Methoden, JavaAPI, Namensgebung, Testen, Debuggen

Lehreinheit im Javakurs der Freitagsrunde
SoSe2008
Technische Universität Berlin

Nadim El Sayed
Kai Dietrich

Agenda

1. Methoden
2. Java API
3. Namensgebung
4. Testen
5. Debuggen

Was wir bisher gelernt haben:

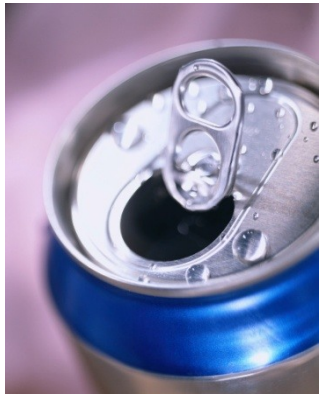
- Variablen und Arrays
- Schleifen (for, while …)
- Fallunterscheidung (if-else)
- Blöcke

Heute wollen wir diese Bausteine *strukturiert* benutzen

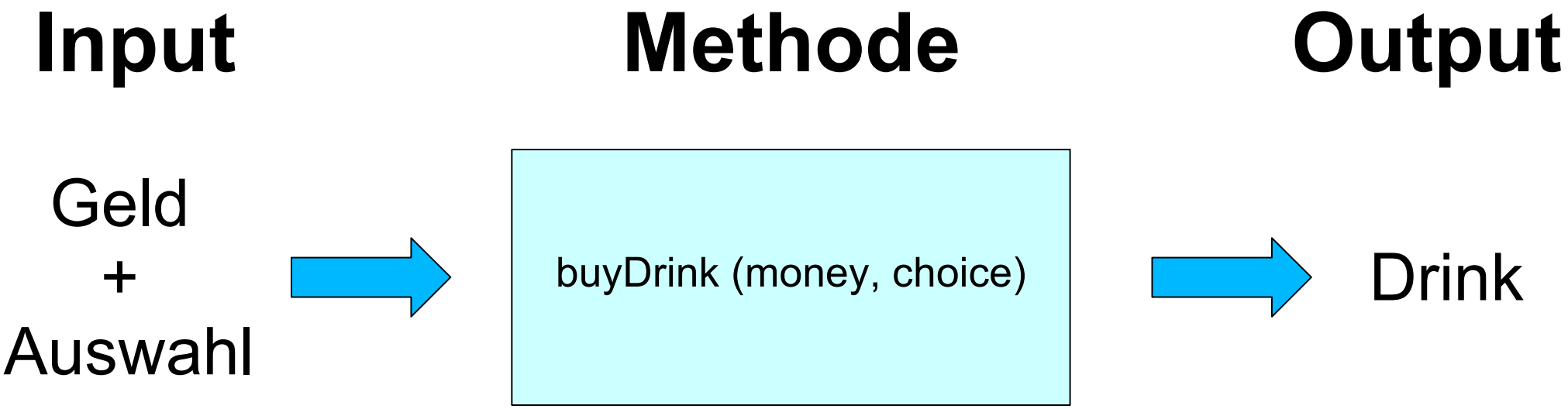
1. Methoden

Methoden - Motivation

+



Blackbox-Prinzip


Input
Methode
Output
Geld
+
Auswahl
buyDrink (money, choice)
Drink

Methoden – mathematische Analogie

abstrakt

Für unser Beispiel

Deklaration

$$f : X_1 \times X_2 \rightarrow Y$$

$$buyDrink : Money \times Choices \rightarrow Drinks$$

Definition

$$f(x_1, x_2) = x_1 \cdot x_2$$

$$buyDrink(money, choice) = \ldots$$

In Java

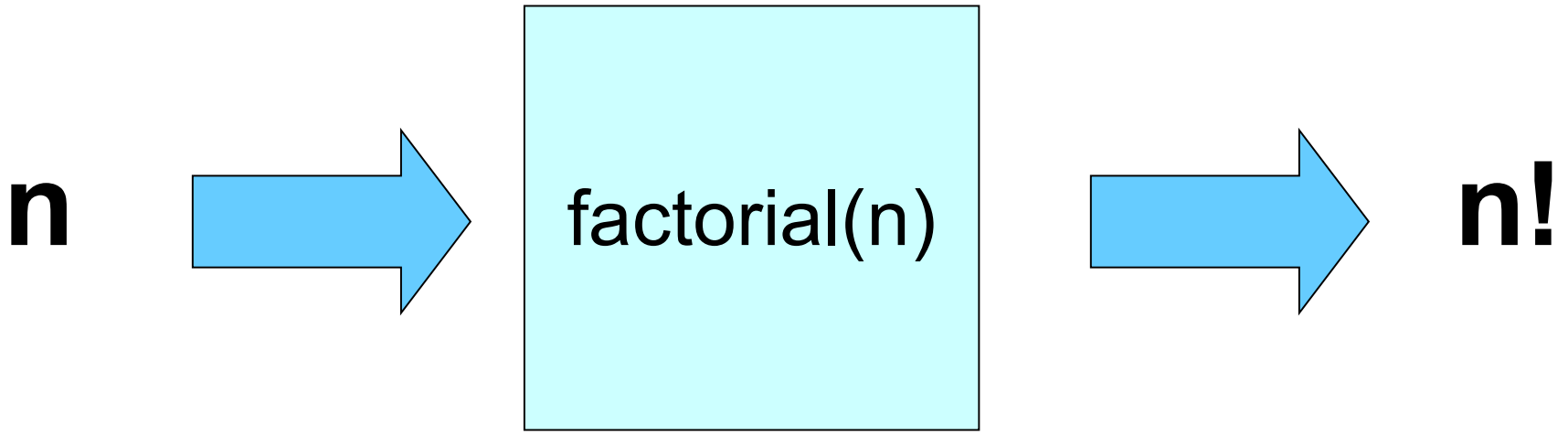
Bestandteile

Methoden bestehen aus:

- Kopf (Deklaration)
- Rumpf (Definition)

Etwas konkreten Beispiel - Fakultätsberechnung :

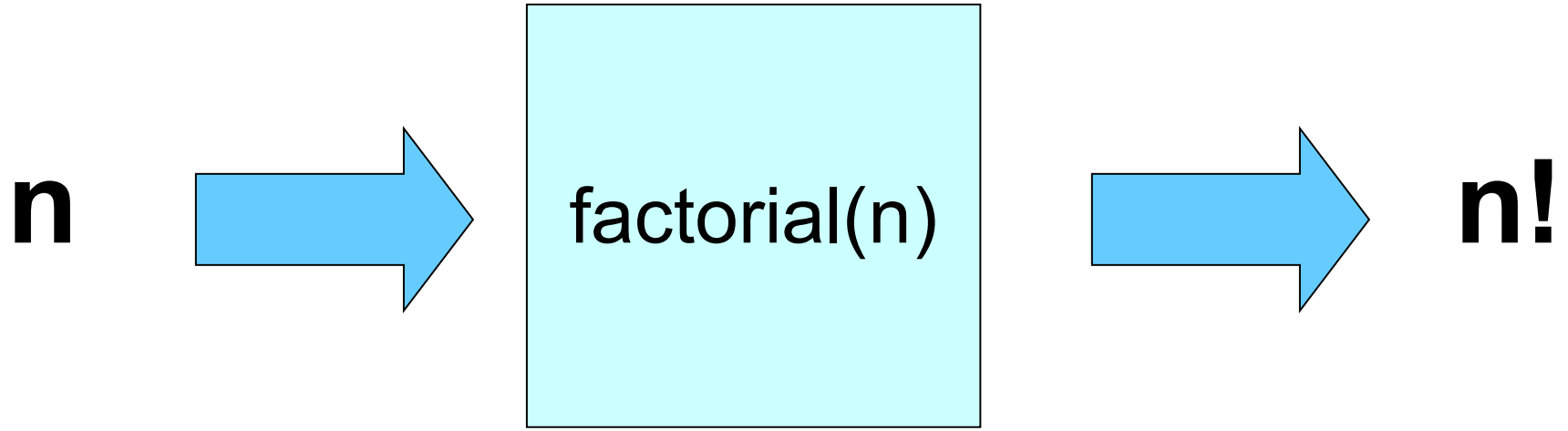
Methode



Methodenkopf hat vier Bestandsteile:

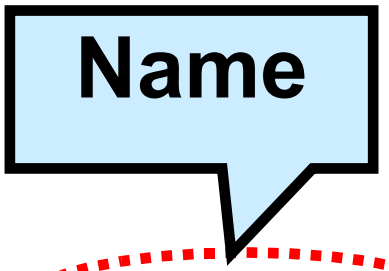
- **Name**
- **Parameterliste** **(INPUT)**
- **Rückgabetyp** **(OUTPUT)**
- **Modifikatoren**

Methode




```
public static int factorial( int n ) {
     .....
}
```

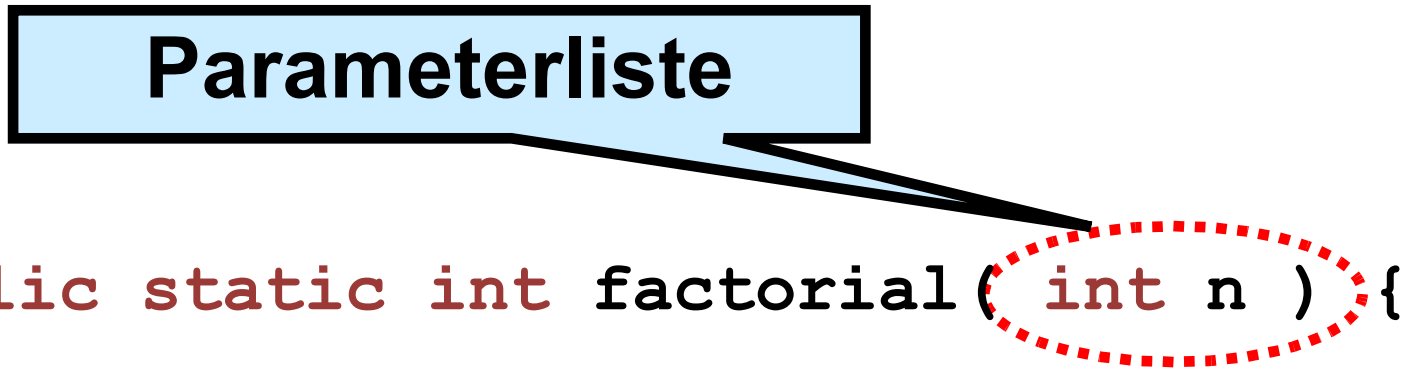
Methodenkopf - Name:




```
public static int factorial( int n ) {
     .....
}
```

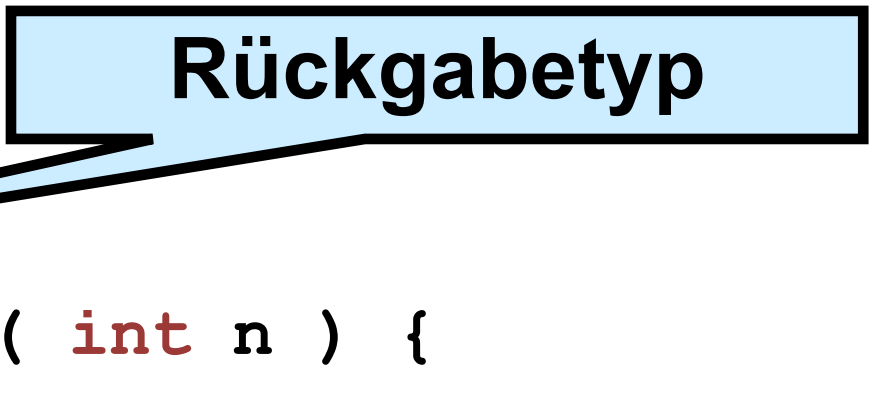
- **Der Name wird zum Aufrufen der Methode verwendet.**
- **Er sollte daher aussagekräftig sein (am besten ein Verb).**
- **Es dürfen nur Buchstaben, Zahlen und ‚_‘ verwendet werden.**

Methodenkopf - Parameter:



- **Die Eingabewerte werden in der Methode als Variablen verwendet.**
- **Mehrere Parameter werden durch Kommas getrennt.**
- **Es ist auch möglich auf Parameter zu verzichten.**

Methodenkopf - Rückgabe:




```
public static int factorial( int n ) {
     .....
}
```

- **Definiert den Typ der Rückgabe: boolean, int, double etc.**
- **Nur eine Variable kann zurückgegeben werden.**
- **Falls nichts zurückgegeben werden soll, schreibt man „void“.**

Methodenkopf - Modifikatoren:

```
public static int factorial( int n ) {
    .....
}
```

Modifikatoren

- **Legen die Art (und Sichtbarkeit) der Methode fest.**
- **Wir schreiben zunächst nur „ public static “.**
- **Mehr dazu im OOP Kapitel (LE 5 & 6).**

Methodenrumpf (body):

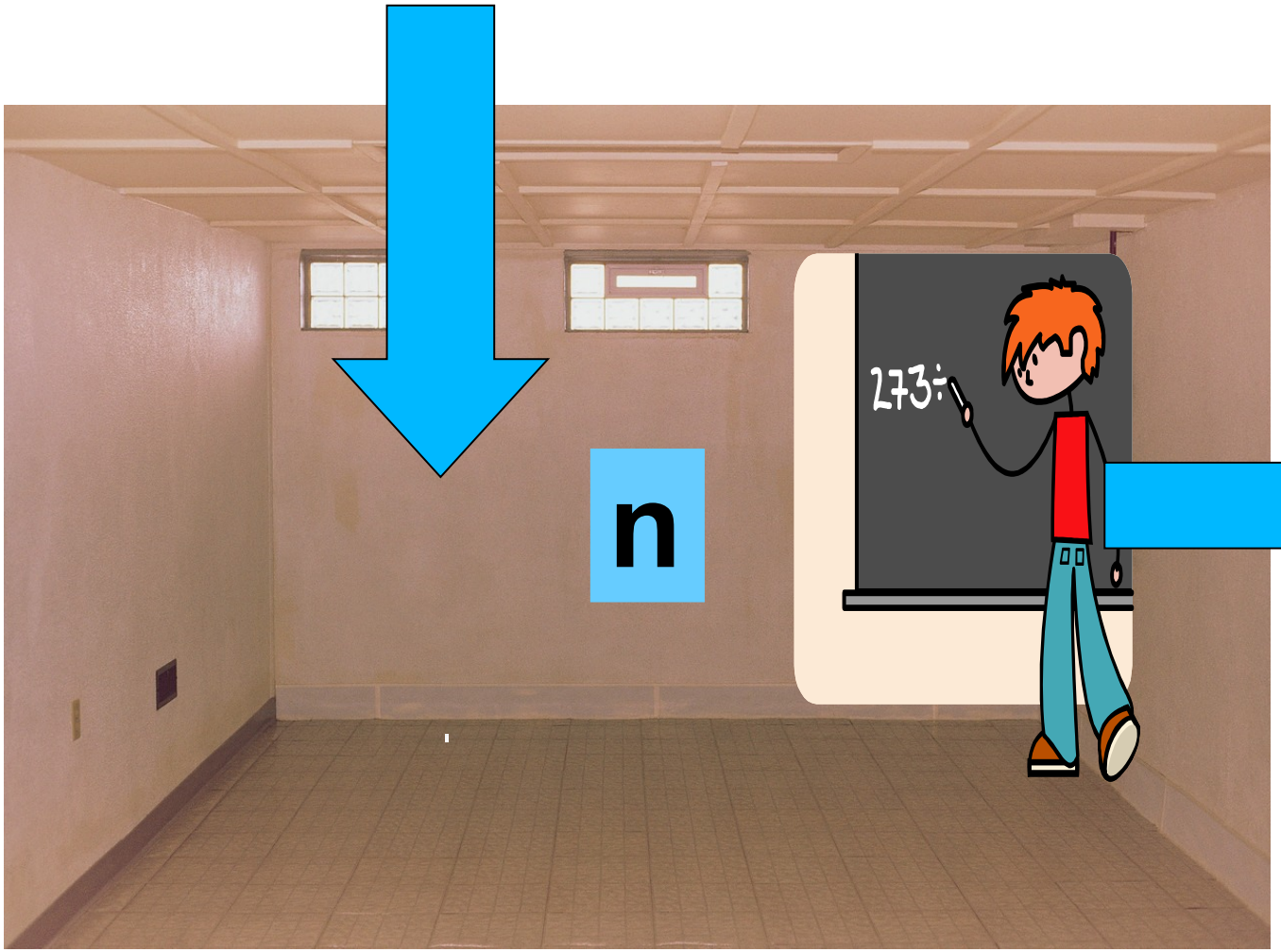
- **Wird zwischen {} eingeschlossen und steht nach dem Kopf.**

- **Enthält die Anweisungen, die (von den Parametern) zur Rückgabe führen.**

- **Die übergebenen Parameter werden innerhalb des Rumpfs als Variablen behandelt.**

Inside

Input



Output

n!

Ziel:

Wie Kann man aus einer ganzen Zahl ihre Fakultät berechnen ?

Man nehme folgende Variablen zu Hilfe :

- Platzhalter fürs Ergebnis: „nFak“
- Ein Zähler „ x “

Eine allgemeine Lösung:

Am Anfang:

nFak und x auf eins setzen .

wiederhole:

1. nFak mit x multiplizieren.
2. x um eins erhöhen.

Wie oft ?

Am Ende: nFactorial = 1*2*3*....*n

```
public static int factorial (int n) {
 // Am Anfang
 int nFak = 1;
 int x = 1;
 // Wiederholung
 while (x <= n) {
   nFak = nFak * counter;
   x = x + 1;
 }
 // Ende
 return nFak;
}
```

```
public static int factorial (int n) {
  // Am Anfang
  int nFak = 1;
  // Wiederholung
  for(int x = 1; x <= n; x++) {
    nFak = nFak * x;
  }
  // Ende
  return nFak;
}
```

Siehe LE 2.

Zusammenhänge

```
public static int factorial( int n ) {

    int nFak = 1;

    for( int x = 1 ; x <= n ; x++){
          // calculate n factorial
          nFak = nFak * x;
    }

    return nFak;
}
```

Wie geben wir die Ergebnisse zurück? :

```
public static int factorial( int n ) {
     int nFak = 1;
     for( int x = 1 ; x <= n ; x++){
            // calculate n factorial
            nFak = nFak * x;
     }
     return nFak;
}
```

- Die return Anweisung beendet die Ausführung.
- Nach return steht ein Ausdruck des Rückgabetyps.
- Falls keine Rückgabe (void), lässt return weg.

Was bedeutet

```
public static void main( String[] args )
```

nun ?

Weiterführenden Beispiel :

- **Binomialkoeffizienten berechnen**

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

```
public static void main( String[] args ) {
    int n = 49, k = 6;
    // init variables
    int nFac = 1 , kFac = 1 , nMinusKFac = 1;
    for( int i= 1 ; i <= n ; i++){     // calculate n factorial
      nFac = nFac * i;
    }
    for( int i= 1 ; i <= k ; i++){     // calculate k factorial
       kFac = kFac * i;
    }
    for( int i = 1 ; i <= n-k ; i++){
       // calculate (n-k) factorial
       nMinusKFac = nMinusKFac * i;
    }
    int result = nFac/(nMinusKFac * kFac);
    // calculate n choose k
    System.out.println(n + " ueber " + k + " ist " + result );
 }
```

Verbesserung:

```
public static int factorial( int n ) {
    int nFactorial = 1;
    for( int i = 1 ; i <= n ; i++){    // calculate n factorial
       nFactorial = nFactorial * i;
    }
    return nFactorial;
}
```

- **Unser Code sieht nun viel einfacher aus:**

```
public static void main( String[] args ) {
    int n = 49, k = 6;
    int nFac = factorial( n ); //calculate n factorial
    int kFac = factorial( k ); //calculate k factorial
    int nMinusKFac = factorial( n-k ); // (n-k) factorial

    int result = nFac/(nMinusKFac * kFac); // n choose k
    System.out.println(n + " ueber " + k + " ist " + result );
}
```

Vorteile von Methoden :

- **Code wird lesbarer und robuster.**
- **Aufruf unabhängig von der Implementierung.**
- **Implementierung kann nachträglich geändert oder verbessert werden bei gleichem Aufruf.**
- **Für größere Projekte ist das eine Möglichkeit der Arbeitsteilung.**
- **Testen einzelner Methoden des Programms möglich.**

Resumé :

- **Kopf**
  - **Besteht aus Name, Parametern, Rückgabetyp.**
  - **WAS für eine Methode ist es? WAS macht sie ?**
  - **Dem Benutzer muss das auf einen Blick klar sein!**

- **Rumpf**
  - **Beinhaltet die Implementierung (Menge v. Anweisungen).**
  - **WIE macht die Methode das, was sie soll?**
  - **Für den Benutzer nicht zwingend relevant.**

- **Einen korrekten Kopf zu schreiben bringt euch bereits Punkte in der Klausur!**

Beispiele

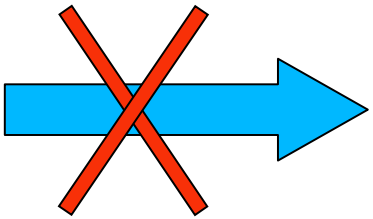
Keine Rückgabe

Automat leer

Zustandsänderung

fill (drinks) ;

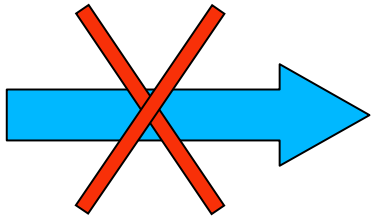
Getränke



Automat voll

Keine Parameter

purse = cash( ) ;



Geld

Call-by-Value :

```
public static void main( String[] args ) {
    int n = 5;
    System.out.println( "n ist: " + n );
    setToZero( n );
    System.out.println( "n ist: " + n );
}

// Method changes <number> to zero
public static void setToZero( int number ) {
    number = 0;
}
```

```
Ausgabe:

    n ist: 5
    n ist: 5
```

- **Die Variable wird nicht verändert, da die Methode mit einer Kopie arbeitet.**
- **Das gilt aber nur für primitive Datentypen wie int, double, boolean usw. (in Java immer kleingeschrieben)**

Call by reference

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

    int[] numArr = new int[1];
    numArr[0] = 7;
    System.out.println( "[0] ist: " + numArr[0] );
    setToZero( numArr );
    System.out.println( "[0] ist: " + numArr[0] );
}

// Method changes <number> to zero
public static void setToZero( int[] numArray ) {
    numArray[0] = 0;
}
```

```
Ausgabe:

    [0] ist: 7
    [0] ist: 0
```

- **Das Array wird verändert, da es kein primitiver Datentyp ist.**

- **Bei nicht-primitiven Datentypen wird nicht kopiert.**

- **Rekursion funktioniert auch in Java!**

```
public static int factorial( int n ) {
    if( n <= 1 ){
        return 1;
    }
    return n * factorial( n - 1 );
}
```

2. Java API

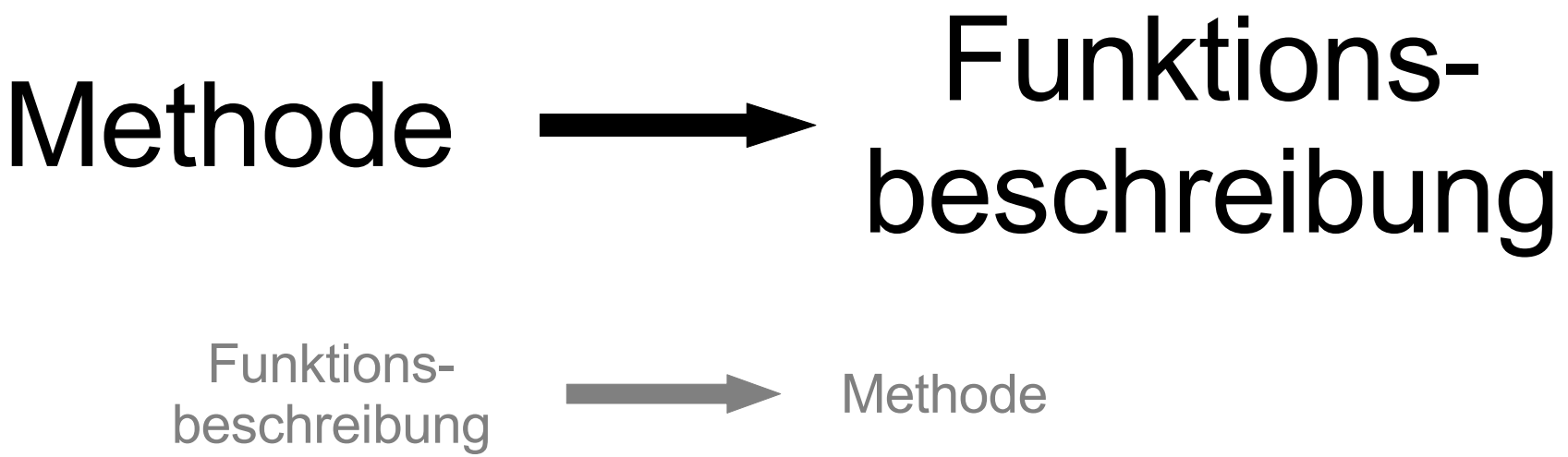
Ihr kennt schon einige Standardmethoden...
z.B.:

```
System.out.println(…);
Math.random();
```

Wo gibt es mehr?
Und wie benutzt man sie?

Java API

http://java.sun.com/j2se/1.5.0/docs/api/


Methode
Funktions-
beschreibung
Funktions-
beschreibung
Methode

java.lang

Class Math

java.lang.Object
  └ **java.lang.Math**

```
public final class Math
extends Object
```

The class `Math` contains methods for performing basic numeric operations s as the elementary exponential, logarithm, square root, and trigonometric func ns.

Unlike some of the numeric methods of class `StrictMath`, all implementation the equivalent functions of class `Math` are not defined to return the bit-for-bi same results. This relaxation permits better-performing implementations w re strict reproducibility is not required.

By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of `Math` methods. ch higher-performance implementations still must conform to the specification or `Math`.

Java™ 2 Platform Standard Ed. 5.0
All Classes
Packages
java.applet
java.awt
ManagementFactory
ManagementPermissi
ManageReferralContr
ManagerFactoryPara
Manifest
Map
Map.Entry
MappedByteBuffer
MARSHAL
MarshalException
MarshalledObject
MaskFormatter
Matcher
MatchResult
Math
MathContext
MatteBorder
MBeanAttributeInfo
MBeanConstructorInf
MBeanException
MBeanFeatureInfo

approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

Since:

JDK1.0

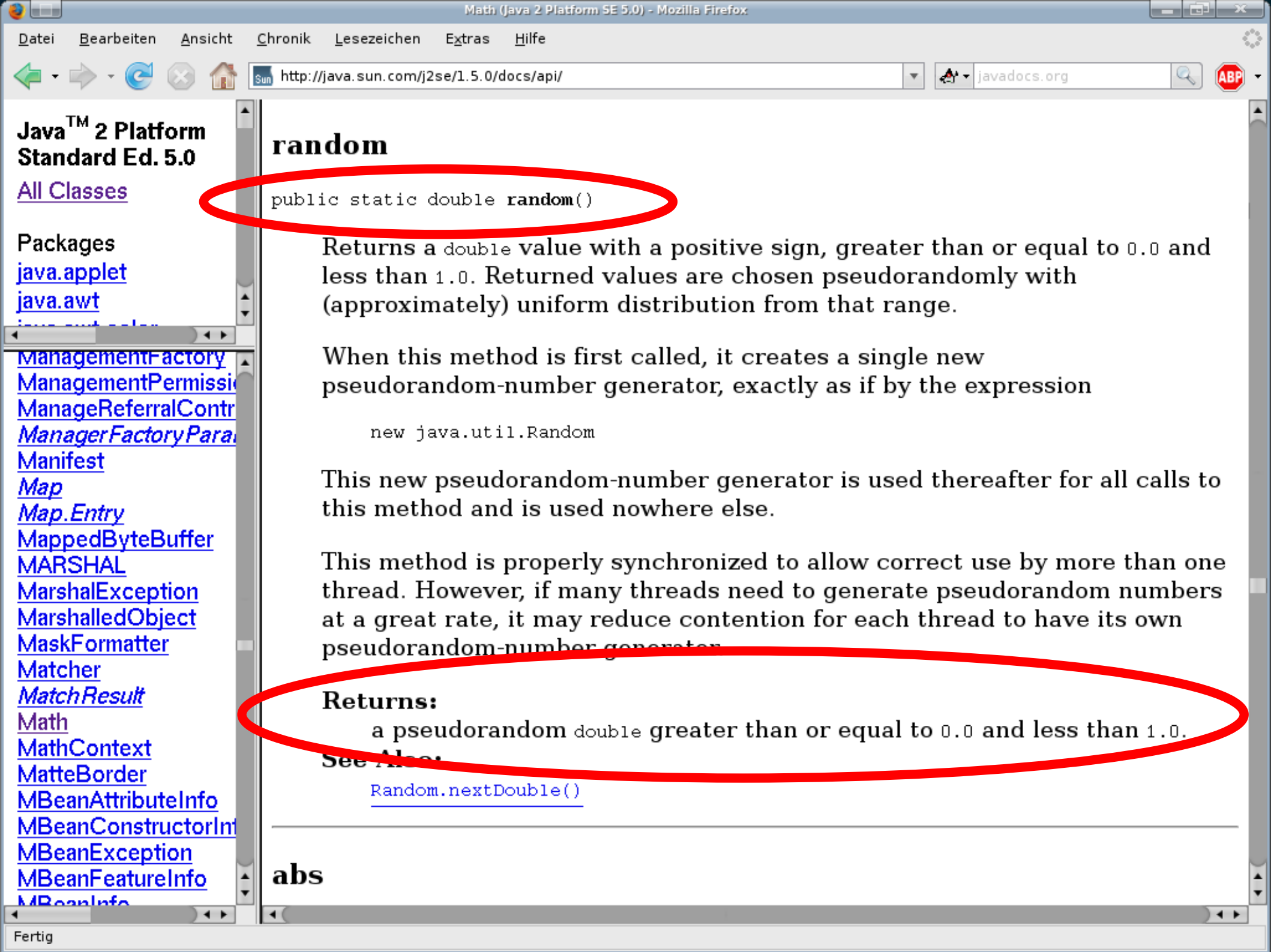
Field Summary

static double	**E** The `double` value that is closer than any other to *e*, the base of the natural logarithms.
static double	**PI** The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary

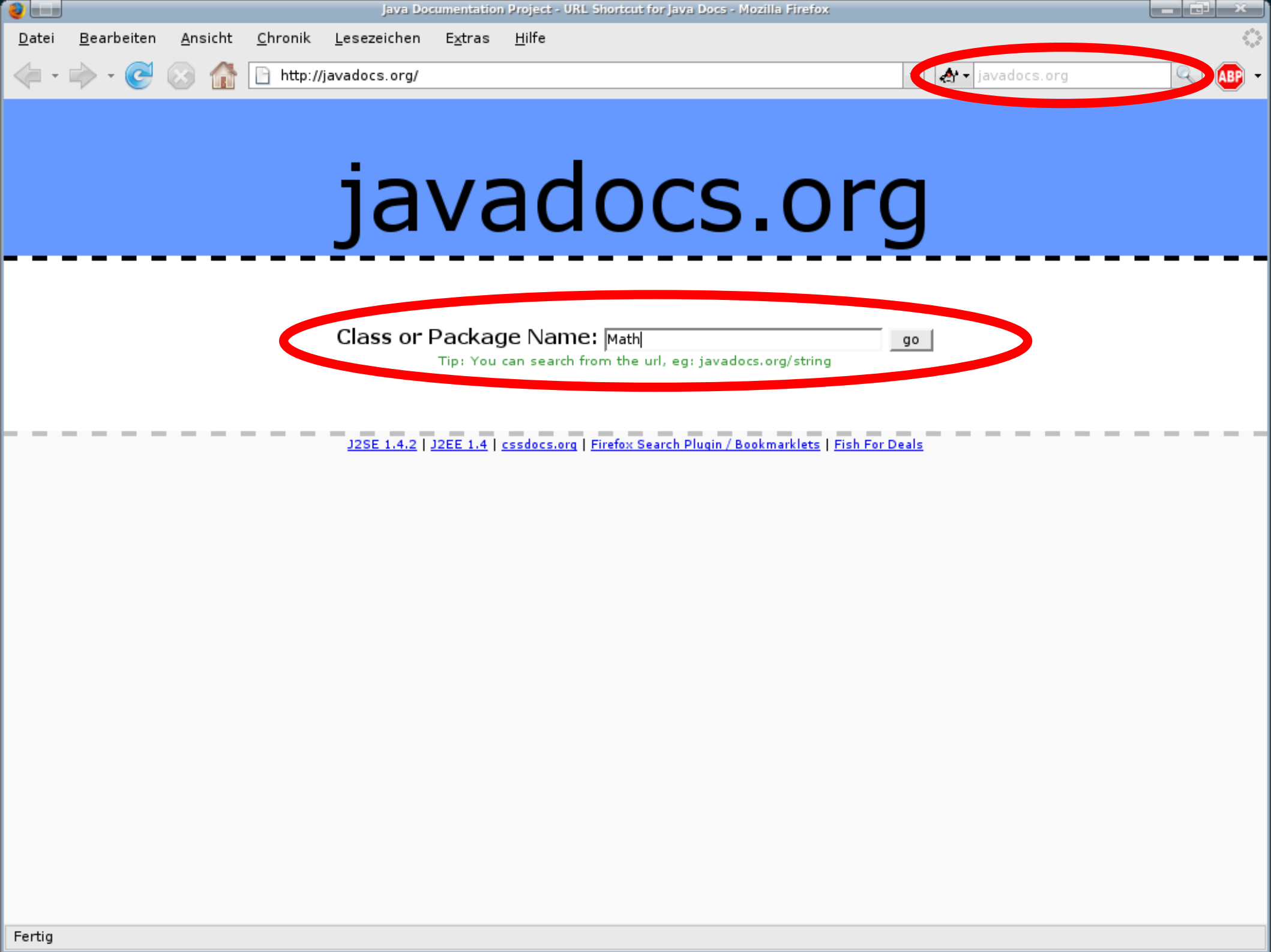
static double	**abs**(double a) Returns the absolute value of a `double` value.
static float	**abs**(float a) Returns the absolute value of a `float` value.
static int	**abs**(int a) Returns the absolute value of an `int` value.
static long	**abs**(long a) Returns the absolute value of a `long` value.

	Returns the smaller of two double values.
static float	min(float a, float b) Returns the smaller of two float values.
static int	min(int a, int b) Returns the smaller of two int values.
static long	min(long a, long b) Returns the smaller of two long values.
static double	pow(double a, double b) Returns the value of the first argument raised to the power of the second argument.
static double	random() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
static double	rint(double a) Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
static long	round(double a) Returns the closest long to the argument.
static int	round(float a) Returns the closest int to the argument.
static double	signum(double d) Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
static float	


Math (Java 2 Platform SE 5.0) - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://java.sun.com/j2se/1.5.0/docs/api/
javadocs.org
Java™ 2 Platform Standard Ed. 5.0
All Classes
Packages
java.applet
java.awt
ManagementFactory
ManagementPermissi
ManageReferralContr
ManagerFactoryPara
Manifest
Map
Map.Entry
MappedByteBuffer
MARSHAL
MarshalException
MarshalledObject
MaskFormatter
Matcher
MatchResult
Math
MathContext
MatteBorder
MBeanAttributeInfo
MBeanConstructorInf
MBeanException
MBeanFeatureInfo
random
public static double random()
Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.
When this method is first called, it creates a single new pseudorandom-number generator, exactly as if by the expression
new java.util.Random
This new pseudorandom-number generator is used thereafter for all calls to this method and is used nowhere else.
This method is properly synchronized to allow correct use by more than one thread. However, if many threads need to generate pseudorandom numbers at a great rate, it may reduce contention for each thread to have its own pseudorandom-number generator.
Returns:
a pseudorandom double greater than or equal to 0.0 and less than 1.0.
See Also:
Random.nextDouble()
abs
Fertig

Java API

www.javadocs.org


Java Documentation Project - URL Shortcut for Java Docs - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://javadocs.org/
javadocs.org
javadocs.org
Class or Package Name: Math
go
Tip: You can search from the url, eg: javadocs.org/string
J2SE 1.4.2 | J2EE 1.4 | cssdocs.org | Firefox Search Plugin / Bookmarklets | Fish For Deals
Fertig

Fragen?

3. Namensgebung und Kommentare

Ein einfacher Grundsatz:

Code ist immer für andere Menschen

Variablen und Methoden brauchen Namen

Kommentare kann man (wirklich) benutzen

Was macht diese Methode?

```
public static boolean myMethod(int a) {

    if (a == 0){
        return true;
    }else if (a > 0) {
        boolean boolValue = myMethod(a - 1);

        // boolValue is the result of myMethod(a - 1)?
        return !boolValue;
    }else {
        boolean boolValue = myMethod(a + 1);

        // we return the complement of boolValue
        return !boolValue;
    }
}
```

Was macht diese Methode?

```
public static boolean isEven( int num ) {

    if (num == 0){
        return true;
    }else if (num > 0) {
        boolean predecessorEven = isEven(num - 1);

        // predecessor is even <=> number is not even
        return !predecessorEven;
    }else {
        boolean successorEven = isEven(num + 1);

        // successor is even <=> number is not even
        return !successorEven;
    }
}
```

Namensgebung und Kommentare

	Dos	**Don'ts**
Kommentare	▪ Schwer ersichtliche Gedankengänge erläutern. ▪ Bedeutung und Anwendung von komplexen Methoden/Klassen erklären ▪ Den generellen Ablaufs des Programms erläutern	▪ beschreiben, **was** man tut und nicht **warum** man es tut ▪ belanglose und sich wiederholende Kommentare schreiben
Namen	▪ Namen wählen, die selbsterklärend sind und ihre Funktion repräsentieren ▪ unbedingt camelCase verwenden	▪ Namen auf Deutsch. ▪ zu lange Namen. ▪ zu kurze Namen ▪ camelCase misachten

4. Testen

Testen

„Unser Code ist immer korrekt!“
(Winzigweich Corp., QA Abteilung)

Testen ist Pflicht um gute Software herzustellen, denn Fehler treten immer auf, egal wie gut (oder eingebildet) man ist.

UND

Frühes, konsequentes Testen hilft Fehler schneller zu finden

UND

Testen

Was sind die Hauptziele beim Testen von Software?

1. Funktionalität testen:
   - Tut das Programm das, was es soll?
   - Funktioniert es auch für sinnlose Eingaben?

2. Stabilität testen:
   - Terminiert das Programm für alle Eingaben?

Am Beispiel:

$$pow(base, exp) = base^{exp}$$

Testen

Wir sollen eine pow-Methode schreiben...

1. Schritt: Denken

```
class Powerizer {

    // returns <base> to the power of <exp>
    public static double pow(int base, int exp) {
        // ...
    }

}
```

2. Schritt: Test-Methode schreiben

```
class Powerizer {

    public static void main(String args[]) {
        testPow();
    }

    public static void testPow() {
        // TODO: call pow(...) with lot's of parameters
    }

    public static double pow(int base, int exp) {
        // ...
    }

}
```

Testen

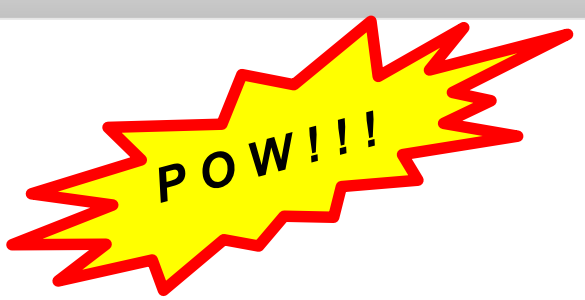
Testfälle: 2^{8}, 3^{2}, 42^{1}, 0^{1}, 0^{0}, 1^{0}, 2^{-8}, 0^{-1}, 1^{-1}

```
public static void testPow() {
    System.out.println("2^8 = " + pow(2,8) + " (256 erwartet)");
    System.out.println("3^2 = " + pow(3,2) + " (9 erwartet)");
    System.out.println("42^1 = " + pow(42,1) + " (42 erwartet)");

    System.out.println("0^1 = " + pow(0,1) + " (0 erwartet)");
    System.out.println("0^0 = " + pow(0,0) + " (1 erwartet)");
    System.out.println("1^0 = " + pow(1,0) + " (1 erwartet)");

    System.out.println("2^-8= " + pow(2,-8)+ " (0.003906 erwartet)");
    System.out.println("0^-1 = " + pow(0,-1) + " (n.d. erwartet)");
    System.out.println("1^-1 = " + pow(1,-1) + " (1 erwartet)");
}
```

Testen

3. Schritt: 

programmieren

```
// returns <base> to the power of <exp>
public static double pow(int base, int exp) {
    double out = 1;

    // compute a^n = 1 * (a * a * ... * a)
    for(int i=1; i<=exp; i++) {
        out = out * base;
    }

    return out;
}
```

Testen

Ausgabe:

```
kai@lankiveil ~/Javakurs/LE3/code $ java Powerizer
2^8 = 256.0 (256 erwartet)
3^2 = 9.0 (9 erwartet)
42^1 = 42.0 (42 erwartet)
0^1 = 0.0 (0 erwartet)
0^0 = 1.0 (1 erwartet)
1^0 = 1.0 (1 erwartet)
2^-8 = 1.0 (0.00390625 erwartet)
0^-1 = 1.0 (n.d. erwartet)
1^-1 = 1.0 (1 erwartet)
```

Hmmm ...

negative Exponenten werden wohl nicht richtig unterstützt ...

Testen

pow-improved:

```
// returns <base> to the power of <exp>
public static double pow(int base, int exp) {
    double out = 1;

    if(exp>=0) {
        //normal computation for positive exponents

        // a^n = 1 * (a * a * ... * a)
        for(int i=1; i<=exp; i++) {
            out = out * base;
        }
    } else {
        //compute the reciprocal for negative exponents
        out = 1 / pow(base, -exp);
    }

    return out;
}
```

Testen

Ausgabe:

```
kai@lankiveil ~/Javakurs/LE3/code $ java Powerizer
2^8 = 256.0 (256 erwartet)
3^2 = 9.0 (9 erwartet)
42^1 = 42.0 (42 erwartet)
0^1 = 0.0 (0 erwartet)
0^0 = 1.0 (1 erwartet)
1^0 = 1.0 (1 erwartet)
2^-8 = 0.00390625 (0.00390625 erwartet)
0^-1 = Infinity (n.d. erwartet)
1^-1 = 1.0 (1 erwartet)
```

Besser!

:-)

Merke:

- Grenzfälle testen! (Kehrwert von 0)
- Unlogische Werte testen! (Fakultät einer neg. Zahl)
- Viel hilft viel!
- Nach der Korrektur immer nochmals testen!

5. Debuggen

Println-Debugging

- Ausgeben von Variablen und Meldungen zur Laufzeit mit `System.out.println();`
- Effektives Vorgehen zur Fehlersuche

Debuggen

Gegeben sei folgende Methode

```
//calculates <num> modulo <div>
public static int mod(int num, int div) {

    while(num > div) {
        num = num - div;
    }

    return num;
}
```

Für div<0 entsteht eine Endlosschleife

Debuggen

Nehmen wir mal an, wir sehen den Fehler nicht...

bei einem Test versucht man mod(4,-1) zu berechnen...

Stunden später: Das Programm läuft immer noch.

Und nun? - Einfügen von Status-Ausgaben...

```
//calculates <num> modulo <div>
public static int mod(int num , int div) {
    System.out.println("mod(" + num + ", " + div + ") aufgerufen");

    while(num > div) {
        num = num - div;
        System.out.println("mod: num = " + num);
    }
    System.out.println("mod: Berechnung abgeschlossen, num = " + num);

    return num;
}
```

Debuggen

Ausgabe:

```
kai@lankiveil ~/Javakurs/LE3/code $ java Modulo
mod(4, -1) aufgerufen
mod: num = 5
mod: num = 6
mod: num = 7
...
mod: num = 64168
mod: num = 64169
mod: num = 64170
...
Strg+C
```

Aha!!! :)

Hinweise:

- Ausgabe aller relevanten Variablen
- Println-Anweisungen nur auskommentieren, nicht löschen
- Später: Debugger benutzen.

Fragen?

Danke und:

Viel Spaß bei den Übungen!