Educational Objectives

- You can encapsulate code fragments in methods.
- You know all elements of method declarations.
- You understand what happens to the parameters upon calling a method: *pass by value*
- You can formualte *pre-* and *postconditions* for given methods.
- You can apply the *stepwise refinement* methodology.

11. Methods

Defining and Calling Methods, Evaluation of Method Calls, the Type `void`, Pre- and Post-Conditions,Stepwise Refinement,Libraries

Methods

Code fragments can be assembled in methods

Advantages:

- Define once – use several times
- clearer, more readable code, easier to comprehend
- code in methods can be tested easier

Example Cookie Calculator

```
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");
  }
}
```

Cookie Calculator – Additional Requirements

We want to make sure that **`kinder`** is positive and that each child gets at least one cookie ⇒ *check input!*

Cookie Calculator – Check Input

From this ...

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

... we go to this:

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

Analogously we have to check that **`kekse >= kinder`**.

Cookie Calculator – Getting Complicated

```
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");
  }
}
```

Read and check number of children (first block)

Read and check number of cookies (second block)

Cookie Calculator – Takeaway

- The two code fragments are *nearly identical*
- The following aspects are different:
  - The prompt, i.e. `"Kinder: "` vs. `"Kekse: "`
  - The minimum, i.e. “1” vs. “**`kinder`**”

We can outsource the code fragment into a method and thus feature *reuse*.

We have to *parameterize* the different aspects.

Declaration and definition of a Method

return type of the method:

modifiers *parameter list*

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

name of the method

implementation

Declaration and definition of a Method

- **Modifiers**: Will be treated later.
- **return type**: data type of the return value. If the method does not return a value, this type is `void`.
- **Name**: a valid name. Should be starting with a lower letter.
- **parameter list**: List of parameters surrounded by parentheses, declared by data type and name. Parameters are set when method is called can can be used like local variables.
- **implementation**: The code that is executed when the method is called.

Method Signature

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

Signature of the method

- Signature is unique within a class.
- It is thus possible to have several methods with the same name but different numbers or types of parameters. - *not recommended*
- Return type is not part of the signature! It is not possible to have several methods that are only distinguishable by their return type.

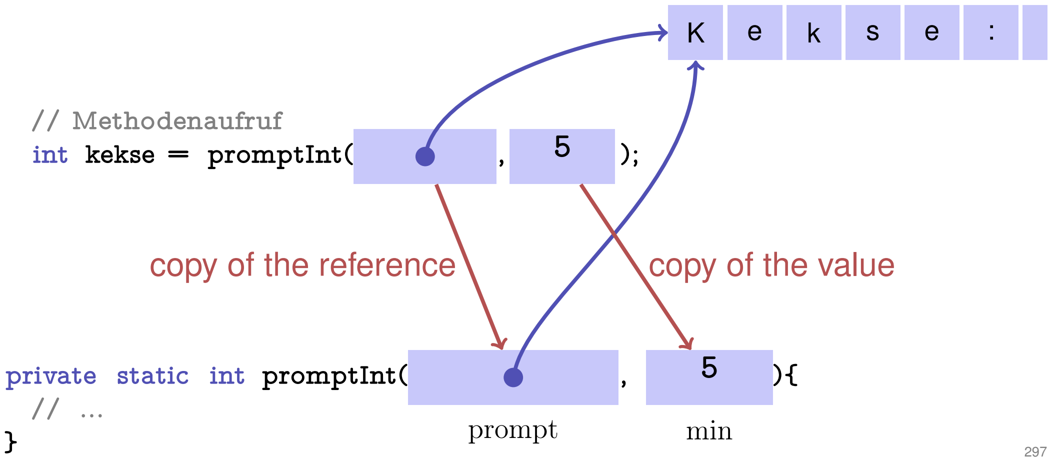
Method Call – Pass By Value

- A method call is an expression with the return value of the method.
- In Java we always have *pass by value* semantics.

Pass by value means: argument values are *copied* into the parameters upon method call.

This corresponds to the same principle as the assignment to a variable.

Method Call – Pass By Value



Back to the Example – Method `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;
}
```


Return Values of Methods

Two cases:

- **Return type** $=$ `void`: The evaluation of the method *can* be ended with the statement `return`.
- **Return type** $\neq$ `void`: The evaluation of the method *must* happen via “`return` value”. The value is passed back to the calling method.

Important: In the second case *every* possible finite execution path must contain a “`return`” statement.

Cookie Calculator – More Comprehensible

```
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;
  }
```

Pre- and Postconditions

- characterize (as complete as possible) what a Method does
- document the Method for users and programmers (we or other people)
- make programs more readable: we do not have to understand *how* the Method works
- are ignored by the compiler
- Pre and postconditions render statements about the correctness of a program possible – provided they are correct.

Example: 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;
}
```


Scope of Formal Parameters

```
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
}
```

Not the formal parameters `b` and `e` of pow but the variables defined here locally in the body of **`main`**

Definition: *Pre- and Postconditions*

“Contracts”*, that specify the behavior of a method. If the precodition holds upon calling a method, the postcondition should hold after the method’s execution.*

Preconditions

precondition:

- What is required to hold when the Method is called?
- Defines the *domain* of the Method

0^e is undefined for $e < 0$

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


Postconditions

postcondition:

- What is guaranteed to hold after the Method call?
- Specifies *value* and *effect* of the Method call.

Here only value, no effect.

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


Pre- and Postconditions

- should be correct:
- *if* the precondition holds when the Method is called *them* also the postcondition holds after the call.

Methode `pow`: works for all numbers $b \neq 0$

Pre- and Postconditions

- We do not make a statement about what happens if the precondition does not hold.

Method `pow`: division by 0

Pre- and Postconditions

- Pre-condition should by *as weak* as possible (large domain of definition)
- Post-condition should be as *strong* as possble (detailled statement)

Example: 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;
}
```


Example: xor

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


Example: 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;
}
```

Example: min

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


White Lies...

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

is formally incorrect:

- Overflow if e or b are too large
- b^e potentially not representable as a double (holes in the domain!)

White Lies are Allowed

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

The exact pre- and postconditions are platform-dependent and often complicated. We abstract away and provide the mathematical conditions. ⇒ compromise between formal correctness and lax practice.

Checking Preconditions...

- Preconditions are only comments.
- How can we ensure that they hold when the Method is called?

...with asserts

```
// 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;
    ...
}
```


Postconditions with Asserts

- The result of “complex” computations is often easy to check.
- Then the use of asserts for the postcondition is worthwhile.

```
// 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;
}
```


Definition: *Stepwise Refinement*

The stepwise breakdown of a complex problem in managable subtasks. Solving all (simple) subtasks solves the original complex problem.

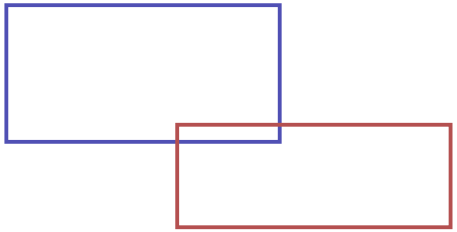
Book on page 225ff

Stepwise Refinement

- Solve the problem step-by-step. Start with a coarse solution on a high level of abstraction (only comments and abstract Method calls)
- At each step comments are replaced by program text and Methods are implemented (using the same principle again)
- The refinement also refers to the development of data representation (more later).
- If the refinement is realized as far as possible by Methods, then partial solutions emerge that might be used for other problems.
- Stepwise refinement supports (but does not replaced) the structural understanding of a problem.

Example Problem

Find out if two rectangles intersect!

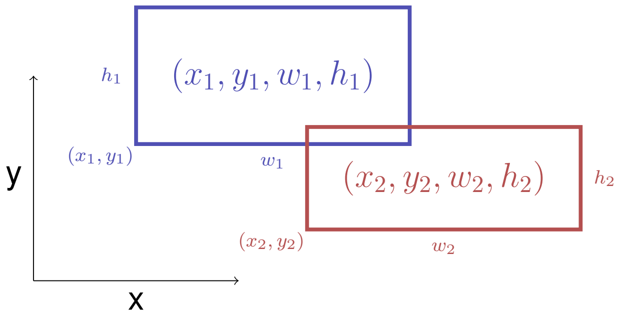


Coarse Solution

(include directives and Main class omitted)

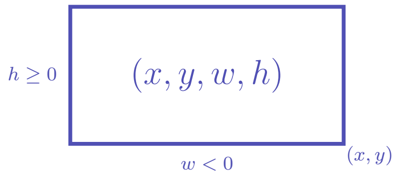
```
static void main(String args[])
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe
}
```


Refinement 1: Input Rectangles



Refinement 1: Input Rectangles

Width w and height h may be negative.



Refinement 1: Input Rectangles

```
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
}
```


Refinement 2: Intersection? and Output

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

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

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


Refinement 3: Intersection Method...

```
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[]){
    Input ✓
    Intersection ✓
    Output ✓
}
```


Refinement 3: Intersection Method...

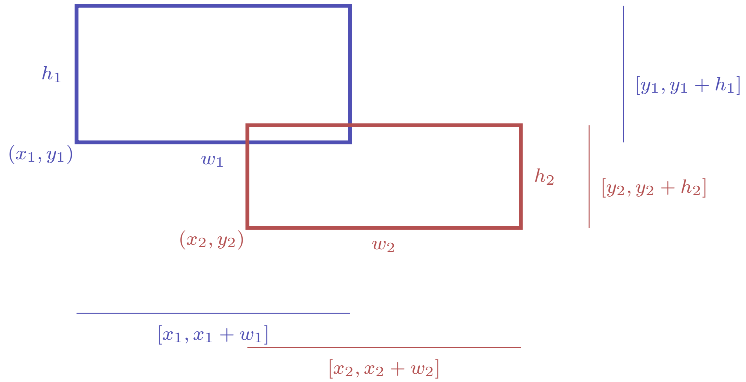
```
static boolean rectanglesIntersect (int x1, int y1, int w1, int h1,
                                    int x2, int y2, int w2, int h2)
{
    return false; // todo
}

Method main ✓
```

Refinement 3: ...with PRE and 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
}
```


Refinement 4: Interval Intersections

Two rectangles intersect if and only if their x and y-intervals intersect.



Refinement 4: Interval Intersections

```
// 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); ✓
}
```


Refinement 4: Interval Intersections

```
// 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` ✓

Refinement 5: Min and 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); ✓
}
```


Refinement 5: Min and 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;
}
```

already existing in the standard library

Methode intervalsIntersect ✓

Methode rectanglesIntersect ✓

Methode main ✓

Back to Intervals

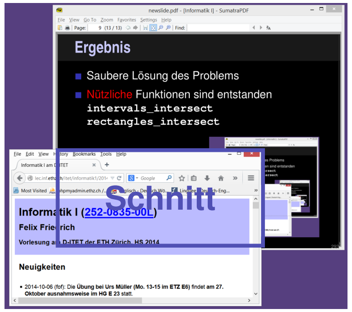
```
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
boolean intervalsIntersect (int a1, int b1, int a2, int b2)
{
   return Math.max(a1, b1) >= Math.min(a2, b2)
      && Math.min(a1, b1) <= Math.max(a2, b2);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


Look what we have Achieved in Steps!

```
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!");
    }
  }
}
```

Result

- Clean solution of the problem
- Useful Methods have been implemented
  `intervalsIntersect`
  `rectanglesIntersect`

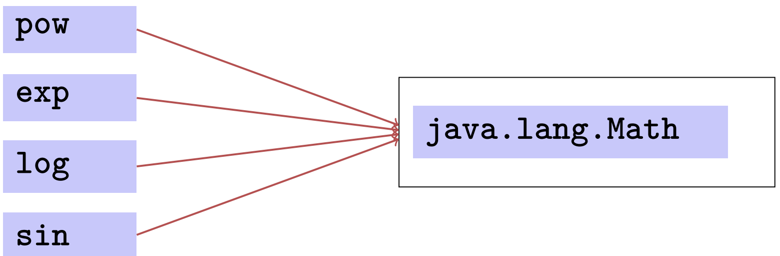


Reusability

- Methods such as `rectanges` and `pow` are useful in many programs.
- “Solution”: copy-and-paste the source code
- Main disadvantage: when the Method definition needs to be adapted, we have to change *all* programs that make use of the Method

Libraries

- Logically grouping of similar Methods



Methods from the Standard Library

- help to avoid re-inventing the wheel (such as with `pow`);
- lead to interesting and efficient programs in a simple way;
- guarantee a quality standard that can not easily be achieved with code written from scratch.

Prime Number Test with `Math.sqrt`

$n \geq 2$ is a prime number if and only if there is no d in $\{2, \ldots, n-1\}$ dividing n .

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


Prime Number test with `sqrt`

$n \geq 2$ is a prime number if and only if there is no d in $\{2, \ldots, n-1\}$ dividing n .

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

- This works because `Math.sqrt` rounds to the next representable `double` number (IEEE Standard 754).