```
//Illegal forward reference
class Demo
{
  static
    {
      i = 100;
    }

  static int i;
}

public class StaticBlockDemo5
{
  public static void main(String[] args)
    {
      System.out.println(Demo.i);  //100
    }
}
```

```
class Demo
{
  static
    {
      i = 100;
       //System.out.println(i); //Illegal forward reference
       System.out.println(Demo.i);
    }

  static int i;
}

public class StaticBlockDemo6
{

  public static void main(String[] args)
    {
      System.out.println(Demo.i);  //100 100
    }
}
```

```
class Test
{
    static
    {
      i = 100;
    }

    static int i = 200;
}

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

Output is 200 because static variable and static block both are having same priority. Executed according to the order.

```
class Test
{
    public static void print()
    {
        i = 100;
        System.out.println(i);
    }

    static int i;
}

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

Note : In order to print the value of i we need class name if it is static block but if we use static method then class name is not required because static method executed after static variable and static block

```
class Test
{
    {
      i = 100;
      //System.out.println(i); //Invalid Illegal Forward Ref
      System.out.println(this.i);
    }

    int i;
}
public class StaticOrderTest
{
    public static void main(String[] args)
    {
        new Test();
    }
}
```

Note : We have rule for Illegal forward reference error for non static variable and non static block

```
class StaticBlockDemo7
{
    static
    {
    System.out.println("Static Block");
      return;
    }

    public static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

Note : return keyword is not allowed for Initializer

```
class Test
{
    public static final Test t1 = new Test();  // t1 = null

    static
    {
    System.out.println("static block");
    }

    {
      System.out.println("Non static block");
    }

    Test()
    {
        System.out.println("No Argument Constructor");
    }
}

public class StaticBlockDemo9
{
    public static void main(String[] args)
    {
        new Test();  //2 steps (class loading + Object creation)
    }
}
```

static variable + static block both are having same priority

```
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 = 120;
        System.out.println("x value is :"+x);
    }

    static int x;
}

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

What is compile time constant ?

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 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.

//Programs

```
class Test
{
    public static final int A = 100;  //Compile time constant

    static
    {
        System.out.println("Static Block");
    }
}

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

Note : Static Block will not be executed

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

    static
    {
        System.out.println("Static Block");
    }

    public static int m1()
    {
      return 500;
    }
}

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

Note : Here we don;t have compile time constant so static block will be executed.

//Program that describe compile time constant value is provided by compiler at compile time.

```
public class Alpha
{
    public static final int VALUE = 9999999;
}

public class CompileTimeConstantDemo3
{
    public static void main(String[] args)
    {
        System.out.println(Alpha.VALUE);
    }
}
```

Note : * Compile both the programs and execute
* Go to Alpha class change the value of VALUE variable
* Re-compile Alpha class (A new .class file will be created)
* Execute CompileTimeConstantDemo3 directly then we will get old value of Alpha class

---

Can we write a Java program without main method ?

```
class WithoutMain
{
  static
    {
      System.out.println("Hello User!");
      System.exit(0);
    }
}
```

It was possible to write a java program without main method till JDK 1.6V.
From JDK 1.7v onwards, at the time of loading the .class file JVM will verify the presence of main method in the .class file. If main method is not available then it will generate a runtime error that "main method not found in so so class".

---

2) Non static field OR Instance variable

Memory allocation done at the time of object creation using new keyword (Instantiation) and initialized as a part of Constructor 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 memebers are also destroyed 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 variable before use. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

4) Parameter variable

Memory allocation done at stack area (Stack Frame) and end user is responsible to pass the value at runtime. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

Note : We can done validation only one parameter variables.

How many ways we can load the .class file into JVM memory :
