11. Methoden

Methodendefinitionen- und Aufrufe, Auswertung von Methodenaufrufen, Der Typ `void`, Vor- und Nachbedingungen,Stepwise Refinement, Bibliotheken

Lernziele

- Sie können Code Fragmente in Methoden kapseln.
- Sie kennen alle Elemente der Methodendeklaration.
- Sie verstehen was genau mit den Parametern beim Methodenaufruf geschieht: **Pass by Value**
- Sie können für gegebene Methoden **Vor-** und **Nachbedingungen** formulieren.
- Sie können das Konzept der **schrittweisen Verfeinerung** anwenden.

Methoden

Code-Fragmente können zu Methoden geformt werden
Vorteile:

- Einmal definieren – mehrmals verwenden/aufrufen
- Übersichtlicherer Code, einfacher zu verstehen
- Code in Methoden kann separat getestet werden

Beispiel Keksrechner

```
public class Keksrechner {

  public static void main(String[] args){

    Out.print("Kinder: ");
    int kinder = In.readInt();

    Out.print("Kekse: ");
    int kekse = In.readInt();

    Out.println("Jedes Kind kriegt " + kekse / kinder + " Kekse");
    Out.println("Papa kriegt " + kekse % kinder + " Kekse");
  }
}
```

Keksrechner - Zusätzliche Anforderung

Wir wollen sicherstellen, dass **`kinder`** positiv ist und jedes Kind mindestens einen Keks kriegt. ⇒ **Eingabe prüfen!**

Keksrechner - Eingabeprüfung

Aus ...

```
Out.print("Kinder: ");
int kinder = In.readInt();
```

... wird demnach:

```
int kinder;
do {
    Out.print("Kinder: ");
    kinder = In.readInt();
    if (kinder < 1){
        Out.println("Wert zu klein. Mindestens " + 1);
    }
} while (kinder < 1 );
```

Analog dazu müssen wir prüfen, dass **`kekse >= kinder`** ist.

Keksrechner - Es wird unübersichtlich

```
public class Keksrechner {
  public static void main(String[] args) {
    int kinder;
    do {
        Out.print("Kinder: ");
        kinder = In.readInt();
        if (kinder < 1){
            Out.println("Wert zu klein. Mindestens " + 1);
        }
    } while (kinder < 1 );
    int kekse;
    do {
        Out.print("Kekse: ");
        kekse = In.readInt();
        if (kekse < kinder){
            Out.println("Wert zu klein. Mindestens " + kinder);
        }
    } while (kekse < kinder);
    Out.println("Jedes Kind kriegt " + kekse / kinder + " Kekse");
    Out.println("Papa kriegt " + kekse % kinder + " Kekse");
  }
}
```

Anzahl Kinder einlesen und prüfen

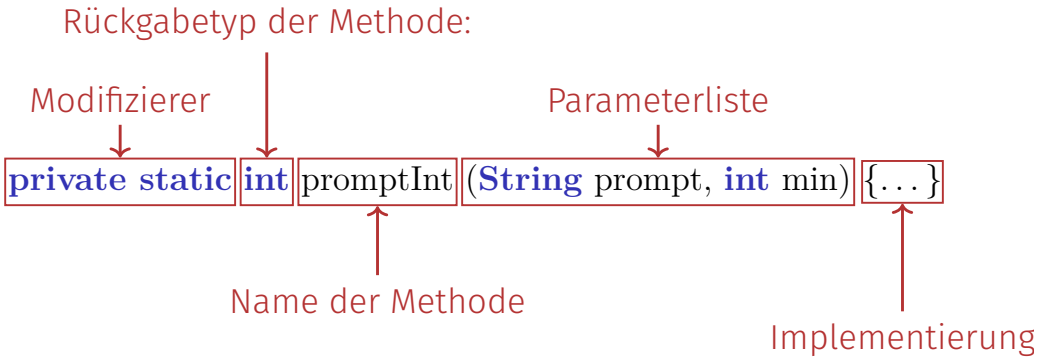
Anzahl Kekse einlesen und prüfen

Keksrechner - Erkenntnisse

- Die beiden Code-Fragmente sind **fast identisch**
- Folgende Aspekte sind unterschiedlich:
  - Der Prompt, also `"Kinder: "` vs. `"Kekse: "`
  - Das Minimum, also “`1`” vs. “**`kinder`**”

Wir können das Code-Fragment in eine Methode auslagern und somit **wiederverwenden**.
Dabei müssen wir die unterschiedlichen Aspekte **parametrisieren**.

Methodendeklaration und -definition



Methodendeklaration und -definition

- **Modifizierer**: Werden später behandelt
- **Rückgabetyp**: Datentyp des Rückgabewertes. Falls die Methode keinen Wert zurückliefert, ist dieser Typ `void`.
- **Name**: Ein gültiger Name. Sollte mit Kleinbuchstaben anfangen.
- **Parameterliste**: Mit rundern Klammern umgebene Liste von Parametern, deklariert mit Datentyp und Name. Parameter werden beim Aufruf der Methode gesetzt und können dann wie lokale Variablen verwendet werden.
- **Implementierung**: Der Code, welcher ausgeführt wird wenn die Methode aufgerufen wird.

Methodensignatur

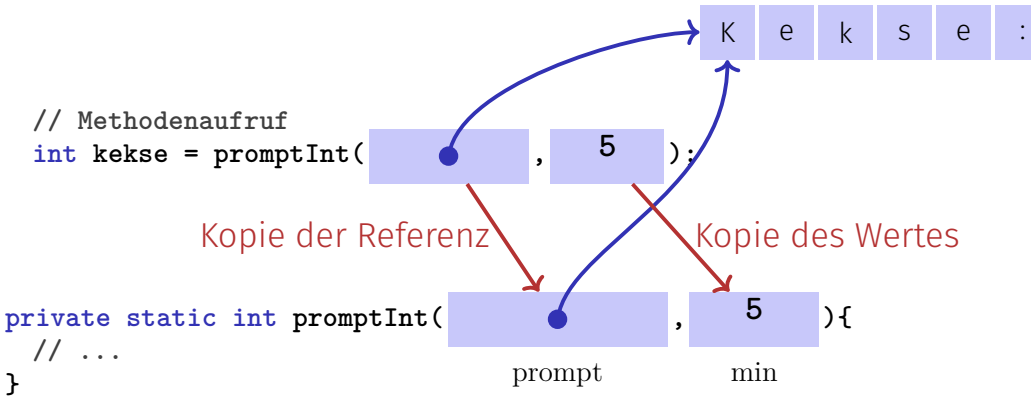
- Signatur ist eindeutig innerhalb einer Klasse.
- Es ist also möglich, mehrere Methoden mit gleichem Namen aber unterschiedlichen Parameter-Anzahl und -Typen zu haben – **aber nicht empfohlen!**
- Rückgabetyp ist nicht Teil der Signatur! Es ist nicht möglich, mehrere Methoden zu haben, die sich nur im Rückgabetyp unterscheiden.

Methodenaufruf - Pass by Value

- Ein Methodenaufruf ist ein Ausdruck, dessen Wert falls vorhanden der Rückgabewert der Methode ist.
- In Java gilt immer die **Pass by Value** Semantik.

Pass by Value bedeutet: Argumentwerte werden beim Methodenaufruf in die Parameter **kopiert**.
Dies entspricht demselben Prinzip wie die Wertzuweisung an eine Variable.

Methodenaufruf - Pass by Value



Zurück zum Beispiel - Methode `promptInt`

```
private static int promptInt(String prompt, int min){
    int number;
    do {
        Out.print(prompt);
        number = In.readInt();
        if (number < min) {
            Out.println("Wert zu klein. Mindestens " + min);
        }
    } while (number < min );
    return number;
}
```


Rückgabewerte von Methoden

Es gibt zwei Fälle

- **Rückgabetyp** = `void`: Die Auswertung der Methode **kann** mittels der Anweisung `return` beendet werden.
- **Rückgabetyp** ≠ `void`: Die Auswertung der Methode **muss** mittels der Anweisung "`return` Wert" beendet werden. Der Wert wird dann an die aufrufende Methode zurückgegeben.

Wichtig: Im zweiten Fall muss **jeder** mögliche endliche Ausführungspfad eine "`return`" Anweisung enthalten.

Keksrechner - Jetzt übersichtlicher

```
public class Keksrechner {

  public static void main(String[] args) {
    int kinder = readInt("Kinder: ", 1);
    int kekse = readInt("Kekse: ", kinder);

    Out.println("Jedes Kind kriegt " + kekse/kinder + " Kekse");
    Out.println("Papa kriegt " + kekse % kinder + " Kekse");
  }

  private static int readInt(String prompt, int min){
    // ... siehe vorige Slide
    return number;
  }
```

Vor- und Nachbedingungen

- beschreiben (möglichst vollständig) was die Methode „macht”
- dokumentieren die Methode für Benutzer / Programmierer (wir selbst oder andere)
- machen Programme lesbarer: wir müssen nicht verstehen, *wie* die Methode es macht
- werden vom Compiler ignoriert
- Vor- und Nachbedingungen machen – unter der Annahme ihrer Korrektheit – Aussagen über die Korrektheit eines Programmes möglich.

Beispiel: pow

```
public static double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) {
        result *= b;
    }
    return result;
}
```


Gültigkeit formaler Parameter

```
public static double
  pow(double b, int e){
    double r = 1.0;
    if (e<0) {
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        r * = b;
    return r;
}
```

```
public static void
  main(String[] args){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    Out.println(z); // 0.25
    Out.println(b); // 2
    Out.println(e); // -2
}
```

Nicht die formalen Parameter **b** und **e** von pow, sondern die hier definierten Variablen lokal zum Rumpf von **main**

Definition: Vor- und Nachbedingungen

“Verträge”, *welche das Verhalten einer Methode spezifizieren. Falls die Vorbedingung beim Aufruf einer Methode gilt, soll am Schlus der Ausführung die Nachbedingung gelten.*

Vorbedingungen

Vorbedingung (precondition):

- Was muss bei Methodenaufruf gelten?
- Spezifiziert **Definitionsbereich** der Methode.

0^e ist für $e < 0$ undefiniert

```
// PRE: e >= 0 || b != 0.0
```


Nachbedingungen

Nachbedingung (postcondition):

- Was gilt nach Methodenaufruf?
- Spezifiziert **Wert** und **Effekt** des Methodenaufrufes.

Hier nur Wert, kein Effekt.

```
// POST: return value is b^e
```


Vor- und Nachbedingungen

- sollten korrekt sein:
- **Wenn** die Vorbedingung beim Methodenaufruf gilt, **dann** gilt auch die Nachbedingung nach dem Methodenaufruf.

Methode `pow`: funktioniert für alle Basen $b \neq 0$

Vor- und Nachbedingungen

- Gilt Vorbedingung beim Methodenaufruf nicht, so machen wir keine Aussage.

Methode `pow`: Division durch 0

Vor- und Nachbedingungen

- Vorbedingung sollte so **schwach** wie möglich sein (möglichst grosser Definitionsbereich)
- Nachbedingung sollte so **stark** wie möglich sein (möglichst detaillierte Aussage)

Beispiel: pow

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
public static double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) {
        result *= b;
    }
    return result;
}
```


Beispiel: xor

```
// post: returns l XOR r
public static boolean xor(boolean l, boolean r) {
    return l && !r || !l && r;
}
```


Beispiel: harmonic

```
// PRE: n >= 0
// POST: returns nth harmonic number
//       computed with backward sum
public static float harmonic(int n) {
    float res = 0;
    for (int i = n; i >= 1; --i) {
        res += 1.0f / i;
    }
    return res;
}
```

Beispiel: min

```
// POST: returns the minimum of a and b
static int min(int a, int b) {
    if (a<b){
        return a;
    } else {
        return b;
    }
}
```


Fromme Lügen...

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

ist formal inkorrekt:

- Überlauf, falls e oder b zu gross sind
- b^e vielleicht nicht als double Wert darstellbar (Löcher im Wertebereich)

Fromme Lügen...sind erlaubt.

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

Die exakten Vor- und Nachbedingungen sind plattformabhängig und meist sehr kompliziert. Wir abstrahieren und geben die mathematischen Bedingungen an. $\Rightarrow$ Kompromiss zwischen formaler Korrektheit und lascher Praxis.

Prüfen von Vorbedingungen...

- Vorbedingungen sind nur Kommentare.
- Wie können wir **sicherstellen**, dass sie beim Methodenaufruf gelten?

...mit Assertions

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
public static double pow(double b, int e) {
    assert e >= 0 || b != 0 : "division by zero";
    double result = 1.0;
    ...
}
```


Nachbedingungen mit Assertions

- Das Ergebnis “komplizierter” Berechnungen ist oft einfach zu prüfen.
- Dann lohnt sich der Einsatz von assert für die Nachbedingung

```
// PRE: the discriminant p*p/4 - q is nonnegative
// POST: returns larger root of the polynomial x^2 + p x + q
static double root(double p, double q) {
    assert(p*p/4 >= q); // precondition
    double x1 = - p/2 + Math.sqrt(p*p/4 - q);
    assert(equals(x1*x1+p*x1+q,0)); // postcondition
    return x1;
}
```


Definition: Schrittweise Verfeinerung

Die schrittweise Zerlegung eines komplexen Problems in machbare Teilaufgaben. Die Lösung aller (einfachen) Teilprobleme löst das ursprüngliche komplexe Problem.

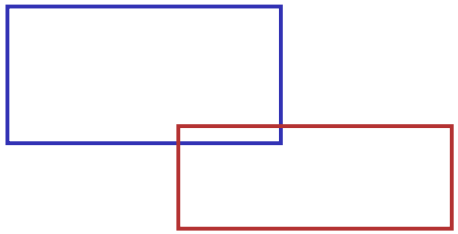
Buch auf Seite 225ff

Stepwise Refinement

- Problem wird schrittweise gelöst. Man beginnt mit einer groben Lösung auf sehr hohem Abstraktionsniveau (nur Kommentare und fiktive Methoden).
- In jedem Schritt werden Kommentare durch Programmtext ersetzt und Methoden implementiert unterteilt (demselben Prinzip folgend).
- Die Verfeinerung bezieht sich auch auf die Entwicklung der Datenrepräsentation (mehr dazu später).
- Wird die Verfeinerung so weit wie möglich durch Methoden realisiert, entstehen Teillösungen, die auch bei anderen Problemen eingesetzt werden können.
- Stepwise Refinement fördert (aber ersetzt nicht) das strukturelle Verständnis des Problems.

Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!

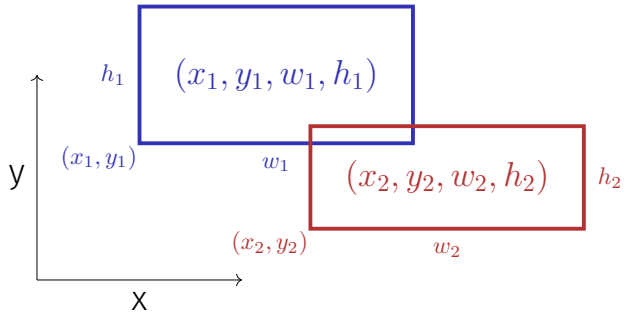


Grobe Lösung

(Include-Direktiven und Main-Klasse ausgelassen)

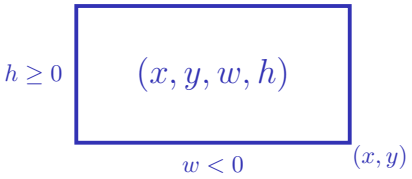
```
static void main(String args[])
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe
}
```


Verfeinerung 1: Eingabe Rechtecke



Verfeinerung 1: Eingabe Rechtecke

Breite w und/oder Höhe h dürfen negativ sein!



Verfeinerung 1: Eingabe Rechtecke

```
static void main(String args[])
{
    Out.println("Enter two rectangles [x y w h each]");
    int x1 = In.readInt();  int y1 = In.readInt();
    int w1 = In.readInt();  int h1 = In.readInt();
    int x2 = In.readInt();  int y2 = In.readInt();
    int w2 = In.readInt();  int h2 = In.readInt();

    // Schnitt?

    // Ausgabe der Loesung
}
```


Verfeinerung 2: Schnitt? und Ausgabe

```
static void main(String args[])
{
    Eingabe ✓

    boolean clash = rectanglesIntersect (x1,y1,w1,h1,x2,y2,w2,h2);

    if (clash){
        Out.println("intersection!");
    } else {
        Out.println("no intersection!");
    }
}
```


Verfeinerung 3: SchnittMethode...

```
static boolean rectanglesIntersect (int x1, int y1, int w1, int h1,
                                    int x2, int y2, int w2, int h2)
{
    return false; // todo
}

static void main(String args[]){
    Eingabe ✓
    Schnitt ✓
    Ausgabe ✓
}
```


Verfeinerung 3: SchnittMethode...

```
static boolean rectanglesIntersect (int x1, int y1, int w1, int h1,
                                    int x2, int y2, int w2, int h2)
{
    return false; // todo
}
```

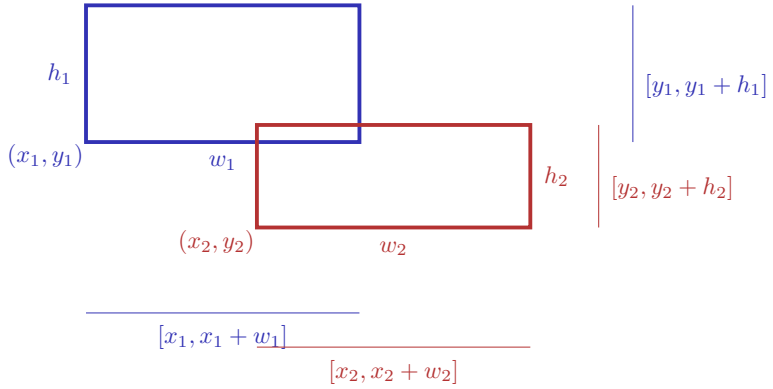
Methode main ✓

Verfeinerung 3: …mit PRE und POST!

```
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles,
//      where w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1) and
//      (x2, y2, w2, h2) intersect
static boolean rectanglesIntersect (int x1, int y1, int w1, int h1,
                        int x2, int y2, int w2, int h2)
{
    return false; // todo
}
```


Verfeinerung 4: Intervallschnitte

Zwei Rechtecke schneiden sich genau dann, wenn sich ihre x- und y-Intervalle schneiden.



Verfeinerung 4: Intervallschnitte

```
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
//      w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1),(x2, y2, w2, h2) intersect
static boolean rectanglesIntersect (int x1, int y1, int w1, int h1,
                        int x2, int y2, int w2, int h2)
{
    return intervalsIntersect (x1, x1 + w1, x2, x2 + w2)
        && intervalsIntersect (y1, y1 + h1, y2, y2 + h2); ✓
}
```


Verfeinerung 4: Intervallschnitte

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
static boolean intervalsIntersect (int a1, int b1, int a2, int b2)
{
    return false; // todo
}
```

Methode `rectanglesIntersect` ✓

Methode `main` ✓

Verfeinerung 5: Min und Max

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
static boolean intervalsIntersect (int a1, int b1, int a2, int b2)
{
    return max(a1, b1) >= min(a2, b2)
        && min(a1, b1) <= max(a2, b2); ✓
}
```


Verfeinerung 5: Min und Max

```
// POST: the maximum of x and y is returned
int max (int x, int y){
    if (x>y) return x; else return y;
}

// POST: the minimum of x and y is returned
int min (int x, int y){
    if (x<y) return x; else return y;
}
```

gibt es schon in der Standardbibliothek

Methode intervalsIntersect ✓

Methode rectanglesIntersect ✓

Methode main ✓

Nochmal zurück zu Intervallen

```
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
boolean intervalsIntersect (int a1, int b1, int a2, int b2)
{
    return Math.max(a1, b1) >= Math.min(a2, b2)
        && Math.min(a1, b1) <= Math.max(a2, b2); ✓
}
```

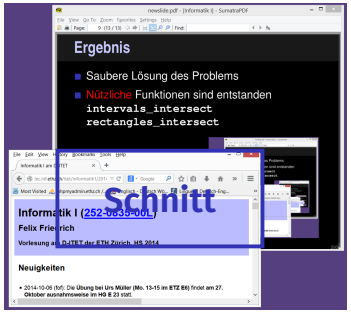

Das haben wir schrittweise erreicht!

```
class Main{
  // PRE: [a1, b1], [a2, h2] are (generalized) intervals,
  //      with [a,b] := [b,a] if a>b
  // POST: returns true if [a1, b1],[a2, b2] intersect
  boolean intervalsIntersect (int a1, int b1, int a2, int b2)
  {
    return Math.max(a1, b1) >= Math.min(a2, b2)
        && Math.min(a1, b1) <= Math.max(a2, b2);
  }

  // PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
  //      w1, h1, w2, h2 may be negative.
  // POST: returns true if (x1, y1, w1, h1),(x2, y2, w2, h2) intersect
  static boolean rectanglesIntersect (int x1, int y1, int w1, int h1,
                                      int x2, int y2, int w2, int h2)
  {
      return intervalsIntersect (x1, x1 + w1, x2, x2 + w2)
          && intervalsIntersect (y1, y1 + h1, y2, y2 + h2);
  }
```

```
  static void main(String args[])
  {
    Out.println("Enter two rectangles [x y w h each]");
    int x1 = In.readInt(); int y1 = In.readInt();
    int w1 = In.readInt(); int h1 = In.readInt();
    int x2 = In.readInt(); int y2 = In.readInt();
    int w2 = In.readInt(); int h2 = In.readInt();

    boolean clash = rectanglesIntersect (x1,y1,w1,h1,x2,y2,w2,h2);

    if (clash){
        Out.println("intersection!");
    } else {
        Out.println("no intersection!");
    }
  }
}
```

Ergebnis

- Saubere Lösung des Problems
- **Nützliche** Methoden sind entstanden
  `intervalsIntersect`
  `rectanglesIntersect`

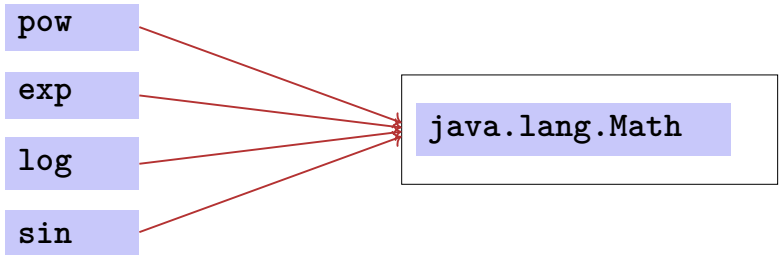


Wiederverwendbarkeit

- Methoden wie **`rectanglesIntersect`** und **`pow`** sind in vielen Programmen nützlich.
- “Lösung:” Methode einfach ins Hauptprogramm hineinkopieren, wenn wir sie brauchen!
- Hauptnachteil: wenn wir die Methodendefinition ändern wollen, müssen wir **alle** Programme ändern, in denen sie vorkommt.

Bibliotheken

- Logische Gruppierung ähnlicher Methoden



Methoden aus der Standardbibliothek

- vermeiden die Neuerfindung des Rades (wie bei **`pow`**);
- führen auf einfache Weise zu interessanten und effizienten Programmen;
- garantierten einen Qualitäts-Standard, der mit selbstgeschriebenen Methoden kaum erreicht werden kann.

Primzahltest mit `Math.sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \ldots, n-1\}$ ein Teiler von n ist.

```
int d;
for (d=2; n % d != 0; ++d);
```


Primzahltest mit `sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \ldots, n-1\}$ ein Teiler von n ist.

```
double bound = Math.sqrt(n);
int d;
for (d = 2; d <= bound && n % d != 0; ++d);
```

- Das funktioniert, weil `Math.sqrt` auf die nächste darstellbare `double`-Zahl rundet (IEEE Standard 754).