

## 15. Pakete und Übersetzungseinheiten

**ARINKO<sup>®</sup>**

- Eine Übersetzungseinheit umfasst den Quellcode, den der Java-Compiler in "einem Rutsch" (einem Übersetzungsvorgang) übersetzt.
- Normalerweise ist das eine Typdeklaration (Klasse, Interface etc.)
- Die Übersetzungseinheit ist also das .java-File
- In einer .java-Datei dürfen mehrere Typdeklarationen enthalten sein, aber nur eine darf den Modifizierer `public` enthalten.
- Beim Aufruf des Java-Compilers `javac` kann man eine oder mehrere Übersetzungseinheiten als Argumente übergeben:

```
javac Bestellung.java Fahrzeug.java Test.java
```

- Aus jeder Typdeklaration im Quellcode entsteht eine *eigene* .class-Datei.

- Der JDK-Compiler geht folgendermaßen vor:
  - wenn er beim Compilationsvorgang auf eine andere Klasse stößt, prüft er, ob es die betreffende .class-Datei bereits gibt und ob sie noch aktuell ist.
  - ansonsten sucht er nach einer .java-Datei mit dem Namen der Klasse und übersetzt sie, falls er sie findet.
- Die gesamten Orte, an denen für die Compilation nach Ressourcen gesucht werden, nennt man auch **compile time class path**
- Der Eclipse-Compiler arbeitet i.d.R. etwas anders, da er "inkrementell" implementiert ist.
- Eclipse compiliert direkt beim Speichern und ist in der Lage, auch nur neue Teile in einem File zu compilieren und es können auch Klassen mit Fehlern teilübersetzt werden.
- Eclipse benötigt somit kein JDK für die Programmentwicklung, ein JRE genügt.
- Die meisten anderen IDEs arbeiten, indem sie den JDK-Compiler aufrufen.

- Eine .java-Datei sollte nur eine Typdefinition enthalten.
- Der Name der .java-Datei sollte mit dem Namen der Typdefinition übereinstimmen.
- Trägt die enthaltene Typdeklaration den Modifizierer `public`, dann muss der Name der Übersetzungseinheit mit dem Namen des Typs übereinstimmen.

```
public class FehlerA
{
}

class FehlerB
{
}
```

Fehler.java

Error: The public type FehlerA must be defined in its own file

```
class FehlerA
{
}

class FehlerB
{
}
```

Fehler.java

compiliert ohne Probleme, Resultat FehlerA.class und FehlerB.class

- Damit die zahllosen Typdefinitionen der Java-Plattform und eigener Projekte nicht unübersichtlich "nebeneinander" zu finden sind, werden sie in Paketen organisiert.
  - Pakete fassen Typen zusammen, die ein gemeinsames Thema (Domain) haben oder die anderweitig eng zusammenarbeiten.
  - Welche Typen in welche Pakete kommen, bestimmt der Entwickler, d.h. man kann auch hier nachvollziehbare Strukturen und weniger nachvollziehbare Strukturen schaffen.
  - Pakete stellen einen eigenen Namensraum dar.
- 
- Es gibt nicht in allen Programmiersprachen ein Namensraum-Konzept. Beispiele: C oder Objective-C.
  - Lösung in Objective-C: um Typen zu unterscheiden, wird deren Modulzugehörigkeit als 2- oder 3-Lettern-Abkürzung als Typnamenpräfix geführt.
  - Bsp: NSObject – NS steht für "next step", die originale Plattform und Object ist die Basisklasse.

- Ein Paket definiert einen eigenen Geltungsbereich für die in ihm enthaltenen Typdefinitionen.
- Innerhalb des Pakets müssen die Typ-Namen eindeutig sein.
- Dafür dürfen sich Typen innerhalb eines Paketes auch direkt mit diesem Namen ansprechen.
- Der Inhalt jeder .java-Datei gehört zu genau einem Paket.
- Als erste Information darf in der .java-Datei eine Deklaration stehen, zu welchem Paket sie gehört.
- Fehlt diese Anweisung, dann gehört die Übersetzungseinheit zum "default package", dem namenlosen Standardpaket.
- Ein Paket besteht somit aus allen Übersetzungseinheiten, die den Namen des Pakets spezifiziert haben.

- Wenn ein Paket benannt wird, dann setzt sich sein Namen aus einer Folge von Bezeichnern, jeweils durch Punkte getrennt, zusammen.

```
package de.arinko.project.start;  
  
public class Run  
{  
  
    public static void main(String[] args)  
    {  
        // ...  
    }  
}
```

- Die Namensvergabe ist relativ frei, es existieren aber Konventionen:
  - nur Kleinbuchstaben, keine Zahlen
  - von der umgedrehten Firmen-URL abgeleitet, um weltweite Eindeutigkeit zu erreichen:  
http://www.arinko.de, Projekt "project", Teil-Modul "start"  
→ de.arinko.project.start
- Eigene Pakete dürfen niemals mit java. oder javax. beginnen!

- Innerhalb eines Paketes ist sozusagen "freie Sicht".
- Wenn ein Typ eines Pakets einen anderen Typ des Pakets benutzen will, kann dies ohne weitere Vorkehrungen geschehen.

```
package de.arinko.project.start;  
  
public class Run  
{  
  
    public static void main(String[] args)  
    {  
        RunHelper.doSomething();  
    }  
}
```

Klasse ist im selben Paket  
und kann direkt  
angesprochen werden

```
package de.arinko.project.start;  
  
class RunHelper  
{ ... }
```



- Soll eine Typdeklaration außerhalb ihres Paketes verwendet werden dürfen, muss sie den Modifizierer `public` tragen.

```
package de.arinko.project.gui;  
  
public class GUI  
{  
    public void open()  
    {  
        //...  
    }  
}
```

Klasse kann außerhalb  
benützt werden

- Will man einen (public) Typ eines anderen Pakets benützen, so hat man 3 Möglichkeiten:
  1. das Paket, das die Typdeklaration beinhaltet "importieren"
  2. die Typdeklaration "importieren"
  3. den Typ mit seinem "voll qualifizierten" Namen ansprechen

- Ist ein Paket pauschal importiert, kann man alle public Typen direkt mit ihrem Namen ansprechen.

```
package de.arinko.project.start;  
  
import de.arinko.project.gui.*;  
  
public class Run  
{  
    public static void main(String[] args)  
    {  
        GUI gui = new GUI();  
        gui.open();  
    }  
}
```

alle public Typen namentlich  
bekannt machen

- Frage: Vor- und Nachteile dieser Methode?

- Wird ein Typ gezielt importiert, dann darf auch nur dieser in der anderen Typdeklaration vorkommen. Andere Typen aus seinem Paket sind unbekannt.

```
package de.arinko.project.start;  
  
import de.arinko.project.gui.GUI;  
  
public class Run  
{  
    public static void main(String[] args)  
    {  
        GUI gui = new GUI();  
        gui.open();  
    }  
}
```

nur diesen Typ bekannt  
machen

- Frage: Vor- und Nachteile dieser Methode?

- Hier wird auf den Import verzichtet und der Typ mit seinem vollen Namen direkt angesprochen.
- Der Paketname gehört zum Klassennamen!

```
package de.arinko.project.start;  
  
public class Run  
{  
    public static void main(String[] args)  
    {  
        de.arinko.project.gui.GUI gui = new de.arinko.project.gui.GUI();  
        gui.open();  
    }  
}
```

- Frage: Vor- und Nachteile dieser Methode?

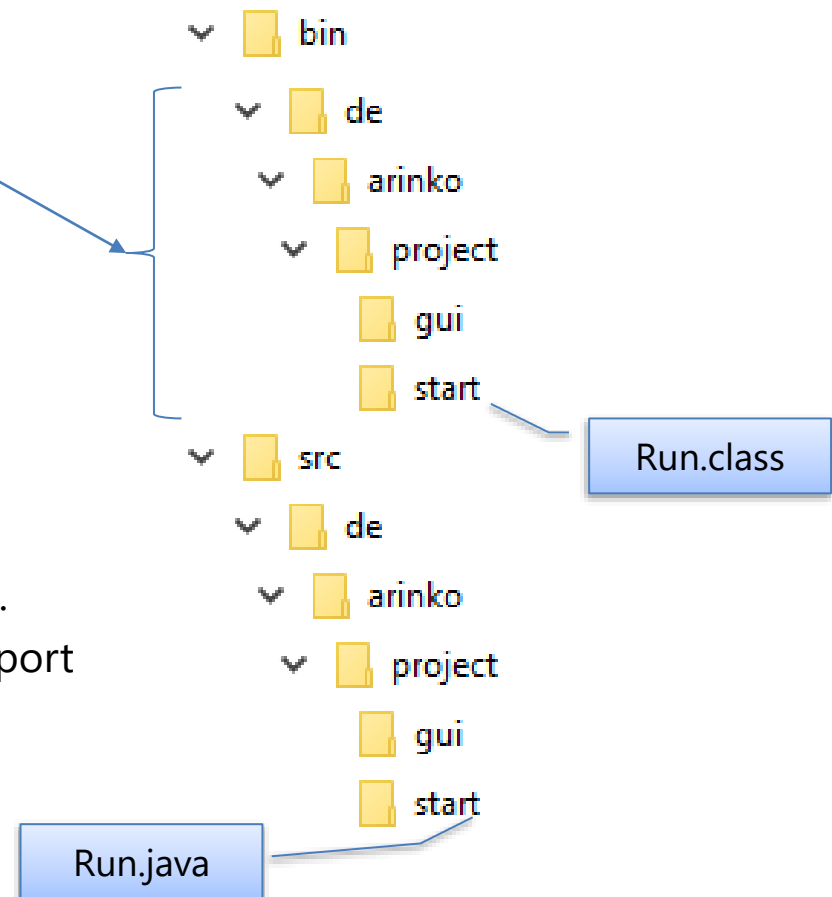
- `java.lang` ist ein ganz besonderes Paket, denn es ist immer "implizit" importiert.
- Alle Typdefinitionen aus diesem Systempaket sind somit immer sichtbar und müssen nie importiert werden.
- Einige Typen aus diesem Paket: `String`, `Object`, `Thread`, `Exception`, `System`
- Auch wenn der Import nicht notwendig ist, so ist dennoch der vollqualifizierte Name dieser Klassen immer mit Paketname, sollte man ihn irgendwo angeben müssen:

```
java.lang.String  
java.lang.Object  
java.lang.System  
...
```

- Paketnamen und der Ablageort der Sourcecode- und Bytecode-Dateien sind unmittelbar verbunden.
- Der Paketname gibt eine Art virtuellen Pfad vor, der sich als Teilbaum einer Dateisystemhierarchie wiederfinden muss.

```
package de.arinko.project.start;  
  
public class Run  
{  
    ...  
}
```

- Der Paketname wird vom Compiler in die .class-Datei hineincompiliert.
- Somit ist ein Ändern der Paketzugehörigkeit nur mit Besitz der Sourcecode-Datei möglich.
- Rekursive Imports sind nicht möglich, der Import importiert immer nur die Klassen aus einem Paket gleichzeitig.



- Mit den JDK-Tools arbeitet man immer vom Root-Level des "default packages" aus.

```
C:\Java\Projectroot> javac de\arinko\project\start\Run.java
```

```
C:\Java\Projectroot> java de.arinko.project.start.Run
```

- Hier ist davon ausgegangen, dass Sourcen und Bytecode im selben Ordner liegen, nicht wie im umseitigen Beispiel aus einer IDE mit Trennung von src und bin.
- Die Trennung nach Sourcen (oft "src") und Bytecode (oft "bin") ist etwas, das sich oft in Entwicklungsumgebungen findet. Der Java-Compiler platziert per Default alles in einem Verzeichnis, es sei denn man benutzt Befehlsargumente:

```
C:\Java\Projectroot> javac -help
```

```
...
```

|                    |   |
|--------------------|---|
| -d <directory>     | Specify where to place generated class files    |
| -sourcepath <path> | Specify where to find source files              |
| -classpath <path>  | Specify where to find user class files...       |
| -encoding <enc>    | Specify character encoding used by source files |

```
...
```

- Seit Java 5 sind auch sog. **statische Imports** erlaubt.
- Ein statischer Import importiert statische Elemente eines anderen Typs, die dann im nutzenden Typ so verwendet werden können, als ob sie hier definiert wären.
- Man verwendet das gerne für oft wiederkehrende statische Methoden (z.B. mathematische Funktionen, oder für Unittests)

```
import static java.lang.Math.*;

public class Berechnung
{
    public static void main(String[] args)
    {
        // ohne statischen Import
        double a = Math.sin(50) * Math.cos(20);

        // mit statischem Import
        double b = sin(50) * cos(20);
    }
}
```

- Statische Imports gehen auch qualifiziert: `import static java.lang.Math.sin;`



- Kann ein Name über mehrere Imports aufgelöst werden, liegt eine Mehrdeutigkeit ("ambiguity") vor.
- Qualifizierte (nicht-statische) Imports erzeugen sofort eine Fehlermeldung.
- Pauschale (nicht-statische) Wildcard-Imports provozieren erst bei der Benutzung Fehler, wenn Typen verwendet werden, die aus mehreren Paketen importiert werden könnten.
- Statische Imports erzeugen stets erst bei der Verwendung einen Fehler.
- Erzeugt das einen Fehler?

```
import static de.arinko.logging.Logger.log;  
import static java.lang.Math.log;
```



## 16. Annotations

**ARINKO<sup>®</sup>**

- Mit Annotations kann Metainformation in einem Java-Programm platziert werden.
- Metainformation sind Daten, die Code mit zusätzlichen Eigenschaften anreichern.
- Beispiele:
  - Persistenzinformation für Daten
  - Hinweise für Generatoren
  - Hinweise für den Compiler (z.B. das man Warnungen zur Kenntnis genommen hat)
  - Hinweise für Default-Verhalten in Frameworks
- Mit dem Java Specification Request (JSR) 175 "A Program Annotation Facility for the Java Programming Language" wurden Annotations erstmals definiert und dann in Java 5 eingeführt.
- Annotations sind typisierte Eigenschaften eines Typs, einer Methode, eines Attributs, Konstruktors, Parameters oder gar einer lokalen Variable.
- Annotations können selbst definiert und ausgewertet werden – viele moderne Frameworks beruhen auf diesem Prinzip.
- Die Java-Plattform liefert einige (wenige) Annotations bereits mit:
  - Standard-Annotations (z.B. für Compiler-Warnungen)
  - Standard-Meta-Annotations (Annotations für den Gebrauch bei der Deklaration von anderen Annotations)

- Annotations werden ähnlich wie Interfaces deklariert, die Unterschiede sind aber recht augenfällig:

```
@interface AnnotationName
{
    type identifier() [default defaultValue];
}
```

- Beispiel einer Standard-Annotation, deklariert mit Standard-Meta-Annotations:

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings
{
    String[] value();
}
```

- Die Spezifikation spricht von 3 Akteuren, die Annotations zu unterschiedlichen Zeitpunkten auswerten können: "General Tools", "Specific Tools", "Introspectors"
- Hierfür kann man mit `@Retention` an Annotations den Grundstein legen.
- `@Retention(RetentionPolicy.SOURCE)`  
Die Annotation wird nur im Sourcecode (z.B. vom Compiler) ausgewertet, danach ist die Information verloren.
- `@Retention(RetentionPolicy.CLASS)`  
Die Annotation wird in den Bytecode hineinkompiliert, die virtuelle Maschine verarbeitet sie aber nicht.
- `@Retention(RetentionPolicy.RUNTIME)`  
Die Annotation wird in den Bytecode hineinkompiliert und zur Laufzeit in die virtuelle Maschine geladen.
- Code-Generatoren arbeiten gerne mit SOURCE oder CLASS.
- Frameworks, die zur Laufzeit basierend auf den geladenen Klassen dynamisch agieren wollen, arbeiten immer mit RUNTIME.  
Beispiele: Java EE (z.B. EJBs, JSF), Spring, JPA-Frameworks etc.

- Durch `@Target` kann man die Orte vorgeben, an denen eine Annotation anwendbar ist:

|                 |  |
|-----------------|--|
| TYPE            | Vor Typdeklarationen (Klassenkopf)         |
| FIELD           | Vor Attributsdeklarationen                 |
| METHOD          | Vor Methodendeklarationen                  |
| PARAMETER       | Vor Parameterdeklarationen                 |
| CONSTRUCTOR     | Vor Konstruktordeklarationen               |
| LOCAL_VARIABLE  | Vor der Deklaration einer lokalen Variable |
| ANNOTATION_TYPE | Vor Deklarationen von Annotations          |
| PACKAGE         | Vor der Paketsdeklaration (selten)         |
| TYPE_PARAMETER  | Vor generischen Typparameterdeklarationen  |
| TYPE_USE        | Vor Typverwendungen (siehe JLS 4.11)       |

- Annotations werden ähnlich zu Interfaces deklariert, aber...
- keine Ableitungen
- Methoden dürfen keine Parameter haben (es handelt sich ja eher um Eigenschaften als um Methoden)
- Generics sind nicht erlaubt
- Rückgabewerte (die eher Typen für die durch Methoden deklarierten Eigenschaften entsprechen) sind:

elementare Datentypen, String, Class, Enums, andere Annotations und Arrays von allen genannten

- keine Exceptions

Annotations existieren in 3 "Geschmacksrichtungen", die sich darin unterscheiden, wie einfach sie sich hinschreiben lassen:

- normale Annotations (Eigenschaften werden aufgezählt und benannt)  
`@Column(name="DBCX5V3", table="DSTTBL", columnDefinition="INTEGER")`
- Single-Member Annotations (nur aus einem (nicht-default) Wert namens "value" bestehende Annotation, dessen Name weggelassen werden darf)  
`@SuppressWarnings("unused")`

kurz für

`@SuppressWarnings(value="unused")`

- Marker Annotations (Annotation, die entweder über lauter default Werte verfügt und deren Angaben somit entfallen dürfen oder über gar keine)  
`@Deprecated`

kurz für

`@Deprecated()`



- In der Java SE gibt es ein paar wenige Standard-Annotations für den direkten Gebrauch in Java-Source-Code.
- `@Deprecated`  
zeigt an, dass eine Methode oder ein Typ veraltet ist und generiert eine Warnung durch den Compiler, wenn das gekennzeichnete Element verwendet wird
- `@Override`  
eine Methode überschreibt eine geerbte Methode (das ist einerseits Dokumentation für den Leser, der Compiler überwacht andererseits, ob tatsächlich eine Überschreibung stattfindet)
- `@SuppressWarnings`  
veranlasst den Compiler, die angegebene Warnung zu unterdrücken, weil der Programmierer sie zur Kenntnis genommen hat
- Diese Annotation sind in `java.lang` definiert und können jederzeit ohne Import verwendet werden.



## 17. Tests

**ARINKO<sup>®</sup>**

- *Testen ist die Ausführung eines Programms mit dem Ziel Fehler zu entdecken.*  
(Meyers, 1979)
- *Testen ist die Vorführung eines Programms oder Systems mit dem Ziel zu zeigen, dass es tut, was es tun sollte.*  
(Hetzl, 1984)
- Ausführung eines Programms.
- Ergebnisse für korrekte Ausführung ist bekannt.
- Ist- und Sollergebnis werden verglichen  
Sind Ist- und Sollergebnis verschieden, liegt ein Fehler vor.

## Vorteile von Softwaretests:

- Natürliches Prüfverfahren
- Reproduzierbar – objektiv
  - Test lässt sich beliebig oft wiederholen.
- Ziel-, Entwicklungsumgebung wird mitgeprüft.
- Systemverhalten wird sichtbar.

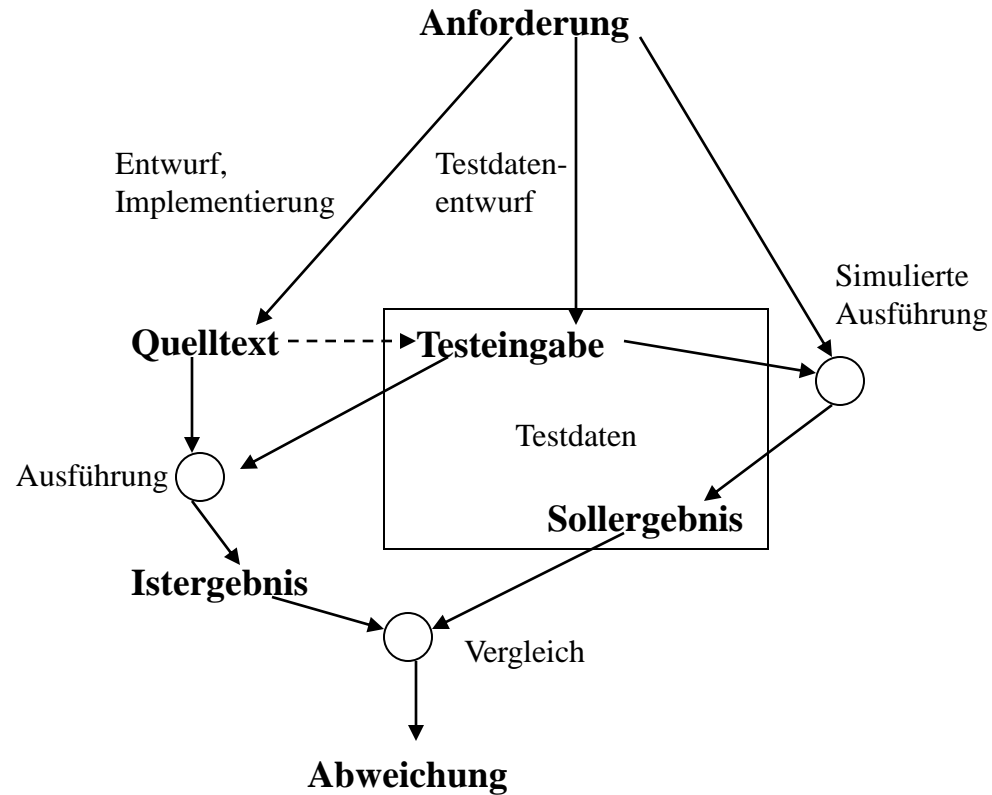
## Nachteile:

- Ergebnisse werden überschätzt!
  - Korrektheitsaussage ist nicht möglich!
- Nicht alle Programmeigenschaften können geprüft werden.
- Nicht alle Anwendungssituationen können nachgebildet werden.
- Test zeigt Fehlerursache nicht.

- Erfolgreiche Tests
  - Testen dient dem Finden von Fehlern
  - Ein Test ist erfolgreich wenn ein Fehler gefunden wird.
- Prüfling
  - Software die getestet wird.
- Regressionstest
  - Wiederholte Ausführung eines Testfalls auf einem geänderten Prüfling.
- Testgeschirr, Testumgebung
  - Umfasst den Prüfling und alle Hilfsmittel für einen Test.


#### Test-Arten:

- Laufversuch – „Code & Fix“-Ansatz
- Wegwerftest – Spontane Eingabe von Testdaten. Fehler werden unter Umständen erkannt. Testdaten sind nicht aufgezeichnet, Test ist nicht reproduzierbar.
- Systematischer Test – Testumgebung, Eingabedaten, Solldaten, Istdaten und Testablauf sind dokumentiert.



- Statische Testverfahren
  - Programm wird nicht ausgeführt
  - Syntaktische Prüfung durch den Übersetzer
  - Reviews
  
- Dynamische Testverfahren
  - Programm wird ausgeführt

- Methoden einer Klasse werden als Black-Box behandelt.
  - Implementierung ist nicht sichtbar.
  - Spezifikation der Schnittstelle bestimmt Testdaten.
  - Menge der Testdaten kann durch Äquivalenzklassen minimiert werden.

```
/**
 * Berechnet die Fakultät einer gegebenen Zahl.
 *
 * @param zahl
 *         Zahl deren Fakultät berechnet werden soll.
 * @return long Fakultät von zahl.
 * @throws CalculateException
 *         bei negativem Parameter.
 */
public static long fakultät(int zahl) throws CalculateException
{
    
}
```

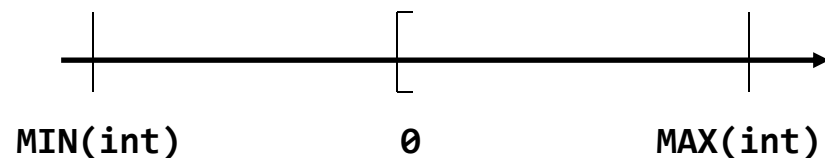


- Äquivalenzklasse enthält Elemente aus der Eingabemenge die den gleichen Fehler (Ausgabe) erzeugen.
  - Aus jeder Äquivalenzklasse wird ein Element ausgewählt.
  - Sinnvoll ist die Wahl von Elementen am Rand der Äquivalenzklasse

```
/**  
 * Berechnet die Fakultät einer gegebenen Zahl.  
 *  
 * @param zahl  
 *       Zahl deren Fakultät berechnet werden soll.  
 * @return long Fakultät von zahl.  
 * @throws CalculateException  
 *       bei negativem Parameter.  
 */
```

Eingabemenge: MIN(int)...MAX(int)

- Teilmenge 1: MIN(int)...-1
- Teilmenge 2: 0...MAX(int)



- Implementierung der zu testenden Methode ist bekannt.
  - Implementierung bestimmt Testdaten.

```
/**
 * Berechnet die Fakultät einer gegebenen Zahl.
 *
 * @param zahl
 *         Zahl deren Fakultät berechnet werden soll.
 * @return long Fakultät von zahl.
 * @throws CalculateException
 *         bei negativem Parameter.
 */
public static long fakultät(int zahl) throws CalculateException
{
    if (zahl < 0)
        throw new CalculateException();
    if (zahl <= 0)
        return 1;
    return zahl * fakultät(zahl - 1);
}
```

- Anweisungsüberdeckung
  - Jede Anweisung einer Methode wird einmal ausgeführt (Achtung: if-else gilt als eine Anweisung)
  - ca. 18% aller Fehler können gefunden werden.
- Zweigüberdeckung
  - Jeder Zweig einer Methode wird ausgeführt.
  - ca. 60% aller Fehler können gefunden werden.
- Bedingungsüberdeckung
  - Bedingungen für Zweige werden überdeckt.
- Pfadüberdeckung
  - Jeder Pfad durch eine Methode wird ausgeführt.
  - Kombinatorische Explosion bei Schleifen!
- „Boundary Interior“-Pfadtest
  - Schleifen werden
  - 0 mal,
  - 1 mal,
  - 2 mal und
  - typisch oft ausgeführt.

- JUnit ist ein (Open-Source) Framework zum Unit-Testen von Java-Programmen.
- JUnit 5 setzt Java 8 voraus, da es mit Annotations & gelegentlich funktional arbeitet.
- Analog zur Idee der Testverfahren (Soll-/Istwerte-Vergleich) wird Testcode geschrieben, der Code ausführt und z.B. die Rückgabewerte mit Sollwerten vergleicht.

```
import static org.junit.jupiter.api.Assertions.assertEquals;  
import org.junit.jupiter.api.Test;
```

wird gerne statisch importiert, damit die vielen assert... Methoden nicht qualifiziert werden müssen

```
public class FakultätTest  
{  
    @Test  
    public void testZero()  
    {  
        long result = Fakultät.fakultät(0);  
        assertEquals(1, result);  
    }  
}
```

Testklassen sind normale Java-Klassen, die allerdings über einen parameterlosen Konstruktor instanzierbar sein müssen

Methode muss parameterlos und void sein, die Annotation @Test tragen, der Name ist beliebig

|                    |   |
|--------------------|---|
| @Test              | Eine Testmethode  |
| @BeforeEach        | Die Methode wird vor jedem Test ausgeführt  |
| @AfterEach         | Die Methode wird nach jedem Test ausgeführt   |
| @BeforeAll         | Die Methode wird nur einmal vor allen Tests ausgeführt  |
| @AfterAll          | Die Methode wird nur einmal nach allen Tests ausgeführt   |
| @Disabled          | Der Test wird ignoriert/zeitweilig abgeschaltet   |
| @Timeout           | Setzt eine maximale Bearbeitungsdauer fest, die, wenn überschritten, einen Fehler zur Folge hat |
| @ParameterizedTest | Definiert einen parametrisierbaren Test. Die Datenquellen für den Test sind recht vielfältig.   |

|   |   |
|---|---|
| <code>assertEquals(long soll, long ist)</code>                                | <code>soll == ist?</code>   |
| <code>assertEquals(Object o1, Object o2)</code>                               | Vergleicht über <code>equals()</code> -Methode                      |
| <code>assertEquals(typ soll, typ ist,<br/>                  typ delta)</code> | <code>double, float</code><br><code> soll - ist  &lt;= delta</code> |
| <code>assertArrayEquals(typ soll, typ ist)</code>                             | <code>Object[], byte[], short[], char[],<br/>int[], long[]</code>   |
| <code>assertFalse(boolean condition)</code>                                   | <code>condition == false?</code>                                    |
| <code>assertTrue(boolean condition)</code>                                    | <code>condition == true?</code>                                     |
| <code>assertNotNull(Object object)</code>                                     | <code>object != null?</code>  |
| <code>assertNull(Object object)</code>  | <code>object == null?</code>  |
| <code>assertSame(Object o1, Object o2)</code>                                 | <code>o1 == o2?</code>  |
| <code>assertNotSame(...)</code>   | <code>o1 != o2?</code>  |
| <code>fail()</code>   | immer Fehler  |

- Tests können zu Suiten zusammengefasst werden (Gruppierung von Tests, damit diese gemeinsam ausgeführt werden können)
- Parametrisierte Tests (Auslagern von Testdaten). Beispiel:

```
@ParameterizedTest
@CsvSource({
    "0,1",
    "1,1",
    "2,2",
    "3,6",
    "4,24",
    "5,120"})
public void test(Long given, Long expected)
{
    long result = Fakultät.fakultät(given);
    assertEquals(expected, result);
}
```

- Vielfältige Datenmöglichkeiten: @NullSource, @EmptySource, @ValueSource, @MethodSource, @CsvSource, @CsvFileSource [Testdaten in Dateien]



## 18. Auslieferung

**ARINKO<sup>®</sup>**



- Wenn die Applikation fertig entwickelt ist, in Module (besser: Pakete) aufgeteilt und gut getestet ist, kann man sie ausliefern.

#### Einschub Realität:

- normalerweise wird jetzt die Applikation für übergreifende Integrationstest in eine andere "Stage" geschoben
- typischerweise trifft man folgende "Stages" an, die Benennungen können variieren:
  - Unit Test (Development)
  - Integration Test (Zusammenarbeit der Programmteile)
  - System Test (das gesamte System inkl. externer Ressourcen)
  - Acceptance Test (Test durch Fachabteilung/Enduser)
- Den Vorgang, die Ressourcen definiert an eine bestimmte Stelle für weitere Ausführung zu bringen, nennt man neudeutsch "Deployment"

- Statt nun den Bytecode Datei für Datei einzeln zu kopieren, packt man zusammengehörige Pakete und Ressourcen in ein sog. **JAR-File** (Java **A**rchive).
- JAR-Files beinhalten die Dateien unkomprimiert oder wahlweise komprimiert als ZIP.
- In JAR-Dateien kommt alles, was für die Ausführung eines Java-Programmes wichtig ist, also der Bytecode in Paketstruktur (d.h. Unterverzeichnisse) und andere Ressourcen, z.B. Bilddateien, Sounddateien, Konfigurationsdateien etc.
- JAR-Dateien kann man mit dem JDK-Tool `jar` oder üblicherweise mit Funktionen der IDE erstellen.
- Beispiel die Struktur der JAR-Datei für das Bestellsystem:

```
META-INF/MANIFEST.MF
de/arinko/wawi/ware/Ware.class
de/arinko/wawi/bestellung/Bestellung.class
de/arinko/wawi/bestellung/Bestellposition.class
de/arinko/wawi/kunde/Kunde.class
de/arinko/wawi/kunde/Kundenrabatt.class
de/arinko/wawi/test/WarenbestellungTest.class
```

- Das JDK-Tool jar ist an das Linux-Tool tar angelehnt (nur dass JAR normalerweise auch komprimiert):

```
Verwendung: jar {ctxui}[vfmn0PMe] [jar-file] [manifest-file] [entry-  
point] [-C dir] Dateien...
```

Optionen:

- c Neues Archiv erstellen
- t Inhaltsverzeichnis für Archiv anzeigen
- x Benannte (oder alle) Dateien aus dem Archiv extrahieren
- u Vorhandenes Archiv aktualisieren
- v Ausgabe im Verbose-Modus aus Standard-Ausgabe generieren
- f Dateinamen für Archiv angeben
- m Manifestinformationen aus angegebener Manifestdatei einschließen
- e Anwendungseinstiegspunkt für Standalone-Anwendung angeben in einer ausführbaren JAR-Datei gebündelt
- 0 Nur speichern; keine ZIP-Komprimierung verwenden...

...uvm...

- Die virtuelle Maschine kann Bytecode aus Verzeichnissen oder aus JAR-Dateien abrufen.
- Im vorherigen Beispiel der Start mit Hilfe der JAR-Datei für die Warenbestellung:

```
C:> java -classpath wawi.jar de.arinko.wawi.test.WarenbestellungTest
```

- Der sog. **Runtime Classpath** kann sich aus mehreren JAR-Dateien und/oder Verzeichnissen speisen.
- Möchte man die länglichen Namen für die Startklassen nicht eintippen, darf in der Manifest-Datei META-INF\MANIFEST.MF eine Startklasse mitgegeben werden:

```
Manifest-Version: 1.0
```

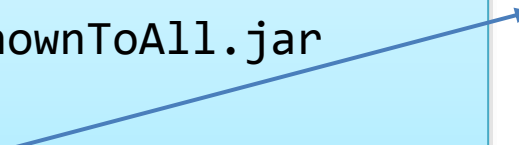
```
Main-Class: de.arinko.wawi.test.WarenbestellungTest
```

- Aufruf dann

```
C:> java -jar wawi.jar
```

- In der Java Enterprise Edition (Java EE) wird die Idee der JAR-Dateien für Teilmodule weitergeführt.
- Dort existieren beispielsweise folgende:
  - .war – Web Archive
  - .jar – für EJB-Module oder Hilfsmodule (Utility-JARs)
  - .rar – für Ressource Adapter
  - .ear – (Enterprise Archive) für die gesamte Zusammenstellung o.g. Teilmodule:

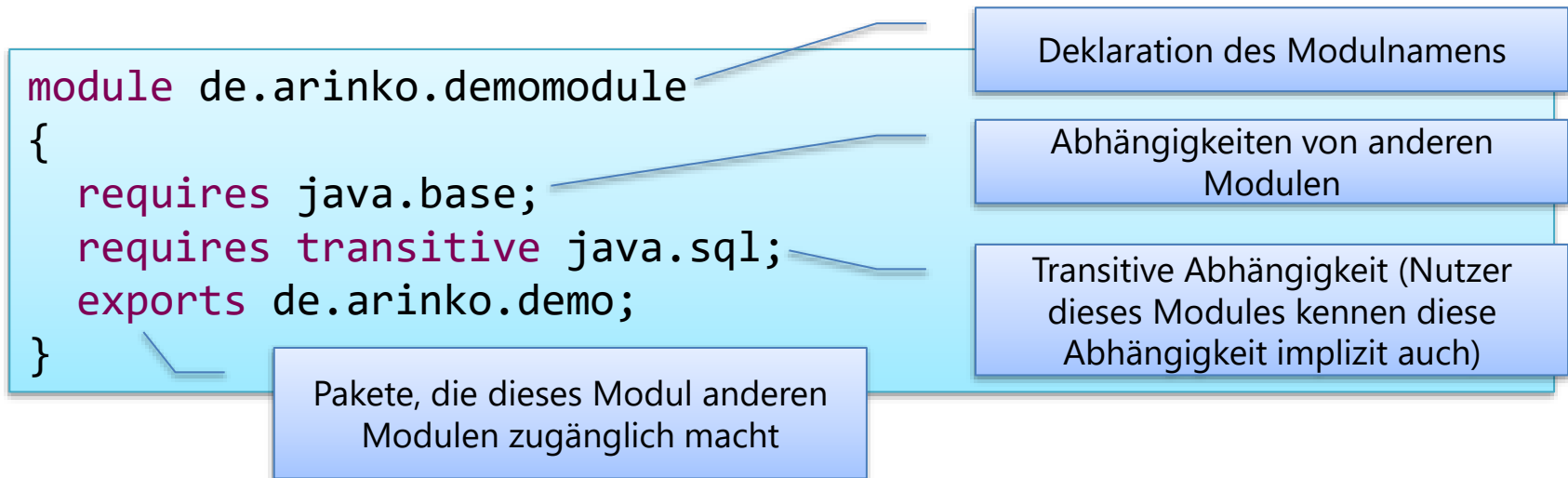
```
META-INF/  
    MANIFEST.MF  
    application.xml  
lib/  
    libsKnownToAll.jar  
webModule.war  
ejbModule.jar  
otherEjbModule.jar
```



```
META-INF/  
    MANIFEST.MF  
WEB-INF/  
    web.xml  
    lib/  
        libsForModule.jar  
    classes/  
        <Bytecode>  
  
<web resources>
```

- Mit Java 9 wurde ein lang erwartetes Modulkonzept eingeführt.
- Probleme in der Vergangenheit waren:
  - Unklare Abhängigkeiten von JAR-Dateien.
  - Abhängigkeiten mussten immer extern gemanaged werden.
  - Modernere Architekturen, die aus vielen kleinteiligen „Modulen“ bestehen sollten, hatten eine sehr unübersichtliche Abhängigkeitsstruktur („JAR-Hell“).
  - Wenn ein JAR im Klassenpfad ist, dann sind alle Pakete darin gleich sichtbar. Es gibt keine Hilfspakete oder Implementationspakete.
- Das mit Java 9 unter dem Codenamen „Jigsaw“ eingeführte Modulkonzept nimmt sich der Probleme an:
  - Aufteilung („zersägen“ - ... „jigsaw“) der Java-Laufzeitumgebung in Module, so dass auch Systeme mit minimalerem Footprint gebaut werden können
  - Module können ihre Abhängigkeiten von anderen Modulen erklären („requires“)
  - Module können den nutzbaren Inhalt erklären („exports“)
- Module werden durch eine Modul-Beschreibung („module descriptor“) definiert und strukturell konfiguriert.

- Die Modulbeschreibung/der Modulkriptor ist eine Datei `module-info.java` (bzw. deren kompilierte Form `module-info.class`).
- Deren Inhalt an einem Beispiel:



- Achtung: exportiert werden Pakete, gelesen werden Module. Modulnamen und Paketnamen sind aber per Konvention in ähnlicher Schreibweise.

- Module können über komplexere Beschreibungen verfügen. Hier nicht vollständig:
  - `requires static`: Abhängigkeit besteht nur „hart“ zur Compile-Time. Zur Runtime ist diese optional.
  - `exports to`: Qualifizierter Export in benannte abhängige Module
  - `uses`: Deklariert die Benutzung eines Services.
  - `provides`: Deklariert die Bereitstellung eines Services.
  - `opens`: Definiert Laufzeitabhängigkeit inkl. Möglichkeiten für Reflection.
- Durch die Einführung von Modulen gibt es nun 2 Pfadarten:
  - `Class-Path`: extern gemanagte Abhängigkeiten
  - `Modulepath`: deklarativ gemanagte Abhängigkeiten
- Neues Kommandozeilen-Tool `jlink`, das „self-contained“ Applikationen auf Modulbasis generieren kann.
- Multi-Release-JAR-Files, in denen verschiedene Versionen von Klassen für unterschiedliche JDK-Level platziert werden können.



