Programmiermethodik 1/ Programmiertechnik

Varargs + Dokumentation + Codeformatierung + Debugging

Wiederholung

- Statische Objektvariablen
- Statische Methoden
- Aufzählungstypen

Ausblick

Worum gehts?

Agenda

- Variable Parameteranzahl in Methoden
- Dokumentation
- Code-Formatierung
- Debugging

Variable Parameteranzahl bei Methoden

Einführung

- bisher feste Anzahl Argumente bei Methodenaufrufen
- void methode(double doubleArgument, int intArgument);
- manchmal Bedarf: variable Anzahl von Argumenten
    - Vararg (variable length argument lists)
    - kann eine variable Anzahl an Argumenten übergeben werden
- Syntax
    - drei Punkte direkt nach Typangabe

```
<Typ>... <Variablenname>
```

- Beispiel:

```
int summe(int... summanden){
   …
}
```

Verwendung

- ausschließlich in Parameterlisten erlaubt!
- im Methodenrumpf verwendbar wie ein Array

```
int summe(int... summanden)
```

- ist im Rumpf der Methode gleichwertig mit ...

```
int summe(int[] summanden)
```

- Beispiel: Addition aller Argumente

```
int summe(int... summanden){
   int ergebnis = 0;
   for(int summand: summanden){
      ergebnis = ergebnis + summand;
   }
   return ergebnis; // Summe zurückliefern
}
```

Methodenaufruf

- Aufrufer liefert beliebig viele Argumente für einen Vararg-Parameter
- jedes einzelne Argument muss kompatibel zum Vararg-Parametertyp sein
- Parameterübergabe:
    - Erzeugen eines neuen Arrays mit Länge = Anzahl Argumente
    - Initialisieren des Arrays mit Argumentwerten
    - Zuweisen des Arrays an den Vararg-Parameter

- Beispiele:

```
System.out.println(summe(1, 2, 3));  // 6
System.out.println(summe());            // 0
System.out.println(summe(97, summe(1, 2))); // 100
```

Einschränkungen

- nur ein Vararg-Parameter pro Parameterliste erlaubt
- Vararg-Parameter muss letzter in der Parameterliste sein
- vorausgehende Parameter werden normal behandelt

Beispiel

- Zähle Anzahl Werte in einem int-Bereich

```
int zaehleWerte(int unten, int oben, int... werte){
   int anzahl = 0;
   for(int x: werte){
      if(x >= unten &&  x <= oben){
         anzahl++;
      }
   }
   return anzahl;
}
```

- mindestens zwei Argumente beim Aufruf nötig:

```
zaehleWerte(5, 10, 6, 2, 12, 8) → 2
zaehleWerte(5, 10) → 0
```

Varargs und Überladen

- Was passiert, wenn man eine Methode überlädt (z.B. zwei int-Parameter) und gleichzeitig eine Variante mit einer varargs Parameterliste (auch int) anbietet?
- Beispiel

```
public void methode(int... zahlen) {
   System.out.println("Methode mit varargs
      Parametern aufgerufen");
}
public void methode(int zahl1, int zahl2) {
   System.out.println("Methode mit zwei Parametern aufgerufen");
}
```

- Auflösung: Variante ohne varargs wird bevorzugt.

Varargs und Überladen

- Regelwerk
  - Widening beats Boxing (Vererbung/Interfaces vs. Auto(un)boxing)
  - Boxing beats Varargs (Auto(un)boxing vs. Varargs)
  - Legacy beats Varargs (konkrete Parameter vs. Varargs)

Übung: Varargs

- Schreiben Sie eine Methode kuerzester(), die beliebig viele Stings als Argumente bekommt und den kürzesten davon zurückgibt.
- Beispiel:

```
kuerzester( „aaa“, „bb“, „c“, „dddd“) → „c“
```

Dokumentation

Einführung

- Dokumentation wichtiger Bestandteil der Software-Entwicklung
    - wie Quellcode
    - wie Tests
- Dokumentation wird teilweise vernachlässigt
    - z.B. weil Programm auch ohne Dokumentation läuft
    - z.B. weil Dokumentation oft an anderem Ort liegt

Javadoc

- Java bietet einen Mechanismus, zum automatischen Erzeugen von API-Dokumentation: Javadoc
    - Integration der Dokumentation in den Entwicklungsprozess
    - Dokumentation findet sich an gleicher Stelle wie Quellcode
- Aufnahme aller Packages, Klassen, Methoden
- Ausgabeformat
    - HTML

Javadoc

- Verwendung von Block-Kommentaren

```
/**
 * ...
 */
```

- wichtig
  - keine Kommentare durch //
  - zweites einleitendes * relevant
- Blockkommentare stehen vor dem beschriebenen Quellcode
  - Klasse oder Interface
  - Objektvariable oder Klassenvariable
  - Methode
- Das Symbol * wird im Blockkommentar ignoriert

Aufbau eines Javadoc-Kommentars

- drei Abschnitte
  - Zusammenfassung in einem Satz mit Punkt am Ende
  - Ausführliche Beschreibung als Freitext
  - Liste von Tags mit besonderen Informationen

```
/**
 * Hinzufügen eines Elementes in die Datenstruktur.
 *
 * Es wird eine zusätzliches Element an die nächste freie
 * Position im Array gesetzt. Der Index auf das neueste Element
 * wird um 1 erhöht. Falls das Array über keine freien Plätze
 * verfügt, wird ein neues Array mit der doppelten Größe erzeugt.
 * Außerdem werden die bestehenden Einträge in das neue Array
 * übertragen.
 *
 * <TAGs>
 */
```

Tags

- markieren Informationen mit bestimmter Bedeutung
- beginnen mit einem @-Zeichen
- dann folgt ein Schlüsselwort
    - z.B. @author
- für jeden Tag wird eine neue Zeile im Doc-Kommentar verwendet
- Tags stehen am Anfang der Zeile
- Text hinter einem Tag kann sich über mehrere Zeilen erstrecken

Tags

- für Klassen und Interfaces
  - @author <text>
    - Name des Autors
    - je einmal pro Autor verwendet
  - @version <text>
    - Versionsnummer des Quelltextes
    - wird teilweise von Systemen zur Verwaltung von Quellcode automatisch gesetzt

Tags

- für Methoden
  - @param <name> <text>
    - erläutert die Bedeutung des Parameters <name>
    - Typ wird nicht genannt
    - Klarstellung des zulässigen Werte
    - Reihenfolge der Parameter in Signatur muss zur Tag-Reihenfolge passen
  - @return <text>
    - beschreibt Methodenergebnis (Rückgabewert)
    - besonders Ausnahmeergebnisse (z.B. -1 als Index, falls Elements nicht gefunden)
    - wird nicht bei Konstruktoren und void-Methoden verwendet
  - @exception <exceptionclass> <text>
    - beschreibt die Umstände, die zum Werfen der Exception führen
    - wird für jede geworfene Exception einzeln durchgeführt

Javadoc Kommandozeilenwerkzeug

- Programm zum Erzeugen der Dokumentation: javadoc
- Syntax
    - javadoc [options] [packagenames] [sourcefiles] [@files]
- Kommandozeilenparameter für [options] (Auszug)
    - -d <path>
        - Zielverzeichnis
    - -public
        - Dokumentation nur von public-Elementen (öffentliche Schnittstelle)
    - -author
        - Übernahme des @author-Tags in Dokumentation
    - -version
        - Übernahme des @version-Tags in Dokumentation
    - -help
        - weitere Informationen zur Verwendung des Kommandozeilenwerkzeugs

Javadoc Kommandozeilenwerkzeug

- Verzeichnisstruktur
- <Projektverzeichnis>
    - src/edu/tipr1/adt/<Quellcode-Dateien>
    - testdoc
- Aufruf von javadoc im Verzeichnis src
    - javadoc
    - -d ../testdoc/
    - -classpath /Applications/eclipse/plugins/ org.junit_4.10.0.v4_10_0_v20120426-0900/junit.jar:.
    - edu.tipr1.adt
- API-Dokumentation im Verzeichnis testdoc

Generierte Dokumentation

- Menge von HTML-Seiten
    - eine Seite pro Klasse
- Abschnitte
    - Field Summary
    - Constructor Summary
    - Method Summary
    - und später
    - Field Detail
    - Constructor Detail
    - Method Detail

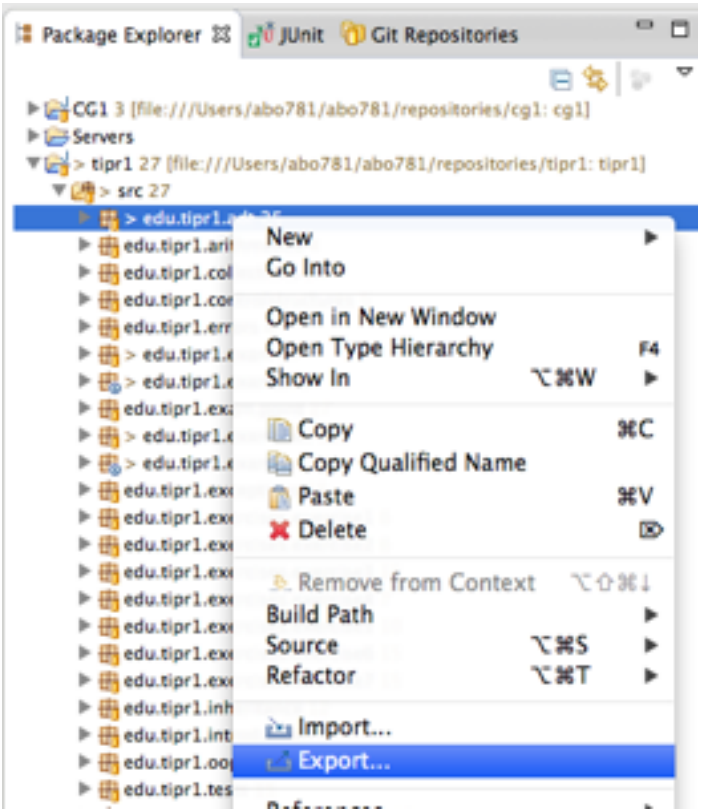
Hinweise

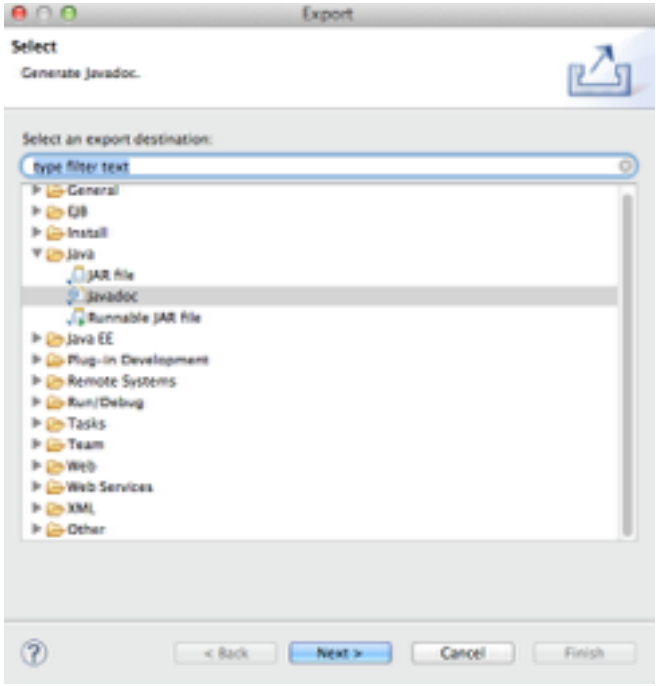
- Kommentare werden vollständig in die HTML-Seiten übernommen
- daher ist es möglich, HTML-Tags zu verwenden
    - nur in Ausnahmesituation verwenden
    - schlechter Stil
    - möglicher (sinnvoller) Einsatz: Verlinken einer E-Mail-Adresse
        - @autor Philipp Jenke

Javadoc aus Eclipse heraus

- Rechtsklick auf das Package
  - Export

Auswahl
Java - Javadoc





Übung: JavaDoc

- Schreiben Sie einen JavaDoc-Kommentar für die folgende Methode

```
private static String kuerzester(String... woerter)
```

Code Konventionen

Layout

- Layout = optisches Erscheinungsbild des Quellcodes
- umfasst Zeilenumbruch, Einrückung, Leerzeichen (Zwischenraum), Kommentare, Leerzeilen
- Compiler ignorieren das Layout weitgehend
- aber: zwischen aufeinanderfolgenden Wörtern muss mindestens ein Leerzeichen (Zwischenraum) stehen. Falsch wäre zum Beispiel: classHello
- Compiler akzeptieren auch:
  class Hallo{public static void main(String[] args){System.out.println("Hello, World!");}}
- Unterstützung in Eclipse
  - Ctrl–Shift-F

Best Practices: Formatierung

- Richtlinien für guten Quellcode:
  - nicht mehr als eine Anweisung pro Zeile
  - Text zwischen geschweiften Klammern einrücken
  - nie mehr als 80 Zeichen pro Zeile
  - nie mehr als 20 Anweisungen pro Block

Kommentare

- Erläuterung zum Programm als Freitext
- Compiler behandeln Kommentare als Zwischenraum und ignorieren den Inhalt
- Kommentare dienen einem menschlichen Leser zur Orientierung und zum Verständnis

Kommentare

- Ziel
- Erläuterung von Sinn, Zweck, Wirkung von Programmabschnitten
- Kommentare sollen nicht:
- offensichtliches wiederholen

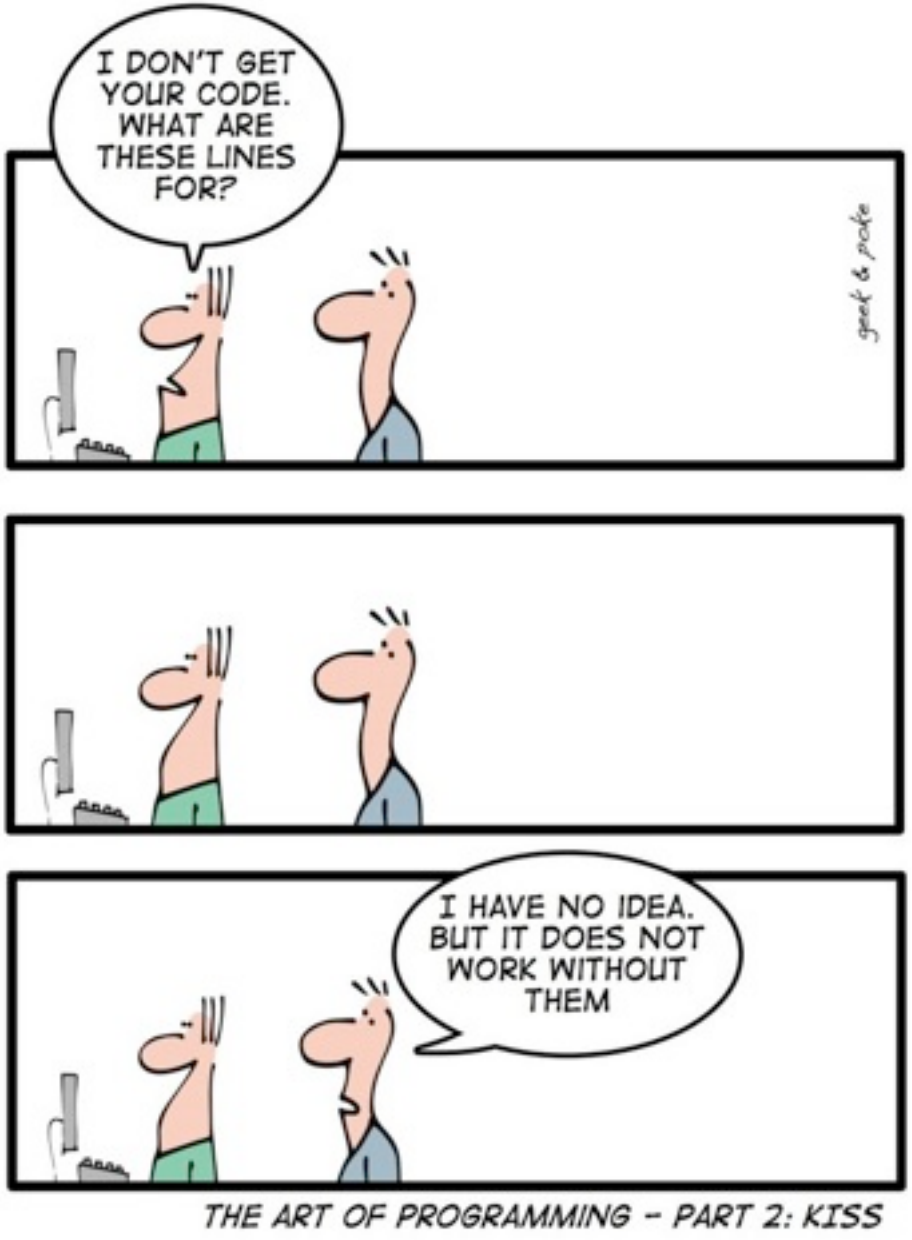
  System.out.println("blah"); // hier wird "blah" ausgegeben

  schlechten Code rechtfertigen

```
/* Das hab ich zwar gestern geschrieben, aber
 * heute verstehe ich nicht mehr, was die folgende
 * Anweisung soll. Wenn man sie löscht, funktioniert
 * nichts mehr. Also einfach stehen lassen, wird
 * schon irgendwie gut gehen. */
...
```

Kommentare

- so nicht!



Kommentare

- Kommentare generell großzügig verwenden
    - zu viel Kommentar schadet kaum
    - zu wenig Kommentar kann funktionierenden Code wertlos machen!

Bezeichner - Namenskonventionen

- generell
  - normalerweise kleine Buchstaben verwenden
  - neue Wortteile mit großen Buchstaben (CamelCode oder CamelCase)
  - aussagekräftige Namen suchen
- später: spezielle Anforderungen je nach "Ding", dass benannt wird

Bezeichner - Namenskonventionen

- Konstantennamen
    - mit Großbuchstaben geschrieben
    - z.B. MAXIMUM
- Klassennamen
    - beginnen mit einem Großbuchstaben
    - z.B. Sum
- Methodennamen
    - beginnen mit einem kleinen Buchstaben
    - beginnen mit einem aussagekräftigen Verb
    - z.B. main

Demo: Refactoring

`Klasse1.java`

Debugging

Fehlersuche

- Situation
  - Festgestellt, dass ein Fehler im Programm ist
  - Ursache (Stelle im Quellcode) noch nicht

Fehlersuche

- Testen entdeckt die Auswirkungen (Symptome) von Fehlern, Debugging beseitigt die Ursachen (Fehlerquellen)
- Debugging nur sinnvoll mit …
    - Zugriff auf Quellcode
    - Verständnis der Arbeitsweise
    - Möglichkeit zur strukturieren Modifikation
- Aufgaben beim Debugging: Fehlerquelle im Quellcode …
    - suchen und lokalisieren
    - sinnvoll und nachhaltig beseitigen
- getrennte Probleme, einzeln zu bewältigen
- anschließend sind neue Tests erforderlich

Tracing

- Einfachste Technik bei der Fehlersuche von logischen Fehlern: Tracing (von engl. "trace" = "Spur") = Protokollierung des Programmablaufs mit Ausgaben des Zustands (Methodenname, Variablenwerte, ..)
- Realisierbar durch Ausgabeanweisungen an Schlüsselstellen im Sourcecode, zum Beispiel
    - nach Eintritt in Methode, vor Rückkehr aus Methode (hilfreich: nur eine return-Anweisung)
    - am Beginn von Schleifenrümpfen
    - am Anfang von if/else-Blöcken
- Vorteil: einfach und ohne Unterstützung von Werkzeugen anwendbar

Tracing-Tipps

- Zur Ausgabe auf der Konsole System.err statt System.out verwenden, da dann keine Pufferung stattfindet
    - wichtig für die Visualisierung parallel laufender Programmteile schon mal dran gewöhnen!
- für Trace-Ausgaben eine eigene Methode verwenden, z.B.

```
void trace(String ausgabe){ ..}
```

- Für das An- und Abschalten der Trace-Ausgaben eine boolean-Variable deklarieren, z.B.

```
private static final boolean TRACE_MODUS = true;
```

- durch final kann der Compiler den Code optimieren, ohne final kann das Tracing zur Laufzeit an- und abgeschaltet werden, wenn nur einzelne Abschnitte verfolgt werden sollen!)

Tracing-Beispiel

```
private static final boolean TRACE_MODUS = true;
...

public void machDochWas(int n){
   trace("Starte machDochWas mit n = " + n));
   ...
}

public void trace(String ausgabe){
   if (TRACE_MODUS)
      System.err.println(ausgabe);
   }
}
```

Debugger

- Programm zur kontrollierten Ausführung eines Anwendungsprogramms
- wichtigste Debugger-Befehle:
    - vor Ausführung des zu überwachenden Programms Haltepunkt ("Breakpoint") in einer Codezeile setzen oder löschen
    - Ausführung bis zum nächsten Haltepunkt fortsetzen / abbrechen
    - eine einzelne Anweisung ausführen und dann wieder anhalten
    - lokale Variablenwerte ausgeben
    - aktuelle Methodenaufrufsituation ("Stack Trace") ausgeben

Java-Debugger

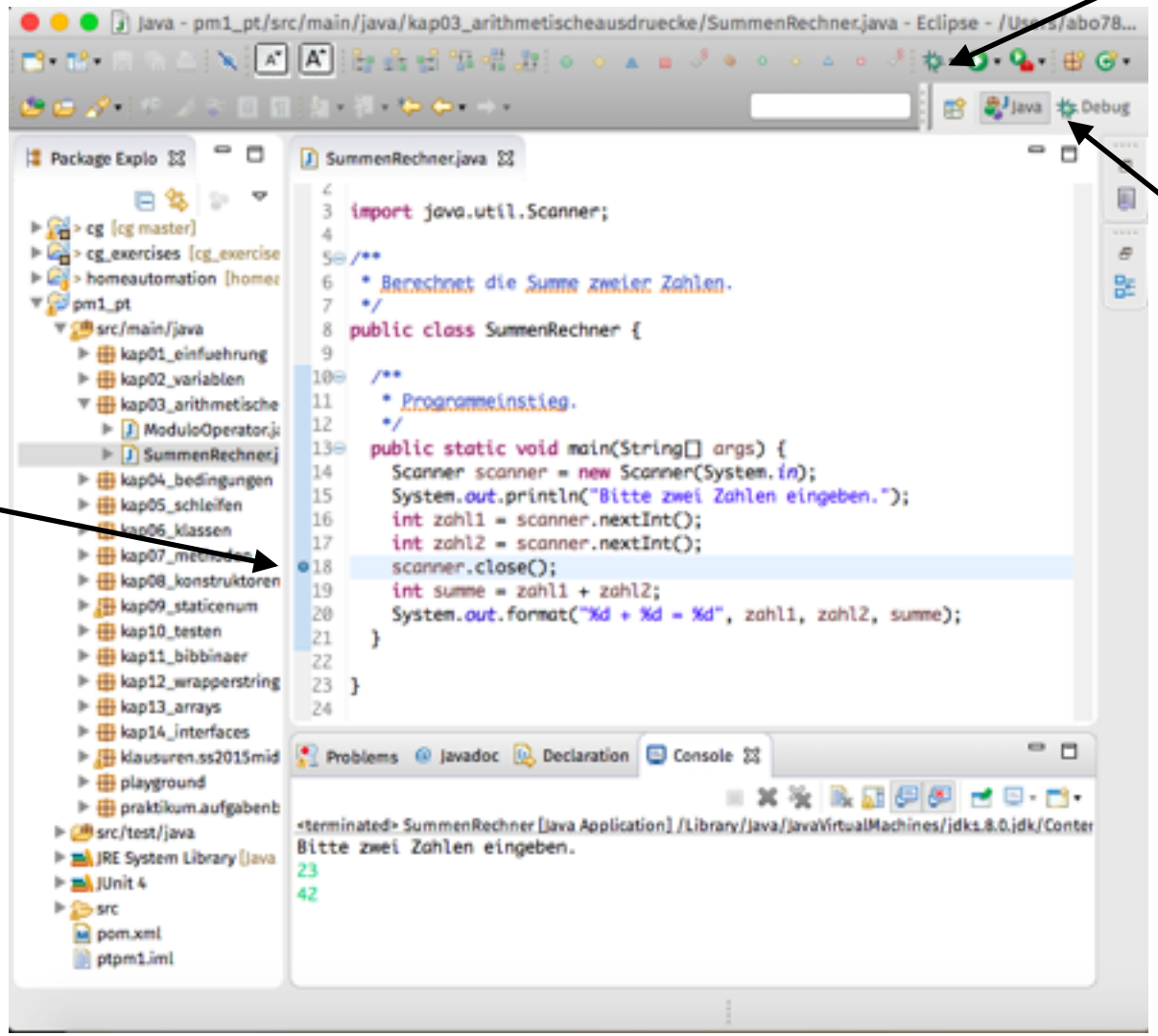
- *jdb*
  - Teil des Java SDK ⇒ mit Entwicklungssystem immer verfügbar
  - Einfache Kommandozeilen-Oberfläche
  - Vorbereitung: Sourcecode mit Schalter -g compilieren, Beispiel:
  - javac –g GGT.java: Class-Datei erzeugen (compilieren)
  - jdb GGT: Start des GGT-Programms mit jdb statt java
  - anschließend werden Befehle akzeptiert
- *jdb*-Befehle:
  - help, stop at, clear, run, cont, step, next, dump, locals, where, exit, ...

Eclipse-Debugger

- Eclipse bringt Debugger-Integration mit

Eclipse-Debugger

Starten des Debuggers



Breakpoint

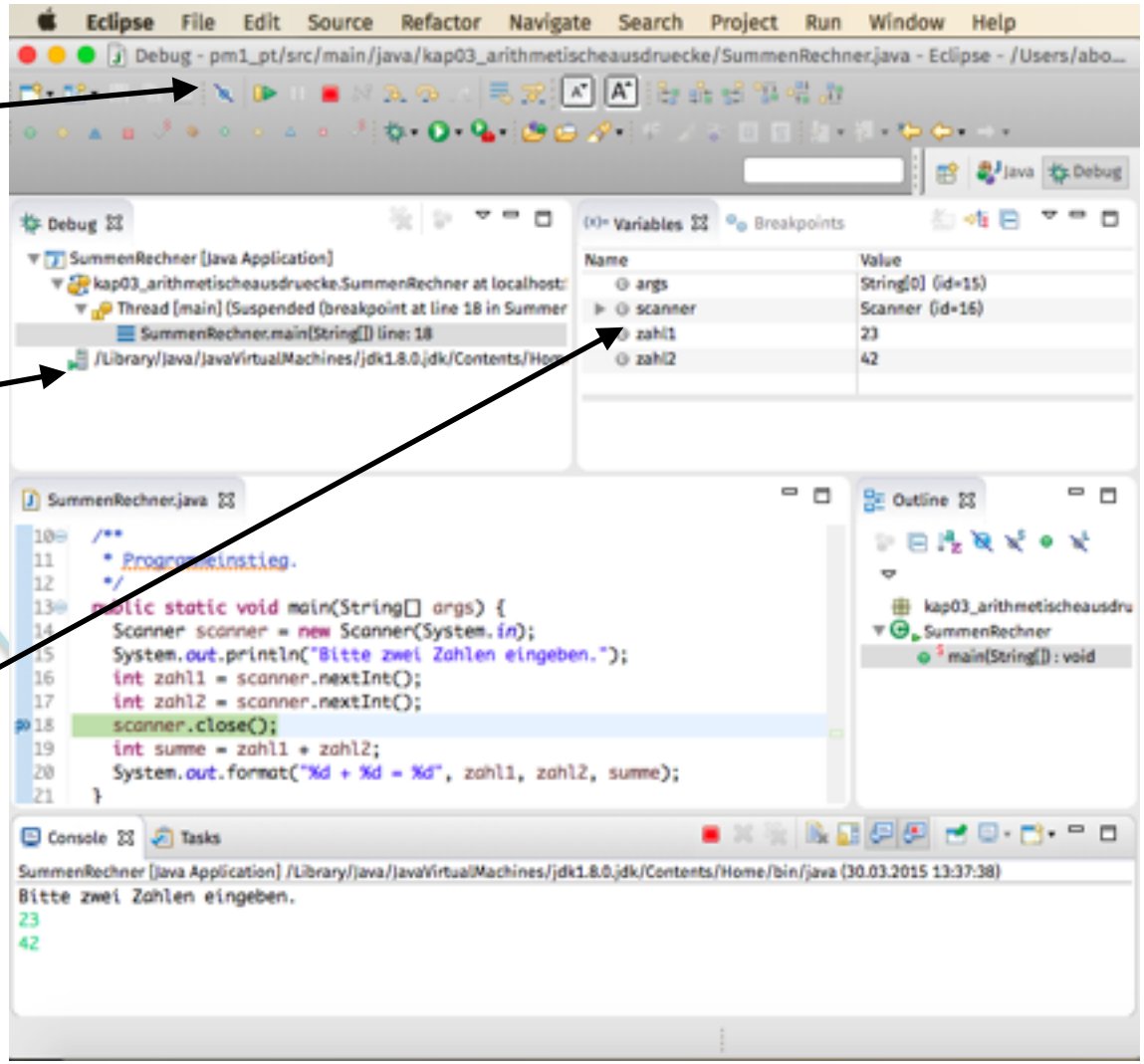
Wechsel Perspektive: Java vs. Debug

Eclipse-Debugger

Steuerung des Programm-ablaufs

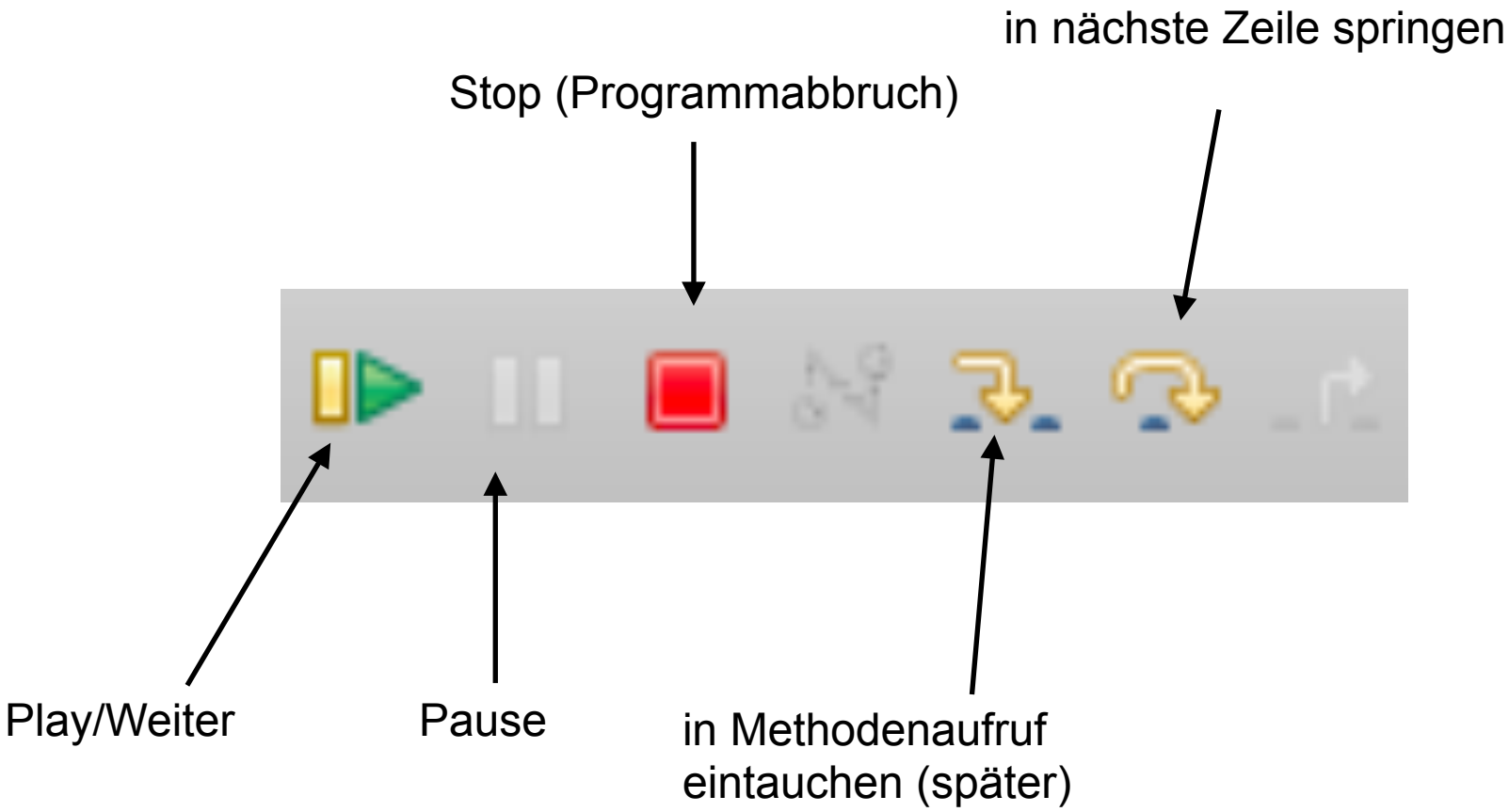
Aktueller Zustand des Programm-ablaufs

Aktuelle Belegung der Variablen



Eclipse-Debugger

- Steuerung des Programmablaufs



Demo: Debugging

`Bruch.java`

Zusammenfassung

- Variable Parameteranzahl in Methoden
- Dokumentation
- Code-Formatierung
- Debugging