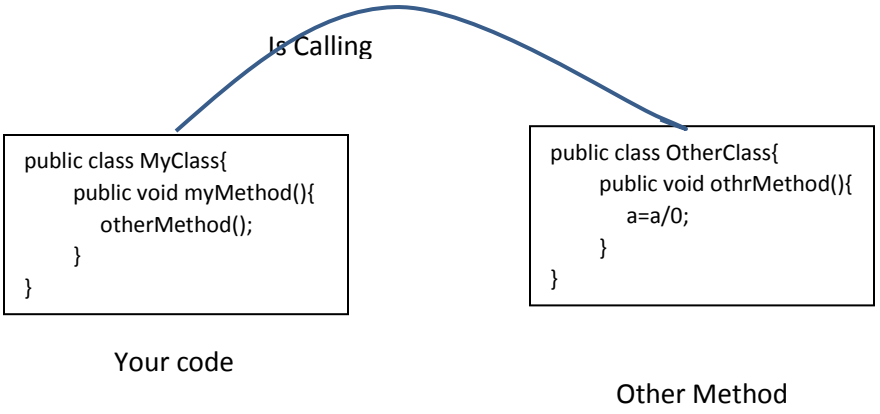
Exception Handling

No matter how good programmer we are, we cannot control the flow of execution of the program. There might be possibility of **abnormal termination** of program at runtime due to several reasons like, coding error, system error, absence of code to which we are calling, reading a file which doesn't exist etc.

Such unwanted, un-avoided and unexpected situations are called as **Exceptions.** Exceptions can terminate the normal execution of the program. Hence it is highly recommended to use **Exception Handling.**

- Exception Handling makes sure that your program terminates NORMALLY.
- Exception Handling doesn't guarantees the repair of exception, but it can continue the normal execution of the program.
- In Exception Handling we can make a decision what to do, when an exception occurs.
- Let's say your code is calling a method that you didn't write.



- In this example, your code will terminate abnormally because otherMethod() is having a risky code.
- In this case, to avoid abnormal execution of myMethod(), we have to handle exception.
- Exceptions can be handled using:
    1. try-catch block
    2. thorws keyword
- To understand exception clearly, let's discuss the **Runtime Stack Mechanism.**

Runtime Stack Mechanism:

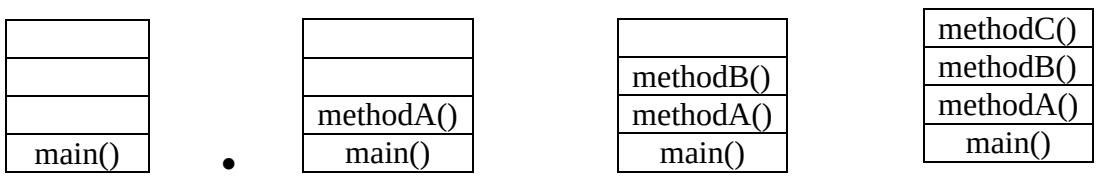
- For every thread, JVM creates a stack in which methods will be stored. (main method can be an example of thread)

- All the methods which are called by thread will be saved on stack.
- After executing the method, JVM deletes its entry from stack.
- After successfully executing all the methods, JVM destroys stack and then terminates the thread.

- Example:

```
public class TestStack{
     public static void main(String[] args){
        methodA();
     }
     Public void methodA(){
        methodB();
     }
      public void methodB(){
         methodC();
      }
      Public void methodC(){
      }
}
```

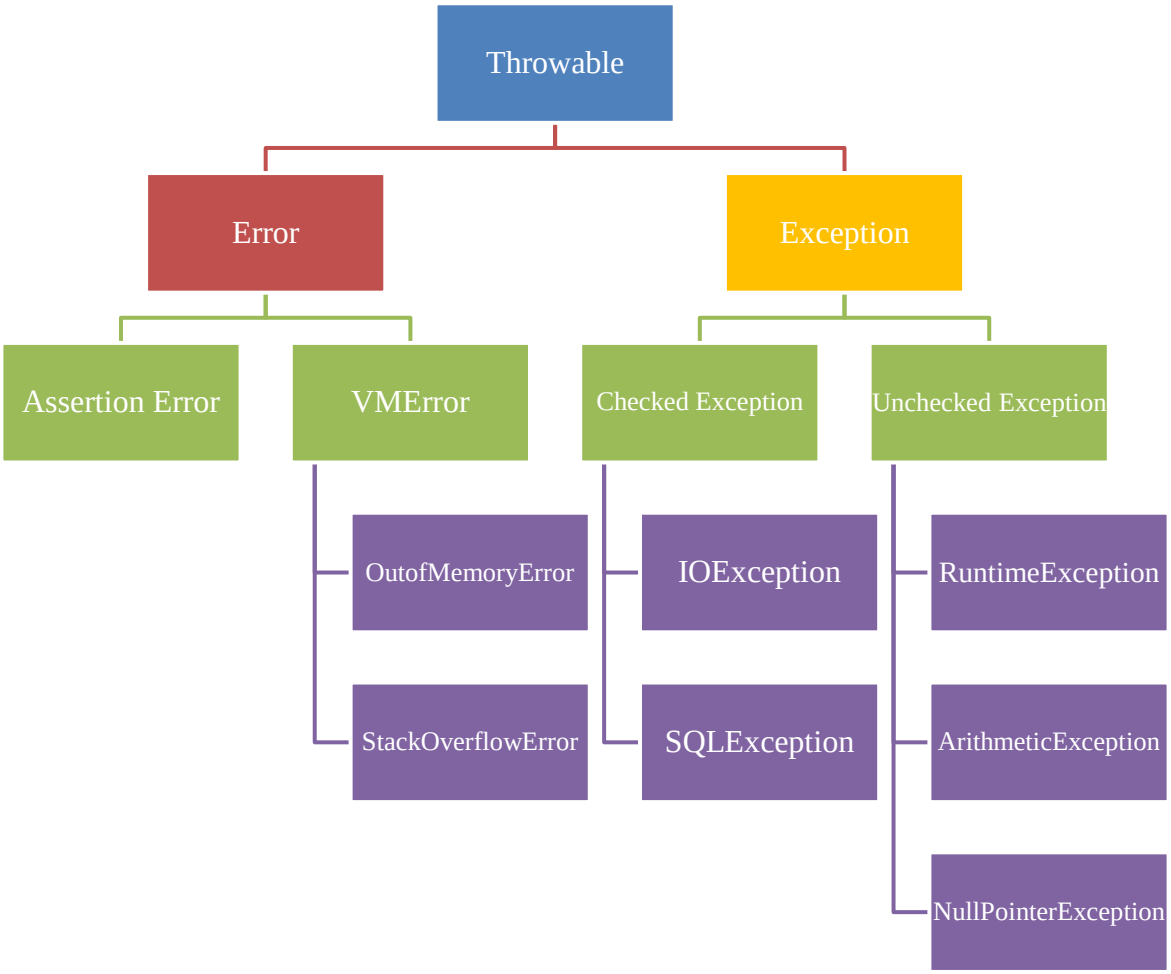
- Here, main() is calling methodA(), methodA() is calling methodB(), methodB() is calling methodC().
- So stack will become:



Step-1 • Step-2 Step-3 Step-4

- Once methods are executed, they are deleted from stack. And at last stack is deleted and thread is terminated.

Exception Hierarchy:



- Throwable class in Java acts as parent of entire exception hierarchy.
- It has two child classes viz:
    1. Error
    2. Exception
- Exceptions are caused by programmers most of the time and they are recoverable.
- Errors are caused by system and they are not recoverable.
- There are two types of Exception:
    1. Checked Exception
    2. Unchecked Exception

1. **Checked Exception:**
   - Exceptions that are not subclass of RuntimeException are called as checked exception
   - Checked exceptions are checked by compiler.
2. **Unchecked Exception:**
   - RuntimeExceptions are unchecked exceptions.
   - They are not checked by compiler.

Exception Handling using try-catch block:

- try block includes a code which may cause exception
- catch block includes a handling code if exception occurs in try block.
- Example:

```
public class ExceptionDemo{
     public static void main(String[] args){
        try{
             "Risky Code"
        }catch(Exception e){
             "Handling Code"
        }
      }
   }
```

Control Flow in try-catch block

```
 Public class ExceptionDemo{
       Public static void main(String[] args){
            try{
                   statement_1
                   statement_2
                   statement_3
             }catch(Exception e){
                   Statement_4
             }
        Statement_5
    }//End of main method
}//End of class
```

- Case_1: If there is no exception, then statements 1,2,3,5 will execute
- Case_2: If exception occurs at statement 2 and catch block matches, then statement 1,4,5 will execute
- Case_3: If exception occurs at statement 2 and catch block DOESN'T matches, then program will terminate abnormally.
- Case_4: If exception occurs at statement 4 or 5, program will terminate abnormally.
- If exception occurs at some statement in try block, then next statement will not be executed and program control will be transferred to catch block.
- Hence while writing program, risky code should be the last line of try block strictly.
- Length of try block should be as less as possible. Probably a single line which may cause exception.

try-block with multiple catch-blocks:

- It is possible to have multiple catch blocks to a single try-block.
- Each and every exception can be handled in different ways.
- Hence it is recommended to write multiple catch-blocks for single try-block.
- Example:

```
try{
   statement_1
}catch(ArrayIndexOutofBoundException ai){
}catch(IndexOutofBound ib){
}catch(Exception e){
}
```

- While writing multiple catch-blocks to single try-block we have to be very careful about the order of catch blocks.
- The catch block with very specific exception should come first and then generic one should come next.
- Above sequence of catch-blocks is allowed because, first catch has child exception class(`ArrayIndexOutofBoundException`), then next catch have its parent(`IndexOutofBound`) and then next catch have it's grand-parent (`Exception`)
- But reverse is not possible. Paren->child->Grand-child this order will give compile time error. Whereas child->parent->Grand-parent will work fine.

Finally Block:

- If we want to write a code which will perform clean-up activity, like closing file, closing database connection, closing Session etc., we cannot write such code inside try-block. Because if exception occurs there is chance of not executing the clean-up code.
- If we write clean-up code inside catch-block and if exception doesn't occur, then clean-up code will not execute.
- Hence we require something which will execute for sure regardless of exception.
- And that thing is nothing but a '*finally*' block.
- Finally block is written after all the catch blocks.
- It will execute for sure even exception occurs or doesn't occurs.
- Finally block is used to write clean-up code.
- *finally* block dominates *return* statement. Even if *return* statement is present in try or catch block, still *finally* block will execute.
- There is only one situation where finally block will not execute: When we write System.exit(0) in try block or when ever JVM shuts down.

```
try{
    "Risky Code"
 }
catch(Exception e){
    "Handling code"
}
finally{
   "Clean-up code"
}
```

Control flow with Finally Block:

If exception occurs	If exception doesn't occurs

try{
S.o.pln("Statement_1");
a=10/0; //DivideByZeroException
}
catch(Exception e){
S.o.pln("Statement_2");
}
finally{
S.o.pln("Clean-up Code");
}
	try{
	S.o.pln("Statement_1");
	a=10/5;
	}
	catch(Exception e){
	S.o.pln("Statement_2");
	}
	finally{
	S.o.pln("Clean-up Code");
	}

Output: Statement_1 Statement_2 Clean-up Code	Output: Statement_1 Clean-up Code

Printing Exception Information

- There are three methods available to print exception information.
  1. printStackTrace()
     - This method prints the exception information in following format: *Name of Exception: Description: Stack Trace*
  2. toString():
     - This method prints the exception information in following format. *Name of Exception: Description*
  3. getMessage():
     - This method prints the exception information in following format. *Description.*

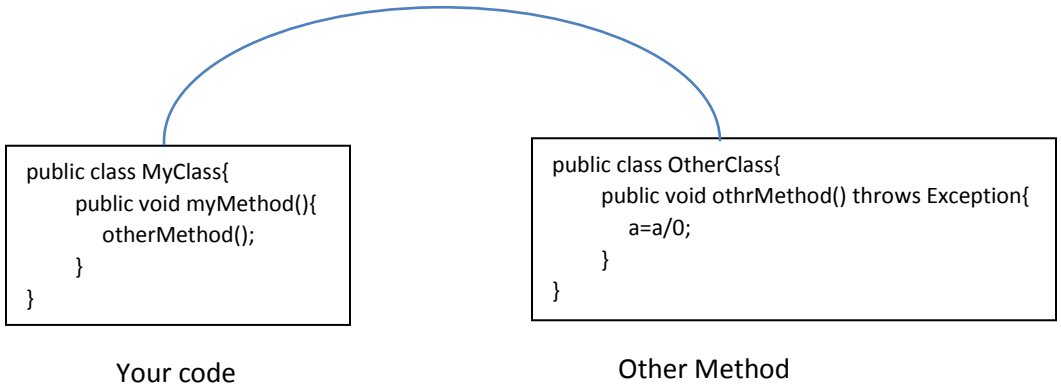
Throws Keyword:

- We have seen how to handle exceptions using try-catch blocks.
- But sometimes we don't take responsibility to handle exception in our own method.
- In such cases, if some other method calls our method, the caller method will get exception because of our method.
- Hence it is recommended to handle exception using try-catch block or to 'Propagate' exception to caller method, so that caller method will have prior-knowledge that the method I am calling may throw an exception.
- Let's say that you have borrowed a bike from your friend. And the bike may stop at any time due to engine exhaust. And if your friend doesn't inform you about this, you may not handle the situation.
- If you have prior knowledge of such exception, you can take necessary prevention steps.
- In same way, if a caller method has prior knowledge of calling a RISKY method, he may handle the exception at his end effectively.
- Hence to intimate caller method about exception we should write *throws* keyword followed by list of exceptions that a method may throw.
- Eg.

```
Public void othersMethod() throws ArrayIndexOutofBoundException,
ArithmeticException{

}
```

- In below diagram, myMethod() will have prior knowledge about exception that may cause in otherMethod().



- Hence we can summarize *throws* as: *throws* is used to propagate exception to the caller method. And we don't take responsibility to handle exception.

Throw Keyword:

- So far, we were only catching exceptions that are thrown by the Java run-time system.
- However it is possible for our program to throw an exception.
- *throw* keyword is used to throw exception explicitly.
- If we write *throw* keyword inside our method, our method will throw exception explicitly even if it doesn't occurs.
- Eg.

```
Public void myMethod(){
throw new ArithmeticException();
}
```

- In above given example, myMethod() will thorw *ArithmeticException* explicitly.
- After *throw* statement we are not allowed to write any statement. If we try to write any statement after *throw* statement, we will get compile time error.
- Throw keyword is also used to throw customized exception mentioned below.

Creating your own Exception:

- Java has provided several exception classes for different exceptions.
- But some projects have requirement to create customized exceptions.
- Eg. NotEnoughBalanceException.
- Hence java has provided a facility to programmers that they can create their own exception.
- To create our own exception, we just has to create a class with customized exception name and extend the class with *Exceptions* class or *Throwable* class.
- Eg.

```
Public class NotEnoughBalanceException extends Exception
OR
Public class NotEnoughBalanceException extends Exception
```

-:Interview Questions:-

1. What is Exception in Java?
2. What is difference between Checked and Unchecked Exception?
3. What is difference between ClassNotFoundException and NoClassDefFoundException?
4. Can we write multiple catch blocks to single try block?
5. Is it necessary to have catch block for every try block?
6. Can we write try block inside try block?
7. Can we write try block inside catch block?
8. What is the difference between final, finally and finalize?
9. How to create your own exception?