TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 01f:
Vererbung mit FopBot

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Erste eigene Roboterklasse: Roboter mit Rechtsdrehung

Das Thema Vererbung sowie seinen Sinn und Zweck schauen wir uns mit Beispielen dafür an, wie man eigene Roboterklassen definieren kann, die mehr können als Klasse Robot selbst. Dies ist das erste solche Beispiel.

Roboter mit Rechtsdrehung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import fopbot.*;

public class SymmTurner extends Robot {
    ..........
    ..........
    ..........
}
```

Die Datei ist SymmTurner.java.

SymmTurner im Einsatz

TECHNISCHE UNIVERSITÄT DARMSTADT

```
SymmTurner st = new SymmTurner ( 3, 4, UP, 20 );
st.turnLeft();
st.move();
st.turnRight();
```

Wir wollen die neue Klasse SymmTurner so realisieren, dass wir ganz normal eine Variable davon einrichten und so wie bisher initialisieren können: mit Spalte, Zeile, Richtung und Anzahl Münzen.

SymmTurner im Einsatz

TECHNISCHE UNIVERSITÄT DARMSTADT

```
SymmTurner st = new SymmTurner ( 3, 4, UP, 20 );
st.turnLeft();
st.move();
st.turnRight();
```

Außerdem wollen wir, dass Klasse SymmTurner alle Methoden hat, die auch Klasse Robot zur Verfügung stellt, und zwar in genau derselben Form und mit genau denselben Effekten.

SymmTurner im Einsatz

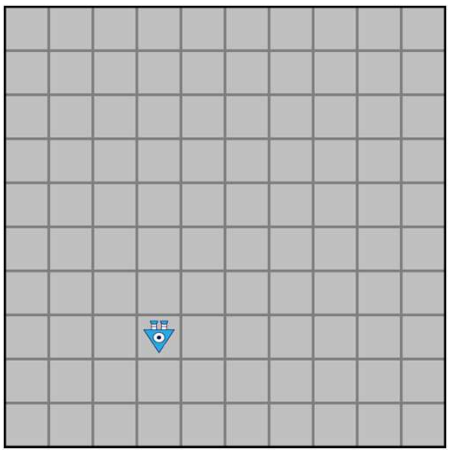
TECHNISCHE UNIVERSITÄT DARMSTADT

```
SymmTurner st = new SymmTurner ( 3, 4, UP, 20 );
st.turnLeft();
st.move();
st.turnRight();
```

Aber wir wollen auch, dass Klasse SymmTurner noch eine weitere Methode hat, die die Klasse Robot *nicht* hat; eine Methode, die wir in früheren Beispielen schon gut hätten gebrauchen können; eine Methode zum Rechtsdrehen, die völlig analog zu turnLeft sein soll, nur eben umgekehrte Richtung.

SymmTurner im Einsatz

TECHNISCHE UNIVERSITÄT DARMSTADT

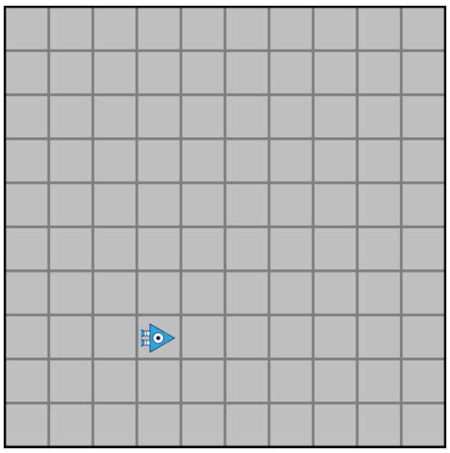


```
SymmTurner st = new SymmTurner ( 3, 2, DOWN, 20 );
```

Schauen wir uns diese vier Anweisungen von der letzten Folie nochmals einzeln im Bild an. Hier wird ein Roboter von Klasse SymmTurner eingerichtet. Alles soll so sein wie bei Klasse Robot, wenn dieselben vier Parameterwerte eingesetzt werden.

SymmTurner im Einsatz

TECHNISCHE UNIVERSITÄT DARMSTADT

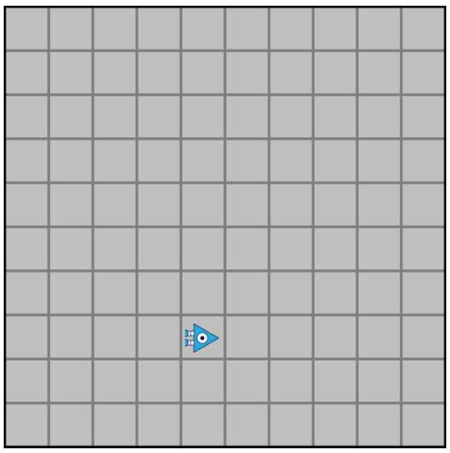


st.turnLeft();

Auch das Verhalten des Roboters bei Aufruf von turnLeft soll exakt dasselbe wie bei Klasse Robot sein.

SymmTurner im Einsatz

TECHNISCHE UNIVERSITÄT DARMSTADT

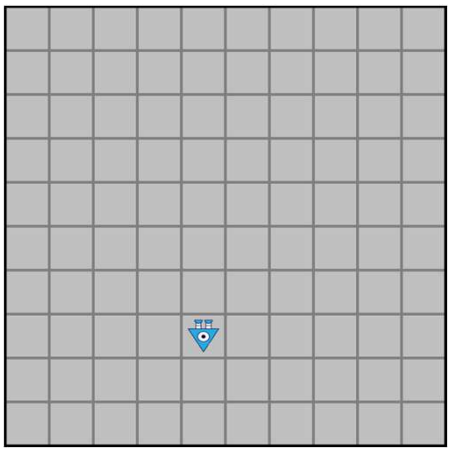


st.move();

Genauso bei Methode move.

SymmTurner im Einsatz

TECHNISCHE UNIVERSITÄT DARMSTADT

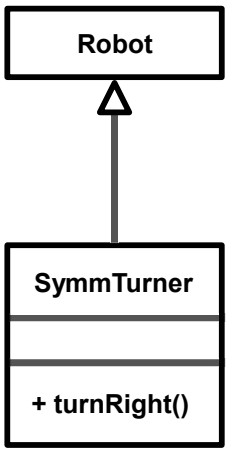


st.turnRight();

Die neue Methode turnRight, die wir für SymmTurner erst noch zu implementieren haben, soll das leisten, was man sich denken würde, nämlich analog zu turnLeft nach rechts statt nach links zu drehen.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT



Zuerst wieder das UML-Klassendiagramm, Erinnerung: verschiedene Stellen in Kapitel 01e.

Die Klasse Robot ist in Kurzform dargestellt, um die geht es jetzt nicht. Der Pfeil mit nicht gefüllter Spitze bedeutet, dass Klasse SymmTurner von Klasse Robot abgeleitet ist. Dies wiederum bedeutet, dass SymmTurner alle Attribute und Methoden von Robot *erbt*. Klasse SymmTurner hat keine zusätzlichen Attribute gegenüber Klasse Robot, daher ist die Attributsektion bei SymmTurner leer. Aber wie wir schon gesehen haben, soll SymmTurner eine zusätzliche Methode namens turnRight haben, die Robot noch nicht hat.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                 int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Das ist die *vollständige* Definition dieser Klasse SymmTurner. Wie man sieht, ist sie recht klein und überschaubar. Das liegt daran, dass sie ihre Funktionalität im Wesentlichen von Klasse Robot einfach übernimmt, auch das Zeichnen in die FopBot-World.

Wir sagen, SymmTurner *erbt* alle diese Funktionalität von Robot.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
    public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
        super ( x, y, direction, numberOfCoins );
    }
    public void turnRight() {
       for ( ... 3 ... ) turnLeft();
    }
}
```

Erinnerung **von Kapitel 01e: In einer Datei kann es mehrere Klassen, Enumerationen und ähnliches geben. Nur bei einer davon darf vorne das Schlüsselwort public stehen. Diese eine Klasse ist nach außen sichtbar, und die Datei muss denselben Namen wie *diese* Klasse haben.**

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Erinnerung **von Kapitel 01e: Das Schlüsselwort class zeigt an, dass wir hier eine neue Klasse definieren wollen und nicht beispielsweise eine Enumeration.**

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
    public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
        super ( x, y, direction, numberOfCoins );
    }
    public void turnRight() {
        for ( ... 3 ... ) turnLeft();
    }
}
```

Das ist jetzt neu: Mit Schlüsselwort extends und Klassennamen Robot sagen wir, dass die neue Klasse SymmTurner von der Klasse Robot abgeleitet werden soll. Wir sagen, dass Robot die *Basisklasse* von SymmTurner ist.

Dadurch erbt Klasse SymmTurner wie gewünscht alle Attribute und Methoden von Klasse Robot. Wir haben vier Attribute von Klasse Robot gesehen: Zeile, Spalte, Richtung und Anzahl Münzen. Diese vier Attribute sind durch private in der Klasse Robot versteckt, wir haben aber gesehen, dass sie sich durch Methoden der Klasse Robot abfragen lassen, nämlich getX, getY, getDirection und getNumberOfCoins. Diese Methoden haben wir am Beginn des Abschnitt beim beispielhaften Umgang mit SymmTurner nicht mitbetrachtet, aber auch sie werden selbstverständlich an SymmTurner vererbt.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
    public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
        super ( x, y, direction, numberOfCoins );
    }
    public void turnRight() {
        for ( ... 3 ... ) turnLeft();
    }
}
```

Genau so, wie Attribute und Methoden für Robot verfügbar sind, sind sie auch für SymmTurner verfügbar. Und auch die interne, für uns unsichtbare Verknüpfung mit der FopBot-World wird mitvererbt, so dass die Roboter unserer neuen Klassen wie gewünscht in der FopBot-World herumlaufen können.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
    public SymmTurner ( int x, int y, Direction direction,
                                int numberOfCoins ) {
        super ( x, y, direction, numberOfCoins );
    }
    public void turnRight() {
       for ( ... 3 ... ) turnLeft();
    }
}
```

Die ganze Klassendefinition ist wieder in geschweiften Klammern, das kennen wir ja schon von unserem Nachbau der Klasse Robot in Kapitel 01e. Wir wissen auch schon, dass – im Gegensatz etwa zu Schleifenrümpfen – die geschweiften Klammern bei einer Klassendefinition niemals weggelassen werden dürfen.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Neben den von Klasse Robot ererbten Methoden soll Klasse SymmTurner ja auch eine Methode turnRight haben, die Klasse Robot nicht hat. Hier implementieren wir diese Methode.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                 int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Die Methode turnRight hat zwar nur eine Anweisung, eine Schleife, aber wie wir schon wissen, müssen die geschweiften Klammern um einen Methodenrumpf auf jeden Fall immer sein.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
     for ( ... 3 ... ) turnLeft();
   }
}
```

Wie üblich: Dreimal links herum ist dasselbe wie einmal rechts herum.

Aus Platzgründen schreiben wir die ganze Schleife auf dieser Folie ohne weitere Formatierung in eine einzige Zeile, was man der Übersichtlichkeit halber eigentlich eher vermeidet.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Da Methode turnLeft von Robot an SymmTurner vererbt ist, steht diese Methode in Klasse SymmTurner zur Verfügung und kann verwendet werden, um Methode turnRight zu implementieren.

Wir müssen nicht wie sonst mit Punkt vorneweg dazuschreiben, welcher Roboter sich nach links drehen soll: Der Roboter, mit dem turnRight aufgerufen wird, ist automatisch der Roboter, mit dem in diesem Aufruf von turnRight dann dreimal turnLeft aufgerufen wird.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Erinnerung **von Kapitel 01e: Das Schlüsselwort public hier besagt, dass die Methode turnRight nach außen sichtbar und allgemein verwendbar sein soll.**

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Erinnerung **von Kapitel 01e: Das Schlüsselwort void zeigt an, dass diese Methode nichts zurückliefert.**

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                                 int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Hier würden wir die Parameter von Methode turnRight spezifizieren. Methode turnRight soll aber wie turnLeft keine Parameter haben. Daher lassen wir die Parameterliste leer. Aber das leere Klammerpaar müssen wir dennoch hinschreiben.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                       int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Erinnerung **von Kapitel 01e: Ein Konstruktor heißt immer genauso wie die Klasse, hier heißt er also SymmTurner. Und ein Konstruktor hat keinen Rückgabetyp, auch nicht void anstelle eines Rückgabetyps.**

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
  public SymmTurner ( int x, int y, Direction direction,
                      int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
  }
  public void turnRight() {
     for ( ... 3 ... ) turnLeft();
  }
}
```

Den Konstruktor brauchen wir public, denn wir wollen ihn ja außerhalb der Klasse SymmTurner verwenden.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
   public SymmTurner ( int x, int y, Direction direction,
                       int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
      for ( ... 3 ... ) turnLeft();
   }
}
```

Der Konstruktor von Klasse SymmTurner tut nichts anderes, als den Konstruktor der Basisklasse, also von Klasse Robot aufzurufen. Das ist auch logisch, denn abgesehen von der neuen Methode turnRight soll SymmTurner ja gar keine zusätzliche Funktionalität haben, also gibt es für SymmTurner auch nichts zu initialisieren. Aber für die Basisklasse gibt es sehr wohl etwas zu initialisieren, nämlich die vier Attribute.

Den Konstruktur der Basisklasse ruft man mit Schlüsselwort super auf, gefolgt von der Parameterliste für den Konstruktor der Basisklasse. Der Aufruf des Konstruktors der Basisklasse muss immer die erste Anweisung im Konstruktor der abgeleiteten Klasse sein.

Definition von SymmTurner

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SymmTurner extends Robot {
    public SymmTurner ( int x, int y, Direction direction,
                                int numberOfCoins ) {
        super ( x, y, direction, numberOfCoins );
    }
    public void turnRight() {
        for ( ... 3 ... ) turnLeft();
    }
}
```

Beim Einrichten von Robot-Objekten hatten wir immer diesen Konstruktor von Klasse Robot aufgerufen: erster Parameter die Spaltennummer, zweiter Parameter die Zeilennummer, dritter Parameter die Richtung und vierter Parameter die Anzahl Münzen.

Klasse SymmTurner bekommt diese vier Attribute von der Klasse Robot geschenkt. Aber dafür müssen wir diese Attribute hier dann eben auch initialisieren, wie wir das auch bei Robot-Objekten gemacht haben.

TECHNISCHE UNIVERSITÄT DARMSTADT

Zweite eigene Roboterklasse: Roboter in Slow Motion

Als nächstes definieren wir eine weitere beispielhafte Klasse.

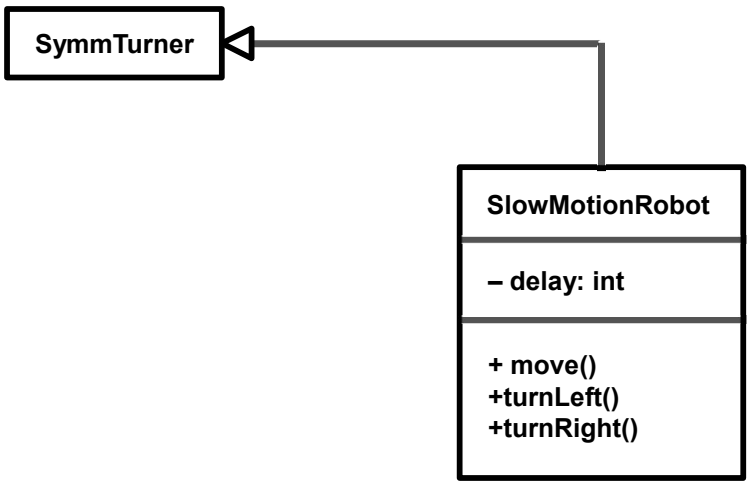
In diesem Beispiel geht es nicht darum, die Funktionalität von Klasse Robot zu erweitern wie bei SymmTurner, sondern es geht darum, das *Verhalten* der Methoden zu verändern.

Genauer gesagt, soll bei jedem Vorwärtsschritt und jeder Drehung eines SlowMotionRobot eine längere Zeitspanne vergehen, damit man die Aktionen des Roboters schrittweise besser nachvollziehen kann.

***Nebenbemerkung*: In FopBot können Sie das durch die Methode setDelay von World global für alle Roboter machen. Wir implementieren das jetzt so, dass jeder Roboter seine *individuelle* Zeitspanne haben kann.**

Robot in Slow Motion

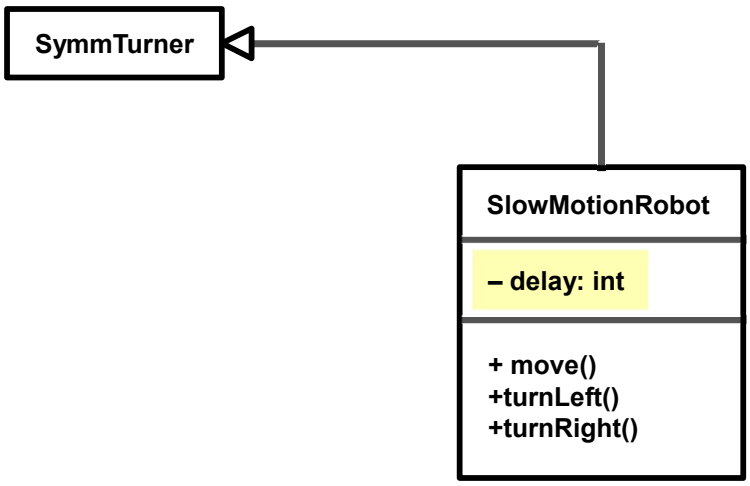
TECHNISCHE UNIVERSITÄT DARMSTADT



Auch hier zuerst das UML-Klassendiagramm. Die neue Klasse ist nicht von Robot, sondern von SymmTurner abgeleitet und erbt daher neben den Attributen und Methoden von Robot auch noch die Methode turnRight von SymmTurner.

Robot in Slow Motion

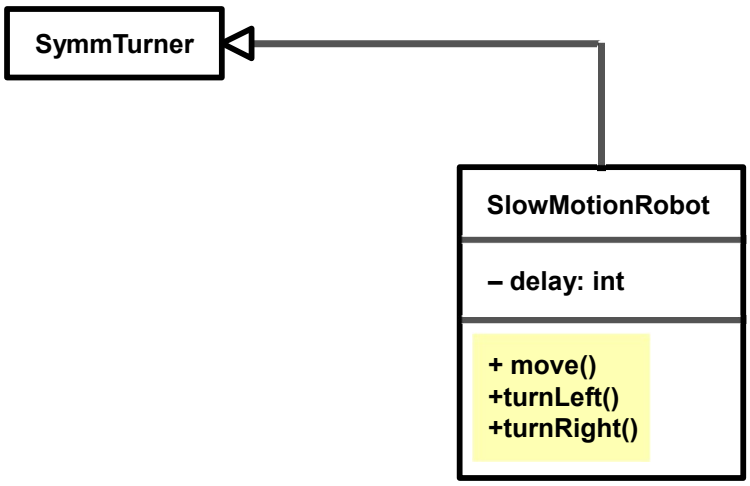
TECHNISCHE UNIVERSITÄT DARMSTADT



Die Klasse hat gegenüber ihrer Basisklasse nicht wie SymmTurner eine public-Methode zusätzlich, sondern ein private-Attribut namens delay.

Robot in Slow Motion

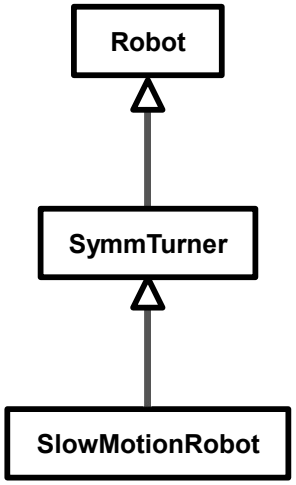
TECHNISCHE UNIVERSITÄT DARMSTADT



Auch wenn diese drei Methoden von Robot beziehungsweise SymmTurner ererbt sind, implementieren wir sie hier nochmals. Wir sagen, dass diese Methoden in SlowMotionRobot „überschrieben“ werden. Der Grund ist, dass diese drei Methoden sich in SlowmotionRobot etwas anders verhalten sollen als in Robot und SymmTurner, nämlich jeweils kurz pausieren nach der Operation.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT



So sieht dann das UML-Klassendiagramm für alle drei Klassen zusammen in Kurzform aus: SymmTurner ist von Robot abgeleitet, SlowMotionRobot von SymmTurner.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   public SlowMotionRobot ( int x, int y, Direction direction,
                            int numberOfCoins, int delay ) {
      super ( x, y, direction, numberOfCoins );
      this.delay = delay;
   }

   !!! ausfuellen !!!
}
```

Die Datei ist SlowMotionRobot.java.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   public SlowMotionRobot ( int x, int y, Direction direction,
                            int numberOfCoins, int delay ) {
      super ( x, y, direction, numberOfCoins );
      this.delay = delay;
   }

   !!! ausfuellen !!!
}
```

Die neue Klasse SlowMotionRobot leiten wir von SymmTurner ab. Sie erbt also alle Methoden wie Vorwärtsschritt und Linksdrehung von Klasse Robot sowie Rechtsdrehung von Klasse SymmTurner. Sie erbt natürlich auch die anderen Methoden von Robot sowie alle Attribute und das Zeichnen in die FopBot-Welt.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   public SlowMotionRobot ( int x, int y, Direction direction,
                            int numberOfCoins, int delay ) {
      super ( x, y, direction, numberOfCoins );
      this.delay = delay;
   }

   !!! ausfuellen !!!
}
```

Die vier Attribute aus Klasse Robot werden jetzt indirekt vererbt: direkt vererbt von Robot an SymmTurner und dann noch einmal direkt vererbt von SymmTurner an SlowMotionRobot. Der Konstruktor von SlowMotionRobot muss daher den Konstruktor seiner Basisklasse SymmTurner aufrufen, der – wie wir schon bei SymmTurner gesehen haben – seinerseits den Konstruktor von Robot aufruft. Letzterer nimmt dann schlussendlich die vier Parameter her und initialisiert damit die vier Attribute.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

  private int delay;

  public SlowMotionRobot ( int x, int y, Direction direction,
                           int numberOfCoins, int delay ) {
     super ( x, y, direction, numberOfCoins );
     this.delay = delay;
  }

  !!! ausfuellen !!!
}
```

Der Klasse SlowMotionRobot geben wir noch ein weiteres Attribut mit, das die direkte Basisklasse SymmTurner und die indirekte Basisklasse Robot noch nicht hatten. Darin soll die Verzögerungszeit für jeden Schritt in Millisekunden gespeichert werden.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   public SlowMotionRobot ( int x, int y, Direction direction,
                                    int numberOfCoins, int delay ) {
      super ( x, y, direction, numberOfCoins );
      this.delay = delay;
   }

   !!! ausfuellen !!!
}
```

Erinnerung **von Kapitel 01e: Mit this und einem Punkt vorneweg wird im Falle eines Namenskonflikts das Attribut angesprochen, ohne this der Parameter.**

Robot in Slow Motion

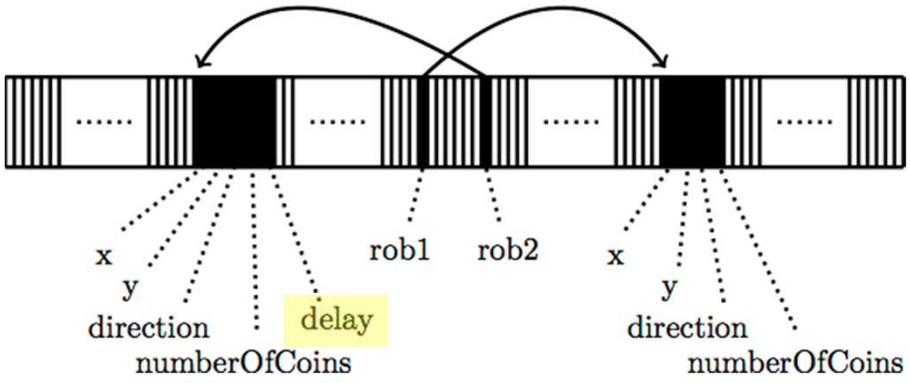
TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   public SlowMotionRobot ( int x, int y, Direction direction,
                                    int numberOfCoins, int delay ) {
      super ( x, y, direction, numberOfCoins );
      this.delay = delay;
   }

   !!! ausfuellen !!!
}
```

Dafür müssen die Methoden von Klasse Robot in Klasse SlowMotionRobot verändert werden. Wir sagen, diese Methoden von Klasse Robot werden in Klasse SlowMotionRobot *überschrieben*.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

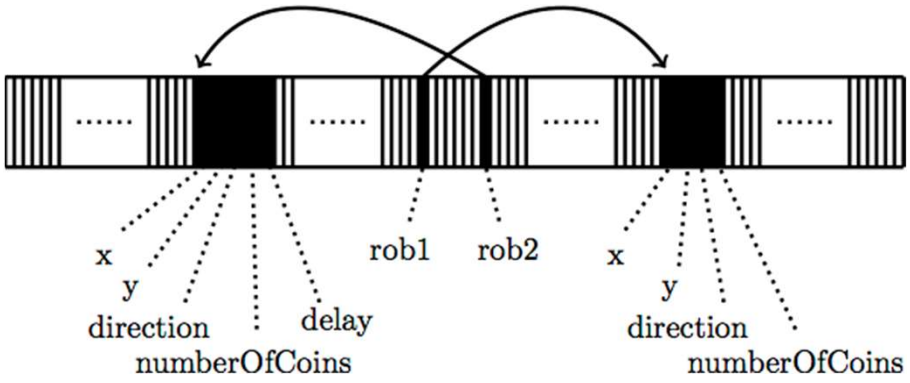



```
Robot rob1 = new Robot ( 1, 2 UP, 0 );
SlowMotionRobot rob2 = new SlowMotionRobot ( 3, 4, RIGHT, 3, 1000 );
```

Im Objekt der Klasse SlowMotionRobot findet sich dann auch das zusätzliche Attribut, das die Klasse Robot noch nicht hatte.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

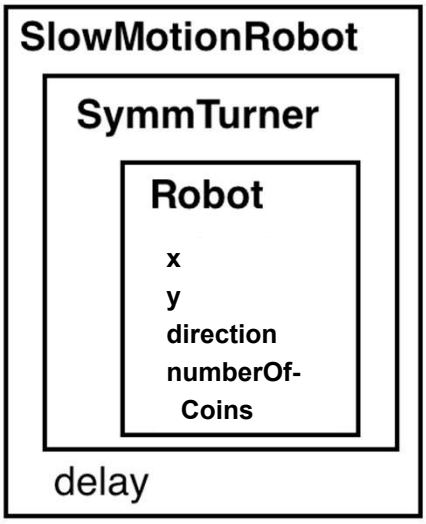



```
SymmTurner rob1 = new SymmTurner ( 1, 2 UP, 0 );
SlowMotionRobot rob2 = new SlowMotionRobot ( 3, 4, RIGHT, 3, 1000 );
```

Wenn wir in der ersten Zeile unten Robot durch SymmTurner ersetzen, dann ändert sich oben im Bild nichts, denn SymmTurner hat das Attribut delay ja auch noch nicht, nur die vier von Robot ererbten Attribute.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT



Dies ist eine gebräuchliche Form, sich das rein abstrakt vorstellen, also ohne Bezug dazu, wie das Ganze im Computerspeicher abgelegt wird: Ein Objekt von Klasse SlowMotionRobot enthält ein Objekt von Klasse SymmTurner, dieses wiederum ein Objekt von Klasse Robot. Klasse SymmTurner hat gegenüber Robot keine zusätzlichen Attribute, Klasse SlowMotionRobot hat hingegen gegenüber SymmTurner ein zusätzliches Attribut namens delay.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   .........

   public void move () {
       super.move();
       Thread.sleep ( delay );
   }


   .........
}
```

Nun machen wir uns wie angekündigt an das Überschreiben der Methode move.

Bei SymmTurner war das noch *so*: Ruft man move auf mit einem SymmTurner, dann wird die Methode move von Robot aufgerufen. Bei SlowMotionRobot hingegen wird die Methode move von SlowMotionRobot aufgerufen. Es wird immer die zuletzt in der Vererbungshierarchie definierte Methode aufgerufen, das ist hier die von SlowMotionRobot.

Vorgriff: Gleich in diesem Kapitel, im Abschnitt zu Vererben vs. Überschreiben, schauen wir uns die Systematik und die Hintergründe an. Bis dahin sammeln wir erst einmal Beispielmaterial.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   .........

   public void move () {
       super.move();
       Thread.sleep ( delay );
   }


   .........
}
```

Diese neue Methode move hält die Abarbeitung des Programms eine Zeitlang an. Diese Zeitspanne ist in delay gespeichert.

Thread ist eine vordefinierte Klasse, die nichts mit FopBot zu tun hat, sondern auch ohne das Paket FopBot allgemein in Java zur Verfügung steht. Die Methode sleep von Klasse Thread hält die Abarbeitung des Programm solange an, wie ihr Parameter es sagt. Maßeinheit ist Millisekunde.

Wir haben schon bei Klasse World gesehen, dass es auch Methoden gibt, die nicht mit dem Namen einer Variablen, sondern mit dem Namen der Klasse aufgerufen werden. Dort waren das die Methoden zur Errichtung von Wänden in der FopBot-World.

***Vorgriff*: Wie dort schon bemerkt, betrachten wir Methoden, die mit Klassen aufgerufen werden, systematisch im Kapitel 03b, Stichwort Klassenmethoden. Klasse Thread kommt dann im Kapitel 09.**

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   .........

   public void move () {
      super.move();
      Thread.sleep ( delay );
   }


   .........
}
```

Eigentlich soll ja Methode move von SlowMotionRobot dasselbe machen wie Methode move von Robot, nur eben mit der kurzen Zeitverzögerung hinterher.

Mit super und einem Punkt dazwischen wird die Methode move der Basisklasse aufgerufen. Die Basisklasse von SlowMotionRobot ist SymmTurner, und SymmTurner hat Methode move von Robot geerbt. Also wird mit super.move hier die Methode move von Robot aufgerufen.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   .........

   public void turnLeft () {
       super.turnLeft();
       Thread.sleep ( delay );
   }


   .........
}
```

Das Überschreiben von turnLeft ist völlig analog zum Überschreiben von move: erst Aufruf von turnLeft der Basisklasse, also indirekt turnLeft von Robot, dann wird die Abarbeitung des Programms um delay viele Millisekunden angehalten.

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   .........

   public void turnRight () {
      super.turnRight();
      Thread.sleep ( delay );
   }


   .........
}
```

Wenn wir bei turnRight analog vorgehen und einfach super.turnRight aufrufen, werden wir feststellen, dass die Pausierung beim Aufruf von turnRight viermal so lang ist, wie delay angibt. Das liegt an einem Phänomen, das wir uns ab der übernächsten Folie ansehen: In turnRight von SymmTurner wird ja dreimal turnLeft aufgerufen. Und wir werden gleich sehen, dass das dann wieder das turnLeft von SlowMotionRobot ist, also jeweils mit Pausierung. Klingt unlogisch? Gleich nicht mehr!

Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   .........

   public void turnRight () {
      super.turnLeft();
      super.turnLeft();
      super.turnLeft();
      Thread.sleep ( delay );
   }


   .........
}
```

Aber bevor wir uns dieses Phänomen genauer ansehen, hier noch schnell eingeschoben eine alternative Implementation, bei der die Pausierung offenbar wirklich nur so lang ist, wie delay angibt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Allgemein:
Methoden vererben / überschreiben

Wikipedia:
Tabelle virtueller Methoden / Virtual method table

Wir haben jetzt mit SymmTurner und SlowMotionRobot schon ausreichend Beispielmaterial gesammelt, um uns die Systematik dahinter anzuschauen. Dazu müssen wir die beiden auf dieser Folie einander gegenübergestellten Begriffe, vererben und überschreiben, sorgfältig unterscheiden.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Robot {
   .........
   public void move () {
      .........
   }
   .........
}
```

```
public class SymmTurner extends
Robot {
   .........   //  kein neues move
}
```

```
public class SlowMotionRobot
extends SymmTurner {
   .........
   public void move () {
      .........
   }
   .........
}
```

Sie sehen hier oben links noch einmal auszugsweise die Klasse Robot mit ihrer Methode move sowie unten links und oben rechts zwei Klassen, die wir in früheren Abschnitten dieses Kapitels von Klasse Robot abgeleitet hatten.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Robot {
   .........
   public void move () {
      .........
   }
   .........
}
```

```
public class SymmTurner extends
Robot {
   .........   // kein neues move
}
```

```
public class SlowMotionRobot
extends SymmTurner {
   .........
   public void move () {
      .........
   }
   .........
}
```

In Klasse SymmTurner hatten wir nur eine neue Methode turnRight eingefügt, ansonsten unterscheidet SymmTurner sich nicht von seiner Basisklasse Robot. Daher wird die Methode move von Robot an SymmTurner *vererbt*, das heißt, sie ist in SymmTurner einfach vorhanden und kann für einen SymmTurner genauso aufgerufen werden wie für einen Robot. Sie macht für SymmTurner auch dasselbe wie für Robot.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Robot {
   .........
   public void move () {
      .........
   }
   .........
}
```

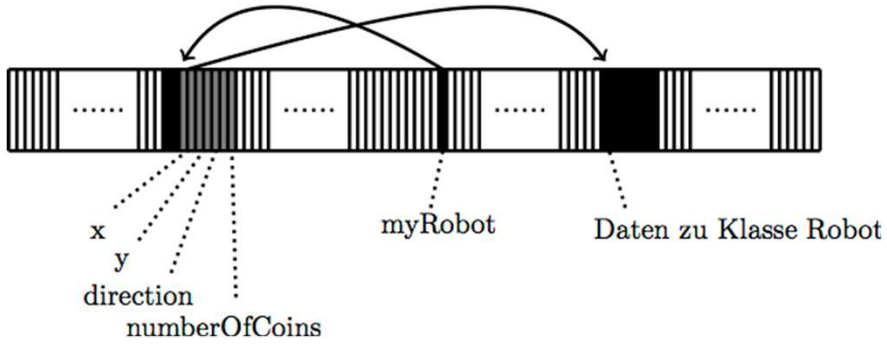
```
public class SlowMotionRobot
extends SymmTurner {
   .........
   public void move () {
      .........
   }
   .........
}
```

```
public class SymmTurner extends
Robot {
   .........   //  kein neues move
}
```

In Klasse SlowMotionRobot haben wir hingegen die Methode move *überschrieben*, das heißt, die Methode move mit derselben Parameterliste – in diesem Fall der leeren – noch einmal implementiert. Wenn Methode move mit einem SlowMotionRobot aufgerufen wird, dann wird die Implementation in SlowMotionRobot ausgeführt, nicht die in Robot.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

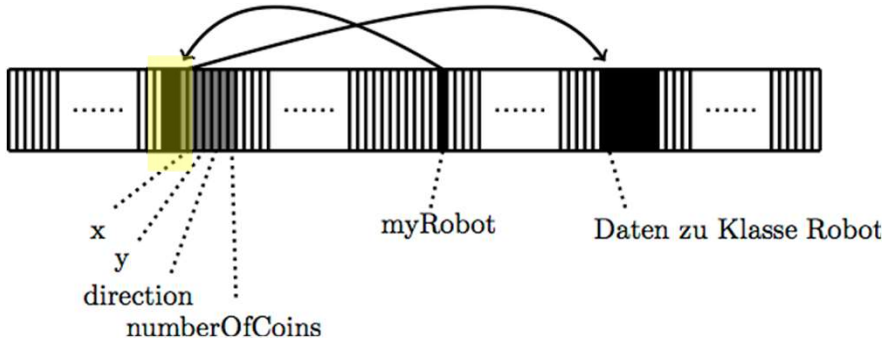


Robot myRobot = new Robot (.....);

Es ist wichtig für das tiefergehende Verständnis, sich wieder zu veranschaulichen, was im Speicher eigentlich vor sich geht. Das ist der Kernpunkt dieses Abschnitts.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

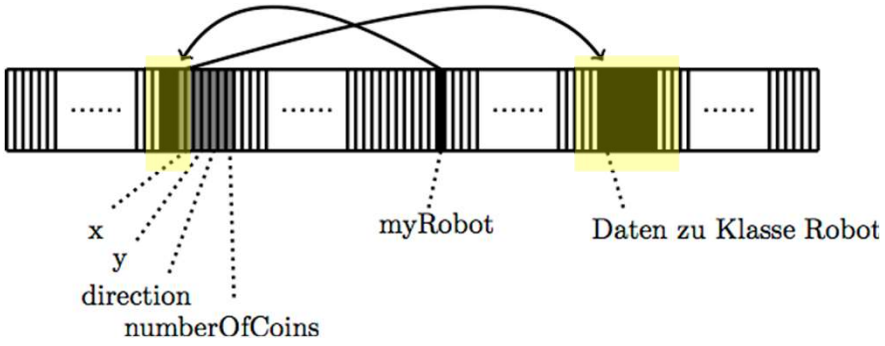


Robot myRobot = new Robot (.....);

Wir waren bisher etwas ungenau: In ein Objekt einer Klasse wird nicht nur Platz für die Attribute eingefügt, die wir der Klasse explizit mitgeben, sondern darüber hinaus noch ein kleiner Bereich, auf den wir nicht zugreifen können, der uns auch beim Programmieren nicht interessiert.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT



Robot myRobot = new Robot (.....);

Wenn der Compiler die Definition einer Klasse übersetzt, dann fügt er Anweisungen ein, die zu Beginn des Programms irgendwo im Speicher bestimmte Daten zu dieser Klasse anlegen. Bei Einrichtung eines Objektes dieser Klasse wird dann die Adresse dieses Datenblocks in den zusätzlichen Bereich geschrieben, so dass sich insgesamt das hier gezeigte Bild ergibt.

Vererben vs. Überschreiben

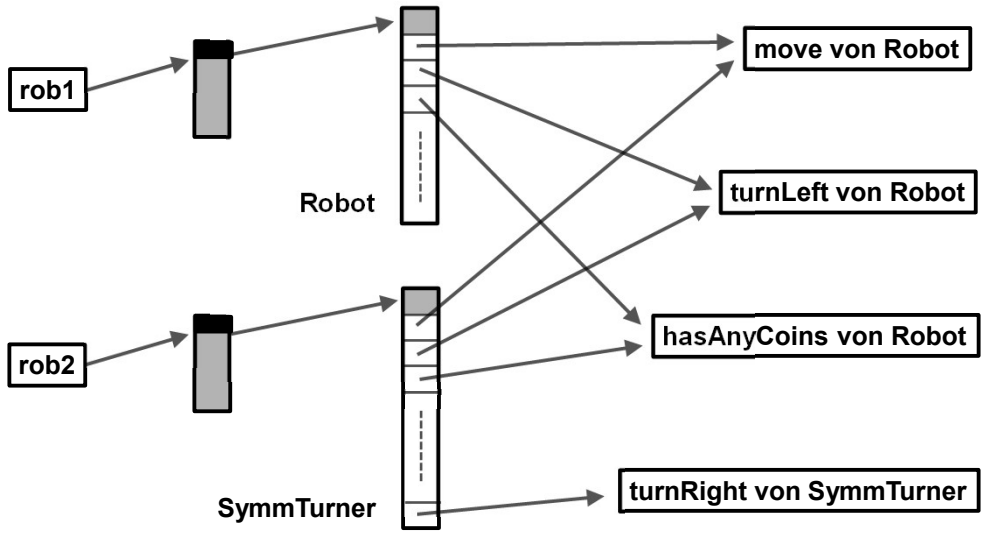
TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rob1 = new Robot ( ..... );

SymmTurner rob2 = new SymmTurner ( ..... );

SlowMotionRobot rob3
        = new SlowMotionRobot ( ..... );
```

Um genauer zu verstehen, was es damit auf sich hat, schauen wir uns beispielhaft an, was im Computerspeicher daraus gemacht wird, wenn diese drei Referenzen und Objekte eingerichtet werden. Die genauen Werte der einzelnen Attribute interessieren uns hier nicht, und daher sind die Parameter der drei Konstruktoraufrufe ausgelassen.

Vererben vs. Überschreiben

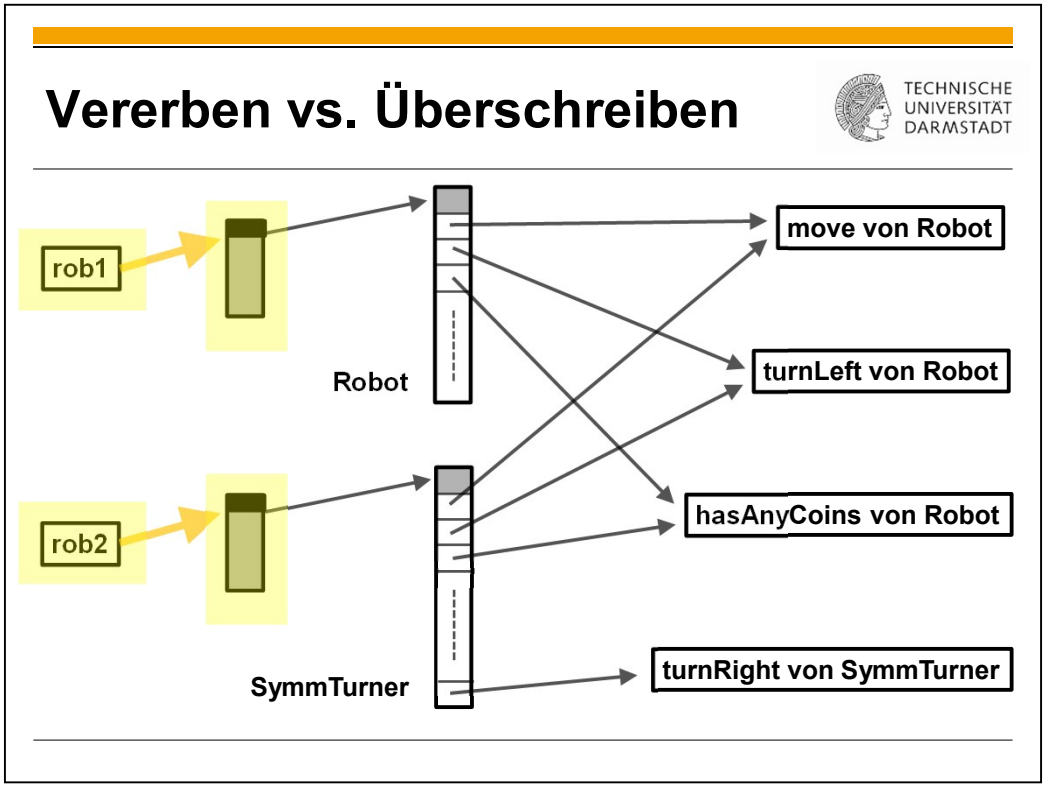
TECHNISCHE UNIVERSITÄT DARMSTADT



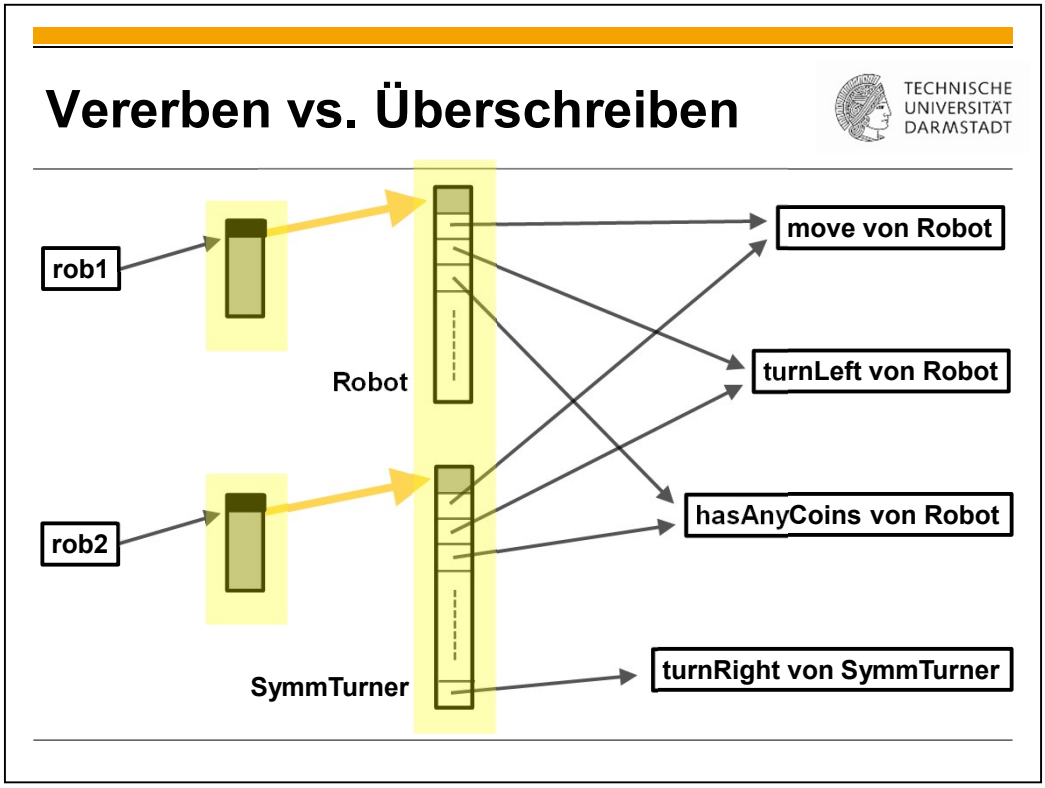
Aus Platzgründen vergleichen wir zuerst nur Robot und SymmTurner, der Vergleich von Robot und SlowMotionRobot kommt auf der nächsten Folie.

Jedes Rechteck auf dieser Folie steht für einen Bereich im Computerspeicher, und jeder Pfeil bedeutet, dass der Startpunkt des Pfeils die Adresse des Zielpunktes enthält. Also im Grunde alles wie bisher, nur dass wir uns jetzt zur besseren Übersicht von den starr linearen Bildern im Computerspeicher lösen.

Worum geht es hier: Wenn in Java eine Methode mit einer Referenz aufgerufen wird, dann passieren im Hintergrund so einige Dinge.

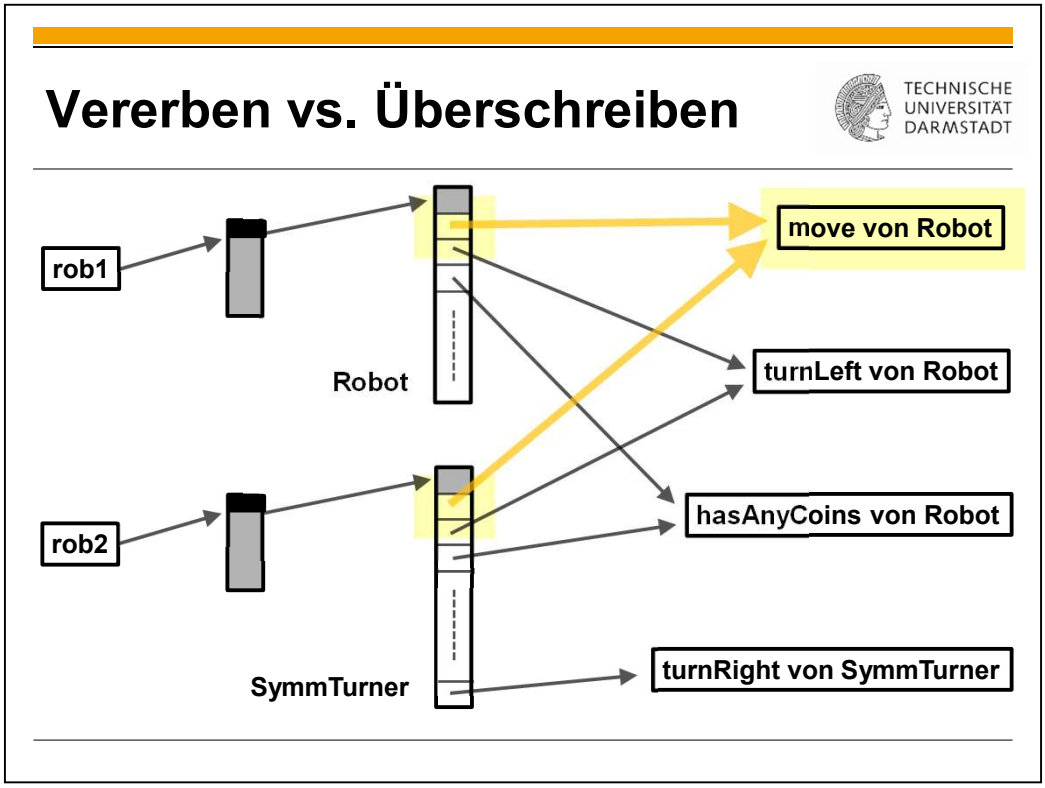


Diesen Teil haben wir schon öfters gesehen: Eine Referenz namens rob1 vom Typ Robot und eine Referenz namens rob2 vom Typ SymmTurner verweisen auf Objekte jeweils ihres eigenen Typs.



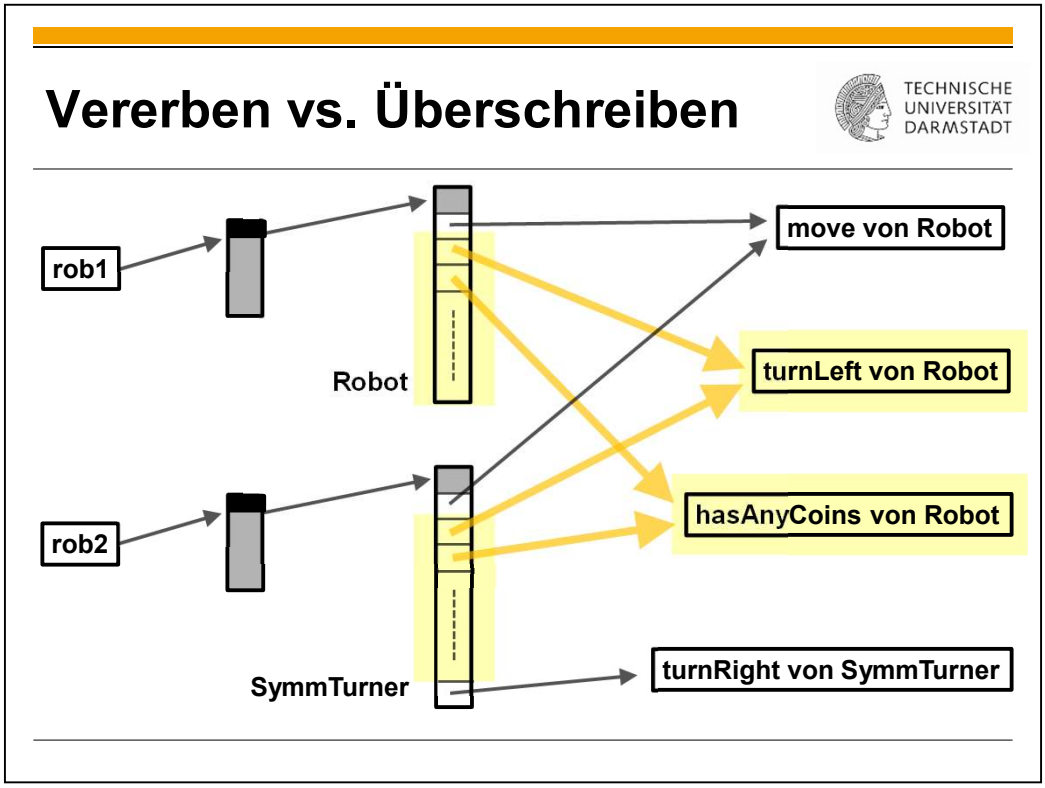
Das sind die Daten zu Klasse Robot beziehungsweise zu Klasse SymmTurner, wie wir es eben erstmals gesehen hatten. Wir hatten uns diese Daten aber noch nicht genauer angeschaut. Das holen wir jetzt nach.

Die grau unterlegten Felder in den Daten zu Robot beziehungsweise SymmTurner deuten an, dass es hier noch weitere Daten gibt, die uns bis auf weiteres nicht interessieren. Was uns jetzt definitiv interessiert, sind die weißen Felder, von denen die Pfeile ausgehen.



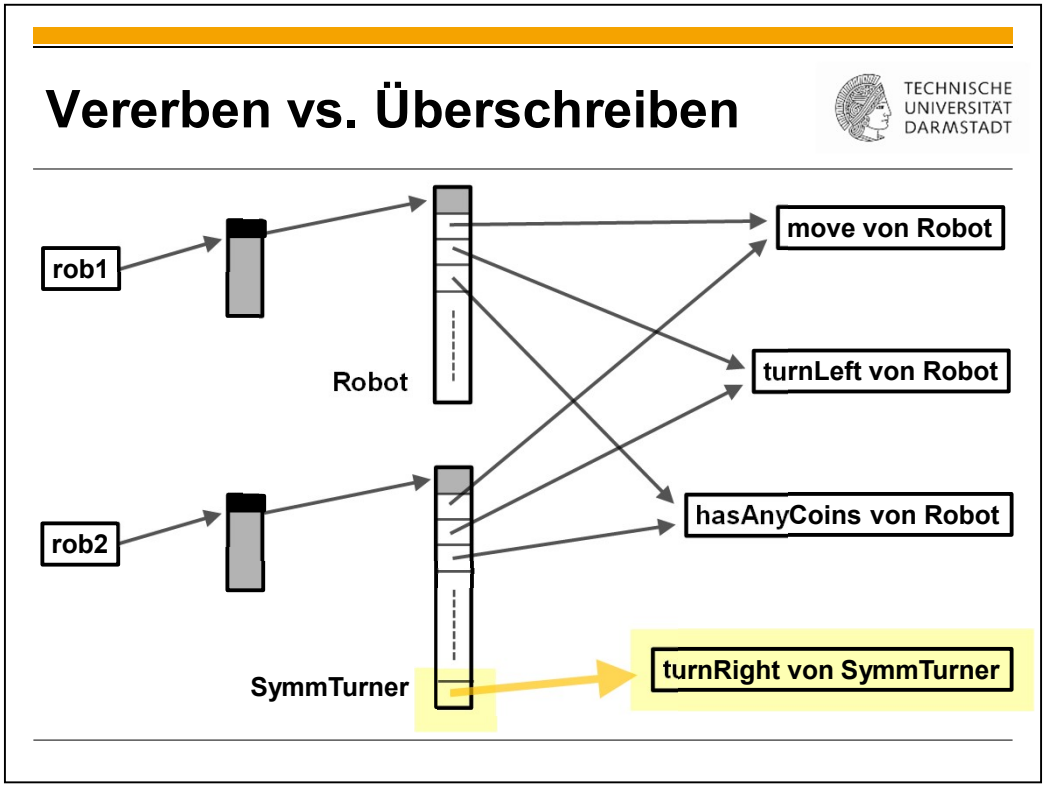
Jedes dieser weißen Felder speichert die Anfangsadresse der Übersetzung einer Methode. Daher nennt sich dieser Bereich die *Methodentabelle*, englisch *method table*. Der Offset einer Methode ist bei Robot und bei einer davon abgeleiteten Klasse wie SymmTurner derselbe. Das heißt, für den Aufruf einer Methode wie etwa move erstellt der Compiler Code, der zunächst einmal über die Referenz auf das Objekt und dann über das Objekt auf die Anfangsadresse der Methodentabelle zugreift. Weiterer Code addiert den festen Offset für die Position, an der der Verweis auf die Übersetzung von move abgelegt ist. Die Adresse, die dort gefunden wird, wird zum Aufruf der Methode move verwendet.

Sie sehen, was es konkret bedeutet, dass die Methode move von Robot an SymmTurner vererbt worden ist: Bei der Übersetzung von Klasse SymmTurner kopiert der Compiler einfach den Inhalt aus der Methodentabelle von Robot in die Methodentabelle von SymmTurner. Daher wird bei jedem Aufruf von move mit einem SymmTurner die Implementation von move aus Robot angesteuert.



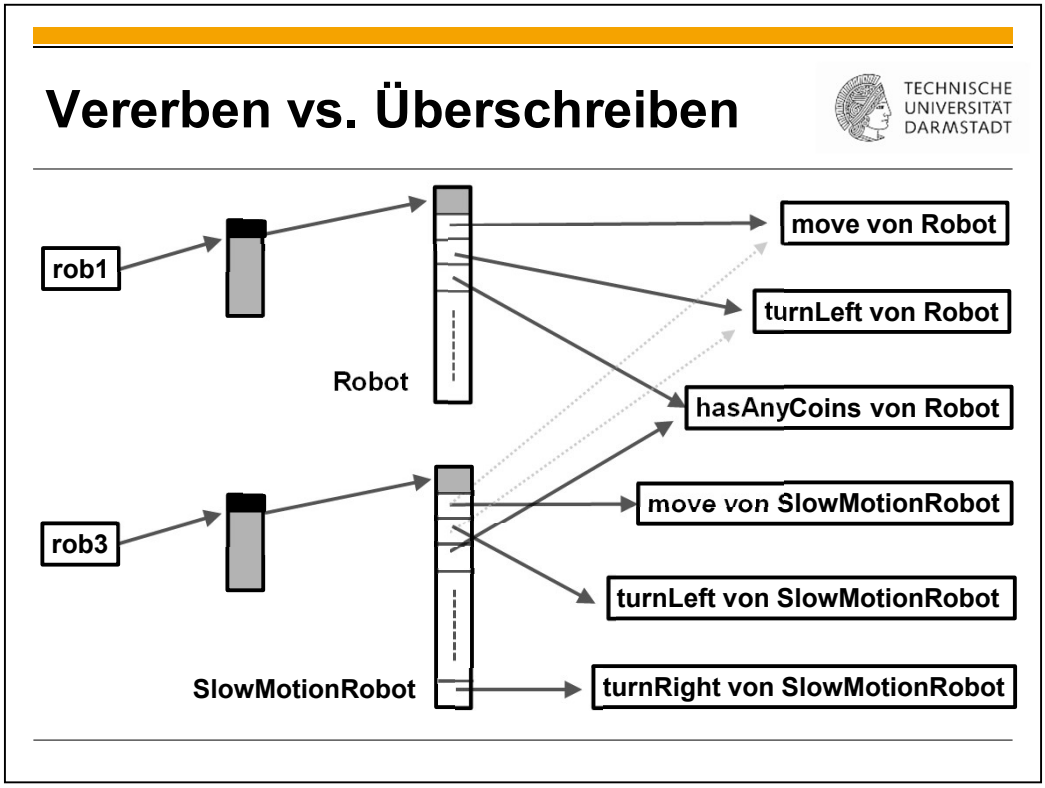
Für die anderen Methoden von Klasse Robot gilt analog dasselbe. Zwei weitere Methoden sind hier explizit dargestellt, die anderen durch Strichlinien angedeutet.

Der Offset für eine Methode ist in beiden Tabellen derselbe, so dass der Code zum Aufruf einer Methode unterschiedslos diesen Offset verwenden kann.

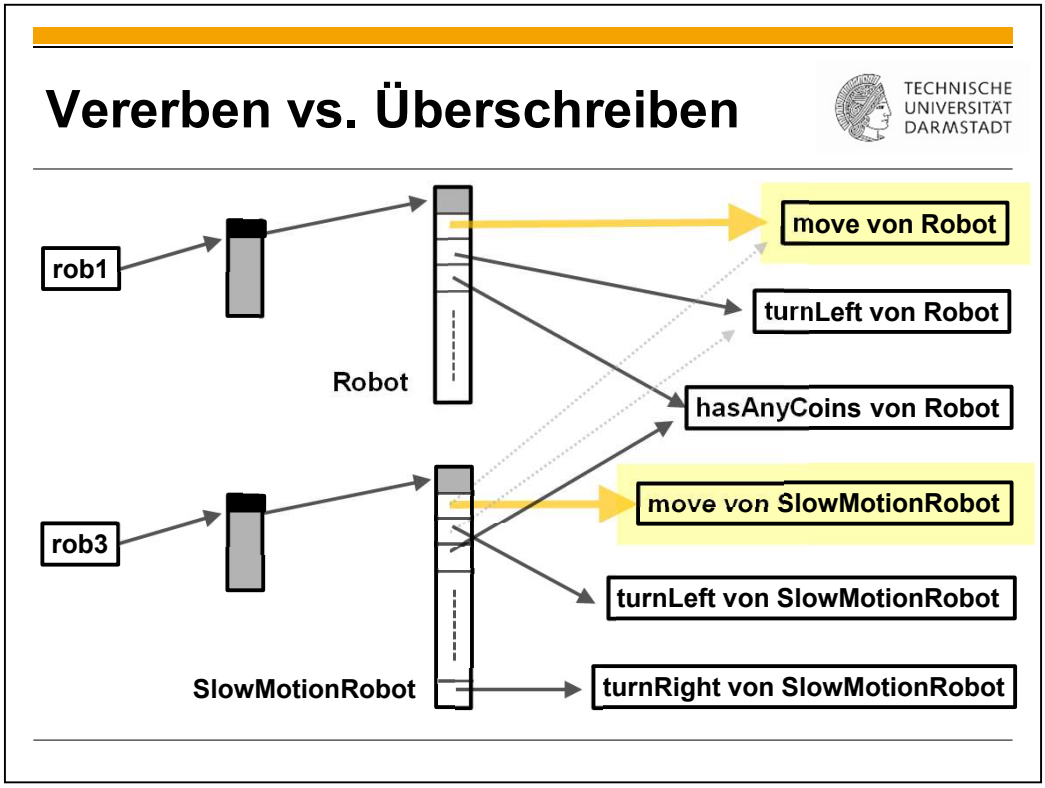


Die Methode turnRight war noch nicht für Klasse Robot definiert, erst für Klasse SymmTurner. Damit die Offsets der schon für Robot definierten Methoden bei SymmTurner und Robot gleich sein können, wird die neue Methode hinten an die Methodentabelle angehängt.

Damit sind wir mit dem Vergleich von Robot und SymmTurner fertig ...

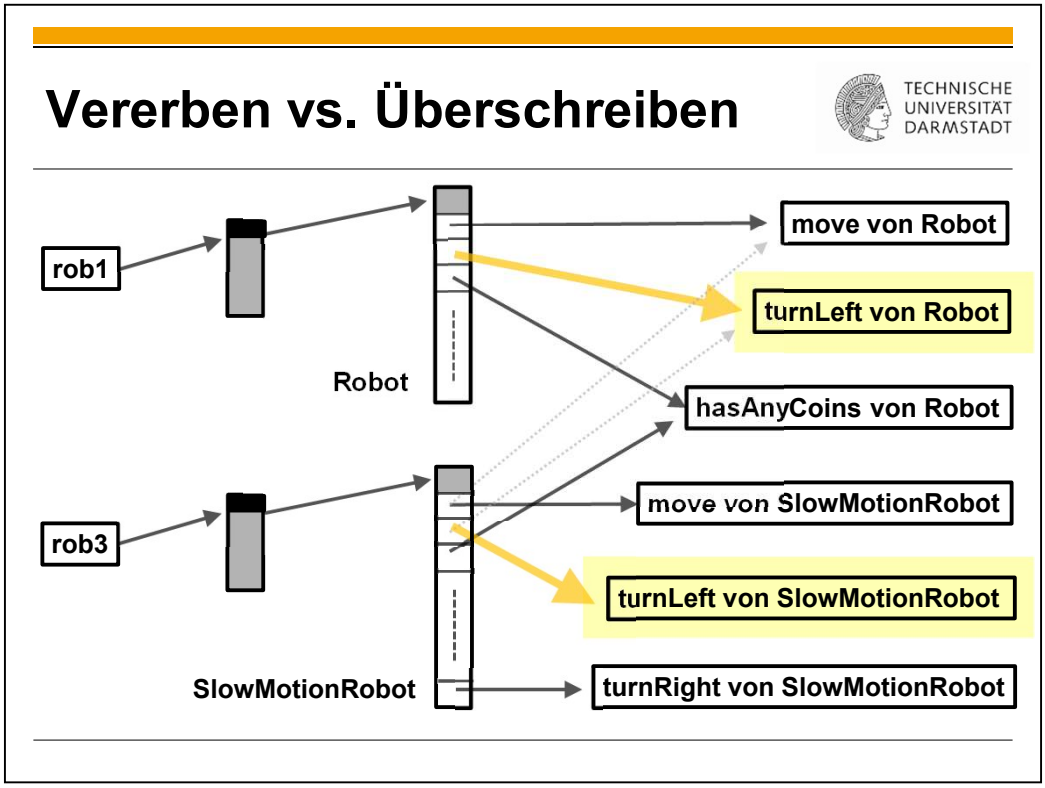


... und wechseln zum Vergleich von Robot und SlowMotionRobot, also rob3 statt rob2. Das Grundkonstrukt ist auch hier dasselbe. Aber da wir die Methoden move, turnLeft und turnRight für SlowMotionRobot überschrieben haben, sieht die Situation anders aus als soeben bei SymmTurner. Wie es bei SymmTurner aussah, zeigen die beiden ausgegrauten, gestrichelten Pfeile als kleine Erinnerungsstütze an.

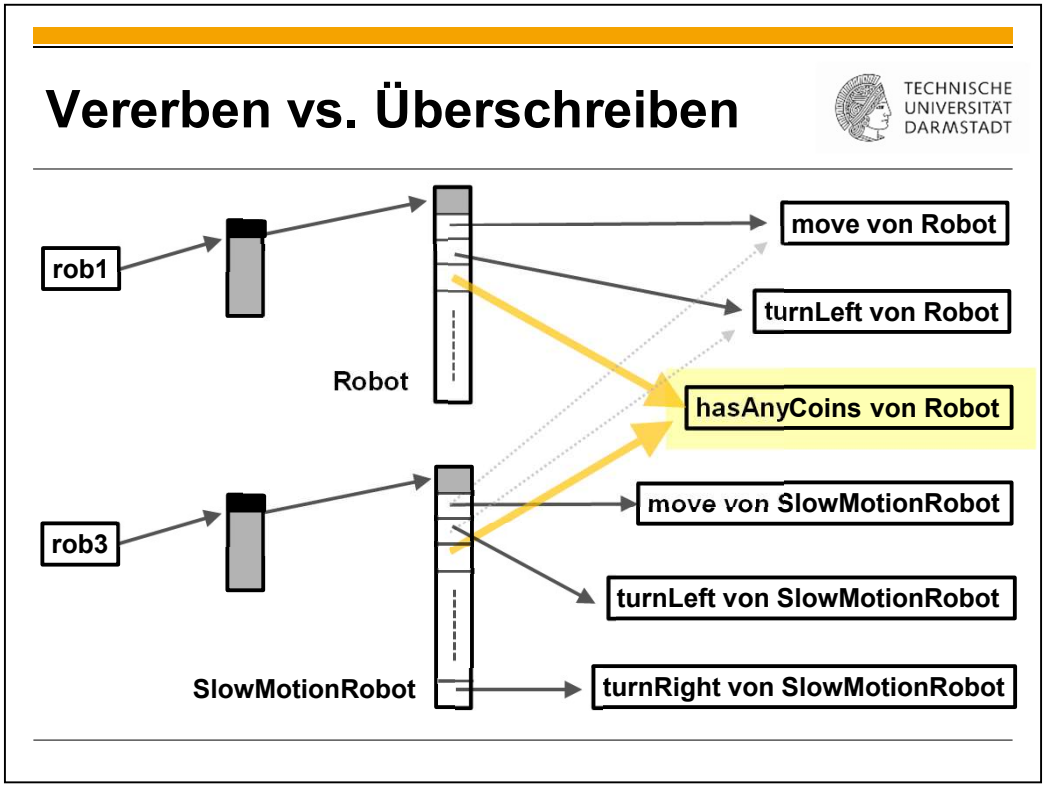


Konkret wird der Compiler bei der Übersetzung der Klasse SlowMotionRobot natürlich auch die Methode move von SlowMotionRobot übersetzen. In der Methodentabelle für SlowMotionRobot wird dann am Offset für move nicht die Adresse der Implementation aus Klasse Robot, sondern die Adresse der Implementation aus Klasse SlowMotionRobot stehen. Genau das war bei SymmTurner anders.

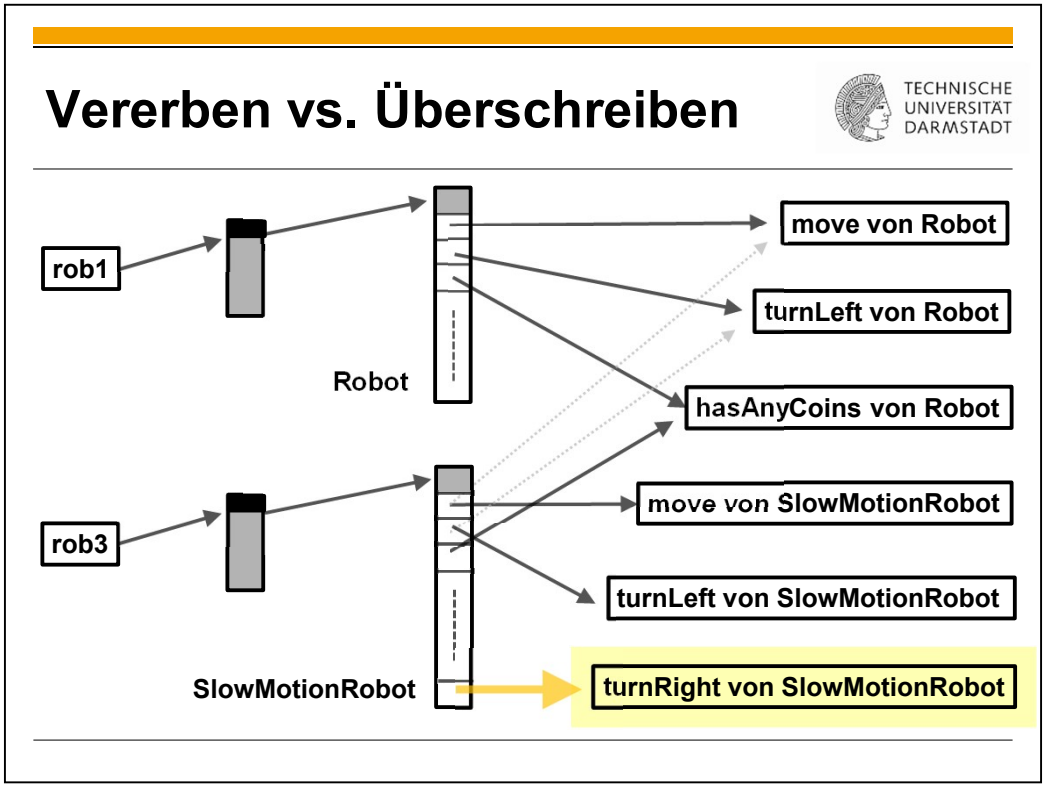
Sollte von SlowMotionRobot eine weitere Klasse abgeleitet werden, dann würde diese die Implementation von move in SlowMotionRobot erben, wenn move in dieser weiteren Klasse nicht überschrieben wird. Vererbt wird immer die letzte Implementation in der Hierarchie.



Dasselbe gilt auch für turnLeft, denn wir haben in SlowMotionRobot nicht nur move, sondern auch turnLeft überschrieben.



Methode hasAnyCoins ist nicht in SlowMotionRobot überschrieben worden. Daher sieht es hier wie vorher bei SymmTurner aus: Der Eintrag in der Methodentabelle von Robot ist vom Compiler zuerst bei der Übersetzung von SymmTurner in die Methodentabelle von SymmTurner und dann bei der Übersetzung von SlowMotionRobot von der Methodentabelle von SymmTurner in die Methodentabelle von SlowMotionRobot kopiert worden.



Die Methode turnRight wurde in SymmTurner eingeführt, also in der direkten Basisklasse von SlowMotionRobot. Wie bei den Methoden aus Robot hat auch diese Methode aus SymmTurner denselben Offset in SlowMotionRobot wie in SymmTurner. Da diese Methode in SlowMotionRobot überschrieben wurde, steht hier die Anfangsadresse der Implementation aus SlowMotionRobot, nicht die der Implementation aus SymmTurner.

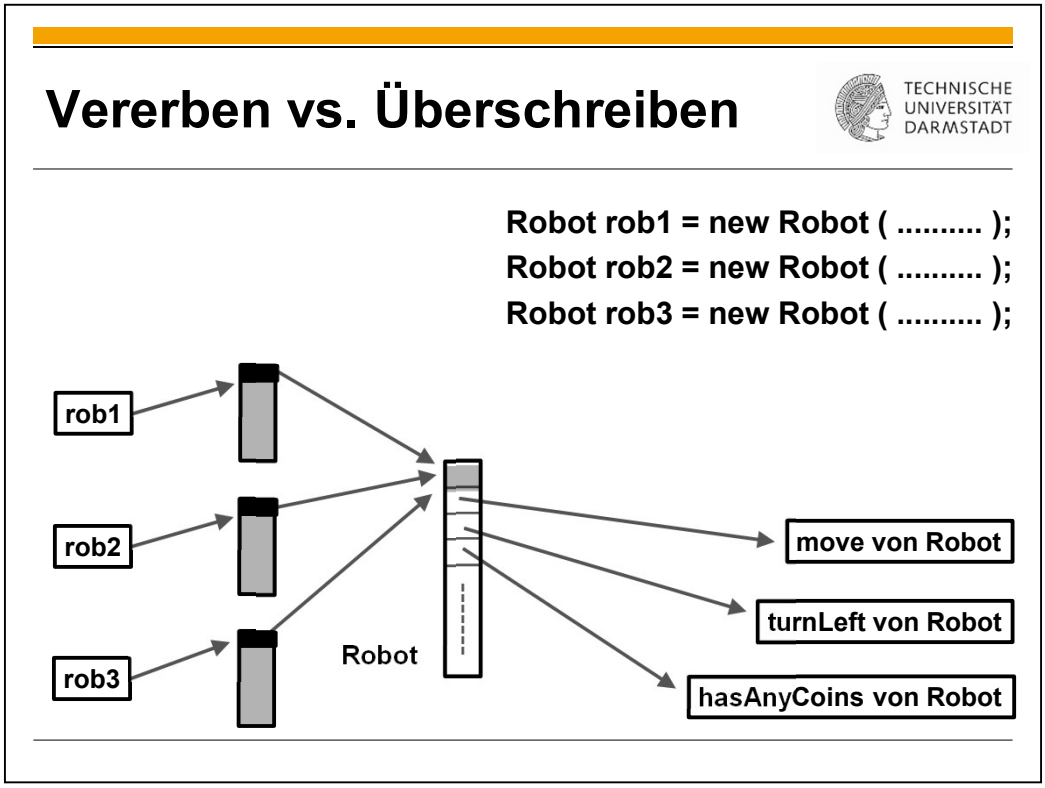
Robot in Slow Motion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class SlowMotionRobot extends SymmTurner {

   private int delay;

   .........

   public void turnRight () {
       super.turnRight();
       Thread.sleep ( delay );
   }


   .........
}
```

```
public class SymmTurner extends Robot {

   public SymmTurner ( int x, int y, Direction direction,
                                    int numberOfCoins ) {
       super ( x, y, direction, numberOfCoins );
   }
   public void turnRight() {
     for ( ... 3 ... ) turnLeft();
   }

}
```

Jetzt können wir das Phänomen bei Methode turnRight von SlowMotionRobot verstehen, das wir unmittelbar vor dem jetzigen Abschnitt gesehen haben: Durch super wird in Methode turnRight von Slowmotion die Methode turnRight von SymmTurner aufgerufen. Darin wiederum wird turnLeft dreimal aufgerufen. Welche Implementation von turnLeft wird da aufgerufen? Nun, ein Objekt von Klasse SlowMotionRobot verweist auf die Methodentabelle von SlowMotionRobot. Also wird in diesem Fall dreimal turnLeft von SlowMotionRobot aufgerufen und somit viermal statt nur einmal pausiert.

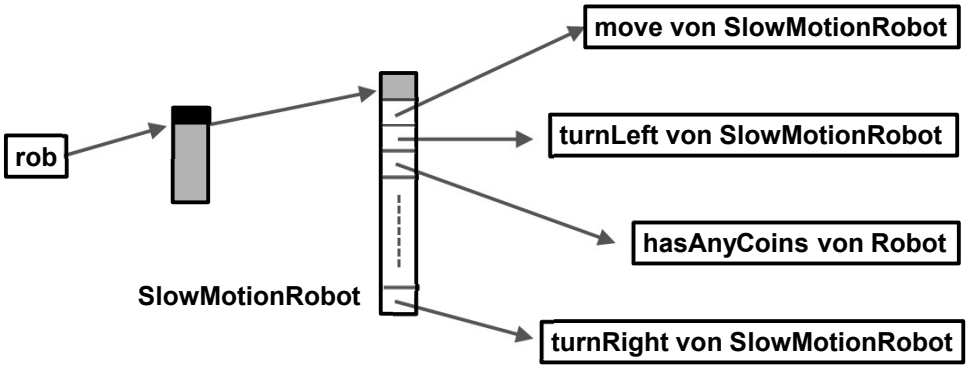


Ein wichtiger Punkt bei den Methodentabellen ist bei den bisherigen Beispielen vielleicht noch nicht klar herausgekommen: Es wird nicht für jedes Objekt einer Klasse eine eigene Methodentabelle eingerichtet, sondern nur eine einzige Methodentabelle für alle Objekte dieser Klasse. Immer wenn ein Objekt einer Klasse eingerichtet wird, wird die Anfangsadresse der Daten zu dieser Klasse, zu denen auch die Methodentabelle gehört, in das entsprechende Feld im neuen Objekt kopiert.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rob = new SlowMotionRobot ( 1, 2, UP, 5, 300 );
```

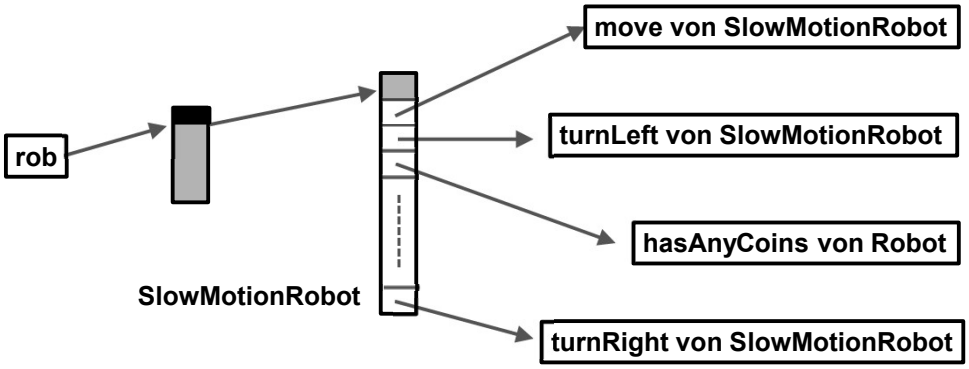


Nun noch ein Vorgriff auf eine Möglichkeit, die später sehr wichtig werden wird. Diese Möglichkeit ist letztendlich auch der Grund, warum bei Klassen überhaupt zwischen Referenz und Objekt unterschieden wird.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rob = new SlowMotionRobot ( 1, 2, UP, 5, 300 );
```

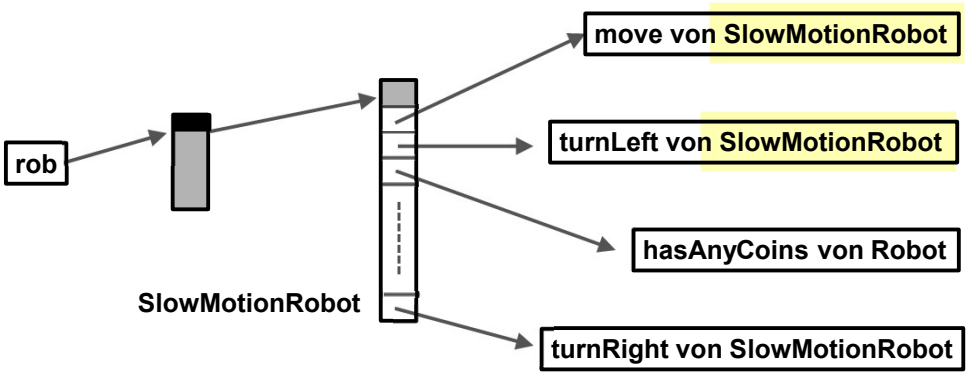


Bisher stand bei uns an den beiden farblich unterlegten Stellen immer derselbe Name einer Klasse. Jetzt stehen zum ersten Mal zwei verschiedene Namen hier. Tatsächlich kann man einer Referenz nicht nur ein Objekt derselben Klasse zuweisen, sondern alternativ auch ein Objekt einer direkt oder indirekt abgeleiteten Klasse. In diesem Beispiel ist es eine indirekt abgeleitete Klasse, denn SlowMotionRobot ist direkt von SymmTurner und SymmTurner direkt von Robot abgeleitet.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rob = new SlowMotionRobot ( 1, 2, UP, 5, 300 );
```

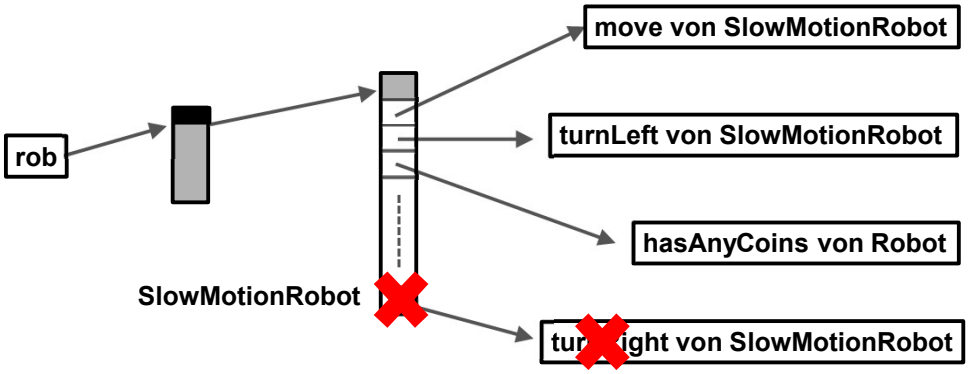


Die Methodentabelle hängt nicht vom Typ der Referenz, sondern vom Typ des Objektes ab. Wenn also die Methoden move und turnLeft mit rob aufgerufen werden, dann werden die Implementationen von move und turnLeft für SlowMotionRobot angesteuert. Hinter der Fassade eines Robot verbirgt sich also tatsächlich ein echter SlowMotionRobot.

Vererben vs. Überschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rob = new SlowMotionRobot ( 1, 2, UP, 5, 300 );
```



Allerdings können Methoden, die nicht in der Klasse, zu der die Referenz gehört, definiert sind, über diese Referenz auch nicht aufgerufen werden. Konkret in diesem Beispiel ist Methode turnRight nicht für Klasse Robot definiert, also darf turnRight nicht durch die Referenz rob aufgerufen werden, obwohl sie vorhanden ist.

***Vorgriff*: In Kapitel 03b, Abschnitt „Begriffsbildung: Subtypen und statischer / dynamischer Typ“ werden wir die genauen Regeln und die Gründe dafür betrachten.**

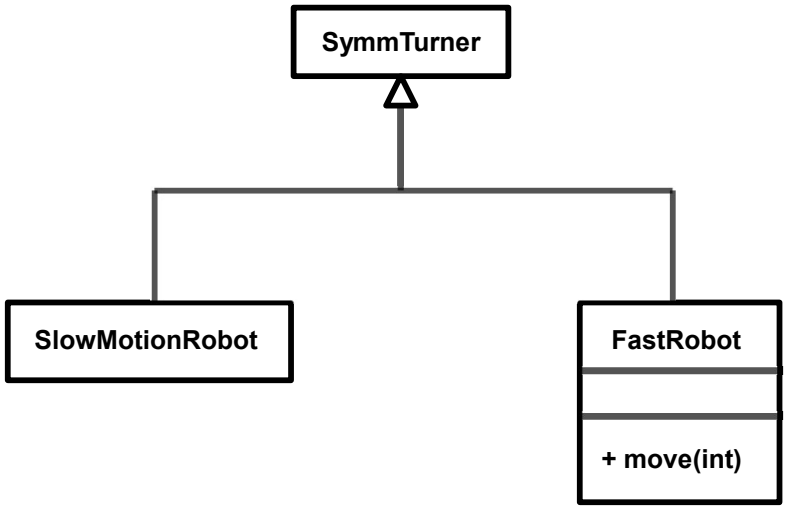
TECHNISCHE UNIVERSITÄT DARMSTADT

Dritte eigene Roboterklasse: zusätzliche move-Methode

Als nächstes eine Klasse namens FastRobot mit einer zweiten move-Methode, die mehrere Schritte auf einmal machen kann, abhängig von ihrem Parameter. Diese Klasse wird als Beispielmaterial dienen für den gleich folgenden Abschnitt zu Überschreiben vs. Überladen.

Êine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

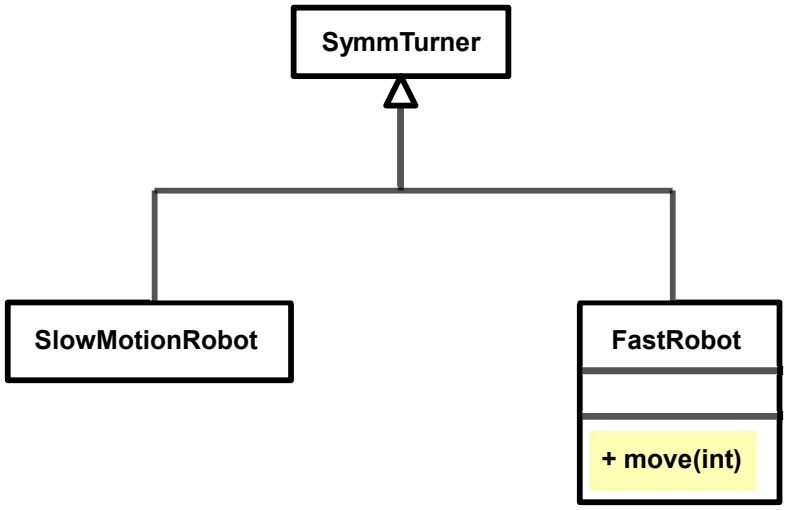


Wir werden FastRobot ebenfalls von SymmTurner ableiten, so dass nun zwei Klassen direkt von SymmTurner abgeleitet sind. In UML-Klassendiagrammen sieht das dann so wie hier gezeigt aus: Zwei Pfeile vereinigen sich zu einem.

Da es jetzt nicht um SymmTurner und SlowMotionRobot geht, sind beide in Kurzform dargestellt.

Êine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT



Wie gesagt, wird die neue Klasse FastRobot eine zweite Methode move haben, die im Gegensatz zu move von Robot noch einen Parameter hat. Dieser Parameter ist vom Typ int.

Eine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class FastRobot extends SymmTurner {

   public FastRobot ( int x, int y, Direction direction,
                      int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }

   public void move ( int numberOfSteps ) {
      for ( ... numberOfSteps ... )
         move();
   }
}
```

Das ist die vollständige Definition der Klasse FastRobot.

Wir sehen in der ersten Zeile, dass FastRobot von SymmTurner abgeleitet ist. Dadurch hat FastRobot die Methode move, wie wir sie bisher kennen gelernt haben, und natürlich auch die anderen Methoden von Robot sowie die Rechtsdrehung von SymmTurner.

Eine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class FastRobot extends SymmTurner {

   public FastRobot ( int x, int y, Direction direction,
                      int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }

   public void move ( int numberOfSteps ) {
      for ( ... numberOfSteps ... )
         move();
   }
}
```

Kopf der Klasse und Konstruktor sehen analog wieder genau so aus, wie wir es schon bei SymmTurner gesehen haben. Das brauchen wir nicht nochmals im Detail durchzugehen.

Eine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class FastRobot extends SymmTurner {

   public FastRobot ( int x, int y, Direction direction,
                      int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }

   public void move ( int numberOfSteps ) {
      for ( ... numberOfSteps ... )
         move();
   }
}
```

Hier definieren wir jetzt eine zweite Methode move. Sie überschreibt nicht die Methode move von Robot, denn sie hat eine andere Parameterliste. Klasse FastRobot hat daher *zwei* Methoden, die denselben Namen move tragen, sich aber in der Parameterliste unterscheiden.

Die Namensgleichheit hat für uns als menschliche Leser natürlich eine Bedeutung, denn sie unterstützt unser Verständnis, dass beide Methoden beinahe dasselbe tun. Aber für Übersetzung und Ausführung des Programms hat diese Namensgleichheit überhaupt keine Bedeutung, es sind einfach zwei verschiedene Methoden, so als wenn wir den beiden Methoden unterschiedliche Namen gegeben hätten.

Eine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class FastRobot extends SymmTurner {

   public FastRobot ( int x, int y, Direction direction,
                      int numberOfCoins ) {
      super ( x, y, direction, numberOfCoins );
   }

   public void move ( int numberOfSteps ) {
      for ( ... numberOfSteps ... )
         move();
   }
}
```

Die neue Methode move macht so viele Schritte vorwärts, wie ihr Parameter es vorgibt. Da die Methode move ohne Parameter *nicht* überschrieben wurde, ist sie weiterhin indirekt von Robot über SymmTurner ererbt und steht daher ganz normal in FastRobot zur Verfügung, muss also – im Gegensatz zur Situation in SlowMotionRobot – nicht mit super angesprochen werden.

Eine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
FastRobot fr = new FastRobot ( 3, 4, UP, 0 );
fr.move();
fr.turnLeft();
fr.move ( 5 );
```

Wir sehen uns jetzt den FastRobot in Aktion an.

Eine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
FastRobot fr = new FastRobot ( 3, 4, UP, 0 );
fr.move();
fr.turnLeft();
fr.move ( 5 );
```

Das ist die von Klasse Robot über Klasse SymmTurner geerbte Methode move. Da sie weder in SymmTurner noch in FastRobot überschrieben wurde, macht sie genau das, was sie auch in Klasse Robot macht.

Eine zweite move-Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
FastRobot fr = new FastRobot ( 3, 4, UP, 0 );
fr.move();
fr.turnLeft();
fr.move ( 5 );
```

Und das ist jetzt die neue Methode move in FastRobot. Durch diesen Aufruf geht der Roboter gleich fünf Schritte auf einmal vorwärts.

Wenn eine Klasse mehrere Methoden mit demselben Namen hat, sagt man, die Methode ist *überladen*. Die Parameterlisten müssen dann allesamt unterschiedlich sein, denn sonst könnte der Compiler bei einem Aufruf nicht erkennen, welche Methode nun genau gemeint ist.

Überladung von Methoden sehen wir uns jetzt sofort im nächsten Abschnitt an.

TECHNISCHE UNIVERSITÄT DARMSTADT

Allgemein:
Methoden überschreiben / überladen

Oracle Java Tutorials:
Defining Methods (Abschnitt Overloading Methods)

***Vorgriff*: Das Thema werden wir nochmals in Kapitel 03c, Abschnitt „Signatur und Überschreiben / Überladen von Methoden“, aufgreifen.**

Überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen einer „normalen“ Methode:

```
public class MyRobot {
    .....
    public void move () { ..... }
    public void move ( int n ) { ..... }
    public void move ( Robot otherRobot ) { ..... }
    public void move ( int n, Robot otherRobot ) { ..... }
    public void move ( Robot otherRobot, int n ) { ..... }
    .....
}
```

Beim Nachbau der Klasse Robot in unserer eigenen Klasse MyRobot in Kapitel 03e hatten wir nur die move-Methode definiert, die auch in Robot vorhanden ist und die eine leere Parameterliste hat.

Überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen einer „normalen“ Methode:

```
public class MyRobot {
    .....
    public void move () { ..... }
    public void move ( int n ) { ..... }
    public void move ( Robot otherRobot ) { ..... }
    public void move ( int n, Robot otherRobot ) { ..... }
    public void move ( Robot otherRobot, int n ) { ..... }
    .....
}
```

Wir könnten aber auch so wie hier beliebig viele Methoden desselben Namens in derselben Klasse definieren. Das nennt man *Überladen*: Der Name move ist hier überladen.

Überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen einer „normalen“ Methode:

```
public class MyRobot {
    .....
    public void move () { ..... }
    public void move ( int n ) { ..... }
    public void move ( Robot otherRobot ) { ..... }
    public void move ( int n, Robot otherRobot ) { ..... }
    public void move ( Robot otherRobot, int n ) { ..... }
    .....
}
```

Allerdings müssen, wie schon gesagt, die Parameterlisten unterschiedlich sein, damit der Compiler bei jedem Aufruf einer Methode move dieser Klasse unzweideutig bestimmen kann, *welche* der Methoden mit Namen move aufgerufen werden soll.

Überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen einer „normalen“ Methode:

```
public class MyRobot {
    .....
    public void move () { ..... }
    public void move ( int n ) {
        for ( int i = 0; i < n; i++ )
            move();
    }
    .....
}
```

Wir spinnen das Beispiel ein wenig fort: Wie Sie sehen, kann man eine Methode move durchaus in einer anderen Methode move aufrufen.

Überladen durch Vererben

TECHNISCHE UNIVERSITÄT DARMSTADT

Erben und Überladen:

```
public class Robot {
   .........
   public void move () { ......... }
   .........
}

public class FastRobot extends Robot {   // erbt move()
   .........
   public void move ( int n ) { ......... }
   .........
}
```

Tatsächlich hatten wir früher in diesem Kapitel schon einen Fall von Überladung, nämlich bei der Klasse FastRobot. Die Methode move mit leerer Parameterliste war ja von Robot an FastRobot vererbt worden, ist also in FastRobot definiert.

Überladen durch Vererben

TECHNISCHE UNIVERSITÄT DARMSTADT

Erben und Überladen:

```
public class Robot {
   .........
   public void move () { ......... }
   .........
}

public class FastRobot extends Robot {    //  erbt move()
   .........
   public void move ( int n ) { ......... }
   .........
}
```

Und zusätzlich hatten wir in FastRobot eine Methode move mit einem Parameter vom Typ int definiert. Damit sind also zwei Methoden move mit unterschiedlichen Parameterlisten in der Klasse FastRobot definiert.

Konstruktor überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen des Konstruktors:

```
public class MyRobot {
   .....
   public MyRobot ( int x, int y ) { ..... }
   public MyRobot ( int x, int y, MyDirection direction ) { ..... }
   public MyRobot ( int x, int y, MyDirection direction,
                    int numberOfCoins ) { ..... }
   .....
}
```

Nicht nur normale Methoden wie move, sondern auch Konstruktoren kann man überladen, das heißt, eine Klasse kann mehrere Konstruktoren mit unterschiedlichen Parameterlisten haben.

Überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen des Konstruktors:

```
public MyRobot ( int x, int y,
                 MyDirection direction,
                 int numberOfCoins ) {
   this.x = x;
   this.y = y;
   this.direction = direction;
   this.numberOfCoins = numberOfCoins;
}
```

Hier sehen wir noch einmal unseren Nachbau des Konstruktors von Klasse Robot, den wir bei Einrichtung jedes Objektes von Klasse Robot bisher verwendet hatten.

Überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen des Konstruktors:

```
public MyRobot ( int x, int y, MyDirection direction ) {
   this.x = x;
   this.y = y;
   this.direction = direction;
   this.numberOfCoins = 0;
}
```

Es ist nicht unüblich, so wie hier Konstruktoren hinzuzufügen, die nicht für jedes Attribut einen initialisierenden Parameter haben, sondern für manche Attribute einen Standardwert einsetzen, zum Beispiel Standardwert 0 für die initiale Anzahl der Münzen, was ja in vielen Fällen der passende Wert ist.

Überladen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überladen des Konstruktors:

```
public class MyRobot {
    .........
    public MyRobot () {
      this ( 1, 1, UP, 0 );
    }
    .........
    public MyRobot ( int x, int y, MyDirection direction, int numberOfCoins ) {
       .........
    }
    .........
}
```

Man kann auch einen Konstruktor einer Klasse in einem anderen Konstruktor der Klasse aufrufen, das muss dann aber die allererste Anweisung sein. Dazu schreibt man das Schlüsselwort this gefolgt von der Parameterliste hin.

TECHNISCHE UNIVERSITÄT DARMSTADT

Array mit gemischten Klassen

Weiter vorne hatten wir schon gesehen, dass eine Referenz von einer Klasse nicht notwendig auf ein Objekt derselben Klasse verweisen muss, sondern alternativ auch auf ein Objekt einer direkt oder indirekt abgeleiteten Klasse verweisen kann. Dadurch lassen sich Arrays mit einer Mischung aus Klassen bauen.

Array mit gemischten Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot[ ] robots = new Robot[6];
robots[0] = new SymmTurner ( ..... );
robots[1] = new SlowMotionRobot ( ..... );
robots[2] = new Robot ( ..... );
robots[3] = new SlowMotionRobot ( ..... );
robots[4] = new FastRobot ( ..... );
robots[5] = new SymmTurner ( ..... );

for ( int i = 0; i < robots.length; i++ )
   robots[i].move();
```

Hier sehen Sie ein solches gemischtes Array. Der Komponententyp des Arrays ist die Basisklasse, das heißt, die Referenzen sind alle von der Basisklasse Robot. Die Objekte können daher von abgeleiteten Klassen sein.

Array mit gemischten Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot[ ] robots = new Robot[6];
robots[0] = new SymmTurner ( ..... );
robots[1] = new SlowMotionRobot ( ..... );
robots[2] = new Robot ( ..... );
robots[3] = new SlowMotionRobot ( ..... );
robots[4] = new FastRobot ( ..... );
robots[5] = new SymmTurner ( ..... );

for ( int i = 0; i < robots.length; i++ )
   robots[i].move();
```

Natürlich dürfen die Komponenten auch von der Basisklasse sein.

Array mit gemischten Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot[ ] robots = new Robot[6];
robots[0] = new SymmTurner ( ..... );
robots[1] = new SlowMotionRobot ( ..... );
robots[2] = new Robot ( ..... );
robots[3] = new SlowMotionRobot ( ..... );
robots[4] = new FastRobot ( ..... );
robots[5] = new SymmTurner ( ..... );

for ( int i = 0; i < robots.length; i++ )
   robots[i].move();
```

Und selbstverständlich dürfen verschiedene Komponenten auch von derselben Klasse sein.

Array mit gemischten Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot[ ] robots = new Robot[6];
robots[0] = new SymmTurner ( ..... );
robots[1] = new SlowMotionRobot ( ..... );
robots[2] = new Robot ( ..... );
robots[3] = new SlowMotionRobot ( ..... );
robots[4] = new FastRobot ( ..... );
robots[5] = new SymmTurner ( ..... );

for ( int i = 0; i < robots.length; i++ )
   robots[i].move();
```

Da neben der Basisklasse nur Klassen vorkommen können, die von dieser Basisklasse abgeleitet sind, ist gewährleistet, dass jedes Objekt die Methode move mit leerer Parameterliste hat – entweder ererbt oder überschrieben.

TECHNISCHE UNIVERSITÄT DARMSTADT

Zugriffsrechte und Packages

Oracle Java Tutorials: Lesson Packages

Wir haben bisher etliche Beispiele für public und private gesehen. Nachdem wir jetzt Vererbung verstanden haben, können wir das bisher gewonnene Bild systematisch abrunden.

Packages

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import fopbot.*;

.........

Robot robot = new Robot ( ......... );
```

Wir werden später, in Kapitel 03a, noch mehr zu Packages sagen, hier nur soviel: Man kann einen oder mehrere Definitionen von Klassen, Enumerationen und ähnlichem zu einem Package zusammenfassen. Den Import eines solchen Packages haben wir in den bereitgestellten Quelltext-Dateien immer wieder gesehen. Nur durch diesen Import ist es möglich, Robot, Direction und World zu verwenden, denn die sind allesamt im importierten Package fopbot zusammengefasst.

Solche import-Anweisungen müssen immer ganz am Anfang der Quelltext-Datei stehen. Wie immer, wird am Ende ein Semikolon gesetzt.

Packages

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import fopbot.*;

.........

Robot robot = new Robot ( ......... );
```

Mit dem Stern nach dem Namen des Packages, getrennt durch einen Punkt, wird gesagt, dass man *alle* Definitionen aus diesem Package importieren möchte.

Packages

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import fopbot.Direction;
import fopbot.Robot;
.....

Robot robot = new Robot ( ..... );
```

Man kann auch einzelne Klassen, Enumerationen und so weiter benennen, dann werden nur diese importiert. Das heißt, die anderen Definitionen aus diesem Package können weiterhin nicht verwendet werden.

Packages

TECHNISCHE UNIVERSITÄT DARMSTADT

Datei MyRobot.java:

```
package myfopbot;
.....

public class MyRobot {
   .....
}
```

Datei MyDirection.java:

```
package myfopbot;
.....

public enum MyDirection {
   .....
}
```

Mit dieser Zeile ganz am Anfang einer Quelltext-Datei geben Sie an, dass die Definitionen in dieser Quelltext-Datei zu dem Package gehören sollen, dessen Namen hinter dem Schlüsselwort package steht.

Der Name eines Package ist ein Identifier. Konvention ist, dass Packagenamen eher kurz gehalten und vollständig klein geschrieben werden.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Bei Klassen, Enumerationen u.ä.:

- **public oder gar nichts**
- **Mit public:**
  - **Nur eine Klasse, Enumeration o.ä. pro Quelltextdatei**
  - **Deren Name ist der Name der Quelldatei**

Quelldatei Y.java:

```
class X { ..... }
public class Y { ..... }
class Z { ..... }
```

Jetzt kommen wir zu den Zugriffsrechten für Klassen, Enumerationen und ähnliches.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Bei Klassen, Enumerationen u.ä.:

- **public oder gar nichts**
- **Mit public:**
  - **Nur eine Klasse, Enumeration o.ä. pro Quelltextdatei**
  - **Deren Name ist der Name der Quelldatei**

Quelldatei Y.java:

```
class X { ..... }
public class Y { ..... }
class Z { ..... }
```

Man kann nur entscheiden, ob die Klasse beziehungsweise Enumeration public sein soll oder ob public weggelassen werden soll. Ein private alternativ zum public ist nicht möglich.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Bei Klassen, Enumerationen u.ä.:

- **public oder gar nichts**
- **Mit public:**
  - **Nur eine Klasse, Enumeration o.ä. pro Quelltextdatei**
  - **Deren Name ist der Name der Quelldatei**

Quelldatei Y.java:

```
class X { ..... }
public class Y { ..... }
class Z { ..... }
```

Wir haben schon mehrfach gesagt, dass es nur eine public-Definition pro Quelltext-Datei geben darf und dass die Datei denselben Namen tragen muss.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Bei Klassen, Enumerationen u.ä.:

- **public oder gar nichts**
- **Mit public:**
  - **Nur eine Klasse, Enumeration o.ä. pro Quelltextdatei**
  - **Deren Name ist der Name der Quelldatei**

Quelldatei Y.java:

```
class X { ..... }
public class Y { ..... }
class Z { ..... }
```

Häufig macht es Sinn, zur Verbesserung der Struktur Funktionalität aus der eigentlich zu erstellenden Klasse in weitere Klassen auszulagern, diese dann aber nicht nach außen sichtbar zu machen. Das ist der Grund hinter der Möglichkeit, Definitionen ohne public hinzuzufügen.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zugriff bei Attributen und Methoden einer Klasse:

- private: nur in der Klasse selbst
- Keine Angabe: zusätzlich zu private auch im Package
- protected: zusätzlich zu „keine Angabe“ in allen direkt oder indirekt abgeleiteten Klassen
- public: zusätzlich zu protected überall wo das Package importiert ist

```
public class X {
    private int i1;
    private void m1() { ..... }
    int i2;
    void m1() { ..... }
    protected int i3;
    protected void m1() { ..... }
    public int i4;
    public void m1() { ..... }
}
```

Kommen wir nun systematisch zu Zugriffsrechten bei Attributen und Methoden einer Klasse. Links sehen Sie die allgemeinen Regeln, rechts ein rein illustratives Beispiel, also ein Beispiel, das nicht wirklich etwas Sinnvolles tun, sondern die Regeln links veranschaulichen soll.

Die Reihenfolge von Attributen und Methoden ist völlig egal. Auch die Reihenfolge der Zugriffsrechte ist egal. Die vier Attribute und vier Methoden rechts hätten auch in beliebiger anderer Reihenfolge in der Klasse X erscheinen können. Sie sind hier so geordnet, dass die linke und die rechte Spalte der Folie möglichst gut zusammenpassen.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zugriff bei Attributen und Methoden einer Klasse:

- private: nur in der Klasse selbst
- Keine Angabe: zusätzlich zu private auch im Package
- protected: zusätzlich zu „keine Angabe“ in allen direkt oder indirekt abgeleiteten Klassen
- public: zusätzlich zu protected überall wo das Package importiert ist

```
public class X {
  private int i1;
  private void m1() { ..... }
  int i2;
  void m1() { ..... }
  protected int i3;
  protected void m1() { ..... }
  public int i4;
  public void m1() { ..... }
}
```

Die restriktivste Form ist private. Die Methoden der Klasse dürfen auf dieses Attribut lesend oder schreibend zugreifen beziehungsweise diese Methode verwenden, ansonsten ist das nirgendwo möglich.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int i1;
   private void m1() { ..... }
   public void m2 ( X x ) {
      i1 = 1;
      x.i1 = 2;
      m1();
      x.m1();
   }
}
```

Wir hatten das schon gesehen, sollten das aber doch noch einmal klarstellen: Zugriffsbeschränkung auf die eigene Klasse bedeutet *nicht* Zugriffsbeschränkung auf das eigene Objekt. In einer Methode einer Klasse kann auch auf die private-Attribute und –methoden anderer Objekte derselben Klasse zugegriffen werden. Anders herum gesagt: Auf die private-Attribute und -methoden eines Objektes kann auch von anderen Objekten derselben Klasse zugegriffen werden.

Mit „zugreifen“ ist hier natürlich bei Attributen lesender und schreibender Zugriff gemeint, bei Methoden ist mit „zugreifen“ der Aufruf gemeint.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zugriff bei Attributen und Methoden einer Klasse:

- private: nur in der Klasse selbst
- Keine Angabe: zusätzlich zu private auch im Package
- protected: zusätzlich zu „keine Angabe“ in allen direkt oder indirekt abgeleiteten Klassen
- public: zusätzlich zu protected überall wo das Package importiert ist

```
public class X {
    private int i1;
    private void m1() { ..... }
    int i2;
    void m1() { ..... }
    protected int i3;
    protected void m1() { ..... }
    public int i4;
    public void m1() { ..... }
}
```

Man muss gar kein Zugriffsrecht dazuschreiben, dann ist das Attribut beziehungsweise die Methode nicht nur in der eigenen Klasse, sondern im gesamten eigenen Package zugreifbar.

Zugriffsrechte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zugriff bei Attributen und Methoden einer Klasse:

- private: nur in der Klasse selbst
- Keine Angabe: zusätzlich zu private auch im Package
- protected: zusätzlich zu „keine Angabe“ in allen direkt oder indirekt abgeleiteten Klassen
- public: zusätzlich zu protected überall wo das Package importiert ist

```
public class X {
    private int i1;
    private void m1() { ..... }
    int i2;
    void m1() { ..... }
    protected int i3;
    protected void m1() { ..... }
    public int i4;
    public void m1() { ..... }
}
```

Das Schlüsselwort protected kannten wir bisher nicht. Es erweitert den Zugriff auf alle direkt oder indirekt abgeleiteten Klassen, auch wenn sie nicht im eigenen Package, sondern in anderen Packages definiert sind.

Zugriffsrechte

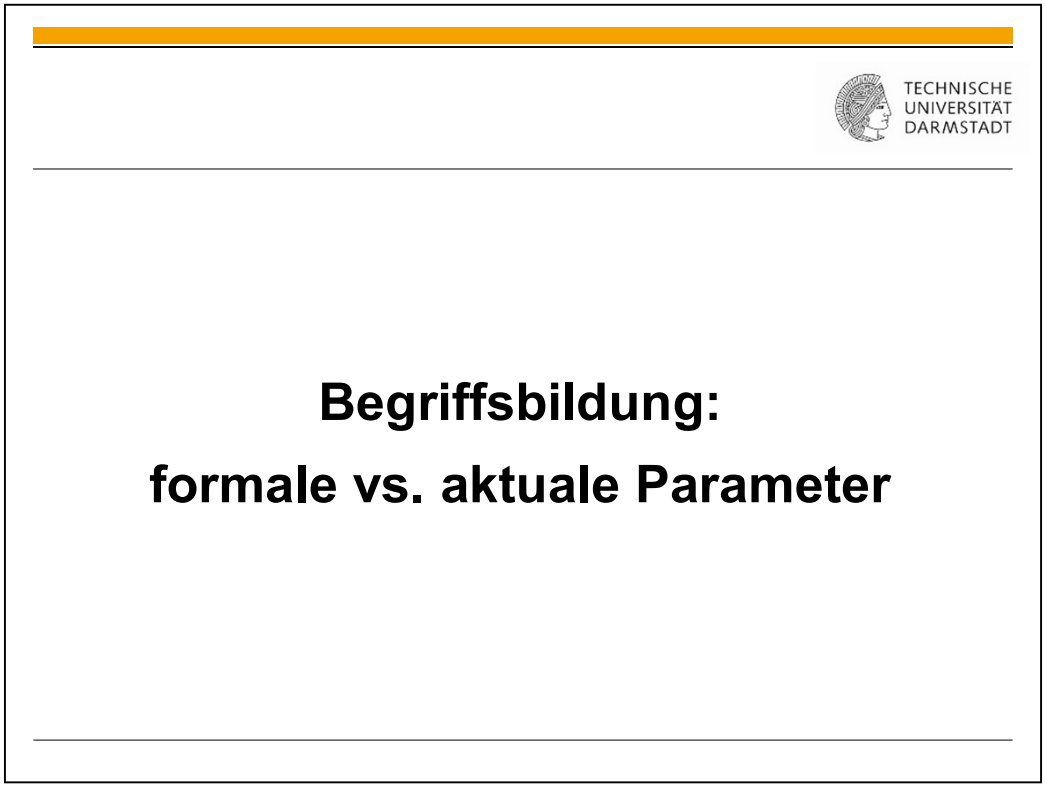
TECHNISCHE UNIVERSITÄT DARMSTADT

Zugriff bei Attributen und Methoden einer Klasse:

- private: nur in der Klasse selbst
- Keine Angabe: zusätzlich zu private auch im Package
- protected: zusätzlich zu „keine Angabe“ in allen direkt oder indirekt abgeleiteten Klassen
- public: zusätzlich zu protected überall wo das Package importiert ist

```
public class X {
    private int i1;
    private void m1() { ..... }
    int i2;
    void m1() { ..... }
    protected int i3;
    protected void m1() { ..... }
    public int i4;
    public void m1() { ..... }
}
```

Und bei public ist der Zugriff dann überall erlaubt, natürlich vorausgesetzt, dass die Klasse selbst durch import bekannt ist.



Für den Umgang mit den Parametern einer Methode führen wir eine nützliche grundlegende Begrifflichkeit ein.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {
   public void m ( double d, X x )  { ..... }
}

public class Y extends X { ..... }

A a = new A();
a.m ( 3+4*2, new Y() );
```

Wieder ein einfaches, rein illustratives Beispiel: eine Klasse A mit einer Methode m, wobei uns nicht interessiert, was m genau tut. Auch die Klassen X und Y interessieren uns nicht, wir verwenden sie nur als Spielmaterial.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {
   public void m ( double d, X x )  { ..... }
}

public class Y extends X { ..... }

A a = new A();
a.m ( 3+4*2, new Y() );
```

Was bei der Methoden*definition* in der Parameterliste steht, das sind die *formalen* Parameter. Jeder formale Parameter wird durch Typnamen und Identifier spezifiziert. Im Prinzip sind das lokale Variable der Methode, die durch den Methodenaufruf initialisiert werden.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {
   public void m ( double d, X x )  { ..... }
}

public class Y extends X { ..... }

A a = new A();
a.m ( 3+4*2, new Y() );
```

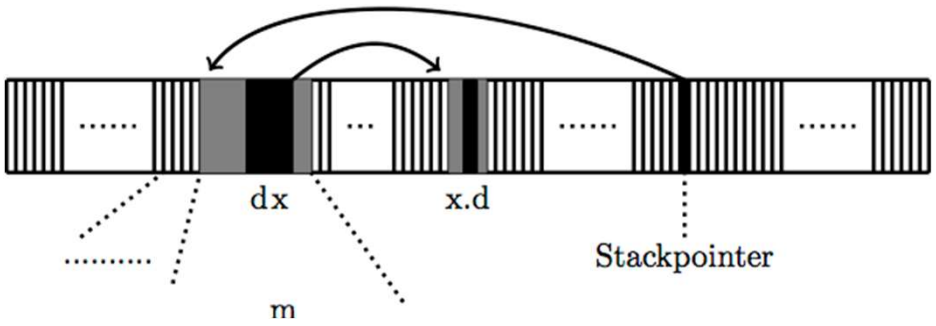
Was hingegen beim Methoden*aufruf* in der Parameterliste steht, das sind die *aktualen* Parameter. Wie man sieht, können die aktualen Parameter beliebig komplizierte Ausdrücke sein, und ein aktualer Parameter muss nicht denselben Typ haben wie der zugehörige formale Parameter. Es reicht, wenn der Typ des aktualen Parameters implizit in den Typ des formalen Parameters konvertierbar ist.

In diesem Beispiel ist der erste aktuale Parameter vom Typ int, und int wird implizit in double konvertiert. Die Klasse Y des zweiten aktualen Parameters ist vom zweiten formalen Parameter, also von X, abgeleitet, auch da passiert implizite Konversion.

Pate bei dieser Begriffsbildung stand das englische Wort actual, das nicht aktuell, sondern unter anderem tatsächlich oder auch vorliegend heißt.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT



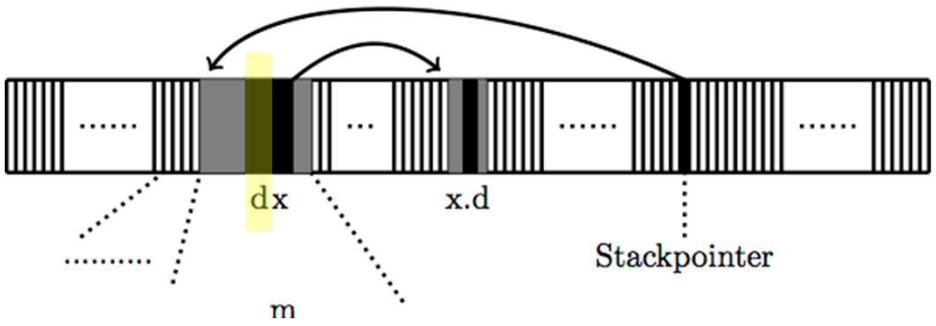

```
public void m ( double d, X x ) {
    x.d = 2 * d – 3;
}
```

```
X x = new X();
.........
a.m ( 3+4*2, x );
```

Wir hatten ja schon des Öfteren gesehen, dass die Ausführung eines Methodenaufrufs zur Ablage eines Frames auf dem Call-Stack führt. Im Frame für eine Methode mit Parametern ist für jeden Parameter ein bestimmter Platz reserviert, der wie bei allen anderen Daten im Frame immer denselben Offset von der Anfangsadresse des Frames hat. An diese Stelle wird beim Aufruf der Methode der Wert des aktualen Parameters geschrieben, und an dieser Stelle wird innerhalb der Methode auf diesen Wert zugegriffen.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT



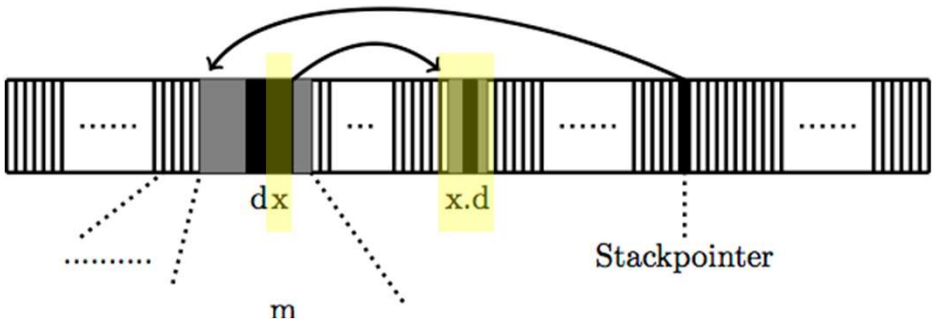

```
public void m ( double d, X x ) {
    x.d = 2 * d – 3;
}
```

```
X x = new X();
.........
a.m ( 3+4*2, x );
```

Auf den Wert 11 des ersten aktualen Parameters wird also zugegriffen, indem der Offset des formalen Parameters d auf die im Stackpointer gespeicherte Anfangsadresse des Frames draufaddiert und der Wert an der resultierenden Adresse ausgelesen wird.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT



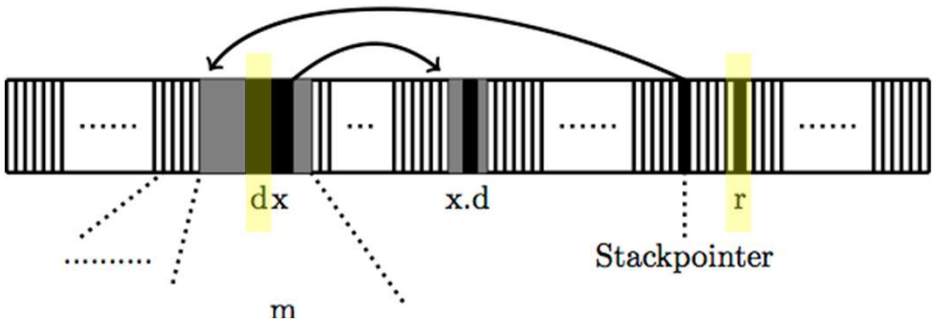

```
public void m ( double d, X x ) {
    x.d = 2 * d – 3;
}
```

```
X x = new X();
.....
a.m ( 3+4*2, x );
```

Bei einem Parameter von einem Referenztyp ist das Gesamtbild wegen der Aufspaltung in Referenz und Objekt naturgemäß etwas komplizierter, aber abgesehen von dieser Aufspaltung eigentlich völlig analog.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT



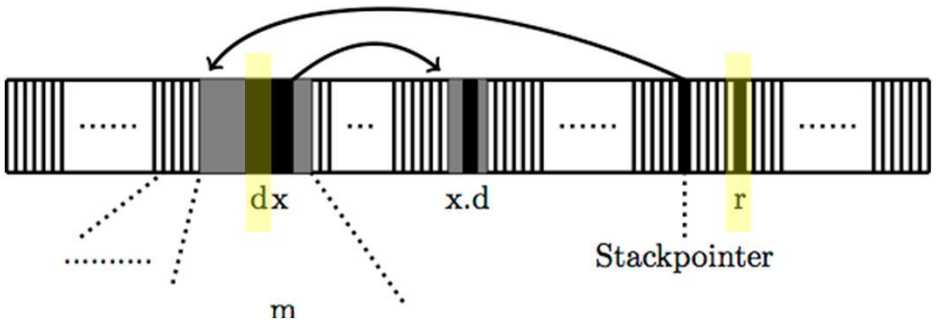

```
public void m ( double d, X x ) {
    x.d = 2 * d – 3;
    d = 3.14;
}
```

```
X x = new X();
double r = 3+4*2;
a.m ( r, x );     // r == 11
```

Eine kleine Variation des Beispiels: Nun ist der erste aktuale Parameter der Name einer Variable. Die interessierende Frage ist, welcher Wert in r nach Ausführung der Methode m steht – die 11 oder die 3.14.

Formale vs. aktuale Parameter

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public void m ( double d, X x ) {
    x.d = 2 * d – 3;
    d = 3.14;
}
```

```
X x = new X();
double r = 3+4*2;
a.m ( r, x );     // r == 11
```

Tatsächlich ist der Wert in r weiterhin 11. Das liegt wieder daran, dass es bei primitiven Datentypen die Aufteilung in Referenz und Objekt nicht gibt. Wie wir es schon mehrfach bei Zuweisungen gesehen haben, wird hier tatsächlich der Zahlenwert selbst aus dem aktualen Parameter in den Speicherplatz im Frame für den formalen Parameter kopiert, keine Adresse. Mit anderen Worten: r und d haben nichts miteinander zu tun, außer dass d zu Beginn der Ausführung der Methode m den Wert von r bekommt. In der farblich unterlegten Zeile links wird dieser Wert dann durch einen anderen überschrieben, aber das hat nichts mit r zu tun.