Variables
  a. Based on the type of value it holds
      1. primitive
      2. instance
 b. Based on the behaviour and its position of declaration
      1. instance variables.[jvm will give default value]
      2. static variables. [jvm will give default value]
      3. local variables.   [jvm will not give, programmer should initialize before using]

local variables
+++++++++++++++
 => Sometimes to meet programming requirement the developer will declare variables inside a method or a block,such type of variables are called as "Local Variables".
 => These variables are also called as "Temporary variables/stack variables".
 => Local variables will be created during the method execution inside stack and they will be destoryed once the control comes out of method execution.
 => Scope of local varible is limited only to that method or to that block, if we try to access them outside the block or method it would result in "CompileTime Error".

eg#1.
```java
class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
                  int x = 10;
                  int y = 20;

                  add(x,y);//local variables
      }
      static void add(int a, int b)
      {
                  int c = a+b;//local variables
                  System.out.println(c);
      }
}
```

eg#2.
```java
class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            try
            {
                  int i = Integer.parseInt("10");
            }
            catch (Exception e)
            {
                  System.out.println(i);
            }
      }

}
```

```
Note: For local variables jvm will not give any default value, we need to initalise
the variable value depending on the type before
      intializing it, otherwise it would result in "CompileTimeError".


eg#3.
class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            int x;
            System.out.println("hello :: "+x);//CE
      }
}

Note:It is not a convention to initialize the local variable inside the block,
because there is no gaurantee that a block will be
     executed and it will be initialized at runtime.
     It is suggestible to initialize the local variable at the time of declaration
itself with a default value depending upon the
     datatype.

class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            int x ;
            if(args.length > 0 )
                  x = 10;
            System.out.println(x);//CE
      }
}

eg#5.
class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            int x ;
            if(args.length > 0 )
                  x = 10;
            else
                  x = 20;

            System.out.println(x);
      }
}
javac Test.java
java Test 100
  x = 10
java Test
  x= 20

How many access modifers are there in java?
  public,private,protected
  static, strictfp, synchronized
  final, abstract,native
  transient, volatile

Note: The only access modifier which is applicable at the local variable level is
"final",if we try to use any other access modifier
       it would result in "CompiletimeError".

eg#1.
class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            final int x = 100;
      }
}

Note:
eg#1.
class Test
{
      //instance variale
      int[] a ; // a = null

      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            Test t= new Test();
            System.out.println(t.a);//null
            System.out.println(t.a[0]);//NPE

      }
}

eg#2.
class Test
{
      //instance variale
      int[] a =new int[3];  // a ---> [0]=0 [1]=0 [2]=0

      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            Test t= new Test();
            System.out.println(t.a);//[I@...
            System.out.println(t.a[0]);//0

      }
}

eg#3.
class Test
{
      //instance variale
      static int[] a =new int[3];  // a ---> [0]=0 [1]=0 [2]=0

      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            System.out.println(a);//[I@...
            System.out.println(a[0]);//0
      }
}

eg#4.
class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            int[] a;
            System.out.println(a);//CE
            System.out.println(a[0]);
      }
}

eg#5.
class Test
{
      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            int[] a =new int[3];
            System.out.println(a);//[I@...
            System.out.println(a[0]);//0
      }
}

Note: combination of varibles can be of the following types
      a. instance -> primitive, reference
      b. static   -> primitive, reference
      c. local    -> primitive, reference

class Test
{
      int[] a =new int[3]; //instance-reference
      static int x= 100;//static-primitive

      //Pre-Defined Method[Entry point/Driving Code]
      public static void main(String[] args)
      {
            String name = "sachin";//local-reference
      }
}
```

+++++++++++++
Polymorphism
++++++++++++

 1. Overloading

```
=> Two or more methods are said to overloaded methods iff they  have the same
methodname but change in the argument types.
=> In c language we did'nt had this Overloading concept so as a C programer for
same tasks with different argument types, programmer should
   remembe mulitple method names like
      a. abs()  -> for int type
      b. labs() -> for long type
      c. fabs() -> for float type
          ;;;

=> Remembering mulitple method names was challenging for developers.
=> In java we have concept of Overloading, where we can write one method name for
same tasks with different argument types.
=> overloading concept reduces the complexity of programming in java.
=> Overloading refers to "CompileTime Polymorphism".

eg#1.
class Calculator
{
      public void add(int a, int b)
      {
            System.out.println(a+b);

      }
      public void add(int a,int b, int c)
      {
            System.out.println(a+b+c);
      }

      public void add(double a, double b)
      {
            System.out.println(a+b);
      }

      public void add(float a, float b)
      {
            System.out.println(a+b);
      }
}
class Test
{
      public static void main(String[] args)
      {
                  Calculator c = new Calculator();
                  c.add(10,20);
                  c.add(10,20,30);
                  c.add(20.5,30.5);
                  c.add(20.5f,30.5f);
      }
}

+++++++
Output
++++++
D:\OctBatchMicroservices>javac Test.java
D:\OctBatchMicroservices>java Test
30
60
```

```
51.0
51.0
```

Conclusion: In overloading compiler is responsible for method resolution(binding) based on the reference type, so we say overloading as
"CompileTime Polymorphism/Eager Binding".

eg#2.

```
class Test
{
      //overloaded method
      public void methodOne(){
            System.out.println("NO arg method");
      }
      public void methodOne(int i){
            System.out.println("Int arg method");
      }
      public void methodOne(double d){
            System.out.println("double arg method");
      }
      public static void main(String[] args)
      {
            Test t= new Test();
            t.methodOne();
            t.methodOne(10);
            t.methodOne(25.5);
      }
}
```

Output

```
D:\OctBatchMicroservices>javac Test.java
D:\OctBatchMicroservices>java Test
NO arg method
Int arg method
double arg method
```

TypePromotion in Overloading
++++++++++++++++++++++++++++

```
public class Test
{
      //overloaded method
      public void methodOne(int i){
            System.out.println("int arg method");
      }
      public void methodOne(float f){
            System.out.println("float arg method");
      }
      public static void main(String[] args)
      {
            Test t= new Test();
            t.methodOne('a');//int arg method
            t.methodOne(10L);//float arg method
            t.methodOne(19.5);//CE
      }
}
```

```
eg#2.
class Test
{
      //Overloaded method
      public void methodOne(int i)
      {
            System.out.println("int arg method");
      }
      public void methodOne(Integer i)
      {
            System.out.println("Integer version");
      }
      public void methodOne(Character c)
      {
            System.out.println("Character version");
      }
      public static void main(String[] args)
      {
            Test t= new Test();
            t.methodOne('a');
      }
}
// primtiive -----> Exact Match(primitive) -----> typecasting -----> bind
// primitive ------> no exact match -> no typecasting -----> Wrapper classes ---->
bind

eg#3.

class Test
{
      //Overloaded method
      public void methodOne(String s)
      {
            System.out.println("String version");
      }

      public void methodOne(StringBuilder sb)
      {
            System.out.println("StringBuilder version");
      }
      public void methodOne(Object o)
      {
            System.out.println("Object version");
      }
      public static void main(String[] args)
      {
            Test t= new Test();
            t.methodOne("sachin");//String version
            t.methodOne(new Object());//Object version
            t.methodOne(new StringBuilder("sachin"));//StringBuilder version

//StringBuilder(child) -> null  String(child) -> null , Object(Parent) --> null
            t.methodOne(null);
      }
```

Note: While resolving Overloaded methods exact match will get high priority.
      While resolving Overloaded methods,child class will get more priority than
the parent class.
      While resolving Overloaded methods, if both the class are from child level

only, then it would result in "Compiletime Error".