Exception handling
==========
1. try catch and finally
2. throws(best suited for checkedException)
3. throw(best sutied for uncheckedException and customException)

Syntax
=====
```
  try{
      //risky code
  }catch(XXXXX e){
      //handling code
  }finally{
      //resource releasing code
  }
```

In realtime application,we use many resources where all the resource should be closed inside finally block.
            resource => In File operations we use FileReader,FileWriter,BufferedReader,BufferedWriter
                                In JDBC Operations we use Connection,Statement,PreapredStatement,CallableStatement,....

Realtime coding
===========
```
 //declaration of resources
  try{
      //risky code
      use the resource
  }catch(XXXXX e){
      //handling code
  }finally{
      //resource releasing code
  }
```

JDK1.6V for developers
==========
eg:
```
BufferedReader br=null;
      FileReader fr =null;
      try{
            fr  = new FileReader("sample.txt");
            br = new BuffereReader();
      }catch(IOException e){
            e.printStackTrace();
      }finally{
            if(br!=null)
                  br.close();
            if(fr!=null)
                  fr.close();
      }
```

boilerplate -> the code which is repeated in multiple modules of project with no change or with small change.
                  whenever boiler plate code comes into pitcutre, we always try to avoid it by using
                        a. using JDK software higher version(jdk1.0,jdk1.2,..... jdk18)
                        b. using 3rd party API's

In JDK1.7 version they made few enhancement in the Excpetion handling area
==========
1. try with resource
2. try with multicatch block

try with resource
==========
Syntax: try(R){
                    use resource as per ur application requirement
                    if exception occurs or not occurs and if it is handled or not handled
                    still Resources will be closed once the control comes out of try block
          }catch(XXXX e){

          }

eg: without using try with resource
```
    ==========
      BufferedReader br=null;
    FileReader fr =null;
    try{
          fr  = new FileReader("sample.txt");
          br = new BuffereReader();
    }catch(IOException e){
          e.printStackTrace();
    }finally{
          if(br!=null)
                br.close();
          if(fr!=null)
                fr.close();
    }
```

eg: try with resource
```
       try(BufferedReader br= new BufferedReader(new FileReader("sample.txt"))){
                //use the resource
       }catch(IOException e){
                e.printStatckTrace();
       }
```

Advantage of try with Resource
==========
  1. The main advantage of try with resource is the resources which are a part of try block gets close automatically.
    once the control comes of out try block automatically that resource will be closed.
    while try block is getting executed
          a. exception occured and handled
          b. exception occured and not handled
     In both these cases also jvm will close the resource automatically, if we use resource with "try with resource".

  2. Using try with resource increases readabilty and reduces redundant code in our application.

Conclusions
==========

1. we can declare any no of resources ,but all the resoures should be seperated with ; symbol.

```
try(R1;R2;R3; ........){

}catch(XXXXX e){

}
```

2.From JDK1.7 for Resource Releasing logic Requirement specification they had come with an interface called "AutoCloseable" which is added in "java.lang" package.

```
interface AutoCloseable{
      public abstract void close() throws Exception;
}
public class BufferedReader implements AutoCloseable{
            @Override
            public void close(){
                  //logic of closing.
            }
}
try(BufferedReader br=new BufferedReader(new FileReader("sample.txt"))){
            //logic of using br
}
  catch(IOException e){
            //handling logic
}
```

Note:

```
try(String name =new String("sachin")){
                  //using name object
   }catch(Exception e){}
```

output: Compile Time Error

All java.io classes and java.sql classes has implemented AutoCloseable interface.

3. All resources reference variable are been made as final automatically when they are used, so we can't re-assign the reference of the Resource Variable.

CompileTime Error

==========

```
try(BufferedReader br=new BufferedReader(new FileReader("sachin.txt"))){
            br=new BufferedReader(new FileReader("kohli.txt")));
}catch(IOException e){}
```

4. Before JDK1.6

```
try{

}catch(XXXX e){

}finally{

}
```

After JDK1.7, do we need finaly block ?

Ans. no

finally block becomes dummy if we use "try with Resource".

5. JDK1.6V

```
try{
```

```
      }finally{

      }
      a. if exception does not occur => normal termination/smoothfull termination
still finally executes.
      b. if exception occurs => abnormal termination still finally executes.

     JDK1.7
      try(R){

      }
      it is possible to write only try also from JDK1.7 version ,but that try
should be associated with Resource.


JDK1.5 features
===========
1.Wrapper classes
2.Var-Args

Wrapper class
==========
      1. To wrap primtive into object form so that we can handle primtive also just
like objects
      2. To define several utility function which are requried for primitives.
      3. Wrapper classes are a part of "java.lang" package.

primtive data types
==============
  1. byte,short,int,long
  2. float,double
  3. char
  4. boolean

For every primitive type we have equavilent Wrapper class as shown below
      byte -> Byte
      short -> Short
      int    -> Integer
      long -> Long
      float -> Float
      double-> Double
      ***char  -> Character(1 constructor)
      ***boolean -> Boolean(2 constructor(String is important))

With Respect to wrapper class how is toString() implemented?

class Object{
      public String toString(){
            // returns the reference(address/hashCodeValue) of the object
      }
}
public final class Integer extends Object{

      @Override
      public String toString(){
            //returns the data present in the Object
      }
}
```

Almost every Wrapper class contains 2 constructors which takes
a. primtive type as the argument.
b. String type as the argument.

eg#1.

```
Integer i1 = new Integer(10);
System.out.println(i1);//jvm calls i1.toString()
Integer i2 = new Integer("10");
System.out.println(i2);//jvm calls i1.toString()
```

output

```
10
10
```

eg#2
If the String input is not properlyformatted,mean if it is not reprsenting any number then we will get an Exception called
"NumberFormatException"

```
  Integer i2 = new Integer("ten");//NumberFormatException
```

eg#3.
Character class contains only constructor which can take only primitive argument of type char only.

```
Character c1=new Character('a');
System.out.println(c1);

Character c1=new Character("a");//Compile Time Error.
System.out.println(c1);
```

eg#4.

```
Boolean b=new Boolean(true);
System.out.println(b);//true

Boolean b=new Boolean(false);
System.out.println(b);//false

Boolean b=new Boolean(True);//CE
Boolean b=new Boolean(False);//CE
```

Note: If we are passing String argument, then case is not important and content is important.
if the content is case insensitive String of true then it is treated as true and in all other cases it is false.
eg#5

```
Boolean b1=new Boolean("false");
System.out.println(b1);//false

Boolean b2=new Boolean("False");
System.out.println(b2);//false
```

eg#6

```
Boolean b1=new Boolean("true");
System.out.println(b1);//true

Boolean b2=new Boolean("True");
System.out.println(b2);//true
```

```
eg#7.
Boolean b1=new Boolean("yes");
System.out.println(b1);//false

Boolean b2=new Boolean("no");
System.out.println(b2);//false

Boolean b1=new Boolean("tRuE");
System.out.println(b1);//true

Boolean b2=new Boolean("TrUe");
System.out.println(b2);//true

Object class methods
===============
public class java.lang.Object {
  public java.lang.Object();
  public final native java.lang.Class<?> getClass();
  public native int hashCode();
  public boolean equals(java.lang.Object);
  protected native java.lang.Object clone() throws
java.lang.CloneNotSupportedException;
  public java.lang.String toString();
  public final native void notify();
  public final native void notifyAll();
  public final native void wait(long) throws java.lang.InterruptedException;
  public final void wait(long, int) throws java.lang.InterruptedException;
  public final void wait() throws java.lang.InterruptedException;
  protected void finalize() throws java.lang.Throwable;
  static {};
}

String toString()
      JVM will always call toString() when we try to print any reference variable.
      reference varaible can be
            a. inbuilt class
            b. user defined class

eg#1.

class Object{
      public String toString(){
            // returns the reference(address/hashCodeValue) of the object
      }
}
public final class String extends Object{

      @Override
      public String toString(){
            //returns the data present in the Object
      }
}

String name= new String("sachin");
System.out.println(name);// jvm internally calls name.toString()

eg#2.

class Object{
```

```
	public String toString(){
		// returns the reference(address/hashCodeValue) of the object
	}
}
public class Student extends Object{
	String name;

	Student(String name){
		this.name =name;
	}

	public String toString(){
		// returns the reference(address/hashCodeValue) of the object
	}
}

Student student = new Student("sachin");
   System.out.println(student);//JVM calls student.toString()

output: hashCode value of Student object


eg#3.
class Object{
	public String toString(){
		// returns the reference(address/hashCodeValue) of the object
	}
}
public class Student extends Object{
	String name;

	Student(String name){
		this.name =name;
	}

	@Override
	public String toString(){
		return this.name;
	}
}

Student student = new Student("sachin");
   System.out.println(student);//JVM calls student.toString()

output: sachin
```