Regular Expression: -
==========

java.util.regex.Pattern:

public static Pattern compile(String regex); static factory method

Pattern p= Pattern.compile("");

- [abc]----------Either 'a' or 'b' or 'c'
- [^abc] ----------Except 'a' and 'b' and 'c'
- [a-z] ----------Any lower case alphabet symbol
- [A-Z] ----------Any upper case alphabet symbol
- [a-zA-Z] ----------Any alphabet symbol
- [0-9] ----------Any digit from 0 to 9
- [a-zA-Z0-9] --------Any alphanumeric character
- [^a-zA-Z0-9] ------Any special character
- [a-z&&[^bc]] ------ a through z, except for b and c
- [a-z&&[^m-p]] ------ a through z, and not m through p

\s----space character
\d----Any digit from o to 9 [o-9]
\w----Any word character [a-zA-Z0-9]
. ----Any character including special characters.
\S----any character except for space character
\D----any character except for digit
\W----any character except for word character (special character)

re* -- 0 or any number of occurrences of the preceding expression.

re+ -- 1 or more(at least) number of occurrences of the preceding expression.

re? -- 0 or 1(at most 1)number of occurrences of the preceding expression.

re{n} -- exactly n number of occurrence of the preceding expression.

re{n,} -- n or more occurrence of the preceding expression.

re{n, m} -- at least n and at most m.

a | b -- either a or b.

public static boolean matches(String regex, CharSequence input):

We can perform pattern matcher application by using some of the methods of the String class also.

public boolean matches(String regex);
public String replaceAll(String regex, String target);
public String replace(CharSequence regex, CharSequence target);
public String[] split(String regex);

Example:

Demo.java:

```
package com.masai;

import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Demo {



	public static void main(String[] args) {

		//System.out.println(Pattern.matches("[789]{1}[0-9]{9}",
"9953038949"));//true

		String message = "Welcome Ramesh, How are you Ramesh?";

		String result= message.replaceAll("Ramesh","Amit");
```

```
                System.out.println(result);

        }

}
```

Exception handling:
==========

The main objective of exception handling is gracefull termination of our application.

--Exception always occurs at runtime time, it never occurs at compile time.

Exception: In java exceptions are the objects of some classes created by the jvm that represents the corresponding logical error.

there r 2 types of error in java:

1.syntax error: improper environment or syntactical error : it result in compile time error.

2.logical error: for some type of logical mistake, jvm would be unable to execute the application this results in logical error.

for each logical error , there r some predefined classes in java, whose object will be created by the jvm whenever it encouters that corespnding logical error.

java.lang.ArithmaticException
java.lang.ArrayOutOfBoundException
java.lang.NullPointerException

--
etc..

Whenever jvm encoutners a logical error, it creates an object of corresponding predefined exception class automatically and put(throw) that object in our application, if we don't handle that object explicitly then that obj will reaches back to the jvm. then jvm will recieve that obj and terminate our program abnormally.

--to avoid the abnormal termination we need to avoid that exception class obj should not reach back to the jvm, and by doing so we can handle that exception class obj.

--we handle that exception class obj by just assigning that object to the corresponding class variable.

--in java we have 2 keywords 1.try 2.catch using which we handle the exception class obj.

try block will recognize the exception class obj created by the jvm and its gets hold on that obj and transfer that obj to the catch block.

--in catch block we should be able to assign that obj to the appropriate class reff variable so that it will not reaches back to the jvm and therefore our program will not terminate abnormally.

Demo.java:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {

		System.out.println("Start of main...");

		try {

			System.out.println("inside try block");

			int x =100;
			int y=0;

			int z = x/y;

			System.out.println("The Result is :"+z);
```

```
                System.out.println("End of try...");
            }
            catch(ArithmeticException ae) {

                System.out.println("inside catch..");
                System.out.println(ae.getMessage());

                //alternate logic

            }
            System.out.println("End of main...");
        }

}
```

some of the important points of the exception handling:

1.the main objective of EH is normal/gracefull termination of the program.

2.we can know in which part of the program the logical error has occured.

3.we r not allowing the jvm to terminate the entire application, only the try block in which the exception obj is
generated will be terminated.

4.if no exception class obj is created inside the try block, control will never enters into the catch block.

5.once the control enters into the catch block, it executes the statement in the catch block and it moves furthur.. it will not come to the try block once again due to which that catch block is executing.

6.every try block should have atleast one catch block.

One try block can have multiple catch block also.

example:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {

		System.out.println("Start of main...");

		try {
		String s1 = "100";
		String s2 = "50";

		A a1 = null;

		int num1= Integer.parseInt(s1);
		int num2= Integer.parseInt(s2);

		int z = num1/num2;

		if(z > 5)
			a1 = new A();

		a1.funA();
		}
		catch (ArithmeticException ae) {
			System.out.println("inside AE");
		}
		catch (NumberFormatException  nfe) {
			System.out.println("inside NFE");
		}
		catch (NullPointerException npe) {
			System.out.println("inside NPE");
		}


		System.out.println("End of main...");
	}

}
```

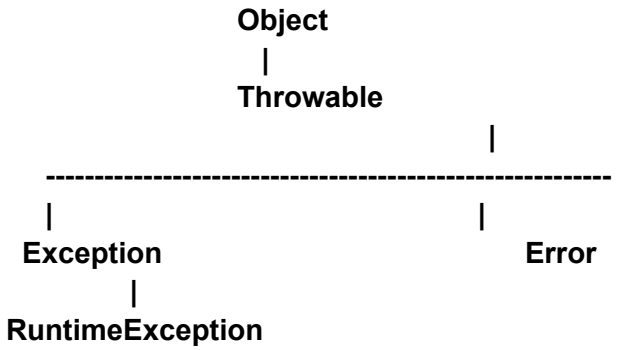
Adv of above approach:

--we have granular way to handle each type of exception, we can write alternate logic for each type exception

Dis Adv:
----

1. developer should have through knowledge about which statement may throw which type of exception, which may not possible always.

2.writing too many catch block is also not feasiable.



******Exception class is the super class of all the exception classes.**

all the above classes belongs to java.lang package.

Rule with mutiple catch block:
----
siblings can be in any order but parent must be the last

example:

```
package com.masai;

public class Demo {

	public static void main(String[] args) {

		System.out.println("Start of main...");

		try {
```

```
		String s1 = "100";
		String s2 = "0";

		A a1 = null;

		int num1= Integer.parseInt(s1);
		int num2= Integer.parseInt(s2);

		int z = num1/num2;

		if(z > 5)
			a1 = new A();

		a1.funA();
		}


		catch (ArithmeticException ae) {
			System.out.println("inside AE");

			//10000
		}
		catch (NullPointerException npe) {
			System.out.println("inside NPE");
		}
		catch (Exception e) {
			System.out.println("Inside Exp"+e);
		}


		System.out.println("End of main...");
	}

}
```

logical errors are also of two types:

1.simple logical error : - these logical error are permited to handle. they are the child of Exception classes
2.serious logical error: these logical errors are not permitted to handle, they r the child class of Error class.

```
while(true){
A a1 = new A();
}
```

--the classes which represents the simple logical error comes under the Exception class whereas the classes which represents the serious logical error comes under the Error class.

--catch block is designed in such a way that it can take only the object of Throwable class and its child classes.

in java we have all together 5 keywords in the concept of exception handling:

1.try
2.catch
3.throw
4.throws
5.finally

--Exception classes are categoried in 2 category:

1.checked exception (checked by java compiler whether we have handled that exception or not)
2.unchecked exception (not checked by the compiler)

****Note: whether exception is checked or unchecked, exception always occurs at runtime.it never occurs at compile time. compile time only occurs compilation error.

kid --------> mother----------->> exam
.java --------> java compiler ---------->.class---------->jvm-------execute the class

responsibility of java compiler:
---

1. convert .java(source code) to the .class(bytecode)
2.if inside the class if we don't put any constructor then it provides the default constructor inside our .class file
3.it will scan our .java file and check any kind of syntax error . and generate compile time error.

4.for each and every statement which may generate an exception, compiler will check whether we are handling the corresponding exception class object or not.

--in the process of checking compiler will not bother even we don't handle object of some exception classes.

--but there are some statements for which applying the concept of exception handling is mandatory at compile time itself.
if we don't handle them then compiler will generate a compile time error.

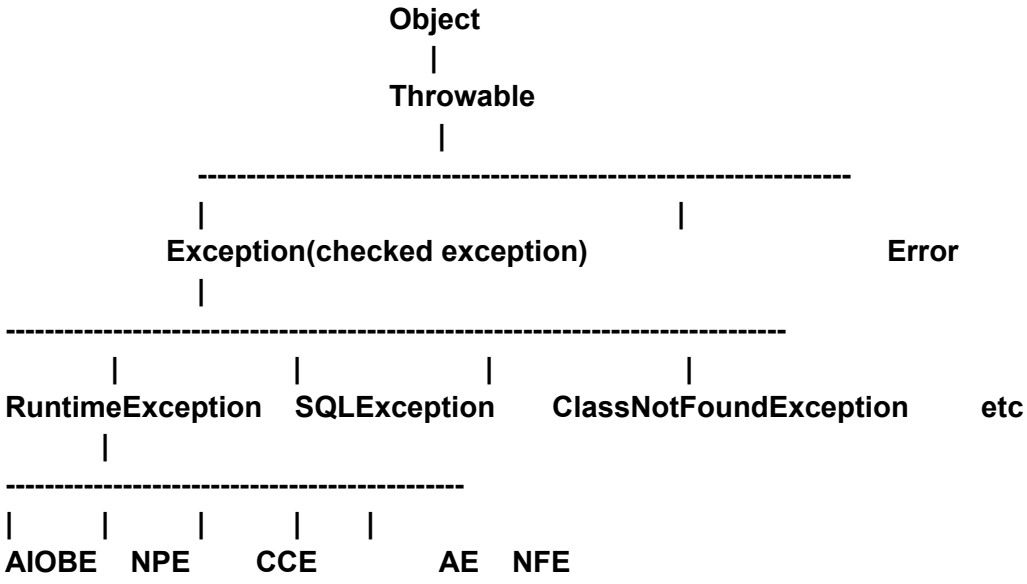
the exception class object for which compiler force us to handle them at compile time only are knwon as checked excception

ex:

ClassNotFoundException
IOException
SQLException

and for those exception classes compiler will not force us (it ignores) at compile time to handle are known as unchecked exceptions:

AE , NPE,CCE,AIOBE, NFE



RuntimeException and its child classes considered as unchecked exception (runtime exceptions)
--if some statement in our java code may generate these classes object, handling these classes object at compiler time is optional.

Exception class and its child class except RuntimeException class is knwon as checked exception,
--if some statement in our java code may generate these classes object, handling these classes object at compile time is mandatory, otherwise compiler will generate compile time error