Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java : Exception Handling/5b1e5a277becc0459deae232

TUTORIAL

Java : Exception Handling

Topics

Java provides superior support for runtime error and exception handling, enabling programs to check for anomalous conditions and respond to them with minimal impact on the normal flow of program execution. This allows error- and exception-handling code to be added easily to existing methods. Exceptions are generated by the Java runtime system in response to errors that are detected as classes are loaded and their methods are executed. The runtime system is said to throw these runtime exceptions. Runtime exceptions are objects of the classes java.lang.RuntimeException, java.lang.Error, or of their subclasses. Runtime exceptions are also referred to as unchecked exceptions.

Exceptions may also be thrown directly by Java code using the throw statement. These exceptions are thrown when code detects a condition that could potentially lead to a program malfunction. The exceptions thrown by user programs are generally not objects of a subclass of RuntimeException. These non-runtime exceptions are referred to as checked exceptions or program exceptions. Both program and runtime exceptions must be caught in order for them to be processed by exception handling code. If a thrown exception is not caught, its thread of execution is terminated, and an error message is displayed on the Java console window.

The approach used by Java to catch and handle exceptions is to surround blocks of statements for which exception processing is performed by a try statement. The try statement contains a catch

clause that identifies what processing is to be performed for different types of exceptions. When an exception occurs, the Java runtime system matches the exception to the appropriate catch clause. The catch clause then handles the exception in an appropriate manner. An optional finally clause is always executed whether or not an exception is thrown or caught. This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Here, ExceptionType is the type of exception that has occurred. All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing. The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type Error, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

An simple example for Exception Handling is:

```java
class Main
{
  public static void main(String args[])
  {
    int d, a;
    try { // monitor a block of code.
      d = 0;
      a = 55 / 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.");
  }
}
```

Multiple Catch Statements

We can have as many catch statements as we want after a try block. The most appropriate catch will be called whenever a exception is occurred. For example,

```java
// Demonstrate multiple catch statements.
class Main {
  public static void main(String args[]) {
    try {
      int a = 0;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
```

```
    }
    System.out.println("After try/catch blocks.");
  }
}

```

It will call ArithmeticException, but if we set a to a non-zero value then, it will call ArrayIndexOutOfBoundsException as c[42] will raise an exception.

The throw Statement

Exceptions are thrown using the throw statement. Its syntax is as follows:

```
throw Expression;
```

Note What to Throw? A throw statement can throw an object of any class that is a subclass of java.lang.Throwable; however, it is wise to stick with the standard convention of only throwing objects that are a subclass of class Exception.

Expression must evaluate to an object that is an instance of a subclass of the java.lang.Throwable class. When an exception is thrown, execution does not continue after the throw statement. Instead, it continues with any code that catches the exception. If an exception is not caught, the current thread of execution is terminated, and an error is displayed on the console window. Specifically, the uncaughtException() method of the current ThreadGroup is invoked to display the error message.

For example, the following statement will throw an exception, using an object of class ExampleException:

```
throw new ExampleException();
```

The new operator is invoked with the ExampleException() constructor to allocate and initialize an object of class ExampleException. This object is then thrown by the throw

statement. A method's throws clause lists the types of exceptions that can be thrown during a method's execution. The throws clause appears immediately before a method's body in the method declaration. For example, the following method throws the ExampleException:

```
public void exampleMethod() throws ExampleException {
throw new ExampleException();
}
```

When more than one exception can be thrown during the execution of a method, the exceptions are separated by commas in the throws clause. For example, the following method can throw either the Test1Exception or the Test2Exception:

```
public void testMethod(int i) throws Test1Exception,
Test2Exception
{
if(i==1) throw new Test1Exception();
if(i==2) throw new Test2Exception();
}
```

Note Runtime Exceptions and Errors Because runtime exceptions or errors can occur almost anywhere in a program's execution, the catch or declare requirement does not apply to them.

The types identified in the throws clause must be capable of being legally assigned to the exceptions that may be thrown. In other words, the class of the thrown exception must be castable to the class of the exceptions identified in the throws clause. If a program exception can be thrown during the execution of a method, the method must either catch the exception or declare the exception in its throws clause. This rule applies even if the exception is thrown in other methods that are invoked during the method's execution.

For example, suppose that method A of object X invokes method B of object Y, which invokes method C of object Z. If method C throws an exception, it must be caught by method C or declared in method C's throws clause. If it is not caught by method C, it must be caught by method B or declared in method B's throws clause. Similarly, if the exception is not caught by method B, it must be caught by method

A or declared in method A's throws clause. The handling of exceptions is a hierarchical process that mirrors the method invocation hierarchy (or call tree). Either an exception is caught by a method and removed from the hierarchy, or it must be declared and propagated back through the method invocation hierarchy.