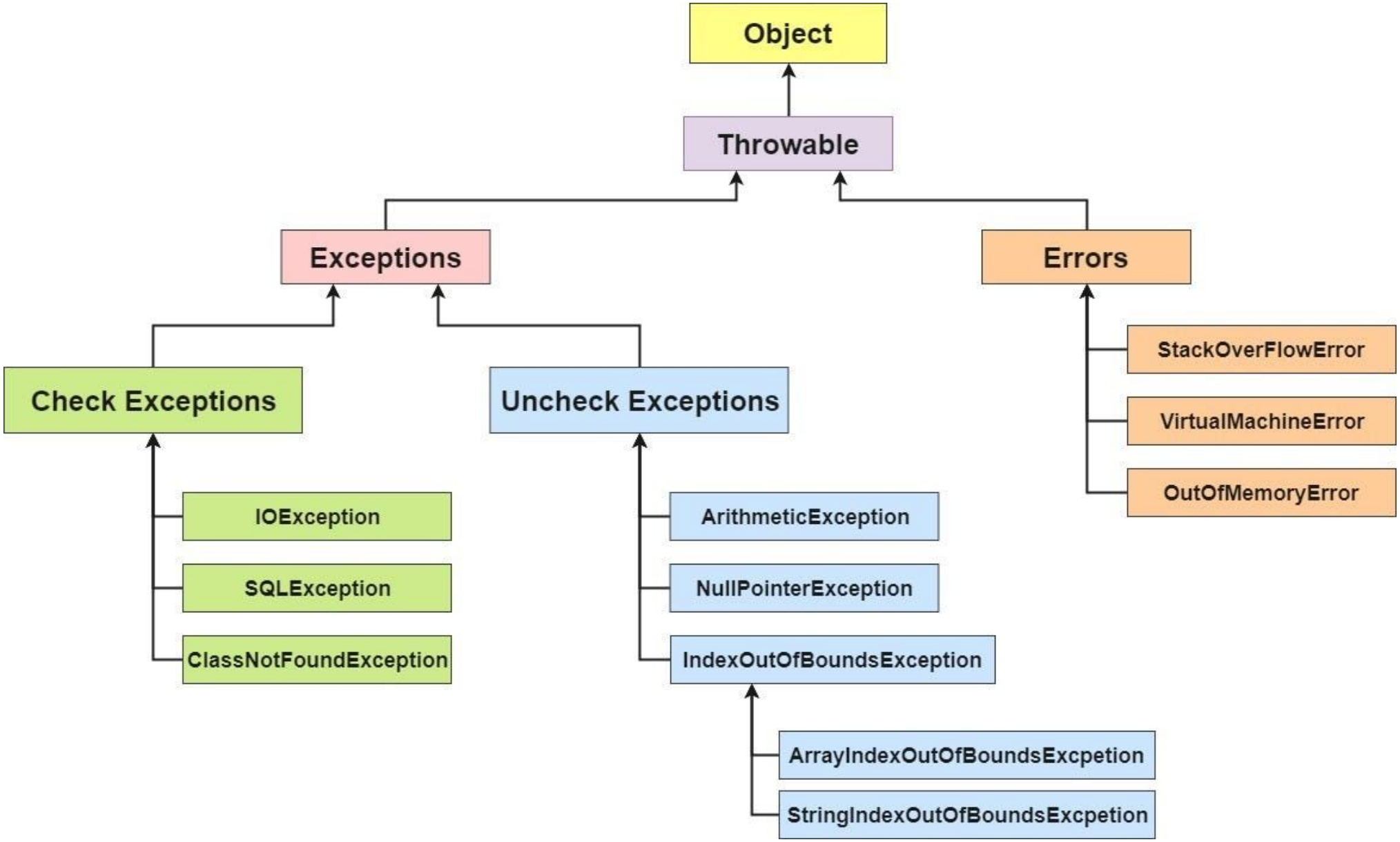
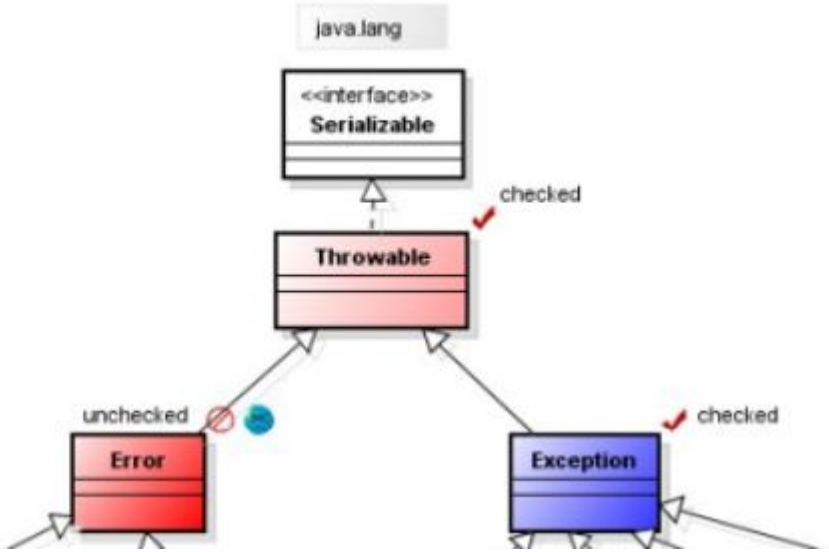
Exceptions

- exceptions


Object
Throwable
Exceptions
Errors
Check Exceptions
Uncheck Exceptions
StackOverFlowError
VirtualMachineError
OutOfMemoryError
IOException
SQLException
ClassNotFoundException
ArithmeticException
NullPointerException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsExcpetion
StringIndexOutOfBoundsExcpetion

• Exceptions

Исключения



Exceptions, on the other hand, indicate “conditions that a reasonable application might want to catch.” These could include problems that can occur at compile-time (checked exceptions) or run-time (unchecked exceptions) and can happen rather frequently in most applications - especially during development. Checked exceptions should be handled in application code, whereas unchecked exceptions don’t need to be handled explicitly.

According to the official documentation, an **Error** “indicates serious problems that a reasonable application should not try to catch.” This refers to problems that the application can not recover from - they should be dealt with by modifying application architecture or by refactoring code.

• Exceptions

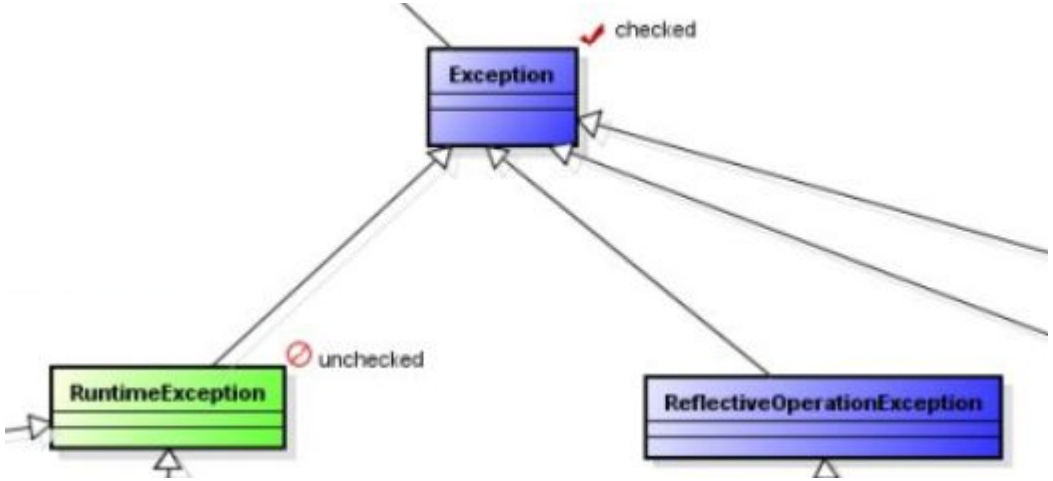
Исключения

1. **try** - Java try block is used to enclose the code that might throw an exception. It must be used within the method. If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
2. **catch** - Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
3. **finally** - You can attach a finally-clause to a try-catch block. The code inside the finally clause will always be executed, even if an exception is thrown from within the try or catch block. If your code has a return statement inside the try or catch block, the code inside the finally-block will get executed before returning from the method.
4. **throw** - If a method needs to be able to throw an exception, it has to declare the exception(s) thrown in the method signature, and then include a throw-statement in the method. When an exception is thrown the method stops execution right after the "throw" statement. Any statements following the "throw" statement are not executed. In the example above the "return numberToDivide / numberToDivideBy;" statement is not executed if a BadNumberException is thrown. The program resumes execution when the exception is caught somewhere by a "catch" block.
5. **twhros** - The throws keyword is used in a method signature and declares which exceptions can be thrown from a method. The throws keyword can be useful for propagating exceptions in the call stack and allows exceptions to not necessarily be handled within the method that declares these exceptions.

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}
```

• Exceptions

Исключения



1. **Checked** exceptions are exceptions that the Java compiler requires us to handle. We have to either declaratively throw the exception up the call stack, or we have to handle it ourselves. More on both of these in a moment. Oracle's documentation tells us to use checked exceptions when we can reasonably expect the caller of our method to be able to recover.

A couple of examples of checked exceptions are IOException and ServletException.

2. **Unchecked** exceptions are exceptions that the Java compiler does not require us to handle.

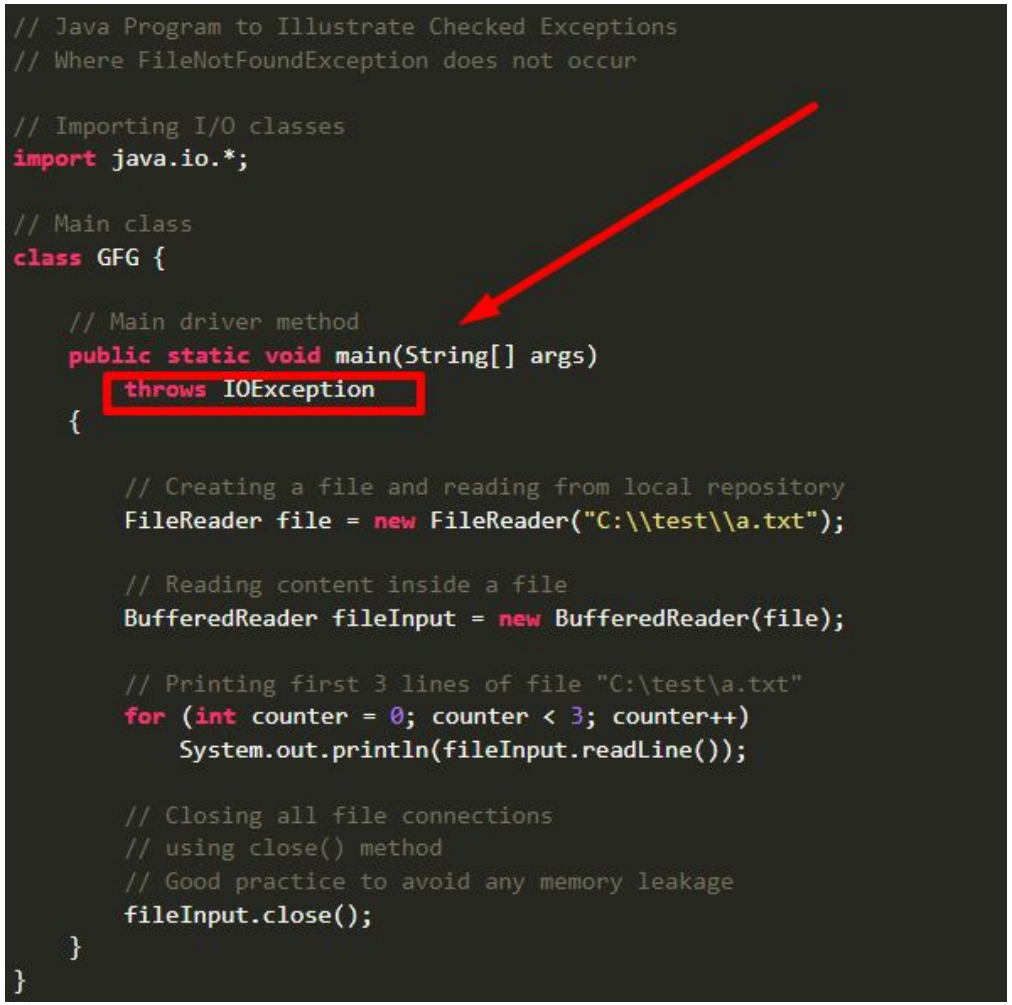
Simply put, if we create an exception that extends RuntimeException, it will be unchecked; otherwise, it will be checked.

And while this sounds convenient, Oracle's documentation tells us that there are good reasons for both concepts, like differentiating between a situational error (checked) and a usage error (unchecked).

Some examples of unchecked exceptions are NullPointerException, IllegalArgumentException, and SecurityException.

• Exceptions

Checked Exceptions in Java



```java
// Java Program to Illustrate Checked Exceptions
// Where FileNotFoundException does not occur

// Importing I/O classes
import java.io.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
        throws IOException
    {

        // Creating a file and reading from local repository
        FileReader file = new FileReader("C:\\test\\a.txt");

        // Reading content inside a file
        BufferedReader fileInput = new BufferedReader(file);

        // Printing first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        // Closing all file connections
        // using close() method
        // Good practice to avoid any memory leakage
        fileInput.close();
    }
}
```

These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the throws keyword. In checked exceptions, there are two types: fully checked and partially checked exceptions. A fully checked exception is a checked exception where all its child classes are also checked, like IOException, and InterruptedException. A partially checked exception is a checked exception where some of its child classes are unchecked, like an Exception.

For example, consider the following Java program that opens the file at location “C:\test\a.txt” and prints the first three lines of it. The program doesn’t compile, because the function main() uses FileReader(), and FileReader() throws a checked exception FileNotFoundException. It also uses readLine() and close() methods, and these methods also throw checked exception IOException

- Exceptions

Unchecked Exceptions in Java

```java
// Java Program to Illustrate Un-checked Exceptions

// Main class
class GFG {

    // Main driver method
    public static void main(String args[])
    {
        // Here we are dividing by 0
        // which will not be caught at compile time
        // as there is no mistake but caught at runtime
        // because it is mathematically incorrect
        int x = 0;
        int y = 10;
        int z = y / x;
    }
}
```

These are the exceptions that are not checked at compile time. In C++, all exceptions are unchecked, so it is not forced by the compiler's to either handle or specify the exception. It is up to the programmers to be civilized and specify or catch the exceptions. In Java, exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile because *ArithmeticException* is an unchecked exception.

• Exceptions

Исключения

Method overriding, when used to handle exceptions, is quite uncertain as the compiler cannot understand which definition to use, the superclass or subclass.

There are two important points to remember while handling exceptions using method overriding.

- If the superclass method does not declare an exception, then the overriding subclass method cannot declare a checked exception, but it can declare an unchecked exception.
- If the superclass method declares an exception, then the overriding subclass method can declare the same exception, subclass exception or no exception, but it cannot declare a parent exception thrown by the superclass method.

Method overriding for exception handling involves two main cases:

- When a superclass does not declare an exception
- When a superclass declares an exception.

- Exceptions

When superclass does not declare an exception

```
class Person
{
  1 usage  1 override
  public void display()
  {
    System.out.println(" I am a Person ");
  }
}

2 usages
class Student extends Person
{
  1 usage
  @Override
  public void display() throws IOException
  {
    System.out.println(" I am a Student ");
  }
}

class Main
{
  public static void main(String[] args)
  {
    Student s1 = new Student();
    s1.display();
  }
}
```

As shown above, the superclass Person does not throw any exception while the overriding method of subclass Student throws a checked exception.This results in a compilation error as the compiler monitors checked exceptions during compile time. Hence, an overriding subclass method can not throw a checked exception when the overriding superclass method has no exceptions.

- Exceptions

When superclass does not declare an exception

```
class Person
{
  1 usage  1 override
  public void display()
  {
    System.out.println(" I am a Person ");
  }
}

2 usages
class Student extends Person
{
  1 usage
  @Override
  public void display() throws NullPointerException
  {
    System.out.println(" I am a Student ");
  }
}

class Main
{
  public static void main(String[] args)
  {
    Student s1 = new Student();
    s1.display();
  }
}
```

The NullPointerException is an unchecked exception. Although the overridden method has no exceptions, the overriding subclass method can throw an unchecked exception and run successfully.

- Exceptions

When the superclass declares an exception

```
class Person
{
  1 usage   1 override
  public void display() throws RuntimeException
  {
    System.out.println(" I am a Person ");
  }
}

2 usages
class Student extends Person
{
  1 usage
  @Override
  public void display() throws Exception
  {
    System.out.println(" I am a Student ");
  }
}

class Main
{
  public static void main(String[] args)
  {
    Student s1 = new Student();
    s1.display();
  }
}
```

The Exception thrown by the subclass overriding method is not a child exception of the RuntimeException, so it throws a compile error. Hence, if the overriding method's exception is not a child exception of the one thrown by the overridden superclass method, then the code does not compile successfully.

- Exceptions

When the superclass declares an exception

```
class Person
{
  1 usage  1 override
  public void display() throws RuntimeException
  {
    System.out.println(" I am a Person ");
  }
}

2 usages
class Student extends Person
{
  1 usage
  @Override
  public void display() throws ArithmeticException
  {
    System.out.println(" I am a Student ");
  }
}

class Main
{
  public static void main(String[] args)
  {
    Student s1 = new Student();
    s1.display();
  }
}
```

ArithmeticException is a child exception of RuntimeException, which is thrown by the overridden superclass method. Hence, the compiler does not give any error, and the code executes successfully.

- Exceptions

When the superclass declares an exception

```
class Person
{
  1 usage  1 override
  public void display() throws IOException
  {
    System.out.println(" I am a Person ");
  }
}

2 usages
class Student extends Person
{
  1 usage
  @Override
  public void display()
  {
    System.out.println(" I am a Student ");
  }
}

class Main
{
  public static void main(String[] args)
  {
    Student s1 = new Student();
    s1.display();
  }
}
```

The superclass method throws an IOException, while the subclass overriding method does not throw any exception. Similar to the previous case, we get no error on compilation

- Exceptions

Custom Checked Exception

```
public class IncorrectFileNameException extends Exception {

    1 usage
    private static String yourErrorMessage;
    no usages
    public IncorrectFileNameException() {
        super(yourErrorMessage);
    }
}
```

```
public class IncorrectFileNameException extends Exception {
    no usages
    public IncorrectFileNameException(String yourErrorMessage) {
        super(yourErrorMessage);
    }
}
```

- Exceptions

Custom Unchecked Exception

```
public class IncorrectFileExtensionException
    extends RuntimeException {
  no usages
  public IncorrectFileExtensionException(String errorMessage, Throwable err) {
    super(errorMessage, err);
  }
}
```

Questions:

1. Exception Definition: Provide the definition of the term "exception."
2. Hierarchy of Exceptions: What is the hierarchy of exceptions?
3. Handling JVM Errors: Is it possible/necessary to handle JVM errors?
4. Methods of Exception Handling: What are the existing methods for handling exceptions?
5. Meaning of the "throws" Keyword: What does the "throws" keyword signify?
6. Characteristic of the "finally" Block: What is the peculiarity of the "finally" block? Is it always executed?
7. Absence of Catch Blocks: Can there be no "catch" blocks when catching exceptions?
8. Scenario without Finally Block Execution: Could you come up with a situation where the "finally" block will not be executed?
9. Catch Block Handling Multiple Exceptions: Can one catch block handle multiple exceptions (from the same and different inheritance branches)?
10. Checked and Unchecked Exceptions: What do you know about checked and unchecked exceptions?
11. Feature of RuntimeException: What is the peculiarity of RuntimeException?
12. Writing Custom Exceptions: How to write a custom (user-defined) exception? What motives guide the choice between checked and unchecked exceptions?
13. Operator for Throwing Exceptions: Which operator allows for forcefully throwing an exception?
14. Additional Conditions for a Method Throwing an Exception: Are there additional conditions for a method that potentially throws an exception?
15. Exception Thrown by the main Method: Can the main method throw an exception externally, and if yes, where will the handling of this exception occur?
16. Return Statement in Catch and Finally Blocks: If a return statement is present in both the catch and finally blocks, which one takes precedence?
17. Knowledge of OutOfMemoryError: What do you know about OutOfMemoryError?
18. Understanding SQLException: What do you know about SQLException? Is it checked or unchecked, and why?
19. Definition of Error: What is an Error? In what case is Error used? Provide an example of an Error.
20. Java Construction for Exception Handling: What construction is used in Java for exception handling?
21. Try-Finally Block Scenario: Suppose there is a try-finally block. An exception occurs in the try block, and execution moves to the finally block. An exception also occurs in the finally block. Which of the two exceptions will "fall out" of the try-finally block? What happens to the second exception?
22. Method with Potential IOException and FileNotFoundException: Suppose there is a method that can throw IOException and FileNotFoundException. In what sequence should catch blocks be arranged? How many catch blocks will be executed?

Литература

- ru https://habr.com/ru/company/golovachcourses/blog/223821/
- ru https://habr.com/ru/company/golovachcourses/blog/225585/
- ru http://www.javable.com/tutorials/fesunov/lesson10/
- ru https://javastudy.ru/interview/exceptions/
- en https://www.baeldung.com/java-exceptions
- en https://www.tutorialspoint.com/java/java_exceptions.htm
- en https://rollbar.com/blog/java-exceptions-hierarchy-explained/
- en https://www.geeksforgeeks.org/exception-handling-with-method-overriding-in-java/
- en https://www.baeldung.com/java-new-custom-exception