

# ESOP - Gleitkommazahlen, Methoden und Arrays

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Repeat ..



```
/**
 * Check for primes, simple version ...
 */
public class Primes {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime; candidate++) {
            boolean isPrime = true;
            // iterate potential dividers
            for (int divider = 2; divider < candidate; divider++) {
                // check for division without rest
                if (candidate % divider == 0) {
                    isPrime = false;
                }
            }
            if (isPrime)
                System.out.println("prime = " + candidate);
        }
    }
}
```

- Find prime numbers < maxPrime

# Floating Point Numbers



- Two data types
  - float ... 32 Bit precision (24/8 in Java 8)
  - double ... 64 bit precision (53/11 in Java 8)

- Syntax

`FloatConstant = [Digits] "." [Digits] [Exponent]  
[FloatSuffix].`

`Digits = Digit {Digit}.`

`Exponent = ("e" | "E") ["+" | "-"] Digits.`

`FloatSuffix = "f" | "F" | "d" | "D".`

# Floating Point Numbers



- **Variables**
  - float x, y;
  - double z;
- **Constants**
  - 3.14                   // type double
  - 3.14f                 // type float
  - 3.14E0               //  $3.14 * 10^0$
  - 0.314E1              //  $0.314 * 10^1$
  - 31.4E-1              //  $31.4 * 10^{-1}$
  - .23
  - 1.E2                 // 100

# Harmonic Series



```
public class HarmonicSequence {  
    public static void main (String[] arg) {  
        float sum = 0;  
        int n = 10;  
        for (int i = n; i > 0; i--)  
            sum += 1.0f / i;  
        System.out.println("sum = " + sum);  
    }  
}
```

- Exchanging  $1.0f / i$  what would happen?
  - $1 / i$  ... 0 (integer division)
  - $1.0 / i$  ... a double value

# Float vs. Double



```
public class HarmonicSequence {  
    public static void main (String[] arg) {  
        float sum = 0;  
        int n = 10;  
        for (int i = n; i > 0; i--)  
            sum += 1.0f / i;  
        System.out.println("sum = " + sum);  
    }  
}
```

D:\Java\JDK\jdk1.6.0\_45\bin\java ...  
sum = 2.9289684

Process finished with exit code 0

```
public class HarmonicSequence {  
    public static void main (String[] arg) {  
        double sum = 0;  
        int n = 10;  
        for (int i = n; i > 0; i--)  
            sum += 1.0d / i;  
        System.out.println("sum = " + sum);  
    }  
}
```

D:\Java\JDK\jdk1.6.0\_45\bin\java ...  
sum = 2.9289682539682538

Process finished with exit code 0

# Assignments and Operations



- Type compatibility
  - $\text{double} \supseteq \text{float} \supseteq \text{long} \supseteq \text{int} \supseteq \text{short} \supseteq \text{byte}$
- Operators possible
  - Arithmetic operators (+, -, \*, /)
  - Comparison (==, !=, <, <=, >, >=)  
Note! Do not check floating point values for equality!

# Assignments and Casts



```
float f; int i;  
f = i;           // works  
i = f;           // does not work  
i = (int) f;     // works, but cuts after comma;  
                 // too large or too small lead to  
                 // Integer.MAX_VALUE, Integer.MIN_VALUE  
f = 1.0;         // does not work, 1.0 is type double
```



# Review: Data Types



- Integer Types: `byte`, `char`, `short`, `int`, `long`
- Floating point types: `float`, `double`
- Characters / Text: `String`
- Boolean: `boolean`

See also <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

# Review: Data Types



- Integer expressions are of type `int`
  - ie. `byte`, `short`, ...
- Floating point and scientific number expressions are type `double`
- Explicit type with suffix
  - „L“ or „l“ -> `long`
  - „d“ -> `double`
  - „f“ -> `float`

# Review: Data Types



Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

# Review: Data Types



- IDEA - DIE supports you by pointing out problems and types
- Suffix for explicit type
  - 120L // that's a long

# Methods

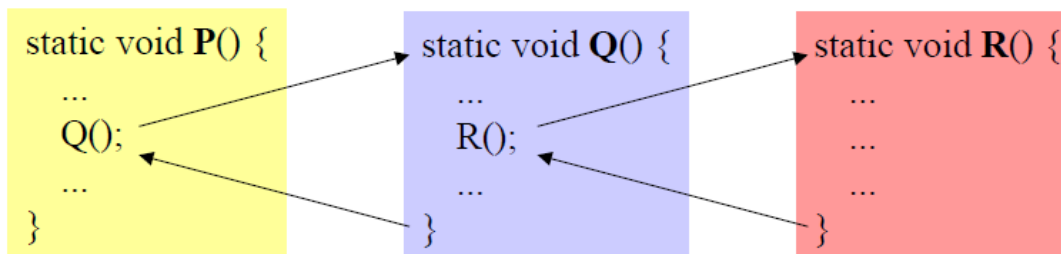


- Cp. functional programming languages
  - subroutines, functions, ...
- Goal is to re-use code
  - Code that would otherwise show up more than once.
- All in all less to write
  - less lines of code, less work
  - easier to find errors and maintain.

# Methods in Java



- We first introduce methods as subroutines.
  - .. that's a non-default interpretation.
- Name conventions for methods
  - start with verb and lower case letter
  - examples:
    - printHeader, findMaximum, traverseList, ...



# Methods in Java



```
public class SubroutineExample {  
    private static void printRule() {           // method head  
        System.out.println("-----");        // method body  
    }  
  
    public static void main(String[] args) {  
        printRule();                           // method call  
        System.out.println("Header 1");  
        printRule();  
    }  
  
}
```

D:\Java\JDK\jdk1.6.0\_45\bin\java ...

-----  
Header 1  
-----

Process finished with exit code 0

# Parameters



- Input of values supported by methods

```
class Sample {  
    static void printMax (int x, int y) {  
        if (x > y) Out.print(x); else Out.print(y);  
    }  
  
    public static void main (String[] arg) {  
        ...  
        printMax(100, 2 * i);  
    }  
}
```

formal parameters

- in the method head
- are the variables in the method body

actual parameters

- in the method call
- can be expressions



# Parameters



- Actual parameters are stored in the variables defined by the formal parameters.
- `x = 100; y = 2 * i;`
  - actual parameters need to be type compatible with the formal parameters.

```
class Sample {  
  
    static void printMax (int x, int y) {  
        if (x > y) Out.print(x); else Out.print(y);  
    }  
  
    public static void main (String[] arg) {  
        ...  
        printMax(100, 2 * i);  
    }  
}
```

# Functions



- Functions are methods that return a value.

```
class Sample {  
    static int max (int x, int y) {  
        if (x > y) return x; else return y;  
    }  
  
    public static void main (String[] arg) {  
        ...  
        int result = 3 * max(100, i + j) + 1;  
        ...  
    }  
}
```

- They have a return type, eg. int instead of void
- They use the return keyword to exit
- Can be used in expressions

# Functions vs. Procedures



- Functions
  - methods with return values
  - static `int` max (int x, int y) {...}
- Procedures
  - methods without return values
  - static `void` printMax (int x, int y) {...}

# Example



```
public class BinomialCoefficient {
    public static void main(String[] args) {
        int n = 5, k = 3;
        int result = factorial(n) /
            (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        int result = 1;
        for (int i = 2; i <= k; i++) {
            result *= i;
        }
        return result;
    }
}
```

$$\binom{n}{k} = \frac{n!}{k! \cdot (n - k)!}.$$

# Return & Rekursion



```
public class BinomialCoefficient {
    static int n = 5, k = 3;

    public static void main(String[] args) {
        int result = factorial(n) /
            (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        if (k>1) {
            return factorial(k-1)*k;
        }
        else {
            return 1;
        }
    }
}
```

- Return ends method
- Can be called at any place
- Method calling itself -> direct recursion

# Prime Numbers



```
/**
 * Primes based on function.
 */
public class PrimesWithMethod {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime;
            candidate++) {
            if (isPrime(candidate))
                System.out.println("prime = " + candidate);
        }
    }
}
```

```
public static boolean isPrime(int candidate) {
    boolean isPrime = true;
    // iterate potential dividers
    for (int divider = 2; divider < candidate; divider++) {
        // check for division without rest
        if (candidate % divider == 0) {
            isPrime = false;
        }
    }
    return isPrime;
}
```

```
/**
 * Check for primes, simple version ...
 */
public class Primes {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime;
            candidate++) {
            boolean isPrime = true;
            // iterate potential dividers
            for (int divider = 2; divider < candidate; divider++) {
                // check for division without rest
                if (candidate % divider == 0) {
                    isPrime = false;
                }
            }
            if (isPrime)
                System.out.println("prime = " + candidate);
        }
    }
}
```

# Scope of Variables



- Based on groups of statements -> blocks
  - `{ ... }`,
  - `for (int i; ...) {...}`
- A variable defined in a block is not known outside

# Example



```
public class BinomialCoefficient {  
    public static void main(String[] args) {  
        int n = 5, k = 3;   
        int result = factorial(n) /  
            (factorial(k) * factorial(n - k));  
        System.out.println("result = " + result);  
    }  
  
    public static int factorial(int k) {  
        int result = 1;  
        for (int i = 2; i <= k; i++) {  
            result *= i;  
        }  
        return result;  
    }  
}
```

Different  
variables with  
different scope



# Example: Scope



```
public class BinomialCoefficient {  
    static int n = 5, k = 3;   
  
    public static void main(String[] args) {  
        int result = factorial(n) /  
            (factorial(k) * factorial(n - k));  
        System.out.println("result = " + result);  
    }  
  
    public static int factorial(int k) {  
        int result = 1;  
        for (int i = 2; i <= k; i++) {  
            result *= i;  
        }  
        return result;  
    }  
}
```

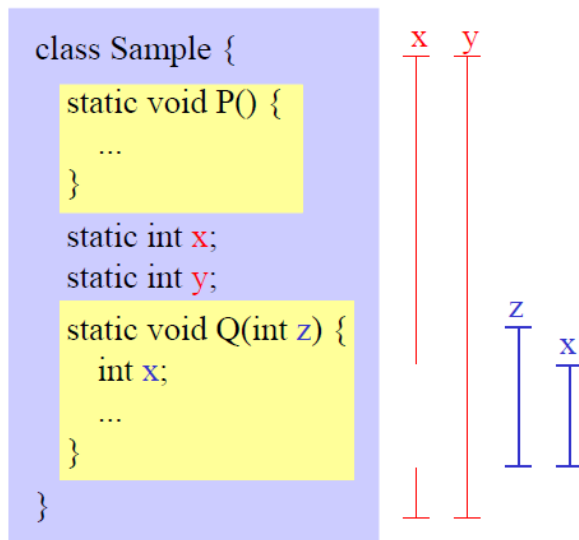
Smallest scope is  
the the actual one.

# Visibility of Names: Local Variables



## Regeln

1. A name can only be declared once within a scope.
2. locale names are prioritized over class scope names.
3. Visibility of a local name starts with ist declaration and ends with the method.
4. Variables in class scope are visible in all methods.



# Local & Static

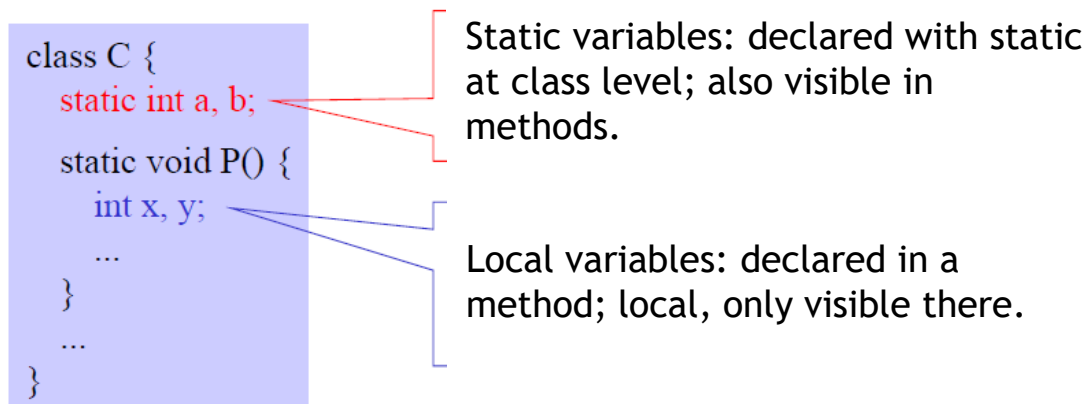


## Static Variables

- Are initialized at program start
- Are released upon program termination

## Local Variables

- Are initialized at each method call
- Are released upon termination of method.



# Locality



Best Practice: declare variables as local as possible. Don't use static unless there is no other way.

## Benefits:

- Clarity: bring together declaration and usage
- Security: Local variables can not be overwritten by other methods
- Efficiency: access to local variable is often faster

# Method Overloading



- Methods can be declared multiple times with different sets of formal parameters (difference in type, not names)

```
static void write (int i) {...}  
static void write (float f) {...}  
static void write (int i, int width) {...}
```

- At call time method implementation fitting to actual parameters is chosen.

```
write(100);    ⇒  write (int i)  
write(3.14f);  ⇒  write (float f)  
write(100, 5); ⇒  write (int i, int width)  
short s = 17;  
write(s);      ⇒  write (int i);
```

# Varargs



- In Java methods with an arbitrary number of arguments can be declared.

```
public class VarargExample {  
    public static void main(String[] args) {  
        printList("one", "two", "three");  
    }  
  
    public static void printList(String... list) {  
        System.out.println("list[0] = " + list[0]);  
        System.out.println("list[1] = " + list[1]);  
        System.out.println("list[2] = " + list[2]);  
    }  
}
```



# Arrays



- Combination of data of the same type
- Arrays have a fixed length
  - which is given at the time of instantiation
- Array variables are references
  - In Java! cp. int, float, etc. -> base types
- Access uses index values
  - first element at index 0

# One-Dimensional Arrays



- Name `a` for the whole array
- elements are accessed by their index
- indexing starts with 0
- elements are „nameless“ variables

	<code>a[0]</code>	<code>a[1]</code>	<code>a[2]</code>	<code>a[3]</code>	<code>...</code>
<code>a</code>					

## Declaration

- declares array with name and type
- length is not (yet) known

```
int[] a;  
float[] b;
```

## Instantiation

- creates a new `int` array with 5 elements
- assigns address `a`

```
a = new int[5];  
b = new float[10];
```



# Accessing Arrays



- array elements are just like variables
- index can be expression
- run time error if array is not instantiated
- run time error if index < 0 oder  $\geq$  length
- *length* is pre-defined operator
- returns number of elements

```
a[3] = 0;  
a[2*i+1] = a[i] * 3;
```

```
int len = a.length;
```

# Example



```
public class ArrayExample {  
    public static void main(String[] args) {  
        int[] myArray = new int[5];  
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}  
        for (int i = 0; i < myArray.length; i++) {  
            myArray[i] = i+1;  
        }  
        // Berechne Durchschnitt:  
        float sum = 0;  
        for (int i = 0; i < myArray.length; i++) {  
            sum += myArray[i];  
        }  
        System.out.println(sum/myArray.length);  
    }  
}
```

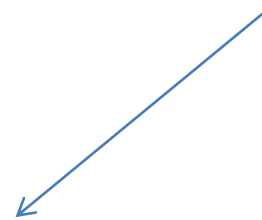
- Computes mean
- implicit cast to float!

# Example: While, For Each



```
public class ArrayExample {  
    public static void main(String[] args) {  
        int[] myArray = new int[5];  
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}  
        int i = 0;  
        while (i < myArray.length) { // while  
            myArray[i] = i+1;  
            i++;  
        }  
        // Berechne Durchschnitt:  
        float sum = 0;  
        for (int myInt : myArray) { // for each  
            sum += myInt;  
        }  
        System.out.println(sum/myArray.length);  
    }  
}
```

- Other loop constructs
- „for each“



# Example: Instantiation



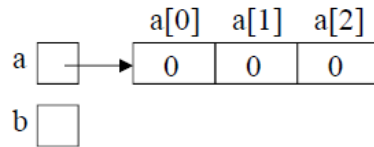
```
public class ArrayExample {  
    public static void main(String[] args) {  
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}  
        int[] myArray = {1, 2, 3, 4, 5};  
        // Berechne Durchschnitt:  
        float sum = 0;  
        for (int myInt : myArray) { // for each  
            sum += myInt;  
        }  
        System.out.println(sum/myArray.length);  
    }  
}
```

- Different way to create!

# Arrays

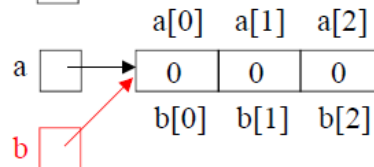


```
int[] a, b;  
a = new int[3];
```



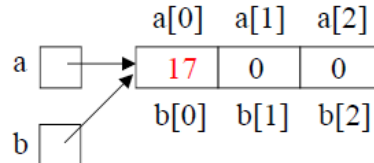
array elements in Java are initialized with 0

```
b = a;
```



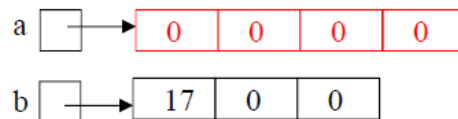
b gets the same value as a. It's a reference!!!

```
a[0] = 17;
```



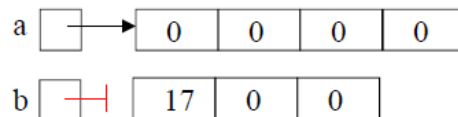
changes b[0] too!

```
a = new int[4];
```



a now points to new array.

```
b = null;
```

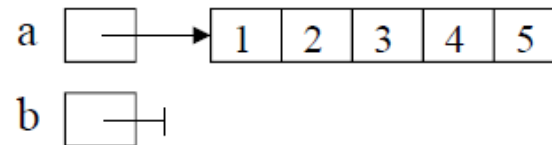


null is a special value, which can be assigned to all reference data type variables.

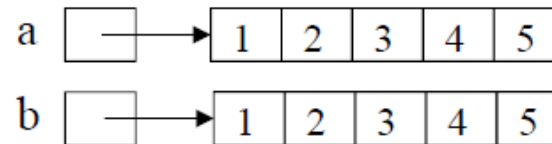
# Copying Arrays



```
int[] a = {1, 2, 3, 4, 5};  
int[] b;
```



```
b = (int[]) a.clone();
```



- Cast necessary, `a.clone()` returns type `Object[]`

# Command Line Parameters



- Calling a program with parameters
  - `java <program> par1 par2 par3 ...`
- Parameters are in a String-Array
  - `main(String[] args)` method of the program.

# Command Line Parameters



```
public class ArrayExample {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++) {  
            String arg = args[i];  
            System.out.println(arg);  
        }  
    }  
}
```

```
$> java ArrayExample one two three  
one  
two  
three
```



# Example: Linear Search



```
public class ArrayExample {  
    public static void main(String[] args) {  
        int[] myArray = {12, 2, 32, 74, 26, 42, 53, 22};  
        int query = 22;  
        for (int i = 0; i < myArray.length; i++) {  
            if (query == myArray[i]) {  
                System.out.println("Found at position " + i);  
            }  
        }  
    }  
}
```

- Each element is touch -> linear
- Needs  $n$  steps - What is the size of  $n$ ?

# Example: Sorting



- How does one sort an array  $a$ ?
- Naive approach:
  1. Create array  $b$  of the same size and type.
  2. Move minimum of  $a$  to next free position of  $b$
  3. If  $a$  is not empty start over with step 2.

# Example: Sorting



```
public class ArrayExample {
    public static void main(String[] args) {
        // o.b.d.A. a[k] > 0 & a[k] < 100
        int[] a = {12, 2, 32, 74, 26, 42, 53, 22};
        // create result array
        int[] b = new int[a.length];
        for (int i = 0; i < b.length; i++) { // set each item of b
            int minimum = 100;
            int pos = 0;
            for (int j = 0; j < a.length; j++) { // find minimum
                if (a[j] < minimum) {
                    minimum = a[j];
                    pos = j;
                }
            }
            b[i] = minimum;
            a[pos] = 100; // set visited.
        }

        for (int i = 0; i < b.length; i++) {
            System.out.print(b[i] + ", ");
        }
    }
}
```

- Can be solved in many different ways.
- Cp. AlgoDat lesson!

# Example: Eratosthenes' Sieve



```
public class Sieve {
    public static void main(String[] args) {
        int maxPrime = 200 000;
        boolean[] sieve = new boolean[maxPrime];
        // init array
        for (int i = 0; i < sieve.length; i++) {
            sieve[i] = true;
        }

        // mark the non-primes
        for (int i = 2; i < Math.sqrt(sieve.length); i++) {
            if (sieve[i] == true) { // if it is a prime
                int k = 2;
                while (k*i < sieve.length) {
                    sieve[k*i] = false;
                    k++;
                }
            }
        }

        // print results
        for (int i = 2; i < sieve.length; i++) {
            if (sieve[i]) System.out.println(i);
        }
    }
}
```

