Methoden

Einführung in die Programmierung 1

Sommersemester 21

TU Wien

Überblick

- Unterprogramme allgemein
- Methoden in Java
- Sichtbarkeit und Lebensdauer von Variablen
- Vor- und Nachbedingungen
- Überladen von Methoden
- Beispiele aus der Java-API

Unterprogramme allgemein

Motivation

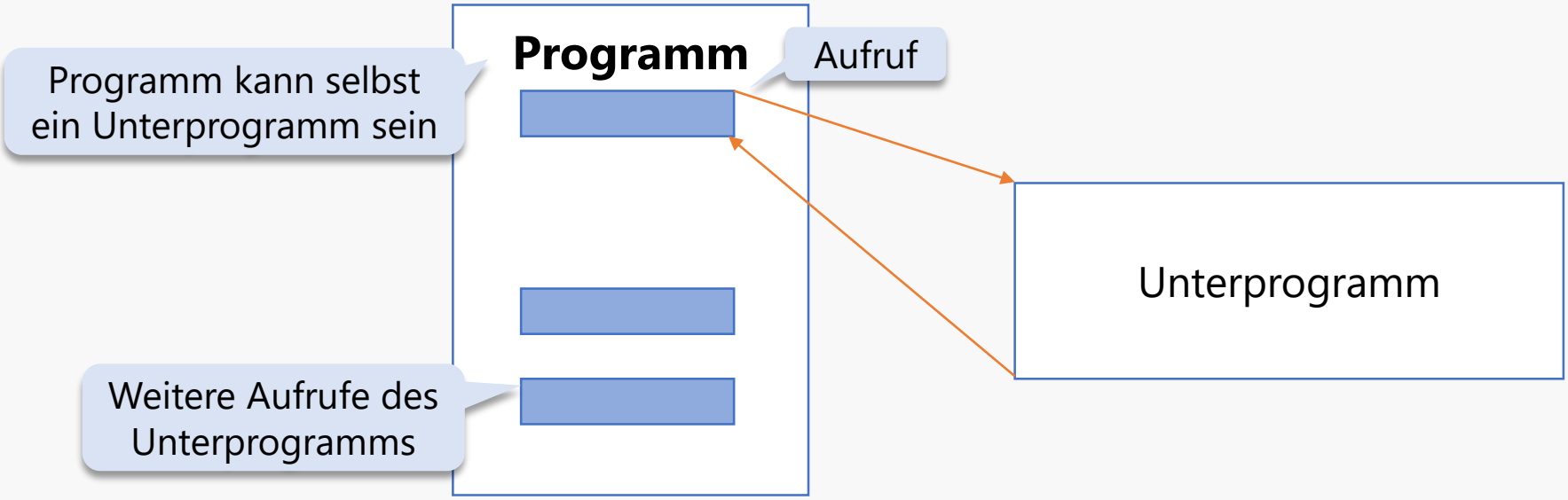
- Bisher gesamter Programmcode in der main-Methode
  - Bei größeren Programmen wird das unübersichtlich und umständlich
  - Einige Teile werden sehr ähnlich sein und immer wieder im Programm vorkommen
- Programm sollte aufgeteilt werden
  - Übersichtlicher
  - Verständlicher

Unterprogramm

- Teil eines Programms
  - Ist eine **isolierbare**, oft an **mehreren Stellen** im Programm **identisch** auszuführende Tätigkeit
  - Wird durch einen Bezeichner identifiziert
- Weiterer Mechanismus zum Steuern des Programmablaufs
  - Neben Verzweigungen und Schleifen

Aufruf eines Unterprogramms

- Sprung zum Unterprogramm
- Abarbeitung des Unterprogramms
- Nach der Abarbeitung Rückkehr zur Aufrufstelle



Vorteile von Unterprogrammen

Wiederverwendung

- Einmal Code schreiben
- Mehrfach an vielen Stellen mit unterschiedlichen Argumenten verwenden

Abstraktion

- Ablauf muss nicht genau bekannt sein
- Für den Aufruf nur wichtig
  - Welcher Input (Argumente) kann übergeben werden?
  - Welche Auswirkung hat der Aufruf (Output, Rückgabewert)?

Unterprogramme in Java

- Werden als Methoden bezeichnet
- Werden in einer Klasse implementiert
  - Sind kein eigenständiger Teil (wie z. B. in C)

Methoden in Java

Methoden in Java

Statische Methoden

- Gehören zur Klasse
- Werden in diesem Foliensatz besprochen

Objektbezogene Methoden

- Gehören zu einem Objekt der Klasse
- Werden bei der Objektorientierung (EP2) besprochen

Aufbau von Methoden

Methodenkopf

Modifikatoren, Rückgabetyp, Methodenname und eventuell Parameter

Methodenrumpf

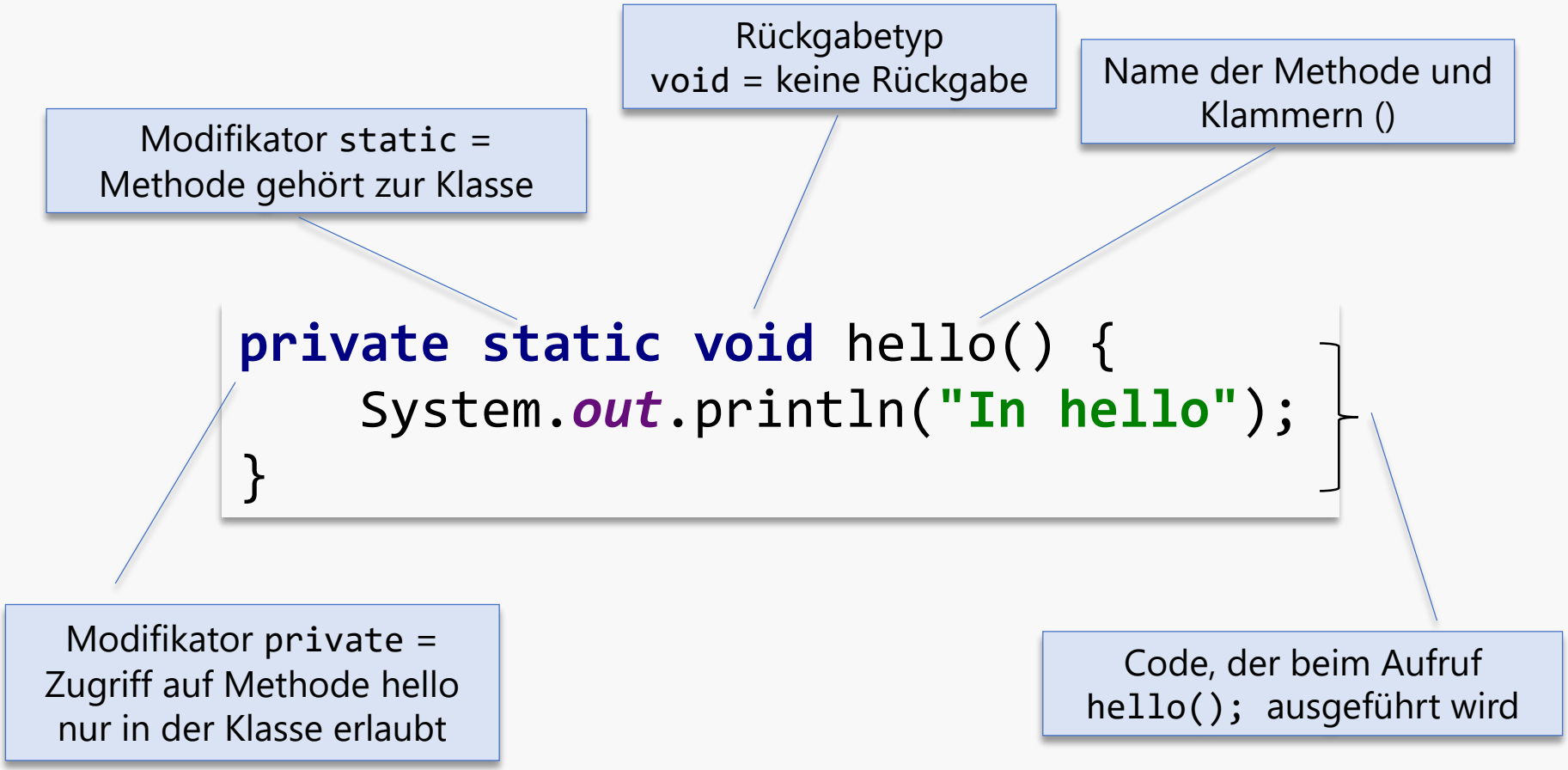
Enthält Variablendeklarationen und Anweisungen

Beispiel (einfache Methode)

```
public class MethodTest1 {

    private static void hello() {
        System.out.println("In hello");
    }

    public static void main(String[] args) {
        System.out.println("In main");
        hello();
        System.out.println("In main");
        hello();
        System.out.println("In main");
    }

}
```

```
In main
In hello
In main
In hello
In main
```

Anatomie einer statischen Methode



Beispiel (einfache Methode)

```
public class MethodTest2 {

    private static void printLine() {
        for (int i = 0; i < 15; i++) {
            System.out.print("-");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        printLine();
        System.out.println("Hello World!");
        printLine();
        System.out.println("Hello EP1!");
        printLine();
    }

}
```

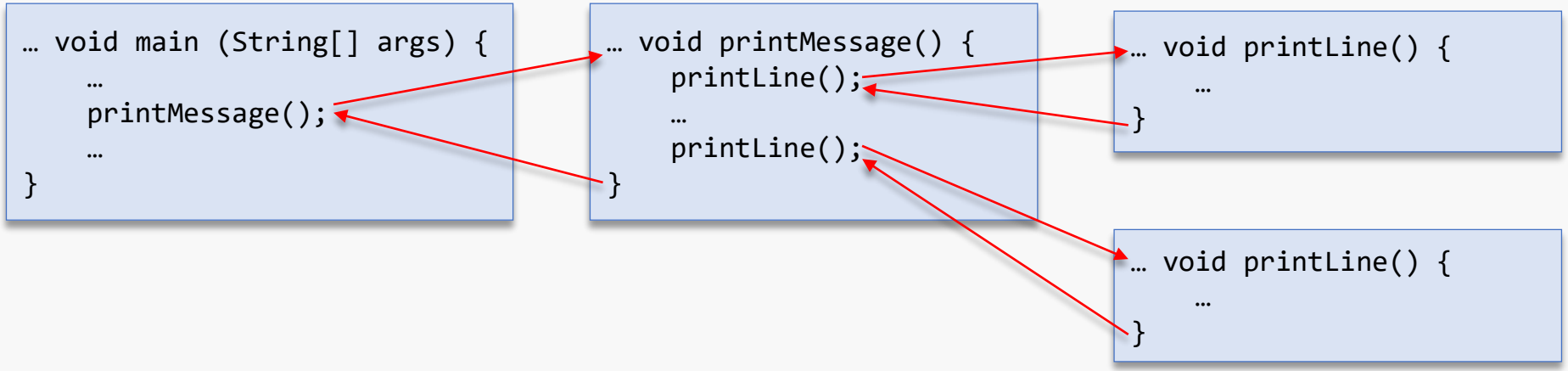
```
---------------
Hello World!
---------------
Hello EP1!
---------------
```

Beispiel (mehrere einfache Methoden)

```java
public class MethodTest3 {

    private static void printLine() {
        for (int i = 0; i < 15; i++) {
            System.out.print("-");
        }
        System.out.println();
    }

    private static void printMessage() {
        printLine();
        System.out.println("Hello World!");
        printLine();
    }

    public static void main(String[] args) {
        System.out.println("In main");
        printMessage();
        System.out.println("In main");
    }

}
```

```
In main
---------------
Hello World!
---------------
In main
```

Beispiel (Ablauf)



Vergleich (alles in main, mit Methoden)

```java
public class TestWithoutMethods {

    public static void main(String[] args) {
        System.out.println("In main");
        for (int i = 0; i < 15; i++) {
            System.out.print("-");
        }
        System.out.println();
        System.out.println("Hello World!");
        for (int i = 0; i < 15; i++) {
            System.out.print("-");
        }
        System.out.println();
        System.out.println("In main");
    }

}
```

Duplizierter Code

```java
public class MethodTest3 {

    private static void printLine() {
        for (int i = 0; i < 15; i++) {
            System.out.print("-");
        }
        System.out.println();
    }

    private static void printMessage() {
        printLine();
        System.out.println("Hello World!");
        printLine();
    }

    public static void main(String[] args) {
        System.out.println("In main");
        printMessage();
        System.out.println("In main");
    }

}
```

Methodenname

- Name
  - Sollte ausdrücken, was die Methode leistet
  - Kann mit einem Verb beginnen
    - Abhängig von der Aufgabe
  - Wird üblicherweise klein geschrieben
  - Falls der Name aus mehreren Worten besteht, sollten die Folgeworte groß geschrieben werden (Konvention)
- Beispiele

```
random, abs, max, min, sin, cos, sqrt ...
printHeader, createTable ...
```

Parameter

- Werte, die beim Aufruf an eine Methode übergeben werden
- Wir unterscheiden

Formale Parameter	• Werden im Methodenkopf angegeben • Werden innerhalb der Methode wie Variablen behandelt
Aktuelle Parameter (Argumente)	• Beim Aufruf der Methode wird ein formaler Parameter durch je einen aktuellen Parameter initialisiert

Beispiel (einfache Methode mit Parameter)

```
public class MethodTest4 {

    private static void printLine(int length) {
        for (int i = 0; i < length; i++) {
            System.out.print("-");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        printLine(10);
        System.out.println("Hello World!");
        printLine(15);
        System.out.println("Hello EP1!");
        printLine(20);
    }

}
```

Formaler Parameter length

Aktueller Parameter 10

```
----------
Hello World!
---------------
Hello EP1!
--------------------
```

Beispiel (Methode mit mehreren Parametern)

```
public class ParameterTest {

    private static void printMax(int a, int b) {
        if (a > b) {
            System.out.println(a);
        } else {
            System.out.println(b);
        }
    }

    public static void main(String[] args) {
        int x = 10;
        short y = 20;
        printMax(x, y);
        printMax(2 * 7, 3 + 4 + 5);
        // printMax(10L,10);
    }

}
```

Funktioniert nicht, da aktueller Parameter vom Typ long übergeben wird

```
20
14
```

Aufruf mit aktuellen Parametern

- Ein formaler Parameter wird als lokale Variable angelegt
- Der Wert des aktuellen Parameters wird berechnet
  - Kann auch ein beliebiger Ausdruck sein, der zuerst ausgewertet wird
- Der Wert wird in den formalen Parameter **kopiert**
  - Es wird implizit eine Zuweisung durchgeführt
  - Wird als **Call-by-value** bezeichnet

Implizite Typkonvertierung

- Der Typ eines aktuellen Parameters muss gleich oder kleiner als der Typ des formalen Parameters sein
- Beim Aufruf erfolgt, falls benötigt, eine **implizite Typkonvertierung**
  - Details siehe Java-Spezifikation

Rückgabe von Werten

- Typ des Rückgabewerts steht vor dem Methodennamen
  - `void`, wenn „Nichts“ zurückgeliefert wird
    - Methode produziert z. B. nur eine Ausgabe (bisherige Beispiele)
- Ergebnis wird durch eine `return`-Anweisung zurückgegeben
  - Kann in einem Ausdruck verwendet werden

Beispiel (Methode mit Rückgabewert)

```
public class MethodReturnTest1 {

    private static int addIntegers(int a, int b) {
        int sum;
        sum = a + b;
        return sum;
    }

    public static void main(String[] args) {
        int x, y;
        int sum;
        sum = addIntegers(10, 20);
        System.out.println("s = " + sum);
        x = sum + 10;
        y = sum + sum;
        System.out.println("x = " + x + ", y = " + y);
        sum = addIntegers(x, y) + y;
        System.out.println("s = " + sum);
    }

}
```

Methode mit zwei Parametern **a** und **b** vom Typ **int** und Rückgabetyp **int**

Die lokale Variable **sum** ist nur in **addIntegers** sichtbar

Rückgabe des Inhalts von **sum**

sum unterscheidet sich von **sum** in **addIntegers**

```
s = 30
x = 40, y = 60
s = 160
```

Beispiel (alternative Variante)

```
public class MethodeReturnTest1 {

    private static int addIntegers(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int x, y, sum;
        sum = addIntegers(10, 20);
        System.out.println("s = " + sum);
        x = sum + 10;
        y = sum + sum;
        System.out.println("x = " + x + ", y = " + y);
        sum = addIntegers(x, y) + y;
        System.out.println("s = " + sum);
    }

}
```

Kürzere Variante von **addIntegers**

main liefert nichts zurück – daher void

```
s = 30
x = 40, y = 60
s = 160
```

Rückgabewert

- Sollte (muss aber nicht) verwendet werden
  - Rückgabewert einer Variable zuweisen
    - Z. B. `sum = addIntegers(10, 20);`
  - Rückgabewert in einem Ausdruck verwenden
    - Z. B. `System.out.println(addIntegers(10, 20) * 2);`
- Methodenaufruf kann auch nur als reine Ausdrucksanweisung (durch Anhängen eines Strichpunkts) vorkommen
  - Methoden, die keinen Rückgabewert (`void`) haben, können nur als Ausdrucksanweisung angeschrieben werden

return-Anweisung

- Mehrere `return`-Anweisungen sind möglich
  - Verkürzen Methoden
  - Aber mehrere Ausstiegspunkte aus der Methode
    - Debugging umständlicher
- Leere `return`-Anweisung
  - `return;`
  - Ermöglicht vorzeitiges Beenden einer void-Methode

Beispiel (mehrere return-Anweisungen)

```
public class PrimeTest {

    private static boolean isPrime(int number) {
        if (number < 2) {
            return false;
        }
        for (int i = 2; i * i <= number; i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isPrime(1));
        System.out.println(isPrime(4));
        System.out.println(isPrime(31));
        System.out.println(isPrime(99991));
    }

}
```

```
false
false
true
true
```

Beispiel (nur eine return-Anweisung)

```
public class PrimeTest2 {

    private static boolean isPrime(int number) {
        boolean flag = true;
        if (number < 2) {
            flag = false;
        }
        for (int i = 2; i * i <= number && flag; i++) {
            if (number % i == 0) {
                flag = false;
            }
        }
        return flag;
    }

    public static void main(String[] args) {
        System.out.println(isPrime(1));
        System.out.println(isPrime(4));
        System.out.println(isPrime(31));
        System.out.println(isPrime(99991));
    }

}
```

```
false
false
true
true
```

Beispiel (ggT-Berechnung)

```
public class GCDWithMethods {

    private static int gcd(int a, int b) {
        int first = a;
        int second = b;
        if (first == 0) {
            return second;
        } else {
            while (second != 0) {
                if (first > second) {
                    first -= second;
                } else {
                    second -= first;
                }
            }
        }
        return first;
    }

    public static void main(String[] args) {
        System.out.println(gcd(132, 1908));
        System.out.println(gcd(2640, 150000));
    }
}
```

Klassischer Euklidischer Algorithmus als Methode mit zwei Parametern.
Siehe auch: https://de.wikipedia.org/wiki/Euklidischer_Algorithmus

```
12
240
```

Beispiel (ggT-Berechnung)

```
public class GCDWithMethods {

    private static int gcd(int a, int b) {
        if (a == 0) return b;
        else
            while (b != 0)
                if (a > b) a -= b;
                else b -= a;
        return a;
    }

    public static void main(String[] args) {
        System.out.println(gcd(132, 1908));
        System.out.println(gcd(2640, 150000));
    }
}
```

Kürzere Variante:
- Parameter als Variablen benutzen und verändern (eigentlich kein guter Stil)
- Keine Klammern, da immer nur eine Anweisung pro Schachtelungstiefe

```
12
240
```

Sichtbarkeit und Lebensdauer von Variablen

Block (Wiederholung)

- Ist eine Programmeinheit, in der lokale Vereinbarungen getroffen werden können
- In Java beginnt ein Block mit **{** und endet mit **}**
- Beispiel

```
public static void main(String[] args) {
      int x = 2, y = x;
      {
          int z = y + 2;
          x = z;
      }
      System.out.println(x + " " + y);
}
```

Block für z

Block für x und y

Hier sind nur x und y bekannt
Ausgabe: 4 2

Lokale Variablen (Wiederholung)

- Methoden können auch lokale Deklarationen enthalten
- Beispiel

```
public static void m(int x) {
        int y;
        short s;
        …
}
```

- Deklariert **3 lokale** Variablen y, s und x
  - Dürfen nur in dieser Methode verwendet werden
  - x wird wie eine lokale Variable behandelt

Klassenvariablen

- Variablen außerhalb von Methoden deklarieren
- Beispiel

```
public class Test1 {
        private static int a, b;
        private static void m(int x){…}
        public static void main(String[] args) {…}
}
```

- a und b sind Klassenvariablen (statische Variablen)
  - Durch `static` gekennzeichnet (meist `private`)
  - Können in **allen** Methoden der Klasse benutzt und verändert werden (globale Variable für die Klasse)

Lokale und globale Variablen – Speicherplatz

- Lokale Variablen
  - Bei jedem neuen Aufruf der Methode wird Speicherplatz angelegt
  - Am Ende der Methode wird der Speicherplatz wieder freigegeben
- Globale Variablen
  - Speicherplatz wird beim Programmstart angelegt bzw. beim Programmende wieder freigegeben
  - Werte existieren über Methodenaufrufe hinweg

Beispiel (Aufsummieren mit globaler Variable)

```
public class CumulativeAdd {

    private static int sum = 0;

    private static void add(int x) {
        sum = sum + x;
    }

    public static void main(String[] arg) {
        add(1);
        add(2);
        add(3);
        System.out.println("sum = " + sum);
    }

}
```

Statische Variable

Aktueller Wert von sum wird verwendet

```
sum = 6
```

Lokale oder globale Variablen?

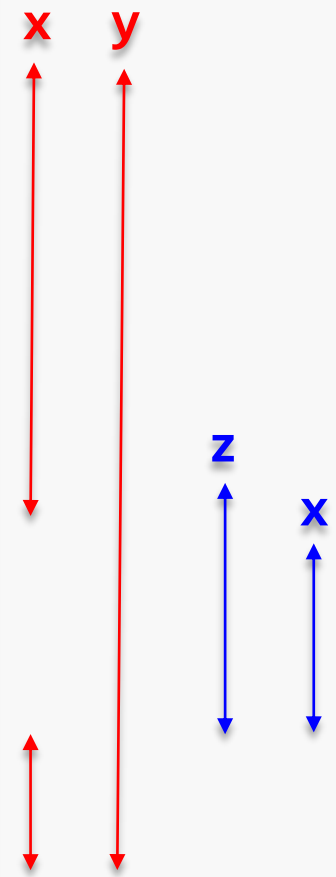
- Wenn möglich, Variablen **immer lokal** deklarieren
- Vorteile lokaler Variablen
  - Vereinfachen Wahl der Bezeichner
  - Fördern Lesbarkeit
    - Deklaration und Verwendung der Variablen sind nah beieinander
  - Fördern Verständlichkeit
    - Jede Methode ist für sich alleine überprüfbar
  - Fördern Sicherheit
    - Lokale Variablen können nicht durch andere Methoden verändert werden
    - Vermindern Seiteneffekte bei globalen Variablen

Sichtbarkeitsbereich (Sichtbarkeit)

- Jener Teil des Programms, in dem auf einen Bezeichner zugegriffen werden kann
- Regeln (vollständige Liste siehe Spezifikation)
  - Ein Bezeichner darf in einem Block nicht mehrmals deklariert werden (auch nicht in geschachtelten Anweisungsblöcken)
  - Lokale Bezeichner verdecken Bezeichner, die auf Klassenebene deklariert wurden
  - Der Sichtbarkeitsbereich eines lokalen Bezeichners beginnt bei seiner Deklaration und geht bis zum Ende der Methode bzw. bis zum Ende des umschließenden Blocks
  - Auf Klassenebene deklarierte Bezeichner sind in allen Methoden der Klasse sichtbar

Sichtbarkeit – Beispiel

```
public class Example {
    private static void m1() {
        ...
    }
    private static int x;
    private static int y;
    private static void m2(int z) {
        int x;
        ...
    }
    ...
}
```



Sichtbarkeit – größeres Beispiel

```
public class Example {
    private static void method1() {
        System.out.println(x);
    }
    private static int x = 0;
    public static void main(String[] arg) {
        System.out.println(x);
        int x = 1;
        System.out.println(x);
        method1();
        if (x > 0) {
            int x;
            int y;
            ...
        } else {
            int y;
            ...
        }
        for (int i = 0; ...) {...}
        for (int i = 1; ...) {...}
    }
}
```

Gibt 0 aus

Gibt 0 aus

Gibt 1 aus

Geht nicht - x wurde in main schon deklariert

Kein Konflikt mit y im if-Zweig

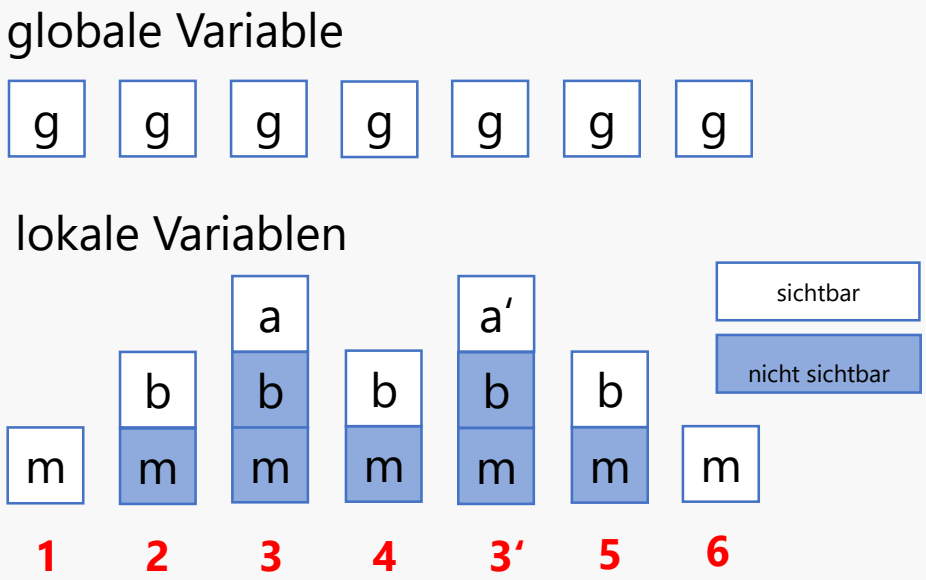
Kein Konflikt mit i aus vorheriger Schleife

Lebensdauer

- Lebensdauer ≠ Sichtbarkeit
  - Variable kann existieren, aber nicht sichtbar sein (z. B. globales x aus vorheriger Folie)
- Hängt davon ab, ob eine Variable lokal oder global deklariert wird
  - Global – Vom Programmanfang bis zum Programmende
  - Lokal – Abhängig vom Block

Lebensdauer – Beispiel

```
public class Tester {
    private static int g;

    public static void main(String[] args) {
        int m;
        1 … method2(); … 6
    }

    private static void method2() {
        int b;
        2 … method1(); … 4 … method1(); … 5
    }

    private static void method1() {
        int a;
        … 3 …
    }
}
```



Vor- und Nachbedingungen

Zusicherung (Assertion)

- Ist eine Aussage über den Zustand eines Programms an einer bestimmten Stelle
- Beispiel bei Verzweigung

```
if (x > y) {     // x > y
    max = x;
} else {         // !(x > y), d.h. x <= y
    max = y;
}
```

- Hinweise
  - Erste Zusicherung ist trivial (kann weggelassen werden)
  - Die zweite Zusicherung ist in diesem Fall wichtig (<=)

Zusicherung (komplexeres Beispiel)

- Beispiel

```
if (0 <= x && x < 10) {
    // ...
} else {
    // !(0 <= x && x < 10) wird umgeformt zu
    // !(0 <= x) || !(x < 10) oder besser
    // x < 0 || x >= 10
}
```

- Hinweis:
  - Umformung zusammengesetzter Ausdrücke mit Hilfe der Gesetze von De Morgan
    - `!(a && b)` ist äquivalent zu `!a || !b`
    - `!(a || b)` ist äquivalent zu `!a && !b`

Vorbedingungen (Preconditions)

- Geben an, unter welchen Voraussetzungen das Verhalten der Methode definiert ist (z. B. korrekte Parameterbereiche)
- Müssen beim Aufruf eingehalten werden

Nachbedingungen (Postconditions)

- Geben an, was nach der Ausführung der Methode (vor dem Zurückkehren zur Aufrufstelle) gelten muss
  - Rückgabewerte, Ausgaben etc.
- Gelten, falls Vorbedingungen eingehalten wurden
- Müssen von der Methode eingehalten werden

Zusicherungen bei Methoden

- Vorbedingungen
  - Können bei einer öffentlichen Methode **nicht** zugesichert werden
  - Zusicherung bei privaten Methoden möglich
- Nachbedingungen
  - Diese kann und muss die Methode garantieren
  - Gelten am Ende einer Methode
  - Zusicherungen zur Überprüfung verwenden

Beispiel

Vorbedingung (keine Zusicherung)
- Ist sie nicht erfüllt, dann wird kein korrektes Verhalten garantiert!

```
// a > 0 && b > 0
public static int gcd(int a, int b) {
    int first = a;
    int second = b;
    while (second != 0) {
        if (first > second) {
            first -= second;
        } else {
            second -= first;
        }
    }
    // first > 0 && a % first == 0 && b % first == 0
    return first;
}
```

Nachbedingungen (Zusicherungen)

Zusicherungen in Java

- Bis jetzt nur als Kommentare geschrieben
- Eigentlich sind das Aussagen, die immer wahr sein müssen
  - Das könnte auch in laufenden Programmen überprüft werden
- In Java gibt es dafür die `assert`-Anweisung

  https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

assert-Anweisung in Java

- Form

```
assert ausdruck;
assert ausdruck : ausdruck;
```

- Der erste Ausdruck ist ein boolescher Ausdruck
  - Muss zur Laufzeit `true` sein
  - Ist er `false`, dann bricht das Programm ab
    - Setzt voraus, dass Assertions überprüft werden
- Der zweite Ausdruck wird ausgewertet (wenn erster Ausdruck false) und mit einer Fehlermeldung ausgegeben
- Beispiele

```
assert a <= b && b <= c;
assert a >= 0 : "Negative a";
```

Arbeitsweise

- Assertions werden zur Laufzeit ausgewertet
- Wenn das Ergebnis `false` ist, stoppt das Programm mit einem `AssertionError`
  - Die nachfolgenden Anweisungen werden nicht mehr ausgeführt
- Bei Abbruch wird Information über den Ort der gescheiterten Assertion ausgegeben

Aktivierung

- Assertions kosten Rechenzeit
- Sie können zur Laufzeit wahlweise aktiviert werden
  - Sie sind automatisch deaktiviert
  - Programm muss nicht neu übersetzt werden
- Aktivierung in einem Programm Test

```
java –ea Test
```

- In IntelliJ
  - Unter `Run->Edit Configurations`
  - `-ea` im Feld `Vm Options` angeben
- Normalerweise während der Entwicklungszeit!

Beispiel – Assertions für Nachbedingungen

```
public class AssertionTest {

    public static int max(int x, int y) {
        int max = 0;
        if (x < y) {
            max = x;
        } else {
            max = y;
        }
        assert max >= x && max >= y && (max == x || max == y): "Maximum is not correct";
        return max;
    }

    public static void main(String[] args) {
        System.out.println(max(10, 20));
    }
}
```

Fehler in der Implementierung!

Ausgabe ohne Überprüfung der Assertion: 10
Ausgabe mit Überprüfung:

```
Exception in thread "main" java.lang.AssertionError: Maximum is not correct
        at AssertionTest.max(AssertionTest.java:9)
        at AssertionTest.main(AssertionTest.java:15)
```

Beispiel mit mehreren Assertions

```
public class AssertionTest2 {

    // a > 0 && b > 0
    public static int fastGcd(int a, int b) {
        int first = a, second = b;
        while (first != 0 && second != 0) {
            if (first >= second) {
                first = first % second;
            } else {
                second = second % first;
            }
        }
        int result = first + second;
        assert result > 0 : "gcd " + result + " is not > 0";
        assert a % result == 0 : "gcd " + result + " is not a divisor of " + a;
        assert b % result == 0 : "gcd " + result + " is not a divisor of " + b;
        return result;
    }

    public static void main(String[] args) {
        System.out.println(fastGcd(12, 8));
        System.out.println(fastGcd(3, 14));
    }

}
```

Überladen von Methoden

Überladen von Methoden

- Zwei Methoden dürfen den gleichen Namen haben
  - Typliste der formalen Parameter muss aber unterschiedlich sein (mindestens eine der folgenden Bedingungen)
    - Unterschiedliche Typen
    - Unterschiedliche Reihenfolge der Typen
    - Unterschiedliche Anzahl der Typen
- Der gleiche Name hat mehrere Bedeutungen
  - Die Methode wurde **überladen**
- Überladen auch bei Operator +
  - + hat unterschiedliche Bedeutung für int, double, String ….

Überladen – Unterscheiden von Methoden

- Unterscheiden mit Hilfe der Signatur
- Signatur einer Methode in Java
  - Name der Methode
  - Liste der Typen der formalen Parameter
    - Die Namen der Parameter spielen keine Rolle, nur die **Typen** und ihre **Reihenfolge**
  - Rückgabetyp gehört nicht dazu
- Achtung
  - Eingeschränkte Definition einer Signatur

Beispiel (Überladen von Methoden)

```java
public class OverloadingTest {
    private static void myPrint(int i) {
        System.out.println("Integer: " + i);
    }

    private static void myPrint(double i) {
        System.out.println("Double: " + i);
    }

    private static void myPrint(int i, int j) {
        System.out.println("Integer + Integer: " + i + " " + j);
    }

    private static void myPrint(int i, double j) {
        System.out.println("Integer + Double: " + i + " " + j);
    }

    private static void myPrint(double i, int j) {
        System.out.println("Double + Integer: " + i + " " + j);
    }

    public static void main(String[] args) {
        myPrint(10);
        myPrint(10D);
        myPrint(10, 'a');
        myPrint(10, 10D);
        myPrint(10F, 10);
    }
}
```

```
Integer: 10
Double: 10.0
Integer + Integer: 10 97
Integer + Double: 10 10.0
Double + Integer: 10.0 10
```

Praktische Überlegungen

- Überladen sollte mit Bedacht eingesetzt werden
  - Schwer lesbar, wenn auf die Details beim Auflösen der Argumente geachtet werden muss
- Einsatz
  - Methoden haben die gleiche Auswirkung
  - Unterschiedliche Parameteranzahl oder Typen erfordern unterschiedliche Methoden
  - Sinnvolles Beispiel `System.out.println()`
    - Methode bewirkt immer eine Ausgabe
    - Für unterschiedliche Typen werden unterschiedliche Ausgaben produziert

Weiteres Beispiel aus der Java-API

- Methode abs in der Klasse Math

```
…
public static int abs(int a) {
    return (a < 0) ? -a : a;
}
…
public static long abs(long a) {
    return (a < 0) ? -a : a;
}
…
public static float abs(float a) {
    return (a <= 0.0F) ? 0.0F - a : a;
}
…
public static double abs(double a) {
    return (a <= 0.0D) ? 0.0D - a : a;
}
…
```

Beispiele aus der Java-API

Vordefinierte statische Methoden

- Java bietet etliche statische Methoden (Klassenmethoden) in speziellen Klassen an
  - Gehören nur zur Klasse, nie zu einem Objekt
- Typischer Aufruf
  - `ClassName.MethodName(Parameter)`
  - Beispiel

    ```
    Math.sqrt(10.0);
    ```
- Manchmal auch in der Form
  - `ClassName.ObjectName.MethodName(Parameter)`
  - Beispiel

    ```
    System.out.println(x);
    ```

Beispiele (1)

- Beispiele für Klasse **`System`**
  - `System.currentTimeMillis()`
  - Statische Variable `out`
    - `System.out.println()`
- Beispiele für Klasse **`Math`**
  - `Math.`*`random`*`()`
  - `Math.`*`sqrt`*`(10.0)`
  - …

Beispiele (2)

- **`Byte, Short, Integer, Float, Double, Character`**
  - Klassen für primitive Datentypen
    - Methoden, Konstanten
  - Beispiele für Klasse Integer
    - `Integer.max(2,3)`
    - `Integer.`*`parseInt`*`(…)`
    - …

Importieren

- Bestimmte Klassen werden automatisch geladen
  - Z. B. Integer, Math, System
- Andere Klassen müssen importiert werden
- 2 Arten
  - Expliziter Import (am Anfang der Datei)

```
import java.util.Scanner;
import java.util.*;
```

Alle Klassen aus dem Paket `java.util`

  - Impliziter Import (bei Verwendung)

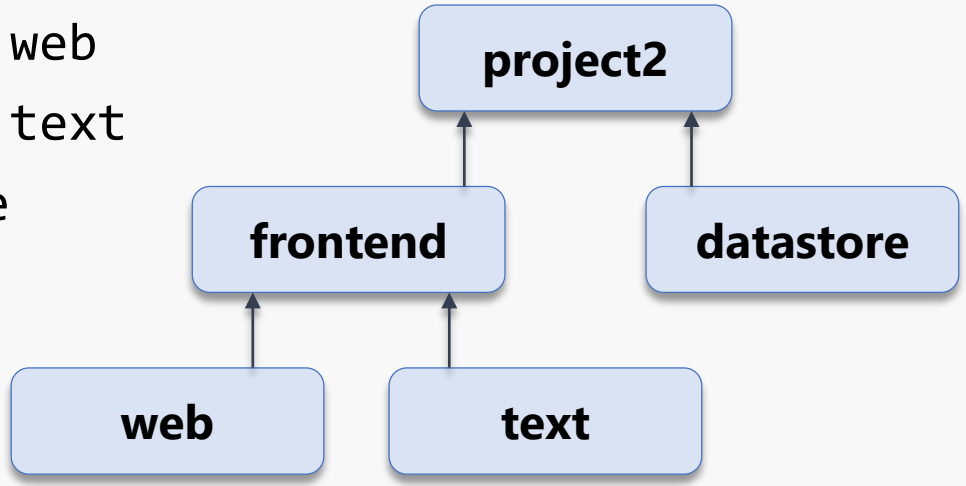
```
java.util.Arrays.sort(x);
```

Pakete – Hinweise

- Große Programme beinhalten viele Klassen
  - Probleme: Orientierung, Namenskonflikte, …
- Paket
  - Sammlung mehrerer logisch zusammengehörender Klassen
- Regeln für Pakete
  - In einem Paket sollte eine Klasse eindeutig benannt werden
  - In unterschiedlichen Paketen kollidieren nicht gleiche Namen
  - Pakete werden benannt (wie Variablen etc.)
  - Pakete können geschachtelt werden
    - Es gibt immer höchstens ein Super-Package
    - Pakete bilden eine Baumstruktur

Beispiel (Pakete)

- Beispiel für Pakete
  - `project2`
  - `project2.frontend`
  - `project2.frontend.web`
  - `project2.frontend.text`
  - `project2.datastore`



Module

- Seit Java 9
- Weitere Stufe über Paketen
- Ermöglicht bessere Modularisierung (Aufteilung größerer Programme)
- Mehr dazu in späteren Semestern

Dokumentation von Methoden

- Startpunkt (Java 15)
  - https://docs.oracle.com/en/java/javase/15/docs/api/index.html
- Informationen (Beispiele)
  - In welchem Modul und Paket finden wir die Klasse?
  - Für eine Klasse
    - Beschreibung der Klasse
    - Liste der Methoden
  - Für jede Methode
    - Kurzbeschreibung
    - Längere Beschreibung (Parameter, Ausnahmen etc.)

Zusammenfassung

- Unterprogramme
- Methoden in Java
- Sichtbarkeit und Lebensdauer von Variablen
- Vor- und Nachbedingungen
- Überladen
- Methoden in der Java-API