Nesting `try`s

We've learned to use `try` and `catch` blocks, and we've learned to catch multiple different kinds of exceptions when a `try` block fails to achieve its goals–but what if we need to try different things depending on different circumstances? Java has us covered there, too.

#Java Nested try block

In Java, a `try` block that lives inside of another `try` block is called a **nested** `try` **block**. It works like this:

```
try {
// Code to try before the nested try block goes here.
// This is our nested try:
try {
// Nested code to try
} catch {
// Code to run if the nested try fails
}
} catch {
// Code to run if an exception occurs in the original try, but
is not caught by the nested catch try/catch patterns.
}
```

Read the above **carefully.** Think about what it says, then answer the question below. Note: the question is about the example in the IDE below.
In the nested `try` example **below**, which print line statement won't be printed? Do your best to trace the code and answer before you run it code yourself. After you've run the code, feel free to add more to your answer here:
There was an error outside of the conditions set by the inner trys!

The example:

Run

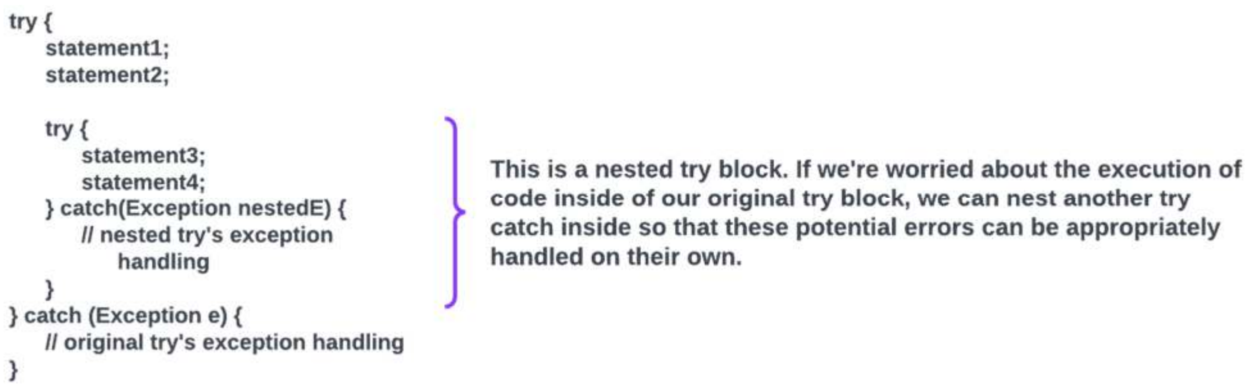
NestedTryBlock.java

```
public class NestedTryBlock{
 public static void main(String args[]){
 // Our outer try block
  try {
    //   int z =39/0;
    try {
        // Our first inner try block:
        System.out.println("Going to divide by 0");
        int b =39/0;
    } catch(ArithmeticException e) {
        //The catch block for our first inner try block:
        System.out.println("You divided by zero: " + e);
    }
    // Our second inner try block
    try{
        int a[]=new int[5];
        // Attempting to assign a value outside of the array's boundaries:
        a[5]=4;
    } catch(ArrayIndexOutOfBoundsException e) {
        // Catch block for our second inner try:
        System.out.println("You attempted to assign a value to an out of bound
s index: " + e);
    }
    System.out.println("The inner try blocks have finished running, and our in
itial try is complete.");
  }  catch(Exception e) {
    // Catch block for our original outer try:
    System.out.println("There was an error outside of the conditions set by th
e inner trys!");
  }
  System.out.println("Try blocks have been run and we can now return to our re
gularly scheduled program.");
  }
}
```

Once you understand your answer, check ours and compare. You're welcome to add more to your short answer above, if you like, and keep it as a place you can return to for clarity in the future.
Reveal Content

#Why Use Nested `try` Blocks?

The example above was a bit simplistic. After all, the outer try was fairly extraneous, given that we had nested `try`s to check the conditions that the outer one would have otherwise covered. However, a situation may arise where a part of a block may cause one exception and the entire block itself may cause another exception. In such cases, exception handlers should be nested so that each individual exception can be handled appropriately.



```
try {
    statement1;
    statement2;

    try {
        statement3;
        statement4;
    } catch(Exception nestedE) {
        // nested try's exception
            handling
    }
} catch (Exception e) {
    // original try's exception handling
}
```

This is a nested try block. If we're worried about the execution of code inside of our original try block, we can nest another try catch inside so that these potential errors can be appropriately handled on their own.

Annotated example of nested try block

In the example image here, there are statements that belong to the initial `try` block itself. The evaluation of these statements would, therefore, trigger the outer `catch` exception if they were to fail. Similarly, if our inner `try` blocks here didn't have appropriate exception handling, the exceptions thrown by their attempts to run would be caught by the outer `catch` block. In this way, we have a couple of chances to `catch` these exceptions, but remember: **our goal is to catch the exceptions we can anticipate**. This is why we call them **exceptions**, and it's why we have `try`/`catch` blocks at all. We want to appropriately handle everything we can, and plan for the contingencies that matter most.

#Summary

- If there are exceptions that can occur in the subprocesses of a larger `try` block, we should nest more try/catch blocks inside of the original `try` and handle their exceptions appropriately.
- When a `try` block does not have a `catch` block for a particular exception, the outer parent's `try` block will be checked for that exception. If the exception occurs and the error matches what the `catch` was looking for, the outer `catch`'s code will be run.
- If you want a `catch` block to handle all general exceptions, remember to simply pass it the parameter `Exception e`. This will ensure that all exceptions in this block are caught. Note: you can rename `e` if you like, but it's common practice to just put `e`.

- If none of your `catch` blocks has been created specifically to handle an exception and you have no default `catch`, the Java runtime system will handle the exception itself, and display its auto-generated message.
- Inner `catch` blocks don't catch the outer block's exceptions. The outer block needs its own `catch` block, otherwise, the Java runtime system will have to handle it.

#Knowledge Check

If the outer `catch` block of the outer try/catch hasn't been run, which of the following may could have happened? Select all that apply.

There were no exceptions in the try block.

There were no exceptions even in the inner try blocks.

There were unspecified errors in the outer try block that occurred.

There were exceptions, but they were all caught by inner try/catch blocks.

#Extra Resources

If you'd like to check out some more examples of try/catch, you can do so here.