❖ Give reason: Java is strongly typed language.

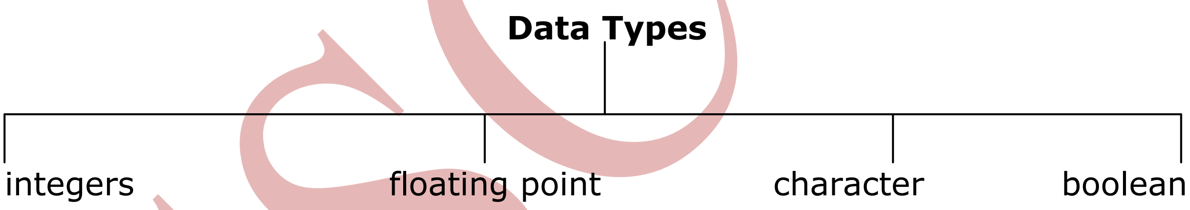
1. Every variable has a type, every expression has a type and every type is strictly defined.
2. All assignments whether explicit or via parameter passing in methods calls are checked for type compatibility.
3. There are no automatic conversions of conflicting types as in some languages.
4. The Java compiler checks all expressions and parameters to ensure that the types are compatible.
5. Any types mismatches are errors that must be corrected before the compiler will finish compiling the class

For example;

```
int x;
float y =45.67;
x=y;  //valid in C/C++ but not in Java
```

❖ How many data types are available in Java?

Java defines eight simple or elemental data types**. They are byte, short, int, long, char, float, double and Boolean**. All these data types can be put in four groups:



integers	floating point	character	boolean
(1) Byte	(1) float		(1) true
(2) Short	(2) double		(2) false
(3) int			
(4) long			

(1) Integers

Java defines four integer types: **byte, short, int, long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them.

The width and ranges of these integer types vary widely, as shown in this table:

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	08	-128 to 127

- **byte:** The smallest integer type is byte. This is a signed 8-bit types that has a range from **-128 to 127**. Byte data type is used when you are working with a stream of data from a network or file.

  The following example declares two byte variables called b & c.

  ```
  byte b,c;
  ```

- **short:** Short is a signed 16-bit type. It has range from -32,768 to 32,767. it is probably the least-used Java type, since it is defined as having its high byte first (called big-endian format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce.

  The following example declares two short variables called s & t.

  ```
  short s;
  short t;
  ```

- **int:** The most commonly used integer type is **int.** It is a signed 32-bit type that has a range from **-2,147,483,648 to 2,147,483,647**. Int type variables are commonly used to control loops and to index arrays. Any time you have an integer expression involving **bytes, shorts, ints and literal numbers,** the entire expression is promoted to **int** before the calculation is done.

  The following example declares two int variables called i & j.

  ```
  int i, j;
  ```

- **long: long** is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is very large. Long data type is used when big numbers are needed.

  The following example declares two long variables called d & s.

  ```
  long d, s;
  ```

(2) Floating-point value

Floating point numbers are also known as **real numbers**. It is useful when we want to calculate square root, sine and cosine which require a floating-point type. There are two types of floating point types float and double respectively. Their width and ranges are shown here:

Name	Width	Range
Double	64	4.93-324 to 1.8e+308
Float	32	1.4e-045 to 3.4e+038

- **float:** This type float specifies a single-precision value that uses **32 bits** of storage. Single precision is faster on some processors and takes half as much space as double precision. It will become rough when the values are either very large or very small. Variables of type float are useful when you need a fractional component but don't require a large degree of precision.

  The following example declares two float variables called f1 & f2.

  ```
  float f1,f2;
  ```

- **double:** This type double specifies a double-precision value that uses **64 bits** of storage. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All math functions, such as **sin()**, **cos()** and **sqrt()** return double values. When you need to maintain accuracy over many iterative calculations or are manipulating large value numbers **double** is the best choice.

  The following example declares two double variables called d1 & d2.

  ```
  double d1,d2;
  ```

Here is a short program that uses double variables to compute the area of a circle.

```
//compute the area of a circle.

class area
{       public static void main(String arg[])
        {       double pi,r,a;
                r=10.8;
                pi=3.14;
                a=pi*r*r;
                System.out.println("Area of circle is: " +a);
        }
}
```

(3) characters

Java used **char type** to store character value. There is difference between char data type in C/C++ and Java. **char** is an integer type and 8 bits wide in C/C++. This is not in the case of Java. Java uses Unicode to represent characters. **Unicode defines** a fully international character set that can represent all of the characters found in all human languages. It defines the set of characters of various languages such as **Latin, Greek, Arabic, Cyrillic, Hebrew, Hangul and many more**.

For the purpose of Unicode **char** data type requires 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as **ASCII** and its range from 0 to 127. The remaining characters are 8 bit wide and its range from 0 to 255.

The following example declares two char variables called c1 & c2.

```
char d1,c2;
```

Here is the program that demonstrates **char** variables:

```
class chardemo
{
        public static void main(String arg[])
        {
                char ch1,ch2;
                ch1=88;
                ch2='Y';
                System.out.println("ch1 and ch2:");
                System.out.println(ch1 +" " +ch2);
        }
}
```

This program displays the following output:

```
        ch1 and ch2: X Y
```

Note: You can found more information about Unicode on website:
http://www.unicode.org

(4) booleans

Java has a simple type called Boolean for logical values. It can have only one of two possible values **true or false**. All the relational operator, if statement and loop statement return logical value either true or false.

here is the program that demonstrates **boolean** variables:

// Demonstrate boolean values.

```
class booltest {
  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));
  }
}
```

❖ **Literal or Constants:** Constant in Java means fixed values that do not change during the execution of program. Java support several types of constants given as bellow.

Integer Literal

An integer constant refers to a sequence of digits. There are three types of integers literal namely, decimal integer, octal integer and hexadecimal integer.

Decimal integer literals consist of a set of digits, 0 through 9, preceded by an optional minus sign. Valid examples of decimal integer constants are:

```
    123        -321        0         456892
```

An octal integer literal consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

```
    037        0          0435       0557
```

Hexadecimal integer literal consist of digit from the set 0 through 15, so A through F (or a through f) are substituted for 10 through 15. You suggest a hexadecimal constant with a leading zero-x (**0X or 0x**). Some examples of hexadecimal integer are:

```
    0X2        0X9        0X12       0X15
```

Integer literal create an int value, which is a 32-bit integer value. You can assign an integer literal to of Java's other integer types, such as **byte or long** without causing a type mismatch error. When a literal value is assigned to a **byte, short or long** variable, no error is generated if the literal value is within the range of the target type. However to specify a long literal, you will need to explicitly tell the compiler that the literal value is of type long. You do this by an appending an **upper or lowercase L/l** to the literal.

For example,

```
class longint
{
    public static void main(String arg[])
    {
    long l1=4578909897654;              //   error!!! you have to use
                                        //   long
      l1=4578909897654L
      l1=56489;
```

```
        int i=l;        // you can assign long value to int if it is within
                        // the range of target data type.

        System.out.println("LONG=" +l1 +"\ninteger" +i);
    }
}
```

Floating-Point Literal

Floating point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation.

Standard notation consists of a whole number component followed by a decimal point followed by a fractional component. **For example,** 2.0, 3.14159 and 0.6667 represent valid standard notation floating point numbers. Some standard notation floating variables may be distance, heights, temperatures and prices and so on.

Scientific notation uses a standard-notation, floating point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. **The general form is**

mantissa e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer with an optional plus or minus sign. The letter e separating the mantissa and the exponent can be written in either uppercase or lowercase.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5e9 or 75e8. Similarly -0.000000368 is equivalent to -3.68e-7.

For example, 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by $\mathbf{10^2}$.

Floating point literals in Java default to double precision. To specify a float literal, you must append an **F or f** to the constant. You can also explicitly specify a double literal by appending a **D or d**.

For example,

```
class floatval
{
      public static void main(String arg[])
      {
             float sum=0.0f;
             float num1=10.50F,num2=10.50F;
             sum=sum+num1+num2;
             System.out.println("sum=" +sum);
      }
}
```

Boolean literal

Boolean literals are simple. There are only two logical values that a Boolean value can have, **true and false**. The values of true or false do not convert into any numerical representation. The true literal in Java does not equal **1**, nor does the false literal equal **0**.

Character literal

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as addition and subtraction operators.

A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes such as 'a', 'z' and '@'.

For characters that are impossible to enter directly, there are several **escape sequences**, which allow you to enter the character you need, such as '\" for the single-quote character itself, and '\n' for the newline character.

There is also a mechanism for directly entering the value of a character in **octal or hexadecimal**. For octal notation use the backslash followed by the three-digit number.

For example, **'\141'** is the letter 'a'

For hexadecimal notation, you enter a **backslash-u (\u)**, then exactly four hexadecimal digits. For example, **'\u0061'** is the ISO-Latin-1 'a' because the top byte is zero.

Following table shows the character escape sequences.

Escape Sequence	**Description**
\ddd	Octal Character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\'	Single quote
\""	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\t	Tab
\b	Backspace
\f	Form Feed

Character Escape Sequences

<u>String Literals</u>

String literals in Java are specified like same as specified in other languages. Strings are actually object types. Strings are not implemented as arrays of characters like in other languages.

One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there in other languages. **Examples** of string literals are as bellow

"Hello World" "two\n lines" "\ "This is in quotes\""

"WEL COME" "I.N.D.I.A" "3+2" "S"

Consider the following program

```
class strdemo
{
        public static void main(String arg[])
        {
                String s1="Radhe Krishna";
                System.out.println("S1=" +s1);
                int len=s1.length(),i=0;

                while(i<len)
                {
                System.out.println("Character " +(i+1) +": " +s1.charAt(i));
                        i++;
                }
                String s2=”Radhe”;
                String s3=”Krishna”;
                String s4=s2 +”  “ + s3;
                System.out.println(“String s4=” +s4);
        }
}
```

❖ How to declare and dynamically initialized variables?

Variable: The variable is the basic unit of storage in Java program. A variable is defined by the combination of an identifier, a type and an optional initialize. All variables have a scope, which defines their visibility and a lifetime.

The syntax to declare variable:

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

type	= any valid Java data type or name of class or interface.
identifier	= name of variable
=value :	you can initialize the variable by specifying an equal sign and a value.

Here are several examples of variable declaration of various types.

```
int a,b,c;              //declares three int a,b and c.
int d=3,e,f=5;          //declares three more ints initializing d and f.
byte z=22;              // initialized z by 22
double pi=3.14159;      //declares an approximation of pi.
char x=’x’;             // the variable x has the value ‘x’.
```

Dynamic initialization: Java allows variables to be initialized dynamically, using any expression valid at the time of variable declaration.

The key point here is that the initialization expression may use any element valid at the time of initialization including calls to methods, other variables or literals.

For example,

//demonstrate dynamic initiazliztion.

```
class dyndemo
{
        public static void main(String arg[])
        {
                double a=3.0,b=4.0;
                //c is dynamically initialized
            double c=Math.sqrt(a*a + b* b);
            System.out.println("c=" +c);
}}
```

❖ What is Scope and lifetime of variables?

Java variables are actually classified into three kinds:

1. instance variable (class object)
2. class variables (class data member)
3. local variables

Instance and class variables are declared inside a class. Instance variables are created when the objects are created and therefore they are associated with the objects. They take different values for each object.

Class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable.

Variables declared and used inside methods are called **local variables**. They are called local variables because they are not available for use outside the method definition.

Local variables can also declared inside program blocks that are defined between an opening brace { and a closing brace }. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will be destroyed.

Each block can contain its own set of local variable declarations. We cannot declare variable to have the same name as one in an outer block. It means any block can not have the same variable name.

Scope: The area of the program where the variable is accessible (i.e usable) is called its scope.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope.

To understand the effect of nested 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);
  }
}
```

Output:

```
X and y: 10 20
X is 40
```

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. **For example**, consider the following program.

```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y iz: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

Output:

```
y is: -1
y is now: 100
y is: -1
```

```
y is now: 100
y is: -1
y is now: 100
```

❖ **Type Conversion and Casting:**

There are two types of conversion in Java.

(1) Automatic Type conversion
(2) Casting Incompatible Types.

➢ **Automatic Type Conversion:**

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:

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

When these two conditions are met, a **widening (promotion) conversion** takes place. For example the **int** type is always large enough to hold all valid short & byte values. So explicit cast statement is not required.

For example,

```
short s1=5432;
float f1;
double d1;
int i;
i=s1;
f1=i1;
d1=f1;
```

The following table list which are guaranteed to result in no loss of information.

From (Source Type)	**To (Destination Type)**
Byte	short, int, long, float, double, char
Short	int, long, float, double
Int	long, float, double
Long	float, double
Float	Double
Char	int, long, float, double

➢ Casting Incompatible Types:

When you want to assign larger data type value into smaller data type automatic conversion will not be possible. So, you have to explicitly convert source data type into destination data type. This kind of conversion is Sometime known as narrowing conversion.

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

(target_type) value

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);
  }
}
```

➢ Automatic Type Promotion In Expressions:

In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, consider the following expression:

```
byte a=40,b=50,c=100;
int d=a*b/c;
```

The result of intermediate term **a*b** easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte or short operand to **int** when evaluating an expression.

This means that the sub expression **a*b** is performed using integers-not bytes. Thus, 2000 the result of the intermediate expression, 50*40 is legal even though **a and b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, the following code generates a problem:

```
byte b=50;
b=b*2;      //error! Can't assign an int to a byte!
                        Or
byte b=50,b1=2;
b=b*b1;     //error! Can't assign an int to a byte!
```

The code generates compile-time error, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of cast.

In this case you should use an explicit cast, such as

```
byte b=50;                  or          byte b=50,b1=2;
b=(byte) (b*2);                         b=(byte)(b*b1);
```

- **The type promotion rules:**

Java defines several type promotion rules that apply to expressions. They are as follows.

1. All byte and short values are promoted to **int**. If one operand is a long, the whole expression is promoted to **long**.
2. If one operand is a **float**, the entire expression is promoted to **float**.
3. If one operand is **double**, the result is **double**.

The following program demonstrates some type conversions in expression.

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

❖ Introduction to arrays in Java

Array: An array is group of like-typed 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.

1-D Array: Like any other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps:

1. Declaring the array
2. Creating memory locations
3. Initialization of arrays

➢ **Declaration of arrays:**

Arrays in Java may be declared in two forms:

Form 1: type arrayname[];
Form 2: type [] arrayname;

For example;

```
int number[];
int month_days[];
                    or
int [] counter;
float [] average;
```

The value of above all variables is set to **null** which represents an array with no value.

➢ **Creation of arrays:** After declaring an array, we need to create in the memory. Java allows us to create arrays using new operator only, as shown below:

```
arrayname=new type[size];
```

For example;

```
month_days=new int[12];
average=new float[10];
```

These lines create necessary memory locations for the arrays number and average and designate them as **int** and **float** respectively. Now, the variable **month_days** refers to an array of 12 integers and average refers to an array of 10 floating point values.

It also possible to combine the two steps-declaration and creation-into single statement as shown bellow:

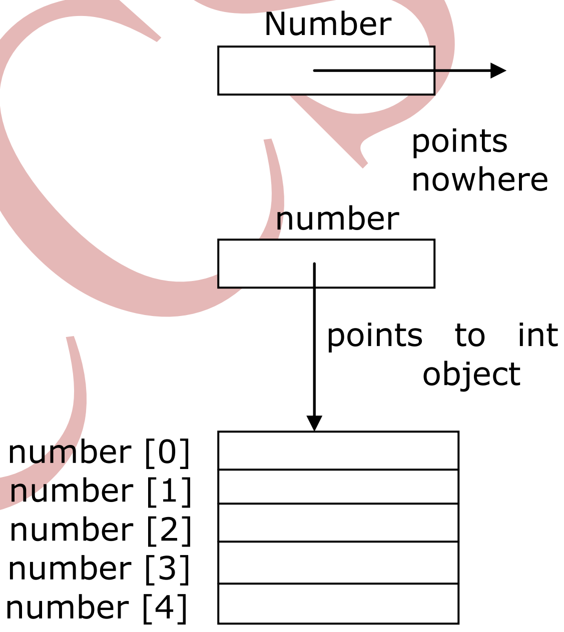
```
int number[]=new int[5];
float average[]=new float[3];
float  marks[]=new float[3];
char c1[]=new char[3];
String s1[]=new String[3];
```

Figure illustrates creation of an array in memory.

Statement **Result**

int number[]

number=new int [5];



➢ **Initialization of Arrays:**

The final step is to put values into the array created. This process is known as initialization. This is done using the array subscripts as shown below:

```
arrayname[subscript]=value;
```

Example:

```
number[0]=10;
number[1]=20;
number[2]=30;
……………………….
……………………….
Number[N]=50;
```

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

```
type arrayname[]={list of values};
```

For example;

```
int number[]={10,20,30,40,50};
String name[]={"MUMBAI","CHENNAI","DELHI","CALCUTTA"};
```

It is possible to assign an array object to another array object. For example

```
int a[]={1,2,3};
int b[];
b=a;
```

are valid in Java. Both the array will have the same values.

➢ Array Length:

In Java, all arrays store the allocated size in a variable named length. We can obtain the length of the array **a** using **a.length**. For example,

```
int a[]={10,20,30};
int len;
len=a.length;
```

This information will be useful in the manipulation of arrays when their sizes are not known:

➢ Remember Points about Arrays:

1. Java creates arrays starting with the subscript 0 and ends with a value one less than the size specified.
2. Unlike C, Java protects arrays from overruns and under runs. Trying to access an array bound its boundaries will generate an error message.

❖ **2-D Array:** A collection of rows and columns is known as 2-D Array.

Declaration:

```
int twod[][]=new int[3][3];
                    or
int twod[][];
twod[][]=new int[3][3];
```

This allocates a 2 by 2 array and assigns it to twod. Internally this matrix is implemented as an array of arrays of int. **Conceptually**, this array will look like the one shown in the following figure.

	Col 0	**Col 1**	**Col 2**
Row->0	[0] [0]	[0] [1]	[0] [2]
Row->1	[1] [0]	[1] [1]	[1] [2]
Row->2	[2] [0]	[2] [1]	[2] [2]

Conceptual view of a 3 by 3 matix, two dimensional array.

❖ **Multidimensional Array or Irregular Array:** In 2-D array you need only specify the memory for the **first (leftmost) dimension**. You can allocate the remaining dimension separately.

For example, this following code allocates memory for the first dimension of twod when it is declared.

```
int twod[][]=new int[4][];

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

The following program creates a two dimensional array in which the sizes of the second dimension are an unequal.

// Manually allocate differing size second dimensions.

```
class mdarray
{
    public static void main(String args[])
    {
       int twod[][]=new int[4][];
       twod[0]=new int[1];
       twod[1]=new int[2];
       twod[2]=new int[3];
       twod[3]=new int[4];
       int i,j,k=0;
       for(i=0;i<4;i++)
       {
              for(j=0;j<i+1;j++)
              {      twod[i][j]=k;
                     k++;
              }
       }
       for(i=0;i<4;i++)
       {
              for(j=0;j<i+1;j++)
              {
    System.out.println(twod[i][j]+" ");
              }
    System.out.println();
       }
    }
}
```

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix.

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {{ 1, 2, 3 },{ 4,5,6 },{ 7,8,9}};
    int i, j;
    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

❖ **3-D Array:** A collection of tables is known as **3-D Array**.

The following program creates a 3by 4 by 5, three dimensional array. It then loads each element with the product of its indexes.

```
// Demonstrate a three-dimensional array.
class threeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
        threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

❖ What is the meaning of String in Java?

String is neither simple type nor it is an array of characters (as are strings in C/C++). String defines an object and full description of it requires an understanding of several object-oriented features. **String** objects have many special features and attributes that make them quite powerful and easy to use.

❖ Why Java does not support Pointer?

Java does not support or allow pointers because it stores the address of another variable in memory. So it would allow Java applets to break the firewall between the Java execution environment and the host computer. Since C/C++ makes extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the boundaries of the execution environment, you will never need to use pointer or would there be any benefit in using one.