Exception Handling

An exception is an abnormal condition that alters or interrupts the flow of execution. Java enables built-in exception handling to deal with such conditions.

Exception handling in Java bases on the following concepts:

- **Protected block.** Statements can be protected against occurence of exceptions/errors.
- **Exception handler.** A protected block has one or more exception handler(s), which catch possible exceptions/errors.
- **Exception.** An error will be indicated by causing an exception. If an error occurs the corresponding exception handler will automatically be searched and encountered.

The Exception Hierarchy

All exceptions and errors inherit from the class `Throwable`, which inherits from the class `Object`.

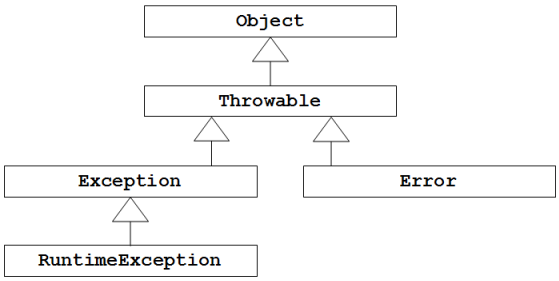


Figure 0.1: The Exception Hierarchy - An Overview.

Checked/Unchecked Exceptions and Errors

There are three categories of exceptions and errors: checked exceptions, unchecked exceptions, and errors.

Checked Exceptions

- ... are checked by the compiler at compile time.
- methods that throw a checked exception must show so in the method declaration using the `throws` clause. This must be done (all the way up the calling stack) until the exception is handled.
- all ... must be be explicitly caught with a `catch` block.
- ... include exceptions of the type `Exception`, and all classes that are subtypes of `Exception`, except for `RuntimeException` and its subtypes.

Algorithm 1 Method that throws a checked exception - An Example.

```
1 // Method declaration that throws and IOException
2 void readFile (String filename) throws IOException {
3     ...
4 }
```

Unchecked Exceptions

- ... are not checked by the compiler at compile time.
- ... occur during runtime due to programmer error (out-of-bounds index, divide by zero, and null pointer exception) or system resource exhaustion.
- ... do not have to be caught.
- methods that may throw an unchecked exception do not have to (but can) show this in the method declaration.
- ... include exceptions of the type `RuntimeException` and all its subtypes.

Errors

- ... are typically unrecoverable (dt. unbehebbar) and present serious conditions.
- ... are not checked at compile time and do not have to be (but can be) caught/handled.

Any checked exceptions, unchecked exceptions, or errors can be caught.

Exception Handling Keywords

Java cleanly separates error-handling from error-generating code. Code that generates the exceptions is said to "throw" an exception, whereas code that handles the exception is said to "catch" the exception:

Algorithm 2 Declaration of an Exception - An Example

```
public void mA() throws IOException {
    ...
    throw new IOException();
    ...
}
```

Algorithm 3 Catching an Exception - An Example.

```
public void mB() {
    ...
    /* Call to method mA must be in a try/catch block
       since the exception is a checked exception;
       otherwise mB could throw the exception */
    try {
       mA();
    } catch (IOException ioe) {
       System.err.println(ioe.getMessage());
       ioe.printStackTrace();
    }
}
```

The `throw` Keyword

... is used to throw an exception. Any checked/unchecked exception and error can be thrown:

```
if (n == -1) throw new EOFException();
```

The `try/catch/finally` Keywords

... are used in block to handle thrown exceptions in Java:

```java
try {
   m();
}
catch (EOFException eofe) {
   eofe.printStackTrace();
}
catch (IOException ioe) {
   ioe.printStackTrace();
}
finally {
   /* cleanup */
}
```

The `try-catch` Statement

... includes one `try` and one or more `catch` blocks.

The `try` block contains code that may throw exceptions. All checked exceptions that may be thrown must have a `catch` block to handle the exception. If no exceptions are thrown, the `try` block terminates normally. A `try` block must have at least one `catch` or `finally` block associated with it.

There cannot be any code between the `try` block and any of the `catch` blocks or the `finally` block.

The `catch` block(s) contain code to handle thrown exceptions. It should never be empty because such "silencing" results makes errors harder to debug.

Within a `catch` clause, a new exception may also be thrown.

The order of the `catch` clauses in a `try/catch` block defines the precedence/ranking for catching exceptions. Always begin with the most specific exception and end with the most general.

The `try-finally` Statement

... includes one `try` and one `finally` block.

The `finally` block is used for releasing resources when necessary. This block is optional and is only used where needed. When used, the `finally` block will always be executed, whether or not the `try` block terminates normally. If the `finally` block throws an exception, it must be handled.

```java
public void testMethod() throws IOException {
   FileWrite fw = new FileWriter("\\data.txt");
   try {
      fw.write("... information ...");
   } finally {
      fw.close();
   }
}
```

The `try-catch-finally` Statement

... includes one `try`, one or more `catch` blocks, and one `finally` block. If the `finally` block throws an exception, it must be handled.

The `try-with-resources` Statement

... is used for declaring resources that must be closed when they are no longer needed. Theses resources are declared in the `try` block.

The `multi-catch` Statement

... is used to allow for multiple exception arguments in one catch clause.

```java
boolean isTest = false;
public void testMethod() {
   try {
      if (isTest) { throw new IOException(); }
```

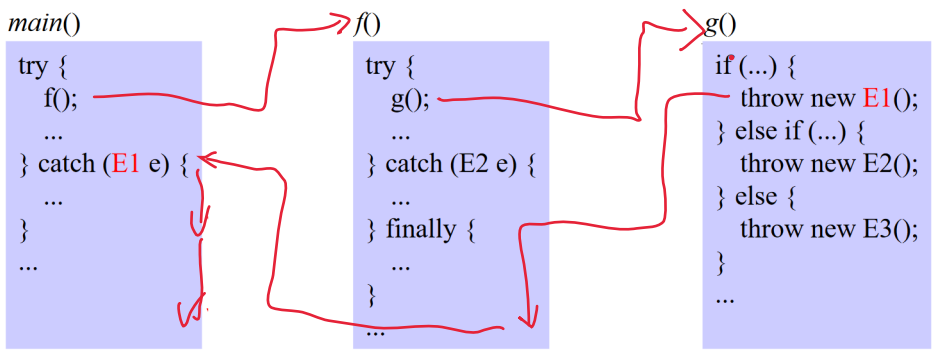
```
        else { throw new SQLException(); }
    } catch (IOException | SQLException e) {
        e.printStackTrace();
    }
}
```

The Exception Handling Process

The following describes the steps of the exception handling process:

1. An exception is encountered (dt. antreffen, darauf gestoßen), which results in an exception object being created.
2. A new exception object is thrown.
3. The runtime system looks for code to handle the exception:
   (a) It begins with the method in which the exception object was created.
   (b) If no handler is found, it traverses the call stack (the ordered list of methods) in reverse looking for an exception handler.
   (c) If the exception is not handled, the program exits and a stack trace is automatically output.
4. The runtime system hands the exception object off to an exception handler to handle (catch) the exception.

Example 1.



Explanation Example 1: Throw new E1();

keine try-Anweisung in g() => Bricht g() ab

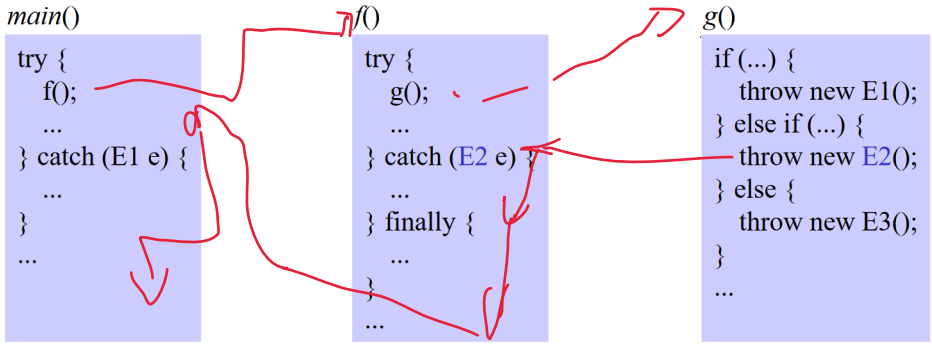
kein passender catch-Block in f() => führt finally-Block in f() aus und bricht f() dann ab

führt catch-Block für E1 in main() aus

setzt nach try-Anweisung in main() fort

Figure 0.2: Example 1: throw new E1();

Example 2.



keine try-anweisung in g() => bricht g() ab

führt catch-Block für E2 in f() aus

führt finally-Block in f() aus

setzt nach try Anweisung in f() fort

Figure 0.3: Example 2: Explanation Example 2. throw new E2();

Example 3.

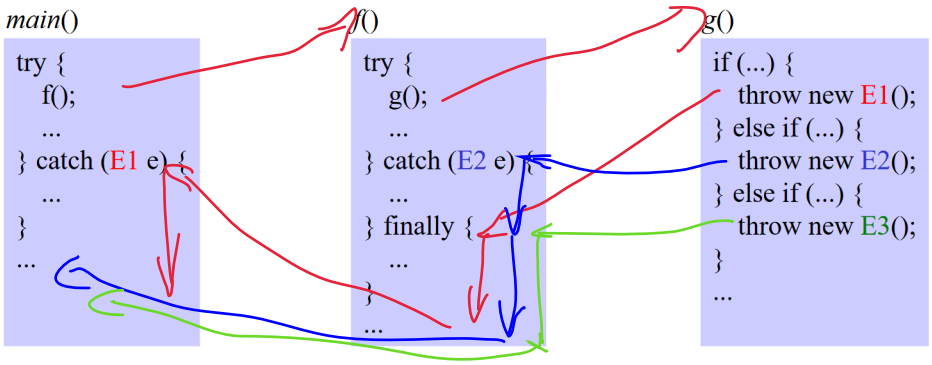


Figure 0.4: Example 3: throw new E3 ();

throw new E3()

Compiler meldet einen Fehler, weil E3 nirgendwo in der Ruferkette abgefangen wird

fehlerfreier Fall

führt g() zu ende aus

führt try block zu ende Aus

führt finally block in f() aus

setzt nach finally block in f() fort

Defining Your Own Exception Class

Java allows creating programmer-defined exceptions. However, the Java exceptions should be reused wherever possible:

- To define a checked exception, the new exception class must extend the `Exception` class, directly or indirectly.
- To define an unchecked exception, the new exception class must extend the `RuntimeException` class, directly or indirectly.
- To define an unchecked error, the new error class must extend the `Error` class.

User-defined exceptions should have at least two constructors - a constructor that does not accept any arguments and a constructor that does.

Algorithm 4 User-defined Exception - An Example

```
1 public class ReportException extends Exception {
2    public ReportException () {}
3    public ReportException (String message, int reportId) { ... }
4 }
```

Printing Information About Exceptions

The class `Throwable` includes methods for providing information about thrown exceptions: `getMessage()`, `toString()`, and `printStackTrace()`. Normally, one of these methods should be called in the catch clause handling the exception.

The `getMessage()` Method

... returns a detailed message string about the exception.

The `toString()` Method

... returns a detailed message string about the exception, including its class name.

The `printStackTrace()` Method

... returns a detailed message string about the exception, including its class name and a stack trace from where the error was caught, all the way back to where it was thrown.

Do It

1. Explain the difference between checked and unchecked exception.
2. Explain the `throws` clause in context with method declaration.
3. Is this code correct or is there anything wrong? Explain.

```
1 try {
2 }
3 catch (Exception e) {
4 }
5 catch (ArithmeticException a) {
6 }
```

4. Is this code correct or is there anything wrong? Explain.

```
1 try {
2 }
3 finally {
4 }
```

5. What is the output of the following fragment if

(a) 1

(b) 0

(c) abc

is entered?

```
Scanner in = new Scanner (System.in);
try {
   int num = in. nextInt ();
   if (num != 0) {
      throw new Exception (" Nicht Null ");  }
      System.out.println ("OK");
   }
   catch ( InputMismatchException ex) {
      System.out.println (" Keine Zahl ");
   }
   catch ( Exception ex) {
      System.out.println (" Fehler : " + ex.getMessage ());
   }
   finally {
      System.out.println (" Ciao ");
   }
```