CHAPTER 7: EXCEPTIONS

LEARNING OBJECTIVES

- Be able to explain the Java error handling concept
- Define local exception handling
- To be able to rethrow an exception
- Be able to develop user-defined exception classes
- Be able to throw exceptions in error situations
- Be able to use the finally block

7.1 HANDLING ERRORS IN JAVA

The following examples are syntactically correct, but lead to errors, why?

```
Student  s;
s.setRegistrationNumber(4711);
```

Problem: s is `null` when calling `setRegistrationNumber`

```java
int[] intArray = { 1, 2, 3 };
intArray[3] = 4;
```

Problem: Index 3 is invalid! intArray contains elements for indices 0 to 2

Unlike "C", Java has an integrated concept for handling runtime errors:

- the intended data processing is coded in a separate block. When a runtime error occurs, a so called exception is thrown which describes the cause and the location of the error
- any type of error handling for the first block is coded in a separate block. This block is entered when an exception is thrown.

EXAMPLE: NULLPOINTEREXCEPTION

```
Student s;
s.setRegistrationNumber(4711);
```

s was declared without a value being assigend

At the time of calling `setRegisterNumber` the value of `s` is `null`. The Java runtime system therefore throws a `NullPointerException`.

EXAMPLE: ARRAYINDEXOUTOFBOUNDSEXCEPTION

```java
int[] intArray = { 1, 2, 3 };
intArray[3] = 4;
```

Array `intArray` is initialized to 1, 2, and 3 at index 0, 1 and 2.

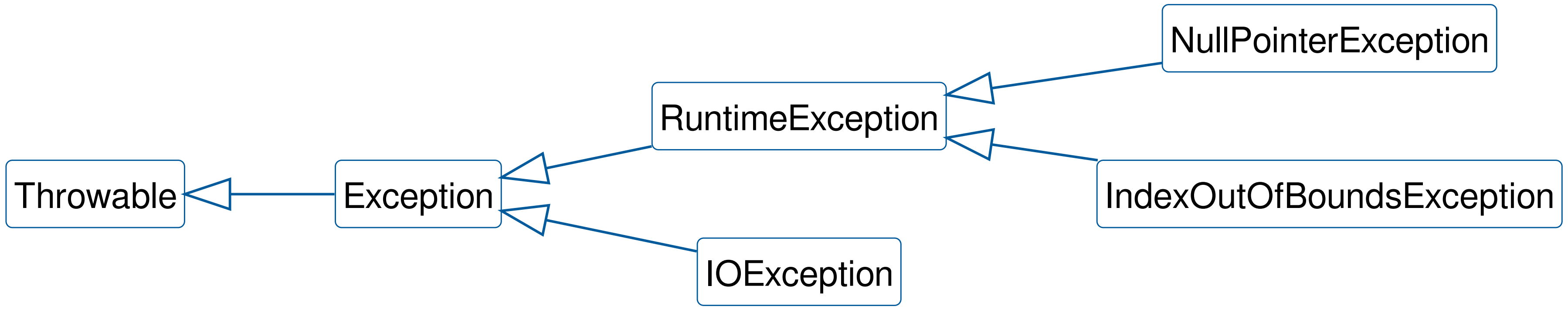
Trying to change the value at index 3 leads to an error since the index is invalid. The Java Runtime System accordingly throws an `ArrayIndexOutOfBoundsException`.

If an exception is "thrown", the execution of the current instruction is cancelled.

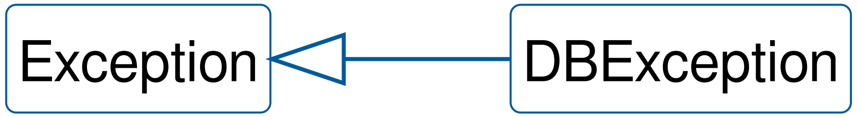
There are now two options for what happens next:

- Exceptions can be caught and handled in the method in which they occur
- or they can be passed on to the calling method.

There are special classes in Java for describing runtime errors, all of which inherit from `Throwable`. Unlike `Exception`, `Throwable` acts as a base class with important information (e.g. stack trace) for all derived classes, but it is not used directly.



The name of the exception class usually indicates the type of error. Customised, application-specific exceptions are typically defined as a subclass of Exception. Errors when accessing a database (DB) could, for example, be thrown in the form of a self-defined DBException, which then contains DB-specific error codes.



EXERCISE: ERROR HANDLING IN JAVA

Which statements are correct?

- ☐ You must use the Exception class or derivatives of it. You cannot define your own classes.
- ☐ In Java exception handling, the main process of a method should be separated from the error handling.
- ☐ An ArrayIndexOutOfBoundsException is thrown when an array is accessed using an invalid index.
- ☐ If an exception occurs in a method, it is always exited immediately.
- ☐ Every exception must either be handled locally by a try-catch block or thrown to the caller.

7.2 DEALING WITH EXCEPTIONS

Each method must specify what exceptions, if any, it throws.

```
public FileInputStream(File file) throws FileNotFoundException {
    ...
}
```

An example is the constructor of the class `java.io.FileInputStream`, which is used to read files.

- The implementation throws a FileNotFoundException if the parameter "file" does not describe an existing file
- The `FileInputStream` is a predefined class from the Java runtime library. However, the above applies analogously to self-developed methods.

(UN)CHECKED EXCEPTIONS

The Java compiler checks for each method whether it calls methods that can throw exceptions. If this is the case, it checks whether the exception is passed on (see 7.2.1) or whether the exception is handled in a local try-catch construct (see 7.2.2).

The above does not hold for objects of class `RuntimeException` and its derivatives: these are typically thrown by the JVM and are therefore not checked. Exceptions and their derivatives are referred to as *checked exceptions*, while a `RuntimeException` and its derivatives are referred to as *unchecked exceptions*.

THE "THROWS" KEYWORD

If handling of an exception does not make sense in the current method, the method can throw the error to the calling method. To do this, it is sufficient to declare the excption with a throws clause in the method header.

```
public class SomeClass {
    public void doSomething() throws SomeException {
        // ...
    }
}
```

The following is an example of reading a text file: Occurring exceptions are thrown twice (from readFile or main) and lead to a stacktrace in the console.

```java
public class TextFileReader {
    public String readFile(String filename) throws IOException {
        String result = "";
        BufferedReader input = new BufferedReader(new FileReader(
            new File(filename)));
        String line;

        while ((line = input.readLine()) != null)
            result += line + "\n";
        return result;
    }

    public static void main(String args[]) throws Exception {
        String fileText = new TextFileReader().readFile("test.txt");
        System.out.println(fileText);
    }
}
```

TRY-CATCH

To handle an exception locally, a try-catch construct can be used. The following shows an example of a try-catch statement with several catch clauses:

```
try {
    // code that might throw several exceptions
} catch (NumberFormatException nfe) {
    // error handler 1
} catch (IOExcption ioe) {
    // error handler 2
} catch (NullpointerException | ArithmeticException ex) {
    // multi-catch für two exception types
} catch (Exception e) {
    // general exception -> always at the end!
}
```

What can you do in a catch block (let the exception reference be e)?

- Ignore exception (empty statement)
- Print exception to the console, e.g. with `e.printStacktrace()`
- Throw exception with throw:
  ```
  throw e;
  ```
- Create and throw new exception:
  ```
  throw new Exception(e.getMessage());
  ```
- Handle exceptions “sensibly”, i.e. resolve the error situation

Example: Division by number read in, the comments contain important explanations

```java
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in); // Scanner on standard input
  int number = 4711;

  while (true) { // forever: read in int number n and output 4711/n
    try {
      System.out.print("Divider: ");
      int i = sc.nextInt();
      System.out.println(number + " / " + i + " = " + (number / i));
    } catch (ArithmeticException e) {
      System.out.println("Fehler: Division durch 0!");
    } catch (InputMismatchException ime) {
      // Scanner error: input can not be converted
      System.out.println("Error: No number entered!");
      String s = sc.next(); // Empty input buffer
    }
  }
}
```

EXERCISE: DEALING WITH EXCEPTIONS

Which statements are correct?

- ☐ All exceptions that can be thrown in a method must be declared in the method header.
- ☐ An unchecked exception is one whose meaning does not need to be understood.
- ☐ Unchecked exceptions do not have to be declared in the method header if they are thrown in a method.
- ☐ To handle problematic statements locally, surround them with a suitable try-catch construct.

EXERCISE: EXCEPTION VS. RUNTIMEEXCEPTION

Which statements are correct?

- ☐ Methods can specify RuntimeException via throws, but do not have to
- ☐ RuntimeException is checked and Exception is unchecked
- ☐ Exception inherits from RuntimeException
- ☐ Exception extends Throwable, RuntimeException does not

7.3 DEFINING YOUR OWN EXCEPTIONS

Java offers a number of predefined exceptions for standard errors that typically occur during program execution, such as the `NullPointerException`.

Basic information such as a stack trace and a message attribute (error message) are contained in the base class. If you want to offer additional information or methods, e.g. to provide more detailed information about the error, you can define your own exception classes by deriving them from Exception.

EXAMPLE: AN EXCEPTION WITH A DETAIL MESSAGE

```java
class MoreDetailledException extends Exception {
  private String detailMessage;
  public MoreDetailledException(String message, String detailMessage) {
    // intialize message which is defined in super class
    super(message);
    this.detailMessage = detailMessage;
  }
  public String getDetailMessage() { return detailMessage; }
}
```

The above class extends Exception with a `detailMessage` attribute and a getter for it. In the constructor, the superclass constructor must be called to store the given `message` value. The next section explains how exceptions (predefined and self-developed) can be thrown.

EXAMPLE: APPLICATION EXCEPTION

```java
public class ApplicationException extends Exception {
    private String userMessage;     // shown to the user
    private String internalMessage; // internal technical error message

    public ApplicationException(String userMessage,
        String internalMessage) {
        this.userMessage = userMessage;
        this.internalMessage = internalMessage;
    }
}
```

An `ApplicationException` can store 3 messages: `userMessage`, `internalMessage` and `message` (derived from `Exception`). However, the `message` attribute is not used here.

EXAMPLE: DATABASE EXCEPTION

```
public class DBException extends ApplicationException {
    private int dbErrorCode;

    public DBException(int errorCode,
        String userMessage, String dbMessage) {
        super(userMessage, dbMessage);
        this.dbErrorCode = errorCode;
    }
}
```

Overall, a `DBException` has the attributes inherited from `ApplicationException` and additionally a database error code.

EXERCISE: DEFINING YOUR OWN EXCEPTIONS

Which statements are correct?

- ☐ Custom exception classes can define a class hierarchy according to the Java inheritance concept.
- ☐ Custom exception classes do not have a stack trace.
- ☐ You can define your own exceptions by deriving from Exception or RuntimeException.

7.4 THROWING EXCEPTIONS

An exception object is created via the constructor `Exception(String message)` and the error message is initialised. Exceptions are thrown by creating an exception object (if one does not yet exist) and then "throwing" it with the `throw` statement:

```
public String parseInput() throws Exception {
    String input = Console.getInputAsString();
    if (input == null || input.length() == 0) {
        throw new Exception("Empty input not allowed!");
    }
    return input;
}
```

After executing a `throw` statement, the execution of the method is interrupted. If there is an enclosing matching `catch` block, the catch clause is executed, otherwise the exception is passed on to the calling method.

EXERCISE: THROWING EXCEPTIONS

Which statements are correct?

- ☐ If a throw statement is executed, you jump to the last line of the method.
- ☐ Exceptions can be thrown in your own methods using the throw statement.
- ☐ If there is a suitable catch block, the catch block will be executed after a throw statement, otherwise the generated exception is thrown to the caller of the method.

7.5 THE FINALLY BLOCK

A try-catch construct can be supplemented with a `finally` clause if certain "clean-up" work, such as closing a database connection, is to be done before a method exits. It does not matter whether the method is exited "normally" or with an exception!

```
try {
    openDatabaseConnection();
    databaseOperationThatMightFail();
} catch (SQLException e) {
    // ...
    // Log error -- fix if necessary
} finally {
    closeDatabaseConnection();
}
```

EXERCISE: THE FINALLY BLOCK

Which statements are correct?

- ☐ A finally block can be programmed at any point in a method.
- ☐ A finally block must be placed at the end of a try-catch block.
- ☐ A finally block is always executed.

7.6 TRY-WITH-RESOURCES

Resources, e.g. files, are often acquired, accessed and then released again. Example:

```java
public String loadFile(String pathname) {
    String text = "";
    File source = new File(pathname);
    Scanner scanner = null;
    try {
        scanner = new Scanner(source);
        while(scanner.hasNextLine()) {
            text += scanner.nextLine();
        }
        return text;
    } catch (IOException e1) {
        e1.printStackTrace();
    } finally {
        scanner.close();
    }
}
```

Please be aware: Often an exception resulting from closing must be handled again in finally.

An alternative for dealing with resources is the so-called Try-With-Resources expression. This allows you to create one or more resources and closes them automatically at the end. The prerequisite for this is that they implement `java.lang.AutoCloseable` - which is the case with the scanner, for example.

```
public String loadFile(String pathname) {
    String text = "";
    File source = new File(pathname);

    try(Scanner scanner = new Scanner(source)) {
        while(scanner.hasNextLine()) {
            text += scanner.nextLine();
        }
        return text;
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}
```

Here the scanner is created using Try-With-Resources and automatically closed when all instructions have been processed.

EXERCISE: TRY-WITH-RESOURCES

Which statements are correct?

- ☐ The resources used in a try-with-resources expression must implement the AutoCloseable interface.
- ☐ The resources used in a try-with-resources expression do not have to meet any prerequisites.
- ☐ A try-with-resources expression automatically closes all resources opened in it at the end.

7.7 EXERCISE

OVERVIEW

The method `DontPanic.handle` throws many errors. Add the missing error handling measures!

Unzip exceptions.zip and import the folder as "Existing Maven Project".

STEP 1: OUTPUT AFTER EXECUTION

Add a try-finally block to the method `Main.runOnce` and output the string `"DontPanic executed!"` with `System.out.println`, regardless of whether an error occurs or not.

STEP 2: ERROR HANDLING

Handle the errors `NullPointerException` and `DontPanicException` in the method `Main.runOnce` with an output of details about the error, this message should start with `"Error"` and then contain the message of the exception.

STEP 3: YOUR OWN ERROR CLASS

- Create a new class `BadInputException` which inherits from `Exception`
- Add a constructor that expects a message and a cause of type `Throwable` and call the appropriate constructor from the `Exception` class
- Handle the errors `IllegalArgumentException`, `StringIndexOutOfBoundsException` and `NumberFormatException` in the method `Main.runOnce` using the new error class `BadInputException` - which errors really need to be handled here?

STEP 4: THROWS

Correct the `throws` statement of the method `Main.runOnce` so that the errors `BadInputException`, `IOException` and `SQLException` are thrown explicitly and not the general class `Exception`.

LAST STEP: FURTHER ERROR HANDLING

- Handle all remaining errors in the method `Main.main` - so that the method can be executed without an exception.
- Remove the `throws` clause from the `Main.main` method header