# Programmiermethodik 1
# Programmiertechnik

## Wahrheitswerte, Bedingte Anweisungen

# Wiederholung



- Arithmetische Ausdrücke
- Fließkommazahlen
- Kompatibilität
- Wahrheitswerte

# Ausblick für heute

# Use Cases



- Abhängig von einer Bedingung (wahr oder falsch) sollen zwei verschiedene Dinge getan werden.
- Ich weiß, dass in meinem Problem ein Fehler ist. Ich versuche daher, das Verhalten (Abläufe, Variablenwerte) nachzuverfolgen.

# Agenda



- Wahrheitswerte
- Bedingte Anweisungen
- Fehlersuche/Debugger

# Bedingte Anweisungen

# Bedingte Anweisung



Falls <Bedingung> dann <tu-dies> ansonsten <tu-das>.

- if-Anweisung, Alternative, bedingte Anweisung, Verzweigung
- besteht aus
  - 1. Bedingung: (engl. condition)
  - 2. Konsequente: untergeordneter Anweisung
- untergeordnete Anweisung wird nur dann ausgeführt
  - wenn die Bedingung zutrifft
  - andernfalls übergangen
- Syntax:
  - `if (<Bedingung>) <Anweisung>`

# Beispiel



```
if ( temperature < 0 ){
    System.out.println("Es gibt Schnee!");
}
```

- Der Text wird nur ausgegeben, wenn die Variable `temperature` einen negativen Wert hat

# Bedingung



- neue Art von Ausdruck:
  - „trifft zu“ oder „trifft nicht zu“
  - keine dritte Möglichkeit
- Bedingung
  - Ausdruck mit ja/nein-Ergebnis
  - z.B. aufgrund eines Vergleichs
- Ergebnis der Auswertung des Bedingungsausdrucks
  - z.B. Durchführung des Vergleichs
  - keine Zahl, sondern ein Wahrheitswert (`true`/`false`)

# Beispiel: Berechnung der Tage eines Monats



```
int month = ...;  // 1..12

if((month == 4) || (month == 6) || (month == 9) ||
   (month == 11)){
        days = 30;
} else if(month == 2){
        days = 28;
} else {
        days = 31;
}
```

# Teilweise Auswertung



- Einzelbedingungen in logischen Ausdrücken sind oft voneinander abhängig
- Beispiel
  - `if( b != 0 && a/b > 0 ) …`
  - Standard:
    - zuerst beide Operanden auswerten: `b != 0` und `a/b > 0`
    - dann Ergebnisse mit AND verknüpfen
  - hier Problem, falls `b == 0`:
    - Division durch 0 → Programmabbruch!
- Lösung: teilweise Auswertung (shortcut evaluation)
  - Auswertung wird beendet, wenn das Ergebnis nach dem ersten Operanden feststeht
  - der verbleibende, zweite Operand wird dann nicht mehr berechnet

# Teilweise Auswertung



- bei `&&`:
  - erster Operand `false` → logischer Ausdruck ist `false`
- bei `||`:
  - erster Operand `true` → logischer Ausdruck ist `true`
- nicht möglich bei `^` (kein vorzeitiges Ergebnis ableitbar)
- Steuerung teilweiser/vollständiger Auswertung durch Operatorenwahl:
  - `&&` und `||` werten teilweise aus
- Beispiel

```
// 1. operand evaluated, expression is false, if b == 0
if ( b != 0 && a/b > 0 ) …
```

# Übung: Max3If



- Erstellen Sie ein Programm `Max3If`, das von 3 übergebenen Integer-Werten den größten Wert ermittelt und ausgibt!

- Anforderungsanalyse
  - Eingabe:
    - Der Benutzer gibt 3 ganzzahlige Werte ein
  - Ausgabe:
    - Der größte der drei Werte wird ausgegeben

→ Verwenden Sie für Ihren Algorithmus keine Bibliotheksmethode!

# Zweiseitige Bedingte Anweisung



- enthält eine Bedingung und zwei untergeordnete Anweisungen
- wenn die Bedingung zutrifft
  - wird die erste Anweisung ausgeführt ("then-Fall", Konsequente)
  - andernfalls die zweite ("else-Fall", Alternative)
- Syntax:

```
if (<Bedingung>)
        <Anweisung>  // Konsequente
else
        <Anweisung>  // Alternative
```

# Zweiseitige Bedingte Anweisung



**Beispiel**

- x enthält einen beliebigen Wert
- in a soll dessen Absolutwert (Betrag) berechnet werden:

```
double x = …;
double a;
if(x >= 0) {
        a = x;
System.out.println("x war positiv oder null");
} else {
a = -x;
System.out.println("x war negativ");
}
```

- es wird immer genau eine der beiden Anweisungen ausgeführt
  - niemals beide
  - niemals keine

# Geschachtelte Bedingte Anweisung



- if-Anweisung ist selbst eine Anweisung
  - kann daher einer anderen untergeordnet werden
  - geschachtelte if-Anweisungen

# Beispiel



- Berechnung des Quartals aufgrund der Monatszahl

```
if ( month <= 3) {
       quarter = 1;
} else {
   if ( month <= 6 ) {
      quarter = 2;
   } else {
      if ( month <= 9 ) {
         quarter = 3;
      } else {
         quarter = 4;
      } // month <= 9
   } // month <= 6
} // month <= 3
```

# Best Practices



- hohe Schachtelungstiefe vermeiden !
  - unübersichtlich!
  - es stehen noch andere Konstrukte zur Verfügung
- Alternative und Konsequente immer als Block in geschweifte Klammern setzen!
  - erzeugt Klarheit
  - hilft, falls später Anweisungen hinzukommen
    - z.B. Ausgabeanweisungen zum Testen
  - Ausnahme/Sonderfall: Alternative ist eine bedingte Anweisung
    - dann für bessere Lesbarkeit: auf geschweifte Klammern verzichten

# Beispiel



- Berechnung des Quartals aufgrund der Monatszahl

```
if ( month <= 3 ) {
   quarter = 1;
   System.out.println("1. Quartal!");
} else if ( month <= 6 ) {
   quarter = 2;
   System.out.println(null, "2. Quartal!");
} else if ( month <= 9 ) {
   quarter = 3;
   System.out.println(null, "3. Quartal!");
} else {
   quarter = 4;
   System.out.println(null, "4. Quartal!");
}
```

# Dreistelliger Bedingter Operator



- ähnlich `if-then-else`, wertet nur einen von zwei Ausdrücken aus
- Syntax
  - `<Bedingung> ? <Konsequente> : <Alternative>`
- Ablauf
  - Bedingung auswerten
  - falls wahr: Ja-Ausdruck auswerten und zurückliefern
  - falls falsch: Nein-Ausdruck auswerten und zurückliefern

# Dreistelliger Bedingter Operator



- Beispiele:

```
int a = …;
int b = (a == 0)? 1 : 2;
System.out.println(b != 1? "ungleich 1": "gleich 1");
```

- Problem:
  - leicht unübersichtlich
  - daher: nur mit sehr einfachen (kurzen) Ausdrücken verwenden
  - im Zweifelsfall vermeiden!

# Übung: Gerade/Ungerade Zahl



- Gegeben ist eine Variable:

```
int zahl;
```

- Schreiben Sie eine Anweisung, die entweder "gerade" oder "ungerade" auf der Konsole ausgibt, je nachdem, ob der Wert von `zahl` gerade oder ungerade ist. Verwenden Sie den Dreistelligen Bedingten Operator.

# Umfangreiche if-else-Kaskaden



- Fallunterscheidungen mit vielen Fällen werden unübersichtlich
  - oft je ein ... else if ... für jeden Fall
- Java bietet Konstrukt, das den Code übersichtlicher macht

# switch-Anweisungen



- Syntax
  - `switch`-Anweisungen ersetzen längere if-Kaskaden

```
switch (<Ausdruck>) {
     case <Konstante>:
              [<Anweisung>]
              [...]
              [break;]
     case <Konstante>:
              [<Anweisung>]
              [...]
              [break;]
     ...
     [default:]
              [<Anweisung>]
              [...]
}
```

# switch-Anweisungen



- Ablauf (Semantik)
  - der Wert des Ausdrucks wird einmal berechnet
  - das Ergebnis des Ausdrucks wird nacheinander mit den `case`-Konstanten ("Labels", "Sprungmarken") verglichen
    - dies können beliebig viele sein
  - `case`-Konstante mit dem ersten übereinstimmenden Wert:
    - folgende Anweisungen werden ausgeführt
    - dies können beliebig viele sein
  - eine `break`-Anweisung beendet die `switch`-Anweisung sofort
    - wie bei Schleifen
- `break` steht üblicherweise am Ende einer `case`-Anweisungsfolge
  - ohne `break` werden die Anweisungen des nächsten `case` ebenfalls ausgeführt

# switch-Anweisungen



Berechnung der Anzahl Tage im Monat:

```
switch(month){
    case 1:
        days = 31;
        break;
    case 2:
        days = 28;
        break;
    case 3:
        days = 31;
        break;
    …
    case 12:
        days = 31;
}
```

# case-Konstanten



- `case`-Konstanten müssen eindeutig sein, doppelte Werte unzulässig
- Wenn keine `case`-Konstante passt, geschieht nichts
  - dann: ganzes `switch` wirkt wie eine leere Anweisung
- leere `case`-Anweisungsfolgen sind zulässig
- Beispiel:

```
switch(month){
        case 1: case 3: case 5:  case 7:  case 8: case 10: case 12:
                days = 31;
                break;
        case 2:
                days = 28;
                break;
        case 4: case 6: case 9: case 11:
                days = 30;
                break;
}
```

# Default-Fall



- `default` = Standardfall
  - hier: spezielle `case`-Konstante, passt auf alle übrigen Werte
- `default` darf nur einmal und nur am Ende genannt werden

```
switch(month){
        case 2:
                days = 28;
                break;
        case 4: case 6: case 9: case 11:
                days = 30;
                break;
        default:  // all remaining months
                days = 31;
}
```

# Zulässige Datentypen



- Ergebnistyp des Ausdrucks im Kopf der `switch`-Anweisung und der Typ der `case`-Konstanten müssen übereinstimmen
- zulässige Typen :
  - einfache ganzzahlige Typen (`byte, short, int, char`)
  - Aufzählungstypen
  - Strings
- nicht zulässig (u.a.):
  - `float, double`: Test von exakten Werten problematisch ⇒ Rundungsfehler
  - `boolean`: nur zwei Werte

# Übung: Switch



- Schreiben Sie eine Switch-Anweisung, die für eine Variable

  int zahl;

- folgende Ausgaben auf der Konsole generiert:
  - falls zahl == 1 → "Eins"
  - falls zahl == 2 → "Zwei"
  - ansonsten → "Alles andere"

# Syntaktischer Zucker



- Switch ist ein Beispiel für Syntaktischen Zucker
  - keine neue Funktionalität, erweitert nicht die Sprache
  - aber: Code wird einfacher/lesbarer/übersichtlicher

**Syntaktischer Zucker** sind Syntaxerweiterungen in Programmiersprachen, welche der Vereinfachung von Schreibweisen dienen. Diese Erweiterungen sind alternative Schreibweisen, die aber nicht die Ausdrucksstärke und Funktionalität der Programmiersprache erweitern. (Wikipedia)

# Debugging

# Fehlersuche



- Situation
  - Fehler gefunden
  - 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, ...*

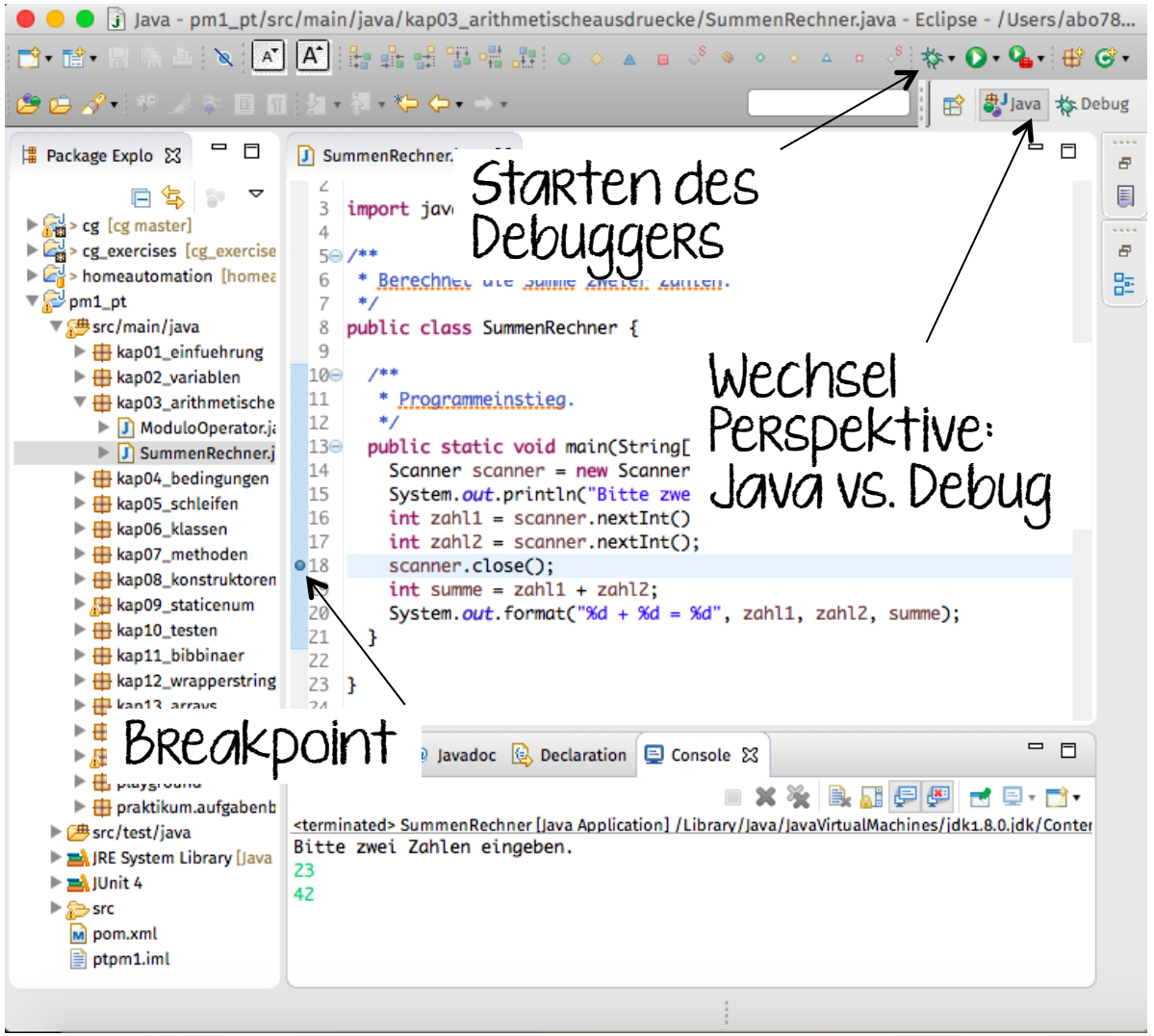
# Eclipse-Debugger



- Eclipse bringt Debugger-Integration mit

# Eclipse-Debugger

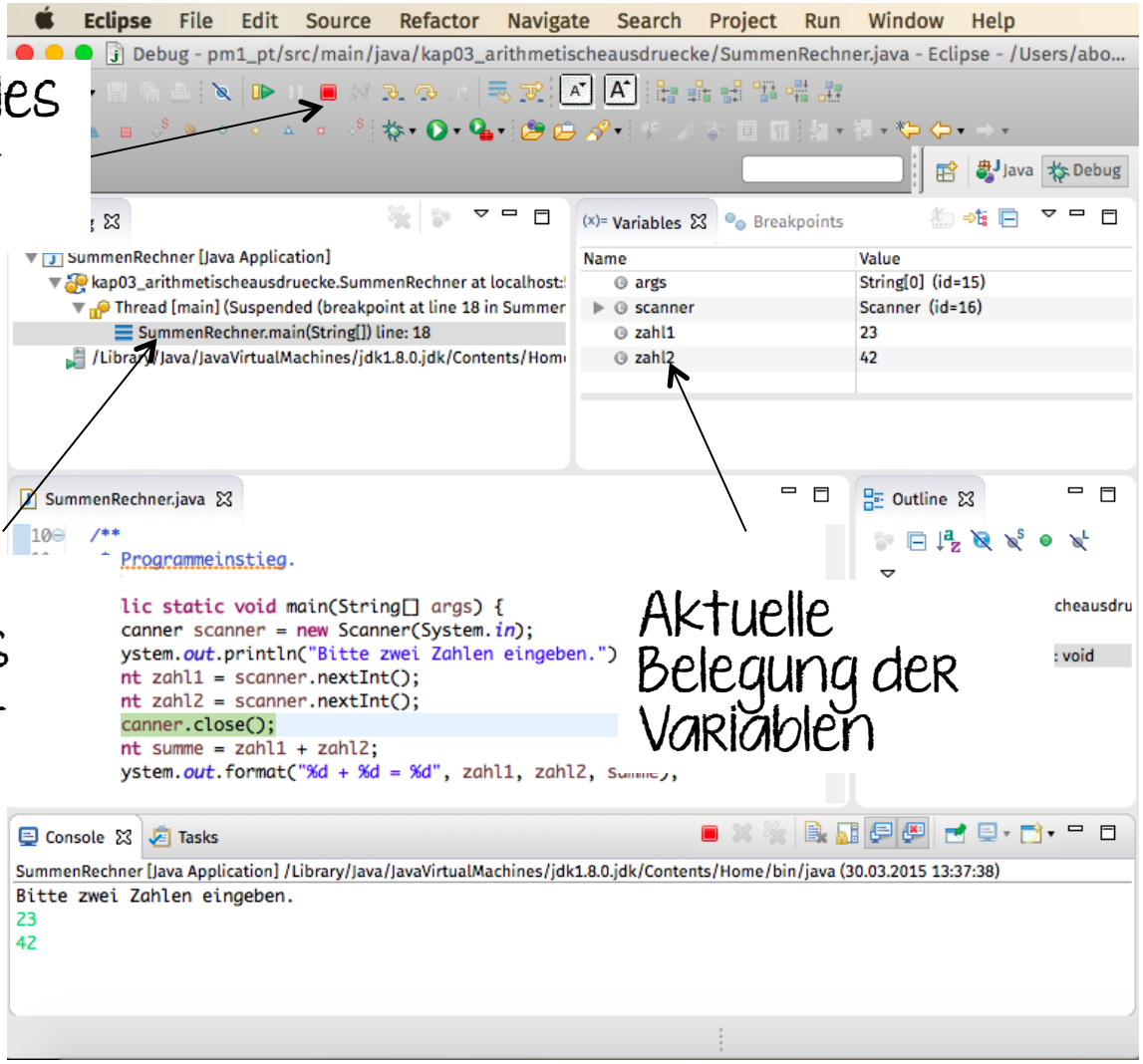




# Eclipse-Debugger



Steuerung des Programm-ablaufs



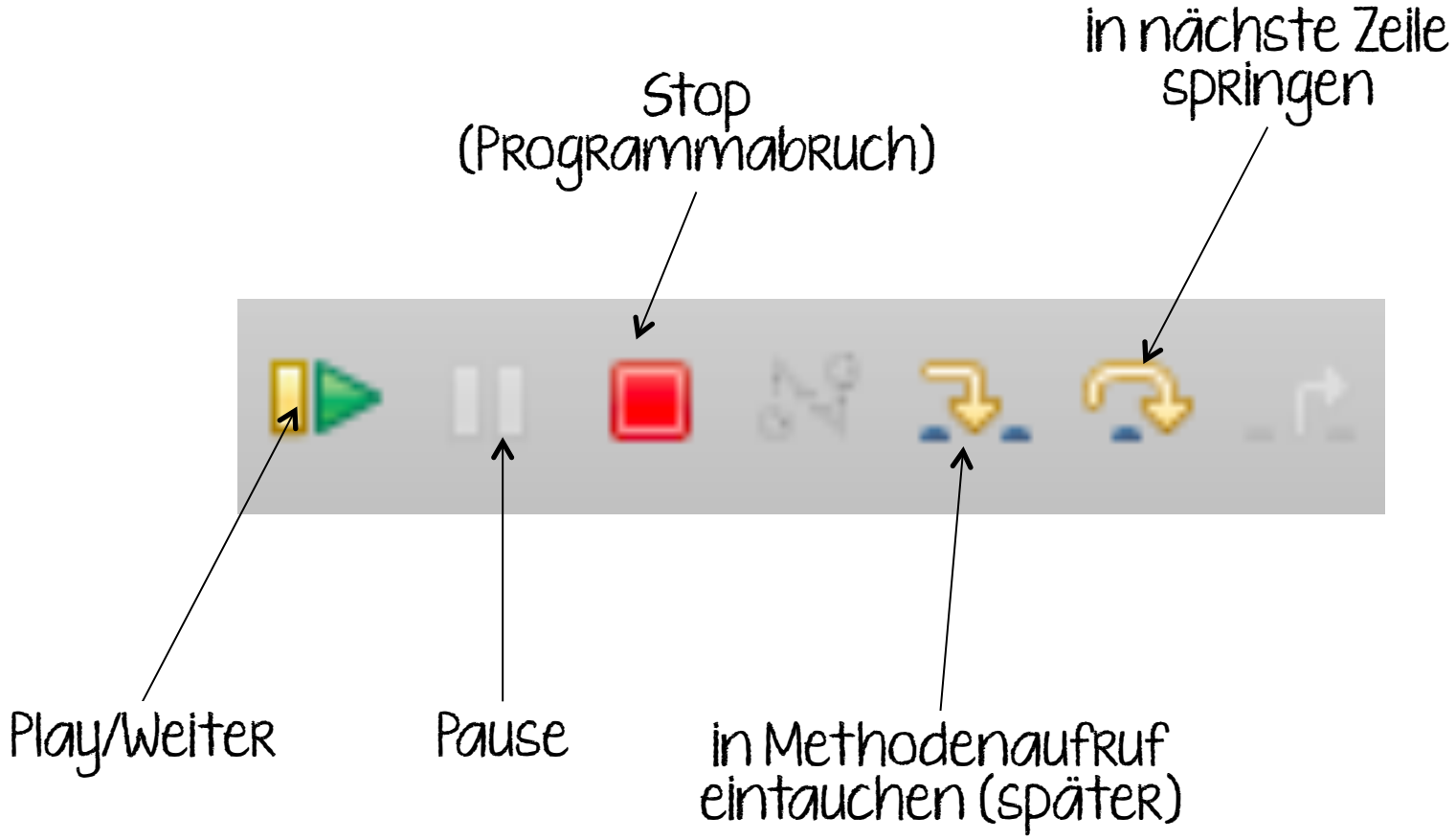

Aktueller Zustand des Programm-ablaufs

Aktuelle Belegung der Variablen

# Eclipse-Debugger



- Steuerung des Programmablaufs



# Zusammenfassung



- Wahrheitswerte
- Bedingte Anweisungen
- Fehlersuche/Debugger