

# Funktionale und objektorientierte Programmierkonzepte



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Präsenz-Sprechstunde B



## Fehlerbehandlung [20min]

- Throwable

- Error

  - AssertionError

- Exception

- RuntimeException

- Exception werfen

- Exception weiterreichen

- Exception fangen

## Arbeitsphase [80min]

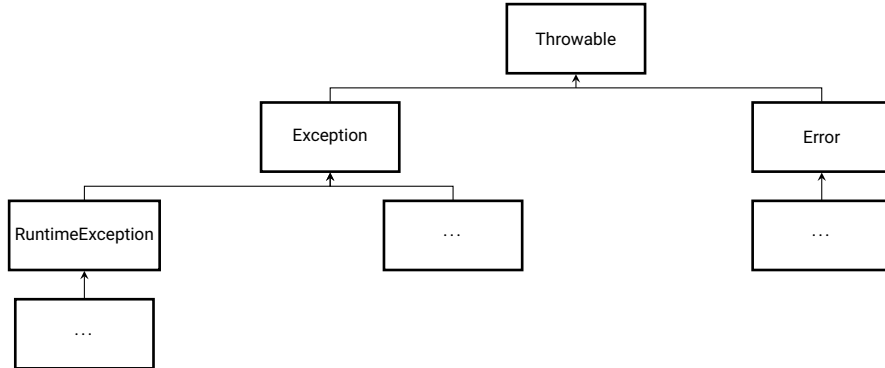


Abbildung: Java Exception Hierarchy

- Basisklasse für Exception und Error
- Bietet `try-catch` Konstrukt an, um Fehlerzustände zu behandeln
  - ▣ Funktioniert für Throwable und alle direkt oder indirekt abgeleiteten Klassen von Throwable

```
1 int a = 5, b = 0;
2 try {
3     int c = a / b;
4     System.out.println(c);
5 } catch (ArithmeticException e) {
6     System.err.println("Divisor cannot be zero!");
7 }
```

- Wird typischerweise vom Laufzeitsystem geworfen
- Fälle: Fehler welche nach menschlichen Ermessen so gewichtig ist, dass wohl keine sinnvolle Fehlerbehandlung möglich ist, sondern der Programmabbruch die beste Lösung zu sein scheint.
- Error-Klassen müssen nicht mit `try-catch`-Block gefangen werden
- Formaler Aufbau (Werfen eines Errors):  

```
throw new ErrorklasseName(Parameter);
```
- Subklassennamen enden mit Error

- Spezielle abgeleitete Error-Klasse
- Java bietet eine sehr bequeme Sonderform mit AssertionError umzugehen
- Schlüsselwort: `assert`
  - ▣ Müssen aktiviert werden!
- Formaler Aufbau:

```
1 if (condition) {  
2     throw new AssertionError(message);  
3 }  
4 // Short form  
5 assert !condition:message;
```

```
1 if (n < 0 || n % 2==1) {  
2     throw new AssertionError("Bad n!");  
3 }  
4 // Short form  
5 assert n >= 0 && n % 2==0:"Bad n!";
```

- „Checked Exception“
- Wird vom Compiler entdeckt und muss
  - ▣ behandelt werden oder
  - ▣ weitergereicht werden

- Formaler Aufbau:

```
throw new Exceptionklassenname(Parameter);
```

- Subklassennamen enden mit Exception



- FileReader Konstruktor wirft eine Exception, falls die Datei nicht existiert

```
1 File file = new File("path-to-file");
2 try {
3     BufferedReader reader = new BufferedReader(new FileReader(file));
4 } catch (FileNotFoundException e) {
5     System.err.println("File could not be found!");
6 }
```

# Wieso eigene Exceptionklassen definieren?

---

**Frage:** Ist es sinnvoll, eigene Exceptionklassen zu definieren? Welche Vorteile ergeben sich hieraus?

# Wieso eigene Exceptionklassen definieren?

**Frage:** Ist es sinnvoll, eigene Exceptionklassen zu definieren? Welche Vorteile ergeben sich hieraus?

**Antwort:** Bei eigenen Exception-Klassen können Informationen an den Konstruktor übergeben werden, was die Fehlerbehandlung erleichtern kann und zum Beispiel für eine detaillierte Ausgabe der Probleme sorgt.



- Fehler, die erst zur Laufzeit auftreten und dann zum Abbruch der Ausführung des Programms führen
- Kann nicht vom Compiler entdeckt werden
- Laufzeitsystem bricht Ablauf ab
- Call-Stack mit Zeilennummern wird in der Fehlermeldung angezeigt
  - ▣ Man sieht sofort, wo der Fehler aufgetreten ist

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         int a = 5, b = 0;  
5         int c = a / b;  
6     }  
7 }
```

Exception in thread "main" java.lang.ArithmeticException: / by zero  
at Main.main(Main.java:5)

- Schlüsselwort: `throw`
- Leitet im Methodenkörper eine Exception ein bzw. eine Exception wird geworfen
- Eine Exception ist ganz konkret ein Objekt der gewählten Exception-Klasse und muss daher mit `new` erzeugt werden, bevor sie geworfen werden kann.
- `throw`-Klausel beendet die Ausführung einer Methode
  - ▣ Im Unterschied zu `return` wird kein Wert zurückgeliefert, sondern das in der Anweisung erzeugte Exception-Objekt wird geworfen.

```
1 public class Person {  
2  
3     final String name;  
4  
5     public Person(String name) {  
6         if (name==null) {  
7             throw new NullPointerException("Name cannot be null!");  
8         }  
9         this.name = name;  
10    }  
11 }
```

```
1 if (name==null) {  
2     NullPointerException e = new NullPointerException("Name cannot be null!");  
3     throw e;  
4 }
```



- Schlüsselwort: `throws`
- Im Methodenrumpf: Gibt an, dass diese Methode potenziell eine Exception werfen kann
- Falls mehrere Exceptions weitergereicht werden, können diese mit einem Komma getrennt werden
- Eine Methode muss eine Exception nicht unbedingt fangen, sondern kann diese dem Aufrufer weiterreichen
- Formaler Aufbau:  
`Modifiers Rückgabetyp Methodenname(Parameter)throws Exceptionname, ...`
- Exceptionname in der `throws`-Klausel muss nicht mit der geworfenen Exception übereinstimmen
  - ▣ Geworfene Exception kann eine direkt oder indirekt abgeleitete Exception von der `throws`-Klausel sein

```
1 public static int factorial(int n) throws IllegalArgumentException {  
2     if (n < 0) {  
3         throw new IllegalArgumentException("n cannot be negative!");  
4     }  
5     int sum = 1;  
6     for (int i = 1; i <= n; i++) {  
7         sum *= i;  
8     }  
9     return sum;  
10 }
```



- Schlüsselwort: `try` und `catch`
- Exception behandeln
- `try`-Block: Codeauschnitt, dass potenziell eine Exception werfen kann
- `catch`-Block: Name der Exception, welche gefangen werden soll
  - ▣ Mehrere `catch`-Blöcke werden untereinander mit ihren eigenen Block eingeführt
  - ▣ Ein Block kann mehrere Exception fangen und werden mit `|` (oder) getrennt
  - ▣ Wenn eine Exception geworfen wird, dann wird die Ausführung beendet und ein passender `catch`-Block wird ausgewertet.

- Falls es keine passenden `catch`-Blöcke gibt, dann werden alle Blöcke ignoriert
- Auswertung ist von oben nach unten, d.h. wird eine Exception im ersten `catch`-Block behandelt, so werden die Operationen in dieser ausgeführt und die anderen `catch`-Blöcke werden nicht mehr beachtet.

```
1 try {  
2     // Code that can potentially throw an exception  
3 } catch (Exceptionname1 e) {  
4     // Execution of operations if an exception 1 is caught  
5 } catch (Exceptionname2 e) {  
6     // Execution of operations if an exception 2 is caught  
7 }
```

```
1 int[] a = {1, 2, 3, 4, 5};
2 try {
3     System.out.println(a[6]);
4     for (int n : a) {
5         System.out.println(factorial(n));
6     }
7 } catch (IndexOutOfBoundsException | NullPointerException e) {
8     System.err.println("Invalid task");
9 } catch (IllegalArgumentException e) {
10     throw e;
11 }
```

## try-catch: Exception vs. RuntimeException



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

**Frage:** Alle Exceptions, die nicht von der Klasse `RuntimeException` direkt oder indirekt abgeleitet sind, müssen spätestens in der `main`-Methode behandelt werden. Wieso müssen `RuntimeException` nicht unbedingt behandelt werden?



**Frage:** Alle Exceptions, die nicht von der Klasse `RuntimeException` direkt oder indirekt abgeleitet sind, müssen spätestens in der `main`-Methode behandelt werden. Wieso müssen `RuntimeException` nicht unbedingt behandelt werden?

**Antwort:** Müsste man all diese Exception fangen, so wäre der komplette Code mit `try-catch` Blöcken versehen werden und der Code wäre nicht mehr zu lesen. Deshalb müssen sie nicht gefangen oder geworfen werden und werden automatisch geworfen.

```
1 Animal animal = new Dog();  
2 Dog dog = (Dog) animal;  
3 int a = 10;  
4 int b = 42;  
5 int c = a / b;
```



```
1 Animal animal = new Dog();
2 try {
3     Dog dog = (Dog) animal;
4 } catch (ClassCastException e) {
5     // ...
6 }
7 int a = 10;
8 int b = 42;
9 try {
10
11     int c = a / b;
12 } catch (ArithmeticException e) {
13     // ..
14 }
```

## Selbstständiges Arbeiten