Snippets
=======

```
Integer i1= new Integer(10);
Integer i2= new Integer(10);
System.out.println(i1==i2);//false

Integer i1 = 10;
Integer i2 = 10;
System.out.println(i1==i2);//true

Integer i1 =Integer.valueOf(10);
Integer i2 =Integer.valueOf(10);
System.out.println(i1==i2);//true

Integer i1 =10;
Integer i2 =Integer.valueOf(10);
System.out.println(i1==i2);//true
```

Note:
When compared with constructors it is recommended to use valueOf() method to create wrapper object.

Var arg method
==============

It stands for variable argument methods.

In java language,if we have variable no of arguments, then compulsorily new method has to be written till jdk1.4.

But jdk1.5 version, we can write single method which can handle variable no of arguments(but all of them should be of same type).

Syntax:: methodOne(dataType... varaibleName)

... => It stands for ellipse

eg#1.

```
class Demo
{
      //JDK1.5V called Var-args(ellipse)
      public void add(int... x){
            System.out.println("var-arg approach");
      }
}
class Test
{
      public static void main(String[] args)
      {
            Demo d =new Demo();
            d.add();
            d.add(10);
            d.add(10,20);
            d.add(10,20,30);
      }
}
```

Output

```
var-arg approach
var-arg approach
var-arg approach
var-arg approach
```

Note:: internally the var arg method will converted to SingleDimension Array, so we can access the

var arg method arguments using index.

eg2::

```
  class Demo
    {
      public void methodOne(int... x){
            int total=0;
            for(int i=0;i<x.length;i++){
                  total+=x[i];
            }
            System.out.println("The sum is "+total);
      }
        public static void main(String[] args){
                Demo d= new Demo();
            d.methodOne();//The sum is 0
            d.methodOne(10);//The sum is 10
            d.methodOne(10,20,30);// The sum is 60
      }
    }
```

eg3::

```
  class Demo
    {
      public void methodOne(int... x){
            int total =0;
            for(int data:x){total+=data;}
                  System.out.println("The sum is "+total);
      }
        public static void main(String[] args){
                Demo d= new Demo();
            d.methodOne();//The sum is 0
            d.methodOne(10);//The sum is 10
            d.methodOne(10,20,30);// The sum is 60
      }
    }
```

case1
=====
Valid Signatures
 1.public void methodOne(int... x)
 2.public void methodOne(int...x)
 3.public void methodOne(int ...x)

case2
=====
   We can mix normal argument with var argument
    public void methodOne(int x,int... y)
    public void methodOne(String s,int... x)

case3
=====
While mixing var arg with normal argument var arg should be always last.
 public void methodOne(int... x,int y); (invalid)

case4
=====
In an argument list there should be only one var argument
  public void methodOne(int... x,int ...y); (invalid)

case5
=====
 We can overload var arg method, but var arg method will get a call only if none of matches are found.
 (just like default statement of switch case)

eg::
```
 class Test
   {
      public void methodOne(int ...i){System.out.println("Var arg method");}
      public void methodOne(int i){System.out.println("Int arg method");}

            public static void main(String[] args)
            {
            Test t= new Test();
                  t.methodOne(10);//Int arg method
             t.methodOne();//Var arg method
             t.methodOne(10,20,30);//Int arg method
            }
 }
```

case6
=====
  public void methodOne(int... x) => it can be replace as int[] x

case7
=====
```
  public void methodOne(int... x)
  public void methodOne(int[] x)
```

  output:: CE because we cannot have two methods with same signature.

SingleDimension Array vs Var Arg method
=======================================
   1. Whereever Singledimesion array is present we can replace it with Var Arg.
        eg:: public static void main(String[] args) => String... args
   2. Whereever Var arg is present we cannot replace it with SingleDimension Array.
        eg:: public void methodOne(String... args) => String[] args (invalid)

Note:
m1(int... x)
      => we can call to this method by passing group of int values and x will become 1D array(int[] x)

m1(int[] x)
      => we can call to this method by passing 1D array only.

Note::
eg1::
```
 class Test
 {
        public void methodOne(int... x){
           for(int data: x){
                System.out.println(data);
           }
```

```
        }
      public static void main(String... args){
                  Test t= new Test();
                  t.methodOne(10,20,30);
        }
 }
```

In the above pgm x is treated as One-D array.

```
eg2::
class Test
 {
        public void methodOne(int[]... x){
           for(int[] OneD: x){
            for(int element:oneD){
              System.out.println(data);
                }
           }
         }
      public static void main(String... args){
                  Test t= new Test();
              int[] a= {10,20,30};
                  int[] b= {30,40};

                  t.methodOne(a,b);
        }
 }
```

In the above program x is treated as 2D array

Note::
methodOne(int...x)
=> we can call this method by passing a group of int values,so it becomes 1-D array.
methodOne(int[]... x)
=> we can call this method by passing a group of 1D int[], so it becomes 2-D array.

Wrapper class
==========
1. AutoBoxing
2. Widening(Implicit TypeCasting done by the Compiler (applicable for both primitive and Wrapper type))
3. Var-Args

case1::
Widening vs Autoboxing

```
class AutoBoxingAndUnboxingDemo {
      public static void methodOne(long l) {
                  System.out.println("widening");
      }
      public static void methodOne(Integer i) {
                  System.out.println("autoboxing");
      }
      public static void main(String[] args) {
                  int x=10;
                  methodOne(x); // primtive ====> do type casting====> found ==>
long(binding happens by compiler)
```

```
	}
}
output: widening

case2::Widening vs var-arg method
class AutoBoxingAndUnboxingDemo {
		public static void methodOne(long l) {
				System.out.println("widening");
		}
		public static void methodOne(int... i) {
				System.out.println("var-arg method");
		}
		public static void main(String[] args) {
				int x=10;
				 methodOne(x); // primtive ====> do type casting====>
found ==> long(binding happens by compiler)
		}
}
output: widening

Case 3: Autoboxing vs var-arg method :
class AutoBoxingAndUnboxingDemo {
	public static void methodOne(Integer i) {
		System.out.println("AutoBoxing");
	}
	public static void methodOne(int... i) {
		System.out.println("var-arg method");
	}
	public static void main(String[] args) {
		int x=10;
		methodOne(x); // int =====> implicit type casting===>
long,float,double
					// int =====> Autoboxing ===> Integer
	}
}
Output: AutoBoxing


Case4:
class AutoBoxingAndUnboxingDemo {
	public static void methodOne(Long l) {
		System.out.println("Long");
	}
	public static void main(String[] args) {
		int x=10;
		methodOne(x); //CE: can't find the method
	}
}

Note:
Widening followed by Autoboxing is not allowed in java,  but Autoboxing followed by
widening is allowed.

Case 5:
class AutoBoxingAndUnboxingDemo {
		public static void methodOne(Object o) {
			System.out.println("Object");
		}
		public static void main(String[] args) {
```

```
                int x=10;
                 methodOne(x); // AutoBoxing ===> int ===> Integer
                                      // Widening   ===> Integer ====>
Number,Object
            }
}
Output: Object
```

Which of the following declarations are valid ?

```
1. int i=10 ;//valid
2. Integer I=10 ; //AutoBoxing(valueof())
3. int i=10L ; // invalid(long===> int)
4. Long l = 10L ; //AutoBoxing(valueOf())
5. Long l = 10 ;  //AutoBoxing=> Integer====> Number,Object , so invalid
6. long l = 10 ; // valid(int===> long)
7. Object o=10 ; //AutoBoxing==> Integer ===> Number,Object  so valid
8. double d=10 ;  // valid(int===> double)
9. Double d=10 ; // AutoBoxing=> Integer====> Number,Object , so invalid
10. Number n=10;  //AutoBoxing==> Integer ===> Number,Object  so valid
```

new Vs newInstance( )

==========

1. new is an operator to create an objects , if we know class name at the beginning then we can create an object by using
   new operator .
2. newInstance( ) is a method presenting class " Class " , which can be used to create object.
3. If we don't know the class name at the beginning and its available dynamically Runtime then we should go for
   newInstance() method

eg#1.

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

            //Take the input of the classname for which object has to be created at
the runtime
            String className = args[0];

            //Load the class file explicitly
            Class c = Class.forName(className);

            //for the loaded class object is created using zero param constructor
only
            Object obj=c.newInstance();

            //Perform type casting to get Student Object
            Student std = (Student)obj;
            System.out.println(std);

      }
}
```

If dynamically provide class name is not available then we will get the RuntimeException saying ClassNotFoundException
To use newInstance( ) method compulsory corresponding class should contains no argument constructor ,
otherwise we will get the RuntimeException saying "InstantiationException".

if the argument constructor is private then it would result in "IllegalAccessException".

Note: During typecasting, if there is no relationship b/w 2 classes as parent to child then it would result in "ClassCastException".

Difference between new and newInstance( ) :

new
===
new is an operator , which can be used to create an object.
We can use new operator if we know the class name at the beginning.

```
Test t= new Test( );
```

If the corresponding .class file not available at Runtime then we will get RuntimeException saying NoClassDefFoundError,It is unchecked.
To used new operator the corresponding class not required to contain no argument constructor.

newInstance( )
============
newInstance( ) is a method , present in class Class , which can be used to create an object .
We can use the newInstance( ) method , If we don't class name at the beginning and available dynamically Runtime.

```
Object o=Class.forName(arg[0]).newInstance( );
```

If the corresponding .class file not available at Runtime then we will get RuntimeException saying ClassNotFoundException , It is checked.
To used newInstance( ) method the corresponding class should compulsory contain no argument constructor , Other wise we will get
RuntimeException saying InstantiationException.

Difference between ClassNotFoundException & NoClassDefFoundError :
1. For hard coded class names at Runtime in the corresponding .class files not available we will get NoClassDefFoundError ,
which is unchecked

```
Test t = new Test( );
```

In Runtime Test.class file is not available then we will get NoClassDefFoundError

2. For Dynamically provided class names at Runtime , If the corresponding .class files is not available then we will get the
RuntimeException saying "ClassNotFoundException".

```
Ex : Object o=Class.forname("Test").newInstance( );
```

At Runtime if Test.class file not available then we will get the ClassNotFoundException , which is checked exception.

Note:
new will create a memory on the heap area
Student => JVM will search for Student.class file in Current Working Directory
if found load the .class file data into MethodArea

During the loading of .class file
a. static variables will get memory set with default vaalue
b. static block gets executed

In the heap area, for the required object memory for instance variables is given jvm will set the default values to it

a. Execute the instance block if available

b. call the constructor to set the meaningful values to the instance varaibles.

JVM will give the address of the object to hashing algorithm which generates the hashCode for the object and that hashCode
will be returned as the reference to the programmer

new => requried class details known to compiler but not available at jvm then it would result in "NoClassDefFoundError"
newInstance() => requried class details not available at jvm then it would result in "ClassNotFoundException"