4.1. Konventionen in einem Java-Programm

Package-Namen komplett klein: **lowercase**, sonst auch camelCase .
Bei **Variablen** ist besser **camelCase**, anstelle von snake_case.
Package-Pfade: Beispiel: *com.mycompany.appname.foo*

4.2. Reservierte Begriffe (Keywords)

Offen: assert, synchronized, goto, instanceof, transient, strictfp, volatile, const, native, _

4.3. Packages, Import und Nutzung

Erstes Statement einer in der Java-file, sonst „unbenanntes Package“. (default, != name)
→Klassen darin nicht importierbar, weil sie keinem bekannten Package zuordenbar sind.
Beispiel java-file: ***package foo.bar*** , optional ***import bei.spiel.****, dann ***public class foo{}***

FQCN = fully qualified class name, sowas wie **com.foo.bar.**TestClass (Package+Klasse)
Import nutzen, Redundanz zu vermeiden. Vermeide Wildcard (*) →Nachverfolgbarkeit.
Außerdem werden bei einer Wildcard nur Klassen, keine sub-Packages importiert ~~(p.**)~~

Werden zwei gleichnamige Klassen aus unterschiedlichen Packages benötigt, kann man eine importieren und die andere im Vollpfad angeben, optimal ist aber bei beiden FQCN.
Wird eine Klasse viel häufiger als die andere benutzt, importiert man am besten nur die.
Ist eine Klasse im selben Package wie die nutzende, wird die externe mit FQCN genutzt.

4.4. Struktur einer Java-Klasse

Neben bekannten Dingen wie static und instance fields bzw. methods und Konstruktor:
static-Initializer → static-Block, der bei erstem Klassenzugriffs/-laden ausgeführt wird.
→ Beispiel: Laden einer Datenbank, die zur Verwendung der Klasse gebraucht wird.
instance-Initializer → „loser“ Block, der **vor** Erstellung neuer Instanzen ausgeführt wird.
→ Beispiel: Ändern statischer Felder/Variablen bei Erstellung neuer Objekte (C.count++)

/** * <h3>Headline als HTML-Tag</h3> , * @see „See Also“ Verweis auf andere Klassen

4.4.4 Beziehung zwischen java- und class-Dateien

Pro Java-Datei nur eine public class auf „top level“, inner / nested classes beliebig viele.
Name der Datei muss gleich der (public) Toplevel-Klasse sein, bei != public besser auch.
Bestenfalls ist jede Klasse in eigener Datei für eine bessere Übersicht auf Dateiebene.

4.5. Erweiterte Kompilation und Ausführung

Beim Kompilieren von **Klassen in Packages**: c:\dir> javac **-d .** Foo.java
-d sagt dem Compiler, er soll eine **Ordnerstruktur gleich der des Packages** anlegen
Die Class-Dateien werden abgelegt, wie die java-Files in den Projektpackages liegen.
Der **Punkt** ist die Anweisung diese Struktur **im aktuellen Verzeichnis** abzulegen, diese Bezug kann ersetzt werden, so dass die Struktur abseits des aktuellen Orts erstellt wird.

- classpass (kurz **-cp**) sagt der JVM, wo die Datei(en) liegen: (z.B. ***hier*** java **-cp .** file.java)

Hat eine Klasse Bezug auf andere sind mehrere Pfadangaben möglich (mit ; getrennt):
c:\> java -classpass c:\targetclass; c:\auxiliaryclass packagename.Targetclass(.class)

Im ist im Einzelnen die **Zielklasse von einer Hilfsklasse abhängig**, so muss man die **Hilfsklasse zuerst kompilieren**. Umgehen lässt sich das, indem man alle zu kompilierenden Klassen gemeinsam mitgibt, so dass der Compiler die Abhängigkeiten selbst ermitteln kann, (auch Zirkelbezüge sind möglich). (Spezifizierbar mit -cp Angabe). Beispiel: **javac -d . A.java B.java**. Besser noch: **javac -d . *.java**

Optional: jar-Files (nicht examensrelevant)
Erstelle jar-File (java-„ZIP), zur Verbreitung des Programms mitsamt Ordner-Struktur.
jar -cvf targetname.jar packagename (sourcefolder), (KI-Info): -cvf = create verbose file

In einer **MANIFEST.MF**-Datei, im Ordner META-INF kann z.B. die **main class angeben**, um den Einstieg für eine **direkte Ausführung einer jar-File** zu ermöglichen. Erstellung einer jar-File mit MANIFEST: jar **-cvfm** targetname.jar **manifestname.txt** packagename

Wenn jar-File Manifest mit main class hat, dann ausführen über: **java -jar filename.jar**

4.6. Das java.lang-Package und andere Standard-Packges

Prüfungsrelevant ist erstmal nur das java.lang-Package (java.util wird empfohlen).

Sub-Package	Kurzbeschreibung	Häufig verwendete Klassen
java.lang:	Elementare Java-Funktionen Wird bereits implizit importiert! Grund(?): Objects, Exceptions…	Object, Math, System, Runtime, Exception, Runtime Exception, wrapper classes
java.util:	Werkzeuge für Datenstrukturen, Internationales, Zeit und Datum	Collection, List, ArrayList, Set, Map, Date, Locate
java.io:	Ein- und Ausgabe-Aktivitäten inkl. über Dateien und Geräte	InputStream, OutputStream, FileReader, FileWriter, IOException
java.sql:	Umgang mit relationalen Datenbanken	DriverManager, Connection, Statement, ResultSet
java.net:	Netzwerk-Kommunikation	Socket, ServerSocket
java.awt/swing:	Für grafische Oberflächen (GUI)	Frame, Dialog, Button, ActionEvent, LayoutManager

wichtige Klassen in java.lang:

Wrapper Klassen: Boolean, Charakter, (<)Byte, Short, Integer, Long, Float und Double.(<)

Java.lang.System: Hat die Variable out (PrintStream für Konsole/CLI) – methoden print/ln
Java.lang.Math: Exponential, Trigonometrie, Logarithmus, Round, Min/Max, random, abs