**Language Fundamentals**

**Identifiers:**

A name in java program is called identifier, which can be used for identification purpose. It can be method name, variable name, class name.

Ex: How many identifier are there in the program

Class Main{

Public static void main(String[] args){

int x=10;

}

}

Ans: 5 Identifiers

1. Main (class Name)
2. main (method name)
3. String (class name)
4. args (parameter name)
5. x (variable name)

**Rules for defining java Identifiers:**

1. The only allowed characters in java identifiers are a-z, A-Z, 0-9, &, \_ If we are using any other character we will get compile time error

Ex: total number

1. Identifiers can’t start with digit. Ex: total123 valid

123total In-valid

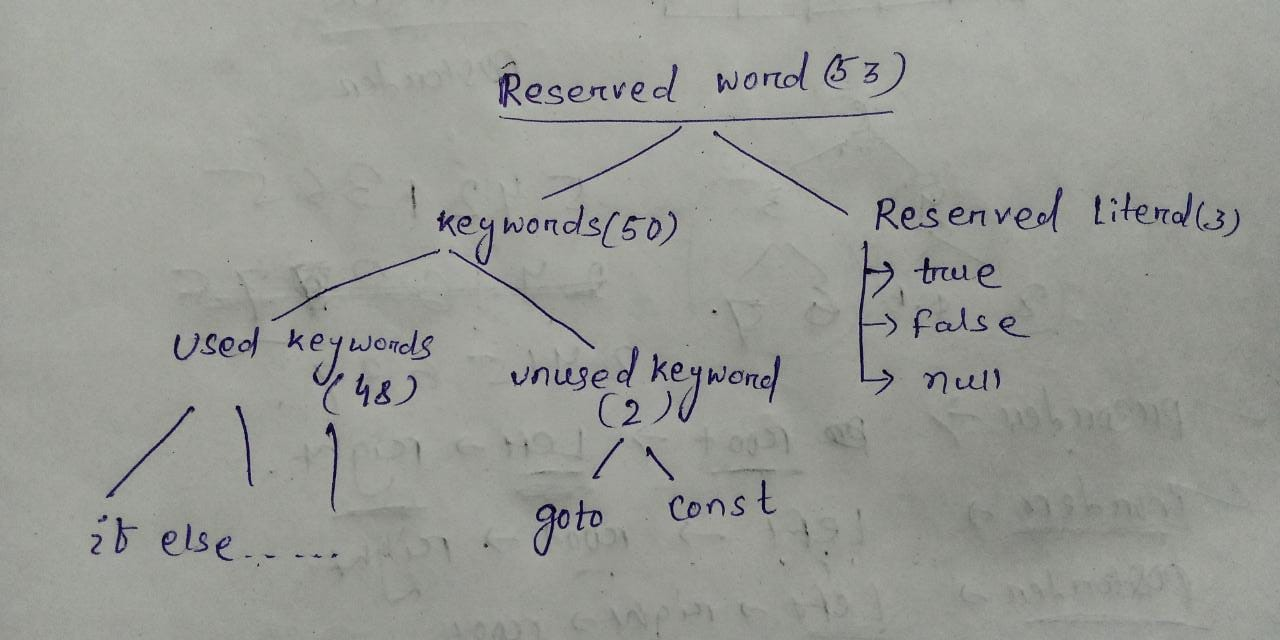
1. Java identifiers are case sensitive of course java language itself is a treated as case sensitive programming language.
2. There is no length limit for java identifier. But it is not recommended to take too lengthy identifier.
3. We can’t use reserved as identifier. Ex: int if = 20; not valid
4. All predefined java class name and interface name we can use as identifier. Ex: int String =2334; valid

Int Runnable = 234;

Even though it is valid but it is not a good programming practice because it reduces readability of code and create confusion.

**Reserved word:**

In java some words are reserved to represent some meaning and functionality. Such type of words are called reserved word.



Keywords for data types:

1. Byte
2. Short
3. Int
4. Long
5. Char
6. Float
7. Double
8. boolean

Keywords for flow control

1. if
2. else
3. switch
4. case
5. default
6. while
7. do
8. for
9. break
10. continue
11. return

Keywords for Modifiers:

1. public
2. private
3. protected
4. static
5. final
6. abstract
7. synchronized
8. native
9. strict
10. transient
11. volatile
12. default (like public private protected)

Keywords for Exception Handling

1. try
2. catch
3. throw
4. finally
5. throws
6. assert

class related keywords:

1. class
2. interface
3. extends
4. implements
5. package
6. import

Object related Keywords:

1. new
2. super
3. this
4. instanceof

Void return type keyword:

1. void – in java return type is mandatory. If a method is not returning any thing we have to take return type void. But in c default return type is int.

**goto**: usage of goto created several problem in old languages so sun people banned goto in java.

**const:** use final instead of const.

**Note:** goto and const are unused keywords and if we are trying to use we will get comile time error.

**Reserved Literal:**  true, false for Boolean data types. null- default value for object reference.

**enum:** we can use enum to group of named constants.

Ex: enum month{

JAN, FAB…

}

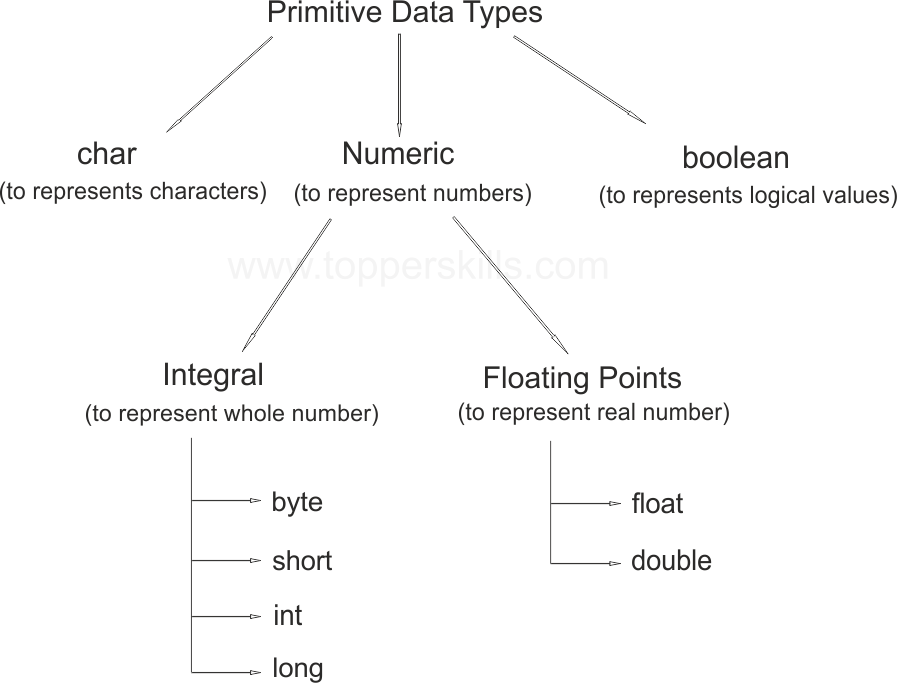
**Conclusion:** All 53 reserved words in java contains only lower case alphabet symbol.

In java we have only new keyword not delete keyword because destruction of useless object is the work of garbage collector.

The following are new keywords in java: strictfp, assert, anum.

* In java every variable and every expression has some type, each and every data type clearly defined. Every expression should be checked for type compatibility. Because of above reason we can conclude java is a strongly typed programming language.
* Java is not considered as pure object oriented programming language because several opp features are not supported by java (operator overloading, multiple inheritances). More over we are depending on primitive data types which are non-objects

**Primitive Data types:**



* Except Boolean and char remaining data types are considered as singed data type because we can give any sign to the value.

**byte:**

size = 1byte = 8 bits

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| \*- sign bit | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

26  25 24 23 22 21 20

Sign bit : 0 🡪 +ve, 1 🡪 -ve 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127

Max Value = +127 Min value = -128

Range: -128 to 127

* The most significant bit acts as sign bit. 0 means positive number and 1 means negative. Positive no is directly represented in memory but negative numbers are represented in 2’-complement form.

Ex: byte b = 128; // CE: possible loss of precision found : int, required : byte

byte b = 10.5; // CE: possible loss of precision found : float, required : byte

byte b = true // CE: incompatible types found : bool required : int

byte b = true // CE: incompatible types found : java.lang.String, required : int

* Byte is the best choice if we want to handel data in the form of streams either from the file or from the network(file supported form or network supported form is byte).

**short:**

This is the most rarely used data type in java.

Size – 2 bytes = 16 bits

Range : -215 to 215-1 [-32768 to 32767]

Ex:

short s = 32768; //CE: possible loss of precision found : int, required : short

short s = true; // CE: incompatible types found : bool required : short

* Short data type is best suitable for 16-bit processor like 8085. But these processor are completely outdated hence corresponding short data type is out dated.

**Int**:

This is the most common java data type. Size = 4 bytes = 32 bits

Range; -231 to 231-1 [-2147483648 to 2147483647]

Ex:

Int x = 2147483648; // CE: integer number too large

Int x = 2147483648L; // CE : : possible loss of precision found : long, required : int

Int x = true; // CE: incompatible types found : bool required : int

**Long**:

Sometimes int may not enough to hold big values then we should go for long type.

Ex: The amount of distance travel by light in 1000 days to hold this value in may not enough so we have to take long value

long l = 126000\*60\*24\*1000;

Ex: The number of characters in a big page may exceeds int type so we have to return the length as **long.**

Size : 8 bytes (64 bits) Range: -263 to 263-1 .

**Note:** All the above data types (byte, short, int, long) meant for representing integral value.

if we want to store decimal values we have to go for floating point data type.

|  |  |
| --- | --- |
| float | double |
| 1. If we want to 5 to 6 decimal places of accuracy then we should go for float. 2. Float follow single precision. 3. Size: 4 bytes 4. Range: -34e38 to 34e38 | 1. If we want 14 to 15 decimal places of accuracy then we should go for double. 2. double follow double precision. 3. Size: 8 bytes 4. -1.7e38 to 1.7e38 |

**boolean :**

size: not applicable [virtual machine dependent]

Range: not applicable [allowed values are true or false]

boolean b = true;

boolean b = false;

boolean b = True; // CE: can’t find symbol, symbol : variable true location : class test

**char**:

size: 2 bytes

Old languages like c/c++ are ASCII code based and the number of different allowed character are less than 256 to represent these 256 character 8 bits are enough. Hence the size of java in old languages is 8 bits.

Java is Unicode based and the number of different 256 and less than equal to 65536. To represent these character 8 bits is may not enough. Hence the size of char in java 2 bytes.

Range: 0 to 65536.

**Note: null** is the default value for object reference and we can’t use in primitive data type otherwise it will give compile time error.

**Literal:**

A constant which can be assigned to the variable is called literal.

Int x = 10;

Here 10 is the constant literal.

**Integral Literal:** for integral data types(byte, short ,int ,long) we can specify literal value in following base.

**Decimal Literal:** (Base 10) allowed digits 0 – 9. Int s = 10;

**Octal form:** base – 8. Allowed digit 0-7.

Literal value should be prefixed with 0.

Int x = 021;

**Hexadecimal form:** base 16. Allowed digit 0 to 9 and a-f.

Int x = 0X23;

For extra digits (a to f) we can use both lower case and upper case . These are very few area where java is not case sensitive. Literal value should be prefixed with 0X or 0x.

These are only possible ways to specify literal value for integral data types.

Q. int x = 0786; // CE : integer number is too large (octal allowed number 0-7)

Q2. Class test{

P s v m(){

int x = 10 ; int y = 010; int z = 0x10;

SOP(x+” ”+ y+ “ ”+z);

}

}

o/p-> 10 8 16

Note: By default every integral is of int type but we can specify as long by suffixed with small l or capital L.

**Float literal:** By default every floating point literal is of double type hence we can’t directly to the float variable. But we can floating point literal as float type by suffixed with small f or capital f.

Flat f= 123.456f;

Double d = 123.456;

We can specify floating point literals only in decimal form and we can’t specify in octal and hexadecimal forms.

Double d = 123.456;

Double d = 0123.456;

Double d = 0x123.345; //CE: malformed floating point literal

We can assign integral literal directly to floating point variable and that integral literal can be specified either in decimal or octal or hexadecimal form.

double d = 0786; CE: integral literal is too large

double d = 0xFace;

double d = 0786.0;

double d = 0xFace.0;

double d = 10;

double d = 0777;

we can specify floating point literal even in exponential form within bracket (scientific notation ).

double d = 1.2e3; 1.2e3 =1.2 x 103 = 1200

**Boolean Literal:**  The only possible value are true or false.

**Char literal:**  We can specify char literal as within sing quotes. Char ch = ‘A’;

char ch =a; //CE: can’t find symbol symbol: variable

char ch = “a”; //CE: Incompatible type

char ch = ‘ab’; //CE: unclosed char literal / not a statement

we can specify char literal as integral literal which represents Unicode value if the character. And the integral letter either in decimal or octal or hexadecimal form but the allowed range is 0 to 65536.

Char ch =97;

Sop(ch); // o/p:- a

Char ch = 0xFace; // valid

Char ch = 0777;

Char ch = 65535;

Char ch = 65536; // CE: possible loss of precision fount : int required : char

Note: we can represent char literal in Unicode representation which is nothing but **‘\uXXXX’** (xxxx – 4 digit hexadecimal number).

Ex: char ch = ‘\u0061’;

Sop(ch); // o/p-> a

Every escape character is a valid char literal.

Ex: char ch = ‘\n’;

Char ch = ‘\t’;

|  |  |
| --- | --- |
| Escape character | Description |
| * **\n** * **\t** * \r * **\b** * **\f** * **\’** * **\”** * **\\** | * **New line** * **Horizontal tab** * **Carriage return** * **Back space** * **Form feed** * **Single Quote** * **Double Quote** * **Back Slash** |

Ex: char c = 0xBeer; **X**

char c = \uface; **X**

char c = ‘\uface’; **valid**

char ch = ‘\m’ ; **X**

char ch = ‘\iface’ ; **x**

**String Literal:** Any sequence of characters within double quotes is treated as string literal.

**Ex:** “Durga”;

* 1. **enhancement with Literals:**

1. Binary Literal:

For integral data until 1.6 version we can specify literal value in the following base decimal octal hexadecimal but from 1.7 version onwards we can specify in binary form also. Literal values are 1 and 0. Literal value should be prefixed with **0b** or **0B.**

Ex: int x = 0x1111;

Sop(x); // 15 in decimal form

1. Usage of underscore Symbol in Numeric Literal:

From 1.7 version onwards we can use underscore symbol between digits of numeric literal.

Ex: double d = 1\_23\_456.7\_8\_9; // it is similar to 123456.789

double d = 123\_456.7\_8\_9;

Both above are same.

The main approach of this enhancement is readability of the code will increase.

At the timer of compilation these underscore symbols will be removed automatically hence after compilation the above line will become 123456.789

We can use more than one underscore symbols between the digits .

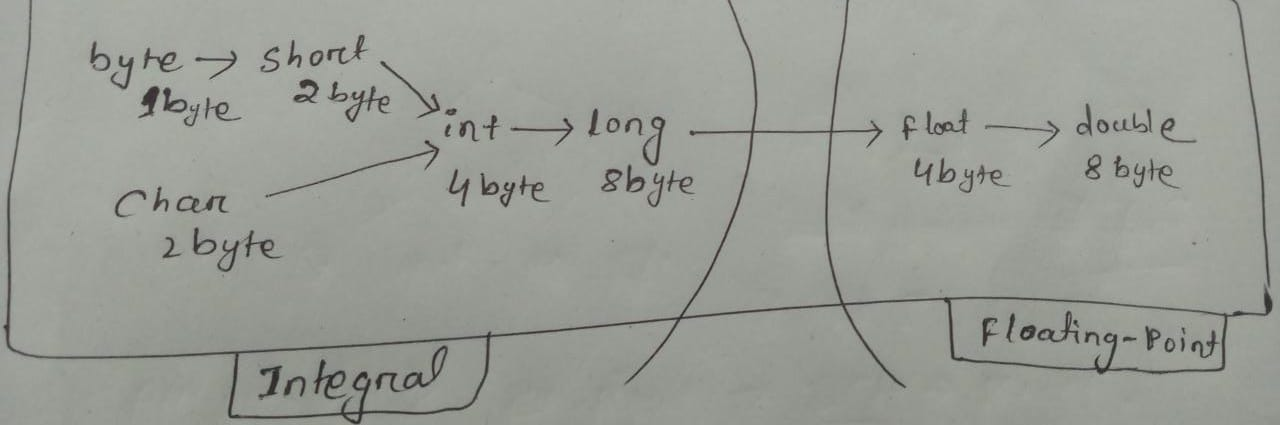
Ex: double d = 1\_\_\_2\_345\_\_6.7\_\_\_\_\_\_89;

We can use underscore symbol only between the digits if we are using any where else we will get compile time error.

Ex: double d = \_1\_2\_3456.\_\_678;

double d = 123\_456.\_348\_\_;

the above line will give compile time error.



Note : 8 byte long value we can assign to 4 byte float variable because both are following different memory representation internally.

float f =10l; // valid

***Array:***

1. Introduction 7. Anonymous Array
2. Array declaration 8. Array Element assignment
3. Array creation 9. Array Variable assignment
4. Array Initialization
5. Array Declaration, creation & Initialization in a single line
6. length vs length()

**Introduction:**

An array is an indexed collection of fixed number of homogenous data elements. The main advantages of array is we can represent huge number of values by using single variable so that readability of the code will be improved.

But the main disadvantage of array is fixed in size i.e. once the array is created we can’t change the size of the array. Hence to use array concept compulsory we should know the size in advance, which may not possible always.

**Array Declaration:**

**1-D Array Declaration:**

**Int[] x;** // Recommended because name is clearly separated for type.

Int []x;

Int x[];

At the time of declaration we can’t specify the size otherwise we will get compile time error. Ex: int[6] x; // Invalid

**2-D Array Declaration:**

int[][] x;

int [][]x;

int x[][];

int[] []x;

int[] x[];

int []x[];

These all above declaration are valid.

Ex: Which of the following are valid

int[] a, b; //a=1 b=1

int[] a[],b; // a=2 b=1

int[] a[],b[]; // a = 2 b =2

int[] []a, b; // a = 2 b = 1

int[] []a, b[]; //a=2 b = 3

int[] []a, []b; // Compile time Error

Note: if we want to specify dimension before the variable the facility is applicable only for first variable in a declaration. If we are trying to apply for remaining variable we will get compile time error.

**int[] [] a, []b , []c;**

valid In-valid In-valid

**3-D Array Initialization:**

int[][][] a;

int [][][]a;

int a[][][];

int[] [][]a;

int[] a[][];

int[] []a[];

int[][] []a;

int[][] a[];

int [][]a[];

int []a[][];

**Array Creation:**

int[] a = new int[4];

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Every array in java is an object only hence we can create array using new operator.

For every array type corresponding classes are available and these classes are part java language and not available to the programmer level.

To get the class of any array type we cant use method like **a.getClass().getName()**.

Sop(a.getClass().getNmae()); // [I

|  |  |
| --- | --- |
| **Array type** | **Corresponding class Name** |
| int[] | [I |
| int[][] | [[I |
| double[] | [D |
| short | [S |
| byte[] | [B |
| boolean[] | [Z |
| .  .  . | .  .  . |

At the time of array creation compulsory we should specify the size otherwise we will get compile time error.

Int[] x = new int[]; //compile time error

Int[] x = new int[7]; // Valid

* It is legal to have an array with size 0 in java. Ex: int[] x = new int[0];
* If we are trying to create array size with some negative int value then we will get Run time Exception saying **NegetiveArraySizeException.**
* To specify array size the allowed data types are byte, short, char, int . if we are trying to take any other type we will get compile time error.

Ex: int[] x = new int [10] ;

int b = new int[‘a’];

byte b = 20;

int x = new int[b];

short s = 13;

int[] x = new int[s];

int[] x = new int[10l]; // C E: possible loss of precision found : long required : int

**Note:**  The maximum array size allowed in java is 2147483647 which is the maximum value of int data.

Ex: int[] x = new int[**2147483647**];

Int[] x = new int[2147483648]; // C E: integer number is too large

Even in the first case we may get runtime exception if sufficient heap memory is not available.

**2-D array creation:**

In java 2-D array not implemented in matrix representation. Sun people followed array of arrays for multidimensional array creation. The main advantages of the approach is memory utilization will be improved.

|  |  |
| --- | --- |
|  |  |

Ex-1:

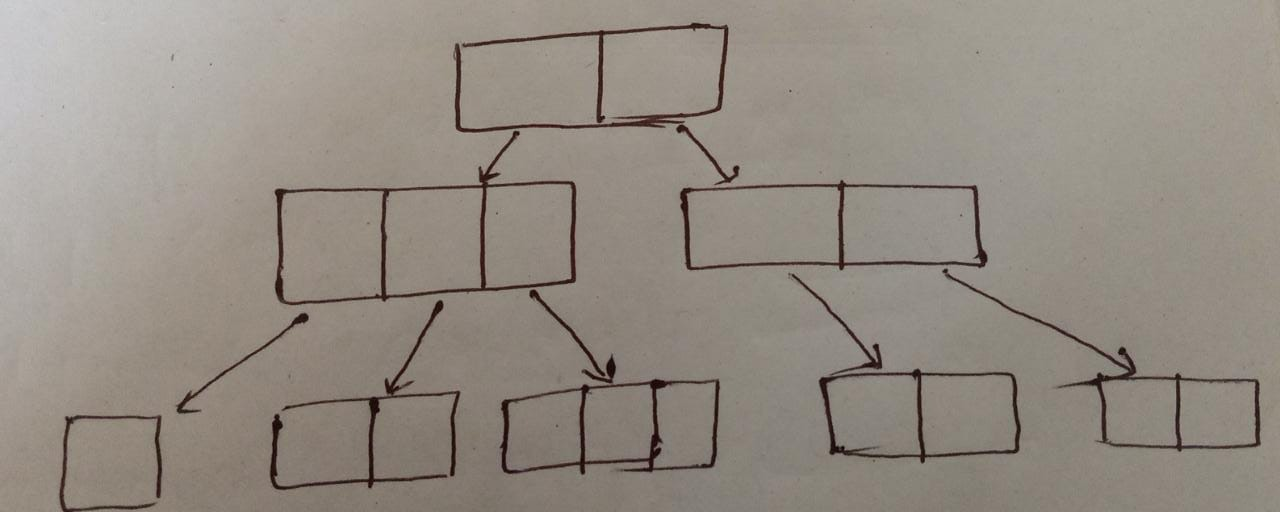
Int[][] x = new int[2][];

|  |  |
| --- | --- |
|  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |

X[0] = new int[2];

X[1] = new int[3];

Ex-2: 

**Code:**

int[][][] x = new int[2][][];

x[0] = new int[3][];

x[1] = new int[2][2];

x[0][0] = new int[1];

x[0][1] = new int[2];

x[0][2] = new int[3];

Q. which of the following array declaration are valid

int[] a = new int[]; // Invalid

int[] a = new int[3]; //valid

int[][] a = new int[][]; //Invalid

int[][] a = new int[3][]; //valid

int[][] a = new int[][4]; //Invalid

int[][] a = new int[3][4]; // valid

int[][][] a = new int[3][4][5]; //valid

int[][][] a = new int[3][4][]; //valid

int[][][] a = new int[3][][5]; //Invalid

int[][][] a = new int[][4][5]; //invalid

**Array Initialization:**

Once we creates an array every element by default initialized with default values.

int[] a = new int[3];

sop(a);

sop(a[0]);

when ever we are trying to print any reference variable internally toString() will be called which is implemented by default to return the string in the following form.

classname@hashCode\_in\_hexadecimalForm

int[][] x = new int[2][3];

System.out.println(x);

System.out.println(x[0]); System.out.println(x.toString());

o/p:-

[[I@71dac704

[I@123772c4

[[I@71dac704

Ex: 3

int[][] x = new int[2][];

System.out.println(x);

System.out.println(x[0]);

System.out.println(x[0][0]);

O/P: [[I@71dac704

null

Exception in thread "main" java.lang.NullPointerException

at Main.main(Main.java:6)

Once we creates an array every array by default initialized with default values if we are not satisfied with these values we can override these values with are customized value.

**Array Declaration Creation and Initialization in a single:**

int[][][] x = {{{10, 20, 30}, {60, 70, 80} }, { {70, 80}, {90, 100, 110}}};

Sop(x[0][1][2]); // 60

Sop(x[1][0][1]); // 80

Sop(x[2][0][0]); // Run time Exception

Sop(x[1][2][0]); // Runtime Exception

Sop(x[1][1][1]); // 100

Sop(x[2][1][0]); // Runtime Exception

**length vs length()**

Length:

length is a final variable applicable for arrays. Length variable represents size of array.

Ex: int[ ] x = new int[6];

Sop(x.length);

O/p:- 6

Sop(x.length()); // CE: cant find the symbol length

Length()

It is a final method applicable for string object. It returns number of character present in the String.

String s = “Durga”;

Sop(s.length); // CE: can’t find Symbol length

Sop(s.length()); // 5

**Note**: length variable applicable for arrays not for String object. Similarly length() method are applicable for String object not for array.

Note: In multidimensional arrays length variable represents only base size but not total size.

Int[][] x = new int[6][3];

Sop(x.length); // 6

Sop(x[0].length); // 3

There is no direct way to find total length of multidimensional array but indirectly we can find by follow x[0].length + x[1].length + x[2].length + ………

**Anonymous Array:**

Sometimes we can declare an array without name. Such type of nameless array are called as anonymous array. The main purpose of anonymous array is just for instance use.

We can create anonymous array as follows:

Syntax: new int[]{10,20,30,40}

While creating anonymous array we can’t specify the size otherwise we will get compile time error.

Ex: new int[3]{10, 20, 30} // CE :

new int[]{10, 20, 30};

We can create multidimensional array in similar way also.

Based on our requirement we can give the name for anonymous array then it is no longer anonymous.

int x = new int[]{12,34,56};

Ex: public class Main{

public static void main(String[] args){

sum(new int[]{10,20,30});

}

public static void sum(int[] x){

int total = 0;

for(int x1 : x ){

total = total + x1;

}

System.out.println("Sum = "+ total);

}

}

In the above example just to call sum method we required an array but after completing call sum method we require no more so for one time use anonymous array is the best choice.

In the case of Object type arrays as array elements we can provide either declared type object or child class object.

Ex: 1

Object[ ] a = new Object[10];

a[0] = new Object();

a[1] = new String(“durga”);

a[2] = new Integer(10);

Ex: 2

Number() n = new Number[10];

n[0] = new Integer(10);

n[1] = new Double(15.7);

n[2] = new String(“durga”); // Compile time error because String class is not in Number Class

For interface as array elements its implementation class are allowed.

Runnable[ ] r = new Runnable[10];

r[0] = new Thread();

r[1] = new String(“durga”); // CE: found java.lang.String required: java.lang.Runnable

Array Variable assignments:

Case 1:

Element level promotion are not applicable at array level.

Ex: char element can be promoted to int type but char array can’t be promoted to int array.

Int[] x = {10,20,30,40};

Char ch = new char{‘a’,’e’,’i’,’o’,’u’};

Int[] b = x;

Int[] c = ch; // In valid compile time error we will get

* Which of the following promotion performed automatically?

Char -> int // Valid

Char[] -> int[] //In- Valid

Int -> double // Valid

Int[] -> double[] //In- Valid

Float -> int // In-Valid

Float[] -> int[] // In-Valid

String -> Object // Valid

String -> Object[] // Valid

But in the case of Object type array child class type array can be promoted parent class type array.

String s = {“A”, “B”, “C”};

Object() a = s;

* Whenever we are assigning one array to another array internal elements won’t be copied just reference variable are assigned.

Int[] a = {10, 20, 30, 40};

Int[] b = {10, 20};

a=b;

b=a;

* Whenever we are assigning one array to another array the dimensions must be matched. For example in the place of one dimensional int array we should provide one dimensional array only if we are trying any other dimension then we will get compile time error.

int[][] a = new int[3][];

a[0] = new int[4][3]; // Compile time error

a[0] = 10; // Compile time error

a[0] = new int[8]; // Valid

**Note**: Whenever we are assigning one array to another array both dimension and type are must be matched but sizes are not required to match.

Ex:

public class Main{

public static void main(String[] args){

String[] argh = {"x", "y", "z"};

args = argh;

for(String s : args){

System.out.println(s);

}

}

}

O/p: For every input output will be x y z.

Ex:

int[][] a = new int[4][3]; // 5 objects

a[0] = new int[4]; // 1 objects

a[1] = new int[2]; // 1 objects

a = new int[3][2]; // 4 objects

Here we created 11 objects. And 7 objects eligible for Garbage Collection.

**Types of variable:**

Division-1:

Based on type of value represented by a variable all variables are divided into 2 types

1. Primitive variables: These are used to represent primitive values

Ex: int x = 10;

1. Reference variable: can be used to represent referance object.

Student s = new Student();

Division-2:

Based on position of declaration and behaviour all variable are divided into 3 categories.

1. **Instance variable:**

* If the value of a variable is varied from object to object such type of variable are called instance variable.
* For every object a separate copy of variables will be created.
* Instance variable should be declared inside the class directly but outside of any method or block or constructor.
* Instance variable will be created at the time of object creation and destroyed at the time of object destruction. Hence the scope of instance variable is exactly same as object.
* Instance variable will be stored in the heap memory at the part of object.
* We can’t access instance variable directly from static area but we can access by using object reference.
* But we can access instance variable directly from instance area.

Ex: public class Main{

int x = 10;

public static void main(String[] args){

Main m = new Main();

System.out.println(m.x);

System.out.println(x); // CE : non-static variable cant be accessed

m.sum();

}

public void sum(){

System.out.println(x);

}

}

* For instance variable Jvm is always providing default values and we are not required to initialize it explicitly.

Ex:

public class Main{

int x;

double d;

boolean b;

String s;

public static void main(String[] args){

Main m = new Main();

System.out.println(m.x); // 0

System.out.println(m.d); // 0 . 0

System.out.println(m.b); // false

System.out.println(m.s); // null

}

}

* Instances variables are also known as object level variable or attribute.

1. **Static Variable:**

* If the value of a variable is not varied from object to object then it is not recommended to declare variable as instance variable.
* We have to declare such type of variable in class level by using static modifier.
* In case of instance variable for every object a separate copy will be created but in the case of static variable a single copy will be created and shared by every object of the class.
* Static variable should be declared within the class directly but outside of the method or constructor.
* Static variable will be created at the time of class loading and destroyed at the class unloading hence scope of static variable is same as scope of that class.

***Note:***

What is happening when we are running the main class ? java Test<-

1. Start JVM
2. Create and start main thread
3. Locate Test.class file
4. Load Test.class // Static variable are created
5. Execute main method
6. Unload Test.class // Static variable are destroyed
7. Terminate main thread
8. Shut down JVM

* Static variable will be should be stored in method area.
* We can access static variable either by object variable or class name but recommended to use class name. within the class it is not required to use class name and we can access directly.

Ex: public class Main{

static int x = 10;

public static void main(String[] args){

Main m = new Main();

System.out.println(m.x); // 10

System.out.println(Main.x); // 10

System.out.println(x); // 10

}

}

* We can access static variable from both static area and instance area.
* For static variable jvm wiil provide default variable and we are not required to perform initialization explicitly.

Ex: public class Main{

static int x;

static double d;

static boolean b;

static String s;

public static void main(String[] args){

System.out.println(x) ; // 0

System.out.println(d); // 0 . 0

System.out.println(b); // false

System.out.println(s); // null

}

}

* Static variable also known as class variable or field.

1. **Local Variable :**

* Sometimes to meet temporary requirements of the programmer we can a variable inside a method or block or constructor such type variable are called **local / stack / automatic / temporary** variable.
* Local variables will be stored in side stack memory.
* Local variable will be created while executing the block in which we declared it once block execution completed local variables are destroyed. Hence the scope of local variable is the block it is declared.
* For local variable JVM won’t provide default value compulsory we have to perform initialization explicitly before using that variable i.e. if we are not using then it’s not required to perform initialization.

Ex: public class Main{

public static void main(String[] args){

int x;

System.out.println("Hello");

System.out.println(x); // error: variable x might not have been initialized

}

}

* Ex: 1

public class Main{

public static void main(String[] args){

int x;

if(args.length > 0)

x = 10;

System.out.println(x); // error: variable x might not have been initialized

}

}

* Ex: 2

public class Main{

public static void main(String[] args){

int x;

if(args.length > 0)

x = 10;

else

x = 20;

System.out.println(x);

}

}

* It is not recommended to perform initialization inside logical blocks because there is no for execution of these blocks always runtime.
* It is highly recommended to perform initialization for local variables with default values at the time of declaration.
* The only applicable modifier for local variable is **final** by mistake if we are trying to apply any other modifier then we will get compile time error.

Ex: final int x = 0; // valid

* If we are not declaring with any modifier then by default it is **default** but this rule is only applicable for instance and static variable not for local variables

**Conclusion:**

* For instance and static variable JVM will provide default value and we are not required to perform initialization explicitly. But for local variable JVM won’t provide default compulsory we should perform initialization explicitly before using that variable.
* Instance and static variable can be assessed by multiple thread simultaneously and hence these are not thread safe. But in case of local variable for every thread a separate copy will be created hence it is thread safe.
* Every variable in java should be either instance or static or local.
* Every variable in java should be either primitive or reference. Hence various possible combination of variables are java are

instance ------- primitive

instance ------reference

static ---- primitive

static ----- reference

local ----- primitive

local ------ reference

**Uninitialized array:**

1. Instance Level:

Ex: public class Main{

int[] x;

public static void main(String[] args){

Main m = new Main();

System.out.println(m.x);

System.out.println(m.x[0]);

}

}

O/p:-

null

Exception in thread "main" java.lang.NullPointerException

Ex: public class Main{

int[] x = new int[3];

public static void main(String[] args){

Main m = new Main();

System.out.println(m.x);

System.out.println(m.x[0]);

}

}

O/P: -

[I@71dac704

0

1. Static level:

What ever happened for instance variable same thing will happen for static variable.

1. Local level:

Ex:

public class Main{

public static void main(String[] args){

int[] x ;

System.out.println(x);

System.out.println(x[0]);

}

}

error: variable x might not have been initialized

Ex:

public class Main{

public static void main(String[] args){

int[] x = new int[3];

System.out.println(x);

System.out.println(x[0]);

}

}

O/p:

[I@71dac704

0

**Note**: one we create array every array element by default initialized with irrespective of whether it is instance or static or local array.

**var-arg methods:** (variable number of argument methods)

Until 1.4 version we can’t declare a method with variable number of arguments if there is a change in number of argument compulsory we should go for new method. It increases length of the code and reduces readability.

To overcome this problem sun people introduced var-arg method in 1.5 version.

According to this we can declare a method which can take variable number of arguments. Such type of methods are called var-arg methods.

We can declare a var-arg method as follows:

m1(int … x);

we can call this method by passing any number of int values including 0 number.

Ex: m1();

m1(10);

m1(20,30);

m1(10,34,67);

Ex:

public class Main{

public static void m1(int ... x){

System.out.println("Var-arg Method!");

}

public static void main(String[] args){

m1();

m1(10);

m1(10, 20);

m1(10,20,30,40,3);

}

}

O/p:

Var-arg Method !

Var-arg Method !

Var-arg Method !

Var-arg Method !

Internally var-arg parameter will be converted into 1-d array. Hence within the var-arg method we can differentiate the value by index.

Ex:

public class Main{

public static void sum(int ... x){

int total = 0;

for(int x1 : x)

total += x1;

System.out.println("The Sum: "+ total);

}

public static void main(String[] args){

sum();

sum(10);

sum(10, 20);

sum(10,20,30,40,3);

}

}

O/P:

The Sum: 0

The Sum: 10

The Sum: 30

The Sum: 103

**Case 1:**

Which of these following are valid.

Method(int… x) // valid

Method(int …x) //valid

Method(int…x) //valid

Method(int x…) // Invalid

Method(int. x..) // Invalid

Method(int .x..) //Invalid

**Case 2:**

We can mix var-args parameter with normal parameter.

Ex: m1(int x, int … y) // 1(x) parameter is always mandatory

M2(int x, int y, String … s ) // 2(x, y) parameter is always mandatory

**Case 3:**

If we mix normal parameter with var-args parameter then var-arg parameter should the last parameter.

Ex: m1(double … d, String s) // Invalid

M1(String s, double … d) // Valid

Inside var-arg parameter we can take only one var-arg parameter and we can’t take more than one var-arg parameter.

M1(int… x, double … d) // Not Valid

**Case 4:**

Inside a class we can’t declare var-arg method and corresponding 1-d array method simultaneously otherwise we will get compile time error.

Ex:

public class Main{

public static void sum(String... x){

System.out.println("The sum ");

}

public static void sum(String[] x){

System.out.println("The sum ");

}

}

Compile time error: cannot declare both sum(String[]) and sum(String...) in Main

public static void sum(String[] x){

**case 5:**

Ex:

public class Main{

public static void sum(int ... x){

System.out.println("Var-Args Method");

}

public static void sum(int x){

System.out.println("General Method ");

}

public static void main(String[] args){

sum();

sum(10);

sum(10,20);

}

}

O/P:

Var-Args Method

General Method

Var-Args Method

Explanation: in general var-arg method will get least priority i.e. if no other method matched then only var-arg method will get the chance it is exactly similar to default case switch.

**Equivalence between var-arg parameter and 1-d array**:

1. Where ever 1-d array present we can replace with var-arg parameter.

Ex: m1(int[] x) == m1(int … x)

main(String[] args) can be replaced with main(String[] args)

1. Where ever var-arg parameter present we can’t replace with 1-d array.

Ex: m1(int … x) != m1(int[] x)

1. M1(int… x) we can call this method by passing a group of int value and x will become 1-d int array.

M1(int[] … x) we can call this method by passing a group of 1-d int array and x will become 2-d int array.

Ex:

public class Main{

public static void vMethod(int[]... x){

for(int[] x1: x){

for(int x2: x1){

System.out.print(" "+ x2);

} }

}

public static void main(String... args){

int[] a = {10, 20, 30};

int[] b = {40, 50, 60};

vMethod(a, b);

}

}

O/P: - 10 20 30 40 50 60

***Main method:***

Whether class contain main method or not and whether main method is declared according to requirement or not these thing won’t be checked by compiler.

At the runtime JVM is responsible to check these things. If JVM unable to find main then we will get runtime exception saying NoSuchMethodError.

Ex:

class Test{

}

Javac Test.java // no compilation error

Java Test // Runtime exception

At runtime JVM always searches for main method with the following prototype :

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

public :- To call JVM from anywhere.

static :- Without existing object JVM has to call this method.

void :- main( ) method won’t return anything to the JVM.

main :- This is the name which is configured inside JVM.

String[ ] args :- Command line arguments.

The above syntax is very strict and if we perform any change then we will get Runtime Exception saying: NoSuchMethodError

Even though above syntax is very strict the following changes are acceptable:

1. Instead of public static we can take static public that means order of modifiers is not important.
2. We can declare String array in any acceptable forms

Ex:

main(String[ ] args)

main(String [ ]args)

main(String args[ ])

1. Instead of args we can take any valid java identifiers.

Ex: main(String[ ] durga)

1. We can replace String[ ] with var-args parameter

Ex: main(String… args)

1. We can declare main method with the following method
2. **final**
3. **synchronized**
4. **strictfp**

The main method is:

**static final synchronized strictfp public void main(String… sonu)**

{

System.out.println(“Valid Main Method ”);

}

Q1. Which of the following main method declaration are valid?

public static void main(String args) // In-Valid

public static void Main(String[ ] args) //In-valid

public void main(String[ ] args) //In-valid

public static int main(String[ ] args) //In-valid

final synchronized strictof public void main(String[ ] args) //In-valid

final synchronized strictof public static void main(String[ ] args) //valid

public static void main(String… args) // Valid

Q2. Which of the above case we will get compilation error?

We won’t get compile time error any where but except last two cases in remaining we will get runtime exception saying NoSuchMethodError;main

**Case 1**: (main method overloading):

Overloading of the main method is possible but JVM will call String[ ] argument main method only the other overloaded method we have to call explicitly like normal method call

Ex:

public class Main{

public static void main(int[ ] x){

System.out.println(" int[ ] ");

}

public static void main(String[ ] args){

System.out.println(" String[ ] ");

main(new int[0]);

}

}

O/P:-

String[ ]

int[ ]

**case 2:** (Inheritance of main)

Inheritance concept applicable for main method hence while executing child class if child does not contain main method then parent class main() will be executed.

Ex:

public class Main{

public static void main(String[ ] args){

System.out.print(" Parent Main ");

}

}

class c extends Main{

}

o/p:

java Main Parent Main

java c Parent Main

**case 3:** (Method Hiding)

Ex:

public class Main{

public static void main(String[ ] args){

System.out.print(" Parent Main ");

}

}

class c extends Main{

public static void main(String[ ] args){

System.out.print(" Child Main ");

}

}

O/p:-

java Main Parent Main

java c Child Main

It seems overriding concept applicable for main method but it is not overriding it is method over hiding.

**Note:** For main method inheritance and overloading but overriding concept is not applicable. Instead of method overriding method hiding is there.

**1.7 version enhancement with respect to main():**

**->** Until 1.6 version if the class does not contains main method then we get runtime exception saying : NOSuchMethodError : main

But from 1.7 version onwards instead of this we will get more elaborated error information. RE: Error: Main method not found in class Main, please define the main method as:

public static void main(String[] args)

* From 1.7 version main method is mandatory. Hence even though class contains static block it won’t be executed if the class doesn’t contain main method.

Ex: public class Main{

static{

System.out.println("Sratic Block");

}

}

1.6 version : O/P: Static Block

RE: NOSuchMethodError : main

1.7 Version : : Error: Main method not found in class Main, please define the main method as:

public static void main(String[] args)

* Ex:

public class Main{

static{

System.out.println("Sratic Block");

System.exit(0);

}

}

1.6 version: Static Block

1.7 Version : : Error: Main method not found in class Main, please define the main method as:

public static void main(String[] args)

* Ex:

public class Main{

static{

System.out.println("Sratic Block");

}

public static void main(String[ ] args){

System.out.println("Main Method");

}

}

O/p: In both case out put is

Sratic Block

Main Method

**Command Line arguments:**

The arguments which are passing from command prompt are called command line arguments.

With these command line arguments JVM will create an array and by passing that array as argument JVM will call main method.

Ex: Java Test A B C

args[0] args[1] args[2]

The main objective of command line argument is we can customize the behaviour of main method.

**Case 1:**

Ex:

public class Main{

public static void main(String[ ] args){

for(int i = 0;i<=args.length;i++)

System.out.println(args[i]);

}

}

O/P:

java Main a b c

a

b

c

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

java Main 1

1

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

java Main

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

In the above program if we replace i<=args.length with i<args.length

Then we will not get runtime exception.

**Case 2:**

Ex: public class Main{

public static void main(String[ ] args){

String[ ] argh = {"X", "Y" , "Z"};

args = argh;

for(String s : args)

System.out.println(s);

}

}

What ever may be the input the output will be

X

Y

Z

Here args is pointing to argh by args = argh; line

**Case 3:**

Within main() method command line arguments are available in String form

public class Main{

public static void main(String[ ] args){

System.out.println(args[0] + args[1]);

}

}

O/P:-

1020

**Case 4:**

Usually space is the separator between command line arguments if our command line arguments contains a space then we have to enclose that command line argument within double quotes.

**Java coding Standards:**

Whenever we are writing any java code it is highly recommended to follow coding stands. When ever we are writing any component its name should reflect the purpose of component (functionality). Readability and maintainability of the code will be improved.

**Coding standards for classes:**

Usually class names are nouns should starts with upper case character and if contains multiple character every inner word starts with upper case character.

Ex: String ,StringBuffer, Account, Dog etc

**Coding standards for interfaces:**

Usually interface names are adjective should starts with upper case character and if contains multiple character every inner word starts with upper case character.

Ex: Runnable

Serializable

Comparable

.

.

**Coding standards for methods:**

Usually method names are verb or verb-noun combination. Should starts with lower case alphabet symbol and if it contains multiple words then every inner word should start with upper case character(camel case conventions).

Ex:

print()

sleep()

run()

eat()

start()

**Coding standards for variables:**

Usually variable names are nouns should starts with lower case alphabet symbol and if contains multiple words then every inner word should starts with uppercase character(camel case convention).

Ex:

name

age

salary

mobileNumber

**Coding standards for constants:**

Usually constant names are nouns. Should contain only uppercase character and if it contains multiple words then these words are separated with underscore symbol.

Ex:

MAX\_VALUE

MIN\_PRIORITY

PI

**Note:** Usually we can declare constants with **public** **static** and **final** modifiers.

**JavaBean Coding Standards:**

A java bean is a simple java class with private properties with public getter and setter method.

Ex:

public class StudentBean{

private String name;

public void setName(String name){

this.name = name;

}

public String getName(){

return name;

}

}

Note: class name ends with ‘Bean’ is not official convention from SUN.

Syntax for setter method:

1. It should be public method.
2. Return type should be void.
3. Method name should be prefixed with set.
4. It should take some argument i.e. it shouldn’t be no argument method.

Syntax for getter method:

1. It should be public method.
2. Return type shouldn’t be void
3. Method name should be prefixed with get.
4. It shouldn’t take some argument.

**Note\* :** for Boolean property getter method name can be prefixed with either get or is but recommended to use is.

Ex:

private boolean empty;

public Boolean getEmpty(){

return empty;

}

public boolean isEmpty(){

return empty;

}

**Coding concepts of Listeners:**

**Case 1:** method name should be prefixed with add.

Ex:

public void addMyActionListener(MyActionListener l) // Valid

public void registerMyActionListener(MyActionListener l) // In-valid

public void addMyActionListener(ActionListener l) // In-valid

**case 2:**

method name should be prefixed with remove.

Ex:

public void removeMyActionListener(MyActionListener l) // Valid

public void unregisterMyActionListener(MyActionListener l) //Invalid

public void removeMyActionListener(ActionListener l) //Invalid

public void deleteMyActionListener(MyActionListener l) //Invalid