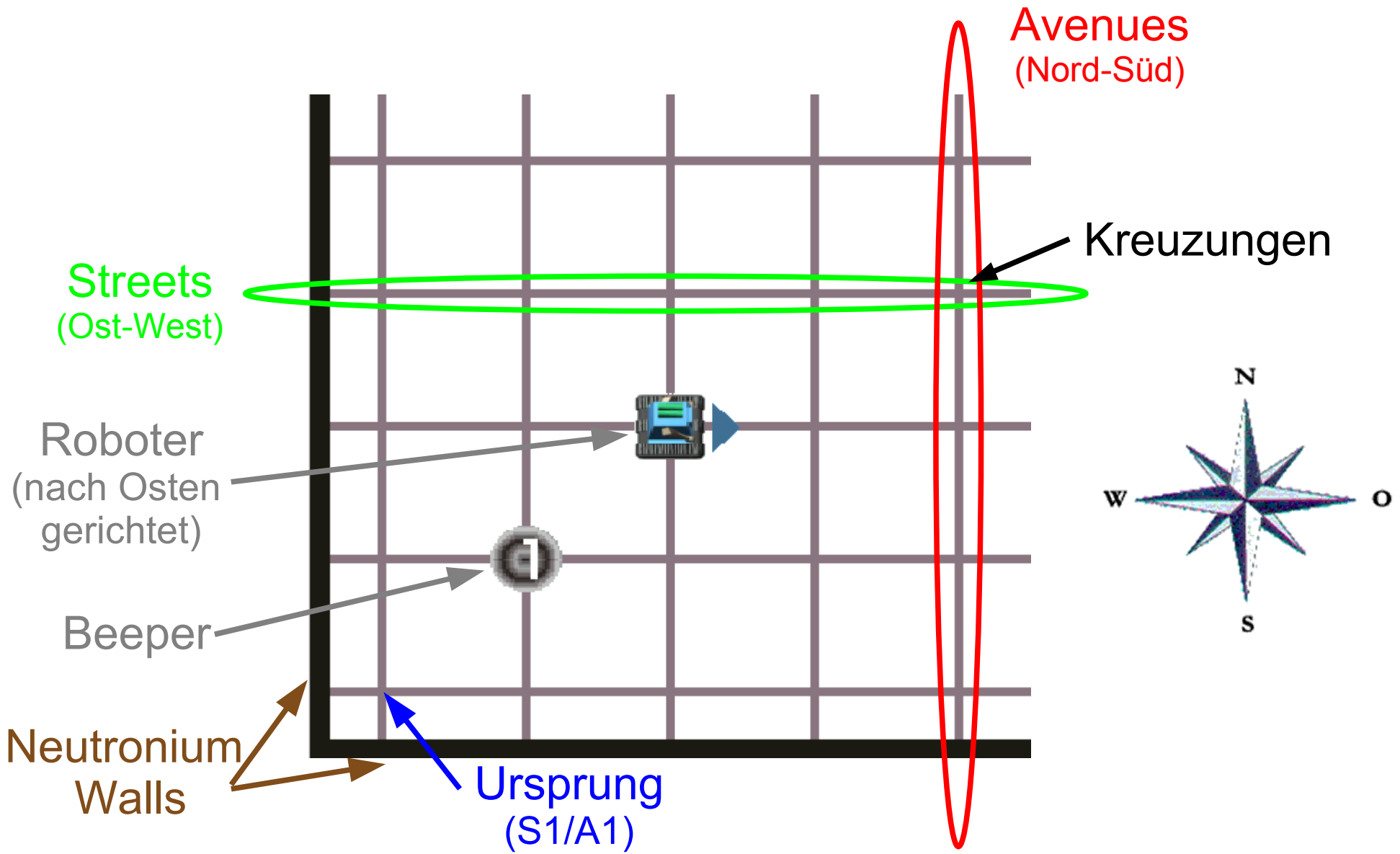
Programmieren mit Karel J. Robot

- KarelJ ist eine Erweiterung der Programmiersprache Java
  - orientiert sich an der Aufgabe, Roboter in einer (sehr) einfachen, simulierten Welt zu steuern
- KarelJ wurde speziell für den Einsatz in der Lehre entwickelt
  - Vereinfachung von komplexen syntaktischen Konstrukten
  - Reduktion auf das Wesentliche (Verständnis objekt-orientierter Programmierung)
  - unmittelbares Feedback durch Beobachtung des Resultats in der Roboter-Welt
  - einfache IDE zur Programmierung (Christoph Bokisch, TU Darmstadt)

Die Roboter-Welt



Roboter

- Roboter können durch einfache Befehle programmiert werden, bestimmte Aufgaben zu erledigen
  - Beeper zu finden oder zu verteilen
  - Muster mit Beepern zu zeichnen
  - Navigation in verwinkelten Welten mit vielen Wänden
  - u.v.m.
- Alle Roboter entstammen (vorerst) derselben Serie der Karel-Werke
  - später lernen wir, neue Modelle zu spezifizieren
- Fähigkeiten dieser Serie
  - Einen Schritt vorwärts gehen
  - Eine Viertel-Drehung nach links machen
  - Feststellen, ob sich am momentanen Standort andere Roboter oder Beeper befinden
  - Feststellen, ob der Weg frei ist

Roboter-Programmierung

- Eine einfache Programmier-Umgebung erlaubt, Kommandos an die Roboter zu schicken.
- Die Kommandos müssen einer einfachen Syntax folgen

- 2 Grund-Typen von Kommandos
  - Aufruf von Konstruktoren:
    - zum Erzeugen neuer Roboter in der Welt
    - neue Roboter müssen immer Namen erhalten!

```
Robot karel = new Robot(1, 2, 0, North);
```

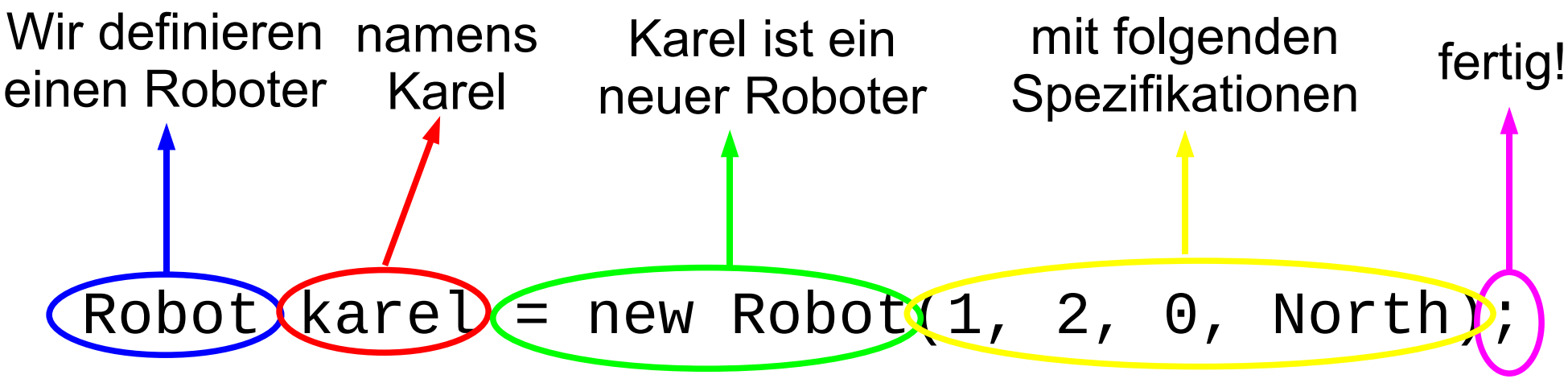
  - Senden von Nachrichten:
    - zum Steuern vorhandener Roboter
    - der Name gibt an, welcher Roboter gesteuert werden soll

```
karel.move();
```

Aufruf von Konstruktoren

```
Robot karel = new Robot(1, 2, 0, North);
```

Aufruf von Konstruktoren

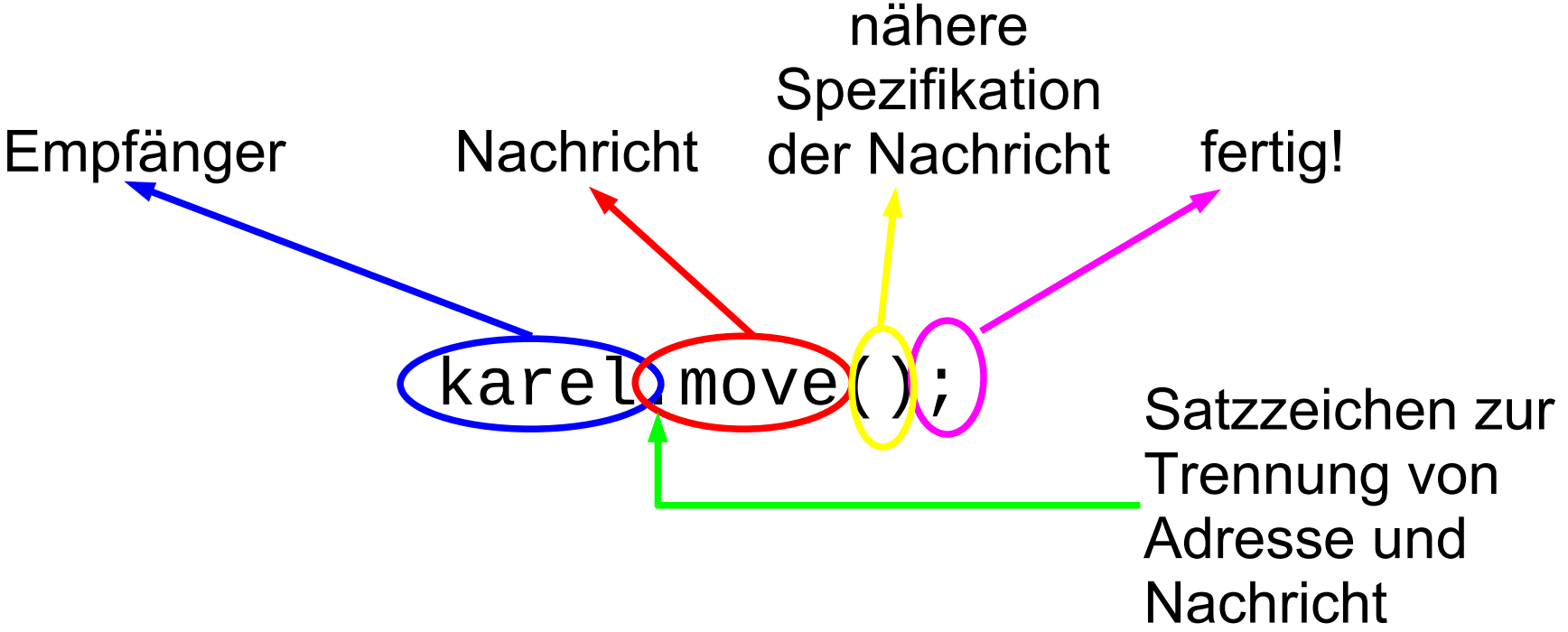


Straße: 1
Avenue: 2
Beeper: 0
Richtung: Norden

Nachrichten / Methoden

```
karel.move();
```

Nachrichten / Methoden



Anmerkung:
Die Spezifikation ist (vorerst) meist leer.

Denkbar wäre z.B. eine Methode `move_n(n)`, die als Parameter die Anzahl der Schritte erwartet, die der Roboter gehen soll.

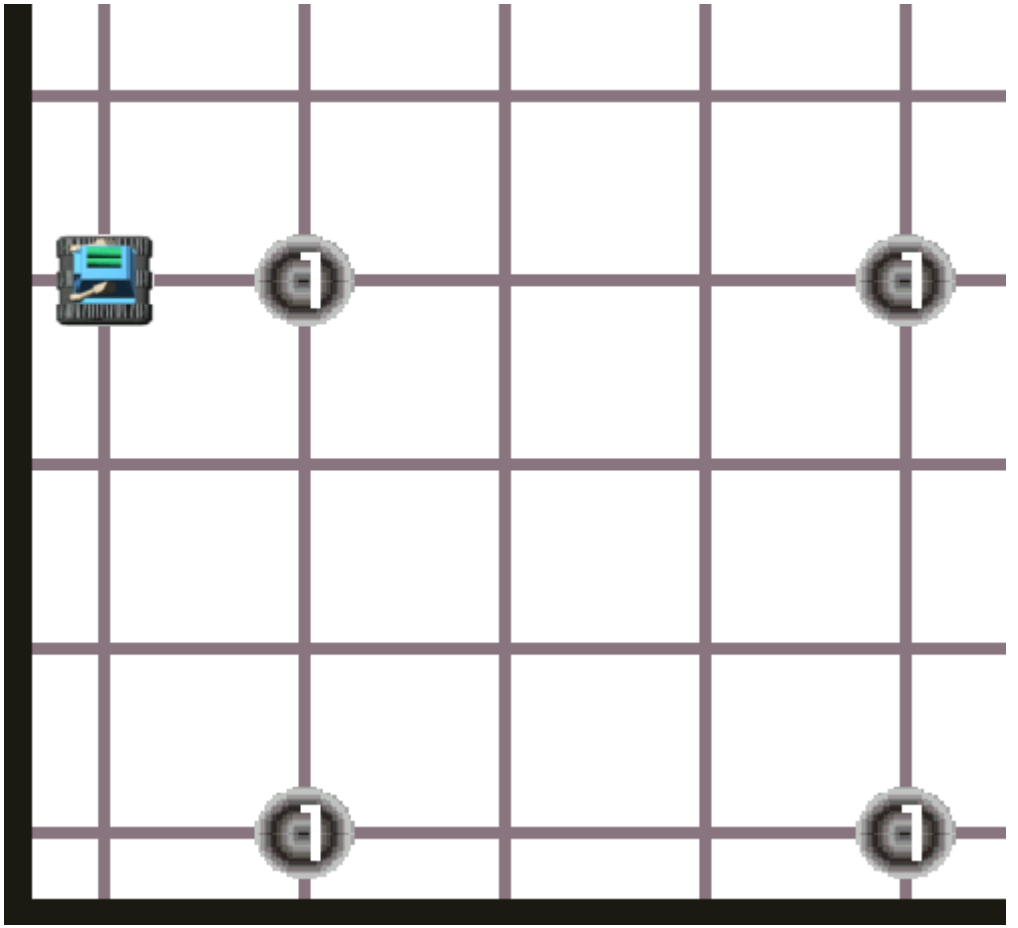
Vordefinierte Methoden

Nachrichten, die eine Aktion durchführen:

- `move()`
  - Bewegung um ein Feld in die Richtung der momentanen Orientierung
  - läuft er gegen eine Wand, gibt's einen Error (und der Roboter schaltet sich ab)
- `turnLeft()`
  - 90º-Drehung nach links (nein, rechts kann er nicht...)
- `turnOff()`
  - Roboter ausschalten
- `pickBeeper(), putBeeper()`
  - einen Beeper aufheben bzw. auf der momentanen Position hinterlassen
  - falls da kein Beeper liegt (bei pick) bzw. der Roboter keinen Beeper mit sich führt (bei put), gibt's einen Error

Aufgabe: Zeichne ein Quadrat

- Der Roboter soll im Ursprung starten
- mit einer Ladung von 4 Beepern
- und soll diese als Ecken eines Quadrats mit Seitenlänge 3 anordnen



Quadrat-Programm

Programme nennen wir tasks

```
task {
    Robot karel = new Robot(1, 1, 4, East);
    karel.move();
    karel.putBeeper();
    karel.move();
    karel.move();
    karel.move();
    karel.putBeeper();
    karel.turnLeft();
    karel.move();
    karel.move();
    karel.move();
    karel.putBeeper();
    karel.turnLeft();
    karel.move();
    karel.move();
    karel.move();
    karel.putBeeper();
    karel.move();
}
```

SW Ecke

Südkante

SO Ecke

Drehung

Ostkante

NO Ecke

Drehung

Nordkante

NW Ecke

Karel entsteht im Ursprung, hat 4 Beeper und blickt nach Osten

Formatierung von Programmen

- Prinzipiell gilt (für die meisten Programmiersprachen)
  - Die Anweisungen müssen durch Whitespace getrennt sein
    - Leerzeichen
    - Tabulatoren
    - Carriage Return
  - daraus kann man ziemlich unleserliche Programme schreiben

- Die Konvention ist, Programme leserlich zu schreiben
  - jeder Befehl in eine eigene Zeile
  - Leerzeilen erhöhen Lesbarkeit
  - jede neue Ebene (umgeben von { }) wird etwas weiter eingerückt als die vorige
  - sprechende Namen verwenden

Eine Implementierung von Missile Command....

By John Williams, Lindon UT, USA
Source: Obfuscated C Contest 2001


```
#include <X11/Xlib.h>
#include <unistd.h>
typedef long o; typedef struct                    { o b,f,u,s,c,a,t,e,d; } C;
Display *d; Window w; GC g; XEvent e;
char Q[] = "Level %d   Score %d", m[222];
#define N(r) (random()%(r))
#define U I[n++]=L[n]=l; n%=222
#define K c=-l.u; l=I[i]; l.t=0; c+=l.u
#define E l.e--&&!--L[l.e].d&&(L[l.e].t=3)
#define M(a,e,i,o) a[0]=e,(a[1]=i)&&XFillPolygon(d,w,g,(void*)a,o,1,1)
#define F return
#define R while(
#define Y if(l.t


                    o p                                                 ,B,
                  D,A=6,Z                                             ,S=0,v=
                 0,n=0,w=400                                        ,H=300,a[7]
                ={ 33,99, 165,                                     231,297,363} ;
               XGCvalues G={ 6,0                                  ,~0L,0,1} ; short
             T[]={ 0,300,-20,0,4                                ,-20,4,10,4,-5,4,5,
           4,-20,4,20,4,-5,4,5,4,                              -10,4,20},b[]={ 0,0,4,
         0,-4,4,-4,-4,4,-4,4,4} ;                             C L[222],I[222];dC(o x){
       M(T,a[x],H,12); } Ne(C l,o                           s) { l.f=l.a=1; l.b=l.u=s;
      l.t=16; l.e=0; U; } nL(o t,o                        a,o b,o x,o y,o s,o p){ C l;
     l.d=0; l.f=s; l.t=t; y-=l.c=b;                      l.e=t==2?x:p; x-=l.s=a;s=(x|1)
    %2*x; t=(y|1)%2*y; l.u=(a=s>t?s:                    t)>>9;l.a=(x<<9)/a;l.b=(y<<9)/a;
   U; } di(C I){ o p,q,r,s,i=222;C l;                 B=D=0; R i--){ l=L[i]; Y>7){ p=I.s
  -l.s>>9; q=I.c-l.c>>9; r=l.t==8?l.b:               l.a; s=p*p+q*q; if(s<r*r||I.t==2&&s<
  26) F S+=10; s=(20<<9)/(s|1); B+=p*s;            D+=q*s; }} F 0; } hi(o x,o d){ o i=A;
 R i--&&(x<a[i]-d||x>a[i]+d)); F i; }      dL(){ o      c,r=0, i=222,h; C l; R i--){ l=L[i];
 Y){ r++;c=l.f; Y==3){c=l.u; l.t=0;      E; }R c--){--    l.u;h=l.c>>9; Y>7){XDrawArc(d,w,g,
(l.s>>9)-++l.a,h-l.a,l.a*2,l.a*2,0    ,90<<8); if(!l.u){    I[i].t-=8; l=I[i]; } } else Y==2)M
(b,l.s>>9,h,6); else XDrawPoint(d     ,w,g,(l.s+=l.a)>>9,     h=(l.c+=l.b)>>9); Y==4&&!l.u){ Ne
(l,20); K; } Y&&l.t<3&&(di(l)||h>     H)){ if(h>H&&(c=hi(     l.s>>9,25))>=0){ dC(c); a[c]=a[--
A]; }Ne(l,30); Y==1){ E;K; } else     c=l.t=0;} Y==1&&h<H     -75&&!N(p*77)){ do{ nL(1,l.s,l.c,
                                       N(W<<9),H<<9,1,i+
                                         1); I[i].d++;
                                           }R N(3)

                                            );          K;
                                           l.u=c; c=0; } Y
                                          ==2){ l.s+=l.a+B;
                                         l.a= (l.e-l.s)/((H+
                                       20-h)|1); l.c+=l.b+D;
                                      M(b,l.s>>9,l.c>>9,6); }
                                    } L[i]=l; } } F r; } J(){
                                  R A) { XFlush(d); v&&sleep(
                                3); Z=++v*10; p=50-v; v%2&&hi
                               ((a[A]=N(W-50)+25),50)<0 &&A++;
                             XClearWindow (d,w); for(B=0; B<A;
                            dC(B++)); R Z|dL()){ Z&&!N(p)&&(Z--
                           ,nL(1+!N(p),N(W<<9), 0,N(W<<9),H<<9,1
                          ,0)); usleep(p*200); XCheckMaskEvent(d,
                        4,&e)&&A&&--S&&nL(4,a[N(A)]<<9,H-10<<9,e.
                       xbutton.x<<9,e.xbutton.y<<9,5,0);}S+=A*100;
                           B=sprintf(m,Q,v,S); XDrawString(d,w
                                  ,g,W/3,H/2,m,B); } }

main ()
{
o i=2;
d=XOpenDisplay(0);
w=RootWindow(d,0);
R i--) XMapWindow(d,w=XCreateSimpleWindow(d,w,0,0,W,H,0,0,0));
XSelectInput(d,w,4|1<<15);
XMaskEvent(d,1<<15,&e);
g=XCreateGC(d,w,829,&G);
srandom(time(0));
J();
puts(m);
}
```

Objekte in der Roboter-Welt

- Roboter
  - die Welt kann mit beliebig vielen Robotern bevölkert werden. Jeder Roboter erhält einen Namen.
  - beliebig viele Roboter können an derselben Kreuzung stehen, sie behindern sich nicht in der Bewegung
- Beeper
  - beliebig viele Beeper können an jeder Kreuzung stehen (die Anzahl wird auf dem Beeper-Symbol angezeigt)
  - Beeper behindern die Roboter ebenfalls nicht
  - Beeper können von Robotern aufgehoben und mitgenommen werden
- Wände
  - die Welt ist im Westen und Süden von undurchdringlichen Wänden begrenzt
  - Welten mit mehr Wänden können konstruiert werden

Namensvergabe

- Roboter (und später andere Objekte oder Klassen) können beliebige Namen erhalten
  - **Richtlinie:** Die Namen sollen “sprechend” sein, d.h. sie sollen etwas über ihre Funktion aussagen

- Ausnahme:
  - reservierte Wörter:
    - `task, new, int, if, while, for, ...`
  - diese sind Teil der Programmiersprache und dürfen für keinen anderen Zweck verwendet werden
    - Methoden und Klassennamen sind keine reservierten Wörter
    - sollten aber auch nur eindeutig verwendet werden

Kommentare

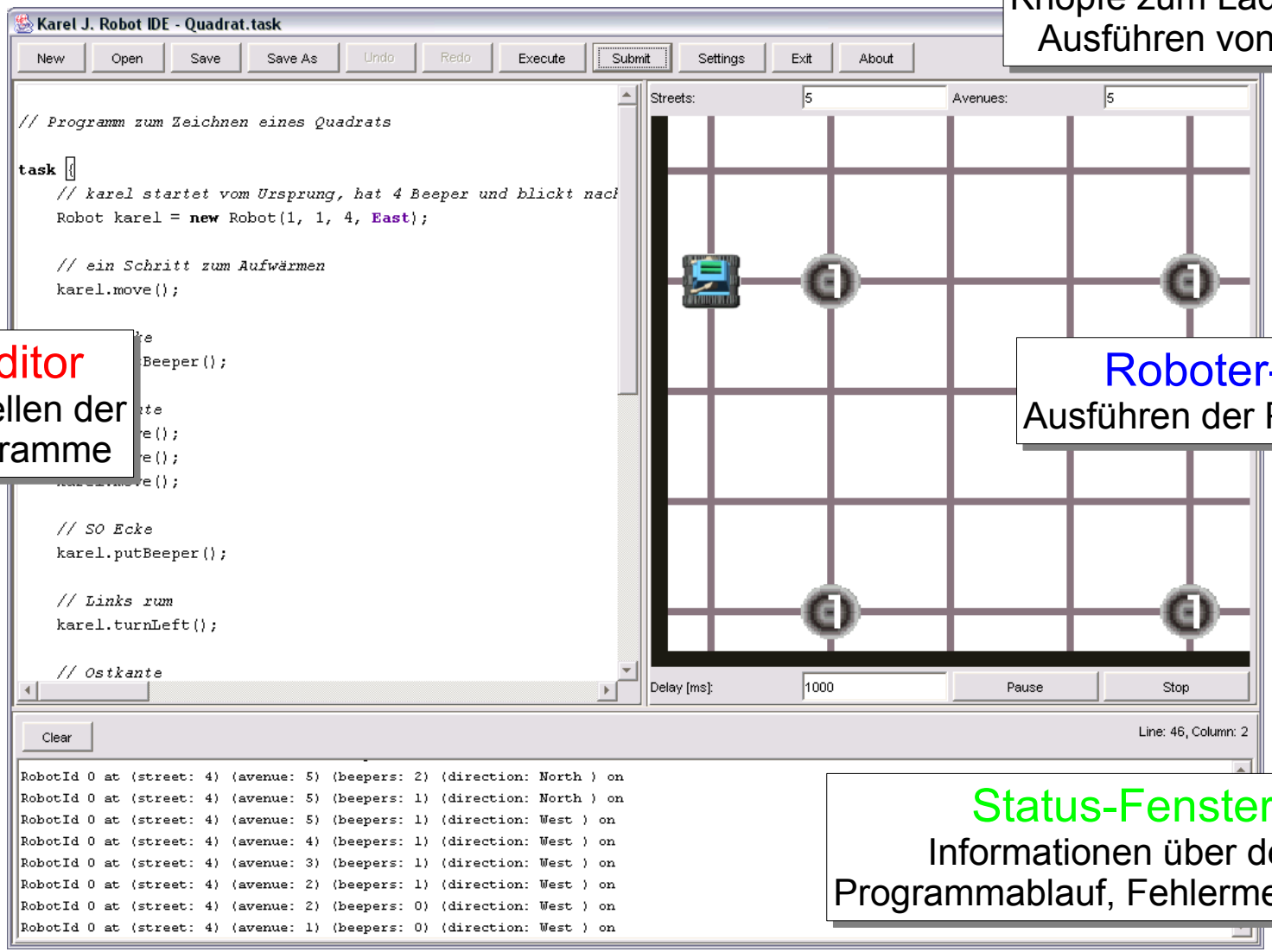
- können beliebigen Text enthalten
  - z.B. Autor des Programms, Datum, Versionsnummer
- dienen zur Erklärung von Programmteilen
  - für andere Programmierer
  - für einen selbst
    - damit man seine Programme auch noch nach 3 Monaten versteht
  - nicht für den Computer
    - der betrachtet den gesamten Kommentar als Whitespace
- In KarelJ-Welt:
  - Nach einem `//` wird der Rest der Zeile als Kommentar betractet
  - Beispiel:

```
karel.move();   // beweg Dich, Karel!
```

Klareres Quadrat-Programm

```
// Programm zum Zeichnen eines Quadrats

task {
    // karel startet vom Ursprung,
    // hat 4 Beeper und blickt nach Osten
    Robot karel = new Robot(1, 1, 4, East);

    // ein Schritt zum Aufwärmen
    karel.move();

    // SW Ecke
    karel.putBeeper();

    // Südkante
    karel.move();
    karel.move();
    karel.move();

    // SO Ecke
    karel.putBeeper();

    // Links rum
    karel.turnLeft();

    // Ostkante
    karel.move();
    karel.move();
    karel.move();

    // NO Ecke
    karel.putBeeper();

    // nochmals Links
    karel.turnLeft();

    // Nordkante
    karel.move();
    karel.move();
    karel.move();

    // NW Ecke
    karel.putBeeper();

    // und langsam auslaufen...
    karel.move();
}
```

IDE: Integrated Development Enviroment



Steuer-Leiste
Knöpfe zum Laden, Speichern, Ausführen von Programmen

Editor
Erstellen der Programme

Roboter-Welt
Ausführen der Programme

Status-Fenster
Informationen über den Programmablauf, Fehlermeldungen

Ablauf der Programmierung

1. (optional) Laden des Programms
2. Erstellen bzw Ändern eines Programms
3. Abspeichern des Programms
   - der Datei (File) unter der das Programm gespeichert wird, wird ein Namen zugeordnet, unter dem es später wieder geladen werden kann
4. Übersetzen des Programms
   - das Programm muß zuerst in einen langen Code von 0/1 übersetzt werden, den der Computer versteht
   - viele Programmierfehler (z.B. falsche Syntax, Tipp-fehler, etc.) werden bereits beim Übersetzen gefunden
   - Bei Fehlern, Fehler suchen, dann bei Schritt 2. weitermachen
5. Testen des Programms
   - Falls das Programm nicht das tut, was man will, Fehler suchen, dann bei Schritt 2. weitermachen

Steuer-Leiste

- New
  - wenn Sie ein neues Programm erstellen wollen
- Open
  - öffnen eines vorhandenen Programms
- Save
  - speichert das Programm und versucht, es zu übersetzen
- Save As
  - speichert das Programm unter einem neuen Namen und versucht, es zu übersetzen
- Undo
  - macht die letze Änderung rückgängig
- Redo
  - macht das letzte Undo rückgängig
- Execute
  - führt das Programm aus (speichert vorher, falls Sie vergessen)

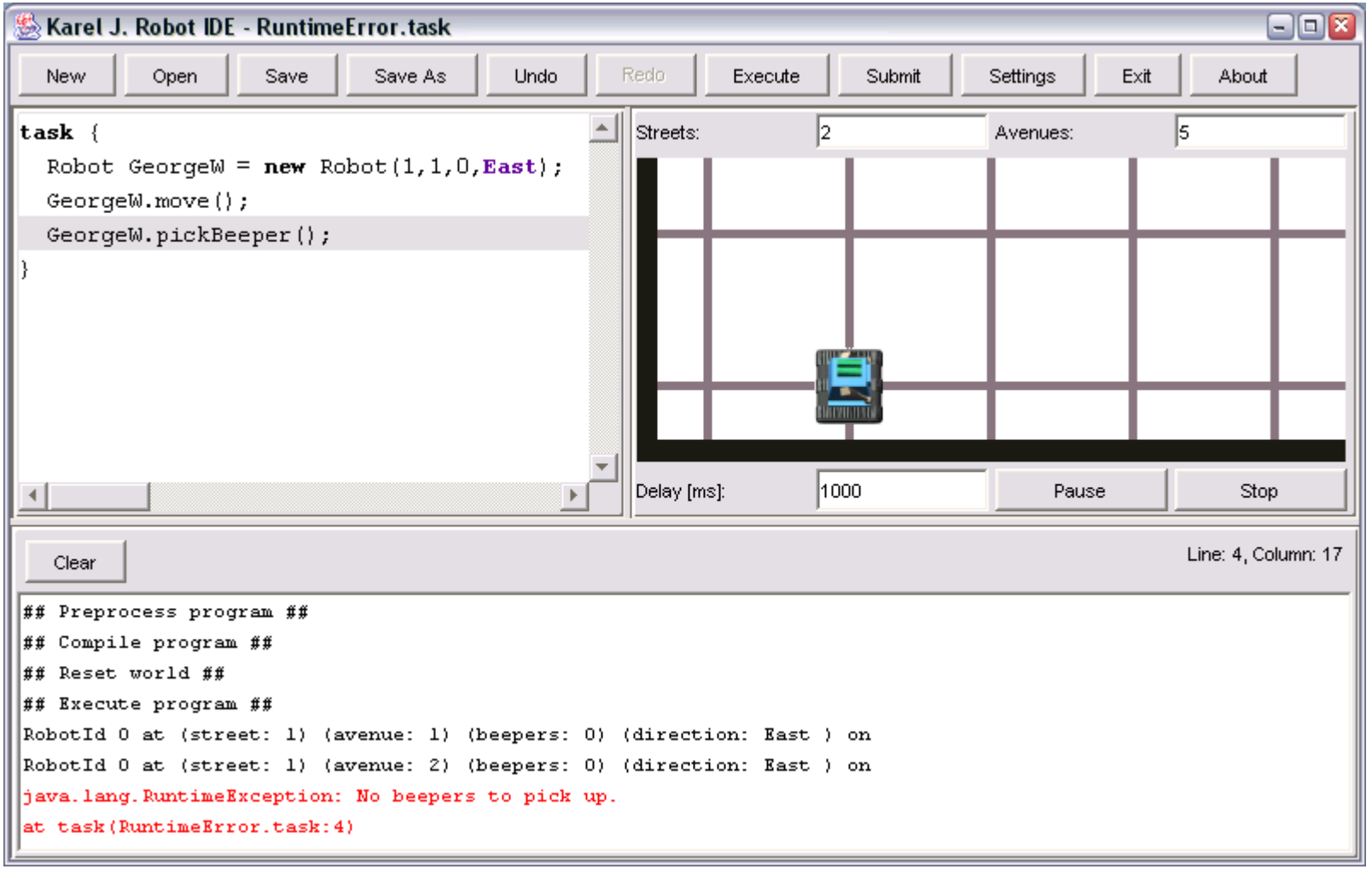
Weitere Steuer-Elemente

- Steuerung der Welt
  - Pause / Resume bzw. Stop
    - hält die Ausführung des Programms an bzw. startet sie wieder
  - Delay:
    - Einstellen der Geschwindigkeit, mit der die einzelnen Schritte ausgeführt werden
  - Avenues / Streets:
    - Größe der *Darstellung* der Welt
      (Größe ist aber im Prinzip unbegrenzt)
- Für die bewertete Übung brauchen Sie eventuell noch
  - Submit
    - schickt das Programm an Ihren Tutor.
  - Settings
    - hier können Sie die einstellen, wer ihr Tutor ist, den Code Ihrer Übungsgruppe eingeben etc.

Programmfehler

- Programmierer machen immer Fehler
  - Schreibfehler, Denkfehler, Design-Fehler, Logische Fehler, ...
- Bugs:
  - Fehler im Programm
- Debugging:
  - Auffinden der Fehler im Programm
- Offensichtliche Fehler werden automatisch erkannt
  - vom Übersetzer (z.B. Schreibfehler)
  - manche auch während der Laufzeit (z.B. Aufheben eines Beepers, der nicht da ist)

Typische Fehlermeldung



Lexical Errors

- Auftreten von Wörtern, die sich nicht im Vokabular des Roboters befinden
- werden bei der Übersetzung erkannt und im Status-Fenster angezeigt

```
taxt
{
  Robot Karel(1,2, East, 0) ;
  karel.move();
  Karel.mvoe();
  Karel.pick();
  Karel.move();
  Karel.turnRight();
  Karel.turn-left();
  Karel.turnleft();
  Karel.move();
}
```

Lexical Errors

- Auftreten von Wörtern, die sich nicht im Vokabular des Roboters befinden
- werden bei der Übersetzung erkannt und im Status-Fenster angezeigt

```
taxt                        // Tippfehler
{
  Robot Karel(1,2, East, 0) ; // fehlendes new...
  karel.move();             // Klein-Großschreibung
  Karel.mvoe();             // Buchstaben vertauscht
  Karel.pick();             // unbekannte Nachricht
  Karel.move();
  Karel.turnRight();        // unbekannte Nachricht
  Karel.turn-left();        // unbekannte Nachricht
  Karel.turnleft();         // Klein-und Großschreibung!
  Karel.move();
}
```

Syntax Errors

- Nicht-Einhalten der festgelegten Regeln der Programmiersprache
- vergleichbar mit ungrammatikalischen Sätzen
- werden auch bei der Übersetzung erkannt und im Status-Fenster angezeigt

```
Robot Karel = new Robot(1,1,0,East);
task
  Robot Karel2 = new Robot(East,2,2,0);

  move();
  Karel.pickBeeper;
  Karel move();
  Karel.turnLeft()
};
  Karel.turnOff()
```

Syntax Errors

- Nicht-Einhalten der festgelegten Regeln der Programmiersprache
- vergleichbar mit ungrammatikalischen Sätzen
- werden auch bei der Übersetzung erkannt und im Status-Fenster angezeigt

```
Robot Karel = new Robot(1,1,0,East); // nicht im Task-Block
task                                 // Klammer auf { fehlt
  Robot Karel2 = new Robot(East,2,2,0);
                                     // falsche Parameterreihenfolge
  move();                            // Wer soll sich bewegen?
  Karel.pickBeeper;                  // keine ()
  Karel move();                      // Punkt vergessen
  Karel.turnLeft()                   // Strichpunkt vergessen
};                                   // Strichpunkt zu viel
  Karel.turnOff()                    // nach Ende des Task-Blocks
```

Execution Errors

- Fehler in der Logik im Programmablauf
- Das Programm übersetzt richtig, es tut etwas, aber nicht das was es soll

Aufgabe: Karel soll den Beeper ein Feld nach Norden verschieben

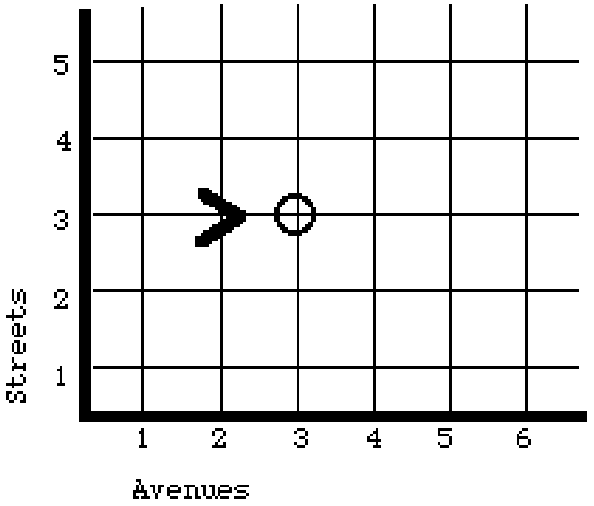


Figure 2-4 Karel's Initial Situation

```
task
{ Robot Karel = new Robot(3,2,0,East);
  Karel.move();
  Karel.pickBeeper();
  Karel.move();
  Karel.turnLeft();
  Karel.putBeeper();
  Karel.move();
  Karel.turnOff();
}
```

Resultat: Karel verschiebt den Beeper ein Feld nach Osten

Wo ist der Fehler?

Debugging

- Lexical und Syntax Errors sind meistens leicht zu finden
  - im Status-Fenster beim Übersetzen findet sich meist eine gute Beschreibung des aufgetretenen Fehlers
  - aber nicht aller Fehler, da die Übersetzung abbricht, wenn Sie dem Programm nicht mehr folgen kann

- Execution Errors sind oft schwer zu finden
  - Anzeige von Fehlermeldungen nur bei offensichtlicher Verletzung von Voraussetzungen
    - gegen die Wand laufen
    - Aufheben eines Beepers, wo keiner ist
    - Niederlegen eines Beepers, wenn man keinen hat
  - logische Fehler im Programmablauf zu finden ist oft ein langwieriger Prozeß des Ausprobierens
    - Trial and Error

Debugging Tricks

- das Programm Schritt für Schritt nachvollziehen
  - nach Möglichkeit nicht von der eigenen Erwartungshaltung beeinflussen lassen
- das Status-Fenster gibt Informationen über den Programm-Ablauf
  - zeigt nach jedem Befehl an, wo sich der Roboter befindet, in welche Richtung er schaut, wie viele Beeper er bei sich hat
- Nachvollziehen in der Roboter-Welt
  - eventuell das Programm durch Einfügen von `turnOff` Befehlen verfrüht stoppen, um an kritischen Punkten zu sehen, ob es noch im richtigen Zustand ist

Verwendung mehrerer Roboter

- Klasse:
  - eine abstrakte Definition einer Familie von (gleichartigen) Objekten (z.B. `Robot`)
- Instanz:
  - eine konkretes Objekt dieser Famile (z.B. `karel`)

- Klarerweise kann ein Programm mehrere Instanzen derselben Klasse enthalten
  - d.h. es können mehrere Roboter gleichzeitig in der Welt herumlaufen

Beispiel

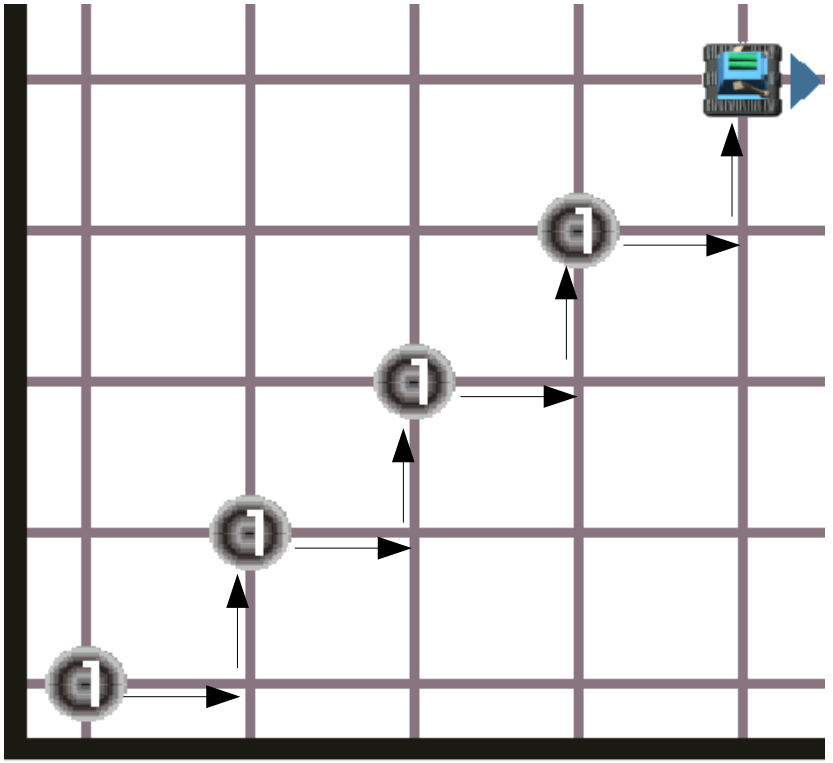
```
task {
  // bolek hält einen Beeper
  Robot bolek = new Robot(2,1,1,East);
  // lolek nicht
  Robot lolek = new Robot(2,3,0,West);
  // bolek läßt den Beeper fallen
  bolek.move();
  bolek.putBeeper();
  bolek.move();
  // und lolek hebt ihn wieder auf
  lolek.move();
  lolek.pickBeeper();
  lolek.move();
  // nun wird getanzt
  bolek.turnLeft();
  lolek.turnLeft();
  bolek.turnLeft();
  lolek.turnLeft();
}
```

Eine neue Aufgabe

Aufgabe:

- zeichne die Diagonale

- Lösung
  - Roboter muß Treppen steigen



Treppen steigen: 1. Lösung

```
task
{
  // neuer TreppenSteiger
  Robot astair =
        new Robot(1,1,4,East);
  // ersten Beeper abladen
  astair.putBeeper();
  // eins links
  astair.move();
  // links um
  astair.turnLeft();
  // eins hinauf
  astair.move();
  // rechts um
  astair.turnLeft();
  astair.turnLeft();
  astair.turnLeft();
  // und nochmals das ganze
  astair.putBeeper();
  astair.move();
  astair.turnLeft();
  astair.move();
  astair.turnLeft();
  astair.turnLeft();
  astair.turnLeft();
  // Nr. 3
  astair.putBeeper();
  astair.move();
  astair.turnLeft();
  astair.move();
  astair.turnLeft();
  astair.turnLeft();
  astair.turnLeft();
  // einmal geht's noch
  astair.putBeeper();
  astair.move();
  astair.turnLeft();
  astair.move();
  astair.turnLeft();
  astair.turnLeft();
  astair.turnLeft();
}
```

Schleife

- Ist das nicht ein wenig umständlich?
  - Ich muß schließlich für jeden Teilschritt (eine Stufe) denselben Block von Anweisungen ausführen
  - Kann mir nicht der Computer die stupide (Schreib-)Arbeit abnehmen?
- Ja, mit einer Schleife:
  - `loop(n) {  }`
    - bedeutet, daß die Anweisungen zwischen den geschwungenen Klammer n-mal ausgeführt werden
- Beispiel:

```
// rechts um
astair.turnLeft();
astair.turnLeft();
astair.turnLeft();
```

```
// rechts um
loop(3) {
   Astair.turnLeft();
}
```

Treppen steigen: 2. Lösung

```
task
{
  Robot astair = new Robot(1,1,4,East);
  loop(4) {
     astair.putBeeper();
     astair.move();
     astair.turnLeft();
     astair.move();
     loop(3) {
        astair.turnLeft();
     }
  }
}
```

4x Wiederholen

Eine Stufe steigen

bis hierher

Eine Schleife kann auch innerhalb einer anderen Schleife definiert werden (verschachtelte Schleifen)

Neue Roboter definieren

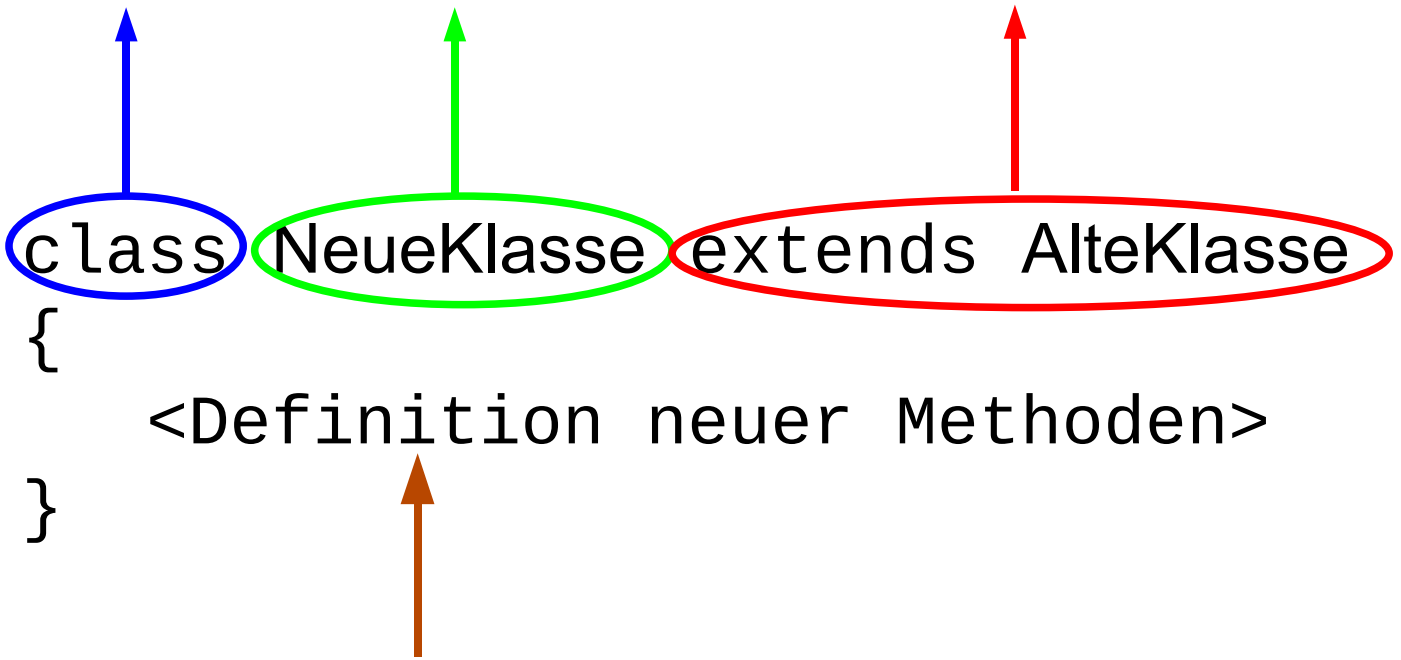
- Die Definition einer Rechtsdrehung mit `loop` ist relativ kurz und prägnant
- Besser wäre allerdings ein Roboter, der ein Kommando `turnRight()` versteht.
- Noch besser wäre eine ganze Klasse von Robotern, die die Rechtsdrehung beherrscht, sodaß man beliebig viele “rechtsdrehende” Roboter produzieren kann.
  - das heißt, wir müssen eine neue Spezifikation an die Karel-Werke schicken
  - die alles kann, was ein Robot so können muß (alle Fähigkeiten des existierenden Robot-Modells übernimmt)
  - und zusätzlich Rechtsdrehungen beherrscht

Syntax für neue Klassen

```
class NeueKlasse extends AlteKlasse
{
   <Definition neuer Methoden>
}
```

Syntax für neue Klassen

Wir definieren
eine neue Klasse namens "NeueKlasse" die eine Erweiterung von "AlteKlasse" ist



```
class NeueKlasse extends AlteKlasse
{
    <Definition neuer Methoden>
}
```

Nur *neue* Methoden müssen definiert werden!
(der Rest wird ***geerbt***, i.e., von der Spezifikation von AlteKlasse übernommen)

Vererbung (Inheritance)

- Definiert man eine neue Klasse als Unterklasse einer existierenden Klasse, erbt diese Klasse alle Methoden der Überklasse
  - unter Verwendung der `extends` Option

- Das heißt, alle Methoden der Überklasse können in der Unterklasse genauso verwendet werden
  - Beispiel:

```
class NewRobot extends Robot { }
```

    - Alle Objekte der Klasse `NewRobot` können genauso definiert und behandelt werden wie Objekte der Klasse `Robot`
    - Insbesondere können alle bisherigen Programme `NewRobots` statt `Robots` verwenden
      (wenn sie obige Definition enthalten oder importieren)

Neue Methoden schreiben

- Zusätzlich zu vererbten Methoden kann eine neue Klasse auch neue Methoden enthalten
  - ACHTUNG: Methoden-Definitionen gehören immer zu einer Klasse und müssen daher innerhalb einer Klassen-Definition definiert werden!

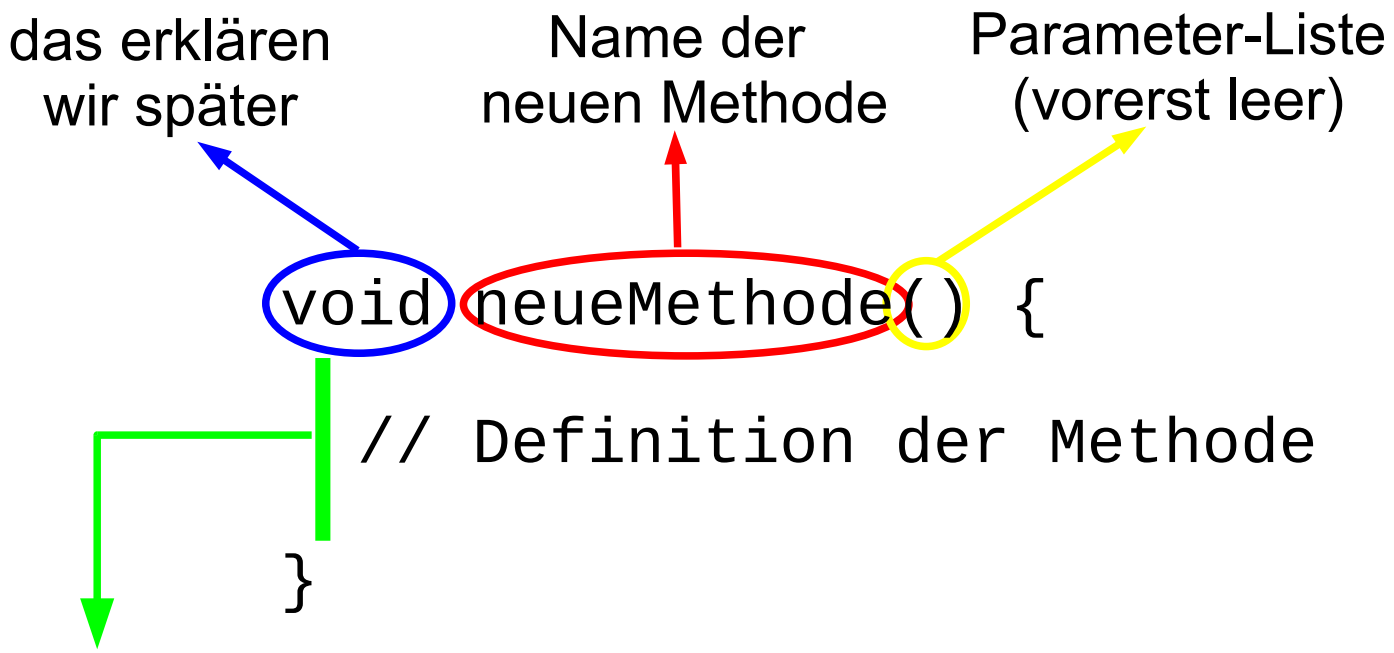
```
class NeueKlasse extends AlteKlasse
{
   <Definition neuer Methoden>
}
```

- neu definierte Methoden
  - können von allen Instanzen dieser Klasse verwendet werden
  - werden an alle Subklassen weitervererbt
  - (genauso wie geerbte Methoden)

Syntax zur Methoden-Definition

```
void neueMethode() {
  // Definition der Methode
}
```

Syntax zur Methoden-Definition



- Alle Befehle, die in der Definition der Methode stehen, werden ausgeführt, sobald
  - `x.neueMethode()` aufgerufen wird
  - und `x` ein Objekt der Klasse ist, für die die Methode definiert wird

Besonderheit

- Methoden werden für eine Klasse von Objekten definiert, nicht für konkrete Instanzen
  - daher ist der Name der Instanz, die diese Methode verwendet, nicht bekannt
    - die Instanz wird erst beim Aufruf der Methode bekanntgegeben (e.g., `karel.move()`)
  - aber es wird immer eine Instanz der Klasse sein, für die die Methode definiert wird
    - daher kennen wir die Fähigkeiten dieser Instanz

- *Daher können (und brauchen) wir den Namen der Instanz bei der Verwendung von (eigenen) Methoden innerhalb der Definition neuer Methoden nicht anzugeben!*

Beispiel

Klassendefinition

Methoden-Definition

```
class RechtsDreher extends Robot {

   void turnRight() {
      turnLeft();
      turnLeft();
      turnLeft();
   }

}

task {
   RechtsDreher karel =
         new RechtsDreher(1,1,0,East);
   karel.turnLeft();   // vererbte Methode
   karel.turnRight();  // neue Methode
}
```

karel ist eine Instanz der Klasse RechtsDreher

Weitervererbung ist möglich

- Natürlich sind abgeleitete Klassen vollwertige Klassen
  - Insbesondere können von abgeleiteten Klassen genauso Unterklassen definiert werden, die dann wiederum alle Methoden (also vererbte und neu definierte) erben.
- Beispiel:

```
class TreppenSteiger extends RechtsDreher {
   void climbStair()
   {
      move();
       turnLeft();   // geerbt von Robot
                     // (über RechtsDreher)
       move();
       turnRight();  // geerbt von RechtsDreher
   }
}
```

Treppen steigen: 3. Lösung

- Unter Verwendung der oben definierten Klassen
  - d.h. die Klassen `RechtsDreher` und `TreppenSteiger` müssen im File definiert werden
    (oder importiert werden, aber das erst später)

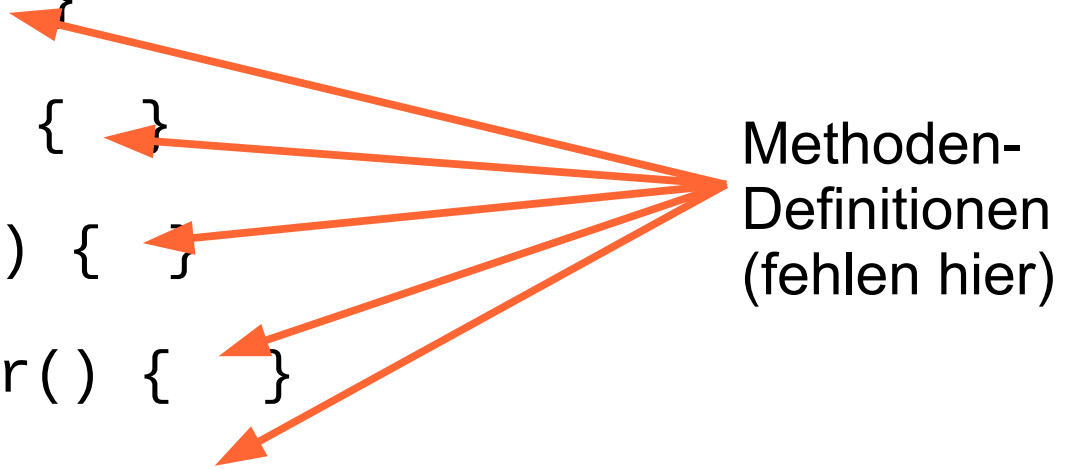
```
task
{
   TreppenSteiger karel =
             new TreppenSteiger(1,1,4,East);
   loop(4) {
     karel.putBeeper();
     karel.climbStair();
   }
}
```

UrRobot

- Alle Roboter Funktionen, die wir bisher verwendet haben, sind allen Robotern gemein
  - ***Was wir dabei verschwiegen haben:*** Die Klasse, die diese Funktionen definiert, heißt eigentlich nicht `Robot`, sondern `UrRobot` (wie Urgroßvater)
- `Robot` ist eine Unterklasse von `UrRobot`, die darüber hinaus noch einige weitere Methoden definiert
  - insbesondere Methoden, die es erlauben, den Zustand der Welt abzufragen
- Natürlich kann man der Einfachheit halber einen `Robot` auch für Aufgaben verwenden, für die ein `UrRobot` genügt hätte
  - wie wir das bisher getan haben
  - Warum geht das? → Vererbung

Spezifikation des UrRobots

```
class UrRobot {

   void move() {  }

   void turnOff() {  }

   void turnLeft() {  }

   void pickBeeper() {  }

   void putBeeper() {  }

}
```



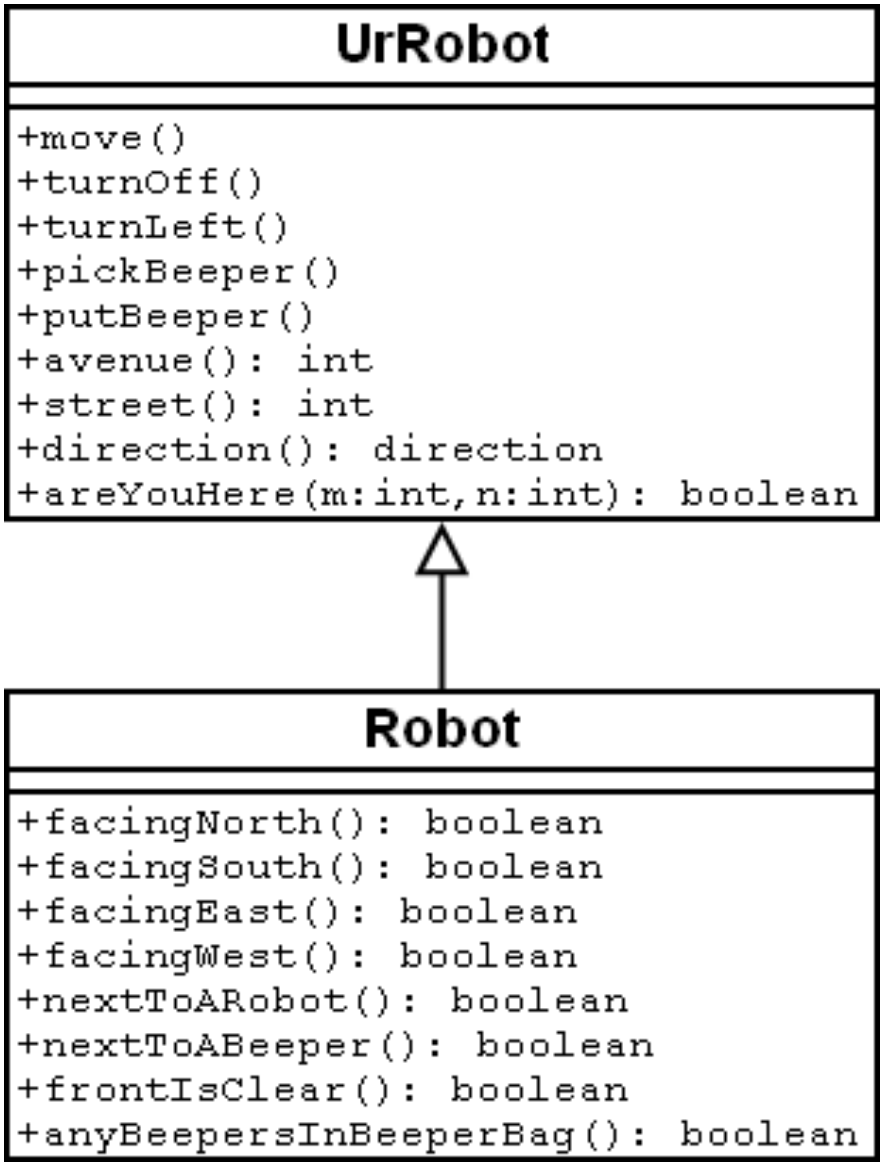
Status-Abfragen für UrRobot

- `avenue(), street()`
  - in welcher Avenue bzw. Straße befindet sich der Roboter
- `direction()`
  - in welche Richtung schaut der Roboter
- `areYouHere(n,m)`
  - befindet sich der Roboter an der Ecke n-te Straße und `m`-te Avenue
  - `n` und `m` sind Parameter, die die genauen Ko-ordinaten der Anfrage spezifizieren

Zusätzliche Status-Abfragen von Robot

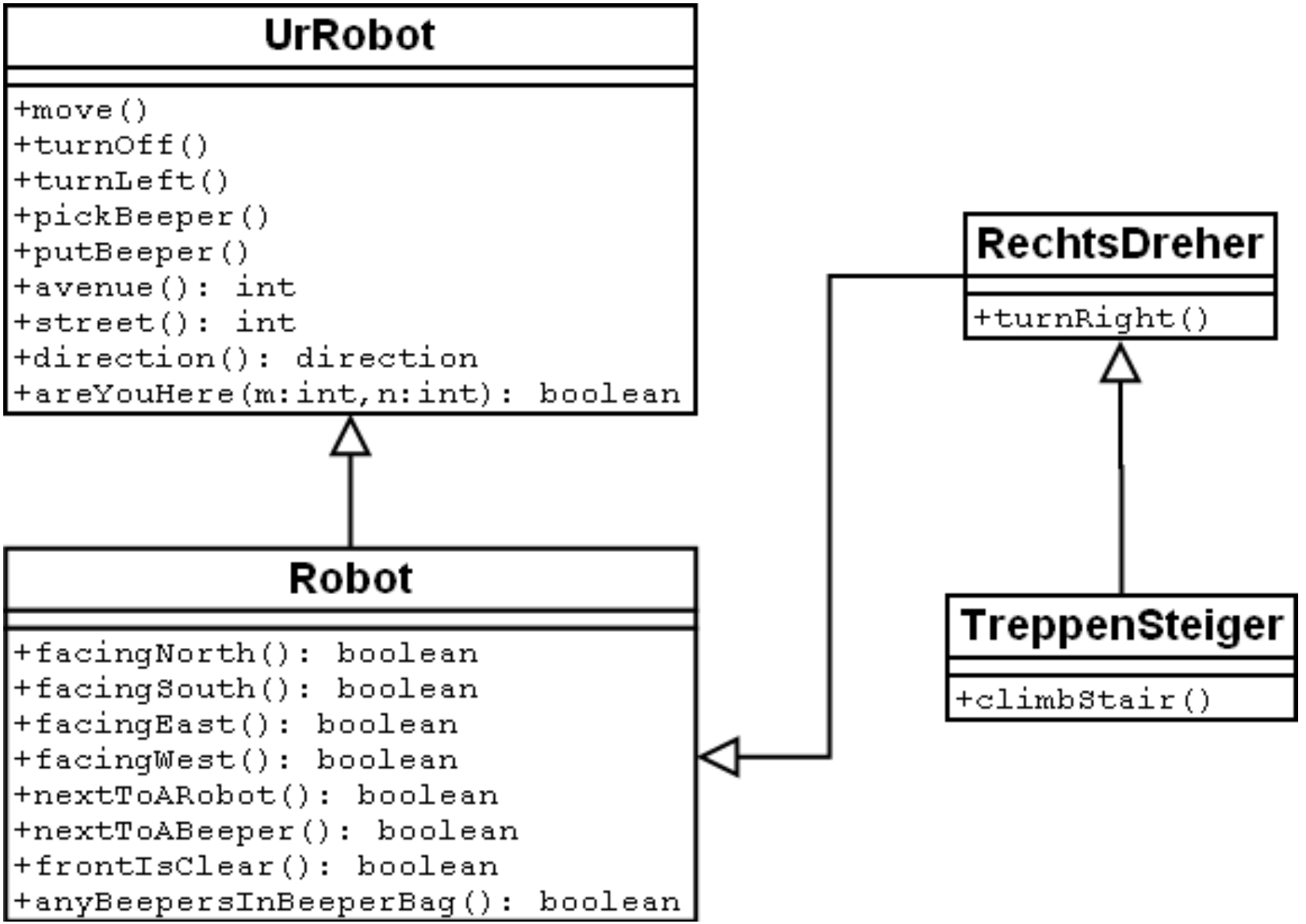
- `facingNorth(), facingSouth(), facingEast(), facingWest()`
  - blickt der Roboter nach Norden? Süden? Osten? Westen?
- `nextToABeeper(), nextToARobot()`
  - befinden sich auf dem Feld, auf dem der Roboter steht, Beeper? Oder andere Roboter?
- `frontIsClear()`
  - kann der Roboter vorwärts gehen, oder befinden sich Hindernisse im Weg (z.B. Neutronium Mauern)
- `anyBeepersInBeeperBag()`
  - schleppt der Roboter noch Beeper herum?

UML Diagramm



- UML
  - Unified Modeling Language
- einfache Diagramme, um Klassenabhängigkeiten darzustellen
  - und mehr...

UML Diagramm



Variablen

- Variablen können Zwischenresultate speichern
  - Wofür?
    - Manche Zwischenresultate sind aufwendig zu berechnen
    - Wenn Sie öfter gebraucht werden, ist es besser, sie sich zu merken, als sie jedes Mal neu zu berechnen
  - bei jedem Auftreten der Variable wird ihr Wert eingesetzt
- Eine Variable ist charakterisiert durch
  - Name
  - Typ (Welche Art von Objekt kann sich die Variable merken?)
  - Wert
- Wir hatten Variablen bereits!
  - Die Namen der Roboter sind nichts anderes als Variablen vom Typ Roboter, in der eine bestimmte Roboter-Instanz gespeichert wurde

Typen

Neben Robotern gibt es noch folgende Typen von Objekten, die in Programmen (z.B. in Spezifikationen) verwendet werden können:

- `int`: ganze Zahlen
  - Nummern von Straßen und Avenues
  - Anzahl der Beeper
- `direction`: Richtungen
  - `North, South, East, West`
- `boolean`: Wahrheitswerte
  - `true, false`
- `void`: Null-Typ
  - verwendet bei Methoden-Deklarationen, die nichts berechnen (war bisher immer der Fall)

Variablen-Deklaration

- `int n;`
  - Spezifiert, daß die Variable n vom Typ `int` sein soll.
  - n wird dabei noch kein Wert zugewiesen!
    - ACHTUNG: Laufzeit-Fehler falls n ohne Initialisierung im Programm verwendet wird!
  - Zuweisung kann in der Folge durch ein Statement wie
    `n = 5;`
    erfolgen.
- `int n = 4;`
  - Spezifiert, daß die Variable n vom Typ `int` sein soll UND weist ihr den Wert `4` zu.
  - Dieser Wert kann natürlich später durch ein Statement wie
    `n = 5;`
    überschrieben werden.
  - Wir hatten diesen Fall bereits:
    `Robot karel = new Robot(1,1,1,East);`

ACHTUNG

- Absolut letzter Termin für die Übungsanmeldung:
  - Freitag 17.11.
- Danach gibt es keine Ausnahmen mehr

Scope von Variablen-Deklarationen

- Eine Variablen-Deklaration ist nur innerhalb des sie umbegebenden Blocks { } gültig

Korrekt

```
{
  {
    int n = 4;

    // Anweisungen

    n = 5;
  }
}
```

Falsch

```
{
  {
    int n = 4;

    // Anweisungen
  }

  n = 5;
}
```

n ist hier undefiniert!

Scope von Variablen-Deklarationen

- Eine Variablen-Deklaration ist nur innerhalb des sie umbegebenden Blocks { } gültig
- ...und darf dort nur einmal deklariert werden!

Korrekt

```
{
  {
    int n = 4;

    // Anweisungen

    n = 5;
  }
}
```

Auch Falsch

```
{
  {
    int n = 4;

    // Anweisungen

    int n = 5;
  }
}
```

n ist hier doppelt definiert!

Scope von Variablen

- Zwei Variablen gleichen Namens in verschiedenen Scopes sind als verschiedene Variable zu betrachten!

dieselbe Variable

```
{
  int n = 5;

  {
    n = 3;

    // Anweisungen
  }

  // hier ist n == 3!
}
```

verschiedene Variablen

```
{
  int n = 5;

  {
    int n = 3;

    // Anweisungen
  }

  // hier ist n == 5!
}
```

Verwendung von Typen

- Variablen-Deklaration
  - damit man weiß, welche Art von Objekt sie speichern können
- Methoden-Deklaration
  - damit man weiß, welche Art von Objekt von einer Methode berechnet wird
- Parameter-Listen
  - damit man weiß, welche Art von Objekt übergeben wird

Methoden für Berechnungen schreiben

Die Methode soll einen `boolean` Wert zurück-liefern

```
// Methode, um festzustellen, ob rechts frei ist
boolean rightIsClear() {

  // dreh Dich nach rechts
  turnRight();

  // Merken, ob nun vorne Platz ist
  boolean clear = frontIsClear();

  // Wieder in den alten Zustand zurückdrehen!
  turnLeft();

  // und das Ergebnis zurückgeben
  return clear;

}
```

`frontIsClear()` ist als `boolean` spezifiziert

`Clear` wird ebenfalls als `boolean` spezifiziert

`return` gibt an, welcher Wert als Ergebnis der Methode zurückgegeben werden soll. Das Ergebnis muß dem im Methodenkopf deklarierten Typ entsprechen.

Treppen steigen: 4. Lösung

- mit einer variablen Anzahl von Stufen!
  - die Anzahl der Stufen wird in der ersten Zeile festgelegt

```
task
{
   int n = 4;
   TreppenSteiger karel =
             new TreppenSteiger(1,1,n,East);
   loop(n) {
     karel.putBeeper();
     karel.climbStair();
   }
}
```

Treppen steigen: 4. Lösung

- mit einer variablen Anzahl von Stufen!
  - die Anzahl der Stufen wird in der ersten Zeile festgelegt

```
task
{
   int n = 4;
   TreppenSteiger karel =
            new TreppenSteiger(1,1,n,East);
   loop(n) {
     karel.putBeeper();
     karel.climbStair();
   }
}
```

Definiere die Variable n als ganze Zahl und weise ihr den Wert 4 zu

Die Variable wird 2-mal verwendet

Um karel eine andere Anzahl von Stufen steigen zu lassen, muß nur der Wert der Variablen geändert werden

Methoden mit Parametern

- Aber wäre es nicht noch besser, wenn wir eine Methode hätten, die n Beepers diagonal anordnet?
  - für beliebig wählbare Werte von `n`
- Problem:
  - Wie teile ich der Methode mit, daß ich n Beepers möchte?
- Lösung:
  - Parameter-Liste:
    - für jeden Parameter, der der Methode übergeben werden soll wird der Typ und der Name angegeben (wie bei einer Typ-Deklaration)
    - die einzelnen Parameter werden durch ein Komma getrennt

Treppen Steigen: 5. Lösung

```
// nicht vergessen: RechtsDreher definieren/importieren

class TreppenSteiger extends RechtsDreher {

    // nicht vergessen: climbStair() definieren

    // make a diagonal of length n
    void makeDiagonal(int n)
    {
        loop(n) {
          putBeeper();
          climbStair();
        }
     }
}

task {
   TreppenSteiger karel = new TreppenSteiger(1,1,4,East);
   karel.makeDiagonal(4);
}
```

Treppen Steigen: 5. Lösung

```
// nicht vergessen: RechtsDreher definieren/importieren

class TreppenSteiger extends RechtsDreher {

    // nicht vergessen: climbStair() definieren

    // make a diagonal of length n
    void makeDiagonal(int n)
    {
        loop(n) {
          putBeeper();
          climbStair();
        }
     }
}

task {
   TreppenSteiger karel = new TreppenSteiger(1,1,4,East);
   karel.makeDiagonal(4);
}
```

Die Methode erwartet ein Argument namens n, vom Typ `int`.

Hier wird n verwendet

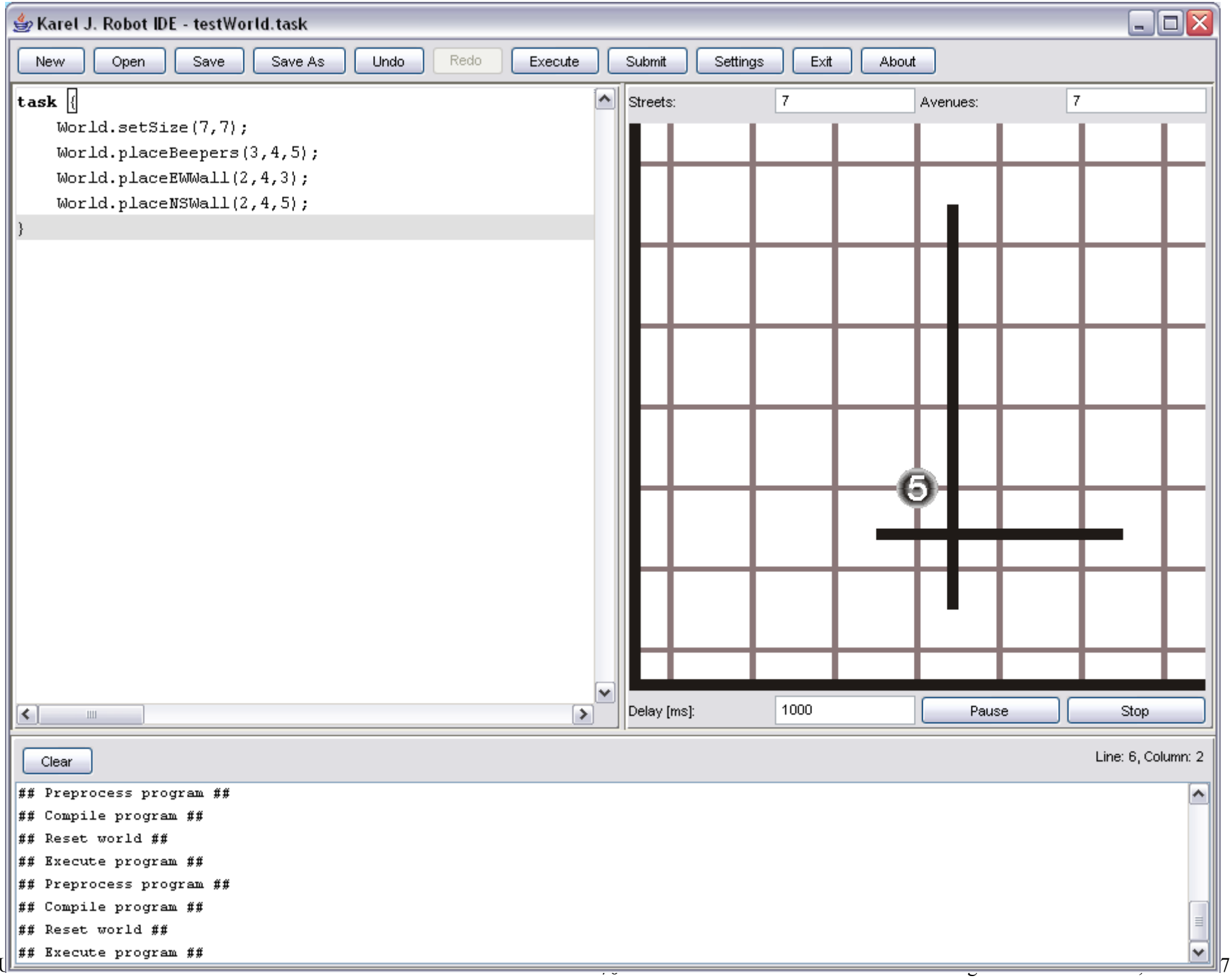
Hier wird der Wert von n festgelegt

Alternative Lösungen

- `makeDiagonal` kann nun Diagonalen beliebiger Länge zeichnen:
  - eine Diagonale der Länge 3:
    ```
    karel.makeDiagonal(3);
    ```
  - eine Diagonale der Länge 4, aber in 2 Teilstücken
    ```
    karel.makeDiagonal(3);
    karel.makeDiagonal(1);
    ```
  - eine Diagonale der Länge 4, in 4 Teilstücken:
    ```
    loop(4) {
      karel.makeDiagonal(1);
    }
    ```
- WICHTIG: Der Wert von n ist lokal für jeden Aufruf der Methode!
  - sobald die Methode fertig ist, verliert n seine Gültigkeit
  - der nächste Aufruf derselben Methode erhält ein neues n

Auf vielfachen Wunsch...

- Wie kann man andere Welten definieren?
  - Beeper setzen: `World.placeBeepers(s,a,n)`
    - plaziert `n` Beeper auf die Kreuzung Strasse `s` / Avenue `a`
  - Wände setzen: World.placeEWWall(s,a,n)
    - plaziert eine Wall zwischen Strasse `s` und `s+1`, beginnend zwischen Avenue `a-1` und `a`, und `n` Blocks lang
  - etc.
- Das sind Methoden der Klasse `World`
  - ACHTUNG: Diese Methoden sind sogenannte Klassen-Methoden
  - statt dem Namen einer Instanz dieser Klasse, muß man den Namen der Klasse angeben!
  - mehr dazu später...
- Eine genaue Auflistung aller Methoden findet sich unter
  - <KarelJIDE>/doc/api/World.html


Karel J. Robot IDE - testWorld.task
New
Open
Save
Save As
Undo
Redo
Execute
Submit
Settings
Exit
About
task {
World.setSize(7,7);
World.placeBeepers(3,4,5);
World.placeEWWall(2,4,3);
World.placeNSWall(2,4,5);
}
Streets:
7
Avenues:
7
5
Delay [ms]:
1000
Pause
Stop
Clear
Line: 6, Column: 2
Preprocess program
Compile program
Reset world
Execute program
Preprocess program
Compile program
Reset world
Execute program

Konditionale

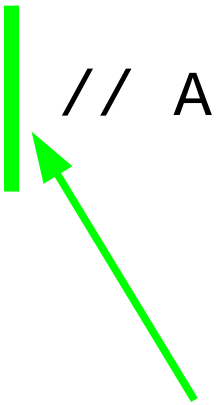
- Ein Problem gibt es aber doch?

```
task {
   TreppenSteiger karel = new TreppenSteiger(1,1,2,East);
   karel.makeDiagonal(4);
}
```

  - `karel` kann nicht 4 Beeper abladen, wenn er nur 2 Beeper trägt!
  - führt zu Laufzeitfehler!
- eine sauberere Lösung würde nur dann einen Beeper hinlegen, wenn noch einer da ist.
  - Realisierung:
    - Wenn-Dann Bedingungen (conditional execution)

Syntax: if

```
if ( Bedingung ) {

   // Anweisungen

}
```



Anweisungen werden nur ausgeführt falls die angegebene Bedingung den Wert true liefert.

Anmerkung: Falls nur eine einzige Anweisung folgt, können die geschwungenen Klammern auch weggelassen werden

Bedingungen

Eine Bedingung kann alles sein, das einen boolean Wert zurückliefert

- also Tests, die wahr oder falsch zurückliefern

- Tests, ob eine Variable einen bestimmten Wert hat

```
if (n == 4) { // n has the value 4
}
```

- eine Methode, die den Wert boolean zurückliefert

```
if (karel.anyBeepersInBeeperBag()) {
}
```

- eine Variable, die selbst von Typ boolean ist

```
boolean bprs = karel.anyBeepersInBeeperBag();
if (bprs) { // karel has beepers
}
```

Boole'sche Operatoren

Logische Verknüpfung von Bedingungen

- `!`: Verneinung
  - z.B. `!anyBeepersInBeeperBag()`
  - ist wahr, wenn die Bedingung falsch ist
- `&&`: Logisches Und
  - ist wahr, wenn beide Bedingungen wahr sind
  - z.B. `(1 <= i) && (i < 10)`
- `||`: Logisches Oder
  - z.B.`(direction()==East) || (direction()==West)`
  - ist wahr, wenn eine der beiden Bedingungen wahr ist

Vergleichs-Operatoren

```
int i,j;
```

Anm: Man kann auch mehrere Variablen in einer Zeile deklarieren!

- `i == j`
  - true wenn i den gleichen Wert wie j hat, false sonst
- `i != j`
  - true wenn i nicht den gleichen Wert wie j hat, false sonst
- `i > j`
  - true wenn der Wert von i größer als der von j ist
- `i < j`
  - true wenn der Wert von i kleiner als der von j ist
- `i >= j`
  - true wenn der Wert von i größer oder gleich dem von j ist
- `i <= j`
  - true wenn der Wert von i kleiner oder gleich dem von j ist

Treppen Steigen: 6. Lösung

```
// nicht vergessen: RechtsDreher definieren/importieren

class TreppenSteiger extends RechtsDreher {

    // nicht vergessen: climbStair definieren

    // make a diagonal of length n
    void makeDiagonal(int n)
    {
        loop(n) {
          if (anyBeepersInBeeperBag())
             putBeeper();
          climbStair();
        }
     }
}

task {
   TreppenSteiger karel = new TreppenSteiger(1,1,2,East);
   karel.makeDiagonal(4);
}
```

Ist noch ein Beeper da?

karel hat nur 2 Beeper

...aber möchte 4 verwenden

if-else

- oft ist es notwendig, bei Eintreten einer Bedingung eine bestimmte Handlung zu setzen, bei Nicht-Eintreten eine andere

Anweisungen werden nur ausgeführt falls die Bedingung den Wert `true` liefert.

Anweisungen werden nur ausgeführt falls die Bedingung den Wert `false` liefert.

```
if ( Bedingung ) {
    // Anweisungen
}
else {
    // andere Anweisungen
}
```

Beispiel if-then-else

```
// make a diagonal of length n
    void makeDiagonal(int n)
    {
        loop(n) {
          if (anyBeepersInBeeperBag())
             putBeeper();
          else
             System.out.println("Keine Beeper mehr!");
          climbStair();
        }
     }
```

Gibt den Text
Keine Beeper mehr!
im Statusfenster aus

Textausgabe

Mit Hilfe von `System.out.print` und `System.out.println` können Sie Text auf dem Bildschirm ausgeben

- `System.out.println(“Hello world!”);`
  - Gibt den Text Hello World! aus, und beginnt danach eine neue Zeile
- `System.out.print(i);`
  - Gibt den Inhalt der Variablen i aus
- `System.out.println(“i hat den Wert “ + i);`
  - das gleiche wie:

```
System.out.print(“i hat den Wert ”);
System.out.println(i);
```

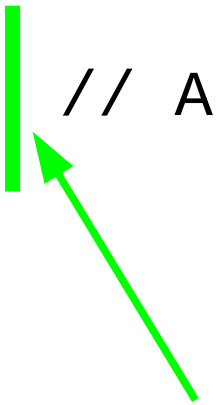
- **Anmerkung:** Warum der Name so umständlich ist, kommt später bei Java-Programmierung

while-Schleife

- Ganz optimal ist unsere Lösung immer noch nicht:
  - karel legt zwar nur mehr Beeper nieder, wenn er wirklich noch welche hat
  - aber er läuft noch unnötig weiter
- besser wäre es, wenn er nach dem letzten Beeper stehen bleiben würde
  - in anderen Worten:
    - solange (karel noch Beeper trägt) soll er weiter machen
- Lösung:
  - while-Schleife:
    - `while (anyBeepersinBeeperbag()) { }`

Syntax: while

```
while ( Bedingung ) {
   // Anweisungen
}
```



Anweisungen werden so lange ausgeführt, so lange die angegebene Bedingung den Wert true liefert.

Anmerkung: Falls nur eine einzige Anweisung folgt, können die geschwungenen Klammern auch weggelassen werden

Treppen Steigen: 7. Lösung

```
// nicht vergessen: RechtsDreher definieren/importieren

class TreppenSteiger extends RechtsDreher {

    // nicht vergessen: climbStair definieren

    // make a diagonal using all beepers
    void makeDiagonal()
    {
        while (anyBeepersInBeeperBag()) {
          putBeeper();
          climbStair();
        }
     }
}

task {
   TreppenSteiger karel = new TreppenSteiger(1,1,2,East);
   karel.makeDiagonal();
}
```

kein Argument, Karel verteilt nun immer alle Beeper

Ist noch ein Beeper da? Dann mach noch eine Runde...

Beliebter Fehler: Endlosschleife

- Was passiert hier?

```
while (anyBeepersInBeeperBag()) {
   climbStair();
}
```

- Die Anweisungen in der Schleife (hier nur eine Anweisung, `climbStair`), haben keinen Einfluß auf den Wahrheitswert der Bedingung
  - das heißt, wenn die Bedingung einmal erfüllt ist, ist sie immer erfüllt
  - deswegen wird die Bedingung nie `false` liefern
  - und der Roboter wird auf immer und ewig Stufen steigen!

→ Immer kontrollieren, ob der Wahrheitswert der Bedingung innerhalb der Schleife verändert wird!

Überschreiben von Methoden

- Wir haben gelernt, Roboter mit neuen Fähigkeiten (i.e., Methoden) zu definieren
- Man kann aber auch die Spezifikation von bekannten Fähigkeiten überschreiben oder modifizieren
  - Dazu verwendet man die gleiche Syntax wie zur Definition neuer Methoden
- Zur Definition kann man zusätzlich zu allen anderen Methoden auch noch die Methoden der Überklasse verwenden!
  - Identifikation der Überklasse durch das Wort `super`

Beispiel

- wir wollen einen Roboter, der bei jeder `move`-Anweisung gleich 2 Schritte auf einmal ausführt

```
class Racer extends UrRobot {

    // mach 2 Schritte auf einmal
    void move()
    {
        super.move();
        super.move();
    }
}

task {
    Racer karel = new Racer(1,1,0,East);
    karel.move();
}
```

Definition durch 2 Aufrufe der Methode der Überklasse `UrRobot`

karel macht 2 Schritte!

Beispiel (2)

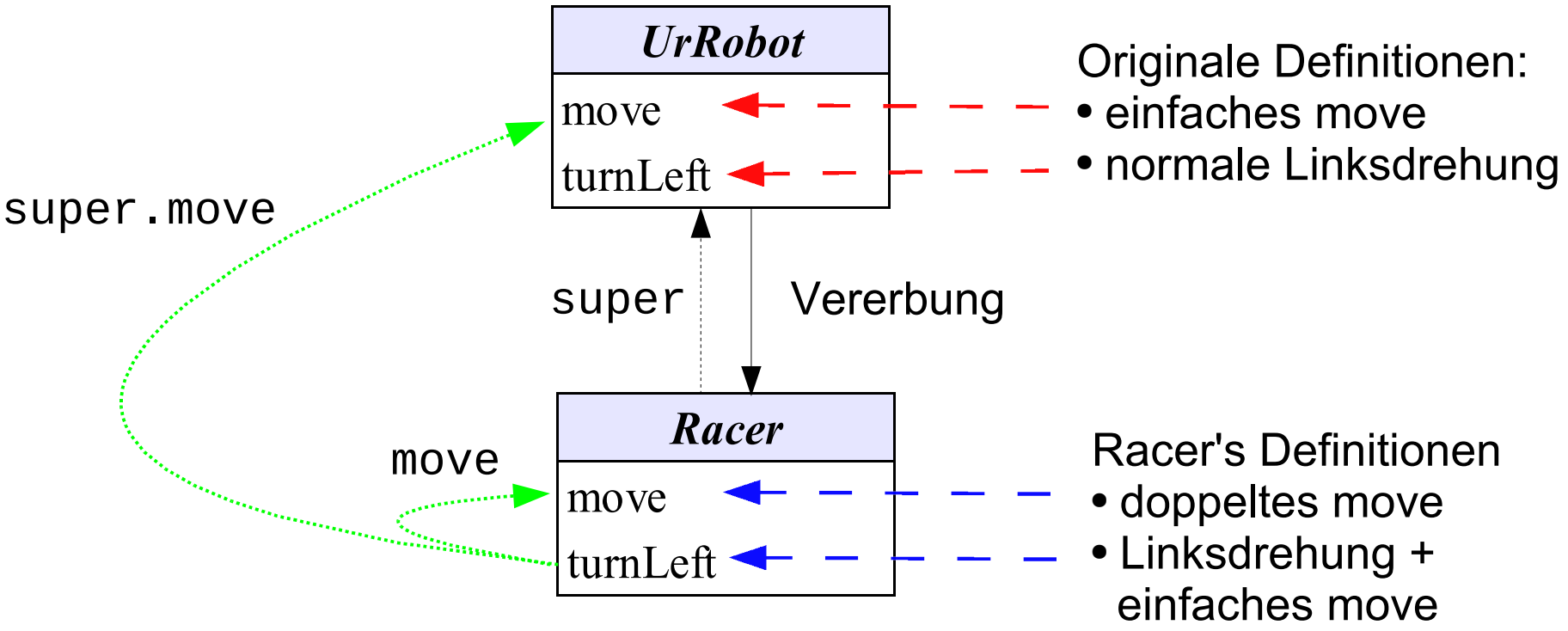
- Zusätzlich wollen wir, daß Racer nach jeder Drehung automatisch einen Schritt vorwärts geht.
- Wir definieren noch eine Methode für diese Klasse:

```
void turnLeft()
{
   super.turnLeft();
   move();
}
```

- oder doch so?

```
void turnLeft()
{
   super.turnLeft();
   super.move();
}
```

Beispiel (3)

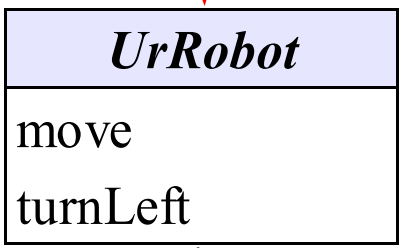


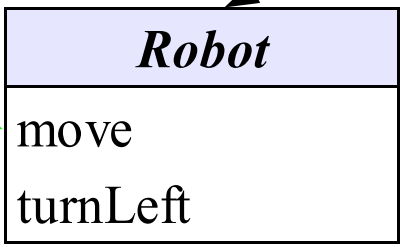
Um in der Definition von Racer's `turnLeft` ein einfaches `move` zu realisieren, muß daher `super.move` aufgerufen werden!

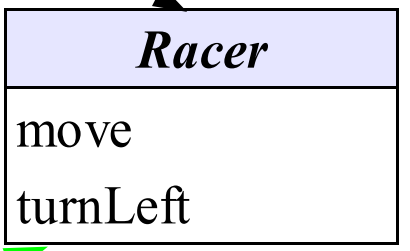
Statischer vs. Dynamischer Typ

```
task {

  UrRobot karel;

    karel = new Robot(1,3,0,East);
    karel.move();

    karel = new Racer(1,3,0,East);
    karel.move();

}
```

Statischer Typ: UrRobot
karel wird als
UrRobot deklariert







Dynamischer Typ:
Robot/Racer
Jeder Robot und jeder
Racer ist auch ein UrRobot
und erfüllt daher die
verlangten Anforderungen

Statischer und Dynamischer Typ

- Statischer Typ
  - legt fest, welche Methoden verwendet werden dürfen
- Dynamischer Typ
  - legt fest, wie diese Methoden implementiert sind

- Beispiel 1:

```
Robot karel = new RechtsDreher(1,3,0,East);
```

  - `karel` wird mit statischem Typ `Robot` und dynamischem Typ `RechtsDreher` deklariert
  - das heißt, daß `karel` nur die Methoden verwenden darf, die jeder `Robot` kann
  - insbesondere darf `karel` kein `turnRight()` durchführen, da er sich als einfacher Robot verkleidet hat.

Statischer und Dynamischer Typ

- Statischer Typ
  - legt fest, welche Methoden verwendet werden dürfen
- Dynamischer Typ
  - legt fest, wie diese Methoden implementiert sind

- Beispiel 2:

```
Robot karel = new Racer(1,3,0,East);
```

  - `karel` wird mit statischem Typ `Robot` und dynamischem Typ `Racer` deklariert
  - das heißt, daß `karel` nur die Methoden verwenden darf, die jeder `Robot` kann
  - es wird aber die in `Racer` definierte Version ausgeführt (also bei einem `move()` ein Doppelschritt gemacht)

Statischer und Dynamischer Typ

- Statischer Typ:
  - der Typ mit dem die Variable deklariert wird
  - dieser Typ bestimmt
    - welche Werte der Variablen zugewiesen werden können
    - welche Methoden verwendet werden können (nur die, die für den statischen Typ deklariert wurden)
  - eine Variable kann im Gültigkeitsbereich ihrer Deklaration immer nur einen Statischen Typ haben
- Dynamischer Typ:
  - der Typ des Objekts, das der Variablen zugewiesen wird
    - muß ein Untertyp des Statischen Typs sein
    - bei Aufruf einer Methode wird die Methode des Dynamischen Typs verwendet
  - eine Variable kann im Gültigkeitsbereich ihrer Deklaration auch mehrere dynamische Typen haben

Statischer und Dynamischer Typ Polymorphie

Polymorphie:
gleiche Anweisung, aber verschiedene Bedeutung.

```
task {

  UrRobot karel;

  karel = new Robot(1,3,0,East)
  karel.move();

  karel = new Racer(1,3,0,East)
  karel.move();

  karel = new TreppenSteiger(1,3,0,East);
  karel.climbStairs();

}
```

Karel ist ein Robot: 1 Schritt

Karel ist ein Racer: 1 Doppel-Schritt

Karel ist als UrRobot deklariert.
Ein UrRobot kann climbStairs nicht!
→ Fehler beim Übersetzen!

Was passiert hier?

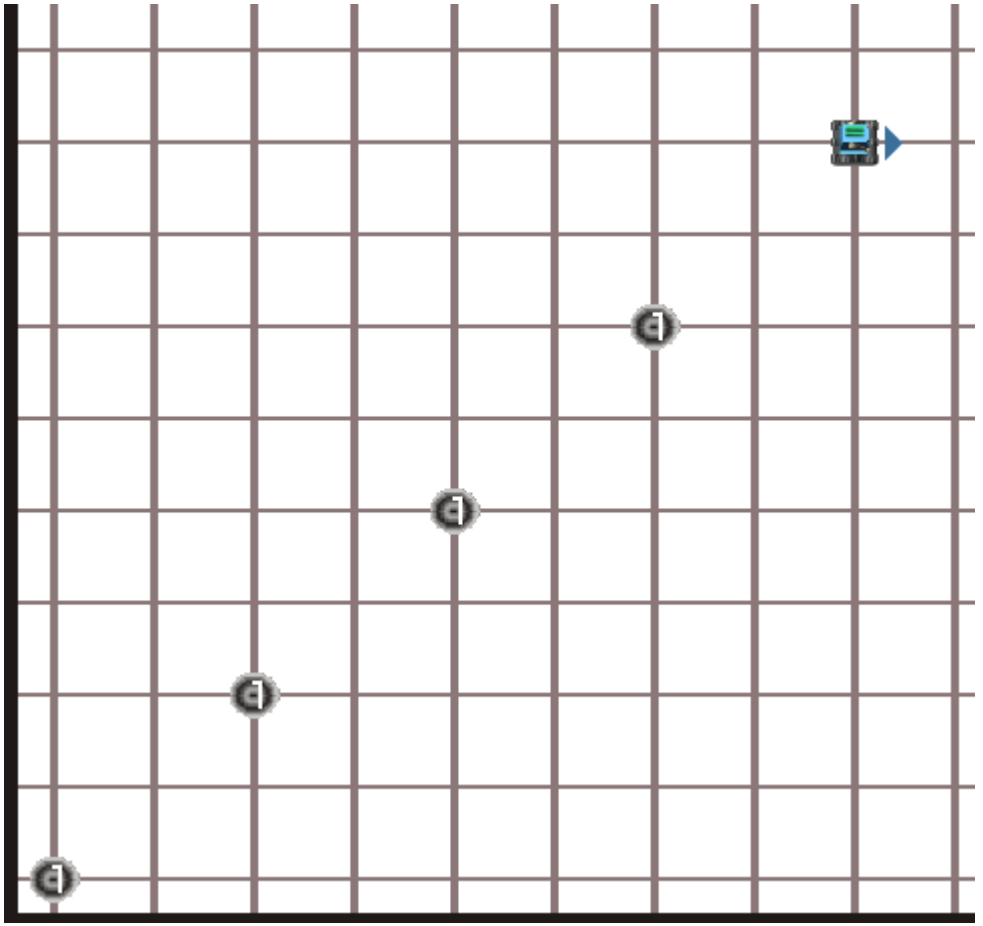
```
class RacerSteiger extends TreppenSteiger
{
    void move() {
        super.move();
        super.move();
    }
}

task {
    RacerSteiger gulliver =
          new RacerSteiger(1,1,4,East);
    gulliver.makeDiagonal();
}
```

Bei jeder Instanz von `RacerSteiger` erfolgt bei *jedem* Aufruf von `move()` ein Doppelschritt...

... auch wenn `move()` nicht direkt aufgerufen wird, sondern wie hier über `makeDiagonal()`!!

Das passiert hier!

Daher klettert `RacerSteiger` Treppen der Höhe 2!



Objekt-Variablen

- Variablen können nicht nur innerhalb einer Methode definiert werden
  - sondern auch für eine gesamte Klasse definiert werden.
- Die Objekt-Variable kann analog zu einer Methode verwendet werden
  - von allen Methoden dieser Klasse mit dem Namen der Variable
  - von allen Methoden außerhalb der Klasse und von der `task` Definition durch Vorsetzen des Namens der Instanz
- ACHTUNG:
  - Jede Instanz hat eine eigene Kopie dieser Variable. Eine Änderung der Variable in einer Instanz bewirkt keine Änderung in einer anderen Instanz!

Variable Schrittweite

```
class RacerSteiger extends TreppenSteiger {

    // Abspeichern der Schrittweite
    int schrittweite = 1;

    // Verwenden von Schrittweite
   void move() {
      loop(schrittweite) {
         super.move();
      }
   }
}

task {
   RacerSteiger gulliver = new RacerSteiger(1,1,4,East);
   gulliver.schrittweite = 2;
   gulliver.makeDiagonal();
}
```

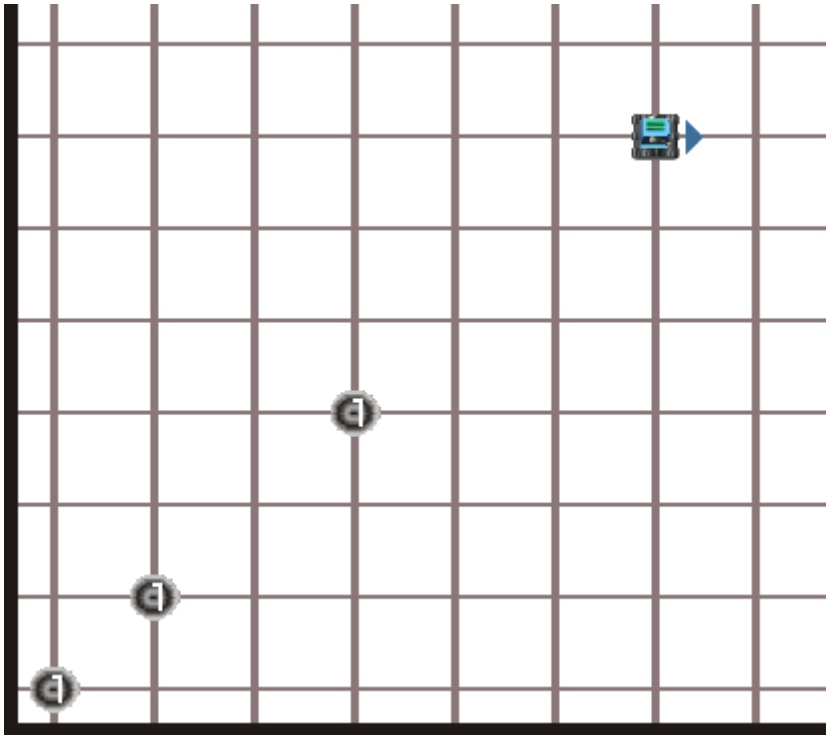
Üblicherweise wollen wir Schrittweite 1

Schrittweite bestimmt die Anzahl der `move`s, die eine Instanz des `RacerSteiger`s ausführt

Hier wollen wir aber Schrittweite 2

Weiterverwendung von Objekt-Variablen

```
task {
    RacerSteiger gulliver =
       new RacerSteiger(1,1,4,East);
    gulliver.schrittweite = 1;
    gulliver.putBeeper();
    gulliver.climbStair();
    gulliver.schrittweite = 2;
    gulliver.putBeeper();
    gulliver.climbStair();
    gulliver.schrittweite = 3;
    gulliver.putBeeper();
    gulliver.climbStair();
}
```



Objekt-Variablen sind klarerweise variabel und können dynamisch verändert werden!

Konstruktoren

- Bis jetzt hatten wir Konstruktoren immer verwendet, um neue Instanzen zu definieren

```
Robot karel = new Robot(1, 2, 0, North);
```

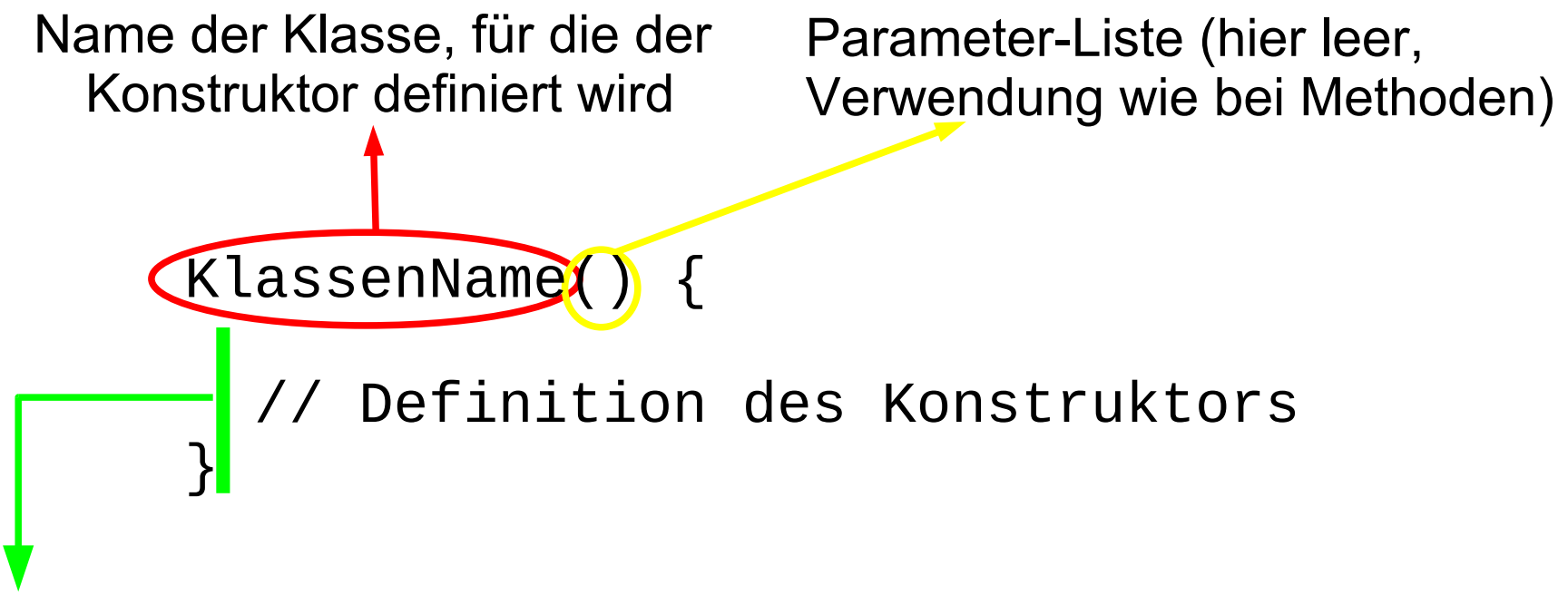
- Für neue Konstruktoren lassen sich aber eigene Konstruktoren definieren
  - können eine eigene Parameter-Liste haben
  - Beispiel:
    - für `RacerSteiger` würde sich anbieten, die Schrittweite bereits als fünftes Argument im Konstruktor zu übergeben

- Spezieller Konstruktor:
  - `super(...)`: ruft den Konstruktor der Überklasse auf

Syntax zur Definition eines Konstruktors

```
KlassenName() {

  // Definition des Konstruktors
}
```

Syntax zur Definition eines Konstruktors



- Alle Befehle, die in der Definition des Konstruktors stehen, werden ausgeführt, sobald
  - `new KlassenName()` aufgerufen wird
  - Resultat des Aufrufs ist eine neue Instanz dieser Klasse

Beispiel

```
class RacerSteiger extends TreppenSteiger {

   int schrittweite = 1;

   RacerSteiger(int s, int a, int b,
                direction d, int schritt)
   {
      super(s,a,b,d);
      schrittweite = schritt;
   }
}


task {
    RacerSteiger gulliver =
       new RacerSteiger(1,1,4,East,2);
    gulliver.makeDiagonal();
}
```

Beispiel

```
class RacerSteiger extends TreppenSteiger {

   int schrittweite = 1;

   RacerSteiger(int s, int a, int b,
                direction d, int schritt)
   {
      super(s,a,b,d);
      schrittweite = schritt;
   }
}

task {
    RacerSteiger gulliver =
       new RacerSteiger(1,1,4,East,2);
    gulliver.makeDiagonal();
}
```

Definition des Konstruktors

Konstruktor hat die vier bekannten Argumente von `Robot`-Konstrukturen plus ein neues (`schritt`)

Zuerst wird ein Objekt der Überklasse konstruiert (ein `TreppenSteiger`)

dann wird die Schrittweite mit dem übergebenen Wert initialisiert

Beim Aufruf des Konstruktors *muß* nun die Schrittweite als 5. Argument angegeben werden

for-Schleife

```
for (<Init>; <Test>; <Update>) {
    // Anweisungen
}
```

- `<Init>`: Initialisierung der Schleife
  - wird genau einmal am Beginn ausgeführt
  - z.B. `int i = 1`
- `<Test>`: Abbruchkriterium
  - wird vor jedem Schleifendurchlauf überprüft
  - solange der Test `true` retourniert, werden die Anweisungen der Schleife durchgeführt.
  - z.B. `i < 10`
- `<Update>`: Veränderung der Schleifenvariablen
  - wird nach jedem Schleifendurchlauf durchgeführt
  - z.B. `i = i + 1` bzw. in Kurzschreibweise `i++`

Beispiel

- `while`-Schleife zur Berechnung von $\sum_{i=1}^{9} i$ und $\prod_{i=1}^{9} i$

```
int sum = 0;
int prod = 1;
int i = 1;

while (i < 10) {
    sum = sum + i;
    prod = prod * i;
    i++;
}
```

Initialisierung der Summe und des Produkts

Initialisierung einer Zählvariablen

Überprüfen der Schleifenbedingung

Addieren des aktuellen Elements zur Summe
Multiplizieren des aktuellen Elements zum Produkt

Hochzählen der Zählvariablen

Anmerkung:
`i++` steht für `i = i+1`

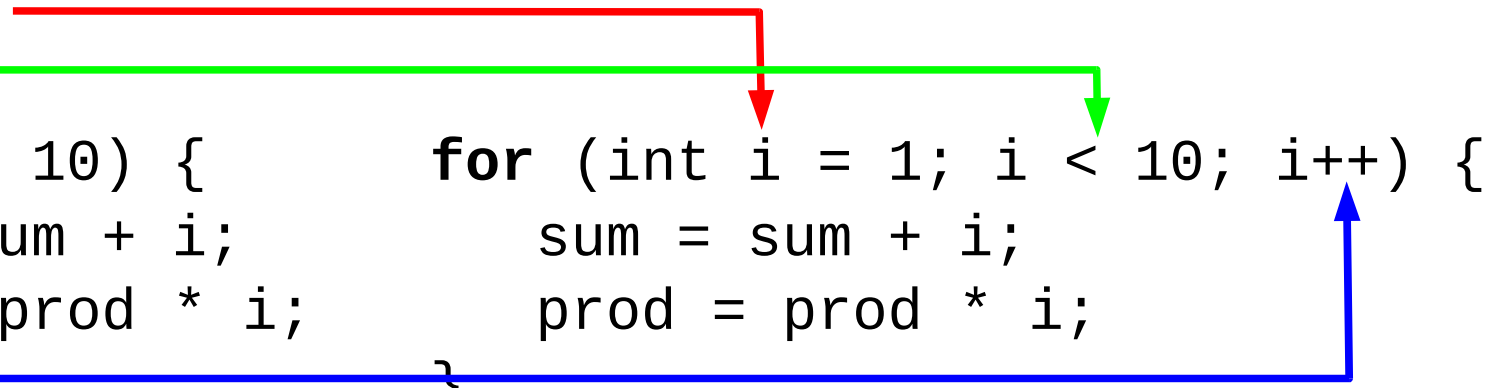
Beispiel

- `while`-Schleife

```
int sum = 0;
int prod = 1;
int i = 1;

while (i < 10) {
    sum = sum + i;
    prod = prod * i;
    i++;
}
```

- äquivalente `for`-Schleife

```
int sum = 0;
int prod = 1;

for (int i = 1; i < 10; i++) {
    sum = sum + i;
    prod = prod * i;
}
```



Anmerkung:

`i++` steht für `i = i+1`

Reelle Zahlen

- Bis jetzt hatten wir nur ganze Zahlen (int)
- Zwei Typen von reell-wertigen Zahlen:
  - `float`
    - definiert eine Gleitkommazahl
    - Beispiel: `float pi = 3.14159`
  - `double`
    - wie `float`, nur werden die Zahlen doppelt so genau abgespeichert (d.h. die Mantisse ist doppelt so lang)
    - wird häufiger als `float` verwendet
- **Beachte:**
  - Das Ergebnis einer arithmetischen Operation ist immer vom selben Typ wie die Operanden!
  - Beispiel:

```
int i = 11;
int j = 3;
System.out.println(i/j);
// 3 wird ausgegeben
```

```
double i = 11;
double j = 3;
System.out.println(i/j);
// 3.666666 wird ausgegeben!
```

Konvertierung von Zahlentypen

- Die verschiedenen Zahlentypen `int`, `float`, `double` können nicht so einfach ineinander übergeführt werden.
  - der Compiler erlaubt nur Zuweisungen zwischen Variablen gleichen Typs!
  - alle Variablen innerhalb einer Berechnung (eines arithmetischen Ausdrucks) müssen denselben Typ haben!
  - Grund: Verschiedene Zahlentypen werden ja (wie wir gesehen haben) intern verschieden abgespeichert!
- Dazu benötigt es eine explizite Konvertierung
  - Diese erfolgt indem man den gewünschten Typ in runden Klammern vor die Variable schreibt!

  - Falsch:

```
int i = 5;
double x = 10.0;
i = i + x;
```

Richtig:

```
int i = 5;
double x = 10.0;
i = i + (int) x;
```

Automatische Konvertierung

- der Compiler führt eine automatische Konvertierung durch, wenn sie ohne Beeinträchtigung der Genauigkeit möglich ist
  - Also: `int` → `float` → `double`
  - Aber nicht: `double` → `float` → `int`

- **Achtung:**
  - Die Konvertierung findet erst statt, wenn sie aufgrund der Inkompatibilität der Typen notwendig ist
    - das ist oft nicht dann, wenn man sie erwartet
  - Beispiel:

```
int x = 5;
int y = 3;
double z1 = x / y;
double z2 = (double) x / y;
```

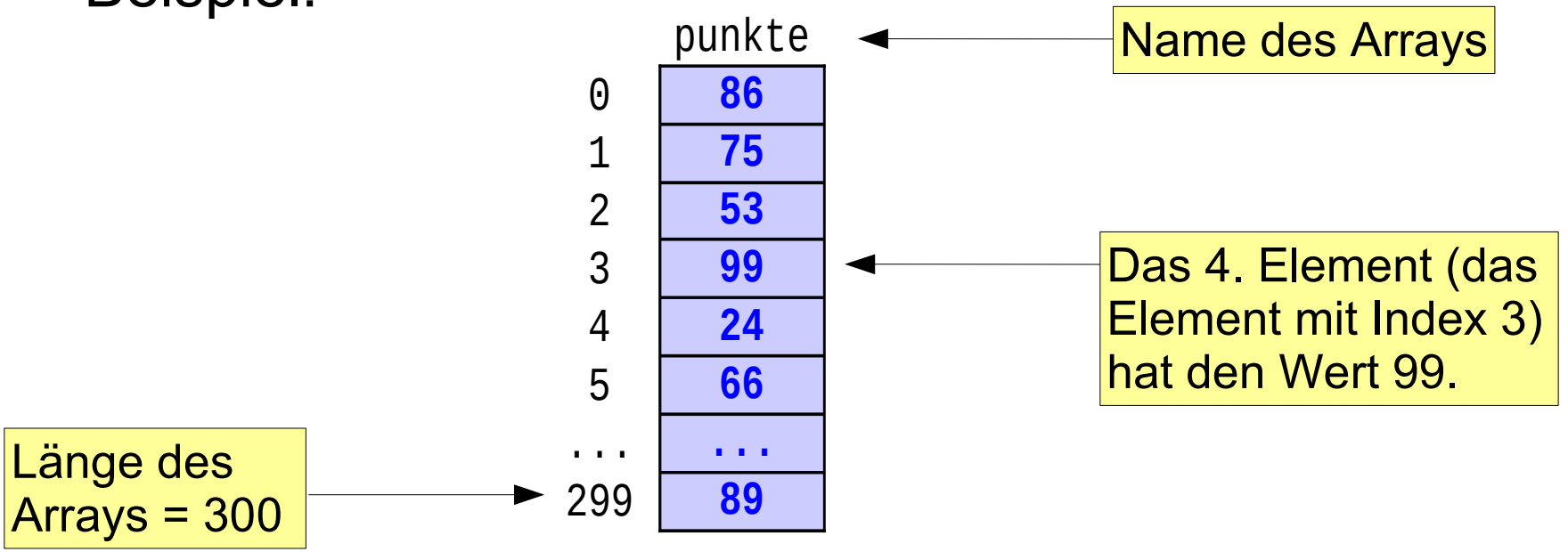
Division zweier `int`-Zahlen
→ `int` Resultat wird auf double konvertiert

`(double) x` bewirkt Konvertierung von `x`
→ y wird auch auf double konvertiert, erst dann wird dividiert!

Arrays

- Ein Array ist eine geordnete Liste von Variablen
  - gleichen Datentyps
  - die mit dem selben Namen angesprochen werden
  - aber jedes Element hat eine Nummer (Index)
    - ein Array mit n Elementen ist von 0 bis n-1 durchnumeriert

- Beispiel:

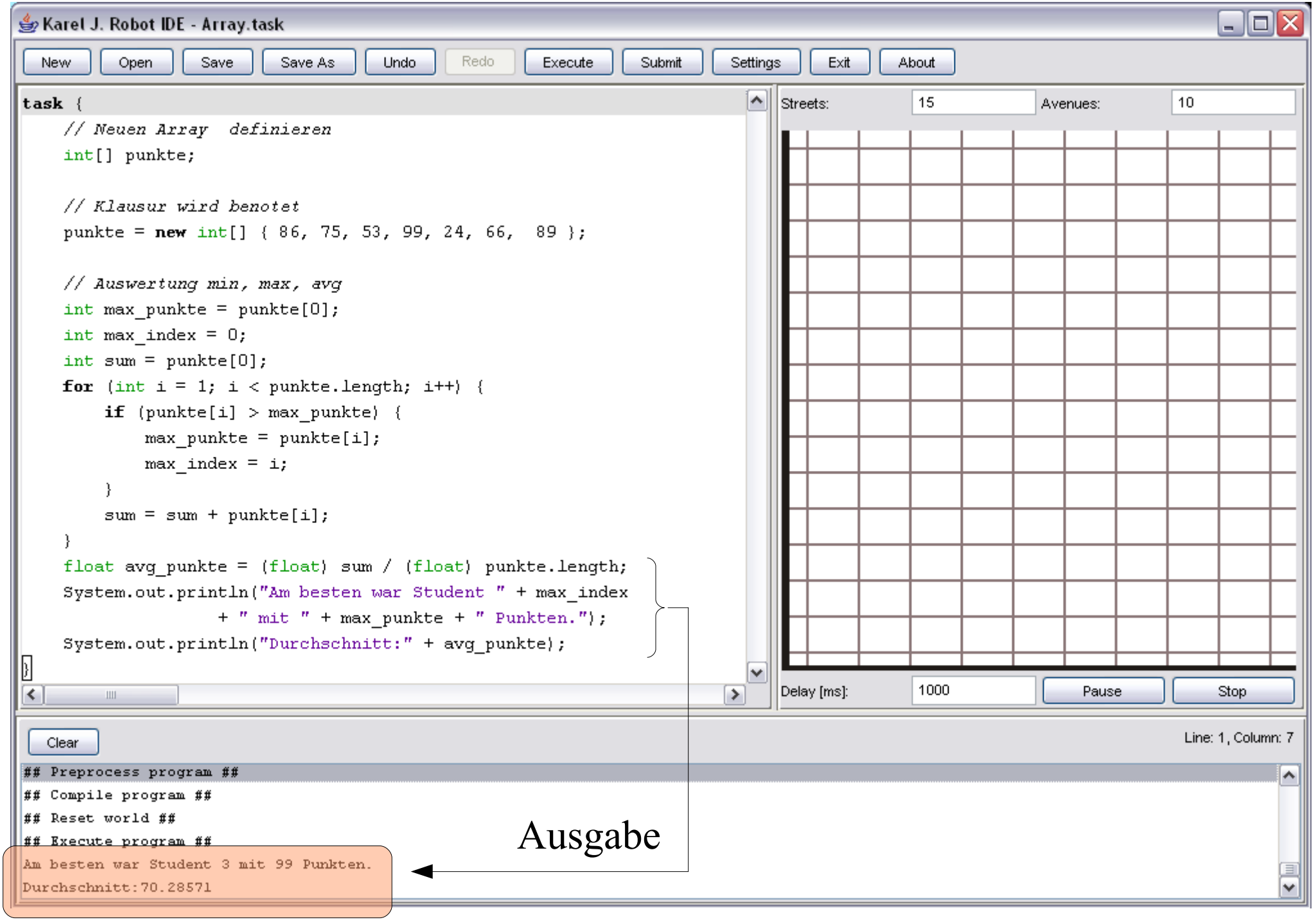


Arrays in Java

- Deklaration einer Array-Variable
  - *Typ*`[]` *Arrayname*
  - z.B. `int[] punkte;`
- Erzeugung eines Arrays
  - new *Typ*`[`*n*`]` oder `new` *Typ*`[] { <Elements> }`
  - z.B. `punkte = new int[3];`
  - z.B. `punkte = new int[] { 86, 75, 53 };`
- Lesen oder Schreiben von Array-Elementen
  - durch Angabe des Namens der Array-Variable
  - und der Indexnummer des Eintrags (0...n-1)
  - z.B. `punkte[2] = 53; int p = punkte[2];`
- Anzahl der Elemente eines Arrays
  - wird bei der Erzeugung unveränderbar festgelegt
  - Abfrage mit *Arrayname*`.length`
  - z.B. `punkte.length`

Beispiel

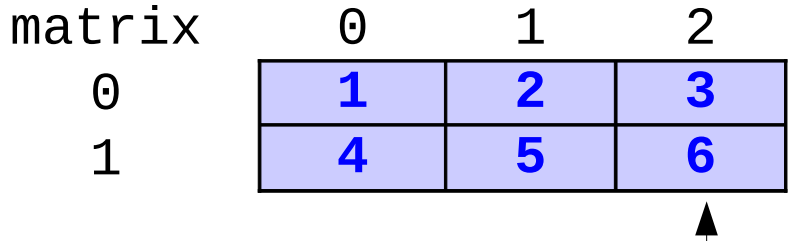
```
// Neuen Array für Punktzahlen definieren
int[] punkte;

// Klausur wird benotet (7 Teilnehmer)
punkte = new int[] { 86, 75, 53, 99, 24, 66, 89 };

// Initialisierung
int max_punkte = punkte[0];  // maximale Punktanzahl (1. Student)
int max_index = 0;           // Index des Studenten mit max_punkte
int sum = punkte[0];         // Summe aller Klausurpunkte

// Schleife zum Durchlauf vom zweiten (Index 1) bis letzten Studi
for (int i = 1; i < punkte.length; i++) {
    if (punkte[i] > max_punkte) {  // neues maximum gefunden
        max_punkte = punkte[i];    // abspeichern in max_punkte
        max_index = i;             // abspeichern des index
    }
    sum = sum + punkte[i];         // Summe anpassen
}

// Durchschnitt ist Summe durch Anzahl der Klausuren
float avg_punkte = (float) sum / (float) punkte.length;
```


Karel J. Robot IDE - Array.task
New
Open
Save
Save As
Undo
Redo
Execute
Submit
Settings
Exit
About
task {
// Neuen Array definieren
int[] punkte;
// Klausur wird benotet
punkte = new int[] { 86, 75, 53, 99, 24, 66, 89 };
// Auswertung min, max, avg
int max_punkte = punkte[0];
int max_index = 0;
int sum = punkte[0];
for (int i = 1; i < punkte.length; i++) {
if (punkte[i] > max_punkte) {
max_punkte = punkte[i];
max_index = i;
}
sum = sum + punkte[i];
}
float avg_punkte = (float) sum / (float) punkte.length;
System.out.println("Am besten war Student " + max_index
+ " mit " + max_punkte + " Punkten.");
System.out.println("Durchschnitt:" + avg_punkte);
}
Streets:
15
Avenues:
10
Delay [ms]:
1000
Pause
Stop
Clear
Line: 1, Column: 7
Preprocess program
Compile program
Reset world
Execute program
Am besten war Student 3 mit 99 Punkten.
Durchschnitt:70.28571
Ausgabe

Mehrdimensionale Arrays

- Arrays können auch mehr als eine Dimension haben.
  - das heißt im Prinzip, daß jeder Eintrag in der Liste wiederum eine Liste ist
  - effektiv wird daher aus der Liste eine Matrix (in 2-dimensionalen Fall, bzw. ein (Hyper-)Würfel im allgemeinen Fall

- Beispiel:




```
int[][] matrix;
matrix = new int[2][3];
matrix = new int[][] { {1,2,3},
                       {4,5,6} };
matrix[1][2] = 7;
```

Zusammenfassung Programm-Elemente

- Variablen-Deklarationen:
  - erklären, daß eine Variable einen bestimmten Typ hat (z.B. `int i;` oder `Robot karel;`)
  - eventuell mit automatischer Wertzuweisung (z.B. `int i=4;` oder
    `Robot karel=new Robot(1,2,3,East);`)
- Zuweisungen
  - weisen einer deklarierten Variable einen neuen Wert zu (z.B. `i = 5;` oder `i = i + 1;`)
  - der Wert kann natürlich auch der Wert eine andere Variable desselben Typs sein (z.B. `bolek = lolek;`)
- Berechnungen
  - alle Grundrechnungsarten (`+`, `-`, `*`, `/`) mit Klammerung sind möglich (z.B. e = m * (c * c);)
  - vorerst rechnen wir nur ohne Komma-Stellen.

Zusammenfassung Programm-Elemente (2)

- Bedingungen
  - Werte von Variablen können miteinander oder mit Konstanten verglichen werden (`==, !=, <, >, >=, <=`)
  - Boole'sche Operatoren: die Ergebnisse von Vergleichen können verneint (`!`), mit und verknüpft (`&&`) oder mit oder verknüpft (`||`) werden (z.B. `(1 <= i) && (i < 10)`)
- Konditionale
  - `if (<Bedingung>) { }`
  - `if (<Bedingung>) { } else { }`
- Iteration
  - `loop(<int>) { }`
  - `while (<Bedingung>) { }`
  - `for (<Init>; <Test>; <Update>) { }`
- Arrays

Zusammenfassung: Objekte

- Klassen
  - definieren Eigenschaften einer Klasse von Objekten
- Instanzen
  - eine Instanz ist ein bestimmtes Objekt einer Klasse
  - mit Instanzen kann man arbeiten, Klassen sind nur abstrakte Begriffe
- Vererbung
  - Klassen kann man definieren, indem man existierende Klassen erweitert
  - dabei werden die Methoden der existierenden Klasse (Super-Class) an die neue Klasse (Sub-Class) weitervererbt
  - können aber überschrieben bzw. ergänzt werden

Zusammenfassung: Methoden

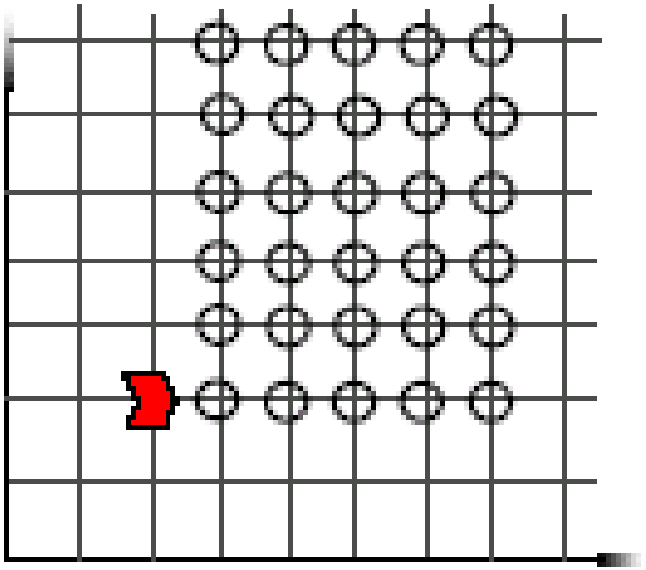
- Methoden (Nachrichten)
  - Operationen, die man auf einem Objekt durchführen kann
  - Methoden sind immer Klassen zugeordnet
- Definition von Methoden
  - für jede Klasse können Methoden definiert werden
  - geerbte Methoden können überschrieben werden
- Konstruktor
  - Methode, die eine neue Instanz des Objekts erzeugt
  - Definition mit Klassennamen, Aufruf mit `new`
- Polymorphie
  - eine Variable mit einem statischen Typ kann in einem Programm verschiedene dynamische Typen haben
  - daher kann derselbe Aufruf zum Aufruf verschiedener Methoden führen

Software Entwicklung

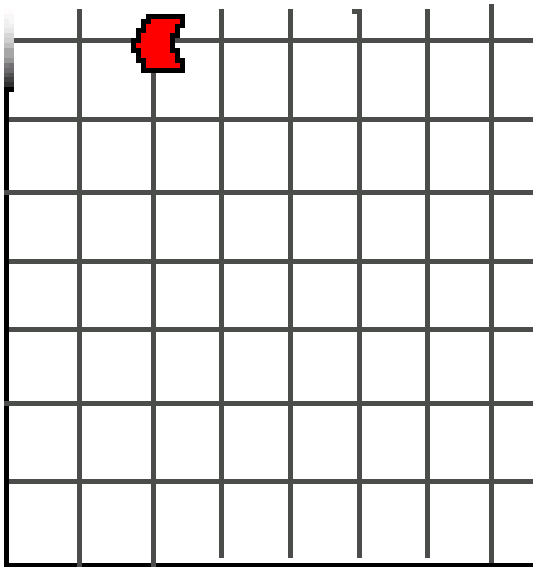
1. Analyse des Problems
2. Planung einer Lösung durch Verfeinerung
   - Wir beginnen mit einer Klassendefinition
   - und fügen sukzessive Methoden dazu
   - vorerst einmal ohne sie tatsächlich zu definieren
3. Implementierung
   - danach werden die Methoden nach und nach implementiert
4. Testen
   - ausprobieren des Programms und von Programm-Teilen in verschiedenen Szenarien
   - nach Möglichkeit so, daß alle Teile des Programms in irgendeiner Form durchlaufen werden

Problem: Ernte-Roboter

- Definieren Sie einen Roboter, der ein Feld von 6x5 Beepers aberntet



Anfangssituation



Endsituation

Planung der Klasse

- Klasse:
  - wir definieren einen ErnteRoboter, der alle Fähigkeiten eines Roboters erbt
  - und zusätzlich ein Feld ernten kann

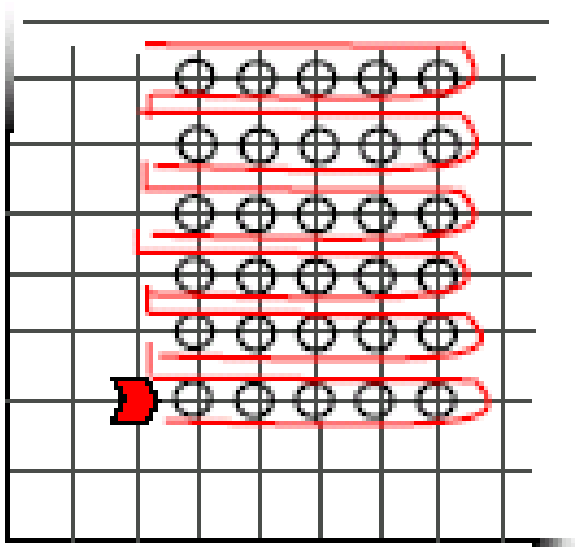
```
class ErnteRoboter extends Robot {

   void ernteFeld() {
     // Implementation folgt hier
   }
}
```

- geplante Verwendung:

```
task {
  ErnteRoboter karel =
          new ErnteRoboter(2,2,0,East);
  karel.ernteFeld();
}
```

Schrittweise Verfeinerung

- Kann ich die Methode `ernteFeld` in Teilprobleme zerlegen?
- z.B: Ernte eine Reihe, dann die nächste, dann die nächste...

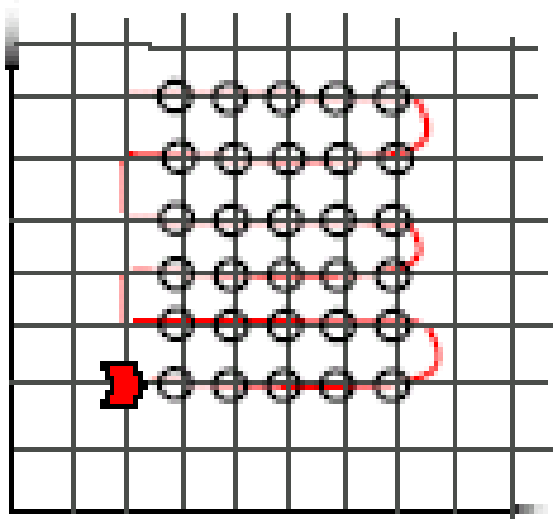


```
void ernteFeld() {
  ernteZeile();
  kehreZurueck();
  eineReiheHinauf();
  ...
  ...
  ernteZeile();
  kehreZureck();
}
```

- Die Methoden `ernteZeile`, `kehreZurueck`, `eineReiheHinauf` wären der nächste Schritt

Verbesserte Lösung

- Wäre es nicht effizienter, wenn der ErnteRoboter beim zurücklaufen die nächste Reihe erntet?



```
void ernteFeld() {
  ernteZeileNachOst();
  eineReiheHinaufOst();
  ernteZeileNachWest();
  eineReiheHinaufWest();
  ...
  ...
  ernteZeileNachOst();
  eineReiheHinaufOst();
  ernteZeileNachWest();
}
```

Weitere Verbesserungen

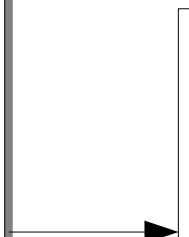
- Brauchen wir wirklich 2 Methoden um eine Zeile Richtung Westen und eine Zeile Richtung Osten zu ernten?
  - Nicht, wenn garantiert wird, daß der Ernte Roboter vor dem Auruf in die richtige Richtung positioniert wird.
  - Das muß die Methode `eineReiheHinauf` leisten.

```
void ernteFeld() {
  ernteZeile();
  eineReiheHinauf();
  ernteZeile();
  eineReiheHinauf();
  ...
  ...
  ernteZeile();
}
```

Weitere Verbesserungen

- Kann ich die Methode nun nicht kürzer schreiben?
  - Verwendung von `loop` bietet sich an
  - Aber Achtung!
    - `ernteZeile` wird 6 Mal verwendet, aber `eineReiheHinauf` nur 5 Mal!

```
void ernteFeld() {
  ernteZeile();
  eineReiheHinauf();
  ernteZeile();
  eineReiheHinauf();
  ...
  ...
  eineReiheHinauf();
  ernteZeile();
}
```



```
void ernteFeld() {
  loop(5) {
    ernteZeile();
    eineReiheHinauf();
  }
  ernteZeile();
}
```

Implementierung

- Das sieht gut aus.
- Implementieren wir nun die Methode `ernteZeile:`
  - wir müssen 5 Mal eine Kreuzung abernten

```
void ernteZeile() {
  loop(5) {
    ernteKreuzung();
  }
}
```

Implementierung (2)

- Nun müssen wir die Methode `ernteKreuzung` implementieren:
  - wir nehmen an, der ErnteRoboter steht (wie in der Startposition auf der ersten Kreuzung vor der Zeile, und blickt in die richtige Richtung
  - das heißt, wir müssen
    - einen Schritt weitergehen
    - und den Beeper dort aufheben

```
void ernteKreuzung() {
  move();
  pickBeeper();
}
```

Implementierung (3)

- Nun müssen wir noch `eineReiheHinauf` implementieren
  - 2 Fälle:
    - der Roboter hat gerade Richtung Osten geerntet
    - der Roboter hat gerade Richtung Westen geerntet

```
void eineReiheHinauf() {
   if (direction() == East)
      eineReiheHinaufOst();
   else
      eineReiheHinaufWest();
}
```

Implementierung (4)

- `eineReiheHinaufOst`
  - Beachte:
    - der Roboter steht auf der letzten Kreuzung, die er geerntet hat
    - er muß auf der ersten Kreuzung vor der nächsten Zeile stehen, damit `ernteZeile` funktioniert.
    - d.h. er muß einen Schritt weiter, nach links drehen, einen Schritt hinauf, und wieder nach links drehen

```
void eineReiheHinaufOst() {
  move();
  turnLeft();
  move();
  turnLeft();
}
```

Implementierung (5)

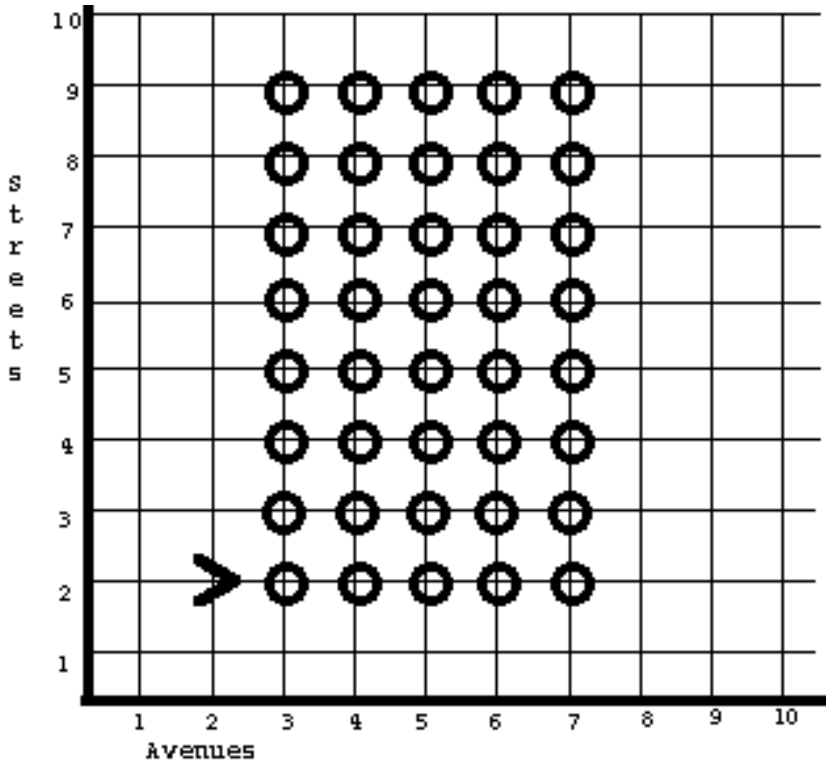
- `eineReiheHinaufWest`
  - analog zu `eineReiheHinaufOst`
  - allerdings muß der Roboter nun zwei Rechtsdrehungen machen
    - entweder `turnRight()` als eigene Methode implementieren
    - oder statt `Robot` eine Unterklasse von `RechtsDreher` bilden!
      - `class ErnteRoboter extends RechtsDreher`

```
void eineReiheHinaufWest()
{
  move();
  turnRight();
  move();
  turnRight();
}
```

```
class ErnteRoboter extends Robot {

  void turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
  }

  void ernteZeile() {
    loop(5) {
      ernteKreuzung();
    }
  }

  void ernteKreuzung() {
     move();
     pickBeeper();
  }

  void eineReiheHinauf() {
    if (direction() == East)
      eineReiheHinaufOst();
    else
      eineReiheHinaufWest();
  }

  void eineReiheHinaufOst() {
    move();
    turnLeft();
    move();
    turnLeft();
  }

  void eineReiheHinaufWest() {
    move();
    turnRight();
     move();
     turnRight();
  }

  void ernteFeld() {
    loop(5) {
      ernteZeile();
      eineReiheHinauf();
    }
    ernteZeile();
  }
}
```

Anpassung des Programms an neue Erfordernisse

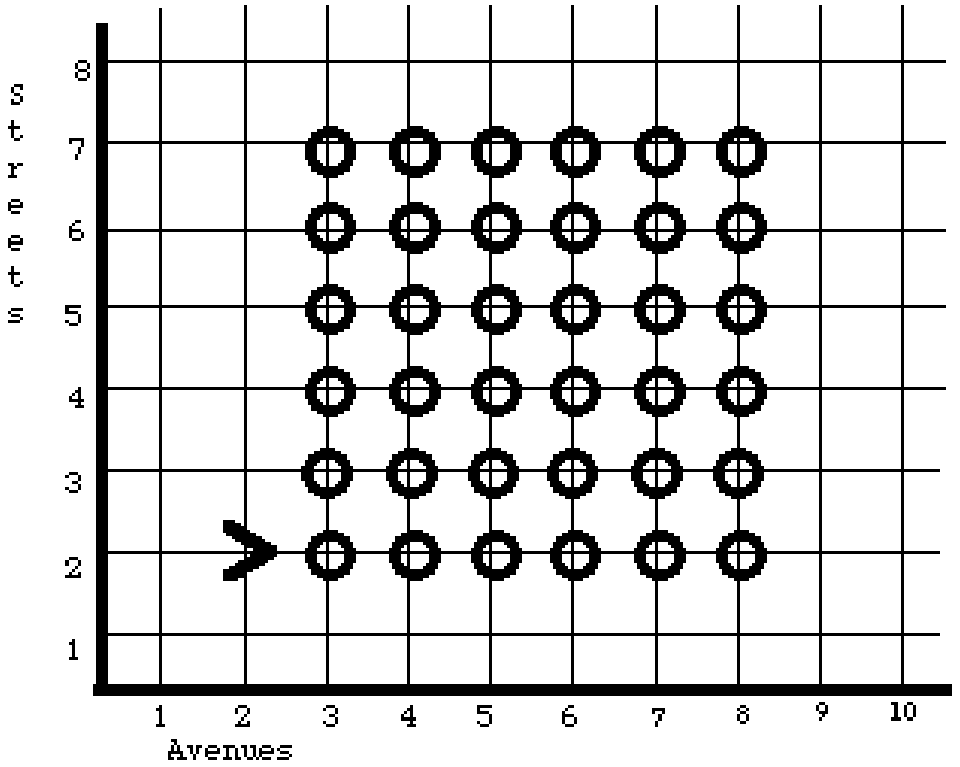
- zwei Zeilen mehr




```
class ErnteRoboter8Strts
      extends ErnteRoboter
{
  void ernteFeld() {

    // ernte 2 Zeilen
    ernteZeile();
    eineReiheHinauf();
    ernteZeile();
    eineReiheHinauf();

    // und dann das Feld
    // wie gehabt
    super.ernteFeld();
   }

}
```

Anpassung des Programms an neue Erfordernisse

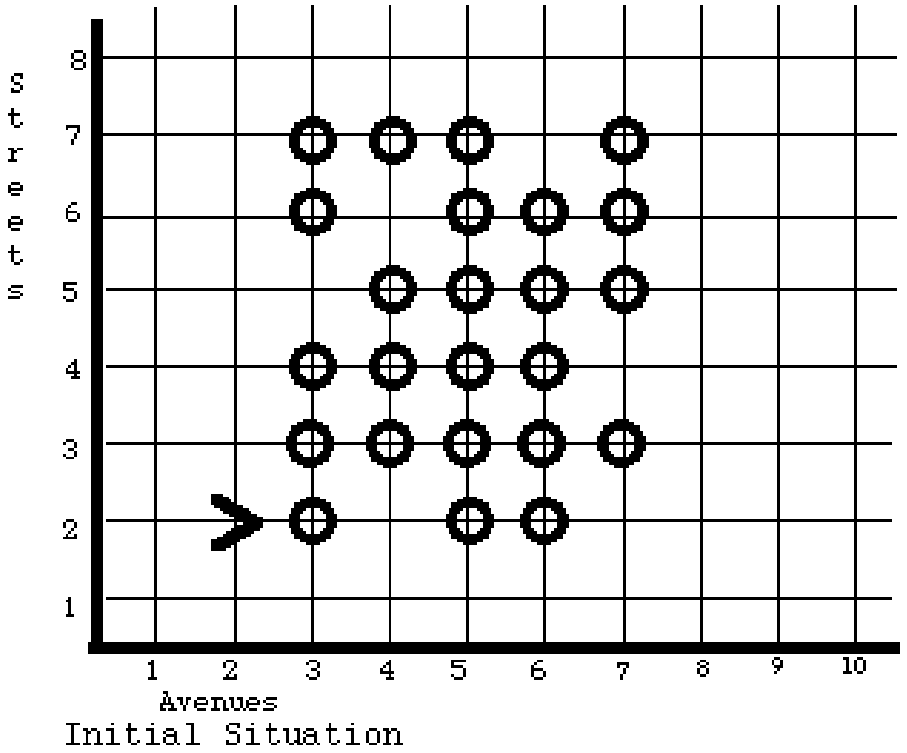
- längere Zeilen




```
class ErnteRoboter6Avs
      extends ErnteRoboter
{

  // wir müssen nur
  // ernteZeile umdefinieren!
  void ernteZeile() {

    // ernte eine alte Zeile
    super.ernteZeile();

    // und eine Kreuzung mehr
    ernteKreuzung();
  }

}
```

Anpassung des Programms an neue Erfordernisse

- nicht alle Kreuzungen sind besetzt



Initial Situation

```
class SensorErnteRoboter
      extends ErnteRoboter
{

  // wir müssen nur
  // ernteKreuzung umdefinieren!
  void ernteKreuzung() {

    move();

    // nur ernten wenn was da
    // ist
    if (nextToABeeper())
       pickBeeper();
  }

}
```

Vorteile eines stark strukturierten Programms

- größere Lesbarkeit
  - ein stark abstrahiertes Programm mit sprechenden Methoden-Namen braucht kaum mehr Dokumentation
- bessere Testbarkeit
  - man kann gezielt die einzelnen Methoden testen und sich von ihrer Funktionstüchtigkeit überzeugen
- Modularität
  - die einzelnen Teile (Methoden, Klassen) können in mehreren Programm-Teilen oder u.U. sogar in anderen Programmen wieder verwendet werden
- Adaptivität
  - Anpassung des Programms an neue Anforderung wird leichter, da oft nur wenige Stellen zu ändern sind
  - Bei Anpassung durch Subclassing bleibt die alte Version nach wie vor verfügbar