L47

Friday, February 4, 2022 8:53 PM

Exception classes & their Hierarchy, Exception Types

Understanding Exception Handling

Q1. An exception by its very definition is something which is not normal.

In programming, if the program does not run as expected due to some abnormal event or situation, the error produced is called an exception.

Java provides easy to use constructs to handle such situations.

Even before we learn how to handle exceptions let us see an exception and try fixing it.

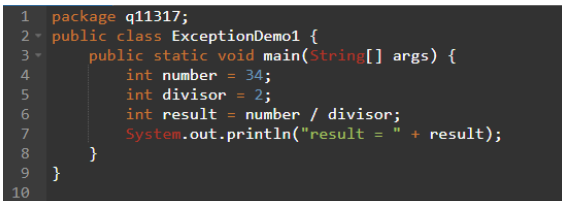
The below example code when run produces an exception called java.lang.ArithmeticException.

Click on **Submit** button to see the exception details as given below.

Caused by: java.lang.ArithmeticException: / by zero *// it has exception class namefollowed by error message* at ExceptionDemo1.main(ExceptionDemo1.java: 5) *// it has the class/method name and the line number which triggered the exception*... 6 more

After you click on **Submit** and see the exception message, replace the value 0 of the divisor variable with 2 to fix the problem.

Important: Please note that the blue animating arrow which is shown when you click **Submit**, is shown by our **intelligent error detection system**. When you start coding using a regular IDE during work, you will not be helped like this. You will only be provided the error information (stack trace) without this animating arrow. Hence, learning how to read and understand exception stack traces becomes an essential part of programming.



```
1  package q11317;
2  public class ExceptionDemo1 {
3      public static void main(String[] args) {
4          int number = 34;
5          int divisor = 2;
6          int result = number / divisor;
7          System.out.println("result = " + result);
8      }
9  }
10
```

Q2. When an exception occurs while the code is running, the JVM tries to provide as much information as possible regarding the li ne of code which triggered the exception.

This information usually spans multiple lines which is called the exception stack trace.

Click on **Submit** button to see the exception stack trace produced while executing the code.

It is very easy to fix the error in this scenario. However, we will first click on the Submit button and then learn to read the stack trace information provided.

Caused by: java.lang.ArithmeticException: / by zero *// it contains the exception class name followed by message*at ExceptionDemo2.divide(ExceptionDemo2.java: 9) *// **line - 9*** at ExceptionDemo2.main(ExceptionDemo2.java: 5) *// **line - 5*** ... 6 more

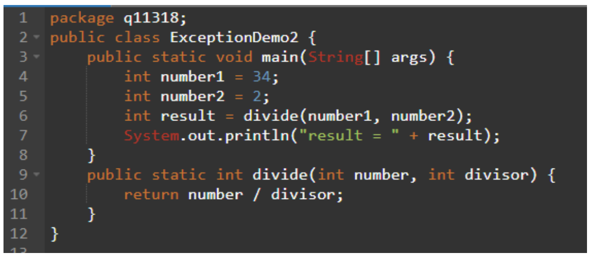
The exception stack trace contains two lines from the ExceptionDemo2 class.

The top most line in the **exception stack trace** which is from our code (meaning from a class written by us) is responsible for causing the exception.

When the statement in **line 9** in our code (ExceptionDemo2 class) is analyzed for the **ArithmeticException** with error message **/ by zero** , we can easily figure out that the value contained in the variable divisor is 0 (zero).

There can be some scenarios where the top most line is from a class available in Java standard classes or a class written by a third party library provider, in such cases we wil l have to ignore such lines till we find the lines which are from classes written by us.

After you have understood the above stated rules, correct the error in the code by replacing the value 0 of variable number2 with 2 and click on **Submit**.



```
1  package q11318;
2  public class ExceptionDemo2 {
3      public static void main(String[] args) {
4          int number1 = 34;
5          int number2 = 2;
6          int result = divide(number1, number2);
7          System.out.println("result = " + result);
8      }
9      public static int divide(int number, int divisor) {
10         return number / divisor;
11     }
12 }
```

Q3. The exception stack trace can include classes which are not written by us. The below code when submitted will terminate with an exception whose stack trace will contain method call information from Java classes in java.lang package.

As a learner it is easy to get lost when we see so many error lines.

However, it is extremely easy to pinpoint the line with error if we follow a simple thumb rule.

To start with let us click on **Submit** button to see the exception stack trace produced by the code.

Do not try to fix the code. First let us try to understand the exception stack trace we get when we click Submit.

```
Caused by: java.lang.NumberFormatException: For input string: "4a"// notice the exception class name and error messageat java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
	at java.lang.Integer.parseInt(Integer.java:580)
	at java.lang.Integer.parseInt(Integer.java:615)
	at ExceptionDemo3.convertAndAdd(ExceptionDemo3.java: 10 )
	at ExceptionDemo3.main(ExceptionDemo3.java:5)
	... 6 more
```

The first thing we should read in the exception stack trace is the line starting with **Caused by:** which contains the **name of the exception class** and the **error message**.

In our case the name of the exception class is NumberFormatException

And the error message is For input string: "4a"

It is always a good practice to read about the exception class. [Hint: Click on the name of the exception class to read about it.]

The exception stack trace in total contains **5 lines**. Out of which **2 lines** are from our class **ExceptionDemo3** and the remaining **3 lines** are from classes in Java's standard library.

Our goal is to find out which line in our code is responsible for triggering the exception. For all practical purposes we can safely assume that classes in Java code are not responsible for this exception.

Which means **we can safely ignore all the top 3 lines** in the stack trace which are from classes java.lang.NumberFormatException and java.lang.Integer.

As mentioned earlier we should scan from top to bottom to find the top most line in the **exception stack trace** which is from our code (meaning from a class written by us, which in our case is **ExceptionDemo3**).

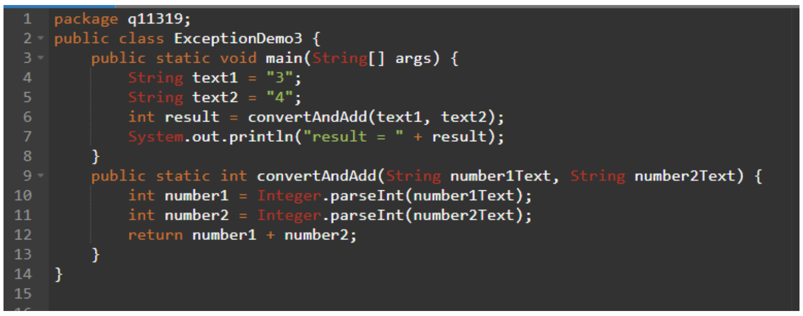
We will notice that **line 10** is what we are looking for.

When the statement in **line 10** in our code (in **ExceptionDemo3** class) is analyzed for the **NumberFormatException** with error message **For input string: "4a"**, we can easily figure out that the value contained in the reference number2Text is 4a which is not being properly parsed into an integer value.

We can see from the stack trace that the current method convertAndAdd is being called by code in line 5 which is inside the main method.

From that line 5 we can again figure out that text2 is being passed as number2Text. Which means that the value of text2 i.e 4a is responsible for the **NumberFormatException.**

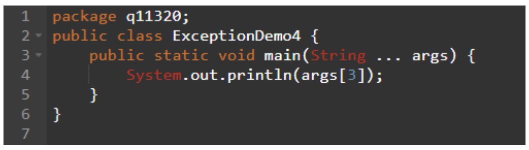
To fix the problem, change the string literal initialized to reference text2 from "4a" to "4" and click **Submit.**



```
1  package q11319;
2  public class ExceptionDemo3 {
3      public static void main(String[] args) {
4          String text1 = "3";
5          String text2 = "4";
6          int result = convertAndAdd(text1, text2);
7          System.out.println("result = " + result);
8      }
9      public static int convertAndAdd(String number1Text, String number2Text) {
10         int number1 = Integer.parseInt(number1Text);
11         int number2 = Integer.parseInt(number2Text);
12         return number1 + number2;
13     }
14 }
15
```

Q4. The code should actually print the **fourth** element passed in the arguments to the main method.

[**Hint:** Click on the **animating blue arrow** in the exception stack trace to understand the error.]



```
package q11320;
public class ExceptionDemo4 {
    public static void main(String ... args) {
        System.out.println(args[3]);
    }
}
```

Exception classes & their Hierarchy

Q1. In Java, Throwable is the super class for all Error classes and Exception classes.

Among hundreds of exception and error classes, below are few commonly used ones and their **exception hierarchy**:

```
              Throwable
                 |
  _______________________|___________________
  |                                                  |
 Error                    Exception
  |                   _________|
  |                                                      |   IOException
  |-- VirtualMachineError      |      |-- ParseException
  |-- OutOfMemoryError          |      |-- InterruptedException
  |-- StackOverflowError        |
                RuntimeException
                  |
                  |-- NullPointerException
                  |-- ArithmeticException
                  |-- IllegalArgumentException
                  |         |
                  |     NumberFormatException
                  |
                  |-- IndexOutOfBoundsException
                       |
                       |--ArrayIndexOutOfBoundsException
                       |--StringIndexOutOfBoundsException
```

In the above hierarchy, Throwable and all subclasses of Throwable that **are not** subclasses of either RuntimeException or Error are called checked exceptions.

All the other subclasses of Error and RuntimeException are called unchecked exceptions.

The Throwable class stores the method call stack trace and also the error message that is printed when an exception occurs.

Even though we have classes called Error and Exception as the main subclasses of Throwable, the whole concept of handling the exception scenarios is called exception handling. We do not say Throwable-handling, Error-handling and Exception-handling separately based on their class names. In general programming terminology, the concepts along with the constructs which are employed to handle erroneous or abnormal situations is called exception handling.

Select all the correct statements given below.

- [ ] ParseException is an unchecked exception.
- [ ] NumberFormatException is an checked exception.
- [✓] RuntimeException is an unchecked exception.
- [✓] NullPointerException is an unchecked exception.
- [✓] ArithmeticException is an unchecked exception.
- [ ] StackOverflowError is a checked exception.

Exception Types

Q1. An exception is thrown to signal an error condition.

Java provides a way to catch the exceptions and handle them by either taking corrective course of action or notifying the use r or doing what ever is needed.

This handling is done using a try-catch block or propagating the exception to the caller.

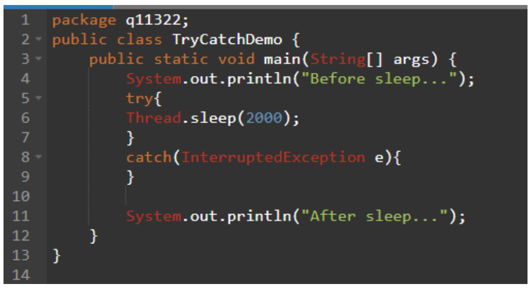
The compiler flags an error if we invoke methods which throw checked exceptions and do not handle them.

For example, the static method Thread.sleep(long milliSeconds) in Thread class throws an InterruptedException, which is of type checked exception. Below code demonstrates the usage of try-catch block. Notice how the statement containing the call to the sleep method is wrapped in the try-catch block.

```
try{// try-catch block startThread.sleep(2000);
}catch (InterruptedException e) {// catch clause e.printStackTrace();
}// try-catch block end
```

Click on Submit to see the error.

To fix the code, write the try-catch block appropriately.



```
package q11322;
public class TryCatchDemo {
    public static void main(String[] args) {
        System.out.println("Before sleep...");
        try{
        Thread.sleep(2000);
        }
        catch(InterruptedException e){
        }

        System.out.println("After sleep...");
    }
}

```

Q2. Unchecked exceptions are those which are not checked/verified during compile time for their exception handling code.

We have earlier used Integer.parseInt(String text) to parse some text and convert it into an integer value. Whenever this method is unable to parse the text, it throws an exception called NumberFormatException. However, the compiler does not enforce us to handle the **NumberFormatException** because **NumberFormatException** is an unchecked exception.

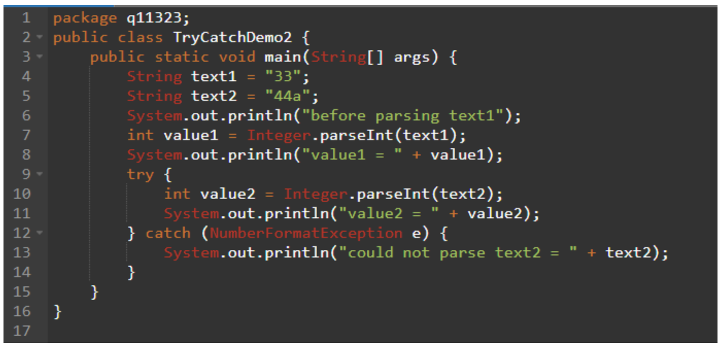
Among the unchecked exceptions, subclasses of RuntimeException can be handled using try-catch block, even though it is not mandated by the compiler. However, subclasses of Error which are also unchecked exceptions should not be handled using try-catch blocks.

Subclasses of Error usually indicate some serious irrecoverable error conditions which should not be caught using the the try-catch block.

See and retype the below code.

You will notice that one call to parseInt is not surrounded in a try-catch block, while the other is surrounded.

When you execute the code, you will also notice how the control transfers into the catch block when the exception occurs, ski pping the statement:
System.out.println("value1 = " + value1);



```java
package q11323;
public class TryCatchDemo2 {
    public static void main(String[] args) {
        String text1 = "33";
        String text2 = "44a";
        System.out.println("before parsing text1");
        int value1 = Integer.parseInt(text1);
        System.out.println("value1 = " + value1);
        try {
            int value2 = Integer.parseInt(text2);
            System.out.println("value2 = " + value2);
        } catch (NumberFormatException e) {
            System.out.println("could not parse text2 = " + text2);
        }
    }
}
```

Q3.

StackOverflowError is a subclass of Error and hence an unchecked exception. It is thrown by JVM.

While executing code when JVM exhausts the stack space available to store the method call information, it throws this error. **StackOverflowError** is not an instance (subclass) of an Exception class. It is a subclass of **Error**.

In the below code, you will notice that the main method calls the someMethod(). And the someMethod() internally calls itself.

The execution starts with the main. At this moment the JVM pushes the main method details in to a stack called **method call stack**.

Method Call Stack	
method-1	main(String[] args)

When the main for the first time calls someMethod(), JVM pushes the method information of someMethod() also into the **method call stack**. At this moment there are two entries in the **method call stack**.

Method Call Stack	
method-call-2	someMethod()
method-call-1	main(String[] args)

When the someMethod() internally call itself again, JVM pushes the method information of someMethod() once again into the **method call stack**. At this moment there are three entries in the **method call stack**.

Method Call Stack	
method-call-3	someMethod()
method-call-2	someMethod()
method-call-1	main(String[] args)

Since there is no condition in someMethod() which will stop itself from calling itself, it will infinitely keep calling itself. Eventually JVM will exhaust the stack spa ce available to store the method call information for each and every call of someMethod().

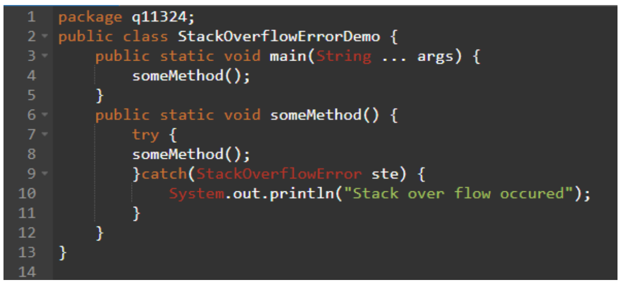
It is at this point when JVM gives up, it throws the StackOverflowError signally that it is out of stack space and just cannot do anything.

As a programmer whenever we see a StackOverflowError, we should analyze our code and remove the infinite recursion.

Subclasses of Error usually signal such fatal problems in JVM, which are not recoverable. Meaning we should not try to enclose such code in a try-catch. We should instead analyze the cause of the problem and try fixing it.

The below code demonstrates how a StackOverflowError occurs.

See and retype the below code. When you execute the code, you will notice how the control transfers into the catch block when the error occurs.



```java
package q11324;
public class StackOverflowErrorDemo {
    public static void main(String ... args) {
        someMethod();
    }
    public static void someMethod() {
        try {
        someMethod();
        }catch(StackOverflowError ste) {
            System.out.println("Stack over flow occured");
        }
    }
}
```