Data Types in Java

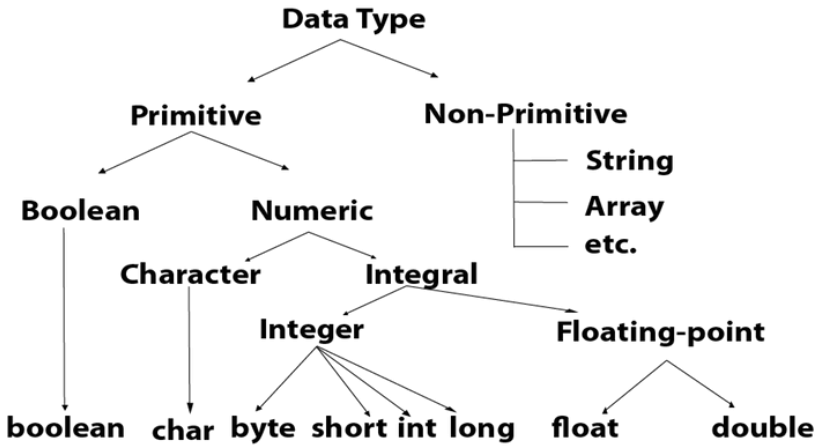
Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

Boolean one = **false**

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 10, **byte** b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

Example:

short s = 10000, **short** r = -5000

Int Data Type
The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is -2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.
The int data type is generally used as a default data type for integral values unless if there is no problem about memory.
Example:

int a = 100000, **int** b = -200000

Long Data Type
The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0.
Example:

long a = 100000L, **long** b = -200000L

Float Data Type
The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.
Example:

float f1 = 234.5f

Double Data Type
The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.
Example:

double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.
Example:

char letterA = 'A'

Literals in Java

Any constant value which can be assigned to the variable is called literal/constant.

In simple words, Literals in Java is a synthetic representation of boolean, numeric, character, or string data.

1. Integral Literals

Integral literals consist of digit sequences and are broken down into these sub-types:

- Decimal Integer: Decimal integers use a base ten and digits ranging from 0 to 9. They can have a negative (-) or a positive (+), but non-digit characters or commas aren't allowed between characters. Example: 2022, +42, -68.
- Octal Integer: Octal integers use a base eight and digits ranging from 0 to 7. Octal integers always begin with a "0." Example: 007, 0295.
- Hexa-Decimal: Hexa-decimal integers work with a base 16 and use digits from 0 to 9 and the characters of A through F. The characters are case-sensitive and represent a 10 to 15 numerical range. Example: 0xf, 0xe.

- Binary Integer: Binary integers uses a base two, consisting of the digits “0” and “1.” The prefix “0b” represents the Binary system. Example: 0b11011.

2. Floating-Point Literals

Floating-point literals are expressed as exponential notations or as decimal fractions. They can represent either a positive or negative value, but if it’s not specified, the value defaults to positive. Floating-point literals come in these formats:

- Floating: Floating format single precision (4 bytes) end with an “f” or “F.” Example: 4f. Floating format double precision (8 bytes) end with a “d” or “D.” Example: 3.14d.
- Decimal: This format uses 0 through 9 and can have either a suffix or an exponent. Example: 99638.440.
- Decimal in Exponent form: The exponent form may use an optional sign, such as a "-," and an exponent indicator, such as "e" or "E." Example: 456.5f.

3. Char Literals

Character (Char) literals are expressed as an escape sequence or a character, enclosed in single quote marks, and always a type of character in Java. Char literals are sixteen-bit Unicode characters ranging from 0 to 65535.

4. String Literals

String literals are sequences of characters enclosed between double quote ("") marks. These characters can be alphanumeric, special characters, blank spaces, etc.
Examples: "John", "2468", "\n", etc.

5. Boolean Literals

Boolean literals have only two values and so are divided into two literals:

- True represents a real boolean value
- False represents a false boolean value

So, Boolean literals represent the logical value of either true or false. These values aren't case-sensitive and are equally valid if rendered in uppercase or lowercase mode. Boolean literals can also use the values of “0” and “1.”
Examples:
boolean b = true;
boolean d = false;

Operators

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.
There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

1. Arithmetic Operators

They are used to perform simple arithmetic operations on primitive data types.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

2. Unary Operators

Unary operators need only one operand. They are used to increment, decrement, or negate a value.

- **– : Unary minus**, used for negating the values.
- **+ : Unary plus** indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
- **++ : Increment operator**, used for incrementing the value by 1. There are two varieties of increment operators.
    - **Post-Increment:** Value is first used for computing the result and then incremented.
    - **Pre-Increment:** Value is incremented first, and then the result is computed.
- **– – : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operators.
    - **Post-decrement:** Value is first used for computing the result and then decremented.
    - **Pre-Decrement: The value** is decremented first, and then the result is computed.
- **! : Logical not operator**, used for inverting a boolean value.

3. Assignment Operator

'=' Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.

The general format of the assignment operator is:

variable **=** value;

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

4. Relational Operators

These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements.

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

5. Logical Operators

These operators are used to perform "logical AND" and "logical OR" operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has "Logical NOT", which returns true when the condition is false and vice-versa

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

6. Bitwise Operators

These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

Operator	Description	Example	Same as	Result	Decimal
&	AND - Sets each bit to 1 if both bits are 1	5 & 1	0101 & 0001	0001	1
|	OR - Sets each bit to 1 if any of the two bits is 1	5 | 1	0101 | 0001	0101	5
~	NOT - Inverts all the bits	~ 5	~0101	1010	10
^	XOR - Sets each bit to 1 if only one of the two bits is 1	5 ^ 1	0101 ^ 0001	0100	4
<<	Zero-fill left shift - Shift left by pushing zeroes in from the right and letting the leftmost bits fall off	9 << 1	1001 << 1	0010	2
>>	Signed right shift - Shift right by pushing copies of the leftmost bit in from the left and letting the rightmost bits fall off	9 >> 1	1001 >> 1	1100	12
>>>	Zero-fill right shift - Shift right by pushing zeroes in from the left and letting the rightmost bits fall off	9 >>> 1	1001 >>> 1	0100	4

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

Structure of Java Program

Structure of Java Program

Java is an object-oriented programming, **platform-independent,** and **secure** programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications.

Documentation Section

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line,** and **documentation** comments.

- **Single-line Comment:** It starts with a pair of forwarding slash **(//)**. For example:

```
1. //First Java Program
```

- **Multi-line Comment:** It starts with a **/*** and ends with ***/.** We write between these two symbols. For example:

```
1. /*It is an example of
2. multiline comment*/
```

- **Documentation Comment:** It starts with the delimiter **(/**)** and ends with ***/**. For example:

```
1. /**It is an example of documentation comment*/
```

Package Declaration

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** statement in a Java program. It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different packages and directories based on the module they are used.

package Examples; //where Examples is the package name

package com.Examples; //where com is the root directory and Examples is the subdirectory

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the import keyword to import the class. It is written before the class declaration and after the package statement. We use the import statement in two ways, either import a specific class or import all classes of a particular package. For example:

import java.util.Scanner; //it imports the Scanner class only

import java.util.*; //it imports all the class of the java.util package

Interface Section

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface is a slightly different from the class. It contains only **constants** and **method** declarations.

```
interface car
{
void start();
void stop();
}
```

Class Definition

In this section, we define the class. It is **vital** part of a Java program. Without the class, we cannot create any Java program. A Java program may conation more than one class definition. We use the **class** keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants.

```
class Student //class definition
{
}
```

Main Method Class

In this section, we define the **main() method.** It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods.

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

Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java.

Creating Arrays

You can create an array by using the new operator with the following syntax –

Syntax

dataType[] arrayRefVar = new dataType[arraySize];

The above statement does two things –

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Alternatively you can create arrays as follows –

dataType[] arrayRefVar = {value0, value1, ..., valuek};

Types of Array in java

There are two types of array.

- Single Dimensional Array

- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Example:-

```
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
OR
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Multidimensional Array

In java, we can create an array with multiple dimensions. We can create 2-dimensional, 3-dimensional, or any dimensional array.

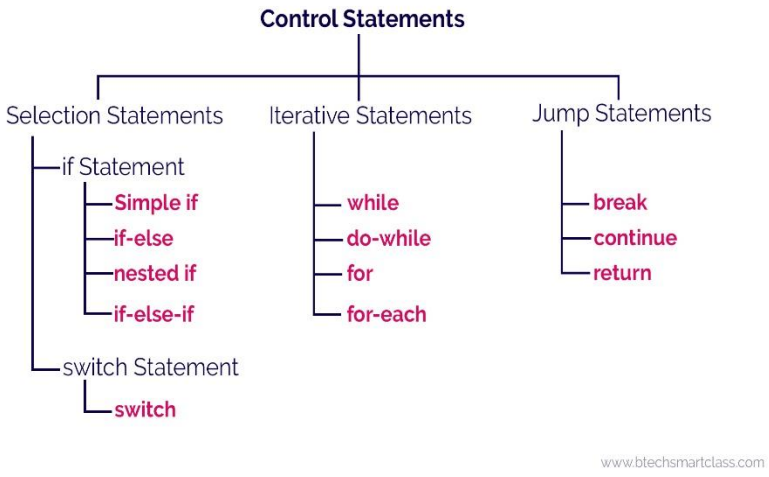
In Java, multidimensional arrays are arrays of arrays. To create a multidimensional array variable, specify each additional index using another set of square brackets. We use the following syntax to create two-dimensional array.

Syntax:

```
data_type array_name[ ][ ] = new data_type[rows][columns];
(or)
data_type[ ][ ] array_name = new data_type[rows][columns];
```

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. It describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.



Decision Making Statement

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

Java's Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

1. **if statement:**

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

Syntax:

```
if(condition)
{
  // Statements to execute if
  // condition is true
}
```

2. **if-else statement**

   An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax

Following is the syntax of an if...else statement –

```
if(Boolean_expression) {
  // Executes when the Boolean expression is true
}else {
  // Executes when the Boolean expression is false
}
```

3. **Nested if statement**

   It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax

The syntax for a nested if...else is as follows –

```
if(Boolean_expression 1) {
  // Executes when the Boolean expression 1 is true
  if(Boolean_expression 2) {
    // Executes when the Boolean expression 2 is true
  }
}
```

4. **The if...else if...else Statement**

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are a few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

Following is the syntax of an if...else statement –

```
if(Boolean_expression 1) {
  // Executes when the Boolean expression 1 is true
```

```
}else if(Boolean_expression 2) {
  // Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3) {
  // Executes when the Boolean expression 3 is true
}else {
  // Executes when the none of the above condition is true.
}
```

5. Switch Statement

   A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The syntax of enhanced for loop is –

```
switch(expression) {
  case value :
    // Statements
    break; // optional

  case value :
    // Statements
    break; // optional

  // You can have any number of case statements.
  default : // Optional
    // Statements
}
```

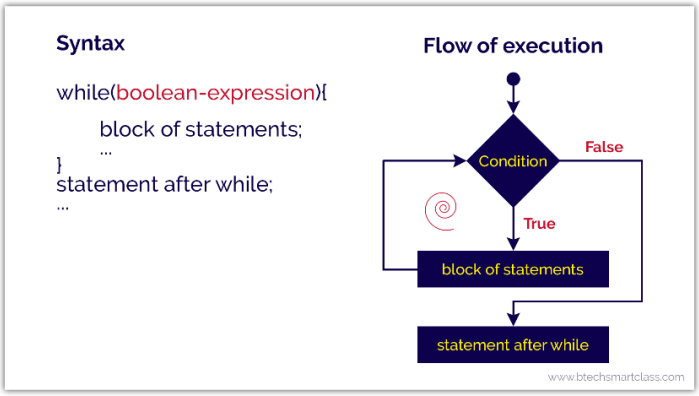
Looping Statement

Java provides a set of looping statements that **executes a block of code repeatedly** while some condition evaluates to true. The iterative statements are also known as looping statements or repetitive statements. Java provides the following iterative statements.

- while statement
- do-while statement
- for statement
- for-each statement

1. while Loop

The while loop statement is the simplest kind of loop statement. It is used to iterate over a single statement or a block of statements until the specified boolean condition is false.

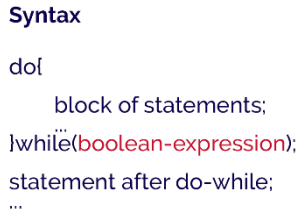


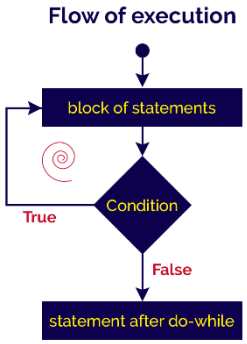
The while loop statement is also called the **entry-control looping statement** because the condition is checked prior to the execution of the statement

and as soon as the boolean condition becomes false, the loop automatically stops.

2. do-while Loop

The Java do-while loop statement works the same as the while loop statement with the only difference being that its boolean condition is evaluated post first execution of the body of the loop. Thus it is also called **exit controlled looping statement**.



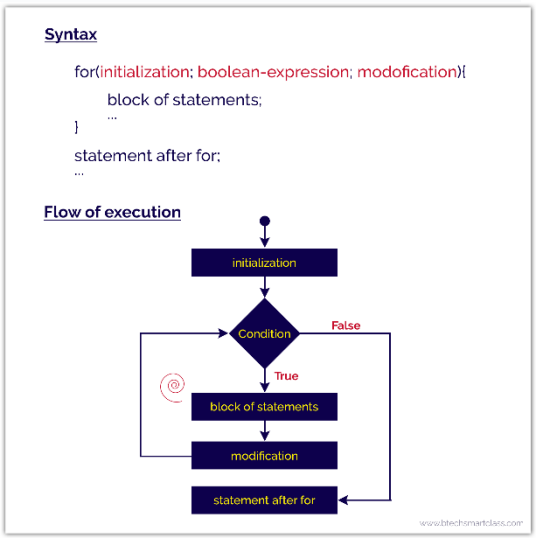


You can use a do-while loop if the number of iterations is not fixed and the body of the loop has to be executed at least once.

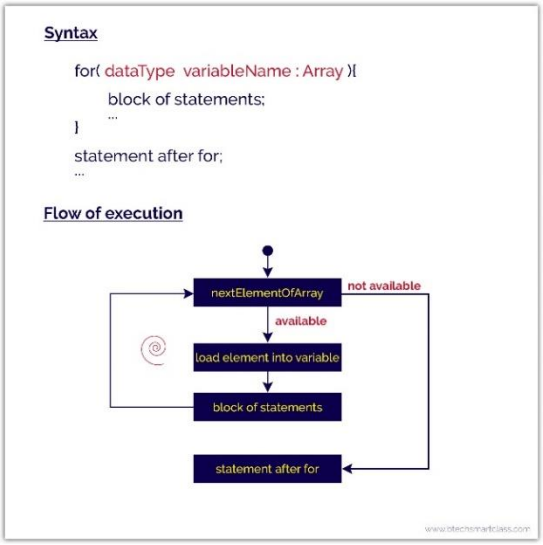
3. for Loop

Unlike the while loop control statements in Java, a for loop statement consists of the initialization of a variable, a condition, and an increment/decrement value, all in one line. It executes the body of the loop until the condition is false. In a for loop statement, execution begins with the initialization of the looping variable, then it executes the condition, and then it increments or decrements the looping variable.

If the condition results in true then the loop body is executed otherwise the for loop statement is terminated.



4. for-each Loop



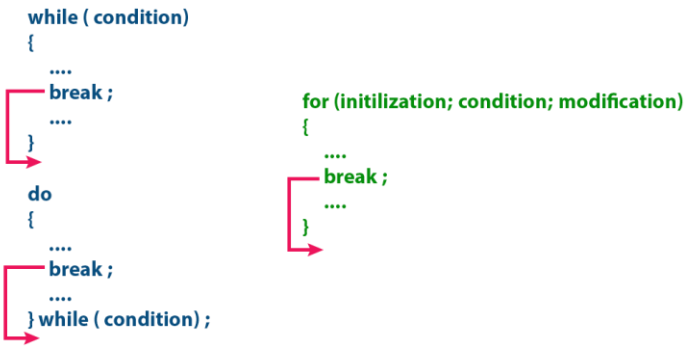
The for-each loop statement provides an approach to traverse through elements of an array or a collection in Java. It executes the body of the loop for each element of the given array or collection. It is also known as the **Enhanced for loop statement** because it is easier to use than the for loop statement as you don't have to handle the increment operation. The major difference between the for and for-each loop is that for loop is a general-purpose loop that we can use for any use case, the for-each loop can only be used with collections or arrays.

In for-each loop statement, you cannot skip any element of the given array or collection. Also, you cannot traverse the elements in reverse order using the for-each loop control statement in Java.

Jump/Branching Statements

Jump statements are used to transfer the control of the program to the specific statements. Jump/Branching control statements in Java transfer the control of the program to other blocks or parts of the program and hence are known as the branch or jump statements.

1. Break Statement

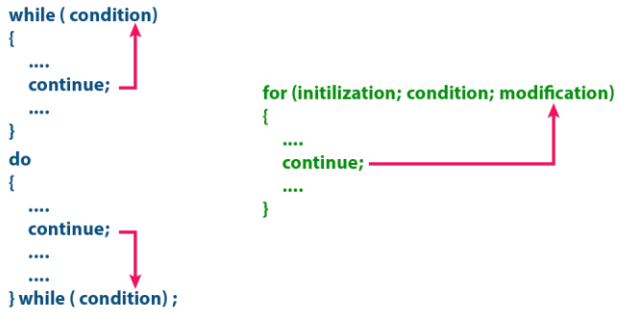


The break statement as we can deduce from the name is used to break the current flow of the program. The break statement is commonly used in the following three situations:

- Terminate a case block in a switch statement as we saw in the example of the switch statement in the above section.
- To exit the loop explicitly, even if the loop condition is true.
- Use as the alternative for the goto statement along with java labels, since java doesn't have goto statements.
-

2. Continue Statement

Sometimes there are situations where we just want to ignore the rest of the code in the loop body and continue from the next iteration. The continue statement in Java allows us to do just that. This is similar to the break statement in the sense that it bypasses every line in the loop body after itself, but instead of exiting the loop, it goes to the next iteration. The continue statement can be used with looping statements like while, do-while, for, and for-each.



3. Return Statement

The return statements are used when we need to return from a method explicitly. The return statement transfers the control back to the caller method of the current method. In the case of the main method, the execution is completed and the program is terminated. Return statements are often used for conditional termination of a method or to return something from the method to the caller method.