Chained Exceptions in Java

baeldung.com/java-chained-exceptions

1. Overview

In this article, we'll have a very brief look at what *Exception* is and go in depth about discussing the chained exceptions in Java.

Simply put, an *exception* is an event that disturbs the normal flow of the program's execution. Let's now see exactly how we can chain exceptions to get better semantics out of them.

2. Chained Exceptions

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.

This concept was introduced in JDK 1.4.

Let's see how chained exceptions are supported in Java.

3. *Throwable* Class

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*.

4. Example

Now, let's look at the example where we will set our own *Exception* description and throw a chained *Exception*:

```
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:

```
Caught: java.lang.ArithmeticException: Top Level Exception.
Actual cause: java.io.IOException: IO cause.
```

5. Why Chained Exceptions?

We need to chain the exceptions to make logs readable. Let's write two examples. First without chaining the exceptions and second, with chained exceptions. Later, we will compare how logs behave in both of the cases.

To start, we will create a series of Exceptions:

```
class NoLeaveGrantedException extends Exception {

    public NoLeaveGrantedException(String message, Throwable cause) {
        super(message, cause);
    }

    public NoLeaveGrantedException(String message) {
        super(message);
    }
}

class TeamLeadUpsetException extends Exception {
    // Both Constructors
}
```

Now, let's start using the above exceptions in code examples.

5.1. Without Chaining

Let's write an example program without chaining our custom exceptions.

```
public class MainClass {

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

    void getLeave() throws NoLeaveGrantedException {
        try {
            howIsTeamLead();
        } catch (TeamLeadUpsetException e) {
            e.printStackTrace();
            throw new NoLeaveGrantedException("Leave not sanctioned.");
        }
    }

    void howIsTeamLead() throws TeamLeadUpsetException {
        throw new TeamLeadUpsetException("Team Lead Upset");
    }
}
```

In the example above, logs will look like this:

```
com.baeldung.chainedexception.exceptions.TeamLeadUpsetException:
  Team lead Upset
    at com.baeldung.chainedexception.exceptions.MainClass
      .howIsTeamLead(MainClass.java:46)
    at com.baeldung.chainedexception.exceptions.MainClass
      .getLeave(MainClass.java:34)
    at com.baeldung.chainedexception.exceptions.MainClass
      .main(MainClass.java:29)
Exception in thread "main" com.baeldung.chainedexception.exceptions.
  NoLeaveGrantedException: Leave not sanctioned.
    at com.baeldung.chainedexception.exceptions.MainClass
      .getLeave(MainClass.java:37)
    at com.baeldung.chainedexception.exceptions.MainClass
      .main(MainClass.java:29)
```

5.2. With Chaining

Next, let's write an example with chaining our custom exceptions:

```
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.");
    }
}
```

Finally, let's look at the logs obtained with chained exceptions:

```
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
```

We can easily compare shown logs and conclude that the chained exceptions lead to cleaner logs.