## History of Java:

**Java** is a general purpose, class based, object oriented, platform independent, portable, architecturally neutral, multithreaded, dynamic, distributed, and robust interpreted programming language.

In 1990, Sun Microsystems has started a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called Stealt h Project but later its name was changed to **Green Project**.

In 1991, James Gosling, Bill Joy, Patric Naughton, Mike Sheradin and several others met in Aspen, Colorado to discuss this project. Mike Sheradin was to focus on business development; Patric Naughton was to begin work on the Graphics system; and James Gosling was to identify proper programming language for the project. Gosling thought C and C++ could be used to develop the project. But the problem he faced with them is that they were system dependent languages and hence could not be used on various processors, which the electronic devices might use. So he started developing a new language, which was completely system independent. This language was initially called **Oak.** Since the name was registered by some other company, later it was changed to **Java**.

**Why the name Java?:** James Gosling and his team members were consuming lot of coffee while developing this language. They felt that they were able to develop a better language because of the good quality coffee they consumed and quality coffee was exported to the entire world from a place called "Java Island". Hence they fixed the name of the place for the language as Java. And the symbol for the Java language is coffee cup and saucer.

By September of 1994, Patric Naughton and others started writing WebRunner- a Java based Web Browser, which was later renamed as HotJava. HotJava was the first browser, having the capabilities of executing Applets, which are programs to run dynamically on Internet. Finally, Sun Microsystems formally announced Java and HotJava in 1995.

**Java Version History:**

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014) - 217 Packages and 4240 Classes.
11. Java SE 9 (September 2017)
12. Java SE 10 (September 2018)

## Importance of Java to Internet:

The Internet helped Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content.

**1. Java Applets:** An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server.

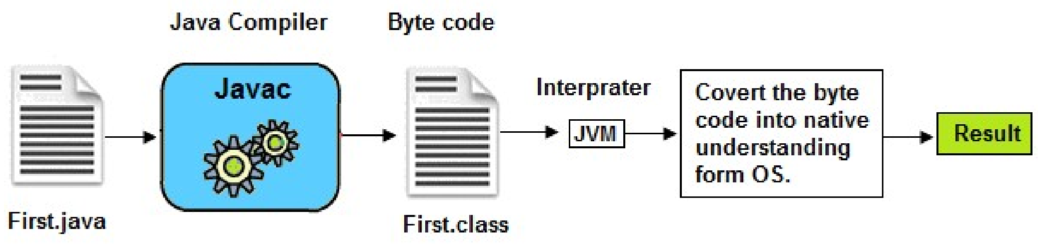
**2. Security:** When we download data from the Internet, there is chance of risks (such as virus, worms, trojans) .For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack. Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

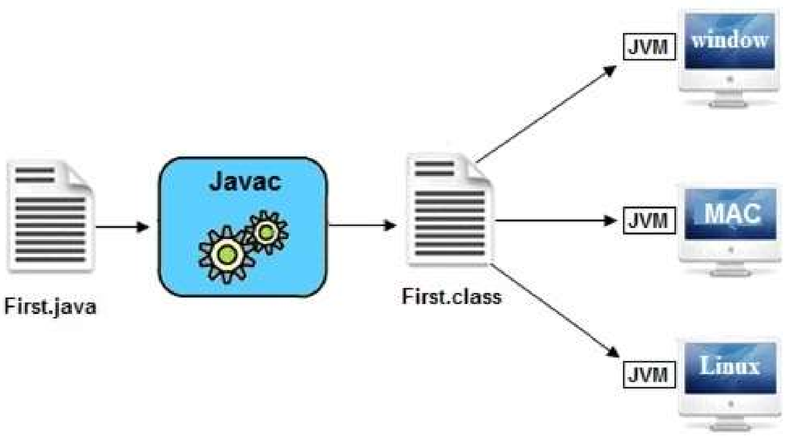
**3. Portability:** If any language supports platform independent and architectural neutral features, then that language is portable and if a program yields the same result on any machine, then that program is also called portable. Java achieves this with **byte code**.

## Java's Magic: The Bytecode

Java solves both the security and the portability problem is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*.

In essence, the original JVM was designed as an *interpreter* for bytecode. Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.

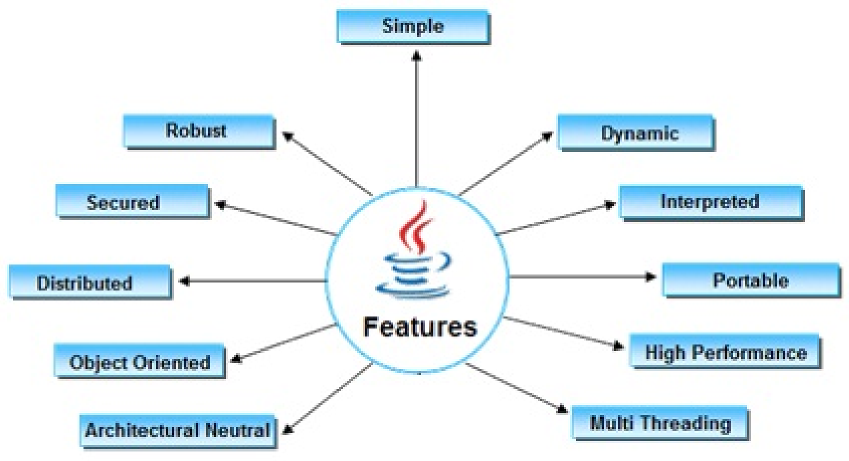




In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect. Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was developed by Java people. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted.

**<u>Features of Java/ Java Buzzwords:</u>**

Features of Java are:



1. **Simple:**

   I. It is free from pointer due to this execution time of application is improve.

   II. It have rich set of APIs (application programming interface).

   III. It have Garbage Collector which is always used to collect un-referenced (unused) memory locations for improving performance of a Java program.

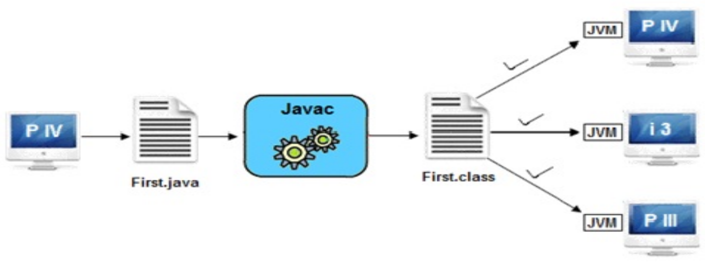
   IV. It contains user friendly syntax for developing any applications.

2. **Object Oriented:** Java supports OOP features and everything is an Object in Java. The object model in Java is simple and easy to extend.

3. **Robust:** Java is robust or strong programming language because of its capability to handle run-time errors, automatic garbage collection, lack of pointer concept, exception handling. These entire things make Java is a robust Language.

4. **Multi Threading:** Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

5. **Architecture-Neutral:** Architecture represents processor. A language or technology is said to be architectural neutral which can run on any processors in the real world without considering type of architecture and vendor (providers).



**6. Interpreted:** Java programs are compiled to generate the byte code. This byte code can be downloaded and interpreted by the interpreter in JVM. If we consider any other language, only an interpreter or a compiler to execute the programs. But in Java, we use both interpreter and compiler for the execution.

**7. High Performance:** The problem with interpreter inside JVM is that it is slow. Because of this, Java programs used to run slow. To overcome this, along with interpreter, Java people have introduced JIT (Just-in-Time) compiler, which enhances the speed of execution. So now in JVM, both interpreter and JIT compiler works together to run the program.

**8. Distributed:** Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. Java also supports *Remote Method Invocation (RMI).* This feature enables a program to invoke methods across a network.

**9. Dynamic:** Java support dynamic memory allocation. Due to this memory wastage is reduced. The process of allocating the memory space to the program at a run-time is known as dynamic memory allocation, To allocate memory space dynamically we use an operator called 'new'. 'new' operator is known as dynamic memory allocation operator.

**10. Secure:** It is more secured language compare to other languages; In this, program is covered into byte code after compilation which is not readable by human and also Java does not support global variable concept.

**11. Portable:** If any language supports platform independent and architectural neutral features, then that language is portable.

**Portable = Platform independent + Architecture-neutral**

**Differences between Java and C++:**

| S. No | Java | C++ |
|---|---|---|
| 1 | Java is a pure object oriented programming language; therefore, everything is an object in Java | C++ supports both procedural and object oriented programming; therefore, it is called a hybrid language. |
| 2 | Java does not support pointers, templates, unions, operator overloading, structures etc. | C+ supports structures, unions, templates, operator overloading, pointers and pointer arithmetic. |
| 3 | Java support automatic garbage collection. It does not support destructors as C++ does. | C++ support destructors, which is automatically invoked when the object is destroyed. |
| 4 | Java has built in support for threads. In Java, there is a `Thread` class that you inherit to create a new thread. | C++ has no built in support for threads. C++ relies on non-standard third-party libraries for thread support. |
| 5 | There is no scope resolution operator (::) in Java. The method definitions must always occur within a class. | C++ has scope resolution operator (::) which is used to define a method outside a class and to access a global variable within from the scope where a local variable also exists with the same name. |
| 6 | There is no *goto* statement in Java. The keywords `const` and `goto` are reserved, even though they are not used. | C++ has *goto* statement. |
| 7 | Java doesn't provide multiple inheritance | C++ support multiple inheritance. The keyword `virtual` is used to resolve ambiguities during multiple inheritance if there is any. |
| 8 | Java is interpreted for the most part and hence platform independent. | C++ generates object code and the same code may not run on different platforms. |

## Java is a strongly typed language:

In Java, every variable has a type, every expression has a type, and every type is strictly defined. And, also all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.

## The Primitive Types:

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types.

These can be put in four groups:

1. **Integers:** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers. Following table specified its width and range.

| **Name** | **Width** | **Range** |
|---|---|---|
| long | 64 | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| int | 32 | –2,147,483,648 to 2,147,483,647 |
| short | 16 | –32,768 to 32,767 |
| byte | 8 | –128 to 127 |

2. **Floating-point numbers:** This group includes **float** and **double**, which represent numbers with fractional precision.

| **Name** | **Width** | **Range** |
|---|---|---|
| double | 64 | 4.9e–324 to 1.8e+308 |
| float | 32 | 1.4e–045 to 3.4e+038 |

3. **Characters:** This group includes **char**, which represents symbols in a character set, like letters and numbers. In Java, character data type occupies two bytes of memory because of UNICODE representation.

4. **Boolean:** This group includes **boolean**, which is a special type for representing true/false values.

   Ex: boolean b;

**Variables:** Variable represents basic unit of storage in a Java program. In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

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

Example:

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three   ints, initializing d and f.
byte z = 22; // initializes z.
```

**/* Java program to illustrate boolean data type*/**

```
class Bool_Example
{
        public static void main(String args[])
        {
                boolean b;
                b = false;
                System.out.println("b is " + b);
                b = true;
                System.out.println("b is " + b);
                // a boolean value can control the if statement
                if(b) System.out.println("This is executed.");
                b = false;
                if(b) System.out.println("This is not executed.");
                // outcome of a relational operator is a boolean value
                System.out.println("10 > 9 is " + (10 > 9));
        }
}
```

## 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.

## Scope and lifetime of variables:

Java allows variables to be declared within any block. 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.

Many other computer languages define 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.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.

**Here is another important point to remember:** variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.

**/* Write a Java program to illustrate scope of a variable*/**

```
// Demonstrate block scope.
class Scope_Example
{
        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 );
        }
}
```

<u>**Type Conversion and Casting:**</u> Assigning a value of one type to a variable of another type. Java supports two types of type casting.

1. Automatic type casting:

   *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.
2. Explicit type casting: 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:

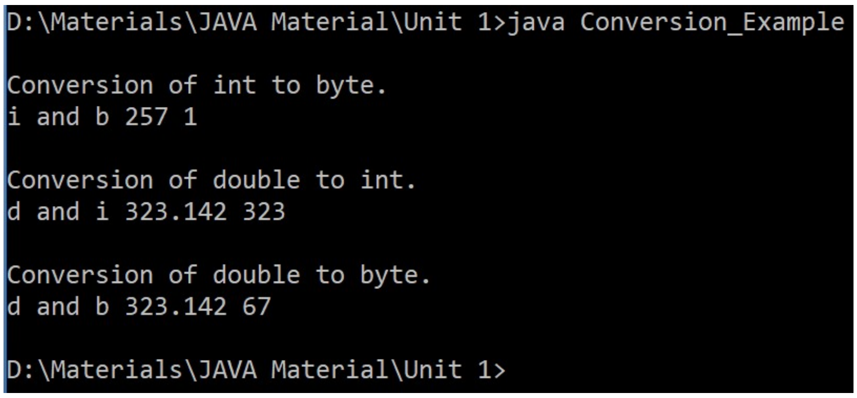
   variable=(*target-type) value*

**/* Java program to illustrate explicit type casting*/**

```
class Conversion_Example
{
        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);
                }
}
```

**Output:**



```
D:\Materials\JAVA Material\Unit 1>java Conversion_Example

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

D:\Materials\JAVA Material\Unit 1>
```

## Arrays:

- An *array is a group of variables that are referred to by a common name.*
- *Arrays of* any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.

**One-Dimensional Arrays:**

The general form of a one-dimensional array declaration is

*type var-name[ ];*

Ex: int month_days[];

Although this declaration establishes the fact that month_days is an array variable, no array actually exists. In fact, the value of month_days is set to null, which represents an array with no value. To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days. new is a special operator that allocates memory.

```
var_name = new type [size];
month_days = new int[12];
```

- Another version:

  It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

  ```
  int month_days[] = new int[12];
  ```

- Another version:
  - int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

  Here, we can initialize arrays in curly braces by separating with comma.

**Alternative Array Declaration Syntax:**

- There is a second form that may be used to declare an array:

  *type[ ] var-name;*

- Here, the square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:

  ```
  int al[] = new int[3];
  int[] a2 = new int[3];
  ```

- The following declarations are also equivalent:

  ```
  char twod1[][] = new char[3][4];
  char[][] twod2 = new char[3][4];
  ```

- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,
  - int[] nums, nums2, nums3; // create three arrays creates three array variables of type int.

  It is the same as writing

  ```
  int nums[], nums2[], nums3[]; // create three arrays
  ```

**/* Java program to illustrate arrays*/**

```
class Array_Example
{
        public static void main(String args[])
        {
                int month_days[];
                month_days = new int[12];
                month_days[0] = 31;
                month_days[1] = 28;
                month_days[2] = 31;
                month_days[3] = 30;
                month_days[4] = 31;
                month_days[5] = 30;

                month_days[7] = 31;
                month_days[8] = 30;
                month_days[9] = 31;
                month_days[10] = 30;
                month_days[11] = 31;
                System.out.println("April has " + month_days[3]+ " days.");
        }
}
```

**Multi-Dimension arrays:**

To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two dimensional

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**.

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

**Operators:**

1. Arithmetic : +, -,*,/,%,++,+=,-=,*=,/=,%=,--
2. Bitwise operators(works on only integer group): &, |, ^, >>,<<, >>>,<<<, &=, |=, ^=
3. Relational: ==,!=, >,<,>=,<=
4. Boolean logical operands(Works only on boolean operands): &, |, ^, ||, &&, !, !=, ?:, ==

**Control Statements:**

Java's program control statements can be put into the following categories: selection, iteration, and jump.

- ***Selection*** statements: allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
  - *if and its variants, switch*
- ***Iteration*** statements enable program execution to repeat one or more statements until some condition is satisfied.
  - while, do-while, for, for-each
- ***Jump*** statements allow your program to execute in a nonlinear fashion.
  - break, continue, return

**Selection statements:** Java supports two selection statements:

1. if
2. switch

**if statement:**

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if (condition)
        statement1;
else
        statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

**Nested if:**

A *nested* if is an if statement that is the target of another if or else. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
        if(j < 20) a = b;
        if(k > 100) c = d; // this if is
        else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100) because it is the closest if within the same block.

**The if-else-if Ladder:** A common programming construct that is based upon a sequence of nested **if**s is the *if-elseif* ladder. It looks like this:

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
        .
        .
        .
else
        statement;
```

**/* Java program to demonstrate the use of if statement*/**

```
class CheckNumber{
    public static void main(String args[]){
        int num = 10;
        if (num % 2 == 0){
            System.out.println(num + " is even number );
        }
        else {
            System.out.println(num + " is odd number );
        }    } }
```

**Switch-case statement:** switch-case can be used as an alternative for if-else condition. If the programmer has to make number of decisions, and all the decisions depend on the value of variable, then switch-case is used instead of if-else condition.

**Syntanx** of switch-case:

```
switch(expression){
        case 1 :
              action1 statement;
               break;
        case 2:
              action2 statement;
               break;
                     .
                     .
                     .
      case N:
               actionN statement;
                 break;
        default:
              default statement;
}
```

Where expression: is variable containing the value to be evaluated. It must be of type byte, short, int, char and enumeration. From Java 7 onwards string types are also allowed.
**default:** is an optional keyword used to specify the statements to be executed only when all the case statements evaluate to false.

**/* Java program to demonstrate the use of switch statement*/**

```
class StringSwitch_Example
{
        public static void main(String args[])
        {
                String str = "two";
                switch(str) {
                        case "one":
                                System.out.println("one");
                                break;
                        case "two":
                                System.out.println("two");
                                break;
                        case "three":
                                System.out.println("three");
                                break;
                        default:
```

```
                System.out.println("no match");
                break;
            }
        }
    }
```

**Nested switch statement:** switch as part of the statement sequence of an outer switch. This is called a *nested* switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch.

For example, the following fragment is perfectly valid:

```
switch(count) {
        case 1:
                switch(target) { // nested switch
                        case 0:
                                System.out.println("target is zero");
                                break;
                        case 1: // no conflicts with outer switch
                                System.out.println("target is one");
                                break;
                }
                break;
                case 2: // ...
                .
```

**Iteration Statements:**

1. **While loop:** The while loop executes till condition is specified. It executes the steps mentioned in the loop only if the condition is true.

   **Syntanx** of while loop:

   ```
   while (condition) {
           statements;
           statements;
   }
   ```

   Where condition, is any boolean expression that returns a true or false value. The loop continues upto condition returns true value.

   **Example :**

   ```
   int i = 1;
   while(i<=5){
           i++;
         System.out.println("value of i is : "+i);
   }
   ```

2. **do-while loop:** The do-while loops execute certain statements till the specified condition is true. This loop ensures that the loop body is executed at least once.

   **Syntax** of do-while loop:

   ```
   do{
               action statements;
   }while(condition);
   ```

   **Example :**

   ```
   int j = 8;
   do{
       j = j+2;
       System.out.println("value of j is : " +j);
   }while(j>=10 && j<=50);
   ```

3. **for loop:** All loops have some common feature: a counter variable that is initialized before the loop begins. In for loop initialization of a variable before loop begins, a condition for counter variable and counter upto which loop lasts.

   **Syntax** of for loop:

   ```
   for(initialization statements; condition; increment/decrement statements){
       action statements;
                 .
                 .
   }
   ```

   where initialization statements : sets or initialize the value for counter variable.

   condition : A boolean expression that returns true or false value. The loop terminates if false value is returned.

   Increment/decrement statements : Modifies the counter variable. These statements are always executed after the action statements, and before the subsequent condition check.

   **Example:**

   ```
   for(int i=1;i<=10;i++){
               System.out.println("Value of i is " +i);
   }
   ```

4. **for-each:** 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:**

```
for(data_type variable : array | collection)
{
        Statements;
}
```

   **Example:**

```
public class ForEachExample
{
  public static void main(String[] args) {
        int arr[]={12,23,44,56,78};
        for(int i:arr)
        {
           System.out.println(i);
        }
     }
  }
```

**Jump statements:**

1. **break statement**: In Java, 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 alternative to goto statement.
     - Syntax:
       - break *label;*

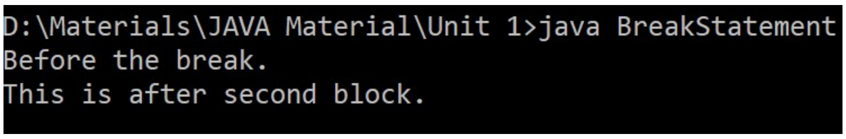
   **/* Java program to demonstrate the use of brake statement*/**

```
class BreakLoop
{
        public static void main(String args[])
        {
                for(int i=0; i<100; i++)
                {
                        if(i == 10) break; // terminate loop if i is 10
                                System.out.println("i: " + i);
                }
                System.out.println("Loop complete.");
        }
}
```

**/* Java program to demonstrate the use of brake statement in labels */**

```
class BreakStatement
{
        public static void main(String args[])
        {
                boolean t = true;
                first:
                {
                        second:
                        {
                                third:
                                {
                                        System.out.println("Before the break.");
                                        if(t)
                                            break second; // break out of second block
                                         System.out.println("This won't execute");
                                }
                                System.out.println("This won't execute");
                        }
                        System.out.println("This is after second block.");
                }
        }
}
```

**Output:**



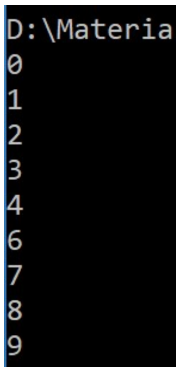
```
D:\Materials\JAVA Material\Unit 1>java BreakStatement
Before the break.
This is after second block.
```

2. **continue statement:** Sometimes the programmer might want to continue a loop, but stop processing the remainder of the code in its body for a particular iteration. The continue statement can be used for such a scenario. It can be also used in Java as alternative to goto statement.

   **/* Java program to demonstrate the use of continue statement*/**

```
class ContinueStatement
{
        public static void main(String args[])
        {
                for(int i=0; i<10; i++)
                {

                        if (i==5)
                                continue;
                        System.out.println(i);  } } }
```



```
D:\Materia
0
1
2
3
4
6
7
8
9
```

**/* Java program to demonstrate the use of continue with labels */**

```
import java.util.*;
public class Factorial
{
        public static void main(String[] args)
        {
                int x = 1;
                int factValue = 1;
                Scanner userInput = new Scanner(System.in);
                restart:
                while(true)
                {
                        System.out.println("Please enter a nonzero, nonnegative  ");
                        int factInput = userInput.nextInt();
                        if(factInput<=0)
                        {
                             System.out.println("Enter a nonzero, nonnegative value  ");
                             factInput = userInput.nextInt();
                        }
                        if(factInput<1)
                        {
                                System.out.println("The number you entered is not valid. Please
                                                        try again.");
                                continue restart;
                        }
                        while(x<=factInput)
                        {
                                factValue*=x;
                                x++;
                        }
                        System.out.println(factInput+"! = "+factValue);
                        break restart;
                  }
             }
        }
```

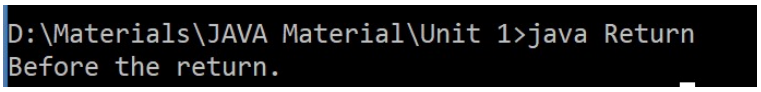
3. **Return statement:**

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.

The following example illustrates this point. Here, return causes execution to return to the Java run-time system, since it is the run-time system that calls main( ):

```
class Return
{
        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:



```
D:\Materials\JAVA Material\Unit 1>java Return
Before the return.
```

**Scanner Class:**

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double etc. and strings. It is the easiest way to read input in a Java program.

1. To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
2. To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
3. To read strings, we use nextLine().
4. To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) funtion returns the first character in that string.

**Methods:**

| Method | Description |
|---|---|
| public String next() | it returns the next token from the scanner. |
| public String nextLine() | it moves the scanner position to the next line and returns the value as a string. |
| public byte nextByte() | it scans the next token as a byte. |
| public short nextShort() | it scans the next token as a short value. |
| public int nextInt() | it scans the next token as an int value. |
| public long nextLong() | it scans the next token as a long value. |
| public float nextFloat() | it scans the next token as a float value. |
| public double nextDouble() | it scans the next token as a double value. |

**/* Java program to demonstrate the use of scanner class*/**

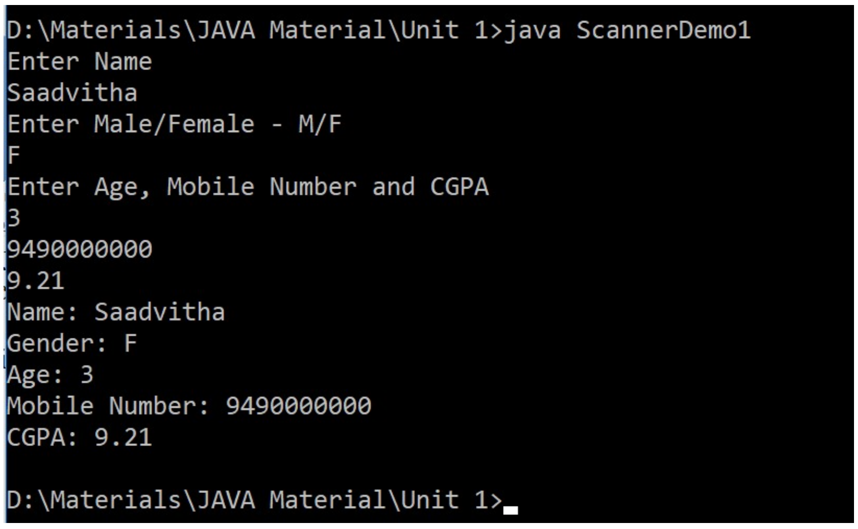
```
import java.util.Scanner;
public class ScannerDemo1
{
    public static void main(String[] args)
    {
             Scanner sc = new Scanner(System.in);
      System.out.println("Enter Name");
      String name = sc.nextLine();

      System.out.println("Enter Male/Female - M/F");
              char gender = sc.next().charAt(0);

             System.out.println("Enter Age, Mobile Number and CGPA");
             int age = sc.nextInt();
              long mobileNo = sc.nextLong();
              double cgpa = sc.nextDouble();

       // Print the values to check if input was correctly obtained.
       System.out.println("Name: "+name);
       System.out.println("Gender: "+gender);
       System.out.println("Age: "+age);
       System.out.println("Mobile Number: "+mobileNo);
       System.out.println("CGPA: "+cgpa);
    }
}
```

**Output:**



```
D:\Materials\JAVA Material\Unit 1>java ScannerDemo1
Enter Name
Saadvitha
Enter Male/Female - M/F
F
Enter Age, Mobile Number and CGPA
3
9490000000
9.21
Name: Saadvitha
Gender: F
Age: 3
Mobile Number: 9490000000
CGPA: 9.21

D:\Materials\JAVA Material\Unit 1>
```