**MODULE-1**

**JAVA Programming Fundamentals**

## WHAT IS JAVA?

* Java is an object-oriented programming language originally developed by Sun Microsystems and released in 1995.
* Java was originally developed by **James Gosling** at **Sun Microsystems** (which has since merge into Oracle Corporation).
* Java programs are platform independent which means they can be run on any operating system with any type of processor as long as the Java interpreter is available on that system.
* Java code that runs on one platform does not need to be recompiled to run on another platform, it’s called “write once, run anywhere” (WORA).
* [Java Virtual Machine (JVM)](http://www.w3schools.in/java-tutorial/java-virtual-machine/) executes Java code, but is written in platform specific languages such as [C](http://www.w3schools.in/c/intro/)/[C++](http://www.w3schools.in/cplusplus/intro/)/ASM etc. JVM is not written in Java and hence **cannot be platform independent** and Java interpreter is actually a part of JVM.

## TYPES OF JAVA APPLICATIONS

1. **Web Application:** Java is used to create server-side web applications. Currently, servlet, jsp, struts, jsf etc. technologies are used.
2. **Standalone Application:** It is also known as desktop application or window-based application. An application that we need to install on every machine or server such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.
3. **Enterprise Application:** An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.
4. **Mobile Application**: Java is used to create application softwares for mobile devices. Currently Java ME is used for creating applications for small devices, and also Java is programming language for Google Android application development.

**JAVA HISTORY(MILESTONES)**

**1990**🡪 Sun Microsystems decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.

**1991🡪** After exploring the possibility of most Object Oriented Programming Language C++, the team announced a new language named “Oak”.

**1992🡪** The team, known as a Green Project team by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch sensitive screen.

**1993🡪** The World Wide Web (WWW) appeared on the internet and transformed the text-based Internet into a Graphical-rich environment. The green Project team came up with the idea of developing Web Applets (tiny Programs) using the new language that could run on all types of computers connected to Internet.

**1994🡪** The team developed a web browser called “Hot Java” to locate and run applet programs on Internet. Hot Java demonstrated the power of the new language, thus making it instantly popular among the Internet users.

**1995🡪** Oak was named “Java”, due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announce to their support to Java.

**1996🡪** Java established itself not only a leader for Internet Programming but also as a general-purpose, object oriented programming language. Java found its home.

The most striking feature of the language is that it is a platform-neutral language. Java is a first programming language that is not tied to any particular hardware or operating system.

### Java Version:

Java SE 18 currently being used

**Key Attributes of Object Oriented Programming:**

* **Simple** − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
* **Object Oriented**- Almost everything in Java is a class, method or an object. Java provides Abstraction, Encapsulation, Inheritance, and Polymorphism.
* **Secure** − With Java's secure feature it enables to develop virus-free, tamper-free systems. Java contains built in security. Java never allows the programmer to memory manipulation (pointer) of the system.
* **Architecture-neutral** – Write once, run anywhere (WORA). With a JVM, you can write one program that will run on any platform.
* **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Java programs can run on any platform without being recompiled.
* **Robust** − Java is a reliable language. It does not have permission to access all of your computer memory, so java program can’t cause a crash. Java programs are less prone to error. It also provides exception and error handling.
* **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
* **Interpreted** – You need an interpreter to run JAVA programs. The programs are compiled into the Java Virtual Machine code called byte code. The byte code is machine independent and can run on any machine that has a Java interpreter, which is a part of JVM.
* **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.
* **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
* **Distributed** − Distributed computing involves several computers working together on a network. Java applications and applets can open and access objects across the web through URL’s as easily as they can access local file system.
* **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Maintaining different versions of application is easy in Java. Java supports dynamic memory allocation with automatic garbage collection.
* **Information Encapsulation (Hiding):-**

Object oriented programming allows you to encapsulate data that you do not want users of the object to access. Typically, attributes of a class are encapsulated.

* **Abstraction:-**

Providing users with only what they need to know is known as abstraction. i.e. Abstraction lets us ignore the irrelevant details and concentrate on the essentials.

* **Inheritance: -** Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the concept of hierarchical classification. In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.
* **Polymorphism: -** Polymorphism means “One Interface, multiple implementations.”

**JAVA DEVELOPMENT KIT(JDK)**

JDK (Java SE Development Kit) includes a complete JRE (Java Runtime Environment) plus tools for developing, debugging, and monitoring Java applications. JDK is needed to develop Java applications and applets as well as run them.

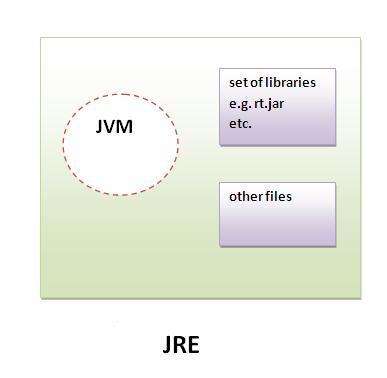
****

Java Development kit contains with the following tools

* **Appletviewer:** Enables us to run java applets(without actually using a java-compatible browser).
* **java:** java interpreter, which runs applets and applications by reading and interpreting bytecode files.
* java: The java compiler, which translates java source code to bytecode files that the interpreter can understand.
* **javadoc :** Creates HTML format documentation from java source code files.
* **javah :** Produces header files for use with native methods
* **javap** : Java disassembler, which enables us to convert bytecode files into a program description.
* **jdb :** Java debugger, which helps us to find errors in our programs.

### JAVA RUNTIME ENVIRONMENT(JRE)

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.



**BLOCK of JAVA CODE:**

Blocks of Code Java allows two or more statements to be grouped into blocks of code, also called code blocks. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can. For example, a block can be a target for Java’s if and for statements.

Consider this if statement:

if(x < y)

{ // begin a block

x = y;

y = 0;

} // end of block

Here, if x is less than y, then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block.



**SIMPLE JAVA PROGRAM**

Create a file with name HelloWorld.java, because the compiler expects the file name to match the class identifier.

The HelloWorld application.

**public class HelloWorld**

**{**

**public static void main(String args[])**

**{**

**System.out.println("Hello World!!");**

**}**

**}**

**Explanation:**

**Class Declaration**

The first line public class HelloWorld declares a class, which is an Object-Oriented construct. As stated earlier Java is true Object-Oriented language and therefore, everything must be placed inside a class. Class is a keyword and declares that a new class definition follows.

**Opening Brace**

Every class definition in Java begins with an opening brace “{“ and ends with a matching closing brace “}”, appearing in the last line in the example.

## public static void main

* The word public means that it is accessible by any other classes.
* The word static means that it is unique.
* The word void means this main method has no return value.
* main is a method where the program starts.

**String args[]** declares a parameter named args, which contains an array of objects of the class type String.

**The Output Line**

The only executable Statement in the program is

**System.out.println("Hello World!!");**

The println method is a member of the out Object, which is static data Member of the System class. This line prints **Hello World!!** to the screen. The method println always appends a newline character to the end of the string.

**KEYWORDS**

Keywords are essential to define language. Specific features of language are implemented using them. There are 50 Keywords in java language. They are,



## IDENTIFIERS

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

In Java, there are several points to remember about identifiers. They are as follows −

* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).
* After the first character, identifiers can have any combination of characters.
* A key word cannot be used as an identifier.
* Most importantly, identifiers are case sensitive.
* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value.
* Examples of illegal identifiers: 123abc, -salary.

**The Java Class Libraries**:

The sample programs shown in this chapter make use of two of Java’s built-in methods: println( ) and print( ). As mentioned, these methods are available through System.out. System is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for a graphical user interface (GUI). Thus, Java as a totality is a combination of the Java language itself, plus its standard classes

**APPLICATION PROGRAMMING INTERFACE (API):**

Java supports several packages which contains hundreds of classes & methods.

**java.lang.\*** - language support package

A collection of classes and methods require for implementing basic features of java.

**java.util.\*** - utilities packages

A collection of classes to provide utility functions such as date & time functions.

**java.io.\*** - Input/Output package

A collection of classes required for input and output manipulations.

**java.net.\*** - networking package

A collection of classes for communicating with other computers via internet.

**java.awt.\*** - AWT(Abstract Window Toolkit) package

This package contains classes that implements platform independent GUI (Graphical User Interface)

**java.applet.\***- Applet package

This includes a set of classes that allow us to create JAVA applets.

## DATA TYPES IN JAVA

1. Primary Data Type (Java supports 8 primitive data types): byte, short, int, long, float, double, char and boolean.
2. Non-Primitive Data Types: string, array etc.

#### Integer Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Contains | Default | Size | Range |
| byte | Signed integer | 0 | 8 bit or 1 byte | -27 to 27-1 or -128 to 127 |
| short | Signed integer | 0 | 16 bit or 2 bytes | -215 to 215-1 or -32768 to 32767 |
| int | Signed integer | 0 | 32 bit or 4 bytes | -231 to 231-1 or -2147483648 to 2147483647 |
| long | Signed integer | 0 | 64 bit or 8 bytes | -263 to 263-1 or -9223372036854775808 to 9223372036854775807 |

#### Rational Numbers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Contains | Default | Size | Range |
| float | IEEE 754 floating point single-precision | 0.0f | 32 bit or 4 bytes | ±1.4E-45 to ±3.40282347E+38F |
| double | IEEE 754 floating point double-precision | 0.0 | 64 bit or 8 bytes | ±439E-324 to ±1.7976931348623157E+308 |

#### Characters

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Contains | Default | Size | Range |
| char | Unicode character unsigned | \u0000 | 16 bits or 2 bytes | 0 to 216-1 or \u0000 to \uFFFF |

#### Conditional

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Contains | Default | Size | Range |
| boolean | true or false | false | 1 bit | true or false |

**LITERALS**

Sequence of characters representing a constant value, to be stored in a variable.

Eg- int a=10; // 10 is an integer literal

boolean b=true; // true is a Boolean literal

**VARIABLE**

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

**. Declaring a Variable In Java**,

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

Here, type is one of Java’s atomic types

The identifier is the name of the variable.

int a, b, c; // declares three ints, a, b, and c.

int d = 3, e, f = 5;

. **Dynamic Initialization**

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

// Demonstrate dynamic initialization.

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

{

double a = 3.0, b = 4.0; // c is dynamically initialized double

c = Math.sqrt(a \* a + b \* b);

System.out.println("Hypotenuse is " + c);

}

} Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem).

**Rules to write Variable/Identifier in Java :**

* They must not begin with digit
* Upper and lowercase are distinct. This means that the variable Total is not the same as total or TOTAL.
* It should not be a keyword.
* White space is not allowed.
* Variable names can be of any length.

There are three kinds of variables in Java −

* 1. **Local variables**
* Local variables are declared in methods, constructors, or blocks.
* Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
  1. **Instance variables**
* Instance variables are declared in a class, but outside a method, constructor or any block.
* Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.

1. **Class/Static variables**

* Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
* There would only be one copy of each class variable per class, regardless of how many objects are created from it.

**The Scope and Lifetime of Variables**

a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

two general categories of scopes: global and local.

In Java, the two major scopes are those defined by a class and those defined by a method. To understand the effect of scopes, consider the following program:

// Demonstrate block scope.

class Scope

{

public static void main(String args[])

{

int x; // known to all code within main x = 10;

if(x == 10)

{ // start new scope

int y = 20; // known only to this block // x and y both known here.

System.out.println("x and y: " + x + " " + y);

x = y \* 2;

} // y = 100; // Error! y not known here // x is still known here.

System.out.println("x is " + x);

}

}

As the comments indicate, the variable x is declared at the start of main( )’s scope and is accessible to all subsequent code within main( ). Within the if block, y is declared. Since a block defines a scope, y is only visible to other code within its block. This is why outside of its block, the line y = 100; is commented out. If you remove the leading comment symbol, a compile-time error will occur, because y is not visible outside of its block. Within the if block, x can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

**OPERATORS**

An operator is a symbol that takes one or more arguments and operates on them to produce a result.

Eg- 2+3, 5\*6 etc..

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups −

* Arithmetic Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Assignment Operators
* Misc Operators

|  |  |
| --- | --- |
| **Operator Precedence** | |
| **Operators** | **Precedence** |
| postfix | *expr*++ *expr*-- |
| unary | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

**Type Conversion and Casting**

a value of one type to a variable of another type.

If the two types are compatible, then Java will perform the conversion automatically.

For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte.

Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

Let’s look at both automatic type conversions and casting.

**Java’s Automatic Conversions** 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. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.

However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

**Casting Incompatible Types**

Although the automatic type conversions are helpful, they will not fulfill all needs.

For example, what if you want to assign an int value to a byte variable

This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type. 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. For example, the following fragment casts an int to a byte. If the integer’s value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte’s range. int a; byte b; // … b = (byte) a

. The following program demonstrates some type conversions that require casts:

// Demonstrate casts.

class Conversion

{

public static void main(String args[])

{

byte b;

int i = 257;

double d = 323.142;

System.out.println("\nConversion of int to byte.");

b = (byte) i;

System.out.println("i and b " + i + " " + b);

System.out.println("\nConversion of double to int.");

i = (int) d;

System.out.println("d and i " + d + " " + i);

System.out.println("\nConversion of double to byte.");

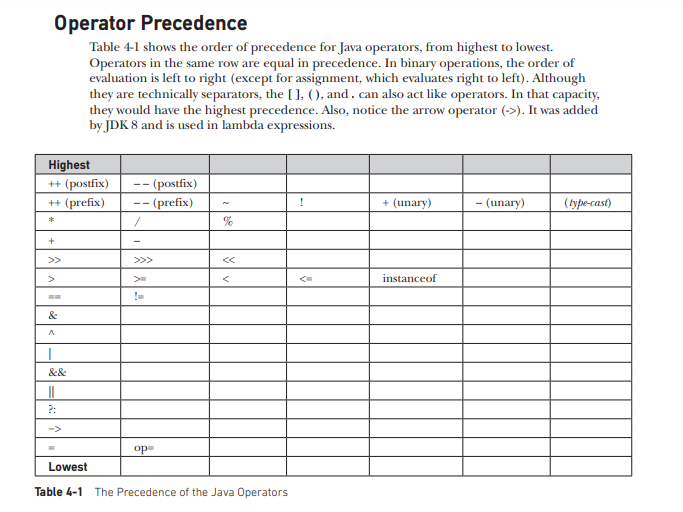
b = (byte) d;

System.out.println("d and b " + d + " " + b);

}

}

This program generates the following 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 Let’s look at each conversion. When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case. When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67

****

**DECISION MAKING STATEMENTS**

Java supports following decision-making statements:

* [if statement](http://www.w3schools.in/java/decision-making/if/)
* [if-else statement](http://www.w3schools.in/java/decision-making/if-else/)
* [else-if statement](http://www.w3schools.in/java/decision-making/else-if/)
* [switch statement](http://www.w3schools.in/java/decision-making/switch/)
  1. **If Statement**

Java if statement is used to execute some code, if a condition is true otherwise if condition is false, execute nothing.

**Syntax:**

if(condition)

{

statement 1;

statement 2;

...

}

**Example:**

public class Sample{

public static void main(String args[]){

int a=20, b=30;

if(b>a)

System.out.println("b is greater");

}

}

**Output:**

b is greater

* 1. **If-else Statement**

Java if else statements is used to execute some code, if a condition is true otherwise if condition is false, execute nothing, or execute some other code.

**Syntax:**

if(condition)

{

//execute your code

}

else

{

//execute your code

}

**Example:**

public class Sample {

public static void main(String args[]) {

int a = 80, b = 30;

if (b > a) {

System.out.println("b is greater");

}

else {

System.out.println("a is greater");

}

}

}

**Output:**

a is greater

* 1. **Else if Statement**

Java elseif is a like doing another if condition for a true or false value.

**Syntax:**

if(condition)

{

//execute your code

}

else if(condition n)

{

//execute your code

}

else

{

//execute your code

}

**Example:**

public class Sample {

public static void main(String args[]) {

int a = 30, b = 30;

if (b > a) {

System.out.println("b is greater");

}

else if(a > b){

System.out.println("a is greater");

}

else {

System.out.println("Both are equal");

}

}

}

**Output:**

Both are equal

* 1. **Switch Statement**

Java switch statement is used when you have multiple possibilities for the if statement.

**Syntax:**

switch(variable)

{

case 1:

//execute your code

break;

case n:

//execute your code

break;

default:

//execute your code

break;

}

**Example:**

public class Sample {

public static void main(String args[]) {

int a = 5;

switch (a) {

case 1:

System.out.println("You chose One");

break;

case 2:

System.out.println("You chose Two");

break;

case 3:

System.out.println("You chose Three");

break;

case 4:

System.out.println("You chose Four");

break;

case 5:

System.out.println("You chose Five");

break;

default:

System.out.println("Invalid Choice. Enter a no between 1 and 5");

break;

}

}

}

**Output:**

You chose five

**LOOPS IN JAVA**

Sometimes we want a Java program to repeat something over and over again, loops are used to make a program do something more than one time.

Java loops execute a block of commands a specified number of times, until a condition is met.

Java supports following types of loops:

* [while loops](http://www.w3schools.in/java/loops/while/)
* [do while loops](http://www.w3schools.in/java/loops/do-while/)
* [for loops](http://www.w3schools.in/java/loops/for/)

1. **While loop**

Java while loops statement allows to repeatedly run the same block of code, until a condition is met.

**Syntax:**

While (condition)

{

statement(s);

Incrementation;

}

**Example:**

public class Sample {

public static void main(String args[]) {

/\* local variable Initialization \*/

int n = 1, times = 5;

/\* while loops execution \*/

while (n <= times) {

System.out.println("Java while loops:" + n);

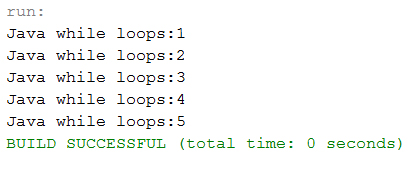
n++;

}

}

}

**Output:**



1. **do while loop**

Java do while loops is very similar to the java while loops, but it always executes the code block at least once and further more as long as the condition remains true.

**Syntax:**

do

{

statement(s);

} while( condition );

**Example:**

public class Sample {

public static void main(String args[]) {

/\* local variable Initialization \*/

int n = 1, times = 0;

/\* do-while loops execution \*/

do {

System.out.println("Java do while loops:" + n);

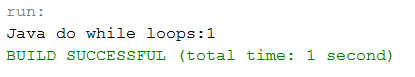
n++;

} while (n <= times);

}

}

**Output:**



1. **For loop**

Java for loops is very similar to Java while loops in that it continues to process a block of code until a statement becomes false, and everything is defined in a single line.

**Syntax:**

for ( init; condition; increment )

{

statement(s);

}

**Example:**

public class Sample {

public static void main(String args[]) {

/\* local variable Initialization \*/

int n = 1, times = 5;

/\* for loops execution \*/

for (n = 1; n <= times; n = n + 1) {

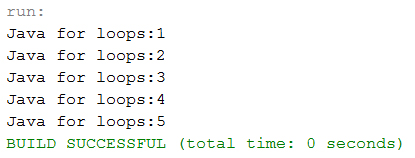
System.out.println("Java for loops:" + n);

}

}

}

**Output:**



## The fore ach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

### Example

The following code displays all the elements in the array myList −

public class TestArray {

public static void main(String[] args) {

double[] myList = {1.9, 2.9, 3.4, 3.5};

// Print all the array elements

for (double element: myList) {

System.out.println(element);

}

}

}

This will produce the following result −

**Output**

1.9

2.9

3.4

3.5

## BRANCHING STATEMENTS IN JAVA

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.

## break Statement

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

public class BreakDemo

{

public static void main(String[] args)

{

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

{

if (i == 5)

{

break; // terminate loop if i is 5

}

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

}

System.out.println("Loop is over.");

}

}

**Output :**

1 2 3 4 Loop is over.

The break statement has two forms: labeled and unlabeled. You can also use an unlabeled break to terminate a for, while, or do-while loop

* 1. **continue Statement**

The continue statement skips the current iteration of a for, while, or do-while loop. When a continue statement is encountered inside the body of a loop, remaining statements are skipped and loop proceeds with the next iteration.

public class ContinueDemo

{

public static void main(String[] args)

{

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

{

if (i % 2 == 0)

{

continue; // skip next statement if i is even

}

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

}

}

}

**Output :**

1 3 5 7 9

The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop.

## return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return statement has two forms: one that returns a value, and one that doesn't.

To return a value simply put the value (or an expression that calculates the value) after the return keyword.

return ++count;

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

return;