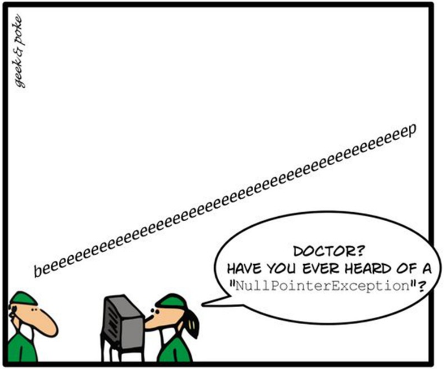
Interfaces, abstrakte Klassen & Exceptions

Vorlesungsinhalt



RECENTLY IN THE OPERATING ROOM

Vererbung

Vererbung

Definition einer Unterklasse

```
... class  Unterklasse extends Oberklasse{
    ... }
```

- die Unterklasse erbt Exemplarmethoden und Exemplarvariablen ihrer Oberklasse, nicht aber deren Konstruktoren
- eine Unterklasse kann zusätzliche Exemplarvariablen definieren
- eine Unterklasse kann zusätzliche Methoden definieren
- eine Unterklasse kann Methoden überschreiben (später)

Vier Tests für den richtigen Einsatz von Vererbung

1. **is-a Test**
   Jedes Objekt der Unterklasse ist ein Objekt der Oberklasse.
2. **behaves-like-a Test**
   Jedes Objekt der Unterklasse verhält sich wie ein Objekt der Oberklasse.
3. **extends Test**
   Jedes Objekt der Unterklasse hat zusätzliches Verhalten oder zusätzliche Statusinformationen.
4. **steadiness Test**
   Ein Objekt will seine Oberklasse niemals wechseln.

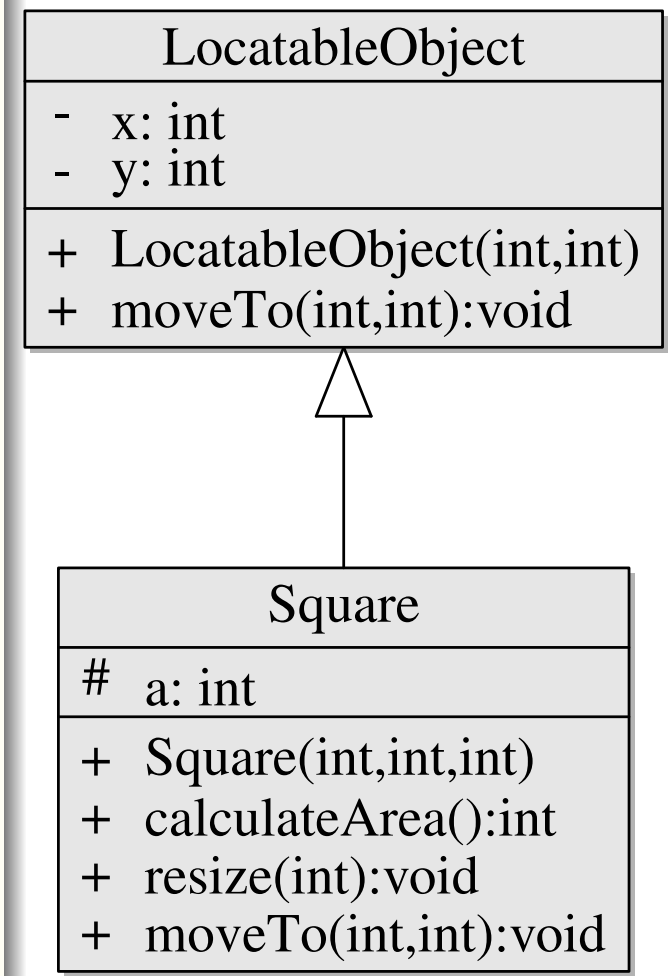
Konstruktoren von Unterklassen

- Bei der Erzeugung eines Objekts der Unterklasse wird zuerst der Default-Konstruktor (No-Argument-Constructor) der Oberklasse aufgerufen.
- Mit **super** kann statt des Default-Konstruktors ein anderer Konstruktor der Oberklasse aufgerufen werden. Dieser Aufruf muss als 1. Anweisung im Konstruktor stehen.
- Ist in der Oberklasse kein Default-Konstruktor definiert, muss **super(...)** im Konstruktor der Unterklasse aufgerufen werden
- Alle Klassen haben die Klasse `java.lang.Object` als Oberklasse

Konstruktoren von Unterklassen - Beispiel

Beispiel

```
class Square extends LocatableObject{
  protected int a;
  public Square(int x,int y,int a){
    super(x,y);
    this.a = a;
  }
  public void resize(int a){
    this.a = a;
  }
  public int calculateArea(){
    return a * a;
  }
  public void moveTo(int x, int y){
    super.moveTo(x,y);
    // Methode so ueberfluessig!
} }
```



Kapselung

- weitere Möglichkeiten zur Datenkapselung
  - mit **public** deklarierte Variablen und Methoden werden vererbt und sind überall benutzbar
  - mit **protected** deklarierte Variablen und Methoden werden vererbt, sind aber außerhalb ihres Paketes (s. später) nicht zugreifbar
  - ohne Modifier deklarierte Variablen (**package-private**) werden nicht vererbt, sind aber im selben Package zugreifbar
  - mit **private** deklarierte Variablen und Methoden werden nicht vererbt und sind nur in der deklarierenden Klasse zugreifbar.
- Standardansatz: Exemplarvariablen sind **private**
- außer man will sie in einer Unterklasse verwenden, dann sind sie **protected**

Access Level Modifiers - Überblick

Modifier	Klasse	Package	Subclass	Global
`public`	ja	ja	ja	ja
`protected`	ja	ja	ja	nein
ohne (`package-private`)	ja	ja	nein	nein
`private`	ja	nein	nein	nein

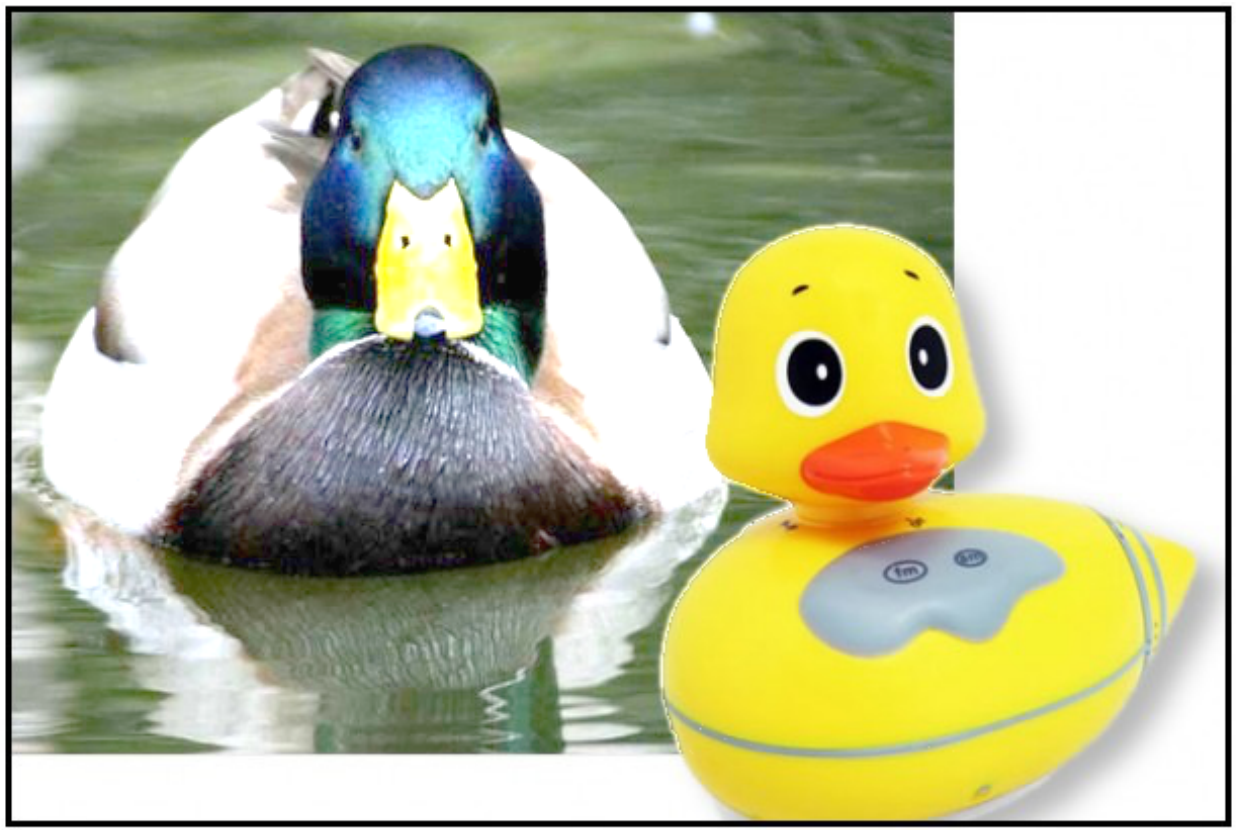
Liskov'sches Substitutionsprinzip

Das Liskov'sche Substitutionsprinzip

G. Starke (2011): Effektive Software-Architekturen

„Klassen sollen in jedem Fall durch ihre Unterklassen ersetzbar sein."

It's all about abtractions...



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Gute vs. schlechte Vererbung

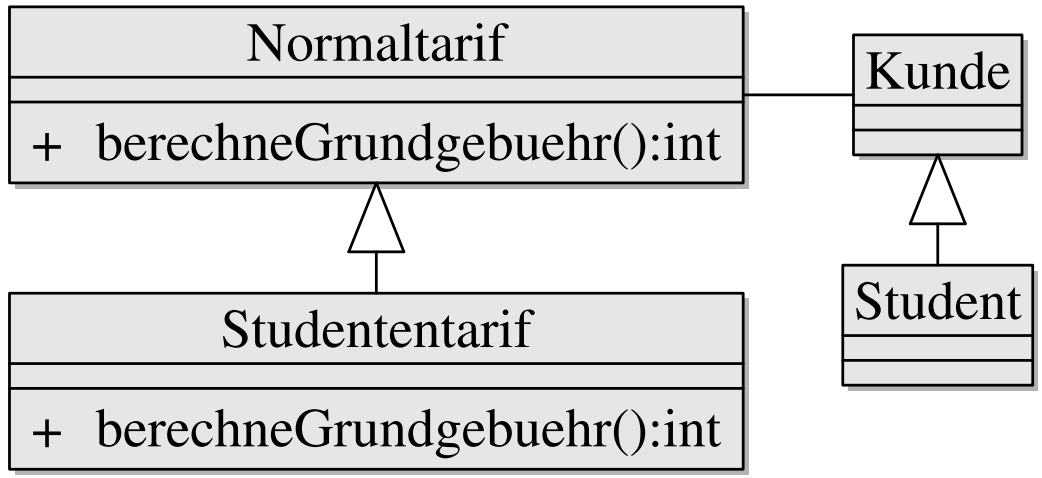
Das Liskovsche Substitutionsprinzip unterscheidet **gute** Vererbung von **schlechter** Vererbung.

Substitutionsforderung: Jede abgeleitete Klasse kann ihre Oberklasse ersetzen, es folgt:

- überschreibende Methoden dürfen in der abgeleiteten Klasse nur schwächere Vorbedingungen haben (größerer Wertebereich bei Eingabeparametern)
- überschreibende Methoden dürfen in der abgeleiteten Klasse nur stärkere Nachbedingungen haben (kleinerer Wertebereich bei der Ausgabe)

Übergeordnetes Verhalten darf nicht durch Unterklassen gebrochen werden.

Negativbeispiel zum LSP

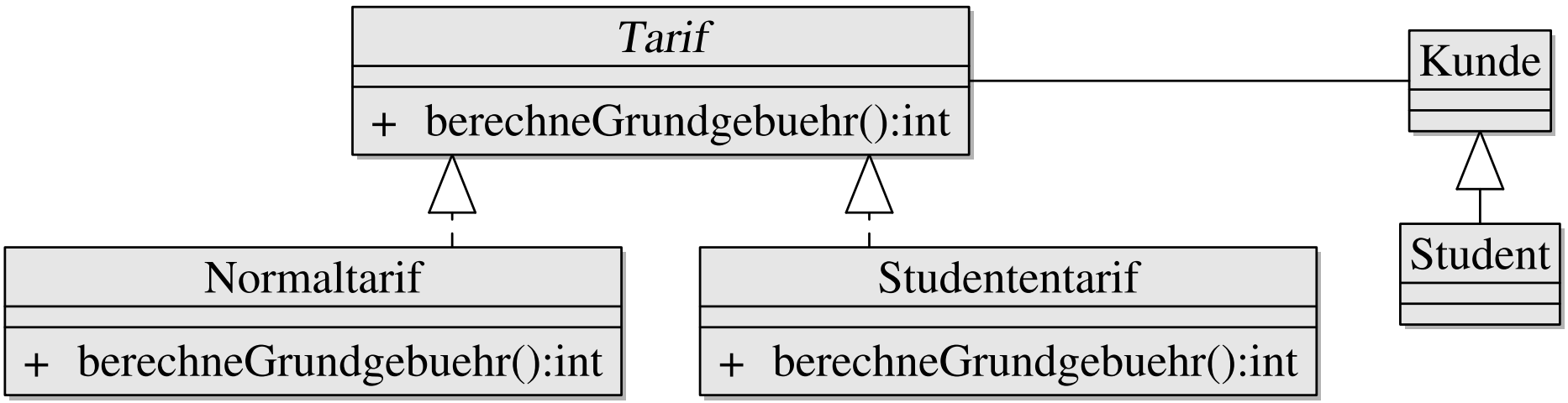


- Studententarif gilt nur für Studenten: Erweiterung der Vorbedingung⇒stärkere Vorbedingung
- berechne Grundgebuehr in Studententarif liefert anderes Ergebnis: das ist eine Änderung der Nachbedingung⇒keine stärkere Nachbedingung

Nachteile:

- Nutzer der Klasse müssen die Unterklassen kennen
- Keine Erweiterung (extends) sondern Modifikation

Korrigiertes Beispiel



- Studententarif ist keine Unterklasse von Normaltarif, weil sie anderes Verhalten zeigt
- abstrakte Klasse Tarif enthält Gemeinsamkeiten

Abstrakte Klassen

Typabstraktion: Abstrakte Klassen

- Eigenschaften
  - abstrakte Klassen sind Schablonen $\Rightarrow$ unvollständig implementiert
  - von ihnen können keine Exemplare gebildet werden (also kein **new** möglich)
  - Konstruktoren möglich, mit super in Unterklasse aufgerufen
- eine Klasse muss mit **abstract** deklariert sein
  - wenn mindestens eine Methode mit **abstract** deklariert wurde, oder
  - wenn eine abstrakte Methode von der Oberklasse geerbt, und nicht überschrieben wird

Abstrakte Methoden

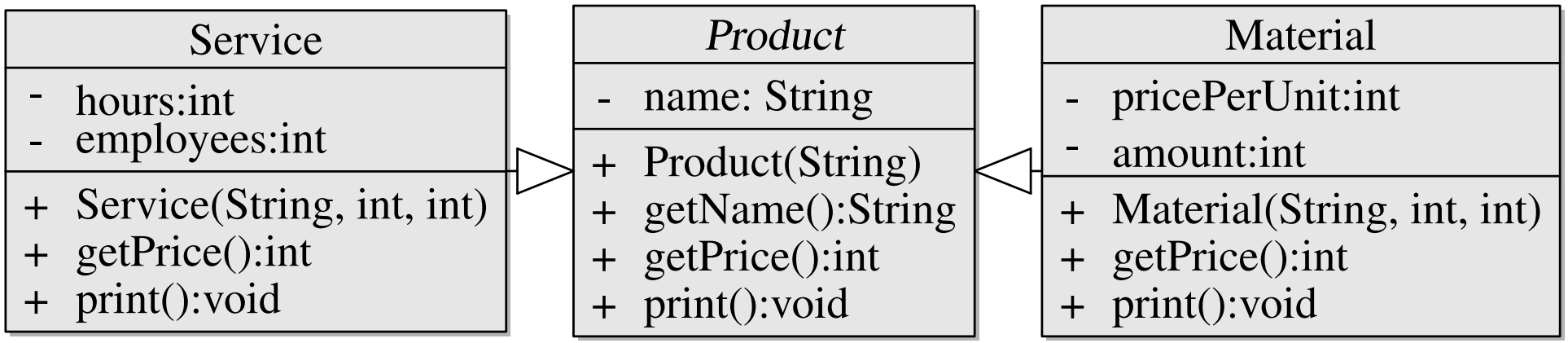
Abstrakte Methoden werden mit Modifier `abstract` deklariert

- benötigt keine Implementierung
- eine abstrakte Methode darf nicht mit static oder final deklariert werden
- muss von der Unterklasse implementiert werden, um Objekte der Unterklasse bilden zu können

Beispiel

```
public abstract class Fahrzeug {
     public abstract void starte();
}

public class Fahrrad extends Fahrzeug{
    public void starte(){
        steigeAuf(); . . .
    }
}
```

Wann benutzt man abstrakte Klassen?



- Wenn man in mehreren Unterklassen dasselbe Verhalten vorfindet.
- Wenn mehrere Unterklassen identische Zustandsvariablen haben.
- Zur Umsetzung des DRY Prinzips.
- Als Wurzelknoten einer Implementationshierarchie.
- Zur Vermeidung modifizierender Implementationsvererbung.
- Sollten wenigstens eine Methode implementieren oder eine Exemplarvariable haben, sonst nutzt man Interfaces.

Bad Smell: Basisklassen referenzieren Subklassen 1/2

Negativbeispiel (Teil 1)

```
public class SubclassA extends BaseClass {
    // ...
}
public class SubclassB extends BaseClass {
    // ...
}
public class SubclassC extends BaseClass {
    // ...
}
```

Bad Smell: Basisklassen referenzieren Subklassen 2/2

Negativbeispiel (Teil 2)

```
public class BaseClass {
  public static void main(String[] args){
        // ...
        BaseClass smellyClass = null;
        if ("A".equals(subclassType)) {
              smellyClass = new SubclassA();
        } else if ("B".equals(subclassType)) {
              smellyClass = new SubclassB();
        } else if ("C".equals(subclassType)) {
              smellyClass = new SubclassC();
        }
        // ...
  }
}
```

Lösung: Basisklassen referenzieren Subklassen 1/5

Nachteile:

- Die Basisklasse kennt ihre Spezialisierungen
- Das Hinzufügen von neuen Subklassen oder Anpassungen an bestehenden resultieren häufig in einer Anpassung der Basisklasse
- Widerspricht dem OO-Gedanken von Basis und Subklassen (Herausfaktorieren gemeinsamer Funktionalität in Basisklassen)

Lösung: Basisklassen referenzieren Subklassen 2/5

Die Klasse `BaseClass` hat zu viele Aufgaben:

- Start des Programms
- Die Auswahl der konkreten Subklasse für das Agieren als Basisklasse
- Agieren als Basisklasse

Empfehlungen für ein neues Design:

- Umsetzung durch Anwendung des Fabrikmethode-Musters
- Aufspaltung der verschiedenen Aufgaben in andere Klassen
- Verwendung von `enum`-Aufzählungen statt eines Strings

Lösung: Basisklassen referenzieren Subklassen 3/5

Lösungsvorschlag (Teil 1)

```
public abstract class BaseClass {
    // ...
}
public class SubclassA extends BaseClass {
    // ...
}
public class SubclassB extends BaseClass {
    // ...
}
public class SubclassC extends BaseClass {
    // ...
}
```

Lösung: Basisklassen referenzieren Subklassen 4/5

Lösungsvorschlag (Teil 2)

```
public class BaseClassFactory {
  public BaseClass create(final SubclassType subclassType){
    if (SUB_TYPE_A.equals(subclassType)) {
      return new SubclassA();
    } else if (SUB_TYPE_B.equals(subclassType)) {
      return new SubclassB();
    } else if (SUB_TYPE_C.equals(subclassType)) {
      return new SubclassC();
    }
    return null;
  }
}
```

Lösung: Basisklassen referenzieren Subklassen 5/5

Lösungsvorschlag (Teil 3)

```
public enum SubclassType {
  SUB_TYPE_A, SUB_TYPE_B, SUB_TYPE_C
}

public class App {
  public static void main(String[] args){
    // ...
    final BaseClassFactory factory = new BaseClassFactory();
    final BaseClass baseClass = factory.create(subclassType);
    // ...
  }
}
```

Interfaces

Good fences make good neighbours

- Implementierungen ändern sich häufig.
- Schnittstellen können und müssen über längere Zeit unverändert gehalten werden.
- Methoden einer Schnittstelle sind sorgsam zu wählen.
- Schnittstellen ermöglichen die Zusammenarbeit mehrerer Personen und Teams. Sie enthalten Verantwortlichkeiten, die eine Klasse übernimmt, ohne die Umsetzung der Verantwortlichkeit einzuschränken.
- Die Trennung von Implementation und Schnittstelle ermöglicht die ungestörte Zusammenarbeit von Entwickler und Nutzer einer Klasse.

Interfaces

- wie bei abstrakten Klassen stellt eine Schnittstelle eine Schablone zur Verfügung
  - die Schnittstelle darf keine Implementierung enthalten
  - ab Java 8 ist **default** Implementierung möglich
  - nur Konstanten sind als Attribute erlaubt
  - von einer Schnittstelle kann kein Exemplar erzeugt werden
- eine Schnittstelle bietet aber weitere Möglichkeiten
  - es können mehrere Schnittstellen von einer Klasse implementiert werden
  - Mehrfachvererbung von Deklarationen wird dadurch ermöglicht

```
// Definition:
interface MyInterface1 {..}
// Realisierung:
class MyClass implements MyInterface1, MyInterface2 {..}
```

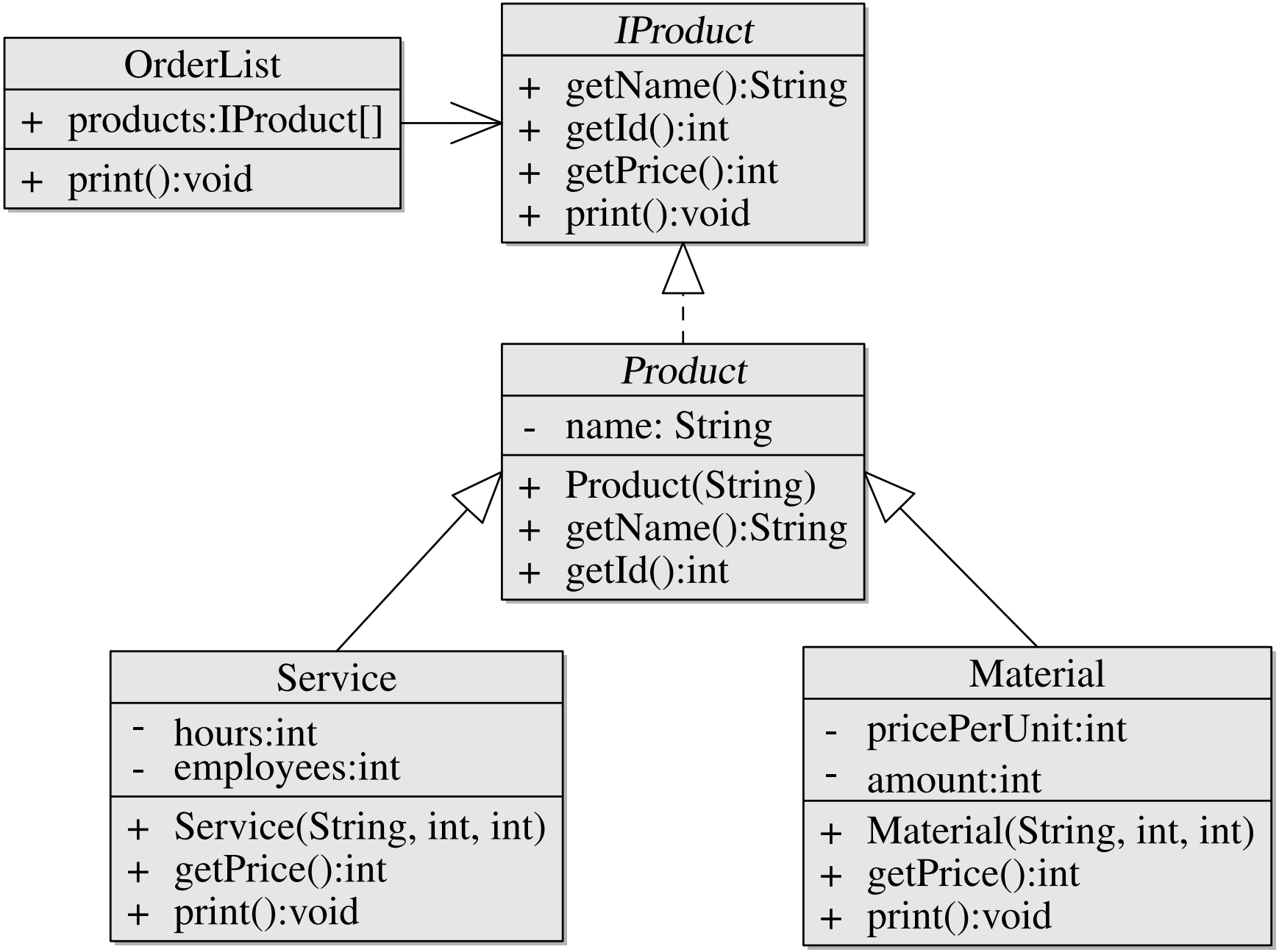
- Schnittstellen sind ein zentrales Konzept von Java, da sie keine Implementationen enthalten, sind sie stabil
- (nur) Interfaces können von Interfaces erben

Beispiel

```
public interface KannSchwimmen {
    void schwimmen();
}

public interface KannLaufen {
    void laufen();
}

public interface KannFliegen {
    void fliegen();
}

public class Ente extends Tier implements
    KannSchwimmen, KannLaufen, KannFliegen {
// hier stehen Implementierungen von
// schwimmen(), laufen(), fliegen()
}
```

Subclassing und Subtyping

- Implementierungsvererbung wird auch als Subclassing bezeichnet.
- Schnittstellenvererbung wird auch als Subtyping bezeichnet.
- Subclassing und Subtyping bewirken unterschiedliche Arten von Wiederverwendung.
  - Beim Subclassing wird der Code der Superklasse wiederverwendet: Die Member-Variablen und Methoden der Superklasse sind in der Subklasse nicht nochmals zu programmieren.
  - Beim Subtyping dagegen wird der Code wiederverwendet, der die Superklasse benutzt, also seine Klienten.
- Subclassing hat geringen Wiederverwendungswert.
- Subtyping hat hohen Wiederverwendungswert.


OrderList
+ products:IProduct[]
+ print():void
IProduct
+ getName():String
+ getId():int
+ getPrice():int
+ print():void
Product
- name: String
+ Product(String)
+ getName():String
+ getId():int
Service
- hours:int
- employees:int
+ Service(String, int, int)
+ getPrice():int
+ print():void
Material
- pricePerUnit:int
- amount:int
+ Material(String, int, int)
+ getPrice():int
+ print():void

Interface

```
public interface IProduct {
    String getName();
    int getId();
    int getPrice();
    void print();
}
```

Realisierung

```
public abstract class Product implements IProduct {

    private static int productNo = 0;
    private String name;
    private int id = productNo++;

    public Product(String name) { this.name = name; }
    public String getName() { return name; }
    public int getId() { return id; }
}
```

Einsatz von Interfaces und abstrakten Klassen

1. Ohne Verwendung abstrakter Klassen und Interfaces kann bessere Wiederverwendbarkeit und Wartbarkeit **nicht** erreicht werden.
2. Nutzer einer Klasse verwenden **immer** das Interface und niemals Implementierungen.
3. Das Interface bietet genau die Methoden an, die der Nutzer wirklich braucht.
4. Gemeinsames Verhalten und gemeinsamer Status verschiedener Implementierungen wird in einer abstrakten Klasse hinterlegt.

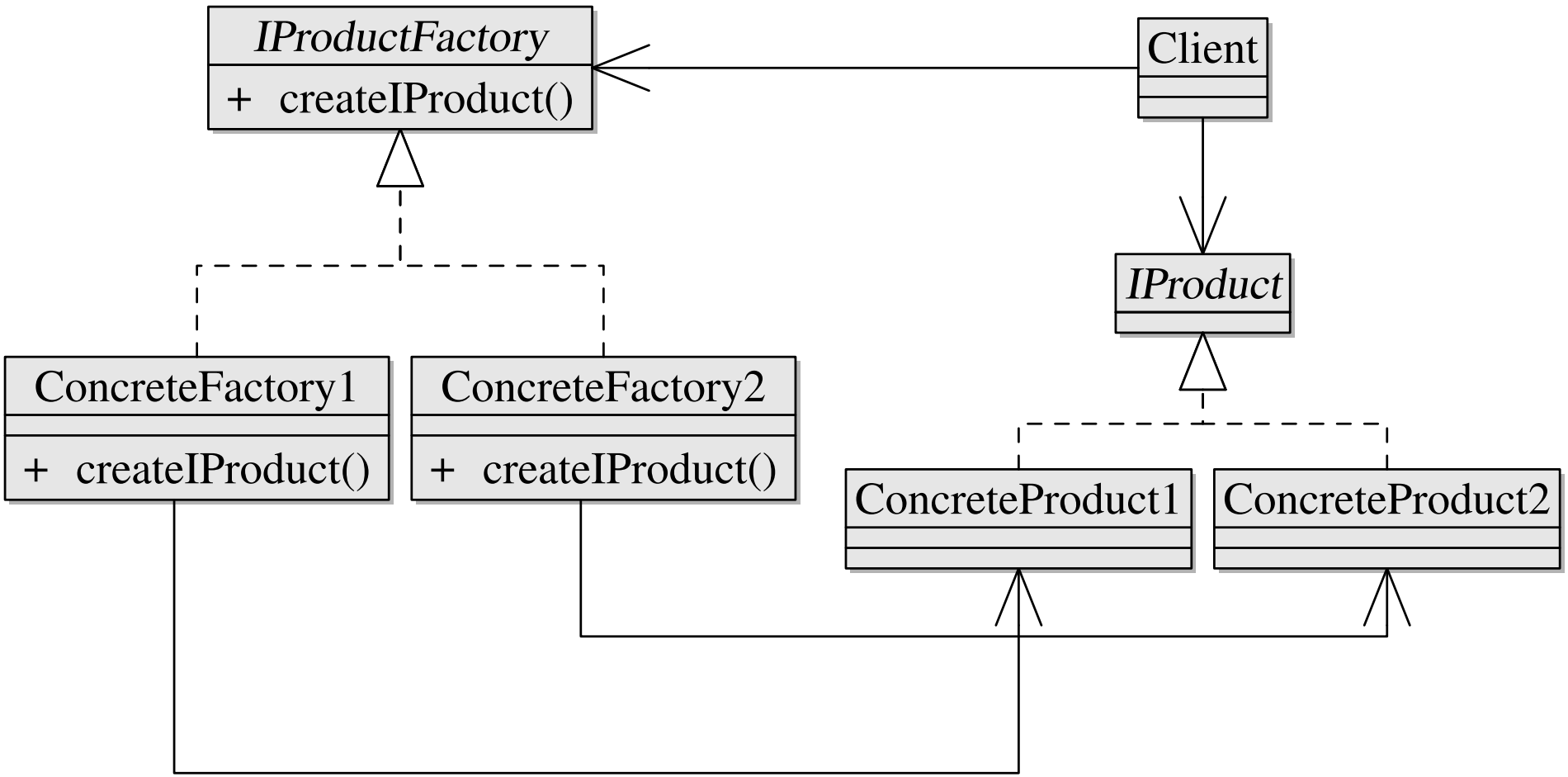
Das Problem mit new Service(...)

orderService(index) in OrderList ist böse

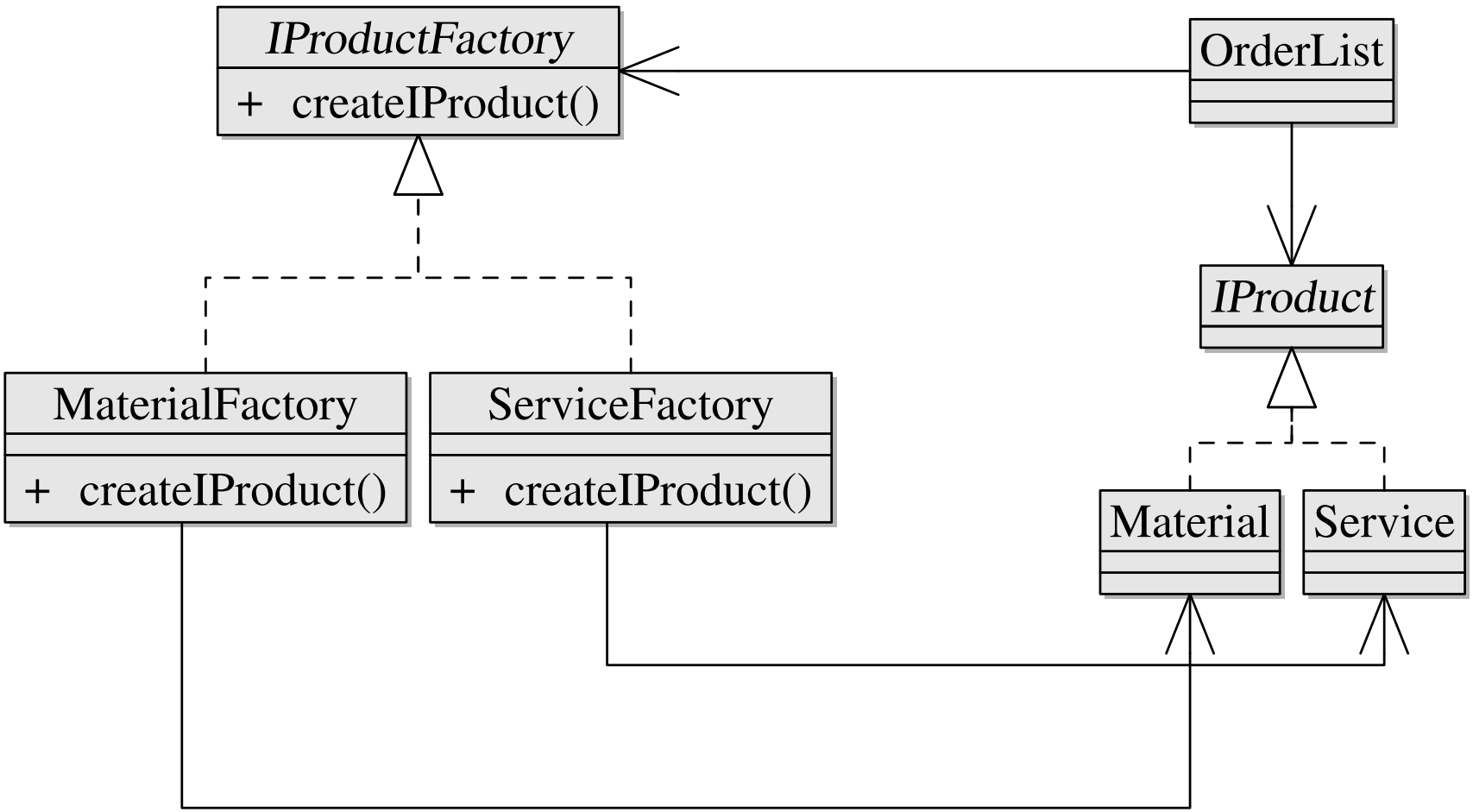
```
private void orderService(int index) {
    System.out.println("Leistungsart?");
    String name = this.scanner.nextLine();
    System.out.println("Personenzahl?");
    int employees = this.scanner.nextInt();
    System.out.println("Stunden?");
    int hours = this.scanner.nextInt();

    this.products[index] = new Service(name, employees, hours);
}
```

1. orderService kennt die konkrete Klasse Service.
2. OrderList hängt von konkreten Klassen ab.

Abstract Factory Pattern



Abstract Factory Pattern im Kontext



Bad Smell: Konstanten in Interfaces

Negativbeispiel

```
public interface BadSmellInterface {
    int BAD_CONSTANT = 0;
    int wrongNotation = 1;
    void doesSomething();
}
```

Nachteile:

1. Gemäß JLS haben Konstanten in Interfaces implizit die Sichtbarkeit `public static final`
2. Sichtbarkeit in Interfaces ist nicht änderbar
3. Wird `BadSmellInterface` in einer Klasse implementiert, geht Verweis auf Definitionsort verloren bzw. wird verschleiert
4. Vermengung durch Implementierung erschwert Erweiterungen, Refactorings und die Nachvollziehbarkeit

Lösung für Bad Smell: Konstanten in Interfaces

Korrektes Beispiel

```
public class GoodConstants {
    private GoodConstants() {}
    public static final int GOOD_CONSTANT = 0;
    static final int CORRECT_NOTATION = 1;
}
```

Empfehlungen:

1. Umsetzung über finale Konstantensammlungsklassen mit privatem Konstruktor
2. Umsetzung über `enum`-Aufzählungen
3. Notation von Konstanten erfolgt in Großbuchstaben mit Unterstrich
4. Sichtbarkeit der Konstante den Anforderungen entsprechend definieren

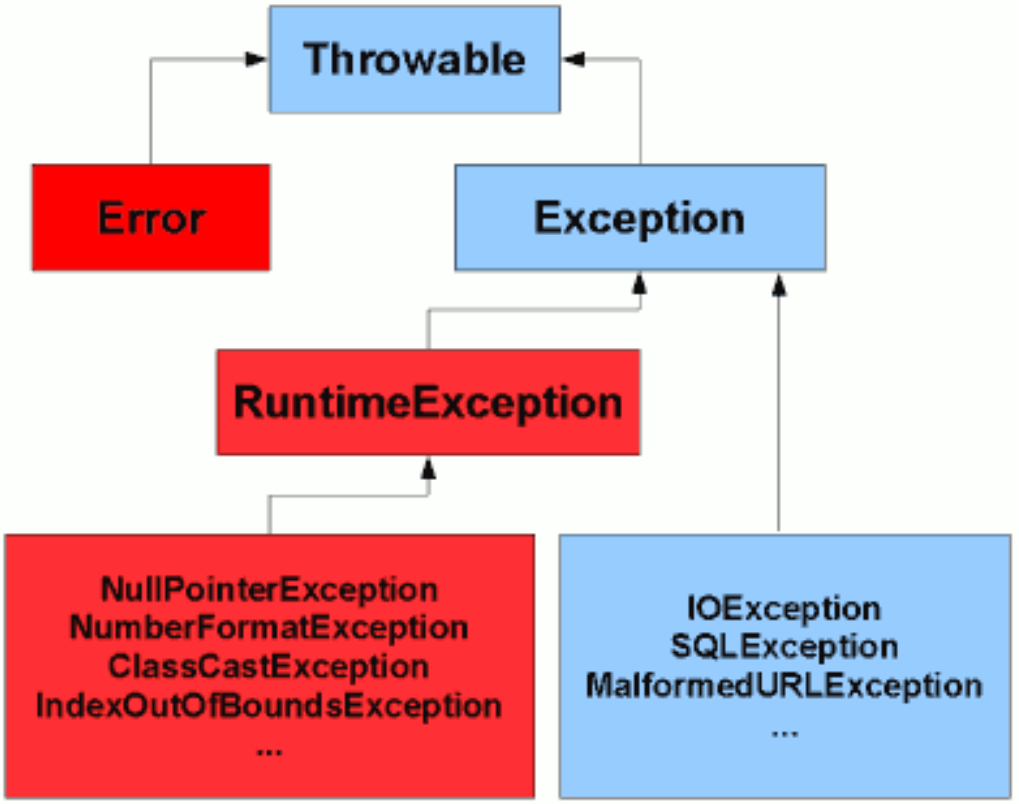
Ausnahmesituationen

Typen von Ausnahmesituationen

- Throwable - Basisklasse für Objekte, die Ausnahmesituationen signalisieren.
- Exception - Fehlersituation, auf die ein sorgfältiges Programm vorbereitet sein sollte und durch Abfangen darauf entsprechend reagieren kann.
- RuntimeException - Zum Zeitpunkt der Kompilierung nicht überwachte Ausnahmesituation (unchecked Exception), die in der JVM auftreten kann. Viele Programmierfehler, z.B. nicht initialisierte Objekte, falsche Arraygrößen u.ä. fallen in diese Kategorie.
- Error - Schwerwiegende Fehler, nach denen der weitere Programmablauf meist nicht möglich ist, z.B. Speichermangel

Java Exception-Hierarchie

- Exception Klassen sind in einer Klassenhierarchie angeordnet.
- Die Zugehörigkeit eines Exception Objekts zu einer Exceptionklasse beschreibt, welche Fehlersituation aufgetreten ist.
- Exception Objekte wissen, an welcher Stelle im dynamischen Programmablauf sie aufgetreten sind. Die printStackTrace() Methode gibt diese Information aus.



Exception-Mechanismus in Java

- Ein Laufzeitfehler oder eine vom Entwickler gewollte Bedingung löst eine Exception aus (“throwing”)
- Der auslösende Programmteil kann die Exception entweder behandeln (“catching”) oder an den Programmteil weiterreichen, der ihn aufgerufen hat.
- Der empfangende Programmteil einer weitergereichten Exception kann diese ebenfalls entweder behandeln oder weiterreichen.
- Wird eine Exception von keinem Programmteil behandelt, kommt es zum Programmabbruch mit einer Fehlermeldung und dem zugehörigen Stacktrace.

Zentrale Methoden von Exceptions

Die Basisklasse Exception

```
public class Exception extends Throwable {
    public Exception();
    public Exception(String message);
    public Exception(Throwable cause);
    public Exception(String message, Throwable cause);
 ...
    public String getMessage();
    public Throwable getCause();
 ...
    public void printStackTrace();
}
```

Exceptionklassen haben kein eigenes Verhalten (stimmt!) und Status (kommt drauf an ...).

Entstehung von Exceptions

```
public String getFirst(String[] s) throws ArrayLeerException {
    if (s == null)
        throw new ArrayLeerException("Array␣ist␣null!");
    else if (s.length == 0)
        throw new ArrayLeerException();
    else
        return s[0];
}
```

- In der getFirst() Methodendeklaration ist angegeben, dass diese Methode eine ArrayLeerException werfen kann.
- Verwender dieser Methode wären nun gezwungen diese Exception zu behandeln oder weiterzugeben.
- **Ausnahme:** Wenn die geworfene Exception vom Typ RuntimeException ist, muss sie nicht zwingend (sie darf aber!) vom Aufrufer behandelt werden.

Fangen und sofortiges Behandeln von Exceptions

```
String[] s = getRandomArray();
try {
    String first = getFirst(s);
 ...
} catch (ArrayLeerException e) {
    System.out.println("Leerer␣Array␣übergeben!");
} finally {
    s = null;
    first = null; // Compile Error!
}
```

- Der Methodenaufruf, welcher eine Exception auslösen könnte sowie der Code für den normalen Programmablauf, stehen im try{}-Block
- Der Code für den Fall, dass eine Exception auftritt, steht im catch{}-Block
- Code der unabhängig davon immer am Ende ausgeführt werden soll, steht im finally{}-Block

Sonderfälle

Mehrere catch{}-Blöcke

```
try {
  ...
} catch (EigeneCheckedException ece) {
  ...
} catch (Exception e) {
  ...
}
```

RuntimeException werfen ohne throws-Deklaration

```
public String getFirst(String[] s) {
  if (s == null) throw new NullPointerException();
  ...
}
```

Korrektes Exceptionhandling

- Je spezifischer eine Exceptionklasse, desto besser lassen sich einzelne Fehler behandeln. Hierbei gilt abzuwägen, ob man eine eigene Exception schreibt oder bestehende wiederverwendet.
- Faustregel: RuntimeExceptions für fachliche Fehler (z.B. schlechte Daten), Exceptions für technische Fehler (z.B. Zugriff auf Festplatte fehlgeschlagen)
- Exceptions dürfen nicht ignoriert werden. Leere catch-Blöcke sind schlechter Stil. Wenigstens loggen.
- Exceptionhandling ist keine leichte Angelegenheit. Es lässt sich ohne viel Nachdenken eine `IllegalArgumentException` werfen, aber am Ende weiß niemand mehr was los ist.
- Exceptions dürfen nicht für die Implementierung des regulären Kontrollflusses benutzt werden. Sie sind **Ausnahmen**!

Weitere Informationen unter https://howtodoinjava.com/best-practices/java-exception-handling-best-practices/

Validierung vs. Exceptions

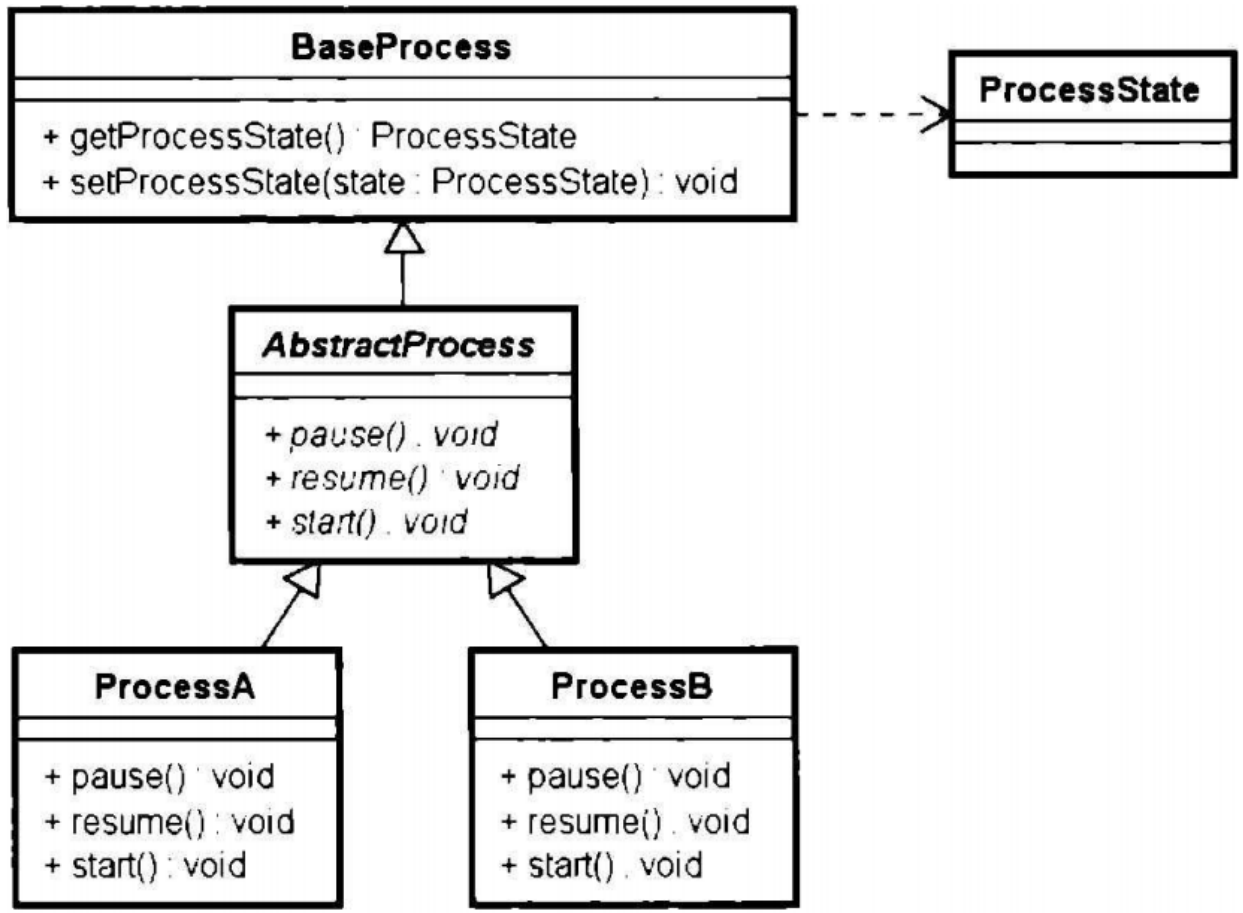
- Validierungen von Benutzereingaben sind nicht durch Exceptions zu steuern, da sie Teil des regulären Kontrollflusses sind.
- Validierungswarnungen sollten im Gegensatz zu Exceptions gesammelt werden. Das erste Auftreten einer Warnung muss nicht zwingend den Programmfluss abbrechen.
- Für Validierungen empfiehlt sich das Notification-Pattern.
- Java bietet bereits den Standard "Bean-Validation"(JSR-303) dafür an.

Weitere Informationen zum Notification-Pattern unter:

`https://martinfowler.com/articles/replaceThrowWithNotification.html`

Aufgabe

Aufgabe Basisklassen



- Tauschen Sie sich zu dem Klassenmodell aus. Sehen Sie Vor- oder Nachteile des Klassenmodells?
- Wie könnte das Klassenmodell verbessert werden?

Übungsziel

- Die folgende Aufgabe beschäftigt sich mit einem Refactoring, also mit einer Umprogrammierung einer bestehenden Anwendung unter vollständiger Beibehaltung der Funktionalität zum Zwecke besserer Programmqualität.
- Ziel dieses Refactorings ist es, dass die Klasse `OrderList` aus der gegebenen Anwendung als wiederverwendbare Komponente nur von Interfaces nicht jedoch von Implementationen abhängig ist. Es soll nach dem Ende des Refactorings möglich sein, eine dritte Implementation von Produkten hinzuzufügen, ohne dass die Klasse `OrderList` angepasst werden muss.
- Es soll am Ende der Übung eine wirkliche Erweiterung realisiert werden. Dabei darf `OrderList` nicht geändert werden.
- Gehen Sie, wie in den Aufgaben beschrieben, bei der Entwicklung inkrementell vor, d.h. schreiben Sie ein kleines Stück Programm und bringen Sie dieses zum Laufen.

Übung Interfaces und abstrakte Klassen 1/3

1. Laden Sie die Ausgangsanwendung aus gitlab (PdSE/order-list) herunter.
2. Machen Sie sich mit dem bestehenden Code vertraut.
3. Legen Sie in dem neuen Projekt die Pakete `de.nordakademie.orderlist.model`, `de.nordakademie.orderlist.model.factory`, `de.nordakademie.orderlist.model.implementation`, `de.nordakademie.orderlist.frontend` an.
4. Bewegen Sie die Klassen in das zugehörige Package und testen Sie die Anwendung.
5. Entfernen Sie die main Methode aus der Bestellliste und erzeugen eine Klasse App, die die main Methode aufnimmt.
6. Machen Sie die Klasse Product abstrakt und entfernen die Methodenrümpfe von `print` und `getPrice`. Machen Sie diese Methoden abstrakt. Testen Sie die Anwendung.

Übung Interfaces und abstrakte Klassen 2/3

7. Erzeugen Sie ein Interface IProduct, das von Product implementiert wird. Welche Methoden gehören ins Interface, welche in die abstrakte Klasse?
8. Ersetzen Sie die Referenzen auf Product in OrderList durch IProduct. Testen Sie die Anwendung.
9. Erzeugen Sie ein Interface IProductFactory mit den Methoden createProduct() und getProductType().
10. Implementieren Sie die Klassen MaterialFactory und ServiceFactory, die selbstverständlich IProductFactory implementieren. Verwenden Sie den Code aus orderMaterial und orderService aus OrderList.

Übung Interfaces und abstrakte Klassen 3/3

11. Die Klasse OrderList erhält nun eine Exemplarvariable productFactories vom Typ IProductFactory[] und eine Setter Methode für die Variable.
12. Füllen Sie die Variable productFactories mittels der Setter Methode in der main Methode von App mit einem Array aus einer MaterialFactory und einer ServiceFactory.
13. Verändern Sie den Ausgabedialog und die Auswahlbearbeitung unter Verwendung von productFactories.
14. Löschen Sie orderMaterial und orderService aus OrderList. Testen Sie Ihre Anwendung.
15. Definieren Sie eine neue Produktart Ihrer Wahl. Erzeugen Sie eine geeignete Factory Implementation. Sie sollten in der Lage sein, die Bestellliste in main mit dieser neuen Factory zu versorgen, Ihre Anwendung sollte nun mit der neuen Produktart umgehen können.