



Neue Themen für die 808

Februar 2025

Plan für die Woche

Montag

- Selbsteinschätzung
- Heap und Stack
- Garbage Collector und Object Lifecycle

Dienstag

- Exceptions

Mittwoch

- StringBuilder vs. String

Donnerstag

- Calender API

Freitag

- Wiederholung ArrayLists
- Lambdas

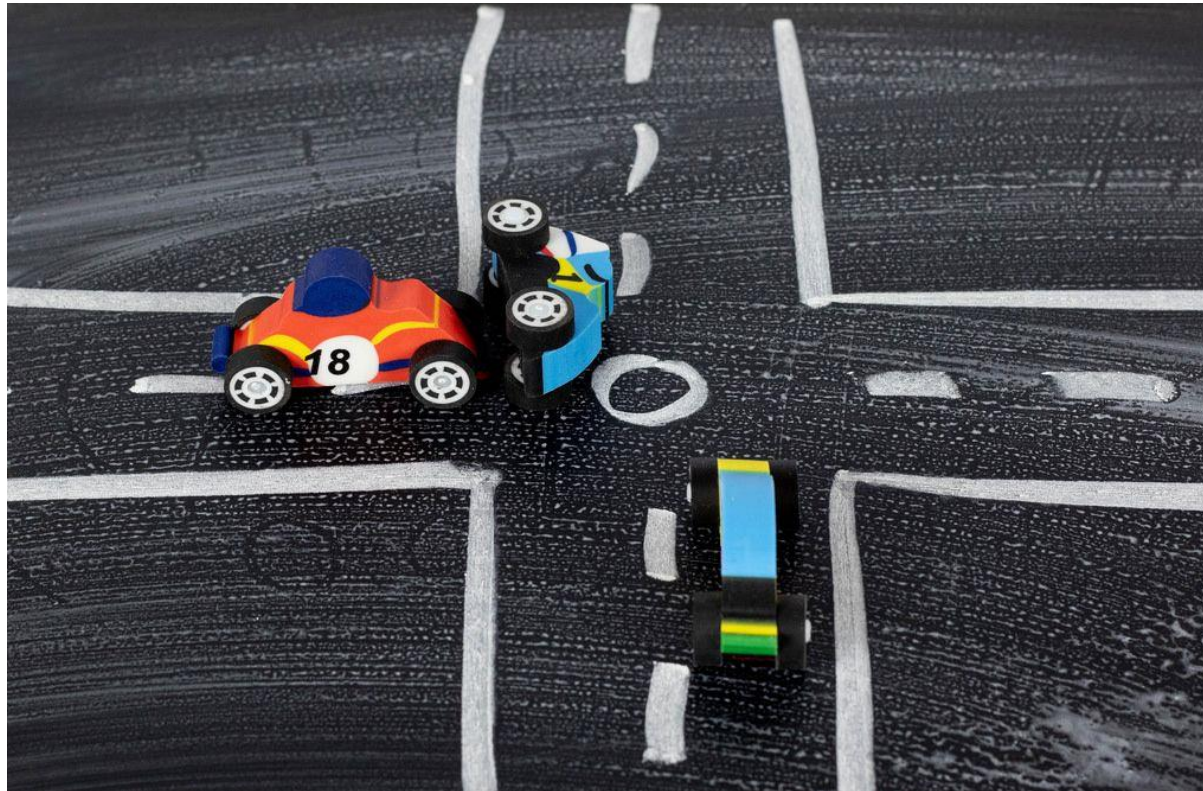
Plan für heute

- checked, unchecked Exceptions und Errors
- try- catches
- Überschriebene Methoden und Exceptions

Exceptions

BUCHSEITEN S.300-324

Grundlagen



"[Dieses Foto](#)" von Unbekannter Autor ist lizenziert gemäß [CC BY](#)

Grundlagen

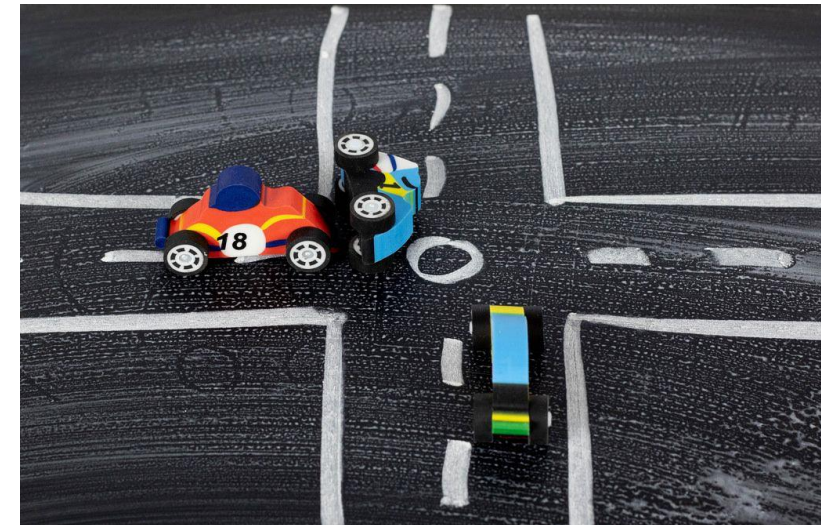
- bei einem Programm kann aus verschiedenen Gründen ein Fehler auftreten
 - Webseite: Not Found 404
 - Zugriff auf ein Element in einem Array mittels invalidem Index
- Exceptions sind Ereignisse, die während der Programmausführung auftreten und den normalen Ablauf eines Programms unterbrechen können
- das Programm kann versuchen auf solche Ereignisse zu reagieren
 - Try/catch/finally - Blöcke

Try-catch-Block

Versuch
Auto zu fahren

Falls ein Unfall
passiert

```
try{  
    autofahren();  
}  
catch(Unfall e){  
    ADAC.abschleppen();  
}
```



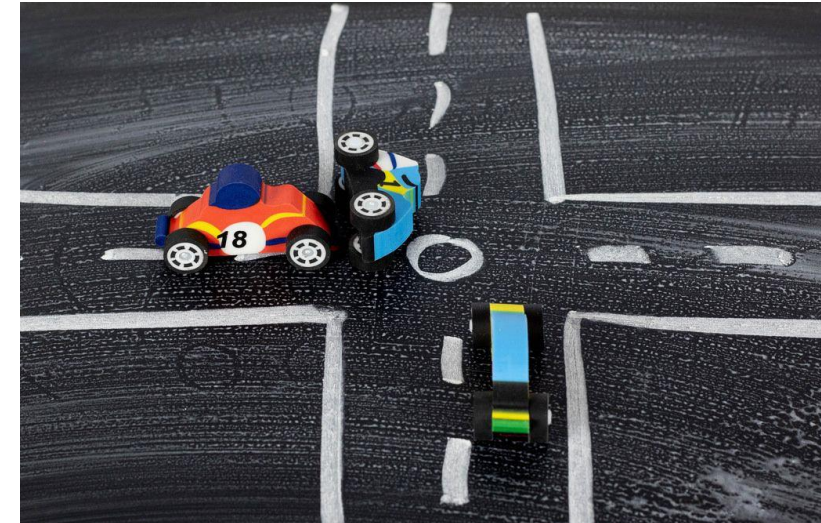
"[Dieses Foto](#)" von Unbekannter Autor ist lizenziert gemäß [CC BY](#)

Try-catch- Block

Versuche den Codeabschnitt
auszuführen

```
try{  
    autofahren();  
}  
  
catch(Unfall e){  
    ADAC.anschleppen();  
}
```

Fange
Exception ab,
falls
etwas passiert.



"[Dieses Foto](#)" von Unbekannter Autor ist lizenziert gemäß [CC BY](#)

Grundlagen

- das Schlüsselwort try darf nicht allein stehen
- es braucht entweder einen catch- oder einen finally-Block
- catch-Block:
 - Anweisungen, die passieren sollen, wenn eine Exception abgefangen wurde
- finally-Block:
 - wird immer ausgeführt, ob eine Exception auftritt oder nicht
- Kann auch beides zusammen stehen
 - Dann aber zuerst catch und dann finally!!

```
try{}  
catch(Exception e){}  
finally{}
```

Try-catch-finally- Block

```
try{  
    autofahren();  
}  
catch(Unfall e){  
    ADAC.abschleppen();  
}  
finally{  
    Auto.abschließen();  
}
```



"Dieses Foto" von Unbekannter Autor ist lizenziert gemäß [CC BY](#)

Try-catch-finally- Block

```
try{
    inDenZooGehen();
    stolpern();
}
catch(Exception e){
    aufstehen();
}
finally{
    weitereTiereAnschauen();
}
```

Try-finally- Block

Finally hat
IMMER das
Schlusswort!

```
try{  
    inDenZooGehen();  
    stolpern();  
}  
finally{  
    System.out.println(„Alles gut“);  
}
```

Übung

Erstelle ein Programm, das eine Zahl durch Null teilt und den Fehler behandelt.
Nutze dafür einen try-catch-Block.



Übung

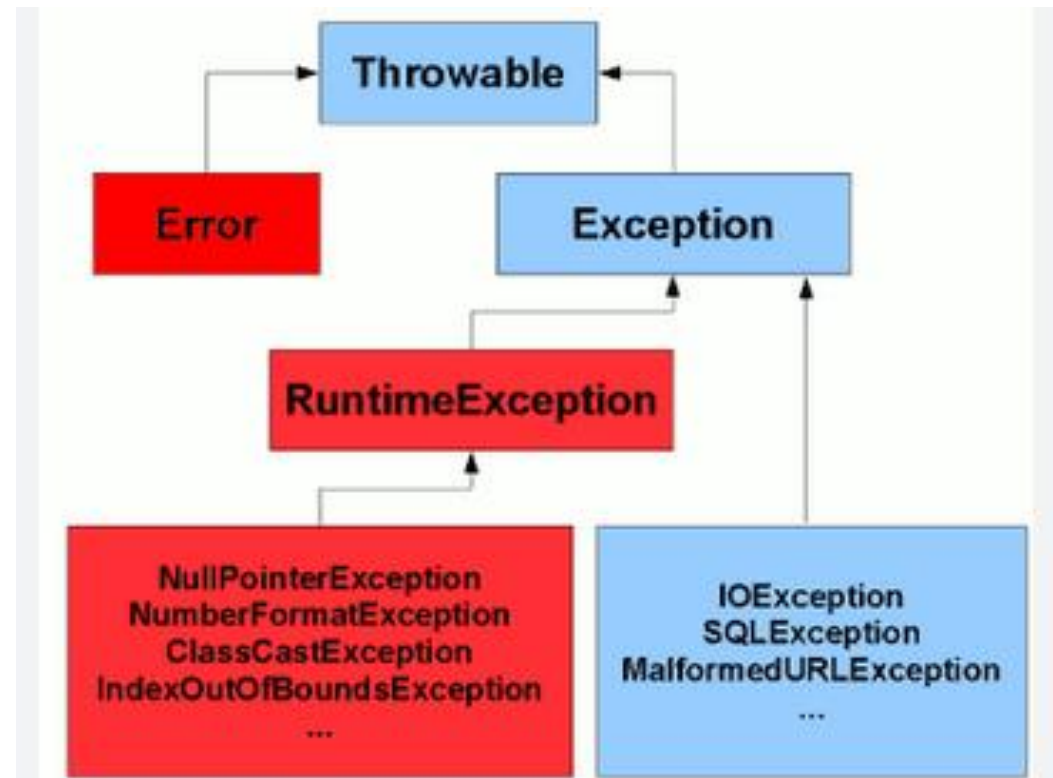
Schreibe ein Java-Programm, das zwei Zahlen vom Benutzer eingibt, ihre Division berechnet und sicherstellt, dass eine Abschlussnachricht immer ausgegeben wird, egal ob ein Fehler auftritt oder nicht.

Schritte:

- Lese zwei Zahlen vom Benutzer ein.
- Versuche, die Division der beiden Zahlen durchzuführen.
- Verwende einen finally-Block, um sicherzustellen, dass eine Abschlussnachricht immer angezeigt wird.



Exceptions



throw new Exception

Signalwort!: Ich kann
Exceptions werfen!

- Exceptions kann man auch selbst werfen:
 - `throw new Exception();`
 - `throw new Exception(„Fehler!“);`
 - `throw new RuntimeException();`
 - `throw new RuntimeException(„Fehler!“);`
- dann muss der Methodenkopf das auch signalisieren, wenn sie nicht direkt wieder abgefangen wird
 - Außer bei RuntimeExceptions

```
static void go() throws Exception {  
    throw new Exception("Ich bin hingefallen!");  
}  
  
public static void main(String[] args) {  
    try{  
        go();  
    } catch (Exception e){  
        System.out.println(e.getMessage());  
    }  
}
```

throw new Exception

- Exceptions kann man auch selbst werfen:
 - `throw new Exception();`
 - `throw new Exception(„Fehler!“);`
 - `throw new RuntimeException();`
 - `throw new RuntimeException(„Fehler!“);`
- dann muss der Methodenkopf das auch signalisieren, wenn sie nicht direkt wieder abgefangen wird

```
static void go() {  
    try{  
        throw new Exception("Ich bin hingefallen!");  
    } catch (Exception e){  
        System.out.println(e.getMessage());  
    }  
}  
  
public static void main(String[] args) {  
    go();  
}
```

Welche Exception würde hier am Ende geworfen werden?

```
try{  
    throw new Exception();  
}  
catch(Exception e){  
    throw new RuntimeException();  
}  
finally{  
    throw new Exception();  
}
```



Welche Exception würde hier am Ende geworfen werden?


```
try{  
    throw new Exception();  
}  
catch(Exception e){  
    throw new RuntimeException();  
}  
finally{  
    throw new Exception();  
}
```

Finally hat das
Schlusswort!



throws Exception

- wenn man in den Methodenkopf throws schreibt, signalisiert man, dass diese Methode eine Exception werfen kann
- somit wird die Verantwortung der Behandlung an den Aufrufer geschoben



```
static void go() throws Exception {  
    throw new Exception("Ich bin hingefallen!");  
}  
  
public static void main(String[] args) {  
    try{  
        go();  
    } catch (Exception e){  
        System.out.println(e.getMessage());  
    }  
}
```


throws Exception

- wenn man in den Methodenkopf throws schreibt, signalisiert man, dass diese Methode eine Exception werfen kann
- somit wird die Verantwortung der Behandlung an den Aufrufer geschoben

```
class FileReader {  
    public void printContent () throws IOException{  
        /* code goes here */  
        throw new IOException();  
    }  
}  
  
class Test {  
    public static void main(String[] args) throws IOException {  
        FileReader xobj = new FileReader();  
        xobj.printContent();  
    }  
}
```

```
class FileReader {  
    public void printContent() throws IOException{  
        /* code goes here */  
        throw new IOException();  
    }  
}  
  
class Test {  
  
    public static void main(String[] args){  
        FileReader xobj = new FileReader();  
        try {  
            xobj.printContent();  
        } catch (IOException e){  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Übung

Schreibe eine Methode, die das Alter einer Person validiert. Die Methode soll sicherstellen, dass das eingegebene Alter eine gültige Zahl ist und im Bereich von 0 bis 120 liegt. Wenn das Alter ungültig ist, soll eine Exception geworfen werden. Schreibe eine Methode `validateAge`, die diese Überprüfung durchführt und eine Exception wirft, wenn das Alter ungültig ist.

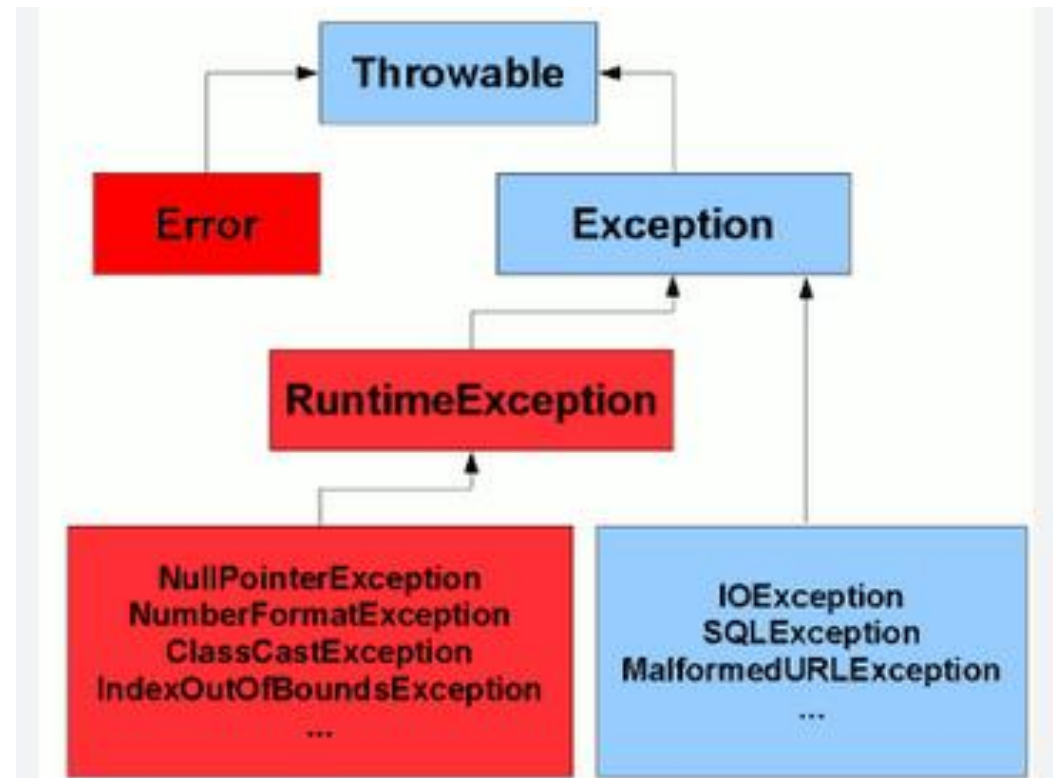
In der Main-Methode soll eine Benutzereingabe entgegen genommen werden. Diese Benutzereingabe soll nun mit der oben definierten Methode überprüft werden. Wie würde man die Fehlerbehandlung machen? Was passiert wenn der Nutzer keine Zahl eingibt (z.B. Abc)? Wie kann man das Problem umgehen?



throw vs throws

Merkmal	throw	throws
Funktion	Wirft eine Ausnahme im Code	Deklariert, dass eine Methode eine oder mehrere Ausnahmen werfen kann
Verwendungsort	Innerhalb des Methodenkörpers	In der Methodensignatur
Syntax	<code>throw new Exception("Nachricht");</code>	<code>public void methodName() throws Exception</code>
Anzahl der Ausnahmen	Nur eine Ausnahme wird geworfen	Mehrere Ausnahmen können deklariert werden
Bedeutung	Löst die Ausnahme aus und übergibt sie an den nächsten Catch-Block oder die aufrufende Methode	Informiert den Compiler und die aufrufende Methode darüber, welche Ausnahmen geworfen werden können

Exceptions



Checked Exceptions

- erben von Exception
- müssen zur Compilerzeit behandelt werden
 - Mittels try/catch oder throws Methodenkopf
- Beispiele:
 - IOException
 - SQLException
 - FileNotFoundException

```
public class WrongExceptionHandling {  
  
    void readCard(int cardNo) throws Exception {  
        System.out.println("Reading Card");  
    }  
  
    public static void main(String[] args) {  
        WrongExceptionHandling ex = new WrongExceptionHandling();  
        int cardNo = 123456789;  
        try {  
            ex.readCard(cardNo);  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Checked Exceptions

- erben von Exception
- müssen zur Compilerzeit behandelt werden
 - Mittels try/catch oder throws Methodenkopf
- Beispiele:
 - IOException
 - SQLException
 - FileNotFoundException

Notwendig!

```
public class WrongExceptionHandling {  
  
    void readCard(int cardNo) throws Exception {  
        System.out.println("Reading Card");  
    }  
  
    public static void main(String[] args) {  
        WrongExceptionHandling ex = new WrongExceptionHandling();  
        int cardNo = 12344;  
        try {  
            ex.readCard(cardNo);  
        } catch (Exception e){  
            System.out.println(e.getMessage());  
        }  
    }  
}
```


Ist folgendes valider Code?

```
public class WrongExceptionHandling {  
    void readCard(int cardNo) throws Exception {           //line1  
        System.out.println("Reading Card");  
    }  
  
    void checkCard(int cardNo) throws RuntimeException {      //line2  
        System.out.println("Checking Card");  
    }  
  
    public static void main(String[] args) {  
        WrongExceptionHandling ex = new WrongExceptionHandling();  
        int cardNo = 123456789;  
        ex.readCard(cardNo);           //line 3  
        ex.checkCard(cardNo);          //line 4  
    }  
}
```

Checked Exceptions:

- müssen zur Compilerzeit behandelt werden
- Mittels try/catch oder throws Methodenkopf

- A. Reading Card
Checking Card
- B. Compilation fails on line 1
- C. Compilation fails on line 2
- D. Compilation fails at line 3
- E. Compilation fails at line 3 and 4



Unchecked Exceptions (Runtime Exceptions)

- erben von RuntimeException
- sind Ausnahmen, die während der Programmausführung auftreten
- müssen nicht zwingend behandelt werden
- zeigen im Regelfall Programmierfehler auf
- Beispiele:
 - NullPointerException
 - ArithmeticException
 - ArrayIndexOutOfBoundsException
 - ClassCastException



Try/catch nicht
notwendig

```
public class UncheckedExceptionExample {  
    public static void main(String[] args) {  
        int[] array = new int[5];  
        System.out.println(array[10]);  
    }  
}
```

Was ist die Ausgabe?

```
public static void main(String[] args) {  
    String[] strs = new String[2];  
    for(String s: strs){  
        s = s.concat(s);  
        System.out.println(s);  
    }  
}
```

- A. null null
- B. nullnullnull
- C. NullPointerException
- D. Keine Konsolenausgabe



Errors

- sind schwerwiegende Probleme, die nicht vom Programm behandelt werden sollten
- erben von Error
- Beispiele:
 - StackOverflowError (oft bei Rekursionen)



Try/catch nicht
notwendig, aber auch
niemals versuchen!

```
public class StackOverflowExample {  
    public static void recursiveMethod() {  
        recursiveMethod();  
    }  
    public static void main(String[] args) {  
        recursiveMethod();  
    }  
}
```

Was ist die Ausgabe?

```
public static void main(String[] args) {  
    ArrayList list = new ArrayList();  
    String[] array;  
    try {  
        for (;;) {  
            list.add("String");  
        }  
    } catch (RuntimeException re) {  
        System.out.println("RuntimeException");  
    } catch (Exception e) {  
        System.out.println("Exception");  
    }  
    System.out.println("Ready");  
}
```

- A. RuntimeException
- B. Exception
- C. Runtime Error in the thread „main“
- D. Ready
- E. Compilation Error



Übung

Erstelle eine Methode, die eine `ArrayOutOfBoundsException` auslöst und eine Methode, die eine `NullPointerException` verursacht.



Unterschied Checked, Unchecked Exception und Errors

Merkmal	Checked Exceptions	Unchecked Exceptions	Errors
Beispiel	IOException, SQLException	NullPointerException, ArrayIndexOutOfBoundsException	OutOfMemoryError, StackOverflowError
Hierarchie	Unterklasse von Exception	Unterklasse von RuntimeException	Unterklasse von Error
Überprüfung zur Kompilierzeit	Ja	Nein	Nein
Muss gefangen oder weitergegeben werden	Ja	Nein	Nein
Verwendung	Handhabung vorhersehbarer Fehler (z.B. Datei nicht gefunden)	Handhabung von Programmierfehlern (z.B. Nullzeiger)	Kritische Fehler, die meist das Programm zum Absturz bringen (z.B. Speicherüberschreitung)
Typische Anwendung	Externe Ressourcen, Datenbankzugriffe	Logikfehler im Programm	Schwere Systemfehler
Behandlungsstrategie	Sollten durch try-catch-Blöcke oder throws-Deklarationen behandelt werden	Können durch try-catch-Blöcke behandelt werden, aber oft nicht notwendig	Selten im Code behandelt, eher Maßnahmen zur Vermeidung notwendig

Mehrere Exceptions abfangen

- es können mehr als eine Exception abgefangen werden
- beim Abfangen mehrerer Exceptions muss man darauf achten, dass sie von spezifisch zu unspezifisch sortiert sind

spezifisch! ←

unspezifisch! ←

```
public static void main (String[] args) {  
    try {  
        int num = 10;  
        int div = 0;  
        int ans = num / div;  
    } catch (ArithmeticException ae) {  
        System.out.println ("No zero divison allowed");  
    } catch (Exception e) {  
        System.out.println ("Invalid calculation");  
    }  
}
```

Einfluss von Exceptions auf den Programmfluss

- eine Exception stoppt das Programm und gibt die Exception weiter

```
import java.io.*;
class CheckedExceptionExample {

    public static void readFile() throws IOException {
        FileReader file = new FileReader("nicht_existierende_datei.txt");
        System.out.println("Datei gefunden");
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("Datei konnte nicht gefunden werden: " + e.getMessage());
        }
    }
}
```

Was ist die A

- A. A
- B. AB
- C. Compilerfehler bei line 1
- D. B
- E. I

```
class MyRuntieException extends RuntimeException {  
  
class Programmfluss {  
    public static void main(String[] args) {  
        try {  
            methodAbcd();  
        } catch (MyRuntieException ne) {  
            System.out.print("A");  
        }  
    }  
  
    public static void methodAbcd() { // line n1  
        try {  
            throw 3 < 10 ? new MyRuntieException() : new FileNotFoundException();  
        } catch (FileNotFoundException ie) {  
            System.out.println("I");  
        } catch (Exception re) {  
            System.out.print("B");  
        }  
    }  
}
```



Vorteile von Exception-Handling

- Verbesserte Lesbarkeit: Fehler können an einer zentralen Stelle behandelt werden
- bessere Wartbarkeit: Der Code bleibt sauber und ist besser zu debuggen
- Verhinderung von Programmabstürzen: Fehler können abgefangen und verarbeitet werden

Exceptions und überschreibende Methoden

- sowohl überschreibende Methoden von Interface- oder Subklassen haben besondere Regeln für den Methodenkopf
- die überschriebenen Methoden dürfen checked Exceptions deklarieren
 - Die überschreibende Methode darf die Exceptions oder Subklassen der Exceptions im Methodenkopf deklarieren, aber KEINE anderen oder übergeordneten checked Exceptions
 - Die überschreibende Methode darf RuntimeExceptions unabhängig deklarieren

Ist folgendes valider Code?

```
import java.io.IOException;

abstract class MyAbstClass {
    void myMethod() throws IOException;
}

class MyClass extends MyAbstClass {
    @Override
    public void myMethod() throws IOException {
    }
}
```

Passt!

- sowohl überschreibende Methoden von Interface- oder Subklassen haben besondere Regeln für den Methodenkopf
- die überschriebenen Methoden dürfen checked Exceptions deklarieren
 - Die überschreibende Methode darf die Exceptions oder Subklassen der Exceptions im Methodenkopf deklarieren, aber KEINE anderen oder übergeordneten checked Exceptions
 - Die überschreibende Methode darf RuntimeExceptions unabhängig deklarieren



Ist folgendes valider Code?

```
import java.io.IOException;

interface MyInterface {
    void myMethod() throws IOException;
}

class MyClass implements MyInterface {
    @Override
    public void myMethod() {

    }
}
```

Passt!

- sowohl überschreibende Methoden von Interface- oder Subklassen haben besondere Regeln für den Methodenkopf
- die überschriebenen Methoden dürfen checked Exceptions deklarieren
 - Die überschreibende Methode darf die Exceptions oder Subklassen der Exceptions im Methodenkopf deklarieren, aber KEINE anderen oder übergeordneten checked Exceptions
 - Die überschreibende Methode darf RuntimeExceptions unabhängig deklarieren



Ist folgendes valider Code?

```
import java.io.IOException;

abstract class MyAbstClass {
    void myMethod() throws IOException;
}

class MyClass extends MyAbstClass {
    @Override
    public void myMethod() throws FileNotFoundException {

    }
}
```

- sowohl überschreibende Methoden von Interface- oder Subklassen haben besondere Regeln für den Methodenkopf
- die überschriebenen Methoden dürfen checked Exceptions deklarieren
 - Die überschreibende Methode darf die Exceptions oder Subklassen der Exceptions im Methodenkopf deklarieren, aber KEINE anderen oder übergeordneten checked Exceptions
 - Die überschreibende Methode darf RuntimeExceptions unabhängig deklarieren

Passt! Da FileNotFoundException hierarchisch unter IOException steht



Ist folgendes valider Code?

```
import java.io.IOException;

interface MyInterface {
    void myMethod() throws IOException;
}

class MyClass implements MyInterface {
    @Override
    public void myMethod() throws IOException, NullPointerException {

    }
}
```

- sowohl überschreibende Methoden von Interface- oder Subklassen haben besondere Regeln für den Methodenkopf
- die überschriebenen Methoden dürfen checked Exceptions deklarieren
 - Die überschreibende Methode darf die Exceptions oder Subklassen der Exceptions im Methodenkopf deklarieren, aber KEINE anderen oder übergeordneten checked Exceptions
 - ❗ Die überschreibende Methode darf RuntimeExceptions unabhängig deklarieren

Passt! NullPointerException gehört zu RuntimeException und darf in der überschreibenden Methode im Methodenkopf stehen



Ist folgendes valider Code?

```
import java.io.IOException;

abstract class MyAbstrClass {
    void myMethod() throws IOException;
}

class MyClass extends MyAbstrClass {
    @Override
    public void myMethod() throws Exception {

    }
}
```

- sowohl überschreibende Methoden von Interface- oder Subklassen haben besondere Regeln für den Methodenkopf
- die überschriebenen Methoden dürfen checked Exceptions deklarieren
 - Die überschreibende Methode darf die Exceptions oder Subklassen der Exceptions im Methodenkopf deklarieren, aber KEINE anderen oder übergeordneten checked Exceptions
 - Die überschreibende Methode darf RuntimeExceptions unabhängig deklarieren

Falsch! Exception steht hierarchisch über IOException -> geht nicht. Nur Exceptions unter IOException funktionieren



Ist folgendes valider Code?

```
import java.io.IOException;

interface MyInterface {
    void myMethod();
}

class MyClass implements MyInterface {
    @Override
    public void myMethod() throws IOException {

    }
}
```

- sowohl überschreibende Methoden von Interface- oder Subklassen haben besondere Regeln für den Methodenkopf
- die überschriebenen Methoden dürfen checked Exceptions deklarieren
 - Die überschreibende Methode darf die Exceptions oder Subklassen der Exceptions im Methodenkopf deklarieren, aber KEINE anderen oder übergeordneten checked Exceptions
 - Die überschreibende Methode darf RuntimeExceptions unabhängig deklarieren

Passt nicht! Überschriebene Methoden können nicht plötzlich eine Exception werfen



Übung

Die verschiedenen Typen der möglichen Exceptions hattet ihr im Kurs präsentiert.
Zur Wiederholung und Auffrischung: Lest euch die Buchseiten 314-318 durch.

