the iteration of the inner loop. Hence, the text *Inner Loop: 2* is skipped from the output.

Labeled continue Statement

Till now, we have used the unlabeled continue statement. However, there is another form of continue statement in Java known as **labeled continue**.

It includes the label of the loop along with the continue keyword. For example,

```

continue label;

```

Here, the continue statement skips the current iteration of the loop specified by *label*.

```

label:
→ while (testExpression) {
    // codes
    while (testExpression) {
        // codes
        if (testExpression) {
            continue label;
        }
        // codes
    }
    // codes
}

```

Working of the Java labeled continue Statement

We can see that the label identifier specifies the outer loop. Notice the use of the continue inside the inner loop.

Here, the continue statement is skipping the current iteration of the labeled statement (i.e. outer loop). Then, the program control goes to the next iteration of the labeled statement.

Example 4: labeled continue Statement

```

class Main {
    public static void main(String[] args) {

        // outer loop is labeled as first
        first:
        for (int i = 1; i < 6; ++i) {

            // inner loop
            for (int j = 1; j < 5; ++j) {
                if (i == 3 || j == 2)

                    // skips the current iteration of outer loop
                    continue first;
            }
        }
    }
}

```

```

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

```

Output:

```

i = 1; j = 1
i = 2; j = 1
i = 4; j = 1
i = 5; j = 1

```

In the above example, the labeled continue statement is used to skip the current iteration of the loop labeled as *first*.

```

if (i==3 || j==2)
    continue first;

```

Here, we can see the outermost for loop is labeled as *first*,

```

first:
for (int i = 1; i < 6; ++i) {...}

```

Hence, the iteration of the outer for loop is skipped if the value of *i* is 3 or the value of *j* is 2.

Note: The use of labeled continue is often discouraged as it makes your code hard to understand. If you are in a situation where you have to use labeled continue, refactor your code and try to solve it in a different way to make it more readable.