

Problemorientiertes Programmieren

RoboCode

1. Termin – Teil I



Christian Riess, Eva Eibenberger

Lehrstuhl für Mustererkennung (Inf. 5)

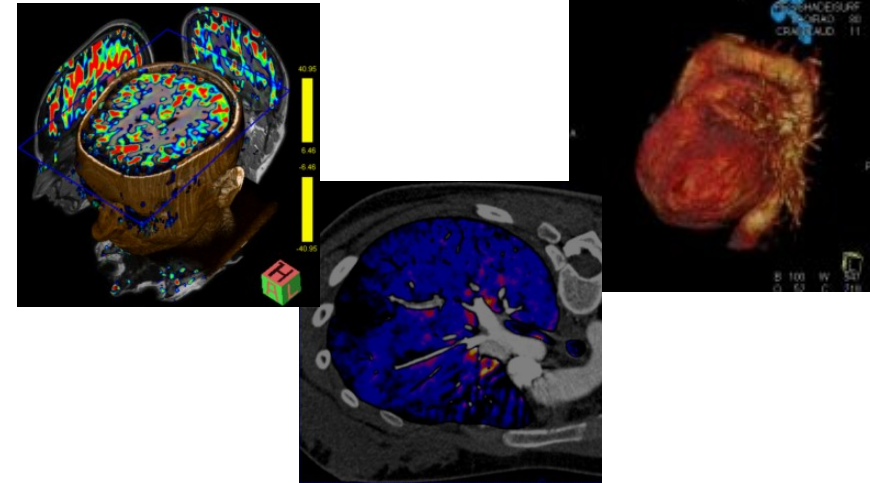
Friedrich-Alexander-Universität Erlangen-Nürnberg

Informatik in der Medizintechnik



■ Bildgebende Systeme:

- CT,
- MRT,
- Endoskopie,
- ...



[www.medical.siemens.com]

■ Medizinische Geräte:

- Cochleaimplantate,
- Herzschrittmacher,
- Insulinpumpe,
- Herz-Lungen-Maschine,
- ...



[www.biotronik.de]

[www.accu-check.de]

Warum „Problemorientiertes Denken“?



- In Medizintechnik I:
 - Welche Probleme müssen gelöst werden?
- Offene Frage:
Wie kann man Lösungen zu (diesen) Problemen finden?
- „Problemorientiertes Denken: RoboCode“
 - Übung: Probleme erkennen, Lösungen finden
 - Erlernen von Java-Grundlagen
 - Und damit: Hinführung zu „Algorithmen und Datenstrukturen“
- Wichtig: Eigeninitiative ist gefordert!



„Problemorientiertes Denken“: Ablauf



- Wichtig: Eigeninitiative ist gefordert!
- Wir Tutoren unterstützen lediglich!
- Eure Aufgabe:
 - durch Beobachten und Überlegen Probleme erkennen,
 - durch Beobachten und Überlegen Lösungen finden
 - ... und in ein Java-Programm umsetzen
- Wichtig:
 - Im eigenen Tempo arbeiten!



„Problemorientiertes Denken“: Ablauf



- 1.-3. Termin:
 - Java-Grundlagen
 - Knobelaufgaben
- 4. Termin: RoboCode-Einführung
- 5.-7. Termin:
 - Entwicklung des eigenen Roboters
 - Eigenständiges Erkennen der Schwächen und Lösungsfindung
- 8. Termin bis Schluss:
 - Verbesserung des Roboters
 - Vorschlag: Umstieg auf „Advanced Robot“, mit Nebenläufigkeit
- Regelmäßige Wettkämpfe
 - Roboter in das EST-System hochladen
 - Die Ergebnisse werden auf der Turnier-Website veröffentlicht





Bevor wir richtig anfangen:

Fragen?

Übersicht



- Grundgerüst eines Programms
 - Grundstruktur
 - Übersetzen und Aufrufen von Programmen
 - Fehlermeldungen beim Übersetzen

- Variablen und Datentypen
 - Variablen und Datentypen
 - Datentypumwandlungen
 - Arithmetische Operationen

- Kontrollstrukturen: Verzweigung und Schleifen
 - If/else-Anweisung, Switch-Anweisung und For-/While-Schleifen

- Methoden



Das Grundgerüst

■ Java-Applikation

- besteht aus einer oder mehrerer Klassen
- jede Klasse steht in einer eigenen Datei (mit der Endung `.java`)
- Eine der Klasse hat eine `main`-Methode

Definition der Klasse `BeispielApp`

Klassen- und Dateiname sind identisch!
Hier: `BeispielApp.java`

```
public class BeispielApp {  
  
    public static void main(String[] args) {  
        // hier folgen die Kommandos  
        // ...  
    }  
  
}
```

Die **Ausführung des Programms** beginnt in der ersten Zeile der `main`-Methode und endet in der letzten Zeile der `main`-Methode



Das Grundgerüst

■ Die **Ausführung des Programms**

- beginnt in der ersten Zeile der `main`-Methode
- endet in der letzten Zeile der `main`-Methode
- die dazwischenliegenden Befehle werden der Reihe nach ausgeführt

■ Die **main-Methode** enthält zum Beispiel:

- Variablen
- Ausgabe von Text und Variablen
- Berechnungen
- Aufrufe von vorhandenen oder eigenen Unterprogrammen
- For-/while-Schleifen oder if-Anweisungen zur Steuerung des Programmablaufs
- ...



Das Grundgerüst

■ Der Quelltext eines Beispiel-Programms (Square.java):

```
public class Square{
    public static void main(String[] args) {
        double length = 1.5;           // length of square
        double area = length*length;   // area computation
        System.out.println("Flaeche: " + area);
    }
}
```

Bildschirm Ausgaben (auf der Kommandozeile der Konsole):

- print: kein Zeilenumbruch
- println: mit Zeilenumbruch am Ende des Texts
- + zur Verknüpfung von Ausgabeparametern

Variablen sind mit bestimmten Datentyp versehen (später mehr dazu)

Kommentare: Wichtig für

- die Dokumentation
- kurzzeitiges Auskommentieren von Teilen des Programmcodes



Ausführung des Programms

■ Aus Computer-Sicht:

- Programm-Datei enthält zunächst nur **Text**
- Ausführung am Computer erfordert **Übersetzung** in **Maschinensprache**
- Übersetzung wird mittels **Compiler** durchgeführt

```
public class Square{  
    public static void ma  
        double length = 1  
        double area = len  
        System.out.printl  
    }  
}
```

Compiler

```
2 0000010: 3ff8 0000 000  
3 0000020: 0a00 0500 1a0  
4 0000030: 0020 0a00 050  
5 0000040: 2508 0026 080  
6 0000050: 002a 0100 063  
7 0000060: 5601 0004 436  
8 0000070: 756d 6265 725  
...
```

■ Warum dieser Umweg?

- Maschinensprache für Menschen schwer verständlich
- Programm in einer Hochsprache (z.B. Java) verwendbar auf Computern mit unterschiedlichen Maschinensprachen



Ausführung des Programms bei Java

- Übersetzung von Java-Programmen:
 - **Programm-Datei (z.B. Square.java) mit Java-Compiler (javac) übersetzen**
 - **Die übersetzte Datei hat die Endung .class (z.B. Square.class)**
- Ausführen des Programms:
 - Interpreter (`java`) mit dem Klassennamen aufrufen
- Beispiel:

```
faii00a [~/test]> javac Square.java
faii00a [~/test]> java Square
Flaeche 2.25
faii00a [~/test]>
```
- Änderungen am Programm (also die Datei `Square.java`) erfordern erneute Übersetzung durch Compiler



Übersetzen des Programms:

- Wie überall im Leben – man kann Fehler machen:
 - **Rechtschreibfehler: betreffen ein Wort/Zeichen**
 - **Grammatikfehler: betreffen die Struktur des Programms**

- **Beispiel:**

```
fau100a [~/test]> javac Square.java
Square.java:5: cannot find symbol
symbol   : variable lenth
Location: class Square
           double area = length*lenth; // area computation
                           ^
1 error
```

- Was ist zu tun?
 - Fehlermeldung lesen – es steht alles darin!
 - Behebung solcher Fehler erfordert nur Genauigkeit und Sorgfalt
 - ...daher ist es auch nicht so nett, Tutoren damit zu quälen – lesen hat man schließlich auch selbst gelernt



```
fau00a [~/test]> javac Square.java
Square.java:5: cannot find symbol
symbol   : variable length
Location: class Square
    double area = length*length; // area computation
                    ^
1 error
```

■ In Fehlermeldung enthaltene Informationen:

- Wie viele Fehler bestehen?
- In welcher Klasse befindet sich ein Fehler
- In welcher Datei und welcher Zeile?
- Um welche Fehler-Art handelt es sich?
- Wie wird Fehler-Art konkretisiert?

■ Quellcode:

```
public class Square{
    public static void main(String[] args) {
        double length = 1.5;           // length of square
        double area = length*length;   // area computation
        System.out.println("Flaeche: " + area);
    }
}
```



■ Welcher Fehler besteht?

```
faii00a [~/test]> javac Square.java
Square.java:7: ';' expected
    System.out.println("Flaeche: " + area)
                                ^
1 error
```

■ Quellcode:

```
public class Square{
    public static void main(String[] args) {
        double length = 1.5;           // length of square
        double area = length*length;    // area computation
        System.out.println("Flaeche: " + area)
    }
}
```

- Wichtig: Ein Befehl muss mit einem Strichpunkt ; beendet werden!



Variablen

■ Bedeutung

- Variablen bezeichnen einen Speicherbereich, in dem Werte eines bestimmten Typs gespeichert werden können.

■ Bestehen aus

■ Bezeichner

- Aus Buchstaben, Zahlen und bestimmten (Sonder-)Zeichen
- Konvention: beginnen mit Kleinbuchstaben

■ Datentyp

- Vordefinierte Typen („built-in“)
- Benutzerdefinierte Typen

- Wert (Speicherinhalt, dem Typ entsprechend interpretiert)

■ Beispiele:

```
int zaehler1 = 1;  
double kommaZahl = 1.5;  
char zeichen = 'a';
```




Datentypen

■ Bei Java werden acht Grundtypen unterschieden:

- Ganzzahlige Werte (integer): `byte`, `short`, `int`, `long`
- Gleitpunktwerte (Fließkommazahlen): `float`, `double`
- Zeichen (character): `char`
- Wahrheitswerte (boolesche Werte): `boolean`

■ Wertebereiche:

- | | | |
|------------------------|-----------------|--|
| ■ <code>byte</code> | 8 bit (1 Byte) | -128 ... +127 |
| ■ <code>short</code> | 16 bit (2 Byte) | -32768 ... +32767 |
| ■ <code>int</code> | 32 bit (4 Byte) | -2147483648 ... +2147483647 |
| ■ <code>long</code> | 64 bit (8 Byte) | -2^{63} ... $+(2^{63})-1$ |
| ■ <code>float</code> | 32 bit (4 Byte) | 1.40239846E-45 ... 3.40282347E+38 |
| ■ <code>double</code> | 64 bit (8 Byte) | 4.940656...E-324 ... 1.797693...E+308 |
| ■ <code>char</code> | 16 bit (2 Byte) | Unicode-Zeichen: <code>'a'</code> , <code>'B'</code> , ... |
| ■ <code>boolean</code> | 1 bit (<1 Byte) | <code>true</code> oder <code>false</code> |



Umwandeln von Datentypen

■ Beispiel:

```
int anzahlWuerste = 5;
int gewichtGesamt = 11;

// Berechnen des Gewichts aller Würste
double gewichtWurst = gewichtGesamt / anzahlWuerste;
```

Ausgabe: `gewichtWurst == 2`

- Explizite Typumwandlung ist nötig, insbesondere bei Umwandlungen auf eine Typ mit kleinerem Wertebereich
- Beispiele für explizite Typumwandlung:

```
...
// Berechnen des Gewichts aller Würste
double gewichtWurst = (double) gewichtGesamt / anzahlWuerste;
```

```
...
// Runden des Gewichts auf eine ganze Zahl
int gewichtGerundet = (int) Math.round(gewichtWurst);
```



Operatoren für Berechnungen

■ Arithmetische Operatoren:

- Addition + und Subtraktion -
- Multiplikation * und Division /
- Modulo (Rest) %

■ Kombination: Zuweisung und arithmetischer Operator

- Addition += und Subtraktion -=
- Multiplikation *= und Division /=
- Modulo %=

```
int a = 9;
int b = 4;
int c1 = a / b;           // c1 == 2
double c2 = (double) a / b; // c2 == 2.25
int d = a % b;           // d == 1
int a += b;              // a == 13
```

■ Regeln:

- Punkt-vor-Strich: $a + b * c = a + (b * c)$
- Klammerung: $a * (b + c) \neq a * b + c$



Kontrollstrukturen

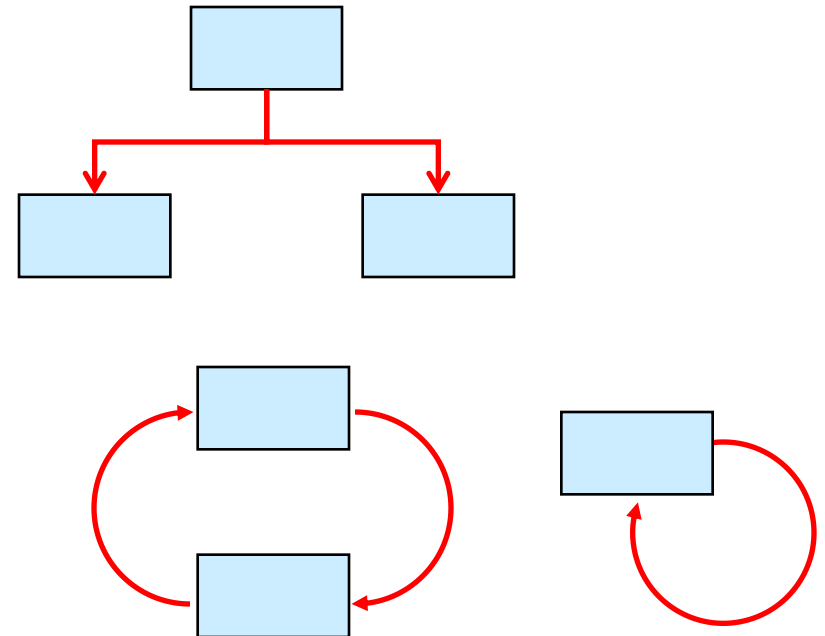
- Normalerweise:
Sequentielles Abarbeiten der Kommandos
- 2 Gruppen von Kontrollstrukturen, um den Programmablauf zu steuern:

- Verzweigungen:

- if/else-Anweisungen
- switch-Anweisungen

- Schleifen/Wiederholungen:

- For-Schleifen
- While-Schleifen





Kontrollstrukturen: Verzweigung

- Ausführung der Anweisung im if-Block nur, wenn Bedingung erfüllt ist
- Syntax:

```
if(Bedingung) {  
    Anweisungen; // Ausführung falls Bedingung erfüllt  
    ...  
}
```

```
if(Bedingung) {  
    Anweisungen; // Ausführung falls Bedingung erfüllt  
    ...  
} else {  
    Anweisungen; // Ausführung falls Bedingung NICHT erfüllt  
    ...  
}
```

- Auch weitere Verschachtelung möglich (Verzweigung in der Verzweigung)



Einschub: Bedingungen

■ Zahlen vergleichen:

```
if(a < b) { // ist a kleiner als b?  
if(a > b) { // ist a groesser als b?  
if(a == b) { // ist a gleich b?  
if(a <= b) { // kleiner oder gleich?  
if(a >= b) { // groesser oder gleich?  
if(a != b) { // ist a ungleich b?
```

■ Vergleiche verknüpfen:

```
&& heisst „und“  
|| heisst „oder“  
! Heisst „nicht“
```

■ Aus Vergleichsoperatoren und Verknüpfungen kann man komplexe Ausdrücke bauen (Klammern nicht vergessen):

```
if ((a < b) || (a < c))  
{  
    System.out.println(„a ist nicht Maximum“);  
}
```



Bedingungen: Ein „echtes“ Beispiel

- Angenommen, wir wollen beim würfeln prüfen, ob wir einen Yahtzee haben, oder zumindest Full House...
- ... und angenommen, wir hätten die Würfel w_0 bis w_4 schon nach Augenzahlen aufsteigend sortiert:

```
if ((w0 == w1) && (w3 == w4) && ((w1 == w2) || (w2 == w3))) {  
    // ein Paar und eine Zahl dreimal! Yahtzee oder Full House ist  
    // sicher - aber welches von beiden?  
    if ((w1 == w2) && (w2 == w3)) {  
        System.out.println(„yahtzee!“);  
        dance_on_the_table();  
    } else {  
        System.out.println(„Full House!“);  
    }  
} else {  
    System.out.println(„Kein Glueck heute“);  
}
```



Kontrollstrukturen: for-Schleife

- Zur Wiederholung eines Programmblocks
- Besteht aus Schleifenkopf und Schleifenkörper
- Syntax:

```
for (init; bedingung; zaehlen) {  
    Anweisungen; // wiederholtes Ausführen  
    ...  
}
```

- Schleifen diesen Typs v.a. dann wenn Anzahl der Schleifendurchläufe bekannt ist
- Beispiel:

```
for (int i = 1; i <= 10; i++) {  
    System.out.println(„i = “ + i);  
}
```

- Verschachtelung von Schleifen ist ebenfalls möglich



Kontrollstrukturen: while-Schleife

■ Syntax:

```
while (bedingung) {  
    Anweisungen; // wiederholtes Ausführen  
    ...  
}
```

■ Vor jedem Schleifendurchlauf:

- Überprüfen der Bedingung
- Nur bei `true`: Ausführen der Anweisungen

■ Beispiel:

```
int i = 1;  
while (i <= 10) {  
    System.out.println („i = " + i);  
    i++;  
}
```

- Initialisierung der Schleifenvariable **vor** der Schleife und Aktualisierung der Schleifenvariable **in** der Schleife



Unterprogramme

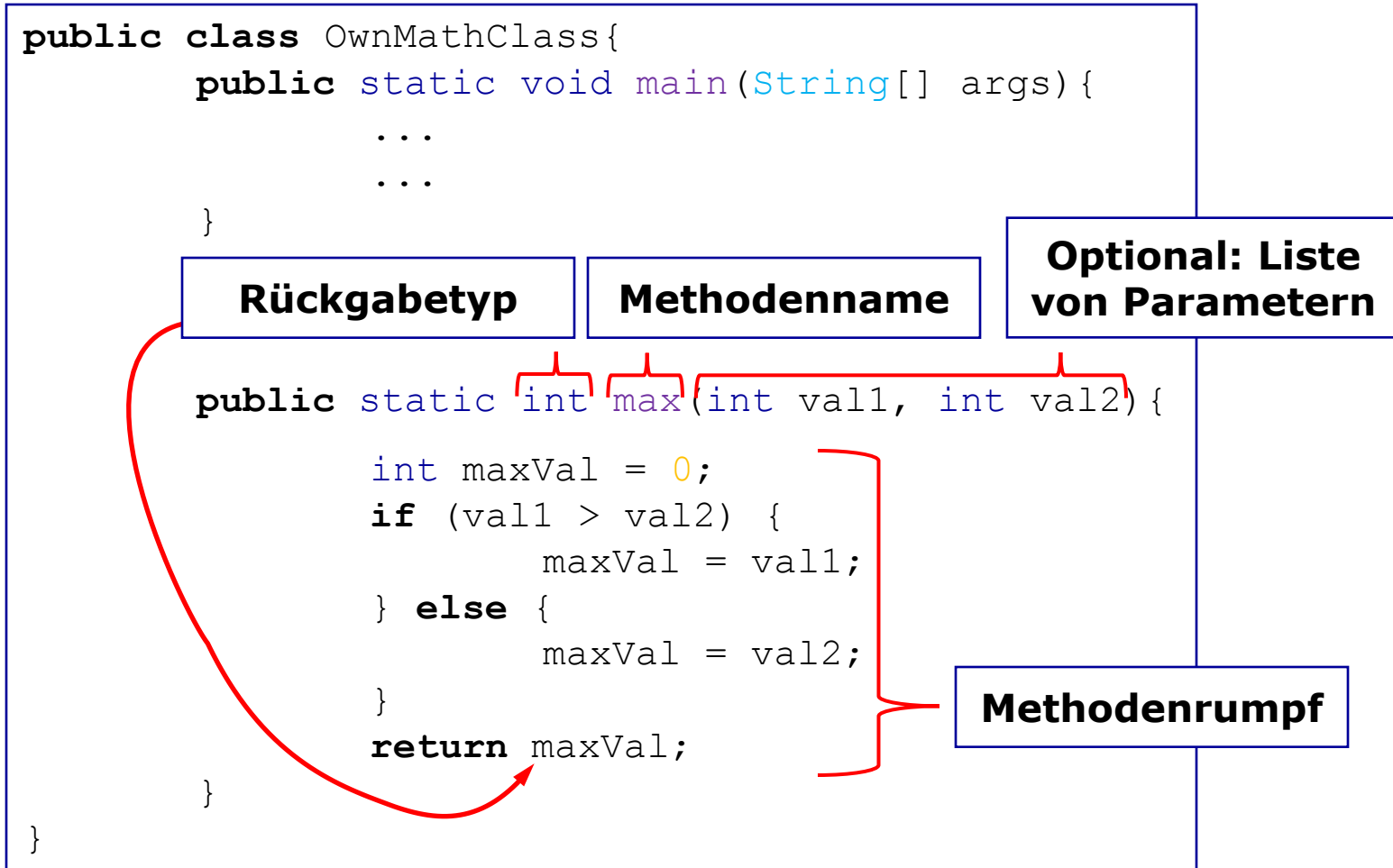
- In einer Klasse kann man auch Unterprogramme (**Methoden**) schreiben:
 - außerhalb der main-Methode
 - aber innerhalb der Klasse
- Vorteile:
 - Strukturierung des Programmes und dadurch größere Übersichtlichkeit
 - Einmal implementieren, aber mehrfach verwenden

```
public class OwnMathClass{  
    public static void main(String[] args){  
        ...  
    }  
    public static int max(int val1, int val2){  
        ...  
    }  
}
```



Unterprogramme

■ Aufbau einer Methode:





Unterprogramme

■ Aufbau einer Methode:

```
public class OwnMathClass{
    public static void main(String[] args){
        int a = 5;
        int b = 10;
        int mAB = max(a, b);
        System.out.println(„Max: " + mAB);
    }

    public static int max(int val1, int val2){
        int maxVal = 0;
        if (val1 > val2)
            maxVal =
        } else {
            maxVal =
        }
        return maxVal;
    }
}
```

Wichtig:

- Um ausgeführt zu werden, muss Methode aufgerufen werden (z.B. aus der `main`-Methode oder aus anderen Methoden)
- `main`-Methode selbst hat auch: Methodenamen, Rückgabewert, Rumpf, Übergabeparameter



Unterprogramme

■ Aufbau einer Methode:

```
public class OwnMathClass{  
    public static void main(String[] args){  
        ...  
    }  
    public static int max(int val1, int val2){  
        ...  
    }  
    public static int min(int val1, int val2){  
        ...  
    }  
    public static double mean(int val1, int val2){  
        ...  
    }  
    ...  
}
```

Wichtig:

- Viele Methoden pro Klasse möglich
- Methoden, die nichts zurück geben, haben den Rückgabety `void`



Unterprogramme

- Java bietet bereits unzählige Methoden, z.B.:
 - In der Klasse `Math`:
 - `double c = Math.cos(Math.PI / 2);`
 - `double s = Math.sin(Math.PI / 2);`
 - In der Klasse `System`:
 - `System.out.println("Cosinus von 90: " + c);`
 - `System.out.println("Sinus von 90: " + s);`
- Besonders nützlich für RoboCode sind die Math-Methoden.
- Die Dokumentation dazu: `google -> java 1.6 api math`
(„api“ steht für „application programming interface“, d.h. Doku.
„1.6“ ist die Java-Version mit der wir arbeiten)



Bevor wir mit dem praktischen Teil anfangen:

Fragen?



Praktischer Teil:

Eigenständiges Bearbeiten des 1. Übungsblatts
zu finden unter

<http://www5.informatik.uni-erlangen.de/en/lectures/ss-12/problemorientiertes-programmieren-robocode-robocode/>

(oder: www5.cs.fau.de -> Courses -> RoboCode)