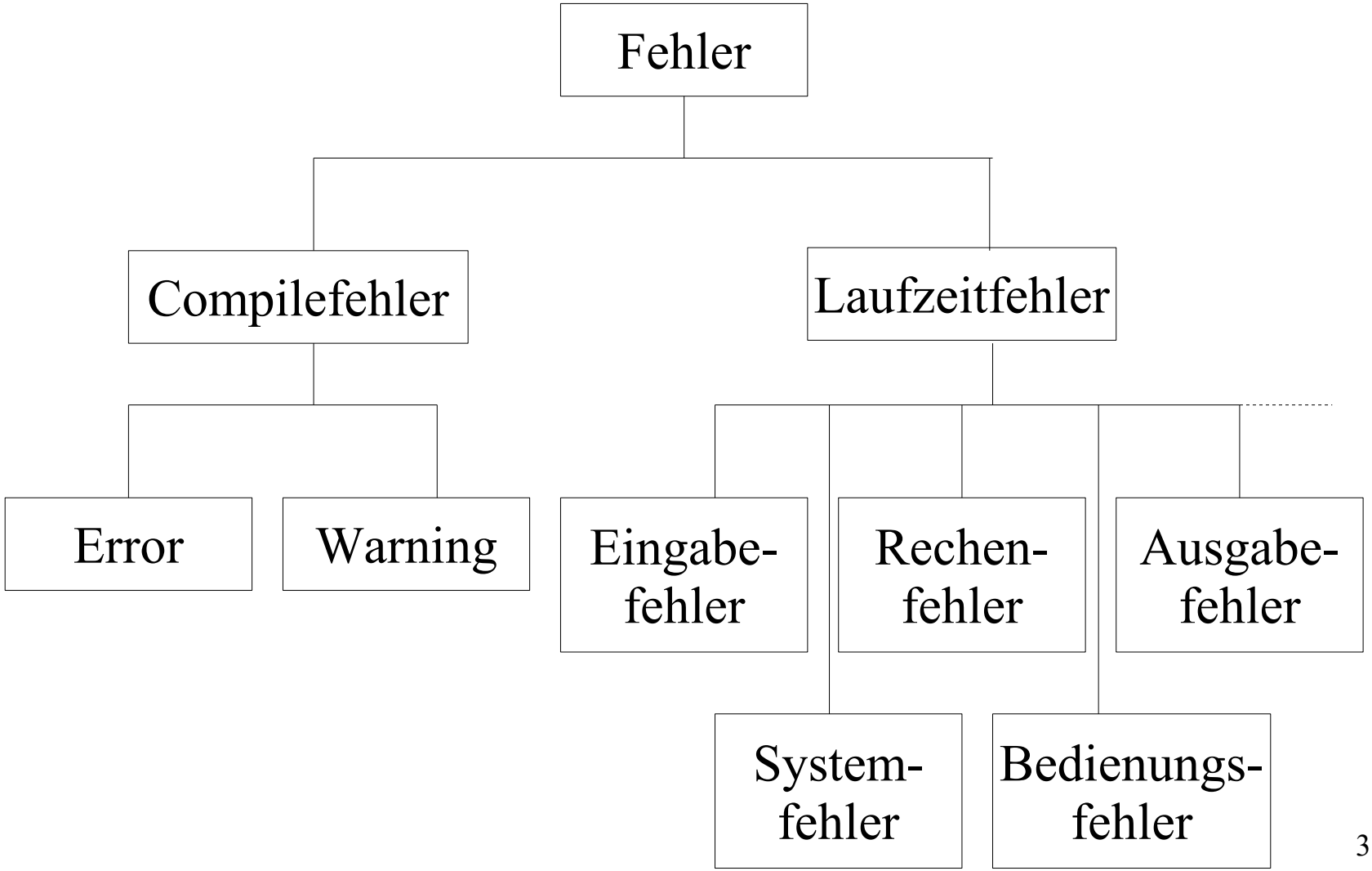
Java Fehlerbehandlung

- Fehlerklassifikation
- Java – Fehlerbehandlung
- Fehlerklassen
- Exceptionklassen

Softwarequalität - Benutzersicht

- Zuverlässigkeit
  - Korrektheit
    Funktioniert wie spezifiziert
  - Robustheit
    Auf alle Eingaben/Ereignisse definierte Reaktion
  - Verfügbarkeit
    Einsatzbereit

Fehlerklassifikation



Fehler

- fault: Unzulässige Eigenschaft, kann zu (Teil-)Versagen führen
- defect:
  - Unzulässige Abweichung eines Merkmals
  - Nichterfüllung von Merkmalswerten
- error: Abweichung zwischen berechnetem und wahrem, spezifiziertem oder theoretisch richtigem Wert
- mistake: Menschliche Handlung, die ein unerwünschtes Ereignis zur Folge haben kann

Fehlerbehandlung

- Fehler können erkannt werden:
  - beim Übersetzen
  - zu Laufzeit
- Java erzwingt für viele Laufzeitfehler eine explizite Behandlung
  - Mögliche Fehler sollen nicht ignoriert werden
- Laufzeitfehler werden durch "Exception-Handler" bearbeitet
  - das Abfragen von Fehler-Codes aus dem Rückgabewert einer Methode ist nicht notwendig

Compilerfehler und -warnungen

- Errors
  - Müssen behoben werden (sonst geht gar nichts)
- Warnungen
  - Muss man genau verstehen, bevor man sie ignorieren kann (aber immer auf eigenes Risiko).
  - Beispiele
    - Attribut, Methode, Parameter oder Variable wird nie benutzt.
      Nur noch nicht fertig? Designfehler? Sollte leicht zu beheben sein.
    - Depricated Hinweise: Bei neuem Code sofort beheben, bei altem Code Behebung planen.

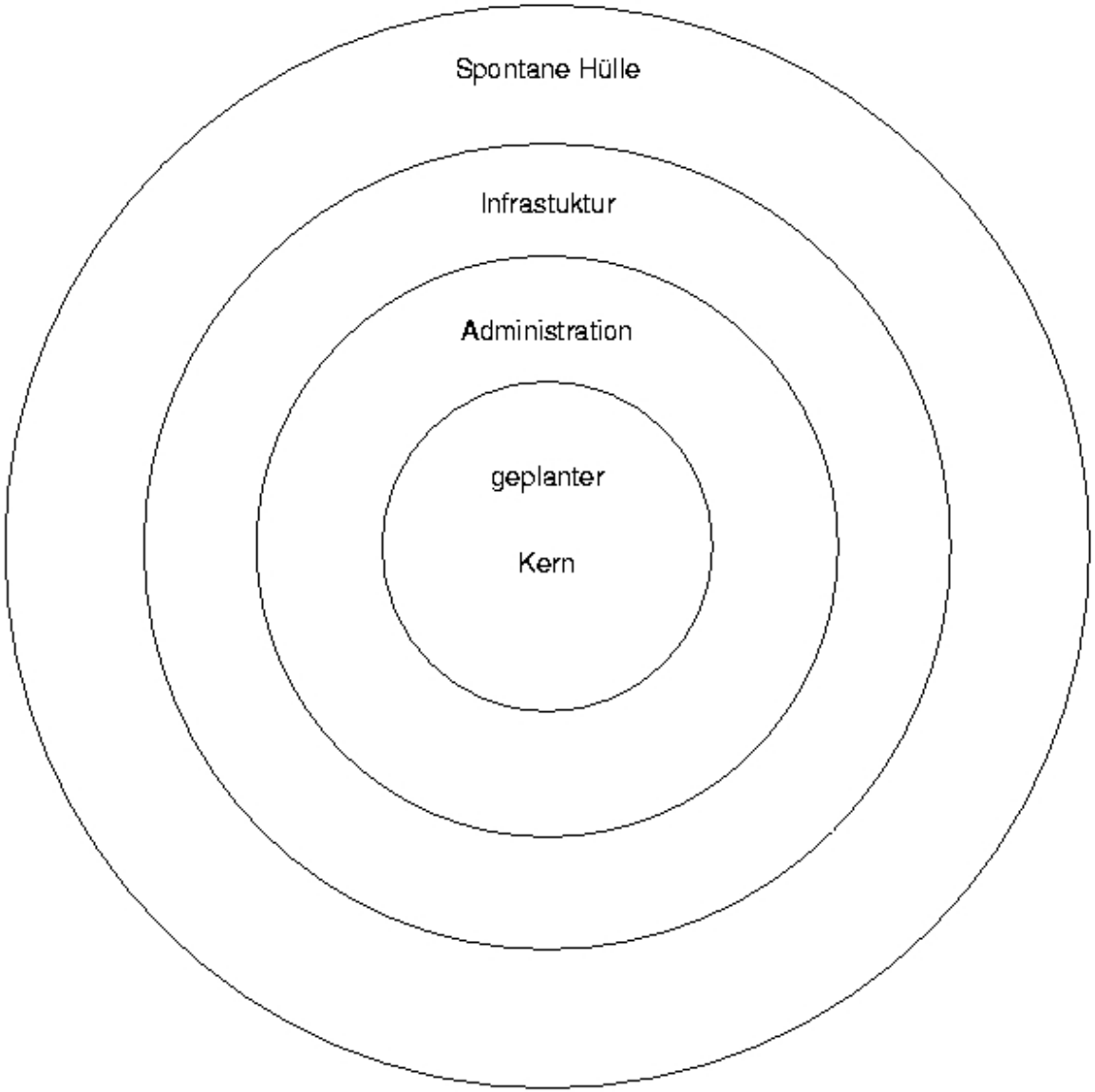
mehr zu Compilerwarnungen

- „List is a raw type. References to generic type List<E> should be parameterized“: In neuem Code sofort beheben. In altem: Umstellung planen.
  Es ist sehr selten, dass ein raw type akzeptabel ist. Siehe Generics, Reflection.
- Generell: Warnungen weisen auf mögliche Probleme für die Weiterentwicklung oder zur Laufzeit hin. Missachten Sie sie nur, wenn Sie wirklich wissen was Sie tun (also wahrscheinlich nie)!

Laufzeitfehler

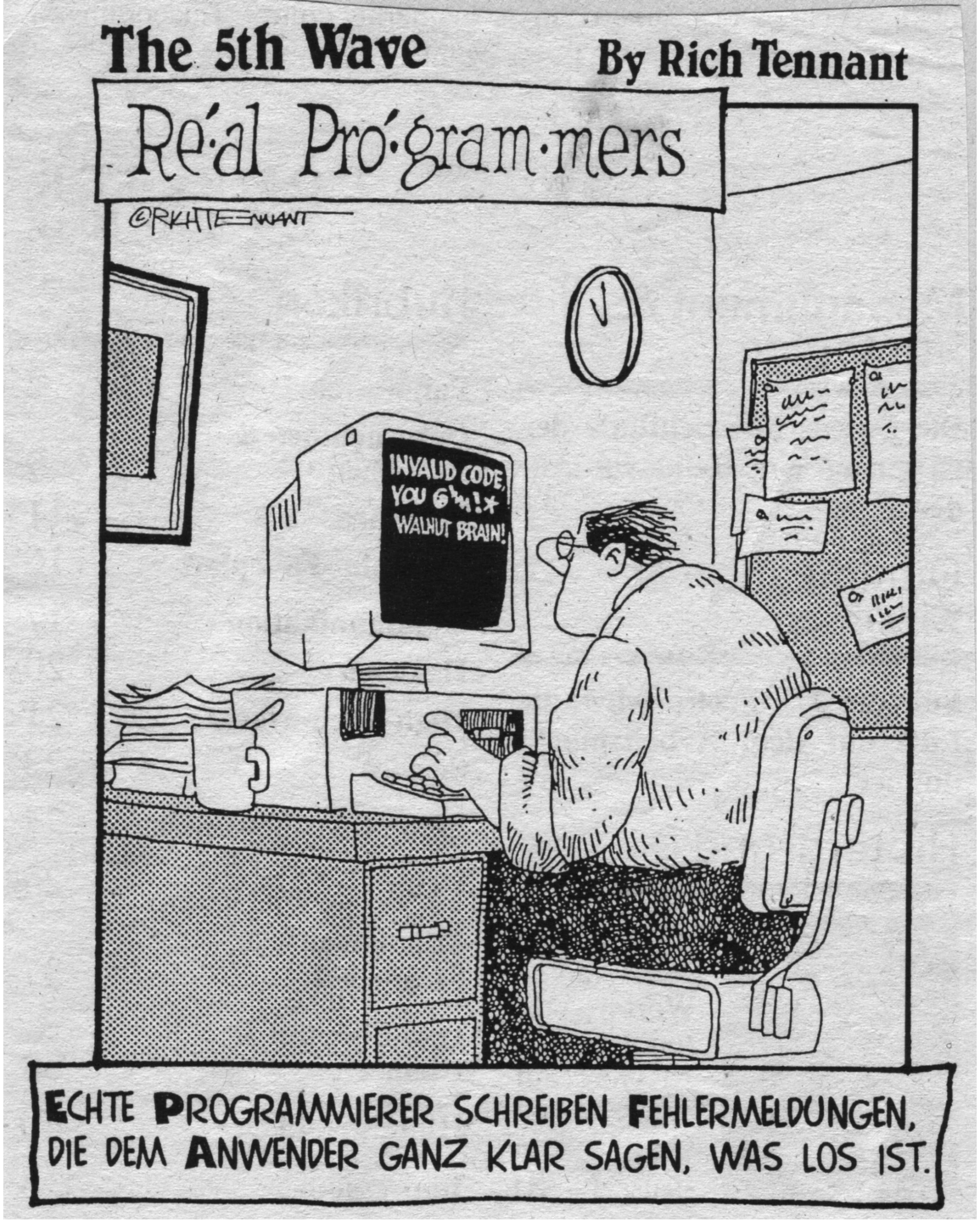
- Fehlerhafte Eingaben
  - Durch den Benutzer
  - Durch andere Methoden
  - Durch andere Teilsysteme
- Programmiererfehler
  - Division durch 0
  - Unter- Überschreiten der Arraygrenzen
  - Verwenden von nicht initialisierten Attributen oder Variablen
- Umgebungsfehler (JVM, Betriebssysteme)
- Hardwarefehler

Systemschichten (Zwiebelmodell)



Eingabeprüfungen

- Einzelne Felder
  - Datum gültig, Wert numerisch …
- Gruppen von Feldern:
  - Kein Wert doppelt (Mastermind)
  - Buchung: von Datum < bis Datum
- Prüfungen mit bereits gespeicherten Eingaben
  - Kunde in Datenbank
- In jedem Fall:
  - Meldung an Benutzer mit Hinweis auf Fehler und Korrekturmöglichkeiten
  - Alles zusammen prüfen und dann melden


The 5th Wave
By Rich Tennant
Re·al Pro·gram·mers
©RKHTennant
INVALID CODE, YOU G*#!+ WALNUT BRAIN!
ECHTE PROGRAMMIERER SCHREIBEN FEHLERMELDUNGEN, DIE DEM ANWENDER GANZ KLAR SAGEN, WAS LOS IST.

Was macht man in anderen Situationen?

- Nicht vorhersehbare ungültige Operationen
  - Neues Objekt anlegen aber kein Platz mehr
  - Objekt sollte da sein, ist aber null
  - Lesen/Schreiben aber Datei existiert nicht
  - …
- Sonstige Systemfehler

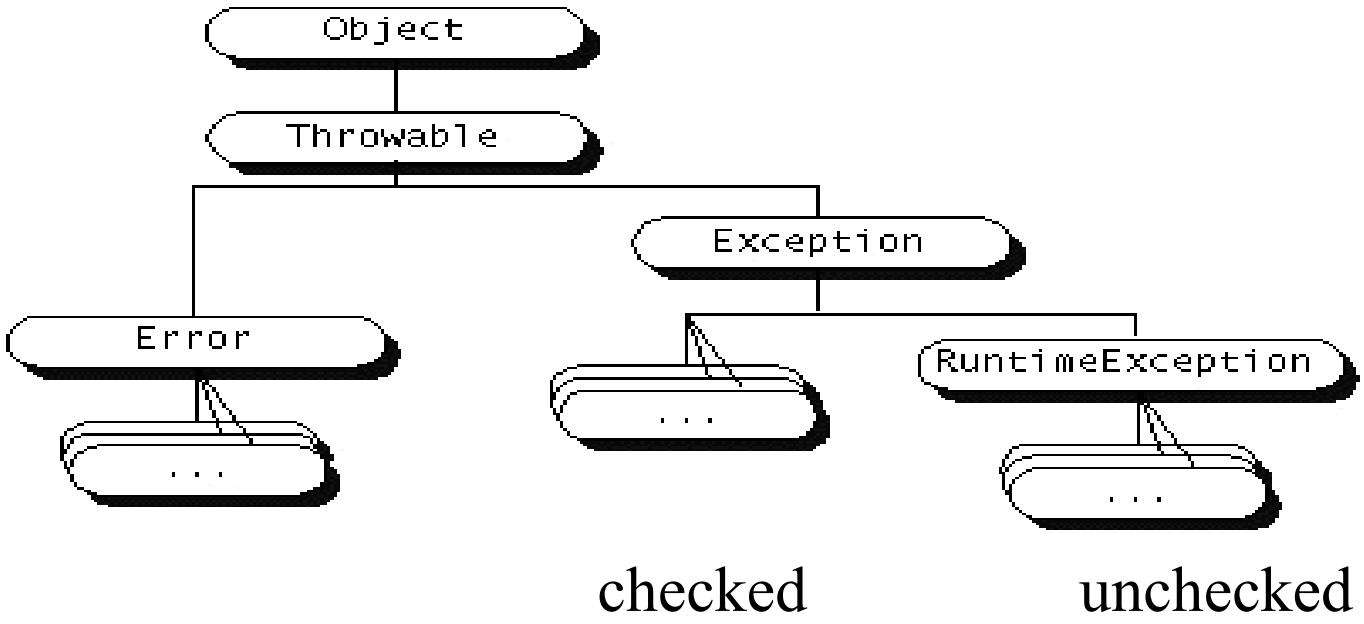
Erzeugung von Exceptions

- In einer nicht auflösbaren Situation wird eine Exception erzeugt - Schlüsselwort `throw`
  - Der normale Programmablauf wird dadurch unterbrochen
  - Ein Exception-Handler wird ausgeführt, um auf das Problem zu reagieren
  - Der Weg von der Erzeugung einer Exception zu dem Exceptionhandler geschieht **nicht** über den normalen Methoden/Funktionen - Weg

```
// Beispiel Null-Pointer
if (t == null) throw new NullPointerException();
// oder - anderer Konstruktor
if (t == null)
   throw new NullPointerException("t = null");
```

Exception - throw

- Exceptions werden durch `throw` ausgelöst
  - **`throw`** übergibt immer ein Objekt, das vom Typ **`Throwable`** ist (kann also abgeleitet von **`Throwable`** sein)
  - **`Throwable`** - Objekte speichern eine Momentaufnahme der Fehlersituation, die später abgefragt werden kann



Exception - Hierarchie

- `Throwable` ist die Superklasse für alle Fehlersituationen
- `Error` sind Fehler, die durch Internals der JVM erzeugt werden - *sollten nicht vorkommen* - werden üblicherweise nicht aufgefangen
- `Exception` ist die Superklasse für eine große Menge vordefinierter Fehlerklassen
- Eine Subklassen-Familie - `RuntimeException` - sind Fehler beim Ablauf der JVM, z.B.:
  - Division durch Null
  - zu großer Array-Index
  - Diese heißen unchecked Exceptions
- Die vordefinierten Klassen haben ausdrucksvolle - den Fehler gut beschreibende Namen

Auffangen von Exceptions

- Code, der eine Exception produziert, kann in einen Versuchsblock (`try`) gekleidet werden.
- Wird die Exception nicht durch einen `try`-Block behandelt, so muss die Exception auf die nächsthöhere Ebene propagiert werden mit `throws :`

```
public void funcException()throws ExceptionTypX,
ExceptionTypY, ExceptionTypZ { ... }
```

- Exceptions können bis zu `main` propagiert werden und führen dann beim Auftreten zum Programmabbruch
- Der `try`-Block endet mit Exception - handler(n) (`catch`)

```
try {
  // Hier kann eine Exception passieren
} catch (ExceptionTypX e) {
  // Bearbeitung der Exception oder
Programmabbruch
}
```

Auffangen von Exceptions II

- Der `try`-Block kann viele mögliche Exception-Situationen enthalten
- Es können mehrere Exception - Handler per `catch` hintereinandergereiht werden
- oder man kann durch Angabe der Super-Klasse `Exception` alle Exceptions in einem `catch`-Block abfangen

```
try {
  // Code, der Exceptions generieren könnte
} catch (ExceptionTyp1 e1) {
  // code zum Bearbeiten des ExceptionTyp1
} catch (ExceptionTyp2 e2) {
  // code zum Bearbeiten des ExceptionTyp2
} // u.s.w.
```

Eigene Auswertung einer Exception

- In dem catch-Block können die Methoden des übergebenen Exception-Objekts aufgerufen werden:
- z.B. aus Throwable:

```
String getMessage()
String toString()
printStackTrace()   // Standard Error Ausgabe
printStackTrace(PrintStream)// anderer Ausgabekanal
Throwable fillInStackTrace() // Füllen mit Info
```

try with Resources

```
try(SomeClazz scz = new SomeClazz())  {
    // Code, der Exceptions generieren könnte
    //Verwendung von scz
  } catch (ExceptionTyp1 e1) {
    // code zum Bearbeiten des ExceptionTyp1
  }
```

- **Möglich, wenn die Ressource `SomeClazz` AutoCloseable implementiert**
- **Dann immer das Konstrukt der Wahl!**
- **Ersetzt weitgehend finalizer (deprecated seit Java 9)**

Weitergabe von Exceptions

- Eine Exception, die durch einen catch-Block aufgefangen wurde, kann weitergegeben werden an die nächsthöhere Handler-Instanz

```
  ...
catch(Exception e) {
    // Bearbeitung nicht vollständig möglich ..
    throw e.fillInStackTrace(); // Weitergabe
    // mit aktueller Stack-Information
  }
```

Eigene Exceptions

- Für spezielle Fehler können eigene Exceptions erzeugt werden

```
triviales Beispiel:
class SimpleException extends Exception {// ... }
public class SimpleExDemo {
  public void f() throws SimpleException {
    System.out.println ("SimpleException from f()");
    throw new SimpleException();
  }
  public static void main(String[] args) {
    SimpleExDemo sed = new SimpleExDemo();
    try { sed.f(); } catch (SimpleException e) {
      System.err.println("Caught it !");
    }
  }
}
```

Eigene Exception mit Konstruktor

- Ein Konstruktor mit String - Parameter, der eine Meldung übergeben kann:

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg); // Übergabe der Meldung
  }
}

// Benutzung in einer anderen Klasse
public class ...
  public void g() throws MyException {
      ......
      throw new MyException("Fehler in g() !");
```

Standard Exceptions

- Die Liste der speziellen Exception-Klassen ist lang und wächst ständig
- In der Doku angucken !
- Beispiele:

```
ClassNotFoundException
IllegalAccessException
IOException : FileNotFoundException,
  SocketException ...
NoSuchFieldException
NoSuchMethodException
RuntimeException :
  ArithmetikException, ClassCastException,
  NegativeArraySizeException, NullPointerException,
  IndexOutOfBoundsException ....
```

Speziell: RuntimeException

- `RuntimeException` ist eine Subklasse von `Exception`
- `RuntimeException` sind Programmierfehler:
  - Division durch 0
  - Falscher Array-Index
  - null - Referenz
- Sie dienen zum schnellen Finden von Fehlern
- Eine Fehlerbehandlung entfällt üblicherweise
- Sie müssen nicht mit `catch` aufgefangen und nicht mit `throws` deklariert werden
- Der auftretende Fehler propagiert automatisch bis zu `main` und stoppt das Programm
- Der Programmierer kann den Code korrigieren

Wer fängt die Exception ?

- Eine Exception-Klassenhierarchie

```
class KopfwehException extends Exception {  ...  }
class GrippeException extends KopfwehException {  .... }
public class Mensch {
  public static void main (String [] args) {
    try {
      ......
      throw new GrippeException ();  // <-----
    } catch (GrippeException g) {
      System.err.println("Grippe eingefangen");
    } catch (KopfwehException k){
      System.err.println("Kopfweh eingefangen");
    }
  }
}
```

Wer fängt die Exception ?

- Eine Exception wird immer durch den nächsten Handler aufgefangen, wenn er "passt"
- Ein Handler passt, wenn die Exception im `catch`-Block denselben Typ hat oder eine Superklasse der auslösenden Exception ist.
- Auf der vorigen Folie wird die Exception durch den ersten `catch`-Block aufgefangen (`Grippe`)
- Würde er fehlen, würde die Exception durch den 2. `catch`-Block aufgefangen werden (`Kopfweh`)
- Die `catch`-Blöcke müssen zunächst die Subklassen, dann die Superklassen aufreihen (vom Speziellen zum Allgemeinen)

Exception-Beispiele

- SimpleExceptionDemo
  - Eine eigene Exception wird ausgelöst
- FullConstructors
  - Exceptions können unterschiedliche Konstruktoren haben
- ExtraFeatures
  - Exceptions können zusätzliche Methoden haben
- ExceptionMethods
  - Demonstriert die Stand.-Methoden von Exceptions
- Rethrowing
  - Exceptions können weitergegeben werden
    - Dabei wird die Herkunft nicht vergessen
    - Mit fillInStackTrack( ) wird die Herkunft neu definiert

Exception-Beispiele II

- RethrowingNew
  - Es kann in einer Exception eine andere Exception ausgelöst werden
    - Dann geht die Information über die alte Exception verloren
- NeverCaught
  - RuntimeExceptions müssen nicht aufgefangen werden
- FinallyWorks
  - Der finally – Block wird immer ausgeführt, egal, ob die Exception passiert oder nicht
- LostMessage
  - Exceptions können verloren gehen

Assert

- assert dient dazu, die Korrektheit des Werts eines Ausdrucks zu überprüfen
- `assert boolscher Ausdruck;`
  - ***Abbruch, wenn nicht true durch Error***
  - ***Sonst Fortsetzen des Programms***
- `assert boolscher Ausdruck : " text ";`
  - ***Wenn nicht true: Ausgabe des Textes, dann Abbruch durch Error***
  - ***Sonst Fortsetzen des Programms***

Assert / Exceptions

- Assertions dienen zur abschaltbaren Überprüfung des Programms
  - Es soll getestet werden, ob die eigene Programmlogik richtig ist
  - Nach Überprüfung der Korrektheit können Assertions abgeschaltet werden
  - Innerhalb der Test- und Debugphase dagegen sind die Assertions eine große Hilfe beim Aufdecken von verdeckten Fehlern.
- Exceptions dienen zur Überprüfung von Übergabeparametern
  - Z.B. wenn falsche Parameter an eine öffentliche Methode übergeben werden
  - Sind nicht abschaltbar
- Assertions sind auch sinnvoll
  - Wenn nach Überprüfung von externen Parameter an interne (nicht öffentliche) Methoden Parameter übergeben werden

Assertions - wann

- In nicht öffentlichen Methoden und Konstruktoren:
  - Auch hier müssen Sie das Einhalten von Vorbedingungen überprüfen.
  - Eine throws Klausel würde „nach oben“ propagiert und damit andere ProgrammiererInnen belasten (wenn checked Exception) oder zu unnötigen Exceptions führen können.
  - Hilft Ihnen Fehler zu finden.
  - Kann bei Bedarf auch in Produktion zur Fehlerlokalisierung eingesetzt werden.

Preconditions / Postconditions

- Exceptions dienen zur Überprüfung von *Preconditions*
- Assertions dienen zur Überprüfung von *Postconditions*
  - Jede wohldokumentierte Methode schließt einen „Vertrag“ mit dem Benutzer ab:
    - Der Gültigkeitsbereich der Eingangsparameter wird genau beschrieben und muss eingehalten werden **– *Preconditions***
    - Wenn dieses der Fall ist, leistet die Methode die dokumentierte Funktionalität
    - Erlangt einen dokumentierten Status -> ***Postconditions***
    - Und gibt ein Ergebnis zurück -> ***Postconditions***

Assert / Aktivierung / Eclipse

- Assertions gibt es seit jdk 1.4
  - Sie sind nur aktiviert, wenn die VM aufgerufen wird mit

    ```
    java –ea programmname
    ```

  - Unter Eclipse muss eingestellt sein:
    - Windows -> Preferences -> Java -> Compiler
      - jdk 1.6 und
      - assert = > Error
    - Unter Run –> Arguments-> VM-Arguments muss eingegeben sein
      - -ea (d.h. enable Assertions)

Exception Handler

- SetDefaultExceptionHandler