

Java Programmierung

Exceptions, Fehlersuche und Testen

Exceptions

Prof. Dr. rer. nat. Andreas Berl
Fakultät für Angewandte Informatik
Technische Hochschule Deggendorf

Lernziele

- Ausnahmen in Java
- Varianten von Ausnahmesituationen
- Ungeprüfte Ausnahmen
 - Ignorieren einer Ausnahme
 - Fangen einer Ausnahme
- Geprüfte Ausnahmen
 - Weiterleiten einer Ausnahme
- Vererbungshierarchie von Ausnahmen
- Die try-catch-Behandlung
 - Aufbau und Regeln
 - Mehrere Ausnahmen
 - Die finally-Klausel
 - Schließen von Ressourcen
- Java Stack Trace
- Catch or throws?
- Ausnahmen erzeugen
 - Ausnahmen werfen mit **throw**
 - Auswahl einer passenden Ausnahme
 - Geprüfte oder ungeprüfte Ausnahmen?
 - Eigene Exceptionklassen erstellen

Ausnahmen (Exceptions)


Möglichkeit zur Fehlerbehandlung in Java

- **Strukturiertes Verfahren** zum Umgang mit Fehlersituationen während der Programmausführung
- **Exceptions** sind spezielle Klassen um solche Ausnahmen zu erzeugen
 - Verhalten sich anders als „normale“ Klassen

Auslösen und Weitergabe von Ausnahmen

- Sobald ein Programm **nicht mehr sinnvoll** weiterlaufen kann, wird eine Ausnahme erzeugt
- Die Ausnahme wird **von Methode zu Methode** an die jeweils aufrufende Methode weitergegeben
- Wenn eine Ausnahme **nicht im Code behandelt** wird landet sie letztendlich bei der main-Methode und das Programm beendet sich mit einer Fehlermeldung

Beispiel: In einem Spiel schlägt das Lesen einer Datei fehl

main() → spiel() → ladeSpielstand() → leseDatei()  **FileNotFoundException!**

Ausnahmen (Exceptions)

Möglichkeit zur Fehlerbehandlung in Java

- **Strukturiertes Verfahren** zum Umgang mit Fehlersituationen während der Programmausführung
- **Exceptions** sind spezielle Klassen um solche Ausnahmen zu erzeugen
 - Verhalten sich anders als „normale“ Klassen

Auslösen und Weitergabe von Ausnahmen

- Sobald ein Programm **nicht mehr sinnvoll** weiterlaufen kann, wird eine Ausnahme erzeugt
- Die Ausnahme wird **von Methode zu Methode** an die jeweils aufrufende Methode weitergegeben
- Wenn eine Ausnahme **nicht im Code behandelt** wird landet sie letztendlich bei der main-Methode und das Programm beendet sich mit einer Fehlermeldung

Beispiel: In einem Spiel schlägt das Lesen einer Datei fehl

Programmabsturz ⚡ ← `main()` ← `spiel()` ← `ladeSpielstand()` ← `leseDatei()` ⚡ *FileNotFoundException!*

Ausnahmen (Exceptions)

- Geprüfte und ungeprüft Ausnahmen, harte Fehler

Drei unterschiedliche Varianten von Ausnahmesituationen

- **Ungeprüfte Ausnahmen** (unchecked Exceptions)
 - Der Compiler **überprüft nicht**, ob diese Ausnahmen behandelt werden
 - **Können** im Code behandelt werden
 - `NullPointerException`, `IllegalArgumentException`, `IndexOutOfBoundsException`, ...
- **Geprüfte Ausnahmen** (checked Exceptions)
 - Der Compiler **stellt sicher**, dass diese Ausnahmen behandelt werden
 - **Müssen** im Code behandelt werden
 - `IOException`, `FileNotFoundException`, `SQLException`, ...
- **Harte Fehler** (Errors) - meist im Zusammenhang mit der JVM
 - Der Compiler **überprüft nicht**, ob harte Fehler behandelt werden
 - **Dürfen nicht** im Code behandelt werden
 - `OutOfMemoryError`, `IOError`, `ThreadDeath`, ...

Ungeprüfte Ausnahmen

Ungeprüfte Ausnahmen (unchecked Exceptions)

- Ungeprüfte Ausnahmen sind von der Klasse **RuntimeException** abgeleitet
- Der Compiler prüft bei diesen Ausnahmen **nicht**, ob die Ausnahme im Code behandelt wird
 - Man kann wählen, ob man den Fehler behandeln möchte oder nicht

Behandeln von ungeprüften Ausnahmen

- **Möglichkeit 1: Ignorieren der Ausnahme**
 - Irgendwann erreicht die Ausnahme die **main**-Methode
 - Das Programm wird mit einer Fehlermeldung, dem sogenannten **Stacktrace** beendet
- **Möglichkeit 2: Fangen der Ausnahme**
 - In einer passenden Methode wird die Ausnahme in einer **try-catch**-Behandlung behoben
 - Das Programm läuft danach **normal weiter**

Eine einfache **try-catch**-Behandlung

```
try {  
    // Code der eine Ausnahme auslösen kann  
} catch (Exceptionklasse e) {  
    // Code zur Fehlerbehandlung  
}
```

Ungeprüfte Ausnahmen

- Möglichkeit 1: Ignorieren der Ausnahme

Beispiel: Ein Programm mit einer unbehandelten Ausnahme: `NumberFormatException`

```

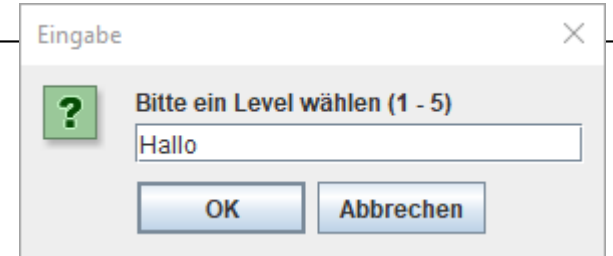
01 package battleship;
02 class BattleShip {
03
04     public static void main(String[] args) {
05         new BattleShip().startGame();
06     }
07
08     private void startGame() {
09         LevelChooser levelChooser = new LevelChooser();
10         int level = levelChooser.letUserSelectLevel();
11         // Code
12     }
13 }

```

```

01 package battleship;
02 import javax.swing.*;
03
04 class LevelChooser {
05
06     int letUserSelectLevel() {
07         String message = "Bitte ein Level wählen (1 - 5)";
08         String input = JOptionPane.showInputDialog(message);
09         return Integer.parseInt(input);
10     }
11 }
12
13

```



```

java.lang.Integer
public static int parseInt(@NotNull String s)
throws NumberFormatException

```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value or an ASCII plus sign '+' ('\u002B') to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(String, int)` method.

Params: s – a String containing the int representation to be parsed

Returns: the integer value represented by the argument in decimal.

Throws: `NumberFormatException` – if the string does not contain a parsable integer.

Ungeprüfte Ausnahmen

- Möglichkeit 1: Ignorieren der Ausnahme

Beispiel: Ein Programm mit einer unbehandelten Ausnahme: `NumberFormatException`

```

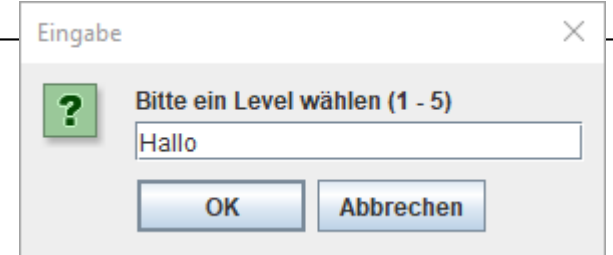
01 package battleship;
02 class BattleShip {
03
04     public static void main(String[] args) {
05         new BattleShip().startGame();
06     }
07
08     private void startGame() {
09         LevelChooser levelChooser = new LevelChooser();
10         int level = levelChooser.letUserSelectLevel();
11         // Code
12     }
13 }

```

```

01 package battleship;
02 import javax.swing.*;
03
04 class LevelChooser {
05
06     int letUserSelectLevel() {
07         String message = "Bitte ein Level wählen (1 - 5)";
08         String input = JOptionPane.showInputDialog(message);
09         return Integer.parseInt(input);
10     }
11 }
12
13

```



Beispiel: Der Stacktrace wird auf der Kommandozeile angezeigt

```

Exception in thread "main" java.lang.NumberFormatException: For input string: "Hallo"
at java.base/java.lang.Integer.parseInt(Integer.java:630)
at java.base/java.lang.Integer.parseInt(Integer.java:786)
at battleship.LevelChooser.letUserSelectLevel(LevelChooser.java:9)
at battleship.BattleShip.startGame(BattleShip.java:10)
at battleship.BattleShip.main(BattleShip.java:5)

```

Spezifische Nachricht
der Ausnahme.

Links: Springen an
die gezeigte Stelle im
Code.

Ungeprüfte Ausnahmen

- Möglichkeit 2: Fangen der Ausnahme

Beispiel: Ein Programm mit einer behandelten Ausnahme: `NumberFormatException`

```
package battleship;
class BattleShip {

    public static void main(String[] args) {
        new BattleShip().startGame();
    }

    private void startGame() {
        LevelChooser levelChooser = new LevelChooser();
        int level;
        try {
            level = levelChooser.letUserSelectLevel();
        } catch (NumberFormatException e) {
            level = 1;
            System.err.println("Falscheingabe des Benutzers!");
            System.err.println("-> " + e.getMessage());
            System.err.println("Level wird auf 1 gesetzt.");
        }
        // Code
    }
}
```

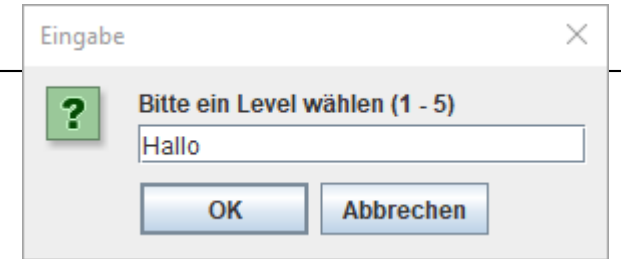
Beispiel: Konsolenausgabe

Falscheingabe des Benutzers!
-> For input string: "Hallo"
Level wird auf 1 gesetzt.

```
package battleship;
import javax.swing.*;

class LevelChooser {

    int letUserSelectLevel() {
        String message = "Bitte ein Level wählen (1 - 5)";
        String input = JOptionPane.showInputDialog(message);
        return Integer.parseInt(input);
    }
}
```



Mit **System.err.** Können Nachrichten auf der Standardfehlerausgabe ausgegeben werden.

Mit **e.getMessage()** kann die spezifische Nachricht der Exception zurück gegeben werden.

Geprüfte Ausnahmen

Geprüfte Ausnahmen (checked Exceptions)

- Geprüfte Ausnahmen sind von der Klasse **Exception** abgeleitet
- Der Compiler **stellt sicher**, dass die Ausnahme im Code behandelt wird

Behandeln von geprüften Ausnahmen

- **Möglichkeit 1: Weitergeben der Ausnahme**
 - Die Ausnahme muss mit **throws** **explizit** an die aufrufende Methode weitergereicht werden
 - Es können auch **mehrere Exceptions** durch Komma getrennt weitergegeben werden
 - Falls auch die main-Methode die Ausnahme mit **throws** weiterreicht, stürzt das Programm ab
- **Möglichkeit 2: Fangen der Ausnahme**
 - Mit einer **try-catch**-Behandlung, genau wie bei ungeprüften Ausnahmen

Beispiel: Eine Methode mit **throws**

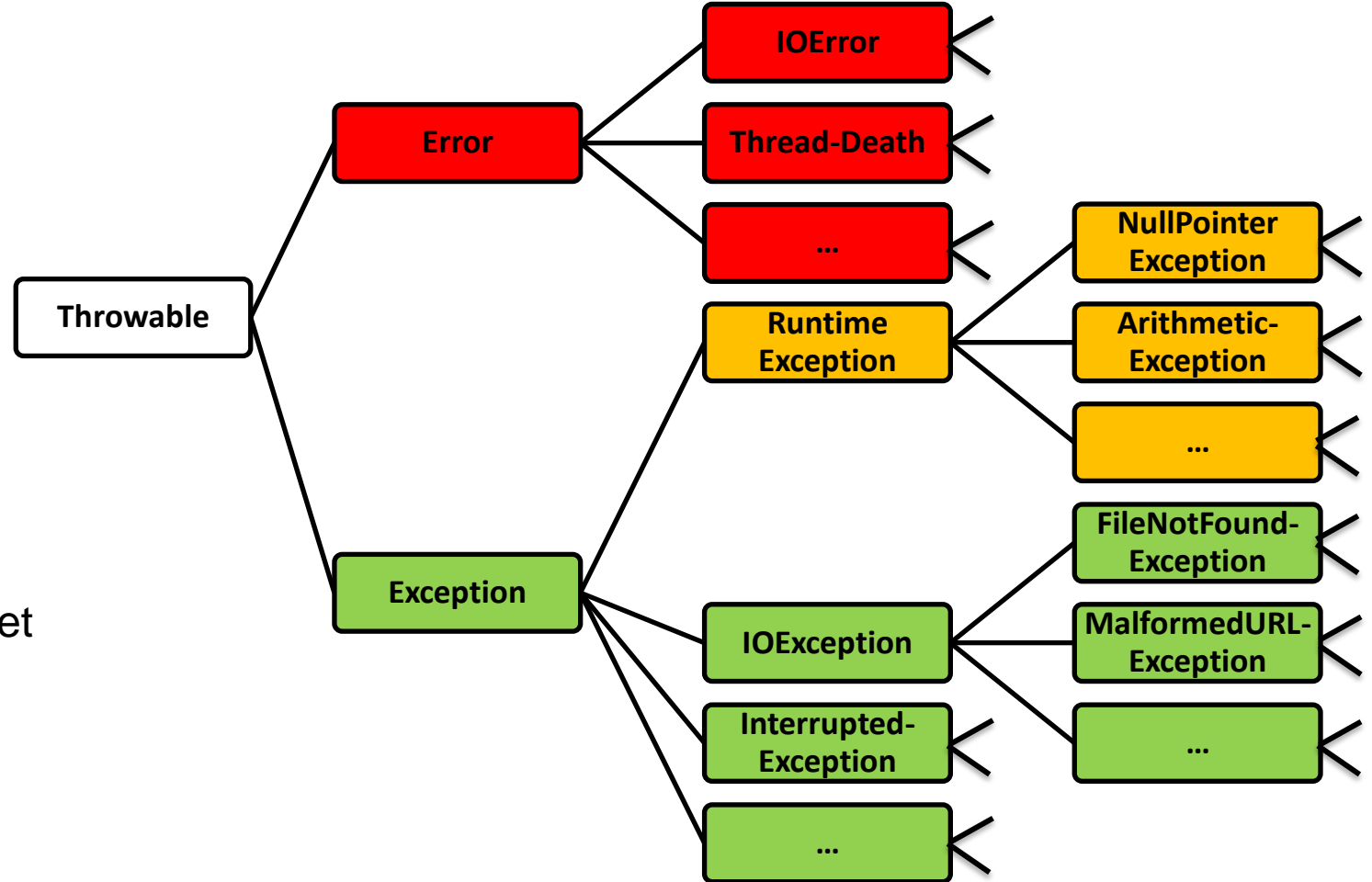
```
String readFromDisc() throws FileNotFoundException {  
    // Code der eine Ausnahme auslösen kann  
}
```

Erweiterung der
Methodensignatur mit **throws**.

Vererbungshierarchie

Vererbungshierarchie von Ausnahmen und Fehlern

- **Alle Ausnahmen und Fehler**
 - Sind direkt oder indirekt von der Klasse **Throwable** abgeleitet
- **Geprüfte Ausnahmen (grün)**
 - Sind direkt oder indirekt von der Klasse **Exception** abgeleitet
- **Ungeprüfte Ausnahmen (orange)**
 - Sind direkt oder indirekt von der Klasse **RuntimeException** abgeleitet
- **Harte Fehler (rot)**
 - Sind direkt oder indirekt von der Klasse **Error** abgeleitet



Die try-catch-Behandlung

Aufbau einer try-catch Behandlung

- Die **try**-Klausel
 - Enthält den Code, der eine Ausnahme erzeugen kann
 - Falls eine Ausnahme auftritt
 - Nachfolgender Code in der **try**-Klausel wird **nicht** ausgeführt
 - Es wird sofort in die **passende** catch-Klausel gesprungen
- Die **catch**-Klausel
 - Enthält den Code um den aufgetretenen Fehler zu beheben
 - Wird nur ausgeführt, falls eine **passende** Exception auftritt
- Die **finally**-Klausel
 - Code der auf **jeden Fall** ausgeführt wird
 - Egal ob ein Fehler aufgetreten ist oder nicht

Die try-catch-Behandlung

```
try {  
    // Code bei eine Ausnahme auslösen kann  
} catch (Exceptionklasse e) {  
    // Code zur Fehlerbehandlung  
} finally {  
    // Code der in jedem Fall ausgeführt wird  
}
```

Beispiel: Lese Spiel-Einstellungen von der Festplatte

```
String settings;  
try {  
    settings = readGameSettingsFromDisc();  
    System.out.print("Lesen erfolgreich!");  
} catch (FileNotFoundException e) {  
    System.err.print("Lesen nicht möglich!");  
    settings = Gameplay.DEFAULT_SETTINGS;  
} finally {  
    game.setCurrentSettings(settings);  
}
```

Die try-catch-Behandlung

- Die catch-Klausel

Allgemeine Regeln für catch-Klauseln

- Es ist verboten die Oberklassen **Exception** oder **Throwable** in einer **catch**-Klausel zu fangen
 - Alle möglichen Arten von Fehlern würden auf diese Weise einfach verschwinden
 - Statt dessen: Eine spezifische Exception fangen, z.B. **FileNotFoundException**
- **Catch or throws-Regel:** Ausnahmen **müssen** entweder **weitergegeben** oder in einer catch-Klausel **behandelt** werden
 - Keine **leeren catch**-Klauseln!
 - Leere catch-Klauseln lassen die Ausnahme verschwinden
 - Mindestanforderung: Ausgabe einer Fehlermeldung oder Logging der Ausnahme
 - Besser: Reparieren des aufgetretenen Fehlers

Die try-catch-Behandlung

- Die catch-Klausel

Allgemeine Regeln für catch-Klauseln

- Es ist verboten die Oberklassen **Exception** oder **Throwable** in einer **catch**-Klausel zu fangen
 - Alle möglichen Arten von Fehlern würden auf diese Weise einfach verschwinden
 - Statt dessen: Eine spezifische Exception fangen, z.B. **FileNotFoundException**
- **Catch or throws-Regel:** Ausnahmen **müssen** entweder **weitergegeben** oder in einer catch-Klausel **behandelt** werden
 - Keine **leeren catch**-Klauseln!
 - Leere catch-Klauseln lassen die Ausnahme verschwinden
 - Mindestanforderung: Ausgabe einer Fehlermeldung oder Logging der Ausnahme
 - Besser: Reparieren des aufgetretenen Fehlers
- **Seltene Sonderfälle**
 - Wenn die Ausnahme wirklich egal ist und nichts passieren kann
→ Parametername **ignored** wählen

Beispiel: Unproblematische InterruptedException

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException ignored) {  
}
```

Die try-catch-Behandlung

- Die catch-Klausel

Behandeln mehrerer unterschiedlicher Ausnahmen

- **Möglichkeit 1**
 - Behandlung in separaten **catch**-Klauseln
 - Nur **eine** der Klauseln wird ausgeführt
- **Möglichkeit 2**
 - Fangen einer gemeinsamen Oberklasse (z.B. **IO-Exception**)
- **Möglichkeit 3**
 - **Multicatch**: Mehrere **catch**-Klauseln werden zusammengefasst

Separate **catch**-Klauseln

```
try {
    // Code der Ausnahmen auslösen kann
} catch (Exceptionklasse e1) {
    // Code zur Fehlerbehandlung von Ausnahme 1
} catch (Exceptionklasse e2) {
    // Code zur Fehlerbehandlung von Ausnahme 2
} catch (Exceptionklasse e3) {
    // Code zur Fehlerbehandlung von Ausnahme 3
}
```

catch-Klausel mit gemeinsamer Oberklasse

```
try {
    // Code der Ausnahmen auslösen kann
} catch (Exceptionklasse e1) {
    // Code zur Fehlerbehandlung von Ausnahme 1
} catch (Exceptionklasse oberklasse) {
    // Code zur Fehlerbehandlung von Ausnahme 2 und 3
}
```

Oberklasse von
e2 und e3.

Multicatch

```
try {
    // Code der Ausnahmen auslösen kann
} catch (Exceptionklasse | Exceptionklasse | Exceptionklasse e) {
    // Code zur Fehlerbehandlung der Ausnahmen 1, 2 und 3
}
```

Die try-catch-Behandlung

- Die finally-Klausel

Besonderheiten der finally-Klausel

- Die **finally**-Klausel wird **immer** ausgeführt
 - Egal ob eine Ausnahme aufgetreten ist oder nicht
 - Sogar dann, wenn in der **try**- oder **catch**-Klausel ein **break**, **continue** oder **return** steht
 - **Vorsicht:** Ein **return**-Befehl der **finally**-Klausel kann einen **return**-Befehl aus der **try**- oder **catch**-Klausel überdecken
- Es darf auch eine **finally**-Klausel ohne eine **catch**-Klausel geben
 - Die Ausnahme wird dann ggf. mit **throws** weitergeleitet
 - Unabhängig vom Auftreten einer Ausnahme wird der Code in der **finally**-Klausel ausgeführt

Beispiel: **finally** wird auf jeden Fall ausgeführt

```
private int berechneZahl() {  
  
    try {  
        // Code der Ausnahmen auslösen kann  
        return 5;  
    } catch (IllegalArgumentException e) {  
        // Fehlerbehandlung  
        return 6;  
    } finally {  
        System.out.print("Finally!");  
        return 7;  
    }  
}
```

Diese Methode liefert in jedem Fall **7** zurück und gibt „Finally!“ aus.

Die try-catch-Behandlung


- Schließen von Ressourcen

Schließen von Ressourcen

- Streams müssen nach ihrem Gebrauch wieder geschlossen werden
- Dabei kann es leicht zu einer verschachtelten **try-catch**-Behandlung kommen

Beispiel: Eine verschachtelte try-catch-Behandlung

```
private void schreibeInTagebuch(File datei) {  
    RandomAccessFile tagebuch = null;  
    try {  
        tagebuch = new RandomAccessFile(datei, "rw");  
        tagebuch.writeChars("Liebes Tagebuch ...");  
        // ...  
    } catch (IOException e) {  
        // Fehlerbehandlung  
    } finally {  
        if (tagebuch != null) {  
            try { tagebuch.close(); } catch (IOException ignored) { }  
        }  
    }  
}
```



Die try-catch-Behandlung

- Schließen von Ressourcen

Schließen von Ressourcen

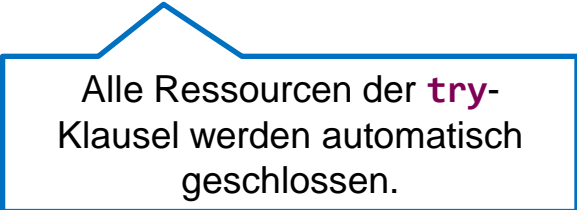
- Streams müssen nach ihrem Gebrauch wieder geschlossen werden
- Dabei kann es leicht zu einer verschachtelten **try-catch**-Behandlung kommen

Automatisches Schließen

- Eine spezielle **try**-Klausel ermöglicht das automatische Schließen von Ressourcen
- Die Ressourcen müssen dazu das Interface **AutoCloseable** implementieren
- Mehrere Ressourcen werden durch ein Semikolon getrennt

Beispiel: Automatisches Schließen von Ressourcen

```
private void schreibeInTagebuch(File datei) {  
    try (RandomAccessFile tagebuch = new RandomAccessFile(datei, "rw")) {  
        tagebuch.writeChars("Liebes Tagebuch ...");  
        // ...  
    } catch (IOException e) {  
        // Fehlerbehandlung  
    }  
}
```



Java Stack Trace

Der Java Stack Trace (Stapelrückverfolgung)

- Zeigt welche Ausnahme erzeugt wurde und enthält ggf. eine konkrete Nachricht zur Ausnahme
- Gibt Informationen über die Methoden, die aufgerufen wurden bevor es zur Ausnahme kam
- Enthält **Links** zum direkten Anspringen der relevanten Stellen im Code

Beispiel: Ein Stacktrace einer NumberFormatException

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hallo"
at java.base/java.lang.Integer.parseInt(Integer.java:630)
at java.base/java.lang.Integer.parseInt(Integer.java:786)
at battleship.LevelChooser让用户selectLevel(LevelChooser.java:9)
at battleship.BattleShip.startGame(BattleShip.java:10)
at battleship.BattleShip.main(BattleShip.java:5)
```

- Die Klasse **Throwable** bietet einige Methoden zum Java Stack Trace, z.B.

```
public void printStackTrace() // Ausgabe des Java Stack Trace auf der Kommandozeile
```

```
public StackTraceElement[] getStackTrace() // Rückgabe als StackTraceElement-Array
```

```
public String getMessage() // Rückgabe der Nachricht zur Ausnahme
```

Catch or throws?

In welcher Methode fängt man eine Ausnahme?

- Soll man in einer Methode die Ausnahme mit **throws** weitergeben (werfen)?
- Oder soll man lieber die Ausnahme mit einer **try-catch**-Behandlung fangen?
- **Fangen einer Ausnahme:** Wenn eine Methode die **notwendigen Informationen** hat um die Situation aufzulösen, sollte sie die Ausnahme fangen und beheben
- **Beispiel**
 - Ausnahme: In einem Spiel schlägt das Lesen eines Spielstandes aus einer Datei fehl
 - Fehlerbehandlung: Ein neuer Spielstand soll erzeugt werden

```
main() → spielStart() → ladeSpielstand() → leseDatei() → findeSpielstand()
```



Catch or throws?

In welcher Methode fängt man eine Ausnahme?

- Soll man in einer Methode die Ausnahme mit **throws** weitergeben (werfen)?
- Oder soll man lieber die Ausnahme mit einer **try-catch**-Behandlung fangen?
- **Fangen einer Ausnahme:** Wenn eine Methode die **notwendigen Informationen** hat um die Situation aufzulösen, sollte sie die Ausnahme fangen und beheben
- **Beispiel**
 - Ausnahme: In einem Spiel schlägt das Lesen eines Spielstandes aus einer Datei fehl
 - Fehlerbehandlung: Ein neuer Spielstand soll erzeugt werden

`main() → spielStart() → ladeSpielstand() → leseDatei() → findeSpielstand()` ⚡ ***FileNotFoundException!***



Catch or throws?

In welcher Methode fängt man eine Ausnahme?

- Soll man in einer Methode die Ausnahme mit **throws** weitergeben (werfen)?
- Oder soll man lieber die Ausnahme mit einer **try-catch**-Behandlung fangen?
- **Fangen einer Ausnahme:** Wenn eine Methode die **notwendigen Informationen** hat um die Situation aufzulösen, sollte sie die Ausnahme fangen und beheben
- **Beispiel**
 - Ausnahme: In einem Spiel schlägt das Lesen eines Spielstandes aus einer Datei fehl
 - Fehlerbehandlung: Ein neuer Spielstand soll erzeugt werden



main() → spielStart() → ladeSpielstand() → leseDatei() → **findeSpielstand()** ⚡ **FileNotFoundException!**

Hier nicht lösbar: werfen!
Keine Infos über Spielstände

Catch or throws?

In welcher Methode fängt man eine Ausnahme?

- Soll man in einer Methode die Ausnahme mit **throws** weitergeben (werfen)?
- Oder soll man lieber die Ausnahme mit einer **try-catch**-Behandlung fangen?
- **Fangen einer Ausnahme:** Wenn eine Methode die **notwendigen Informationen** hat um die Situation aufzulösen, sollte sie die Ausnahme fangen und beheben
- **Beispiel**
 - Ausnahme: In einem Spiel schlägt das Lesen eines Spielstandes aus einer Datei fehl
 - Fehlerbehandlung: Ein neuer Spielstand soll erzeugt werden



main() → spielStart() → ladeSpielstand() → leseDatei() → findeSpielstand() ⚡ **FileNotFoundException!**

Hier nicht lösbar: werfen!
Keine Infos über Spielstände

Catch or throws?

In welcher Methode fängt man eine Ausnahme?

- Soll man in einer Methode die Ausnahme mit **throws** weitergeben (werfen)?
- Oder soll man lieber die Ausnahme mit einer **try-catch**-Behandlung fangen?
- **Fangen einer Ausnahme:** Wenn eine Methode die **notwendigen Informationen** hat um die Situation aufzulösen, sollte sie die Ausnahme fangen und beheben
- **Beispiel**
 - Ausnahme: In einem Spiel schlägt das Lesen eines Spielstandes aus einer Datei fehl
 - Fehlerbehandlung: Ein neuer Spielstand soll erzeugt werden



main() → spielStart() → ladeSpielstand() → leseDatei() → findeSpielstand() ⚡ **FileNotFoundException!**

Hier lösbar: fangen!
Anstatt einen Spielstand zu laden,
wird ein neuer Spielstand erzeugt

Erzeugen von Ausnahmen

Ausnahmen erzeugen mit throw

- Programmierer können in Methoden und Konstruktoren jederzeit Ausnahmen erzeugen
 - Dann, wenn ein Programm nicht mehr weiter sinnvoll ausgeführt werden kann
- Das Schlüsselwort **throw** signalisiert eine Ausnahme
 - Eine **Exception** wird mit dem **new**-Operator erzeugt und sollte eine hilfreiche Nachricht als Übergabeparameter enthalten
 - Die Abarbeitung wird dann sofort an dieser Stelle abgebrochen

Beispiel: Erzeugen einer InputMismatchException

```
private String benutzerAuswahl() {  
    String eingabe = JOptionPane.showInputDialog("Bitte A oder B eingeben");  
    if (!eingabe.equals("A") && !eingabe.equals("B")) {  
        throw new InputMismatchException("Nur A oder B sind als Eingabe möglich!");  
    }  
    return eingabe;  
}
```

Erzeugen von Ausnahmen

- Welche Ausnahme ist passend?

Auswahl von Ausnahmen

- Es ist **verboten** eine Ausnahme vom Typ **Throwable** oder **Exception** zu werfen
 - Statt dessen sollte eine **möglichst passende** Ausnahme ausgewählt werden, wie z. B.
 - `IllegalArgumentException` // Übergabe von fehlerhaften Parametern
 - `IllegalStateException` // Falls im aktuellen Zustand eine Aktion nicht möglich ist
 - `UnsupportedOperationException` // Wenn eine geerbte Methode nicht implementiert wurde
 - `IndexOutOfBoundsException` // Bei falschen Index-Zugriffen
 - `NullPointerException` // Wenn auf eine Referenz zugegriffen wird, die auf null zeigt
 - `InputMismatchException` // Wenn eine Eingabe nicht dem gewünschten Muster entspricht
- Falls es keine passende Exceptionklasse gibt, ist es üblich eine **eigene Klasse** zu erstellen
 - **Ungeprüfte Ausnahmen** werden direkt oder indirekt von **RuntimeException** abgeleitet
 - **Geprüfte Ausnahmen** werden direkt oder indirekt **Exception** abgeleitet

Erzeugen von Ausnahmen

- Geprüfte oder ungeprüfte Ausnahmen?

Ungeprüfte Ausnahmen signalisieren meistens Programmierfehler

- **Ziel:** Der Programmierer soll seinen Fehler beheben
- **Beispiel:** Eine `ArrayIndexOutOfBoundsException` tritt auf, wenn der Programmierer den Index im Array falsch wählt
 - Dieser Fehler **muss** durch den Programmierer im Code behoben werden
 - Eine Behandlung mit **catch** ist **nicht sinnvoll** → das Programm sollte beendet werden

Geprüfte Ausnahmen signalisieren Fehler, die unter gewissen Umständen auftreten können

- **Ziel:** Die Anwendung soll sich von diesem Fehler „erholen“ können
- **Beispiel:** Eine `FileNotFoundException` tritt auf, wenn ein Programm mit einem falschen Dateipfad arbeitet
 - Diese Situation kann **zur Laufzeit** durch eine erneute Auswahl des Dateipfads **behooben werden**
 - Ein Beenden des Programms ist in diesem Fall nicht notwendig
- Manchmal werden ungeprüfte statt geprüfter Ausnahmen genutzt um Code übersichtlich zu halten

Erzeugen von Ausnahmen

- Eigene Exceptionklassen erstellen

Vorgehensweise

- Oberklasse wählen
 - **Geprüfte** oder **ungeprüfte** Ausnahme?
 - Gibt es bereits eine **passende Klasse**, von der geerbt werden kann?
- Aussagekräftigen Namen wählen
- Konstruktoren implementieren
 - Z.B. Konstruktor mit Nachricht
 - Die Nachricht wird im **Stack Trace** ausgegeben und kann mit **getMessage()** abgerufen werden.
 - Z.B. Konstruktor mit Nachricht und einer Ausnahme
 - Dies ermöglicht es eine andere Ausnahme aufzufangen und z.B. als **LevelException** weiterzugeben
 - Die Original-Ausnahme wird im **Stack Trace** ausgegeben und kann mit **getCause()** abgerufen werden.

Beispiel: Erzeugen einer eigenen Exceptionklasse

```
package game;  
  
public class LevelException extends Exception {  
  
    public LevelException(String message) {  
        super(message)  
    }  
  
    public LevelException(String message, Throwable cause) {  
        super(message, cause)  
    }  
  
}
```

Zusammenfassung

- Ausnahmen in Java
- Varianten von Ausnahmesituationen
- Ungeprüfte Ausnahmen
 - Ignorieren einer Ausnahme
 - Fangen einer Ausnahme
- Geprüfte Ausnahmen
 - Weiterleiten einer Ausnahme
- Vererbungshierarchie von Ausnahmen
- Die try-catch-Behandlung
 - Aufbau und Regeln
 - Mehrere Ausnahmen
 - Die finally-Klausel
 - Schließen von Ressourcen
- Java Stack Trace
- Catch or throws?
- Ausnahmen erzeugen
 - Ausnahmen werfen mit **throw**
 - Auswahl einer passenden Ausnahme
 - Geprüfte oder ungeprüfte Ausnahmen?
 - Eigene Exceptionklassen erstellen