INTRODUCTION TO JAVA PROGRAMMING

Basics of Java, The Key Attributes of Object-Oriented Programming, The Java Development Kit, A First Simple Program, Java Magic: Byte Code, Difference between JDK, JVM, and JRE. Java's Primitive Types, Literals, Variables, Type Conversion and Casting, Operators, Shorthand Assignments. Input characters from the Keyword, if statement, Nested ifs, if-else-if Ladder, Switch Statement, Nested switch statements, For Loop, Enhanced For Loop, While Loop, do-while Loop, Use break, Use continue, Nested Loops.

Learning Objective :

- Basics of Java
- JDK
- Java magic : Byte Code
- Difference between JDK, JRE and JVM
- Primitive data types
- Literals and Variables
- Type conversion and Casting
- Different types of Operators
- Shorthand assignment
- Control statements : Loops and if-else ladder
- Break and Continue keywords

Basics of JAVA

Java is a general purpose Programming language. Computer language innovation & development occurs for two fundamental reasons

- ✔ To adapt to changing environments.
- ✔ To implement refinements & improvements in the art of programming.

In 1991, a group of people in Sun Microsystem led by *Patric Naughton & James Gosling*, wanted to design a small computer language for consumer devices such as TV switch boxes. Since these devices require small power & memory, the language had to be small and tight. Also it is important that language not be tied to any single architecture. This project team was named as "Green".

Nikalus Writh, the inventor of Pascal, designed a portable language that generated intermediate language for a hypothetical machine (often called virtual machine). Thus Green

Project used this Virtual machine, which solved their main problem. Finally it took 18 months to develop the first working version. Gosling initially called this language “OAK” but was renamed “JAVA” in 1995.

In 1992, the Green Project delivered its first product called “*7”. It was an intelligent remote control. Unfortunately no one was interested in producing this at Sun. The Green Project with a new name “First Person, Inc” spent all of 1993 and half of 1994 looking for people to buy its technology, but no one was found. Thus First Person was dissolved in 1994.

About the time of Java being worked out at Sun, the World Wide Web part of the internet was growing bigger & bigger. Java was then made use to advance the World Wide Web browser. Sun released the first version of Java in early 1996.

Java as Programming Platform

Java is certainly a good programming language. There is no doubt that it is one of the better languages available to serious programmers. But, Java was never just a language; rather it was a whole Platform, with a huge library, containing lots of reusable code, and an execution environment that provides services such as security, portability and automatic garbage collection.

As a programmer, we want a language with pleasant syntax and comprehensible semantics. Java fits this issue. Java has everything – a good language, a high-quality execution environment, and a vast library.

The Key Attributes of Object Oriented Programming

Object-oriented programming (or OOP for short) is the dominant programming paradigm, having replaced the "structured," procedural programming techniques that were developed in the early 1970s.

Any object oriented programming language like C++, Java etc are designed to achieve,

1. Simplicity in code, and
2. Code Reusability.

In order to make the code simple and reusable, there are four pillars of object oriented programming language,

1. Encapsulation
2. Abstraction
3. Polymorphism
4. Inheritance

Encapsulation: The property of binding the data and its behavior together. It ensures that the data is safe and is not freely available for outside interference and misuse.

Abstraction: The Property of hiding internal implementation from external view. This makes building applications easier.

Polymorphism: The property of providing a single method to take different forms. It means that it is possible to generate a generic interface for a group of related activities.

Inheritance: The Property of acquiring qualities of one class into another class. This is important because it supports the concept of hierarchical classification. By the use of inheritance, a class has to define only those qualities that make it unique. The general qualities can be inherited from the parent class.

Java Development Kit (JDK)

JDK refers to a collection of tools that are used for developing and running Java programs. JDK consists of a collection of tools such as,

1. appletviewer (to launch applets)
2. javac (Java Compiler)
3. java (Java Interpreter)
4. javap (Java Disassembler)
5. javah (For C header files)
6. javadoc (For HTML documents)
7. jdb (Java Debugger)

A source program written in Java is compiled using “javac” and executed using “java”. The “jdb” is used to locate errors if any in the source program.

A First Simple Program

Consider the following simple java program,

```
/*  This is simple Java Program  call this file as “Example.java” */

class Example{
// Program begins with the call to main( )
      public static void main(String [ ] args){
            System.out.println("This is Simple Java Program");
      }
 }
```

Output:

This is Simple Java Program

The code of a java program must be saved in a file with .java as an extension. By convention, the name of the file must be the name of the class that contains the main( ) method. i.e., Example.java in the above example. To compile the java program, execute the compiler, javac, specifying the name of the source file on the command line, i.e., javac Example.java

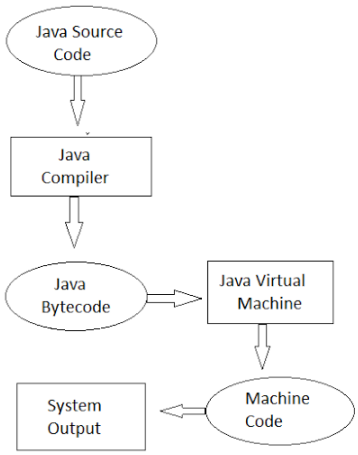
The javac compiler creates a file called Example.class, that contains the bytecode version of the program. To actually run the program, use the java application launcher, called java. To this java, pass class name as a command-line argument, i.e., java Example. When the program is run, the respective output is obtained.

Java Magic: Byte Code

The concept of "Bytecode" is the main reason for Java to achieve portability. When the source program is given as an input to the java compiler, the java compiler does not generate the machine level language executable code. Rather it generates Bytecode.

Bytecodes are a highly optimized set of instructions designed to be executed by the Java runtime system called JVM (Java Virtual Machine).

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only JVM needs to be implemented for each platform. Hence portability of Java programs can be achieved.



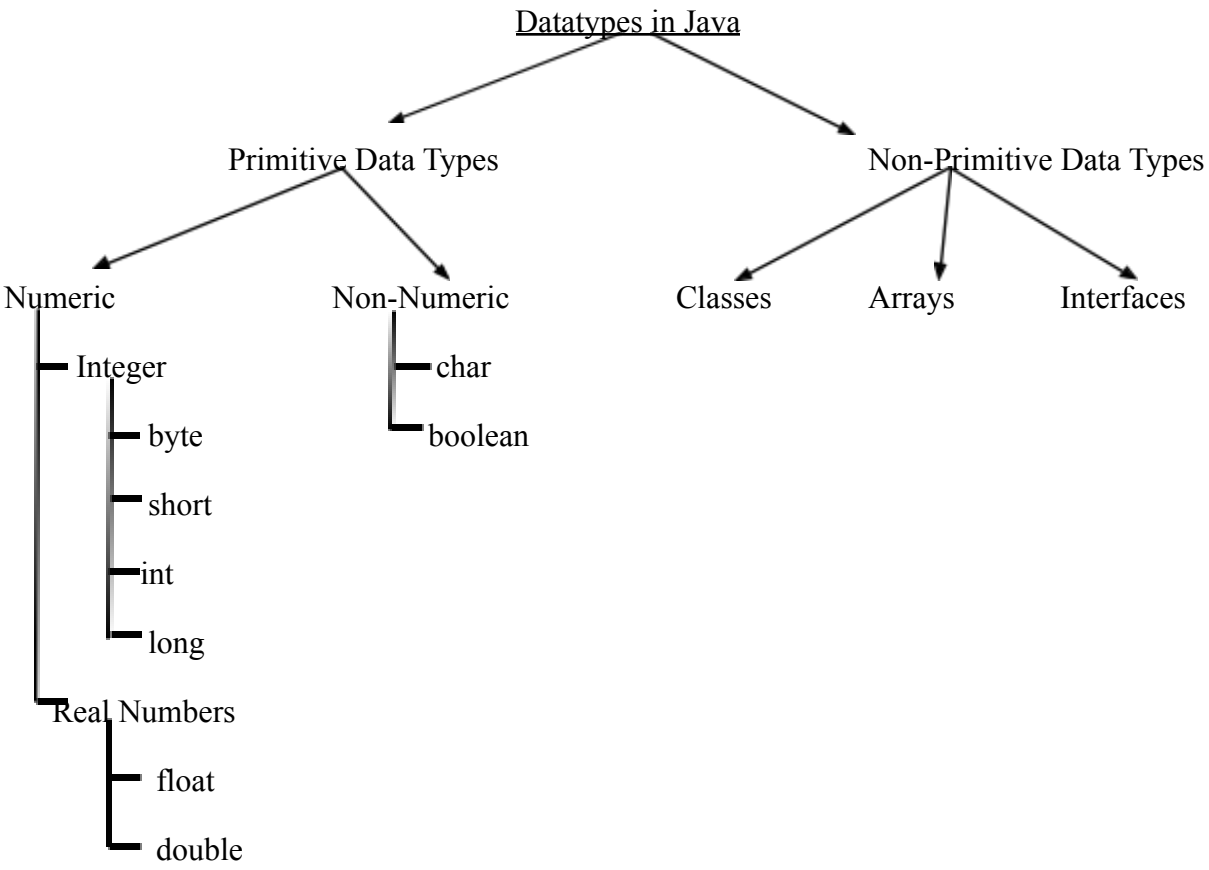
JVM is software that accepts the intermediate bytecode as the input and generates the required machine level language executable code as its output. SunMicrosystems developed different versions of JVM for each environment (OS) and hence achieved portability.

Difference between JDK, JVM, and JRE

JDK- Java Development Kit	JVM – Java Virtual Machine	JRE – Java Runtime Environment
JDK is a tool that facilitates building and running java programs.	JVM is a software to convert byte code to machine code.	JRE is also a tool which can be used to run java programs but can't be used to build the program.
JDK can both compile and run the program.	JVM is specific to the specific version of the Operating System.	JRE can only run the program.

The Primitive Types

Java is a strongly typed language. The various data types supported by Java are as shown below,



Java defines eight primitive types (simple types) of data: byte, short, int, long, char, float, double and boolean.

Integer Types: are signed, positive and negative values. Java does not support unsigned values. Integer types hold the whole numbers, they cannot hold real numbers. There are 4 data types related to Integer groups such as- byte, short, int, long. The size (width) of Integer group is as shown below,

Type	Width in bits
Byte	8
Short	16
Int	32
Long	64

Floating-Point Numbers (Real Types): can hold real numbers, there are 2 datatypes related to floating-point numbers, such as, float and double. 'Float' stores real numbers using single precision, whereas, 'double' stores real numbers using double precision. The size of these data types are shown below,

Type	Width in bits
Float	32
Double	64

Character Types: in Java are used to store characters. It occupies 2bytes in the memory unit and uses Unicode (not ASCII) to represent characters.

Boolean: is used to deal with the logical values. It can have one of the two values namely 'true' or 'false'. This is the type returned by all relational operators.

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown below,

type identifier [= value][, identifier [= value] ...] ;

The *type* is one of Java's primitive types, or the name of a class or interface.

e.g., int a, b, c;
float x = 20.2;

Type Conversion and Type Casting

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(*target-type*) *value*

Here, *target-type* specifies the desired type to convert the specified value to. Consider the following example,

```
1  package example.program;
2
3  class Conversion {
4      public static void main(String args[]) {
5              byte b;
6              int i = 257;
7              double d = 323.142;
8              System.out.println("\nConversion of int to byte.");
9              b = (byte) i;
10             System.out.println("i and b " + i + " " + b);
11             System.out.println("\nConversion of double to int.");
12             i = (int) d;
13             System.out.println("d and i " + d + " " + i);
14             System.out.println("\nConversion of double to byte.");
15             b = (byte) d;
16             System.out.println("d and b " + d + " " + b);
17     }
18 }
```

Output:

```
Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67
```

Operators

The operators are helpful to perform specific operations. There are rich set of operators supported by like such as,

1. Arithmetic Operators
2. Bitwise Operators
3. Relational Operators
4. Logical Operators
5. Assignment Operator
6. Ternary Operator

Arithmetic Operators

Arithmetic operators are used in mathematical expressions. The operands to an arithmetic operator must be of numeric or character type. In other words, boolean type variables cannot be used as operands for arithmetic operators. The following table lists the arithmetic operators,

Operator	Result
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Consider the following example,

```
1  package example.program;
2
3  class BasicMath {
4      public static void main(String args[]) {
5          int a = 1 + 1;
6          int b = a * 3;
7          int c = b / 4;
8          int d = c - a;
9          int e = -d;
10
11         System.out.println("a = " + a);
12         System.out.println("b = " + b);
13         System.out.println("c = " + c);
14         System.out.println("d = " + d);
15         System.out.println("e = " + e);
16     }
17 }
```

Bitwise Operators

Bitwise operators, as the name suggests, operate upon the individual bits of the operand. The operands to the bitwise operators must be of numeric or of character type. The bitwise operators supported by Java is as given below,

Operator	Result
~	Bitwise Unary NOT
&	Bitwise AND
|	Bitwise OR
^	Exclusive OR
>>	Shift Right
>>>	Shift Right Zero Fill
<<	Shift Left

Consider the following example,

```
1  package example.program;
2
3  class BitwiseDemo{
4      public static void main(String [ ] args){
5          int a = 3; int b = 6;
6          int c = a | b;
7          int d = a & b;
8          int e = a ^ b;
9          int f = ~e;
10         System.out.println("c = " + c);
11         System.out.println("d = " + d);
12         System.out.println("e = " + e);
13         System.out.println("f = " + f);
14     }
15 }
```

Relational Operators

Relational operators determine the relationship that one operand has with the other. The outcome of the relational operator is a boolean value. The relational operators are more frequently used in expressions that control the various loop statements and if statements. The relational operators in Java are as listed below,

Operator	Result
==	Equal to
!=	Not Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Consider the following example,

```
1  package example.program;
2
3  class Demo{
4      public static void main(String [ ] args){
5          int a = 4;
6          int b = 1;
7          boolean c = a < b;
8          boolean d = a > b;
9          boolean e = (a == b);
10         boolean f = (a!= b);
11         System.out.println("c = " + c);
12         System.out.println("d = " + d);
13         System.out.println("e = " + e);
14         System.out.println("f = " + f);
15     }
16 }
```

Logical Operators

Logical operators operate upon boolean operands. The list of logical operators supported by Java is as given below,

Operator	Result
!	Logical NOT
&&	Logical AND
||	Logical OR
^	Exclusive OR

The following table shows the effect of each logical operation,

A	B	A&&B	A||B	A^B	!A
False	False	False	False	False	True

True	False	False	True	True	False
False	True	False	True	True	True
True	True	True	True	False	False

Consider the following example,

```
1  package example.program;
2
3  class LogicalDemo{
4      public static void main(String [ ] args){
5          boolean a = true;
6          boolean b = false;
7          boolean c = a | b;
8          boolean d = a & b;
9          boolean e = a ^ b;
10         boolean f = !a;
11         System.out.println("c = " + c);
12         System.out.println("d = " + d);
13         System.out.println("e = " + e);
14         System.out.println("f = " + f);
15     }
16 }
```

Assignment Operator

The general form of the assignment operator is,

var =expression;

e.g., int a = 2*9;

However, the assignment operator allows the programmer to create a chain of assignments,

e.g., int a, b, c;

a = b = c = 9;

Using a chain of assignments is the simplest way to set a group of variables to a common value.

The ? Operator

Java supports a special operator called ternary operator (?), which can be used instead of certain types of if-then-else. The general form of ? operator is,

Expression1 ? Expression2 : Expression3

Here, Expression1 can be any expression that evaluates to a boolean value. If Expression1 is true, then Expression2 is evaluated, otherwise Expression3 is evaluated.

```
1  package example.program;
2
3  class TernaryDemo{
4      public static void main(String [ ] args){
5          int a;
6          a = 28;
7          a=(a>0) ? a+10 : a-10;
8          System.out.println("a= " + a);
9      }
10 }
```

Shorthand Assignment

Most of the binary operators can be combined with assignment operators to make the code simple. These kinds of operators are called shorthand assignment operators.

Operator	Example	Is same as
+=	a+= 10	a = a + 10
-=	a-= 10	a = a - 10
*=	a*= 10	a = a * 10
/=	a/= 10	a = a / 10
%=	a%= 10	a = a % 10
&=	a&= 10	a = a & 10
|=	a|= 10	a = a | 10
^=	a^= 10	a = a ^ 10

```
1  package example.program;
2
3  class ShortHandDemo{
4      public static void main(String [ ] args){
5          int a, b;
6          a = 28;
7          b = 12;
8          a += b;  //same as a=a+b
9          System.out.println("a= " + a);
10     }
11 }
```

Input Character from Keyboard

Java supports various classes to read characters from the keyboard, one such simple and most widely used class for this is Scanner, which is present in the package java.util.

The Scanner class provides methods to read integer, float, string, character etc.

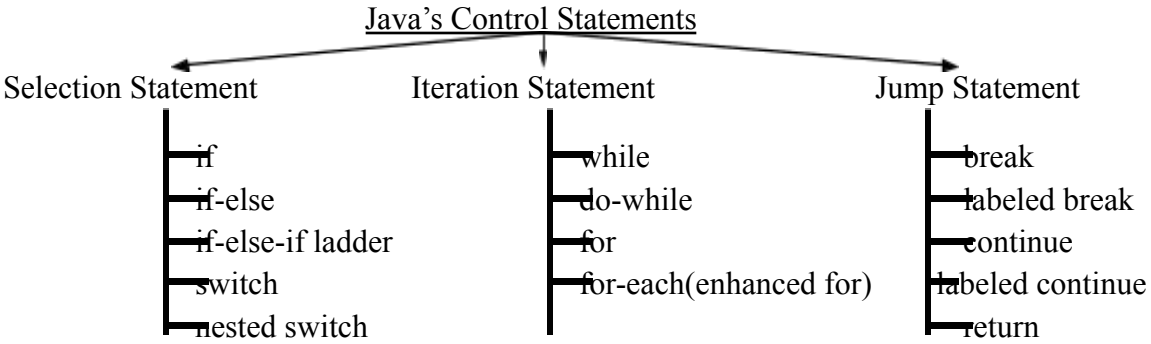
Method	Description
nextInt()	Reads integer
nextLine()	Reads string until new line character
next()	Reads string until space
nextFloat()	Reads floating point value
nextChar()	Reads character value

```
1  package example.program;
2
3  import java.util.Scanner;
4
5  class Demo{
6      public static void main(String [ ] args){
7          int a;
8          Scanner in = new Scanner(System.in);
9          System.out.println("Enter the Number");
10         a = in.nextInt();
11         System.out.println("Value of a = " + a);
12         in.close();
13      }
14  }
```

Introduction to Control Statement

Java supports three types of control statements such as

- ✔ Selection Statement
- ✔ Iteration Statement
- ✔ Jump Statement



Selection Statements

Java supports two selection statements: *if* and *switch*. These statements allow control of the flow of program execution based upon conditions. The *if* statement is Java's conditional branch statement. It is used to route program execution through two different paths. There are different versions of *if-statements*, such as,

- ✔ Simple if-statement
- ✔ if-else
- ✔ if-else-if ladder
- ✔ Nested if-else statement

The general form of *if*-statements are as given below,

Simple if-statement

```
if(condition){
        …..
        …..
 }
```

if-else statement

```
if(condition){
        …..
        …..
  }
else{
        …..
        …..
  }
```

if-else-if ladder

```
if(condition){
        …..
        …..
  }
else if{
```

```
                    .....
                    .....
  }
else if{
                    .....
                    .....
  }
else{
                    .....
                    .....
  }
```

Nested if-else statement

```
if(condition){
        if(condition){
                    .....
                    .....
          }
        else{
                    .....
                    .....
         }
  }
else{
        if(condition){
                    .....
                    .....
          }
        else{
                    .....
                    .....
         }
  }
```

The condition in the above general syntax is any expression that returns a Boolean value or it is a Boolean type by itself. The below program illustrates the working of if-statements,

```java
package example.program;

class ifElse{
    public static void main(String [ ] args){
        int month = 4;
        String season;

        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Summer";
        else if(month == 6 || month == 7 || month == 8)
            season = "Spring";
        else if(month == 9 || month == 10 || month == 11)
            season = "Rainy";
        else
            season = "Bogus Month";

        System.out.println("The season in month " + month + "is: " + season);
    }
}
```

Output:

```
The season in month 4 is: Summer
```

Switch: The switch statement is Java's multiway branch statement. The general form of switch statement is as shown below,

```
switch(expression){
        case value1: // statement sequence
                     break;
        case value2: // statement sequence
                     break;
        case value3: // statement sequence
                     break;
                       .
                       .
                       .
        case valueN: // statement sequence
                     break;
        case value1: // statement sequence
                     break;
        default: // statement sequence
    }
```

The ‘expression’ in switch statement must be of type- byte, short, int or char. Each of the values specified in the case statements must be of a type compatible with the expression. The value of the expression in the switch statement is compared with each of the values in the case statements. If a match is found, the code sequence following that case is executed. If none of the constants matches the value of the expression, then the default statement is executed. The break statement is used inside the switch to terminate a statement sequence.

Consider the following example,

```
1  package example.program;
2
3  class DemoSwitch {
4      public static void main(String [ ] args) {
5          for(int i=0; i<6; i++)
6              switch(i) {
7                  case 0: System.out.println("i is zero.");
8                          break;
9                  case 1: System.out.println("i is one.");
10                         break;
11                 case 2: System.out.println("i is two.");
12                         break;
13                 case 3: System.out.println("i is three.");
14                         break;
15                 default: System.out.println("i is greater than 3.");
16             }
17     }
18 }
```

Output:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Iteration Statements

Java supports 4 types of iteration statements,

→ while loop
→ do-while loop
→ for loop
→ enhanced for loop (for each loop)

while Loop:

The while loop is Java's fundamental loop statement. It repeats a statement or block of statements until the expression is true in a while loop. The general form of while-loop is as shown below,

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

Here 'condition can be any boolean expression. Consider the following example,

```
1  package example.program;
2
3  class While {
4      public static void main(String [ ] args) {
5          int n = 10;
6          while(n > 0) {
7              System.out.println("tick " + n);
8              n--;
9          }
10     }
11 }
```

Output:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.

do-while Loop:

The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

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

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

Consider the following example,

```
1  package example.program;
2
3  class DoWhile {
4      public static void main(String args[]) {
5          int n = 10;
6          do {
7              System.out.println("tick " + n);
8              n--;
9          } while(n > 0);
10     }
11 }
```

Output:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

for Loop:

The general form of for-statement:

```
for(initialization; condition; iteration) {
        // body
 }
```

If only one statement is being repeated, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable,* which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression.

Consider the following example,

```
1 package example.program;
2
3 class ForTick {
4     public static void main(String [ ] args) {
5         int n;
6         for(n=10; n>0; n--)
7             System.out.println("tick " + n);
8     }
9 }
```

Output:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

For-Each Loop (Enhanced for-Loop):

Beginning with JDK 5, a new form of for-loop was defined as 'for-each loop'. The general form of enhanced for is as shown below,

for(type itr-var:collection)
    Statement-block

Here, type specifies the data type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to the end. The collection being cycled through is specified by collection.

Consider the following example,

```
1  package example.program;
2
3  class ForEach{
4      public static void main(String [ ] args){
5          int num [ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
6          int sum = 0;
7
8          for(int x:num){
9              System.out.println("Value is: " + x);
10             sum = sum + x;
11          }
12         System.out.println("Summation: " + sum);
13     }
14 }
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Jump Statement

Java supports three jump statements namely,

→ break
→ continue
→ return.

These statements transfer control to another part of your program.

break Statement:

The break statement has three uses.

- First, it terminates a statement sequence in a switch statement.
- Second, it can be used to exit a loop.
- Third, it can be used as a "civilized" form of goto.

By using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. For example,

```
1  package example.program;
2
3  class BreakLoop {
4      public static void main(String [ ] args) {
5          for(int i=0; i<100; i++) {
6              if(i == 10) break; // terminate loop if i is 10
7              System.out.println("i: " + i);
8          }
9          System.out.println("Loop complete.");
10     }
11 }
```

Output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Using break as a form of 'goto' i.e., labeled break. The general form of the labeled break statement is,

break label;

Here, label is the name of a label that identifies a block of code. When this form of break executes, control is transferred out of the named block of code. The labeled block of code must enclose the break statement, but it does not need to be the immediately enclosing block. This means that you can use a labeled break statement to exit from a set of nested blocks. But you cannot use break to transfer control to a block of code that does not enclose the break statement.

Consider the following example,

```
1  package example.program;
2
3  class Break {
4      public static void main(String [ ] args) {
5          boolean t = true;
6          first: {
7              second: {
8                  third: {
9                         System.out.println("Before the break.");
10                        if(t) break second; // break out of second block
11                        System.out.println("This won't execute");
12                    }
13                 System.out.println("This won't execute");
14                 }
15         System.out.println("This is after second block.");
16          }
17     }
18 }
```

Output:

```
Before the break.
This is after second block.
```

continue Statement:

continue statement skip the remainder of the code in its body for a particular iteration. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Consider the example,

```
1  package example.program;
2
3  class Continue {
4      public static void main(String [ ]args) {
5          for(int i=0; i<10; i++) {
6              System.out.print(i + " ");
7              if (i%2 == 0)
8                  continue;
9              System.out.println(" ");
10         }
11     }
12 }
```

Output:

```
0 1
2 3
4 5
6 7
8 9
```

Similar to a break statement, even a continue statement can be specified with a label. Consider the following example,

```
1  package example.program;
2
3  class ContinueLabel {
4      public static void main(String [ ] args) {
5          outer: for (int i=0; i<10; i++) {
6                  for(int j=0; j<10; j++) {
7                      if(j > i) {
8                          System.out.println();
9                          continue outer;
10                     }
11                     System.out.print(" " + (i * j));
12                 }
13         }
14     System.out.println();
15     }
16 }
```

The continue statement in this example terminates the loop counting j and continues with the next iteration of the loop counting i. Here is the output of this program:

```
0
```

```
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

return Statement:

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.

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

Consider the following example,

```
1  package example.program;
2
3  class Return {
4      public static void main(String args[]) {
5          boolean t = true;
6          System.out.println("Before the return.");
7          if(t) return; // return to caller
8          System.out.println("This won't execute.");
9      }
10 }
```

Output:

```
Before the return.
```