- Ausnahmebehandlung -

Überblick Gesamt

- Einführung in das Programmieren (S. 1-140, S. 181-226, = 185 Seiten)
  - Grundlagen (Kapitel 1, im Schnelldurchlauf!)
  - Datentypen und Operatoren (Kapitel 2)
  - Kontrollanweisungen (Kapitel 3)
  - Arrays (Kapitel 5)
- Objektorientierte Programmierung (S. 141-180, S. 227-388, = 200 Seiten)
  - Klassen, Objekte Methoden (Kapitel 4)
  - Genauere Betrachtung der Methoden und Klassen (Kapitel 6)
  - Vererbung und Polymorphie (Kapitel 7)
  - Pakete und Schnittstellen (Kapitel 8)
  - Ausnahmebehandlung (Kapitel 9)

Überblick Heute

- Fehlerbehandlung
- defensives Programmieren
- Exceptions werfen
- 2 Arten von Exceptions
- Auswirkung einer geworfenen Exception
- unchecked und checked Exceptions werfen
- geworfene Exceptions abfangen
- die finally Klausel

Fehlerbehandlung

- Frei nach Wolf Haas: "Jetzt ist schon wieder etwas passiert"
  - unterschiedliche Gründe für Fehler
    - logischer Fehler in unserer Software (Bug)
    - falsche Anwendung (z.b. kaputter Zustand durch erweiternde Klasse)
    - Probleme ausserhalb der Applikation (Platte voll, Netzwerkausfall, ...)
    - ...

- Behandlung von Fehlern als fixer Bestandteil von robuster Software
  - Auftreten von Fehlern erwarten
  - sinnvoll reagieren: je nach Kontext
  - sorglos vs. defensiv vs. paranoid
  - Unterrichtsbeispiel vs. Pragmatik vs. Militär/Finanzen

- Keywords
  - Error Handling, Exception Handling
  - Error Reporting
  - Error Avoidance vs. Error Recovery (!)

defensives Programmieren (1)

- "Client – Server" Interaktion
  - unsere Klassen (Server) werden von anderen Klassen (Client) benutzt
  - zwei Sichtweisen bzgl. Fehlerbehandlung möglich:
    - annehmen, dass der Client unsere Klassen wie vorgesehen benutzt
    - sich auf nichts verlassen, alles gegenchecken

- Umfeld beeinflusst Entscheidung
  - z.B. (ab)geschlossene, kleinere Applikationen
  - z.B. eine öffentliche Klassenbibliothek
  - guter Kompromiss: defensives Programmieren

```
public class Client {
    public static void main(String args[]) {
        int r = MathTools.remainder(10, 0); // Div. durch 0
        System.out.println("Rest bei Division: " + r);
    }
}
```

defensives Programmieren (2)

- ... aus der Sicht des Servers
  - Strategie z.B.: frühzeitig Methodenparameter überprüfen
  - allerdings: was tun im Fehlerfall?
  - folgende Überlegungen:
    - Client benachrichtigen, z.B. per Rückgabewert Fehler signalisieren
    - Anwendungen benachrichtigen, z.B. per Ausgabe an die Konsole/GUI
  - beide Überlegungen sind problematisch! Warum?

- allgemeinere Lösung gesucht

```
public class MathTools {
    public static int remainder(int z, int n) {
        if (n == 0) {
            // Ausnahmesituation! Wie reagieren?
        }
        return (z % n)
    }
}
```

Exceptions werfen

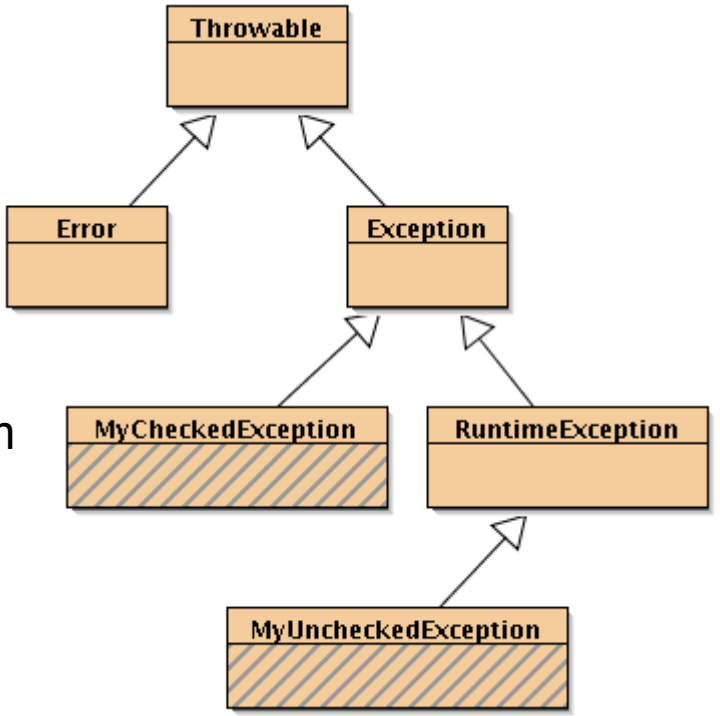
- Exceptions werfen
  - ... ist die effektivste Möglichkeit um dem Client mitzuteilen, dass man eine Aufgabe nicht erfüllen kann.

- Vorteile
  - geworfene Exceptions können nicht einfach ignoriert wrden
  - Exceptions sind unabhängig vom Rückgabewerts einer Methode
  - Exceptions sind Objekte, die Informationen über den Fehler kapseln

```
public class MathTools {
    public static int remainder(int z, int n) {
        if (n == 0) {
            throw new IllegalArgumentException("Nenner 0!");
        }
        return (z % n)
    }
}
```

2 Arten von Exceptions

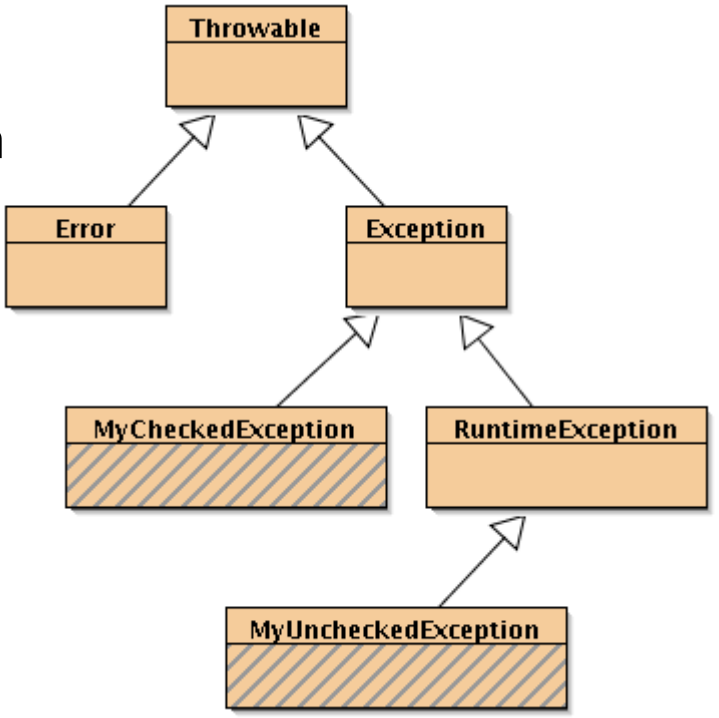
- Klassenhierarchie aus `java.lang`



- man spricht von 2 Arten von Exceptions
  - unchecked Exceptions
    - = `RuntimeException` und Subklassen
    - sollten im regulären Ablauf nicht auftreten
    - "Reparatur" der Situation unwahrscheinlich
    - oft Hinweis auf logischen Fehler im Programm
    - einigen Klassen sind wir bereits begegnet
      - `IndexOutOfBoundsException`
      - `ClassCastException`
      - `...`
  - checked Exceptions
    - = was sonst von `Exception` erbt
    - müssen vorausschauend behandelt werden
    - Reparatur oder sinnvolle Behandlung meist möglich
    - z.B.: Netzausfall, erneuter Versuch in 3 Minuten

übrigens: Errors

- Errors vs. Exceptions
  - selten benutzt, meist nur für fatale Fehler
  - in den allerseltensten Fällen selber geworfen
  - sollten nicht mehr behandelt werden
  - z.B. `VirtualMachineError`
  - z.B. `OutOfMemoryError`



Auswirkung einer geworfenen Exception

- die Ausführung der betroffenen Methode endet sofort
  - die noch folgenden Statements werden nicht mehr ausgeführt
  - keine Möglichkeit mehr, per `return` einen Wert zurückzugeben
  - die Exception "fällt nach oben" zum Aufrufer zurück
  - it's a Good Thing™

- Kontrolle liegt danach bei der aufrufendn Methode, ABER:
  - es wird nicht einfach nach dem problematischen Aufruf fortgesetzt
  - die Exception "fällt weiter"

- aufrufende Methode kann die Exception abgefangen & behandeln ...
  - ... oder nicht: dann fällt sie der nächsten aufrufenden Methode zu
  - das geht rauf bis zur `public static void main(...)`, wo die Exception schliesslich bis zum Anwender durchdringt und ausgegeben wird. Der Programmablauf wird dabei abgebrochen

unchecked Exceptions werfen

- wie bereits besprochen
  - einfach neue Exception Instanz erzeugen, dann per `throw` werfen
  - der Compiler setzt weder bei der werfenden, noch bei der aufrufenden Methode irgendwas voraus; deshalb auch der Name "unchecked"

- häufig geworfen: `IllegalArgumentException`
  - beim Methodenparameter überprüfen, bevor die eigentliche Operation beginnt, um den validen Zustand des Objekts nicht zu gefährden.
  - selbes Prinzip: Konstruktorenparameter überprüfen und ggf. Exception werfen, um keine Instanz mit einem ungültigen Zustand zu erzeugen.

```
public static int remainder(int z, int n) {
    if (n == 0) {
        throw new IllegalArgumentException("Nenner 0!");
    }
    return (z % n)
}
```

checked Exceptions werfen

- der Compiler setzt voraus, dass ...
  - ... Methoden alle checked Exceptions, die sie eventuell werfen, in ihrer Signatur deklarieren (per `throws` Klausel, siehe unten)
  - ... der Aufrufer solcher Methoden entweder die aufrufenden Exceptions abfängt oder deklariert, dass er sie selber wirft.

- immer gut: javadoc Kommentare schreiben!

```
public class AddressBook {
    /**
     * @throws FileNotFoundException ...
     * @throws EOFException ...
     */
    public void loadFromFile(String filename)
        throws FileNotFoundException, EOFException {
        ...
    }
}
```

geworfene Exceptions abfangen (1)

- das `try/catch` Statement
  - im `try` Block stehen die "geschützten" Statements
  - im `catch` Block wird eine Exception abgefangen und behandelt
  - letzterer wird nur dann ausgeführt, wenn tatsächlich eine passende Exception abgefangen wurde
  - auch Subklassen der angegebenen Exception werden gefangen

```
String filename = "addresses.csv";

try {
    // geschützte Statements
    loadFromFile(filename);
    updateDisplay();
} catch (FileNotFoundException ex) {
    // Fehlerbehandlung (Reporting und Recovery)
    System.out.println("Datei nicht gefunden: " + filename);
}
```

geworfene Exceptions abfangen (2)

- mehrere catch Blöcke möglich
  - aber: eine geworfene Exception kann nur einmal abgefangen werden
  - das erste passende `catch` Statement gewinnt (Vererbung beachten!)
  - "blinde" Behandlung *aller* Exceptions oft keine gute Idee

- wann fangen? wann fallen lassen?
  - Exceptions nicht abfangen, bevor nicht sinnvoll(!) reagiert werden kann

```
try {
    loadFromFile(filename);
    updateDisplay();
} catch (FileNotFoundException ex) {
    System.out.println("Datei nicht gefunden: " + filename);
} catch (EOFException ex) {
    System.out.println("Kann Datei nicht lesen: " + filename);
} catch (Exception ex) { // <- keine gute Idee
    System.out.println("Exception: " + ex);
}
```

and finally...

- die `finally` Klausel
  - optionale dritte Komponente für das `try/catch` Statement
  - wird ausgeführt, unabhängig davon ob eine Exc. auftritt oder nicht
  - steht entweder nach dem letzten `catch` Block: try – catch(es) – finally
  - oder anstelle von `catch` Blöcken: try – finally

- hauptsächlich für Aufräumarbeiten bei propagierten Exceptions

```
void doSomethingWithDatabase() throws SomeException {
    DatabaseConnection cx = null;
    try {
        cx = new DatabaseConnection();
        cx.doSomething();
    } finally {
        // belegte Ressourcen freigeben
        if (cx != null) cx.close();
    }
}
```