Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

11. Vorlesungswoche

Abschluss zur Objektorientierten Programmierung:

Interfaces, Exceptions und Packages

Unterlagen:

Echtle, Goedicke; Einführung in die objektorientierte Programmierung mit Java, dpunkt-Verlag.

Doberkat, Dissmann; Einführung in die objektorientierte Programmierung mit Java, Oldenbourg-Verlag, 2. Auflage.

K. Arnold, J. Gosling, The Java Programming Language, 2nd ed, Addison-Wesley

Folien nach V.Gruhn, Vorlesung Programmierung WS 99/00

Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Prinzipien objektorientierter SW Entwicklung

- Verbindung von Daten und Methoden zu Objekten
- Klassen als Schema zu gleichartigen Objekten
  - Objekte als Instanzen von Klassen
- Vererbung
  - Mehrfachvererbung
  - Einfachvererbung
- Polymorphie und Late-Binding
  - d.h. Methoden gleichen Namens dürfen auftreten, jeweils kontextabhängig wird die zugehörige Methode ausgewählt

- Datenkapselung, Geheimnisprinzip (Information hiding)
  - durch Klassenhierarchie, aber auch durch Schnittstellen / Interfaces unterstützt,

  Interfaces werden im folgenden betrachtet

Schnittstellen, Interfaces

- im Doberkat/Dißmann-Buch “Abstraktionen” genannt
- Grundidee:
  - Spezifikation von Funktionalität ohne Implementierung
    - das „was“ wird bekanntgegeben, das „wie“ bleibt offen/unbekannt
    - Trennung von Spezifikation und Implementierung
    - Geheimnisprinzip, Kapselung von Daten
- bisher bereits bei abstrakten Klassen aufgetreten
  - wirkte als Spezifikation jedoch nur innerhalb der Vererbungshierarchie
- allgemein: Modulkonzept
  - erlaubt Nutzung unbekannter Objekte gemäß einer bekannten Schnittstellenvereinbarung, die diese Objekte einhalten
  - erlaubt konkurrierende Implementierungen, wobei gewisse Eigenschaften festgelegt werden, die eine Implementierung gewährleistet muss.
- Was gehört zu einer Interface Beschreibung:
  - Signatur einer Methode legt Eingabe-/Ausgabe bzgl des Typs fest
  - Verhalten wird(würde) durch zusätzliche Axiome festgehalten (Theorie) oder durch zusätzliche natürlich sprachliche Beschreibung

Beispiel: Signatur der Algebra der natürlichen Zahlen

Signatur läßt sich auch formal einführen, ...

NatSig = (S, F) mit

S={Nat, Bool} Typsymbole, Sorten, Symbole für Mengen

$F=\{0_{Nat}, succ_{Nat}, +_{Nat}, =_{Nat}\}$ Funktionssymbole, Symbole für Werte

jedes f aus F hat einen Typausdruck als Typ der aus x, ->, Typvariablen und Elementen aus S besteht: bei natürlichen Zahlen

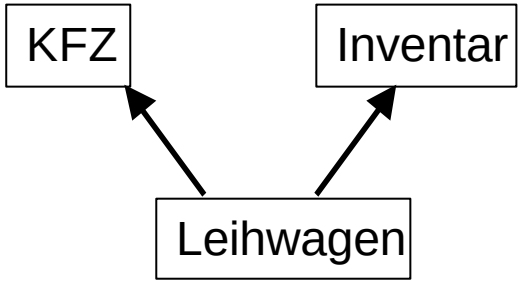
type: $F \rightarrow S^* \times S$

$0_{Nat} \rightarrow () \times Nat$	oder $0_{Nat}:() \rightarrow Nat$
$succ_{Nat} \rightarrow Nat \times Nat$	oder $succ_{Nat} : Nat \rightarrow Nat$
$+_{Nat} \rightarrow Nat \times Nat \times Nat$	oder $+_{Nat} : Nat \times Nat \rightarrow Nat$
$=_{Nat} \rightarrow Nat \times Nat \times Bool$	oder $=_{Nat} : Nat \times Nat \rightarrow Bool$
$true_{Bool} \rightarrow () \times Bool$	oder $true_{Bool} :() \rightarrow Bool$
$false_{Bool} \rightarrow () \times Bool$	oder $false_{Bool} :() \rightarrow Bool$
$=_{bool} \rightarrow Bool \times Bool \times Bool$	oder $=_{bool} : Bool \times Bool \rightarrow Bool$
$1_{Nat} \rightarrow () \times Nat$	oder $1_{Nat}:() \rightarrow Nat$

Eine weitere Motivation für Interfaces

Mehrfacherbung (multiple inheritance) =

eine Klasse besitzt mehrere Oberklassen



Problem: Welche Methode wird bei Namenskonflikten gewählt?

Lösung: In Java nicht möglich!

aber: hilfreiches Ersatzkonstrukt verfügbar: Interface

Interfaces

`extends` ist das Schlüsselwort für Interfacevererbung:

```
interface Unterinterface extends Oberinterface {
    Interfacebody
}
```

Klassen erben von Interfaces über das Schlüsselwort `implements`

- Klassen können von einer *Oberklasse* erben und zusätzlich ein oder mehrere *Interfaces* implementieren (unterstützen/realisieren) :

```
class NameUnterklasse extends NameOberklasse
                      implements Interface1, Interface2, ... {
...
}
```

- Werden von einer Klasse nicht alle Methoden des Interfaces implementiert, muß sie mit `abstract` gekennzeichnet werden
- Analog zum Erben von Oberklassen ist Polymorphie möglich

Klasse versus Interface

Alle Interface-Methoden sind implizit `public` und `abstract`.

- Begründung: `public` weil alle realisierenden Klassen wissen müssen, was sie realisieren sollen, `abstract` weil das das Merkmal von Interfaces ist.

=> Interface ist Spezialfall einer abstrakten Klasse,

(Eine Klasse ist abstrakt, wenn mindestens eine ihrer Methoden abstrakt ist!)

Alle Attribute sind implizit `public`, `static` und `final`

- Begründung: Es kann nur um Konstanten gehen, die immer gleich verwendet werden (also final). Diese braucht man dann nicht pro Objekt, sondern nur pro Klasse (also static).

Vorteil im Hinblick auf Mehrfacherbung:

- ohne Implementierungen keine Konflikte bezüglich Methoden möglich

```
public interface Konto {
      public void einzahlen();
      public void auszahlen();
      ...
}
```

Interfaces

- Interface wird durch Klasse implementiert

```
public class Privatkonto
implements Konto {
   public void einzahlen() {...}
   public void auszahlen() {...}
}
```

```
modifier class
cName
implements iName {
        ...
}
```

- Interfaces können eigene Vererbungshierarchie bilden,

  für Interfaces ist Mehrfacherbung erlaubt:

```
public interface Konto
extends Vertrag, Historisierbar {
   public void einzahlen() {...}
   public void auszahlen() {...}
}
```

```
interface I extends
I1,I2 {

Methodendeklarationen;
}
```

Interfaces

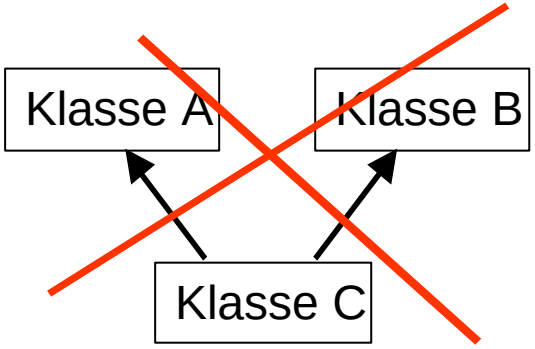
- eine Klasse kann mehrere Interfaces implementieren
  - alle vorgegebenen Methoden müssen realisiert werden

```
public class Privatkonto
  implements Konto, Zähler {
     public void einzahlen() {
        ...
     }
     public void auszahlen() {
        ...
     }
     public int inkrementiere() {
        ...
     }
  }
```

```
class C
implements I1,I2 {
        ...
}
```

Mehrfacherbung

Mit Interfaces läßt sich etwas ähnliches wie Mehrfacherbung erreichen

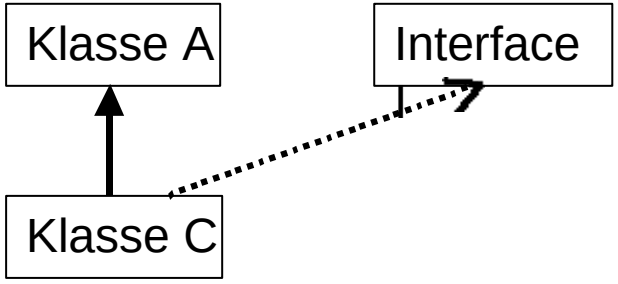



```
public class SubName
extends SuperName
implements InterfaceName,
..., InterfaceName {

   ...

}
```

Lösung: Implementierung eines Interfaces



aber Achtung:
wesentlicher Nachteil bleibt,
Klasse C muss Funktionen mit eigenem Code realisieren!

Interfaces

Interface kann als Typ für eine Referenz dienen, d.h.

Interfaces unterstützen polymorphes Verhalten wie Oberklassen.

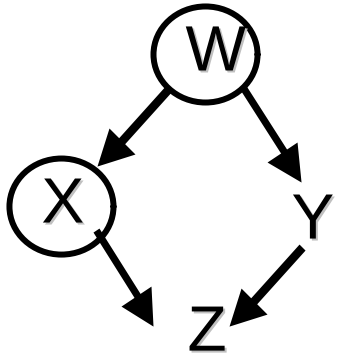
```
Girokonto dasGiro = new Girokonto();
Konto kto = dasGiro;
kto.einzahlen(500);
```

Das durch `kto` referenzierte Objekt bestimmt die Methode, die ausgeführt wird!

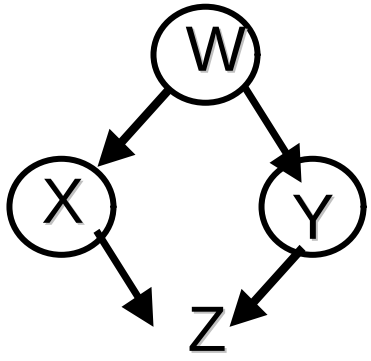
Interfaces haben keine ausgezeichnete Wurzel, von der alle Interfaces erben (anders als Klassen, die alle von `Object` erben). Dennoch können Ausdrücke eines beliebigen Interface-Typen an eine Referenz auf ein Objekt der Klasse `Object` zugewiesen werden, denn ein Objekt, das ein Interface implementiert ist halt irgendein Objekt und damit auch vom Typ `Object`.

Interfaces

```
Bsp 1:
interface W {}
interface X extends W {}
class Y implements W {}
class Z extends Y implements X  {}
```




```
Bsp 2:
interface W {}
interface X extends W {}
interface Y extends W {}
class Z implements X, Y  {}
```



Namenskonflikte bei Implementierung mehrerer Interfaces

- Was passiert, wenn eine Methode mit dem gleichen Namen in zwei zu realisierenden Interfaces vorkommt?
  - 1. bei unterschiedlichen Parametern: Überladen
  - 2. bei gleichen Parametern: nur eine Realisierung verfügbar
  - 3. bei gleichen Parameter, unterschiedlicher Rückgabetypen: unlösbarer Konflikt
- Was passiert bei gleichnamigen Attributen?
  - Referenzierung über **`InterfaceName.Attribute`**

Trennung von Deklaration und Implementierung

Deklaration

```
public interface Konto {
   public void einZahlen();
   public void ausZahlen();
   ...
}
```

```
public interface iName {
   Methodendeklarationen;
}
```

Interfaces können von anderen Interfaces erben (`extends`)

Implementierung

```
public class Girokonto
implements Konto {
   public void einZahlen() {
      ...
   }
   public void ausZahlen() {
      ...
   }
}
```

```
public class cName
implements iName {

}
```

Klassen können mehrere Interfaces implementieren (`implements i1, i2, ...`)

Interfaces versus abstrakte Klassen

- Interfaces erlauben eine Art Mehrfachvererbung.
- Eine Klasse kann nur von einer anderen Klasse erben, selbst dann wenn die vererbende Klasse nur abstrakte Methoden hat.

- Interfaces können nur **`public`** Konstanten und Methoden ohne Realisierungen beinhalten.
- Eine abstrakte Klasse kann teilweise implementiert sein, sie kann über **`protected`** Attribute, **`static`** Methoden usw. verfügen.

Zwischenstand: Interfaces

- allgemein:
  - übliches Hilfsmittel um Information Hiding zu erreichen,
    eine Aufteilung von Programmen in Pakete/Module zu ermöglichen
- speziell in Java:
  - Hilfsmittel um Restriktionen der Einfachvererbung auszugleichen
  - Achtung: Vorteile/Nachteile der Einfachvererbung bleiben erhalten!
    - pro: einfachere Struktur, leichter erkennbar, welcher Code ausgeführt wird
    - con: Code kann nur aus 1 Klasse direkt genutzt werden,
      Implementierung eines Interfaces erfordert eigenen Code,
      ggfs Hilfskonstruktion:
      Objekte nutzen Hilfsobjekte einer Klasse, die eine Schnittstelle implementiert, anstatt von der implementierenden Klasse zu erben

nochmals:

wesentlicher Nutzen ist die Möglichkeit der Modularisierung:

Aufteilen einer komplexen Aufgabe in eine Menge einfacherer Aufgaben (wie auch bereits bei Definition von Klassen möglich)

Steuerung des Kontrollflusses: Erweiterung um Ausnahmen

bisher einige Standardkonstrukte imperativer Sprachen betrachtet:

- Sequenz, Verzweigung, Iteration, Funktion

Sichtweise

- datenabhängige Verfolgung einer Befehlsfolge im Programmcode, um parameterabhängige Berechnungen durchführen zu können

Beobachtung: 2 Situationen

- Berechnung ist zielführend, erfolgt gemäß obiger Sichtweise
- Auftreten von Fehlersituationen erfordert Maßnahmen / Entscheidungen, die zwar problemabhängig erfolgen, aber nicht im ursprünglichen Sinne zu einer gewünschten Lösung führen

- in imperativer Programmierung:
  - beide Situationen auf gleiche Art behandelt,
  - im Code vermischen sich diese Anteile,
    - z.B. bei Funktionsrückgabewerten entweder Ergebniswerte oder Fehlercodes

im folgenden: Behandlung von Ausnahmen in objektorientierter Programmierung am Bsp von Java

Ausnahmen - Wieso eigentlich?

(Etwas naive) Annahme:

- Algorithmen sind so entworfen, dass alle denkbaren Zustände behandelt werden. Ungewollte Terminierungen (Abstürze) können deshalb gar nicht vorkommen.

  - Beispiel: beim Einfügen eines Elementes in eine Liste werden alle denkbaren Fälle (Einfügen in leere Liste, Einfügen vor erstem Listenelement, Einfügen zwischen Listenelemente, Einfügen hinter dem letzten Listenelement) unterschieden und korrekt behandelt.

- Wozu also Ausnahmen?

Ausnahmen - Wieso eigentlich?

- Algorithmen durch Zusammenwirken von Objekten realisiert.
  - Objekte entscheiden autonom über die Ausführung von Methoden, können dabei aber nicht den Gesamtzustand des Systems berücksichtigen (lokale Sicht).

    => Ausnahmen möglich, falls lokale Sicht unzureichend.
- Alle globalen Zustände können nicht berücksichtigt werden:
  - kombinatorische Explosion der Möglichkeiten bei vielen Objekten mit internen Zuständen
- ein pragmatischer Grund:
  - Klarheit von Programmen,
  - Vermischung von Algorithmus und Ausnahmebehandlung erhöht Komplexität

Fazit:

Code für eigentlichen Algorithmus und Ausnahmebehandlung sollten innerhalb einer Programmiersprache bereits unterschieden werden.

Übersicht

- Was sind Ausnahmen? / Grundidee der Ausnahmebehandlung
  - Die Klasse java.lang.Exception
- „Werfen“ von Exceptions
- Weiterleiten von Exceptions
- „Fangen“ von Exceptions
- Die Klassen Exception, Error, RuntimeException

Was sind Ausnahmen?

- Ausnahmen können im Programmablauf auftreten
  - Division durch Null
  - Datei nicht vorhanden
  - ...
- Ziel ist aber: Erstellen eines stabilen Programms
- Ausnahme: Zustand, der das Programm daran hindert, im normalen Ablauf fortzufahren

- Trennung von Fehlerbehandlungscode und regulärem Code
- Es kann leichter sein, Ausnahmen eintreten zu lassen und sie ordentlich zu behandeln als sie unbedingt zu vermeiden.

Grundidee der Ausnahmebehandlung

- Es gibt vordefinierte Ausnahmen, diese können erweitert werden.
- In der Definition von Methoden wird definiert, welche Ausnahmen auftreten können.
- Beim Aufruf von Methoden wird definiert, wie die möglicherweise auftretenden Methoden behandelt werden. Hierzu wird der potentiell Ausnahmen verursachende Code in einen **`try{...}`**-Block eingebettet.
- Wenn im **`try{...}`**-Block Ausnahmen auftreten, werden sie gefangen. Was dann passiert, wird im **`catch{...}`**-Block festgelegt.
- Nach dem Fangen und Verarbeiten wird im **`finally{...}`**-Block definiert, was zum Abschluß der Ausnahmebehandlung passiert.
- Und nun erst mal ein Beispiel!

Die Klasse java.lang.Exception

```
public class Exception ... {
   public Exception() {..};
   public Exception(String s) {...};
   // und von java.lang.Throwable geerbt:
   // Kurzbeschreibung des Fehlers
   public String toString() {...};

   // Details des Fehlers
   public String getMessage() {...};

   // Details mit Aufrufstack
   public String printStackTrace() {...};
   ...
}
```

„Werfen“ von Ausnahmen

```
if (kontostand-betrag > dispo) {
     zahle(-betrag);
     return (betrag);
} else throw new Exception();
```

throw new *Exception ();*
throw new *Exception(„Text“);*

Definition des Werfens von Ausnahmen in Methoden

```
Modifier Rückgabetyp Methodenname (...) throws Exception {
```

```
public int auszahlen(int betrag) throws Exception {
   Datum heute = Datum.aktuellesDatum();
   int zinstage = heute.gibDifferenz(letzteTransaktion);
   int zinsen = berechneZinsen(zinstage);
   zahle(zinsen);
   if (kontostand-betrag > dispo) {
       zahle(-betrag);
       return (betrag);
   } else throw new Exception(“Konto nicht gedeckt“);
}
```

Definition des Werfens von Ausnahmen in Methoden

- Ausnahmen, die geworfen werden können, müssen deklariert werden,
  - damit Ausnahmeverhalten ebenso bekannt wird wie normalesVerhalten (Signatur).
  - in der Methodendefinition deklarierten Ausnahmen heißen „checked“
- Achtung, Laufzeitausnahmen: **`RuntimeException`** (z.B. **`ClassCastException`**, **`ArithmeticException`**) nicht explizit deklariert, können von jeder Methode geworfen werden,

  => nicht vom Compiler überprüft („unchecked“),

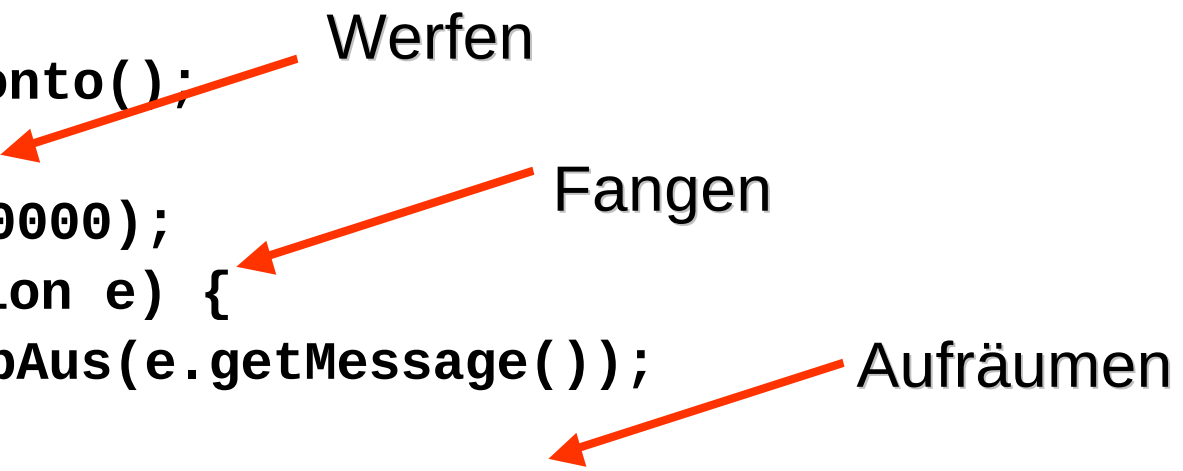
  Exceptions aus **`java.lang.Error`**, **`java.lang.RuntimeException`** müssen nicht in **`throws`**-Deklaration auftreten.
- Nutzung von Methoden, die Ausnahmen werfen können, erlaubt 3 Fälle:
  - Fangen und Behandeln
  - Fangen, Abbilden auf eigene Ausnahme, Werfen der eigenen Ausnahme
  - Deklarieren der Ausnahme in der aufrufenden Methode und dann Fangen und Weiterleiten der Ausnahme

  also stets: Fangen von Ausnahmen!

Fangen von Ausnahmen

- Mit **`try`** und **`catch`** werden Exceptions aufgefangen
  - **`try`**-Block: Bereich, in dem Ausnahmen auftreten können
  - **`catch`**-Block: Bereich, in dem die Fehlerbehandlung stattfindet
  - **`finally`**-Block: optional, wenn vorhanden, wird er auf jeden Fall ausgeführt

```
...
Konto k = new Konto();
try {
  k.auszahlen(10000);
} catch (Exception e) {
  Bildschirm.gibAus(e.getMessage());
} finally {
  Bildschirm.gibAus(k.holeKontostand());
}
```



Fangen von Ausnahmen

```
try {
      statements
} catch (exception_type1 identifier1) {
      statements
} catch (exception_type2 identifier2) {
      statements
....
} finally {
      statements
}
```

Mehrere catch-Blöcke!

Ablauf für das Verarbeiten von Ausnahmen

- Rumpf eines **`try`**-Blocks bis zum Ende ausgeführt (falls keine Ausnahme auftritt) oder: falls Ausnahme auftritt
  - durchsuche **`catch`**-Blöcke (von oben nach unten), nach passendem.
  - bei 1. passendem **`catch`**-Block, setze Identifier auf das aufgetretene Exception-Objekt
  - fehlt passender **`catch`**-Block, Ausnahme nach außen liegende **`try`**-Blöcke durchreichen
- **`finally`**-Block nach Beendigung **`try`**-Block ausgeführt,
  - durch normale Beendigung, Ausnahme oder **`return/break`**.
  - sinnvoll für Resourcenfreigabe (z. B. Schließen einer Datei), denn Programmcode außerhalb des **`try`**-Blocks nicht mehr erreicht, falls innerhalb des **`catch`**-Blocks wieder Ausnahme geworfen wird.

Ausnahmebehandlung

- Ausnahmen können auftreten durch
  - explizite **`throw`**-Anweisung
  - Aufruf einer Methode mit **`throws`**- Deklaration
- Auffangen und Weiterwerfen (**`throw`** im **`catch`**-Block)
  - sinnvoll, um wichtige Zusatzinformationen anzuhängen
- Kann in einer Methode eine Ausnahme auftreten, muß sie
  - entweder durch **`try`** und **`catch`** aufgefangen werden
  - oder durch **`throws`** in der Methodendeklaration weitergeleitet werden

Exception, Error, RuntimeException

- Exception:
  - nicht für ernsthafte, kritische Fehler
  - Einsatz als Feature im Programmablauf
- Error:
  - Schwerwiegende Fehler der Virtual Machine
  - Sollten nicht selbst geworfen oder aufgefangen werden
- RuntimeException
  - Systemfehler, die nicht deklariert werden müssen
  - Beispiel: Teilung durch Null (0)

Schreiben eigener Ausnahmen

Neue Klasse wird von java.lang.Exception abgeleitet :

```
public class KeineDeckungException extends Exception {
   public KeineDeckungException() {super();}
   public KeineDeckungException(String s) {super(s);}
}
public class Girokonto extends Konto { ...
   public int auszahlen(int betrag) throws KeineDeckungException {
        ...
        if (kontostand-betrag > dispo) {
               zahle(-betrag);
               return (betrag);
        } else throw new KeineDeckungException("Konto nicht
        gedeckt");
   }
}
```

Auffangen der Exception:

```
try {
   k.auszahlen(1000);
} catch(KeineDeckungException e){Bildschirm.gibAus(e.getMessage());}
```

Zwischenstand

- Interfaces
  - allgemeines Konzept zur Modularisierung
  - in Java: zusätzlich speziell zur Ergänzung des Vererbungskonzeptes
- Exceptions
  - normaler Algorithmus vs Fehlerbehandlung
  - in Java: syntaktisch getrennt, try-catch-finally und throws
  - erlaubt klarere Struktur von Programmen,
  - erlaubt einfacheres Erkennen der Ausnahmebehandlung:
    - welche Ausnahmen werden wie berücksichtigt

Klassenbibliothek

- Umfangreiche Klassenbibliotheken unterstützen die Nützlichkeit einer Sprache (hier: Java).
  - erhöhen die Produktivität durch Code Sharing
  - verringern die Fehleranfälligkeit
- Klassenbibliothek
  - plattformübergreifend einsetzbar
  - in Pakete gegliedert, dadurch handhabbar.
  - Pakete können wie selbstdefinierte Pakete verwendbar.
- Hier nur Einblick in grobe Einteilung und wichtige Funktionalitäten am Bsp von Java
  - Vollständige Übersicht in "JDK 1.2 Documentation"
    => Nachschlagewerk

Übersicht: Die Java Klassenbibliotheken

- `java.applet` (JDK 1.0) Java-Applets (für WWW-Browser)
- `java.awt` (JDK 1.0) AWT-Komponenten
- `java.beans` (JDK 1.1) Beans-Development
- `java.io` (JDK 1.0) Input / Output / Datenströme
- `java.lang` (JDK 1.0) Basispackage für Java
- `java.math` (JDK 1.1) Zahlen großer Genauigkeit
- `java.rmi` (JDK 1.1) Remote Method Invocation
- `java.security` (JDK 1.1) Sicherheit, Kryptographie
- `java.sql` (JDK 1.1) JDBC-Paket
- `java.util` (JDK 1.0) Verschiedene Hilfsklassen
- `javax.swing` (JDK 1.2) Swing-Komponenten
  Für JDK 1.1.x: `com.sun.java.swing`
- `org.omg.CORBA` (JDK 1.2) CORBA-Schnittstelle

Einige Beispiele: java.lang

- Basispaket der Sprache
- alle Systemklassen sind hier untergebracht
- Klassen dürfen ohne `import`-Deklaration direkt angesprochen werden (vgl. System, String)

`java.lang: Object`, `String` und `StringBuffer`

- Klasse `Object`
  - Die Mutter aller Klassen
- Klassen `String` und `StringBuffer`
  - Zeichenketten

Einige Beispiele: `Class`

Klasse `Class`

- Klasse um Klassennamen und Instanzen zu verwalten
- Die Klasse `Object` hat die Methode
  `public final Class getClass()`
  um zugehöriges Class-Objekt zu erhalten
- Methode `public String getName()` gibt Namen der Klasse zurück
- Es lassen sich mit Hilfe des Class-Objektes verfügbare Konstruktoren und Methoden ermitteln (seit JDK 1.1)
- Neue Objekte der Klasse können so erzeugt werden, ohne daß man Vorabinformationen hat (seit JDK 1.1)

Einige Beispiele: `System`

Klasse `System`

- Sammlung von Klassenattributen und -methoden
- Anbindung an Standard-In, Standard-Out und Standard-Error-Stream (`System.in`, `System.out` und `System.err`)
- Systemnahe Methoden
- `public static long currentTimeMillis()`
  - gibt Millisekunden seit 1. Janur 1970 zurück
- `public static void exit(int status)`
  - Beendet das Programm (und die Virtuelle Maschine)
- `public static void gc()`
  - Empfehlung an den Garbage - Kollektor, Speicher freizugeben. In der Regel nicht nötig.

Einige Beispiele: java.lang

- `java.lang.System`
  - Anbindung an Standard-Err, Standard-In, Standard-Out-Stream

```
{
...
System.out.println(“Hallo”);
...
}
```

```
public static final PrintStream err
public static final PrintStream in
public static final PrintStream out
```

  - Systemnahe Methoden

```
public static void exit(int status)
public static void gc()
```

Einige Beispiele: Einhüllende Klassen

- Klassen `Boolean, Byte, Character, Double, Float, Integer, Long, Short`
- Nützlich, wenn primitive Datentypen als Objekt benötigt werden, z.B. für objektbasierte Datenstrukturen
- Alle einhüllenden Klassen haben den entsprechenden einfachen Datentypen als Konstruktor
  - Beispiel:
    ```
    Float zahl=3.5;
    Float zahlHülle = new Float(zahl);
    ```
- Alle einhüllenden Klassen (bis auf Character) haben `String` als Konstruktorargument
  - Beispiel:
    ```
    Long l = new Long("2.142352123");
    ```

Einige Beispiele: Einhüllende Klassen

- `Byte`, `Double`, `Float`, `Integer`, `Long` und `Short` (sowie `java.math.BigDecimal` und `java.math.BigInteger`) sind von abstrakter Klasse `Number` abgeleitet
- Für Objekte der Klasse `Number` sind Wandlungen zu primitiven Datentypen möglich mit `byteValue()`, `doubleValue()`, `floatValue()`, `intValue()`, `longValue()` und `shortValue()`
- Beispiel: Konvertierung von `String` nach `double`

```
String s = "1.345";
double d = new Double(s).doubleValue();
```

Einige Beispiele: java.lang

- `java.lang.Math`
  - Konstanten `E` und `Pi`
  - Methoden für Betrag, Rundung, Trigonometrische Funktionen, Exponent / Logarithmus, etc.

```
public static int abs(int a)
public static native double sin(double a)
public static native double sqrt(double a)
```

Zusammenfassung

- Interfaces
  - allgemeines Konzept zur Modularisierung
  - in Java: zusätzlich speziell zur Ergänzung des Vererbungskonzeptes
- Exceptions
  - normaler Algorithmus vs Fehlerbehandlung
  - in Java: syntaktisch getrennt, try-catch-finally und throws
  - erlaubt klarere Struktur von Programmen,
  - erlaubt einfacheres Erkennen der Ausnahmebehandlung:
    - welche Ausnahmen werden wie berücksichtigt
- Klassenbibliotheken, Pakete
  - umfangreiche, plattformunabhängige Bibliothek erlaubt produktive SW Erstellung (hoffentlich)