Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java : try-catch-finally/5b1e5b727becc0459deae27d

TUTORIAL

Java : try-catch-finally

Topics

Statements for which exception processing is to be performed are surrounded by a try statement with a valid catch or finally clause (or both). The syntax of the try statement is as follows:

```
try TryBlock CatchClauses FinallyClause;
```

At least one catch clause or finally clause must be defined. More than one catch clause may be used, but no more than one finally clause may be identified. The try block is a sequence of Java statements that are preceded by an opening brace ({) and followed by a closing brace (}). The catch clauses are a sequence of clauses of the form:

```
catch (Parameter) {
/*
* Exception handling statements
*/
}
```

The Parameter is a variable that is declared to be a subclass of Throwable. The statements within the catch clause are used to process the exceptions that they "catch," as I'll explain shortly. The finally clause identifies a block of code that is to be executed at the conclusion of the try statement and after any catch clauses. Its syntax is as follows:

```
finally {
/*
* Statements in finally clause
*/
}
```

The finally clause is always executed, no matter whether or not an exception is thrown. The try statement executes a statement block. If an exception is thrown during the block's execution, it terminates execution of the statement block and checks the catch clauses to determine which, if any, of the catch clauses can catch the thrown exception. If none of the catch clauses can catch the exception, the exception is propagated to the next level try statement. This process is repeated until the exception is caught or no more try statements remain. A catch clause can catch an exception if its argument may be legally assigned the object thrown in the throw statement. If the argument of a catch clause is a class, the catch clause can catch any object whose class is a subclass of this class. The try statement tries each catch clause in order, and selects the first one that can catch the exception that was thrown. It then executes the statements in the catch clause. If a finally clause occurs in the try statement, the statements in the finally clause are executed after execution of the catch clause has been completed. Execution then continues with the statement following the try statement.

When an exception is caught in the catch clause of a try statement, that exception may be re-thrown. When an exception is re-thrown, it can then be caught and processed by the catch clause of a higher level try statement. A higher-level catch clause can then perform any secondary clean-up processing.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, includes a catch clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a try block and a catch clause which processes the ArithmeticException generated by the division-by-zero error:

Java

```java
class Main {
  public static void main(String args[]) {
    int d, a;
    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    }
    catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}

```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Put differently, catch is not “called,” so execution never “returns” to the try block from a catch. Thus, the line “This will not be printed.” is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

try statement

> *try statement*
>
> *exception is generated*
>
> *catch clause(s) of inner try statement*

catch clause(s) of outer try statement

other higher levels of the exception-handling try statement

So, java provides a finally clause to execute the code with 100% guarantee. Consider the following: -

```java
class Main {
  public static void main(String args[]) {
    int d, a;
    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
    }
    catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    finally {   // finally block will always be executed
      System.out.println("This will be printed.");
    }
    System.out.println("After catch statement.");
  }
}

```

If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks sequentially. If a catch-block catches that exception class or catches a superclass of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.

If none of the catch-block matches, the exception will be passed up the call stack. The current method executes the finally clause (if any) and popped off the call stack. The caller follows the same procedures to handle the exception. The finally block is almost certain to be executed, regardless of whether or not exception occurs (unless JVM encountered a severe error or a System.exit() is called in the catch block).

Java

```java
import java.util.Scanner;
// Other imports go here
// Do NOT change the class name
class Main
{
  public static void main(String[] args)
  {
    System.out.println("Enter main()");
    methodA();
    System.out.println("Exit main()");
  }

  public static void methodA()
  {
    System.out.println("Enter methodA()");
    try
    {
      System.out.println(1 / 0);
      // A divide-by-0 triggers an ArithmeticException - an unchecked exception
      // This method does not catch ArithmeticException
      // It runs the "finally" and popped off the call stack
    } finally
    {
      System.out.println("finally in methodA()");
    }
    System.out.println("Exit methodA()");
  }
}
```

try-catch-finally

- A try-block must be accompanied by at least one catch-block or a finally-block.
- You can have multiple catch-blocks. Each catch-block catches only one type of exception.

- A catch block requires one argument, which is a `throwable` object (i.e., a subclass of `java.lang.Throwable`), as follows:

```
catch (AThrowableSubClass aThrowableObject) {
   // exception handling codes
}
```

- You can use the following methods to retrieve the type of the exception and the state of the program from the `Throwable` object:
- `printStackTrace()`: Prints this `Throwable` and its call stack trace to the standard error stream `System.err`. The first line of the outputs contains the result of `toString()`, and the remaining lines are the stack trace. This is the most common handler, if there is nothing better that you can do. For example,

```
try {
   Scanner in = new Scanner(new File("test.in"));
   // process the file here
   ......
} catch (FileNotFoundException ex) {
   ex.printStackTrace();
}
```

- You can also use `printStackTrace(PrintStream s)` or `printStackTrace(PrintWriter s)`.
- `getMessage()`: Returns the `message` specified if the object is constructed using constructor `Throwable(String message)`.
- `toString()`: Returns a short description of this `Throwable` object, consists of the name of the class, a colon `':'`, and a message from `getMessage()`.
- A catch block catching a specific exception class can also catch its *subclasses*. Hence, `catch(Exception ex) {...}` catches all kinds of exceptions. However, this is not a good practice as the exception handler that is too general may unintentionally catches some subclasses' exceptions it does not intend to.
- The order of catch-blocks is important. A subclass must be caught (and placed in front) before its superclass. Otherwise, you receive a compilation error "exception `XxxException` has already been caught".
- The finally-block is meant for cleanup code such as closing the file, database connection regardless of whether the try block succeeds.

The finally block is always executed (unless the catch-block pre-maturely terminated the current method).

Some important points about exceptions in java

- Certainly not advisable other than writing toy programs. But to bypass the compilation error messages triggered by methods declaring unchecked exceptions, you could declare "`throws Exception`" in your `main()` (and other methods), as follows:

```
public static void main(String[] args) throws Exception {  // 
throws all subclass of Exception to JRE
   Scanner in = new Scanner(new File("test.in"));   // declares 
"throws FileNotFoundException"
   ......
   // other exceptions
}
```

- An overriding method cannot declare exception types that were not declared in its original. However, it may declare exception types are the same as, or subclass of its original. It needs not declare all the exceptions as its original. It can throw fewer exceptions than the original, but not more.