Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 11)

Klassendiagramme: Überblick

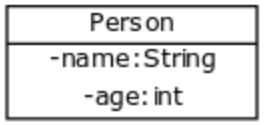
- **Klassendiagramme** sind eine visuelle Darstellung der Struktur von Software.
- Sie zeigen:
  - Klassen und ihre **Attribute**.
  - **Konstruktoren** und **Methoden**.
  - Beziehungen zwischen Klassen, wie Vererbung und Schnittstellen.
- Sie sind nützlich für:
  - **Planung** und **Modellierung** von Software.
  - Kommunikation zwischen Entwicklern auf hoher Abstraktionsebene.

Klassen und Attribute beschreiben

- Eine Klasse wird durch ein **Rechteck** dargestellt:
  - Oben: Name der Klasse.
  - Mitte: **Attribute** (z.B. `name: String`).
  - Unten: **Konstruktoren und Methoden**.

```java
public class Person {
    private String name;
    private int age;
}
```

- Diagramm:



- **+** bedeutet **öffentlich**, **-** bedeutet **privat**.

Methoden und Konstruktoren darstellen

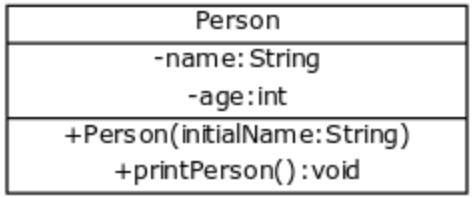
- Methoden und Konstruktoren werden unter den Attributen dargestellt.

```java
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.name = initialName;
        this.age = 0;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " + this.age + " years");
    }
}
```

- Diagramm:

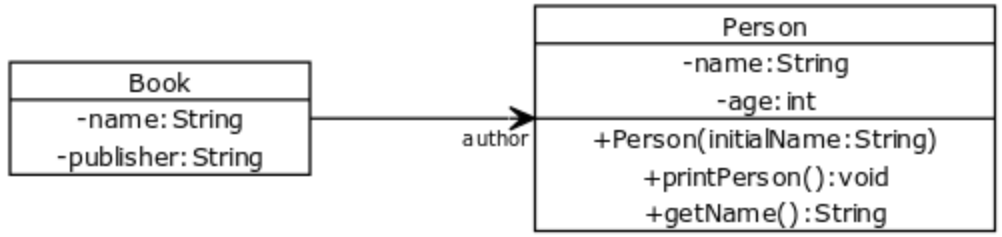


Verbindungen zwischen Klassen

- Klassen können **Beziehungen** zueinander haben, die durch **Pfeile** dargestellt werden.
- Beispiel:

```
public class Book {
    private String name;
    private String publisher;
    private Person author;
}
```

- Diagramm:



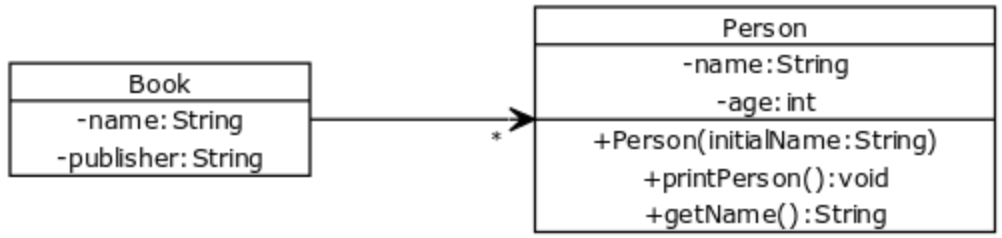
- Pfeile zeigen, dass ein `Book` einen `Person` als Autor hat.

Komplexe Beziehungen: M:N-Beziehungen

- Eine Klasse kann zu mehreren Objekten in **m:n** Beziehungen stehen.

```
public class Book {
    private ArrayList<Person> authors;
}
```

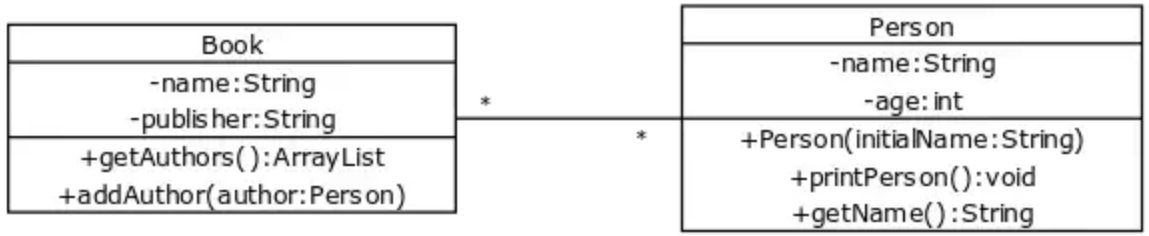
- Diagramm:



- **Sterne** zeigen an, dass mehrere Verbindungen möglich sind (z.B. viele Autoren und viele Bücher).

M:N Beziehungen: Bücher und Autoren

- In einer **Many-to-Many**-Beziehung kann eine Klasse mehrere Instanzen einer anderen Klasse kennen und umgekehrt.
- Beispiel:
  - Eine **Person** kann mehrere Bücher schreiben.
  - Ein **Buch** kann von mehreren Personen (Autoren) geschrieben werden.
- Darstellung:



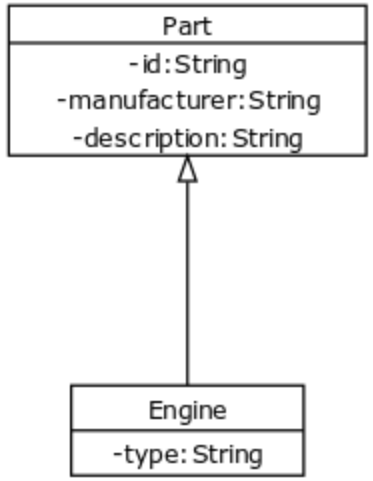
- Der Stern (`*`) am Ende der Pfeile zeigt an, dass beide Klassen viele Instanzen der jeweils anderen Klasse kennen/haben können.

Vererbung und Schnittstellen

- **Vererbung** wird durch einen **Pfeil mit Dreieck** dargestellt.

```
public class Engine extends Part {
}
```

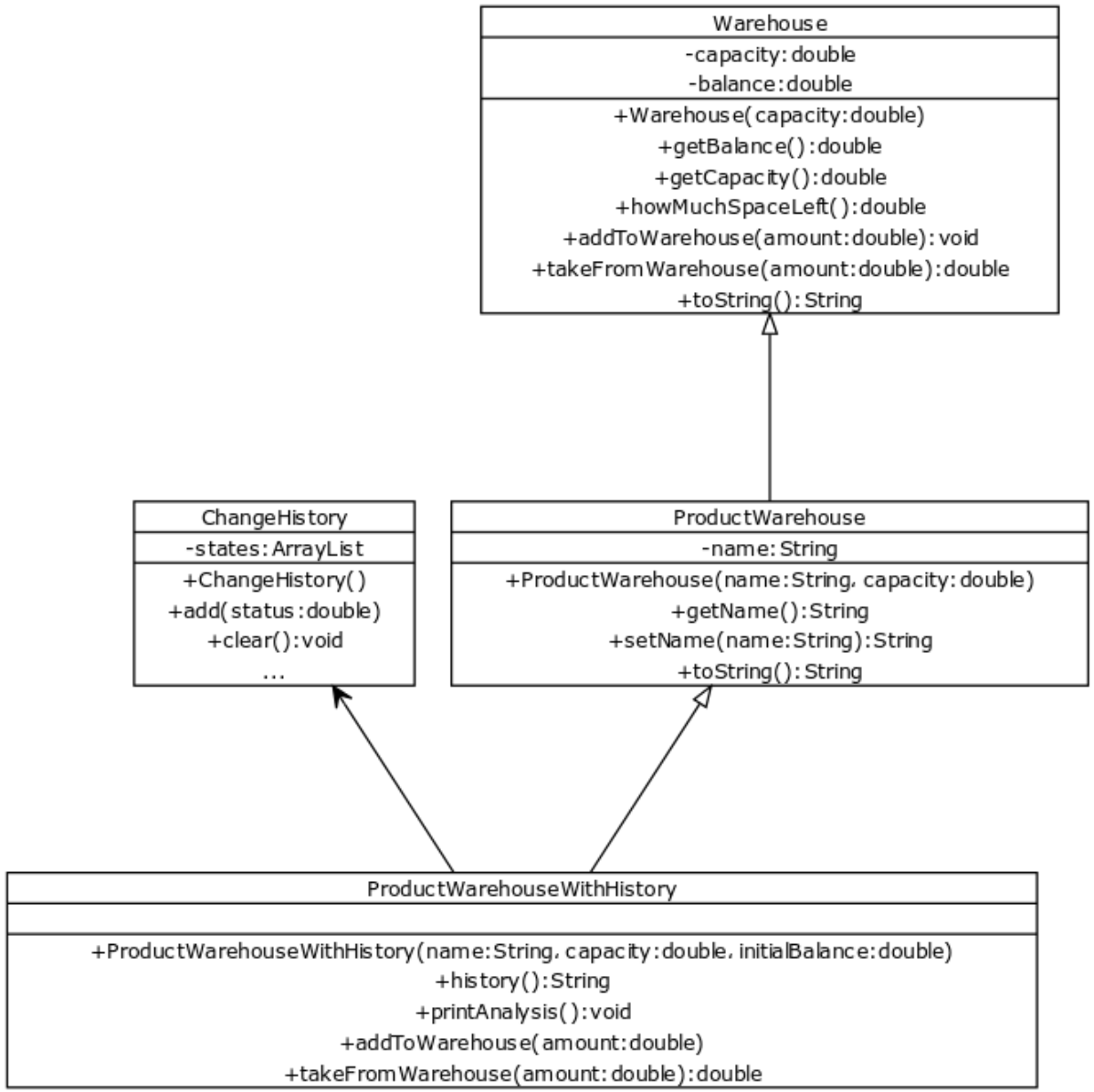
- Diagramm:



- Eine Klasse kann von einer abstrakten Klasse erben oder eine Schnittstelle implementieren.

Beispiel: Produktlager-Vererbung

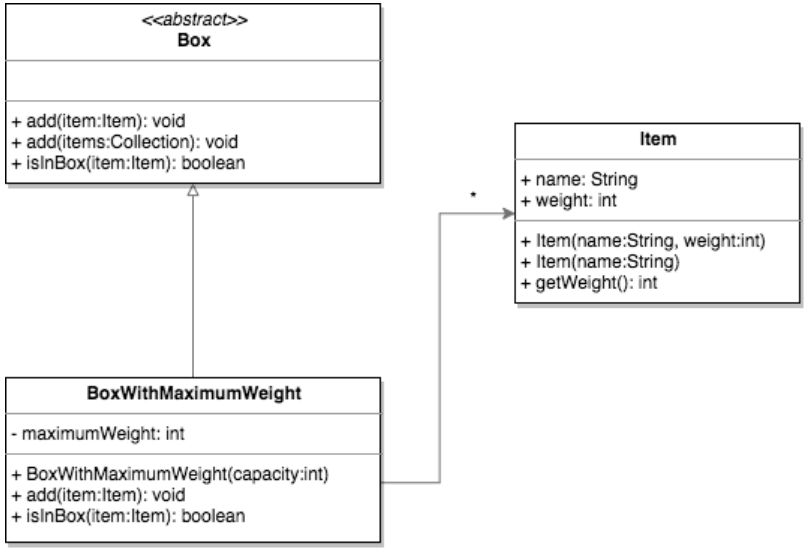
- **Mehrstufige Vererbung**: `ProductWarehouseWithHistory` erbt von `ProductWarehouse`, das von `Warehouse` erbt.
- **Verbindung zwischen Klassen**:
  - `ProductWarehouseWithHistory` kennt `ChangeHistory`, aber `ChangeHistory` kennt `ProductWarehouseWithHistory` nicht.


Warehouse
-capacity: double
-balance: double
+Warehouse(capacity: double)
+getBalance(): double
+getCapacity(): double
+howMuchSpaceLeft(): double
+addToWarehouse(amount: double): void
+takeFromWarehouse(amount: double): double
+toString(): String
ChangeHistory
-states: ArrayList
+ChangeHistory()
+add(status: double)
+clear(): void
...
ProductWarehouse
-name: String
+ProductWarehouse(name: String, capacity: double)
+getName(): String
+setName(name: String): String
+toString(): String
ProductWarehouseWithHistory
+ProductWarehouseWithHistory(name: String, capacity: double, initialBalance: double)
+history(): String
+printAnalysis(): void
+addToWarehouse(amount: double)
+takeFromWarehouse(amount: double): double

Abstrakte Klassen und Vererbung

- **Abstrakte Klassen** dienen als Vorlage für andere Klassen und können nicht direkt instanziiert werden.
- Sie definieren gemeinsame Attribute und Methoden, die von Unterklassen geerbt werden.
- Methoden in abstrakten Klassen können **abstrakt** sein, d.h. sie werden von den Unterklassen überschrieben.

- Darstellung:



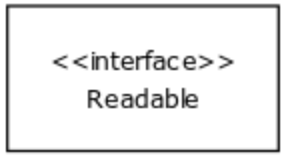
- Im Diagramm:
  - Die Klasse `<<abstract>>` `Box` definiert allgemeine Attribute und abstrakte Methoden.
  - `BoxWithMaximumWeight` erbt von `Box` und implementiert die abstrakten Methoden. Die `Box` hat mehrere `Item` s.

Schnittstellen in Klassendiagrammen

- In Klassendiagrammen werden Schnittstellen als `<<interface>> NameOfTheInterface` notiert.

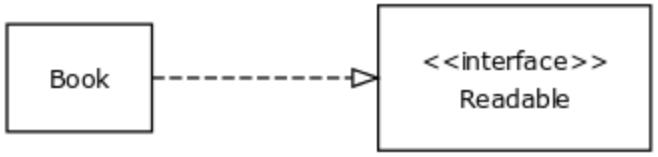
```
public interface Readable {
}
```

- Darstellung:



Implementierung von Schnittstellen

- Die Implementierung einer Schnittstelle wird durch einen **gestrichelten Pfeil** mit einem **dreieckigen Kopf** dargestellt.
- Beispiel:
  - Die Klasse `Book` implementiert die Schnittstelle `Readable`.
- Darstellung:



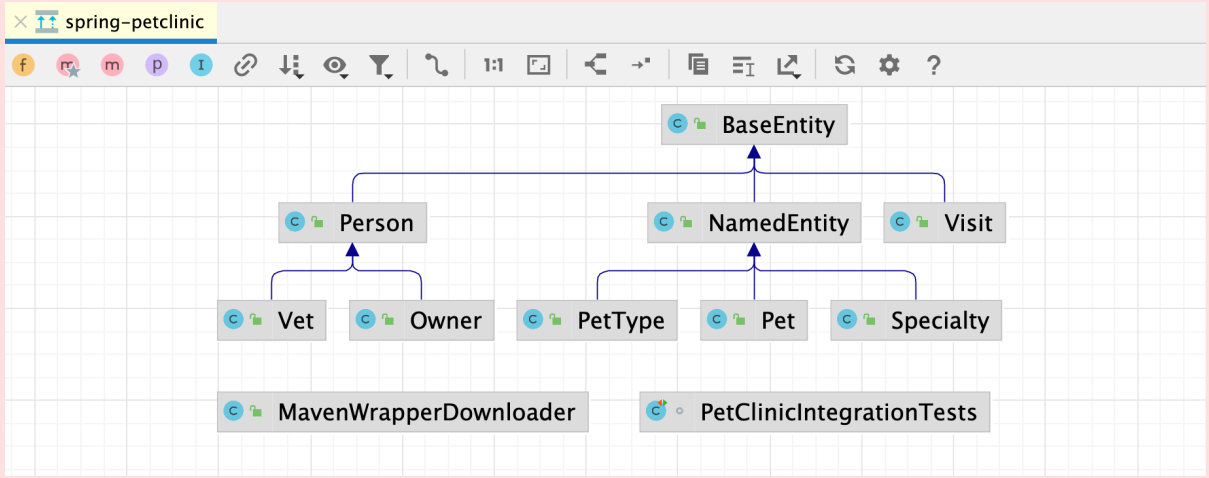
- Der gestrichelte Pfeil zeigt an, dass `Book` die Methoden von `Readable` implementiert.

Grundprinzipien: Warum Klassendiagramme wichtig sind

- **Abstraktion**: Sie trennen die **Struktur** eines Programms von der **Implementierung**.
- **Klarheit**: Sie ermöglichen Entwicklern, den Überblick über komplexe Systeme zu behalten.
- **Modularität**: Sie fördern die Wiederverwendbarkeit von Klassen und Konzepten.
- **Kommunikation**: Sie sind eine universelle Sprache für Entwickler, um komplexe Ideen zu teilen.

Klassendiagramme in IntelliJ

- Benötigt Ultimate Lizenz
- Details hier: https://www.jetbrains.com/help/idea/class-diagram.html



Klassendiagramme: Bedeutung im Software Engineering

- **Klassendiagramme** sind ein zentrales Werkzeug im Software Engineering.
- Sie ermöglichen:
  - **Abstrakte Modellierung** der Softwarestruktur.
  - **Dokumentation** von Softwarearchitekturen.
  - **Planung und Design** von Systemen vor der Implementierung.
- Klassendiagramme verbessern:
  - **Kommunikation** zwischen Entwicklern, Kunden und anderen Stakeholdern.
  - Die **Wartbarkeit** und **Erweiterbarkeit** von Softwareprojekten.

Vorteile von Klassendiagrammen im Entwicklungsprozess

1. **Frühzeitige Fehlererkennung**:
    - Probleme im Design können bereits in der Planungsphase erkannt werden.
2. **Modularität und Wiederverwendbarkeit**:
    - Durch die visuelle Darstellung der Beziehungen zwischen Klassen wird klar, welche Module wiederverwendbar sind.
3. **Systemverständnis**:
    - Neue Teammitglieder können schneller das **Systemdesign** und die **Architektur** verstehen.
4. **Wartung und Erweiterung**:
    - Klassendiagramme helfen, den Überblick über **Änderungen** zu behalten und die Auswirkungen von Erweiterungen zu erkennen.

Rolle von Klassendiagrammen in der Softwarearchitektur

- Klassendiagramme unterstützen die **Definition von Softwarearchitekturen**:
  - Sie beschreiben **Komponenten**, **Module** und deren **Interaktionen**.
  - Sie helfen, **Schichtenarchitekturen** zu planen (z.B. Präsentations-, Geschäfts- und Datenzugriffsschicht).
- Sie fördern die **Trennung von Verantwortlichkeiten**:
  - Klassen werden nach spezifischen Verantwortlichkeiten gruppiert (z.B. MVC: Model, View, Controller).
- **Design Patterns**:
  - Klassendiagramme helfen beim Erklären und Anwenden von **Entwurfsmustern** (z.B. Singleton, Observer, Factory).

Klassendiagramme und Entwurfsmuster

- **Entwurfsmuster** sind bewährte Lösungsansätze für wiederkehrende Probleme in der Softwareentwicklung.
- Klassendiagramme helfen, Entwurfsmuster klar und verständlich zu vermitteln:
  - **Singleton**: Eine Klasse, die nur eine Instanz zulässt.
  - **Observer**: Eine Klasse, die andere über Änderungen informiert.
  - **Factory**: Eine Klasse, die Objekte erstellt, ohne deren konkrete Typen zu kennen.
- Klassendiagramme zeigen:
  - **Beziehungen** zwischen den beteiligten Klassen.
  - **Vererbungs- und Implementierungsstrukturen** für die jeweilige Lösung.

Herausforderungen bei der Arbeit mit Klassendiagrammen

1. **Komplexität**:
    - Große Systeme führen zu komplexen Klassendiagrammen, die schwer zu lesen und zu pflegen sind.
2. **Veraltete Diagramme**:
    - Wenn die Diagramme nicht kontinuierlich aktualisiert werden, können sie vom tatsächlichen Code abweichen.
3. **Übermäßige Detaillierung**:
    - Es ist wichtig, das richtige **Abstraktionsniveau** zu finden, um Diagramme übersichtlich zu halten.
4. **Automatische Generierung**:
    - Werkzeuge wie IntelliJ oder UML-Tools können helfen, aber die manuelle Anpassung ist oft nötig.

Best Practices für Klassendiagramme

1. **Fokussierung auf Abstraktion**:
    - Konzentrieren Sie Sich auf die wichtigsten Klassen und Beziehungen, ohne jedes Detail zu zeigen.
2. **Modularität fördern**:
    - Zerlegen Sie große Systeme in kleinere, verständliche **Module** oder **Schichten**.
3. **Fortlaufende Aktualisierung**:
    - HaltenSie die Diagramme aktuell, um Änderungen am Systemdesign nachzuverfolgen.
4. **Dokumentation und Kommunikation**:
    - Nutzen Sie Klassendiagramme als Teil der **Dokumentation** und um mit dem Team und den Stakeholdern zu kommunizieren.

Pakete (Packages) in Java

- **Pakete** in Java sind eine Möglichkeit, Klassen zu organisieren und zu strukturieren.
- Sie gruppieren Klassen, die ähnliche **Funktionalitäten** oder **Zwecke** haben, und helfen, den Code übersichtlicher zu gestalten.
- Pakete ermöglichen es:
    - **Namenskonflikte** zu vermeiden.
    - Den Code in **logische Einheiten** zu unterteilen.
    - Die **Wartbarkeit** und **Wiederverwendbarkeit** zu verbessern.

Klassen in Paketen platzieren

- Jede Klasse kann einem Paket zugeordnet werden.
- Die Paketzugehörigkeit wird zu Beginn der Datei mit der Anweisung `package` definiert.

Beispiel:

```
package library;

public class Program {
    public static void main(String[] args) {
        System.out.println("Hello packageworld!");
    }
}
```

- Hier befindet sich die Klasse `Program` im Paket **library**.

Hierarchische Pakete

- Pakete können **hierarchisch** organisiert werden.
- Ein Paket kann weitere Pakete enthalten. Zum Beispiel:

```java
package library.domain;

public class Book {
    private String name;
    public Book(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

- Hier liegt die Klasse `Book` im Paket **library.domain**, einem Unterpaket von `library`.

Import von Klassen aus Paketen

- Um auf eine Klasse aus einem anderen Paket zuzugreifen, wird die `import` **-Anweisung** verwendet.

Beispiel:

```
package library;

import library.domain.Book;

public class Program {
    public static void main(String[] args) {
        Book book = new Book("The ABCs of packages!");
        System.out.println("Hello packageworld: " + book.getName());
    }
}
```

- Hier wird die Klasse `Book` aus dem Paket **library.domain** importiert.

Vorteile der Verwendung von Paketen

1. **Organisation**:
    - Pakete organisieren den Code und verbessern die Lesbarkeit.
2. **Namenskonflikte vermeiden**:
    - Durch Pakete können Klassen mit gleichem Namen in unterschiedlichen Paketen existieren.
3. **Modularität**:
    - Pakete fördern die Modularität und erleichtern die **Wartung** und **Erweiterung** von Code.
4. **Zugriffskontrolle**:
    - Klassen und Methoden können paketprivat gemacht werden, um ihre Sichtbarkeit zu begrenzen.

IDE-Unterstützung für Pakete

- Moderne IDEs wie **IntelliJ IDEA** unterstützen die Verwaltung von Paketen:
  - Neue Pakete können einfach erstellt werden: **File > New > Package**.
  - Klassen werden automatisch in den entsprechenden Verzeichnissen platziert.
  - Pakete ermöglichen es, Code in großen Projekten effizient zu organisieren.
- IDEs bieten zudem **Werkzeuge zur Paketverwaltung** und helfen, Klassen und Pakete konsistent zu organisieren.

Zugriffskontrolle in Paketen

- **Zugriffsspezifizierer** steuern den Zugriff auf Klassen und deren Mitglieder:

1. **public**:
    - Klasse oder Methode ist von überall zugänglich.
2. **package-private** (Standard):
    - Ohne Modifikator ist eine Klasse nur innerhalb desselben Pakets zugänglich.
3. **protected**:
    - Zugriff innerhalb des Pakets und in abgeleiteten Klassen.
4. **private**:
    - Zugriff nur innerhalb der Klasse selbst.

```
class PackagePrivateClass {
    // Nur innerhalb des Pakets zugänglich
}
```

Best Practices für die Arbeit mit Paketen

1. **Logische Gruppierung**:
    - Gruppiere Klassen nach ihrer **Funktionalität** (z.B. utils, models, services).
2. **Klarer Paketname**:
    - Verwende eindeutige, hierarchische Namen, die oft mit der **Domain** des Unternehmens beginnen (z.B. `com.company.project`).
3. **Minimale Abhängigkeiten**:
    - Halte die Abhängigkeiten zwischen Paketen gering, um die **Modularität** zu fördern.
4. **Zugriffskontrolle beachten**:
    - Mache nur das **public**, was wirklich von außen zugänglich sein muss.

Was ist ein Namespace?

- Ein **Namespace** ist ein **logischer Container** für Namen (z.B. Klassennamen), um Namenskonflikte zu vermeiden.
- Namen in einem Namespace sind **eindeutig** und können nicht mit Namen in einem anderen Namespace kollidieren.
- Namespaces werden in vielen Programmiersprachen verwendet, um Klassen, Funktionen oder Variablen zu **organisieren**.
- In Java wird ein Namespace durch **Pakete** implementiert.

Der "package"-Befehl in Java

- Der `package` **-Befehl** legt fest, in welchem **Namespace** sich eine Klasse befindet.
- Es definiert, zu welchem **Paket** eine Klasse gehört, und bestimmt ihre **Position** im Verzeichnisbaum.

Beispiel:

```
package com.example.myapp;
public class MyClass { }
```

- Die Klasse `MyClass` gehört hier zum Paket **com.example.myapp**.
- Ohne den `package` -Befehl befindet sich eine Klasse im **Standardpaket**.

Der "import"-Befehl in Java

- Der `import`-**Befehl** erlaubt den Zugriff auf Klassen, die in einem **anderen Paket (Namespace)** definiert sind.
- Er macht es möglich, auf Klassen aus anderen Paketen zuzugreifen, ohne den **vollständigen Paketnamen** bei jedem Zugriff zu schreiben.

Beispiel:

```
import com.example.myapp.MyClass;
public class Main {
    MyClass obj = new MyClass();
}
```

- `MyClass` aus **com.example.myapp** wird hier mit dem `import`-Befehl zugänglich gemacht.

Namespaces und Pakete in Java

- Ein **Paket** in Java dient als **Namespace**, der alle darin enthaltenen Klassen **eindeutig** identifiziert.
- Klassen im selben Paket können direkt aufeinander zugreifen, ohne einen Import.
- **Pakete** bieten eine klare Strukturierung und verhindern **Namenskonflikte**.
- `package` erstellt den Namespace und `import` ermöglicht den Zugriff auf Klassen in anderen Namespaces.

Beispiel für package und import

```java
// Definiert das Paket (Namespace) der Klasse
package library.domain;

public class Book {
    private String title;
    public Book(String title) { this.title = title; }
    public String getTitle() { return this.title; }
}
```

```java
// Importiert die Klasse aus einem anderen Paket (Namespace)
package library;

import library.domain.Book;

public class Program {
    public static void main(String[] args) {
        Book book = new Book("Learning Packages");
        System.out.println(book.getTitle());
    }
}
```

Zusammenfassung: package und import

1. **package**:
    - Definiert den **Namespace** einer Klasse.
    - Stellt sicher, dass Klassennamen **eindeutig** sind, indem sie in einem Paket (Namespace) gruppiert werden.
2. **import**:
    - Erlaubt den Zugriff auf Klassen aus **anderen Namespaces** (Paketen).
    - Erspart es, den **vollständigen Paketnamen** jedes Mal zu schreiben.
    - `java.util.ArrayList<String> list = new java.util.ArrayList<>();`
3. **Namespaces**:
    - Ein Werkzeug zur **Organisation** und **Namenskonfliktvermeidung**.
    - In Java wird dies durch **Pakete** realisiert.

Vorteile von Namespaces und Paketen

- **Vermeidung von Namenskonflikten**:
  - Zwei Klassen können denselben Namen haben, solange sie sich in unterschiedlichen Paketen befinden.
- **Struktur und Ordnung**:
  - Pakete helfen, den Code zu strukturieren und **logische Gruppierungen** von Klassen zu schaffen.
- **Wartbarkeit**:
  - Gut organisierte Pakete und Namespaces machen den Code einfacher zu pflegen und zu verstehen.

Zusammenfassung

- Pakete sind ein zentrales **Organisationswerkzeug** in Java.
- Sie fördern **Modularität**, **Namenskonfliktvermeidung** und **Zugriffskontrolle**.
- Durch die Verwendung von Paketen wird der Code übersichtlicher, wartbarer und wiederverwendbarer.
- In den nächsten Übungen werden wir regelmäßig Pakete verwenden, um den Code zu strukturieren.

Namespaces in Haskell und Python

Vergleich mit Java-Packages

In Java:

- **Packages** sind Sammlungen von Klassen und Schnittstellen.
- Sie strukturieren den Code und verhindern Namenskollisionen.

Haskell und Python haben ähnliche Konzepte:

- **Haskell**: Module
- **Python**: Module und Pakete

Namespaces in Haskell: Module

- **Module** sind Sammlungen von Funktionen, Typen und anderen Definitionen.
- Sie werden durch das `module` -Schlüsselwort definiert.

Beispiel

```haskell
module MyNamespace.MyModule (myFunction) where

myFunction :: Int -> Int
myFunction x = x + 1
```

Nutzung

```haskell
import MyNamespace.MyModule
main :: IO ()
main = print (myFunction 42)
```

Qualifizierte Importe in Haskell

- Um Namenskollisionen zu vermeiden, können Module qualifiziert importiert werden.

Beispiel

```haskell
import qualified MyNamespace.MyModule as M

main :: IO ()
main = print (M.myFunction 42)
```

Namespaces in Python: Module

- Jede **.py-Datei** ist ein Modul.
- Module strukturieren den Code und verhindern Namenskollisionen.

Beispiel

`mynamespace/mymodule.py`:

```
def my_function(x):
    return x + 1
```

Nutzung

```
from mynamespace.mymodule import my_function
print(my_function(42))
```

Namespaces in Python: Pakete

- Ein **Paket** ist ein Verzeichnis mit einer `__init__.py`-Datei.
- Pakete können Module und Unterpakete enthalten.

Beispielstruktur

```
mynamespace/
    __init__.py
    mymodule.py
```

Nutzung

```
import mynamespace.mymodule as mm
print(mm.my_function(42))
```

Einführung in Ausnahmen (Exceptions)

- **Ausnahmen** sind spezielle Ereignisse, die während der **Programmausführung** auftreten und den normalen Fluss eines Programms unterbrechen.
- In Java (und anderen Sprachen) werden Ausnahmen verwendet, um **Fehler** oder **unerwartete Situationen** zu handhaben, z.B.:
  - Division durch Null.
  - Zugriff auf eine ungültige Speicherposition.
  - Öffnen einer nicht existierenden Datei.
- Das Exception-Handling in Java stellt sicher, dass Programme trotz **Fehlern** stabil bleiben.

Warum Ausnahmen verwenden?

- **Trennung von normalem Code und Fehlerbehandlung**:
  - Der reguläre Programmablauf bleibt klar und lesbar, während Fehler separat behandelt werden.
- **Vorhersagbarkeit**:
  - Ausnahmen ermöglichen es, auf **bestimmte Fehler** zu reagieren und Maßnahmen zu ergreifen, anstatt dass das Programm unerwartet abstürzt.
- **Wartbarkeit**:
  - Besser organisierter Code, da Fehlerbehandlung an einem Ort zentralisiert wird.

Arten von Ausnahmen in Java

1. **Checked Exceptions**:
   - Müssen entweder behandelt oder in der Methodensignatur angegeben werden (`throws`).
   - Beispiel: `IOException` bei Dateioperationen.
2. **Unchecked Exceptions**:
   - Tritt zur Laufzeit auf, wird aber **nicht erzwungen**, um sie zu behandeln.
   - Beispiel: `NullPointerException` , `ArrayIndexOutOfBoundsException` .
3. **Errors**:
   - Schwere Fehler, die nicht vom Programm behandelt werden sollten.
   - Beispiel: `OutOfMemoryError` .

Unterschied zwischen Error und Exception

Kriterium	`Error`	`Exception`
Ursprung	Probleme der JVM oder Umgebung	Probleme der Anwendung oder Eingabe
Behandelbarkeit	Sollte nicht behandelt werden	Kann und sollte behandelt werden
Typische Beispiele	`OutOfMemoryError`, `StackOverflowError`	`IOException`, `NullPointerException`
Vererbung	Erbt von `Throwable`	Erbt von `Throwable`
Prüfung	Keine Kompilierzeitprüfung	Kann Checked oder Unchecked sein

Umgang mit Ausnahmen: try-catch

- **try-catch**-Blöcke fangen Ausnahmen ab und behandeln sie.

```
try {
    // Code, der eine Ausnahme auslösen könnte
    int result = 10 / 0;
} catch (ArithmeticException e) {
    // Behandlung der Ausnahme
    System.out.println("Division durch Null ist nicht erlaubt.");
}
```

- **try** : Block des risikobehafteten Code, **catch** fängt die Ausnahme und behandelt sie.

Mehrere catch-Blöcke

- Es können mehrere **catch-Blöcke** verwendet werden, um unterschiedliche Ausnahmetypen zu behandeln.

```
try {
    int[] arr = new int[5];
    arr[10] = 50;
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array-Index außerhalb des gültigen Bereichs.");
} catch (Exception e) {
    System.out.println("Ein allgemeiner Fehler ist aufgetreten.");
}
```

- Jede Ausnahme wird individuell behandelt.

finally-Block

- Der **finally-Block** wird immer ausgeführt, unabhängig davon, ob eine Ausnahme auftritt oder nicht.

```
try {
    // risikobehafteter Code
} catch (Exception e) {
    // Behandlung der Ausnahme
} finally {
    // Code, der immer ausgeführt wird
    System.out.println("Dieser Block wird immer ausgeführt.");
}
```

- Wird oft verwendet, um **Ressourcen** wie Dateien oder Datenbankverbindungen zu schließen.

```java
import java.io.*;

public class FinallyExample {
    public static void main(String[] args) {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader("test.txt"));
            System.out.println(reader.readLine());
        } catch (IOException e) {
            System.out.println("Fehler beim Lesen der Datei: " + e.getMessage());
        } finally {
            try {
                if (reader != null) {
                    reader.close(); // Aufräumen der Ressource
                    System.out.println("Datei geschlossen.");
                }
            } catch (IOException e) {
                System.out.println("Fehler beim Schließen der Datei: " + e.getMessage());
            }
        }
    }
}
```

Checked Exceptions: Java erzwingt Behandlung

- **Checked Exceptions** müssen vom Programmierer behandelt werden.
- Tritt zur Kompilierzeit auf, wenn keine Behandlung erfolgt.
- **Beispiele:**
  - `IOException`
  - `SQLException`
  - `FileNotFoundException`

Beispiel: Dateioperation ohne Ausnahmebehandlung

```
import java.io.FileReader;

public class Example {
    public static void main(String[] args) {
        FileReader reader = new FileReader("example.txt");
    }
}
```

Fehler:

```
Unhandled exception: java.io.FileNotFoundException
```

Lösung 1: Behandlung mit `try-catch`

```
import java.io.FileReader;
import java.io.IOException;

public class Example {
    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader("example.txt");
            System.out.println("Datei geöffnet.");
        } catch (IOException e) {
            System.out.println("Fehler beim Öffnen der Datei: " + e.getMessage());
        }
    }
}
```

- Die Ausnahme wird im `catch` -Block behandelt.
- Der Code kompiliert und läuft fehlerfrei, falls die Datei fehlt.

Lösung 2: Weitergeben mit `throws`

```java
import java.io.FileReader;
import java.io.IOException;

public class Example {
    public static void main(String[] args) throws IOException {
        FileReader reader = new FileReader("example.txt");
        System.out.println("Datei geöffnet.");
    }
}
```

- Die Methode gibt die Verantwortung weiter, die Ausnahme zu behandeln.
- Der Aufrufer der Methode muss die Ausnahme behandeln.

Unchecked Exceptions: Keine Pflicht zur Behandlung

- **Unchecked Exceptions** müssen nicht behandelt werden.
- Beispiele:
  - `NullPointerException`
  - `ArithmeticException`

Beispiel: Unchecked Exception

```java
public class Example {
    public static void main(String[] args) {
        int result = 10 / 0; // ArithmeticException
    }
}
```

Keine Kompilierfehler, aber Laufzeitfehler:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Eigene Ausnahmen erstellen

- Eigene **benutzerdefinierte Ausnahmen** können durch das Erstellen einer neuen Klasse, die von `Exception` erbt, erstellt werden.

```
public class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}
```

- Verwendung:

```
throw new MyException("Benutzerdefinierte Ausnahme ausgelöst!");
```

- Nützlich, um **spezifische Fehlerzustände** in Anwendungen zu definieren.

Prinzipien der Ausnahmebehandlung

1. **Nur Ausnahmen fangen, die man auch behandeln kann**:
    - Fange nur solche Ausnahmen ab, die sinnvoll verarbeitet werden können.
2. **Spezifischere Ausnahmen zuerst abfangen**:
    - Mehrere `catch`-Blöcke sollten von spezifisch zu allgemein geordnet sein.
3. **Fehlerbehandlung zentralisieren**:
    - Ermöglicht eine einfachere Wartung und sorgt für konsistente Fehlerverarbeitung.

Ausnahmen und Ressourcen: try-with-resources

- Die `try-with-resources`-Struktur vereinfacht den Umgang mit Ressourcen, wie Dateien oder Netzwerkverbindungen.
- Ressourcen werden automatisch geschlossen, sobald sie nicht mehr benötigt werden.

```
try (Scanner reader = new Scanner(new File("file.txt"))) {
    while (reader.hasNextLine()) {
        System.out.println(reader.nextLine());
    }
} catch (Exception e) {
    System.out.println("Exception: " + e.getMessage());
}
```

- Diese Technik verhindert, dass Ressourcen wie Dateien oder Streams offen bleiben und dadurch Systemressourcen blockieren.

Verantwortung weitergeben: `throws`

- Methoden können Ausnahmen nicht nur behandeln, sondern auch an den Aufrufer **weitergeben**.
- Eine Methode, die eine Ausnahme weitergibt, verwendet die Deklaration `throws Exception`.

```
public List<String> readLines(String fileName) throws Exception {
    ArrayList<String> lines = new ArrayList<>();
    Files.lines(Paths.get(fileName)).forEach(line -> lines.add(line));
    return lines;
}
```

- Die Ausnahme wird hier **nicht behandelt**, sondern an den Aufrufer delegiert.

Ausnahmen werfen: `throw`

- Programmierer können Ausnahmen auch manuell mit dem Befehl `throw` auslösen.
- Zum Beispiel: `IllegalArgumentException` bei ungültigen Argumenten.

```
public class Grade {
    public Grade(int grade) {
        if (grade < 0 || grade > 5) {
            throw new IllegalArgumentException("Grade must be between 0 and 5.");
        }
    }
}
```

```
Grade validGrade = new Grade(3);
Grade invalidGrade = new Grade(22);  // Exception wird hier geworfen
```

- Diese Technik ist nützlich für **Validierungen** oder **Fehlerfälle**, die nicht implizit durch das System erkannt werden.

Einige Übliche Ausnahmen in Java

1. **NullPointerException**
   Wird ausgelöst, wenn versucht wird, auf eine Referenz zuzugreifen, die `null` ist.

2. **ArrayIndexOutOfBoundsException**
   Wird ausgelöst, wenn versucht wird, auf ein Arrayelement außerhalb seiner Grenzen zuzugreifen.

3. **ArithmeticException**
   Wird bei mathematischen Fehlern, wie Division durch Null, ausgelöst.

4. **NumberFormatException**
   Tritt auf, wenn versucht wird, eine ungültige Zeichenkette in eine Zahl zu parsen.

5. **IOException**
   Wird ausgelöst, wenn ein Eingabe-/Ausgabefehler auftritt, z.B. beim Lesen einer Datei.

6. **FileNotFoundException**
   Wird ausgelöst, wenn versucht wird, auf eine Datei zuzugreifen, die nicht existiert.

7. **ClassNotFoundException**
   Wird ausgelöst, wenn versucht wird, eine Klasse dynamisch zu laden, die nicht gefunden werden kann.

8. **IllegalArgumentException**
   Tritt auf, wenn einer Methode ungültige Argumente übergeben werden.

9. **IllegalStateException**
   Wird ausgelöst, wenn der Zustand eines Objekts ungültig ist.

10. **InterruptedException**
    Tritt auf, wenn ein Thread unterbrochen wird, während er wartet oder schläft.

Beispiel 1: NullPointerException

```
String text = null;
System.out.println(text.length());
```

- **Ausnahme:** `NullPointerException`
- Grund: Der Aufruf `text.length()` wird auf einem `null`-Objekt durchgeführt.

Beispiel 2: ArrayIndexOutOfBoundsException

```
int[] numbers = {1, 2, 3};
System.out.println(numbers[3]);
```

- **Ausnahme:** `ArrayIndexOutOfBoundsException`
- Grund: Der Index `3` ist außerhalb der Grenzen des Arrays.

Beispiel 3: NumberFormatException

```
String numberString = "ABC";
int number = Integer.parseInt(numberString);
```

- **Ausnahme:** `NumberFormatException`
- Grund: Die Zeichenkette `"ABC"` kann nicht in eine Ganzzahl umgewandelt werden.

Beispiel 4: ArithmeticException

```
int result = 10 / 0;
```

- **Ausnahme:** `ArithmeticException`
- Grund: Division durch Null ist nicht definiert.

Aber(!):

```
double result = 10.0 / 0.0;
System.out.println(result);  // Ausgabe: Infinity
```

Beispiel 5: FileNotFoundException

```
Scanner reader = new Scanner(new File("non_existent_file.txt"));
```

- **Ausnahme:** `FileNotFoundException`
- Grund: Die Datei `"non_existent_file.txt"` existiert nicht.

Beispiel für `throws Exception` (Delegation)

```java
public void readFile(String fileName) throws FileNotFoundException {
    Scanner reader = new Scanner(new File(fileName));
    while (reader.hasNextLine()) {
        System.out.println(reader.nextLine());
    }
}
```

- **Delegation:** Diese Methode behandelt nicht die Ausnahme, sondern überlässt sie dem Aufrufer durch `throws FileNotFoundException`.

Beispiel für try-catch

```java
public void readNumber() {
    try {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Geben Sie eine Zahl ein: ");
        int number = Integer.parseInt(scanner.nextLine());
        System.out.println("Die Zahl ist: " + number);
    } catch (NumberFormatException e) {
        System.out.println("Das war keine gültige Zahl.");
    }
}
```

- **Exception:** `NumberFormatException`
- **Erklärung:** Die Eingabe wird in einen Integer konvertiert, und falls eine ungültige Zahl eingegeben wird, fängt der `catch`-Block die Ausnahme ab.

(Teil der) Vererbungshierarchie

```
Throwable
├── Exception
│   ├── IOException
│   ├── SQLException
│   ├── ClassNotFoundException
│   └── RuntimeException (unchecked)
│       ├── NullPointerException
│       ├── ArithmeticException
│       ├── ArrayIndexOutOfBoundsException
│       └── IllegalArgumentException
└── Error
    ├── OutOfMemoryError
    └── StackOverflowError
```

Zusammenfassung

- **Ausnahmen** sind unvermeidlich, aber durch **Exception-Handling** können Programme auf Fehler reagieren und **stabil** bleiben.
- Verwenden Sie **try-catch-finally** für sicheren und wartbaren Code.
- **Checked und Unchecked Exceptions**: Verstehen, wann und wie sie behandelt werden müssen.
- **Eigene Ausnahmen** erstellen, um spezialisierte Fehlerzustände zu handhaben.

Quiz

Dateien lesen und schreiben

(zum Teil als Wiederholung)

Lernziele

- Sie wissen, wie Daten aus Dateien gelesen werden.
- Sie können Daten in eine Datei schreiben.

Einführung

- Java bietet verschiedene Klassen zum Lesen und Schreiben von Dateien.
- Wir verwenden die Klasse `PrintWriter`, um Daten in Dateien zu schreiben.

Schreiben in eine Datei

```
PrintWriter writer = new PrintWriter("file.txt");
writer.println("Hello file!");
writer.println("More text");
writer.print("And a little extra");
writer.close();
```

- `println` schreibt eine neue Zeile.
- `print` fügt keinen Zeilenumbruch hinzu.
- `close()` speichert und schließt die Datei.

Ausnahmen behandeln

```java
public class Storer {
    public void writeToFile(String fileName, String text) throws Exception {
        PrintWriter writer = new PrintWriter(fileName);
        writer.println(text);
        writer.close();
    }
}
```

- `PrintWriter`-Konstruktor kann eine Ausnahme werfen.
- Die Methode gibt die Ausnahme weiter (`throws Exception`).

Beispiel: Schreiben in eine Datei

```java
public static void main(String[] args) throws Exception {
    Storer storer = new Storer();
    storer.writeToFile("diary.txt", "Dear diary, today was a good day.");
}
```

- Erstellt eine Datei "diary.txt" und schreibt Text hinein.
- Existierender Inhalt wird überschrieben.
- `main` in Kombination mit `throws Exception` : Indikator für schlechten Stil

Dateiinhalt anhängen

- Wenn der neue Text zum bestehenden Inhalt hinzugefügt werden soll, verwenden wir die Klasse `FileWriter`.

Unterschied: FileWriter vs. PrintWriter

Merkmal	`FileWriter`	`PrintWriter`
Primäre Funktion	Grundlegendes Schreiben von Zeichen.	Formatierte und "lesbare" Ausgabe.
Methoden	`write(char c)`, `write(String s)`	`print()`, `println()`, `printf()`.
Pufferung	Keine Pufferung	Gepufferte Ausgabe (effizienter).
Formatierung	Keine Unterstützung	Unterstützt Formatierung (z. B. `printf`).
Einsatzgebiet	Schreiben von Basis-Textdaten.	Schreiben von formatiertem Text.
Fehlerbehandlung	Kann `IOException` werfen.	Kann Fehler unterdrücken (optional).

Zusammenfassung

- Verwenden Sie `PrintWriter`, um in Dateien zu schreiben.
- `print` und `println` unterscheiden sich in der Handhabung von Zeilenumbrüchen.
- Ausnahmen sollten behandelt oder delegiert werden.