***Types of Variables***

***Division One***

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

1. Primitive Variables can be used to represent primitive values

EG: int x = 10;

1. Reference variables can be used to refer objects.

EG: student s = new student();

***Division two***

Based on position of declaration and behaviour variables are divided into three types.

1. Instance variables
2. Static variables
3. Local variables

***1.Instance Variables:***

If the value of the variable is varied from object to object such types of variables are called instance variables.

For every object a separate copy of instance variables will be created.

Instance variables should be declared within the class directly 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 the scope of object.

Instance variables will be stored in the heap memory as the part of object.

We cannot access instance variables directly from static area but we can access by using object reference.

But we can access instance variables directly from instance area.

For instance variables JVM will always provide default values and we are not required to perform initialization explicitly

Instance variables are also known as object level variables or attributes

EG:

**public** **class** Rough {

**int** x = 10;

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

System.***out***.print(x);

(Cannot make a static reference to the non-static field x)

Rough r = **new** Rough();

System.***out***.print(r.x);

}

**public** **void** m1() {

System.***out***.print(x);

}

}

***2.Static Variables***

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 types of variables at class level by using static modifier.

In case of instance variables for every object a separate copy will be created in the case of static variable a single copy is created at class level and shared by every object of the class.

Static variables should be declared within the class directly but outside of any method or block or constructor.

Static variables are created at the time of class loading and destroyed at the time of class unloading hence scope of static variable is exactly same as scope of .class file.

Java rough

1. Start JVM
2. Create and start main thread
3. Locate Rough.class file
4. Load Rough.class ***..Static Variable Creation***
5. Execute main() method
6. Unload Rough.class ***..Static Variable destruction***
7. Terminate main thread
8. Shut down JVM

Static variables will be stored in method area.

We can access static variables either by object reference or by class name but recommended is use class name.

Within the same class it is not required to use class name and we can access directly.

EG:

**public** **class** Rough {

**static** **int** *x = 10*;

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

Rough r = **new** Rough(); ..10

System.***out***.println(r.*x*); ..10

System.***out***.println(Rough.*x*); ..10

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

}

}

We can access static variables directly from both instance and static areas/methods(Can be accessed from methods).

For static variables JVM will provide default values and we are not required to perform initialisation explicitly.

Static variables are also known as class level variables or fields.

***3.Local Variables***

Sometimes to meet temporary requirements of the programmer we can declare variables inside a method or block or constructor. Such type of variables are called local variables or temporary or stack or automatic variables.

Local variables are stored inside stack memory.

Local variables will be created while executing the block in which we declared it. Once block execution completes automatically local variables will be destroyed. Hence the scope of local variable is the block in which we declared it.

EG:

**public** **class** Rough {

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

**int** i = 0;

**for**(**int** j=0; j<3;j++) {

i=i+j;

}

System.***out***.println(i+"...."+j);

}

}

Error: j cannot be resolved to a variable

For local variables JVM doesn’t provide default values, compulsorily we should perform initialisation explicitly before using that variable.

NOTE:

It is not recommended to perform initialisation for local variables inside logical blocks because there is no guarentee for the execution of these blocks always at runtime

It is highly recommended to perform initialisation for local variables at the time of declaration at least with default values.

The only applicable modifier for local variables is ‘final. By mistake if we are trying to apply any other modifier then we will get compile type error.

public int x = 10;

private int x = 10;

protected int x = 10; Illegal modifier for parameter x; only final is permitted

static int x = 10;

transient int x = 10;

volatile int x = 10;

final int x = 10; ..valid

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

**CONCLUSIONS:**

For instance and static variables JVM will provide default values and initialisation is not required explicitly but for local variables JVM will not provide default values. Compulsorily we should perform initialisation explicitly before using that variable.

Instance and static variables can be accessed by multiple threads simultaneously and hence these are not thread safe but in the case of local variables for every thread a separate copy will be created and hence local variables are thread safe.

Every variable in java should be either instance or static or local.

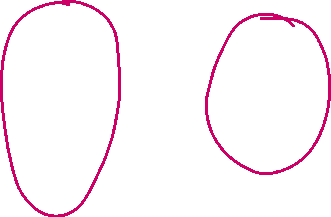
Every variable in java should be either primitive or reference.

Hence, various possible combinations of variables in java are:

Instance

Primitive

Static



Reference

Local

Class Test

{

Int x = 10; ..instance primitive

Static String s = “durga”; ..static reference

Public static void main(String[] args)

{

Int[] y = new int[3]; ..local reference

}

}

***Uninitialized Arrays***

1. ***Instance level***

**EG 1:**

**public** **class** Rough {

**int**[] x;

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

Rough r = **new** Rough();

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

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

}

}

Output:

null

java.lang.NullPointerException: Cannot load from int array because "r.x" is null

**EG 2:**

**public** **class** Rough {

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

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

Rough r = **new** Rough();

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

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

}

}

Output:

[I@5e265ba4

0

1. ***Static level***

**EG 1:**

**public** **class** Rough {

**static** **int**[] *x*;

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

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

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

}

}

Output: null

Exception in thread "main" java.lang.NullPointerException: Cannot load from int array because "Rough.x" is null

**EG 2:**

**public** **class** Rough {

**static** **int**[] *x* = **new** **int**[3];

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

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

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

}

}

Output:[I@5e265ba4

0

1. ***Local level***

**EG 1:**

**public** **class** Rough {

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

**int**[] x;

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

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

}

}

Output:The local variable x may not have been initialized

The local variable x may not have been initialized

**EG 2:**

**public** **class** Rough {

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

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

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

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

}

}

Output:[I@5e265ba4

0

**Note:**

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

***Var args Method***

Until 1.4 version we cannot declare a method with variable number of arguments. If there is a change in number of arguments compulsorily we should go for new method. It increases length of the code and reduces readability. To overcome this problem var arg methods is introduced 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 args methods.

***Case 1.***

We can declare a var-arg method as follows: sum(int… x)

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

EG:

Sum();

Sum(10, 20);

Sum(10, 20, 30);

Internally var-arg parameter will be converted into one dimensional array .hence within the var-arg method we can differentiate values by using index.

**Which of the following method declarations?**

* sum(int… x); ..valid
* sum(int …x); ..valid
* sum(int…x); ..valid
* sum(int x…); ..invalid
* sum(int. ..x); ..invalid
* sum(int .x..); ..invalid

***Case 2.***

We can mix var-arg parameters with normal parameters.

EG:

Sum(int x, int…y);

Sum(String s, double…y);

***Case 3.***

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

EG:

Sum(double… d, String s); ..invalid

Sum(char ch, String… s); ..valid

***Case 4.***

Inside var-arg method we can take only one var-arg parameter and we cannot take more than one var-arg parameter.

Sum(int… x, double… y); ..invalid

***Case 5.***

Inside a class we cannot declare var-arg method and corresponding one dimensional array method simultaneously otherwise we will get compile time error.

***Case 6.***

In general var-arg method will get least priority i.e if no other method is matched then only var-arg method will get the chance.

It is exactly same as default case inside switch statement.

**public** **class** Rough {

**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, 20);

*sum*(10);

}

}

Output:

var-args method

var-args method

general method

***Equivalence between var-arg parameter and one dimensional array***

**Case 1:**

Wherever one dimensional array is present we can replace it with var-arg parameter.

Sum(int[] x) Sum(int… x)

EG: main(String[] args) main(String… args)

**Case 2:**

Wherever var-args parameter is present we cannot replace it with one dimensional array

Sum(int… x) Sum(int[] x) ..invalid

NOTE:

1. Sum(int… x) ---🡪 int[] x;

We can call this method by a group in int values and x will become an one dimensional array

1. Sum(String… x) ---🡪 String[] x;
2. Sum(int[]… x) ---🡪int[][] x;

We can call this method by passing a group of one dimensional int arrays and x will become two dimensional int array.

1. Sum(int[][]… x) ---🡪int[][][] x;