

Sun's Java Code Conventions

18.04.2002

Quelle:

<http://java.sun.com/docs/codeconv/>

“Convention is the ruler of all.” (Pindar)

Warum Code Conventions?

- **Pflege und Wartung** beanspruchen **80%** der Lebenszeit einer Software
- Software wird in **Teams** erstellt – auf ihre Lebenszeit gesehen von **mehreren**
- Mit Code Conventions wird **Software lesbarer**

Konvention*

allgemein Sitte,
Überlieferung;
Zusammenkunft,
Übereinkunft.

[*http://www.wissen.de](http://www.wissen.de)

- **Motivation für Euch:**

„Was wir nicht in begrenzter Zeit verstehen können, können wir auch nicht als gut bewerten.“

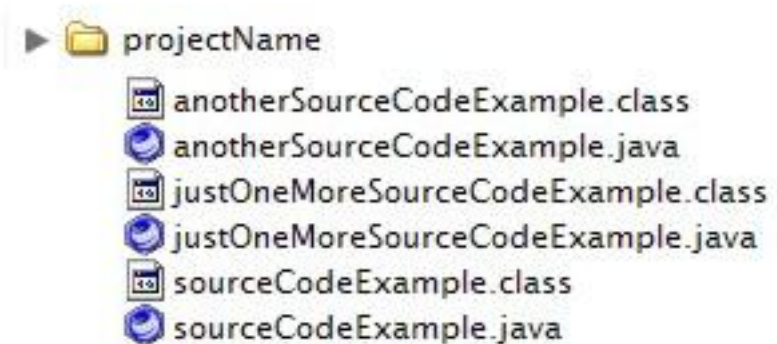
(PI2-Homepage)

Java-Files: Endungen

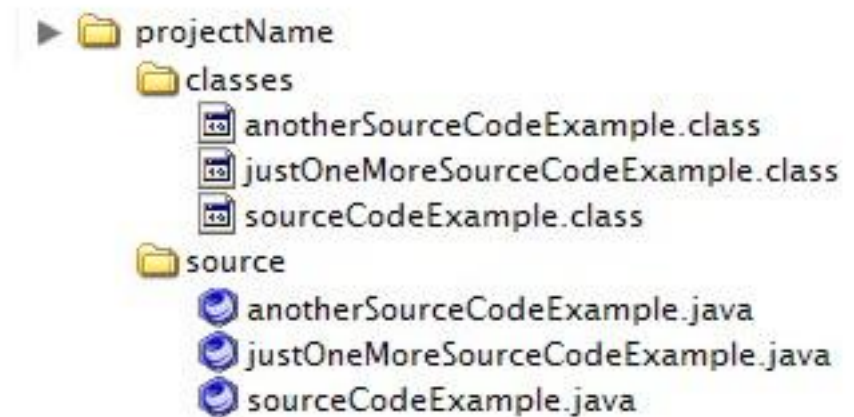
Datei-Typ	Datei-Endung
Java Sourcecode	* .java
Java Bytecode	* .class

Java-Files: Organisation

- **Kleine** Projekte (z.B. Übungszettel):
 - Source- und Bytecode-Dateien im **gleichen** Verzeichnis



- **Grosse** Projekte (z.B. Softwarepraktikum):
 - **Sinnvolle** Ordnerstruktur!!



Source-Code

- Jede **public-Klasse** in **eigene** Datei
 - **private-Klasse** und/oder **private-Interface** *kann* in Datei der **zugehörigen** **public-Class**
 - **public-Klasse** immer **erste** Klasse in Datei
 - Keine Datei **> 2000** Zeilen lang
- **Ordnungsstruktur** in einem Source-File:
 - 1) Datei-Kommentar
 - 2) package- und import-Anweisung
 - 3) Klassen- bzw. Interfacedeklarationen (dazu später mehr)
 - 4) Implementierte Klasse

Source-Code: Struktur

```
/*  
 * Classname  
 *  
 * Version information  
 *  
 * Date  
 *  
 * Copyright notice  
 */
```

Datei-Kommentar (evtl. teilweise in Java-Doc zu verfassen)

- Klassename
- Version und Information über Klasse
- Datum der Erstellung/letzen Änderung der Klasse
- Copyright

```
package java.awt;
```

```
import java.util.ArrayList;
```

package- und **import-** Anweisung

```
public class Example00 {
```

```
    ArrayList arrayList = new ArrayList();
```

```
}
```

Implementierte **Klasse**

Source-Code: Klassen- bzw. Interface-Deklarationen

```
* Copyright 2002 by BS
*/
```

```
package pi2_tutorium.examples;
```

```
/**
 * Klasse zur Repräsentation eines Autos
 * [...blabla...]
 */
```

```
public class Auto {
```

```
    private static int anzahlReifen = 4;
    protected static int anzahlErsatzReifen = 1;
    static int anzahlLenkrad = 1;
    public static int anzahlHupe;
```

```
    private String autoname;
    protected String herstellername = "Porsche";
    String reifenherstellername;
    public double kaufpreis;
```

```
    public Auto() {
    }
```

```
    public static void setAnzahlHupe(int anzahlHupe) {
        Auto.anzahlHupe = anzahlHupe;
    }

    public double berechneHerstellungskosten() {
        double Herstellungskosten = 100000.0; // mindestens!!
        // jede Menge Berechnungen
        return Herstellungskosten;
    }
}
```

- (1) Class/interface **Dokumentations-Kommentar** (`/** ... */`)
- (2) Klassen- oder Interface-**Statement**
- (3) **Klassen-Variablen** (= static)
(private, protected, package, public)
- (4) **Instanz-Variablen**
(private, protected, package, public)
- (5) **Konstruktoren**
- (6) **Methoden**
(private, protected, package, public)

Source-Code: Einrückung

- **Generell**
 - 4 Leerzeichen (bzw. 1 Tab = 8 Leerzeichen)
- **Zeilenlänge**
 - nicht länger als 80 Zeichen
- **Zeilenumbrüche**
 - Nach einem Komma
 - Vor einem Operator
 - Neue Zeile ab gleicher Einrückung wie vorherige

Source-Code: Einrückung

- Beispiele für Zeilenumbrüche (1)

```
someMethod(longExpression1, longExpression2,  
           longExpression3, longExpression4, longExpression5);
```

```
var = someMethod1(longExpression1,  
                 someMethod2(longExpression2,  
                             veryVeryVeryVeryVeryVeryVeryVerylongExpression3));
```

```
longName1 = longName2 * (longName3 + longName4 - longName5)  
            + 4 * longname6;  
            // zu bevorzugen!
```

```
longName1 = longName2 * (longName3 + longName4  
                        - longName5) + 4 * longname6;  
                        // zu vermeiden!
```

Source-Code: Einrückung

- Beispiele für Zeilenumbrüche (2)

```
// nicht diese Einrückung benutzen!!
if ( (condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { // Sch...-Umbruch
    doSomethingAboutIt(); // Diese Zeile könnte man
                          // übersehen
}
```

```
// Diese Einrückung benutzen!!
if ( (condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Source-Code: Einrückung

- Beispiele für Zeilenumbrüche (2)
 - Konditionaler Operator

```
alpha = (aLongBooleanExpression) ? beta : gamma;
```

```
alpha = (aLongBooleanExpression) ? beta  
                                     : gamma;
```

```
alpha = (aLongBooleanExpression) // Favorit!!!  
    ? beta  
    : gamma;
```

Source-Code: Kommentare

- Notation

```
//
```

ab hier **bis Ende Zeile** kommentieren

```
/* blabla */
```

von `/* bis */` kommentieren

```
/** blabla */
```

von `/** bis */` per **JavaDoc**-Konvention
kommentieren

Source-Code: Kommentare

- Kommentare **verwenden**
 - **sinnvoll!**
 - **sparsam!**
 - Code schreiben, der **ohne Kommentare** auskommt!
- wenn **JavaDoc**
 - **durchgängig!**
 - **Funktion** dokumentieren!
 - **Tags** nutzen!
 - Leser **kennen** den **Code nicht!**

Source-Code: Kommentare

- Kommentar-Formatierungen **anwenden** (1)

- **Block-Kommentar**

```
/*  
 * Dies ist ein Block-Kommentar  
 */
```

- **Einzeiler-Kommentare** (lieber nicht verwenden)

```
/* Dies ist ein Einzeiler-Kommentar */
```

- **Folge-Kommentare**

```
if ( (a % 2) == 0) {  
    return TRUE;          /* Spezialfall */  
} else {  
    return isPrime(a);    /* nur ungerade a */  
}
```

Source-Code: Kommentare

- Kommentar-Formatierungen **anwenden** (2)
 - **Kommentare am Zeilenende**

```
if ( foo > 1) {  
    // Do a double-flip.  
    ...  
} else {  
    return false; // Explain why here.  
}
```

- Achtet auf das geschickte Auskommentieren
 → **Blockkommentare**

Source-Code: Deklarationen

- Eine Deklaration pro Zeile
 - **richtig** (untereinander):

```
int zeilenlaenge;  
int zeilenabstand;
```
 - **falsch** (nebeneinander):

```
int zeilenlaenge, zeilenabstand;
```
 - auch **richtig** (Tabbing):

```
int         level;  
int         size;  
Object     currentEntry;
```
 - auch **falsch** (unterschiedliche Typen):

```
int foo, foarray[];
```


Source-Code: Deklarationen

- **Initialisierung**

- lokale Variablen dort initialisieren, wo sie auch gebraucht werden

falsch:

```
int index = 0;
... // do some stuff
for (index; index <= array.length; index++) {
    ...
}
```

richtig:

```
for (int index = 0; index <= array.length; index++) {
    ...
}
```

Source-Code: Deklarationen

- *Keine gleichen Variablen-Namen* in ineinander greifenden Blöcken verwenden

– falsch:

```
int count;
...
myMethod() {
    if ( condition) {
        int count; // Vermeiden!
        ...
    }
    ...
}
```

Source-Code: Statements

- Pro Zeile *ein* Statement

- richtig:

- `argc++;`

- `argv++;`

- falsch:

- `argv++; argc--;`

Source-Code: Statements

- **Return-Statements** (möglichst „**inlinen**“)

- **richtig:**

```
return;  
return myDisk.size();  
return (( size < 100)  
        ? size  
        : DEFAULT_SIZE);
```

- **falsch:**

```
int myDiskSize = myDisk.size();  
return myDiskSize;
```

- **auch falsch:**

```
int actualSize;  
if ( size < 100){  
    actualSize = size;  
} else {  
    actualSize = DEFAULT_SIZE;  
}  
return actualSize
```

Source-Code: Statements

- if-then-else-Statement

– immer { } benutzen!! **Wirklich IMMER !!!**

richtig:

```
if ( condition) {  
    statements;  
}
```

falsch:

```
if ( condition)  
    statements;
```

Source-Code: Statements

- **Schreibweisen** des if-else-Statements
 - richtig:

```
if ( condition) {  
    statements;  
}
```

```
if ( condition) {  
    statements;  
} else {  
    statements;  
}
```

```
if ( condition) {  
    statements;  
}  
if (!condition) {  
    statements;  
}
```

Source-Code: Statements

– falsch (mag Jan P. gar nicht gerne):

```
if ( condition) {  
    statements;  
} else if ( condition) {  
    statements;  
} else {  
    statements;  
}
```

Source-Code: Statements

- **for-Statements** sollten so aussehen

```
for (initialization; condition; update) {  
    statements;  
}
```

```
for (initialization; condition; update) {  
}
```

- **while-Statements** sollten so aussehen

```
while ( condition) {  
    statements;  
}
```

```
while ( condition) {  
}
```


Source-Code: Statements

- **do-while-Statements** sollten so aussehen

```
do {  
    statements;  
} while ( condition);
```

- **switch-case-Statements** sollten so aussehen

```
switch ( condition) {  
    case ABC:  
        statements;  
    case DEF:  
        statements;  
        break;  
    case XYZ:  
        statements;  
        break;  
    default:  
        statements;  
        break;  
}
```

Source-Code: Statements

- **try-catch-Statement** sollte so aussehen

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

- ... oder so (mit **finally**)

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

Source-Code: White Space

- **White Space** sind
 - Leerzeichen
 - Zeilenumbrüche `\n`
 - Tabulatoren `\t`
 - Systemrelevante Zeichen `\r` und `\f`
- **Leerzeichen**
 - zwischen Schlüsselwort und Parenthese
 - nach Kommata
 - trennen binäre Operatoren von den Operanden (nicht unäre Operatoren!)
 - trennen Ausdrücke in for-Schleifen
 - trennen Cast von Objekt

Source-Code: Namens-Konventionen

- **Packages**

```
com.sun.eng  
com.apple.quicktime.v2  
edu.cmu.cs.bovik.cheese  
folder1.folder2.myClass
```

- **Klassen & Interfaces**

```
class Raster;  
class ImageSprite;
```

- **Methoden**

```
run();  
runFast();  
getBackground();
```

- **Variablen**

```
int i;  
char c;  
float myWidth;
```

- **Konstanten**

```
static final int MIN_WIDTH  
    = 4;  
final int MAX_WIDTH = 999;  
static final int  
    GET_THE_CPU;
```

Programmierung: Vorgehensweisen

- **Sichtbarkeit und Zugriff** auf Klassen und deren Member
 - Interner oder externer Gebrauch?
 - Muss vererbt werden?
 - Wird dies überhaupt gebraucht?
- **Sparsam mit den Objekten**
 - Objekte erzeugen kostet (Rechner-)Zeit
 - vermeidbar:
 - richtig:**
`AClass.classMethod() ;`
 - falsch:**
`aClass.classMethod() ;`

Programmierung: Vorgehensweisen

- aus „**Magic Numbers**“
Konstanten machen

falsch:

```
berechneGebuehr ( 2.50 );
```

richtig:

```
double GRUNDGEBUEHR = 2.50;  
berechneGebuehr ( GRUNDGEBUEHR );
```

- **korrekte Parenthese**

bitte vermeiden:

```
if ( a == b && c == d )
```

so isses besser:

```
if ( ( a == b )  
    && ( c == d ) )
```

Programmierung: Vorgehensweisen

- Boolsche Ausdrücke (1)

falsch!!!

```
if ( booleanExpression)
{
return true;
} else {
return false;
}
```

richtig:

```
return
    booleanExpression;
```

- Boolsche Ausdrücke (1)

nicht so gut:

```
if ( condition) {
return x;
}
return y;
```

viel schöner:

```
return (( condition)
        ? x
        : y);
```

Programmierung: Vorgehensweisen

- **schlecht, gaaaanz schlecht:**

```
fooBarChar = barFooChar = 'c';
```

```
d = (a = b + c) + r;
```

- **viel besser:**

```
fooBarChar = 'c';
```

```
barFooChar = 'c';
```

```
a = b + c;
```

```
d = a + r;
```

- **Ausdruck vor ? in konditionalen Operatoren**

wenn binärer Operator, dann klammern:

```
(x >= 0) ? x : -x;
```