Session 13: Streams and Error Handling

Softwaretechnologie: Java I

Nils Reiter
nils.reiter@uni-koeln.de

January 18, 2023

INSTITUT FÜR
DIGITAL HUMANITIES
UNIVERSITÄT ZU KÖLN

Section 1

Input and Output

Introduction

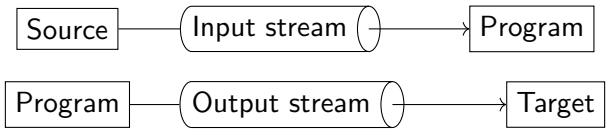
- So far: All data is defined within our programs
- Reality: Data is external to our program
  - Read from files
  - Downloaded via network
  - Recorded from microphone

Introduction

- So far: All data is defined within our programs
- Reality: Data is external to our program
  - Read from files
  - Downloaded via network
  - Recorded from microphone
- Input/Output (IO)
  - Input to the program
  - Output from the program

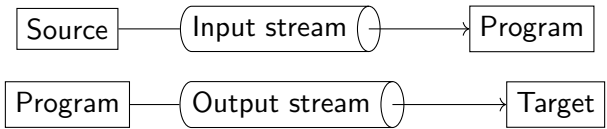
Stream

- A channel through which bytes/characters are transmitted
  - Generic computer concept (not Java-specific)



Stream

- A channel through which bytes/characters are transmitted
  - Generic computer concept (not Java-specific)



- Stream
  - Can provide a single unit only once (i.e., if something has been read from a stream, it's no longer in the stream)
  - Has an end (e.g., if the end of a file has been reached)
  - Has an order (i.e., after we have read something from a stream, we have to read the next unit)
  - Need to be closed after use

Streams in Java

- Abstract class java.io.InputStream
  - `int read()` – Reads the next byte from the stream
  - `void close()` – Closes this stream and releases system resources

Streams in Java

- Abstract class java.io.InputStream
  - `int read()` – Reads the next byte from the stream
  - `void close()` – Closes this stream and releases system resources
- Abstract class java.io.OutputStream
  - `void write(int b)` – Writes the specified byte to this output stream
  - `void flush()` – Flushes this output stream and forces any buffered output bytes to be written out
  - `void close()` – Closes this stream and releases system resources

Streams in Java

- Abstract class java.io.InputStream
  - `int read()` – Reads the next byte from the stream
  - `void close()` – Closes this stream and releases system resources
- Abstract class java.io.OutputStream
  - `void write(int b)` – Writes the specified byte to this output stream
  - `void flush()` – Flushes this output stream and forces any buffered output bytes to be written out
  - `void close()` – Closes this stream and releases system resources
- Implementations
  - java.io.FileInputStream / java.io.FileOutputStream – for reading/writing from files
  - java.io.ObjectInputStream / java.io.ObjectOutputStream – to read/write objects from/into other streams
  - ... for many other use cases

demo

What's with int and byte?

- `InputStream.read()` returns an `int`
  - "The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned."

What's with int and byte?

- `InputStream.read()` returns an `int`
  - "The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned."
- Why not the data type `byte`?
  - Because `byte` can distinguish 256 values, but we need 267 to signal the end of the stream

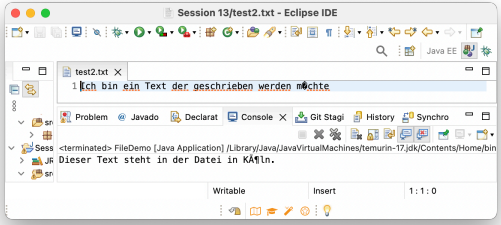
What's with int and byte?

- `InputStream.read()` returns an `int`
  - "The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned."
- Why not the data type `byte`?
  - Because `byte` can distinguish 256 values, but we need 267 to signal the end of the stream
- Why not `short`?
  - Because `int` is actually faster than `short` ...

What's with int and byte?

- `InputStream.read()` returns an `int`
  - "The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned."
- Why not the data type `byte`?
  - Because `byte` can distinguish 256 values, but we need 267 to signal the end of the stream
- Why not `short`?
  - Because `int` is actually faster than `short` ...
- The good news: `int` can be cast into a `char`, which is what we mostly really want

Why are some characters broken?



- Some characters are represented as more than one byte
  - E.g., “ö”: | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
  - It's an interpretation step to convert 1100001110110110 into an ö
- Readers are an abstraction layer on top of streams that can handle this

Readers

- java.io.InputStreamReader
  - `int read()` – Reads a single character
- java.io.OutputStreamWriter
  - `void write(int ch)` – Writes a single character

```
1  InputStream fis = new FileInputStream("path/to/file");
2  InputStreamReader isr = new InputStreamReader(fis, "UTF-8");
3  char ch = isr.read();
4  isr.close();
5
6  OutputStream os = new FileOutputStream("path/to/file");
7  OutputStreamWriter osw = new OutputStreamWriter(os, "UTF-8");
8  osw.write('a');
9  osw.flush();
10 osw.close();
```

What can go wrong with files?

- When dealing with the program-external world, there are many new error sources
- When reading a file

What can go wrong with files?

- When dealing with the program-external world, there are many new error sources
- When reading a file
  - File is not there
  - File is there, but we have no (read) access
  - File is deleted while being read

What can go wrong with files?

- When dealing with the program-external world, there are many new error sources
- When reading a file
  - File is not there
  - File is there, but we have no (read) access
  - File is deleted while being read
- When writing to a file

What can go wrong with files?

- When dealing with the program-external world, there are many new error sources
- When reading a file
  - File is not there
  - File is there, but we have no (read) access
  - File is deleted while being read
- When writing to a file
  - Directory isn't there
  - File is already there
  - Directory is there and file isn't, but we have no (write) access
  - Disk becomes full during writing

Section 2

Exception Handling

Introduction

- Exceptions can appear in various places and for many reasons
- An exception signals something unexpected that happened – usually an error of some kind

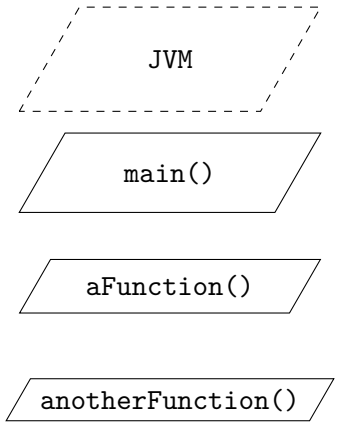
Introduction

- Exceptions can appear in various places and for many reasons
- An exception signals something unexpected that happened – usually an error of some kind
- Exceptions are thrown and can be caught
- But this happens beside to the usual program flow

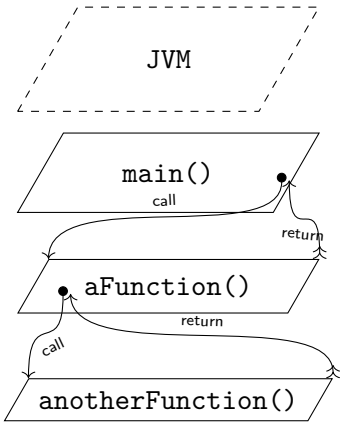
Introduction

- Exceptions can appear in various places and for many reasons
- An exception signals something unexpected that happened – usually an error of some kind
- Exceptions are thrown and can be caught
- But this happens beside to the usual program flow
- Exceptions are instances of the class `java.lang.Exception` (or one of its many subclasses)

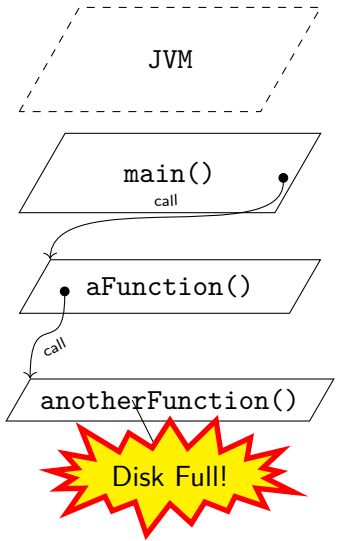
Exception Handling Visualised



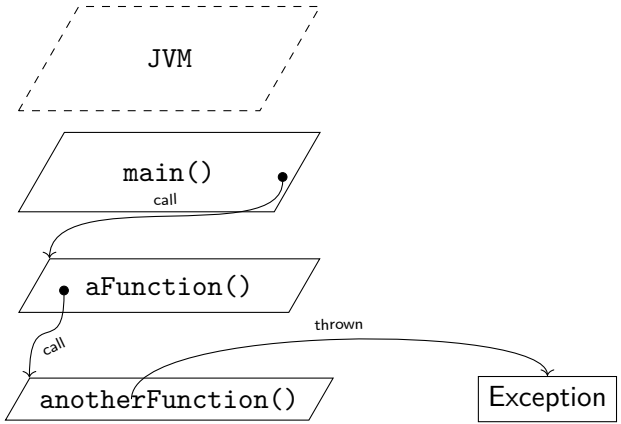
Exception Handling Visualised



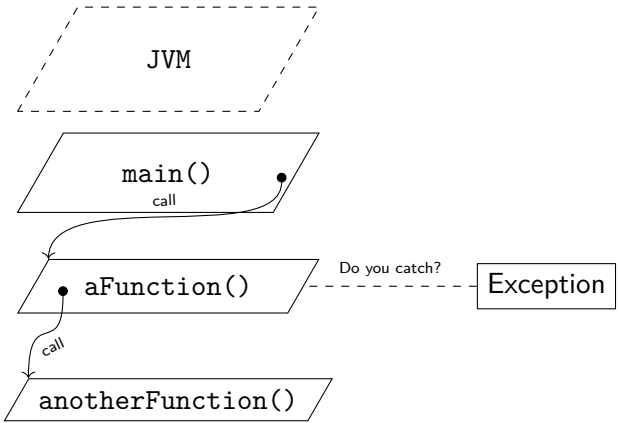
Exception Handling Visualised



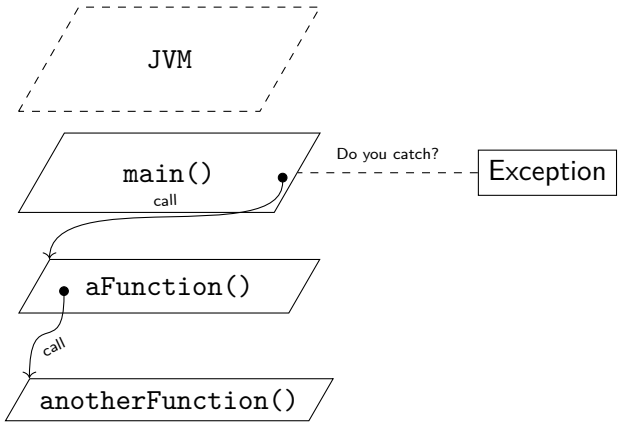
Exception Handling Visualised



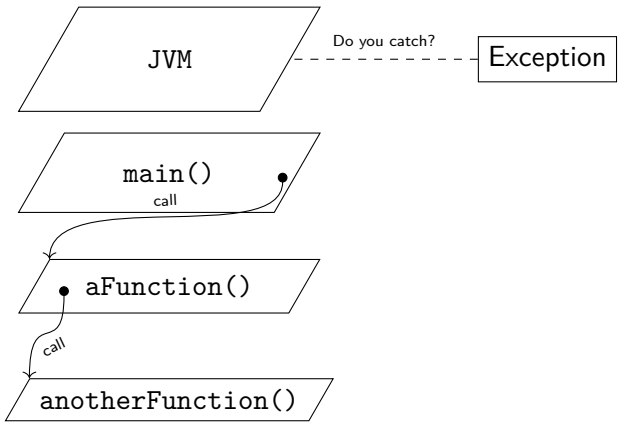
Exception Handling Visualised



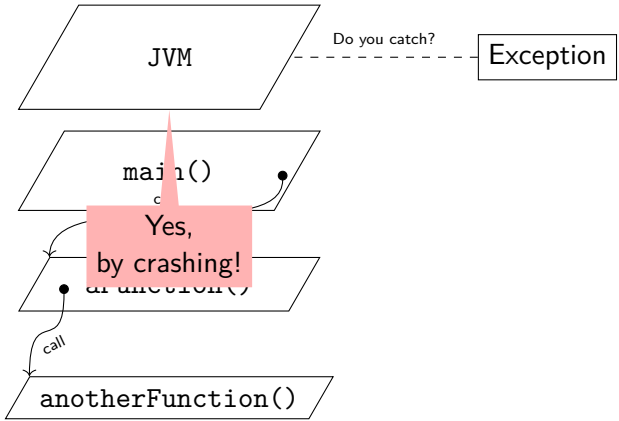
Exception Handling Visualised



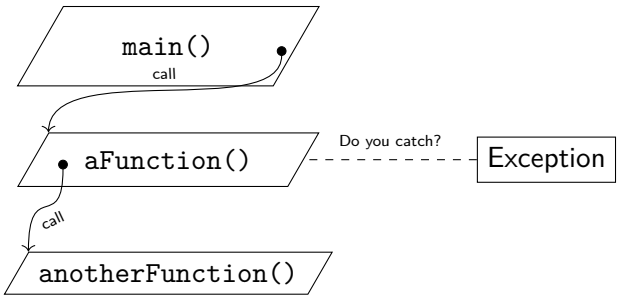
Exception Handling Visualised



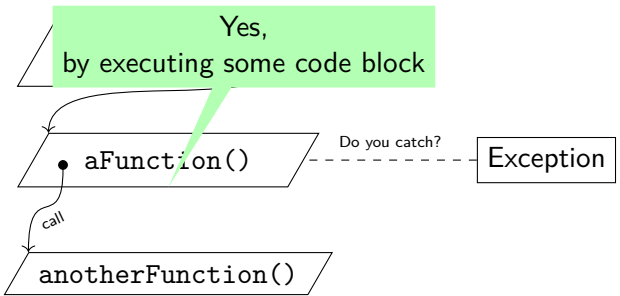
Exception Handling Visualised



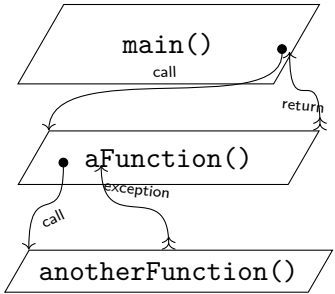
Exception Handling Visualised



Exception Handling Visualised



Exception Handling Visualised



Implementation in Java

- Three components to implement in methods:
  1. Signal that an exception can be thrown
  2. Throw an exception
  3. Catch an exception

Implementation in Java

- Three components to implement in methods:
  1. Signal that an exception can be thrown
  2. Throw an exception
  3. Catch an exception

Kinds of exceptions

- Regular exceptions are objects of the class `java.lang.Exception`
- Runtime exceptions are objects of the class `java.lang.RuntimeException`
  - The potential for a runtime exceptions does not need to be signalled
    I.e., we don't need step 1 from a above
  - A runtime exception can happen anytime, anywhere

Implementation in Java

- Three components to implement in methods:
  1. Signal that an exception can be thrown
  2. Throw an exception
  3. Catch an exception

Kinds of exceptions

- Regular exceptions are objects of the class `java.lang.Exception`
- Runtime exceptions are objects of the class `java.lang.RuntimeException`
  - The potential for a runtime exceptions does not need to be signalled
    I.e., we don't need step 1 from a above
  - A runtime exception can happen anytime, anywhere
- We can define our own exceptions by creating sub classes of java.lang.Exception

Implementation in Java

Signal that an exception can be thrown

```
1 public void someMethod() throws Exception {
2   // some code
3 }
```

- New modifier for method declaration: `throws`
- Modifier followed by a class name
- Needs to match the type of exception

Implementation in Java

Throw an exception

```
1 public void someMethod() throws Exception {
2   // some code
3   if (SOME TEST) {
4     throw new Exception("some error occurred");
5   }
6   // some code
7 }
```

- New keyword `throws`
- Followed by an object of type `java.lang.Exception`
  - Regular rules for creating an object of a class
  - In $99\,\%$ of the time, we create a new one with `new`

Implementation in Java

Catching an exception

```
1 // some code
2 try {
3   // some code
4   object.someMethod();
5   // some code
6 } catch (Exception e) {
7   // deal with the error
8   // if needed, access fields/methods of the exception with the variable e
9 }
10 // some code
```

- New statement kind: `try \{ ... \} catch (TYPE VARIABLE) \{\}`
- If line 4 throws an exception, code in line 5 is not executed – but code in lines 7 and 8
- Program continues in line 10

demo

Section 3

Exercise