Java Control Statements:

# Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](https://www.javatpoint.com/java-tutorial) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   * if statements
   * switch statement
2. Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop
3. Jump statements
   * break statement
   * continue statement

### **Decision-Making statements:**

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

### **1) If Statement:**

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

Suppose a condition is true**if a statement**is used to run the program. It is also known as a one-way selection statement. If a condition is used, an argument is passed, and if it is satisfied, the corresponding code is executed; otherwise, nothing happens.

### Syntax

The syntax of the If statement is:-

if (expression) {

    // You can enter the code here

}

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y > 20) {
6. System.out.println("x + y is greater than 20");
7. }
8. }
9. }

**Output:**

x + y is greater than 20

### **2) if-else statement**

The [if-else statement](https://www.javatpoint.com/java-if-else) is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y < 10) {
6. System.out.println("x + y is less than      10");
7. }   **else** {
8. System.out.println("x + y is greater than 20");
9. }
10. }
11. }

**Output:**

x + y is greater than 20

### **3) if-else-if ladder:**

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. String city = "Delhi";
4. **if**(city == "Meerut") {
5. System.out.println("city is meerut");
6. }**else** **if** (city == "Noida") {
7. System.out.println("city is noida");
8. }**else** **if**(city == "Agra") {
9. System.out.println("city is agra");
10. }**else** {
11. System.out.println(city);
12. }
13. }
14. }

**Output:**

Delhi

## Nested If Statement

If conditions inside another if conditions are called as Nested If conditional statements in java.

### Syntax

The syntax of the nested if conditional statements in java are:-

if (condition1) {

   // Statement 1 will execute

   if (condition2) {

     // Statement 2 will execute

   }

}

### Working of Nested If Statement

The way nested if statements operate assumes that they must first become true and sufficient for the other states with the second condition for them to be used, even though other statements may choose to proceed with a false condition if the first condition is met.

### Example

The example below helps us understand the Nested If conditional statements in Java.

import java.util.\*;

class sample {

 public static void main(String[] args) {

   int i = 10;

   if (i == 10) {

     // First if statement

     if (i < 20)

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

     // Nested - if statement

     if (i < 15)

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

     else

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

   }

 }

}

### **Switch Statement:**

In Java, [Switch statements](https://www.javatpoint.com/java-switch) are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

**Student.java**

1. **public** **class** Student **implements** Cloneable {
2. **public** **static** **void** main(String[] args) {
3. **int** num = 2;
4. **switch** (num){
5. **case** 0:
6. System.out.println("number is 0");
7. **break**;
8. **case** 1:
9. System.out.println("number is 1");
10. **break**;
11. **default**:
12. System.out.println(num);
13. }
14. }
15. }

**Output:**

2

### **Loop Statements**

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

Let's understand the loop statements one by one.

### **Java for loop:**

In Java, [for loop](https://www.javatpoint.com/java-for-loop) is similar to [C](https://www.javatpoint.com/c-programming-language-tutorial) and [C++](https://www.javatpoint.com/cpp-tutorial). It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

1. **for**(initialization, condition, increment/decrement) {
2. //block of statements
3. }

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

**Calculation.java**

1. **public** **class** Calculattion {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** sum = 0;
5. **for**(**int** j = 1; j<=10; j++) {
6. sum = sum + j;
7. }
8. System.out.println("The sum of first 10 natural numbers is " + sum);
9. }
10. }

**Output:**

The sum of first 10 natural numbers is 55

**Calculation.java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. String[] names = {"Java","C","C++","Python","JavaScript"};
5. System.out.println("Printing the content of the array names:\n");
6. **for**(String name:names) {
7. System.out.println(name);
8. }
9. }
10. }

**Output:**

Printing the content of the array names:

Java

C

C++

Python

JavaScript

### **Java while loop**

The [while loop](https://www.javatpoint.com/java-while-loop) is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

1. **while**(condition){
2. //looping statements
3. }

The flow chart for the while loop is given in the following image.



Consider the following example.

**Calculation .java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **while**(i<=10) {
7. System.out.println(i);
8. i = i + 2;
9. }
10. }
11. }

**Output:**

Printing the list of first 10 even numbers

0

2

4

6

8

10

### **Java do-while loop**

The [do-while loop](https://www.javatpoint.com/java-do-while-loop) checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

**Calculation.java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **do** {
7. System.out.println(i);
8. i = i + 2;
9. }**while**(i<=10);
10. }
11. }

**Output:**

Printing the list of first 10 even numbers

0

2

4

6

8

10

### **Jump Statements**

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

### **Java break statement**

As the name suggests, the [break statement](https://www.javatpoint.com/java-break) is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

**The break statement example with for loop**

Consider the following example in which we have used the break statement with the for loop.

**BreakExample.java**

1. **public** **class** BreakExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
5. **for**(**int** i = 0; i<= 10; i++) {
6. System.out.println(i);
7. **if**(i==6) {
8. **break**;
9. }
10. }
11. }
12. }

**Output:**

0

1

2

3

4

5

6

### **Java continue statement**

Unlike break statement, the [continue statement](https://www.javatpoint.com/java-continue) doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

1. **public** **class** ContinueExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
6. **for**(**int** i = 0; i<= 2; i++) {
8. **for** (**int** j = i; j<=5; j++) {
10. **if**(j == 4) {
11. **continue**;
12. }
13. System.out.println(j);
14. }
15. }
16. }
18. }

**Output:**

0

1

2

3

5

1

2

3

5

2

3

5

# Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](https://www.javatpoint.com/java-tutorial) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   * if statements
   * switch statement
2. Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop
3. Jump statements
   * break statement
   * continue statement

### **Decision-Making statements:**

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

### **1) If Statement:**

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

### **1) Simple if statement:**

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }

Consider the following example in which we have used the **if** statement in the java code.

Student.java

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y > 20) {
6. System.out.println("x + y is greater than 20");
7. }
8. }
9. }

**Output:**

*x + y is greater than 20*

### **2) if-else statement**

The [if-else statement](https://www.javatpoint.com/java-if-else) is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

**Syntax:**

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }
4. **else**{
5. statement 2; //executes when condition is false
6. }

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y < 10) {
6. System.out.println("x + y is less than      10");
7. }   **else** {
8. System.out.println("x + y is greater than 20");
9. }
10. }
11. }

**Output:**

*x + y is greater than 20*

### **3) if-else-if ladder:**

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

1. **if**(condition 1) {
2. statement 1; //executes when condition 1 is true
3. }
4. **else** **if**(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. **else** {
8. statement 2; //executes when all the conditions are false
9. }

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. String city = "Delhi";
4. **if**(city == "Meerut") {
5. System.out.println("city is meerut");
6. }**else** **if** (city == "Noida") {
7. System.out.println("city is noida");
8. }**else** **if**(city == "Agra") {
9. System.out.println("city is agra");
10. }**else** {
11. System.out.println(city);
12. }
13. }
14. }

**Output:**

*Delhi*

### **4. Nested if-statement**

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

1. **if**(condition 1) {
2. statement 1; //executes when condition 1 is true
3. **if**(condition 2) {
4. statement 2; //executes when condition 2 is true
5. }
6. **else**{
7. statement 2; //executes when condition 2 is false
8. }
9. }

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. String address = "Delhi, India";
5. **if**(address.endsWith("India")) {
6. **if**(address.contains("Meerut")) {
7. System.out.println("Your city is Meerut");
8. }**else** **if**(address.contains("Noida")) {
9. System.out.println("Your city is Noida");
10. }**else** {
11. System.out.println(address.split(",")[0]);
12. }
13. }**else** {
14. System.out.println("You are not living in India");
15. }
16. }
17. }

**Output:**

*Delhi*

### **Switch Statement:**

In Java, [Switch statements](https://www.javatpoint.com/java-switch) are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

* The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
* Cases cannot be duplicate
* Default statement is executed when any of the case doesn't match the value of expression. It is optional.
* Break statement terminates the switch block when the condition is satisfied.  
  It is optional, if not used, next case is executed.
* While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

1. **switch** (expression){
2. **case** value1:
3. statement1;
4. **break**;
5. .
6. .
7. .
8. **case** valueN:
9. statementN;
10. **break**;
11. **default**:
12. **default** statement;
13. }

Consider the following example to understand the flow of the switch statement.

**Student.java**

1. **public** **class** Student **implements** Cloneable {
2. **public** **static** **void** main(String[] args) {
3. **int** num = 2;
4. **switch** (num){
5. **case** 0:
6. System.out.println("number is 0");
7. **break**;
8. **case** 1:
9. System.out.println("number is 1");
10. **break**;
11. **default**:
12. System.out.println(num);
13. }
14. }
15. }

**Output:**

*2*

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

### **Loop Statements**

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

Let's understand the loop statements one by one.

### **Java for loop**

In Java, [for loop](https://www.javatpoint.com/java-for-loop) is similar to [C](https://www.javatpoint.com/c-programming-language-tutorial) and [C++](https://www.javatpoint.com/cpp-tutorial). It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

1. **for**(initialization, condition, increment/decrement) {
2. //block of statements
3. }

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

**Calculation.java**

1. **public** **class** Calculattion {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** sum = 0;
5. **for**(**int** j = 1; j<=10; j++) {
6. sum = sum + j;
7. }
8. System.out.println("The sum of first 10 natural numbers is " + sum);
9. }
10. }

**Output:**

*The sum of first 10 natural numbers is 55*

### **Java for-each loop**

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

1. **for**(data\_type var : array\_name/collection\_name){
2. //statements
3. }

Consider the following example to understand the functioning of the for-each loop in Java.

**Calculation.java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. String[] names = {"Java","C","C++","Python","JavaScript"};
5. System.out.println("Printing the content of the array names:\n");
6. **for**(String name:names) {
7. System.out.println(name);
8. }
9. }
10. }

**Output:**

*Printing the content of the array names:*

*Java*

*C*

*C++*

*Python*

*JavaScript*

### **Java while loop**

The [while loop](https://www.javatpoint.com/java-while-loop) is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

1. **while**(condition){
2. //looping statements
3. }

The flow chart for the while loop is given in the following image.



Consider the following example.

**Calculation .java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **while**(i<=10) {
7. System.out.println(i);
8. i = i + 2;
9. }
10. }
11. }

**Output:**

*Printing the list of first 10 even numbers*

*0*

*2*

*4*

*6*

*8*

*10*

### **Java do-while loop**

The [do-while loop](https://www.javatpoint.com/java-do-while-loop) checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

1. **do**
2. {
3. //statements
4. } **while** (condition);

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

**Calculation.java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **do** {
7. System.out.println(i);
8. i = i + 2;
9. }**while**(i<=10);
10. }
11. }

**Output:**

*Printing the list of first 10 even numbers*

*0*

*2*

*4*

*6*

*8*

*10*

### **Jump Statements**

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

### **Java break statement**

As the name suggests, the [break statement](https://www.javatpoint.com/java-break) is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

**The break statement example with for loop**

Consider the following example in which we have used the break statement with the for loop.

**BreakExample.java**

1. **public** **class** BreakExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
5. **for**(**int** i = 0; i<= 10; i++) {
6. System.out.println(i);
7. **if**(i==6) {
8. **break**;
9. }
10. }
11. }
12. }

**Output:**

*0*

*1*

*2*

*3*

*4*

*5*

*6*

**break statement example with labeled for loop**

**Calculation.java**

1. **public** **class** Calculation {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
5. a:
6. **for**(**int** i = 0; i<= 10; i++) {
7. b:
8. **for**(**int** j = 0; j<=15;j++) {
9. c:
10. **for** (**int** k = 0; k<=20; k++) {
11. System.out.println(k);
12. **if**(k==5) {
13. **break** a;
14. }
15. }
16. }
18. }
19. }

22. }

**Output:**

*0*

*1*

*2*

*3*

*4*

*5*

### **Java continue statement**

Advertisement

Unlike break statement, the [continue statement](https://www.javatpoint.com/java-continue) doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

1. **public** **class** ContinueExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
6. **for**(**int** i = 0; i<= 2; i++) {
8. **for** (**int** j = i; j<=5; j++) {
10. **if**(j == 4) {
11. **continue**;
12. }
13. System.out.println(j);
14. }
15. }
16. }
18. }

**Output:**

*0*

*1*

*2*

*3*

*5*

*1*

*2*

*3*

*5*

*2*

*3*

*5*

## **Java Control Statement MCQ**

**1. What is the key difference between the while loop and the do-while loop in Java**

1. The while loop executes its body at least once, while the do-while loop may not execute its body at all.
2. The while loop always checks the loop condition at the end of the loop body, while the do-while loop checks it at the beginning.
3. The do-while loop is suitable for scenarios where the loop body must execute a fixed number of times, while the while loop is more flexible.
4. There is no difference between the while loop and the do-while loop.

Show Answer Workspace

**2. When is the continue statement commonly used in Java?**

1. To exit the loop completely
2. To skip the rest of the loop body and continue with the next iteration
3. To restart the loop from the beginning
4. To break out of nested loops

Show Answer Workspace

**3. What happens if the break statement is used inside a nested loop in Java?**

1. It exits only the innermost loop and continues with the outer loop.
2. It exits all the nested loops and continues with the code after the outermost loop.
3. It causes a compilation error.
4. It exits only the outer loop and continues with the inner loop.

Show Answer Workspace

**4. What is the significance of the default case in a switch statement?**

1. It is executed when none of the other cases match the value of the expression.
2. It is executed before any other case in the switch statement.
3. It is optional and not required in a switch statement.
4. It is executed if the value of the expression is null.

Show Answer Workspace

**5. In a for loop in Java, which of the following components is optional?**

1. Initialization
2. Condition
3. Increment/Decrement
4. All components are mandatory

Show Answer Workspace

# Java if-else Statement

In the realm of programming, decision-making plays a pivotal role in determining the flow of execution. Whether it's directing traffic in a complex intersection or navigating through various scenarios in a program, making decisions is essential. In Java, one of the fundamental constructs for decision-making is the 'if-else' statement. Let's delve into what 'if-else' statements are, how they work, and how they can be effectively utilized in Java programming.

The Java if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in Java.

* if statement
* if-else statement
* if-else-if ladder
* nested if statement

## **How Does It Work?**

When the 'if-else' statement is encountered, the condition within the parentheses is evaluated. If the condition evaluates to true, the block of code inside the 'if' block is executed. However, if the condition evaluates to false, the block of code inside the 'else' block is executed. It allows the program to take different paths based on the outcome of the condition.

## **Java if Statement**

The Java if statement tests the condition. It executes the if block if condition is true.

**Syntax:**

1. **if**(condition){
2. //code to be executed
3. }



**Example:**

**File Name: IfExample.java**

1. //Java Program to demonstate the use of if statement.
2. **public** **class** IfExample {
3. **public** **static** **void** main(String[] args) {
4. //defining an 'age' variable
5. **int** age=20;
6. //checking the age
7. **if**(age>18){
8. System.out.print("Age is greater than 18");
9. }
10. }
11. }

[Test it Now](javascript:void(0))

**Output:**

*Age is greater than 18*

## **Java if-else Statement**

The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

**Syntax:**

1. **if**(condition){
2. //code if condition is true
3. }**else**{
4. //code if condition is false
5. }



**Example:**

**File Name: IfElseExample.java**

1. //A Java Program to demonstrate the use of if-else statement.
2. //It is a program of odd and even number.
3. **public** **class** IfElseExample {
4. **public** **static** **void** main(String[] args) {
5. //defining a variable
6. **int** number=13;
7. //Check if the number is divisible by 2 or not
8. **if**(number%2==0){
9. System.out.println("even number");
10. }**else**{
11. System.out.println("odd number");
12. }
13. }
14. }

[Test it Now](javascript:void(0))

**Output:**

*odd number*

**Leap Year Example Using IfElse:**

A year is leap, if it is divisible by 4 and 400. But, not by 100.

**File Name: LeapYearExample.java**

1. **public** **class** LeapYearExample {
2. **public** **static** **void** main(String[] args) {
3. **int** year=2020;
4. **if**(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
5. System.out.println("LEAP YEAR");
6. }
7. **else**{
8. System.out.println("COMMON YEAR");
9. }
10. }
11. }

**Output:**

*LEAP YEAR*

## **Java if-else-if ladder Statement**

The if-else-if ladder statement executes one condition from multiple statements.

**Syntax:**

1. **if**(condition1){
2. //code to be executed if condition1 is true
3. }**else** **if**(condition2){
4. //code to be executed if condition2 is true
5. }
6. **else** **if**(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. **else**{
11. //code to be executed if all the conditions are false
12. }



**Example:**

**File Name: IfElseIfExample.java**

1. //Java Program to demonstrate the use of If else-if ladder.
2. //It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
3. **public** **class** IfElseIfExample {
4. **public** **static** **void** main(String[] args) {
5. **int** marks=65;
7. **if**(marks<50){
8. System.out.println("fail");
9. }
10. **else** **if**(marks>=50 && marks<60){
11. System.out.println("D grade");
12. }
13. **else** **if**(marks>=60 && marks<70){
14. System.out.println("C grade");
15. }
16. **else** **if**(marks>=70 && marks<80){
17. System.out.println("B grade");
18. }
19. **else** **if**(marks>=80 && marks<90){
20. System.out.println("A grade");
21. }**else** **if**(marks>=90 && marks<100){
22. System.out.println("A+ grade");
23. }**else**{
24. System.out.println("Invalid!");
25. }
26. }
27. }

**Output:**

*C grade*

Program to check POSITIVE, NEGATIVE or ZERO using if-else-if:

**File Name: PositiveNegativeExample.java**

1. **public** **class** PositiveNegativeExample {
2. **public** **static** **void** main(String[] args) {
3. **int** number=-13;
4. **if**(number>0){
5. System.out.println("POSITIVE");
6. }**else** **if**(number<0){
7. System.out.println("NEGATIVE");
8. }**else**{
9. System.out.println("ZERO");
10. }
11. }
12. }

**Output:**

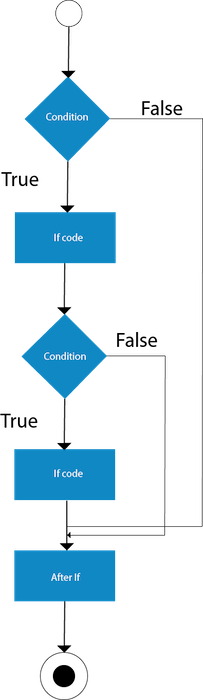
*NEGATIVE*

## **Java Nested if statement**

The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

**Syntax:**

1. **if**(condition){
2. //code to be executed
3. **if**(condition){
4. //code to be executed
5. }
6. }



**Example:**

Advertisement

**File Name: JavaNestedIfExample.java**

1. //Java Program to demonstrate the use of Nested If Statement.
2. **public** **class** JavaNestedIfExample {
3. **public** **static** **void** main(String[] args) {
4. //Creating two variables for age and weight
5. **int** age=20;
6. **int** weight=80;
7. //applying condition on age and weight
8. **if**(age>=18){
9. **if**(weight>50){
10. System.out.println("You are eligible to donate blood");
11. }
12. }
13. }}

[Test it Now](javascript:void(0))

**Output:**

*You are eligible to donate blood*

**Example 2:**

**File Name: JavaNestedIfExample2.java**

1. //Java Program to demonstrate the use of Nested If Statement.
2. **public** **class** JavaNestedIfExample2 {
3. **public** **static** **void** main(String[] args) {
4. //Creating two variables for age and weight
5. **int** age=25;
6. **int** weight=48;
7. //applying condition on age and weight
8. **if**(age>=18){
9. **if**(weight>50){
10. System.out.println("You are eligible to donate blood");
11. } **else**{
12. System.out.println("You are not eligible to donate blood");
13. }
14. } **else**{
15. System.out.println("Age must be greater than 18");
16. }
17. }  }

[Test it Now](javascript:void(0))

**Output:**

*You are not eligible to donate blood*

## **Ternary Operator**

We can also use ternary operator (? :) to perform the task of if...else statement. It is a shorthand way to check the condition. If the condition is true, the result of ? is returned. But, if the condition is false, the result of : is returned.

**Example:**

**File Name: TernaryExample.java**

1. **public** **class** IfElseTernaryExample {
2. **public** **static** **void** main(String[] args) {
3. **int** number=13;
4. //Using ternary operator
5. String output=(number%2==0)?"even number":"odd number";
6. System.out.println(output);
7. }
8. }

**Output:**

*odd number*

## **Conclusion:**

'If-else' statements are indispensable tools in Java programming for making decisions based on conditions. They provide the flexibility to execute different code blocks based on varying circumstances, thus enabling developers to create dynamic and responsive applications. By mastering the usage of 'if-else' statements, programmers can unlock the full potential of Java for crafting efficient and robust software solutions.

# Java Switch Statement

The Java switch statement executes one statement from multiple conditions. It is like [if-else-if](https://www.javatpoint.com/java-if-else) ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, we can use [strings](https://www.javatpoint.com/java-string) in the switch statement.

Let's understand about switch statement in Java in detail.

The switch statement can be described as control flow type statement which is utilized for manipulating the flow of program execution and invoking various branches of code using the value of an expression.

In other words, the switch statement tests the equality of a variable against multiple values.

### **Points to Remember**

* There can be one or N number of case values for a switch expression.
* The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow [variables](https://www.javatpoint.com/java-variables).
* The case values must be unique. In case of duplicate value, it renders compile-time error.
* The Java switch expression must be of byte, short, int, long (with its Wrapper type), *[enums](https://www.javatpoint.com/java-switch)* and string.
* Each case statement can have a break statement which is optional. When control reaches to the [break statement](https://www.javatpoint.com/java-break), it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
* The case value can have a default label which is optional.

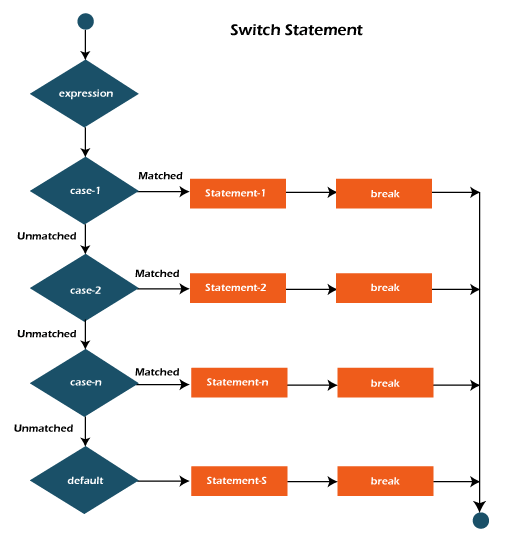
In Java, switch statement mainly provides a more detailed alternative that avoids the usage of nested or several if-else statements when associated with an individual variable.

The syntax of the Java switch statement contains the **switch** keyword which is followed by the expression that needs to be evaluated using parentheses. The mentioned expression must definitely evaluate to a definite data type which is primitive such as int, char, or enum.

**Syntax:**

1. **switch**(expression){
2. **case** value1:
3. //code to be executed;
4. **break**;  //optional
5. **case** value2:
6. //code to be executed;
7. **break**;  //optional
8. ......
10. **default**:
11. code to be executed **if** all cases are not matched;
12. }

**Flowchart of Switch Statement**



In Java, the switch statement can also contain a default label. The default label will be executed only in the situation when none of the case labels are matching the expressions value. The declaring of default label is considered optional, but can be useful in the events of unexpected values or inputs.

**Example:**

**SwitchExample.java**

1. **public** **class** SwitchExample {
2. **public** **static** **void** main(String[] args) {
3. //Declaring a variable for switch expression
4. **int** number=20;
5. //Switch expression
6. **switch**(number){
7. //Case statements
8. **case** 10: System.out.println("10");
9. **break**;
10. **case** 20: System.out.println("20");
11. **break**;
12. **case** 30: System.out.println("30");
13. **break**;
14. //Default case statement
15. **default**:System.out.println("Not in 10, 20 or 30");
16. }
17. }
18. }

[Test it Now](javascript:void(0))

**Output:**

*20*

**Finding Month Example:**

**SwitchMonthExample.javaHTML**

1. //Java Program to demonstrate the example of Switch statement
2. //where we are printing month name for the given number
3. **public** **class** SwitchMonthExample {
4. **public** **static** **void** main(String[] args) {
5. //Specifying month number
6. **int** month=7;
7. String monthString="";
8. //Switch statement
9. **switch**(month){
10. //case statements within the switch block
11. **case** 1: monthString="1 - January";
12. **break**;
13. **case** 2: monthString="2 - February";
14. **break**;
15. **case** 3: monthString="3 - March";
16. **break**;
17. **case** 4: monthString="4 - April";
18. **break**;
19. **case** 5: monthString="5 - May";
20. **break**;
21. **case** 6: monthString="6 - June";
22. **break**;
23. **case** 7: monthString="7 - July";
24. **break**;
25. **case** 8: monthString="8 - August";
26. **break**;
27. **case** 9: monthString="9 - September";
28. **break**;
29. **case** 10: monthString="10 - October";
30. **break**;
31. **case** 11: monthString="11 - November";
32. **break**;
33. **case** 12: monthString="12 - December";
34. **break**;
35. **default**:System.out.println("Invalid Month!");
36. }
37. //Printing month of the given number
38. System.out.println(monthString);
39. }
40. }

[Test it Now](javascript:void(0))

**Output:**

*7 - July*

**Program to check Vowel or Consonant:**

If the character is A, E, I, O, or U, it is vowel otherwise consonant. It is not case-sensitive.

**SwitchVowelExample.java**

1. **public** **class** SwitchVowelExample {
2. **public** **static** **void** main(String[] args) {
3. **char** ch='O';
4. **switch**(ch)
5. {
6. **case** 'a':
7. System.out.println("Vowel");
8. **break**;
9. **case** 'e':
10. System.out.println("Vowel");
11. **break**;
12. **case** 'i':
13. System.out.println("Vowel");
14. **break**;
15. **case** 'o':
16. System.out.println("Vowel");
17. **break**;
18. **case** 'u':
19. System.out.println("Vowel");
20. **break**;
21. **case** 'A':
22. System.out.println("Vowel");
23. **break**;
24. **case** 'E':
25. System.out.println("Vowel");
26. **break**;
27. **case** 'I':
28. System.out.println("Vowel");
29. **break**;
30. **case** 'O':
31. System.out.println("Vowel");
32. **break**;
33. **case** 'U':
34. System.out.println("Vowel");
35. **break**;
36. **default**:
37. System.out.println("Consonant");
38. }
39. }
40. }

**Output:**

*Vowel*

## **Java Switch Statement is fall-through**

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

**Example:**

**SwitchExample2.java**

1. //Java Switch Example where we are omitting the
2. //break statement
3. **public** **class** SwitchExample2 {
4. **public** **static** **void** main(String[] args) {
5. **int** number=20;
6. //switch expression with int value
7. **switch**(number){
8. //switch cases without break statements
9. **case** 10: System.out.println("10");
10. **case** 20: System.out.println("20");
11. **case** 30: System.out.println("30");
12. **default**:System.out.println("Not in 10, 20 or 30");
13. }
14. }
15. }

[Test it Now](javascript:void(0))

**Output:**

*20*

*30*

*Not in 10, 20 or 30*

## **Java Switch Statement with String**

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

**Example:**

**SwitchStringExample.java**

1. //Java Program to demonstrate the use of Java Switch
2. //statement with String
3. **public** **class** SwitchStringExample {
4. **public** **static** **void** main(String[] args) {
5. //Declaring String variable
6. String levelString="Expert";
7. **int** level=0;
8. //Using String in Switch expression
9. **switch**(levelString){
10. //Using String Literal in Switch case
11. **case** "Beginner": level=1;
12. **break**;
13. **case** "Intermediate": level=2;
14. **break**;
15. **case** "Expert": level=3;
16. **break**;
17. **default**: level=0;
18. **break**;
19. }
20. System.out.println("Your Level is: "+level);
21. }
22. }

[Test it Now](javascript:void(0))

**Output:**

*Your Level is: 3*

## **Java Nested Switch Statement**

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

**Example:**

**NestedSwitchExample.java**

1. //Java Program to demonstrate the use of Java Nested Switch
2. **public** **class** NestedSwitchExample {
3. **public** **static** **void** main(String args[])
4. {
5. //C - CSE, E - ECE, M - Mechanical
6. **char** branch = 'C';
7. **int** collegeYear = 4;
8. **switch**( collegeYear )
9. {
10. **case** 1:
11. System.out.println("English, Maths, Science");
12. **break**;
13. **case** 2:
14. **switch**( branch )
15. {
16. **case** 'C':
17. System.out.println("Operating System, Java, Data Structure");
18. **break**;
19. **case** 'E':
20. System.out.println("Micro processors, Logic switching theory");
21. **break**;
22. **case** 'M':
23. System.out.println("Drawing, Manufacturing Machines");
24. **break**;
25. }
26. **break**;
27. **case** 3:
28. **switch**( branch )
29. {
30. **case** 'C':
31. System.out.println("Computer Organization, MultiMedia");
32. **break**;
33. **case** 'E':
34. System.out.println("Fundamentals of Logic Design, Microelectronics");
35. **break**;
36. **case** 'M':
37. System.out.println("Internal Combustion Engines, Mechanical Vibration");
38. **break**;
39. }
40. **break**;
41. **case** 4:
42. **switch**( branch )
43. {
44. **case** 'C':
45. System.out.println("Data Communication and Networks, MultiMedia");
46. **break**;
47. **case** 'E':
48. System.out.println("Embedded System, Image Processing");
49. **break**;
50. **case** 'M':
51. System.out.println("Production Technology, Thermal Engineering");
52. **break**;
53. }
54. **break**;
55. }
56. }
57. }

[Test it Now](javascript:void(0))

**Output:**

*Data Communication and Networks, MultiMedia*

## **Java Wrapper in Switch Statement**

Java allows us to use four [wrapper classes](https://www.javatpoint.com/wrapper-class-in-java): Byte, Short, Integer and Long in switch statement.

**Example:**

**WrapperInSwitchCaseExample.java**

1. //Java Program to demonstrate the use of Wrapper class
2. //in switch statement
3. **public** **class** WrapperInSwitchCaseExample {
4. **public** **static** **void** main(String args[])
5. {
6. Integer age = 18;
7. **switch** (age)
8. {
9. **case** (16):
10. System.out.println("You are under 18.");
11. **break**;
12. **case** (18):
13. System.out.println("You are eligible for vote.");
14. **break**;
15. **case** (65):
16. System.out.println("You are senior citizen.");
17. **break**;
18. **default**:
19. System.out.println("Please give the valid age.");
20. **break**;
21. }
22. }
23. }

[Test it Now](javascript:void(0))

**Output:**

*You are eligible for vote.*

## **Important Points About Java Switch Statement**

* One of the major important features about Java Switch statement is it's fall through behaviour. It means that in case a case label does not contain a break statement, then the execution will be passed on directly to the next case label. A switch statement can also include a default label, which is executed if none of the case labels match the expression's value. The default label is optional but can be useful for handling unexpected or unspecified values.
* Switch statements can only match exact values, and they cannot check ranges of values. This means that if you need to check for a range of values, you would need to use multiple case labels or resort to other control flow statements like if-else.
* Switch statements can only be used to check for equality between the expression and the case labels. They cannot perform more complex checks, such as checking conditions or using comparison operators.
* The expression in a switch statement must evaluate to a primitive data type (int, char, or enum) or to a String (since Java 7). This limitation restricts the types of expressions that can be used with switch statements, making them less flexible in certain situations compared to other control flow statements.
* It's worth noting that starting from Java 12, switch statements have been enhanced to support switch expressions, which allow the switch statement to be used as an expression that returns a value. This feature provides more flexibility and can lead to more concise and readable code in certain situations.
* Overall, the switch statement in Java is a powerful tool for controlling the flow of a program based on the value of an expression, offering a clear and efficient way to handle multiple branching scenarios.
* Switch statements can sometimes lead to code duplication, especially when multiple case blocks perform similar actions. This can make the code harder to maintain and debug.
* While switch statements can be used with String objects since Java 7, they cannot be used directly with other object types. This limitation can make switch statements less useful in scenarios where complex objects need to be compared.

Let us understand about Java Switch statement with the help of a Java example program.

### **Example 1:**

**Filename:**

1. // Java program for demonstrating the features and functionalities of Java Switch statement.
2. **import** java.util.Scanner;
4. **public** **class** JavaSwitch {
5. **public** **static** **void** main(String[] args) {
6. Scanner scanner = **new** Scanner(System.in);
8. System.out.println("Select an option:");
9. System.out.println("1. Print a message");
10. System.out.println("2. Perform a calculation");
11. System.out.println("3. Exit");
13. **int** choice = scanner.nextInt();
15. **switch** (choice) {
16. **case** 1:
17. System.out.println("You selected to print a message.");
18. System.out.println("Enter the message:");
19. scanner.nextLine(); // Consume the newline character
20. String message = scanner.nextLine();
21. System.out.println("Message: " + message);
22. **break**;
23. **case** 2:
24. System.out.println("You selected to perform a calculation.");
25. System.out.println("Enter two numbers:");
26. **int** num1 = scanner.nextInt();
27. **int** num2 = scanner.nextInt();
28. System.out.println("Sum: " + (num1 + num2));
29. **break**;
30. **case** 3:
31. System.out.println("Exiting program.");
32. **break**;
33. **default**:
34. System.out.println("Invalid choice. Please select a valid option.");
35. }
37. scanner.close();
38. }
39. }

**Output:**

*Select an option:*

*1. Print a message*

*2. Perform a calculation*

*3. Exit*

*1*

*You selected to print a message.*

*Enter the message:*

*Hello*

*Message: Hello*

This is all the required explanation about switch statement in Java.

# Java for Loop

**For loops** in Java are a fundamental control structure used to repeat a block of code a specific number of times or iterate through a sequence of values. They are incredibly useful for tasks that require repetition, such as processing items in an array, generating repetitive output, or executing a block of code a predetermined number of times.

The Java for loop is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop.

There are the following three types of for loops in Java.

* Simple for Loop
* [For-each](https://www.javatpoint.com/for-each-loop) or Enhanced for Loop
* Labelled for Loop

## **Java Simple for Loop**

A simple for loop is the same as [C](https://www.javatpoint.com/c-programming-language-tutorial)/[C++](https://www.javatpoint.com/cpp-tutorial). We can initialize the [variable](https://www.javatpoint.com/java-variables), check condition and increment/decrement value. It consists of four parts:

1. **Initialization:** It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition:** It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Increment/Decrement:** It increments or decrements the variable value. It is an optional condition.
4. **Statement:** The statement of the loop is executed each time until the second condition is false.

**Syntax:**

1. **for**(initialization; condition; increment/decrement){
2. //statement or code to be executed
3. }

**Flowchart:**



**Example:**

**Filename:** ForExample.java

1. //Java Program to demonstrate the example of for loop
2. //which prints table of 1
3. **public** **class** ForExample {
4. **public** **static** **void** main(String[] args) {
5. //Code of Java for loop
6. **for**(**int** i=1;i<=10;i++){
7. System.out.println(i);
8. }
9. }
10. }

[Test it Now](javascript:void(0))

**Output:**

*1*

*2*

*3*

*4*

*5*

*6*

*7*

*8*

*9*

*10*

**Filename:** FactorialExample.java

1. // Define a class named FactorialExample
2. **public** **class** FactorialExample {
3. // The main method, which is the entry point of the program
4. **public** **static** **void** main(String[] args) {
5. // Declare an integer variable 'number' and initialize it to 5. This is the number whose factorial is to be calculated.
6. **int** number = 5;
7. // Declare an integer variable 'factorial' and initialize it to 1. This variable will hold the result of the factorial calculation.
8. // Starting with 1 because the factorial of a number n is the product of all positive integers less than or equal to n,
9. // and any number multiplied by 0 is 0, which would not be a correct starting point for multiplication.
10. **int** factorial = 1;
11. // Start a for loop with the loop variable 'i' initialized to 1.
12. // Continue looping as long as 'i' is less than or equal to 'number'.
13. // After each iteration, increment 'i' by 1.
14. **for** (**int** i = 1; i <= number; i++) {
15. // Multiply the current value of 'factorial' by 'i' and store the result back in 'factorial'.
16. // This effectively accumulates the product of numbers from 1 to 'number'.
17. factorial \*= i; // This is equivalent to factorial = factorial \* i;
18. }
19. // After the loop has completed, print the calculated factorial to the console.
20. // This line constructs a string that includes the value of 'number' and the calculated 'factorial',
21. // then prints it to the standard output.
22. System.out.println("Factorial of " + number + " is: " + factorial);
23. }
24. }

**Output:**

*Factorial of 5 is: 120*

## **Java Nested for Loop**

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

The structure allows us to perform complex iterations by running a for loop for each iteration of the outer for loop. Nested for loops are particularly useful when working with multi-dimensional data structures like arrays or when we need to perform operations that require multiple levels of looping, such as generating a matrix or table.

**Syntax:**

1. **for** (initialization; termination; increment) {
2. // Outer loop block
3. **for** (initialization; termination; increment) {
4. // Inner loop block
5. }
6. }

**Example:**

**Filename:** NestedForExample.java

1. **public** **class** NestedForExample {
2. **public** **static** **void** main(String[] args) {
3. //loop of i
4. **for**(**int** i=1;i<=3;i++){
5. //loop of j
6. **for**(**int** j=1;j<=3;j++){
7. System.out.println(i+" "+j);
8. }//end of i
9. }//end of j
10. }
11. }

**Output:**

Advertisement

*1 1*

*1 2*

*1 3*

*2 1*

*2 2*

*2 3*

*3 1*

*3 2*

*3 3*

**Pyramid Example 1:**

**Filename:** PyramidExample.java

1. **public** **class** PyramidExample {
2. **public** **static** **void** main(String[] args) {
3. **for**(**int** i=1;i<=5;i++){
4. **for**(**int** j=1;j<=i;j++){
5. System.out.print("\* ");
6. }
7. System.out.println();//new line
8. }
9. }
10. }

**Output:**

*\**

*\* \**

*\* \* \**

*\* \* \* \**

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

**Pyramid Example 2:**

**Filename:** PyramidExample2.java

1. **public** **class** PyramidExample2 {
2. **public** **static** **void** main(String[] args) {
3. **int** term=6;
4. **for**(**int** i=1;i<=term;i++){
5. **for**(**int** j=term;j>=i;j--){
6. System.out.print("\* ");
7. }
8. System.out.println();//new line
9. }
10. }
11. }

**Output:**

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

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

*\* \* \* \**

*\* \* \**

*\* \**

*\**

## **Java for-each Loop**

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on the basis of elements and not the index. It returns element one by one in the defined variable.

**Syntax:**

1. **for**(data\_type variable : array\_name){
2. //code to be executed
3. }

**Example:**

**Filename:** ForEachExample.java

1. //Java For-each loop example which prints the
2. //elements of the array
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. //Declaring an array
6. **int** arr[]={12,23,44,56,78};
7. //Printing array using for-each loop
8. **for**(**int** i:arr){
9. System.out.println(i);
10. }
11. }
12. }

[Test it Now](javascript:void(0))

**Output:**

*12*

*23*

*44*

*56*

*78*

## **Java Labelled for Loop**

A labelled for loop in Java is a for loop that has been assigned a label. Labels in Java provide a way to identify a block of code, making it possible to break out of or continue an outer loop from within a nested loop. This feature is particularly useful when working with nested loops and we need to control the flow of the outer loop from within an inner loop. Labels enhance the control flow in complex looping structures, allowing for more precise and flexible loop management.

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

#### **Note: The break and continue keywords breaks or continues the innermost for loop respectively.**

**Syntax:**

1. labelname:
2. **for**(initialization; condition; increment/decrement){
3. //code to be executed
4. }

**Example:**

**Filename:** LabeledForExample.java

1. //A Java program to demonstrate the use of labeled for loop
2. **public** **class** LabeledForExample {
3. **public** **static** **void** main(String[] args) {
4. //Using Label for outer and for loop
5. aa:
6. **for**(**int** i=1;i<=3;i++){
7. bb:
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. **break** aa;
11. }
12. System.out.println(i+" "+j);
13. }
14. }
15. }
16. }

**Output:**

*1 1*

*1 2*

*1 3*

*2 1*

If you use **break bb;**, it will break inner loop only which is the default behaviour of any loop.

**Filename:** LabeledForExample2.java

1. **public** **class** LabeledForExample2 {
2. **public** **static** **void** main(String[] args) {
3. aa:
4. **for**(**int** i=1;i<=3;i++){
5. bb:
6. **for**(**int** j=1;j<=3;j++){
7. **if**(i==2&&j==2){
8. **break** bb;
9. }
10. System.out.println(i+" "+j);
11. }
12. }
13. }
14. }

**Output:**

*1 1*

*1 2*

*1 3*

*2 1*

*3 1*

*3 2*

*3 3*

## **Java Infinitive for Loop**

An infinitive for loop in Java is a loop that has no termination condition, or the condition is always true, causing the loop to run indefinitely until the program is manually terminated or interrupted by a break statement or an exception. This type of loop is used when you want to create a continuous loop that keeps running until an external condition or user interaction dictates otherwise.

If we use a pair of semicolons (;;) in the for loop, it will be infinitive for loop.

**Syntax:**

1. **for**(;;){
2. //code to be executed
3. }

**Example:**

**ForExample.java**

1. //Java program to demonstrate the use of infinite for loop
2. //which prints an statement
3. **public** **class** ForExample {
4. **public** **static** **void** main(String[] args) {
5. //Using no condition in for loop
6. **for**(;;){
7. System.out.println("infinitive loop");
8. }
9. }
10. }

**Output:**

*infinitive loop*

*infinitive loop*

*infinitive loop*

*infinitive loop*

*infinitive loop*

Now, we need to press Ctrl+C to exit from the program.

## **Java for Loop vs while Loop vs do-while Loop**

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **for loop** | **while loop** | **do-while loop** |
| Introduction | The Java for loop is a control flow statement that iterates a part of the [programs](https://www.javatpoint.com/java-programs) multiple times. | The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. | The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition. |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. | If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop. |
| Syntax | for(init;condition;incr/decr){ // code to be executed } | while(condition){ //code to be executed } | do{ //code to be executed }while(condition); |
| Example | //for loop for(int i=1;i<=10;i++){ System.out.println(i); } | //while loop int i=1; while(i<=10){ System.out.println(i); i++; } | //do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10); |
| Syntax for infinitive loop | for(;;){ //code to be executed } | while(true){ //code to be executed } | do{ //code to be executed }while(true); |

# Java While Loop

In programming, loops play a pivotal role in iterating over a set of statements repeatedly until a specific condition is met. One such loop in [Java](https://www.javatpoint.com/java-tutorial) is the 'while' loop, known for its simplicity and versatility.

**The Java while loop** is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

**Syntax:**

1. **while** (condition){
2. //code to be executed
3. Increment / decrement statement
4. }

Here, condition is a boolean expression that determines whether the loop should continue iterating or not. The statements within the curly braces are executed repeatedly as long as the condition evaluates to true.

The different parts of do-while loop:

**1. Condition:** It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

**Example:**

1. i <=100

**2. Update Expression:** Every time the loop body is executed, this expression increments or decrements loop variable.

**Example:**

1. i++;

### **Basic Usage**

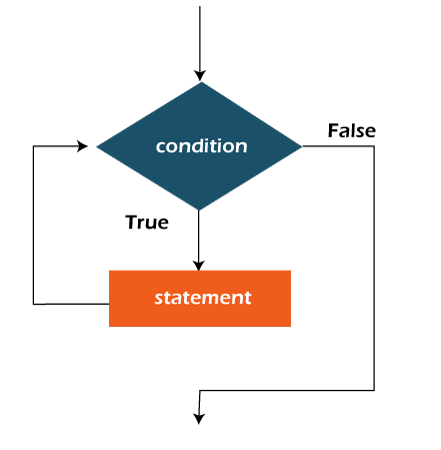
Let's delve into a simple example to grasp the fundamental usage of a while loop. Consider a scenario where we want to print numbers from 1 to 5:

1. **int** i = 1;
2. **while** (i <= 5) {
3. System.out.println(i);
4. i++;
5. }

In this example, the loop starts with i initialized to 1. The condition i <= 5 ensures that the loop continues as long as i is less than or equal to 5. Within each iteration, i is incremented by 1, ensuring that the loop doesn't become infinite.

### **Java While Loop Flowchart**

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



**Example:**

In the following example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

**WhileExample.java**

1. **public** **class** WhileExample {
2. **public** **static** **void** main(String[] args) {
3. **int** i=1;
4. **while**(i<=10){
5. System.out.println(i);
6. i++;
7. }
8. }
9. }

[Test it Now](javascript:void(0))

**Output:**

*1*

*2*

*3*

*4*

*5*

*6*

*7*

*8*

*9*

*10*

### **Java Infinitive while Loop**

While loops can inadvertently become infinite if the condition always evaluates to true. For instance: If we pass true in the while loop, it will be infinitive while loop.

**Syntax:**

1. **while**(**true**){
2. //code to be executed
3. }

This loop will continue indefinitely because the condition true is always true. Therefore, it is crucial to ensure that the condition in a while loop eventually becomes false to exit the loop.

**Example:**

**WhileExample2.java**

1. **public** **class** WhileExample2 {
2. **public** **static** **void** main(String[] args) {
3. // setting the infinite while loop by passing true to the condition
4. **while**(**true**){
5. System.out.println("infinitive while loop");
6. }
7. }
8. }

**Output:**

*infinitive while loop*

*infinitive while loop*

*infinitive while loop*

*infinitive while loop*

*infinitive while loop*

*ctrl+c*

## **Preventing Infinite Loops**

To prevent infinite loops, it is essential to incorporate mechanisms that modify the loop's condition within the loop block. For instance, using a variable to control the loop:

1. **int** count = 0;
2. **while** (count < 5) {
3. System.out.println("Count: " + count);
4. count++;
5. }

In this example, the loop will execute five times, incrementing the count variable with each iteration until it reaches 5, satisfying the condition and ending the loop.

The following example demonstrates how to prevent infinite loops with a while loop:

**PreventingInfiniteLoop.java**

1. **public** **class** PreventingInfiniteLoop {
2. **public** **static** **void** main(String[] args) {
3. **int** count = 0;
4. **while** (count < 5) {
5. System.out.println("Count: " + count);
6. count++;
7. }
8. }
9. }

**Output:**

*Count: 0*

*Count: 1*

*Count: 2*

*Count: 3*

*Count: 4*

The while loop in Java offers a versatile mechanism for executing repetitive tasks based on a condition. Its simplicity and flexibility make it a valuable tool for various programming scenarios, from simple iteration to complex input validation. However, care must be taken to prevent infinite loops by ensuring that the loop's condition eventually evaluates to false. Understanding and mastering the while loop is essential for any Java programmer aiming to write efficient and reliable code.

[**← prev**](https://www.javatpoint.com/java-while-loop)[**next →**](https://www.javatpoint.com/java-break)

# Java do-while Loop

The Java do-while loop is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java do-while loop is executed at least once because condition is checked after loop body.

**Syntax:**

1. **do**{
2. //code to be executed / loop body
3. //update statement
4. }**while** (condition);

**The different parts of do-while loop:**

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

**Example:**

**i <=100**

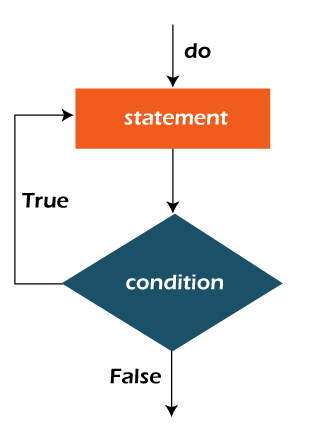
2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

**Example:**

**i++;**

#### **Note: The do block is executed at least once, even if the condition is false.**

**Flowchart of do-while loop:**



**Example:**

In the below example, we print integer values from 1 to 10. Unlike the for loop, we separately need to initialize and increment the variable used in the condition (here, i). Otherwise, the loop will execute infinitely.

**DoWhileExample.java**

1. **public** **class** DoWhileExample {
2. **public** **static** **void** main(String[] args) {
3. **int** i=1;
4. **do**{
5. System.out.println(i);
6. i++;
7. }**while**(i<=10);
8. }
9. }

[Test it Now](javascript:void(0))

**Output:**

*1*

*2*

*3*

*4*

*5*

*6*

*7*

*8*

*9*

*10*

## **Java Infinitive do-while Loop**

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

**Syntax:**

1. **do**{
2. //code to be executed
3. }**while**(**true**);

**Example:**

**DoWhileExample2.java**

1. **public** **class** DoWhileExample2 {
2. **public** **static** **void** main(String[] args) {
3. **do**{
4. System.out.println("infinitive do while loop");
5. }**while**(**true**);
6. }
7. }

**Output:**

*infinitive do while loop*

*infinitive do while loop*

*infinitive do while loop*

*ctrl+c*

**Filename:** DoWhileLoop.java

1. **import** java.util.Scanner;
2. **public** **class** DoWhileLoop {
3. **public** **static** **void** main(String[] args) {
4. // Creating a Scanner object for reading input from the user
5. Scanner sc = **new** Scanner(System.in);
6. // Declare variables to store user input and the s of numbers
7. **int** n;
8. **int** s = 0;
9. **int** c = 0; // Variable to c the n of iterations
10. // Display a welcome message
11. System.out.println("Welcome to the Do-While Loop Program!");
12. // Start a do-while loop
13. **do** {
14. // Prompt the user to enter a n
15. System.out.print("Please enter the value of a and n (or -1 to exit): ");
16. // Read the n entered by the user
17. n = sc.nextInt();
18. // Check if the n is not equal to -1 (exit condition)
19. **if** (n != -1) {
20. // Add the n to the s
21. s += n;
22. // Increment the c of iterations
23. c++;
24. }
25. // Continue looping until the user enters -1
26. } **while** (n != -1);
27. // Display the s of the numbers entered
28. System.out.println("Sum of the numbers entered: " + s);
29. // Display the n of iterations (excluding the exit condition)
30. System.out.println("Number of iterations: " + c);
31. // Calculate and display the average of the numbers entered
32. **double** average = (**double**) s / c;
33. System.out.println("Average of the numbers entered: " + average);
34. // Close the Scanner object to release system resources
35. sc.close();
36. }
37. }

**Output:**

*Welcome to the Do-While Loop Program!*

*Please enter the value of a and n (or -1 to exit): 5*

*Please enter a n (or -1 to exit): -1*

*Sum of the numbers entered: 5*

*Number of iterations: 1*

*Average of the numbers entered: 5.0*

[**← prev**](https://www.javatpoint.com/java-do-while-loop)[**next →**](https://www.javatpoint.com/java-continue)

# Java Break Statement

The `break` statement in Java is a powerful tool for controlling the flow of your program. It is used to exit a loop or switch statement prematurely. It allows us to customize the behavior of code based on specific conditions.

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

The Java break statement is used to break loop or [switch](https://www.javatpoint.com/java-switch) statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

For example, consider a situation if we are searching for a specific value in an array using a for loop. We can use the break statement to exit the loop as soon as we find the value, like this:

1. **int**[] array = {1, 2, 3, 4, 5};
2. **int** targetValue = 3;
3. **for** (**int** i = 0; i < array.length; i++) {
4. **if** (array[i] == targetValue) {
5. System.out.println("Value found at index " + i);
6. **break**;
7. }
8. }

In this example, as soon as the value 3 is found in the array, the break statement is executed, and the loop is terminated. Without the break statement, the loop would continue iterating through the remaining elements of the array that is unnecessary once the value has been found.

Overall, the break statement is a versatile tool that allows us to customize the flow of Java programs, making them more efficient and easier to read. However, it is important to use break judiciously and understand its impact on the flow of code to avoid unintended consequences.

## **Use of break Statement**

We can use Java break statement in all types of loops such as [for loop](https://www.javatpoint.com/java-for-loop), [while loop](https://www.javatpoint.com/java-while-loop) and [do-while loop](https://www.javatpoint.com/java-do-while-loop).

**Syntax:**

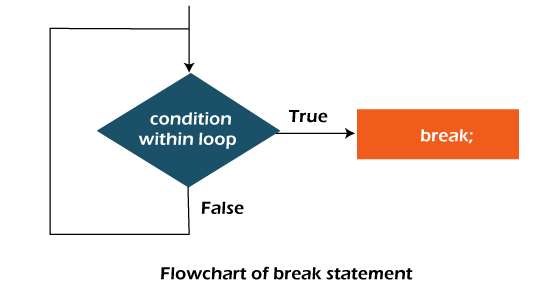
1. jump-statement;
2. **break**;

### **Flowchart of break Statement**

The flowchart of a break statement starts with the condition check of the loop. If the condition is true, the loop body is executed, and then the condition is checked again. If the condition becomes false, the loop is exited normally, and the program continues with the statement after the loop.

However, if a break statement is encountered inside the loop body, the control jumps to the statement immediately after the loop, bypassing the normal exit condition check. This behavior allows us to prematurely exit a loop based on certain conditions, even if the loop's condition is still true.

It is important to note that a break statement only affects the innermost loop that contains it. If we have nested loops and we use a break statement in the inner loop, only that inner loop will be exited, and the outer loops will continue executing normally unless they also have their own break statements.



## **Java Break Statement with Loop**

**Example:**

**BreakExample.java**

1. //Java Program to demonstrate the use of break statement
2. //inside the for loop.
3. **public** **class** BreakExample {
4. **public** **static** **void** main(String[] args) {
5. //using for loop
6. **for**(**int** i=1;i<=10;i++){
7. **if**(i==5){
8. //breaking the loop
9. **break**;
10. }
11. System.out.println(i);
12. }
13. }
14. }

**Output:**

*1*

*2*

*3*

*4*

## **Java Break Statement with Inner Loop**

It breaks inner loop only if you use break statement inside the inner loop.

**Example:**

**BreakExample2.java**

1. //Java Program to illustrate the use of break statement
2. //inside an inner loop
3. **public** **class** BreakExample2 {
4. **public** **static** **void** main(String[] args) {
5. //outer loop
6. **for**(**int** i=1;i<=3;i++){
7. //inner loop
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using break statement inside the inner loop
11. **break**;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

**Output:**

*1 1*

*1 2*

*1 3*

*2 1*

*3 1*

*3 2*

*3 3*

## **Java break Statement with Labelled for Loop**

We can use break statement with a label. The feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer or inner loop.

**Example:**

**BreakExample3.java**

1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to break outer loop
3. **public** **class** BreakExample3 {
4. **public** **static** **void** main(String[] args) {
5. aa:
6. **for**(**int** i=1;i<=3;i++){
7. bb:
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using break statement with label
11. **break** aa;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

**Output:**

*1 1*

*1 2*

*1 3*

*2 1*

## **Java break Statement with while loop**

**Example:**

**BreakWhileExample.java**

1. //Java Program to demonstrate the use of break statement
2. //inside the while loop.
3. **public** **class** BreakWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //while loop
6. **int** i=1;
7. **while**(i<=10){
8. **if**(i==5){
9. //using break statement
10. i++;
11. **break**;//it will break the loop
12. }
13. System.out.println(i);
14. i++;
15. }
16. }
17. }

**Output:**

*1*

*2*

*3*

*4*

## **Java break Statement with do-while loop**

**Example:**

**BreakDoWhileExample.java**

1. //Java Program to demonstrate the use of break statement
2. //inside the Java do-while loop.
3. **public** **class** BreakDoWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //declaring variable
6. **int** i=1;
7. //do-while loop
8. **do**{
9. **if**(i==5){
10. //using break statement
11. i++;
12. **break**;//it will break the loop
13. }
14. System.out.println(i);
15. i++;
16. }**while**(i<=10);
17. }
18. }

**Output:**

*1*

*2*

*3*

*4*

## **Java Break Statement with Switch**

The break statement can be utilized alongside switch statement in Java for successfully exiting

Let's understand about the usage of break statement in Java along with switch keyword in the below mentioned example program.

**Filename:** JavaBreak1.java

1. **import** java.util.Scanner;
2. **public** **class** JavaBreak1 {
3. **public** **static** **void** main(String[] args) {
4. Scanner scanner = **new** Scanner(System.in);
5. System.out.println("Choose an option:");
6. System.out.println("1. Option 1");
7. System.out.println("2. Option 2");
8. System.out.println("3. Option 3");
9. System.out.println("4. Exit");
10. **while** (**true**) {
11. System.out.print("Enter your choice: ");
12. **int** choice = scanner.nextInt();
13. **switch** (choice) {
14. **case** 1:
15. System.out.println("You chose Option 1");
16. **break**; // Exit the switch statement
17. **case** 2:
18. System.out.println("You chose Option 2");
19. **break**; // Exit the switch statement
20. **case** 3:
21. System.out.println("You chose Option 3");
22. **break**; // Exit the switch statement
23. **case** 4:
24. System.out.println("Exiting...");
25. scanner.close();
26. **return**; // Exit the program
27. **default**:
28. System.out.println("Invalid choice. Please try again.");
29. **break**; // Exit the switch statement
30. }
31. }
32. }
33. }

**Output:**

*Choose an option:*

*1. Option 1*

*2. Option 2*

*3. Option 3*

*4. Exit*

*Enter your choice: 1*

*You chose Option 1*

*Enter your choice: 2*

*You chose Option 2*

*Enter your choice: 3*

*You chose Option 3*

*Enter your choice: 4*

*Exiting...*

To understand the example of break with switch statement, go with the link: [Java Switch Statement](https://www.javatpoint.com/java-switch).

## **Uses of Java break Statement**

The break statement in Java is commonly used in various applications to control the flow of a program and manage loop iterations or switch cases. One common application is in searching algorithms, where break can be used to exit a loop once a desired condition is met. For example, in linear search, the loop can be terminated early if the target element is found, saving unnecessary iterations.

Another application is in error handling, where break can be used to exit a loop if an error condition is encountered. It can prevent the program from continuing to execute potentially faulty code and causing further issues. Additionally, break can be used in switch statements to exit the switch block once a specific case is matched, improving the efficiency of the code.

In user input validation, break can be used to exit a loop that prompts the user for input until a valid input is provided. Once the valid input is received, the loop can be exited using break, and the program can continue with the next steps. It helps in ensuring that the program does not proceed with invalid or unexpected input.

Let's understand about the break statement of Java with the help of an example program that demonstrates the usage of break statement in all types of scenarios in a program.

**Filename:** JavaBreak.java

**import** java.util.Scanner;

1. **public** **class** JavaBreak {
2. **public** **static** **void** main(String[] args) {
3. Scanner scanner = **new** Scanner(System.in);
4. **int** number;
5. // Using break in a for loop
6. System.out.println("Using break in a for loop:");
7. **for** (**int** i = 1; i <= 5; i++) {
8. System.out.print("Enter a number (or -1 to exit): ");
9. number = scanner.nextInt();
10. // Check if the user wants to exit the loop
11. **if** (number == -1) {
12. // If the user wants to exit, break out of the loop
13. **break**;
14. }
15. // If the user doesn't want to exit, display the number
16. System.out.println("You entered: " + number);
17. }
18. // Using break in a while loop
19. System.out.println("\nUsing break in a while loop:");
20. **int** i = 1;
21. **while** (i <= 5) {
22. System.out.print("Enter a number (or -1 to exit): ");
23. number = scanner.nextInt();
24. // Check if the user wants to exit the loop
25. **if** (number == -1) {
26. // If the user wants to exit, break out of the loop
27. **break**;
28. }
29. // If the user doesn't want to exit, display the number
30. System.out.println("You entered: " + number);
31. i++;
32. }
33. // Using break in a do-while loop
34. System.out.println("\nUsing break in a do-while loop:");
35. i = 1;
36. **do** {
37. System.out.print("Enter a number (or -1 to exit): ");
38. number = scanner.nextInt();
39. // Check if the user wants to exit the loop
40. **if** (number == -1) {
41. // If the user wants to exit, break out of the loop
42. **break**;
43. }
44. // If the user doesn't want to exit, display the number
45. System.out.println("You entered: " + number);
46. i++;
47. } **while** (i <= 5);
48. // Close the scanner
49. scanner.close();
50. }
51. }

**Output:**

*Using break in a for loop:*

*Enter a number (or -1 to exit): 6*

*You entered: 6*

*Enter a number (or -1 to exit): 5*

*You entered: 5*

*Enter a number (or -1 to exit): 1*

*You entered: 1*

*Enter a number (or -1 to exit): 2*

*You entered: 2*

*Enter a number (or -1 to exit): 36*

*You entered: 36*

*Using break in a while loop:*

*Enter a number (or -1 to exit): 5*

*You entered: 5*

*Enter a number (or -1 to exit): 2*

*You entered: 2*

*Enter a number (or -1 to exit): 2*

*You entered: 2*

*Enter a number (or -1 to exit): 2*

*You entered: 2*

*Enter a number (or -1 to exit): 2*

*You entered: 2*

*Using break in a do-while loop:*

*Enter a number (or -1 to exit): 2*

*You entered: 2*

*Enter a number (or -1 to exit): 2*

*You entered: 2*

*Enter a number (or -1 to exit): 2*

# Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java continue statement is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

**Syntax:**

1. jump-statement;
2. **continue**;

## **Java Continue Statement Example**

**ContinueExample.java**

1. //Java Program to demonstrate the use of continue statement
2. //inside the for loop.
3. **public** **class** ContinueExample {
4. **public** **static** **void** main(String[] args) {
5. //for loop
6. **for**(**int** i=1;i<=10;i++){
7. **if**(i==5){
8. //using continue statement
9. **continue**;//it will skip the rest statement
10. }
11. System.out.println(i);
12. }
13. }
14. }

[Test it Now](javascript:void(0))

**Output:**

*1*

*2*

*3*

*4*

*6*

*7*

*8*

*9*

*10*

As you can see in the above output, 5 is not printed on the console. It is because the loop is continued when it reaches to 5.

## **Java Continue Statement with Inner Loop**

It continues inner loop only if you use the continue statement inside the inner loop.

**ContinueExample2.java**

1. //Java Program to illustrate the use of continue statement
2. //inside an inner loop
3. **public** **class** ContinueExample2 {
4. **public** **static** **void** main(String[] args) {
5. //outer loop
6. **for**(**int** i=1;i<=3;i++){
7. //inner loop
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using continue statement inside inner loop
11. **continue**;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

**Output:**

*1 1*

*1 2*

*1 3*

*2 1*

*2 3*

*3 1*

*3 2*

*3 3*

## **Java Continue Statement with Labelled For Loop**

We can use continue statement with a label. This feature is introduced since JDK 1.5. So, we can continue any loop in Java now whether it is outer loop or inner.

**Example:**

**ContinueExample3.java**

1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to continue outer loop
3. **public** **class** ContinueExample3 {
4. **public** **static** **void** main(String[] args) {
5. aa:
6. **for**(**int** i=1;i<=3;i++){
7. bb:
8. **for**(**int** j=1;j<=3;j++){
9. **if**(i==2&&j==2){
10. //using continue statement with label
11. **continue** aa;
12. }
13. System.out.println(i+" "+j);
14. }
15. }
16. }
17. }

**Output:**

*1 1*

*1 2*

*1 3*

*2 1*

*3 1*

*3 2*

*3 3*

## **Java Continue Statement in while loop**

**ContinueWhileExample.java**

1. //Java Program to demonstrate the use of continue statement
2. //inside the while loop.
3. **public** **class** ContinueWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //while loop
6. **int** i=1;
7. **while**(i<=10){
8. **if**(i==5){
9. //using continue statement
10. i++;
11. **continue**;//it will skip the rest statement
12. }
13. System.out.println(i);
14. i++;
15. }
16. }
17. }

[Test it Now](javascript:void(0))

**Output:**

*1*

*2*

*3*

*4*

*6*

*7*

*8*

*9*

*10*

## **Java Continue Statement in do-while Loop**

**ContinueDoWhileExample.java**

1. //Java Program to demonstrate the use of continue statement
2. //inside the Java do-while loop.
3. **public** **class** ContinueDoWhileExample {
4. **public** **static** **void** main(String[] args) {
5. //declaring variable
6. **int** i=1;
7. //do-while loop
8. **do**{
9. **if**(i==5){
10. //using continue statement
11. i++;
12. **continue**;//it will skip the rest statement
13. }
14. System.out.println(i);
15. i++;
16. }**while**(i<=10);
17. }
18. }

[Test it Now](javascript:void(0))

**Output:**

*1*

*2*

*3*

*4*

*6*

*7*

*8*

*9*

*10*

# Java Comments

The [Java](https://www.javatpoint.com/java-tutorial) comments are the statements in a program that are not executed by the compiler and interpreter.

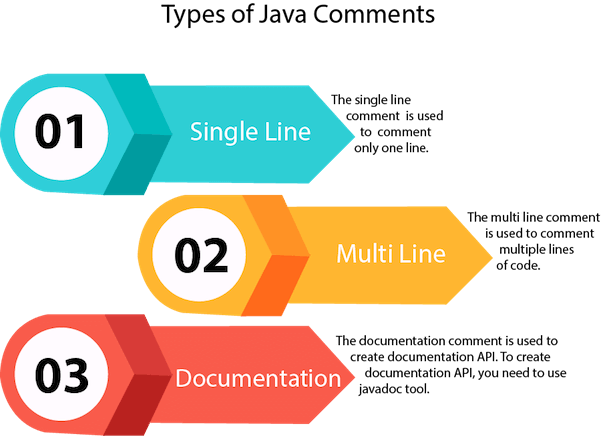
## **Why do we use comments in a code?**

* Comments are used to make the program more readable by adding the details of the code.
* It makes easy to maintain the code and to find the errors easily.
* The comments can be used to provide information or explanation about the [variable](https://www.javatpoint.com/java-variables), method, [class](https://www.javatpoint.com/object-and-class-in-java), or any statement.
* It can also be used to prevent the execution of program code while testing the alternative code.

## **Types of Java Comments**

There are three types of comments in Java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment



### **1) Java Single Line Comment**

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements.

Single line comments starts with two forward slashes **(//)**. Any text in front of // is not executed by Java.

**Syntax:**

1. //This is single line comment

Let's use single line comment in a Java program.

**CommentExample1.java**

1. **public** **class** CommentExample1 {
2. **public** **static** **void** main(String[] args) {
3. **int** i=10; // i is a variable with value 10
4. System.out.println(i);  //printing the variable i
5. }
6. }

**Output:**

*10*

### **2) Java Multi Line Comment**

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there).

Multi-line comments are placed between /\* and \*/. Any text between /\* and \*/ is not executed by Java.

**Syntax:**

1. /\*
2. This
3. is
4. multi line
5. comment
6. \*/

Let's use multi-line comment in a Java program.

**CommentExample2.java**

1. **public** **class** CommentExample2 {
2. **public** **static** **void** main(String[] args) {
3. /\* Let's declare and
4. print variable in java. \*/
5. **int** i=10;
6. System.out.println(i);
7. /\* float j = 5.9;
8. float k = 4.4;
9. System.out.println( j + k ); \*/
10. }
11. }

**Output:**

*10*

#### **Note: Usually // is used for short comments and /\* \*/ is used for longer comments.**

### **3) Java Documentation Comment**

Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API. These APIs are needed for reference, i.e., which classes, methods, arguments, etc., are used in the code.

To create documentation API, we need to use the **[javadoc tool](https://www.javatpoint.com/creating-api-document)**. The documentation comments are placed between /\*\* and \*/.

**Syntax:**

1. /\*\*
2. \*
3. \*We can use various tags to depict the parameter
4. \*or heading or author name
5. \*We can also use HTML tags
6. \*
7. \*/

### **Functions in Java**

**Functions** (also called methods in Java) are reusable blocks of code designed to perform a specific task. They make programs modular and easier to maintain.

#### **Key Points about Functions:**

1. **Declaration:** Functions are defined with a return type, name, and parameters (optional).
2. **Calling:** Functions are called by their name followed by parentheses. If they have parameters, values must be passed in the same order as defined.
3. **Return Type:** Specifies what type of value the function returns. Use void if the function doesn’t return anything.

#### **Example 1: Function to Calculate the Square of a Number**

java

Copy code

public class FunctionExample {

public static void main(String[] args) {

int number = 5;

int result = calculateSquare(number); // Call the function

System.out.println("Square of " + number + " is: " + result);

}

// Function to calculate the square

public static int calculateSquare(int num) {

return num \* num; // Return the square of the number

}

}

#### **Example 2: Function to Greet a User**

java

Copy code

public class GreetingExample {

public static void main(String[] args) {

greetUser("Lynn"); // Call the function with a parameter

}

// Function to greet a user

public static void greetUser(String name) {

System.out.println("Hello, " + name + "! Welcome to Java programming.");

}

}

### **Arrays - Parallel Array Sorting**

**Parallel Array Sorting** is a method introduced in Java 8 to sort arrays using parallelism for better performance. It divides the array into parts, sorts them concurrently, and merges the results.

#### **Advantages:**

* Faster sorting for large arrays.
* Automatically uses multithreading for efficiency.

#### **Example: Sorting an Array of Integers**

java

Copy code

import java.util.Arrays; // Import Arrays utility class

public class ParallelArraySortingExample {

public static void main(String[] args) {

int[] numbers = {42, 15, 8, 23, 4, 16};

System.out.println("Before Sorting: " + Arrays.toString(numbers));

// Sort the array using parallel sort

Arrays.parallelSort(numbers);

System.out.println("After Sorting: " + Arrays.toString(numbers));

}

}

**Explanation:**

1. **Arrays.toString(numbers)** converts the array into a readable string format for display.
2. **Arrays.parallelSort(numbers)** sorts the array in ascending order using parallel sorting.

#### **Example: Sorting an Array of Strings**

java

Copy code

import java.util.Arrays;

public class ParallelStringSorting {

public static void main(String[] args) {

String[] names = {"Lynn", "Zara", "Adam", "Cathy"};

System.out.println("Before Sorting: " + Arrays.toString(names));

// Sort the array of strings

Arrays.parallelSort(names);

System.out.println("After Sorting: " + Arrays.toString(names));

}

}

### **Combining Functions and Parallel Array Sorting**

#### **Example: Function to Sort an Array**

java

Copy code

import java.util.Arrays;

public class SortFunctionExample {

public static void main(String[] args) {

int[] scores = {95, 70, 88, 65, 99};

System.out.println("Original Scores: " + Arrays.toString(scores));

// Call the function to sort the array

sortArray(scores);

System.out.println("Sorted Scores: " + Arrays.toString(scores));

}

// Function to sort an array using parallel sort

public static void sortArray(int[] array) {

Arrays.parallelSort(array); // Sort the array

}

}

**Explanation:**

1. The function sortArray takes an integer array as input.
2. Inside the function, Arrays.parallelSort is used to sort the array.
3. The sorted array is displayed in the main method.

### **Output Examples**

#### **Example 1 (Integer Sorting):**

**Input Array:** {42, 15, 8, 23, 4, 16}  
**Output:**

less

Copy code

Before Sorting: [42, 15, 8, 23, 4, 16]

After Sorting: [4, 8, 15, 16, 23, 42]

#### **Example 2 (String Sorting):**

**Input Array:** {"Lynn", "Zara", "Adam", "Cathy"}  
**Output:**

less

Copy code

Before Sorting: [Lynn, Zara, Adam, Cathy]

After Sorting: [Adam, Cathy, Lynn, Zara]