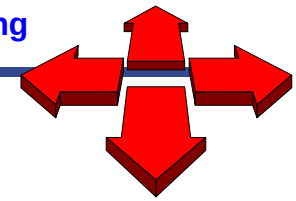
Hinweis zu den Übungen

- In dieser Woche wird das letzte Aufgabenblatt der **Laborphase 1** ausgegeben und bearbeitet, Blatt 5.
- Da in der nächsten Woche die **Laborphase 2 in 4er-Gruppen** beginnt, habt ihr für das **Blatt 5 nur maximal eine Woche** zur Bearbeitung!
- Wer also mit seinen Abnahmen hinterher hängt, sollte in dieser Woche Gas geben!

SE2 – OOPM – Teil 1 2

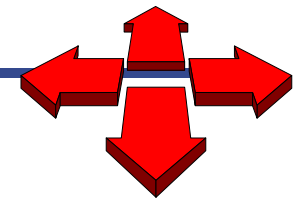
Heute: Fehlerbehandlung / Modularisierung



- Fehlerkategorien
- Traditionelle Techniken der Fehlerbehandlung
- Ausnahmen (Exceptions)

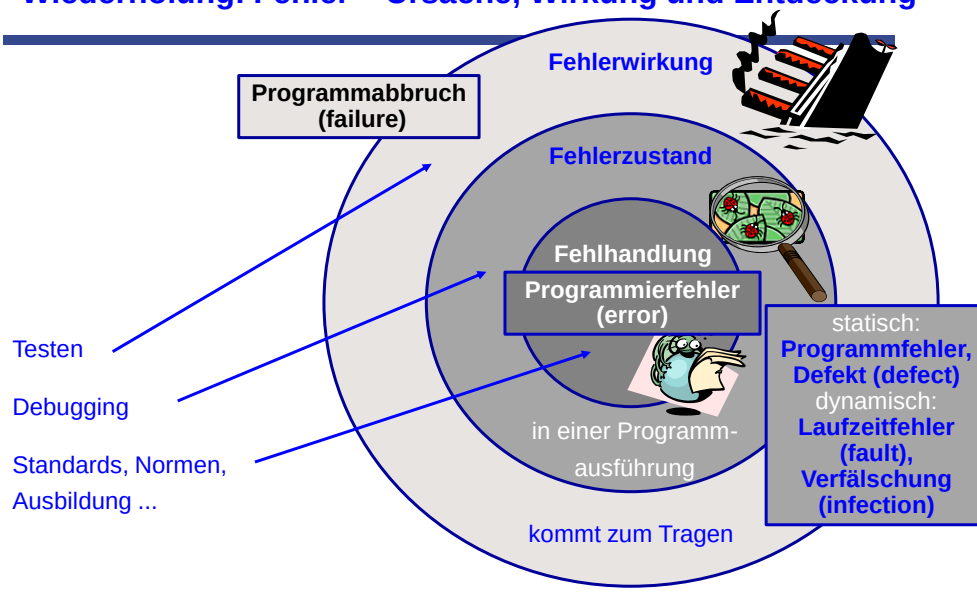
- Das Modulkonzept
- Javas Klassen als Module
- Javas Pakete als Module

Fehlerbehandlung mit Exceptions



- Fehlerkategorien
- Traditionelle Techniken der Fehlerbehandlung
- Grundsätzliches zur Ausnahmebehandlung
- Exceptions in Java: Auslösen und Behandeln
- Javas Exception-Hierarchie

Wiederholung: Fehler – Ursache, Wirkung und Entdeckung



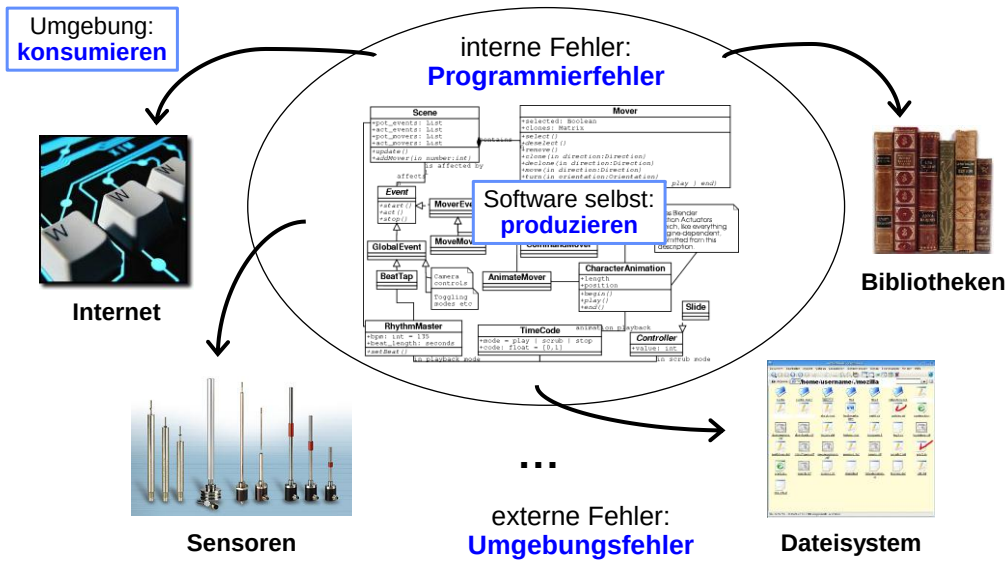
Fehlersituationen in Programmen

- In Softwaresystemen kann es zu vielfältigen Fehlersituationen kommen, beispielsweise:
  - Eine Implementation erfüllt nicht die Anforderungen ihrer Spezifikation.
  - Bei einem schreibenden Zugriff auf eine Datei hat ein Programm keine Schreibrechte für die Datei.
  - Ein Klient hält sich bei der Anforderung einer Dienstleistung nicht an die Vorbedingungen.
  - Für den Zugriff auf eine Webseite hat der Browser keinen Zugang zum Internet.
  - Bei einem Array-Zugriff wird ein ungültiger Index benutzt.
  - Durch konkurrierenden (gleichzeitigen) Zugriff zweier Prozesse auf ein Objekt gerät dieses in einen inkonsistenten Zustand.
- **Wie können wir bei der Softwareentwicklung systematisch (konstruktiv) mit solch unterschiedlichen Fehlersituationen umgehen?**

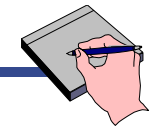
Softwareentwicklung: Einflussebenen

- Bei der Softwareentwicklung geht es um das **Erstellen** oder **Bearbeiten** von Software.
- Häufig wird Software im **Team** bearbeitet.
- Reale Software läuft in einer **Umgebung**, die sich sehr unterschiedlich zusammensetzen kann: Datenbanken, Dateisysteme, Sensoren, Bibliotheken, das Internet und nicht zuletzt Menschen als die Benutzer interaktiver Software haben teilweise großen Einfluss.
- Aus Sicht des Entwicklungsteams können zwei Einflussebenen unterschieden werden:
  - **Unmittelbaren (verändernden) Einfluss** hat das Team nur auf den bearbeiteten Quelltext: Wie viele Klassen gibt es, welche Konventionen werden eingehalten, wird das Vertragsmodell eingesetzt, …
  - **Kaum Einfluss** (meist nur konsumierend) hat das Team auf die Umgebung: Bibliotheken können (meist) nicht selbst verändert werden, Datenbanken sind oft nicht unter der Kontrolle der Softwareentwicklung, das Dateisystem des Produktivsystems wird von Administratoren verwaltet, …

Interne und externe Fehler bei der Softwareentwicklung



Programmierfehler versus Umgebungsfehler



- Aus Sicht eines Programmierteams können wir demnach unterscheiden:
  - **Programmierfehler**
  - **Umgebungsfehler**
- **Programmierfehler** werden bei der Entwicklung vom Team gemacht. Obwohl sie prinzipiell vermeidbar sind, treten sie auf. Programmierfehler, die entdeckt werden, werden behoben und sind dann weg. Zur Laufzeit kann es für sie keinen **korrekten** Umgang in der Software selbst geben; das ausprogrammierte Behandeln eines Programmierfehlers in der Software dient ausschließlich der **Robustheit** der Software!
- **Umgebungsfehler** liegen außerhalb des Einflussbereichs des Teams und können somit **nicht** von diesem **behoben** werden. Sie können **immer wieder** auftreten und sind **prinzipiell vorhersehbar**; es können deshalb **Vorkehrungen in der Software** getroffen werden, um mit diesen Fehlern umzugehen.

Umgebungs- oder Programmierfehler?

Noch einmal die Beispiele:

P • Eine Implementation erfüllt nicht die Anforderungen ihrer Spezifikation.

U • Bei einem schreibenden Zugriff auf eine Datei hat ein Programm keine Schreibrechte für die Datei.

P • Ein Klient hält sich bei der Anforderung einer Dienstleistung nicht an die Vorbedingungen.

U • Für den Zugriff auf eine Webseite hat der Browser keinen Zugang zum Internet.

P • Bei einem Array-Zugriff wird ein ungültiger Index benutzt.

P • Durch konkurrierenden (gleichzeitigen) Zugriff zweier Prozesse auf ein Objekt gerät dieses in einen inkonsistenten Zustand.

U • Ein Benutzer tippt in ein Eingabefeld einer grafischen Oberfläche Daten im falschen Format.

Umgang mit Fehlersituationen

- Wir können in objektorientierten Systemen zwei Situationen unterscheiden:
  - Das Erkennen und Signalisieren eines Fehlers durch einen **Dienstleister**.
  - Das Behandeln eines Fehlers durch einen **Klienten**.
- Für das **Erkennen und Signalisieren** stellen sich Fragen wie:
  - Wann, wie, wie häufig soll ein Dienstleister prüfen, ob er korrekt verwendet wird?
  - Was soll er tun, wenn ein Klient sich fehlerhaft verhält?
- Beim **Behandeln** stellen sich Fragen wie:
  - Wie kann sich ein Klient auf mögliche Fehler einstellen?
  - Wie soll der Klient auf Fehler reagieren? Kann er nach Fehlerart differenzieren und unterschiedlich reagieren?

Traditionelle Ansätze zur Fehlerbehandlung

- Traditionell werden Fehlersituationen mit Hilfe der folgenden Techniken behandelt:
  1. Das Programm wird aus der fehlerhaften Methode heraus **abgebrochen**.
  2. Die fehlerhafte Methode liefert einen **Return-Wert**, der vom Aufrufer als Fehler-Code interpretiert werden soll.
  3. Die fehlerhafte Methode setzt eine **global lesbare Variable** auf den eingetretenen Fehlerstatus.
  4. Die fehlerhafte Methode ruft eine sog. **Callback-Operation**, die zuvor bei ihr für den Fall eines auftretenden Fehlers bekannt gemacht worden ist.

Kritik an den traditionellen Ansätzen

- Die traditionellen Techniken besitzen folgende Nachteile:
  1. **Abbruch**: Bei vielen Anwendungssystemen ist ein Programmabbruch nicht akzeptabel, etwa bei **sicherheitskritischen** Systemen (z.B. Flugzeugsteuerungen).
  2. **Return-Wert**: Nicht immer lässt sich ein **geeigneter Wert** finden, der als Fehlercode interpretiert werden kann, z.B. wenn alle Werte des deklarierten Rückgabetyps auch sinnvolle Rückgabewerte sind. Prinzipiell muss jeder Operationsaufruf auf einen Fehlerfall abgeprüft werden, dies erscheint vielen Programmierern als **inakzeptabler Aufwand**.
  3. **Globale Fehlervariable**: Fehlerhafte Operationen mit ungeprüften Fehlermeldungen werden aus Sicht eines Aufrufers **scheinbar "normal"** ausgeführt, d.h. der Fehler wird nicht erkannt.
  4. **Callbacks**: Fehlerbehandlung über Callback-Operationen weist bereits in die Richtung einer Ausnahmebehandlung, **bläht** aber die **Schnittstellen** von Operationen unnötig **auf**.

Ausnahmen als eigenständiges Sprachkonzept

- In Programmiersprachen wurden für das systematische Umgehen mit Fehlersituationen eigene Sprachkonzepte für so genannte **Ausnahmen** (engl.: exceptions) entwickelt.
  - Eine der ersten Programmiersprachen mit einem Exception-Konzept war **CLU**; die erste weiter verbreitete Sprache war **Ada**.
  - **Eiffel** war die erste objektorientierte Programmiersprache mit Exceptions.
- Mindestens folgende **Ziele** sollen mit einer Sprachunterstützung erreicht werden:
  - Fehler sollen nicht einfach ignoriert werden können.
  - Der Quelltext zur Fehlerbehandlung soll den Normalfall nicht unnötig „verschütten".
  - Durch eine Typisierung der Fehler sollte je nach Fehlerart unterschiedlich reagiert werden können.

Exceptions in Java sind Objekte

- Eine Exception ist in Java ein **Exemplar einer Exception-Klasse**.
- Es existieren etliche **vordefinierte Exception-Klassen** in den Java-Bibliotheken.
  - Jede Exception hat damit einen **Typ**, der sich zur Laufzeit abfragen lässt; dies ermöglicht differenziertes Reagieren auf verschiedene Fehlerarten.
- Ein Programmierer kann zusätzlich **eigene** Exception-Klassen definieren.
  - Dies hat den Vorteil, dass die Exemplare dieser Klassen mit beliebigen **zusätzlichen Informationen** über den Grund der Fehlersituation versehen werden können.

Geprüfte und ungeprüfte Exceptions

- In Java wird grundsätzlich unterschieden zwischen **geprüften** und **ungeprüften** (engl.: checked and unchecked) Exceptions.
- Wenn eine Operation in ihrer Implementation eine **geprüfte Exception** auslösen könnte, muss sie dies **deklarieren**. Geprüfte Exceptions werden für vorhersehbare Fehler eingesetzt, also primär für **Umgebungsfehler**.
  - Beispiele für geprüfte Exceptions: `ServerNotActiveException`, `IOException`, `TimeoutException`
- **Ungeprüfte Exceptions** müssen hingegen **nicht deklariert** werden. Sie werden primär für **Programmierfehler** definiert, da diese Fehler praktisch jederzeit auftreten können und ihre Deklarationen den Quelltext überschwemmen würden.
  - Beispiele für ungeprüfte Exceptions: `ArrayIndexOutOfBoundsException`, `NullPointerException`, `OutOfMemoryException`

Das „geprüft“ und „ungeprüft“ ist **statisch** zu verstehen, denn es bezieht sich auf die **Übersetzungszeit**. Zur Laufzeit führen alle Exceptions zu einer Spezialbehandlung in der Virtual Machine.

Javas Schlüsselwörter für die Ausnahmebehandlung

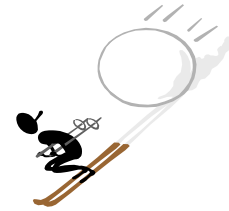
`throw`	**Auslösen** einer Exception.
`throws`	**Deklarieren** eine (geprüften) Exception an der Operationsschnittstelle.
`try`	Einleiten eines Blocks, in dem mit dem **Auftreten einer Exception** gerechnet wird bzw. werden muss.
`catch`	Einleiten eines Blocks, in dem eine aufgetretene Exception **behandelt** (engl.: to handle) werden kann. Der catch-Block wird deshalb auch als **Exception-Handler** bezeichnet.
`finally`	Einleiten eines Blocks, der **immer ausgeführt** wird, wenn der try-Block betreten wurde, selbst wenn dort eine Exception zum vorzeitigen Beenden des try-Blocks geführt hat und ein catch-Block ausgeführt wurde.

Ausnahmebehandlung in Java (1)

- Grundsätzliches zum Auslösen von Exceptions:
  - Eine **Ausnahme** bzw. **Exception** ist ein Signal, dass irgendeine Ausnahmebedingung aufgetreten ist, z.B. ein Fehler beim Öffnen einer Datei oder das Überschreiten von Array-Grenzen.
  - Eine Ausnahme **auszulösen** bedeutet, eine Ausnahmebedingung zu signalisieren.
  - Die auslösende Methode wird abgebrochen und es wird zur aufrufenden Methode **zurückgesprungen**.
  - Eine Ausnahme wird auf diese Weise von Methode zu Methode **weiterpropagiert** und durchläuft die Methoden-Aufrufkette, bis sie abgefangen wird.

Ausnahmebehandlung in Java (2)

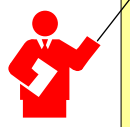
- Grundsätzliches zum Behandeln:
  - Eine Ausnahme **abzufangen** (`catch`) heißt, sie zu behandeln; entweder werden Aktionen durchgeführt, die den normalen Zustand wieder herstellen, oder es wird organisiert abgebrochen, inklusive möglicher Aufräumarbeiten.
  - Wird eine Ausnahme überhaupt nicht abgefangen, so bahnt sie sich ihren Weg bis zum Aufruf der **`main`-Methode**.
  - Fällt sie durch die **`main`**-Methode, wird der Java-Interpreter eine Fehlermeldung ausgeben und die Programmausführung **abbrechen**.



Das Auslösen einer Exception

- Wann löst man eine Exception aus?
  - Eine Methode, die ihre Aufgabe **nicht erfüllen** kann (z.B., weil bestimmte Bedingungen in ihrer Umgebung nicht erfüllt sind), kann eine Exception auslösen.
  - Bevor sie eine Exception auslösen kann, muss sie ein **Exception-Objekt erzeugen**. Erst durch das **Werfen** dieses Objektes (mit dem Schlüsselwort `throw`) wird eine Exception ausgelöst.
  - **Spezialfall geprüfte Exception**: Für eine geprüfte Exception muss die Methode in ihrem Kopf **deklarieren** (mit dem Schlüsselwort `throws`), dass sie diese Exception im Bedarfsfall auslösen wird.

Das Umgehen mit geprüften Exceptions



Regel:

Wenn eine Methode A eine andere Methode B benutzt, die eine geprüfte Exception deklariert, dann müssen in der Methode A Vorkehrungen für den Fall getroffen werden, dass die deklarierte Exception wirklich auftritt.

- Es gibt zwei Möglichkeiten für die Methode A:
  - Sie kann die Exception **behandeln**.
  - Sie kann selbst deklarieren, dass sie diese Exception auslöst, sie also **weiterpropagieren**.

Das Behandeln von Exceptions

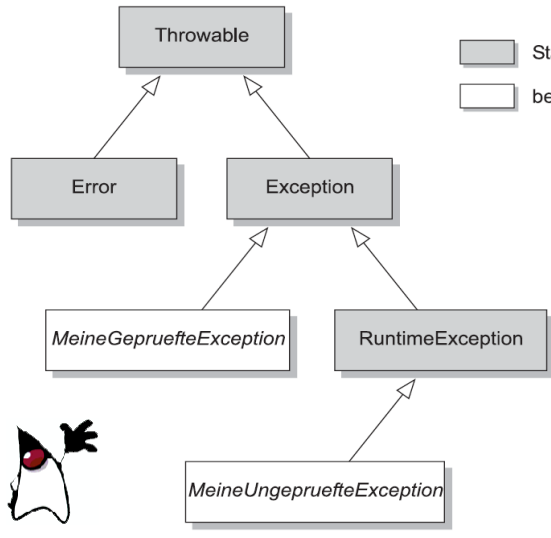
- Allgemeine Struktur eines Exception-Handlers:

```
try
{
    // Anweisungen,
    // die Exceptions auslösen können
}
catch ( <Exception-Typ> e)
{
    // Behandeln der Fehlersituation
}
finally
{
    // Anweisungen, die in jedem Fall
    // ausgeführt werden sollen
}
```

Ausnahmen differenziert behandeln

- Mit welchem Exception-Handler ?
  - Auf einen try-Block können **mehrere** catch-Blöcke folgen.
  - Für jeden catch-Block wird der Typ der Exceptions spezifiziert, die er behandeln soll.
  - Es wird **maximal ein** Exception-Handler ausgeführt.
  - Bei mehreren möglichen Exception-Handlern wird der **erste passende** gewählt.
  - Ein catch-Block passt zu einer Exception, wenn der Typ der erzeugten Exception gleich oder ein **Subtyp** ist.

Vererbung: Ausschnitt aus Javas Exception-Hierarchie



- Alles, was mit `throw` geworfen werden kann, **ist ein** `Throwable`.
- Exceptions der Klasse `Error` oder ihrer Unterklassen weisen auf Probleme der virtuellen Maschine hin, z.B. einen Speicherüberlauf. Sie sind nicht zu beheben und sollten daher auch nicht abgefangen werden.
- Die `RuntimeException`s stehen für Programmierfehler, die nicht deklariert werden müssen, aber dürfen.

Beispiel: Eigene Exception-Klasse definieren

- Die Klasse **`Exception`** verfügt über ein String-Attribut, das mit einem Konstruktor gesetzt werden kann. Wir können es über einen super-Aufruf zum Ablegen eines **Fehlertextes** nutzen.
- Mit der von **`Exception`** geerbten Operation **`getMessage()`** kann ein Klient diesen Text auslesen.

```
class RangeException extends Exception
{
  public RangeException() { super(); }
  public RangeException(String s){super(s); }
  public RangeException(int i)
  {
    super("Access at position " + i + " was invalid.");
  }
}
```

SE2 – OOPM – Teil 1 25

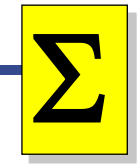
Exceptions als Abstraktionshilfe

"From the Software Engineering point of view, one can regard **exceptions** and **exception handling** as yet another technique for **software abstraction**: being able to abstract away rare, special cases during a first pass in writing and understanding a program. The **anticipated exceptions and their handlers** can then be considered as refinements to the program which appear as **footnotes**, ..."

Borgida, A: "Exceptions in Object-oriented Languages", SIGPLAN Notices 21:10, S. 107-119, 1986.

SE2 – OOPM – Teil 1 26

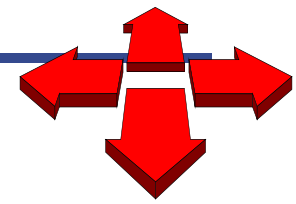
Zusammenfassung



- In Softwaresystemen kann es sehr **verschiedene Fehlerursachen** für auftretende Fehler geben.
- Eine grundsätzliche Unterscheidung aus Sicht der Softwareentwicklung ist die Unterscheidung in **Umgebungsfehler** und **Programmierfehler**.
- Für den systematischen Umgang mit Fehlern wurde für Programmiersprachen das Konzept der **Ausnahmen** bzw. **Exceptions** entwickelt.
- In Java sind Exceptions **Exemplare von Exception-Klassen**, die zur Laufzeit im Fehlerfall erzeugt und geworfen werden.
- In Java kann in **Exception-Handlern** jede geworfene Exception zur Laufzeit gefangen und behandelt werden.

SE2 – OOPM – Teil 1 27

Namensräume und Modularisierung



- Das Modulkonzept
- Javas Klassen als Module
- Javas Pakete als Module

Modularisierung	Schnittstelle/Schutz	Benennung
Klasse		
Paket		

SE2 – OOPM – Teil 1 28

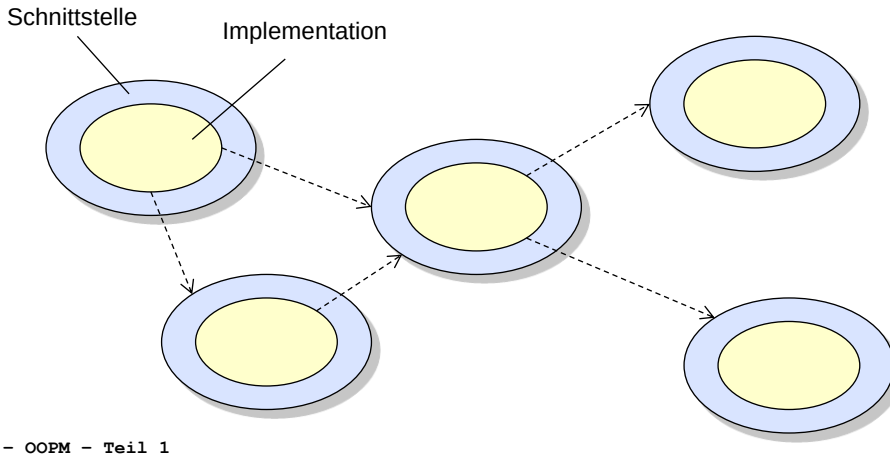
Das Modulkonzept

- Die klassische imperative Programmierung ist eng mit dem **Modulkonzept** verknüpft.
- Entstanden aus der Notwendigkeit, große Programmtexte in für den Übersetzer fassliche Einheiten zu zerlegen, wurde das Modulkonzept **zum zentralen Organisationskonzept für Entwürfe und Programmtexte**.
- Für nicht-objektorientierte Sprachen sind Module (soweit in der Sprache vorhanden) die Programmeinheiten, in denen fachliche oder technische Entwurfsentscheidungen gekapselt werden (Geheimnisprinzip).
- Die neueren Entwicklungen bei objektorientierten Programmiersprachen zeichnen sich u.a. durch eine neue Interpretation des Modulkonzepts aus (wir stellen das Package-Konzept in Java vor).
- Module werden hier als wichtiges Konstruktionsmerkmal der imperativen Programmierung in Verbindung zur objektorientierten Programmierung gesetzt.

SE2 – OOPM – Teil 1 29

Die Grundidee: ein Software-System besteht aus Modulen

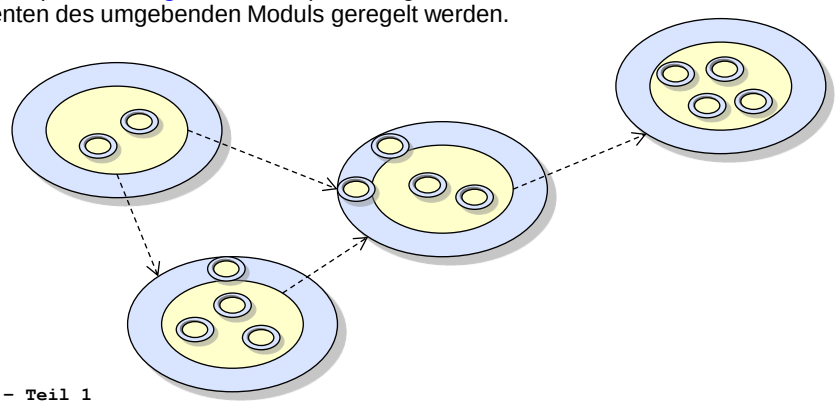
- Module sind statische Einheiten des Quelltextes mit einer **Schnittstelle** und einer **Implementation**. Sie benutzen sich gegenseitig ausschließlich über ihre Schnittstellen.



SE2 – OOPM – Teil 1 30

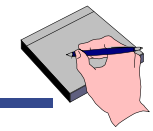
Weiteres Konzept: Module können Module enthalten

- In großen Systemen sind Module häufig auf **mehreren Ebenen** (beispielsweise Schichten, Subsysteme, Pakete) definiert, um die hohe Anzahl an Einheiten strukturierbar zu halten.
- Wenn Module selbst wieder Module enthalten können, dann muss der Zugriff (**Benennung** und **Schutz**) auf die geschachtelten Module für Klienten des umgebenden Moduls geregelt werden.

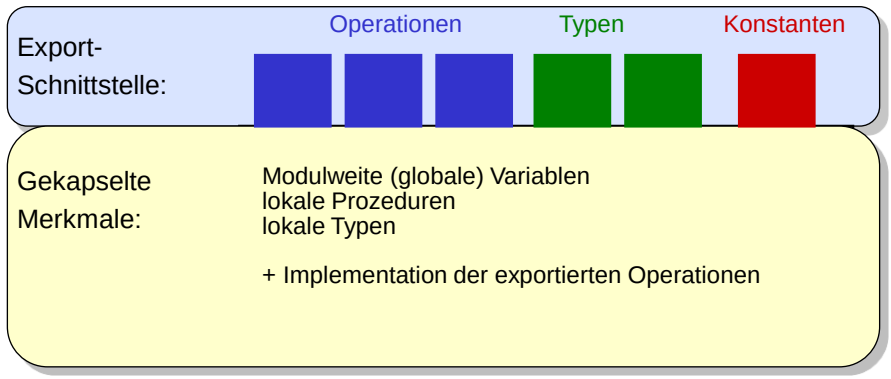


SE2 – OOPM – Teil 1 31

Klassisch: Modul – Exportschnittstelle

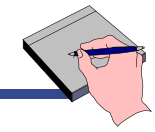


- Um Dienstleistungen über Modulgrenzen hinaus verwendbar zu machen, müssen sie nach außen bekannt gemacht werden.
- Die **Exportschnittstelle** definiert, welche deklarierten Bestandteile ein Modul seiner Umgebung zur Verfügung stellt.

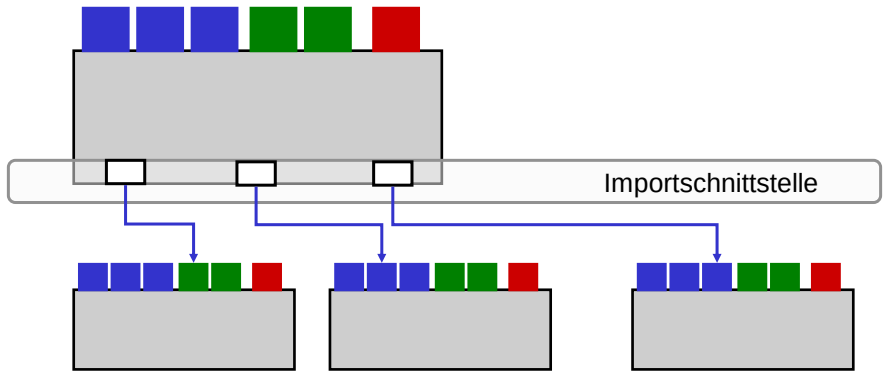


SE2 – OOPM – Teil 1 32

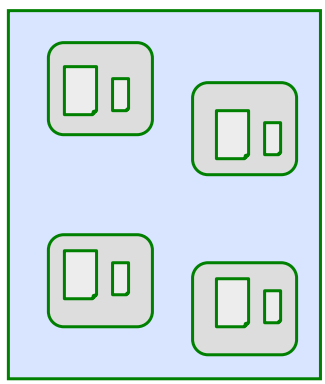
Klassisch: Modul – Importschnittstelle



- Um Programmobjekte und ihre Bezeichner über Modulgrenzen hinaus zu verwenden, müssen sie gezielt angefordert werden.
- Die **Importschnittstelle** beschreibt explizit, welche deklarierten und exportierten Objekte ein Modul von seiner Umgebung benötigt.



Wesentliche übertragbare Modulkonzepte



- Das **Modulkonzept** leistet vorrangig zweierlei:
  - Es führt einen **Namensraum** (engl.: name space) ein, um Programmeinheiten zu gliedern und zu benennen:
    - Block
    - Prozedur,
    - Klasse (Interface),
    - Paket (in Java),
    - Programm.
  - Es definiert einen **Zugriffsschutz** (engl.: access control) für die enthaltenen Elemente:
    - Elemente innerhalb eines Moduls sind zunächst verborgen.
    - Sie können durch gezielten Export für die Benutzung zugreifbar gemacht werden.

Klassen sind auch Module

- Die **Klassen** von objektorientierten Programmiersprachen können auch **als Module angesehen** werden:
  - Sie sind logische Einheiten des statischen Quelltextes, um große Programme handhabbarer zu machen.
  - Sie lassen sich meist einzeln (und damit getrennt) übersetzen.
  - Sie definieren eine Schnittstelle mit den Dienstleistungen, die sie ihren Klienten anbieten.
  - Sie definieren einen Namensraum für ihre Dienstleistungen.
  - Sie können Teile ihrer Implementation verbergen.
- Wir wissen aber auch, dass **Klassen** deutlich **mehr** können **als Module**: Sie definieren Typen mit Exemplaren, die polymorph verwendbar sind, und können in Vererbungsbeziehungen zueinander stehen.
- Wenn wir die Klassen in Java betrachten, erkennen wir ein weiteres wichtiges Element: **Klassen** können ineinander **geschachtelt** werden.

SE2 – OOPM – Teil 1 35

Geschachtelte Klassen in Java

- In Java können Klassen in Klassen (beliebig oft) geschachtelt werden. Eine Klasse, die in einer anderen Klasse enthalten ist, heißt **geschachtelte Klasse** (engl.: nested class).
- *„A nested class is any class whose declaration occurs within the body of another class or interface.“ (Gosling 2005)*
- Diese sind abzugrenzen von den uns bisher bekannten **Top-Level Klassen** (engl.: top level class), die in Übersetzungseinheiten „ganz außen“ (nicht geschachtelt) aufgeführt sind.
- Geschachtelte Klassen ermöglichen:
  - Bessere Strukturierung (Verstecken von Details)
  - Bequemeres Programmieren (Beispiel: Listener für GUIs)

SE2 – OOPM – Teil 1 36

Geschachtelte und innere Klassen in Java

Java unterscheidet vier Arten von geschachtelten Klassen:

1. **Static Member Classes** } ¬Innere Klassen
2. **Member Classes**
3. **Local Classes**
4. **Anonymous Classes**

} Innere Klassen (engl.: inner classes)

- this-Referenz
- keine static-Felder erlaubt

Die drei Arten von **inneren Klassen** (engl.: inner classes) definieren eine spezielle Untermenge aller geschachtelten Klassen; ihre Exemplare halten implizit immer eine Referenz auf ein Exemplar der umgebenden Klasse; sie definieren also spezielle Eigenschaften auch für die Laufzeit. Wir werden innere Klassen hier nicht näher betrachten.

Die **statischen geschachtelten Klassen** (static member classes) hingegen kommen dem Konzept von geschachtelten Modulen am nächsten.

Statische geschachtelte Klassen in Java

- Deklariert mit dem Schlüsselwort `static` im Rumpf einer umgebenden Klasse:

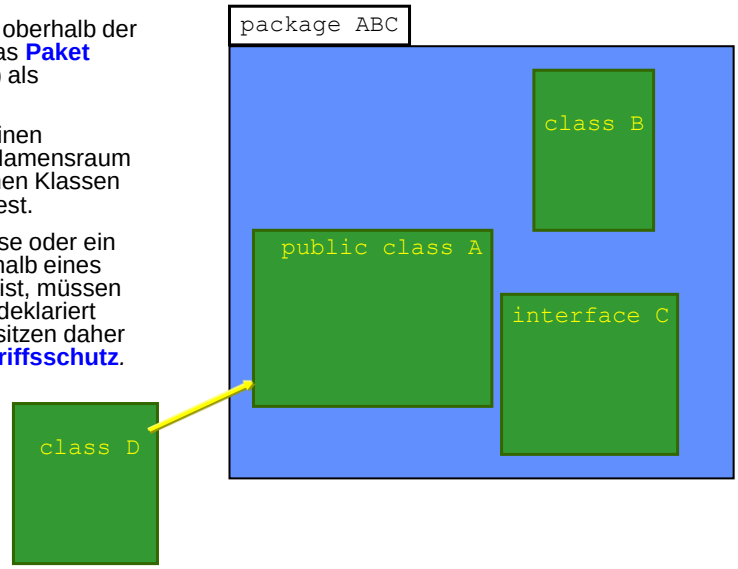
```
class A
{
    public void operation()
    {   ...
    }

    private static class MyStaticMemberClass()
    {    ... // beliebige Felder und Methoden
    }
}
```

- Reine textuelle Schachtelung, ohne Auswirkungen zur Laufzeit.
- Ermöglicht u.a. das Verstecken von Hilfsklassen vor Klienten der umgebenden Klasse (im Beispiel durch den Modifikator `private`).
- Die Klassen sind sehr eng miteinander vertraut: Beide haben **Zugriff auf die privaten Eigenschaften** der Exemplare der jeweils anderen Klasse!

Auch sehr ähnlich zu Modulen: Pakete in Java

- In Java existiert oberhalb der Klassen noch das **Paket** (engl.: package) als **Namensraum**.
- Ein Paket legt einen gemeinsamen Namensraum für die enthaltenen Klassen und Interfaces fest.
- Damit eine Klasse oder ein Interface außerhalb eines Pakets sichtbar ist, müssen sie als `public` deklariert sein. Pakete besitzen daher auch einen **Zugriffsschutz**.



Pakete als Namensräume in Java

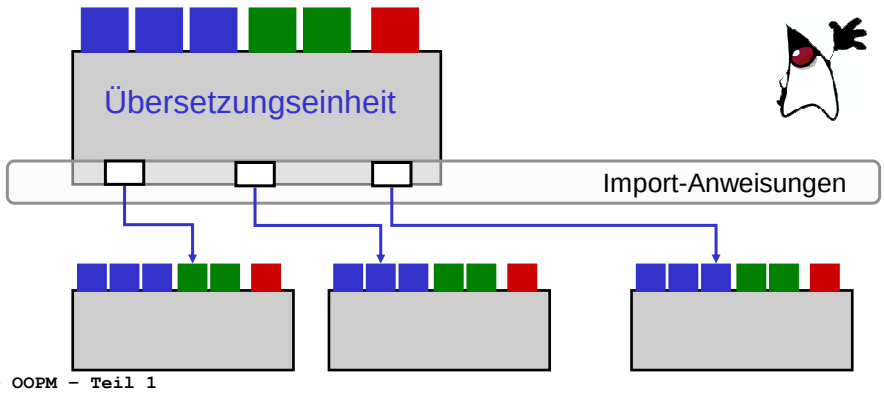
- Pakete als eigene Namensräume ermöglichen, allgemein übliche Namen (wie `List` oder `File`) in einem begrenzten Kontext ohne Konflikte zu verwenden.
- Pakete werden in Übersetzungseinheiten deklariert. Beispiel:
```
package graphics;
```
- Alle Klassen und Interfaces eines Pakets haben implizit den Paketnamen als Prefix. Der **voll qualifizierte Name** setzt sich aus dem Paketnamen und dem Typnamen zusammen. Beispiel:
```
graphics.Punkt
```
- Jede Klasse (und jedes Interface) kann **nur zu einem** Package gehören. Damit können Operationen einer exportierten Klasse aus einem anderen Paket eindeutig über den voll qualifizierten Bezeichner angesprochen werden. Beispiel:
```
graphics.Punkt.ursprung();
```

Package-Deklarationen stehen am Anfang einer Übersetzungseinheit.

Dateien ohne explizite Deklaration gehören zu einem Default-Package

Import von Paketelementen

- Java verfügt über einen **Import-Mechanismus**, der wenig mit der klassischen Import-Schnittstelle zu tun hat:
  - Alle von anderen Paketen exportierten Klassen und Interfaces können durch eine import-Anweisung in einer Übersetzungseinheit einfacher benutzbar gemacht werden. Die importierten Klassen und Interfaces müssen dann nicht mehr voll qualifiziert werden; damit ist der Import nur eine **syntaktische Vereinfachung**.



SE2 – OOPM – Teil 1 41

Bereits bekannt: Import von Paketelementen

- Die **import-Anweisung** hat zwei Formen:

```
import somepackage.SomeClass;
```

  Die Klasse `SomeClass` von `somepackage` ist als Typ verfügbar.

```
import somepackage.*;
```

  Alle von `somepackage` exportierten Klassen und Interfaces sind als Typ verfügbar.

- Die importierten Typen sind anschließend in verkürzter Form benutzbar. Z.B.:

```
import graphics.Punkt;
...
Punkt pkt = new Punkt();
```

- Ohne Package-Import würde das Beispiel so aussehen:

```
graphics.Punkt pkt = new graphics.Punkt();
```

Softwaretechnisch ist es sinnvoll, nur die tatsächlich benötigten Typen zu importieren; dies verdeutlicht Abhängigkeiten.

SE2 – OOPM – Teil 1 42

Namensvergabe für Pakete in Java

- Pakete können ineinander geschachtelt werden. Auf diese Weise entstehen Paketnamen, die aus mehreren Teilen bestehen können, die mit einem Punkt voneinander getrennt werden.
- Da Java-Anwendungen leicht über das Internet ausgetauscht werden können, müssen Namenskonflikte vermieden werden. Daher hat sich eine **Namenskonvention** für Pakete durchgesetzt:
  - Die Konvention der Internet Domänennamen wird als Basis der Paketnamen genommen, wobei die Reihenfolge umgedreht wird. Nach Konvention werden die Bestandteile von Paketnamen aus Kleinbuchstaben gebildet.
  - Die „top-level“ Bezeichner `java` und `sun` sind per Konvention reserviert.
  - Beispiel:
    `de.uni-hamburg.informatik.swt.graphics`

SE2 – OOPM – Teil 1 43

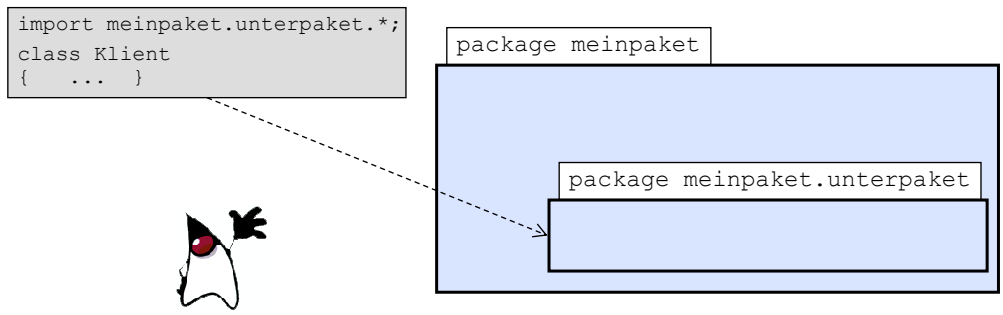
Beschränkter Zugriff: Modifikatoren in Java

- Objektorientierte Programmiersprachen bieten die Möglichkeit, die **Sichtbarkeit**, genauer den Zugriff, auf Klassen, Interfaces und ihre Bestandteile zu **steuern**.
- Dazu dienen unterschiedliche Zugriffsrechte. Diese werden meist durch reservierte Schlüsselwörter (in Java „**Modifikatoren**“ z.B. `private`, `public`) notiert.
- Dabei müssen wir in Java unterscheiden:
  - Sichtbarkeit einer Klasse oder eines Interfaces selbst, **außerhalb des Packages**.
  - Zugriff auf **Bestandteile einer Klasse**.

SE2 – OOPM – Teil 1 44

Zugriffsrechte in Java: nicht einschränkbar zwischen Paketen

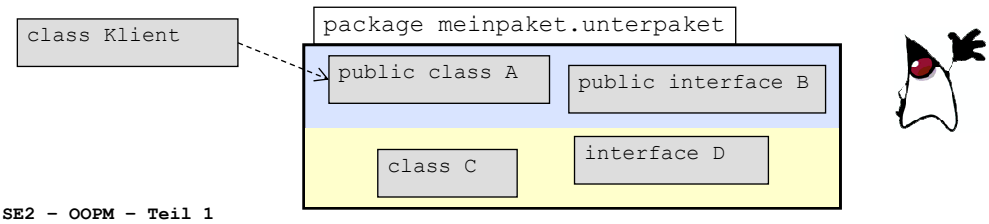
- Ein **Paket** selbst ist immer zugreifbar; es gibt keine Schutz-Mechanismen zwischen Paketen.
- Auch die **Schachtelung** von Paketen bewirkt in Java **keinerlei Einschränkung der Zugreifbarkeit**; die Schachtelung dient ausschließlich der Strukturierung von Namen.



SE2 – OOPM – Teil 1 45

Zugriffsrechte in Java: Elemente von Paketen

- Die **Klassen** (u. Interfaces) innerhalb eines Paketes sind für andere Klassen (Interfaces) je nach Zugriffsrecht zugreifbar:
  - ***<Standardzugriff>*** : Eine Klasse (Interface) ohne Zugriffsmodifikator besitzt Standardzugriff, d.h. ist ausschließlich zugreifbar für Klassen (Interfaces) im selben Paket.
  - **`public`** : Eine als **`public`** deklarierte Klasse (Interface) ist zusätzlich zugreifbar für Klassen (Interfaces) in (allen!) anderen Paketen.
- Pakete haben somit eine implizite **Export-Schnittstelle**: Alle als **`public`** deklarierten Klassen und Interfaces gehören zur Schnittstelle eines Paketes.



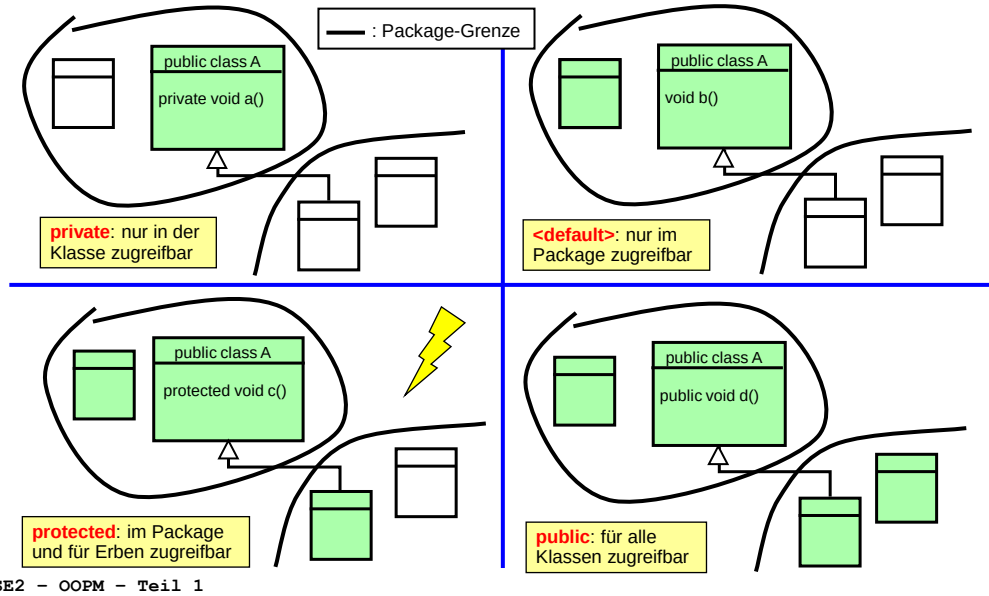
SE2 – OOPM – Teil 1 46

Zugriffsrechte in Java: Elemente von Klassen (vollständig)

- **`private`**: Ein als **`private`** deklariertes Element (Feld, Methode oder Klasse) einer Klasse kann nicht von anderen (Top-Level) Klassen zugegriffen werden. Dies gilt unabhängig von der Paketzugehörigkeit oder der Vererbungsbeziehung, auch wenn die Klasse selbst **`public`** ist.
- ***<Standardzugriff>*** : Ein Element ohne deklarierten Zugriffsschutz ist zugreifbar für alle Klassen im selben Paket.
- **`protected`**: Ein als **`protected`** deklariertes Element einer **`public`** Klasse ist zugreifbar für Klassen im selben Paket und für Subklassen in anderen Paketen.
- **`public`**: Ein als **`public`** deklariertes Element einer **`public`** Klasse ist zugreifbar für Klassen im selben Paket und in anderen Paketen.

SE2 – OOPM – Teil 1 47

Übersicht: Zugriffsmodifikatoren für Elemente von Klassen



SE2 – OOPM – Teil 1 48

Jenseits von Packages: SuperPackages?

com.sun.myModule

Computational Theology
*Gilad Bracha's Sun Weblog**

```
super package com.sun.myModule {
export com.sun.myModule.myStuff.*;
export com.sun.myModule.yourStuff.Interface;
com.sun.myModule.myStuff;
com.sun.myModule.yourStuff;
com.sun.SomeOtherModule.theirStuff;
org.someOpenSource.someCoolStuff;
}
```

JSR 294: super packages auf dem Weg in die nächste Version von Java?

SomeOtherModule.theirStuff

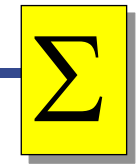
someCoolStuff

myModule.yourStuff

myModule.myStuff

*http://blogs.sun.com/gbracha/entry/developing_modules_for_development

Zusammenfassung: Modul und Klasse



- **Klassen** sind wie **Module** Einheiten der (statischen) Softwarearchitektur. Große Systeme werden aus solchen Einheiten zusammengesetzt. Sie sind üblicherweise auch **Übersetzungseinheiten**.
- Klassen und Pakete bilden wie Module **Namensräume**, deren Elemente vor äußerem Zugriff geschützt werden können.
- Im Gegensatz zu (klassischen) Modulen definieren **Klassen** gleichzeitig einen **Typ**. So lassen sich zur Laufzeit Exemplare von Klassen erzeugen.
- **Module** besitzen verbindliche **Export- und Import-Schnittstellen**. Klassen und Pakete in Java deklarieren ihre Export- und Import-Schnittstellen implizit im Quelltext; sie müssen aus dem Programmtext „extrahiert“ werden.

Modularisierung in Java	Exportschnittstelle/ Schutz	Import-schnittstelle	Benennung
Klasse (Typ)	explizit (Modifikatoren), rekursiv (auch für Typen)	nur implizit	Punkt-notation
Paket	explizit (Modifikatoren), nur für Typen	nur implizit	Punkt-notation

