Kommentare

- Motivation
- Notation: 2 Arten von Kommentaren
- Anwendung: Einsatz zum Testen
- JavaDoc
  - Überlegungen
  - Notation
  - Exkurs: Eclipse

Motivation

- Kommentare dienen hauptsächlich der *Dokumentation* von Programmen
  - damit sie auch Personen, die nicht an der Programmierung beteiligt waren, verstehen können und
  - damit derjenige, der sie programmiert hat, sie besser nachvollziehen, erklären und bei Bedarf ändern kann, wenn er sich einige Zeit nicht mit ihnen beschäftigt hat

- **60 – 80%** aller Entwicklungsarbeiten sind Wartung und Weiterentwicklung!

2 Arten von Kommentaren in Java 1/2

- Kommentare können mit // eingeleitet werden; diese Kommentare enden am Zeilenende

Kommentar am Zeilenende (selten verwendet)

Beispiele

```
mid = min + (max – min) / 2;        // Mitte des Suchbereichs bestimmen

// Wiederholen, solange das gesuchte Element im Bereich liegen kann
while ( element[min] <= search) && (search <= element[max]) && !found ) {
    if ( element[mid] == search ) {
        ...
```

Kommentarzeile

Bei Code-Änderungen (Zeilen kürzer/länger) müssen die Kommentare ständig umformatiert werden, um "schön" auszusehen

Negativ-Beispiele für Kommentare

```
min = 0;                            // Hier werden die Grenzen
max = ARRAY_LEN;                    // für die zukünftige Suche
mid = min + (max – min) / 2;        // erstmalig festgelegt
i = i + 1;                          // die Variable i wird um 1 erhöht
```

Das sieht man (hoffentlich!) ohne Kommentar

2 Arten von Kommentaren in Java 1/2

- Kommentare können mit // eingeleitet werden; diese Kommentare enden am Zeilenende

Kommentar am Zeilenende (selten verwendet)

Beispiele

```
mid = min + (max – min) / 2;        // Mitte des Suchbereichs bestimmen
```

```
// Wiederholen, solange das gesuchte Element im Bereich liegen kann
while ( element[min] <= search) && (search <= element[max]) && !found ) {
    if ( element[mid] == search ) {
        ...
```

Kommentarzeile

Bei Code-Änderungen (Zeilen kürzer/länger) müssen die Kommentare ständig umformatiert werden, um "schön" auszusehen

Verbessertes Code-Fragment

Ein lesbarer Name erspart den Kommentar von oben

```
// Hier werden die Grenzen für die zukünftige Suche erstmalig festgelegt
linkeGrenze = 0;
rechteGrenze = ARRAY_LEN;
mitteDesSuchbereichs = linkeGrenze + (rechteGrenze – linkeGrenze) / 2;
i = i + 1;
```

Das sieht man (hoffentlich!) ohne Kommentar

2 Arten von Kommentaren in Java 2/2

- Kommentare können mit /* eingeleitet und mit */ abgeschlossen werden; diese Kommentare können sich über viele Zeilen erstrecken
- Beispiel

```
/*  Hier wird eine binäre Suche implementiert:
 *  Innerhalb der Schleife wird das Suchelement iterativ eingegrenzt:
 *  Ist es gleich dem mittleren Element, ist die Suche erfolgreich beendet.
 *  Liegt es in der linken Hälfte, wird dort weitergesucht
 *  Liegt es in der rechten Hälfte, wird dort weitergesucht
 *  Die Schleife endet, wenn es außerhalb des Feldes liegt
 */
while ( element[min] <= search) && (search <= element[max]) && !found ) {
    if ( element[mid] == search ) {
        found = true;
    } else if ( element[mid] > search ) {
        ...
    } else { // element[mid] < search
        ...
    }
}
```

Eclipse fügt die Sterne ein, das muss aber *nicht* so sein!!

Das muss nicht mehr in einem weiteren „else if“ getestet werden, der Kommentar ist vielleicht gut für das Verständnis beim Lesen

Einsatz von Kommentaren beim Testen

- (Spätestens) Nachdem man ein Programm entwickelt hat, sollte es gründlich getestet werden
- Es gibt viele verschiedene Test-Strategien;
  dazu in anderen Vorlesungen mehr
- Tritt zum Beispiel ein Berechnungsfehler auf, ist es hilfreich, Kontrollausgaben in das zu testende Programm einzubauen, um den Fehler einzugrenzen

  Beispiel

```
while ( element[min] <= search) && (search <= element[max]) && !found ) {
/*  println( "min: " + min );                    ← diese Ausgaben
    println( "mid: " + mid );                    ← sind
    println( "max: " + max );                    ← lediglich
    println( "search: " + search );              ← Kontrollausgaben
*/  if ( element[mid] == search ) {
        ...
```

Läuft das Programm, werden die Kontrollausgaben
zunächst nicht mehr benötigt: vielleicht aber später noch einmal?

Einsatz von Kommentaren beim Testen

- (Spätestens) Nachdem man ein Programm entwickelt hat, sollte es gründlich getestet werden
- Es gibt viele verschiedene Test-Strategien;
  dazu in anderen Vorlesungen mehr
- Tritt zum Beispiel ein Berechnungsfehler auf, ist es hilfreich, Kontrollausgaben in das zu testende Programm einzubauen, um den Fehler einzugrenzen

  Beispiel

```
while ( element[min] <= search) && (search <= element[max]) && !found ) {
//  println( "min: " + min );                    ← diese Ausgaben
//  println( "mid: " + mid );                    ← sind
//  println( "max: " + max );                    ← lediglich
//  println( "search: " + search );              ← Kontrollausgaben
    if ( element[mid] == search ) {
        ...
```

Läuft das Programm, werden die Kontrollausgaben
zunächst nicht mehr benötigt: vielleicht aber später noch einmal?

Haben Sie Fragen?

An welchen Teilen eines Programms orientiert sich ein Leser?

- Am Programmaufbau (= eine oder mehrere Klassen)
- An Methoden, deren Parametern, Rückgabewerten... (die in Schnittstellen deklariert werden)

→ Genau diese Programmteile müssen sorgfältig kommentiert werden!

Was passiert mit Programmen und ihrer Dokumentation?

- Dokumentation (in begleitenden Dokumenten) wird oft vernachlässigt
  - Sie wird ungern erstellt (unpraktisch, mit mehreren Dateien zuarbeiten)
  - Sie wird ungern gelesen (unpraktisch, woanders nachschauen zu müssen)
  - Sie wird nicht / nicht gerne weiterentwickelt / vergessen, wenn das Programm weiterentwickelt wird (weil man verschiedene Dateien vergleich muss)
  - → veraltet relativ schnell

→ Die Dokumentation von Klassen, Interfaces, Methoden etc. sollte *im Programm selbst* erfolgen und dort unterstützt werden!

→ Das Ergebnis: JavaDoc

Beispiele

```java
/**
 * Dieser Filter filtert alle (potenziellen) Fehler aus
 * der ContactManager-Implementierung weg und gibt sie in
 * einer Dialogbox aus. Dann wird bei Bedarf ein Default-Wert
 * als Ergebnis geliefert.
 */
public class ContactManagerWrapper implements ContactManager {
```

```java
/**
 * Dieses Interface legt fest, welche Funktionalität die
 * Kontaktverwaltung bietet.
 * Die Bedeutung der einzelnen Methoden ist bei jeder
 * Methode beschrieben.
 */
public interface ContactManager {
```

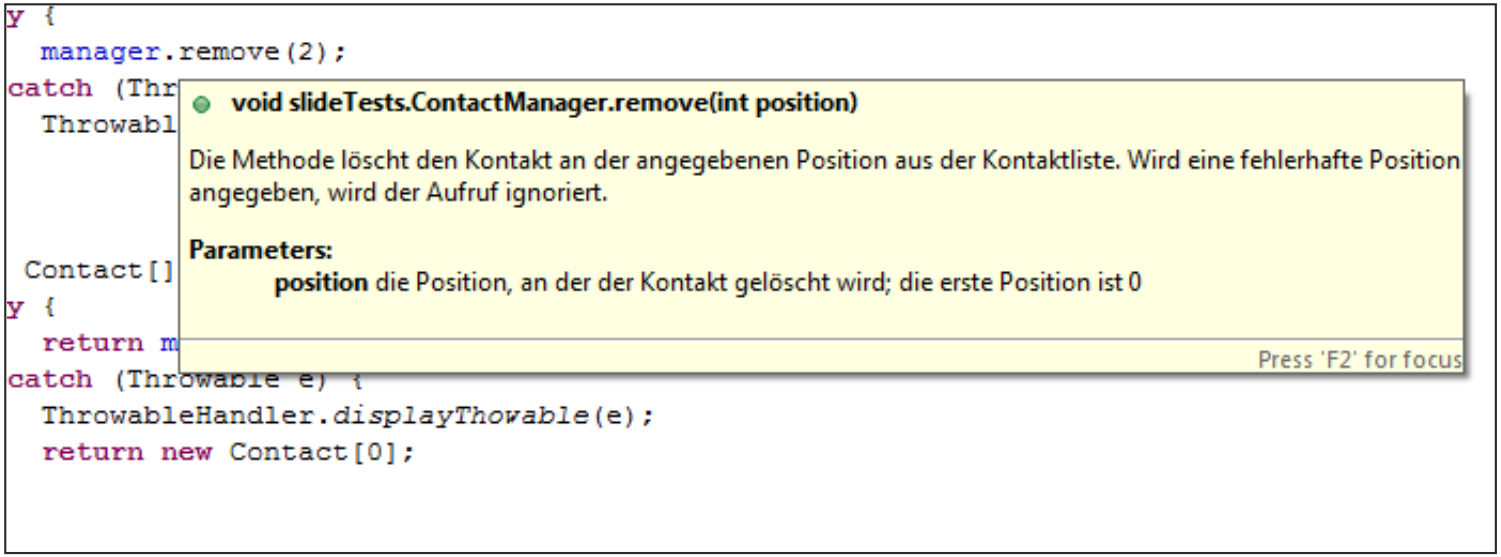
```java
/**
 * Die Methode fügt den neuen Kontakt zu den
 * bereits vorhandenen Kontakten hinzu.
 * @param contact der Kontakt, der neu hinzugefügt wird
 */
void add( Contact contact );

/**
 * Die Methode löscht den Kontakt an der angegebenen
 * Position aus der Kontaktliste. Wird eine fehlerhafte Position
 * angegeben, wird der Aufruf ignoriert.
 * @param position die Position, an der der Kontakt gelöscht wird;
 * die erste Position ist 0
 */
void remove( int position );
```

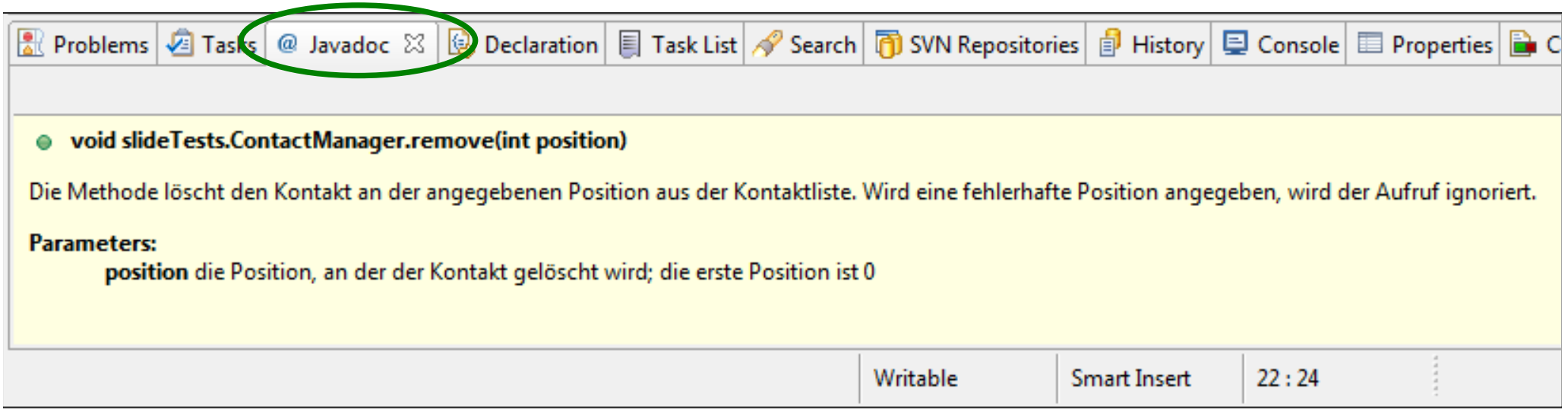
Syntax

- Javadoc-Kommentare werden mit /** eingeleitet und mit */ abgeschlossen
- Sie können sich über beliebig viele Zeilen erstrecken
- Besondere *tags* darin sind unter anderem

@param	Beschreibung der Parameter
@return	Rückgabewert einer Methode
@throws	Beschreibung der Situationen, in denen eine Exception geworfen wird
@author	Autor des Interface‘ / der Klasse / der Methode
@see	Verweis auf ein anderes Interface, eine andere Klasse oder eine andere Methode; Beispiele @see java.util.Date @see java.lang.String#length()

Exkurs: Anzeige in Eclipse





Nutzung von Javadoc

- Nutzen Sie Javadoc für die Dokumentation von Methoden die Sie sich selbst überlegt (entworfen) haben!

- Dokumentieren Sie nichts Überflüssiges, Negativ-Beispiele
  - Text der Aufgabenstellung auf dem Übungsblatt wiederholen
  - @param vorname der Vorname
  - @return Maximum der übergebenen Zahlen
    int maximum(int... zahlen) {...
- Nutzen Sie Javadoc *zukünftig* auch für
  - Ihre Klassen
  - Ihre Interfaces