

## 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.

285

## 11. Methoden

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

286

## 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

287

## 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");  
    }  
}
```

288

## Keksrechner - Zusätzliche Anforderung

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

## Keksrechner - Eingabepprüfung

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 );
```

289

Analog dazu müssen wir prüfen, dass `kekse`  $\geq$  `kinder` ist.

290

## 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 );
```

*Anzahl Kinder einlesen und prüfen*

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

*Anzahl Kekse einlesen und prüfen*

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

```
}
}
```

291

## Keksrechner - Erkenntnisse

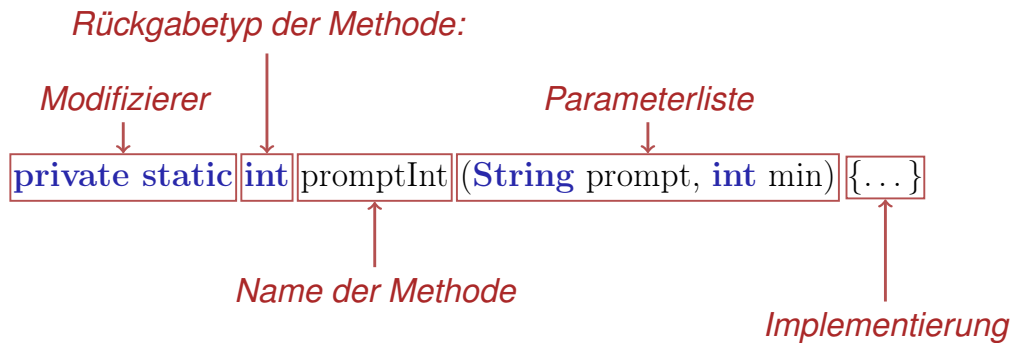
- Die beiden Code-Fragmente sind *fast identisch*
- Folgende Aspekte sind unterschiedlich:
  - Der Prompt, also "Kinder: " vs. "Kekse: "
  - Das Minimum, also "1" vs. "kinder"

Wir können das Code-Fragment in eine Methode auslagern und somit *wiederverwenden*.

Dabei müssen wir die unterschiedlichen Aspekte *parametrisieren*.

292

## Methodendeklaration und -definition



293

## Methodendeklaration und -definition

- **Modifizierer:** Werden später behandelt
- **Rückgabebetyp:** Datentyp des Rückgabewertes. Falls die Methode keinen Wert zurückliefert, ist dieser Typ `void`.
- **Name:** Ein gültiger Name. Sollte mit Kleinbuchstaben anfangen.
- **Parameterliste:** Mit runden 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.

294

## Methodensignatur

```
private static int promptInt (String prompt, int min) { ... }
```

↑  
Signatur der Methode

- 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!*
- Rückgabebetyp ist nicht Teil der Signatur! Es ist nicht möglich, mehrere Methoden zu haben, die sich nur im Rückgabebetyp unterscheiden.

295

## Methodenaufruf - Pass by Value

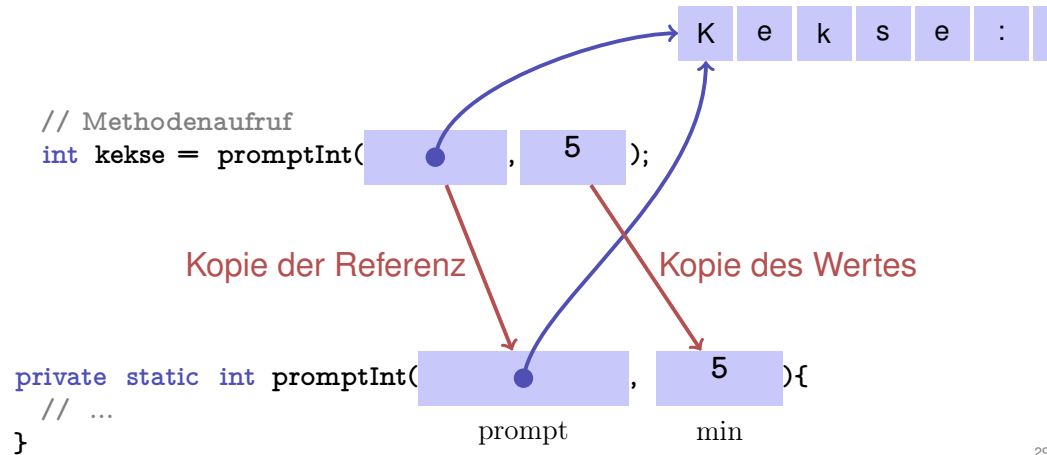
- Ein Methodenaufruf ist ein Ausdruck, dessen Wert falls vorhanden der Rückgabewert der Methode ist.
- In Java gilt immer die *Pass by Value* Semantik.

Pass by Value bedeutet: Argumentwerte werden beim Methodenaufruf in die Parameter *kopiert*.

Dies entspricht demselben Prinzip wie die Wertzuweisung an eine Variable.

296

## Methodenaufruf - Pass by Value



297

## 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;
}
```

298

## Rückgabewerte von Methoden

Es gibt zwei Fälle

- **Rückgabotyp** = void: Die Auswertung der Methode *kann* mittels der Anweisung `return` beendet werden.
- **Rückgabotyp**  $\neq$  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.

299

## 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;
    }
}
```

300

## 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) { // be = (1/b)(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) {
        result *= b;
    }
    return result;
}
```

301

302

## 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`

303

## Definition: *Vor- und Nachbedingungen*

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

304

## 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
```

305

## Nachbedingungen

Nachbedingung (postcondition):

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

Hier nur Wert, kein Effekt.

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

306

## 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  $b \neq 0$

307

## Vor- und Nachbedingungen

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

Methode `pow`: Division durch 0

308

## 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;
}
```

309

310

## Beispiel: xor

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

311

## 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;
}
```

312

## 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;
    }
}
```

313

## 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)

314

## 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.

315

## Prüfen von Vorbedingungen...

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

316



## ...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;
    ...
}
```

317

## 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 + sqrt(p*p/4 - q);
    assert(equals(x1*x1+p*x1+q,0)); // postcondition
    return x1;
}
```

318

## 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

319

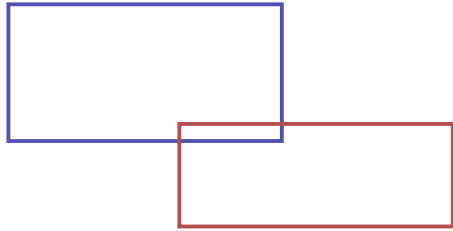
## Stepwise Refinement

- 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.

320

## Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!



## Grobe Lösung

(Include-Direktiven und Main-Klasse ausgelassen)

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

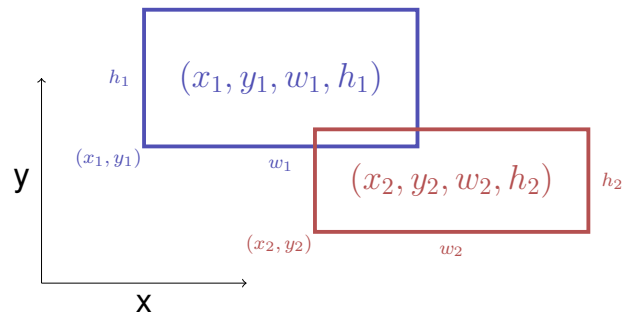
    // Schnitt?

    // Ausgabe
}
```

321

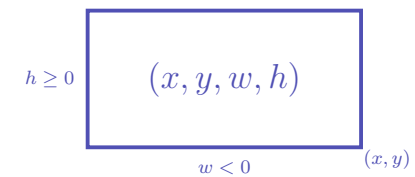
323

## Verfeinerung 1: Eingabe Rechtecke



## Verfeinerung 1: Eingabe Rechtecke

Breite  $w$  und/oder Höhe  $h$  dürfen negativ sein!



324

325

## 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
}
```

326

## 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!");
    }
}
```

327

## 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 ✓
}
```

328

## 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 ✓
```

329

## Verfeinerung 3:

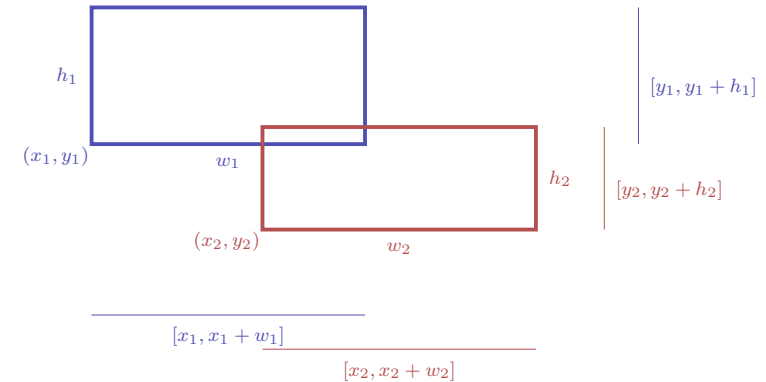
## ...mit PRE und POST!

```
// 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  
}
```

330

## Verfeinerung 4: Intervallschnitte

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



331

## 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); ✓  
}
```

332

## 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 ✓

333

## 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); ✓
}
```

334

## 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 ✓

335

## 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); ✓
}
```

336

## 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();

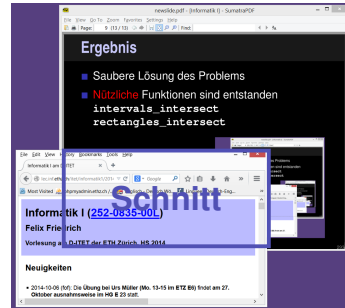
    boolean clash = rectanglesIntersect (x1,y1,w1,h1,x2,y2,w2,h2);

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

337

## Ergebnis

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



338

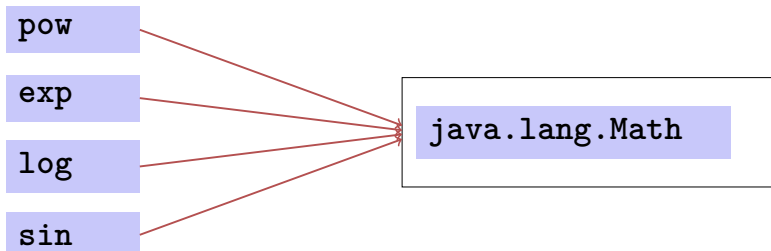
## Wiederverwendbarkeit

- 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.

339

## Bibliotheken

- Logische Gruppierung ähnlicher Methoden



340

## Methoden aus der Standardbibliothek

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

341

## Primzahltest mit `Math.sqrt`

$n \geq 2$  ist Primzahl genau dann, wenn kein  $d$  in  $\{2, \dots, n - 1\}$  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  $\{2, \dots, n - 1\}$  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).