**Java For Loop**

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 three types of for loop in java.

* Simple For Loop
* For-each or Enhanced For Loop
* Labeled For Loop

**Java Simple For Loop**

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

**Syntax:**

1. for(initialization;condition;incr/decr){
2. //code to be executed
3. }

**Example:**

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

Output:

1

2

3

4

5

6

7

8

9

10

**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 elements basis not index. It returns element one by one in the defined variable.

**Syntax:**

1. for(Type var:array){
2. //code to be executed
3. }

**Example:**

1. public class ForEachExample {
2. public static void main(String[] args) {
3. int arr[]={12,23,44,56,78};
4. for(int i:arr){
5. System.out.println(i);
6. }
7. }
8. }

Output:

12

23

44

56

78

**Java Labeled For Loop**

We can have name of each for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

Normally, break and continue keywords breaks/continues the inner most for loop only.

**Syntax:**

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

**Example:**

1. public class LabeledForExample {
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 aa;
9. }
10. System.out.println(i+" "+j);
11. }
12. }
13. }
14. }

Output:

1 1

1 2

1 3

2 1

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

1. public class LabeledForExample {
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**

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

**Syntax:**

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

**Example:**

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

Output:

infinitive loop

infinitive loop

infinitive loop

infinitive loop

infinitive loop

ctrl+c

# while Loop in java

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

## Syntax

The syntax of a while loop is −

while(Boolean\_expression) {

// Statements

}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

When executing, if the *boolean\_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

## Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

## Example

public class Test {

public static void main(String args[]) {

int x = 10;

while( x < 20 ) {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

# do while loop in java

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

## Syntax

Following is the syntax of a do...while loop −

do {

// Statements

}while(Boolean\_expression);

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

## Flow Diagram



## Example

public class Test {

public static void main(String args[]) {

int x = 10;

do {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}while( x < 20 );

}

}

This will produce the following result −

## Output

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

value of x : 15

value of x : 16

value of x : 17

value of x : 18

value of x : 19

**Need of for statement**

**The for Statement**

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

for (*initialization*; *termination*; *increment*) {

*statement(s)*

}

When using this version of the for statement, keep in mind that:

* The *initialization* expression initializes the loop; it's executed once, as the loop begins.
* When the *termination* expression evaluates to false, the loop terminates.
* The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

# Nested Loop in Java

If a loop exists inside the body of another loop, it's called nested loop. Here's an example of nested for loop.

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

// codes inside the body of outer loop

for (int j = 1; j <=2; ++j) {

// codes inside the body of both outer and inner loop

}

// codes inside the body of outer loop

}

Here, a for loop is inside the body another for loop.

It should be noted that, you can put one type of loop inside the body of another type. For example, you can put a while loop inside the body of a for loop.

## Example 1: Java Nested for Loop

class NestedForLoop {

public static void main(String[] args) {

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

System.out.println("Outer loop iteration " + i);

for (int j = 1; j <=2; ++j) {

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

}

}

}

}

When you run the program, the output will be:

Outer loop iteration 1

i = 1; j = 1

i = 1; j = 2

Outer loop iteration 2

i = 2; j = 1

i = 2; j = 2

Outer loop iteration 3

i = 3; j = 1

i = 3; j = 2

Outer loop iteration 4

i = 4; j = 1

i = 4; j = 2

Outer loop iteration 5

i = 5; j = 1

i = 5; j = 2

Here, the outer loop iterates 5 times. In each iteration of outer loop, the inner loop iterates 2 times.

Let's take another example.

### Example 2: Java Nested Loop

class NestedLoop {

public static void main(String[]args) {

int i = 1;

while (i <= 5) {

System.out.println("Outer loop iteration " + i);

for (int j = 1; j <= 2; ++j) {

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

}

++i;

}

}

}

The output of this program and above program is same.

### Example 3: Program to create a pattern

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

Here is a program to create the above pattern using nested loops.

class Pattern {

public static void main(String[] args) {

int rows = 5;

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

{

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

{

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

}

System.out.println("");

}

}

}

# Types of Loops in Java

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



Java programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/java/java_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/java/java_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [do...while loop](https://www.tutorialspoint.com/java/java_do_while_loop.htm)  Like a while statement, except that it tests the condition at the end of the loop body. |

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Java supports the following control statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/java/java_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [continue statement](https://www.tutorialspoint.com/java/java_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

## Enhanced for loop in Java

As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.

### Syntax

Following is the syntax of enhanced for loop −

for(declaration : expression) {

// Statements

}

* **Declaration** − The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
* **Expression** − This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

### Example

public class Test {

public static void main(String args[]) {

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

for(int x : numbers ) {

System.out.print( x );

System.out.print(",");

}

System.out.print("\n");

String [] names = {"James", "Larry", "Tom", "Lacy"};

for( String name : names ) {

System.out.print( name );

System.out.print(",");

}

}

}

This will produce the following result −

### Output

10, 20, 30, 40, 50,

James, Larry, Tom, Lacy,

# Nested Loop in Java

If a loop exists inside the body of another loop, it's called nested loop. Here's an example of nested for loop.

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

// codes inside the body of outer loop

for (int j = 1; j <=2; ++j) {

// codes inside the body of both outer and inner loop

}

// codes inside the body of outer loop

}

Here, a for loop is inside the body another for loop.

It should be noted that, you can put one type of loop inside the body of another type. For example, you can put a while loop inside the body of a for loop.

## Example 1: Java Nested for Loop

class NestedForLoop {

public static void main(String[] args) {

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

System.out.println("Outer loop iteration " + i);

for (int j = 1; j <=2; ++j) {

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

}

}

}

}

When you run the program, the output will be:

Outer loop iteration 1

i = 1; j = 1

i = 1; j = 2

Outer loop iteration 2

i = 2; j = 1

i = 2; j = 2

Outer loop iteration 3

i = 3; j = 1

i = 3; j = 2

Outer loop iteration 4

i = 4; j = 1

i = 4; j = 2

Outer loop iteration 5

i = 5; j = 1

i = 5; j = 2

Here, the outer loop iterates 5 times. In each iteration of outer loop, the inner loop iterates 2 times.

Let's take another example.

### Example 2: Java Nested Loop

class NestedLoop {

public static void main(String[] args) {

int i = 1;

while (i <= 5) {

System.out.println("Outer loop iteration " + i);

for (int j = 1; j <= 2; ++j) {

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

}

++i;

}

}

}

The output of this program and above program is same.

### Example 3: Program to create a pattern

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

Here is a program to create the above pattern using nested loops.

class Pattern {

public static void main(String[] args) {

int rows = 5;

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

{

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

{

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

}

System.out.println("");

}

}

}

class NestedForLoopExample  
{  
    public static void main(String arg[])  
    {  
        for(int outer = 0; outer <= 2;  outer++)  
        {  
            for(int inner = 0; inner <= 3; inner++)  
            {  
                System.out.println(outer + "   " + inner);  
            }  
            System.out.println();  
        }         
      
    }  
}

# Nested Loops

Nested loops are quite helpful in processing information. We will learn how nested loops are used by taking the example of a pattern problem. An integer value is to be taken as an input from the user and the following pattern is to be printed based on the input. If the input is seven, the following pattern needs to be printed.

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

One thing that is obvious on observing this pattern is that loops are used to print the pattern. This is because, we are not aware in advance of the number of \*'s that are to be printed. Further, we need nested loops: a for loop nested within another for loop. the outer for loop keeps track of the line number. Or, in other words, it is used to count the number of lines we are printing. The inner for loop is used to keep track of the number of \*'s we are printing. Urther, the number of stars on a particular line is equal to the number line number which in some way can be realted to the outer loop control variable. keeping these things in mind, we can write the following code which prints this pattern. The number n is atken as an input from the user:

for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= i; j++) {  
        System.out.print("\*");  
    }  
    System.out.println();  
}

Note that we have used the print() method and not the println() method to display the stars as we want the \*'s to be displayed on the same line. And after the inner loop is executed, we go to the next line using the println() statement.

# Formulating Algorithms

Now that we have learnt about two of the control structures, we shall see how we can formulate algorithms using theses structures. We do have another group of control structures- branching statements, about which we shall see later on. We have two basic approaches, each of which is used for a particular situation. The first is counter controlled repetition and the other is sentinel controlled repetition. Counter control repetition is followed when we know in advance the number of times the loop has to be executed while the sentinel controlled approach is taken when this number is not known before entering the loop. We will deal with the problem of adding numbers to understand these two approaches.

## Counter Controlled Repetition

We have already used controlled repetition when discussing the basics of loops. This example would make the approach more clear. This is the problem: the number of numbers to be added is to be first taken as an input. The user is then prompted for the numbers which will be added by the program. The sum of the numbers entered is then displayed on the screen. This is the program which performs the task:

import java.util.Scanner;  
  
public class AddNumbers

{  
    public static void main(String args[]) {  
        Scanner s = new Scanner(System.in);  
        System.out.print("Enter the number of numbers that you wish to add: ");  
        int count = s.nextInt();  
        int sum = 0;  
        for (int i = 1; i <= count; i++) {  
            System.out.print("Enter number " + i + ": ");  
            sum = sum + s.nextInt();  
        }  
        System.out.println("The sum of the numbers is " + sum);  
    }  
}

Given below is a sample output:

Enter the number of numbers that you wish to add: 4  
Enter number 1: 34  
Enter number 2: 7  
Enter number 3: 4  
Enter number 4: 347  
The sum of the numbers is 392

As you can see in the program above, we have taken the count as an input from the user. The loop is executed these many number of times. A counter, i is used to keep track of the number of numbers that have been added. Each time, the value of i is also displayed along with a prompt for the number. Within the for loop itself, the number taken as input is added to sum.

## Sentinel Controlled Repetition

Sentinel is a value which the user enters to indicate that there is no more input to be given to the program. For example, you can take input from the user as long as he doesn't enter the number 0. In that case,0 is called the sentinel since entering this value causes to the program to stop taking the input. Given below is a program which uses a while loop driven by a sentinel to calculate the sum of numbers entered by the user.

import java.util.Scanner;  
  
public class AddNumbers {  
  
    public static void main(String args[]) {  
        Scanner s = new Scanner(System.in);  
        int sum = 0;  
        int num;  
        do {  
         System.out.print("Enter a number (0 to stop): ");  
         num=s.nextInt();  
         sum=sum+num;  
        }while(num!=0);  
        System.out.println("The sum of the numbers is " + sum);  
    }  
}

And here is a sample output:

Enter a number (0 to stop): 3  
Enter a number (0 to stop): 4  
Enter a number (0 to stop): 7  
Enter a number (0 to stop): 34  
Enter a number (0 to stop): 0  
The sum of the numbers is 48

Note that we have used a do while loop instead of a while loop or a for loop. We could have used them also and modified the program accordingly. In this particular program, we are first adding the number entered by the user to sum and then checking if it is the sentinel. Since the sentinel is 0, it doesn't make any difference if we add the number first and then check or check the number and then add it to sum. However, if the sentinel is another value like -1, then the order does matter. In that case, we can take several approaches. One is to subtract the sentinel from the sum after coming out of the loop and before displaying the sum. But this doesn't look quite good and therefore we present below, a few alternate versions to be used when the sentinel is a value other than 0.

Version 1:

int sum = 0;  
int num = 0;  
do {  
    sum = sum + num;  
    System.out.print("Enter a number (0 to stop): ");  
    num = s.nextInt();  
} while (num != 0);

Version 2:

int sum = 0;  
System.out.print("Enter a number (0 to stop): ");  
int num = s.nextInt();  
while (num != 0) {  
    sum = sum + num;  
    System.out.print("Enter a number (0 to stop): ");  
    num = s.nextInt();  
}

Out of all the above versions, the preferred is the last one. This is because in all other cases, we need to know the identity number which when accumulated with the existing result doesn't change the value of result. In this particular case, where we are adding numbers, that number is 0. If we were to multiply the numbers entered instead of adding them, we would have initialised num to 1. But for an operation whose identity value isn't known, version 2 is the only approach that can be taken.

# Types of Loops in Java

The for loop makes it possible to specify precisely how many times to perform a task in a Java application. Using the break statement makes it possible to stop performing a task when conditions aren’t right, and using the continue statement makes it possible to ignore just one loop (a single value) and then continue processing the next task.

## Using the break statement in Java

There are times when you want to stop a for loop early. For example, you might detect a condition that would cause the loop to fail. Perhaps there aren’t enough items to process, or your application detects some other issue. In this case, you can use the break statement to stop the for loop from doing any more work.

## Using the continue statement in Java

A simple version of the for loop performs a specific number of loops and another version stops at a specific point using a break statement. A third version of the for loop performs a specific number of loops, but it skips some of the loops when conditions don’t warrant performing the task.

For example, the data that you need to process might not be of the right type or might be missing completely. This third form relies on the continue statement. The continue statement tells the for loop to go immediately to the next loop, rather than complete the current loop.

Just because some code examples look at the break and continue statements separately, that doesn’t mean you can’t combine them in a single for loop. The break and continue statements can appear wherever and whenever you need them.

For example, you could choose to process part of a for loop and then continue on to the next loop if things aren’t working out as expected. If an error occurs in the same for loop, you could choose to use a break statement to end it. A single for loop can also contain multiple instances of both the break and continue statements.

The best way to understand the difference between the break and continue statements is to contrast the output they provide.

## Nesting for loops

Sometimes you need to process something using multiple loops. For example, when working with tabular data, you might use one loop to process the rows and another loop to process the columns. There are multiple columns for each row, so the Columns loop appears within the Rows loop. Placing one repeating loop within another is called nesting the loops.

Each iteration of the main loop executes the entire subordinate loop. So, when you start processing the first row, it executes all the column tasks for that row before moving to the next row.

*Nesting* is the process of enclosing one structure within another of the same type. Java uses nesting in a number of ways, so you’ll see this term used quite often. When working with structures, one structure acts as a container to hold the other structure.

The container structure is called the *main*, or *parent*, structure. The structure within the main structure is called the *subordinate*, or *child*,structure.

The multiplication tables are one of the better ways to demonstrate nesting because you need to create a loop for rows and another for columns. In addition, you need to create the headings that show the numbers being multiplied, which means using an additional loop.

# Difference between while loops and FOR loops?

For Loops allow you to run through the loop when you know how many times you'd like it to run through the problem such as for (var i; i < 10; i++); this will continually increase i untill that condition returns false, any number can replace the 10 even a variable. but it will quit once the condition is no longer being met. This is best used again for loops that you know how when they should stop.

While Loops allow you a little more flexability in what you put in it, and when it will stop such as while ( i < 10) you can also substitue in a boolean(true/false) for 10 as well as many other types of varibles.

The key difference between the two is organization between them, if you were going to increase to 10 it'd be a lot cleaner and more readable to use a for statement, but on the other hand if you were to use an existing variable in your program in your loop parameters it'd be cleaner to just wright a while loop. In the For loop you MUST create a new variable, thats not true for the While loop.

# Java – Jump Statements

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

1. break.

2. continue.

3. return.

## 1 The break statement

* This statement is used to jump out of a loop.
* Break statement was previously used in switch – case statements.
* On encountering a break statement within a loop, the execution continues with the  next statement outside the loop.
* The remaining statements which are after the break and within the loop are skipped.
* Break statement can also be used with the label of a statement.
* A statement can be labeled as follows.

statementName : SomeJavaStatement

* When we use break statement along with label as

break statementName;

The execution continues with the statement having the label. This is equivalent to a goto statement of c and c++..

An example of break statement

class break1

{

public static void main(String args[])

{

int i = 1;

while (i<=10)

{

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

i++;

if (i==5)

{

break;

}

}

}

}

**Output :**

1

2

3

4

An example of break to a label

class break3

{

public static void main (String args[])

{

boolean t=true;

a:

{

b:

{

c:

{

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

if(t)

break b;

System.out.println("This will not execute");

}

System.out.println("This will not execute");

}

System.out.println("This is after b");

}

}

}

**Output :**

Before the break

This is after b

## 2 Continue statement

* This statement is used only within looping statements.
* When the continue statement is encountered, the next iteration starts.
* The remaining statements in the loop are skipped. The execution starts from the top of loop again.

The program below shows the use of continue statement.

class continue1

{

public static void main(String args[])

{

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

{

if (i%2 == 0)

continue;

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

}

}

}

**Output :**

1

3

5

7

9

## 3 The return statement

* The last control statement is return. The return statement is used to explicitly return from a method.
* That is, it causes program control to transfer back to the caller of the method.
* the return statement immediately terminates the method in which it is executed.

The program below shows the use of return statement.

class Return1

{

public static void main(String args[])

{

boolean t = true;

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

if(t)

return;       // return to caller

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

}

}

**Output :**

Before the return.

**NOTE :**the if(t) statement is necessary. Without it, the Java compiler would flag an “unreachable code” error, because the compiler would know that the last println( ) statement would never be executed. To prevent this error, the if statement is used here to trick the compiler for the sake of this demonstration.