(/)

Java Exceptions Interview Questions (+ Answers)

Last modified: November 11, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/) +

Exception (https://www.baeldung.com/tag/exception/)

Interview (https://www.baeldung.com/tag/interview/)

This article is part of a series:

1. Overview

Exceptions are an essential topic that every Java developer should be familiar with. This article provides answers to some of the questions that might pop up during an interview.

2. Questions

Q1. What is an exception?

An exception is an abnormal event that occurs during the execution of a program and disrupts the normal flow of the program's instructions.

Q2. What is the purpose of the *throw* and *throws* keywords?

The *throws* keyword is used to specify that a method may raise an exception during its execution. It enforces explicit exception handling when calling a method:

```
1 public void simpleMethod() throws Exception {
2     // ...
3 }
```

The *throw* keyword allows us to throw an exception object to interrupt the normal flow of the program. This is most commonly used when a program fails to satisfy a given condition:

```
if (task.isTooComplicated()) {
    throw new TooComplicatedException("The task is too complicated");
}
```

Q3. How can you handle an exception?

By using a *try-catch-finally* statement:

```
try {
    // ...
} catch (ExceptionType1 ex) {
    // ...
} catch (ExceptionType2 ex) {
    // ...
} finally {
    // ...
}
```

The block of code in which an exception may occur is enclosed in a *try* block. This block is also called "protected" or "guarded" code.

If an exception occurs, the *catch* block that matches the exception being thrown is executed, if not, all *catch* blocks are ignored.

The *finally* block is always executed after the *try* block exits, whether an exception was thrown or not inside it.

Q4. How can you catch multiple exceptions?

There are three ways of handling multiple exceptions in a block of code.

The first is to use a *catch* block that can handle all exception types being thrown:

```
try {
    // ...
} catch (Exception ex) {
    // ...
}
```

You should keep in mind that the recommended practice is to use exception handlers that are as accurate as possible.

Exception handlers that are too broad can make your code more error-prone, catch exceptions that weren't anticipated, and cause unexpected behavior in your program.

The second way is implementing multiple catch blocks:

```
try {
    // ...
} catch (FileNotFoundException ex) {
    // ...
} catch (EOFException ex) {
    // ...
}
```

Note that, if the exceptions have an inheritance relationship; the child type must come first and the parent type later. If we fail to do this, it will result in a compilation error.

The third is to use a multi-catch block:

```
try {
    // ...
} catch (FileNotFoundException | EOFException ex) {
    // ...
}
```

This feature, first introduced in Java 7; reduces code duplication and makes it easier to maintain.

Q5. What is the difference between a checked and an unchecked exception?

A checked exception must be handled within a *try-catch* block or declared in a *throws* clause; whereas an unchecked exception is not required to be handled nor declared.

Checked and unchecked exceptions are also known as compile-time and runtime exceptions respectively.

All exceptions are checked exceptions, except those indicated by *Error*, *RuntimeException*, and their subclasses.

Q6. What is the difference between an exception and error?

An exception is an event that represents a condition from which is possible to recover, whereas error represents an external situation usually impossible to recover from.

All errors thrown by the JVM are instances of *Error* or one of its subclasses, the more common ones include but are not limited to:

- *OutOfMemoryError* – thrown when the JVM cannot allocate more objects because it is out memory, and the garbage collector was unable to make more available

- *StackOverflowError* – occurs when the stack space for a thread has run out, typically because an application recurses too deeply
- *ExceptionInInitializerError* – signals that an unexpected exception occurred during the evaluation of a static initializer
- *NoClassDefFoundError* – is thrown when the classloader tries to load the definition of a class and couldn't find it, usually because the required *class* files were not found in the classpath
- *UnsupportedClassVersionError* – occurs when the JVM attempts to read a *class* file and determines that the version in the file is not supported, normally because the file was generated with a newer version of Java

Although an error can be handled with a *try* statement, this is not a recommended practice since there is no guarantee that the program will be able to do anything reliably after the error was thrown.

Q7. What exception will be thrown executing the following code block?

```
1 Integer[][] ints = { { 1, 2, 3 }, { null }, { 7, 8, 9 } };
2 System.out.println("value = " + ints[1][1].intValue());
```

It throws an *ArrayIndexOutOfBoundsException* since we're trying to access a position greater than the length of the array.

Q8. What is exception chaining?

Occurs when an exception is thrown in response to another exception. This allows us to discover the complete history of our raised problem:

```
try {
    task.readConfigFile();
} catch (FileNotFoundException ex) {
    throw new TaskException("Could not perform task", ex);
}
```

Q9. What is a stacktrace and how does it relate to an exception?

A stack trace provides the names of the classes and methods that were called, from the start of the application to the point an exception occurred.

It's a very useful debugging tool since it enables us to determine exactly where the exception was thrown in the application and the original causes that led to it.

Q10. Why would you want to subclass an exception?

If the exception type isn't represented by those that already exist in the Java platform, or if you need to provide more information to client code to treat it in a more precise manner, then you should create a custom exception.

Deciding whether a custom exception should be checked or unchecked depends entirely on the business case. However, as a rule of thumb; if the code using your exception can be expected to recover from it, then create a checked exception otherwise make it unchecked.

Also, you should inherit from the most specific *Exception* subclass that closely relates to the one you want to throw. If there is no such class, then choose *Exception* as the parent.

Q11. What are some advantages of exceptions?

Traditional error detection and handling techniques often lead to spaghetti code hard to maintain and difficult to read. However, exceptions enable us to separate the core logic of our application from the details of what to do when something unexpected happens.

Also, since the JVM searches backward through the call stack to find any methods interested in handling a particular exception; we gain the ability to propagate an error up in the call stack without writing additional code.

Also, because all exceptions thrown in a program are objects, they can be grouped or categorized based on its class hierarchy. This allows us to catch a group exceptions in a single exception handler by specifying the exception's superclass in the *catch* block.

Q12. Can you throw any exception inside a lambda expression's body?

When using a standard functional interface already provided by Java, you can only throw unchecked exceptions because standard functional interfaces do not have a "throws" clause in method signatures:

```
List<Integer> integers = Arrays.asList(3, 9, 7, 0, 10, 20);
integers.forEach(i -> {
    if (i == 0) {
        throw new IllegalArgumentException("Zero not allowed");
    }
    System.out.println(Math.PI / i);
});
```

However, if you are using a custom functional interface, throwing checked exceptions is possible:

```
@FunctionalInterface
public static interface CheckedFunction<T> {
    void apply(T t) throws Exception;
}
```

```
public void processTasks(
  List<Task> taks, CheckedFunction<Task> checkedFunction) {
    for (Task task : taks) {
        try {
            checkedFunction.apply(task);
        } catch (Exception e) {
            // ...
        }
    }
}

processTasks(taskList, t -> {
    // ...
    throw new Exception("Something happened");
});
```

Q13. What are the rules we need to follow when overriding a method that throws an exception?

Several rules dictate how exceptions must be declared in the context of inheritance.

When the parent class method doesn't throw any exceptions, the child class method can't throw any checked exception, but it may throw any unchecked.

Here's an example code to demonstrate this:

```
class Parent {
    void doSomething() {
        // ...
    }
}

class Child extends Parent {
    void doSomething() throws IllegalArgumentException {
        // ...
    }
}
```

The next example will fail to compile since the overriding method throws a checked exception not declared in the overridden method:

```
class Parent {
    void doSomething() {
        // ...
    }
}

class Child extends Parent {
    void doSomething() throws IOException {
        // Compilation error
    }
}
```

When the parent class method throws one or more checked exceptions, the child class method can throw any unchecked exception; all, none or a subset of the declared checked exceptions, and even a greater number of these as long as they have the same scope or narrower.

Here's an example code that successfully follows the previous rule:

```
class Parent {
    void doSomething() throws IOException, ParseException {
        // ...
    }

    void doSomethingElse() throws IOException {
        // ...
    }
}

class Child extends Parent {
    void doSomething() throws IOException {
        // ...
    }

    void doSomethingElse() throws FileNotFoundException, EOFException {
        // ...
    }
}
```

Note that both methods respect the rule. The first throws fewer exceptions than the overridden method, and the second, even though it throws more; they're narrower in scope.

However, if we try to throw a checked exception that the parent class method doesn't declare or we throw one with a broader scope; we'll get a compilation error:

```
class Parent {
    void doSomething() throws FileNotFoundException {
        // ...
    }
}

class Child extends Parent {
    void doSomething() throws IOException {
        // Compilation error
    }
}
```

When the parent class method has a throws clause with an unchecked exception, the child class method can throw none or any number of unchecked exceptions, even though they are not related.

Here's an example that honors the rule:

```
class Parent {
    void doSomething() throws IllegalArgumentException {
        // ...
    }
}

class Child extends Parent {
    void doSomething()
      throws ArithmeticException, BufferOverflowException {
        // ...
    }
}
```

Q14. Will the following code compile?

```
void doSomething() {
    // ...
    throw new RuntimeException(new Exception("Chained Exception"));
}
```

Yes. When chaining exceptions, the compiler only cares about the first one in the chain and, because it detects an unchecked exception, we don't need to add a throws clause.

Q15. Is there any way of throwing a checked exception from a method that does not have a *throws* clause?

Yes. We can take advantage of the type erasure performed by the compiler and make it think we are throwing an unchecked exception, when, in fact; we're throwing a checked exception:

```
public <T extends Throwable> T sneakyThrow(Throwable ex) throws T {
    throw (T) ex;
}

public void methodWithoutThrows() {
    this.<RuntimeException>sneakyThrow(new Exception("Checked Exception"));
}
```

3. Conclusion

In this article, we've explored some of the questions that are likely to appear in technical interviews for Java developers, regarding exceptions. This is not an exhaustive list, and it should be treated only as the start of further research.

We, at Baeldung, wish you success in any upcoming interviews.

Next »

Java Annotations Interview Questions (+ Answers)
(https://www.baeldung.com/java-annotations-interview-questions)

« Previous

Java Flow Control Interview Questions (+ Answers)
(https://www.baeldung.com/java-flow-control-interview-questions)

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)