Language Fundamentals

Language Fundamentals

To prepare java applications , we need some fundamentals provided by Java programming language. we need some building blocks which are not created by the developers , which must be provided by the programming languagebydefault, such building block are language fundamentaks

1) **Tokens**
2) **Data Types**
3) **Type Casting**
4) **Java Statements**
5) **Arrays**

1)Tokens:

Smallest logical unit in java programming is called as "Lexeme".

The Collection of Lexemes come under a particular group is called as "Token"

int a=b+c*d;

Lexemes: int, a, =, b,+, c, *, d, ;-----> 9

Tokens:
1)Data Types: int
2)Identifiers: a, b, c, d
3)Operators: =, +, "*
4)Special Symbol: ;

Types of tokens: 4

To prepare java applications, java has provided the following list of tokens.

1) **Identifiers**
2) **Literals**
3) **Keywords/ Reserved Words**
4) **Operators**

Identifiers:

Identifier is a name assigned to the programming elements like variables, methods, classes, abstract classes, interfaces,.....

EX:
int a=10;
int ----> Data Types
a ------> variable[Identifier]
= ------> Operator
10 -----> constant
; ------> Terminator

To provide identifiers in java programming, we have to use the following rules and regulations.
Identifiers should not be started with any number, identifiers may be started with an alphabet, '_' symbol, '$' symbol, but, the subsequent symbols may be a number, an alphabet, '_' symbol, '$' symbol.

int eno=111;------> Valid
int 9eno=999;-----> Invalid
String _eaddr="Hyd";---> Valid
float $esal=50000.0f;----> valid
String emp9No="E-9999";----> Valid
String emp_Name="Durga";----> Valid
float emp$Sal=50000.0f;------> Valid

Identifiers are not allowing all operators and all special symbols except '_' and '$' symbols.
int empNo=111; -------> valid
int emp+No=111;-------> Invalid
String emp*Name="Durga";-----> Invalid
String #eaddr="Hyd";--->Invalid
String emp@Hyd="Durga";-----> Invalid
float emp.Sal=50000.0f;-----> Invalid
String emp-Addr="Hyd";------> Invalid
String emp_Addr="Hyd";------> Valid

Identifiers are not allowing spaces in the middle.
concat(--)----> Valid
forName(--)---> Valid
for Name()----> Invalid
getInputStream()--> valid
get Input Stream()----> Invalid

Identifiers should not be duplicated with in the same scope, identifiers may be duplicated in two different scopes.

```
class A {
     int i=10;-----> Class level
     short i=20;----> Error
    double f=33.33---> No Error
    void m1() {
           float f=22.22f; -----> local variable
           double f=33.33;---> Error
           long i=30;---> No Error
    }
}
```

In java applications, we can use all predefined class names and interface names as identifiers.

EX1:
int Exception=10;
System.out.println(Exception);
Status: No Compilation Error
Output: 10

EX2:
String String="String";
System.out.println(String);
Status: No Compilation Error
Output: String

EX3:
int System=10;
System.out.println(System);
Status: COmpilation Error

Reason: Once if we declare "System"[Class Name] as an integer variable then we must use that "System" name as integer variable only in the remaining program, in the remaining program if we use "System" as class name then compiler will rise an error.
In the above context, if we want to use "System" as class name then we have to use its fully qualified .

Note: Specifying class names and interface names along with package names is called as Fully Qualified Name.

EX: java.io.BufferedReader
java.util.ArrayList

EX:
```
int System=10;
java.lang.System.out.println(System);
System=System+10;
java.lang.System.out.println(System);
System=System+10;
java.lang.System.out.println(System);
```
Status: No Compilation Error

Output:
```
10
20
30
```

Along with the above rules and regulations, JAVA has provided a set of suggessions to use identifiers in java programs
In java applications, it is suggestible to provide identifiers with a particular meaning.

EX:
String xxx="abc123";------> Not Suggestible
String accNo="abc123";----> Suggestible

In java applications, we dont have any restrictions over the length of the identifiers, we can declare identifiers with any length, but, it is suggestible to provide length of the identifiers around 10 symbols.

EX:
String permanentemployeeaddress="Hyd";----> Not Suggestible
String permEmpAddr="Hyd";----> Suggestible

If we have multiple words with in a single identifier then it is suggestible to seperate multiple words with special notations like '_' symbols.

EX:
String permEmpAddr="Hyd";----> Not Suggestible
String perm_Emp_Addr="Hyd";----> Suggestible

Literals

Literal is a constant assigned to the variables .

EX:
int a=10;
int ----> data types
a ------> variables/ identifier
= ------> Operator
10 -----> constant[Literal].
; ------> Special symbol.
To prepare java programs, JAVA has provioded the following set of literals.

1.Integer / Integral Literals:
byte, short, int, long ----> 10, 20, 30,....
char -----> 'A','B',.....

2.Floating Point Literals:
float ----> 10.22f, 23.345f,.....
double----> 11.123, 456.345,....

3.Boolean Literals:
boolean -----> true, false

4.String Literals:
String ---> "abc", "def",......

Note: JAVA7 has given a flexibility like to include '_' symbols in the middle of the literals inorder to improve readability.

EX:
float f=12345678.2345f;
float f=1_23_45_678.2345f;

If we provide '_' symbols in the literals then compiler will remove all '_' symbols which we provided, compiler will reformate that number as original number and compiler will process that number as original number.

Number Systems in Java:

In general, in any programming language, to represent numbers we have to use a particular system .

There are four types of number systems in programming languages.

1. Binary Number Systems[BASE-2]
2. Octal Number Systems[BASE-8]
3. Decimal Number Systems[BASE-10]
4. Hexa Decimal Number Systems[BASE-16]

In java , all number systems are allowed, but, the default number system in java applications is "Decimal Number Systems".

Binary Number Systems [BASE-2]:

If we want to represent numbers in Binary number system then we have to use 0's and 1's, but, the number must be prefixed with either '0b' or '0B'.

int a=10;-----> It is not binary number, it is decimal num.
int b=0b10;---> valid
int c=0B1010;---> valid
int d=0b1012;---> Invalid, 2 symbol is not binary numbers alphabet.

Note: Binary Number system is not supported by all the java versions upto JAVA6, but , JAVA7 and above versions are supporting Binary Number Systems, because, it is a new feature introduced in JAVA7 version.

Octal Number Systems [BASE-8]:

If we want to prepare numbers in Octal number System then we have to use the symbols like 0,1,2,3,4,5,6 and 7, but, the number must be prefixed with '0'[zero].

EX:
int a=10; ------> It is decimal nhumber, it is not octal number.
int b=012345;---> Valid
int c=O234567;---> Invalid, number is prefixed with O, not zero
int d=04567;----> Valid
int e=05678;----> Invalid, 8 is not octal number systems alphabet.

Decimal Number Systems [BASE-10]:

If we want to represent numbers in Decimal number system then we have to use the symbols like 0,1,2,3,4,5,6,7,8 and 9 and number must not be prefixed with any symbols.
EX:
int a=10;----> Valid

int b=20;----> Valid
int c=30;----> valid

Hexa Decimal Number Systems [BASE-16]:

If we want to prepare numbers in Hexa decimal number system then we have to use the symbols like 0,1,2,3,4,5,6,7,8,9, a,b,c,d,e and f, but the number must be prefixed with either '0x' or '0X'.

EX:
int a=10;-----> It is not hexa decimal number, it is decimal number.
int b=0x102345;---> Valid
int c=0X56789;---> Valid
int d=0x89abcd;---> valid
int e=0x9abcdefg;----> Invalid, 'g' is not in hexa decimal alphabet.

Note: If we provide numbers in all the above number systems in java applications then compiler will recognize all the numbers and their number systems on the basis of their prefix values , compiler will convert these numbers into decimal system and compilers will process that numbers as decimal numbers.

Keywords / Reserved Words

If any predefined word having both word recognization and internal functionality then that predefined word is called as Keyword.

If any predefined word having only word recognization with out internal functionality then that predefined word is called as Reserved word.

EX: goto   const.

To prepare java applications, Java has provided the following list of keywords.

Data types and Return types:
bytes, short, int, long, float, double, char, boolean, void.

Access Modifiers:
public, protected, private, static, final, abstract, native, volatile, transient, synchronized, strictfp,.....

Flow Controllers:
if, else, switch, case, default, for, while, do, break, continue, return,....

Class/ Object Related:
class,enum, extends, interface, implements, package, import, new, this, super,....

Exception Handling Related Keywords:
throw, throws, try, catch, finally

OPERATORS

Operator is a symbol, it will perform a particular operation over the provided operands.

To prepare java applications, JAVA has provided the following list of operators.

1) Arithmetic Operators:
+, -, *, /, %, ++, --

2) Assignment Operators:
=, +=, -=, *=, /=, %=,.....

3) Comparision Operators:
==, !=, <, >, <=, >=,.....

4) Boolean Logical Operators:
&, |, ^

5) Bitwise Logical Operators:
&, |, ^, <<, >>,...

6) Short-Circuit Operators:
&&, ||

7. Ternary Operator:
Expr1? Expr2: Expr3;

Ex1:

```
class Test
{
  public static void main(String[] args)
  {
    int a=10;
    System.out.println(a);
    System.out.println(a++);
```

```
        System.out.println(++a);
        System.out.println(a--);
        System.out.println(--a);
        System.out.println(a);
    }
}
```

Status: No Compilation Error
OUTPUT: 10

```
    10
    12
    12
    10
    10
```

EX: ++, --,(),*/%,+-$ Priority

```
class Test
{
    public static void main(String[] args)
    {
        int a=5;
        System.out.println(++a-++a);
    }
}
```

Status: No Compilation Error
OUTPUT: -1

EX:

```
class Test
{
    public static void main(String[] args)
    {
        int a=5;
        System.out.println((--a+--a)*(++a-a--)+(--a+a--)*(++a+a++));
    }
}
```

Status: No Compilation Error

OUTPUT: 16

```
A    B    A&B    A|B   A^B
-----------------------
T    T    T       T       F
T    F    F       T       T
F    T    F       T       T
F    F    F       F       F
```

EX:

```
class Test
{
   public static void main(String[] args)
   {
      boolean b1=true;
      boolean b2=false;

      System.out.println(b1&b1);//true
      System.out.println(b1&b2);//false
      System.out.println(b2&b1);//false
      System.out.println(b2&b2);//false

      System.out.println(b1|b1);//true
      System.out.println(b1|b2);//true
      System.out.println(b2|b1);//true
      System.out.println(b2|b2);//false

      System.out.println(b1^b1);//false
      System.out.println(b1^b2);//true
      System.out.println(b2^b1);//true
      System.out.println(b2^b2);//false
   }
}
```

```
A   B   A&B   A|B   A^B
--------------------
0   0    0     0     0
0   1    0     1     1
1   0    0     1     1
1   1    1     1     0
```

T----> 1
F----> 0

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      int a=10;
      int b=2;
      System.out.println(a&b);
      System.out.println(a|b);
      System.out.println(a^b);
      System.out.println(a<<b);
      System.out.println(a>>b);
   }
}
```

int a=10;----> 1010
int b=2;-----> 0010

a&b--->10&2--> 0010-----> 2
a|b--->10|2--> 1010-----> 10
a^b--->10^2--> 1000-----> 8

```
a<<b ---> 10<<2 -----> 00001010
                       00101000---> 40
```

--> Remove 2 symbols at left side and append 2 0's at right side.

```
a>>b ---> 10>>2 -----> 00001010
            00000010
```

--> Remove 2 symbols at right side and append 2 0's at left side.

Note: Removable Symbols may be o's and 1's but appendable symbols must be 0's.

Short-Circuit Operators:

The main intention of Short-Circuit operators is to improve java applications performance.

EX: 1.&& 2.||

| Vs ||

In the case of Logical-OR operator, if the first operand value is true then it is not required to check second operand value, directly, we can predict the result of overall expression is true.

In the case of '|' operator, even first operand value is true , still, JVM evalutes second opoerand value then only JVM will get the result of overall expression is true, here evaluating second operand value is unneccessary, it will increase execution time and it will reduve application performance.

In the case of '||' operator, if the first operand value is true then JVM will get the overall expression result is true with out evaluating second operand value, here JVM is not evaluating second operand expression unneccessarily, it will reduce execution time and it will improve application performance.

__Note:__ If the first operand value is false then it is mandatory for JVM to evaluate second operand value inorder to get overall expression result.
__EX:__

```java
class Test
{
   public static void main(String[] args)
   {
      int a=10;
      int b=10;
      if( (a++ == 10) | (b++ == 10) )
      {
         System.out.println(a+"    "+b);//OUTPUT: 11   11
      }
      int c=10;
      int d=10;
      if( (c++ == 10) || (d++ == 10) )
      {
         System.out.println(c+"    "+d);//OUTPUT: 11   10
      }
   }
}
```

__&   Vs   &&__

In the case of Logical-AND operator, if the first operand value is false then it is not required to check second operand value, directly, we can predict the result of overall expression is false.

In the case of '&' operator, even first operand value is false , still, JVM evalutes second opoerand value then only JVM will get the result of overall expression is false, here evaluating second operand value is unneccessary, it will increase execution time and it will reduve application performance.

In the case of '&&' operator, if the first operand value is false then JVM will get the overall expression result is false with out evaluating second operand value, here JVM is not

evaluating second operand expression unneccessarily, it will reduce execution time and it will improve application performance.

<u>Note:</u> If the first operand value is true then it is mandatory for JVM to evaluate second operand value inorder to get overall expression result.

<u>EX:</u>

```
class Test
{
    public static void main(String[] args)
    {
        int a=10;
        int b=10;
        if( (a++ != 10) & (b++ != 10) )
        {
        }
        System.out.println(a+"    "+b);//OUTPUT:  11    11
        int c=10;
        int d=10;
        if( (c++ != 10) && (d++ != 10) )
        {
        }
        System.out.println(c+"    "+d);//OUTPUT:  11    10
    }
}
```

2)Data Types:

Java is strictly a typed programming language, where in java applicatins before representing data first we have to confirm which type of data we representing. In this context, to represent type of data we have to use "data types".

<u>EX:</u> i = 10;----> invalid, no data type representation.
int i=10;--> Valid, type is represented then data is rpersented.

In java applications , data types are able to provide the following advatages.

1. We are able to identify memory sizes to store data.
<u>EX:</u> int i=10;--> int will provide 4 bytes of memory to store 10 value.

2.We are able to identify range values to the variable to assign.
<u>EX:</u> byte b=130;---> Invalid
byte b=125;---> Valid

Reason: 'byte' data type is providing a particular range for its variables like -128 to 127, in this range only we have to assign values to byte variables.

To prepare java applications, JAVA has provided the following data types.

1. Primitive Data Types / Primary Data types

Numeric Data Types

Integral data types/ Integer Data types:

```
    byte ------> 1 bytes ----> 0
    short------> 2 bytes-----> 0
    int--------> 4 bytes-----> 0
    long-------> 8 bytes-----> 0
```

Non-Integral Data Types:

```
    float------> 4 bytes----> 0.0f
    double-----> 8 bytes----> 0.0
```

Non-Numeric Data types:

```
   char ---------> 2 bytes---> ' ' [single space]
   boolean-------> 1 bit-----> false
```

2. User defined data types / Secondary Data types
All classes, all abstract classes, all interfaces, all arrays,......

No fixed memory allocation for User defined data types

If we want to identify range values for variablers onthe basis of data types then we have to use the following formula.

-2^{n-1} to $2^{n-1} - 1$
Where 'n' is no of bits.

EX: Data Type: byte , size= 1 byte = 8 bits.

-2^{8-1} to $2^{8-1} - 1$

-2^{7} to $2^{7} - 1$

-128 to 128 - 1

-128 to 127

Note: This formula is applicable upto Integral data types, not applicable for other data types.

To identify "min value" and "max value" for each and every data type, JAVA has provided the following two constant variables from all the wrapper classes.

MIN_VALUE and MAX_VALUE

Note: Classes representation of primitive data types are called as Wrapper Classes

Primitive Data Types	Wrapper Classes
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

EX:

```
class Test{
    public static void main(String[] args){
        System.out.println(Byte.MIN_VALUE+"----->"+Byte.MAX_VALUE);
        System.out.println(Short.MIN_VALUE+"---->"+Short.MAX_VALUE);
        System.out.println(Integer.MIN_VALUE+"----->"+Integer.MAX_VALUE);
        System.out.println(Long.MIN_VALUE+"----->"+Long.MAX_VALUE);
        System.out.println(Float.MIN_VALUE+"----->"+Float.MAX_VALUE);
        System.out.println(Double.MIN_VALUE+"----->"+Double.MAX_VALUE);
        System.out.println(Character.MIN_VALUE+"----->"+Character.MAX_VALUE);
        //System.out.println(Boolean.MIN_VALUE+"----->"+Boolean.MAX_VALUE); --> Error
    }
}
```

3)Type Casting:

The process of converting data from one data type to another data type is called as "Type Casting".
There are two types of type castings are existed in java.

1.Primitive data Types Type Casting
2.User defined Data Types Type Casting

<u>Note:</u> To perform User defined data types type casting we need either "extends" relation or "implements" relation between user defined data types.

<u>1.Primitive data Types Type Casting:</u>

The process of converting data from one primitive data type to another primitive data type is called as Primitive data types type casting.

There are two types of primitive data types type castings.

1.Implicit Type Casting
2.Explicit Type Casting

<u>Implicit Type Casting:</u>

The process of converting data from lower data type to higher data type is called as Implicit Type Casting.

To cover all the possibilities of implicit type casting JAVA has provided the following chart.

```
 1      2      4     8     4      8
byte ---> short ---> int ---> long ---> float ---> double
                ^
                |
               char
```

If we want to perform implicit type casting in java applications then we have to assign lower data type variables to higher data type variables.

<u>EX:</u>

```
byte b=10;
int i = b;
System.out.println(b+"   "+i);
```

Status: No Compilation Error

OUTPUT: 10   10

If we compile the above code, when compiler encounter the above assignment stattement then compiler will check whether right side variable data type is compatible with left side variable data type or not, if not, compiler will rise an error like "possible loss of precision". If right side variable data type is compatible with left side variable data type then compiler will not rise any error and compiler will not perform any type casting.

When we execute the above code, when JVM encounter the above assignment statement then JVM will perform the following two actions.

1.JVM will convert right side variable data type to left side variable data   type implicitly[Implicit Type Casting]
2.JVM will copy the value from right side variable to left side variable.

<u>Note:</u> Type Checking is the responsibility of compiler and Type Casting is the responsibility of JVM.

EX2:

```
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      byte b=i;
      System.out.println(i+"   "+b);
   }
}
```

Status: Compilation Error, Possible loss of precision.

EX3:

```
class Test
{
   public static void main(String[] args)
   {
      byte b=65;
      char c=b;
      System.out.println(b+"    "+c);
   }
}
```

Status: Compilation Error

EX4:

```
class Test
{
   public static void main(String[] args)
   {
      char c='A';
      short s=c;
      System.out.println(c+"   "+s);
   }
}
```

Status: Compilation Error, Possible loss of precision.
Reason: byte and short internal data representations are not compatible to convert into char.

EX5:

```
class Test
{
   public static void main(String[] args)
   {
      char c='A';
      int i=c;
      System.out.println(c+"   "+i);
   }
}
```

Status: No Compilation Error
OUTPUT: A   65

Reason: Char internal data representation is compatible with int.

EX6:

```
class Test
{
   public static void main(String[] args)
   {
      byte b=128;
      System.out.println(b);
   }
}
```

Status: Compilation Error,possible loss of precision.

Reason: When we assign a value to a variable of data type, if the value is greater the max limit of the left side variable data type then that value is treated as of the next higher data type value.

<u>Note:</u> For both byte and short next higher data type is int only.

EX7:

```java
class Test
{
   public static void main(String[] args)
   {
      byte b1=60;
      byte b2=70;
      byte b=b1+b2;
      System.out.println(b);
   }
}
```

Status: Compilation Error, possible loss of precision.

EX8:

```java
class Test
{
   public static void main(String[] args)
   {
      byte b1=30;
      byte b2=30;
      byte b=b1+b2;
      System.out.println(b);
   }
}
```

Status: Compilation Error, Possible loss of precision.
Reason: X,Y and Z are three primitive data types.

X+Y=Z

1.If X and Y belongs to {byte, short, int} then Z should be int.
2.If either X or Y or both X and Y belongs to {long, float, double} then Z should be higher(X,Y).

byte+byte=int
byte+short=int
short+int=int

byte+long=long
long+float=float
float+double=double

EX9:

```java
class Test
{
   public static void main(String[] args)
   {
      long l=10;
      float f=l;
      System.out.println(l+"    "+f);
   }
}
```

Status: No Compilation Error
OUTPUT: 10   10.0

EX10:

```java
class Test
{
   public static void main(String[] args)
   {
      float f=22.22f;
      long l=f;
      System.out.println(f+"    "+l);
   }
}
```

Status: Compilation Error, possible loss of precision.
Reason:
two class rooms
Class Room A: 25[Size] banches---> 3 members per bench---> 75
Class Room B: 50[Size] banches---> 1 member per bench----> 50

long---> 8 bytes--> less data as per its internal data arrangement.
float--> 4 bytes--> more data as per its internal data arrangement.

Due to the above reason, float data type is higher when compared with long data type so that, we are able to assign long variable4 to float variable directly, but, we are unable to assign float variable to long variable directly.

Explicit Type Casting:

The process of converting data from higher data type to lower data type is called as Explicit Type Casting.

To perform explicit type casting we have to use the following pattern.

P a = (Q) b;

(Q) → Cast operator
Where P and Q are two primitive data types, where Q must be either same as P or lower than P as per implicit type casting chart.

EX1:

```java
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      byte b=(byte)i;
      System.out.println(i+"   "+b);
   }
}
```

Status: No Compilation Error
OUTPUT: 10   10

When we compile the above code, when compiler encounter the above assignment statement, compiler will check whether cast operator provided data type is compatible with left side variable data type or not, if not, compiler will rise an error like "Possible loss of precision". If cast operator provided data type is compatible with left side variable data type then compiler will not rise any error and compiler will not perform type casting.

When we execute the above program, when JVM encounter the above assignment statement then JVM will perform two actions.

1.JVM will convert right side variable data type to cast operator provided data type.
2.JVM will copy value from right side variable to left side variable.

EX2:

```java
class Test
{
   public static void main(String[] args)
   {
```

```
        int i=10;
        short s=(byte)i;
        System.out.println(i+"   "+s);
    }
}
```

Status: No Compilation Error
OUTPUT: 10   10

EX3:

```
class Test
{
    public static void main(String[] args)
    {
        byte b=65;
        char c=(char)b;
        System.out.println(b+"   "+c);
    }
}
```

Status: No Compilation Error
OUTPUT: 65   A

EX4:

```
class Test
{
    public static void main(String[] args)
    {
        char c='A';
        short s=(short)c;
        System.out.println(c+"   "+s);
    }
}
```

Status: No Compilation Error
OUTPUT: A   65

<u>Note:</u> In Implicit type casting, conversions are not possible between char and byte, short and char, but, in explicit type casting conversions are possible in between char and byte, char and short, why because, explicit type casting is forcable type casting.

EX5:

```
class Test
{
   public static void main(String[] args)
   {
      short s=65;
      char c=(byte)s;
      System.out.println(s+"   "+c);
   }
}
```

Status: COmpilation Error.

EX6:

```
class Test
{
   public static void main(String[] args)
   {
      byte b1=30;
      byte b2=30;
      byte b=(byte)b1+b2;
      System.out.println(b);
   }
}
```

Status: Compilation Error

EX7:

```
class Test
{
   public static void main(String[] args)
   {
      byte b1=30;
      byte b2=30;
      byte b=(byte)(b1+b2);
      System.out.println(b);
   }
}
```

Status: No Compilation Error
OUTPUT: 60

EX8:

```
class Test
{
   public static void main(String[] args)
   {
      double d=22.22;
      byte b=(byte)(short)(int)(long)(float)d;
      System.out.println(b);
   }
}
```

Status: No Compilation Error
OUTPUT: 22

EX9:

```
class Test
{
   public static void main(String[] args)
   {
      int i=130;
      byte b=(byte)i;
      System.out.println(b);
   }
}
```

Status: No Compilation Error
OUTPUT: -126
Reason: REf Diagram

b = 130-127 = 3 -1 =2 -128 = -126
b = 135-127 = 8 -1 =7 -128 = -121

4.Java Statements:

Statement is the collection of expressions.

To design java applications JAVA has provided the following statements.

General Purpose Statements
Declaring variables, methods, classes,....
Creating objects, accessing variables, methods,.....

Conditional Statements:
 1. if 2.switch

Iterative Statements:
 1.for 2.while 3.do-while

Transfer statements:
 1.break 2.continue 3.return

Exception Handling statements:
 throw, try-catch-finally

Synchronized statements:
 1.synchronized methods
 2.synchronized blocks

Conditional Statements:
These statements are able to allow to execute a block of instructions under a particular condition.

EX:
1. if 2.switch

1. if:

Syntax-1:
```
if(condition)
{
  ---instructions----
}
```

Syntax-2:
```
if(condition)
{
  ---instuctions----
}
else
{
  ----instructions----
}
```

Syntax-3:
```
if(condition)
{
 ---instructions----
}
```

```
else if(condition)
{
  ---instruction----
}
else if(condition)
{
 ---instructions----
}
----
----
else
{
 ----instructions----
}
```

EX1:

```
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      int j;
      if(i==10)
      {
         j=20;
      }
      System.out.println(j);
   }
}
```

Status: Compilation Error, Variable j might not have been innitialized.

EX2:

```
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      int j;
      if(i==10)
      {
         j=20;
      }
      else
```

```
        {
            j=30;
        }
        System.out.println(j);
    }
}
```

Status: No Compilation Error
OUTPUT: 20

EX3:

```
class Test
{
    public static void main(String[] args)
    {
        int i=10;
        int j;
        if(i==10)
        {
            j=20;
        }
        else if(i==20)
        {
            j=30;
        }
        System.out.println(j);
    }
}
```

Status: Compilation Error, Variable j might not have been initialized.

EX4:

```
class Test
{
    public static void main(String[] args)
    {
        int i=10;
        int j;
        if(i==10)
        {
            j=20;
        }
        else if(i==20)
        {
```

```
        j=30;
      }
      else
      {
        j=40;
      }
      System.out.println(j);
    }
}
```

Status: no Compilation Error
OUTPUT: 20

EX5:

```
class Test
{
    public static void main(String[] args)
    {
      final int i=10;
      int j;
      if(i == 10)
      {
        j=20;
      }
      System.out.println(j);
    }
}
```

Status: No Compilation Error
OUTPUT: 20

EX6:

```
class Test
{
    public static void main(String[] args)
    {
      int j;
      if(true)
      {
        j=20;
      }
      System.out.println(j);
    }
}
```

Status: No Compilation Error

Reasons:

1.In java applications, only class level variables are having default values, local variables are not having default values. If we declare local variables in java applications then we must provide initializations for that local variables explicitlty, if we access any local variable with out having initialization explicitly then compiler will rise an error like "Variable x might not have been initialized".

EX:

```
class A{
 int i;-----> class level variable, default value is 0..
 void m1(){
  int j;---> local variable, no default value.
  System.out.println(i);// OUTPUT: 0
  //System.out.println(j);--> Error
  j=20;
  System.out.println(j);---> No Error
 }
}
```

Note: Local variables must be declared in side methods, blocks, if conditions,... and these variables are having scope upto that method only, not having scope to outside of that method. Class level variables are declare at class level that is in out side of the methods, blocks,.... these variables are having scope through out the clsas that is in all methods, in all blocks which we provided in the respective class.

2. In java, there are two types of conditional Expressions.

1. Constant Expressions
2. Variable Expressions

Constant Expressions:

These expressions includes only constants including final variables and these expressions would be evaluated by "Compiler" only, not by JVM.

EX:
1.if(10 == 10){   } ----> Constant Expression
2.if(true){   } ------> Constant Expression
3.final int i=10;
  if(i == 10){   } ----> Constant Expression

Note: If we declare any variable as final variable with a value then compiler will replace final variables with their values in the remaining program, this process is called "Constant

Folding", it is one of the code optimization tech followed by Compiler.

Variable Expressions:

These expressions are including at least one variable [not including final variables] and these expressions are evaluated by JVM, not by Compiler.

EX:

```
1.int i=10;
  int j=10;
  if( i == j ){   } ----> variable expression.

2.int i=10;
  if( i == 10 ){  } ----> Variable expression
```

switch

'if' is able to provide single condition checking by default, but, switch is able to provide multiple conditions checkings.

Syntax:

```
switch(Var)
{
   case 1:
                -----instructions-----
                break;
   case 2:
               ----instructions------
               break;
               ----
               ----
   case n:
                 ----instructions-----
                 break;
   default:
                   ----instructions-----
                   break;
}
```

Note: We will utilize switch programming element in "Menu Driven" Applications.

Stack Operations
1.PUSH
2.POP
3.PEEK
4.EXIT

```
Enter Your Option: 1
Enter element to PUSH: A
PUSH operation success

Stack Operations
1.PUSH
2.POP
3.PEEK
4.EXIT
Enter Your Option:2
POP operation success.

Stack Operations
1.PUSH
2.POP
3.PEEK
4.EXIT
Enter Your Option:3
PEEK Operation Success, TOUTPUT: A

Stack Operations
1.PUSH
2.POP
3.PEEK
4.EXIT
Enter Your Option:4
Thanks for using STACK Operations.
```

EX:

```java
class Test
{
    public static void main(String[] args)
    {
        int i=10;
        switch(i)
        {
            case 5:
                System.out.println("Five");
            break;
            case 10:
                System.out.println("Ten");
            break;
            case 15:
                System.out.println("Fifteen");
            break;
            case 20:
               System.out.println("Twenty");
            break;
            default:
               System.out.println("Default");
            break;
         }
      }
   }
```

Rules to write switch:

1.switch is able to allow the data types like byte, short, int and char.

EX: byte b=10;
switch(b)
{

}
Status: No Compilation Error

EX: long l=10;
switch(l)
{

}
Status: Compilatiopn Error.

EX:

```
class Test
{
   public static void main(String[] args)
   {
      char c='B';
      switch(c)
      {
         case 'A':
            System.out.println("Five");
         break;
         case 'B':
            System.out.println("Ten");
         break;
         case 'C':
            System.out.println("Fifteen");
         break;
```

```
            case 'D':
                System.out.println("Twenty");
            break;
            default:
                System.out.println("Default");
            break;
        }
    }
}
```

Status: no Compilation Error

<u>Note:</u>
Upto JAVA6 version, switch is not allowing "String" data type as parameter, "JAVA7" version onwards switch is able to allow String data type.

<u>EX:</u>

```
class Test
{
    public static void main(String[] args)
    {
        String str="BBB";
        switch(str)
        {
            case "AAA":
                System.out.println("AAA");
            break;
            case "BBB":
                System.out.println("BBB");
            break;
            case "CCC":
                System.out.println("CCC");
            break;
            case "DDD":
                System.out.println("DDD");
            break;
            default:
                System.out.println("Default");
            break;
        }
    }
}
```

Status: No Compilation Error
OUTPUT: BBB

2. In switch, all cases and default are optional, we can write switch with out cases and with default, we can write switch with cases and with out default, we can write switch with out both cases and default.

EX:

```
int i=10;
switch(i)
{

}
```

Status: No Compilation Error
OUTPUT: No Output.

EX:

```
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      switch(i)
      {
         default:
            System.out.println("Default");
            break;
      }

   }
}
```

Status:No Compilation Error
OUTPUT: Default

EX:

```
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      switch(i)
      {
         case 5:
            System.out.println("Five");
         break;
         case 10:
```

```
            System.out.println("Ten");
         break;
         case 15:
            System.out.println("Fifteen");
         break;
         case 20:
            System.out.println("Twenty");
         break;
      }
   }
}
```

Status: No Compilation Error
OUTPUT: Ten

<u>EX:</u>

```
class Test
{
   public static void main(String[] args)
   {
      int i=50;
      switch(i)
      {
         case 5:
            System.out.println("Five");
         break;
         case 10:
            System.out.println("Ten");
         break;
         case 15:
            System.out.println("Fifteen");
         break;
         case 20:
            System.out.println("Twenty");
         break;
      }
   }
}
```

Status: No Compilation Error
OUTPUT: No Output

In switch, "break" statement is optional, we can write switch with out break statement, in this context, JVM will execute all the instructions continously right from matched case untill it encounter either break statement or end of switch.

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      int i=10;
      switch(i)
      {
         case 5:
            System.out.println("Five");

         case 10:
            System.out.println("Ten");

         case 15:
            System.out.println("Fifteen");

         case 20:
            System.out.println("Twenty");

         default:
            System.out.println("Default");
      }
   }
}
```

Status: No Compilation Error

```
   OUTPUT:     Ten
   Fifteen
   Twenty
   Default
```

In switch, all case values must be provided with in the range of the data type which we provided as parameter to switch.

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      byte b=126;
      switch(b)
      {
         case 125:
```

```
            System.out.println("125");
        break;
        case 126:
            System.out.println("126");
        break;
        case 127:
            System.out.println("127");
        break;
        case 128:
            System.out.println("128");
        break;
        default:
            System.out.println("Default");
        break;
    }
  }
}
```

Status: Compilation Error

In switch, all case values must be constants including final variables, they should not be normal variables.

EX:

```
class Test
{
    public static void main(String[] args)
    {
        final int i=5,j=10,k=15,l=20;
        switch(10)
        {
            case i:
                        System.out.println("Five");
                        break;
            case j:
                        System.out.println("Ten");
                        break;
            case k:
                        System.out.println("Fifteen");
                        break;
            case l:
                        System.out.println("Twenty");
                        break;
            default:
                            System.out.println("Default");
```

```
                      break;
        }
    }
}
```

Status: No Compilation Error
OUTPUT: 10

Note: in the above example, if we remove final keyword then compiler will rise an error.

Iterative Statements:

These statements are able to allow JVM to execute a set of instructions repeatedly on the basis of a particular condition.

EX: 1.for   2.while   3.do-while

1) for

Syntax:
```
for(Expr1; Expr2; Expr3)
{
        ----instructions-----
}
```

EX:

```
class Test
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;i++)
      {
         System.out.println(i);
      }
   }
}
```

Status: No Compilation Error

Output:
```
0
1
---
---
9
```

Expr1-----> 1 time
Expr2-----> 11 times
Expr3-----> 10 times
Body -----> 10 times

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      int i=0;
      for(;i<10;i++)
      {
         System.out.println(i);
      }
   }
}
```

Status: No Compilation Error
OUTPUT: 0 ---- 9

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      int i=0;
      for(System.out.println("Hello");i<10;i++)
      {
         System.out.println(i);
      }
   }
}
```

Status: No Compilation Error
OUTPUT: Hello 0 1 ---- 9

Reason:

In for loop, Expr1 is optional, we can write for loop even with out Expr1 , we can write any statement like System.out.println(--) as Expr1, but, always, it is suggestible to provide loop variable declaration and initialization kind of statements as Expr1.

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      for(int i=0, float f=0.0f ;i<10 && f<10.0f; i++,f++)
      {
         System.out.println(i+"   "+f);
      }
   }
}
```

Status: Compilation Error

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      for(int i=0, int j=0 ;i<10 && j<10; i++,j++)
      {
         System.out.println(i+"   "+j);
      }
   }
}
```

Status: Compilation Error

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      for(int i=0, j=0 ;i<10 && j<10; i++,j++)
      {
         System.out.println(i+"   "+j);
      }
   }
}
```

Status: No Compilation Error

```
OUTPUT:  0   0
         1   1
         2   2
         ------
         ------
         9   9
```

Reason:

In for loop, Expr1 is able to allow at most one declarative statement, it will not allow more than one declarative statement, we can declare more than one variable within a single declarative statement.

EX:

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

Status: No Compilation Error
OUTPUT: Infinite Loop

EX:

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

Status: Compilation Error

Reason: In for loop, Expr2 is optional, we can write for loop even without Expr2, if we write for loop without Expr2 then for loop will take "true" value as Expr2 and it will make

for loop as an infinite loop. If we want to write any statement as Expr2 then that statement must be boolean statement, it must return either true value or false value.

EX:

```
class Test
{
   public static void main(String[] args)
   {
      System.out.println("Before Loop");
      for(int i=0;i<=0 || i>=0 ;i++)
      {
         System.out.println("Inside Loop");
      }
      System.out.println("After Loop");
   }
}
```

Status: No Compilation Error

```
OUTPUT:  Before Loop
         Inside Loop
         ----
         ----
```

EX:

```
class Test
{
   public static void main(String[] args)
   {
      System.out.println("Before Loop");
      for(int i=0; true ;i++)
      {
         System.out.println("Inside Loop");
      }
      System.out.println("After Loop");
   }
}
```

Status: Compilation Error, Unreachable Statement

EX:

```
class Test
{
   public static void main(String[] args)
   {
      System.out.println("Before Loop");
      for(int i=0;;i++)
      {
         System.out.println("Inside Loop");
      }
      System.out.println("After Loop");
   }
}
```

Status: Compilation Error, Unreachable Statement

Reasons:
In java applications, if we provide any statement immediately after infinite loop then that statement is called as "Unreachable Statement". If compiler identifies the provided loop as an infinite loop and if compiler identifies any followed statement for that infinite loop then compiler will rise an error like "Unreachable Statement". If compiler does not aware the provided loop as an infinite loop then there is no chance for compiler to raise "Unreachable Statement Error".

Note: Deciding whether a loop as an infinite loop or not is completely depending on the conditional expression, if the conditional expression is constant expression and it returns true value always then compiler will recognize the provided loop as an infinite loop. If the conditional expression is variable expression then compiler will not recognize the provided loop as an infinite loop even the loop is really infinite loop.

```
int i=0;
for(int i=10; i<10;i++)
{
}
```

EX:

```
class Test
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;)
      {
         System.out.println(i);
```

```
            i=i+1;
        }
    }
}
```

Status: No Compilation Error
OUTPUT: 0 ,1 9

<u>EX:</u>

```
class Test
{
    public static void main(String[] args)
    {
        for(int i=0;i<10;System.out.println("Hello"))
        {
            System.out.println(i);
            i=i+1;
        }
    }
}
```

Status: No Compilation Error

<u>Output:</u>
```
0
Hello
1
Hello
----
----
9
Hello
```

<u>Note:</u> In for loop, Expr3 is optional, we can write for loop without expr3, we can provide any statement as expr3, but, it is suggestible to provide loop variable increment/decrement kind of statements as expr3.

<u>EX:</u>

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

Status: Compilation Error

EX:

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

Status: No Compilation Error
OUTPUT: No Output, but, JVM will be in infinite loop

EX:

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

Status: No Compilation Error

OUTPUT: No Output, but, JVM will be in infinite loop

Reason: In for loop, if we want to write single statement in body then curly braces [{ }] are optional, if we don't want to write any statement as body then we must provide either ; or curly braces to the for loop.
In general, we will utilize for loop when we aware no of iterations in advance before writing loop.

EX:

```
int[] a={1,2,3,4,5};
int size=a.length;
for(int i=0; i<size; i++)
{
  System.out.println(a[i]);
}
```

If we use above for loop to retrieve elements from arrays and from Collection objects then we are able to get the following problems.

1) We have to manage a separate loop variable.

2) At each and every iteration we have to execute a conditional expression that is expr2, which is more streangthful operation and it may consume more no of system resources [Memory and execution time].
3) At each and every iteration we have to perform either increment operation or decrement operation explicitly.

4) In this approach, we are retrieving elements from arrays on the basis of the index value , if it is not proper then there may be a chance to get an exception like "java.lang.ArrayIndexOutOfBoundsException".

The above drawbacks are able to reduce java application performance.

To overcome the above problems and to improve java application performance explicitly we have to use "for-Each" loop provided by JDK5.0 version.

Syntax:

```
for(Array_Data_Type var: Array_Ref_Var)
{
  ----
}
```

EX:

```
int[] a={1,2,3,4,5};
for(int x: a)
{
  System.out.println(x);
}
```

EX:

```
class Test
{
    public static void main(String[] args)
    {
        int[] a={1,2,3,4,5};
        for(int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }
```

```
        System.out.println();
        for(int x: a)
        {
           System.out.println(x);
        }
    }
}
```

Status: No Compilation Error

Output:

```
1
2
3
4
5

1
2
3
4
5
```

2) While Loop:

In java applications, when we are not aware the no of iterations in advance before writing loop there we have to utilize 'while' loop.

EX:

```
D:\javaapps>java Insert.java
Enter Employee Number : 111
Enter Employee Name   : AAA
Enter Employee Salary : 5000
Enter Employee Address: Hyd
Employee Inserted Successfully
One more Employee[yes/no]?  :yes

Enter Employee Number : 222
Enter Employee Name   : BBB
Enter Employee Salary : 6000
Enter Employee Address: Hyd
Employee Inserted Successfully
One more Employee[yes/no]? : yes

Enter Employee Number : 333
Enter Employee Name   : CCC
```

```
Enter Employee Salary : 7000
Enter Employee Address: Hyd
Employee Inserted Successfully
One more Employee[yes/no]? no

--- Thank You for using this appl.----
```

Syntax:

```
while(Condition)
{
  ---instructions-----
}
```

EX:

```java
class Test
{
    public static void main(String[] args)
    {
        int i=0;
        while(i<10)
        {
            System.out.println(i);
            i=i+1;
        }
    }
}
```

EX:

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

Status: Compilation Error
Reason: Conditional Expression is mandatory.

EX:

```
class Test
{
   public static void main(String[] args)
   {
      System.out.println("Before Loop");
      while(true)
      {
         System.out.println("Inside Loop");
      }
      System.out.println("After Loop");
   }
}
```

Status: Compilation Error, Unreachable Statement.

EX:

```
class Test
{
   public static void main(String[] args)
   {
      System.out.println("Before Loop");
      int i=0;
      while(i<=0 || i>=0)
      {
         System.out.println("Inside Loop");
      }
      System.out.println("After Loop");
   }
}
```

3) do-while:

Q) What are the differences between while loop and do-while loop?

1) While loop is not giving any guarantee to execute loop body minimum one time. do-while loop will give guarantee to execute loop body minimum one time.

2) In case of while, first, conditional expression will be executes, if it returns true then only loop body will be executed.

In case of do-while loop, first loop body will be executed then condition will be executed.

3) In case of while loop, condition will be executed for the present iteration.
In case of do-while loop, condition will be executed for the next iteration.

Syntaxes:

```
while(Condition)
{
  ---instructions-----
}

do
{
   ---instructions---
}

While(Condition):
```

EX:

```
class Test
{
    public static void main(String[] args)
    {
        int i=0;
        do
        {
            System.out.println(i);
            i=i+1;
        }
        while (i<10);
    }
}
```

Status: No Compilation Error
OUTPUT: 0, 1, 2,.... 9

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      System.out.println("Before Loop");
      do
      {
         System.out.println("Inside Loop");
      }
      while (true);
      System.out.println("After Loop");
   }
}
```

Status: Compilation Error, Unreachable Statement

4.Transfer Statements:

These statements are able to bypass flow of execution from one instruction to another instruction.

EX: 1.break 2.continue 3.return

1) break:

break statement will bypass flow of execution to outside of the loops or outside of the blocks by skipping the remaining instructions in the current iteration and by skipping all the remaining iterations.

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;i++)
      {
         if(i==5)
         {
            break;
         }
         System.out.println(i);
      }
   }
```

```
}
```

Status: No Compilation Error

Output:

```
0
1
2
3
4
```

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      System.out.println("Before loop");
      for(int i=0;i<10;i++)
      {
         if(i==5)
         {
            System.out.println("Inside loop, before break");
            break;
            System.out.println("Inside loop, after break");
         }
      }
      System.out.println("After Loop");
   }
}
```

Status: Compilation Error, unreachable Statement
Reason: If we provide any statement immediatly after break statement tghen that statement is Unreachable Statement, where compiler will rise an error.

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;i++)// Outer loop
      {
         for(int j=0;j<10;j++)// Nested Loop
         {
```

```
                if(j==5)
                {
                    break;
                }
                System.out.println(i+"   "+j);
            }
            // Out side of nested Loop
        }
    }
}
```

Status: No Compilation Error

Output:

```
0 0
0 1
0 2
0 3
0 4
-------
-------
-------
9  0
9  1
9  2
9  3
9  4
```

Note: If we provide "break" statement in nested loop then that break statement is applicable for only nested loop, it will not give any effect to outer loop.

In the above context, if we want to give break statement effect to outer loop , not to the nested loop then we have to use "Labelled break" statement.

Syntax:
break label;

Where the provided label must be marked with the respective outer loop.

EX:

```
class Test
{
    public static void main(String[] args)
    {
        l1:for(int i=0;i<10;i++)// Outer loop
        {
            for(int j=0;j<10;j++)// Nested Loop
            {
               if(j==5)
               {
                  break l1;
               }
               System.out.println(i+"   "+j);
            }
            // Out side of nested Loop
         }
         //Out side of outer loop
      }
   }
```

Status: no Compilation Error

```
OUTPUT: 0  0
        0  1
        0  2
        0  3
        0  4
```

2) Continue:

This transfer statement will bypass flow of execution to starting point of the loop by skipping all the remaining instructions in the current iteration inorder to continue with next iteration.

EX:

```
class Test
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;i++)
      {
         if(i == 5)
         {
            continue;
         }
         System.out.println(i);
      }
   }
}
```

Status: No Compilation Error

OUTPUT:

```
0
1
2
3
4
6
7
8
9
```

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      System.out.println("before Loop");
      for(int i=0;i<10;i++)
      {
         if(i == 5)
         {
            System.out.println("Inside Loop, before continue");
            continue;
            System.out.println("Inside Loop, After continue");
         }
      }
      System.out.println("After loop");
   }
}
```

Status: Compilation Error, Unreachable Statement.
Reason: If we provide any statement immediatly after continue statement then that statement is unreachable statement, where compiler will rise an error.

EX:

```java
class Test
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;i++)
      {
         for(int j=0;j<10;j++)
         {
            if(j==5)
            {
               continue;
            }
            System.out.println(i+"   "+j);
         }
      }
   }
}
```

Status: No Compilation Error

OUTPUT:

```
---
--
---
--
```

If we provide continue statement in netsted loop then continue statement will give effect to nested loop only, it will not give effect to outer loop

EX:

```
class Test
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;i++)
      {
         for(int j=0;j<10;j++)
         {
            if(j==5)
            {
               continue;
            }
            System.out.println(i+"   "+j);
         }
      }
   }
}
```

In the above context, if we want to give continue statement effect to outer loop, not to the nested loop then we have to use labelled continue statement.

Syntax:

continue label;

Where the provided label must be marked with the respective outer loop.

EX:

```
class Test
{
   public static void main(String[] args)
   {
      l1:for(int i=0;i<10;i++)
      {
         for(int j=0;j<10;j++)
         {
            if(j==5)
            {
               continue l1;
            }
            System.out.println(i+"   "+j);
         }
      }
   }
}
```

Status: No Compiloation Error

OUTPUT:

```
0  0
0  1
0  2
0  3
0  4
1  0
1  1
1  2
1  3
1  4
---
---
---
9  0
0  1
9  2
9  3
9  4
```