Java - Exception

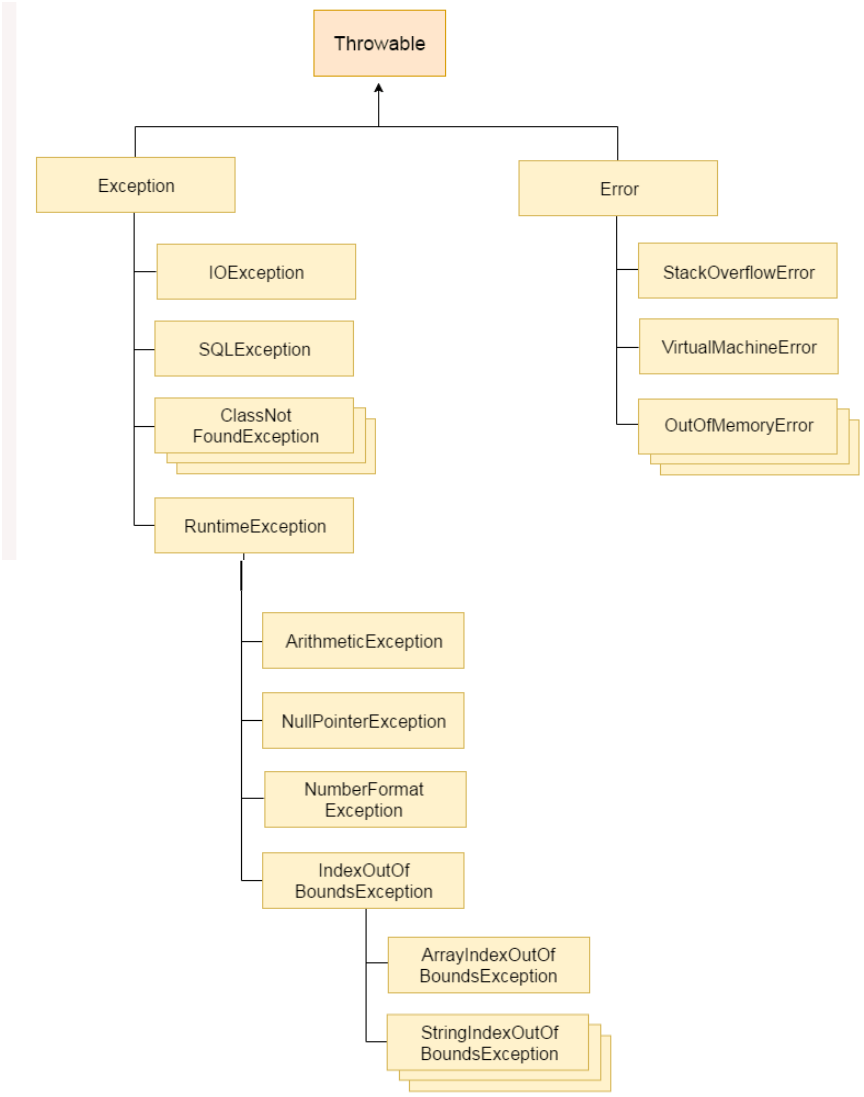
Why use multiple catch block?

Is there any possibility when finally block is not executed?

What is exception propagation?

What are the 4 rules for using exception handling with method overriding?

The java.lang.**Throwable** class is the ***root class*** of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

If we divide any number by zero, there occurs an **ArithmeticException**

```
int a=50/0;//ArithmeticException
```

If we have a null value in any variable, performing any operation on the variable throws a **NullPointerException**

```
String s=null;

System.out.println(s.length());//NullPointerException
```

The wrong formatting of any value may occur **NumberFormatException**. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";

int i=Integer.parseInt(s);//NumberFormatException
```

If you are inserting any value in the wrong index, it would result in **ArrayIndexOutOfBoundsException** as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

If we use different type of exception handling in catch block which is irrelevant to the exception occurring in try block then the exception is not handled properly and will result in interruption of the process, eg:

```
try        {
int data=50/0; // Will throw Arithmetic Exception
}
catch(ArrayIndexOutOfBoundsException e)        {
   System.out.println(e);
}
```

Exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation

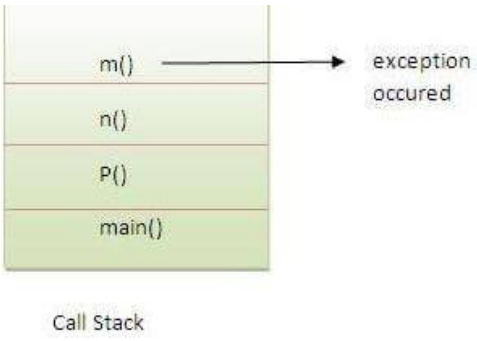
Only **Runtime exceptions** are propagated

Checked exceptions are not propagated, they will give a ***compile time error***

```
class TestExceptionPropagation1{
 void m() {     int data=50/0;    }   // This is a Runtime exception
 void n(){     m();    }
 void p(){
                try{      n();   }
                 catch(Exception e){System.out.println("exception handled");}
 }
```

```
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

```
Output:exception handled
       normal flow...
```



In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to **p() method where exception is handled**.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

CHECKED EXCEPTION SCENARIO

```
class TestExceptionPropagation1{
  void m() {    throw new java.io.IOException("device error");    //checked exception    }
  void n(){     m();    }
  void p(){
                try{     n();   }
                 catch(Exception e){System.out.println("exception handled");}
  }
    public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  } }
```

```
Output:  Compile Time Error
```

“throws” keyword

Used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

```
import java.io.IOException;

class Testthrows1{

  void m()    throws IOException{

                 throw new IOException("device error");          //checked exception

    }

  void n()     throws IOException    {

              m();

     }

  void p(){

              try{

                    n();

              }catch(Exception e)   {System.out.println("exception handled");}

       }

  public static void main(String args[]){

   Testthrows1 obj=new Testthrows1();

   obj.p();

   System.out.println("normal flow...");

  }

}
```

```
exception handled
normal flow...
```

If you are calling a method that declares an exception, you must either caught or declare the exception.

What the above line says is, m() and n() declares the exception but p() catches the exception, that is why this code is working... You should either declare(throws) the exception or catch() the exception.

There are two cases:

1. **Case1:** You caught the exception i.e. handle the exception using try/catch.
2. **Case2:** You declare the exception i.e. specifying throws with the method.

If you declare the exception("throws") then if it is not handled("try and catch") and so when an exception occurs it will throw a Runtime exception.

Can we rethrow an exception?

Yes, by throwing same exception in **catch block**

Difference between throw and throws in Java

throw	throws
Java throw keyword is used to **explicitly throw an exception**.	Java throws keyword is used to **declare an exception**.
Checked exception **cannot be propagated** using throw only.	Checked exception **can be propagated** with throws.
Throw is followed by an **instance**.	Throws is followed by **class.**
Throw is used **within** the **method**.	Throws is used **with** the **method** signature.
You cannot **throw multiple exceptions.**	You can **declare multiple exceptions** e.g. public void method()throws IOException**,** SQLException.

Difference between final, finally and finalize

final	finally	finalize
Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
Final is a **keyword**.	Finally is a **block**.	Finalize is a **method**.

ExceptionHandling with MethodOverriding

*Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but **can declare unchecked exception***

```
class Parent{
        void msg(){System.out.println("parent");}
}
class TestExceptionChild  extends  Parent  {
        void msg()  throws IOException {          //this is a checked exception
                System.out.println("TestExceptionChild");
  }
        public static void main(String args[]){
          Parent p=new TestExceptionChild();
          p.msg();
        }
}
```

```
Output:Compile Time Error
```

*Rule: If the superclass method declares an exception, subclass overridden method **can declare same, subclass exception** or no exception but **cannot declare parent exception***

```
class Parent{
        void msg()     throws ArithmeticException
                       {System.out.println("parent");}
}
class TestExceptionChild2 extends Parent{
        void msg()     throws Exception    // this is parent exception
                    {System.out.println("child");}
        public static void main(String args[]){
                    Parent p=new TestExceptionChild2();
                       try{ p.msg(); }
                       catch(Exception e){}
  }
}
```

```
Output:Compile Time Error
```

```
class Parent{
        void msg()     throws Exception
                       {System.out.println("parent");}
}
class TestExceptionChild2 extends Parent{
        void msg()     throws ArithmeticException    // this is child/sub exception
                    {System.out.println("child");}
        /* Same code as above for “main” method /*
```

```
Output: child
```

Custom Exception

```
class InvalidAgeException extends Exception  {
 InvalidAgeException  (String s)  {    // this is a parameterized constructor
  super(s);  }
}
```

```
class TestCustomException1{
        static void validate (int age)      throws    InvalidAgeException{
                if(age<18)
                        throw   new InvalidAgeException("not valid");
                else
                        System.out.println("welcome to vote");
        }

        public static void main(String args[]){
                try{
                        validate(13);
                }
                catch  (Exception m){
                System.out.println  ("Exception occured: "+m);
                }
                System.out.println("rest of the code...");
       }
}
```

```
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```

What Is the Difference Between an Exception and Error?

An exception is an event that represents a condition from which is possible to recover, whereas error represents an external situation usually impossible to recover from.

All errors thrown by the JVM are instances of *Error* or one of its subclasses, the more common ones include but are not limited to:

- *OutOfMemoryError* – thrown when the JVM cannot allocate more objects because it is out memory, and the garbage collector was unable to make more available
- *StackOverflowError* – occurs when the stack space for a thread has run out, typically because an application recurses too deeply
- *ExceptionInInitializerError* – signals that an unexpected exception occurred during the evaluation of a static initializer
- *NoClassDefFoundError* – is thrown when the classloader tries to load the definition of a class and couldn't find it, usually because the required *class* files were not found in the classpath
- *UnsupportedClassVersionError* – occurs when the JVM attempts to read a *class* file and determines that the version in the file is not supported, normally because the file was generated with a newer version of Java

Although an error can be handled with a *try* statement, this is not a recommended practice since there is no guarantee that the program will be able to do anything reliably after the error was thrown.

What Is Exception Chaining?

Occurs when an exception is thrown in response to another exception. This allows us to discover the complete history of our raised problem:

```
try {
    task.readConfigFile();
} catch (FileNotFoundException ex) {
    throw new TaskException("Could not perform task", ex);
}
```

What Is a Stacktrace and How Does It Relate to an Exception?

A stack trace provides the names of the classes and methods that were called, from the start of the application to the point an exception occurred.

It's a very useful debugging tool since it enables us to determine exactly where the exception was thrown in the application and the original causes that led to it

Can You Throw Any Exception Inside a Lambda Expression's Body?

When using a standard functional interface already provided by Java, you can only throw unchecked exceptions because standard functional interfaces do not have a “throws” clause in method signatures:

```
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
integers.forEach(i -> {
    if (i == 0) {
        throw new IllegalArgumentException("Zero not allowed");
    }
    System.out.println(Math.PI / i);
});
```

However, if you are using a custom functional interface, throwing checked exceptions is possible:

```
@FunctionalInterface
public static interface CheckedFunction<T> {
    void apply(T t) throws Exception;
}

public void processTasks(
  List<Task> taks, CheckedFunction<Task> checkedFunction) {
    for (Task task : taks) {
        try {
            checkedFunction.apply(task);
        } catch (Exception e) {
            // ...
        }
    }
}

processTasks(taskList, t -> {
    // ...
    throw new Exception("Something happened");
});
```

Will the Following Code Compile?

```
void doSomething() {
    // ...
    throw new RuntimeException(new Exception("Chained Exception"));
}
```

Yes. When chaining exceptions, the compiler only cares about the first one in the chain and, because it detects an unchecked exception, we don't need to add a throws clause.

- What is an exception in Java?
- How does exception handling work in Java?
- What are exception handling keywords in Java?
- What is the purpose of the throw and throws keywords?
- How can you handle an exception?
- Explain the Java exception hierarchy.
- How can you catch multiple exceptions?
- What is the difference between checked and unchecked exceptions in Java?
- What is the difference between throw and throws keyword in Java?
- What is the difference between an exception and error?
- What is the `OutOfMemoryError` in Java?
- What are chained exceptions in Java?
- How do you write a custom exception in Java?
- What is the difference between final, finally, and finalize in Java?
- What happens when an exception is thrown by the main method?
- What is a try-with-resources statement?
- What is a stacktrace and how does it relate to an exception?
- What are the advantages of Java exceptions?
- Can you throw any exception inside a lambda expression's body?
- What are the rules that we need to follow when overriding a method that throws an exception?
- What are some of the exception handling best practices?

https://dzone.com/articles/java-exception-handling-interview-questions-and-an-1