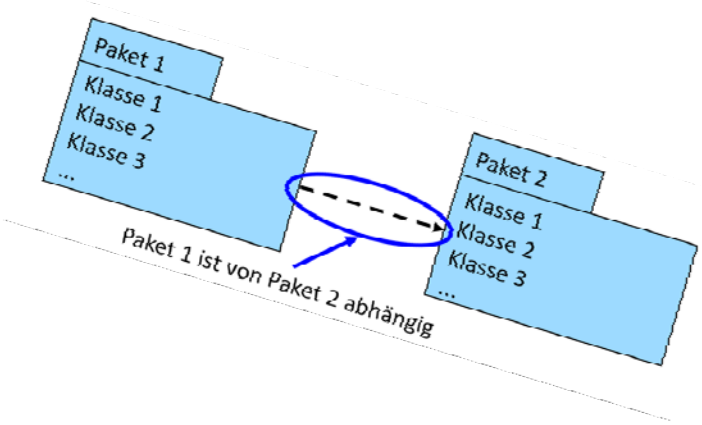
Vorlesung Programmieren
Thema 7: Objektorientierung (II)

Olaf Herden
Fakultät Technik
Studiengang Informatik

Modifier	Symbol UML
public	+
private	-
protected	#
<Default>	<nichts>

CLASSPATH



Stand: 12/2023

Gliederung

- Pakete
- Zugriffsrechte für Klassen
- Attribute (Vertiefung)
- Methoden (Vertiefung)
- Kapselung, ADT

Organisation von Klassen (I)

- Bisher: Entwicklung und Abspeicherung von Klassen
- Bei größeren Projekten: Strukturierung/Ordnung/Organisation der Klassen
- In Java Strukturierungsmöglichkeit über Pakete (Packages):
  - Ermöglichen das Zusammenfassen (inhaltlich) verwandter Klassen
  - Stellen für die Klassen des Pakets einen Namensraum zur Verfügung
- Paket benannte Sammlung von Klassen
- Java liefert eine Reihe von Paketen mit, z.B.:
  - `java.lang`: Enthält die fundamentalsten Klassen
  - `java.util`: Enthält diverse Hilfsklassen
  - `java.io`: Enthält Klassen für Ein-/Ausgabefunktionalität
  - `...`

Organisation von Klassen (II)

- Pakete können Unterpakete (Subpackages) enthalten
- Beispiel: Paket `java.lang` Unterpaket `ref`, Ansprechen mit `java.lang.ref`
- Jede Klasse hat:
  - einfachen Namen (aus Klassendefinition)
  - vollqualifizierten Namen (enthält auch Namen des Paketes)
- Beispiel: Klasse `String` hat
  - einfachen Namen `String`
  - vollqualifizierten Namen `java.lang.String`

Pakete definieren

- Zuordnung Klasse zu Paket: erste Anweisung im Quellcode `package <Paketname>;`
- Beispiel:

```
package myPackage;
class myClass{
   ...
}
```

- Zuordnung aller Klassen dieser Datei zum Paket
- Weglassen `package`-Anweisung $\Rightarrow$ Klassen gehören zu unbenanntem Standardpaket (Default Package)

Verwendung von Klassen

- Klassen aus gleichem Paket: Ansprechen über Namen
- Klassen aus anderen Paketen: Ansprechen über vollqualifizierten Namen
- Ausnahme: Paket `java.lang` enthält essentielle Klassen, können über einfachen Namen angesprochen werden (`String` anstelle von `java.lang.String`)

Import (I)

- Import: Bekanntmachen von Klassen aus anderen Paketen
- Vier Möglichkeiten:
  - Einzelne Klasse:
    - Import: `import <Paketname>.<Klassenname>;`
    - Verwendung: `<Klassenname> k = new <Klassenname>();`
  - Alle Klassen eines Pakets:
    - Import: `import <Paketname>.*;`
    - Verwendung: siehe oben
  - Ganzes Paket:
    - Import: `import <Paketname>.<Unterpaketname>;`
    - Verwendung:
```
<Unterpaketname>.<Klassenname> k =
                    new <Unterpaketname>.<Klassenname>();
```
  - Verwendung ohne `import`-Anweisung:
```
<Paketname>.<Unterpaketname>.<Klassenname> k
  = new <Paketname>.<Unterpaketname>.<Klassenname>();
```

Import (II)

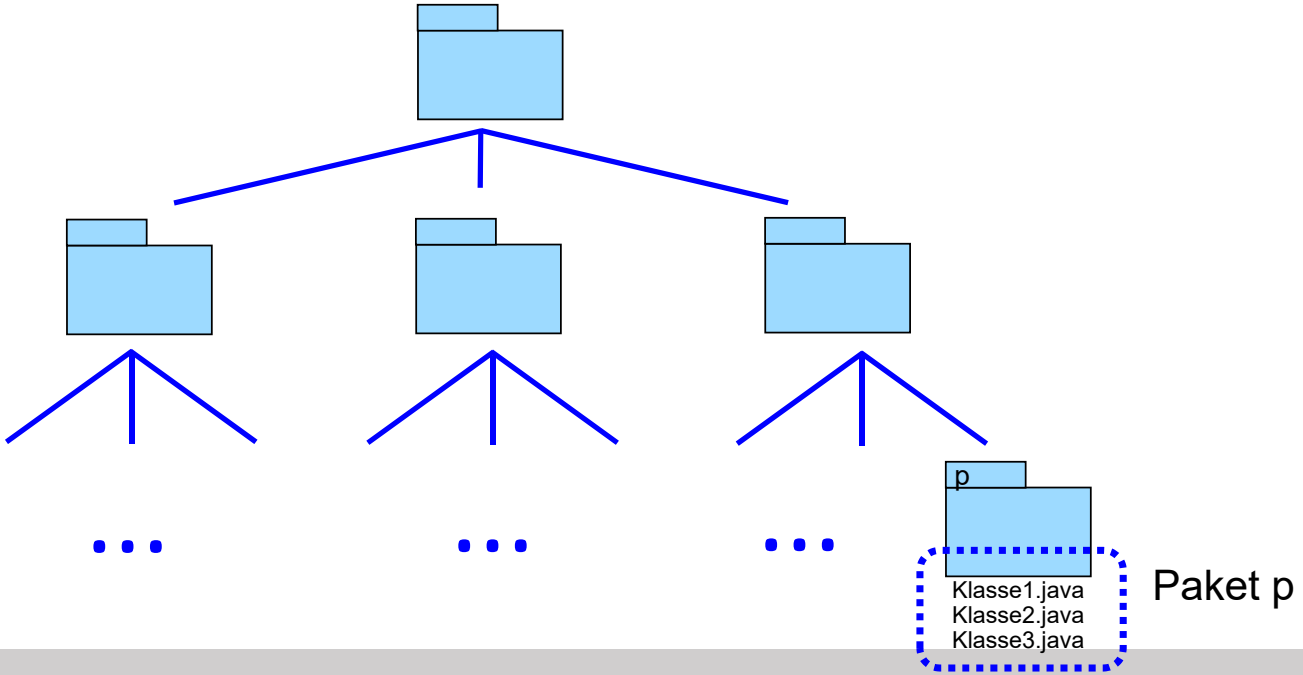
- Anmerkungen zum Importieren:
  - `import`-Anweisungen müssen direkt hinter der `package`-Anweisung stehen
  - Pro Datei beliebig viele `import`-Anweisungen
  - Anweisung `import <Paketname>.*;` importiert keine Unterpakete
  - Muss explizit durch `import <Paketname>.<Unterpaketname>.*;` geschehen
  - `import`-Anweisungen können aus zwei Paketen gleich benannte Methoden importieren
  - Namenskonflikt wird durch Nennung des vollqualifizierten Namens aufgelöst

Organisation von Klassen

- Global eindeutige Paketnamen:
  - Jede*r Entwickler*in benennt „einfach so“ ⇒ Namenskonflikte möglich
  - Daher: Notwendigkeit globaler Benennungskonventionen
  - Oracle kontrolliert alle Pakete, die mit `java`, `javax`, `sun` und `oracle` beginnen
  - Für alle gilt: Vor Paketnamen umgedrehte eigene Internetdomäne setzen
  - Bsp.: Lautet die Internetdomäne „dhbw-stuttgart.de“, so sind die Pakete alle mit dem Präfix „de.dhbwstuttgart“ zu versehen
  - Innerhalb dieses Namensraums Freiheit in Organisation der Pakete bzw. Unterpakete

Compilieren von Paketen

- Alle Klassen eines Paketes in einem Verzeichnis
- Pro Verzeichnis nur ein Paket
- Verzeichnisname muss Paketnamen entsprechen
- Kompilieren vom Verzeichnis oberhalb mit `javac <Paketname>\*.java`
- Zur Verwendung des Paketes Umgebungsvariable `CLASSPATH` auf das Verzeichnis oberhalb setzen



Umgebungsvariable CLASSPATH (I)

- Sammlung von Verzeichnissen, in denen Laufzeitsystem nach Klassen, Paketen und anderen Ressourcen sucht
- Umgebungsvariable des Betriebssystems
- Default: Aktuelles Verzeichnis
- Default ändern:
  - Anwendung Option `–classpath` oder `–cp` in Kommandozeile
  - Setzen CLASSPATH-Umgebungsvariable

- Option beim Aufruf (gilt für diesen Aufruf):
  - `java -cp .;<path_1>;<path_2>;... myProgram`
    oder
  - `java -classpath .;<path_1>;<path_2>;... myProgram`
  - Linux: „:“ statt „;“ verwenden

Umgebungsvariable CLASSPATH (II)

- Setzen CLASSPATH-Umgebungsvariable:
  - Konsole (Session):
    - Linux mit csh: `setenv CLASSPATH "<Pfad1>:<Pfad2>:..."`
    - Linux mit bash: `setenv CLASSPATH = <Pfad1>:<Pfad2>:...`
    - Windows: `set CLASSPATH <Pfad1>;<Pfad2>;...`
  - System (permanent):
    - Windows: Systemeigenschaften → Register „Erweitert“→ Button „Umgebungsvariablen“ (Permanent)
    - Linux: Eintragen im Profile

- Anmerkungen:
  - „.“ muss für anonyme Pakete in CLASSPATH aufgenommen werden
  - Verzeichnis der JDK-Klassenbibliothek muss nicht in CLASSPATH aufgenommen werden

Umgebungsvariable CLASSPATH (III): Wildcards

- Einbinden Klassen: Angabe kompletter Pfad des Verzeichnis der Klassen
- Einbinden jar-Dateien:
  - Angabe kompletter Pfad des Verzeichnis der jar-Datei
  - Verwendung * für alle jar-Dateien in Verzeichnis
  - Achtung:
    - Keine automatische Einbindung von jar-Dateien in Unterverzeichnissen
    - Explizite Angabe notwendig
- Beispiele:

```
//Einbinden aller Klassen im Verzeichnis 'c:/temp/lib‘
java -cp c:/temp/lib myProg
//Einbinden aller jars im Verzeichnis 'c:/temp/lib‘
java -cp c:/temp/lib/* myProg
//Einbinden aller Klassen und jars im Verzeichnis 'c:/temp/lib‘
java -cp c:/temp/lib;c:/temp/lib/* myProg
```

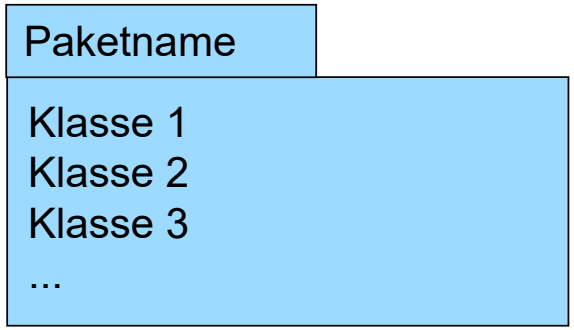
Umgebungsvariable CLASSPATH (IV)

- Zurücksetzen CLASSPATH-Umgebungsvariable:
  - `set CLASSPATH=`

  oder
  - `set CP=`
  - Bei Linux : `setenv`

UML (I)

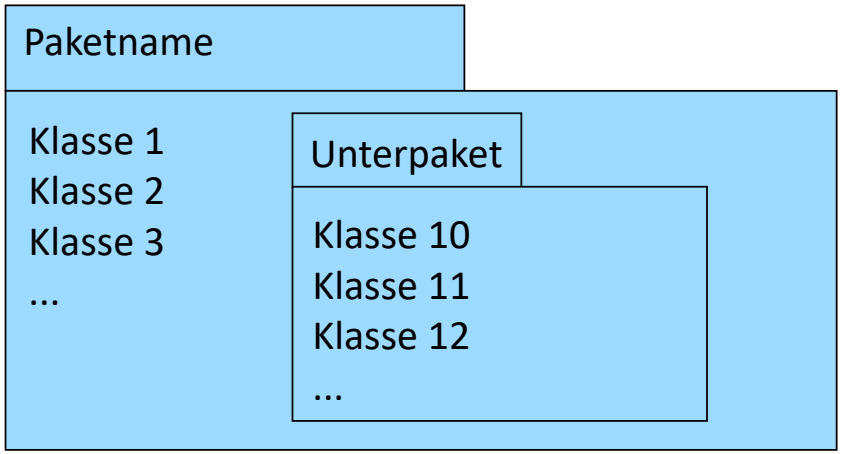
- Notation Pakete in UML:



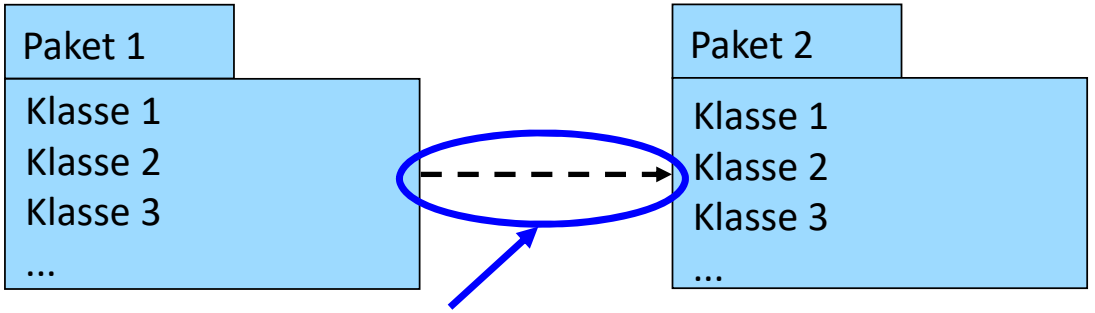
- Anmerkung:
  - Häufig keine Auflistung in Modellierungswerkzeugen
  - Stattdessen: Anzeige der Klassen mit „Click and Zoom“

UML (II)

- Schachtelung von Paketen:



- Abhängigkeiten zwischen Paketen:



Paket 1 ist von Paket 2 abhängig

Übersicht

- Pakete
- Zugriffsrechte für Klassen
- Attribute (Vertiefung)
- Methoden (Vertiefung)
- Kapselung, ADT

Zugriff

- Bisher: Keine Berücksichtigung über Zugriff auf Klassen von außen
- Zugriffsschutz auf verschiedenen Ebenen:
  - Pakete
  - Klassen
  - Attribute/Methoden

- Zugriff auf Pakete:
  - Paket für Code im selben Paket erreichbar
  - Zugriff auf andere Pakete:
    - Keine Regelung in Java selber
    - Abhängig von Ort des Paketes, Classpath und Zugriffsrechte auf Verzeichnisse

Zugriffsrechte Klassen

- Klassen können bei ihrer Definition mit einem Zugriffsmodifier versehen werden
- Syntax: `[Modifier] class <Klassenname>`
- Default (Weglassen des Modifier): Klasse innerhalb Paket zugreifbar
- Beispiel: `class Landfahrzeug{...}`
- Zugreifbar auch außerhalb Paket: Modifier `public`
- Beispiel: `public class Landfahrzeug{...}`
- Weitere Klassen-Modifier:
  - `abstract`: Klasse ist abstrakt (darf keine Instanzen haben)
  - `final`: Klasse ist final (darf keine Superklasse sein)
- Kombination Modifier: `abstract final` verboten
- Alle möglichen Kombinationen: kein Modifier, `final`, `abstract`, `public`, `public final`, `public abstract`

Übersicht

- Pakete
- Zugriffsrechte für Klassen
- Attribute (Vertiefung)
- Methoden (Vertiefung)
- Kapselung, ADT

Klassenvariablen (I)

- Wiederholung:
  - Anlegen Attribute einer Klasse durch Angabe von Datentyp und Namen
  - Anlegen Konstante: Voranstellen Schlüsselwort `final`

- Bisher angelegte Variablen sog. Instanzvariablen
- Beschreiben Eigenschaft einzelnen Objektes, pro Objekt eine Variable

- Klassenvariablen: Existieren nur einmal pro Klasse
- Syntax: `static <Datentyp> <Attributname>`
- Ansprechen von Objekten der Klasse über Punktnotation

Klassenvariablen (II)

- Beispiel: Instanzzähler einer Klasse

```
( 1) public class Fahrzeug{
( 2)   static int anzObjekte = 0;
( 3)   public Fahrzeug(){
( 4)      ++anzObjekte;
( 5)      <Weitere Anweisungen>
( 6)   }
( 7)   public void finalize(){
( 8)      --anzObjekte;
( 9)      <Weitere Anweisungen>
(10)   }
(11)   ...
(12) }
```

Klassen als Datentypen

- Klassen können Datentyp von Attributen sein
- Beispiel:

```
( 1) class Rad{...}
( 2)
( 3) class Motor{...}
( 4)
( 5) class Fahrzeug{
( 6)   String name;
( 7)   Motor  m;
( 8)   Rad[]  r;
( 9)    ...
(10) }
```

Attributzugriffsrechte (I)

- Zugriff auf Attribute:
  - Innerhalb Klasse: Alle Methoden haben Zugriff
  - Beschränkungen für Methoden anderer Klassen möglich
- Syntax: `[Modifier] <Datentyp> <attribute_name>`
- Standardwert (Weglassen des Modifier):
  - Attribut paketsichtbar, d.h. Zugriff auf Attribut von allen Klassen des Paketes
  - Beispiel: `int i;`
- Zugriffsmodifier `public`:
  - Zugriff auf Attribut aus allen anderen Klassen und Paketen
  - Beispiel: `public int i;`

Attributzugriffsrechte (II)

- Zugriffsmodifier `protected`:
  - Zugriff auf Attribut im eigenen Paket und in allen Subklassen auch in anderen Paketen (Unterschied zu paketsichtbar)
  - Bsp.: `protected int i;`
- Modifier `private`:
  - Zugriff auf Attribut nur innerhalb der eigenen Klasse
  - Bsp.: `private int i;`

Attributzugriffsrechte (III): UML

- Notation:

Modifier	Symbol UML
public	+
private	-
protected	#
<Default>	<nichts>

- Beispiel:

Fahrzeug
+ Bezeichnung : Textfeld(50) - PS_Zahl : Zahl - Räder : Zahl ...
...

Übersicht

- Pakete
- Zugriffsrechte für Klassen
- Attribute (Vertiefung)
- Methoden (Vertiefung)
- Kapselung, ADT

Methodenzugriffsrechte (I)

- Zugriff auf Methoden:
  - Innerhalb Klasse: Alle Methoden haben Zugriff
  - Beschränkungen für Methoden anderer Klassen möglich
- Syntax: `[Modifier] <Rückgabewert> <Name> (<Parameterliste>)`

- Standardwert (Weglassen des Modifier):
  - Methode paketsichtbar, d.h. Zugriff auf Methode von allen Klassen des Paketes
  - Beispiel: `int square(int i) { ... }`
- Zugriffsmodifier `public`:
  - Zugriff auf Methode aus allen anderen Klassen und Paketen
  - Beispiel: `public int square(int i) { ... }`

Methodenzugriffsrechte (II)

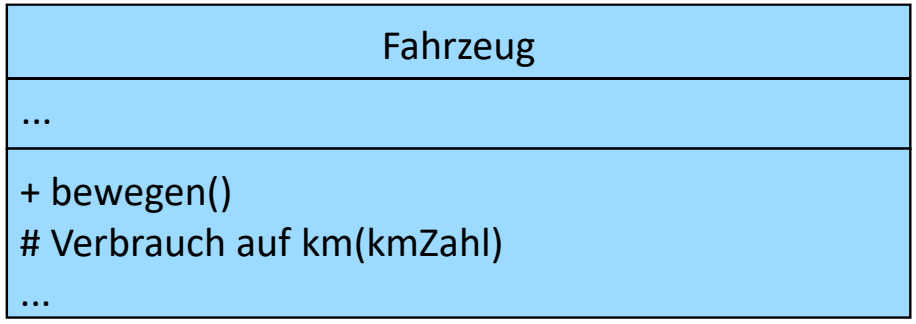
- Modifier `protected`:
  - Zugriff auf Methode im eigenen Paket und in allen Subklassen auch in anderen Paketen (Unterschied zu paketsichtbar)
  - Bsp.: `protected int square(int i) { ... };`
- Modifier `private`:
  - Zugriff auf Methode nur innerhalb der eigenen Klasse
  - Bsp.: `private int square(int i) { ... };`
  - Anm.: Kein Zugriff für Subklassen auf private Methoden ihrer Superklasse

Methodenzugriffsrechte (III): UML

- Notation:

Modifier	Symbol UML
public	+
private	-
protected	#
<Default>	<nichts>

- Beispiel:



Klassen als Rückgabe- und Parametertyp

- Klassen können Parameter- und Rückgabetypen von Methoden sein
- Beispiel:

```
( 1) public class Fahrzeug{
( 2)
( 3)   static Fahrzeug kopieren(Fahrzeug f){
( 4)     return f;
( 5)   }
( 6)
( 7)   public static void main (String[] args){
( 8)      Fahrzeug f = new Fahrzeug();
( 9)      Fahrzeug g = new Fahrzeug();
(10)      g = kopieren(f);
(11)   }
(12) }
```

Klassen- und Instanzmethoden (I)

- Instanzmethoden: Für jedes Objekt einer Klasse definiert
- Klassenmethoden:
  - Nur einmal für Klasse definiert
  - Kennzeichnung durch Schlüsselwort `static`
  - Heißen auch statische Methoden
- Beispiele:
  - `main`-Methode
  -
```
public static double square(double i){
   return i*i;
}
```

Klassen- und Instanzmethoden (II)

- Aufruf statischer Methode außerhalb ihrer Klasse durch Angabe von Klassen- und Methodennamen, z.B.:

```
double d = GanzeZahl.square(5.0);
```

- Klassenmethoden:
  - Können keine Instanzmethoden aufrufen
  - Dürfen nur überschrieben werden, wenn Methode der Subklasse auch wieder statisch

Statischer Import

- Bisher: Ansprechen statischer Methoden (und Attribute) durch Voranstellen des Klassennamens, z.B.:

```
float radius = 4;
System.out.println("Umfang = " + 2*Math.PI*radius);
System.out.println("Sinus("+radius+") = "+ Math.sin(radius));
```

- Statischer Import `import static java.lang.Math.*;`:
  Importiert alle statischen Attribute und Methoden der Klasse
- Damit:

```
System.out.println("Umfang = " + 2*PI*radius);
System.out.println("Sinus("+radius+") = " + sin(radius));
```

- Anmerkung:
  - `import java.util.*` und `import static java.lang.Math.*` sehen ähnlich aus
  - Aber: Einmal werden alle Klassen aus Paket importiert, einmal alle statischen Attribute und Methoden einer Klasse

Finale Methoden

- Dürfen in keiner Unterklasse überschrieben werden
- Schlüsselwort `final`
- Beispiel:

```
final int berechne(int a, float b);
```

- Zweck: Implementierung der Methode ist Standard, der nicht verändert werden soll

Abstrakte Methoden (I)

- Werden in Klasse deklariert, aber nicht implementiert
- Syntax: Schlüsselwort `abstract`, kein Methodenrumpf
- Beispiel:

```
abstract void draw();
```

- Zweck: Methodendeklaration passend, Realisierung in Subklassen, meistens in unterschiedlicher Art und Weise
- Beispiele:
  - Jedes Fahrzeug hat Methode `bewege()`, Implementierung individuell pro Subklasse
  - Jedes graphische Objekt kann gezeichnet werden, Implementierung individuell pro Subklasse

Abstrakte Methoden (II): Anmerkungen

- Deklaration nur in abstrakten Klassen, sonst Fehlermeldung:
  `<Klasse> should be declared abstract; it does not define <Methode> in <Klasse>`
- Keine Deklaration statischer Methoden als `abstract`, sonst Fehlermeldung:
  `illegal combination of modifiers: abstract and static`
- Keine Deklaration finaler Methoden als `abstract`, sonst Fehlermeldung:
  `illegal combination of modifiers: abstract and final`
- Deklaration Konstruktoren als `abstract` erlaubt (d.h. beim Compilieren keine Fehlermeldung), jedoch nicht sinnvoll
- Subklasse muss
  - Entweder alle abstrakten Methoden überschreiben
  - oder selber als `abstract` deklariert werden

Weitere Methodenmodifier

- Native Methoden:
  - Nicht in Java, sondern meistens in C/C++ geschrieben
  - Beispiel: `native int berechne(int a, float b);`
  - Zweck: Verwendung vorhandener (meist systemnaher) Bibliotheken
  - Problem: Plattformunabhängigkeit geht verloren

- Methoden synchronisieren:
  - Zugriff zweier Methoden auf gleiche Daten ⇒ Zugriffsproblem, Datenverletzung
  - Vermeidung durch Synchronisation
  - Beispiel: `synchronized int berechne(int a, float b);`
  - In Objekt dürfen nie zwei als `synchronized` deklarierte Methoden gleichzeitig ablaufen
  - Anm.: Behandlung synchronisierter Methoden in VL-Abschnitt „Multithreading“

Namenskonventionen

- Zwingend: Methodennamen
  - dürfen unbeschränkte Länge haben (bis auf technische Einschränkungen durch System)
  - müssen (wie alle anderen Java-Bezeichner) mit Buchstaben oder „_“ oder „$“ beginnen, alle weiteren Zeichen dürfen Buchstaben, Zahlen und ein paar Sonderzeichen sein
  - werden (wie bei allen Java-Bezeichnern) nach Groß-und Kleinschreibung unterschieden
  - dürfen (wie andere Bezeichner) nicht einem Java-Schlüsselwort entsprechen
- Üblich: Methodennamen
  - sollten möglichst aussagekräftig sein („Sprechende Bezeichner“)
  - sollten mit einem Kleinbuchstaben beginnen, dann sollten weitere Kleinbuchstaben folgen
  - aus mehreren Wörtern sollten zusammengeschrieben werden, neues Wort mit Großbuchstaben beginnen

Konstruktoren und Modifier

- Konstruktoren:
  - dürfen nicht als `native`, `static`, `synchronized` oder `final` deklariert werden
  - dürfen somit paketsichtbar (keine Modifier-Angabe) oder als `public` deklariert werden
  - werden meistens als `public` deklariert

- Anmerkung: Überschreiben von Konstruktoren technisch nicht möglich, da immer gleicher Name wie Klasse

Zusammenfassung Sichtbarkeit

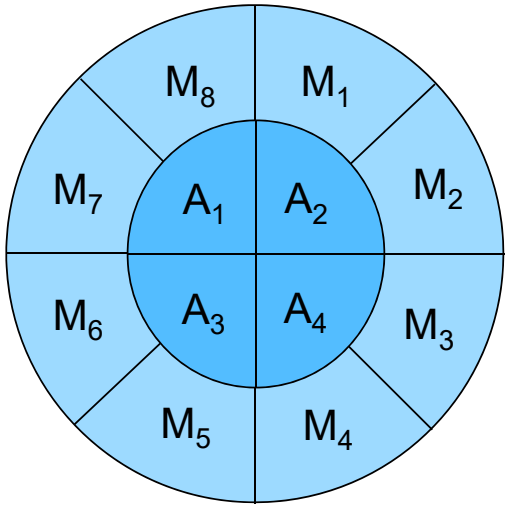
Modifier	Sichtbarkeit
`public`	• In Methoden der Klasse selbst • In Methoden abgeleiteter Klassen • In Methoden des Paketes • In Methoden anderer Pakete, keine Unterklasse
`protected`	• In Methoden der Klasse selbst • In Methoden abgeleiteter Klassen, auch in anderen Paketen • In allen Klassen desselben Pakets
`private`	• Nur in Methoden der Klasse selbst
Default (paketsichtbar)	• In allen Klassen desselben Paketes

Übersicht

- Pakete
- Zugriffsrechte für Klassen
- Attribute (Vertiefung)
- Methoden (Vertiefung)
- Kapselung, ADT

Kapselung

- Wichtiges Konzept der Objektorientierung
- Ansprechen Klassen von außen nur über Schnittstelle, Abstraktion von interner Realisierung
- Realisierung:
  - Deklaration aller Attribute als `private`
  - Methoden zum Verändern und Lesen von außen
  - Namen typischerweise `get<Attributname>`
    bzw. `set<Attributname>` („Getter und Setter“)



Beispiel

- Kapselung aller Attribute:

```
( 1) public class Fahrzeug{
( 2)   private String name;
( 3)   private int raeder;
( 4)   private int psZahl;
( 5)   ...
( 6)   public void setName(String s){name = s;}
( 7)   public String getName(){return name;}
( 8)   public void setRaeder(int r){raeder = r;}
( 9)   public int getRaeder(){return raeder;}
(10)   public void setPsZahl(int z){psZahl = z;}
(11)   public int getPsZahl(){return psZahl;}
(12)   ...
(13) }
```

Abstrakter Datentyp

- Kapselung Umsetzung bzw. Weiterführung des ADT-Konzepts (Abstrakter Datentyp) (Siehe Informatik-VL)
- Abstrakte Datentypen auch in nicht OO-Sprachen
- Abstrakter Datentyp:
  - Name
  - Funktionen/Methoden auf dem Datentypen
  - Evtl. weitere Axiome
- Beispiel: ADT Stapel
  - Speicherstruktur, die das LIFO-Prinzip realisiert (Last In First Out)
  - Funktionen:
    - push(o): Legt das Objekt o auf dem Stapel ab
    - pop(): Liest und entnimmt das oberste Element vom Stapel
    - peek(): Liest oberstes Element vom Stapel ohne es zu entfernen
    - isEmpty(): Prüft, ob der Stapel leer ist

Zusammenfassung (I)

- Pakete:
  - Organisation von Klassen
  - Schlüsselworte `package` und `import`
  - Umgebungsvariable CLASSPATH
  - UML-Notation

- Zugriffsrechte für Klassen:
  - `public`, `abstract` und `final`

- Attribute (Vertiefung):
  - Klassenvariablen (Statische Attribute)
  - Klassen als Datentypen
  - Zugriffsrechte (`public`, `private`, `protected`)

Zusammenfassung (II)

- Methoden (Vertiefung):
  - Zugriffsrechte (`public`, `private`, `protected`)
  - Klasse als Parameter und Rückgabetyp
  - Klassenmethoden (Statische Methoden)
  - Statischer Import
  - Finale und abstrakte Methoden
  - Modifier `native` und `synchronized`
  - Namenskonventionen
  - Konstruktoren und Modifier

- Kapselung, ADT:
  - Kapselung
  - `get`- und `set`-Methoden
  - ADT

Aufgaben (I)

- **Aufgabe 1**

  Erweitern Sie Ihre Lösung der ersten Aufgabe aus dem letzten Abschnitt wie folgt:

  a) Ordnen Sie alle Klassen (außer Anwendung) einem Paket „personal“ zu!

  b) Realisieren Sie für alle Klassen des Pakets das Prinzip der Kapselung wie es in der Vorlesung vorgestellt worden ist!

Aufgaben (II)

- **Aufgabe 2**
  a) Entwerfen Sie eine Klassenbibliothek für zweidimensionale Objekte (Quadrat, Rechteck, Dreieck, Kreis, ...) in Form eines UML-Diagramms. Überlegen Sie sich die Klassenhierarchie. Kennzeichnen Sie alle Attribute und Methoden (sollen Berechnungen der Fläche und des Umfangs ermöglichen) mit Zugriffsmodifiern.
  b) Implementieren Sie das in a) entworfene Schema exemplarisch für mindestens zwei konkrete Klassen in einem Paket.
  c) Realisieren sie eine Klasse Anwendung, in der sie zu den konkreten Klassen Objekte anlegen und für diese jeweils die umgesetzten Methoden aufrufen.