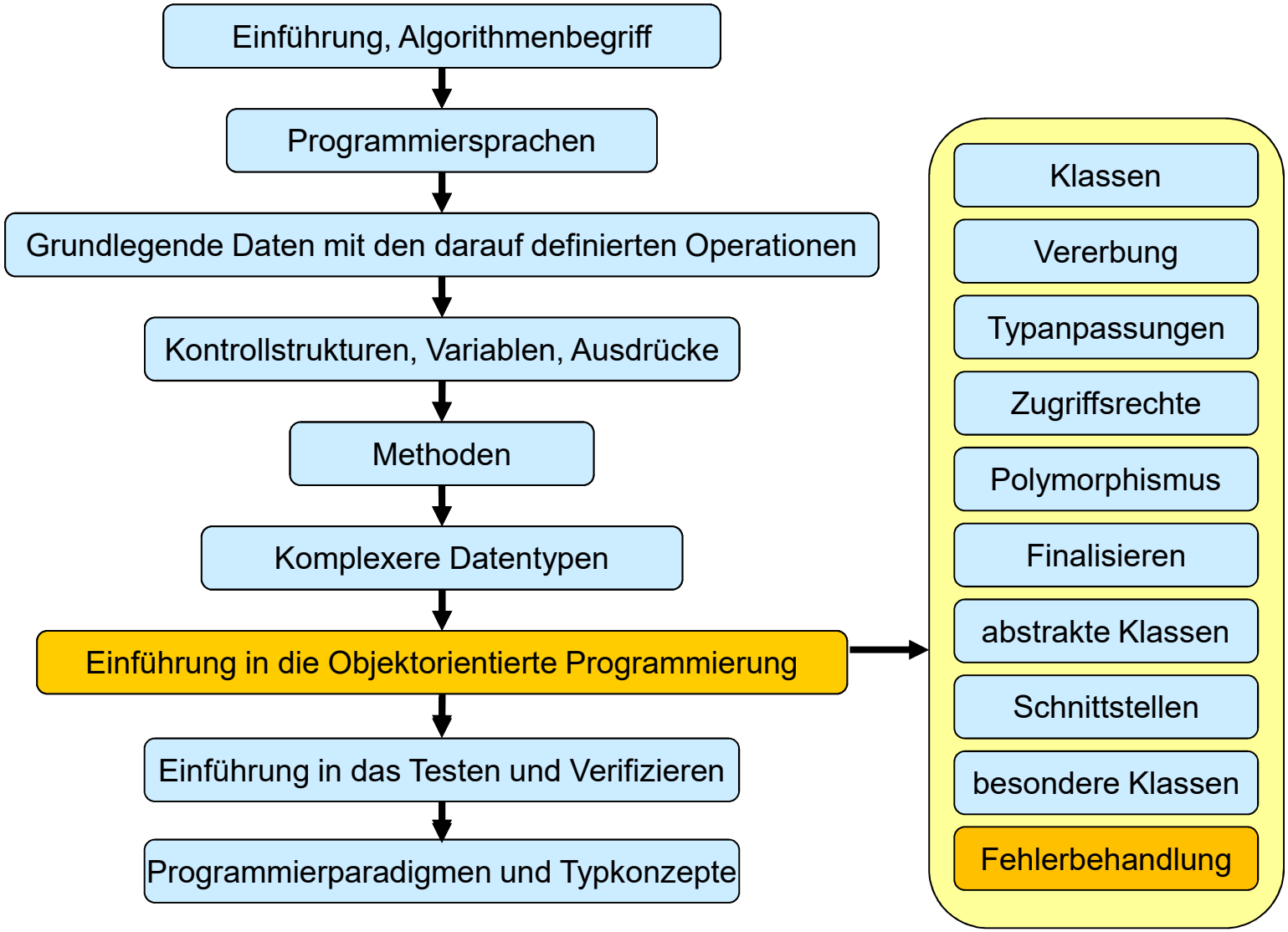
# Inhalt dieser Veranstaltung



# Fehler während der Ausführung des Programms

- In Programmen gibt es Situationen, die zu einem Laufzeitfehler führen können:

```
public class AusnahmeFall {
  public static void main(String[] args) {
    // was passiert, wenn weniger als 2 Argumente uebergeben werden?
    // was passiert, wenn args[0]/args[1] keine Darstellung eines int enthaelt?
    int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    // Was ist, wenn j==0 ist?
    System.out.println(i / j);
  }
}
```

- `java Ausnahmefall` (zu wenige Argumente)

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at AusnahmeFall.main(AusnahmeFall.java:7)
```

- `java Ausnahmefall true 1` (falsche Zahlendarstellung)

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "true"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:492)
        at java.lang.Integer.parseInt(Integer.java:527)
        at AusnahmeFall.main(AusnahmeFall.java:7)
```

- `java Ausnahmefall 1 0` (es kommt zur Division durch Null)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at AusnahmeFall.main(AusnahmeFall.java:11)
```

# Eigene Fehlerbehandlung?

- Ursprungscode

```
double[] methode1(int i, double value) {
  double[] a = new double[10];
  a[i] = 1.0 / value;
  return a;
}
```

- Mit eigener Fehlerbehandlung, die dazu nötig wäre. Wollen wir das???

```
double[] methode2(int i, double value) {
  if(System.verfuegbarerSpeicher() > 10*8) { // Speicher vorhanden?
    double[] a = new double[10];
    if(a != null) {                          // Referenz != null?
      if((i >= 0) && (i < 10)) {             // erlaubter Index beim Zugriff?
        if(value != 0.0) {                   // Division durch 0?
          a[i] = 1.0 / value;
          return a;
        } else {                             // Fehlerbehandlung 1
          System.out.println("Division durch 0");
          return null;
        }
      } else {                               // Fehlerbehandlung 2
         System.out.println("illegaler Index");
         return null;
      }
    } else {                                 // Fehlerbehandlung …
… }
```

# Oder doch eigene Fehlerbehandlung?

- Andere Kategorie von Fehlersituation: Fehlerfall z.B. aufgrund falscher Eingabe und dadurch nicht eingetroffener logischer Annahmen
- Der Normalfall (und nicht die Ausnahme) ist, dass ihre Programmlogik sicherstellt, dass nur mit fehlerfreien Werte für `monat` weiter gerechnet wird

```
public class AusnahmeFall {
  public static void main(String[] args) {
    …
    int monat;
    boolean fehlerFall;
    do {
      System.out.println("Geben Sie eine gueltige Monatsangabe ein.");
      monat = sc.nextInt();

      // gueltiger Wert?
      fehlerFall = (monat < 1) || (monat > 12);
      if(fehlerFall){
          System.out.println("Fehler in Monatsangabe");
      }
    } while(fehlerfall);
    … // ab hier muss monat immer zwischen 1 und 12 sein
  }
}
```

# Erkenntnis

- Es können potentiell an sehr vielen Stellen zur Laufzeit Fehler auftreten
- Die explizite Überprüfung **jeder** Fehlersituation würde ein Programm unnötig aufblähen und unlesbar machen

- Lösung: Unterscheidung zwischen

  – sinnvoller expliziter Überprüfung im Programmcode, wenn logische Bedingungen überprüft werden sollen ($1 \leq$ monat $\leq 12$ muss gelten; Teil der Programmlogik)

  – Überprüfung von Standardproblemsituationen, die aber (in normalen Programmen) nur sehr selten wirklich ein Fehler sein werden (Ausnahme; teilweise dann auch Programmierfehler) und deren Behandlung außerhalb der eigentlichen Programmlogik vorgenommen wird

# Zwischenstand

- An sehr vielen Stellen in einem Programm können potentiell Fehler auftreten (jeder Feldzugriff, jede Derefenzierung einer Referenz,…)
- Man muss solche allgemeinen potentiellen Fehlersituationen unterscheiden von Fehlerüberprüfungen der eigentlichen Programmlogik

**Reflektion**

- Geben Sie zu diesen beiden Kategorien jeweils ein eigenes Beispiel in einem Programm an.

# Exceptions in Java

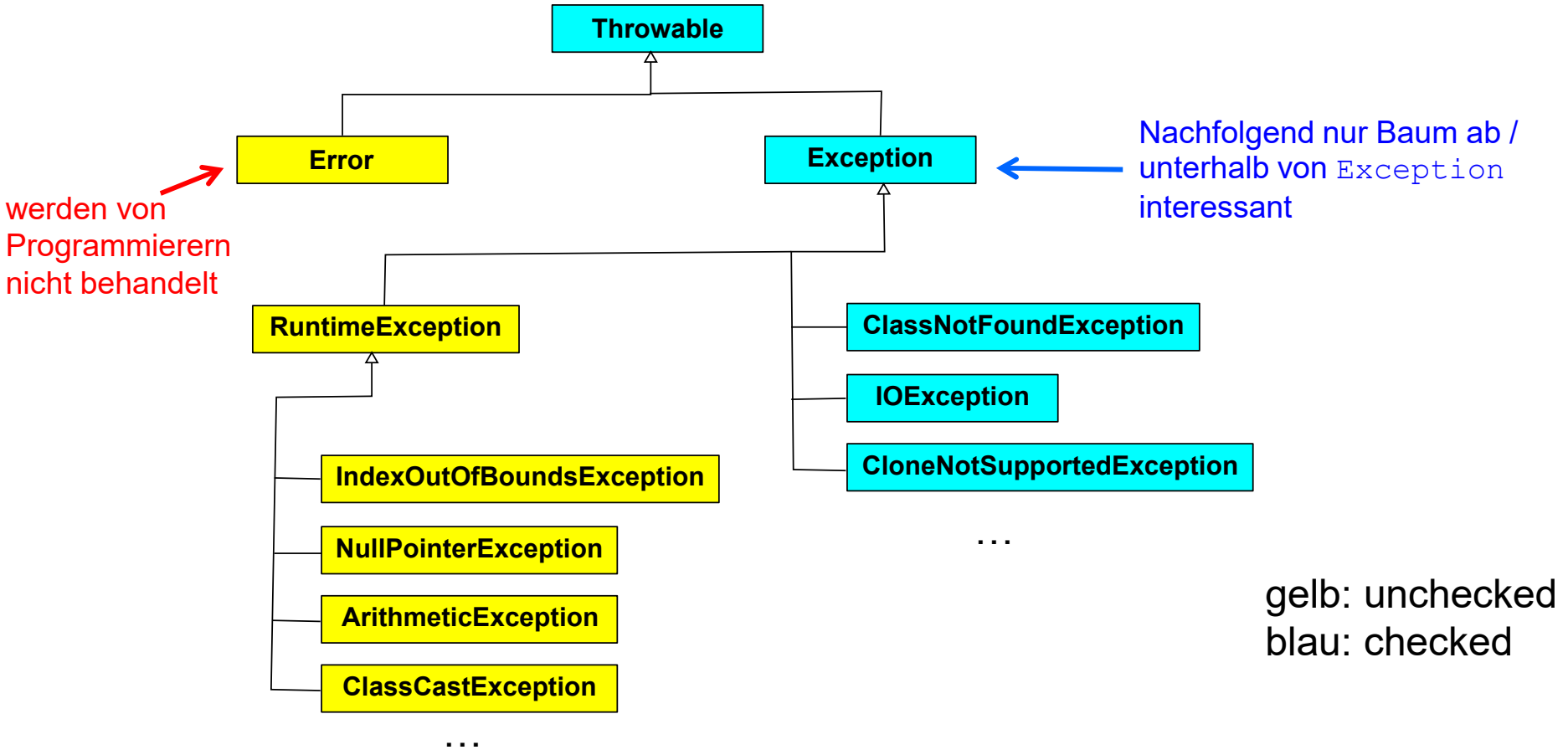
- Java kennt das Konzept der Ausnahmen / Exceptions
- Über Exceptions kann eine bestimmte Fehlerart signalisiert werden, worauf (ggfs. an einer anderen Stelle im Programm) reagiert werden kann
- Trennung von normaler Programmlogik und Ausnahmebehandlung

- Dazu gehört:
  1. Definition von Exception-Klassen für gleichartige Fehlerfälle
  2. Auslösen von Exceptions im Fehlerfall
  3. Behandlung aufgetretener Exceptions
  4. Propagieren von Exceptions

- Eine Klasse von Fehlerfällen wäre z.B. die Nutzung eines Index bei der Indizierung eines Feldes, der aber außerhalb der erlaubter Indexgrenzen dieses Feldes ist

# 1.Definition: Vordefinierte Exceptions

- Java kennt eine Reihe vordefinierter Fehlerarten, die in einer Klassenhierarchie beschrieben sind
- Ausschnitt daraus:

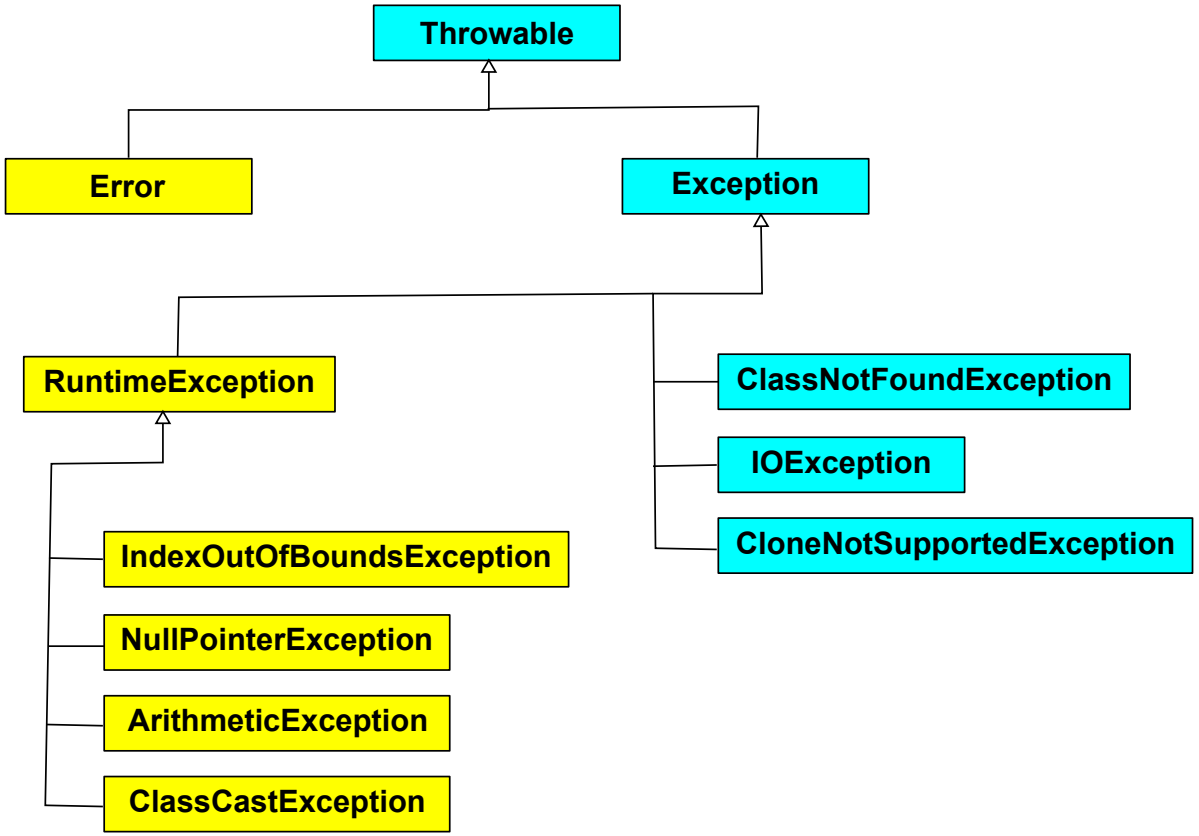


# checked / unchecked

- Eigentlich sollten alle Fehler innerhalb eines Programms (irgendwo) behandelt werden (Code-Sicherheit)
- Aber manche Fehlerarten können potentiell an so vielen Stellen auftreten, dass der Code dadurch unlesbar würde
- Beispiel: Indizierung von Feldern, Zugriff über Referenzen

- Kompromiss: definiere zwei Kategorieren von Fehlerarten
  - checked exceptions: Fehler dieser Art **müssen** im Programm behandelt werden (der Compiler überprüft das zur Übersetzungszeit). Dies sind i.Allg. Fehlerarten, die in den meisten Programmen nur an wenigen Stellen relevant werden können. Beispiel: Öffnen einer Datei, die nicht existieren könnte.
  - unchecked exceptions: Fehler dieser Art **können** im Programm behandelt werden, ansonsten werden sie vom Laufzeitsystem in einer standardisierten Weise behandelt (siehe Ausgabe des ersten Beispielprogramms). Dies ist die Kategorie von Fehlern, die an sehr vielen Stellen in Programmen auftreten könnte.

# Bedeutung der Hierarchie




allgemeine Fehlerart
(Generalisierung)

sehr spezielle Fehler
(Spezialisierung)

Eine Basisklasse umfasst damit auch alle Fehler daraus abgeleiteter Klassen

# Gemeinsame Eigenschaften dieser Klassen

- Die Klasse `Throwable` (und damit alle abgeleiteten Klassen) besitzt ein String-Attribut, das zu einer Instanz von `Throwable` (bzw. davon abgeleitet) mit `getMessage()` ausgelesen werden kann (siehe spätere Beispiele)

- Dieser String sollte eine textuelle Darstellung zum Fehler sein (sehr kurze Fehlerbeschreibung)

# 1.Definition: Eigene Exceptions

- Neben den vordefinierten Fehlerarten kann man auch eigene Fehlerarten definieren
- Dazu programmiert man eine Klasse, die von einer vorhandenen Fehlerklasse (`Exception` und unterhalb) abgeleitet ist
- Je nach Basisklasse erhält man damit selbst wieder eine checked oder unchecked Exception

- Anwendung: Mit einer solchen eigenen Fehlerklasse möchte man also eine spezielle Art von Fehler beschreiben, auf die man auch gezielt im Fehlerfall reagieren kann

- Die eigene Fehlerklasse sollte enthalten:
  - parameterlosen Konstruktor, der den Konstruktor der Oberklasse mit einem sinnvollen String aufruft (kurze Fehlerbeschreibung)
  - einen Konstruktor mit einem String-Argument, der den Konstruktor der Oberklasse mit diesem String aufruf (falls man selbst Basisklasse ist/wird oder der Programmierer einen Fehlertext mitgeben will)

# Beispiel

```
class MeineException extends Exception {

    // parameterloser Konstrukor
    MeineException() {
        // Aufruf des Konstruktors von Exception
        super("MeineException wurde geworfen");
    }

    // falls diese Klasse selber wieder abgeleitet werden soll
    MeineException(String message) {
        // Aufruf des Konstruktors von Exception
        super(message);
    }
}
```

# Zwischenstand

- In Java sind Fehlerarten in Exception-Klassen definiert
- Diese Klassen bilden eine Hierarchie
- Es gibt vordefinierte Fehlerklassen, aber auch eigene sind möglich
- Wichtige Unterscheidung: checked Exceptions müssen überprüft werden, unchecked können überprüft werden

**Reflektion**

- Überlegen Sie sich ein Szenario, wo es vorteilhaft ist eigene Fehlerklassen zu haben.

# 2.Auslösen: durch das System

- Tritt ein Fehler auf, so wird zu genau dieser Fehlerart eine **Instanz** einer passenden Exception-Klasse erzeugt und dem Laufzeitsystem zur Behandlung übergeben
- Fachbegriff: eine Exception wird geworfen („to throw“)
- Tritt an der gleichen Stelle später der gleiche Fehler auf, so wird eine weitere Exception der gleichen Art geworfen, aber mit einem **neuen** Objekt der Fehlerklasse

- Szenario 1 hier: ein Fehler wird durch das System erkannt
- Beispiel: falscher Indexausdruck bei Zugriff auf ein Feld
- Das System erzeugt eine Instanz der passenden Fehlerklasse `IndexOutOfBoundsException` (also eine möglichst spezielle Fehlerklasse) und wirft diese Exception

# 2.Auslösen: selbst auslösen

- Man kann an beliebiger Stelle im Kontrollfluss auch selbst eine (beliebige) Exception werfen

- Dazu notwendige Aktionen:
  - Erkennen der Fehlersituation
  - Instanziieren eines Objekts einer passenden (aber im Prinzip beliebigen) Fehlerklasse
  - das Werfen dieser Exception mit einer `throw`-Anweisung

# Beispiel

```
if((monat < 1) || (monat > 12)) {      // Fehlerfall

    // Exception-Objekt erzeugen
    MeineException e = new MeineException();

    // Exception werfen
    throw e;

    // was passiert jetzt hier?
}
```

oder kürzer

```
if((monat < 1) || (monat > 12)) {      // Fehlerfall

    // Exception werfen (hier mit eigenem Text)
    throw new MeineException("monat ist falsch");

    // was passiert jetzt hier?
}
```

# Eine Exception wird geworfen…

- Wird an einer Programmstelle eine Exception geworfen, so wird das zugehörige Exception-Objekt dem Java-Laufzeitsystem übergeben
- Die Abarbeitung des Programms wird an dieser Programmstelle **nicht** fortgesetzt

- Das Laufzeitsystem sucht nach dem zuständigen Exception-Handler
- Dieser wird aktiviert und mit dem Code des Exception-Handlers die Programmausführung fortgesetzt (gleich mehr dazu)

- Es gibt keine Möglichkeit, später an der Stelle des Auftretens der Fehlers die Programmausführung fortzusetzen

# 3.Behandlung

- Idee: trenne Fehlerbehandlung vom normalen Code (weil Ausnahme)
- Ein Programmcode, in dem es zu Exceptions kommen kann, umschließt man mit einer try-catch-Anweisung
- Bei checked Exceptions **muss** man dies tun, bei unchecked Exceptions **kann** man dies tun

# Prinzipieller Aufbau von try-catch-Anweisungen

Spezifikation der zu behandelnden Fehlerart und Benennung einer Fehlervariablen ähnlich einer Variablendeklaration: `Typ NameDerVariablen`

```
try {
      ...  // normaler Code
   } catch (ExceptionKlasse1 name) {
      ...  // Behandlung eines Fehlers zur ExceptionKlasse1
   } catch (ExceptionKlasse2 name) {
      ...  // weitere catch-Bloecke
   }  ...
   } finally {
      ...  // Code, der auf jeden Fall ausgefuehrt wird
   }
}
```

Programmcode, in dem eine Exception auftreten kann

Hier drei Exception Handler: Programmcode, in dem der jeweilige Fall behandelt wird

- Mindestens ein catch-Block (oder 0 catch-Blöcke und dann 1 finally-Block)
- finally-Block 0-mal oder 1-mal vorhanden

# Beispiel

```
class MeineExceptionFangen {
    public static void main(String[] args) {

        try {

            // Exception werfen
            throw new MeineException();
            // hier hin kommen wir nicht in der Ausfuehrung
            System.out.println("ueberweise 1 Mrd. Euro auf mein Konto");

        } catch(MeineException e) {

            // Fehlermeldung ausgeben
            System.out.println("Exception geworfen: " + e.getMessage());
            // Aufrufhierarchie von Methoden als Hinweis ausgeben
            e.printStackTrace();

        }
    }
}
```

1 catch-Block, 0 finally-Block

# Abarbeitung einer try-catch-Anweisung

- Tritt keine Exception im try-Block auf, so wird die Ausführung nach dem try-Block hinter der try-catch-Anweisung fortgesetzt (aber: siehe finally)

- Tritt eine Exception in einem try-Block auf, so werden die catch-Fälle von oben nach unten überprüft
- Dazu wird jeweils überprüft, ob das geworfene Exception-Objekt eine Instanz der angegebenen Exception-Klasse ist (Achtung: inklusive Vererbung)
- Der Exception Handler des ersten zutreffenden Falls wird dann ausgeführt, alle nachfolgenden ignoriert.
- Nach dem Exception-Handler wird die Ausführung nach der try-catch-Anweisung fortgesetzt (aber: siehe finally)

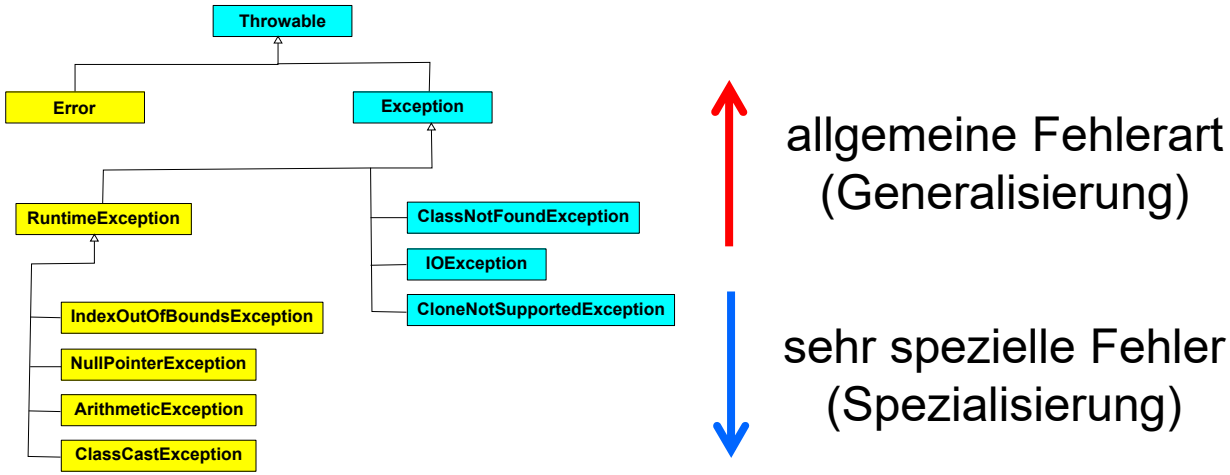
- Falls vorhanden, wird der finally-Block **immer** (in Worten: IMMER) ausgeführt, egal ob eine Exception geworfen wurde oder nicht, egal ob ein catch-Fall vorher zutraf oder nicht
- finally wird oft genutzt, um in jedem Fall einen definierten Endzustand herzustellen

# Nutzung der Exception-Klassenhierarchie

- Die Klassenhierarchie nutzt man z.B. in komplexen Fehlersituationen aus
- Wiederholung: Oberklassen sind allgemeiner, abgeleitete Klassen spezieller

- Eine Überprüfung der Fälle in catch-Blöcken macht nur Sinn von speziellen Fällen zu Beginn hin zu allgemeinen Fällen in späteren catch-Blöcken

- Würde man den allgemeinsten Fall zuerst überprüfen, würde dieser für alle Fehlerarten **immer** zutreffen und es würde nie zu einer Überprüfung der anderen Fälle kommen (siehe nachfolgendes Beispiel für eine korrekte Anwendung)



# Beispiel für die Nutzung von finally / Hierarchie

```
Scanner sc = null;
try {
    sc = new Scanner(new File(args[0]));
    int summe = 0;
    for(int i=0; i<10; i++) {
        summe += sc.nextInt();
    }
    System.out.println("summe=" + summe);

} catch(FileNotFoundException e) {
    System.out.println("Datei nicht gefunden");

} catch(NumberFormatException e) {
    System.out.println("Zahl in Datei falsch angegeben");

} catch(Exception e) {
    System.out.println("anderer Fehler");

} finally {
    // der Scanner muss auf jeden Fall geschlossen werden
    if(sc != null) {
        sc.close();
    }
}
```

spezielle Fehler

Reihenfolge wichtig!

allgemeiner Fehler

immer

# 4. Propagieren

- Szenario: Sie schreiben eine Methode, in der es zu Fehlerfällen kommen kann, die aber dort nicht behandelt werden sollen (z.B. innerhalb einer Software-Bibliothek)

- Bei checked Exception **muss**, bei unchecked Exceptions **kann** man die Methode markieren, dass sie Exceptions werfen kann. Eine möglicherweise auftretende Exception wird dann propagiert und muss nicht selber an dieser Stelle behandelt werden

- Notwendiger Zusatz im Methodenkopf:

```
throws Exceptionklasse_1,…,Exceptionklasse_n
```

- Wird eine Exception geworfen, die in der Methode selbst nicht behandelt wird, so wird die Exception entlang der Aufrufhierarchie der Methodenaufrufe propagiert, bis ein passender Exception Handler gefunden wird
- Spätestens das Laufzeitsystem / main (oberste Stufe der Aufrufhierarchie) hat zu jeder unchecked Exception einen Standard-Handler

# Beispiel

```
static Scanner sc = null;
public static void main(String[] args) {
    // Beispiel 1: Exception wird in Methode selbst gefangen
    initializeScanner1(args[0]);

    // Beispiel 2: propagierte Exception fangen
    try {
       initializeScanner2(args[0]);
    } catch(FileNotFoundException e) { S.o.p("Datei nicht gefunden"); }
}

static void initializeScanner1(String filename) {
    try {
       sc = new Scanner(new File(filename));
    } catch(FileNotFoundException e) {
        System.out.println("Datei nicht vorhanden");
        return;
    }
}

static void initializeScanner2(String filename) throws FileNotFoundException {
    // Scanner von der Datei "test.txt" anlegen
    sc = new Scanner(new File(filename));
}
```

# Zwischenstand

- Exceptions werden durch das System oder mit der throw-Anweisung geworfen
- Das Laufzeitsystem sucht den passenden Exception-Handler; dort wird die Ausführung fortgesetzt
- Die Exception-Hierarchie sollte man sinnvoll nutzen
- Ein finally-Block wird immer ausgeführt
- Methoden können/müssen Exceptions propagieren

## Reflektion

- Diskutieren Sie folgenden Code:

```
try { throw /* überlegen Sie sich einige Fälle hier */;
} catch (Exception e)      { S.o.p("hallo 1");
} catch(MeineException e) { S.o.p("hallo 2");
} finally                  { S.o.p("Hallo 3");
}
```

# Assertions

- Exception: behandle Laufzeitfehler
- Jetzt: Spezifiziere logische Annahmen (Assertions) an Programmstellen, die bei der Programmentwicklung (und nur dann) überprüft werden sollen und so helfen sollen fehlerfreien Code zu entwickeln
- Über eine Option des Java Laufzeitsystems `java -ea` kann man Assertions einschalten / scharf schalten (Programmentwicklung) und ausschalten (Produktionscode), **ohne** den Programmcode ändern zu müssen
- Default: aus
- Später dazu: Programmkorrektheit über Assertions

- Wichtiger Unterschied in Bezug auf die Programmausführung:
  - Alle Exception-Fälle werden immer überprüft und ggfs. geworfen
  - Assertions sollen nur in der Programm**entwicklung** überprüft werden, in der Produktionsversion des Codes sollte es **keine** dieser Überprüfungen für Assertions mehr geben (Code ist dann schneller; Assertions haben ihren Zweck erfüllt)
- Assertions ersetzen nicht Programmlogik zu Fehlerüberprüfungen

# Assertions formulieren

- Assertion-Anweisung in zwei Formen
  1. `assert` <Bedingungsausdruck> `;`
  2. `assert` <Bedingungsausdruck> `:` <Ausdruck2> `;`

- Falls der Wert des Bedingungsausdrucks `true` ist, setze Ausführung mit nächster Anweisung fort.

- Ansonsten Unterschied bei den Varianten:
  1. Ansonsten werfe eine Exception `AssertionError`
  2. Ansonsten werte den zweiten Ausdruck aus. Mit dessen Rückgabewert wird der Konstruktor von `AssertionError` aufgerufen und dieses Objekt dann geworfen

- `AssertionError` ist von `Error` und nicht von `Exception` abgeleitet
- Eine `AssertionError`-Exception sollte nur vom Laufzeitsystem behandelt werden und nicht selbst in einem eigenen Handler

# Beispiel

```
public static void main(String[] args) {
    float degreeCelsius = Float.parseFloat(args[0]);
    // hier sollte keine Assertion sein,
    // weil dies das normale Abfangen falscher Benutzereingaben ist
    if(degreeCelsius < -273.15f) {
        System.out.println("falsche Eingabe fuer Temperatur in Grad C");
        System.exit(1);
    }
    // rechne in Kelvin um
    float degreeKelvin = celsiusToKelvin(degreeCelsius);
    assert(degreeKelvin >= 0);
    System.out.println("Die Temperatur ist " + degreeKelvin + " Grad K");
}

// Umrechnen von Celsius nach Kelvin
static float celsiusToKelvin(float c) {
    // hier ist ein Fehler in der Logik: -1 am Ende ist falsch
    return c + 273.15f - 1;
}
```

```
prompt> java -ea AssertionTest -273
Exception in thread "main" java.lang.AssertionError
        at AssertionTest.main(AssertionTest.java:22)
```

Entwicklungsphase

```
prompt> java AssertionTest -273
Die Temperatur ist -0.8500061 Grad Kelvin
```

Produktionsphase

# Zusammenfassung

- Exceptions dienen der Fehlerbehandlung
- Definition von Fehlerklassen und Objekte dieser Klassen als konkrete Fehlerinstanzen
- Checked Exceptions helfen bei der Entwicklung von sicherem Code
- Trennung von eigentlichem Programmcode und Fehlerbehandlung möglich
- Durch Fehlerklassenhierarchie kaskadierte Fehlerüberprüfung möglich
- Exceptions können entlang einer Aufrufhierarchie propagiert werden

- Assertions haben einen anderen Zweck: Hilfe bei der Programmentwicklung
- Assertions bleiben im Code, lassen sich aber gezielt beim Übersetzen aktivieren / deaktivieren