7. Java Input/Output

User Input/Console Output, File Input and Output (I/O)

User Input (half the truth)

- e.g. reading a number: `int i = In.readInt();`
- Our class `In` provides various such methods.
- Some of those methods have to deal with wrong inputs: What happens with `readInt()` for the following input?

```
"spam"
```

User Input (half the truth)

```
public class Main {
  public static void main(String[] args) {
    Out.print("Number: ");
    int i = In.readInt();
    Out.print("Your number: " + i);
  }
}
```

It seems not much happens!

```
Number: spam
Your number: 0
```

User Input (the whole truth)

- e.g. reading a number using the class `Scanner`

```
import java.util.Scanner;
public class Main {
  public static void main(String[] args) {
    Out.print("Number: ");
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    Out.print("Your number: " + i);
  }
}
```

What happens for the following input?

```
"spam"
```

User Input (the whole truth)

```
Number: spam
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at Main.main(Main.java:7)
        at TestRunner.main(TestRunner.java:330)
```

Oh, we come back to this in the next chapter...

Console Output

- Until now, you knew: `Out.print("Hi")` oder `Out.println("Hi")`
- Without our `Out` class:

```
System.out.print("The answer is: ");
System.out.println(42);
System.out.println("What was the question?!");
```

This leads to the following output:

```
The answer is: 42
What was the question?!
```

So: User Input/Console Output

- Reading of input via the input stream `System.in`
- Writing of output via output stream `System.out`

Reading/Writing Files (line by line)

- Files can be read byte by byte using the class `java.io.FileReader`
- To read entire lines, we use in addition a `java.io.BufferedReader`
- Files can be written byte by byte using the class `java.io.FileWriter`
- To read entire lines, we use in addition a `java.io.BufferedWriter`

Reading Files (line by line)

```
import java.io.FileReader;
import java.io.BufferedReader;

public class Main {
  public static void main(String[] args) {
    FileReader fr = new FileReader("gedicht.txt");
    BufferedReader bufr = new BufferedReader(fr);
    String line;
    while ((line = bufr.readLine()) != null){
      System.out.println(line);
    }
  }
}
```

Reading Files (line by line)

We get the following compilation error:

```
./Main.java:6: error: unreported exception FileNotFoundException;
    must be caught or declared to be thrown
    FileReader fr = new FileReader("gedicht.txt");
                    ^
./Main.java:9: error: unreported exception IOException; must be
    caught or declared to be thrown
    while ((line = bufr.readLine()) != null){
                                ^
2 errors
```

It seems we need to understand more about the topic “Exceptions”

... therefore ...

8. Errors and Exceptions

Errors, runtime-exceptions, checked-exceptions, exception handling, special case: resources

Errors and Exceptions in Java

Errors and exceptions interrupt the normal execution of the program abruptly and represent an *unplanned event*.

Exceptions are bad, or not?

- Java allows to catch such events and deal with it (as opposed to crashing the entire program)
- Unhandled errors and exceptions are passed up through the call stack.

Errors

This glass is broken for good

Errors happen in the virtual machine of Java and are *not repairable*.

Examples

- No more memory available
- Too high call stack ($\rightarrow$ recursion)
- Missing libraries
- Bug in the virtual machine
- Hardware error

Exceptions

Exceptions are triggered by the virtual machine or the program itself and can typically be handled in order to *re-establish the normal situation*

Clean-up and pour in a new glass

Examples

- De-reference `null`
- Division by zero
- Read/write errors (on files)
- Errors in business logic

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

Example of a Runtime Exception

```
import java.util.Scanner;
class ReadTest {
    public static void main(String[] args){
        int i = readInt("Number");
    }
    private static int readInt(String prompt){
        System.out.print(prompt + ": ");
        Scanner input = new Scanner(System.in);
        return input.nextInt();
    }
}
```

Input: `Number: asdf`

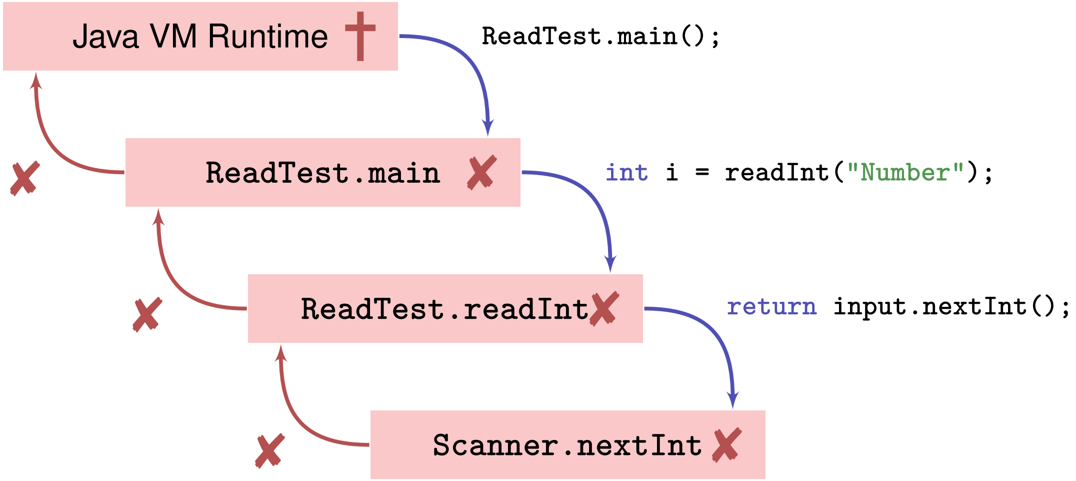
Unhandled Errors and Exceptions

The program crashes and leaves behind a *stack trace*. In there, we can see the where the program got interrupted.

```
Exception in thread "main" java.util.InputMismatchException
            [...]
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

⇒ Forensic investigation based on this information.

Exception gets Propagated through Call Stack



Unstanding Stack Traces

Output:

An unsuited input ...

```
Exception in thread "main" java.util.InputMismatchException
        ~~at java.util.Scanner.throwFor(Scanner.java:864)~~
        ~~at java.util.Scanner.next(Scanner.java:1485)~~
        ~~at java.util.Scanner.nextInt(Scanner.java:2117)~~
        ~~at java.util.Scanner.nextInt(Scanner.java:2076)~~
        at ReadTest.readInt(ReadTest.java:9)
        at ReadTest.main(ReadTest.java:4)
```

... in method `readInt` on line 9 ...

called by method `main` on line 4

Unstanding Stack Traces

```
1  import java.util.Scanner;
2  class ReadTest {
3      public static void main(String[] args){
4          int i = readInt("Number");
5      }
6      private static int readInt(String prompt){
7          System.out.print(prompt + ": ");
8          Scanner input = new Scanner(System.in);
9          return input.nextInt();
10     }
11 }
```

```
at ReadTest.readInt(ReadTest.java:9)
at ReadTest.main(ReadTest.java:4)
```

Runtime Exception: Bug in the Code?!

Where is the bug?

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    return input.nextInt();
}
```

$\uparrow$

Not guaranteed that the next input is an `int`

⇒ The scanner class provides a test for this

Runtime Exception: Bug Fix!

Check first!

```
private static int readInt(String prompt){
    System.out.print(prompt + ": ");
    Scanner input = new Scanner(System.in);
    if (input.hasNextInt()){
        return input.nextInt();
    } else {
        return 0; // or do something else ...?!
    }
}
```

First Finding: often no Exceptional Situation

Often, those “exceptional” cases aren’t that unusual, but pretty foreseeable. In those cases *no* exceptions should be used!

Kids are tipping over cups. You get used to it.

Examples

- Wrong credentials when logging in
- Empty required fields in forms
- Unavailable internet resources
- Timeouts

Second Finding: Avoid Exceptions

Problem solved.

Instead of letting a runtime exception happen, *actively prevent* such a situation to arise.

Examples

- Check user inputs early
- Use optional types
- Predict timeout situations
- Plan B for unavailable resources

Exception Types

Runtime Exceptions

- Can happen anywhere
- *Can* be handled
- Cause: bug in the code

Checked Exceptions

- Must be declared
- *Must* be handled
- Cause: Unlikely but not impossible event

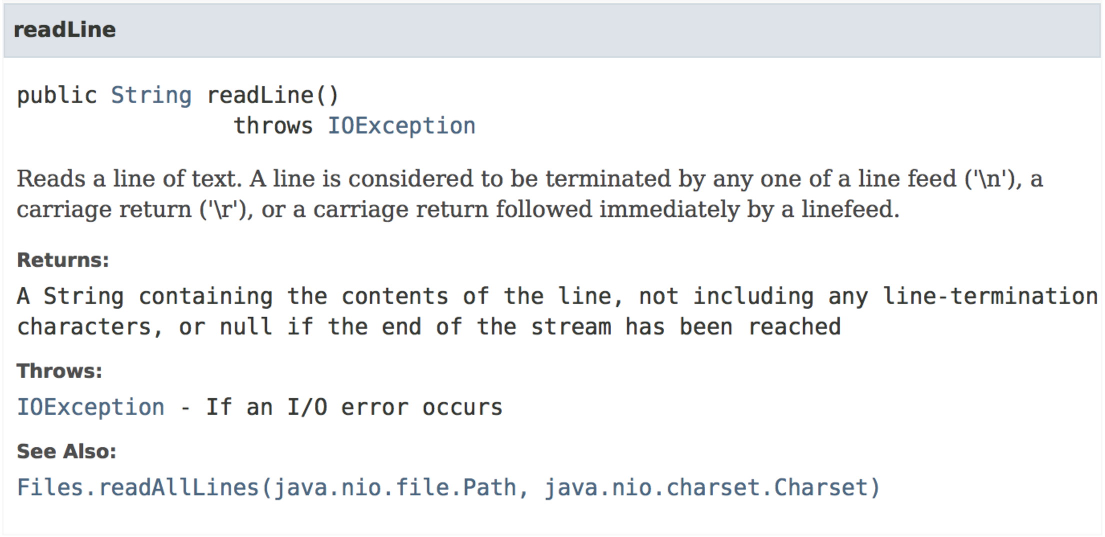
Example of a Checked Exception

```java
private static String[] readFile(String filename){
    FileReader fr = new FileReader(filename);
    BufferedReader bufr = new BufferedReader(fr);
    ...
    line = bufr.readLine();
    ...
}
```

Compiler Error:

```
./Root/Main.java:9: error: unreported exception FileNotFoundException; must be caught or declared to be
    FileReader fr = new FileReader(filename);
                    ^
./Root/Main.java:11: error: unreported exception IOException; must be caught or declared to be thrown
    String line = bufr.readLine();
                               ^
```

Quick Look into Javadoc



readLine

```
public String readLine()
                throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

See Also:

Files.readAllLines(java.nio.file.Path, java.nio.charset.Charset)

Why use Checked Exceptions?

The following situations justify checked exception:

- Fault is *unprobable but not impossibe* – and can be fixed by taking suitable measures at runtime.

The caller of a method with a declared checked exception is forced to deal with it – catch it or pass it up.

Handling Exceptions

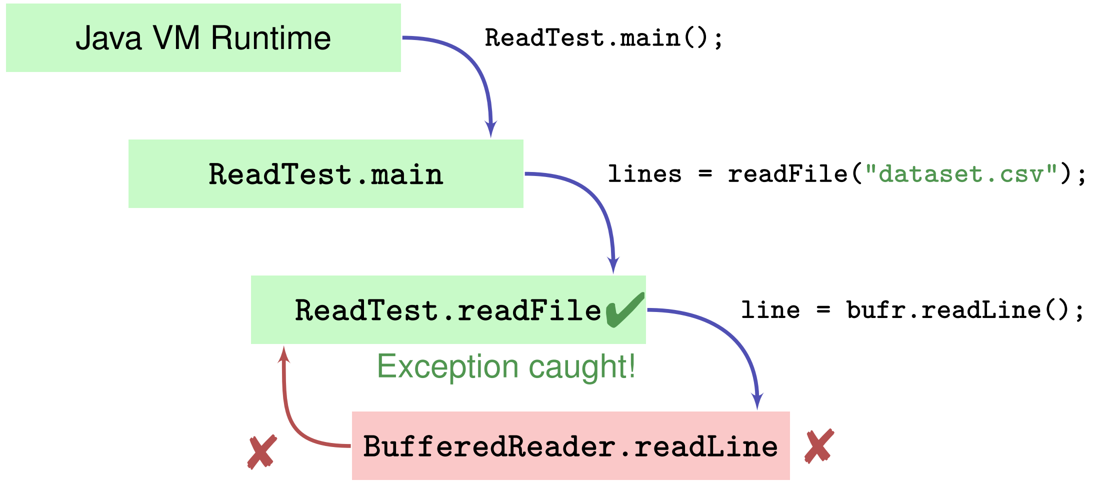
```
private static String[] readFile(String filename){
    try{
        FileReader fr = new FileReader(filename);
        BufferedReader bufr = new BufferedReader(fr);
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    } finally {
        // close resources
    }
}
```

Protected scope

Measures to re-establis the normal situation

Gets executed in any case, at the end, always!

Handling Exceptions: Stop Propagation!



Finally: Closing Resources

In Java, *resources* must be closed after use at all costs. Otherwise, memory won't get freed.

Resources:

- Files
- Data streams
- UI elements
- ...

Try-With-Resources Statement

Specific syntax to close resources *automatically*:

```
private static String[] readFile(String filename){
    try ( FileReader fr = new FileReader(filename);
      BufferedReader bufr = new BufferedReader(fr)) {
        ...
        line = bufr.readLine();
        ...
    } catch (IOException e){
        // do some recovery handling
    }
}
```

Resources get opened here

Resources get closed automatically here