Static block prepare overid...

```
class Sample
{
    static
    {
        System.out.println("Static Block");
        x = m1();
        System.out.println("x from static block :"+Sample.x);
    }

    public static int m1()
    {
        System.out.println("Static Method");
        return 100;
    }

    static int x;
}
public class StaticBlockDemo10
{
    public static void main(String[] args)
    {
        System.out.println("x from main method :"+Sample.x);
    }
}

class Demo
{
    public static void print()
    {
        x = 128;
        System.out.println("x value is :"+x);
    }

    static int x;
}

public class StaticBlockDemo11
{
    public static void main(String[] args)
    {
        Demo.print();
    }
}
```

Compile Time Constant :

A compile time constant is a constant that is evaluated and replaced with its value at compile time rather than runtime.

It must be declared with static and final modifier as well as must be initialized with constant expression. (Must not be initialized by method call)

At compile time, constant value will be converted by compiler at the time of compilation itself so, at runtime JVM can see the value but not the class name so class will not be loaded as shown in the program.

Example :- public static final int A = 100; //Valid [Here class will
not be loaded by JVM]

public static final int A = m1(); //valid [Here
class will be loaded by JVM]

CompileTimeConstantDemo1.java

```
class Alpha
{
    public static final int A = 100; //Compile Time Constant

    static
    {
        SO.println("Static Block");
    }
}

public class CompileTimeConstantDemo1
{
    public static void main(String[] args)
    {
        System.out.println(Alpha.A);
    }
}
```

Note :- Here Alpha.class file will not be loaded

```
class Beta
{
    public static final int A = m1();

    static
    {
        SO.println("Static Block");
    }

    public static int m1()
    {
        return 100;
    }
}

public class CompileTimeConstantDemo2
{
    public static void main(String[] args)
    {
        System.out.println(Beta.A);
    }
}
```

Here Beta class will be loaded

```
public class Test
{
    public static final int MAX_VALUE = 999; //variable time constant
}

public class Main
{
    public static void main(String[] args)
    {
        System.out.println(Test.MAX_VALUE); // 999
    }
}
```

999 change to 1500

Note : If we compile and execute both programs we will get 999 value, now change the 999 to 1500, the compile only Test.java or a new .class file will be created which will hold 1500 value. Now without re-compilation of main.java if we execute Main.java then we will get 999 as it takes the value at compile time.

Runtime Data Areas :

It is also known as Memory Areas.

Once a class is loaded then based on field/variable type/ method type it is divided into different memory areas which are as follows :

1) Method Area [Only One per JVM]
2) HEAP Area [Only One per JVM]
3) Stack Area [We have multiple Stack Area for a single JVM] [Multithreading]
4) PC Register [Hold the current execution instruction of each Thread]
5) Native Method Stack [To hold native method information]

Method Area :

Whenever a class is loaded then the class is dumped inside method area and returns java.lang.Class class object.

It provides all the information regarding the class, like name of the class, name of the package, static and non static fields available in the class, methods available in the class and so on.

We have only one method area per JVM that means for a single JVM we have only one Method area.

This Method Area OR Class Area is shareble by all the objects.

HEAP AREA :

Whenever we create an object in java then the properties and behavior of the object are stored in a special memory area called HEAP AREA.

We have only one HEAP AREA per JVM.

STACK Area :

All the methods are executed as a part of Stack Area.

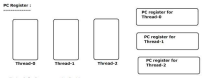
Whenever we call a method in java then internally one stack Frame will be created to hold method related information.

Every Stack Frame contains 3 parts :
1) Local variable arrays
2) Frame Data
3) Operand Stack.

We have multiple stack area for a single JVM.

EveryTime we create a new thread in java then JVM will create a separate Runtime Stack Memory [Multithreading].

PC Register :



It stands for Program counter Register.

In order to hold the **current executing instruction of running thread** we have separate PC register for each and every thread as shown in the diagram.

Native Method Stack

Native method means, the java methods which are written by using native languages like C and C++.

In order to write native method we need native method library support.

Native method stack will hold the native method information in a separate stack.

Execution Engine : [Interpreter + JIT Compiler]

Interpreter

In java, JVM contains an interpreter which executes the program line by line.

Interpreter is slow in nature because at the time of execution if we make a mistake at line number 8 then we will get the error message but after solving the exception again it will start the execution from line number 1 so it is slow in execution that is the reason to boost up the execution java software people has provided JIT compiler.

JIT Compiler :-[Will convert Bytecode to native code]

It stands for just in time compiler. The main purpose of JIT compiler to boost up the execution so the execution of the program will be completed as soon as possible.

JIT compiler holds the repeated instruction like method signature, variables, native method code and make it available to JVM at the time of execution so the overall execution becomes very fast.

Variable Memory Allocation and Initialization :
---

1) static field OR Class variable :

Memory allocation done at prepare phase of class loading and initialized with default value even variable is final.

It will be initialized with Original value [If provided by user at the time of declaration] at class initialization phase.

When JVM will shutdown then during the shutdown phase class will be un loaded from JVM memory so static data members are destroyed. They have long life.

2) Non static field OR Instance variable

Memory allocation done at the time of object creation using new keyword [Instantiation] and initialized with default values even the variable is final. [Object class> at the time of declaration -> instance block -> constructor].

When object is eligible for GC then object is destroyed and all the non static data members are destroyed along with corresponding object. It has less life in comparison to static data members because they belongs to object.

3) Local Variable

Memory allocation done at stack area [Stack Frame] and developer is responsible to initialize the local variables. Once method execution is over, it will be deleted from stack Frame hence it has shortest life.

4) Parameter variable

Memory allocation done at stack area [Stack Frame], and end user is responsible to pass the parameters. Once method execution is over, it will be deleted from stack Frame hence it has shortest life.

Index No.	Variable Type	Declaration Position	Scope	Memory Area	Implicit Initialization	Accessed By	Lifetime
01	Class Variable OR Static Field	Inside Class	Belongs to Class	Method Area	private, protected, public, Default, static, final variable.	Class name followed by dot operator. Reference.	Since class loaded to JVM memory and unloaded
02	Instance variable OR Non Static Field	Inside Class	Belongs to an object instance	Heap Area	private, default, protected, public, Static, final variable.	Object reference.	As long as object is live.
03	Local Variables	Inside Method	Inside a method/constructor/ instance block/ static block/initialization block.	Stack Area	Method/ constructor/ instance/ static block/ initialization block.	Only inside method/ constructor/ block.	Until method/ block/ constructor/ block.
04	Parameter Variable	Inside Method	Inside method/constructor.	Stack Area	Method/ constructor/ instance/ static block/ initialization block.	Through value passing method/ constructor.	Until method/ block/ constructor/ block.

- Field/Variable for **static field** [static, synchronized, native, strictfp and final] variable.
- Field/Variable for **Instance field** [Only Non static field] data is shared.
- Field/Variable for **Local Variable** [Only Stack is shared]
- Field/Variable for **Parameter Variable** [Only Stack is shared]

Method Overriding in Java :