Language Fundamentals

Topics

1. **Identifiers**
2. **Reserved Words**
3. **Data types**
4. **Literals**
5. **Arrays**
6. **Types of Variable**
7. **Var-arg Method**
8. **Main methods**
9. **Command line Argument**
10. **Java Coding Standards**

Identifiers

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

Example:

```
class Test {
      public static void main (String [] args) {
            int x =10;
      }
}
```

Identifiers - Test, main, String, args, x

Rules for defining Java Identifiers

1. Allowed character in Java identifiers are a to z, A to Z, 0 to 9, $ and _.
   Example: total_number, total#
2. Identifiers should not start with digit.
   Example: total123, 123total
3. Java identifiers are case-sensitive.
4. In java identifiers, there is no length limit.
5. We can't use reserved words as identifiers.
   Example: int x =10; int if =20;
6. All predefined Java class name and interface names we can use as identifiers. But it is not recommended.

   Example:

```
Class Test {
      public static void main (String[] args) {
            int String =888;
            System.out.println (String); //888
      }
}
Class Test {
      public static void main (String[] args) {
            int Runnable =999;
            System.out.println (Runnable); //999
      }
}
```

Q) Which of the following is valid java Identifiers?

total_number
total#
123total
total123
ca$h
_$_$_$_
all@hands
java2share
Integer
Int
int

Note: Invalid identifiers gives compile time error.

Reserved Word (53)

1. **50 keywords** - if reserved word is associated with function.
    a. **48 used keyword**
        i. 8 keywords for data types
            byte, short, int, long, float, double, bool, char
        ii. 11 keywords for flow control
            if, else, switch, case, default, for, while, do, break, continue, return
        iii. 11 keywords for modifiers
            public, private, protected, static, final, abstract, synchronized, native, strictfp(1.2), transient, volatile
        iv. 6 keywords for exception handling
            try, catch, finally, throw, throws, assert (1.4)
        v. 6 class related keyword
            class, interface, extends, implements, package, import
        vi. 4 object related keywords
            new, instanceof, super, this
        vii. 1 return type keyword
            void
        viii. enum
    b. **2 unused keywords**
        i. **goto, const** //uses of this gives compile-time error.
2. **3 literals** - if reserved word is associated with values.
    a. **true** →value for boolean data types.
    b. **false** → value for boolean data types.
    c. **null** → default value for object reference.

enum (1.5) - Used to define a group of named constants.

Notes: All 53 Reserved words in Java contain only lower-case alphabet symbols.

Q) Which of the following list contains only Java reserved words?

new, delete
goto, constant
break, continue, return, exit
final, finally, finalize
throw, throws, thrown
notify, notifyAll,
implements, extends, imports
sizeof, instanceof
instanceOf, strictfp,
byte, short, Int
None of the above

Data types

Java is not considered as pure object-oriented programming language, because several OOPS features are not satisfied by Java like operator overloading, multiple inheritance moreover we are depending upon primitive data type which are non-object.

1. Primitive Data Types (8)
    a. Numeric Data Types (6)/Signed Data Types
        i. Integral Data Types

Sl No.	Data Type	Size	Range
1.	***byte***	(8 bit)/1 bytes	(-128) to (127)
2.	***short***	(16 bit)/2 bytes	(-2^{15}) to $(2^{15}-1)$/ (-32768) to (32767)
3.	***int***	(32 bit)/4 bytes	(-2^{31}) to $(2^{31}-1)$
4.	***long***	(64 bit)/8 bytes	(-2^{63}) to $(2^{63}-1)$

        ii. Floating point Data Types
            1. **float**
            2. **double**

float	double
Less accuracy.	More accuracy.
Single precision.	double precision.
4 bytes	8 bytes
$-3.4e^{38}$ to $3.4e^{38}$	$-1.7e^{308}$ to $1.7e^{308}$

    b. Non-numeric data types (2)

Sl No.	Data Type	Size	Range
1	***Char***	2 bytes	0 to 65535
2	***boolean***	NA (Virtual Machine dependency).	NA (values are true or false)

```
byte b =10;
byte b =127;
byte b =128; CE: Possible Loss of Precision, found: int, required: byte
byte b =10.5; CE: Possible Loss of Precision, found: double, required: byte
byte b =true; CE: incompatible types, found: boolean, required: byte
byte b =128; CE: incompatible types, found: int, required: byte
short s = 32767;
short s = 32768; CE: Possible Loss of Precision, found: int, required: short
short s = 10.5; CE: PLP, found: double, required: short
short s = true; CE: incompatible types, found: boolean, required: short
int x = 2147483647;
int x = 2147483648; CE: Integer Number to Large
int x = 21474836481; CE: PLP, found: long, required: int
int x = true; CE: incompatible types, found: boolean, required: int
boolean b = true;
boolean b =0; CE: incompatible types, found: int, required: boolean
boolean b =True; CE: cannot find symbols, Symbol: Variable True, location:
Class Test
boolean b ="True"; CE: incompatible types, found: java.lang.String, required:
boolean
```

Summary of Java Primitive data type

Data types	Size	Range	Wrapper Class	Default Value
byte	1 byte	-2^7 to 2^7-1 (-128 to 127)	Byte	0
short	2 bytes	-2^{15} to $2^{15}-1$ (-32768 to 32767)	Short	0
Int	4 bytes	-2^{31} to $2^{31}-1$	Int	0
long	8 bytes	-2^{63} to $2^{63}-1$	Long	0
float	4 bytes	$-17e^{38}$ to $1.7e^{38}$	Float	0.0
double	8 bytes	$-3.4e^{308}$ to $3.4e^{308}$	Double	0.0
boolean	NA	NA (true or false)	Boolean	false
char	2 bytes	0 to 65535	Character	0 (0 represent space character)

Literals

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

Integral literals

Example: int x =10; // int→data types, x→ variable 19 → literal

- Decimal literal (base 10)
- Octal literal (base 8) if a number prefixed with 0, it is treated as Octal.
- Hexadecimal literals (base 16)- If a number is prefixed with 0X, it is treated as hexadecimal.

Q) Which of the following declarations are valid?

```
int x = 10;
int x = 0786;
int x = 0777;
int x = 0XBeef;
int x = 0XFace;
int x = 0XBeer;
```

Char literals

```
char ch = 'a';
char ch = 97; //allowed range is 65535.
char ch ='\u 0061'; 4-digit hexadecimal number
```

Escape Character	Description
\n	New line
\t	Horizontal tab
\r	Carriage return
\b	Backspace Character
\f	Form feed
\'	Single Quote
\"	Double Quote
\\	Back Slash

Q) Which of the following are valid char literals?

```
char ch = 65536;
```

```
char ch = 0XBeer;
char ch = \uFace;
char ch = '\ubeef';
char ch = '\m';
char ch = '\iface';
```

1.7 version enhancement with respect to literals

1. **Binary literals**
   int x = 0B1111; //15
   Allowed digits are 0 & 1.
2. **Usage of underscore between digits of numeric literals**
   Example: double d = 123_456_789.9_8_7; //used only in Java file, during compilation these underscore symbol will be removed automatically. Used between digits.

Note:

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

Example: `float f = 10l; //10.0`

Arrays

Indexed Collection of Fixed no. of Homogeneous data elements.

Limitations

- Arrays are fixed in size.
- Homogeneous data types.
- We cannot increase/decrease the size of array based on our requirement.

1-d Array declaration

```
int[]  x; //recommended because name is clearly separated from type.
int  []x;
int  x[];
```

2-d Array declaration

```
int[][]  x;
int  [][]x;
int  x[][];
int[]  []x;
int[]  x[];
int  []x[];
```

Q) 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→2
int[] []a, b[];a→2, b→3
int[] []a, []b; //Compile time error
```

If we want to specify dimension before the variable that facility is applicable only for first variable in a declaration if we try to apply for remaining variable, we will get compile-time error.

3-d Array declaration

```
int[][][]  a;
int  [][][]a;
int  a[][][];
int[]  [][]a;
int[]  []a[];
int[]  a[][];
int[][]  a[];
int[][]  []a;
int  [][]a[];
int[]  a[][];
```

1-d Array Creation

Every Array in java is an object only hence, we can create arrays by using new operator.

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

Every Array type has a class. We can get class name through Syso (a.getClass ().getName());

Array Type	Class name
int[]	[I
int [][]	[[I
int [][][]	[[[I
double []	[D
short []	[S
byte []	[B
boolean []	[Z

This class is part of java language, but not available for programmer level.

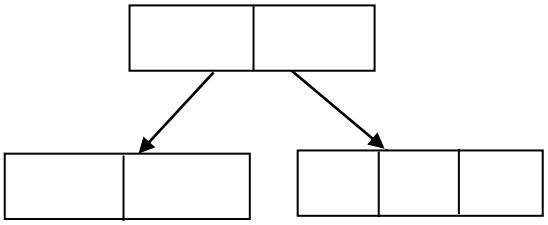
1. At the time of array, we must specify the size else we it gives Compile-time error.
   ```
   int[] x = new int[];
   int[]x = new int[6];
   ```
2. It is legal to have array with size 0 in Java.
3. ```
   int[] x = new int[-3];
   Perfectly valid, compiler only check whether it is valid int or not.
   RE: NegativeArraySizeException
   ```
4. Size of array can be byte, short, int, char.
   ```
   int[] x = new int['a'];
   ```
5. Maximum allowed array size in Java is 2147483647
   ```
   int[] x = new int[2147483647]; //Even in this case we may get RE due to lack of resources.
   int[] x = new int[2147483648]; CE: Integer number too large.
   ```
   ```

2-d Array Creation

In Java 2-d array is not implemented by using matrix style. SUN people followed Array of Arrays approach for multi-dimensional creation. The ***main advantage of this approach is memory utilization***.

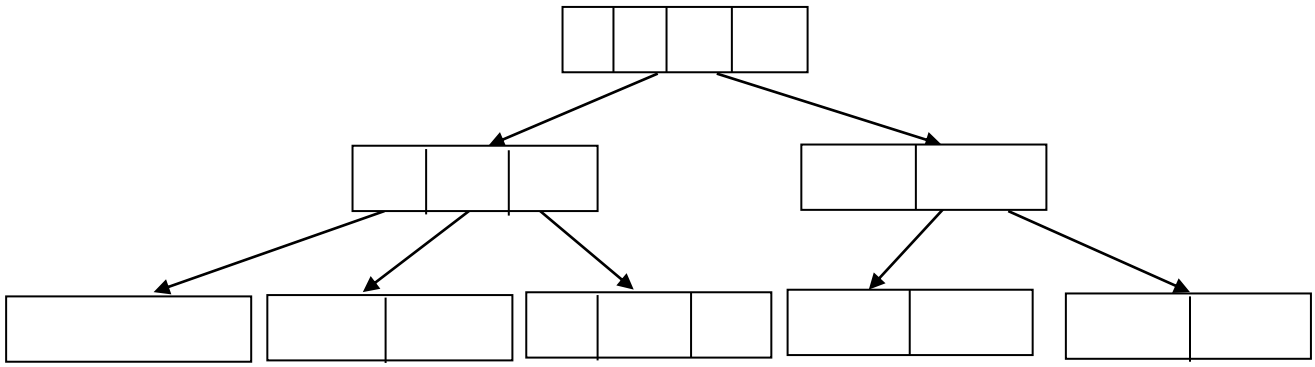
Example 1:

```
int [][] x = new int[2][];
x [0] = new int [2];
x [1] = new int [3];
```



Example 2:

```
int[][][] x = new int [2][][];
X[0] = new int[3][];
X[0][0] = new int[1];
X[0][1] = new int[2];
X[0][2] = new int[3];
X[1] = new int[2][2];
```



Q) Which of the following are valid?

```
int [] a = new int [];
int [] a = new int [3];
int [][]a = new int [][];
int [][] a = new int [3][];
int [][] a = new int [][4];
int [][] a = new int [3][4];
int [][][] a = new int [3][4][5];
int [][][] a = new int [3][4][];
int [][][] a = new int [3][][5];
int [][][] a = new int [][4][5];
```

Array Initialization

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

Note: Whenever we are trying to print any references variable internally **toString** method will be classed, which is implemented by default to return the String in the following form.

Classname@ hashCode in Hexadecimal Form

Example1:
```
int [] [] x = new int [2][3];
Syso (x); //[[I@Some no.
Syso (x [0]); //[I@Some no.
Syso (x [0] [0]); //0
```

Example2:
```
int [][] x = new int[2][];
Syso (x); //[[I@Someno.
Syso (x[0]); //null.
Syso (x[0][0]); //NullPointerException
```

Note: If we are trying to perform an operation on null, then we will get NullPointerException.

Array Declaration, Creation and Initialization in 1 line

```
int [] x;
x = new int [3];
x [0] = 10;
x [1] = 20;
x [3] = 30;
```

```
int [] x = {10,20,30};
char [] ch = {'a', 'e', 'i', 'o', 'u'};
string [] s = {"A", "AA", "AAA"}
int [][] x ={{10,20},{30,40,50}};
```

Length vs length ()

Length

- **Length is a final variable applicable for arrays.**
- Length variable represents the size of the array.
- **Example:**
```
int [] x = new int[6];
Syso (x.length ()); //CE: cannot find symbol
Syso (x.length); //6
```

Length ()

- **Length () is a final method applicable for String object.**
- Length () returns no. of character present in String.
- **Example:**
```
String s = "dhruv";
Syso (s.length); //CE: cannot find symbol
Syso (s.length ()); //5
```

```
Q) String [] s = {"A", "AA", "AAA"};
      Syso (s.length); //3
      Syso (s.length); //CE
      Syso (s[0].length); //CE
      Syso(s[0].length()); //1
Q) int [] [] x = new int[6][3];
      Syso (s.length); //6
      Syso (s[0].length); //3
```

In multi-dimensional array, length variable represents only base size but not total size.

Anonymous Array

An array without name declared for instantaneous use.

```
public class Test {
	public static void main(String[] args) {
		sum (new int [] {10,20,30,40,50});
	}
		public static void sum (int [] x) {
			int total = 0;
			for (int x1:x) {
				total = total+x1;
			}
			System.out.println ("The Sum:"+total);
		}
}
```

We can create anonymous array as follows:

```
new int [] {10, 20, 30, 40}
new int [3] {10, 20, 30}
```

2-d anonymous Array

```
new int [][] {{10,20},{30,40,50}}
```

Based on our requirement we can give the name for anonymous array then it is no longer anonymous. **int[] x = new int [] {10,20,30};**

Array element Assignment

Case 1:

```
int [] x = new int [5];
x [0] =10;
x [1] = 'a';
byte b =20; x[2] =b;
short s =30; x[3]=s;
x[4] = 10L; //CE: Possible Loss of Precision
```

In float arrays, the allowed data-type are byte, short, char, int, long, float.

Case 2: In Object type array, we can provide either declared type object or its child class object.

Example 1:

```
Object [] a = new Object [10];
a[0] =new Object ();
a[1] =new String ("Dhruv ");
a[2] =new Integer (10);
```

Example 2:

```
Number [] n = new Number [10];
n [0] = new Integer [10];
n [1] = new Double (10.5);
n [2] = new String ("dhruv"); //CE: incompatible types
```

Case 3:

```
Runnable [] r = new Runnable [10];
r [0] = new Thread ();
r [1] = new String ("dhruv"); //CE: incompatible types
```

For interface type array, as array elements its implementation class objects are allowed.

Array type	Allowed element types
Primitive Array	Any type which can be implicitly promoted to declared type.
Object type array	Either declared type or its child class objects.
Abstract Class Type Array	Its child class objects are allowed.

Interface Array Type	Their implementation class objects are allowed.

Array Variable Assignment

Case 1:

```
int[][]
int [] x = {10, 20, 30, 40};
char [] ch = {'a', 'b', 'c', 'd'};
int [] b =x;
int [] c = ch; //CE: incompatible types
```

Element level promotion is not allowed in memory.

Notes: char variable can be promoted to int variable, but char array cannot promote to int array.

Q) Which of the following promotion is valid?

```
char → int
char [] → int []
int → double
int [] → double []
float → int
float [] → int []
string → Object
string [] → Object []
```

But in case of object type Arrays, child class type Array can be promoted to parent class type Array.

```
string [] → object []
```

Example:

```
String [] s = {"A","B","C","D"}
Object [] o =s;
```

Case 2: Whenever we are assigning one array to another array, internal elements won't be copied, just reference variable will be reassigned.

```
int[] a = {10, 20, 30, 40, 50, 60};
int b = {70,80};
1. a=b;
2. b=a;
```

Case 3: Whenever we are assigning one array to another, dimension must be matched.

```
int[][] a = new int[3][];
a[0] = new int [4][3] //CE: incompatible types, found :int[][] require: int[]
a[0] =10; //CE: incompatible types, found :int require: int[]
a[0] = new int [4];
```

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

```
public class Test {
      public static void main(String[] args) {
            for (int i=0; i<=args.length; i++) {
                  System.out.println (args[i]);
            }
      }
}//Java Test A B C → A B C ArrayIndexOutofBoundException
 //Java Test A B → A B ArrayIndexOutofBoundException
```

```
 //Java Test → ArrayIndexOutofBoundException
public class Test {
	public static void main(String[] args) {
		String [] argh = {"x","y","z"};
		args= argh;
		for (String s: args) {
			System.out.println(s);
		}
	}
}//Java Test A B C → X Y Z
 //Java Test A B → X Y Z
 //Java Test → X Y Z
int[][] a = new int[4][3];
		a[0] = new int [4];
		a[1] = new int [2];
		a = new int [3][2];
```

Q) Total how many Objects are created? → 11

Q) Total how many objects eligible for Garbage Collection? → 7

Types of Variables

1. **Based on types of value represented by a variable. Variable are of two types**
   - **a. Primitive Variables**

     Can be used to represent primitive values. **Ex: int x=10;**
   - **b. Reference Variables**

     Can be used to refer objects. **Ex: Student s = new Student ();**
2. **Based on position of declaration and behavior all variable is divided into three types.**
   - **a. Instance Variables**
   - **b. Static Variables**
   - **c. Local Variables**

Instance Variable

- When the value of a variable is varied from object to object.
- For every object, a separate copy of instance variable will be created.
- It must be declared within the class directly but outside of any method/block/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 scope of object.
- Instance variable will be stored in heap memory.
- 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.

```
public class Test {
	int x = 10;
	public static void main(String[] args) {
		System.out.println (x); //CE
		Test t = new Test ();
```

```
            System.out.println (t.x); //1   0
        }
        public void m1() {
            System.out.println(x);
        }
    }
```

- For instance variables, JVM will always provide default value and we are not required to perform initialization explicitly.

```
public class Test {
        int x;
        double d;
        boolean b;
        String s;
    public static void main(String[] args) {
            Test t = new Test ();
            System.out.println (t.x); //0
            System.out.println (t.d); //0.0
            System.out.println (t.b); //false
            System.out.println (t.s); //null
        }
}
```

- It is also known as object level variable/attribute.

Static Variable

- If the value of a variable not varied from object to object then it is not recommended to declare variable as instance variable, we must declare such type of variable at class level by using static modifier.
- In case of instance variable for every object a separate copy will be created, but in case of static variable a single copy will be created at class level and shared by every object of the class.
- Static variable is declared within the class directly but outside of any method/ block/constructor.
- Static variable will be created at the time of class loading and destroyed at the time of class unloading. Scope of static variable is exactly same as scope of .class file.
- **Note:** Method Area is a subset of permanent generation which holds class definition.
    - Java Test
        - Start JVM.
        - Create and Start main Thread.
        - Locate Test.class file.
        - Load Test.class → **static variable creation**
        - Execute main () method
        - Unload Test.class →**static variable destroyed**
        - Terminate main Thread.
        - Shut down JVM.
- Static variable will be stored in method area.

- We can access static variable either by object reference or by class name but ***recommended to use class name***. Within the same class it is not required to use class name and we can access directly.
- We can access static variable directly from both instance and static areas.

```
public class Test {
	static int x = 10;
	public static void main(String[] args) {
		System.out.println(x); //10
	}
	public void m1() {
		System.out.println(x); //10
	}
}
```

- For static variables, JVM will provide default value and we are not required to perform initialization explicitly.

```
public class Test {
	static int x;
	static double d;
	static String s;
	public static void main(String[] args) {
		System.out.println (x); //0
		System.out.println (d); //0.0
		System.out.println (s); //null
	}
}
```

- Static variable also known as class-level variable or fields.

```
public class Test {
	static int x =10;
	int y =20;
	public static void main(String[] args) {
		Test t1 = new Test ();
		t1.x =888;
		t1.y =999;
		Test t2 = new Test ();
		System.out.println (t2.x+"    "+t2.y); //888    20
	}
}
```

Local variable

- Variable which are declared within method/block/constructor to meet temporary requirement.
- Also known as temporary variable/ stack variable/Automatic variables.
- It is stored inside stack memory.
- Local variable will be created while executing the block in which we declared it, one block execution completes automatically local variables will be destroyed. Hence its scope is exactly same as the block in which we declared it.

- For local variables, JVM won't provide default values. Compulsory we should perform initialization explicitly before using that variable i.e. if we are not using then it is not required to perform initialization.

```
public class Test {
	public static void main(String[] args) {
		int x;
		System.out.println ("Hello"); //Hello
	}
}
public class Test {
	public static void main(String[] args) {
		int x;
		System.out.println(x); //CE: Variable x might not initialized
	}
}
public class Test {
	public static void main(String[] args) {
		int x;
		if(args.length>0)
			x=10;
		System.out.println(x); //CE: Variable x might not initialized
	}
}
public class Test {
	public static void main(String[] args) {
		int x;
		if(args.length>0)
			x=10;
		else
			x=20;
		System.out.println(x); //Java Test A → 10 // Java Test → 20
	}
}
```

Note:

- It is not recommended to perform initialization inside logical block because there is no guarantee for the execution of this block always at runtime.
- It is highly recommended to perform initialization for local variable at the time of declaration at least with default values.
- ***Only applicable modifier for local variable is "final",*** by mistake if we are trying to apply any other modifier then we will get **CompileTimeError.**
- If we are not declaring with any modifier then it is default modifier, but this rule is applicable only for instance and static variables but not for local variables.

Conclusions:

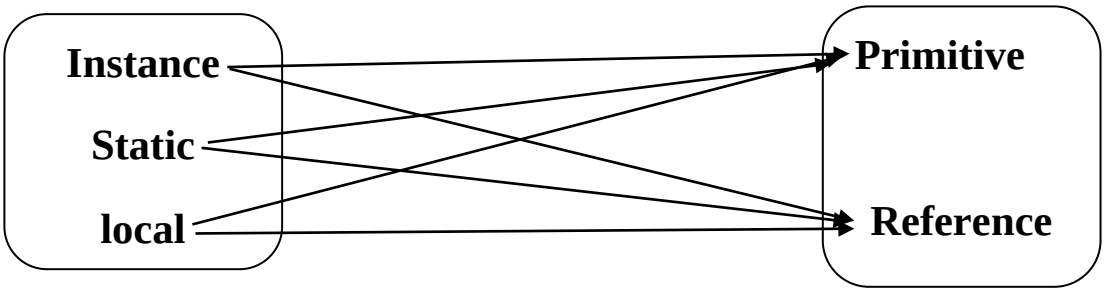
1. For instance and static variable, JVM will provide default values and we are not required to perform initialization explicitly, but for local variables JVM won't provide default values. Compulsory we should perform initialization explicitly before using that variable.

2. Instance and static variable can be access by multithread simultaneously and hence these are not thread safe. But in case of local variable, for every thread a separate copy will be created, and hence local variables are thread safe.

Types of Variables	Is thread Safe?
Instance variable	**No**
Static Variable	**No**
Local Variable	**Yes**

3. Every variable in Java should be either instance or static or local.
4. Every variable in Java should be either primitive or reference.

Hence, various possible combination of variable in Java is:



Uninitialized Array

```
public class Test {
	int [] x;
	public static void main(String[] args) {
		Test t = new Test ();
		System.out.println (t.x); //null
		System.out.println (t.x [0]); //NullPointerException
	}

}
```

1. **Instance Variable**
   a. `int[] x;`
      `System.out.println (Obj.x); //null`
      `System.out.println (Obj.x [0]); //NullPointerException`
   b. `int[] x = new int[3];`
      `System.out.println (Obj.x); //[I@someno.`
      `System.out.println (Obj.x[0]); //0`
2. **Static Variable**
   a. `static int[] x;`
      `System.out.println (x); //[I@null`
      `System.out.println (x[0]); //NullPointerException`
   b. `static int[] x = new int[3];`
      `System.out.println (x); //[I@someno.`
      `System.out.println (x[0]); //0`
3. **Local Variable**
   a. `int[] x;`
      `System.out.println(x); //CE: Variable x might not have been initialized.`
      `System.out.println(x[0]); //CE: Variable x might not have initialized.`
   b. `int [] x = new int[3];`
      `System.out.println (x); //[I@someno.`
      `System.out.println(x[0]); //0`

Note: Once we create an array element by default initialized with default values, irrespective of whether it is instance or static or local Array.

Var-arg methods (Variable number of argument methods)

We can declare a method which can take variable number of arguments such type of methods is called Var-arg methods. It is introduced in 1.5 version.

1. **We can declare a var-arg method as follows:**

```
m1 (int… x)
```

2. **We can call this method by passing any no. of int value including 0 number.**

```
m1 ();
m1 (10);
m1 (10, 20);
m1 (10, 20, 30, 40);
m1 (int… x)→ int [] x;
```

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

```
public class Test {
	public static void main(String[] args) {
		sum ();
		sum (10,20);
		sum (10,20,30);
		sum (10,20,30,40);
	}
	public static void sum (int... x) {
		int total = 0;
		for (int x1:x) {
			total = total+x1;
		}
		System.out.println ("The Sum:"+ total);
	}
}
```

Q) Which of the following are valid Var-arg methods?

Case 1:

```
m1 (int… x);
m1 (int …x);
m1 (int…x);
m1 (int x…);
m1 (int. ..x);
m1 (int .x..);
```

Case 2: We can mix var-arg parameter with normal parameter.

```
m1 (int x, int… y);
m2 (String s, double… d);
```

Case 3: If we mix normal parameter with var-arg parameter then var-arg parameter should be last parameter.

```
m1 (double… d, String s);
m2 (char ch, String… s);
```

Case 4: Inside var-arg method we can take only 1 var-arg parameter and we can't take more than 1 var-arg parameter.

```
m1 (int… x, double… d);
```

Case 5: Inside a class we can't declare var-arg method and corresponding 1-d array method simultaneously otherwise, we will get compile-time error.

```
public class Test {
        public static void m1 (int... x) {
        }
        public static void m1 (int[] x) {
        }
 }//CTE: cannot declare both m1 (int []) and m1(int... ) in Test.
```

Case 6:

```
public class Test {
        public static void m1 (int... x) {
                System.out.println ("var-arg method");
        }
        public static void m1 (int x) {
                System.out.println ("general method");
        }
        public static void main (String [] args) {
                m1 (); //var-arg method
                m1 (10, 20); //var-arg method
                m1 (10); //General method
        }
}
```

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 same as default case inside switch.

Equivalence between var-arg parameter and 1- d array

Case 1: Wherever 1-d array present we can replace with var-arg parameter.

```
m1 (int[] x ) → m1 (int… x);
```

Example: main (String[] args) can be replaced by main(String… args)

Case 2: Wherever var-arg present, we can't replace with 1-d array.

```
m1 (int… x) → m1 (int [] x);
```

Note:

1. m1 (int… x),call this method by passing a group of int value and x will become 1-d array.
2. m1 (int[]… x),call this method by passing a group of 1-d int arrays and x will become 2-d int array.

```
public class Test {
        public static void m1 (int[]... x) {
                for (int [] x1 : x) {
                        System.out.println (x1[0]); //10,40
                }
        }
        public static void main (String [] args) {
                int [] a = {10,20,30};
                int [] b = {40,50,60};
                m1 (a, b);
        }
}
```

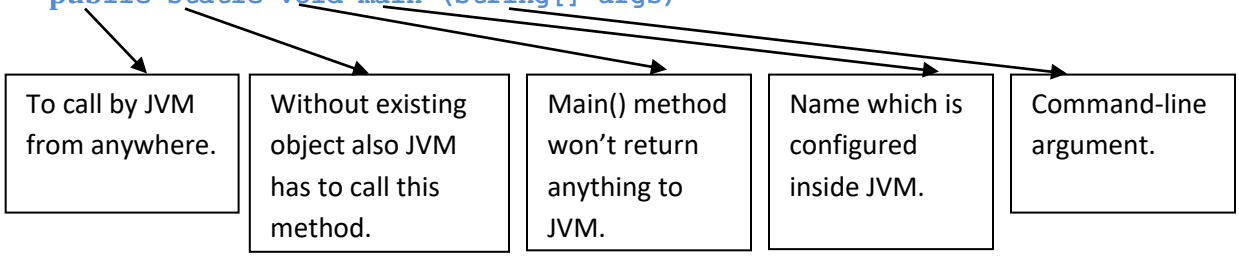
main () method

Weather the class contains main () method or not and weather main () method is declared according to requirement or not. These things won't be checked by compiler. At runtime, JVM is responsible to check these things. If JVM unable to find main () method then we will get **RE: NoSuchMethodError: main.**

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

At runtime, JVM always search for the main () method with the following prototype.

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



The above syntax is very strict and if we perform any change then we will get **RE: NoSuchMethodError: main.**

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

1. Instead of public static we can take static public, i.e. the order of modifiers is not important.
2. We can declare String [] in any acceptable form.

```
main (String[] args);
main (String []args);
main (String args[]);
```

3. Instead of args we can take any valid java identifier.

```
main(String[] dhruv);
```

4. We can replace String[] with var-arg parameter.

```
main(String… args);
```

We can declare main () method with the following modifier.

- **final**
- **synchronized**
- **strictfp**

Q)

```
public static void main (String args)
public static void Main (String [] args)
public void main (String [] args)
public static int main (String[] args)
final synchronized strictfp public void main (String[] args)
final synchronized strictfp public static void main (String[] args)
public static void main (String… args)
```

Q2) In which of the above cases we will get compile-time error?

Ans: We won't get CE anywhere but except last 2 cases in remaining will get RE: NoSuchMethodError: Main

Case1: Overloading of the main () method is possible but JVM will always call String [] argument main method only. The other overloaded method we must call explicitly like normal method call.

```
public class Test {
	public static void main (String [] args) {
		System.out.println ("String []");
	}
	public static void main (int [] args) {
		System.out.println ("int []");
	}
}
```

Case2: Inheritance concept applicable for main method. While executing child, child doesn't contain main method then parent main method will be executed.

```
public class Test {
	public static void main (String [] args) {
		System.out.println ("Parent main");
	}
}
public class Child extends Test {
} Java Test //Parent main   Java Child //Parent main
```

Case3: It seems overriding concept for main, but it is not overriding but it is method hiding.

```
public class Test {
	public static void main (String [] args) {
		System.out.println ("Parent main");
	}
 }//it is method hiding but not method overriding.
 public class Child extends Test {
	public static void main (String [] args) {
		System.out.println ("Child main");
	}
} Java Test //Parent main    Java Child //Child main
```

Note: for main method inheritance and overloading concept is applicable. Overriding is not applicable. Instead of overriding, method hiding is applicable.

Enhancement of 1.7 version

1. Until 1.6 version, If the class doesn't contain main method then we will get RE: NoSuchMethodError: main.

   But from 1.7 version onwards instead of NoSuchMethodError. We will get more elaborated error information.

   Class Test { }

1.6	1.7
RE: NoSuchMethodError: main	Error: main method not found in class Test. Please define main method as public static void main (String [] args)

2. From 1.7 version onwards main () method is mandatory to start program execution. Hence even though class contains static block it won't be executed.

```
public class Test {
   static {
      System.out.println ("Static block");
```

```
    }
}
```

1.6	1.7
Output: static block RE: NoSuchMethodError: main	RE: main method not found in class Test. Please define the main method as public static void main (String [] args)

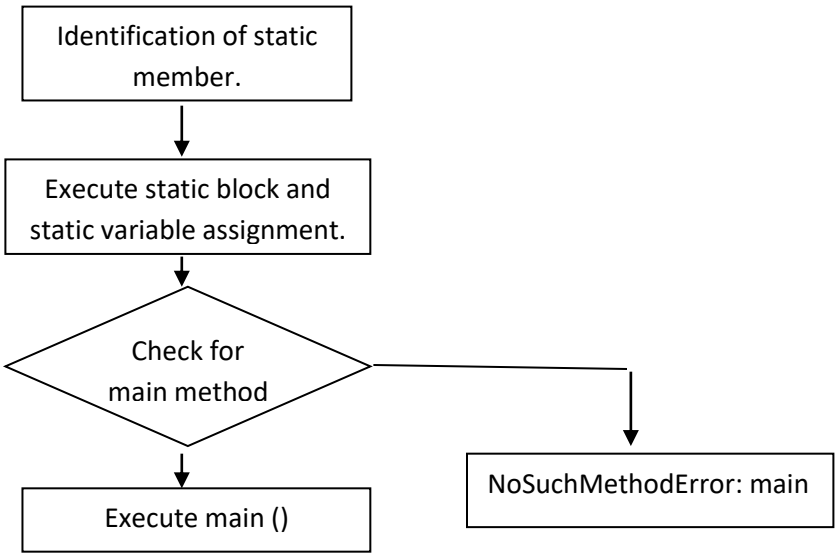
3.
```
public class Test {
        static {
                System.out.println ("Static block");
                System.exit (0);
        }
}
```

1.6	1.7
Output: static block	RE: main method not found in class Test. Please define the main method as public static void main (String [] args)

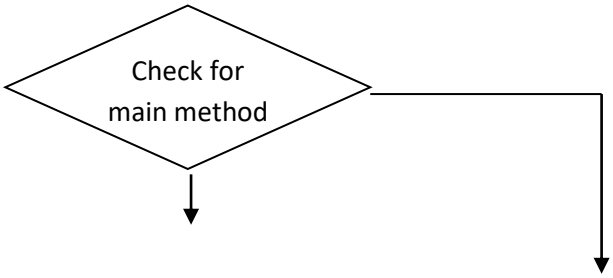
4.
```
public class Test {
        static {
                System.out.println ("Static block");
        }
        public static void main (String[] args) {
                System.out.println ("main method");
        }
}
```

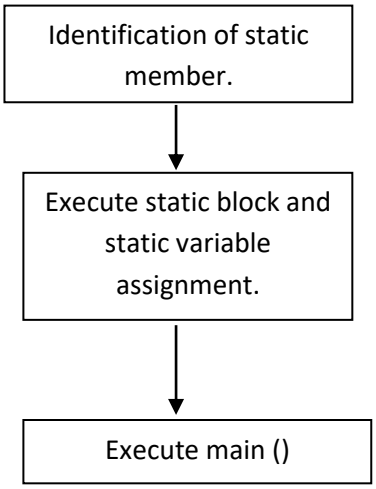
1.6	1.7
static block main method	static block main method

1.6 version



1.7 version



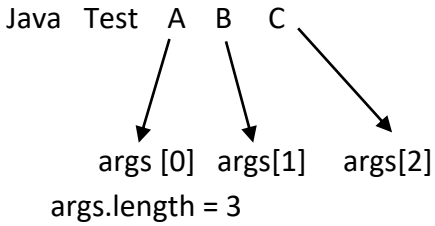


Q) Without writing main () method is it possible to print some statement to the console?

Ans: Yes, by using static block, but this is applicable till 1.6 version. It is impossible from 1.7 version onwards.

Command-line arguments

Arguments which are passing from command prompt are called command line Argument. With these command line arguments, JVM will create an array and by passing that array as argument. JVM will call main method.



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

```
1. public class Test {
        public static void main (String[] args) {
            for (int i=0; i<=args.length; i++) {
                System.out.println (args[i]);
            }
        }
   }
```

- `Java Test A B C`

  ```
  A
  B
  C
  RE: ArrayIndexOutofBoundException
  ```

- `Java Test A B`

  ```
  A
  B
  RE: ArrayIndexOutofBoundException
  ```

- `Java Test`

RE: ArrayIndexOutofBoundException

If we replace <= with < then we won't get any RE.

2.
```
public class Test {
	public static void main (String[] args) {
		String [] argh = {"x","y","z"};
		args = argh;
		for(String s : args) {
			System.out.println(s);
		}
	}
}
```

- Java Test A B C
  X
  Y
  Z
- Java Test A B
  X
  Y
  Z
- Java Test
  X
  Y
  Z

3.
```
public class Test {
	public static void main (String[] args) {
		System.out.println (args[0] + args[1]);
	}
}//Java Test 10 20
//1020
```

Within main method command-line argument are available in String form.

4.
```
public class Test {
	public static void main (String[] args) {
		System.out.println (args[0]);
	}
}	//Java Test "Note Book"
	//Note Book
```

Usually space itself is the separator between command-line arguments, if our command-line argument itself contains space then we must enclose that command-line argument within double-quotes.

Java Coding Standard

Whenever we are writing java codes. It is highly recommended to follow coding standard. Whenever we are writing any component its name should reflect purpose of that component (functionality). The main advantage of this approach is Readability and maintainability of the code will be improved.

Example:

Without Coding Standard	With Coding Standard
`public class A {`	`package com.dhruv.springdemo;`

```
        public int m1(int x, int y)
                return x+y;
}
```

```
public class Calculator {
        public static int add(int num1, int
num2)
                return num1+num2;
}
```

Coding standard for classes

Usually class names are Nouns, should starts with uppercase character and if it contains multiple words every inner word should start with uppercase character.

Example: String, StringBuffer etc.

Coding standard for Interfaces

Usually interfaces names are Adjective, should start with uppercase character and if it contains multiple words, every inner word should start with uppercase character.

Example: Runnable, Serializable etc.

Coding standard for Methods

Usually method names are verbs and verbs-noun combination should start with lowercase, if it contains inner words, every inner words starts with uppercase i.e. camel-case.

Example: print (), eat (), getName (), setSalary () etc.

Coding standard for Variables

Usually Variable names are noun should start with lower case alphabet symbol and if it contains multiple words then every inner word should start with uppercase character. (Camel-case convention)

Example: name, age, mobileNumber etc.

Coding standard for constants

Usually constant names are nouns should contain only uppercase character and if it contains multiple words then these words are separated with (-) symbol.

Example: MAX_VALUE, MIN_VALUE etc.

Note: Usually we can declare constant with public, static and final modifier.

Java Bean Coding Standard

A java bean is a simple java class with ***private properties and public getter and setter methods***. Class name ends with bean is not official convention from SUN.

Syntax for Setter method

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

Syntax for Getter method

1. public method.
2. return type should not be void.
3. no-argument method.
4. method-name should be prefixed with get.

Note: For boolean properties getter method name can be prefixed with either get or Is but recommended to use Is.

```
public boolean empty;
	public boolean getEmpty () {
		return empty;
	}
	public boolean IsEmpty () {
		return empty;
	}
}
```

Coding Standard for listeners

Case 1: to register a listener

1. method-name should be prefixed with add.
2. public method.
3. return type must be void.

```
public void addMyActionListner (myActionListner l);
public void registerMyActionListner (myActionListner l);
public void addMyActionListner (ActionListner l);
```

Case 2: to unregister a listener.

1. Method-name prefixed with remove.

```
public void removeMyActionListner (myActionListner l);
public void unRegisterMyActionListner (myActionListner l);
public void removeMyActionListner (ActionListner l);
public void deleteMyActionListner (myActionListner l);
```