

# Exception-Handling

---

André Matutat & Carsten Gips (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

# Fehlerfälle in Java

```
int div(int a, int b) {  
    return a / b;  
}
```

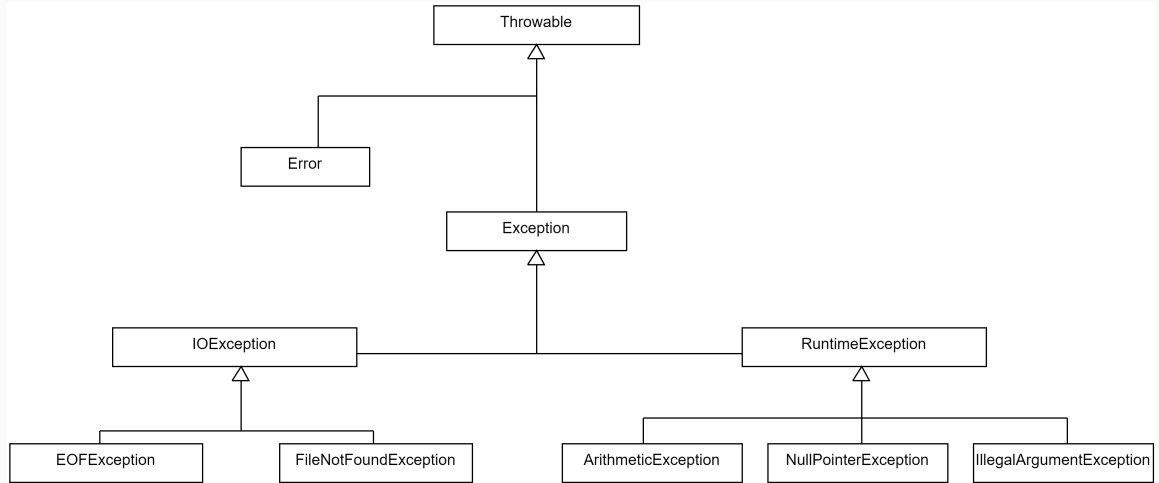
```
div(3, 0);
```

# Lösung?

```
Optional<Integer> div(int a, int b) {  
    if (b == 0) return Optional.empty();  
    return Optional.of(a / b);  
}
```

```
Optional<Integer> x = div(3, 0);  
if (x.isPresent()) {  
    // do something  
} else {  
    // do something else  
}
```

# Vererbungsstruktur *Throwable*



Hinweis: checked vs. unchecked

# Throws

```
int div(int a, int b) throws ArithmeticException {  
    return a / b;  
}
```

```
int div(int a, int b) throws IllegalArgumentException {  
    if (b == 0) throw new IllegalArgumentException("Can't divide by zero");  
    return a / b;  
}
```

Hinweis: throws und checked vs. unchecked

# Try-Catch

```
int a = getUserInput();
int b = getUserInput();

try {
    div(a, b);
} catch (IllegalArgumentException e) {
    e.printStackTrace(); // Wird im Fehlerfall aufgerufen
}

// hier geht es normal weiter
```

## \_Try\_und mehrstufiges *Catch*

```
try {  
    someMethod(a, b, c);  
} catch (IllegalArgumentException iae) {  
    iae.printStackTrace();  
} catch (FileNotFoundException | NullPointerException e) {  
    e.printStackTrace();  
}
```

Hinweis: catch und Vererbungshierarchie

# Finally

```
Scanner myScanner = new Scanner(System.in);

try {
    return 5 / myScanner.nextInt();
} catch (InputMismatchException ime) {
    ime.printStackTrace();
} finally {
    // wird immer aufgerufen
    myScanner.close();
}
```



## Try-with-Resources

```
try (Scanner myScanner = new Scanner(System.in)) {  
    return 5 / myScanner.nextInt();  
} catch (InputMismatchException ime) {  
    ime.printStackTrace();  
}
```

# Eigene Exceptions

*// Checked Exception*

```
public class MyCheckedException extends Exception {  
    public MyCheckedException(String errorMessage) {  
        super(errorMessage);  
    }  
}
```

*// Unchecked Exception*

```
public class MyUncheckedException extends RuntimeException {  
    public MyUncheckedException(String errorMessage) {  
        super(errorMessage);  
    }  
}
```

## Stilfrage: Wie viel Code im *Try*?

```
int getFirstLineAsInt(String pathToFile) {  
    FileReader fileReader = new FileReader(pathToFile);  
    BufferedReader bufferedReader = new BufferedReader(fileReader);  
    String firstLine = bufferedReader.readLine();  
  
    return Integer.parseInt(firstLine);  
}
```

Zeigen: exceptions.HowMuchTry

## Stilfrage: Wo fange ich die Exception?

```
private static void methode1(int x) throws IOException {
    JFileChooser fc = new JFileChooser();
    fc.showDialog(null, "ok");
    methode2(fc.getSelectedFile().toString(), x, x * 2);
}

private static void methode2(String path, int x, int y) throws IOException {
    FileWriter fw = new FileWriter(path);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write("X:" + x + " Y: " + y);
}

public static void main(String... args) {
    try {
        methode1(42);
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

# Stilfrage: Wann checked, wann unchecked

## “Checked” Exceptions

- Für erwartbare Fehlerfälle, deren Ursprung nicht im Programm selbst liegt
- Aufrufer kann sich von der Exception erholen

## “Unchecked” Exceptions

- Logische Programmierfehler (“Versagen” des Programmcodes)
- Aufrufer kann sich von der Exception vermutlich nicht erholen

- `Error` und `Exception`: System vs. Programm
- Checked und unchecked Exceptions: `Exception` vs. `RuntimeException`
- `try`: Versuche Code auszuführen
- `catch`: Verhalten im Fehlerfall
- `finally`: Verhalten im Erfolgs- und Fehlerfall
- `throw`: Wirft eine Exception
- `throws`: Deklariert eine Exception an Methode
- Eigene Exceptions durch Ableiten von anderen Exceptions

# LICENSE



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.