**class** Demo

{

**static** **int** *a*;

**static** **int** *b*;

**static**

{

*a*=10;

*b*=20;

System.***out***.println("static block");

}

**static** **void** disp()

{

System.***out***.println("static method");

}

**int** x,y;

{

//This is Java Block

x=10;

x=20;

System.***out***.println("Java Block");

}

**void** disp1()

{

System.***out***.println("Non Static Method");

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

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

}

Demo()

{

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

}

}

**public** **class** Static {

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

Demo.*disp*();

Demo d=**new** Demo();

d.disp1();

}

}

Output:

static block

static method

Java Block

Constructor

Non Static Method

10

20

Static Execution Flow :

**public** **class** StaticFlowController {

**static** **int** *a*;

**static** **int** *b*;

**static**

{

*a*=10;

*b*=20;

System.***out***.println("static block");

*disp*();

}

**static** **void** disp()

{

System.***out***.println("static method");

}

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

// **TODO** Auto-generated method stub

System.***out***.println("Inside the Main Method");

}

}

Output:

static block

static method

Inside the Main Method

static :

A call

static variables

static block

static method

Non-Static varaibles(instane)

Java Block

Normal method

We can use static varaible in

static block

static method

Java Block

Normal method

where as for instance variable it can be present in Java Block or Normal method but not static block /static method beacuse static members are Object independent there might be chance no Object has been created for the instance variabnle used in static block/Method

Static members are Object independent.

Order of Program Excecution if all the below are there in program:

1.static variables

2.static Blocks

3.static Methods

4.Instance variables

5.Java Block

6.Constructor

7.Methods

class loader subsystem

JVM Architecture:

A diagram of a computer system

Description automatically generated

class Loader Subsystem : Loading +Linking+intilization

Loading: Loads the entire .class into method area

Application Class Loader: It loads all the inbuilt API classes

Bootstrap Class Loader :Loads the programs written by the user.

Extension class Loader : Loads the programs written by the user.

linking :

verify: Byte code verifier--Generated in the correct format/not.it checks whether this file is properly formatted and generated by a valid compiler or not.

preparation : Checks if there static variables and if present memory for them will be allocate on the Heap area and will be assigned with default values.

Resolution :It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

Initialization:

In this phase, all static variables are assigned with their values defined in the code and static block(if any) will get Excecuted.

JVM Data Areas:

Method Area :

In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

Heap area:

Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.

Stack area:

For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.

PC Registers: Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.

Native method stacks: For every thread, a separate native stack is created. It stores native method information.

Execution Engine

Execution engine executes the “.class” (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:

Interpreter: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.

Just-In-Time Compiler(JIT) : It is used to increase the efficiency of an interpreter. It compiles the bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.

Garbage Collector: It destroys un-referenced objects. For more on Garbage Collector, refer Garbage Collector.

when a new Object is created below is the Order of Execution

Instance variables

Java Block

Constructor

when a constructor call is made during object creation the java block which is written in the class internally will be moved to constructor body first the Java block will be present at the starting and constructor body will be present.

{

//Java Block

}

Constructor()

{

//Constructor Body

}

Internally it will be like below:when an constructor call is made it will be in the below format.

constructor

{

{

//Java Block

}

//Constructor Body

}

For every Object instance variables will be created separately in heap with in the Object.

Where as for the static variables will be created only once in the heap area.

We can call the static methods using Object Reference.

Memory for the instance variables will get created during the Object Creation. So we will not able to use the instance variables in the static block/method.

static members gets memory during the class loading.

class A{

static void disp()

{

}

public static void main(String args[])

{

A obj=new A();

obj.disp();//valid

disp();//Valid

}

}

If a method is trying to call the method in the same class then we doesnot need class name /Reference we can call directly using the method name.

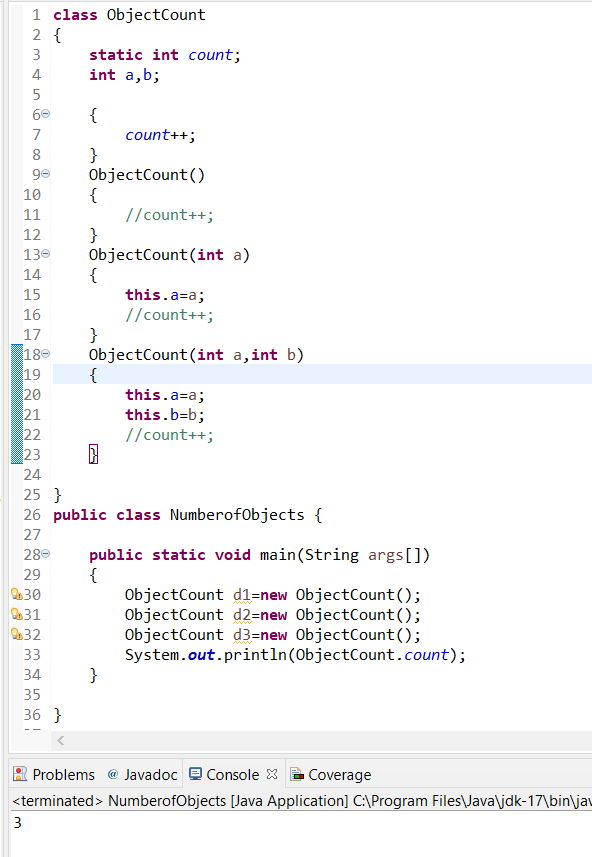
If a static method is present in the different class then we can call that Method using the Classname

A.disp();

Static variables/static block will get executed during class loading

Main Method will get Executed by JVM.

Static variables are also called as class variables.



Static variables:

Created using static keyword.

Executed during class loading

Memory will be created in Heap during class Loading.

It is also class variable because one copy of static variable is shared among all the Objects.

We can access this using Class Name

Static variables are Object independent.

They can be accessed in static and Non\_static elements.

Among static block and main method—static block will get executed first.

Static block is used to initialize the static variables.

If there are some statements that needs to be executed before main method we can keep them in the Static block

static

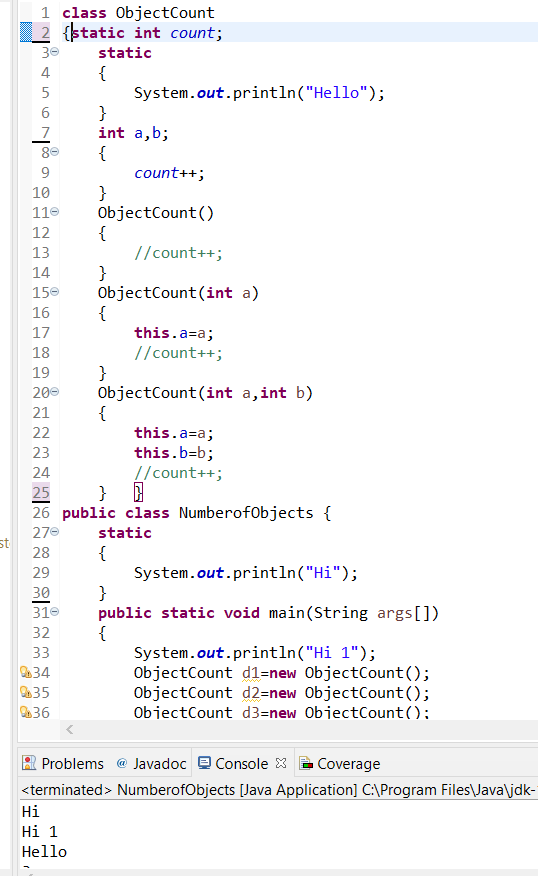
{

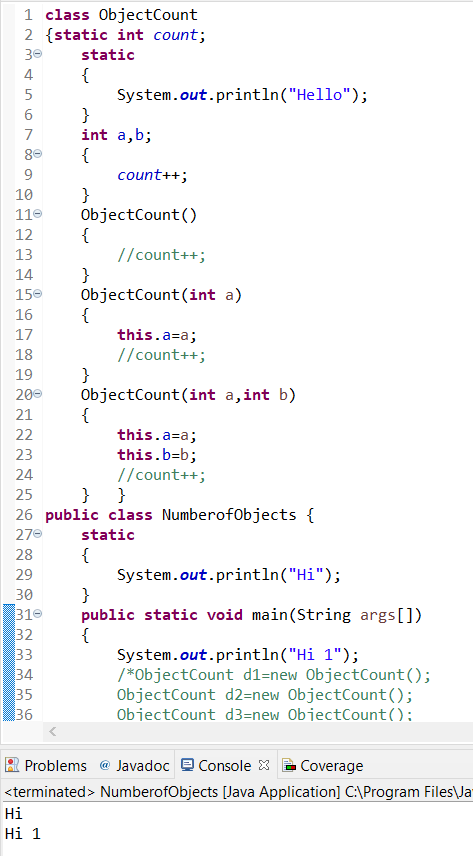
}

If there are some statements that needs to be executed every time an Object is created we can keep them in the Java block/Non-instance block

{

}





|  |  |
| --- | --- |
| static Method | Instance Method |
| We can call this using class Name | We can call this method from other class using Object |
| Object dependent | Object Independent |
| Can be accessed using reference | Can be Accessed using Reference |