Java programs always written inside class definition

**Variables in Java**

A variable is a named memory location that holds the data value of a particular data type. A variable in Java is a kind of container that contains the value during program execution.

Variables are also called symbolic variables because they are named.

A variable is a memory area, and this memory location gets affected when any manipulation or operation is done on the variables.

Each variable has a specific data type, which determines its memory and the type of data that it can store.

**Declaration of a Variable**

To declare the variable, we must specify the data type followed by the unique name of the variable.

**datatype variablename; int n;**

A variable name is an **identifier,** thus all the naming conventions/rules of an identifier must be applied for naming a variable.

### Idenifier

It is a name given by the programmer to identify a variable name, function name, class name, array name etc.

### Naming a n Idenifier

1. A variable name can begin with a special character dollar ($) and underscore ( \_ ).
2. The variable names cannot contain white spaces, for example, long dist ance = 1000; is invalid because the variable name has a space in it.
3. The first letter of a variable cannot be a digit.
4. A variable name should begin with a lowercase letter, for example, int number. For lengthy variable names having more than one word, we can use camelCase, for example, int salaryPerDay; float rateOfInterest; ,etc. are valid.
5. We cannot use keywords like int, for, while, class, etc as a variable name.
6. Variable names are case-sensitive in Java.

## Initialization of a Variable

dataType variableName = value ; variableName = value ;

**double area = 378.87 ;**

**Initial Values of Variables**

There is no default value for local variables, so we have to assign a value to a local variable before its first use. However, each class variable, instance variable or array component is initialized with a default value when it is created:

**Type Initial/Default value**

**byte 0(Zero) of byte type**

**short 0(Zero) of byte type**

**int 0**

**long 0L**

**float 0.0F**

**double 0.0D**

**char null character i.e., ‘¥u0000’**

**boolean false**

**All reference types null**

## Types of Java Variables

Java allows the variables to be declared at any place or within any block. That is, in Java, we can declare the variables at many places, we can declare them either at the start of the program or inside any classes, method/functions or inside the main method.

Scope determines which variables are visible to other parts of your program and also what is the lifetime of those variables. Depending upon the scope, visibility, and access to the variables, they can be classified under 3 categories.

**The 3 types of variables in Java are –**

**Static Variables Instance Variables Local Variables**

**Static Variables in Java**

A variable that is declared inside a class but not inside the method, constructor or a block, with the static keyword is called static or class variable.

Static variables are also called class variables because they are associated with the class and are common for all the instances of the class. That is, one copy of the static variable is shared among all objects of the class.

Static variables are created at the start of program execution and get destroyed automatically after the execution of the program.

Unlike instance variables, there is only one copy of a static variable per class, irrespective of the number of objects we create from the class.

The initialization of static variables is not mandatory. If we do not initialize it with a value, it gets a default value similar to the instance variables.

Static variables can be declared as public/private, final, and static. These variables are the constant variables that never change from their initial values.

Static memory allocation is very helpful to store the static/class variables. We can access the static variables by calling it with the class name, that is

**ClassName.variableName**. We can also directly access the static variables inside static methods

and static blocks.

The visibility of the static variable is similar to instance variables. However, we mostly declare the static variables as public, since they must be available for all objects of the class.

The values of static variables can also be assigned during the declaration or within the **constructor.**

**Also, we can assign the values in special static initializer blocks. class MyClass{**

**static int number; //number is a static variable**

**}**

**Instance Variables in Java**

A variable that is declared inside the class but outside any method, constructor or block is called an instance variable. An instance variable is a non-static variable that is, we can not declare it as static.

It is called an instance variable because its value is instance-specific (related to objects) and is not shared with other instances/objects as each object of the class has its own set of values for these non-static variables.

As we declare an instance variable in a class, these variables are created when an object of the class

is created with the use of a “new” keyword and destroyed when the object is destroyed.

We can access instance variables only by creating objects. Also, we can use these variables using the

“this” pointer inside the same class.

We can also declare instance variables with access specifiers (private, public or default). If we do not explicitly set any access specifier for the instance variables then Java assumes them as default access specifiers.

It is not necessary to initialize an instance variable. If we do not initialize it with a value, it gets a default value. We have already discussed the list of the default values of instance variables in the introductory part of the variables, in this article.

Heap Allocation is used to store the instance variables and a slot of memory is created for each instance variable value. Each object has its own copy of the instance variables that is, these variables cannot be shared among objects.

The instance variables are visible to all methods, constructors, and blocks in the class. Normally, it is

recommended that we should declare the instance with a “private” access specifier.

**Class Circle**

**{**

**//These are instance variables, present inside the class double radius;**

**double circleArea;**

**}**

**Local Variables in Java**

**A local variable in Java is a variable that we declare inside a body of a method, block or a constructor. We can use the local variable only within that method and the other methods of the class are unaware of the existence of this variable.**

**A block starts with an opening curly brace and ends with a closing curly brace. The scope of a local variable is restricted to a particular block. Its lifetime is within the parenthesis in which it is declared.**

**That is, it is created when a function is called or a block is entered and gets destroyed once it exits that method, block or constructor.**

**We cannot define the local variables as static”. We can use the “final” keyword before a local**

**variable.**

**The scope and access of these variables exist only inside the block in which we declare them. We cannot use access modifiers for the local variables.**

**Internally, a stack implementation is used to implement the local variables.**

**Initialization of Local Variable is necessary – there is no default value for local variables, so we should declare a local variable and assign it with an initial value before its first use.**

**If we do not initialize the value of a local variable, then we will get a Compile Time Error.**

**Ex1**

**for(int i = 10 ;i >= 1; i--)**

**{**

**// Body of for loop**

**}**

**Ex2**

**class ClassName**

**{**

**methodName(){**

**<DataType> localVariableName; localVariableName = value;**

**}**

**}**

**Data Type in Java**

**It specifies the range of values and its allowed operations.**

It defines how the values of that data type are stored in memory and what operations can be performed on the data.

Data type tells the compiler or interpreter how the programmer aims to use the data.

**There are majorly two types of languages:**

**statically typed language :** data type of each variable has to be defined during the compile time.

That is, we have to declare the type of the variable before we can use it.

Once we declare a variable of a specific data type, then we cannot change its data type again. However, they can be converted to other types by using explicit type casting

C, C++, C#, Java,

**dynamically typed language**: the data types can change with respect to time and the variables are checked during run-time.

Ruby, Python, Erlang, Perl, VB, and PHP.

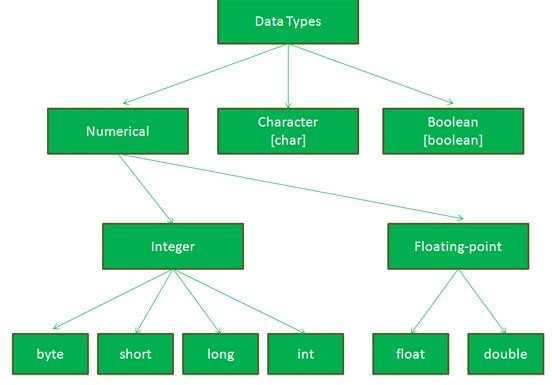
There are two categories of data types in Java: Primitive Data Types

Non-Primitive DataTypes

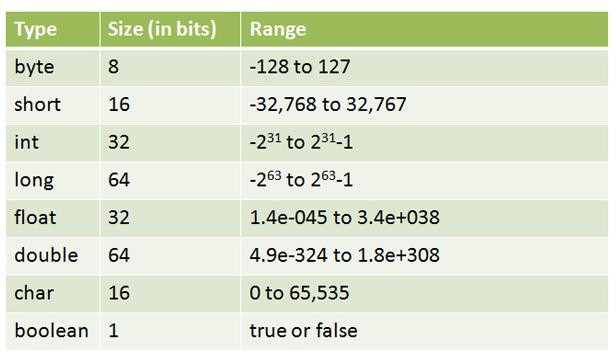
1. Primitive Data Types in Java

Primitive types are the most basic data types available in Java. There are 8 primitive data types in Java:

**byte, char, short, int, long, float, double and boolean.**



**Memory taken for each datatype(size)**



**How to calclate range of values:**

**If n is the number of bits**

**then range= -2n-1 - 0 - +2n-1-1**

**byte:** The smallest integer type available in Java is byte. Its size is 8 bits and can store values within the range -128 to 127. byte data type can be useful while working with a stream of data over a network or a file.

byte a, b;

**short:** Perhaps the least used integer type in Java is short. Its size is 16 bits and it can store values within the range -32,768 to 32,767.

short s;

**int:** Most widely used type for working with integers in Java programs is int. Its size is 32 bits and it can store values within the range -2 billion to +2 billion. Variables of type int are frequently used in loops and for indexing arrays.

int i, j;

**long:** The size of long integer type is 64 bits and can store up to quite a large range of integer values. It is generally used in programs which work with large integer values.

long a, b;

**float:** The type float is used to specify a single-precision value that uses 32 bits for storage. Single precision is faster on some processors and takes half as much space as double precision. The type float is used to work with fractional values where the precision is not that important.

float f;

**double:** The type double is used to specify a double-precision value that uses 64 bits of storage. Double precision is faster on most of the modern processors which are optimized for mathematical calculations. The type double is used to work with larger fractional values and when the precision of the fractional value is important.

double d;

**char:** The data type which allows us to store characters is the char type. Unlike C and C++, the size of char type in Java is 16 bits. Java uses Unicode to represent characters. **Unicode** supports most of the international languages, whereas, C and C++ only supports ASCII.

Unicode is an encoding systen which represents every character in the world in numeric format ,which is represented in 2 bytes. Unicode system is developed by unicode consortium.ASCII values are also in unicode system.

The char type can also be used to store integer values from 0 to 65,535. Operators allowed on integer types are also allowed on char type. '\u0000'

char ch;

**boolean:** The boolean type in Java is used to store a logical value, either true or false. Size of boolean is 1 bit. Variables of the type boolean are used in control statements extensively.

boolean b;

**Literal**

These are the data value used inside the program. 56 :integer literal

7.9d double

8.6f float

'v' character int n=69; "ggg" string

true boolean "\n"

1. Integer literals
2. Floating literals
3. Boolean literals
4. Character literals
5. String literals
6. Null literal

**Integer Literals are further divided into three categories:**

1. **Decimal Integer Literals**

int x1 = 100; int x2 = -467;

#### Octal Integer Literals

int x1 = 0765; //Octal constant

int x2 = 0987; //Invalid as it contains 8 and 9

#### HexaDecimal Integer Literals

int x1 = 0x54; //Hexadecimal integer

int x2 = -0X6BF8; //Valid negative hexadecimal literal

**Floating literals in Java**

Floating literals are the literals that have fractional parts and decimal points in them. They are also known as the real literals. We can write to them in two forms-either in fractional form or in exponential form.

2.0f, 16.7, -13.0, -0.00987

Every floating type is a double type and this is the reason why we cannot assign it directly to the float variable, to escape this situation we use f or F as a suffix, and for double we use d or D.

**Boolean Literals in Java**

The Boolean literals have only two values- true and false. boolean b1 = true;

boolean b2 = false;

**Java Character literals**

A character literal represents a single character that is enclosed in a single quote ‘’, as in ‘Z’. The rule for

writing a character literal is that it must contain a single character enclosed within a single quote.

char ch = ‘A’;

ch = ‘065’; is valid.

Java allows us to have certain non-graphic characters as character literals. Nongraphic characters are the characters that we can’t directly from the keyboard;

\n

\b

\t

**Java String Literals**

The String literals represent multiple characters. A sequence of zero or more characters written within double quotes ” ” is known as a string. Each character in a string literal can also represent an escape sequence.

String a = “Abc”

**Null Literals**

The Null literals are the values that represent null values. The null literals have only a single value (null). It shows the null reference. A null literal is of the null type.

Constant

final datatype constantname=value; final double PI=3.141;

once the value is defined we can not change the value.

# Operators in Java

Operator in Java is a symbol which is used to perform operations.

# Unary Operator

The Java unary operators require only one operand.

**++ , --. ~ and !**

**++c; c++;**

# Arithmetic Operators

**+. -. \* , /, %**

# Relational operator

**<,>,<=,>=.==,!=**

# logical operator

**&&,||,!**

# Bitwise operator

**&,|,^**

# Shift operator

**<<,>>, >>>(unsigned right shif)**

# compound operator +=,-=...

**a+=10;**

# instanceof

# used to test whether an object is an instance of a class or not. It always returns boolean value.

# System.out.println(objectname instanceof classname);

**Unsigned Right Shift Operator (>>>)**

It shifts a zero at the leftmost position and fills 0. It is denoted by the symbol >>>. Note that the leftmost position after >> depends on the sign bit. It does not preserve the sign bit.

Example: If a=11110000 and b=2, find a>>>b?

a >>> b = 11110000 >>> 2 = 00111100

The left operand value is moved right by the number of bits specified by the right operand and the shifted bits are filled up with zeros. Excess bits shifted off to the right are discarded.

# Control Statement

if(condition){

//code to be executed

}

# if-else

if(condition){

//code if condition is true

}else{

//code if condition is false

}

# if-else-if

if(condition1){

//code to be executed if condition1 is true

}else if(condition2){

//code to be executed if condition2 is true

}

else if(condition3){

//code to be executed if condition3 is true

}

...

else{

//code to be executed if all the conditions are false

}

**switch-case** switch(expression){ case value1:

//code to be executed; break; //optional

case value2:

//code to be executed;

break; //optional

......

default:

code to be executed if all cases are not matched;

}

# for loop

for(initialization;condition;incr/decr){

//statement or code to be executed

}

# while loop

while(condition){

//code to be executed

}

do-while do{

//code to be executed

}while(condition);

break; continue;

# for-each loop

for(datatype loopcounter: collectionname)

{

}

System.out.println(-5&2); 00000000000000000000000000000101

1's 11111111111111111111111111111010

2's=1's+1

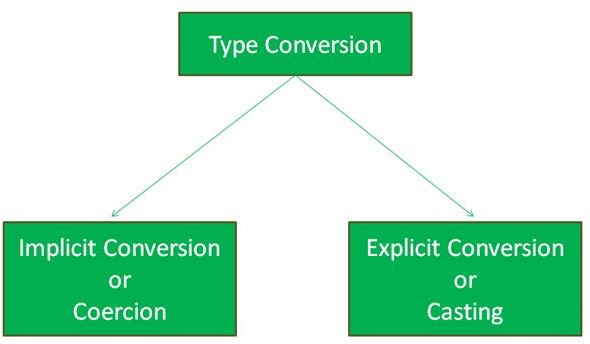
111111111111111111111111111111011

000000000000000000000000000000010

System.out.println(5%-2);

**Type conversion and casting**

It is the process of converting value of one datatype to another datatype.



There are two types of type conversion

### Implicit Casting or Coercion

This type of conversion is performed automatically by Java with the constraint that The source and destination types must be compatible with each other.

The size of the destination type must be larger than the source type.

Since a smaller range type is converted into a larger range type this conversion is also known as

**widening a type. int i=8;**

**long l=i;**

### Explicit typeCasting

There may be situations where you want to convert a value having a type of size less than the destination type size. programmer does this manually.

It is also known as **narrowing a type**. There is some loss of values.

**(destination-type) value int a = 10;**

**byte b = (byte) a; or**

**float a=4.8; int b=(int)a;**

**Note: boolean values are never typecasted.ie boolean value can not be converted to other datatype.**

**boolean b=true;**

**int b1=(int)b; //not possible**

### Type promotion rules of Java for expressions

**All char, short and byte values are automatically promoted to int type.**

**If at least one operand in an expression is a long type, then the entire expression will be promoted to long.**

**If at least one operand in an expression is a float type, then the entire expression will be promoted to float.**

**If at least one operand in an expression is a double type, then the entire expression will be promoted to double.**

**class Sample**

**{**

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

**{**

**int i = 1000000; char c = 'z'; short s = 200; byte b = 120; float f = 3.45f;**

**double d = 1.6789;**

**double result = (f \* b) + (i / c) - (d \* s); System.out.println("Result = "+result);**

**}**

**}**

**The datatype of result is of double type.**

**Another two types of type castings are there**

**object type to primitive type conversion // wrapper class methods are used primitive type to object type conversion**

**Wrapper class**

**int Integer**

**short Short boolean Boolean char Character**

**double Double**

### Array in java

Array is the collection of homogeneous data in **contigeous memory location** using same variable name.

Individual elements of an array are referred using an index value or also known as subscript.

The index value of an array always starts from zero.

Types of Array

one-dimensional two-dimensional multi-dimensional

**One-dimensional Array** Declation of array datatype[] arrayname; int[] num;

int n[]; or

datatype arrayname[]; Define the array

Allocating memory for the array elements. array-name = new data-type[size]; num=new int[5];

Both declaration and definition in one step

### int[] num=new int[5];

**num is reference variable. Reference variable holds the address of a data. num holds address of num[0] or base address**

Initializing Array elements num[0]=10;

num[1]=20; num[2]=20; num[3]=276;

num[4]=45;

Note: By default, when an array is created, all the elements of numeric type are initialized to zero, all elements of boolean type are initialized to false, all elements of type char are initialized to ‘¥u0000’ and all reference types are initialized to null.

Array Initializer

Array declaration,definition and initialization will be in the same line. It is also known as static initialization.

int[] num={10,20,30,40,40};

length is an attribute of array which displays the size of the array.

Traversing an Array using for loop using for-each loop

int[] num={34,76,98,33};

for(int i=0;i<4;i++) System.out.println(num[3]);

for(int i:num) System.out.println(i);

**Two-dimensional array**

* **Declaration** int[][] num;
* **Define**

num=new int[rowsize][colsize];

* **Initialization**

int[][] mat=new int[2][2]; mat[0][0]=12;

mat[0][1]=16;

mat[1][0]=15;

mat[1][1]=18;

Array Initializer

int[][] mat={{12,45},{45,87}};

Traversing an Array class Arraydemo1

{

public static void main(String[] s)

{

int[][] mat=new int[2][2]; mat[0][0]=12;

mat[0][1]=16;

mat[1][0]=17;

mat[1][1]=10;

for(int i=0;i<mat.length;i++) //mat.length is the row size

{

for(int j=0;j<mat[0].length;j++) //mat[0].length is the column size

{ System.out.print(mat[i][j]+" ");

}

System.out.println();

}

//using for each loop

for(int[] r:mat)

{

for(int c:r)

{

System.out.print(c+” “);

}

System.out.println();

}

}

}

**ArrayIndexOutOfBoundsException**

**Ragged Array**

**or**

**Jagged Array**

**It is a 2-D array ,where each row contains variable length of columns 45 67 76**

**23 56**

**78 98 56**

**45**

## Declaration and Initialization of Jagged Array

**int[][] num=new int[4][]; num[0]=new int[3]; num[1]=new int[2]; num[2]=new int[3];**

**num[3]=new int[1];**

There are other ways to declare and initialize jagged arrays. Let us take a look at the other ways.

**int arr[][] = new int[][]{**

**new int[] { 1, 2, 3, 4 },**

**new int[] { 4, 5},**

**new int[] { 6, 7, 8},**

**};**

**or**

**int arr[][] ={**

**new int[] { 1, 2, 3, 4 },**

**new int[] { 4, 5},**

**new int[] { 6, 7, 8},**

**};**

**or**

**int arr[][] ={**

**{ 1, 2, 3, 4 },**

**{ 4, 5},**

**{ 6, 7, 8},**

**};**

The jagged array is stored in heap memory and each individual element of this jagged array is a one-dimensional array. Each 1-D array has a different size.

**import java.util.\*;**

**import java.io.\*;**

**public class testjag {**

**public static void main(String[] s) {**

**// Declaring a 2-D array with 3 rows**

**int arr[][] = new int[3][];**

**// create a jagged array**

**arr[0] = new int[]{99,100,101};**

**arr[1] = new int[]{199,200};**

**arr[2] = new int[]{299,300,301,302,303};**

**// Displaying the elements of 2-D Jagged array**

**System.out.println("Elements of 2-D Jagged Array");**

**for (int i = 0; i < arr.length; i++) {**

**for (int j = 0; j < arr[i].length; j++)**

**System.out.print(arr[i][j] + " ");**

**System.out.println();**

**}**

**}**

**}**

**Advantages**

* Jagged arrays make storage easy as it provides a variable number of columns.
* The performance of a program can be improved as here we don't need to store unwanted elements and have flexibility over the number of elements in the row.
* Jagged arrays have a greater speed than multidimensional arrays and single-dimensional arrays as traversal is faster in jagged arrays.

num is a reference variable

**num points to num[0] num[0] points to num[0][0]**

**copy one array to another array class copy**

**{**

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

**{**

**int[] num1={10,20,30,40,40};**

**int[] n=new int[num1.length];**

**for(int i=0;i<num1.length;i++) n[i]=num1[i];**

**for(int i:n) System.out.println(i);**

**//copy using predefined method arracopy System.arraycopy(num1,0,n,0,4);**

**for(int i:n) System.out.println(i);**

**// copy using clone() method int[] n2=n.clone();**

**for(int i:n2) System.out.println(i);**

**}**

**}**