Java Exceptions

Exceptions are the errors detected during the program **execution** (at runtime), whereas syntax errors are those detected during **compiling** the program into byte-code. An exception interrupts the normal execution of a program.

Why exceptions?

Imagine we need to write a method for loading a file in memory:

First approach (ignore error checking)

- Short
- Readable
- **Not reusable**
- **Not dependable**

```
void loadFile()  {
  open file;
  determine file size;
  allocate memory;
  read file into memory;
  close file;
}
```

Second approach (error checking with magic numbers)

- **Long**
- **Obscure**
- Reusable
- Dependable

```
void loadFile()  {
  open file;
  if (operationFailed)
    return -1;

  determine file size;
  if (operationFailed)
    return -2;

  allocate memory;
  if (operationFailed)
    return -3;

  read file into memory;
  if (operationFailed)
    return -4;

  close file;
  if (operationFailed)
    return -5;
}
```

Third approach (error checking with exceptions)

- Reusable
- Dependable
- Readable

```
void loadFile() {
  try {
      open file;
      determine file size;
      allocate memory;
      read file into memory;
      close file;
  } catch (fileOpenFailed) {
      doSomething;
  } catch(determineSizeFailed) {
    doSomething;
  } catch (memoryAllocationFailed) {
      doSomething;
  } catch (readFailed) {
      doSomething;
  } catch (fileCloseFailed) {
      doSomething;
  }
}
```

Exceptions examples

ArithmeticException

Suppose you are writing a program that reads two integers from the standard input and then outputs the result of the integer division of the first number by the second one. Look at the code below.

```
import java.util.Scanner;

public class ArithmeticExceptionDemo {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int a = scanner.nextInt();
        int b = scanner.nextInt();

        System.out.println(a / b); // an exception is possible here
        System.out.println("finished");
    }
}
```

If the second number is `0`, the program throws an exception because of the division by zero.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at ArithmeticExceptionDemo.main(ArithmeticExceptionDemo.java:11)
```

As you can see, the program fails with an `ArithmeticException`, and the `"finished"` string is not printed at all. All the code **before** the exception is executed properly, and everything **after** is not.

The displayed message shows the cause of the exception (`"/ by zero"`), the file and the line number where it has occurred (`ArithmeticExceptionDemo.java:11`). The provided information is useful for developers, but it is not very meaningful for the end-users of the program.

To avoid the exception, we can check the value before the division, and, if the value is zero, print a message. Here is another version of the program. If the second number is zero, the program will print the "**Division by zero!**" string.

```
import java.util.Scanner;

public class ArithmeticExceptionDemo {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int a = scanner.nextInt();
        int b = scanner.nextInt();

        if (b == 0) {
            System.out.println("Division by zero!");
        } else {
            System.out.println(a / b);
        }
        System.out.println("finished");
    }
}
```

NumberFormatException

Another situation to consider is when you are trying to convert a string into an integer number. If the string has an unsuitable format, the code will throw an exception.

The following program reads a line from the standard input and then outputs the number that follows it.

```
import java.util.Scanner;

public class NumberFormatExceptionDemo {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();

        int number = Integer.parseInt(input); // an exception is po
        System.out.println(number + 1);
    }
}
```

If the input is not correct (e.g. `"121a"`), the program will fail:

```
Exception in thread "main" java.lang.NumberFormatException: For inpu
  at java.base/java.lang.NumberFormatException.forInputString(Numbe
  at java.base/java.lang.Integer.parseInt(Integer.java:652)
  at java.base/java.lang.Integer.parseInt(Integer.java:770)
  at NumberFormatExceptionDemo.main(NumberFormatExceptionDemo.java:
```

To avoid this exception, it is possible to check the input string by using a regular expression:

```java
import java.util.Scanner;

public class NumberFormatExceptionDemo {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();

        if (input.matches("\\d+")) { // it checks if the input line
            int number = Integer.parseInt(input);
            System.out.println(number + 1);
        } else {
            System.out.println("Incorrect number: " + input);
        }
    }
}
```

ArrayIndexOutOfBoundsException

This is a fairly common exception that occurs while working with arrays. It is caused by attempting to access a non-existent element of the array.

```java
int[] array = { 1, 2, 3 }; // an array of ints

int n1 = array[2];  // n1 is 3
int n2 = array[3];  // Exception
int n3 = array[-1]; // Exception
```

To avoid the `ArrayIndexOutOfBoundsException`, we may check if the given index belongs to the interval **[0, length - 1].**

```
public class NoIndexOutOfBoundsExceptions {

    public static void main(String[] args) {
        int[] hardCodedArray = { 3, 2, 4, 5, 1 };

        Scanner scanner = new Scanner(System.in);

        int index = scanner.nextInt();

        if (index < 0 || index > hardCodedArray.length - 1) {
            System.out.println("The index is out of bounds.");
        } else {
            System.out.println(hardCodedArray[index]);
        }
    }
}
```

NullPointerException (NPE)

Java provides a special value called `null` to indicate that no actual value is assigned to a reference variable. This value may cause one of the most frequent exceptions called `NullPointerException`. It occurs when a program attempts to use a variable with the `null` value. To avoid an **NPE**, the programmer must ensure that the objects are initialized before their use.

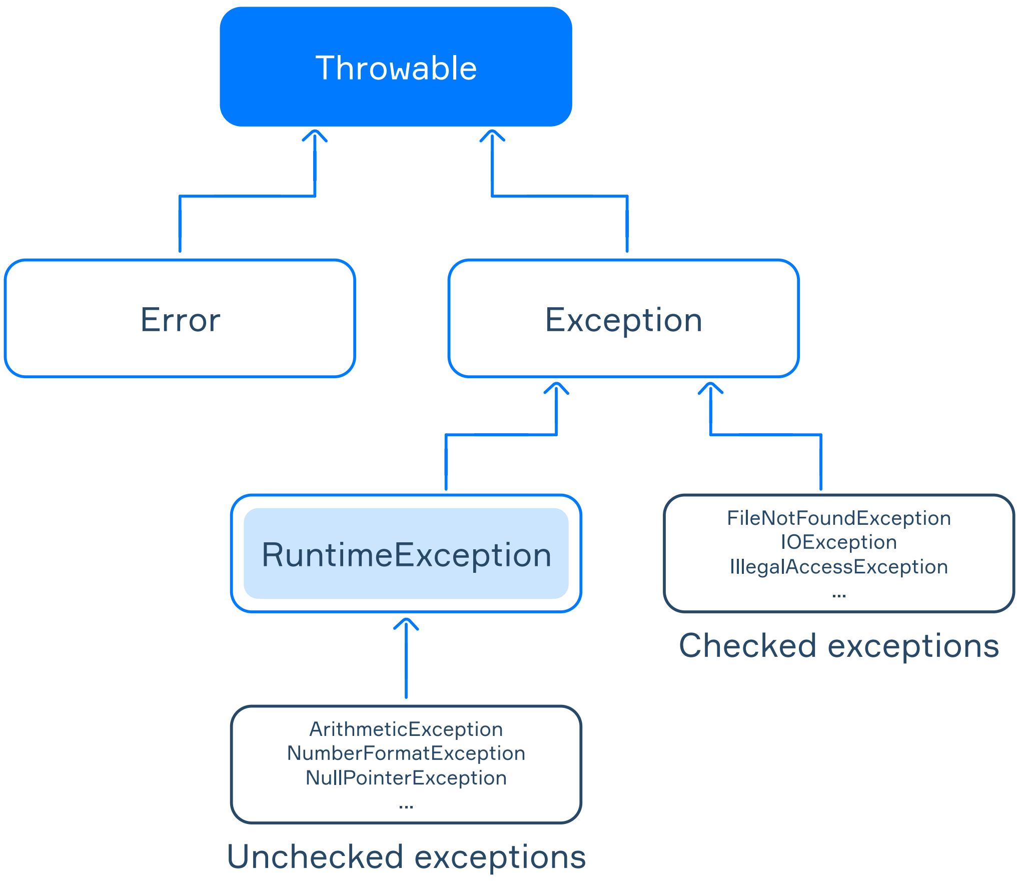
```
String someString = null; // a reference type can be null

int size = someString.length(); // NullPointerException (NPE)
```

There are several general rules on how to avoid NPEs in your programs:

- for reference types, use a conditional statement to check whether the given variable is `null` before using it;
- try to avoid assigning `null` to variables whenever possible;
- use NPE-safe features from the standard library (i.e., *Optional*).

Hierarchy of exceptions

Exceptions are considered objects of special classes organized into a class hierarchy. The following picture illustrates the simplified hierarchy of exceptions:



The base class for all exceptions is `java.lang.Throwable`. This class provides a set of common methods for all exceptions:

- `String getMessage()` returns the detailed string message of this exception object;
- `Throwable getCause()` returns the cause of this exception or `null` if the cause is nonexistent or unknown;
- `printStackTrace()` prints the stack trace on the standard error stream.

The `Throwable` class has two direct subclasses: `java.lang.Error` and `java.lang.Exception`.

- subclasses of the `Error` class represent low-level events in the JVM, for example: `OutOfMemoryError`, `StackOverflowError`;
- subclasses of the `Exception` class deal with events inside applications, such as: `RuntimeException`, `IOException`;
- the `RuntimeException` class is a rather special subclass of `Exception`. It represents so-called **unchecked** exceptions, including: `ArithmeticException`, `NumberFormatException`, `NullPointerException`.

Checked exceptions

Checked exceptions are represented by the `Exception` class, excluding the `RuntimeException` subclass. The compiler checks whether the programmer expects the occurrence of such exceptions in a program or not.

If a method throws a checked exception, this must be marked in the declaration using the special `throws` keyword. Otherwise, the program will not compile.

```
public static String readLineFromFile() throws FileNotFoundExceptio
    Scanner scanner = new Scanner(new File("file.txt")); // java.io
    return scanner.nextLine();
}
```

Here, `FileNotFoundException` is a standard checked exception. This constructor of `Scanner` declares a `FileNotFoundException` exception, because we assume that the specified file may not exist. Most importantly, there is a single line in the method that may throw an exception, so we put the `throws` keyword in the method declaration.

Unchecked exceptions

Unchecked exceptions are represented by the `RuntimeException` class and all its subclasses. The compiler does not check whether the programmer expects the occurrence of such exceptions in a program.

```
public static Long convertStringToLong(String str) {
    return Long.parseLong(str); // It may throw a NumberFormatExcept
}
```

This code always successfully compiles without the `throws` keyword in the declaration.

The `Error` class and its subclasses are also considered as unchecked exceptions.

Throwing exceptions

The throw keyword

Any object of the `Throwable` class and all its subclasses can be thrown using the **throw** statement. The general form of the statement consists of the `throw` keyword and an object to be thrown.

```
public class Main {
    public static void main(String[] args) {
        throw new RuntimeException("Something's bad.");
    }
}
```

The program stops and prints the error with the message we provided:

```
Exception in thread "main" java.lang.RuntimeException: Something's b
    at Main.main(Main.java:3)
```

It is only possible to throw an object of the `Throwable` class or a class that extends `Throwable`.

Throwing checked exceptions

Let's take a look at the following method that reads text from a file. In case the file is not found, the method throws an `IOException`:

```
public static String readTextFromFile(String path) throws IOExceptio
    // find a file by the specified path

    if (!found) {
        throw new IOException("The file " + path + " is not found")
    }

    // read and return text from the file
}
```

The `throws` keyword following the method parameters is required since an `IOException` is a checked exception. Otherwise, the code won't compile.

If a method throws two or more checked exceptions, they must be separated by a comma in the declaration:

```
public static void method() throws ExceptionType1, ExceptionType2, I
  // ...
}
```

If a method is declared as throwing an exception, it can also throw any subclass of the specified exception. In the example below, `FileNotFoundException` does not require a specific annotation because it is a subclass of `IOException`:

```
public static String readLine(String filename) throws IOException {
  BufferedReader reader = new BufferedReader(new FileReader(filenam
  return reader.readLine(); // IOException
}
```

Throwing unchecked exceptions

The `Account` class contains the method called `deposit`, that adds the specified amount to the current balance. If the amount is not positive or exceeds the limit, the method throws an `IllegalArgumentException`.

```
class Account {

    private long balance = 0;

    public void deposit(long amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Incorrect sum " + a
        }

        if (amount >= 100_000_000L) {
            throw new IllegalArgumentException("Too large amount");
        }

        balance += amount;
    }

    // ...
}
```

If a method throws an unchecked exception such as `IllegalArgumentException`, the keyword `throws` is not required in the method declaration.

When to throw an exception?

Technically, throwing an exception is a rather straightforward task. But the question is, when do you need to do this? The answer is not always obvious. **The common practice is to throw an exception when and only when the method preconditions are broken, that is when it cannot be performed under the current conditions.**

Imagine a method that parses the input string in the dd-MM-yyyy format to get a month. Here, if the string is invalid, the method throws an `InvalidArgumentException`. Another example is reading a non-existing file that will lead to a `FileNotFoundException`.

It is recommended to throw exceptions that are most relevant (specific) to the problem: it is better to throw an object of `InvalidArgumentException` than the base `Exception` class.

Handling exceptions

After an exception is thrown, the JVM attempts to find a suitable handler for it. Such a handler can be located in the same method where the exception occurred or in the calling methods (up to the main method). As soon as a suitable handler is found and executed, the exception is considered as handled and the program runs normally.

The best approach to handle an exception is to do it in a method that has sufficient information to make the correct decision based on this exception.

The try-catch statement

Here is a simple `try-catch` template for handling exceptions:

```
try {
    // code that may throw an exception
} catch (Exception e) {
    // code for handling the exception
}
```

The `try` block is used to wrap the code that may throw an exception.

The `catch` block is a handler for the specified type of exception and all of its subclasses. This block is executed when an exception of the corresponding type occurs in the `try` block.

```
public static void main(String[] args) {
  System.out.println("before the try-catch block"); // it will be pı

  try {
    System.out.println("inside the try block before an exception");
    System.out.println(2 / 0); // it throws ArithmeticException
    System.out.println("inside the try block after the exception");
  } catch (Exception e) {
    System.out.println("Division by zero!"); // it will be printed
  }

  System.out.println("after the try-catch block"); // it will be pr

}
```

The output:

```
before the try-catch block
inside the try block before an exception
Division by zero!
after the try-catch block
```

The program does not print `"inside the try block after the exception"` since the `ArithmeticException` aborted the normal flow of the execution. Instead, it executes the print statement in the `catch` block. After completion of the `catch` block, the program executes the next statement (printing `"after the try-catch block"`) without returning to the `try` block again.

As we noted earlier, the `try-catch` statement can handle both checked and unchecked exceptions. But there is a difference: checked exceptions must be wrapped with a `try-catch` block or declared to be thrown in the method, while unchecked exceptions don't have to.

Catching multiple exceptions

It is always possible to use a single handler for all types of exceptions:

```
try {
    // code that may throw exceptions
} catch (Exception e) {
    System.out.println("Something goes wrong");
}
```

Obviously, this approach does not allow us to perform different actions depending on the type of exception that has occurred. Fortunately, Java supports the use of several handlers inside the same `try` block.

```
try {
    // code that throws exceptions
} catch (IOException e) {
    // handling the IOException and its subclasses
} catch (Exception e) {
    // handling the Exception and its subclasses
}
```

Important, the `catch` block with the base class has to be written below all blocks with subclasses. In other words, the more specialized handlers (like `IOException`) must be written before the more general ones (like `Exception`). Otherwise, the code won't compile.

Since Java 7, you can also use the syntax below to have several exceptions handled in the same way:

```
try {
    // code that may throw exceptions
} catch (SQLException | IOException e) {
    // handling SQLException, IOException and their subclasses
    System.out.println(e.getMessage());
} catch (Exception e) {
    // handling any other exceptions
    System.out.println("Something goes wrong");
}
```

The finally keyword

There is another possible block called `finally`. All statements present in this block will always execute regardless of whether an exception occurs in the `try` block or not.

```
try {
    // code that may throw an exception
} catch (Exception e) {
    // exception handler
} finally {
    // code that will always be executed
}
```

The following example illustrates the order of execution of the `try-catch-finally` statement.

```
try {
    System.out.println("inside the try block");
    Integer.parseInt("101abc"); // throws a NumberFormatException
} catch (Exception e) {
    System.out.println("inside the catch block");
} finally {
    System.out.println("inside the finally block");
}

System.out.println("after the try-catch-finally block");
```

The output:

```
inside the try block
inside the catch block
inside the finally block
after the try-catch-finally block
```

If we remove the line that throws a `NumberFormatException`, the `finally` block is still executed after the `try` block.

```
inside the try block
inside the finally block
after the try-catch-finally block
```

Custom exceptions

To create a custom exception you need to extend the `Exception` (checked) or `RuntimeException` (unchecked) classes.

```
public class MalformedRequestException extends Exception {
  public MalformedRequestException() {
  }

  public MalformedRequestException(String message) {
    super(message);
  }

  public MalformedRequestException(String message, Throwable cause)
    super(message, cause);
  }

  public MalformedRequestException(Throwable cause) {
    super(cause);
  }

  public MalformedRequestException(String message, Throwable cause,
    super(message, cause, enableSuppression, writableStackTrace);
  }
}
```

In the example above, a new class of exceptions is declared. It is a checked exception because it extends the `Exception` class. The declared class has five constructors for creating instances, and they call the corresponding constructor of the base class. They can be all auto-generated in IntelliJ.

Now, we can throw an instance of the class:

```java
public static void someMethod() throws MyAppException {
    throw new MalformedRequestException("Something bad");
}
```

Best practices for custom exceptions

- Make sure that your application will benefit from creating a custom exception. Otherwise, use standard Java exceptions.
- Follow the naming convention e.g. end the class name with "Exception", for example `MyAppException`
- In general, it is advisable to use standard exceptions whenever possible for a number of reasons, such as:
    - Standard exceptions are widely known by other programmers. One can understand the type of problem just by looking at the name of the exception.
    - By opting for standard exceptions, you follow the reusability principle. It makes your code clearer and more professional.

Closing resources

When an input stream is created, the JVM notifies the OS about its intention to work with a file. If the JVM process has enough permissions and everything is fine, the OS returns a **file descriptor**. The problem is that the number of file descriptors is limited. This is a reason why it is important to notify the OS that the job is done and the file descriptor that is held can be released for further reusing.

Resource releasing works if the JVM calls the `close` method, but it is possible that the method will not be called at all. Look at the example:

```java
Reader reader = new FileReader("file.txt");
// code which may throw an exception
reader.close();
```

Suppose something goes wrong before the `close` invocation and an exception is thrown. It leads to a situation in which the method will never be called and system resources won't be released. It is possible to solve the problem by using the **try-catch-finally** construction:

```
Reader reader = null;

try {
    reader = new FileReader("file.txt");
    // code which may throw an exception
} finally {
    reader.close();
}
```

try-with-resources

A more elegant way for avoiding resource leaks is called **try-with-resources** and was introduced in Java 7.

```
try (Reader reader = new FileReader("file.txt")) {
    // some code
}
```

This construction has two parts enclosed by round and curly brackets. Round brackets contain statements that create an input stream instance. It is possible to create several objects as well. The code below is also fine:

```
try (Reader reader1 = new FileReader("file1.txt");
     Reader reader2 = new FileReader("file2.txt")) {
    // some code
}
```

The second part just contains some code for dealing with the object that was created in the first part. As you see, there are no explicit calls of the `close` method at all. It is implicitly invoked for all objects declared in the first part. The construction guarantees closing all resources in a proper way.

Surely we do our best to write error-free programs. However, it is difficult to foresee all possible problems. **The best practice is to wrap any code dealing with system resources by the try-with-resources construction.**

Closeable resources

We have dealt with a file input stream to demonstrate how try-with-resources is used. However, not only resources based on files should be released. Closing is crucial for other outside sources like web or database connections. Classes that handle them have a `close` method and therefore can be wrapped by the try-with-resources statement.

For example, let's consider `java.util.Scanner`. Earlier we used `Scanner` for reading data from the standard input, but it can read data from a file as well. `Scanner` has a `close` method for releasing outside sources.

```
try (Scanner scanner = new Scanner(new File("file.txt"))) {
    int first = scanner.nextInt();
    int second = scanner.nextInt();
    System.out.println("arguments: " + first + " " + second);
}
```

Exceptions and loops

For errors affecting a single iteration (or items of a collection!), the try-catch blocks is nested in the loop. In case of exception the execution goes to the catch block and then proceed with the next iteration.

```
while (something) {
  try {
    // potential exceptions
  } catch (Exception e) {
    //  discard iteration
  }
}
```

For errors compromising the whole loop, the loop is nested within the try block. In case of exception the execution goes to the catch block, thus exiting the loop.

```
try {
  while (something) {
    //  potential exceptions
  }
} catch (Exception e) {
  // discard whole loop
}
```

Resources

- https://www.baeldung.com/java-exceptions
- https://www.baeldung.com/java-checked-unchecked-exceptions