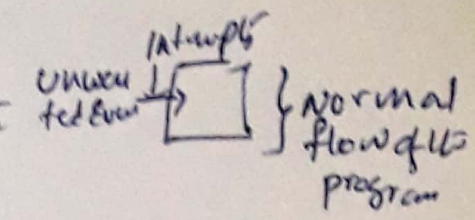
Exception Handling In Java

An Exception is an unwanted event that interrupts the normal flow of the program.

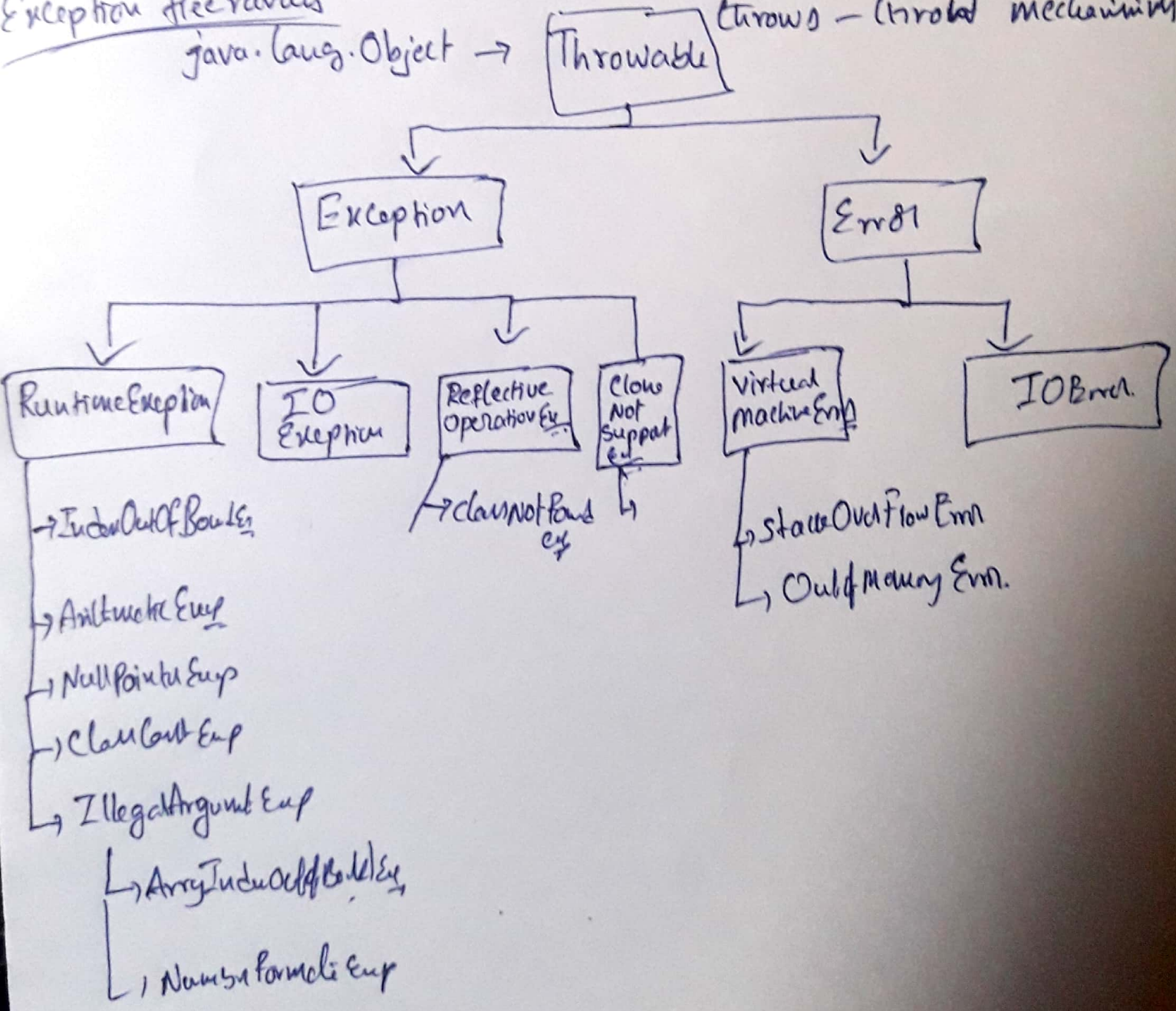


Note: When an exception occur program execution gets terminated. In such cases we get a System generated error message.

Exception Causes :- eg. opening a non-existing file in your program
Network Connection problem, bad input data provided by user.

How exceptions are Handled in java :- By using try-catch-finally. throws - throw keywords mechanism.

Exception Hierarchy



1) Scenario where ArithmeticException occurs

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

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

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPoint
```

3) Scenario where NumberFormatException occurs

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

```
String s="abc";

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

4) Scenario where ArrayIndexOutOfBoundsException occurs

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

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

Types of Exceptions :- ① Checked exception
② Unchecked exception.

→ ① Checked exception :- Checked exceptions as the Compiler checks them during Compilation to see whether the programmer has handled them or not.
→ If these ex. are not handled / declared in the program, you will get Compilation error. Eg IOException, ClassNotFoundException, SQLException.

→ ② Unchecked Exception :-

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at Compile-time so Compiler does not check whether the programmer has handled them or not but it is the responsibility of the programmer to handle these exceptions and provide a safe exit. Eg ArithmeticException, NullPointerException, ArrayIndexOutOfBoundEx.

Note:- Compiler will never force you to catch such exceptions & force you to declare it in the method using Throws keyword.

Try :- Try block contains set of statements where an exception can occur.
→ Try block is always followed by a catch block, which handles the exception that occurs in associated try block
→ A try block must be followed by catch blocks & finally block & / or both.

Catch => A catch block is where you handle the exceptions, this block must follow the try block.
→ A single try block can have several catch blocks associated with it.
→ You can catch different exceptions in different catch blocks, when an exception occurs corresponding catch block handles that particular exception.

Finally → A finally block contains all the crucial statements that must be executed whether exceptions occurs & not. Eg closing a Connection, Stream, File.

try-catch :- Java try block is used to enclose the code that might throw an exception. Note: It must be used within the method.

Rule ② Java try block must be followed by either catch or finally block

Syntax

try-catch

```
try {
  // code that may throw ex
} catch(Exception_class_name ref) {
  } // catch code.
```

try-finally

Syntax

```
try {
  // code may throw ex
} finally {
  // finally block code
}
```

Nested try block

Syntax

```
try {
  // statement 1
  // statement 2
  nested try
  try {
    // statement 3
    // statement 4
  } catch(Exception_class_name ref) { }
} catch(Exception_class_name ref) { }
```

Multiple catch block

```
try {
  ----
  ----
} catch(IOException e) { }
} catch(ArithmeticException e) { }
} catch(Exception e);
```

Note: catch block should follow the catch hierarchy by child-to-parent. because if you declare parent exception before the child, child exception block will not be handle.

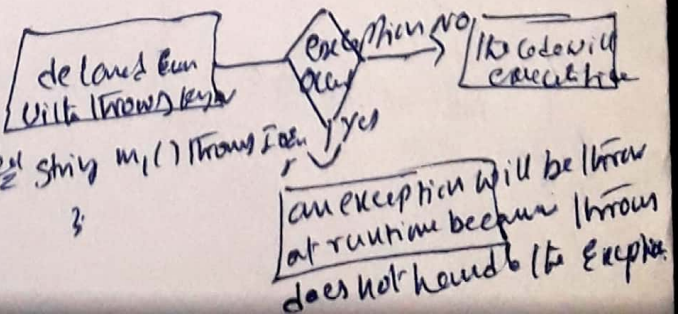
throws :- Syntax

```
return_type method_name( ) throws exception_name {
  ==
}
```

Note: at the time of compilation we do not declare their exceptions then the program will throw an compilation error.

Propagating an exception:-

→ if you declare the exception,



Throws Clause :- throws keyword is used for handling checked Exceptions.

→ By using throws we can declare multiple exceptions in one go.

Syn
```
return_ty method() throw excpt_ngt {
}
```

advantage :- by using throws checked Exception can be propagated (forwarded in call stack).

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

Throw Keyword :- we can define our own set of conditions @ rules and throw an exception explicitly using throw keyword.

Syn `throw new exception_class("custom + error message");`

→ Create our own exceptions class and throw that exception using throw keyword is called user-defined / custom exceptions

java.lang.Exception

→ public class Exception extends Throwable

java.lang.Throwable

public class Throwable extends Object implements Serializable

→ void addSuppressed(Throwable exception)

→ public String getMessage() → returns a detailed message about exp, this is initialized in the Throwable Constructor

→ public Throwable getCause() → cause of the exception as represented by the Throwable object.

→ public String toString() → the name of the class concatinated with the result of getMessage(),

→ public void printStackTrace() → prints the result of toString() along with the stack trace to System.err, the error output stream.

→ public StackTraceElement[] getStackTrace() → Returns an array containing each element on the stack trace. the element at index '0' represents the top of the call stack, last represents the bottom of the call stack.

→ public Throwable fillInStackTrace() → fills current stack trace, adding to any previous information in stack trace.

throws keyword or throws clause:

- ***throws keyword*** is used to declare the exception that might raise during program execution
- whenever exception might thrown from program, then programmer doesn't necessarily need to handle that exception using ***try-catch block*** instead simply ***declare that exception*** using throws clause next to method signature
- But this ***forces or tells*** the caller method to handle that exception, but again caller can handle that exception using ***try-catch block*** or ***re-declare those exception*** with throws clause
- In other words it can also be stated that, it provides information to the caller method that possible exception might raise during program execution and it need to be handled
- **Note:** use of throws clause doesn't necessarily mean that program will terminate normally rather it is the information to the caller to handle for normal termination
- Any ***number of exceptions*** can be specified using throws clause, but they are all need to be separated by commas(,)
- ***throws clause*** is applicable to ***methods*** & ***constructor*** but strictly not applicable to classes
- Compiler forces programmer to handle only ***checked exception;*** so it can be handled either way as discussed in the above points
- For ***unchecked exception***, it is programmer choice, as unchecked exception by default propagated back to the caller method when it isn't handled
- In addition to checked & unchecked exception, ***user-defined exception*** or ***custom exception*** can also be specified using ***throws clause***
- But user-defined exception or custom exception ***must be of type Throwable class*** or ***one of its sub-class*** (i.e.: extend either Throwable class or any one of its sub-class like Exception, IOException, RuntimeException, Error, etc)
- Otherwise, compile-time error will be thrown stating "***incompatible type***"

User-defined / Custom Exception class :-

In Java 2-types of exception ① Checked – java.lang.Exception
② Un-checked – java.lang.RuntimeExcep

① Checked :- Extends java.lang.Exception, for recoverable Condition try-catch like exception explicitly, Compile error.

Note: IOException, FileNotFoundException.

② Un-checked :- Extends java.lang.RuntimeException. for unrecoverable Condition, like programming error, no need try-catch, runtime error

Note: NullPointerException, IndexOutOfBoundException, IllegalArgumentExcep

① Custom Checked Exception

```
public class NameNotFoundException
        extends Exception {
    public NameNotFoundException(String message) {
        super(message);
    }
}
```

```
p s v m (String args) throws
        NameNotFoundExcep
    if (" ".equals(new)) {
        throw new NameNotFoundExcep
            ("name not");
    }
```

② Unchecked Exception

```
public class ListTooLargeException extends RuntimeException {
    public ListTooLargeException(String message) {
        super(message);
    }
}
```

Checked Exception:

- Exception which are checked at compile-time during compilation is known as *Checked Exception*
- **Alternate definition:** any line of code that could possibly throw exception, and if it is raised to handle during compilation is said to be checked exception
- **For example,** accessing a file from remote location could possibly throw file not found exception
- It is the programmer's responsibility to handle the checked exception for **successful compilation**
- This way, if any exception is raised during execution then respective handling code will be executed
- **Note:** if it isn't handled then program will throw compile-time error
- **Example:** IOException, FileNotFoundException, InterruptedException, SQLException, etc
- **Except** Runtime exception & its child classes and error & its child classes, all other exception fall under the category of Checked Exception

Unchecked Exception:

- Exception which are **NOT** checked at compile-time is known as ***Unchecked Exception***
- **Alternate definition:** any line of code that could possibly throw exception at runtime is said to be unchecked exception
- Unchecked exception are because of programming error
- **For example,** accessing out of index position to assign some value during execution could possibly throw exception at runtime
- So, it is again programmer's responsibility to handle the unchecked exception by providing alternate solution in the exception handling code
- **Note:** if it isn't handled properly then program will **terminate** abnormally at runtime
- **Example:** Runtime exception & its child classes and error & its child classes are examples of Unchecked Exception
- Like ArithmeticException, NullPointerException, NumberFormatException, ArrayIndexOutOfBoundsException, StatckOverflowError, etc

UncheckedException.java

UncheckedException.java

```
1  package in.bench.resources.exception.handling;
2
3  public class UncheckedException {
4
5      public static void main(String[] args) {
6
7          char[] ch = new char[4];
8          ch[7] = 'B';
9
10         System.out.println(ch);
11     }
12 }
```

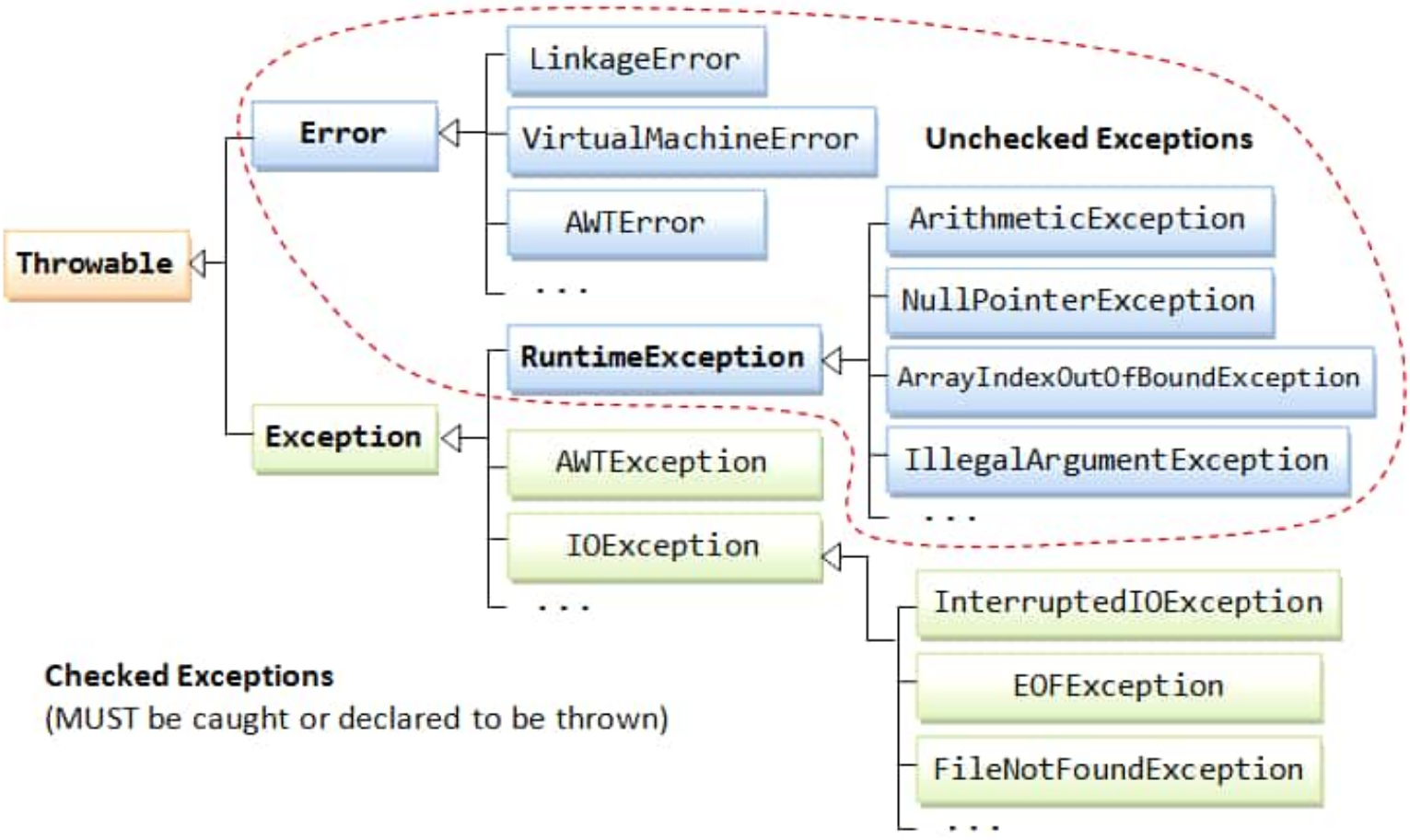
Problems | Javadoc | Declaration | Console

<terminated> UncheckedException [Java Application] **C:\java\javaw.exe** (Feb 1, 2017 7:47:22 PM)

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
        at in.bench.resources.exception.handling.UncheckedException.main(UncheckedException.java:8)
```

Misconception about checked and unchecked exception:

- Sometimes, checked exception are also referred as compile-time exception and unchecked exception are referred as runtime exception
- But this is mis-leading because every exception (whether it is checked or unchecked) occurs/raised only at the runtime i.e.; during program execution only
- **Reason:** during compilation; checked exception are caught and raises compile-time error, due to which programmer has to handle the exception by providing either try-catch blocks or using throws keyword
- Whereas unchecked exception aren't caught during compilation, rather it raises exception during execution because of programming error



Checked Exceptions
(MUST be caught or declared to be thrown)

Rule for User-defined exception or Custom exception:

- *throw keyword* is used to throw User-defined exception ***explicitly***
- Whenever User-defined exception is created, then it is must to ***extend*** one of the exception type from the **Exception hierarchy**
- User-defined exception ***extending checked exception forces the programmer*** to handle either by **try-catch block** or declaring using **throws clause**
- User-defined exception ***extending unchecked exception allows the choice to programmer*** to handle the exception either by **try-catch block** or declaring using **throws clause**
- Always ***provide*** public 1-arguemnt *Constructor* of String type
- Always, ***provide*** one statement inside constructor i.e.; ***call to*** *super constructor* by passing string argument
- This helps to supply error/exception description available to *printStackTrace()* **method of Throwable class** through **Constructor chaining process**

Runtime mechanism in Java – what happens when exception is thrown?

February 9, 2017 SJ Exception Handling 0

In this article, we will discuss runtime mechanism i.e.: what happens internally when any Java program executes

- Normally (graceful termination)
- Abnormally (abnormal termination)

TestRuntimeMechanismForNormalCondition.java

```java
package in.bench.resources.exception.handling;

public class TestRuntimeMechanismForNormalCondition {

    // main() method - start of JVM execution
    public static void main(String[] args) {
        callMethodOne();
    }

    // callMethodOne() method
    public static void callMethodOne() {
        callMethodTwo();
    }

    // callMethodTwo() method
    public static void callMethodTwo() {
        callMethodThree();
    }

    // callMethodThree() method
    public static void callMethodThree() {
        System.out.println("Invoked method Three SUCCESSFULLY from method Two");
    }
}
```

Output:

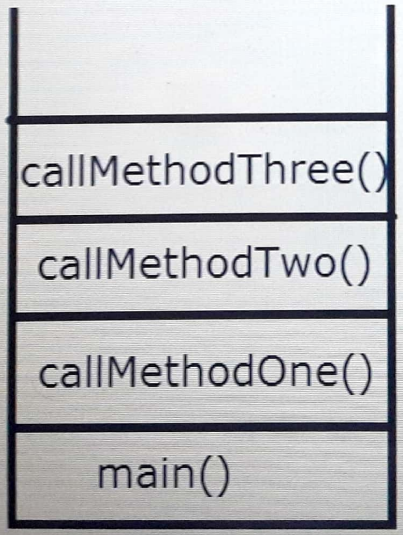
```
Invoked method Three SUCCESSFULLY from method Two
```

Explanation:

- In the above example, there is no system or programmatic exception/error
- Therefore, program executed successfully without any exception and prints message in the console
- Finally program terminates i.e. graceful termination

Runtime Stack:

- When program executes (normally or abnormally), JVM creates runtime stack for every thread spawned from that particular program (to store all its information like method call, etc)
- Since, in the above example there is only one thread is spawned i.e.: main thread
- Therefore, JVM creates one runtime stack for main thread and stores every method in the order (invoking order)
- Each entry in the runtime stack is known as ***Activation Record***. Alternatively, it is also called as ***Stack frame***
- For the above example, which executes normally JVM invokes first method call i.e.: ***main()***
- So, an entry in the runtime stack is stored



Runtime Stack in Java

- **Note:** generally, this is 1st method call from JVM for standalone program
- ***main()*** method executes which has call to another static method called ***callMethodOne()***
- ***callMethodOne()*** method has a call to another method called ***callMethodTwo()***
- Similarly, ***callMethodTwo()*** method has a call to another method called ***callMethodThree()***
- As there is no further invocation from method three, this is last entry into the runtime stack and thus there are 4 entries in the runtime stack as shown in the above figure
- In the reverse order, all entries into runtime stack will be exited one-by-one after corresponding method execution
- So, ***callMethodThree()*** will be executed and its entry from runtime stack exited
- In this way, next in line will be ***callMethodTwo()***, ***callMethodOne()*** and finally ***main()*** method
- Now, runtime stack won't contain any entries i.e.; it will be empty
- Finally, JVM destroys empty runtime stack

Exception scenario:

This sample code throws exception during execution

TestRuntimeMechanismForAbnormalCondition.java

```
package in.bench.resources

public class TestRuntimeMec

    // main() method - star
    public static void main
        callMethodOne();
    }

    // callMethodOne() meth
    public static void call
        callMethodTwo();
    }

    // callMethodTwo() meth
    public static void call
        callMethodThree();
    }

    // callMethodThree() me
    public static void call
        int result = 19/0;
    }
}
```

Output:

```
Exception in thread "main"
    at in.bench.resources.ex

    at in.bench.resources.ex

    at in.bench.resources.ex

    at in.bench.resources.ex

```

Explanation:

- This program is similar to the one, we have discussed in the earlier scenario i.e.: graceful termination
- But in this exception scenario example, last method is tweaked to throw exception
- Let us understand what happens in the runtime stack

Runtime Stack for Exception:

- When this program is executed, similar to last program there will be 4 entries into runtime stack for each method call as there is no exception raised until 4^{th} method
- Now when 4^{th} entry (i.e.; 4^{th} method) executes, it encounters an exception and it looks for exception handling code in the same method
- But there is no handling code in the method therefore it create exception object and passes control over to JVM along with exception object
- JVM exits currently executing method (i.e.; 4^{th} entry removed from stack) abnormally and looks for handler code in the caller method (i.e.; 3^{rd} entry in the stack)
- Since JVM doesn't find any handling code here, so similar to last step currently executing method exits (i.e.; 3^{rd} entry removed from stack) and method terminates abnormally
- This continues until it reaches main() method and every entry in the runtime stack will be exited
- Even in main method, there is no handler code therefore corresponding entry will be removed from runtime stack and main method terminates abnormally
- Finally JVM destroys runtime stack after it becomes empty and passé the control to ***Default exception handler*** along with the exception object
- Finally, overall program terminates abnormally and default exception handler prints exception information in the console

Questions:

1. Whether it is must to ***extend*** Exception class for User-defined Exception ?
2. If extending Exception is must, then what ***type of Exception*** need to be extended (i.e.; whether to extend checked exception or unchecked exception ?)
3. What is the ***significance*** of ***extending checked exception*** ?
4. What is the ***significance*** of extending ***unchecked exception*** ?
5. Whether ***public 1-argument constructor*** of ***String type*** must always be provided ?
6. ***What happens***, if we don't provide public 1-argument constructor of String type ?
7. Explain how user-defined exception's description is made available to ***printStackTrace()*** method (as this method is defined inside Throwable class –> top of Exception hierarchy)

User-defined Exception or Custom Exception in Java

Answers:

1. It is ***very must to extend*** one of the exception type in the **Exception hierarchy**, otherwise it won't act as User-defined exception rather plain-old-java-object
2. To define User-defined exception, any one of the ***Exception type from Exception hierarchy*** needs to be ***extended***. Well it can be checked exception or unchecked exception
3. Whenever user-defined exception ***extends checked exception***, then compiler ***forces/tells*** the programmer to ***handle*** this exception either by **try-catch blocks** or declaring this exception using **throws clause**
4. Whenever user-defined exception ***extends unchecked exception***, then compiler ***never forces*** the programmer to ***handle*** the exception. Rather, it is ***the choice to programmer*** to handle either by **try-catch block** or declaring this exception using **throws clause**
5. Though, it isn't strict to provide public 1-argument Constructor of String type but providing ***will help the programmer to pass the user-defined error/exception description*** along with the User-defined exception type (read above example to understand this statement)
6. By not providing public 1-argument Constructor of String type, ***stops*** the programmer to ***define user-defined error/exception description*** whenever explicit User-defined exception is thrown using **throw keyword**
7. To make user-defined error/exception description available to ***printStackTrace()*** **method of Throwable class**, just provide ***call to super() constructor using super keyword*** along with user-defined error/exception description of String type as its argument

Exception propagation in Java

February 24, 2017 SJ

In this article, we will discuss *exception propagation* in detail with example

Exception propagation:

- Whenever exception is raised from method and if it isn't handled in the same method, then it is propagated back to the caller method
- This step is repeated until handler code is found in one of the caller method in the runtime stack or else it reaches the bottom of the runtime stack
- This is known as Exception propagation

Rules for Exception propagation:

- By default, unchecked exception is propagated back to the runtime stack one-by-one until it finds handler code or it reached the bottom of the stack
- Checked exception isn't propagated, rather compiler forces the programmer to handle checked exception in the same method by surrounding with try-catch block or declaring with throws keyword

1. Example on Unchecked exception:

In this example,

- When arithmetic exception is raised in methodThree(), then it isn't handled. Therefore, it is propagated back to the caller method i.e.; methodTwo()
- Similarly, there is no handler code available in methodTwo() also
- Therefore, again it is propagated back to the caller method i.e.; methodOne()
- Like this, it will be repeated until if finds a suitable handler code or till bottom of the stack is reached
- In this case, there is no handler code till it reaches the bottom of the stack
- Finally JVM passes the control to Default exception handler along with exception object when no handler code is found (i.e.; propagation reaches the main() method i.e.; last entry in the runtime stack)
- Default exception handler prints the exception information it has got from exception object and terminates the methods abnormally

DefaultPropagationForUncheckedExcepti on.java

```
package in.bench.resources

public class DefaultPropaga

    // main() method - star
    public static void main
        callMethodOne();
    }

    // callMethodOne() meth
    public static void call
        callMethodTwo();
    }

    // callMethodTwo() meth
    public static void call
        callMethodThree();
    }

    // callMethodThree() me
    public static void call

        // performing arith
        int result = 19/0;
        System.out.println(
    }
}
```

Output:

```
Exception in thread "main"
    at in.bench.resources.ex
.callMethodThree(DefaultProp
    at in.bench.resources.ex
.callMethodTwo(DefaultPropag
    at in.bench.resources.ex
.callMethodOne(DefaultPropag
    at in.bench.resources.ex
.main(DefaultPropagationForU
```

2. Example on Checked exception:

Whenever checked exception is thrown, then compiler throws compile-time error stating "*Unhandled exception type exception-class-name*"

- So default propagation like in earlier case for unchecked exception isn't possible for this case (with checked exception)
- Because, compiler forces/tells with an compile-time error to handle the checked exception either with a **try-catch block** combination or declaring **throws clause**
- Therefore, it is must for checked exception to handle
- Not providing handler code leads to compile-time error
- **Note:** default propagation isn't possible for checked exception but programmer can manually propagate using **throw keyword**
- Move over to next example for explanation

NoPropagationForCheckedException.java

DefaultPropagationForUncheckedException.java | NoPropagationForCheckedException.java

```
1  package in.bench.resources.exception.handling;
2
3  import java.io.FileReader;
4
5  public class NoPropagationForCheckedException {
6
7      // main() method - start of JVM execution
8      public static void main(String[] args) {
9          callMethodOne();
10     }
11
12     // callMethodOne() method
13     public static void callMethodOne() {
14         callMethodTwo();
15     }
16
17     // callMethodTwo() method
18     public static void callMethodTwo() {
19         callMethodThree();
20     }
21
22     // callMethodThree() method
23     public static void callMethodThree() {
24
25         // performing IO operation
26         // assumed that, we are trying to access file from remote location
27         FileReader fileReader = new FileReader("D:/Folder/test.txt");
28     }
29 }
```

Unhandled exception type FileNotFoundException

2 quick fixes available:

- Add throws declaration
- Surround with try/catch

3. Explicit propagation for checked exception:

- Explicitly, we can propagate checked exception too by declaring with throws clause
- But it must be handled in one of the method in the runtime stack
- Otherwise compile-time error will be thrown stating "*Unhandled exception type exception-class-name*"
- In the below example for explicitly throwing checked exception, last entry in the runtime stack i.e., main() method handled exception by surrounding the call with try-catch block

ExplicitPropagationForCheckedException.java

```
package in.bench.resources

import java.io.FileNotFound
import java.io.FileReader;

public class ExplicitPropag

    // main() method - star
    public static void main

        try {
            callMethodOne()
        }
        catch (FileNotFound
            System.out.prin
        }

        System.out.println(
    }

    // callMethodOne() meth
    public static void call
        callMethodTwo();
    }

    // callMethodTwo() meth
    public static void call
        callMethodThree();
    }

    // callMethodThree() me
    public static void call

        // performing IO op
        // assumed that, we
        FileReader fileRead
    }
}
```

Output:

```
Exception handled successfu
By handling for checked exce
```

Conclusion:

- By default, unchecked exception are propagated back to the runtime stack until it reaches bottom of the stack or else it finds handler code
- By default, checked exception aren't propagated; because whenever there is possibility of raising checked exception then compiler forces/tells to handle it
- But explicit propagation for checked exception is possible by use of **throws keyword**

In this article, we will discuss the difference between *final, finally and finalize* in detail with example

This is one of the *top interview question* for Java fresher's and at junior level to check the knowledge of core Java concepts

First of all, final & finally & finalize is used in different scenarios and also they aren't similar in any context like

- ***final*** is a keyword used for ***restricting*** further alteration in inheritance
- ***finally*** is a block & it is associated with ***try-catch block*** in exception handling for ***clean-up*** activity
- ***finalize()*** is a method associated with garbage collector to ***de-allocate*** resources associated with the Object

Let us discuss each one in detail with example

1. final keyword:

- final is keyword or modifier
- applicable only for variable, method & classes
- variable declared as final can't be ***changed*** or ***re-assigned*** once it is initialized
- method declared as final can't be ***overridden*** in the sub class (inheriting class or extending class)
- class declared as final can't be ***extended*** i.e., inherited further (or sub-class'd)
- Read more about final keyword or modifier

1. final keyword:

- final is keyword or modifier
- applicable only for variable, method & classes
- variable declared as final can't be ***changed*** or ***re-assigned*** once it is initialized
- method declared as final can't be ***overridden*** in the sub class (inheriting class or extending class)
- class declared as final can't be ***extended*** i.e.; inherited further (or sub-class'd)
- Read more about final keyword or modifier

1.1 Variable declared as final can't be re-assigned

Compile-time error: The final field FinalKeyword.count cannot be assigned

1.2 Method declared as final can't be overridden

Compile-time error: Cannot override the final method from ParentClass

1.3 Class declared as final can't be extended or inherited or sub-class'd

Compile-time error: The type ChildClass cannot subclass the final class ParentClass

2. finally block:

- finally is a block associated with ***try-catch block*** in exception handling concept
- It is used to provide ***clean-up code*** for ***releasing resources*** used in try block like database connection or IO resources or network resources
- The beauty of finally block is that it is ***always gets executed***, irrespective of whether exception is thrown or NOT and whether exception is handled or NOT
- Read more about finally block

FinallyBlockExample.java

```
1  package in.bench.resource
2
3  public class FinallyBlockEx
4
5      public static void main
6
7          try {
8
9              // code which m
10
11             int result = 19
12             System.out.prin
13         }
14         catch(ArithmeticExc
15
16             // correspondin
17
18             System.out.prin
19         }
20         finally {
21
22             // finally bloc
23
24             System.out.prin
25             // rest of the
26         }
27     }
28 }
```

3. finalize() method:

- finalize() is a method is associated with garbage collector
- this method is ***invoked*** just before ***destroying an Object*** i.e., to provide clean-up activities
- After Garbage Collector invokes finalize() method, then ***immediately*** it ***destroys*** an Object
- Programmer doesn't have any control over the invocation of finalize() method because it is ***internally*** invoked by garbage collector for null objects (although, we can make Object as null by assigning to null reference)

Method signature:

```
protected void finalize()
```

FinalizeMethodExample.java

```
package in.bench.resource

public class FinalizeMethod

    protected void finalize

        System.out.println(
    }

    public static void main

        // create Object of
        FinalizeMethodExamp

        // explicitly makin
        fme = null;
    }
}
```

Explanation:

- When above program is executed, it doesn't print any sysout statement from finalize() method
- Because, it is the garbage collector's duty to invoke finalize() method just before destroying the Object
- And hence programmer can't make sure that it is compulsorily invoked although we can make any Object as null explicitly

Lets us move on and discuss key differences between these **ClassNotFoundException** & **NoClassDefFoundError**

ClassNotFoundException

This is generally occurs, when required .class is missing when program encounters class load statement such as,

- Class.forName("class.name");
- ClassLoader.loadClass("class.name");
- ClassLoader.findSystemClass("class.name");

Reason: required file ***missing*** in the class-path due to execution of program ***without updating JAR file*** at runtime

Fully qualified class name is *java.lang.ClassNotFoundException*

It falls under the category of Exception i.e.; direct sub-class of ***java.lang.Exception***

It is a ***checked exception***, therefore it needs to be ***handled***, whenever class loading statement is encountered as stated in point no.1

As it is ***checked exception***, programmer can provide handling code either using ***try-catch*** block or can declare ***throws clause***

Therefore, it is *recoverable*

Example 1

NoClassDefFoundError

This is very much similar but the difference is required .class file is available during compile-time & missing at runtime

Possible Reason:

- It is ***deleted*** after compilation or
- there could be ***version*** mismatch

Fully qualified class name is *java.lang.NoClassDefFoundError*

It falls under the category of Error i.e.; sub-class of ***java.lang.Error*** through ***java.lang.LinkageError***

All errors come under ***unchecked exception*** category, therefore ***NoClassDefFoundError*** is also unchecked exception

Errors are thrown by ***Java Runtime system*** during program execution

Therefore, it is *non-recoverable*

Example 2

Example 1: Demo example on ClassNotFoundException:

JdbcConnectionExample.java

```
 in.bench.resources.top.exception.

class JdbcConnectionExample {

lic static void main(String[] args

 // declare variables

 // Step 1: Loading or registering
 try {
     Class.forName("oracle.jdbc.dr
 }
 catch(ClassNotFoundException cnfe
     System.out.println("Problem :
     cnfex.printStackTrace();
 }

 // Step 2: Opening database conne
```

Output:

```
java.lang.ClassNotFoundExcept
Problem in loading Oracle JDB
    at java.net.URLClassLoade
    at java.security.AccessCo
    at java.net.URLClassLoade
    at java.lang.ClassLoader
    at sun.misc.Launcher$AppC
    at java.lang.ClassLoader
    at java.lang.Class.forNar
    at java.lang.Class.forNar
    at in.bench.resources.top
.main(JdbcConnectionExample.j
```

Explanation:

In the above example,

- we are trying to load ***driver file*** for Oracle databases using ***forName()*** method of Class class, but it isn't available at runtime
- ***Possible reason*** for this type of exception is, executing a JDBC program without updating class-path with required JAR files
- **Solution:** to rectify this exception, just include required ***ojdbc14.jar*** into the class-path and then again execute same program

Example 2: Demo example on NoClassDefFoundError:

SimilarException.java

```
package in.bench.resources

public class SimilarExcepti

    // using below declared
    static TestFile tf = ne

    public static void main

        // invoke method
        tf.display();
    }
}

class TestFile {

    public void display() {
        System.out.println(
    }
}
```

Output:

```
java.lang.NoClassDefFoundE
    at in.bench.resources.t
Caused by: java.lang.ClassN
    at java.net.URLClassLoa
    at java.security.Access
    at java.net.URLClassLoa
    at java.lang.ClassLoade
    at sun.misc.Launcher$Ap
    at java.lang.ClassLoade
    ... 1 more
Exception in thread "main"
```

Explanation:

In the above example,

- We are trying to ***execute*** a program and required ***.class*** files is ***missing*** from class-path
- ***Possible reason*** for this exception-type is. required file is present during compilation but missing while executing the same program
- Above program exhibits "***HAS-A***" relationship & compilation succeeds whereas during program execution JVM unable to find required .class file
- **Note:** deliberately deleted TestFile.class after compilation to showcase this exception-type