Programare avansata pe obiecte - laborator 6 (231)

Butan Silvia
silvia.butan@endava.com
butan.silvia@gmail.com

Object oriented programming - lab 6

Exceptions:

An exception is an unwanted or unexpected event which occurs during the execution of a program that disrupts the normal flow of that program.

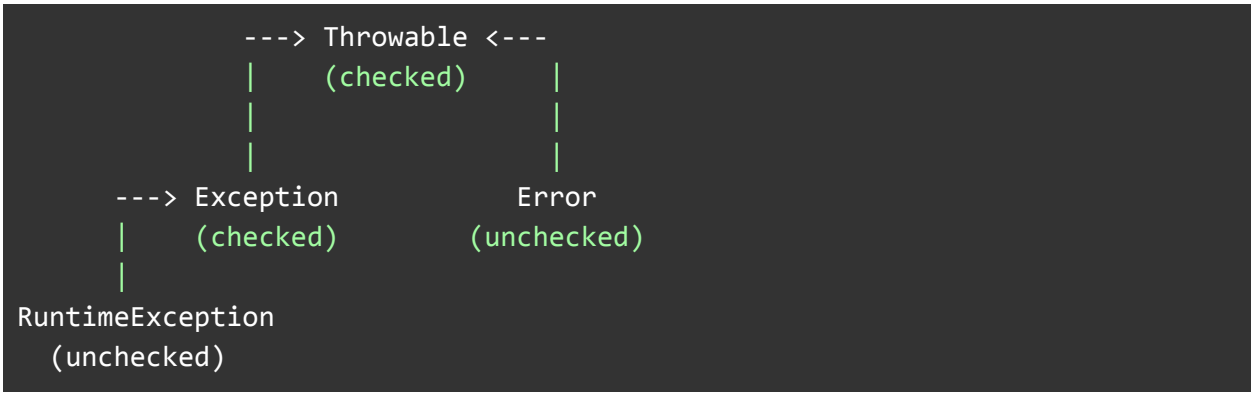
Error: indicates a serious problem that a reasonable application should not try to catch.

Exception: indicates conditions that a reasonable application might try to catch.

The Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

Exceptions are Java objects with all of them extending from Throwable. There are three main categories of exceptions:

- Checked exceptions
- Unchecked exceptions / Runtime exceptions
- Errors



➢ **Checked Exceptions:**

- 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
- examples of checked exceptions: IOException, SQLException

➢ **Unchecked Exceptions:**

- exceptions that the Java compiler does not require us to handle.
- if an exception extends **RuntimeException**, it will be unchecked;
- examples of unchecked exceptions:
  - NullPointerException - This exception means we tried to reference a null object
  - NumberFormatException - This exception means that we tried to convert a String into a number, but the string contains illegal characters
  - ArrayIndexOutOfBoundsException – this exception means that we tried to access a non-existent array index, like when trying to get index 4 from an array of length 3.

➢ **Errors:**

- represent serious and usually irrecoverable conditions
- even though they don't extend RuntimeException, they are also unchecked
- examples of errors: StackOverflowError and OutOfMemoryError.

Handling Exceptions:

There are plenty of places where things can go wrong. Some of these places are marked with exceptions, either in the signature or the Javadoc:

Eg. from **java.io.FileOutputStream**:

```
public FileOutputStream(String name) throws FileNotFoundException {
    this(name != null ? new File(name) : null, false);
}
```

When we call these methods, we must handle the exceptions:

1. **throws**

The simplest way to handle an exception is to rethrow it:

```
package com.paolabs.lab6.ex1;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;

public class FileService {

    public static void write() throws FileNotFoundException {
        FileOutputStream outputStream = new FileOutputStream("ex1.txt");
        //todo
    }
}
```

2. **try-catch**

If we want to try and handle the exception ourselves, we can use a try-catch block.

```
 public static void writeToFile()  {
        try {
           FileOutputStream outputStream = new FileOutputStream("ex1.txt");
            //todo
        } catch (FileNotFoundException e) {
            //todo handle the exception
        }
    }
```

3. **finally**

There are times when we have code that needs to execute regardless of whether an exception occurs, and this is where the **finally** keyword comes in.

Even if a FileNotFoundException is thrown up the call stack, Java will call the contents of finally before doing that.

```
 public static void writeToFileAndPrintInfo()  {
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
            //todo
        } catch (FileNotFoundException e) {
            //todo handle the exception
```

```
        } finally {
            System.out.println("This is some info");
        }
    }
```

4. **multiple catch Blocks**

Sometimes, the code can throw more than one exception, and we can have more than one catch block handle each individually.

```
 public static void writeNumberToFile() {
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
            outputStream.write(String.valueOf(53).getBytes());
        } catch (FileNotFoundException e) {
            //todo handle FileNotFoundException exception
        } catch (IOException e) {
            //todo handle IOException exception
        } finally {
            System.out.println("This is some info");
        }
    }
```

Java 7 introduced the ability to catch multiple exceptions in the same block:

```
    public static void writeTextToFile() {
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
            int number = Integer.parseInt("2f5");
            outputStream.write(String.valueOf(number).getBytes());
        } catch (IOException | NumberFormatException e) {
            //todo handle exception
        } finally {
            System.out.println("This is some info");
        }
    }
```

Throwing Exceptions:

- If we don't want to handle the exception ourselves or we want to generate our exceptions for others to handle

1. Throwing a Checked Exception

We have the following checked exception we've created ourselves:

```
package com.paolabs.lab6.ex1;

public class AccessDeniedException extends Exception {

    public AccessDeniedException(String message) {
        super(message);
    }
}
```

and we have a method that doesn't allow students to write grades:

```
    public static void writeGrades(Object o) throws AccessDeniedException {
        if (o instanceof Student) {
            throw new AccessDeniedException("Access denied for students");
        } else if (o instanceof Professor) {
            // todo write grades
        }
    }
```

2. Throwing an Unchecked Exception

If we want to do something like, say, validate input, we can use an unchecked exception:

Because IllegalArgumentException is unchecked, we don't have to mark the method.

```
public static void readFromFile(String file) {
        if (file.isBlank() || file.isEmpty()) {
            throw new IllegalArgumentException("Filename isn't valid!");
        }
        //todo
    }
```

3. **Wrapping and Rethrowing**

We can also choose to rethrow an exception we've caught:

```
public static void writeInfoToFile() throws FileNotFoundException {
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            throw e;
        }
    }
```

We can also do a wrap and rethrow:

```
   public static void writeSomethingToFile() throws AccessDeniedException {
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
            // todo
        } catch (FileNotFoundException e) {
            throw new AccessDeniedException(e.getMessage());
        }
    }
```

4. **Rethrowing unchecked exceptions:**

If the only possible exceptions that a given block of code could raise are unchecked exceptions, then we can catch and rethrow Throwable or Exception without adding them to our method signature:

```
public static void addGrade() {
        //todo add grade
        try {
            throw new NullPointerException();
        } catch (Throwable t) {
            throw t;
        }
    }
```

Inheritance:

When we mark methods with a throws keyword, it impacts how subclasses can override our method. ***Subclasses can throw fewer checked exceptions than their superclass, but not more.***

Anti-Patterns:

1. swallowing Exceptions

   eg:

```
 public static void writeTimetable() {
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
            // todo
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
```

2. using return in a finally Block

   Eg:

```
   public static boolean getGrade() {
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
            // todo
            return true;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            return false;
        }
    }
```

3. using throw in a finally Block

```
   public static boolean getGrades() throws AccessDeniedException {
```

```
        try {
            FileOutputStream outputStream = new
FileOutputStream("ex1.txt");
            return true;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            throw new AccessDeniedException("there is a problem");
        }
    }
```

Input/Output:

Java I/O (Input and Output) is used to process the input and produce the output.

The java.io package contains different ways to perform input and output (I/O) in Java. All these streams represent an input source and an output destination.

A stream can be defined as a sequence of data. There are two kinds of Streams:

- InputStream - The InputStream is used to read data from a source.
- OutputStream - The OutputStream is used for writing data to a destination.

Byte Streams:

- are used to perform input and output of 8-bit bytes
- frequently used classes are FileInputStream and FileOutputStream

Eg. FileInputStream

```
 public static void readUsingFileOutputStream() throws DocumentedException
{
        try (FileInputStream in = new FileInputStream("input.txt")) {
            int c;
            while ((c = in.read()) != -1) {
                System.out.print((char) c);
            }
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
readUsingFileOutputStream method", e);
        }
```

```
    }
```

Eg: FileOutputStream

```
package com.paolabs.lab6.ex2;

import java.io.FileOutputStream;
import java.io.IOException;

public class WriteToFileService {

    public static void writeUsingFileOutputStream(String text) throws
DocumentedException {
        try (FileOutputStream out = new FileOutputStream("output.txt")) {
            out.write(text.getBytes());
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
writeUsingFileOutputStream method", e);
        }
    }
}
```

Character Streams

- used to perform input and output for 16-bit unicode.
- frequently used classes: FileReader and FileWriter
- internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time

Eg: FileReader

```
 public static void readUsingFileReader() throws DocumentedException {
        try (FileReader fileReader = new FileReader("input.txt")){
            int content;
            while ((content = fileReader.read()) != -1) {
                System.out.print((char) content);
            }
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
```

```
readUsingFileReader method", e);
        }
    }
```

Eg: FileWriter

```
    public static void writeUsingFileWriter(String text) throws
DocumentedException {
        try (FileWriter fileWriter = new FileWriter("output.txt")) {
            fileWriter.write(text);
            fileWriter.append("-");
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
writeUsingFileWriter method", e);
        }
    }
```

File Navigation

A directory is a File which can contain a list of other files and directories. You use File object to create directories, to list down files available in a directory.

```
    public static void createDirectory(String dirName) throws
DocumentedException {
        File dir = new File(dirName);
        boolean mkdir = dir.mkdir();
        if (mkdir) {
            System.out.println("Directory: " + dirName + " created");
        } else {
            throw new DocumentedException("Something went wrong in
createDirectory method");
        }
    }
```

You can use list() method provided by File object to list down all the files and directories available in a directory. Other methods provided by File class are: getAbsolutePath(), getName(), createNewFile(), listFiles().

DataInputStream & DataOutputStream

DataInputStream class allows an application to read primitive data from the input stream..

Java application generally uses the data output stream to write data that can later be read by a data input stream.

The constructor to create an InputStream:

```
InputStream in = new DataInputStream(InputStream in);
```

Eg: DataInputStream

```
public static void readUsingDataInputStream() throws DocumentedException {
        try (DataInputStream dataInputStream = new DataInputStream(new
FileInputStream("input.txt"))) {
            while (dataInputStream.available() > 0) {
                char content = (char) dataInputStream.read();
                System.out.print(content);
            }
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
readUsingFileReader method", e);
        }
    }
```

Eg: DataOutputStream

```
 public static void writeUsingDataOutputStream(String text) throws
DocumentedException {
        try (DataOutputStream dataOutputStream = new DataOutputStream(new
FileOutputStream("output.txt"))) {
            dataOutputStream.write(text.getBytes());
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
writeUsingDataOutputStream method");
        }
    }
```

BufferedInputStream & BufferedOutputStream

BufferedInputStream

- provides transparent reading of chunks of bytes and buffering for a Java InputStream, including any subclasses of InputStream
- Reading larger chunks of bytes and buffering them can speed up IO.
- BufferedInputStream reads a larger block at a time into an internal buffer. When you read a byte from the Java BufferedInputStream you are therefore reading it from its internal buffer. When the buffer is fully read, the BufferedInputStream reads another larger block of data into the buffer. This is much faster than reading a single byte at a time from an InputStream, especially for disk access and larger data amounts.

To add buffering to an InputStream simply wrap it in a BufferedInputStream.

```
 public static void readUsingBufferedInputStream() throws
DocumentedException {
        try (BufferedInputStream bufferedInputStream = new
BufferedInputStream(
                new FileInputStream("input.txt"))) {

            while (bufferedInputStream.available() > 0) {
                int read = bufferedInputStream.read();
                System.out.print((char) read);
            }
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
readUsingBufferedInputStream method", e);
        }
    }
```

BufferedOutputStream

- is used to capture bytes written to the BufferedOutputStream in a buffer, and write the whole buffer in one batch to an underlying Java OutputStream for increased performance.
- Buffering can speed up IO, especially when writing data to disk access or network.

```
   public static void writeUsingBufferedOutputStream(String text) throws
DocumentedException {
        try (BufferedOutputStream output = new BufferedOutputStream(
```

```
                new FileOutputStream("output.txt"))) {
            output.write(text.getBytes());
        } catch (IOException e) {
            throw new DocumentedException("Something went wrong in
writeUsingBufferedOutputStream method");
        }
    }
```

BufferedReader & BufferedWriter

BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.

```
public static void readUsingBufferedWriter() {
        try (BufferedReader buffer = new BufferedReader(new
FileReader("input.txt"))) {
            String line = buffer.readLine();
            while (line != null) {
                System.out.println(line);
                line = buffer.readLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

```
public static void writeUsingBufferedWriter(String text) {
        try (BufferedWriter buffer = new BufferedWriter(new
FileWriter("output.txt"))) {
            buffer.write(text);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Applications:

Develop an application for **Home management**:

Purpose:

- A home management platform where all the family members can be added, whenever a task needs to be done (such as buying milk).
- The platform should provide functionalities such as:
    - members can create tasks and assign them to other members
    - members can mark tasks as completed

Requirements:

- Implement a custom Exception that will be used to wrap thrown exceptions when needed.
- Use I/O mechanism to load and write data to files (.txt). You should have initial data for family members.
- Define a service for I/O using Singleton pattern:
    - **Info**: https://www.geeksforgeeks.org/singleton-class-java/