1. **What are the Exception Handling Keywords in Java?**

There are four keywords used in java exception handling.

* 1. **throw**: Sometimes we explicitly want to create exception object and then throw it to halt the normal processing of the program. **throw** keyword is used to throw exception to the runtime to handle it.
  2. **throws**: When we are throwing any checked exception in a method and not handling it, then we need to use throws keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it’s caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with **main()** method also.
  3. **try-catch**: We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.
  4. **finally**: finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurrs or not.

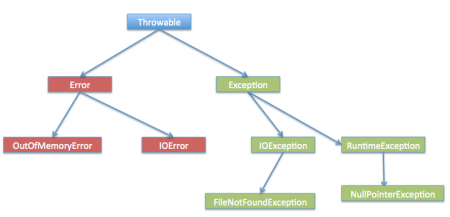
### 2.Explain Java Exception Hierarchy?

### Java Exceptions are hierarchical and [inheritance](http://www.journaldev.com/644/inheritance-in-java-example) is used to categorize different types of exceptions. Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception. Exceptions are further divided into checked exceptions and runtime exception.

1. **Errors** are exceptional scenarios that are out of scope of application and it’s not possible to anticipate and recover from them, for example hardware failure, JVM crash or out of memory error.

**Checked Exceptions** are exceptional scenarios that we can anticipate in a program and try to recover from it, for example FileNotFoundException. We should catch this exception and provide useful message to user and log it properly for debugging purpose. Exception is the parent class of all Checked Exceptions.

**Runtime Exceptions** are caused by bad programming, for example trying to retrieve an element from the Array. We should check the length of array first before trying to retrieve the element otherwise it might throw ArrayIndexOutOfBoundException at runtime. RuntimeException is the parent class of all runtime exceptions.

[](http://cdn.journaldev.com/wp-content/uploads/2013/07/exception-hierarchy.png)

### What are important methods of Java Exception Class?

**String getMessage()**

**String getLocalizedMessage()**

**String toString()**

**synchronized Throwable getCause()**

**void printStackTrace()**

### What is difference between Checked and Unchecked Exception in Java?

1. Checked Exceptions should be handled in the code using try-catch block or else main() method should use throws keyword to let JRE know about these exception that might be thrown from the program. Unchecked Exceptions are not required to be handled in the program or to mention them in throws clause.
2. Exception is the super class of all checked exceptions whereas RuntimeException is the super class of all unchecked exceptions.
3. Checked exceptions are error scenarios that are not caused by program, for example FileNotFoundException in reading a file that is not present, whereas Unchecked exceptions are mostly caused by poor programming, for example NullPointerException when invoking a method on an object reference without making sure that it’s not null.

### 79) Is there any case when finally will not be executed?

finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

80) What is difference between throw and throws?

|  |  |
| --- | --- |
| **throw keyword** | **throws keyword** |
| 1)throw is used to explicitly throw an exception. | throws is used to declare an exception. |
| 2)checked exceptions can not be propagated with throw only. | checked exception can be propagated with throws. |
| 3)throw is followed by an instance. | throws is followed by class. |
| 4)throw is used within the method. | throws is used with the method signature. |
| 5)You cannot throw multiple exception | You can declare multiple exception e.g. public void method()throws IOException,SQLException. |

### 82) Can subclass overriding method declare an exception if parent class method doesn't throw an exception ?

Yes but only unchecked exception not checked.

### 83) What is exception propagation ?

Forwarding the exception object to the invoking method is known as exception propagation.

### What is OutOfMemoryError in Java?

OutOfMemoryError in Java is a subclass of java.lang.VirtualMachineError and it’s thrown by JVM when it ran out of heap memory. We can fix this error by providing more memory to run the java application through java options.

$>java MyProgram -Xms1024m -Xmx1024m -XX:PermSize=64M -XX:MaxPermSize=256m

### What are different scenarios causing “Exception in thread main”?

Some of the common main thread exception scenarios are:

* **Exception in thread main java.lang.UnsupportedClassVersionError**: This exception comes when your java class is compiled from another JDK version and you are trying to run it from another java version.
* **Exception in thread main java.lang.NoClassDefFoundError**: There are two variants of this exception. The first one is where you provide the class full name with .class extension. The second scenario is when Class is not found.
* **Exception in thread main java.lang.NoSuchMethodError: main**: This exception comes when you are trying to run a class that doesn’t have main method.
* **Exception in thread “main” java.lang.ArithmeticException**: Whenever any exception is thrown from main method, it prints the exception is console. The first part explains that exception is thrown from main method, second part prints the exception class name and then after a colon, it prints the exception message.

### What is difference between final, finally and finalize in Java?

final and finally are keywords in java whereas finalize is a method.

final keyword can be used with class variables so that they can’t be reassigned, with class to avoid extending by classes and with methods to avoid overriding by subclasses, finally keyword is used with try-catch block to provide statements that will always gets executed even if some exception arises, usually finally is used to close resources. finalize() method is executed by Garbage Collector before the object is destroyed, it’s great way to make sure all the global resources are closed.

**Q=What is exception chaning?**

Ans=Chained *Exception* helps to identify a situation in which one exception causes another *Exception* in an application.

**For instance, consider a method which throws an *ArithmeticException*** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero.The method will throw the *ArithmeticException* to the caller. The caller would not know about the actual cause of an *Exception*. Chained *Exception* is used in such situations.

*Throwable* class has some constructors and methods to support chained exceptions. Firstly, let’s look at the constructors.

* ***Throwable(Throwable cause)****–* *Throwable*has a single parameter, which specifies the actual cause of an *Exception*.
* ***Throwable(String desc, Throwable cause)****–*this constructor accepts an *Exception* description with the actual cause of an *Exception* as well.

Next, let’s have a look at the methods this class provides:

* ***getCause()* method**–This method returns the actual cause associated with current *Exception*.
* ***initCause(\_)* method** – It sets an underlying cause with invoking *Exception*.
* Now, let’s look at the example where we will set our own *Exception* description and throw a chained *Exception*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class MyChainedException {        public void main(String[] args) {          try {              throw new ArithmeticException("Top Level Exception.")                .initCause(new IOException("IO cause."));          } catch(ArithmeticException ae) {              System.out.println("Caught : " + ae);              System.out.println("Actual cause: "+ ae.getCause());          }      }  } |

As guessed, this will lead to:

|  |  |
| --- | --- |
| 1  2 | Caught: java.lang.ArithmeticException: Top Level Exception.  Actual cause: java.io.IOException: IO cause. |

Or

public class MainClass {

    public void main(String[] args) throws Exception {

        getLeave();

    }

    public getLeave() throws NoLeaveGrantedException {

        try {

            howIsTeamLead();

        } catch (TeamLeadUpsetException e) {

             throw new NoLeaveGrantedException("Leave not sanctioned.", e);

        }

    }

    public void howIsTeamLead() throws TeamLeadUpsetException {

        throw new TeamLeadUpsetException("Team lead Upset.");

    }

}

Output:

Exception in thread "main" com.baeldung.chainedexception.exceptions

  .NoLeaveGrantedException: Leave not sanctioned.

    at com.baeldung.chainedexception.exceptions.MainClass

      .getLeave(MainClass.java:36)

    at com.baeldung.chainedexception.exceptions.MainClass

      .main(MainClass.java:29)

Caused by: com.baeldung.chainedexception.exceptions

  .TeamLeadUpsetException: Team lead Upset.

    at com.baeldung.chainedexception.exceptions.MainClass

  .howIsTeamLead(MainClass.java:44)

    at com.baeldung.chainedexception.exceptions.MainClass

  .getLeave(MainClass.java:34)

    ... 1 more

### ****Q15. Is there any way of throwing a checked exception from a method that does not have a****throws****clause?****

Yes. We can take advantage of the type erasure performed by the compiler and make it think we are throwing an unchecked exception, when, in fact; we’re throwing a checked exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public <T extends Throwable> T sneakyThrow(Throwable ex) throws T {      throw (T) ex;  }    public void methodWithoutThrows() {      this.<RuntimeException>sneakyThrow(new Exception("Checked Exception"));  } |

# Q=[**How do you assert that a certain exception is thrown in JUnit 4 tests?**](https://stackoverflow.com/questions/156503/how-do-you-assert-that-a-certain-exception-is-thrown-in-junit-4-tests)

Ans=JUnit 4 has support for this:

@Test(expected = IndexOutOfBoundsException.class)

public void testIndexOutOfBoundsException() {

ArrayList emptyList = new ArrayList();

Object o = emptyList.get(0);

}

<https://stackoverflow.com/questions/156503/how-do-you-assert-that-a-certain-exception-is-thrown-in-junit-4-tests?rq=1>