Exception Handling Keywords in Java

Java exception handling is managed via five keywords. In this article, we will learn all these five keywords, and how to use them with examples.

Overview

In Java, exception handling is facilitated using five primary keywords: `try`, `catch`, `throw`, `throws`, and `finally`.
1. try: The `try` block is used to enclose a segment of code that might throw an exception, ensuring that any exception arising from the enclosed code can be gracefully managed.

2. catch: The `catch` block follows the try block and defines how to handle specific types of exceptions. Multiple catch blocks can be used after a single try to handle different exception types individually.

3. throw: The `throw` keyword is employed to manually trigger or throw an exception from the code. It's often used in conjunction with user-defined or system exceptions to indicate when something abnormal occurs.

4. throws: Used in method signatures, the `throws` keyword indicates that the method might throw specified exceptions. It's a way of notifying callers that they should be prepared to handle (or further propagate) these exceptions.

5. finally: The `finally` block, used after the try-catch structure, ensures that a particular segment of code runs regardless of whether an exception was thrown in the try block. This is typically used for cleanup operations, like closing resources.

Let's discuss each of the above 5 exception-handling keywords with syntax and examples.

1. try Block

Enclose the code that might throw an exception within a `try` block. If an exception occurs within the try block, that exception is handled by an exception handler associated with it. The try block contains at least one `catch` block or `finally` block.

The syntax of the try-catch block:

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

The syntax of a try-finally block:

```
try{
//code that may throw exception
}finally{}
```

Example:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println(e.getMessage());
}
```

In the above code, dividing by zero will cause an *ArithmeticException*. The statements inside the try block are where we anticipate this error might occur.

Nested try block

The try block within a try block is known as a nested try block in java.

Why use nested try block?

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
public class NestedTryBlock {

        public static void main(String args[]) {
                try {
                        try {
```

```
                                        System.out.println(" This gives divide by zero
error");
                                        int b = 39 / 0;
                                } catch (ArithmeticException e) {
                                        System.out.println(e);
                                }

                                try {
                                        System.out.println(" This gives Array index out of
bound exception");
                                        int a[] = new int[5];
                                        a[5] = 4;
                                } catch (ArrayIndexOutOfBoundsException e) {
                                        System.out.println(e);
                                }

                                System.out.println("other statement");
                        } catch (Exception e) {
                                System.out.println("handeled");
                        }

                        System.out.println("normal flow..");
                }
}
```

2. catch Block

Java catch block is used to handle the Exception. It must be used after the *try* block only. You can use multiple catch blocks with a single try.

Syntax:

```
try
{
      //code that cause exception;
}
catch(Exception_type  e)
{
      //exception handling code
}
```

Examples:

Example 1: catch *ArithmeticException* exception:

```
public class Arithmetic {

 public static void main(String[] args) {

  try {
      int result = 30 / 0; // Trying to divide by zero
  } catch (ArithmeticException e) {
       System.out.println("ArithmeticException caught!");
  }
      System.out.println("rest of the code executes");
 }
}
```

Output:

```
ArithmeticException caught!
rest of the code executes
```

Example 2: catch *ArrayIndexOutOfBoundsException* exception:

```
try {
    int[] arr = {1, 2, 3};
    System.out.println(arr[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index is out of bounds!");
}
```

In this code, we're trying to access an index that doesn't exist in the array. The corresponding catch block catches this exception and handles it by printing a custom error message.

Multi-catch Block

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

Example: Consider a scenario where we want to parse an integer from a string and then use that integer as an array index. Both of these operations can throw exceptions - **NumberFormatException** and **ArrayIndexOutOfBoundsException**.

```
public class MultiCatchExample {

    public static void main(String[] args) {
        String numStr = "10a";  // This will cause NumberFormatException
        int[] numbers = {1, 2, 3, 4, 5};

        try {
            int num = Integer.parseInt(numStr);   // Parsing integer from string
            System.out.println(numbers[num]);     // Accessing array element
        } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Output:

```
An error occurred: For input string: "10a"
```

In the example above, the try block contains two statements, each of which can throw an exception. The catch block can handle both exceptions because of the multi-catch feature.

3. throw Keyword

The *throw* keyword is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exceptions using the *throw* keyword. The throw keyword is followed by an instance of the exception.

Syntax:

```
throw exception_instance;
```

Example:

Let's consider a simple example where we have a method *setAge()* that sets the age of a person. If someone tries to set a negative age, it's clearly an incorrect value. In such a case, we can throw an **IllegalArgumentException**.

```
public class ThrowExample {
```

```
    private int age;

    public void setAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative!");
        }
        this.age = age;
    }

    public static void main(String[] args) {
        ThrowExample person = new ThrowExample();

        try {
            person.setAge(-5);   // This will cause an exception
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output:

```
Error: Age cannot be negative!
```

In the *setAge* method, if the age provided is negative, we throw an **IllegalArgumentException** with a relevant message.

4. throws Keyword

The *throws* keyword is used to declare exceptions. It doesn't throw an exception but specifies that a method might throw exceptions. It's typically used to inform callers of the exceptions they might encounter.

Syntax:

```
return_type method_name() throws exception_class_name{
//method code
}
```

Example:

```
public class ExceptionHandlingWorks {

    public static void main(String[] args) {
```

```
        exceptionHandler();
    }

    private static void exceptionWithoutHandler() throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader(new
File("/invalid/file/location")))) {
            int c;
            // Read and display the file.
            while ((c = reader.read()) != -1) {
                System.out.println((char) c);
            }
        }
    }

    private static void exceptionWithoutHandler1() throws IOException {
        exceptionWithoutHandler();
    }

    private static void exceptionWithoutHandler2() throws IOException {
        exceptionWithoutHandler1();
    }

    private static void exceptionHandler() {
        try {
            exceptionWithoutHandler2();
        } catch (IOException e) {
            System.out.println("IOException caught!");
        }
    }
}
```

5. finally Block

- Java *finally* block is a block that is used to execute important code such as closing connection, stream, etc.
- Java *finally* block is always executed whether an exception is handled or not.
- Java *finally* block follows try or catch block.
- For each try block, there can be zero or more catch blocks, but only one *finally* block.
- The *finally* block will not be executed if the program exits(either by calling *System.exit()* or by causing a fatal error that causes the process to abort).

Syntax:

```
try {
    // Code that might throw an exception
} catch (ExceptionType1 e1) {
```

```
    // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Code to handle ExceptionType2
}
// ... more catch blocks if necessary ...
finally {
    // Code to be executed always, whether an exception occurred or not
}
```

Example 1:

In this example, we have used **FileInputStream** to read the *simple.txt* file. After reading a file the resource **FileInputStream** should be closed by using *finally* block.

```
public class FileInputStreamExample {
    public static void main(String[] args) {

        FileInputStream fis = null;
        try {
            File file = new File("sample.txt");
            fis = new FileInputStream(file);
            int content;
            while ((content = fis.read()) != -1) {
                // convert to char and display it
                System.out.print((char) content);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Example 2:

When we use a JDBC connection to communicate with a database. The resource that we have used are *ResultSet*, *Statement*, and *Connection* (in that order) should be closed in a *finally* block when you are done with them, something like that:

```
Connection conn = null;
```

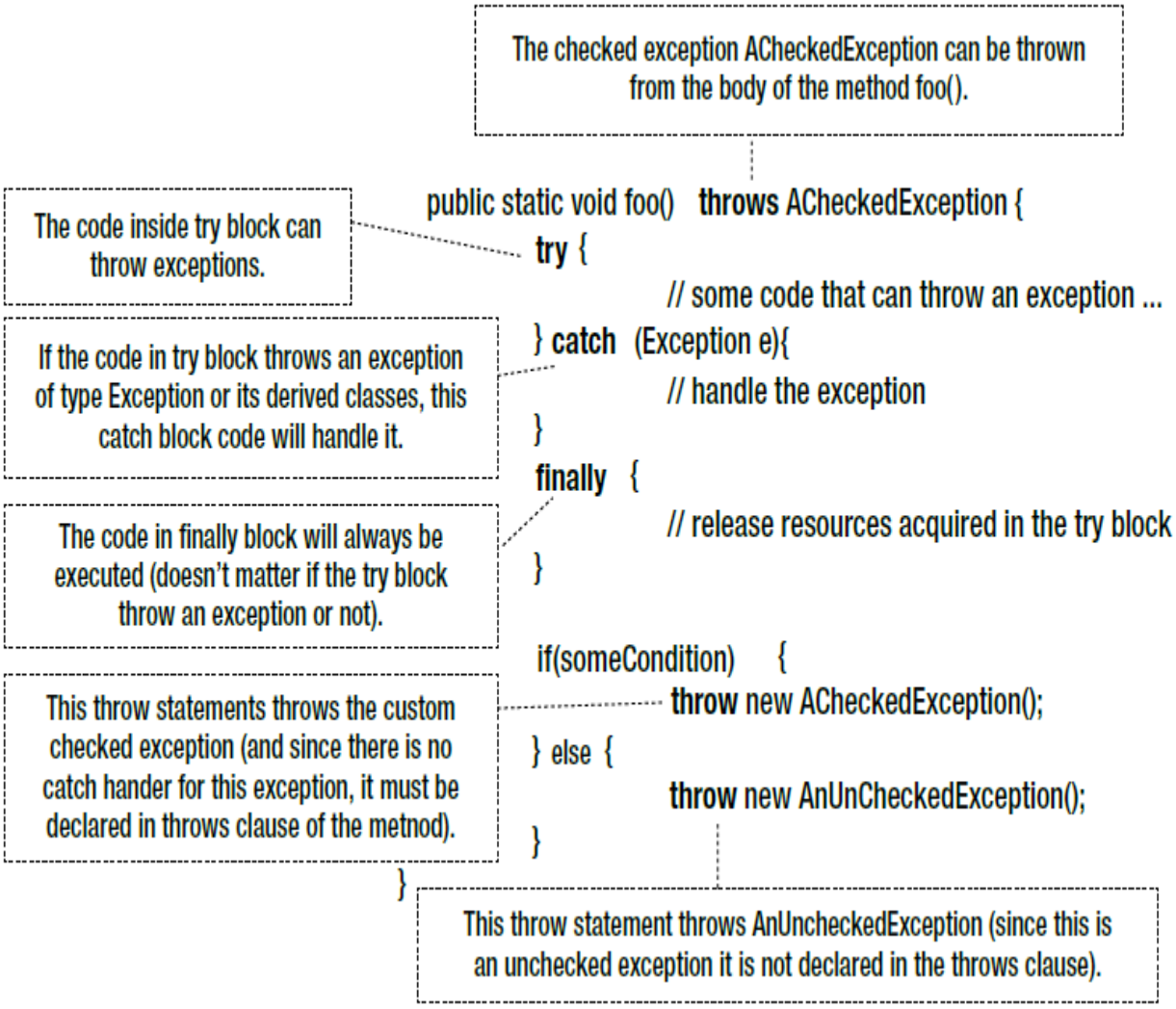
```
PreparedStatement ps = null;
ResultSet rs = null;

try {
    // Do stuff
    ...

} catch (SQLException ex) {
    // Exception handling stuff
    ...
} finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException e) { /* ignored */}
    }
    if (ps != null) {
        try {
            ps.close();
        } catch (SQLException e) { /* ignored */}
    }
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) { /* ignored */}
    }
}
```

Summary

This diagram summarizes the usage of try, catch, throw, throws, and finally keywords:


The checked exception ACheckedException can be thrown from the body of the method foo().
The code inside try block can throw exceptions.
If the code in try block throws an exception of type Exception or its derived classes, this catch block code will handle it.
The code in finally block will always be executed (doesn't matter if the try block throw an exception or not).
This throw statements throws the custom checked exception (and since there is no catch hander for this exception, it must be declared in throws clause of the metnod).
This throw statement throws AnUncheckedException (since this is an unchecked exception it is not declared in the throws clause).
public static void foo() throws ACheckedException {
try {
// some code that can throw an exception ...
} catch (Exception e){
// handle the exception
}
finally {
// release resources acquired in the try block
}
if(someCondition) {
throw new ACheckedException();
} else {
throw new AnUnCheckedException();
}
}