Exception handling:

Introduction:

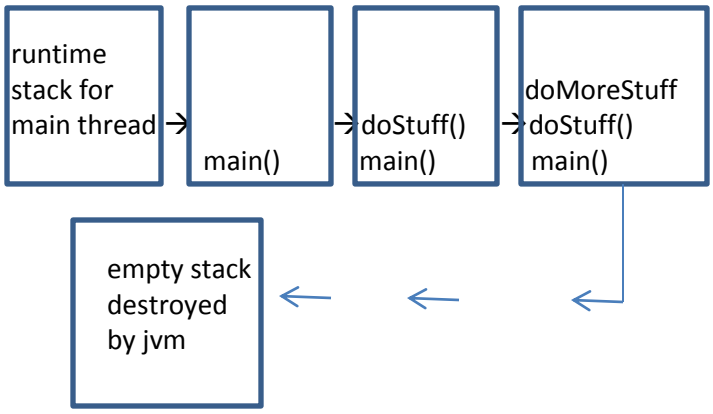
- An unexpected unwanted event that disturbs normal flow of program is called exception.
  Eg: missing file, missing property etc.
- It is highly recommended to handle exceptions and the main objective of exception handling is graceful termination of the program.
- Exception handling doesn't mean repairing an exception, we have to provide alternative way to continue rest of the program normally is the concept of exception handling.
  Eg: a program reads data from remote file located in London, at run time if London file in not available our program should not be terminated abnormally.
  We have to provide some alternate file to continue rest of the program normally.
  This way of defining alternative is nothing but Exception handling

```
try {
        //read data from file located in London
}catch(FileNotFoundException f){
        //use local file and continue
}
```

Runtime Stack mechanism:

- For every thread JVM will create a runtime stack, each and every method call performed by that thread will be stored in the corresponding stack.
- Each entry in stack is called stack frame or activation record, after completing every method call the corresponding entry from the stack will be removed.
- After completing all method calls stack will become empty and that empty stack will be destroyed by JVM just before terminating the thread.

Eg:
```
class Test{
        p s v main(String args[]){
                doStuff();
        }
        p s v doStuff(){
                doMoreStuff();
        }
        p s v doMoreStuff(){
                Sop("hello");
        }
}
```



Default Exception Handling in java:

- Inside a method if any exception occurs the method in which it is raised is responsible to create exception object by including the following information.
  Name , description, location at which exception occurs(stack trace)
- After creating exception object, method hand overs that object to the JVM.
- JVM will check whether the method contains any exception handling code or not.
- If the method doesn't contain exception handling code then JVM terminates that method abnormally and removes the corresponding entry from the stack.

- Then JVM identifies caller method and checks whether caller method contains any handling code or not.
- If the caller method doesn't contain handling code then JVM terminates that caller method also abnormally and removes the corresponding entry from the stack, this process will be continued until main method and if the main method also doesn't contain handling code then JVM terminates main method also abnormally and removes corresponding entry from the stack
- Then JVM handovers responsibility of exception handling to "***default Exception handler***" which is the part of JVM.
- Default exception handler prints Exception information in the following format and terminates program abnormally.
- Eg: Exception in thread "xxx" Name of Exception: Description
  Stack Trace

```
class Test{
p s v main(String args[]){
        doStuff();
}
p s v doStuff(){
        doMoreStuff();
}
p s v doMoreStuff(){
        sop(10/0);
}
}
```

```
Exception in thread "main" AE:division by 0
        at Test.doMoreStuff()
        at Test.doStuff()
        at Test.main()
```

Note: In a program if atleast one method terminates abnormally then the program termination is abnormal termination. Else if all methods terminated normally then only program termination is normal termination

Exception Hierarchy:

Throwable class act as root for Java Exception hierarchy.
Throwable class defines two child classes

1. Exception 2. Error

Exception:
Most of the time exceptions are caused by our program and these are recoverable.
Eg: File not Found … use a local file instead.

Errors:
Most of the time Errors are not caused by our program and these are due to lack of system resources.
Errors are non- recoverable
Eg:OutOfMemory error

Throwable
-Exception
-Error

Exception

Checked Exception Vs Unchecked Exceptions:

- The Exceptions which are checked by compiler for smooth execution of the program are called **checked Exceptions**.
  Eg: HallTicketMissingException, PenNotWorkingException, FileNotFoundException etc.
- In our program if there is any chance of raising checked exception then compulsory we should handle that checked exception(either by try-catch or throws keyword). Otherwise we will get compile time error.
- The Exceptions which are not checked by compiler, whether programmer handles or not are called **Unchecked Exceptions**.
  Eg:ArithematicException,BombBlastException etc.
- Unchecked Exceptions are rarely occurring.

Note:
-whether it is checked or Unchecked every exception occurs at runtime only. There is no chance of occurring any exception at compile time.
-RuntimeException and its child classes , Error and its child classes are UncheckedExceptions. Except these remaining are checked exception.

Fully checked vs partially checked:

- A checked exception is said to be fully checked if and only if all its child classes are also checked.
  Eg: IOException, InterupptedException.
- A checked exception is said to be partially checked if any only if some of its child classes are unchecked.
  Eg: Exception, Throwable

  Note:
  -The only possible partially checked Exceptions in java are Exception,Throwable.

  Interview:
  Describe the behavior of following exceptions:
  1. IOException→checked(fully)
  2. RuntimeException→unchecked
  3. InterruptedException→checked(fully)
  4. Error→Unchecked
  5. Throwable→checked(partially)
  6. ArithmeticException→Unchecked
  7. NullPointerException→Unchecked
  8. Exception→checked(partially)
  9. FileNotFoundException→checked(fully)

Customized Exception handling by using try-catch:

It is highly recommended to handle exceptions. The code which may raise an exception is called risky code, and we have to define that code inside **try block** and corresponding handling code inside the **catch block.**

```
try{
        //risky code
}
Catch(Exception e)
{
        //handling code
}
```

```
Eg; class Test{
        p s v main(String args[]){
                sop("stmt1");
                try{
                        sop(10/0);
                }
                catch(Exception e){
                        sop(10/2);
                }
                sop("stmt3");
        }
}
```

Control flow in try-catch:

```
try{
        stmt1;
        stmt2;
        stmt3;
}catch(x e){
        Stmt4;
}
Stmt5;
```

Case1: If there is no exception →1,2,3,5 Normal termination

Case2: If an exception raised a stmt2 and corresponding catch block matched→1,4,5 Normal Termination.

Case3: If an exception raised a stmt2 and corresponding catch block not matched→1 abnormal Termination

Case4: If exception raised at stmt4 or stmt5 →It is always Abnormal termination.

Note:

- if an Exception is raised within try block, then rest of the try block won't be executed eventhough we handled that exception. Hence within the try block we have to take only the risky code and length of try block should be as less as possible.
- In addition to try block, there may be a chance of raising exception inside catch and finally blocks.

- If any stmt which is not part of try block raises an exception then it is always abnormal termination.

Methods to print Exception Information:

Throwable class defines the following methods to print Exception information.

Method	**Printable format**
e.printStacktrace()	Name:description Stack trace
e or e.toString()	Name:description
e.getmessage()	description

```
Class Test{
        P s vmain(String args[]){
                Try{
                        Sop(10/0);
                }
                Catch(ArithmeticException e){
                        e.printStacktrace();            →java.lang.AE:/ by 0
                                                                at Test.main()
                        sop(e) or sop(e.toString());    →java.lang.AE:/ by 0
                        sop(e.getMessage();             → / by 0
                }
        }
}
```

Internally default exception handler will use printStackTrace() method to print exception information to the console.

Try with multiple catch blocks:

The way of handling an exception is varied from Exception to Exception, hence for every exception type it is highly recommended to take separate catch block i.e. try with multiple catch block is always possible and recommended to use.

```
Eg: try{
        //risky code
}
Catch(ArithematicException e){
//operations performed
}
Catch(SQLException e){
//use other DB
}
Catch(FileNotFoundException e){
//use other file
}
Catch(Exception e){
//default handling
}
```

If try with multiple catch blocks is present then the order of catch blocks is very important. We have to take child first and then parent otherwise we will get compile time error saying "Exception xxx has already been caught"

```
Eg: try{
        //risky code
}
catch(ArithematicException e){
}
Catch(Exception e){
}
```

We cannot declare two catch blocks for the same Exception otherwise we will get compile time error.

Final:

Final is the modifier applicable for class methods and variables.
If class is declared as final we cannot extend that class.
If method is final we cannot override that method in the child class.
If variable declared as final we cannot perform re-assignment for that variable.

Finally:

Finally is a block always associated with try-catch to maintain cleanup code.
The specialty of finally block is it will be executed always irrespective of whether exception is raised or not raised and whether handled or no handled.

Finalize:

Finalize is a method always invoked by garbage collector just before destroying an object to perform cleanup activities.
Once finalize() method completes immediately garbage collector destroys that object.

Note:
Finally block is responsible to perform cleanup activities related to try block i.e. whatever resources we opened as a part of try block will be closed inside finally block.
Finalize() method is responsible to perform cleanup activities related to Object i.e . whatever resources associated with the object will be de allocated before destroying an object by using finalize() method.

Various possible combinations of try-catch finally:

1. In try catch finally order is important.
2. Whenever we are writing try compulsory we should write either catch or finally, otherwise we will get compile time error i.e try without catch or finally is invalid.
3. Whenever we are writing catch block compulsory try block must be required i.e. catch without try is invalid.
4. Whenever we are writing finally block compulsory we should write try block i.e. finally without try is invalid.
5. Inside try, catch and finally blocks we can declare try, catch and finally blocks, i.e. nesting of try catch finally is allowed.
6. For try catch and finally blocks curly braces are mandatory.

Throw:

Programmer → Exception object → JVM
Sometimes we can create Exception object explicitly and handover to the JVM manually for this we have to use **Throw** key word

➔ throw new ArithmaticException("/ by 0");

Hence the main objective of throw keyword is to handover our created Exception object to the JVM manually.

```
Class Test{
        P s v main(String args[]){
        Sop(10/0);
        }
}
```

```
class Test{
p s v main(String args[]){
        throw new AE("/ by 0");
}
}
```

In case 1, main method is responsible to create exception object and handover to the JVM.
In case 2, programmer is creating exception object explicitly and handover to the JVM manually.

Note: Best use of throw keyword is for user defined exceptions or customized exceptions

Case1:
Throw e;
If "e" refers null then we will get null pointer exception.

```
Eg: class Test{
 AE e=new AE();
P s v main(String args[]_{
        Throw e;
}
}
```

→AE

```
class test{
static AE e= new AE();
p s v main(String args[]){
        throw e;
}
}
```

→ Null pointer exception

Case2:
After throw stmt we are not allowed to write any statement directly otherwise we will get compile time error saying unreachable statement.

```
class Test{
        p s v main(String args[]){
                sop(10/0);
                sop("hello");
        }
}
```

➔ RuntimeException

```
class Test{
p s v main(String args[]){
throw new AE:/ by 0
sop("hello");
}
}
```

→ compile time error

Case3:
We can use throw keyword only for throwable types. If we are trying to use for normal java objects we will get compile time error saying "incompatible types"

```
Eg: class Test{
        Psvm(String args[]){
                Throw new Test();
        }
}
```

CE:incompatible types
Found:test
Reuired :throwable

```
class Test extends RuntimeException{
        psvm(String args[]){
                throw new Test();
        }
}
```

RE: exception in thread "main" Test
        at Test.main()

Throws:

- In our program if there is a possibility of raising checked exception then compulsory we should handle that checked exception otherwise we will get compile time error saying "Unreported Exception xxx must be caught or declared to be thrown".

```
Eg:
Class test{
        Psvm(string args[]){
                printWriter  pw =new PrintWriter("abc.txt");
                pw.println("hello");
        }
}

Eg: class Test{
        Psvm(String args[]){
                Thread.sleep(10000);
        }
}
```

Unreported Exception java.lang.interrupted Exception must be caught or declared to be thrown.

- We can handle this compiletime error by using following 2 ways:
  Approach1: using try-catch
  Approach2: Using throws keyword
- We can use throws keyword to delegate the responsibility of exception handling to the caller(it may be another method or JVM). Then caller method is responsible to handle that Exception.
- Eg:

```
class Test{
        Psvm(String args[]) throws InterruptedException
        {
                Thread.sleep(10000);
        }
  }
```

Conclusions:

1. Throws keyword is required only for checked exceptions and usage of throws keyword for unchecked Exceptions there is no use or Impact.
2. Throws keyword is required only to convince compiler and usage of throws keyword doesn't prevent abnormal termination of the program.

```
Class Test{
Psvm(String args[]) throws IE{
```

```
        doStuff();
    }
    Psv doStuff() throws IE{
        doMoreStuff();
    }
    Psv doMoreStuff() throws IE{
        Thread.sleep(10000);
    }
    }
```

In the above program if we remove atleast one throws stmt, then the code won't compile.

Purpose of Throws:

1. We can use to delegate responsibility of Exception handling to the caller.(it may be a method or JVM).
2. It is required only for checked Exceptions and usage of Throws keyword for Unchecked Exceptions there is no impact.
3. It is required only to convince the compiler and usage does not prevent abnormal termination of program.

Note: It is recommended to use try-catch over Throws keyword.

Case1:

- We can use throws keyword for methods and constructors but not for classes.

Case2:

- We can use throws keyword only for Throwable types if we are trying to use for Normal java classes then we will get compile time error saying incompatible types.

Case3:

```
Class Test{
        Psvm(String args[]){
                throw new Exception();
        }
}
```

CE: Unreported Exception

```
class Test{
        Psvm(String args[]){
                throw new Error();
        }
}
```

RE: Exception in Thread main j.l.Error

Since Exception is Checked we need to handle using try-catch or throws.

Case4:

Within a try block if there is no chance of raising an Exception hen we cannot write catch block for that Exception otherwise we will get compile time error saying "Exception xxx is never thrown in body of corresponding try statement".This rule is aaplicable only for fully checked Exceptions.

Eg:

```
class Test{
        Psvm(String args[]){
                Try{
                Sop("hello");
                }catch(IOException e){}
                }
}
```

```
class Test{
        psvm(String args[]){
                try{
                sop("hello")
        }catch(Exception e){}
        }
}
```

PROG1 gives CE and PROG2 compiles fine

Exception handling Keywords Summary:

1. Try→ to maintain risky code
2. Catch→to maintain Exception handling code
3. Finally→to perform cleanup activities
4. Throw→to hand-over our created Exception object to JVM manually
5. Throws→to delegate responsibility of Exception handling to caller method

Compile time Errors in Exception Handling:

1. Unreported Exception xxx must be caught or declared to be thrown.
2. Exception xxx has already been caught
3. Exception xxx is never thrown in body of corresponding try statement.
4. Unreachable code
5. Incompatible types found:test required:java.lang.throwable
6. Try without catch or finally
7. Catch without try
8. Finally without try

Customized or user defined Exceptions:

Sometimes to meet the programing requirement we can define our own exceptions, such type of Exceptions are called customized or user defined Exceptions
Eg: TooYoungException, TooOldException, InsufficientFundsException etc.

```
Class TooYoungException extends RuntimeException
{
        TooYoungException(String s)
        {
                Super(s);
        }
}
Class TooOldException extends RuntimeException
{
        TooOldException(String s)
        {
                Super(s);
        }
}
Class CustException
{
        Psvm(String args[]){
                Int age=Interger.parseInt(args[0]);
                If(age>60){
                        Throw new TooYoungException(“message”);
                }else if(age<18){
                        Throw new TooOldException(“message”);
                }else{
                        Sop(“message”);
```

```
            }
        }
}
```

Note:

- Throw keyword is best suitable for user defined or customized Exceptions but not for pre-defined Exceptions.
- It is highly recommended to define customized Exceptions as unchecked. i.e we have to extend RuntimeException but not Exception.
- Super(s) is needed to make description available to default Exception handler.

Top 10 Exceptions in java:

- ArrayIndexOutOFBoundsException:
  -It is the child class of RuntimeException and hence is unchecked.
  -Raised automatically by the JVM whenever we are trying to access any array element with out of range index.
- NullPointerException:
  -It is the child class of RuntimeException and hence is unchecked.
  -Raised by JVM whenever we are trying to perform any operation on null.
- ClassCastException:
  -It is the child class of RuntimeException and hence is unchecked.
  -Raised automatically by the JVM whenever we are trying to typecast parent class object to child class.
- StackOverflowError:
  -It is the child class of Error and hence is unchecked.
  -Raised automatically by the compiler whenever we are using recursive method call.
- NoClassDefFoundError:
  -It is the child class of Error and hence is unchecked.
  -Raised automatically by the JVM whenever JVM is unable to find the required .class file.
- ExceptionInInitializerError:
  -It is the child class of Error and hence is unchecked.
  -Raised automatically by the JVM whenever there is any exception in execution of static variable assignment or static block.
  Eg: static int x=10/0;
  RE: ExceptionInInitializerError caused by java.lang.AE: /by 0.
- IllegalArgumentException:
  -It is the child class of RuntimeException and hence is Unchecked.
  -Raised explicitly by the programmer or API developer to indicate that a method has been invoked with incorrect argument.
  Eg: Thread t=new Thread();
  t.setPriority(15);
- NumberFormatException:
  -it is the direct child of IllegalArgumentException which child of RuntimeException and hence is unchecked.
  - Raised explicitly by the programmer or API developer to indicate that we are trying to convert a string to a number which is not formatted correctly.
  Eg: int i=Integer.parseInt(“ten”);

- IllegalStateExeption:
  -it is the child of RuntimeException and hence is unchecked.
  - Raised explicitly by the programmer or API developer to indicate that a method has been invoked at wrongtime.
  Eg:Thread t=new Thread();
  t.start();
  t.start(); //calling twice
- AssertionError:
  -it the child class of Error and hence is unchecked.
  - Raised explicitly by the programmer or API developer to indicate that a assert statement is failed.
  Eg: assert(x>10) ; //if x is not > 10 then error will be raised

  Note: 1-6 are raised by JVM and 7-10 are programmatic Exceptions

Try with Resources:

- Until ver 1.6 it was highly recommended to write finally block to close the resources which are opened as part of try block.
- The problem with this approach is, since finally block is compulsory it increases complexity and reduces code readability.
- To overcome this problem, with ver1.7 concept of try with resources was introduced.
- Eg: try(BR br=new BR(new FR("input.txt"))){
  //us br as per reuirement
  }catch(IOExeption e){
  }

Conclusion based on above example:

1. We can declare multiple resources with try, these resources should be separated by ";".
   Try(R1;R2,R3){
   }
2. All resources declared should be auto-closable. A resource is said to auto-closable if annf only if its corresponding class implements auto-closable interface which consists only one method close().
3. All resources ref variable are implicitly final and hence we cannot dp reassignment in try block. If we try doing so we get compile time Error "auto-closable resource xxx may not be assigned"
4. For using try with resources catch or finally is not required.

MultiCatch block

- Until 1.6 ver eventhough multiple different different exceptions having same handling code , for every exception type we have to write a separate catch block.
- It increases length of the code and reduces readability.
  Eg:
  Try{
  }
  Catch(AE e){

```
        Sop(e.printStackTrace());
}
Catch(IOException e){
        Sop(e.printStackTrace());
}
Catch(NPE e){
        SOP(e.getmessage());
}
Catch(InterruptedException e){
        SOP(e.getmessage());
}
```

To overcome this problem sun people introduced, multicatch block in 1.7ver

- According to this, we can write a single catch block that can handle multiple different types of Exceptions

```
Eg:
Try{
}catch(AE|IOException e){
        Sop(e.printStackTrace());
}catch(NPE|InterruptedException e){
        Sop(e.getMessage());
```

The main advantage is length of the code will be reduced and readability will be improved.

- In multicatch block there should not be any relation between, Exception types either child→parent or parent→child or same type.

Otherwise we will get compile time error

```
Try{
}
Catch(AE|Exception e){
        Se.printStackTrace();
}
```

CE:Alternatvies in a multi-catch statement cannot be relatead by subclassing.

Excepion propogation:

Inside a method if an Exception is raised and if we are not handling that Exception then, Exceptionobject will be propogated to caller.Caller method is responsible to handle Exception. This process is called Exception Propogation.

Rethrowing Exception:

We can use thiis approach to convert one Exception type to another Exception.

```
Eg:
Try{
        Sop(10/0);
}
Catch(AE e){
        Throw new NullPointerException();
}
```