

Exception Handling in Java (Part 1)

Problems that arise in a Java program may generate exceptions or errors. An *exception* is an object that defines an unusual or erroneous situation. An exception is thrown by a program or the run-time environment and can be caught and handled appropriately if desired. An *error* is similar to an exception except that an error generally represents an unrecoverable situation and should not be caught (*don't worry about errors for now – we'll concentrate on exceptions and explain errors later.*) Java has a predefined set of exceptions and errors that may occur during the execution of a program.

Problem situations represented by exceptions and errors can have various kinds of root causes. Here are some situations that cause exceptions to be thrown:

- Attempting to divide by zero.
- An array index that is out of bounds.
- A specified file that could not be found.
- A requested I/O operation that could not be completed normally.
- An attempt was made to follow a null reference.
- An attempt was made to execute an operation that violates some kind of security measure.

These are just a few examples. There are dozens of others that address very specific situations.

As many of these examples show, an exception can represent a truly erroneous situation. But as the name implies, they may simply represent an exceptional situation. That is, an exception may represent a situation that won't occur under usual conditions. Exception handling is set up to be an efficient way to deal with such situations, especially given that they don't happen too often.

We have several options when it comes to dealing with exceptions. A program can be designed to process an exception in one of three ways. It can:

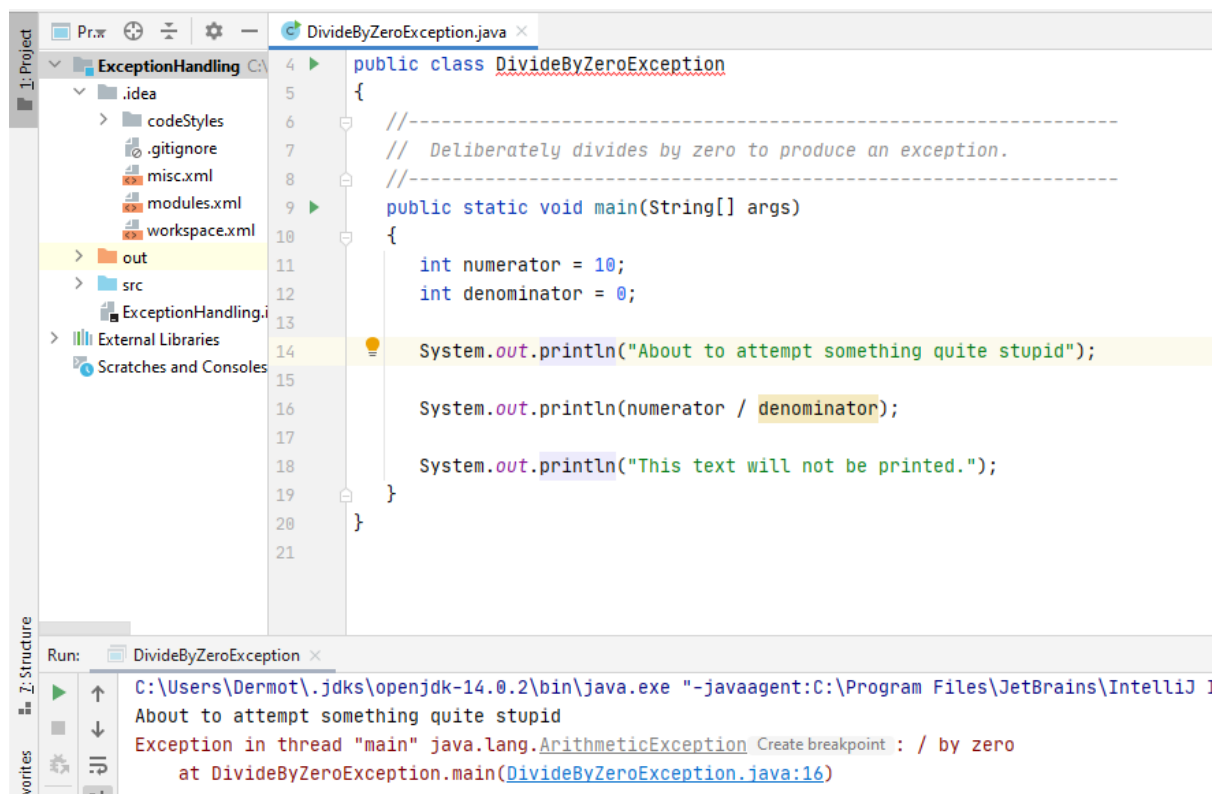
1. not handle the exception at all,
2. handle the exception where it occurs, or
3. handle the exception at another point in the program.

Not Handling the Exception (an uncaught exception)

If a program does not handle the exception at all, it will terminate abnormally and produce a message that describes what exception occurred and where it was produced. The information associated with an exception is often helpful in tracking down the cause of a problem.

Let's look at the output of an exception. The program shown below throws an `ArithmeticException` when an invalid arithmetic operation is attempted. In this case, the program attempts to divide by zero.

Because there is no code in this program to handle the exception explicitly, it terminates when the exception occurs, printing specific information about the exception. Note that the last `println` statement in the program never executes, because the exception occurs first.



```
public class DivideByZeroException
{
    //-----
    // Deliberately divides by zero to produce an exception.
    //-----
    public static void main(String[] args)
    {
        int numerator = 10;
        int denominator = 0;

        System.out.println("About to attempt something quite stupid");

        System.out.println(numerator / denominator);

        System.out.println("This text will not be printed.");
    }
}
```

Run: DivideByZeroException

```
C:\Users\Dermot\.jdk\openjdk-14.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ I
About to attempt something quite stupid
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivideByZeroException.main(DivideByZeroException.java:16)
```

Figure 1: DivideByZeroException

Although the example above is trivial, it simulates an error that could occur in reality, perhaps due to user input.

Handling Exceptions - the try-catch statement

Let's now examine how we catch and handle an exception when it is thrown. The *try-catch* statement identifies a block of statements that may throw an exception. A *catch clause*, which follows a *try* block, defines how a particular kind of exception is handled. A *try* block can have several *catch* clauses associated with it. Each *catch* clause is called an *exception handler*.

When a *try* statement is executed, the statements in the *try* block are executed. If no exception is thrown during the execution of the *try* block, processing continues with the statement following the *try* statement (after all of the *catch* clauses). This situation is the normal execution flow and should occur most of the time.

If an exception is thrown at any point during the execution of the *try* block, control is immediately transferred to the appropriate catch handler if it is present. That is, control transfers to the first *catch* clause whose exception class corresponds to the exception that was thrown. After executing the statements in the *catch* clause, control transfers to the statement after the entire *try-catch* statement.

Although the flowchart below doesn't encapsulate all the possibilities of try-catch, it can serve as a useful enough introduction to the basic concept.

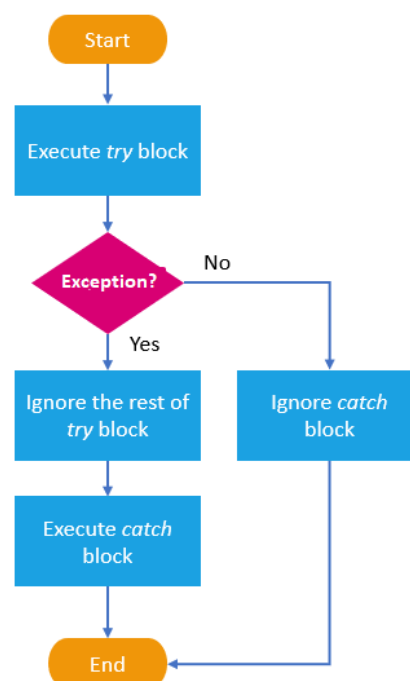


Figure 1: Basic flow of control for try-catch

Below is an example which illustrates the use of a try block with two associated catch blocks (because we realise that there are two obvious issues with user input of the codes).

You can try it out yourself. However, try to anticipate how it will execute before you test with various inputs.

```
import java.util.Scanner;
/**
 * Demonstrates the use of a try-catch block.
 */

/*
A product code includes:
- a character in the tenth position that represents the zone from which that product was made
- a four-digit integer in positions 4 through 7 that represents the district in which it will be sold.

Due to some reorganization, products from zone R are banned from being
sold in districts with a designation of 2000 or higher.

Example Codes:
123456789ABCD <- code position #. I used hex to represent 10,11,12,13

TRD1704A7R-12 <- Valid code and not banned
TRV2475A5R-14 <- Valid code but banned (District 'R' but district code of 2475
TRL2k74A5R-11 <- District is not an integer (NumberFormatException)
TRQ2949A6M-04 <- Valid code
TRV2105A2 <- Improper Length (StringIndexOutOfBoundsException)
*/
public class ProductCodes{
    //-----
    // Counts the number of product codes that are entered with a
    // zone of R and and district greater than 2000.
    //-----
    public static void main(String[] args){
        String code;
        char zone;
        int district, valid = 0, banned = 0;

        Scanner scan = new Scanner(System.in);

        System.out.print("Enter product code (XXX to quit): ");
        code = scan.nextLine();

        while (!code.equals("XXX")){
            try
            {
                zone = code.charAt(9);
                district = Integer.parseInt(code.substring(3, 7));
                valid++;
                if (zone == 'R' && district > 2000)
                    banned++;
            }
            catch (StringIndexOutOfBoundsException exception)
            {
                System.out.println("Improper code length: " + code);
            }
            catch (NumberFormatException exception)
            {
                System.out.println("District is not numeric: " + code);
            }

            System.out.print("Enter product code (XXX to quit): ");
            code = scan.nextLine();
        }
    }
}
```

```
        System.out.println("# of valid codes entered: " + valid);  
        System.out.println("# of banned codes entered: " + banned);  
    }  
}
```

Points to note about the code sample above:

- If there was no try-catch surrounding the code to extract the zone and district, the invalid codes mentioned in the comments would cause the program to crash.
Verify this for yourself!
- At the point an exception occurs, control is immediately transferred to the corresponding catch block.
When the catch block finishes, execution continues at the first line of code outside the try-catch blocks; this is also true if the try block completes successfully.
- You should note that the catch blocks accept an exception argument. This is actually an object that contains information about the exception.
Our code does nothing with this parameter, but we'll return to explore this further quite soon.

Exceptions are “Thrown”

We saw that exceptions are handled by a catch block when they occur.

However, the actual mechanism that actually generates the exception object is done using a `throw`.

What???

`throw` is actually a keyword in Java. We can use it to throw Exception objects and then they will be caught using a `catch` block (if such a block exists.)

However, you might say, “but we never had a `throw` (in the previous example).”

And that’s because it wasn’t our code that threw the exception – it was the Java library code related to our calls to `zone = code.charAt(9); district = Integer.parseInt(code.substring(3, 7));`

Let’s try running the code again without any of our own exception handling.

The following output is displayed for the first problem (that there is no char at index 9):

```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Educational Ed
Enter product code (XXX to quit): TRV2105A2
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 9
    at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)
    at java.base/java.lang.String.charAt(String.java:711)
    at com.dermot.exceptionhandling.ProductCodes.main(ProductCodes.java:46)


Process finished with exit code 1
```

The three lines beginning with ‘`at`’ are called a **stack trace**.

What this means is that our code - `zone = code.charAt(9);` - invoked a call to `java.lang.String.charAt` which, in turn, called `java.lang.StringLatin1.charAt`.

So, the code that actually **threw** the exception was line 48 of the class `StringLatin1` which contains the method `charAt()`.

In fact, we can even click on the file location in the Console and it will bring us directly to the code:

```
final class StringLatin1 {  
  
    public static char charAt(byte[] value, int index) {  
        if (index < 0 || index >= value.length) {  
            throw new StringIndexOutOfBoundsException(index);    
        }  
        return (char)(value[index] & 0xff);  
    }  
}
```

So, basically, when an exception is thrown, the run-time system (the Java Virtual Machine or JVM) looks to see if the current method can handle – catch – the exception. If it can't it proceeds up through the call stack to each level, seeing if there's a corresponding catch-block.

If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then the run-time system hands over the Exception Object to **default exception handler** , which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**:

```
Exception in thread "xxx" Name of Exception : Description  
... .. // Call Stack
```

Illustrative example of how run-time system searches appropriate exception handling code on the call stack

```
// Java program to demonstrate exception is thrown
// how the runTime system searches th call stack
// to find appropriate exception handler.
class ExceptionThrown
{
    // It throws the Exception(ArithmeticException).
    // Appropriate Exception handler is not found within this method.
    static int divideByZero(int a, int b){

        // this statement will cause ArithmeticException(/ by zero)
        int i = a/b;

        return i;
    }

    // The runTime System searches the appropriate Exception handler
    // in this method also but couldn't have found. So looking forward
    // on the call stack.
    static int computeDivision(int a, int b) {

        int res =0;

        try
        {
            res = divideByZero(a,b);
        }
        // doesn't matches with ArithmeticException
        catch(NumberFormatException ex)
        {
            System.out.println("NumberFormatException is occurred");
        }
        return res;
    }

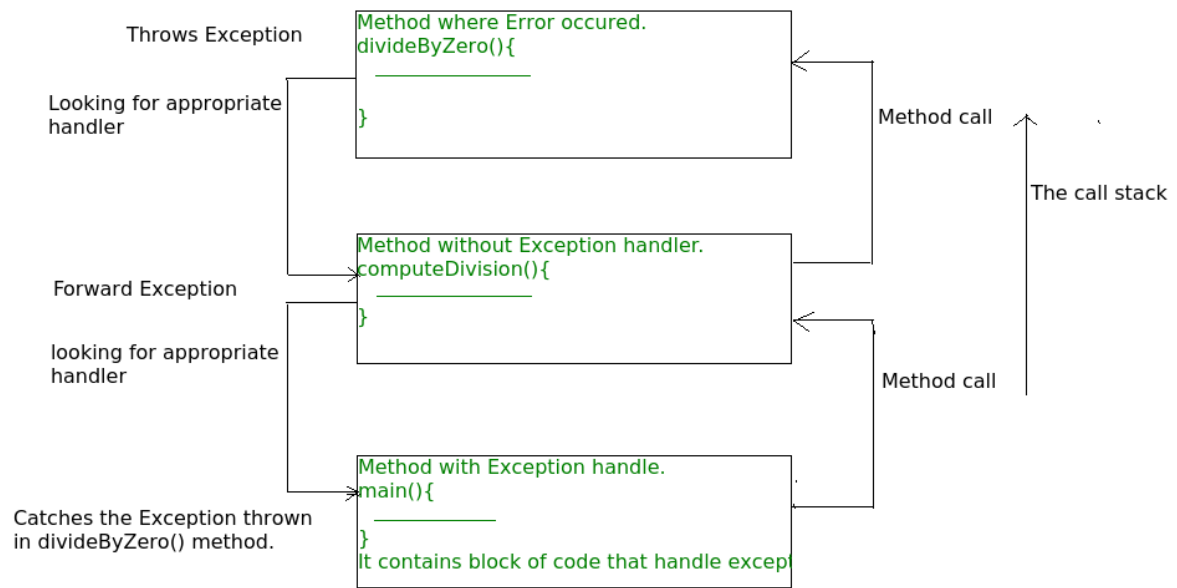
    // In this method found appropriate Exception handler.
    // i.e. matching catch block.
    public static void main(String args[]){

        int a = 1;
        int b = 0;

        try
        {
            int i = computeDivision(a,b);
        }

        // matching ArithmeticException
        catch(ArithmeticException ex)
        {
            // getMessage will print description of exception(here / by zero)
            System.out.println(ex.getMessage());
        }
    }
}
```


Here's the sequence of events presented diagrammatically:

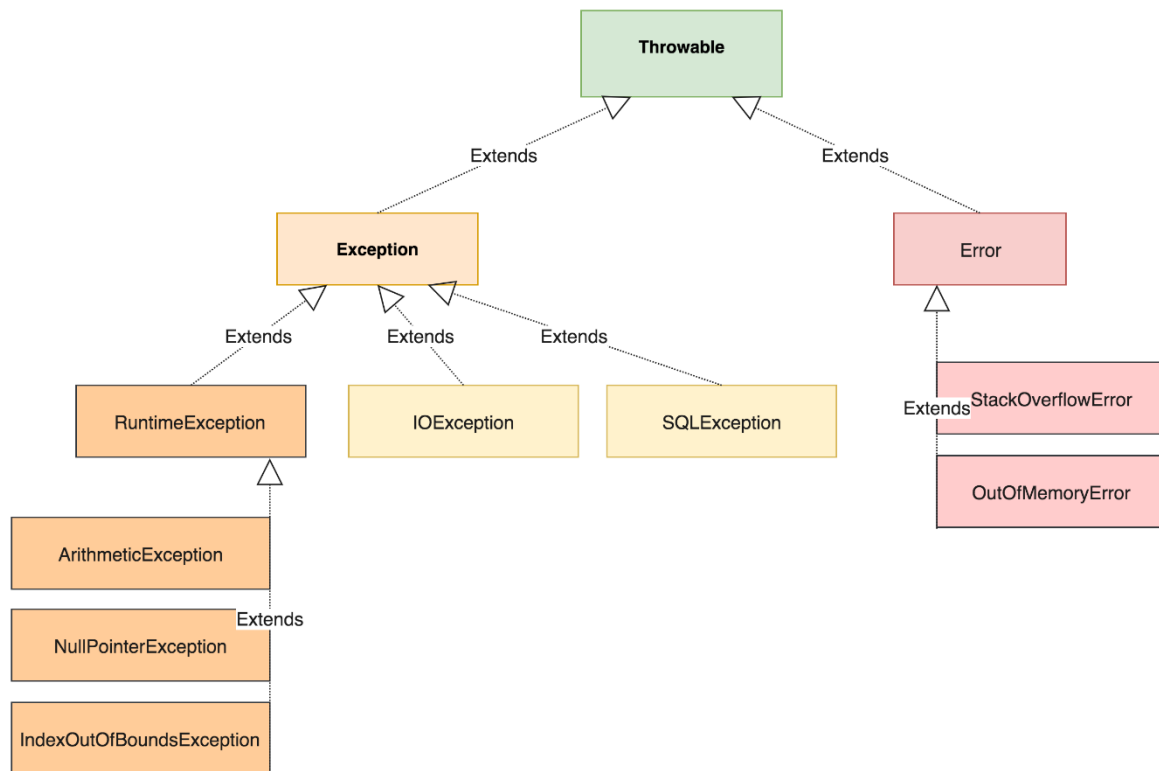


The call stack and searching the call stack for exception handler.

Java Exception Hierarchy

Before I finish this particular lecture, I think it's important to at least see part of Java's Exception Hierarchy (there are more classes than this.)

As you can see, it's an inheritance hierarchy.



Question: What exception types can be caught by the following handler?

```
catch (Exception e) {  
}
```

Question: Is there anything wrong with this exception handler as written? Will this code compile?

```
try {  
}  
catch (Exception e) {  
}  
catch (ArithmeticException a) {  
}
```

Okay, there's more to say about exception handling, but that'll be enough to digest in one lecture.