universität innsbruck

Exceptions

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Motivation

- In einem ablaufenden Programm können Fehler auftreten.
  - Logische Fehler im Programm
  - Fehlerhafte Bedienung
  - Probleme im Java-Laufzeitsystem
  - Technische Fehler (Speicherplatz, Klasse von Laufzeitumgebung nicht gefunden)
- Programme müssen auf Fehlersituationen vorbereitet werden und **kontrolliert** darauf reagieren.
- In C:
  - Kein einheitliches Vorgehen.
  - Fehler können über Rückgabewerte erkannt werden.
- In Java:
  - Ausnahmebehandlung (Exception-Handling)
  - Exceptions sind ein Sprachmittel zur kontrollierten Reaktion auf Laufzeitfehler!

Exception-Handling

- Exceptions (Ausnahmen) sind ein Sprachmittel zur kontrollierten Reaktion auf Laufzeitfehler!
- Sofern eine Exception auftritt, wird durch das Exception-Handling der normale Programmfluss verlassen.
  - Die Kontrolle geht an den Mechanismus der Ausnahmebehandlung über.
  - Im Exception-Handler wird die Ausnahme behandelt.
  - Nach der Behandlung wird der Kontrollfluss wieder an das Programm zurückgegeben.
- Das Auslösen einer Exception wird als Werfen bezeichnet.
- Das Behandeln einer Exception wird als Fangen bezeichnet.

Vorteile durch Exception-Handling

- Mechanismus zur strukturierten Behandlung von Ausnahmesituationen.
- Code für den regulären Programmablauf und die Fehlerbehandlung wird getrennt.
- Exceptions können entlang der Aufrufhierarchie propagiert werden. Dadurch kann die Ausnahme an der Stelle behandelt werden, die am Besten dafür geeignet ist.
  - Eine Exception wird an bestimmten Punkten im Programm geworfen.
  - An einer anderen Stelle im Programm steht Code zum Fangen potenzieller Ausnahmesituationen.
- Bestimmte Exceptions können nicht ignoriert werden und es müssen Maßnahmen zur Behandlung getroffen werden.

Exception-Handling in Java

- Information über die Exception wird in ein spezielles Objekt verpackt, welches ein Exemplar der Klasse `Throwable` oder einer Subklasse von `Throwable` ist.
- Durch die `throw`-Anweisung können Exceptions explizit ausgelöst werden.
- Die Ausnahme kann an einer bestimmten Stelle (äußerer Block, aufrufende Methode etc.) in einer `catch`-Klausel abgefangen werden.
- Grundkonstrukt: `try`-Anweisung

try-Anweisung

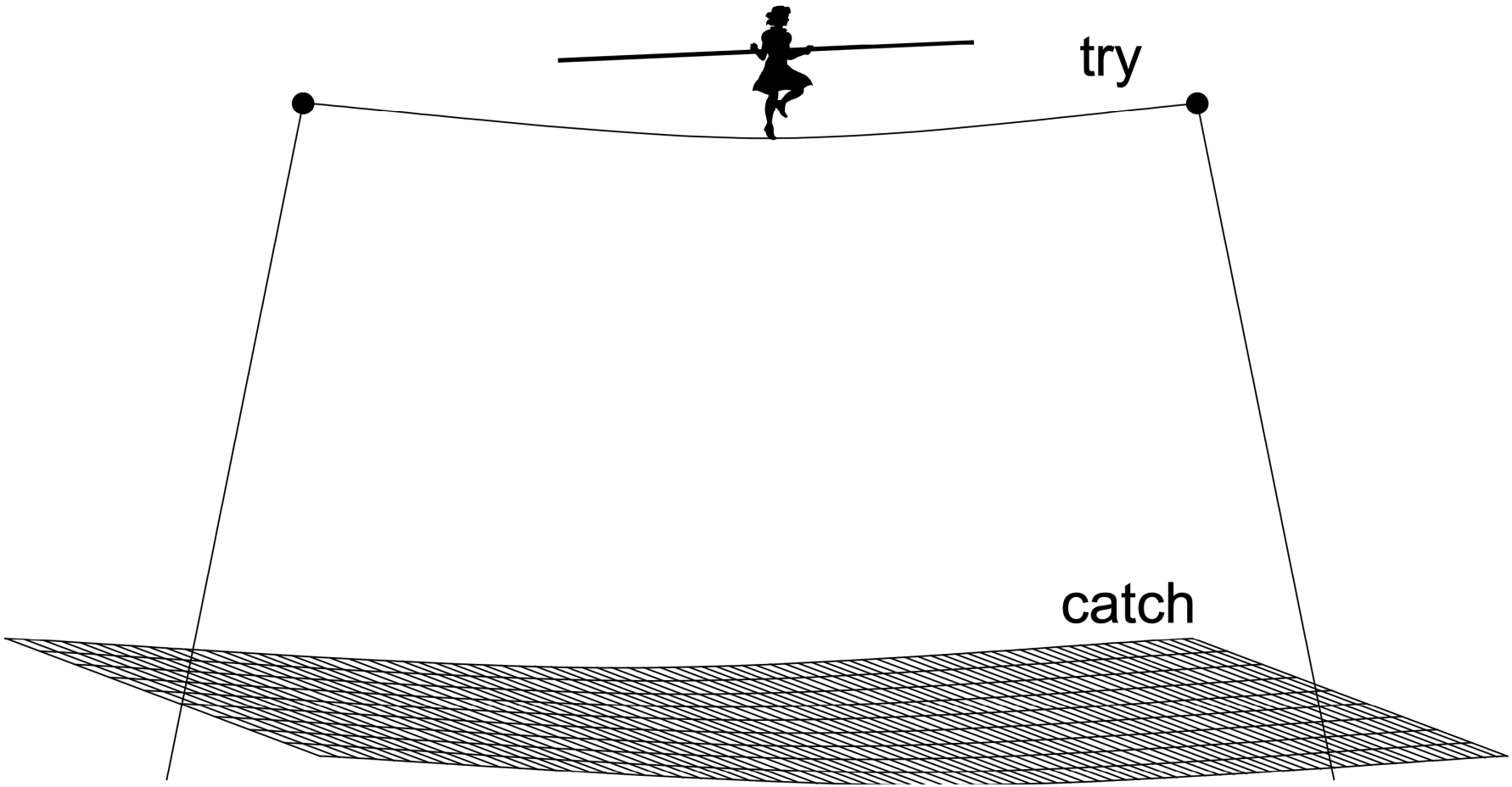


Abbildung übernommen aus „Java als erste Programmiersprache“, Goll & Heinisch

try-Anweisung (Grundform)

```
try {
  …
  …
} catch (ExceptionType1 e) {
  …
  …
} catch (ExceptionType2 e) {
  …
  …
}
```

regulärer Code, in dem Fehler auftreten kann

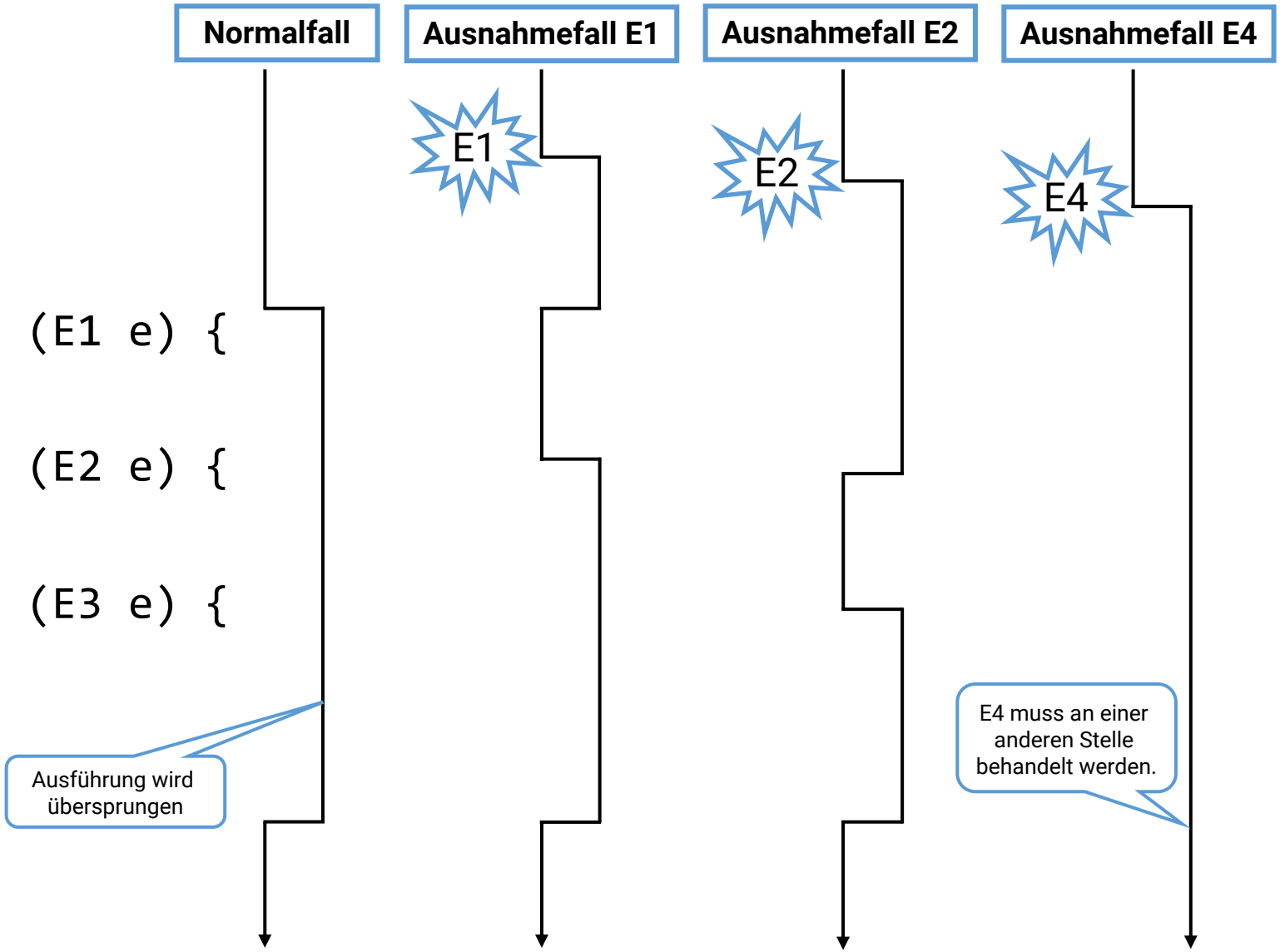
Code für Fehlerbehandlung der Exception e vom Typ ExceptionType1

Code für Fehlerbehandlung der Exception e vom Typ ExceptionType2

- Nach dem `try`-Block folgen die `catch`-Klauseln.
- Es können beliebig viele `catch`-Klauseln verwendet werden.
- Eine `catch`-Klausel hat genau einen Parameter (Exception-Parameter).
- Der Typ des Exception-Parameters bestimmt, welche Exceptions durch diese `catch`-Klausel gefangen werden können.
- Mindestens eine `catch`-Klausel oder ein `finally`-Klausel muss vorhanden sein.

Ablauf (einfach)

```
try {
  …
  …
  …
} catch (E1 e) {
  …
} catch (E2 e) {
  …
} catch (E3 e) {
  …
}
…
```



Ablauf (1)

- Wird eine Ausnahme vom Typ `E` geworfen:
  - Dann wird eine entsprechende `catch`-Klausel für den Typ `E` gesucht.
  - Statt `E` kann auch eine Klausel mit einer Superklasse von `E` verwendet werden, da `E` dort substituiert werden kann!
- Reihenfolge der `catch`-Klauseln ist entscheidend.
  - Eine Ausnahme wird der Reihe nach mit den `catch`-Klauseln verglichen.
  - Ausgeführt wird die erste `catch`-Klausel, zu der die Ausnahme kompatibel ist.
  - Nachfolgende `catch`-Klauseln werden ignoriert.
  - Daher muss die Ordnung der `catch`-Klauseln beachtet werden!
    - Von speziell zu allgemein!
  - Wenn zuerst eine allgemeinere Ausnahme angegeben wird (und dann eine speziellere) kommt es zu einem Compiler-Fehler.

Ablauf (2)

- Falls keine `catch`-Klausel gefunden wird, wird in den äußeren `try`-Anweisungen gesucht.
  - In verschachtelten `try`-Anweisungen
  - Entlang der Methodenaufrufkette (wird noch besprochen)
- Wird nie eine `catch`-Klausel gefunden (auch nicht in `main`), dann bricht das Programm ab.

Termination Model / Resumption Model

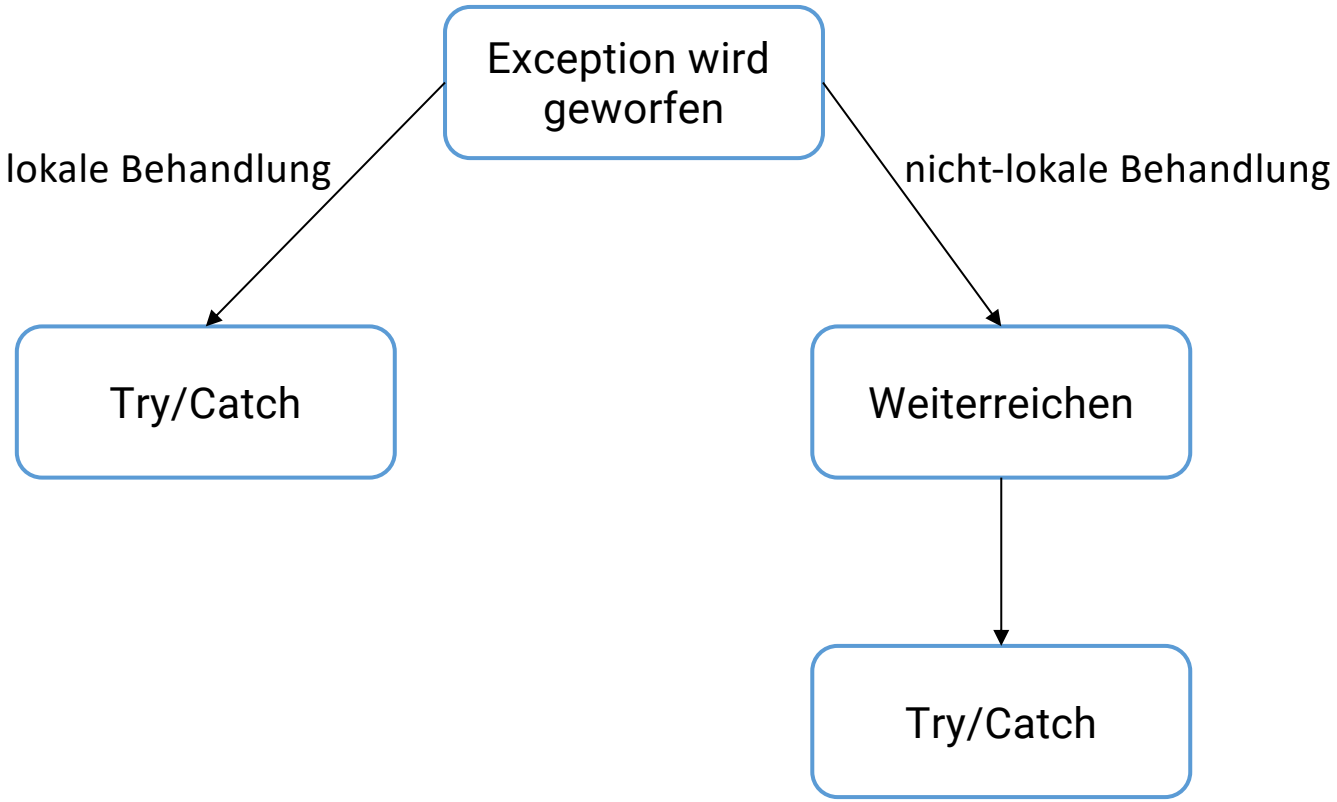
Termination Model

- Der Kontrollfluss kehrt nicht mehr an die Stelle zurück, an der die Ausnahme aufgetreten ist (kehrt an Position nach try-Anweisung zurück).
- Die Ausnahmebehandlung in Java folgt dem **Termination Model**.

Resumption Model

- Bei diesem Modell erfolgt eine Rückkehr an die Aufrufstelle, d.h. das Programm setzt an der Stelle der Ausnahme fort.
- Das **Resumption Model** wird nicht in Java verwendet.

Behandlung von Exceptions



Beispiel lokale Behandlung

```
public class ExceptionTest {
    public static void main (String[] args) {
        try {
            int x = Integer.parseInt(args[0]);
            int y = Integer.parseInt(args[1]);
            System.out.println(x / y);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Please provide at least two command-line parameters!");
            // additional error handling
        } catch (NumberFormatException e) {
            System.out.println("Please provide two integers!");
            // additional error handling
        } catch (ArithmeticException e) {
            System.out.println("Division by zero!");
            // additional error handling
        }
    }
}
```

Kann in `parseInt()` auftreten.

```
java ExceptionTest 4 2
2
```

```
java ExceptionTest 1 z
Please provide two integers!
```

```
java ExceptionTest
Please provide at least two command-line parameters!
```

Hierarchie von Exceptions

- Ausnahmen sind in Java Objekte.
  - Enthalten Informationen über die aufgetretenen Fehler
  - Bieten Methoden an, um auf die Informationen zuzugreifen
- Alle Ausnahmen sind in einer Klassenhierarchie gruppiert.
  - Dient zur Differenzierung
  - Eine catch-Klausel, die eine Klasse E behandeln kann, kann auch alle Unterklassen von E behandeln.
- Oberste Klasse ist `java.lang.Throwable`
  - Enthält Methoden zur Fehleranalyse
    - `String getMessage()` (Text bei Ausnahme)
    - `String toString()` (Klassenname + getMessage())
    - `void printStackTrace()`
  - Konstruktoren (leer, mit Fehlermeldung,…)
- Unterklassen
  - `java.lang.Error`
    - Irreparable Fehler
  - `java.lang.Exception`
    - Fehler, die sinnvoll behandelt werden können

Arten von Ausnahmen

Unchecked Exceptions

- Werden automatisch ausgelöst (z.B. illegale Instruktionen)
- Exemplare, die aus den Klassen `Error`, `RuntimeException` sowie davon abgeleiteten Klassen erzeugt wurden.
- Können abgefangen werden, müssen es aber nicht!

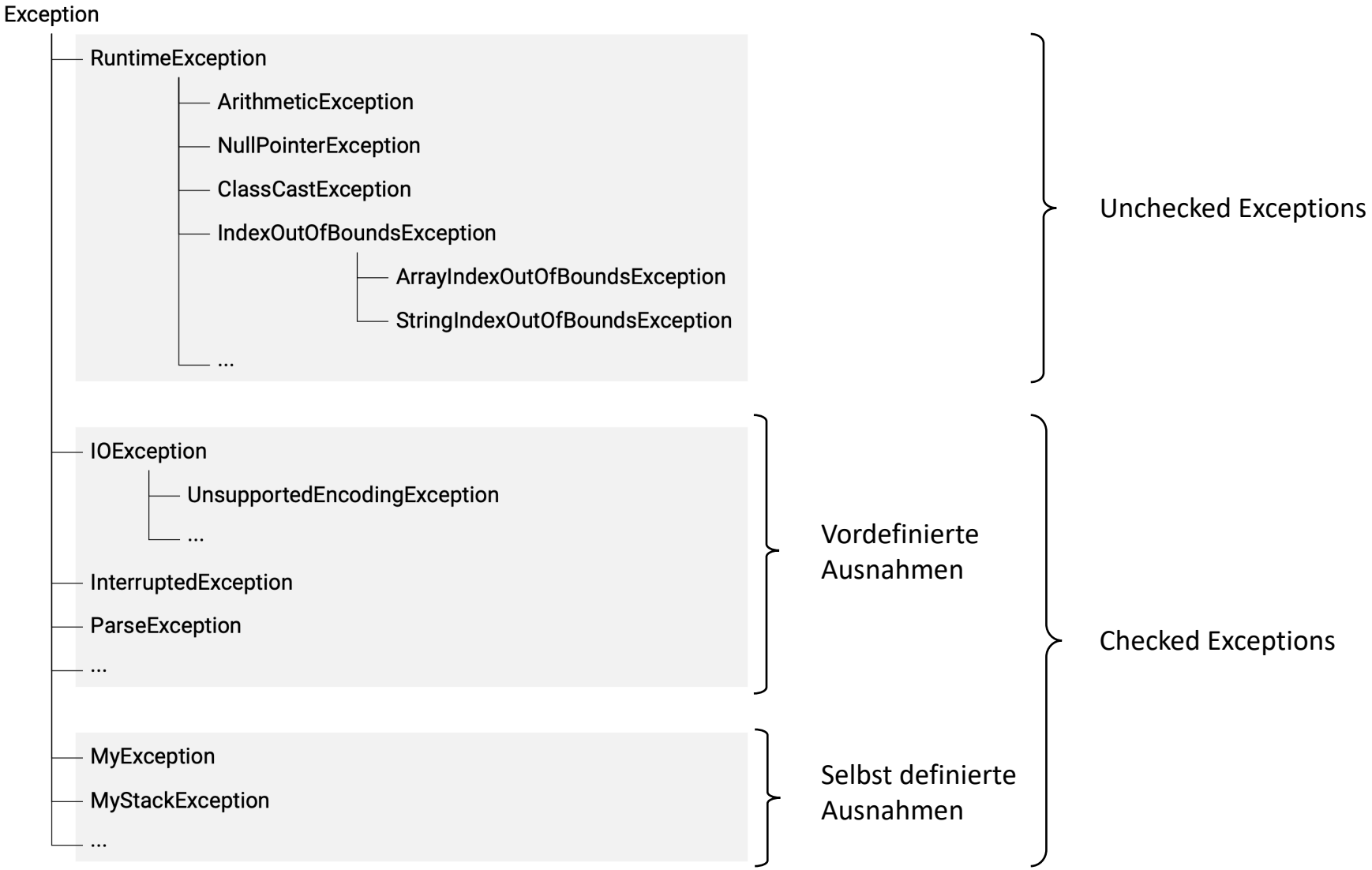
Checked Exceptions

- Müssen explizit beim Programmieren mit der `throw` – Anweisung ausgelöst werden.
- Alle Exceptions bis auf Exemplare, die von `Error`, `RuntimeException` sowie davon abgeleiteten Klassen entstammen, sind Checked Exceptions.
- Der Compiler überprüft, ob die Exceptions behandelt werden.

Error-Klasse

- `Error` sind Fehler, die mit der JVM in Verbindung stehen.
- Beispiel:
  - `OutOfMemoryError:` Zu wenig Speicher für die JVM bei der Objekterzeugung vorhanden
- Da die Fehler „abnormales“ Verhalten anzeigen, müssen sie auch nicht mit einer `catch`-Klausel aufgefangen werden.
  - Wie RuntimeException können sie aber abgefangen werden.
- Ein Auffangen macht wenig Sinn.
  - Wie sollte auf solche Fehler reagiert werden?

Exception-Hierarchie



Eigene Exception erstellen

- Exceptions müssen direkt oder indirekt von der Klasse Throwable erben.
  - Eigene Exceptions sollten immer nur direkt oder indirekt von der Klasse Exception erben.
- Von der Klasse Exception ableiten:

```
public class MyException extends Exception {…}
```

- Wenn eine Nachricht mitgegeben werden soll:

```
public class MyException extends Exception {

    public MyException() {
    }

    public MyException(String specialInfo) {
        super(specialInfo);
    }
}
```

Exceptions auslösen

- Auslösen („Werfen“) einer Ausnahme mit der `throw`–Anweisung:

```
throw new MyException();
throw new MyException("Falsche Eingabe");
```

- Die `throw`-Anweisung unterbricht den laufenden Code sofort!

- Siehe Beispiel:


src/at/ac/uibk/pm/exceptions/stack/StackException.java


src/at/ac/uibk/pm/exceptions/stack/resourceloader/ResourceLoader.java

Weiterreichen von Exceptions (1)

- Methoden müssen ihre Ausnahmen nicht lokal behandeln, sondern können sie auch weiterreichen.
- Exception wird an aufrufende Methode weitergereicht.
- Immer dann verwenden, wenn die Exception von der aufrufenden Methode sinnvoller behandelt werden kann (z.B. Parameter-Neueingabe).
- Die throws-Klausel einer Methode weißt darauf hin welche Exceptions geworfen bzw. weitergereicht werden könnten.
  - Typischerweise werden nur Checked Exceptions in der throws-Klausel angegeben
- Form

```
[Zugriffsmodifikatoren] Rückgabetyp Methodenname([Parameter]) throws
  ExceptionType1, ExceptionType2, …
```

- Beispiele

```
int test1(int x) throws IOException {…}
int test2() throws FileNotFoundException, UnsupportedEncodingException {…}
```

Weiterreichen von Exceptions (2)

- Ausnahmen können über mehrere Aufrufe durchgereicht werden.
- Eine Methode kann nur Checked Exceptions werfen und weiterreichen, welche in der throws-Klausel angegeben wurden.
- Unchecked Exceptions können immer geworfen werden.
- Compiler stellt sicher, dass keine Checked Exception einer untergeordneten Methode übersehen wird.
- Weitergereichte Exceptions sollten möglichst spezifisch sein.
- Siehe Beispiel


src/at/ac/uibk/pm/exceptions/stack/ArrayStack.java


src/at/ac/uibk/pm/exceptions/stack/StackApplication.java

Beispiel nicht-lokale Behandlung

```java
public void push(Object element) throws StackFullException {
    if (position >= data.length) {
        throw new StackFullException(position, data.length);
    }
    data[position] = element;
    ++position;
}

public void push(Object[] elements) throws StackFullException {
    for (Object element : elements) {
        push(element);
    }
}
```

Exception wird hier geworfen.

Exception wird hier weitergereicht.

```java
public static void main(String[] args) {
    Stack stack = new ArrayStack(5);
    try {
        stack.push(args);
    } catch (StackFullException e) {
        System.out.println("Stack is full!");
    }
}
```

Exception wird hier behandelt.

src/at/ac/uibk/pm/exceptions/stack/ArrayStack.java

throws-Klausel (Überschreiben, Überladen)

- Überschreiben
  - throws-Klausel darf bei der Redefinition der Methode
    - Dieselben Exceptions wie Oberklasse auslösen
    - Exceptions spezialisieren
    - Exceptions weglassen
  - Bei der Redefinition dürfen in der throws-Klausel alle Typen aufscheinen, die zu irgendeinem Typ in der Signatur irgendeiner direkten oder indirekten Basisklassenmethode kompatibel sind (auch mehrere Subklassen für eine Exceptionklasse).
- Überladen
  - throws-Klauseln überladener Methoden sind völlig unabhängig.
  - Unterschiedliche throws-Klauseln reichen nicht aus, um Methoden zu überladen.
- Siehe externes Beispiel:


src/at/ac/uibk/pm/exceptions/stack/ArrayStack.java


src/at/ac/uibk/pm/exceptions/stack/StackApplication.java

Exception-Chaining

- Wenn Methoden Ausnahmen immer weiterreichen
  - Sehr lange throws-Klausel
  - Detailfehler auf niederer Ebene müssen an ganz anderer Stelle behandelt werden.
  - Änderungen in der throws-Klausel würden viele Veränderungen nach sich ziehen.
- Lösung – Exception-Chaining
  - Mehrere Ausnahmen untergeordneter Aufrufe werden aufgefangen.
  - Die Ausnahmen werden zusammengefasst und in einer neuen Ausnahme weitergegeben.

Exception-Chaining (Allgemeines Schema)

```
public void exceptionChaining() throws MyException
    try {
        …
    } catch(E1 ex) {
        throw new MyException(ex);
    } catch(E2 ex) {
        throw new MyException(ex);
    }
}
```

Ausnahme vom Typ `E2` wird in eine Ausnahme vom Typ `MyException` verpackt.

Chaining bei eigenen Ausnahmen

- Entsprechende Konstruktoren implementieren, um Grund für Exception rekonstruieren zu können (Cause).
  - Exception-Klasse gibt Beispiele für solche Konstruktoren:
    `Exception(String message, Throwable cause)`
    `Exception(Throwable cause)`
  - Beispiel für Konstruktor für die MyException-Klasse:

```
MyException(Throwable cause) {
    super(cause);
}
```

- Oder `initCause`-Methode verwenden:

```
…
catch (E1 ex) {
    MyException m = new MyException();
    m.initCause(ex);
    throw m;
}
```

- Siehe externes Beispiel:

src/at/ac/uibk/pm/exceptions/stack/resourceloader/ResourceLoader.java

Nachverfolgung von Ausnahmen: StackTrace

- Methode `printStackTrace` gibt Aufrufstack aus, erbt von `Throwable` (Superklasse von `Error` und `Exception`).
- Damit kann man den genauen Verlauf der Exception (wie wurde weitergereicht?) inspizieren.

```
public static void main(String[] args) {
    Stack stack = new ArrayStack(5);
    try {
        stack.push(args);
    } catch (StackFullException e) {
        System.out.println("Stack is full!");
        e.printStackTrace();
    }
}
```

```
java StackApplication a b c d e f
Stack is full!
at.ac.uibk.pm.exceptions.stack.StackFullException: Expected a stack with less than 5
elements but got stack with 5 elements.
   at at.ac.uibk.pm.exceptions.stack.ArrayStack.push(ArrayStack.java:41)
   at at.ac.uibk.pm.exceptions.stack.ArrayStack.push(ArrayStack.java:56)
   at at.ac.uibk.pm.exceptions.stack.StackApplication.main(StackApplication.java:18)
```

src/at/ac/uibk/pm/exceptions/stack/StackApplication.java

Multi-Catch (1)

- Manchmal sollen mehrere Ausnahmetypen gleichartig behandelt und dafür nur eine catch-Klausel verwendet werden.
- Fangen mehrerer Ausnahmen mit nur einem catch-Block möglich.
  - Wird als Multi-Catch bezeichnet.
  - Bei der Aufzählung werden die einzelnen Ausnahmen mit | getrennt.
  - Die Variable (im unteren Beispiel e) ist implizit final.

```
try {
    …
} catch (MyException | OtherException e) {
    …
}
```

Hier können sowohl Ausnahmen vom Typ MyException als auch vom Typ OtherException gefangen werden.

Gemeinsame Behandlung der unterschiedlichen Ausnahmen

Multi-Catch (2)

- Die Abarbeitung erfolgt wie bei mehreren catch-Blöcken.
- Neben den Standard-Tests kommen neue Überprüfungen hinzu:
  - Kommt die exakt gleiche Ausnahme mehrfach in der Liste vor?
  - Gibt es Mehrdeutigkeit aufgrund von Mengenbeziehungen?

- Beispiel für fehlerhaftes Multi-Catch (Mengenbeziehung):

```
try {
    …
} catch (FileNotFoundException | IOException e) {
    …
}
```

Final Rethrow (1)

- Immer dann, wenn in einem catch-Block ein throw stattfindet, ermittelt der Compiler die im try-Block tatsächlich aufgetretenen Typen der in der catch-Klausel geprüften Ausnahmen.
  - Der im catch genannte Typ für das rethrow wird beim Weiterreichen nicht berücksichtigt.
  - Statt dem gefangenen Typ wird der Compiler den durch die Codeanalyse gefundenen Typ beim rethrow melden.

  → Allgemeine Exception fangen, spezielle Exception werfen.

Final Rethrow (2)

```
public class Exception1 extends Exception {}
```

```
public class Exception2 extends Exception {}
```

```
public class Test {
    public void method1() throws Exception1 {…}
    public void method2() throws Exception2 {…}
    …
    public void method3() throws Exception1, Exception2 {
        try {
            method1();
            method2();
        } catch (Exception e) {
            // do something
            throw e;
        }
    }
}
```

Konstruktoren und Ausnahmen

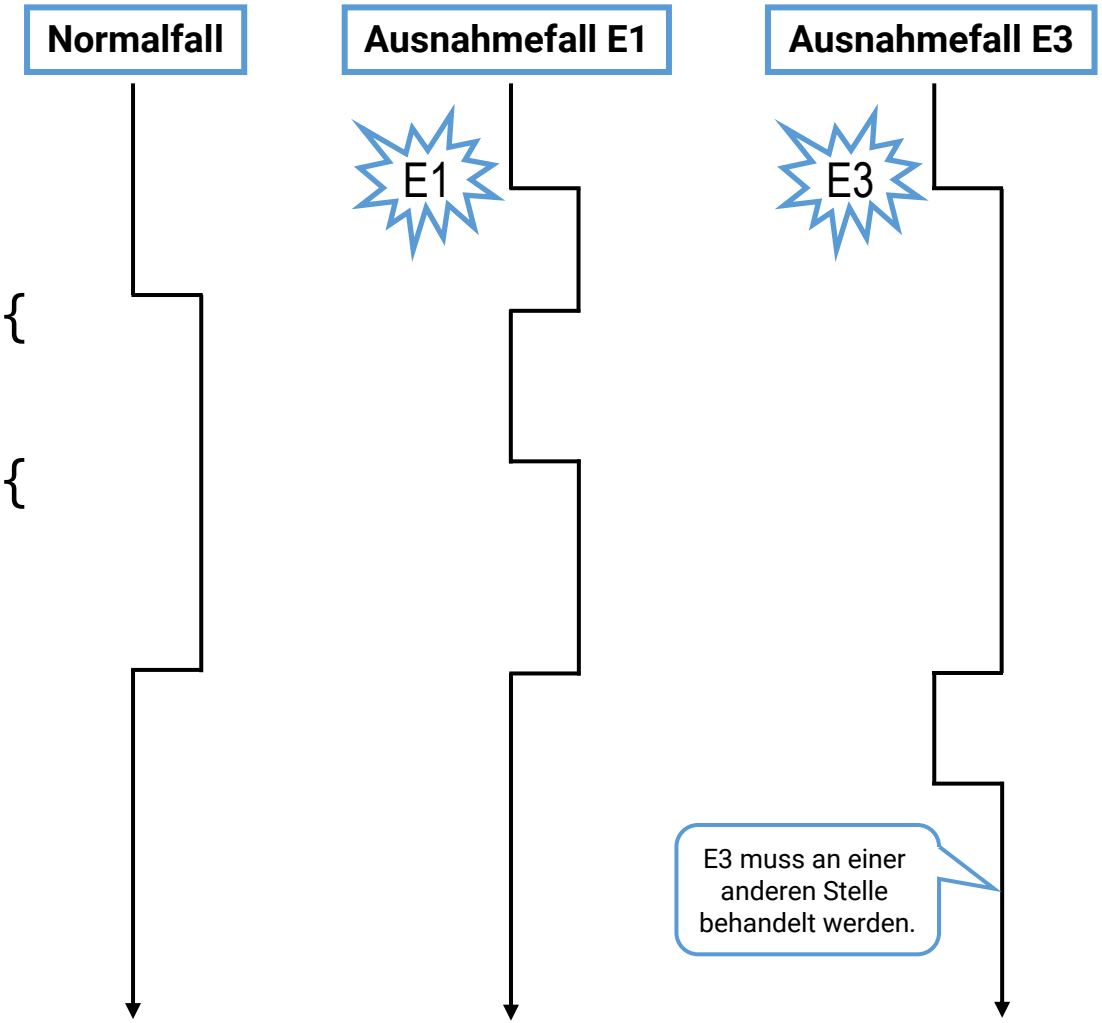
- Ausnahmen können auch in Konstruktoren verwendet werden.
  - Fehler beim Anlegen eines Objekts sollten vermieden werden.
  - Objekt könnte in einem inkonsistenten Zustand sein.
- Der Konstruktor sollte die Ausnahme weiterreichen (`throws`).
- Aufrufende Methode kann dann entsprechend auf die gescheiterte Erzeugung des Objekts reagieren.
- Vorsicht bei `finally` → wird immer aufgerufen

finally

- Die `try`-Anweisung kann nach den `catch`-Blöcken **eine** `finally`-Klausel enthalten.
- Die Anweisungen im `finally`-Block werden immer als Abschluss der `try`-Anweisung ausgeführt, egal ob eine Ausnahme auftrat oder nicht.
- Hat den Zweck Arbeiten, die im `try`-Block begonnen wurden, sauber abzuschließen.
- Beispiel
  - Im `try`-Block wird eine Datei geöffnet.
  - Die Datei muss wieder geschlossen werden (egal ob bei der Verarbeitung der Daten ein Fehler auftritt oder nicht).
  - Hinweis: Ab Java 7 gibt es noch eine andere Möglichkeit. Diese wird im Foliensatz über Streams noch ausführlich besprochen.
- Auch bei verschachtelten `try`-Anweisungen möglich!

finally (Abarbeitung)

```
try {
   …
   …
} catch (E1 e) {
   …
} catch (E2 e) {
   …
} finally {
   …
}
…
```



src/at/ac/uibk/pm/exceptions/stack/FinallyApplication.java

Empfehlungen Exception-Handling (1)

- Art der Ausnahme
  - Wenn ein Aufrufer eine außergewöhnliche Situation behandeln kann, so sollte eine Checked Exception geworfen werden.
  - Ist ein Aufrufer nicht in der Lage, eine Fehlersituation zu korrigieren, so sollte eine Unchecked Exception verwendet werden.
- Behandlung von Ausnahmen
  - Behandle auftretende Ausnahmen, wenn dies sinnvoll möglich ist.
  - Propagiere im Zweifelsfall Ausnahmen an höhere Aufrufebenen weiter.
    - Dabei sollte aber immer ein möglichst aussagekräftiger Ausnahmetyp gewählt werden!
- Verwendung der `try`-Anweisung
  - `try`-Anweisung findet sich meist in übergeordneten Methoden.
  - Je mehr über den aktuellen Programmzustand vorhanden ist, desto besser lässt sich angemessen auf einen Fehler reagieren.

Empfehlungen Exception-Handling (2)

- Standard Exceptions der Java API sollten bevorzugt werden.
- Ausnahmen werden oft für Vorbedingungen verwendet.
  - Vorbedingungen beziehen sich oft auf Parameter.
  - Die Parameter dürfen meist nicht beliebige Werte annehmen.
  - Ohne Überprüfung kann ein Objekt in einen falschen (inkonsistenten) Zustand geraten.

Exception	Mögliche Anwendung
`IllegalArgumentException`	Wert eines Parameters weist einen ungeeigneten Wert auf
`IllegalStateException`	Zustand eines übergebenen Objektes ist unzureichend
`NullPointerException`	Parameter weist den Wert `null` auf
`IndexOutOfBoundsException`	Indexwert ist außerhalb des Bereiches
`UnsupportedOperationException`	Hinweis auf eine fehlende Implementierung

Handling von unerwarteten Parametern

```
private static final int MAX_REPETITIONS = 5;

public static void printHelloWorld(int repetitions) {
    if (repetitions < 0) {
        throw new IllegalArgumentException();
    } else if (repetitions <= MAX_REPETITIONS) {
        for (int i = 0; i < repetitions; ++i) {
            System.out.println("Hello World!");
        }
    } else {
        throw new IllegalArgumentException();
    }
}
```

Fail Fast

```java
private static final int MAX_REPETITIONS = 5;

public static void printHelloWorld(int repetitions) {
    if (repetitions < 0 || repetitions > MAX_REPETITIONS) {
        throw new IllegalArgumentException();
    }
    for (int i = 0; i < repetitions; ++i) {
       System.out.println("Hello World!");
    }
}
```

- Vorher:
  - Alle Zweige sind verbunden und müssen verstanden werden.
- Nachher:
  - Lesbarer Code
  - Zuerst wird der Parameter validiert und anschließend wird der Hauptteil der Methode abgearbeitet.

Explain Cause in Message (1)

```
private static final int MAX_REPETITIONS = 5;

public static void printHelloWorld(int repetitions) {
    if (repetitions < 0) {
        String error = String.format("Expected repetitions >= 0 but got %s.", repetitions);
        throw new IllegalArgumentException(error);
    }
    if (repetitions > MAX_REPETITIONS) {
        String error = String.format("Expected repetitions <= %d but got %s.",
            MAX_REPETITIONS, repetitions);
        throw new IllegalArgumentException(error);
    }
    for (int i = 0; i < repetitions; ++i) {
       System.out.println("Hello World!");
    }
}
```

Explain Cause in Message (2)

- Vorher:
  - Keine detaillierten Informationen über die aufgetretene Exception.
- Nachher:
  - Grund der Exception wird klarer gemacht.
  - Alle Parameter und Felder, welche zum Auftreten der Exception beigetragen haben, wurden kommuniziert.
  - Grundschema:
    - Was wurde erwartet?
    - Was wurde übergeben?
    - Was liegt im Objekt vor?

Verwendung von eigenen Exceptions

- Falls keine der Standard-Exceptions zutrifft, können eigene Exceptions erstellt werden.
  - Subklassen von `Exception`, `RuntimeException` oder einer Subklasse
- Trade-Off
  - Wenige Einzelklassen: schlechte Differenzierung, aber kurze throws-Klauseln oder catch-Listen
  - Viele Einzelklassen: feingranulare Behandlung, aber lange throws-Klauseln oder catch-Listen
- Hierarchien aufbauen
  - Spezielle Klassen für differenzierte Fehlersituationen
  - Spezielle Klassen nach Fehlerart gruppieren und mit gemeinsamen Basisklassen versehen
  - Detaillierte Ausnahmen anbieten
    - Ausnahmen beinhalten weitere Objektvariablen.
    - Beim Werfen der Ausnahme werden diese Objektvariablen belegt (z.B. mit den falschen Parameterwerten etc.).
    - Damit stehen bei der Fehlerbehandlung mehr Informationen zur Verfügung!
- Wann soll von `RuntimeException` abgeleitet werden?
  - Wenn das aufgezeigte Problem von vornherein vermieden werden könnte!

Verwendung der verschiedenen Basisklassen

- Abfangen in `catch`-Klauseln
  - `Error` – nein!
  - `Exception` – nein (würde alle Subklassen abfangen), nur Subklassen davon!
  - `RuntimeException` – nein (zu allgemein), meist ein Indiz für einen Fehler im Programm, der behoben werden sollte! (siehe clean code Tipp nächste Folien)
- Werfen mit `throw`-Anweisung
  - `Error` – nein, im Allgemeinen der JVM bzw. Assertions vorbehalten!
  - `Exception` – nein (zu allgemein), nur mit einer entsprechenden Subklasse!
  - `RuntimeException` – nein (zu allgemein), nur mit einer entsprechenden Subklasse!

Beispiel Runtime-Probleme

```
public boolean equals(Object other) {
    if (this == other) {
        return true;
    }

    Product otherProduct = (Product) other;
    return productID == otherProduct.productID;
}
```

Always Check Type Before Cast

```java
public boolean equals(Object other) {
    if (this == other) {
        return true;
    }
    if (!(other instanceof Product otherProduct)) {
        return false;
    }
    return productID == otherProduct.productID;
}
```

- Vorher:
  - ClassCastException (Runtime-Exception) möglich
- Nachher:
  - ClassCastException nicht mehr möglich – zur Laufzeit kann dieser Fehler nicht mehr auftreten

Ausnahmen - Don'ts (1)

- Eine Ausnahme sollte **niemals** grundlos ignoriert werden!
  - Fehler verschwinden nicht von selbst!
  - Programm wird möglicherweise nicht richtig funktionieren (inkonsistente Daten etc.).
  - Soll eine Exception ignoriert werden:
    - Soll der `catch`-Block eine Begründung mittels Kommentar enthalten
    - Soll die Variabe `ignored` benannt werden.

```
try {
    …
} catch (MyException e) {
}
```

```
try {
    …
} catch (MyException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Ausnahmen - Don‘ts (2)

- Ausnahmen sollten niemals Kontrollstrukturen ersetzen!!!
  - z.B. bei einem Arraydurchlauf nicht auf die Länge überprüfen, sondern auf Ausnahme warten und dann abfangen.
- Rückgabe von `null` statt einer Ausnahme im Fehlerfall
  - Ausnahmen müssen behandelt werden, `null` kann ignoriert werden!
  - Rückgabe von `null` ist nur in sehr wenigen Fällen sinnvoll.
  - Muss in der aufrufenden Methode separat behandelt werden.

Assertions

Assertions allgemein

- Eine Assertion (Zusicherung) ist ein boolescher-Ausdruck, der immer zutreffen muss.
  - Assertions werden beim Programmablauf von der JVM überwacht.
  - Programmabbruch, wenn eine Assertion **nicht** zutrifft (Error)!
- Assertions sind einfache Anweisungen.
  - Form

    ```
    assert Ausdruck1;
    assert Ausdruck1 : Ausdruck2;
    ```

  - `Ausdruck1` muss `true` liefern, ansonsten wirft die Assertion eine Exception vom Typ `AssertionError`.
  - `Ausdruck2` ist optional und wird nur ausgewertet, wenn `Ausdruck1` `false` ist.
  - `Ausdruck2` wird dem Konstruktor der Klasse `AssertionError` übergeben, um den Fehler genauer zu beschreiben.
  - Beispiele

    ```
    assert a <= b && b <= c;
    assert x > 0 : "x must be positive!";
    ```

Arbeitsweise von Assertions

- Assertions werden zur Laufzeit ausgewertet.
- Wenn das Ergebnis `false` ist, stoppt das Programm mit einem `AssertionError`.
- Die nachfolgenden Anweisungen werden nicht mehr ausgeführt.
- Bei Abbruch wird Information über den Ort der gescheiterten Assertion ausgegeben.

Aktivierung von Assertions

- Assertions sind standardmäßig deaktiviert.
- Assertions kosten Rechenzeit.
- Sie können zur Laufzeit wahlweise aktiviert werden.
  - Deaktiviert sind sie automatisch.
  - Programm muss nicht neu übersetzt werden.
- Aktivierung im Programm Test (enable assertions)

```
java –ea Test
```

- Deaktivierung (default, muss nicht angeführt werden; disable assertions)

```
java –da Test
```

- In Eclipse
  - Unter `Run->RunConfigurations` Reiter `Arguments` auswählen.
  - `-ea` im Feld `Vm arguments` angeben.
- In Intellij
  - Unter `Run -> Edit Configurations…`
  - Klick auf `Modifiy options` und `Add VM options`
  - `-ea` im Feld `VM options` eingeben
- Normalerweise nur **während der Entwicklungszeit** aktiviert!

Selektive Aktivierung

- Assertions können auch selektiv auf der Ebene von Klassen und Packages getrennt aktiviert werden.
- Assertions in Klasse `classname` aktivieren:

```
-ea:classname
```

- Assertions in Package `packagename` (3 Punkte müssen angegeben werden):

```
-ea:packagename...
```

- Mehrere Schalter erlaubt:

```
java -ea:project2... -da:project2.datastore...
-ea:project2.datastore.Store  Test
```

Anwendung von Assertions

- Überprüfung von Parametern, die an nicht-öffentliche Methoden übergeben werden (nur falsch, wenn das eigene Programm fehlerhaft ist).
- Verwendung in Nachbedingungen, die am Ende einer Methode erfüllt sein müssen.
- Überprüfung von Schleifeninvarianten.
- Markierung von Codeblöcken die nicht erreicht werden sollten (Assertion mit `false` als Argument).
- Nicht für Bedingungen geeignet, die von irgendwelchen äußeren Einflüssen abhängen.
  - z.B. am Anfang einer öffentlichen Methode die Werte der Parameter überprüfen.

Beispiel (1)

```
private static int min(int a, int b) {
    int minimum = a <= b ? a : b;
    assert minimum <= a && minimum <= b;
    return minimum;
}

private static int max(int a, int b) {
    int maximum = a <= b ? a : b;
    assert maximum >= a && maximum >= b;
    return maximum;
}

public static void print(int a, int b) {
    System.out.printf("%d is less than or equal to %d%n", min(a, b), max(a, b));
}

public static void main(String[] args) {
    print(1, 1);
    print(5, 2);
    print(3, 9);
}
```

```
java AssertionApplication
1 is less than or equal to 1
2 is less than or equal to 2
3 is less than or equal to 3
```

Beispiel (2)

```java
private static int min(int a, int b) {
    int minimum = a <= b ? a : b;
    assert minimum <= a && minimum <= b;
    return minimum;
}

private static int max(int a, int b) {
    int maximum = a <= b ? a : b;
    assert maximum >= a && maximum >= b;
    return maximum;
}

public static void print(int a, int b) {
    System.out.printf("%d is less than or equal to %d%n", min(a, b), max(a, b));
}

public static void main(String[] args) {
    print(1, 1);
    print(5, 2);
    print(3, 9);
}
```

Programmierfehler

```
java -ea AssertionApplication
1 is less than or equal to 1
Exception in thread "main" java.lang.AssertionError
   at at.ac.uibk.pm.exceptions.AssertionApplication.max(AssertionApplication.java:12)
   at at.ac.uibk.pm.exceptions.AssertionApplication.print(AssertionApplication.java:17)
   at at.ac.uibk.pm.exceptions.AssertionApplication.main(AssertionApplication.java:22)
```

Quellen

- Bernhard Lahres, Gregor Rayman, Stefan Strich: **Objektorientierte Programmierung: Das umfassende Handbuch**, Rheinwerk Verlag, 5. Auflage, 2021
- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018