Einführung in die Programmierung

Vorlesungsteil 4
Methoden selbst programmieren und daraus Klassen bauen

PD Dr. Thomas Hinze

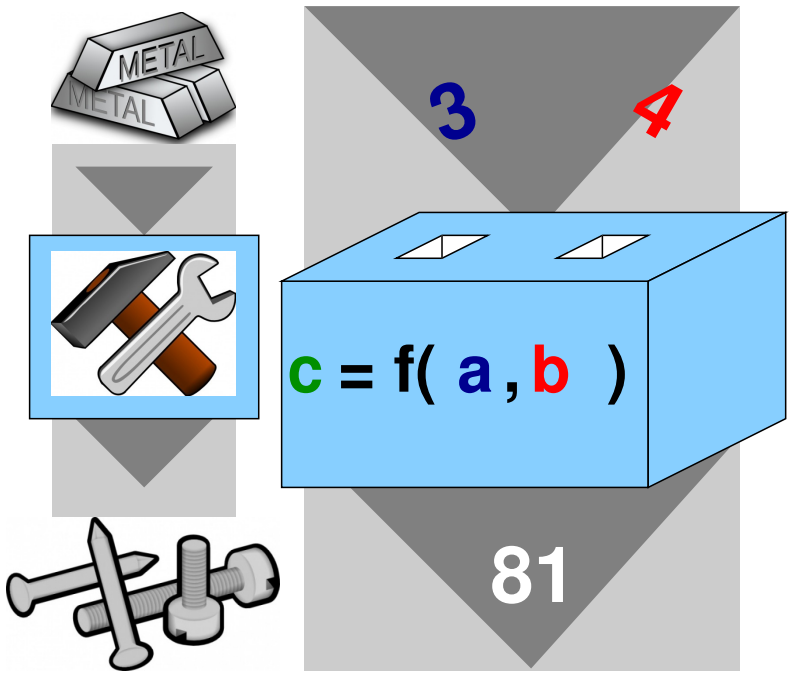
Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik

Sommersemester 2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Funktion als Werkzeug, das auf Daten einwirkt

Der Begriff **Funktion** stammt aus dem Lateinischen und bedeutet *„Tätigkeit“* oder *„Verrichtung“* im Sinne einer zielgerichteten Anwendung eines **Werkzeugs** auf **Werkstücke**. Mathematik und Informatik greifen den Begriff Funktion auf, wobei die Werkstücke dann Datenwerte sind.



Wofür sind Funktionen sinnvoll?

Beispiel: Eine Funktion zur Berechnung von Nullstellen quadratischer Polynome

$0 = x^2 + px + q$ Meine Parameter: $p = -6.1561, q = 8.03636$

Wofür sind Funktionen sinnvoll?

Beispiel: Eine Funktion zur Berechnung von Nullstellen quadratischer Polynome

$0 = x^2 + px + q$ Meine Parameter: $p = 3.9876, q = -15.353$

Wofür sind Funktionen sinnvoll?

Beispiel: Eine Funktion zur Berechnung von Nullstellen quadratischer Polynome

$0 = x^2 + px + q$ Meine Parameter: $p = -0.701, q = -2.2222$

Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.

Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.
- Das kann auch an unterschiedlichen Stellen innerhalb ein und desselben Programms der Fall sein, denken Sie zum Beispiel an einen Sortieralgorithmus, der in einer Tabellenkalkulation häufig benötigt wird.

Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.
- Das kann auch an unterschiedlichen Stellen innerhalb ein und desselben Programms der Fall sein, denken Sie zum Beispiel an einen Sortieralgorithmus, der in einer Tabellenkalkulation häufig benötigt wird.
- Es wäre als Programmierer unklug, dann jedes Mal „das Fahrrad neu zu erfinden“ und die Berechnungsvorschrift jedes Mal erneut gesondert in den Quelltext zu schreiben.

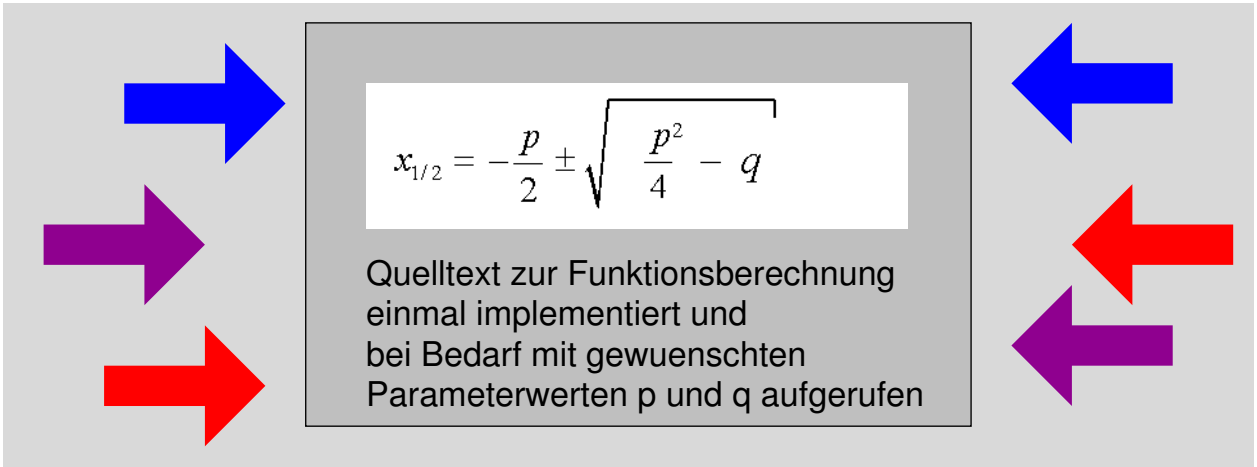
Wofür sind Funktionen sinnvoll?

- In ganz unterschiedlichen Zusammenhängen und für ganz verschiedene Anwendungen benötigt man *gleiche* oder ähnliche *Berechnungsvorschriften*, nur *mit jeweils anderen Eingabewerten*.
- Das kann auch an unterschiedlichen Stellen innerhalb ein und desselben Programms der Fall sein, denken Sie zum Beispiel an einen Sortieralgorithmus, der in einer Tabellenkalkulation häufig benötigt wird.
- Es wäre als Programmierer unklug, dann jedes Mal „das Fahrrad neu zu erfinden“ und die Berechnungsvorschrift jedes Mal erneut gesondert in den Quelltext zu schreiben.

$\Longrightarrow$ *Funktionen* ermöglichen es, eine Berechnungsvorschrift oder einen Algorithmus *einmal* zu *implementieren* und diese Implementierung *immer bei Bedarf* wieder *aufzurufen*.

Vorteile von Funktionen in der Programmierung

Wiederverwendbarkeit von Quelltext

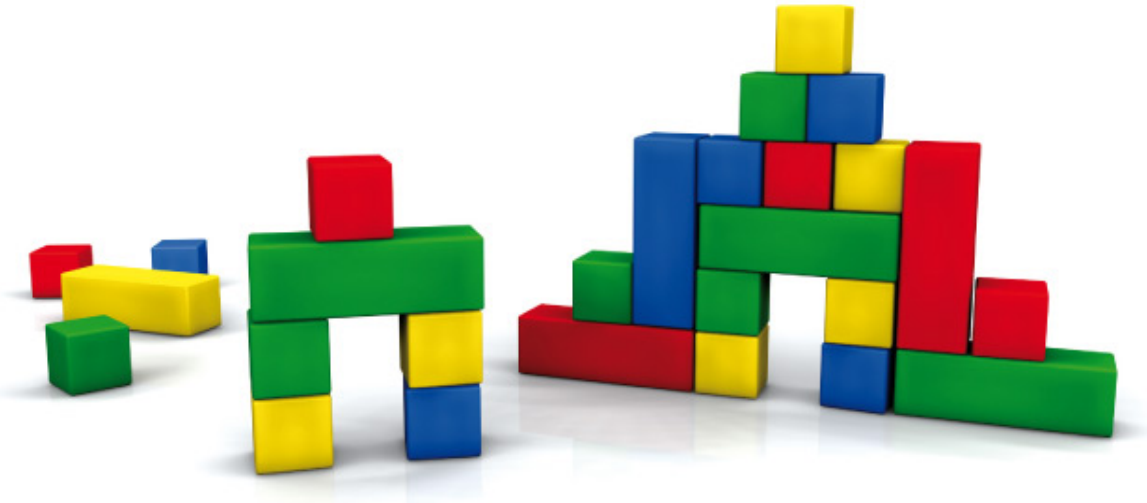


- führt zu deutlich kürzeren Quelltexten mit weniger Redundanz
- führt zu weniger Fehlern im Quelltext
- führt zu effizienterem Programmieren
  (mehr Features pro Zeiteinheit implementierbar)

$\Longrightarrow$ Funktionen als wiederverwendbare *Werkzeuge*

Vorteile von Funktionen in der Programmierung

Leichtere Wartbarkeit von Software



- Änderungen nur an einer Stelle im Quelltext vorzunehmen
- Implementierung einer Funktion unabhängig vom restlichen Quelltext austauschbar, Gesamtquelltext überschaubarer
- Funktion leicht ersetzbar durch schnellere, genauere oder einfach nur fehlerbereinigte Version

$\Longrightarrow$ Funktionen als anwendungsfertige *Bausteine*

Vorbild: Funktionen in der Mathematik

$$f_1 : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

$$f_1(p, q) = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$$

- Funktion besitzt einen klar zuordenbaren ***Namen***, z.B. f_1
- Funktion besitzt ***Argumente*** (Parameter), die bei Aufruf mit konkreten Werten aus Definitionsbereich belegt werden, z.B. p und q

Vorbild: Funktionen in der Mathematik

Aufruf:

$y = f_1(-6.1561, 8.03636)$

$$f_1 : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

$$f_1(p, q) = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$$

- Funktion besitzt einen klar zuordenbaren ***Namen***, z.B. f_1
- Funktion besitzt ***Argumente*** (Parameter), die bei Aufruf mit konkreten Werten aus Definitionsbereich belegt werden, z.B. p und q
- In der Reihenfolge der Argumente von links nach rechts werden die Werte zugeordnet, also z.B. $p = -6.1561$ und $q = 8.03636$
- Funktion berechnet den Funktionswert und gibt ihn als Berechnungsergebnis zurück

Vorbild: Funktionen in der Mathematik

Aufruf:
$y = f_1(-6.1561, 8.03636)$

Aufruf:
$z = f_1(\frac{\pi}{2} + 1, -2 \cdot 9.81)$

$$f_1 : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$$

$$f_1(p, q) = -\frac{p}{2} + \sqrt{\frac{p^2}{4} - q}$$

- Funktion besitzt einen klar zuordenbaren ***Namen***, z.B. f_1
- Funktion besitzt ***Argumente*** (Parameter), die bei Aufruf mit konkreten Werten aus Definitionsbereich belegt werden, z.B. p und q
- In der Reihenfolge der Argumente von links nach rechts werden die Werte zugeordnet, also z.B. $p = -6.1561$ und $q = 8.03636$
- Funktion berechnet den Funktionswert und gibt ihn als Berechnungsergebnis zurück
- Funktion kann erneut (beliebig oft) mit weiteren Werten aufgerufen werden

Vorbild: Funktionen in der Mathematik

$$g : \mathbb{R} \longrightarrow \mathbb{R} \qquad f : \mathbb{R} \longrightarrow \mathbb{R}$$
$$g(x) = \sin(x) \qquad f(x) = x^2$$

Aufruf: $z = f(g(\frac{\pi}{4})) = f(\sin(\frac{\pi}{4})) = \left(\sin(\frac{\pi}{4})\right)^2$

- Funktionsaufrufe dürfen ineinander verschachtelt werden (beliebig, aber endlich tief)

Vorbild: Funktionen in der Mathematik

$$g : \mathbb{R} \longrightarrow \mathbb{R} \qquad f : \mathbb{R} \longrightarrow \mathbb{R}$$
$$g(x) = \sin(x) \qquad f(x) = x^2$$

Aufruf: $z = f(g(\frac{\pi}{4})) = f(\sin(\frac{\pi}{4})) = \left(\sin(\frac{\pi}{4})\right)^2$

- Funktionsaufrufe dürfen ineinander verschachtelt werden (beliebig, aber endlich tief)
- Aus der Verschachtelung in Kombination mit Priorisierung der genutzten Operationen und Abarbeitung von links nach rechts bei gleichrangigen Operationen resultiert klare Reihenfolge, in der die Berechnungsschritte ausgeführt werden

Vorbild: Funktionen in der Mathematik

$$g : \mathbb{R} \longrightarrow \mathbb{R} \qquad f : \mathbb{R} \longrightarrow \mathbb{R}$$
$$g(x) = \sin(x) \qquad f(x) = x^2$$

Aufruf: $z = f(g(\frac{\pi}{4})) = f(\sin(\frac{\pi}{4})) = \left(\sin(\frac{\pi}{4})\right)^2$

- Funktionsaufrufe dürfen ineinander verschachtelt werden (beliebig, aber endlich tief)
- Aus der Verschachtelung in Kombination mit Priorisierung der genutzten Operationen und Abarbeitung von links nach rechts bei gleichrangigen Operationen resultiert klare Reihenfolge, in der die Berechnungsschritte ausgeführt werden
- Bei Funktionsaufruf spielt die Benennung der Argumente in der Funktionsdefinition keine Rolle, es ist also egal, ob die Funktion g definiert ist als $g(x) = \sin(x)$ oder $g(w) = \sin(w)$ oder z.B. $g(y) = \sin(y)$

Funktionen heißen in der objektorientierten Programmierung Methoden

In der objektorientierten Programmierung verallgemeinert man den Begriff **Funktion** noch etwas und nennt eine Funktion üblicherweise **Methode**, was für

„Verfahren zur Gewinnung von Erkenntnissen“

steht. Erkenntnisse sind hier eine Verallgemeinerung von Werkstücken oder Daten, die Funktionen liefern.

Vorlesung Einführung in die Programmierung mit Java

1. **Einführung und erste Schritte** . . . . . Installation Java-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . Java als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen, Schleifen und logische Ausdrücke programmieren
4. **Methoden selbst programmieren** . . . . . Methoden als wiederverwendbare Funktionen, Werteübernahme und -rückgabe
5. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
6. **Objektorientiert programmieren** . . . . . Klassen, Objekte, Attribute, Methoden, Sichtbarkeit, Vererbung, Polymorphie
7. **Felder und Graphen** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Netzwerken
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeichenketten, Dateiarbeit, Ausnahmen** . . . . . Texte analysieren, ver-/entschlüsseln, Dateien lesen/schreiben, Fehler behandeln
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Methodendefinition in Java: Syntaktische Struktur

Jede Methode wird *innerhalb einer Klasse* im Quelltext notiert:

```
<Modifizierer> <Ergebnistyp> <Name>(<getypte Argumentliste>)
{
   <Methodenrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Methodennamen.

Methodendefinition in Java: Syntaktische Struktur

Jede Methode wird *innerhalb einer Klasse* im Quelltext notiert:

```
<Modifizierer> <Ergebnistyp> <Name>(<getypte Argumentliste>)
{
   <Methodenrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Methodennamen.
- Die `<getypte Argumentliste>` enthält durch Kommas getrennt selbstgewählte Namen für die einzelnen Argumente, vor jedes Argument wird noch der entsprechende Typ gesetzt.

Methodendefinition in Java: Syntaktische Struktur

Jede Methode wird *innerhalb einer Klasse* im Quelltext notiert:

```
<Modifizierer> <Ergebnistyp> <Name>(<getypte Argumentliste>)
{
   <Methodenrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Methodennamen.
- Die `<getypte Argumentliste>` enthält durch Kommas getrennt selbstgewählte Namen für die einzelnen Argumente, vor jedes Argument wird noch der entsprechende Typ gesetzt.
- `{<Methodenrumpf>}` entspricht einem Block. Darin können Variablen oder Konstanten vereinbart werden, die nur innerhalb der Methode benötigt werden und deren Speicherplatz nach Abarbeitung des Methodenaufrufs wieder freigegeben wird.

Methodendefinition in Java: Syntaktische Struktur

Jede Methode wird *innerhalb einer Klasse* im Quelltext notiert:

```
<Modifizierer> <Ergebnistyp> <Name>(<getypte Argumentliste>)
{
   <Methodenrumpf>
}
```

- `<Name>` ist frei wählbarer Bezeichner für den Methodennamen.
- Die `<getypte Argumentliste>` enthält durch Kommas getrennt selbstgewählte Namen für die einzelnen Argumente, vor jedes Argument wird noch der entsprechende Typ gesetzt.
- `{<Methodenrumpf>}` entspricht einem Block. Darin können Variablen oder Konstanten vereinbart werden, die nur innerhalb der Methode benötigt werden und deren Speicherplatz nach Abarbeitung des Methodenaufrufs wieder freigegeben wird.
- `<Modifizierer>` sind Schlüsselwörter, mit denen der Zugriff auf die Methode (z.B. `public`) und/oder ein bestimmtes Verhalten (z.B. `static`) festgelegt wird.

Methode zur Berechnung des Rechteckumfangs

MeineRechtecksberechnung.java

```
public class MeineRechtecksberechnung {

  // Methode zur Berechnung eines Rechteckumfangs

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  // main-Methode ruft Berechnung mit konkreten Werten auf

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Der Umfang betraegt: %f\n", z);
  }
}
```

Bei Erreichen des Schlüsselwortes **`return`** Rückgabe des Funktionswertes

Merke

- Methoden definiert man in Java stets *innerhalb* von Klassen (d.h. innerhalb der Klassen-Begrenzungsklammern `{` und `}`).

Merke

- Methoden definiert man in Java stets *innerhalb* von Klassen (d.h. innerhalb der Klassen-Begrenzungsklammern `{` und `}`).
- Eine Klasse kann *beliebig viele* Methoden enthalten.

Merke

- Methoden definiert man in Java stets *innerhalb* von Klassen (d.h. innerhalb der Klassen-Begrenzungsklammern `{` und `}`).
- Eine Klasse kann *beliebig viele* Methoden enthalten.
- Die Methodendefinitionen einer Klasse stehen *untereinander*.

Merke

- Methoden definiert man in Java stets *innerhalb* von Klassen (d.h. innerhalb der Klassen-Begrenzungsklammern `{` und `}`).
- Eine Klasse kann *beliebig viele* Methoden enthalten.
- Die Methodendefinitionen einer Klasse stehen *untereinander*.
- Die Reihenfolge der Methodendefinitionen in der Klasse ist egal.

Merke

- Methoden definiert man in Java stets *innerhalb* von Klassen (d.h. innerhalb der Klassen-Begrenzungsklammern `{` und `}`).
- Eine Klasse kann *beliebig viele* Methoden enthalten.
- Die Methodendefinitionen einer Klasse stehen *untereinander*.
- Die Reihenfolge der Methodendefinitionen in der Klasse ist egal.
- Der **Kopf** jeder Methode enthält die *Modifizierer*, den *Ergebnistyp* (bzw. `void`, wenn nichts zurückgegeben wird), den *Methodennamen* und in runden Klammern die *getypte Argumentliste*. Mehrere Argumente durch Kommas trennen.

Merke

- Methoden definiert man in Java stets *innerhalb* von Klassen (d.h. innerhalb der Klassen-Begrenzungsklammern `{` und `}`).
- Eine Klasse kann *beliebig viele* Methoden enthalten.
- Die Methodendefinitionen einer Klasse stehen *untereinander*.
- Die Reihenfolge der Methodendefinitionen in der Klasse ist egal.
- Der **Kopf** jeder Methode enthält die *Modifizierer*, den *Ergebnistyp* (bzw. `void`, wenn nichts zurückgegeben wird), den *Methodennamen* und in runden Klammern die *getypte Argumentliste*. Mehrere Argumente durch Kommas trennen.
- Ist ein Ergebnistyp angegeben, der von `void` abweicht, muss jeder Abarbeitungspfad im *Methodenrumpf* mit einer `return`-Anweisung abschließen, hinter der ein (Variablen)wert vom Ergebnistyp steht. Dieser liefert das zurückgegebene Ergebnis (den Funktionswert).

Merke

- Methoden definiert man in Java stets *innerhalb* von Klassen (d.h. innerhalb der Klassen-Begrenzungsklammern `{` und `}`).
- Eine Klasse kann *beliebig viele* Methoden enthalten.
- Die Methodendefinitionen einer Klasse stehen *untereinander*.
- Die Reihenfolge der Methodendefinitionen in der Klasse ist egal.
- Der **Kopf** jeder Methode enthält die *Modifizierer*, den *Ergebnistyp* (bzw. `void`, wenn nichts zurückgegeben wird), den *Methodennamen* und in runden Klammern die *getypte Argumentliste*. Mehrere Argumente durch Kommas trennen.
- Ist ein Ergebnistyp angegeben, der von `void` abweicht, muss jeder Abarbeitungspfad im *Methodenrumpf* mit einer `return`-Anweisung abschließen, hinter der ein (Variablen)wert vom Ergebnistyp steht. Dieser liefert das zurückgegebene Ergebnis (den Funktionswert).
- Selbstdefinierte Methoden dürfen sich innerhalb derselben Klasse beliebig gegenseitig aufrufen (verschachtelte Aufrufe).

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
public static float f(float x) {...}
public static float g(float x) {...}
public static float h(float x) {...}
```

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
public static float f(float x) {...}
public static float g(float x) {...}
public static float h(float x) {...}
```

Dann ist der aufrufende Quelltext

```
float y = ...;
float a = f(y);
float b = g(a);
float c = h(b);
```

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
public static float f(float x) {...}
public static float g(float x) {...}
public static float h(float x) {...}
```

Dann ist der aufrufende Quelltext

```
float y = ...;
float a = f(y);
float b = g(a);
float c = h(b);
```

semantisch äquivalent zu

```
float y = ...;
float c = h(g(f(y)));
```

Verschachtelte Aufrufe vs. sequentielle Abarbeitung

$\Longrightarrow$ Aufrufschachtelung gleichwertig zu sequentieller Abarbeitung

Angenommen, es sind drei Funktionen `f`, `g` und `h` definiert:

```
public static float f(float x) {...}
public static float g(float x) {...}
public static float h(float x) {...}
```

Dann ist der aufrufende Quelltext

```
float y = ...;
float a = f(y);
float b = g(a);
float c = h(b);
```

semantisch äquivalent zu

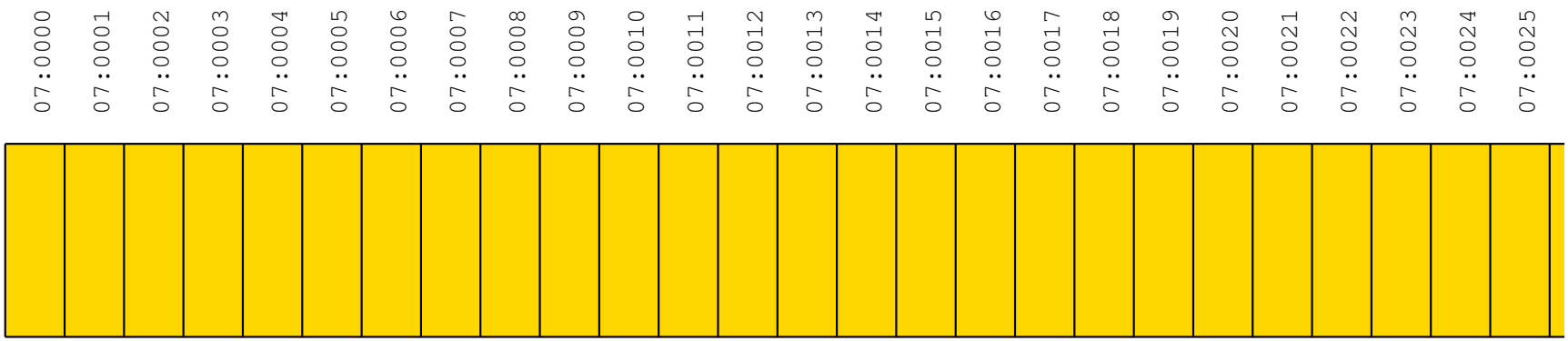
```
float y = ...;
float c = h(g(f(y)));
```

Merke: Die „Transfervariablen“ `a` und `b` brauchen bei Aufrufschachtelung nicht angelegt zu werden.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

Speicheradressen (fiktiver Adressbereich)



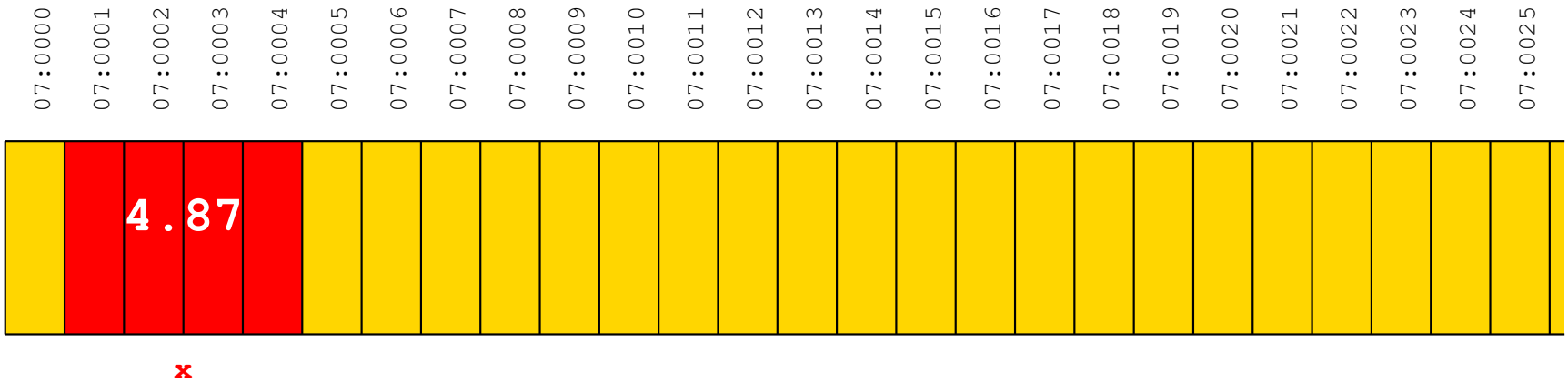
Wie erfolgt Übergabe der Parameterwerte (Argumente) im Speicher?

Zu welchen Zeitpunkten ist wofür Speicherplatz reserviert?

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

Speicheradressen (fiktiver Adressbereich)

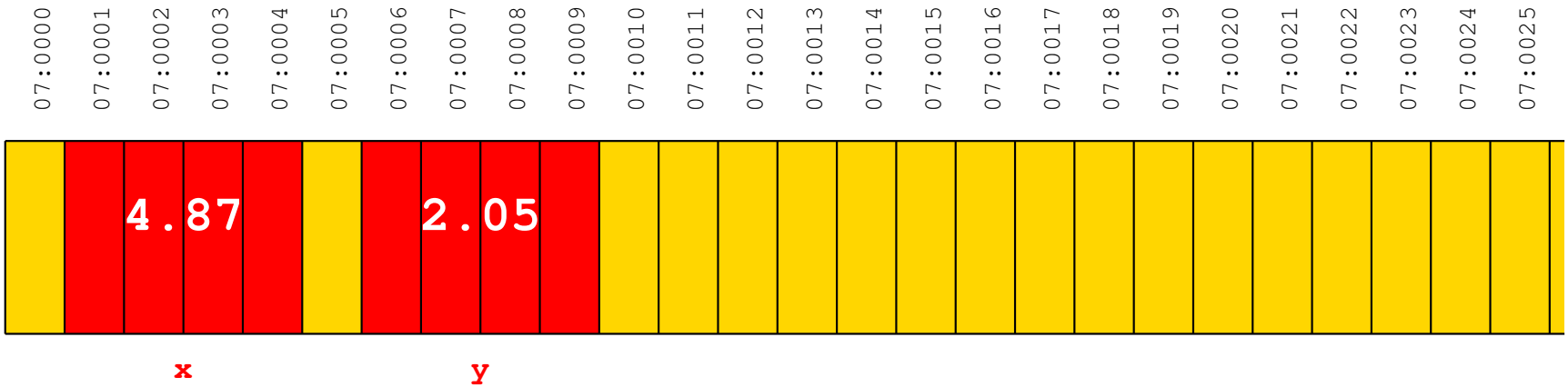


Für die **`float`**-Variable **`x`** werden 4 Bytes (32 Bit) im Speicher reserviert und mit dem Bitmuster des Wertes **`4.87`** gemäß Kodierung IEEE754 belegt.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

Speicheradressen (fiktiver Adressbereich)

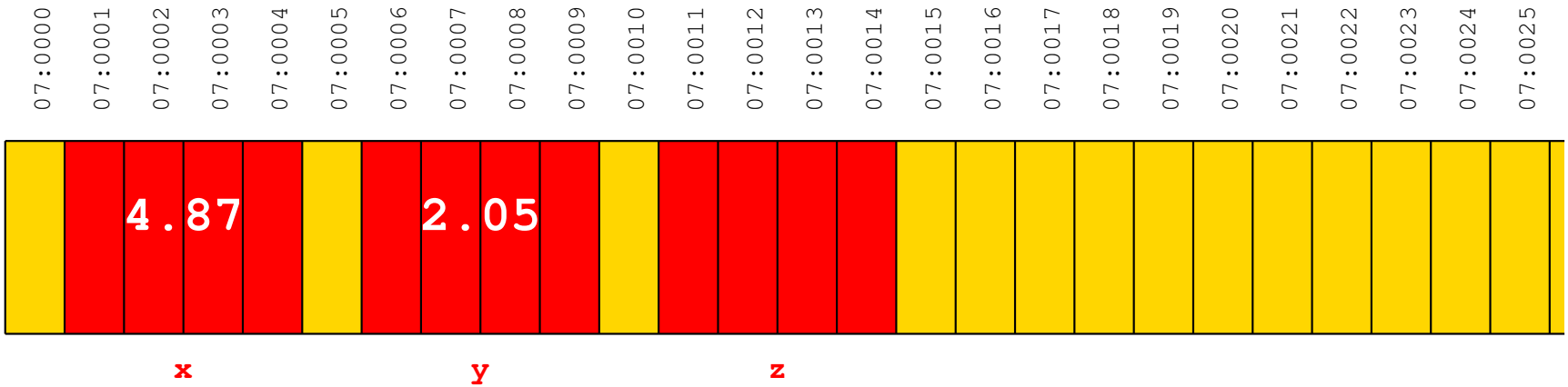


Für die **`float`**-Variable **`y`** ebenfalls 4 Bytes (32 Bit) im Speicher reserviert und mit dem Bitmuster des Wertes **`2.05`** gemäß Kodierung IEEE754 belegt.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

Speicheradressen (fiktiver Adressbereich)

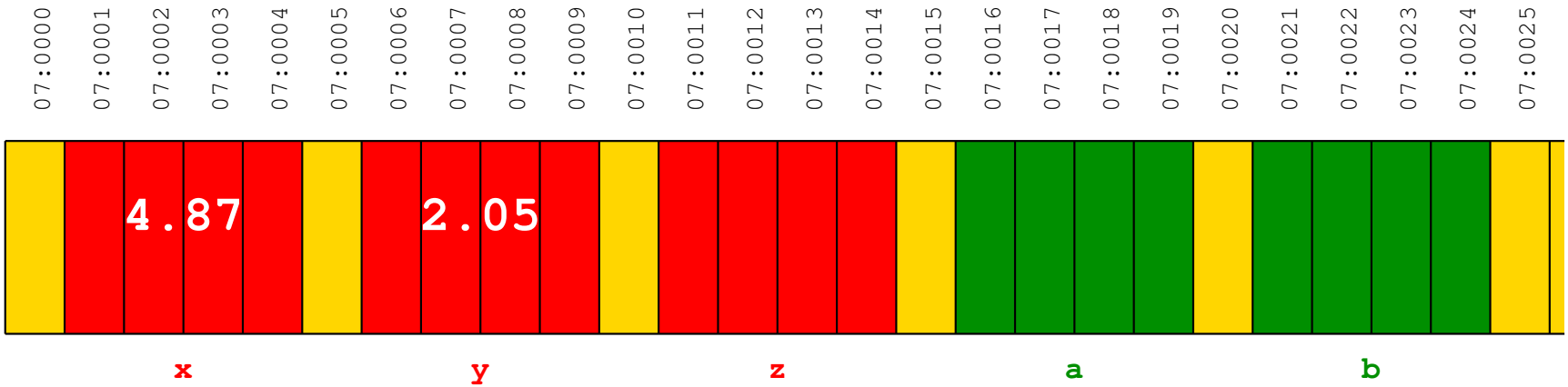


Für die **`float`**-Variable **`z`** auch 4 Bytes (32 Bit) im Speicher reserviert und mit Nullen gefüllt. Dieses Bitmuster wird als Wert gemäß Kodierung IEEE754 interpretiert.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

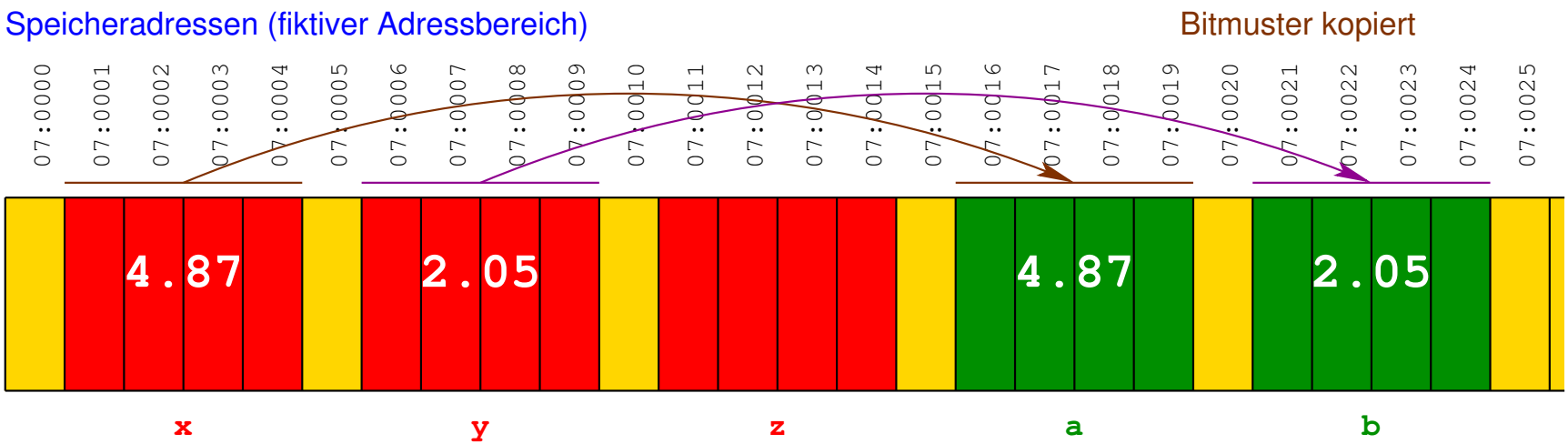
Speicheradressen (fiktiver Adressbereich)



Bei Methodenaufruf für jedes Argument eine entsprechende Variable angelegt und dafür *neuer Speicherplatz* reserviert, hier also für die **`float`**-Variablen **`a`** und **`b`**.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

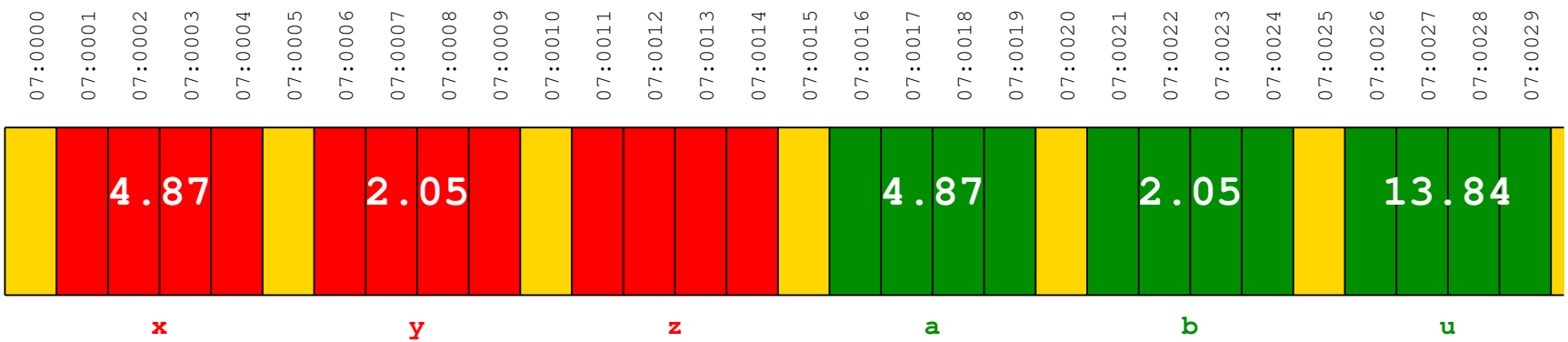


Das Bitmuster des Speicherbereiches **x** wird in den Speicherbereich **a** *kopiert* und das Bitmuster des Speicherbereiches **y** in den Speicherbereich **b**.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

Speicheradressen (fiktiver Adressbereich)

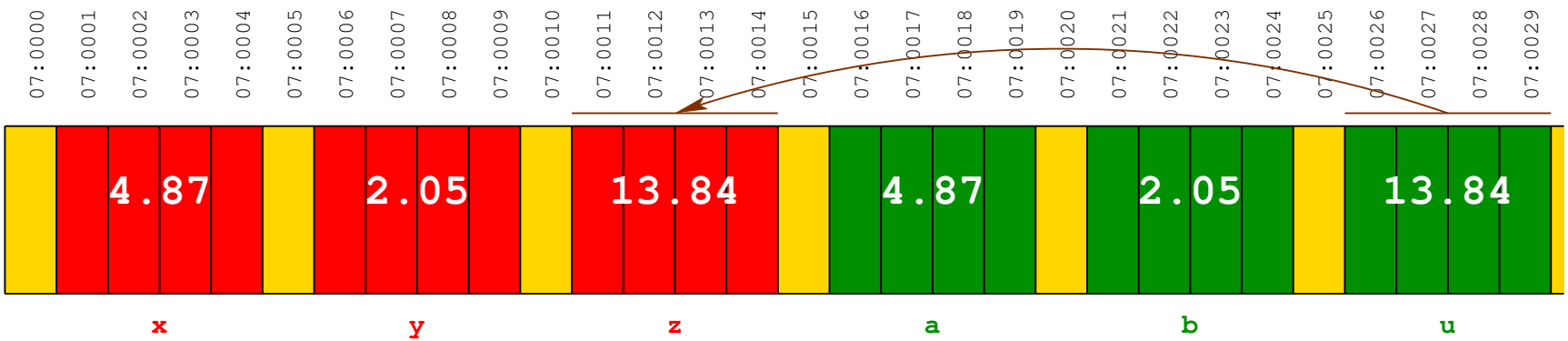


Für die **`float`**-Variable **`u`** werden 4 Bytes (32 Bit) im Speicher reserviert und mit dem Bitmuster des **`float`**-Wertes **`13.84`** ($= 2 \cdot (4.87 + 2.05)$) belegt.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

Speicheradressen (fiktiver Adressbereich)

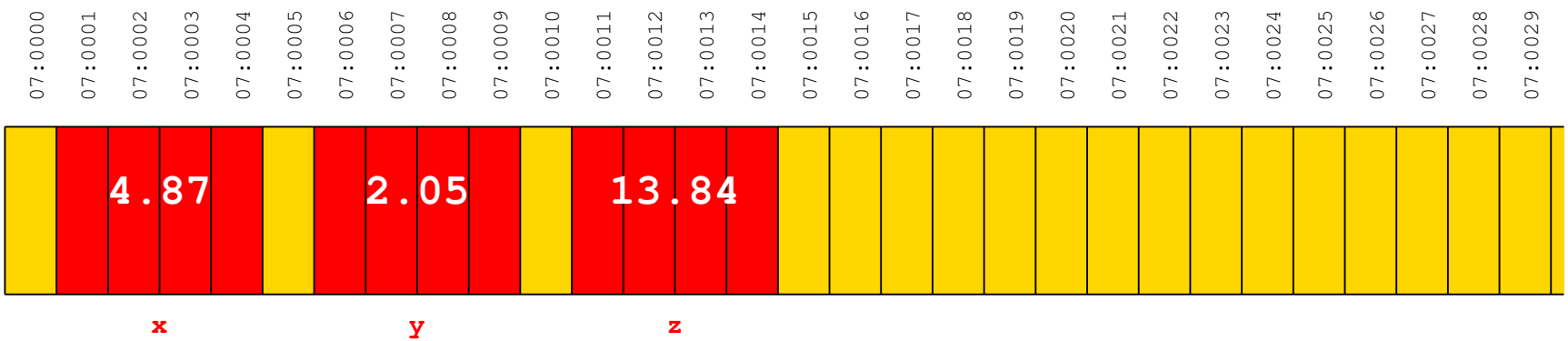


Das Bitmuster des Speicherbereiches **u** wird in den Speicherbereich **z** kopiert zur Rückgabe des berechneten Funktionswertes.

Übergabe der Parameterwerte (call by value)

```
public class MeineRechtecksberechnung {

  public static float rechteckumfang(float a, float b)
  {
    float u = 2*(a+b);
    return u;
  }

  public static void main(String[] args)
  {
    float x = 4.87f;
    float y = 2.05f;
    float z;

    z = rechteckumfang(x, y);
    System.out.printf("Umfang: %f\n", z);
  }
}
```

Speicheradressen (fiktiver Adressbereich)



Bei Erreichen des Blockendes der Methode **`rechteckumfang`** werden alle dort ausgefassten Speicherbereiche wieder freigegeben, die Variablen **`a`**, **`b`** und **`u`** existieren nicht mehr.

Parameterübernahme von der Kommandozeile

MainArgsZinseszins.java – Zinseszinsberechnung

```
public class MainArgsZinseszins {
  public static void main(String[] args) {

    float startkapital, zinssatz;
    int laufzeit;
    float endkapital;

    if (args.length < 3)   //Anzahl eingelesene Argumente pruefen
    {
      System.out.println("Ungueltige Eingabe!");
    }
    else
    {
      startkapital = Float.parseFloat(args[0]); //wandelt Zeichenkette in float-Wert um
      zinssatz = Float.parseFloat(args[1]);
      laufzeit = Integer.parseInt(args[2]); //wandelt Zeichenkette in int-Wert um

      endkapital = (float) Math.pow(1.0f+zinssatz/100.0f,laufzeit) * startkapital;
      System.out.printf("Endkapital: %.2f\n", endkapital);
    }
  }
}

//Aufruf: java MainArgsZinseszins <Startkapital> <jaehrl_Zinssatz_Prozent> <Laufzeit_in_Jahren>
//Aufrufparameter stets durch genau ein Leerzeichen voneinander trennen
//also z.B.: java MainArgsZinseszins 5000.00 3.5 20
```

Parameterübernahme von der Kommandozeile

immer an die Methode **`main(String[] args)`**

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Aufruf z.B.: `java MainArgsZinseszins 5000.00 3.5 20`

Parameterübernahme von der Kommandozeile

immer an die Methode `main(String[] args)`

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Aufruf z.B.: `java MainArgsZinseszins 5000.00 3.5 20`
- `args.length` liefert *Anzahl* eingelesener Parameter

Parameterübernahme von der Kommandozeile

immer an die Methode `main(String[] args)`

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Aufruf z.B.: `java MainArgsZinseszins 5000.00 3.5 20`
- `args.length` liefert *Anzahl* eingelesener Parameter
- `args` ist ein Feld von Zeichenketten. Jeder Parameter als Zeichenkette bereitgestellt
- Die einzelnen Zeichenketten haben die Namen `args[0]`, `args[1]`, `args[2]`

Parameterübernahme von der Kommandozeile

immer an die Methode `main(String[] args)`

- Parameterwerte bei Programmaufruf in Kommandozeile mitgeben
- Parameterwerte jeweils durch ein Leerzeichen trennen
- Aufruf z.B.: `java MainArgsZinseszins 5000.00 3.5 20`
- `args.length` liefert *Anzahl* eingelesener Parameter
- `args` ist ein Feld von Zeichenketten. Jeder Parameter als Zeichenkette bereitgestellt
- Die einzelnen Zeichenketten haben die Namen `args[0]`, `args[1]`, `args[2]`

Bibliotheksklassen **`Float`** und **`Integer`** (*Wrapper* für elem. Typen)

- `parseFloat` wandelt Zeichenkette in `float`-Wert um
- `parseInt` wandelt Zeichenkette in `int`-Wert um, sofern möglich

Methoden einer Klasse überladen

Überladen ist in der Programmierung sinnvoll,
aber nicht im Straßenverkehr

Mehrere namensgleiche Methoden in einer Klasse

Methoden überladen

```
      double x = pseudozufallszahl( 10.0 );    //0 <= x < max

double y = pseudozufallszahl( 23.5, 98.7 );    //min <= y < max

         long z = pseudozufallszahl( 1, 6);    //ganzzahlig min <= z <= max

      double w = pseudozufallszahl( true );    //normalverteilt? ja/nein
```

- Für den Nutzer einer Klasse ist es bequem, wenn sich der Methodenaufruf flexibel nach den übergebenen Argumenten (Anzahl und Typen) richtet.
- Beispiel: Szenarien bei Erzeugung von *Pseudozufallszahlen*.

Mehrere namensgleiche Methoden in einer Klasse

Methoden überladen

```
      double x = pseudozufallszahl( 10.0 );    //0 <= x < max

double y = pseudozufallszahl( 23.5, 98.7 );    //min <= y < max

         long z = pseudozufallszahl( 1, 6);    //ganzzahlig min <= z <= max

      double w = pseudozufallszahl( true );    //normalverteilt? ja/nein
```

- Für den Nutzer einer Klasse ist es bequem, wenn sich der Methodenaufruf flexibel nach den übergebenen Argumenten (Anzahl und Typen) richtet.
- Beispiel: Szenarien bei Erzeugung von *Pseudozufallszahlen*.
- Alle bereitgestellten Methoden heißen **`pseudozufallszahl`**, unterscheiden sich aber in der *Anzahl* und *Typung* der jeweils *übergebenen Parameter*.

Mehrere namensgleiche Methoden in einer Klasse

Methoden überladen

```
      double x = pseudozufallszahl( 10.0 );    //0 <= x < max

double y = pseudozufallszahl( 23.5, 98.7 );    //min <= y < max

         long z = pseudozufallszahl( 1, 6);    //ganzzahlig min <= z <= max

      double w = pseudozufallszahl( true );    //normalverteilt? ja/nein
```

- Für den Nutzer einer Klasse ist es bequem, wenn sich der Methodenaufruf flexibel nach den übergebenen Argumenten (Anzahl und Typen) richtet.
- Beispiel: Szenarien bei Erzeugung von *Pseudozufallszahlen*.
- Alle bereitgestellten Methoden heißen **`pseudozufallszahl`**, unterscheiden sich aber in der *Anzahl* und *Typung* der jeweils *übergebenen Parameter*.
- Es gibt *mehrere Varianten* der Methode **`pseudozufallszahl`**. Man sagt dann, diese Methode ist *überladen*.

Mehrere namensgleiche Methoden in einer Klasse

Klasse MeineZufallszahlen mit überladener Methode pseudozufallszahl

```
public class MeineZufallszahlen {
  public static double pseudozufallszahl(double max) { // 0...< max
    return Math.abs(max) * Math.random();
  }

  public static double pseudozufallszahl(double min, double max) { // min... < max
    if (min == max) {return min;}
    if (min > max) {double h = min; min = max; max = h;}
    return min + Math.random() * (max - min);
  }

  public static long pseudozufallszahl(long min, long max) { // min...max ganzzahlig
    if (min == max) {return min;}
    if (min > max) {long h = min; min = max; max = h;}
    return min + Math.round( Math.random() * (max - min) );
  }

  public static double pseudozufallszahl(boolean glocke) {
    if (glocke) {  //bei true arith. Mittel aus drei PZF-Zahlen bilden, 0...< 1
      return (Math.random() + Math.random() + Math.random()) / 3;
    }
    return Math.random();
  }
```

Mehrere namensgleiche Methoden in einer Klasse

Klasse MeineZufallszahlen mit überladener Methode pseudozufallszahl

```
  // main-Methode ruft pseudozufallszahl-Methoden mit verschiedenen Typsignaturen auf

  public static void main(String[] args) {
     System.out.println("Pseudozufallszahl 0 ... < 10.0:  "+pseudozufallszahl(10.0));
     System.out.println("Pseudozufallszahl 23.5 ... 98.7: "+pseudozufallszahl(23.5, 98.7));
     System.out.println("PZZ ganzzahlig    1 ... 6:       "+pseudozufallszahl(1, 6));
     System.out.println("PZZ gemischt      1 ... 7.5:     "+pseudozufallszahl(1, 7.5));
     System.out.println("PZZ normalverteilt 0 ... < 1:    "+pseudozufallszahl(true));
  }
} //class
```

Mehrere namensgleiche Methoden in einer Klasse

Klasse MeineZufallszahlen mit überladener Methode pseudozufallszahl

```
  // main-Methode ruft pseudozufallszahl-Methoden mit verschiedenen Typsignaturen auf

  public static void main(String[] args) {
     System.out.println("Pseudozufallszahl 0 ... < 10.0:  "+pseudozufallszahl(10.0));
     System.out.println("Pseudozufallszahl 23.5 ... 98.7: "+pseudozufallszahl(23.5, 98.7));
     System.out.println("PZZ ganzzahlig    1 ... 6:       "+pseudozufallszahl(1, 6));
     System.out.println("PZZ gemischt      1 ... 7.5:     "+pseudozufallszahl(1, 7.5));
     System.out.println("PZZ normalverteilt 0 ... < 1:    "+pseudozufallszahl(true));
  }
} //class
```

```
hinzet@bruch:~/Dokumente/Lehre-Einf-in-die-Prog-SoSe15/vorlesungsteil04$ java MeineZufallszahlen
Pseudozufallszahl 0 ... < 10.0:  5.71377879343547
Pseudozufallszahl 23.5 ... 98.7: 33.7524207022193
PZZ ganzzahlig    1 ... 6:       3
PZZ gemischt      1 ... 7.5:     5.704111409295469
PZZ normalverteilt 0 ... < 1:    0.4852915535953876
hinzet@bruch:~/Dokumente/Lehre-Einf-in-die-Prog-SoSe15/vorlesungsteil04$ java MeineZufallszahlen
Pseudozufallszahl 0 ... < 10.0:  5.247600541284356
Pseudozufallszahl 23.5 ... 98.7: 41.728382466963694
PZZ ganzzahlig    1 ... 6:       1
PZZ gemischt      1 ... 7.5:     2.2834702721315008
PZZ normalverteilt 0 ... < 1:    0.3957662495012249
```

Merke

- Mehrere namensgleiche Methoden in derselben Klasse müssen sich durch die *Anzahl Parameter* und/oder die *Typung der Parameter* in der *Argumentliste* unterscheiden.

Merke

- Mehrere namensgleiche Methoden in derselben Klasse müssen sich durch die *Anzahl Parameter* und/oder die *Typung der Parameter* in der *Argumentliste* unterscheiden.
- Das Schlüsselwort `void` („Leere“) ist in Java *kein Typ*. Wird eine Methode überladen, darf es von ihr *keine Variante* geben, deren *Argumentliste* aus dem Schlüsselwort `void` besteht.

Merke

- Mehrere namensgleiche Methoden in derselben Klasse müssen sich durch die *Anzahl Parameter* und/oder die *Typung der Parameter* in der *Argumentliste* unterscheiden.
- Das Schlüsselwort `void` („Leere“) ist in Java *kein Typ*. Wird eine Methode überladen, darf es von ihr *keine Variante* geben, deren *Argumentliste* aus dem Schlüsselwort `void` besteht.
- Es ist in Java unzulässig, in einer Klasse mehrere Methoden gleichen Namens und typungsgleicher Argumentliste zu haben, die sich nur durch ihren *Rückgabetyp* unterscheiden. Der Rückgabetyp lässt sich bei Methodenaufrufauswahl *nicht immer zweifelsfrei* ermitteln, z.B.

```
System.out.println(pseudozufallszahl(a, b));
```

Merke

- Mehrere namensgleiche Methoden in derselben Klasse müssen sich durch die *Anzahl Parameter* und/oder die *Typung der Parameter* in der *Argumentliste* unterscheiden.
- Das Schlüsselwort `void` („Leere“) ist in Java *kein Typ*. Wird eine Methode überladen, darf es von ihr *keine Variante* geben, deren *Argumentliste* aus dem Schlüsselwort `void` besteht.
- Es ist in Java unzulässig, in einer Klasse mehrere Methoden gleichen Namens und typungsgleicher Argumentliste zu haben, die sich nur durch ihren *Rückgabetyp* unterscheiden. Der Rückgabetyp lässt sich bei Methodenaufrufauswahl *nicht immer zweifelsfrei* ermitteln, z.B.
  ```
  System.out.println(pseudozufallszahl(a, b));
  ```
- Bei Aufruf wird diejenige Methodenvariante ausgewählt, bei der die Anzahl übergebener Argumente übereinstimmt und in der Argumentliste von links nach rechts die Typungen passen, so dass *so wenig implizite Typcasts wie möglich* notwendig sind (am besten gar keine), z.B.
  ```
  System.out.println(pseudozufallszahl(1L, 7.5));
  ```
  Mehrdeutigkeiten durch Compilierfehler signalisiert

Modularisierung – Methodenbibliotheken anlegen

Idee der Modularisierung

- Zum Zubereiten einer guten Mahlzeit nutzt man viele, teilweise vorgefertigte Zutaten
- Diese Zutaten sind *Bausteine*, aus denen sich die *Mahlzeit zusammensetzt*
- Die Zutaten ergänzen sich sinnvoll in der Mahlzeit und sind weitgehend unabhängig voneinander
- *Methoden* sind die Zutaten für eine *Methodenbibliothek*, auch *Modul* oder *zustandslose Klasse* genannt

Methodenbibliothek als eigenständige Klasse

- Eine Sammlung von Methoden, die in einem sinnvollen Zusammenhang zueinander stehen und sich hinsichtlich ihrer Aufgaben gegenseitig ergänzen, bildet eine *Methodenbibliothek*, die man üblicherweise als *eigenständige Klasse* implementiert.

Methodenbibliothek als eigenständige Klasse

- Eine Sammlung von Methoden, die in einem sinnvollen Zusammenhang zueinander stehen und sich hinsichtlich ihrer Aufgaben gegenseitig ergänzen, bildet eine *Methodenbibliothek*, die man üblicherweise als *eigenständige Klasse* implementiert.
- Beispiel: Die Bibliotheksklasse **`Math`**, die etwa 50 verschiedene mathematische Funktionen jeweils als Methode zur Verfügung stellt.

Methodenbibliothek als eigenständige Klasse

- Eine Sammlung von Methoden, die in einem sinnvollen Zusammenhang zueinander stehen und sich hinsichtlich ihrer Aufgaben gegenseitig ergänzen, bildet eine *Methodenbibliothek*, die man üblicherweise als *eigenständige Klasse* implementiert.
- Beispiel: Die Bibliotheksklasse **`Math`**, die etwa 50 verschiedene mathematische Funktionen jeweils als Methode zur Verfügung stellt.
- Auch die Bibliotheksklasse **`System.out`** mit Methoden zur Bildschirmausgabe,

Methodenbibliothek als eigenständige Klasse

- Eine Sammlung von Methoden, die in einem sinnvollen Zusammenhang zueinander stehen und sich hinsichtlich ihrer Aufgaben gegenseitig ergänzen, bildet eine *Methodenbibliothek*, die man üblicherweise als *eigenständige Klasse* implementiert.
- Beispiel: Die Bibliotheksklasse **`Math`**, die etwa 50 verschiedene mathematische Funktionen jeweils als Methode zur Verfügung stellt.
- Auch die Bibliotheksklasse **`System.out`** mit Methoden zur Bildschirmausgabe,
- ebenso die Wrapper-Klassen **`Double`**, ..., **`Integer`**, von denen wir bisher parse-Methoden kennengelernt haben wie **`parseDouble`**.

Double

```
Double (double value)
Double (String s) ¦
```

Static Methods

```
    int compare (double d1, double d2)
   long doubleToLongBits (double value)
   long doubleToRawLongBits (double value)
boolean isInfinite (double v)
boolean isNaN (double v)
 double longBitsToDouble (long bits)
 double parseDouble (String s) ¦
 String toString (double d)
 Double valueOf (String s) ¦
```

Accessors

```
boolean isInfinite ()
boolean isNaN ()
```

Object

```
boolean equals (Object obj)
    int hashCode ()
 String toString ()
```

Other Public Methods

```
    int compareTo (Double anotherDouble)
    int compareTo (Object o)
```

```
double POSITIVE_INFINITY,
       NEGATIVE_INFINITY, NaN, MAX_VALUE,
       MIN_VALUE
Class TYPE
```

www.falkhausen.de

Methodenbibliothek als eigenständige Klasse

- Eine Sammlung von Methoden, die in einem sinnvollen Zusammenhang zueinander stehen und sich hinsichtlich ihrer Aufgaben gegenseitig ergänzen, bildet eine *Methodenbibliothek*, die man üblicherweise als *eigenständige Klasse* implementiert.
- Beispiel: Die Bibliotheksklasse **`Math`**, die etwa 50 verschiedene mathematische Funktionen jeweils als Methode zur Verfügung stellt.
- Auch die Bibliotheksklasse **`System.out`** mit Methoden zur Bildschirmausgabe,
- ebenso die Wrapper-Klassen **`Double`**, ..., **`Integer`**, von denen wir bisher parse-Methoden kennengelernt haben wie **`parseDouble`**.

Double

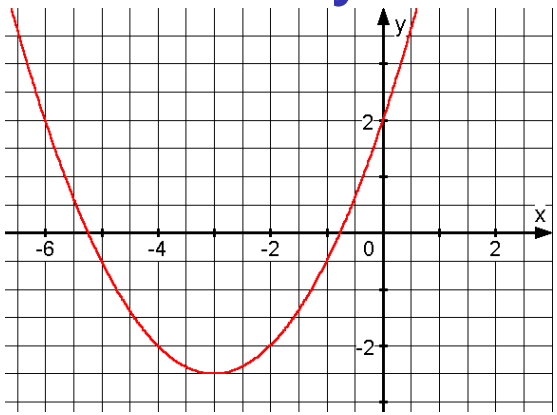
```
Double (double value)
Double (String s) ¦

Static Methods
    int compare (double d1, double d2)
   long doubleToLongBits (double value)
   long doubleToRawLongBits (double value)
boolean isInfinite (double v)
boolean isNaN (double v)
 double longBitsToDouble (long bits)
 double parseDouble (String s) ¦
 String toString (double d)
 Double valueOf (String s) ¦
Accessors
boolean isInfinite ()
boolean isNaN ()
Object
boolean equals (Object obj)
    int hashCode ()
 String toString ()
Other Public Methods
    int compareTo (Double anotherDouble)
    int compareTo (Object o)

double POSITIVE_INFINITY,
       NEGATIVE_INFINITY, NaN, MAX_VALUE,
       MIN_VALUE
Class TYPE
```

www.falkhausen.de

Wie schreibt man eine eigene Methodenbibliothek als Klasse?

Bibliothek zum Handling quadratischer Polynome

Als Beispiel bauen wir uns eine Bibliothek
mit Methoden auf quadratischen Polynomen



$f(x) = a_2 \cdot x^2 + a_1 \cdot x + a_0$

mit $a_2, a_1, a_0 \in \mathbb{R}$ und $a_2 \neq 0$

`float f(float a2, float a1, float a0, float x)`
. . . liefert $f(x)$

`boolean hatReelleNst(float a2, float a1, float a0)`
. . . liefert `true`, falls reelle Nullstelle vorhanden, sonst `false`

`float reelleNst1(float a2, float a1, float a0)`
. . . liefert erste Nullstelle

`float reelleNst2(float a2, float a1, float a0)`
. . . liefert zweite Nullstelle

`float scheitelx(float a2, float a1, float a0)`
. . . liefert x-Koordinate des Scheitelpunkts

`float scheitely(float a2, float a1, float a0)`
. . . liefert y-Koordinate des Scheitelpunkts

Methodenbibliothek QuadratischesPolynom

```
public class QuadratischesPolynom {
  public static float f(float a2, float a1, float a0, float x) {
    return a2*x*x + a1*x + a0;
  }

  public static boolean hatReelleNst(float a2, float a1, float a0) {
    return (a1*a1/4 >= a0);
  }

  public static float reelleNst1(float a2, float a1, float a0) {
    if (hatReelleNst(a2, a1, a0)) {
      return -a1/(2*a2) - (float) Math.sqrt(a1*a1/(4*a2) - a0/a2);
    }
    return -1.7e+38f; //Rueckgabewert im Fehlerfall
  }

  public static float reelleNst2(float a2, float a1, float a0) {
    if (hatReelleNst(a2, a1, a0)) {
      return -a1/(2*a2) + (float) Math.sqrt(a1*a1/(4*a2) - a0/a2);
    }
    return -1.7e+38f; //Rueckgabewert im Fehlerfall
  }

  public static float scheitelx(float a2, float a1, float a0) {
    return -a1/(2*a2);
  }

  public static float scheitely(float a2, float a1, float a0) {
    return f(a2, a1, a0, scheitelx(a2, a1, a0));
  }
} //class
```

Alle definierten Methoden werden in die Klasse **`QuadratischesPolynom`** geschrieben und als Datei QuadratischesPolynom.java abgespeichert.

Durch die Kennzeichnung der Klasse sowie ihrer Methoden als `public` kann von anderen Klassen direkt darauf zugegriffen werden.

Die Bibliothek nutzen (MeinPolynomTest.java)

```java
import java.util.Scanner;

public class MeinPolynomTest {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    float p, q;

    System.out.printf("Quadratisches Polynom f(x) = x*x + p*x + q\n\n");
    System.out.print("Bitte Koeffizienten p eingeben: ");
    p = Float.parseFloat(sc.next());
    System.out.print("Bitte Koeffizienten q eingeben: ");
    q = Float.parseFloat(sc.next());

    if (QuadratischesPolynom.hatReelleNst(1, p, q)) {
      float x1 = QuadratischesPolynom.reelleNst1(1, p, q);
      float x2 = QuadratischesPolynom.reelleNst2(1, p, q);
      System.out.printf("Nullstellen: %f und %f\n", x1, x2);
    }
    float sx = QuadratischesPolynom.scheitelx(1, p, q);
    float sy = QuadratischesPolynom.scheitely(1, p, q);
    System.out.printf("Scheitelpunkt: (%f, %f)\n", sx, sy);
  }
}
```

Merke

- Alle involvierten Java-Quelltextdateien, hier also **`QuadratischesPolynom.java`** und **`MeinPolynomTest.java`**, sollten im *gleichen Verzeichnis* abgelegt sein.

Merke

- Alle involvierten Java-Quelltextdateien, hier also **`QuadratischesPolynom.java`** und **`MeinPolynomTest.java`**, sollten im *gleichen Verzeichnis* abgelegt sein.
- Beim Compilieren ruft man nur die Klasse auf, in welcher die **`main`**-Methode steht, hier also:
  `javac MeinPolynomTest.java`

Merke

- Alle involvierten Java-Quelltextdateien, hier also **`QuadratischesPolynom.java`** und **`MeinPolynomTest.java`**, sollten im *gleichen Verzeichnis* abgelegt sein.
- Beim Compilieren ruft man nur die Klasse auf, in welcher die **`main`**-Methode steht, hier also:
  `javac MeinPolynomTest.java`
- Ebenso beim Ausführen: **`java MeinPolynomTest`**

```
hinzet@bruch:~/Dokumente/Lehre-Einf-in-die-Prog-SoSe15/vorlesungsteil04$ javac MeinPolynomTest.java
hinzet@bruch:~/Dokumente/Lehre-Einf-in-die-Prog-SoSe15/vorlesungsteil04$ java MeinPolynomTest
Quadratisches Polynom f(x) = x*x + p*x + q

Bitte Koeffizienten p eingeben: -6
Bitte Koeffizienten q eingeben: 8
Nullstellen: 2,000000 und 4,000000
Scheitelpunkt: (3,000000, -1,000000)
```

Methodenbibliotheks-Programmierregeln und Tipps

- Eine Bibliotheksklasse ist `public`
- Alle Methoden der Bibliotheksklasse, auf die der Zugriff von außen erlaubt wird, als `public` markieren
- Alle Methoden in einer Bibliotheksklasse werden zudem als `static` gekennzeichnet
- Methoden einer Bibliotheksklasse dürfen sich untereinander beliebig aufrufen
- Klassenglobale Konstanten sind in einer Bibliotheksklasse erlaubt (z.B. `Math.PI`)
- Guter Stil: In eigene Bibliotheken keine Bildschirmausgaben einbauen, sondern ausschließlich Rückgabewerte bereitstellen
- Zugriff von außen über Punkt-Operator:
  `<Bibliotheksklasse>.<Methodenname>(<Argumente>)`

Methodenbibliotheks-Programmierregeln und Tipps

- Eine Bibliotheksklasse ist `public`
- Alle Methoden der Bibliotheksklasse, auf die der Zugriff von außen erlaubt wird, als `public` markieren
- Alle Methoden in einer Bibliotheksklasse werden zudem als `static` gekennzeichnet
- Methoden einer Bibliotheksklasse dürfen sich untereinander beliebig aufrufen
- Klassenglobale Konstanten sind in einer Bibliotheksklasse erlaubt (z.B. `Math.PI`)
- Guter Stil: In eigene Bibliotheken keine Bildschirmausgaben einbauen, sondern ausschließlich Rückgabewerte bereitstellen
- Zugriff von außen über Punkt-Operator:
  `<Bibliotheksklasse>.<Methodenname>(<Argumente>)`

Methodenbibliotheks-Programmierregeln und Tipps

- Eine Bibliotheksklasse ist `public`
- Alle Methoden der Bibliotheksklasse, auf die der Zugriff von außen erlaubt wird, als `public` markieren
- Alle Methoden in einer Bibliotheksklasse werden zudem als `static` gekennzeichnet
- Methoden einer Bibliotheksklasse dürfen sich untereinander beliebig aufrufen
- Klassenglobale Konstanten sind in einer Bibliotheksklasse erlaubt (z.B. `Math.PI`)
- Guter Stil: In eigene Bibliotheken keine Bildschirmausgaben einbauen, sondern ausschließlich Rückgabewerte bereitstellen
- Zugriff von außen über Punkt-Operator:
  `<Bibliotheksklasse>.<Methodenname>(<Argumente>)`

Methodenbibliotheks-Programmierregeln und Tipps

- Eine Bibliotheksklasse ist `public`
- Alle Methoden der Bibliotheksklasse, auf die der Zugriff von außen erlaubt wird, als `public` markieren
- Alle Methoden in einer Bibliotheksklasse werden zudem als `static` gekennzeichnet
- Methoden einer Bibliotheksklasse dürfen sich untereinander beliebig aufrufen
- Klassenglobale Konstanten sind in einer Bibliotheksklasse erlaubt (z.B. `Math.PI`)
- Guter Stil: In eigene Bibliotheken keine Bildschirmausgaben einbauen, sondern ausschließlich Rückgabewerte bereitstellen
- Zugriff von außen über Punkt-Operator:
  `<Bibliotheksklasse>.<Methodenname>(<Argumente>)`

Methodenbibliotheks-Programmierregeln und Tipps

- Eine Bibliotheksklasse ist **`public`**
- Alle Methoden der Bibliotheksklasse, auf die der Zugriff von außen erlaubt wird, als **`public`** markieren
- Alle Methoden in einer Bibliotheksklasse werden zudem als **`static`** gekennzeichnet
- Methoden einer Bibliotheksklasse dürfen sich untereinander beliebig aufrufen
- Klassenglobale Konstanten sind in einer Bibliotheksklasse erlaubt (z.B. **`Math.PI`**)
- Guter Stil: In eigene Bibliotheken keine Bildschirmausgaben einbauen, sondern ausschließlich Rückgabewerte bereitstellen
- Zugriff von außen über Punkt-Operator:
  `<Bibliotheksklasse>.<Methodenname>(<Argumente>)`

Methodenbibliotheks-Programmierregeln und Tipps

- Eine Bibliotheksklasse ist **`public`**
- Alle Methoden der Bibliotheksklasse, auf die der Zugriff von außen erlaubt wird, als **`public`** markieren
- Alle Methoden in einer Bibliotheksklasse werden zudem als **`static`** gekennzeichnet
- Methoden einer Bibliotheksklasse dürfen sich untereinander beliebig aufrufen
- Klassenglobale Konstanten sind in einer Bibliotheksklasse erlaubt (z.B. **`Math.PI`**)
- Guter Stil: In eigene Bibliotheken keine Bildschirmausgaben einbauen, sondern ausschließlich Rückgabewerte bereitstellen
- Zugriff von außen über Punkt-Operator:
  `<Bibliotheksklasse>.<Methodenname>(<Argumente>)`

Methodenbibliotheks-Programmierregeln und Tipps

- Eine Bibliotheksklasse ist **`public`**
- Alle Methoden der Bibliotheksklasse, auf die der Zugriff von außen erlaubt wird, als **`public`** markieren
- Alle Methoden in einer Bibliotheksklasse werden zudem als **`static`** gekennzeichnet
- Methoden einer Bibliotheksklasse dürfen sich untereinander beliebig aufrufen
- Klassenglobale Konstanten sind in einer Bibliotheksklasse erlaubt (z.B. **`Math.PI`**)
- Guter Stil: In eigene Bibliotheken keine Bildschirmausgaben einbauen, sondern ausschließlich Rückgabewerte bereitstellen
- Zugriff von außen über Punkt-Operator:
  `<Bibliotheksklasse>.<Methodenname>(<Argumente>)`

Methodenbibliotheken als „gedächtnislose“ Klassen

Kaffeetasse	Klasse in Java
	**Daten:** ~~enthaltenes Kaffeevolumen~~ ~~in Kubikzentimeter~~ ~~Kapazitaet~~ ~~in Kubikzentimeter~~

	**Werkzeuge (Methoden):**
	`eingiessen (bestimmtes Volumen)`
	`trinken (bestimmtes Volumen)`
	`testen auf leere Tasse`

Klasse allgemein als Zusammenfassung von Zustandsdaten und Methoden

Methodenbibliotheken als „gedächtnislose“ Klassen

- In einer Methodenbibliothek gibt es *keine veränderlichen Datenwerte*, die den Aufruf von Methoden *überdauern*.

Methodenbibliotheken als „gedächtnislose“ Klassen

- In einer Methodenbibliothek gibt es *keine veränderlichen Datenwerte*, die den Aufruf von Methoden *überdauern*.
- Folglich müssen bei der Programmausführung auch nicht mehrere Exemplare („Objekte“) für verschiedene Settings von Zustandsdaten der Klasse unterschieden werden können.

Methodenbibliotheken als „gedächtnislose“ Klassen

- In einer Methodenbibliothek gibt es *keine veränderlichen Datenwerte*, die den Aufruf von Methoden *überdauern*.
- Folglich müssen bei der Programmausführung auch nicht mehrere Exemplare („Objekte“) für verschiedene Settings von Zustandsdaten der Klasse unterschieden werden können.
- Interne Speicherverwaltung bei Arbeit mit Klassen lässt sich einfacher halten, wenn gleichzeitig nur ein Exemplar einer Klasse existieren kann, wie bei Methodenbibliotheken der Fall.

Methodenbibliotheken als „gedächtnislose“ Klassen

- In einer Methodenbibliothek gibt es *keine veränderlichen Datenwerte*, die den Aufruf von Methoden *überdauern*.
- Folglich müssen bei der Programmausführung auch nicht mehrere Exemplare („Objekte“) für verschiedene Settings von Zustandsdaten der Klasse unterschieden werden können.
- Interne Speicherverwaltung bei Arbeit mit Klassen lässt sich einfacher halten, wenn gleichzeitig nur ein Exemplar einer Klasse existieren kann, wie bei Methodenbibliotheken der Fall.
- Das Schlüsselwort `static` vor jeder Bibliotheksmethode veranlasst diese einfachere Speicherverwaltung und bedingt, dass es zu jedem Zeitpunkt der Programmausführung nur höchstens *ein Exemplar* der Klasse geben darf, welches *Klassenobjekt* genannt wird.

Methodenbibliotheken als „gedächtnislose“ Klassen

- In einer Methodenbibliothek gibt es *keine veränderlichen Datenwerte*, die den Aufruf von Methoden *überdauern*.
- Folglich müssen bei der Programmausführung auch nicht mehrere Exemplare („Objekte“) für verschiedene Settings von Zustandsdaten der Klasse unterschieden werden können.
- Interne Speicherverwaltung bei Arbeit mit Klassen lässt sich einfacher halten, wenn gleichzeitig nur ein Exemplar einer Klasse existieren kann, wie bei Methodenbibliotheken der Fall.
- Das Schlüsselwort `static` vor jeder Bibliotheksmethode veranlasst diese einfachere Speicherverwaltung und bedingt, dass es zu jedem Zeitpunkt der Programmausführung nur höchstens *ein Exemplar* der Klasse geben darf, welches *Klassenobjekt* genannt wird.
- Klassenglobale Konstanten sind unveränderliche Datenwerte, die zwar den Aufruf von Methoden überdauern, aber nicht das Anlegen unterschiedlicher Objekte erfordern.