

Kapitel 01a: Einführung in Java mit FopBot

Karsten Weihe

Don't Panic!



Keine Panik: Wir werden mit Hilfe von FopBot sanft in die Programmiersprache Java einsteigen!

Don't Panic!

- **Java ist eine sehr komplexe Programmiersprache**
 - **Für reale Anwendungen konzipiert**
 - **Ursprünglich nicht als Lernsprache gedacht**
- **Auch in den FopBot-Beispielen kann man die Komplexität nicht vollständig verbergen**

Das ist auch notwendig, da Java eine eher unsanft zu lernende Sprache ist. Man kann auch nicht mit ein paar einfachen Teilen beginnen und alle anderen Aspekte von Java auslassen, denn zum Funktionieren auch des allerkleinsten Programms sind etliche Aspekte von Java notwendig, die nicht alle leicht zu verstehen sind.

Don't Panic!



- **Java ist eine sehr komplexe Programmiersprache**
 - Für reale Anwendungen konzipiert
 - Ursprünglich nicht als Lernsprache gedacht
- **Auch in den FopBot-Beispielen kann man die Komplexität nicht vollständig verbergen**
- **Unsere Herangehensweise bei FopBot:**
 - In *Vorlesung* und *Übungen* wird immer klar getrennt, was zu verstehen ist und was nicht
 - In den *Übungen* können Sie die Teile, die noch nicht zu verstehen sind, ignorieren

Das betrifft nur dieses erste Kapitel, das Kapitel zu FopBot. Aber in unserer Herangehensweise ist das für Sie absolut kein Problem. Sie werden in diesem Kapitel und in den darauf basierenden weiteren Kapiteln durch die Java-Programme so geführt, dass Sie sich nur immer die Details genauer anschauen, um die es gerade geht. Diese Details sollten Sie damit gut verstehen können.

In den Übungen werden wir Sie ebenfalls um die Komplexität von Java erst einmal herumleiten. In den Java-Programmen, mit denen Sie arbeiten, werden Sie konkret diverse Lücken finden, und nur diese Lücken haben Sie zu füllen. Und diese Lücken werden sich mit Ihrem Vorwissen aus der Vorlesung dann auch gut füllen lassen.

Alles alte Hüte für Sie?



- Sie kennen schon alles und langweilen sich?

Sie haben vielleicht Vorkenntnisse in Java, vielleicht sogar gute Vorkenntnisse. Dann wird Ihnen sicherlich vieles am Anfang und punktuell auch später schon bekannt sein. Sie könnten versucht sein, Langeweile zu entwickeln und heraushängen zu lassen.

Alles alte Hüte für Sie?



- Sie kennen schon alles und langweilen sich?
- Das wird sich eher früher als später ändern!

Wenn Sie kein Java-Vollprofi mit jahrelangen Erfahrungen sind, dann ist das aber nur eine vorübergehende Situation. Sie werden irgendwann plötzlich merken, dass Sie gar nicht mehr so alles kennen und auf Anhieb verstehen.

Alles alte Hüte für Sie?



- Sie kennen schon alles und langweilen sich?
- Das wird sich eher früher als später ändern!
- Und dann geht es genauso schnell weiter!

Sie werden dann allerdings merken, dass die Vorlesung und der Übungsbetrieb nicht langsamer werden, nur weil *Sie* an die Grenze *Ihrer* individuellen Vorkenntnisse gelangt sind.

Alles alte Hüte für Sie?



- **Sie kennen schon alles und langweilen sich?**
- **Das wird sich eher früher als später ändern!**
 - **Und dann geht es genauso schnell weiter!**
 - **Verpassen Sie also nicht den Einstieg ins Neue!**

Wenn Sie das nicht rechtzeitig merken, werden Sie im weiteren Verlauf von Vorlesung und Übungsbetrieb erst einmal abgehängt und müssen das Ganze dann erst wieder nacharbeiten.

Alles alte Hüte für Sie?



- Sie kennen schon alles und langweilen sich?
- Das wird sich eher früher als später ändern!
 - Und dann geht es genauso schnell weiter!
 - Verpassen Sie also nicht den Einstieg ins Neue!
- Auch während der „alten Hüte“ werden Sie wahrscheinlich neue Details und Aspekte sehen (*übersehen?*)

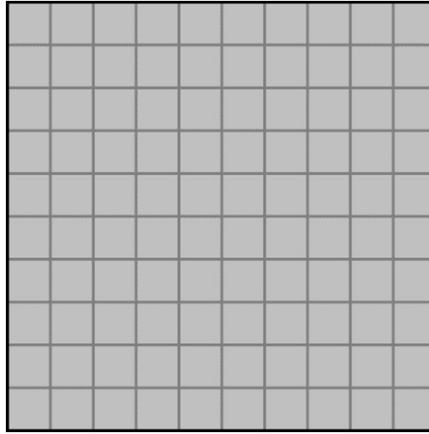
Aber auch bevor Sie an Ihre Grenze gelangt sind, werden Sie so einiges verpassen, wenn Sie sich dafür entscheiden, sich zu langweilen statt konzentriert mitzudenken.

Egal, auf welchem Niveau Ihre Vorkenntnisse sind: Professionelle Haltung ist, die Präsentation Ihnen schon bekannter Konzepte als Gelegenheit zu nutzen, Ihre Erinnerung noch einmal aufzufrischen. Es ist, wie gesagt, *Ihre* Entscheidung, ob Sie sich langweilen.

Erste Schritte in Java mit FopBot

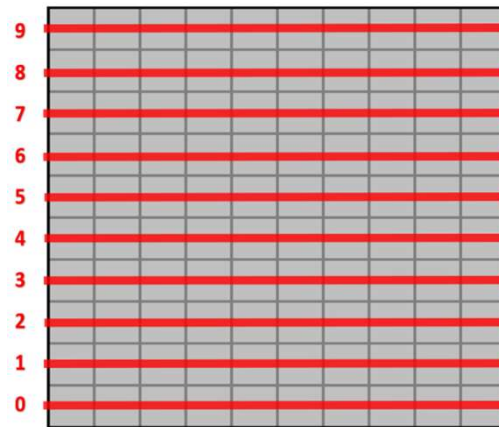
Wir beginnen mit ein paar ersten einfachen Schritten. Das sind schon echte Java-Anweisungen, bei denen wir das Java-Programm FopBot benutzen. Dieses Programm wurde von Lukas Röhrig geschrieben, einem Tutor der FOP, angelehnt an Programme, die den Namen KarelJ tragen.

Die leere *World*



FopBot hat eine sehr einfache Welt, im Grunde nicht mehr als ein rechteckiger Bereich aus Feldern mit ein paar möglichen Objekten darin.

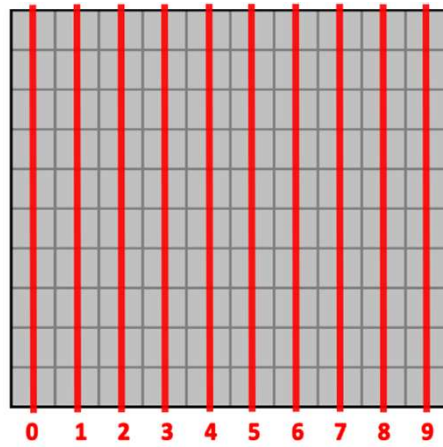
Zeilen



9									
8									
7									
6									
5									
4									
3									
2									
1									
0									

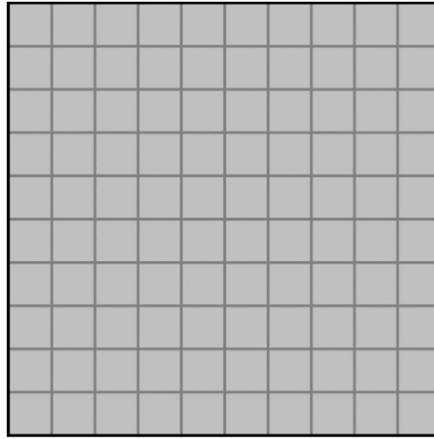
In horizontaler Richtung reden wir von *Zeilen*. Zeilen werden von unten nach oben durchnummeriert. Wie in der Informatik üblich, beginnt die Nummerierung bei 0, nicht bei 1.

Spalten



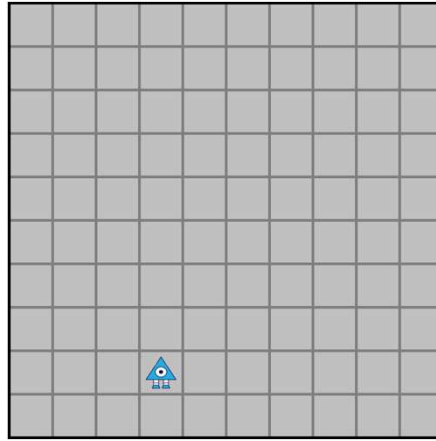
In vertikaler Richtung reden wir von Spalten.

Ersten Robot platzieren



In diesem Koordinatensystem bewegen sich Roboter, die wir einrichten müssen.

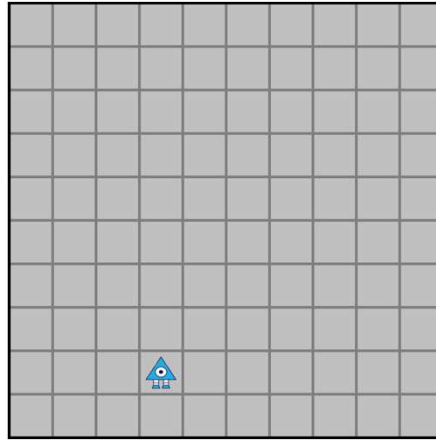
Ersten Robot platzieren



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Mit dieser Anweisung wird ein erster Roboter eingerichtet.

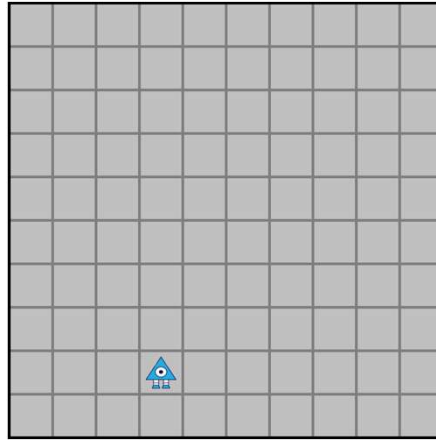
Ersten Robot platzieren



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Im Rahmen der Benennungsregeln von Java, die wir uns bald ansehen, können wir den Roboter beliebig nennen. Wir nennen ihn hier myRobot.

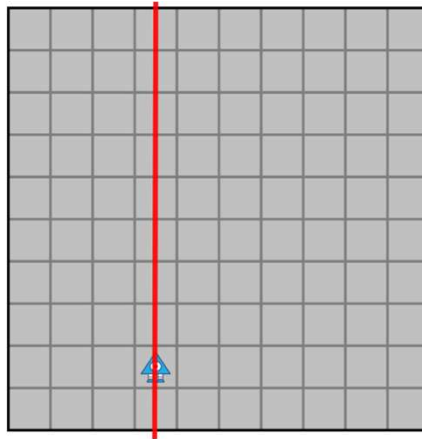
Ersten Robot platzieren



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Hier und hier wird angezeigt, dass myRobot ein Roboter, englisch Robot, und eben nichts anderes ist. Wir sagen, Robot ist die *Klasse* von myRobot. Die Klasse Robot ist in FopBot definiert.

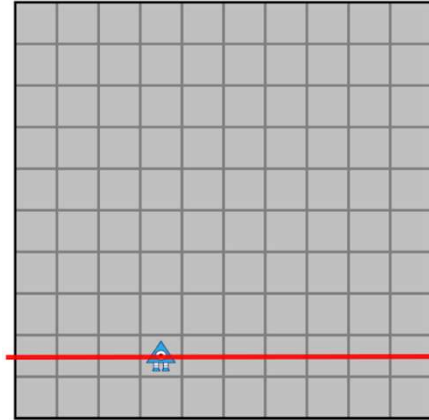
Ersten Robot platzieren



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Die Klasse Robot hat für die Einrichtung eines Roboters vier Parameter. Der erste Parameter ist die Position nach rechts. Der Roboter myRobot wird also in der Spalte mit der Nummer 3 platziert. Das ist die vierte Spalte von links, da die Nummerierung der Spalten von links nach rechts ja mit 0 beginnt.

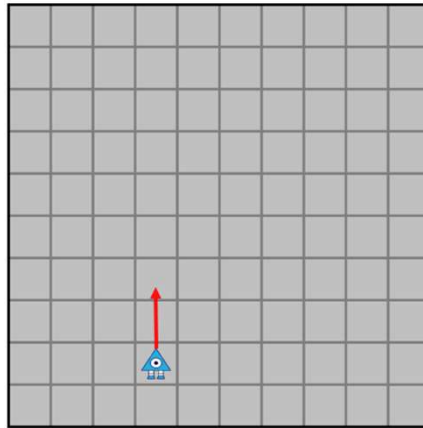
Ersten Robot platzieren



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Der zweite Parameter ist die Position nach oben, also: Der Roboter wird in die Zeile mit Nummer 1 platziert. Da die Nummerierung der Zeilen ebenfalls mit 0 beginnt, ist das die zweite Zeile von unten.

Ersten Robot platzieren

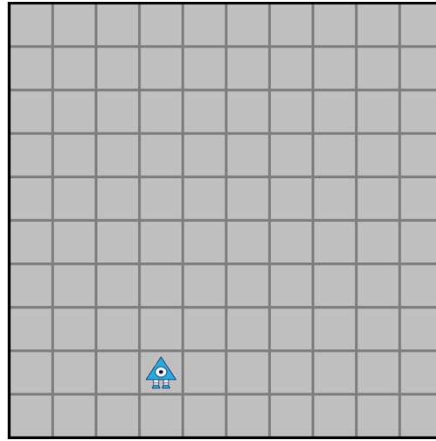


```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Der dritte Parameter gibt die Richtung an, in die der Roboter schaut, also die Richtung seiner Spitze.

UP ist nach oben, DOWN nach unten, RIGHT nach rechts und LEFT nach links, immer alles in Großbuchstaben.

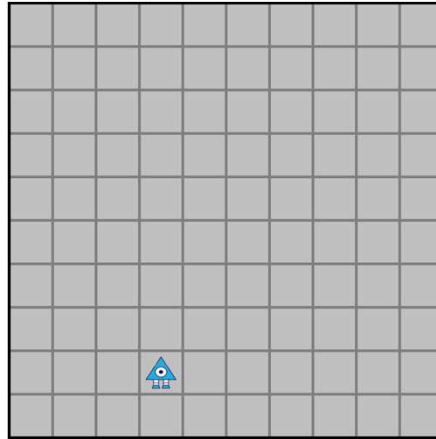
Ersten Robot platzieren



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Den vierten Parameter stellen wir zurück, mit dem befassen wir uns gleich.

Ersten Robot platzieren



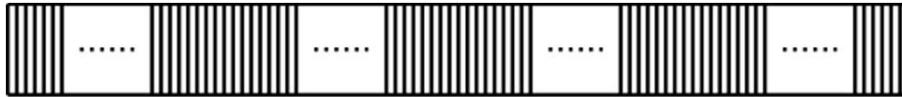
```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Jede Anweisung wird in Java durch ein Semikolon beendet, egal ob danach noch eine weitere Anweisung kommt oder nicht.

Mentales Modell: Daten im Computerspeicher

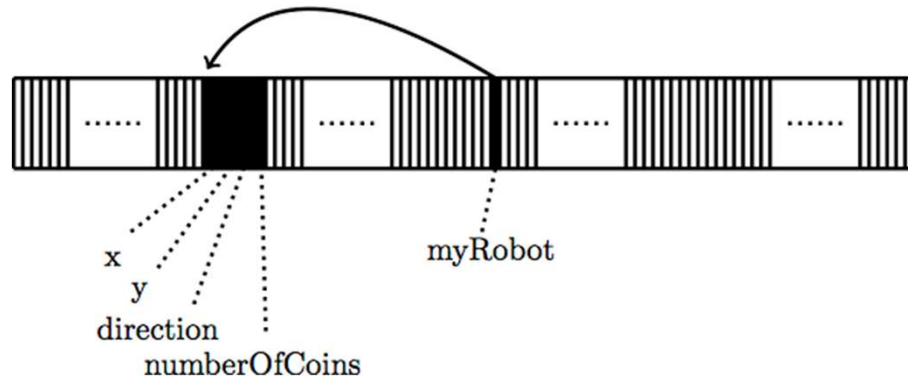
Wichtig für das Verständnis ist, sich ein mentales Modell davon zu machen, was eigentlich so passiert, wenn ein Programm ausgeführt wird. Dazu gehört ein mentales Modell, wie die Objekte im Speicher des Computers prinzipiell abgelegt sind.

Daten im Computerspeicher



Der Speicher ist eigentlich ein sehr komplexes System bestehend aus Hauptspeicher, Cache, Registern, Hintergrundspeicher, Datenbussen und so weiter. Aber für unsere Zwecke reicht es sich vorzustellen, dass der Speicher einfach ein solches großes Feld von einzelnen Maschinenwörtern ist. Jedes Maschinenwort besteht aus derselben Anzahl Bits, aber diese Anzahl Bits ist hardwarespezifisch. Jedes Maschinenwort hat eine eindeutige Adresse, das sind im Prinzip fortlaufende Zahlen 0, 1, 2 ... Je nach ihrer Größe, wird jede Information in einem oder mehreren aufeinanderfolgenden Maschinenwörtern gespeichert.

Daten im Computerspeicher

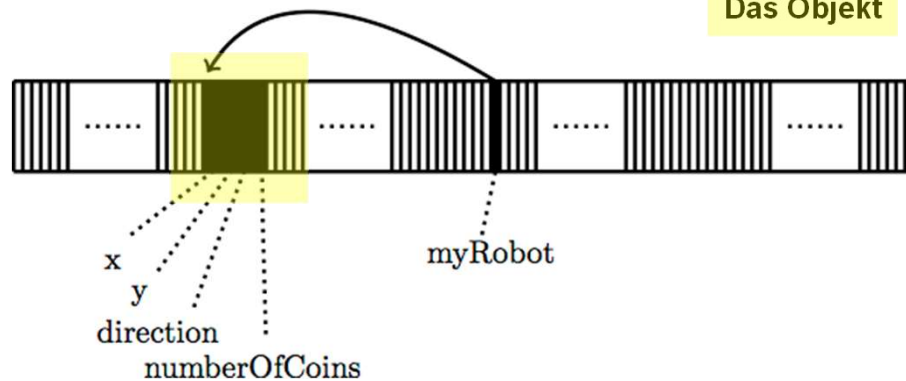


```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

So ungefähr sieht es aus, wenn die Anweisung von eben ausgeführt wird. Sie finden diese Anweisung noch einmal unten rechts.

Daten im Computerspeicher

Das Objekt



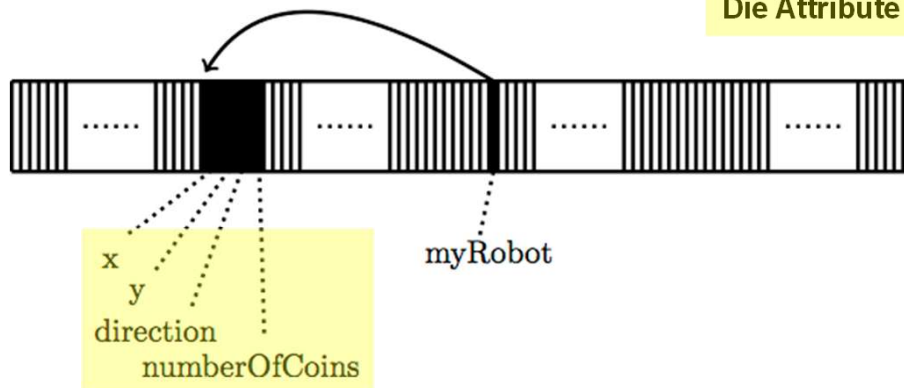
```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Operator new wird gefolgt vom Namen einer Klasse. Alle Objekte einer Klasse sind im Prinzip gleich aufgebaut und gleich groß.

Das Laufzeitsystem, das die Abarbeitung Ihres Programms steuert und kontrolliert, sucht daraufhin ungenutzten Speicherplatz von ausreichender Größe und reserviert ihn für ein Objekt dieser Klasse. Wir können nicht beeinflussen, wo das Laufzeitsystem den Speicher reserviert, und wir können auch nicht sehen, wo es das tut. Aber das brauchen wir auch nicht, denn diese schematischen Skizzen dienen „nur“ dem wichtigen Hintergrundverständnis.

Daten im Computerspeicher

Die Attribute



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

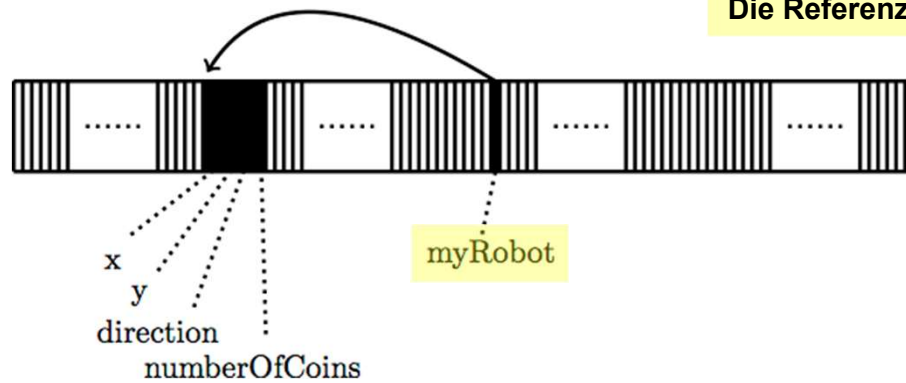
Der Speicherplatz für ein Objekt der Klasse Robot muss groß genug sein, um alle Informationen zu speichern, die für die Klasse Robot vorgesehen sind. Solche Informationen in einer Klasse nennt man die *Attribute* dieser Klasse. In jedem Objekt der Klasse ist jedes Attribut immer an derselben Position relativ zur Anfangsadresse des Objekts.

Klasse Robot hat diese vier Attribute. Die vier Werte in runden Klammern werden den vier Attributen zugewiesen. Auf das vierte Attribut kommen wir gleich zu sprechen.

Nebenbemerkung: Neben diesen vieren hat Klasse Robot auch noch weitere Attribute, die für das Zeichnen des Roboters notwendig sind, hier aber nicht thematisiert werden.

Daten im Computerspeicher

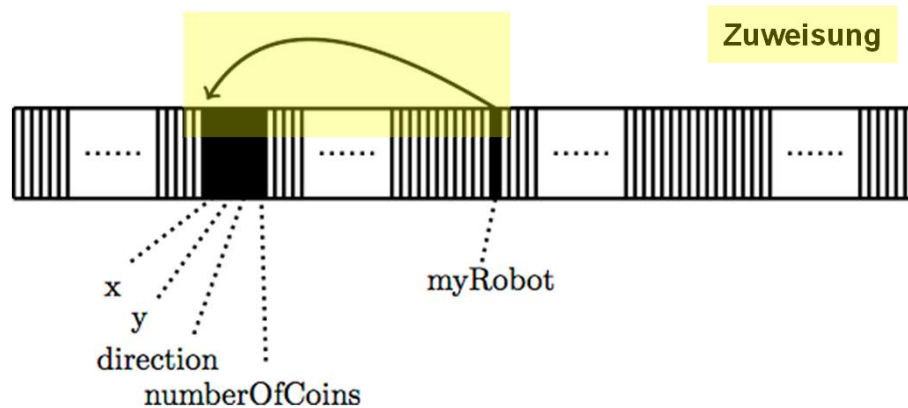
Die Referenz



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Was wir myRobot genannt haben, ist nicht das Objekt selbst, sondern eine weitere, kleine Information, für die das Laufzeitsystem ebenfalls irgendwo Speicherplatz reserviert. Dieses Stück Information nennen wir eine *Referenz*. Objekt und Referenz werden im Prinzip unabhängig voneinander platziert und können an ganz verschiedenen Stellen im Speicherplatz sein.

Daten im Computerspeicher



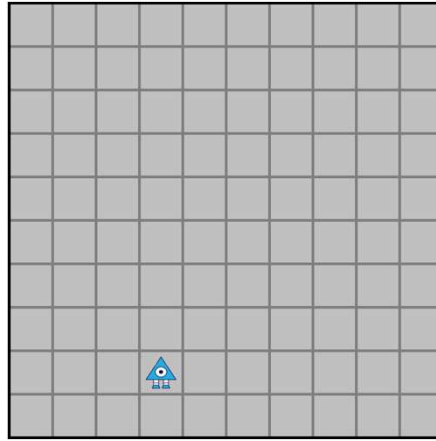
```
Robot myRobot = new Robot ( 3, 1, UP, 0 );
```

Operator new reserviert nicht nur Speicherplatz, sondern liefert auch einen Wert zurück, nämlich die Anfangsadresse des reservierten Speicherplatzes. Das Gleichheitszeichen ist in Java das Zuweisungszeichen: In der Speicherstelle auf der linken Seite des Zuweisungszeichens wird der Wert von der rechten Seite gespeichert. Konkret hier wird also in myRobot die Anfangsadresse der Objekts gespeichert, was wir in unseren schematischen Zeichnungen durch einen Pfeil anzeigen.

Weiter: Erste Schritte in Java mit FopBot

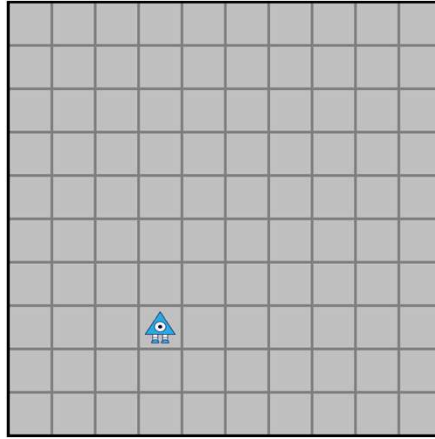
Nachdem dies soweit geklärt ist, geht es weiter mit unserem ersten Java-Programm.

Vorwärtsschritt



Wir betrachten wieder die Situation von eben.

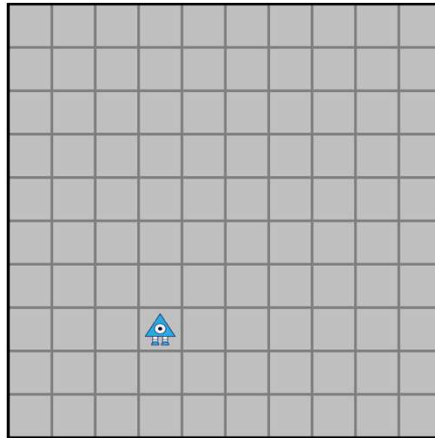
Vorwärtsschritt



`myRobot.move();`

Nun die erste echte Aktion des Roboters: ein Schritt vorwärts in seine momentane Richtung, also nach oben, von der zweiten zur dritten Zeile, innerhalb der vierten Spalte.

Vorwärtsschritt

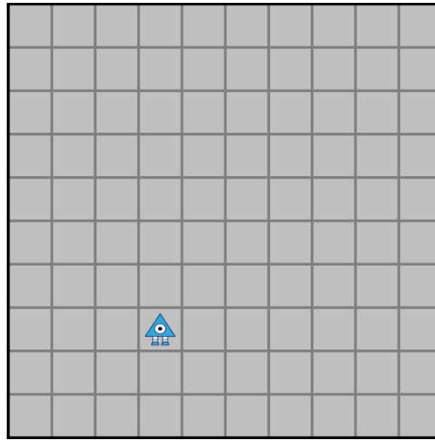


```
myRobot.move();
```

Wir sagen, dieses move ist eine *Methode* der Klasse Robot. Methoden werden aufgerufen, indem sie durch Punkt getrennt hinter den Namen des Roboters geschrieben werden.

Die Methoden einer Klasse werden in der Klasse selbst definiert. Da die Klasse Robot in FopBot definiert ist, ist also auch die Methode move in FopBot definiert, genauso wie alle weiteren Methoden von Klasse Robot, die wir im Laufe der Zeit kennen lernen werden.

Vorwärtsschritt

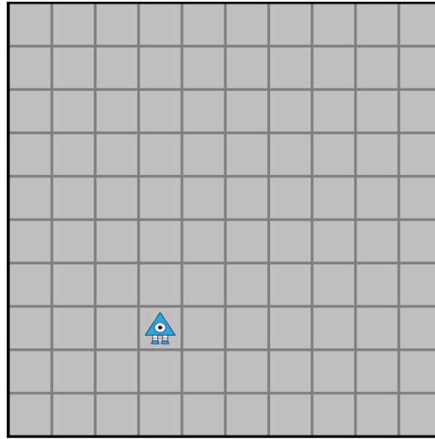


```
myRobot.move();
```

Die Einrichtung des Roboters eben hatte vier Parameter in einem Klammerpaar, durch Kommas getrennt.

Methode move hat im Gegensatz dazu *keine* Parameter, das wird durch ein leeres Klammerpaar angezeigt.

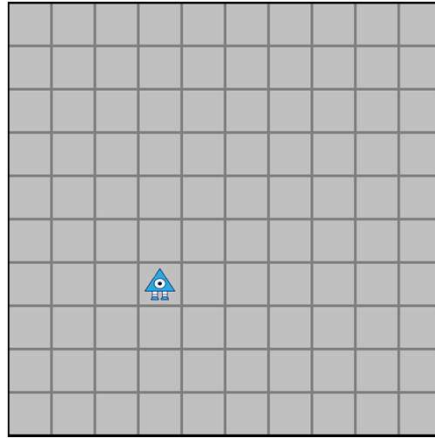
Vorwärtsschritt



```
myRobot.move();
```

Auch bei dieser Anweisung und bei allen folgenden gilt das, was wir eben bei der Einrichtung von myRobot gesagt haben: Jede Anweisung wird durch ein Semikolon abgeschlossen.

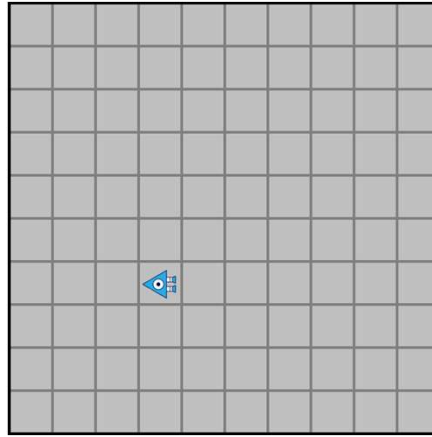
Vorwärtsschritt



```
myRobot.move();
```

Noch ein *zweiter* Vorwärtsschritt, also ein zweiter Aufruf von Methode `move` mit `myRobot`. Nach zwei Vorwärtsschritten nach oben sind wir jetzt in der vierten Zeile.

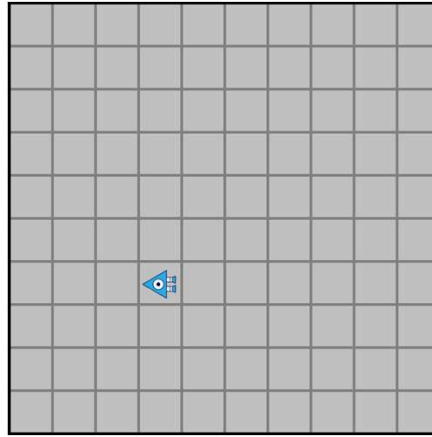
Linksdrehung



```
myRobot.turnLeft();
```

Jetzt eine andere Methode namens `turnLeft`. Bei dieser Methode ändert sich nicht die *Position* des Roboters, sondern seine *Ausrichtung*, und zwar um 90 Grad im Gegenuhrzeigersinn.

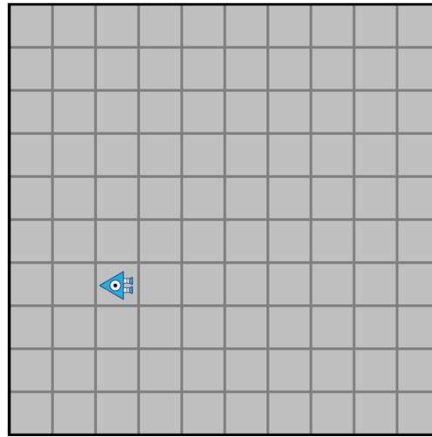
Linksdrehung



```
myRobot.turnLeft();
```

Auch diese Methode hat keine Parameter, also wieder ein leeres Klammerpaar; und wie bei jeder Anweisung ein Semikolon am Ende.

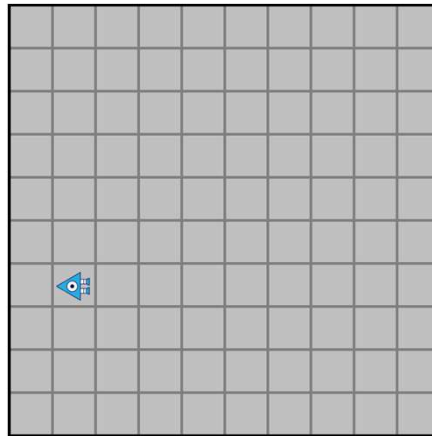
Vorwärtsschritt



`myRobot.move();`

Und wieder ein Vorwärtsschritt mit Methode move. Da der Roboter nach der Linksdrehung nach links zeigt, geht der Schritt also nach links, von der vierten zur dritten Spalte.

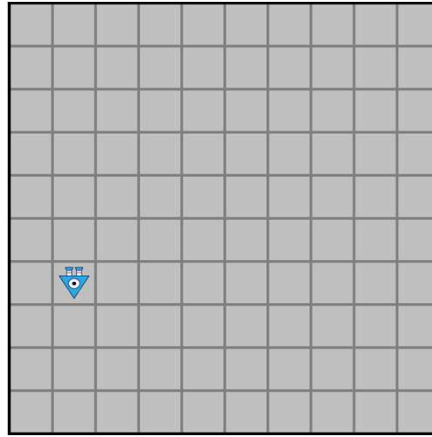
Vorwärtsschritt



`myRobot.move();`

Und noch ein Vorwärtsschritt nach links, von der dritten zur zweiten Spalte.

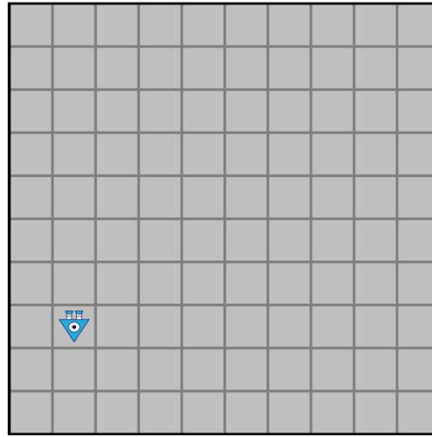
Linksdrehung



```
myRobot.turnLeft();
```

Zur Abwechslung wieder einmal eine Linksdrehung.

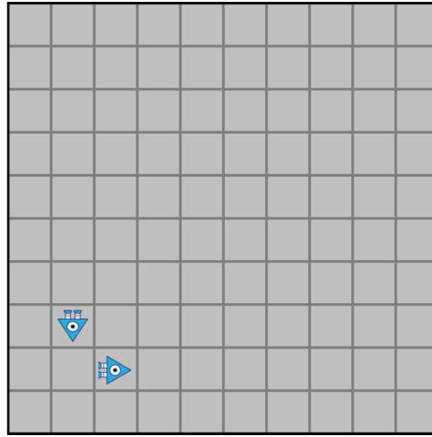
Vorwärtsschritt



`myRobot.move();`

Und ein Vorwärtsschritt nach unten. Damit lassen wir myRobot erst einmal in Ruhe.

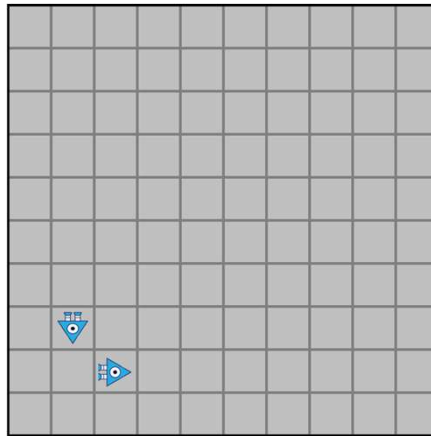
Neuer Robot mit Münzen



```
Robot myRobot2 = new Robot ( 2, 1, RIGHT, 5 );
```

Denn wir richten jetzt einen zweiten Roboter ein und machen erst einmal mit dem weiter.

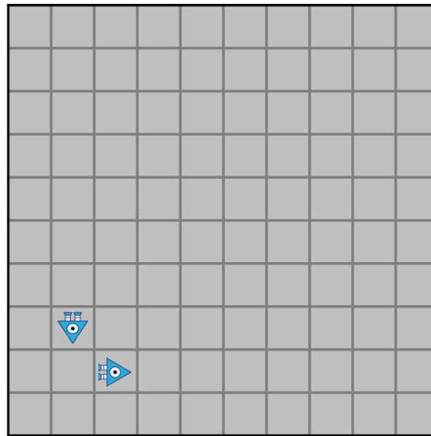
Neuer Robot mit Münzen



```
Robot myRobot2 = new Robot ( 2, 1, RIGHT, 5 );
```

Phantasielos, wie wir sind, nennen wir ihn einfach myRobot2 und sehen dabei: Bezeichner können auch Ziffern enthalten. Das erste Zeichen muss aber immer ein Buchstabe sein.

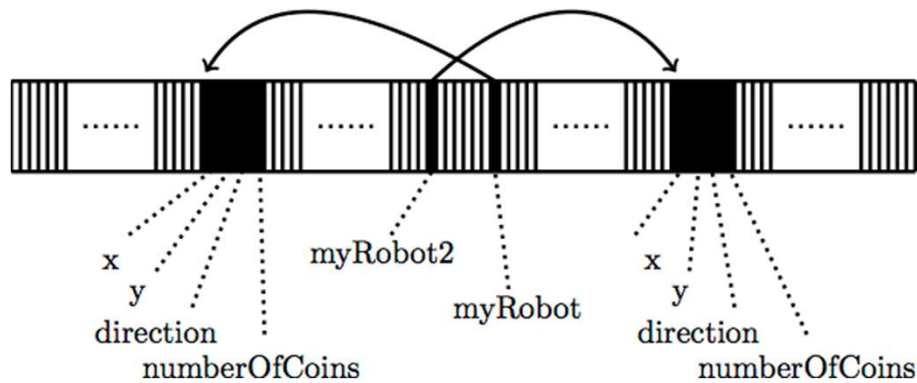
Neuer Robot mit Münzen



```
Robot myRobot2 = new Robot ( 2, 1, RIGHT, 5 );
```

Den vierten Parameter hatten wir bei der Einrichtung von myRobot noch übersprungen, dort hatte er den Wert 0. Bei myRobot2 hat er einen positiven Wert. Jeder Roboter kann Münzen mit sich tragen. Der Roboter myRobot hatte keine Münzen, also 0 Münzen; myRobot2 hingegen hat 5 Münzen.

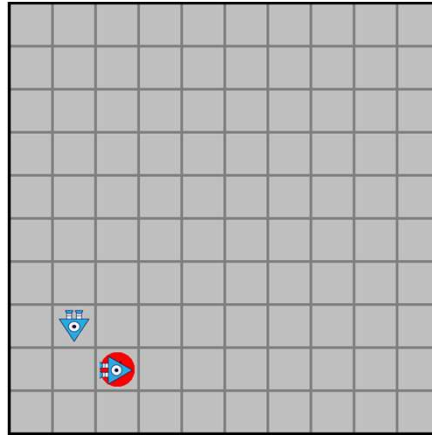
Neuer Robot mit Münzen



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );  
Robot myRobot2 = new Robot ( 2, 1, RIGHT, 5 );
```

Wir sehen uns kurz wieder die Situation im Speicher an. Durch die beiden Aufrufe von Operator `new` wird zweimal jeweils ein Objekt eingerichtet. Beim ersten Objekt werden die Attribute mit den Werten in runden Klammern in der ersten Anweisung initialisiert, beim zweiten Objekt mit den entsprechenden Werten in der zweiten Anweisung. Die Anfangsadresse des ersten Objekts wird der Referenz `myRobot` zugewiesen, die des zweiten analog der Referenz `myRobot2`. Wie diese vier Entitäten im Speicher platziert werden, ist völlig dem Laufzeitsystem überlassen. Die Reihenfolge kann so aussehen wie hier oder auch völlig anders.

Münze ablegen

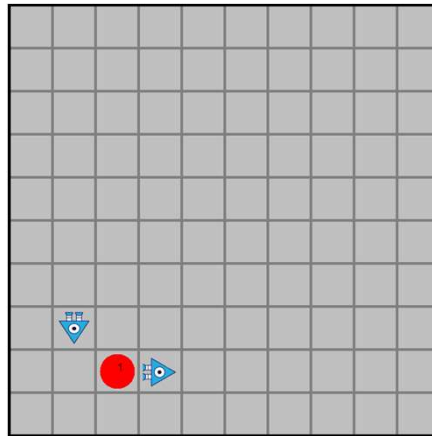


```
myRobot2.putCoin();
```

Mit Methode putCoin legt myRobot2 gleich eine der fünf Münzen an seiner momentanen Position ab. Daher hat myRobot2 statt fünf nur noch vier Münzen. Die abgelegte Münze ist in diesem Bild etwas unklar zu sehen, weil myRobot2 draufsteht.

Auch putCoin hat keine Parameter, also leeres Klammerpaar. Und wie immer ein Semikolon.

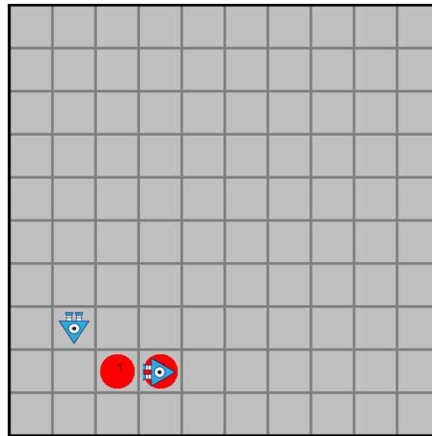
Vorwärtsschritt



```
myRobot2.move();
```

Jetzt ist der Roboter einen Schritt zur Seite gegangen, und wir sehen die Münze auf der vorherigen Position des Roboters genauer. Auf einer Position können mehrere Münzen sein, deshalb steht in dem Symbol für Münzen noch die Zahl der Münzen, die auf dieser Position platziert sind, in diesem Fall eine 1.

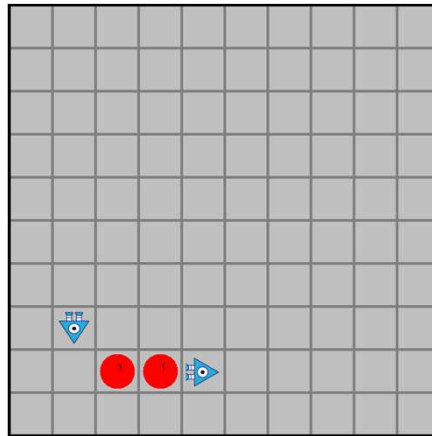
Münze ablegen



```
myRobot2.putCoin();
```

Und noch eine Münze auf dem momentanen Feld abgelegt, jetzt hat myRobot2 nur noch drei Münzen.

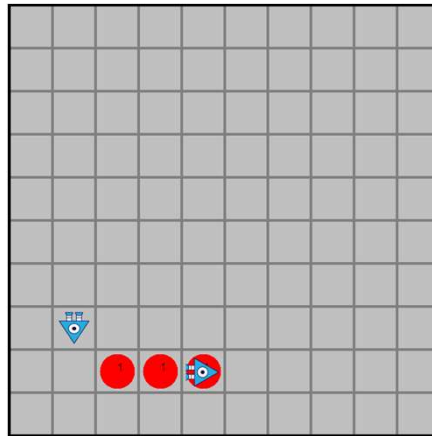
Vorwärtsschritt



`myRobot2.move();`

Durch einen weiteren Schritt nach rechts ist auch diese zweite Münze gut zu sehen.

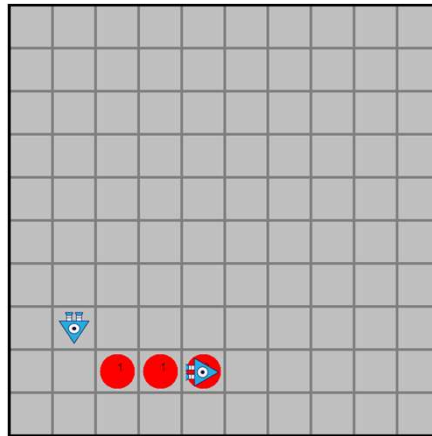
Münze ablegen



```
myRobot2.putCoin();
```

Wir legen an der momentanen Position wieder eine Münze ab.

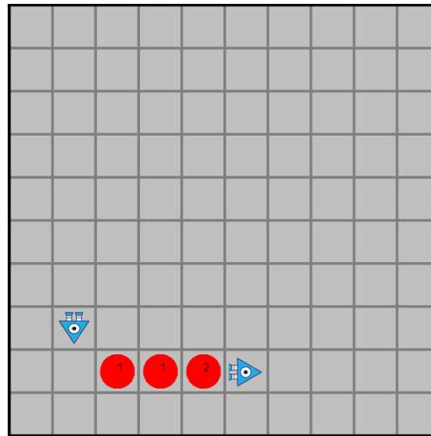
Münze ablegen



```
myRobot2.putCoin();
```

Diesmal legen wir aber noch eine zweite Münze an derselben Position ab, also zwei Aufrufe von putCoin nacheinander, ohne ein move dazwischen. Jetzt hat myRobot2 nur noch eine Münze, und auf dem Feld, auf der myRobot2 steht, sind zwei Münzen.

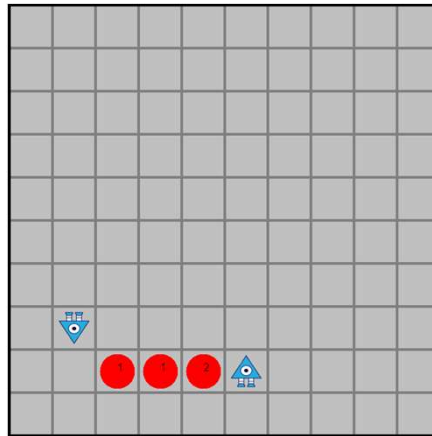
Vorwärtsschritt



`myRobot2.move();`

Nach einem weiteren Schritt sehen wir besser, dass an der letzten Position tatsächlich jetzt *zwei* Münzen sind.

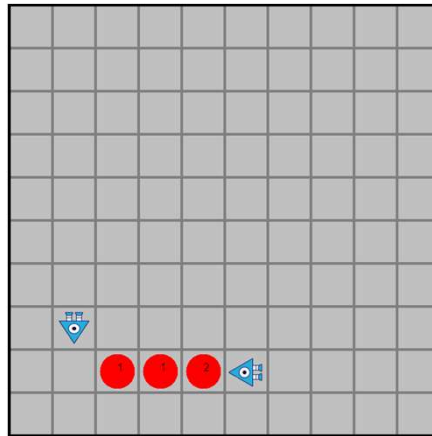
Linksdrehung



```
myRobot2.turnLeft();
```

Eine Linksdrehung, myRobot2 dreht sich von rechts nach oben.

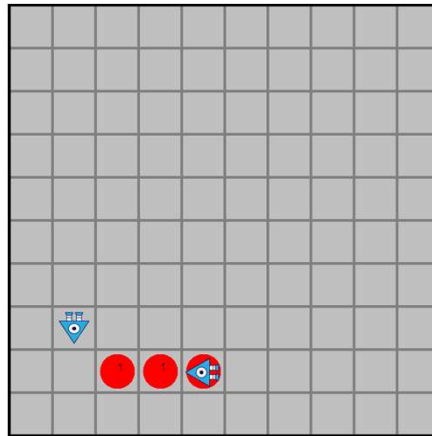
Linksdrehung



```
myRobot2.turnLeft();
```

Und noch eine, also insgesamt eine 180-Grad-Wendung.

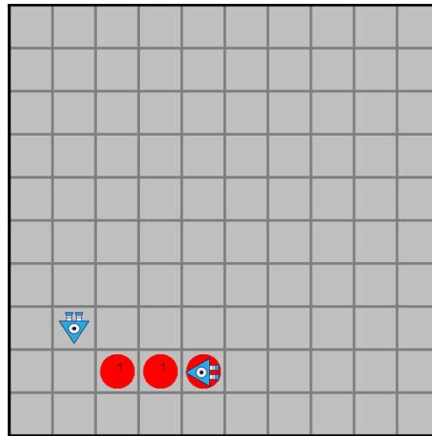
Vorwärtsschritt



`myRobot2.move();`

Der nächste Vorwärtsschritt ist also eigentlich ein Schritt zurück, wenn man so will.

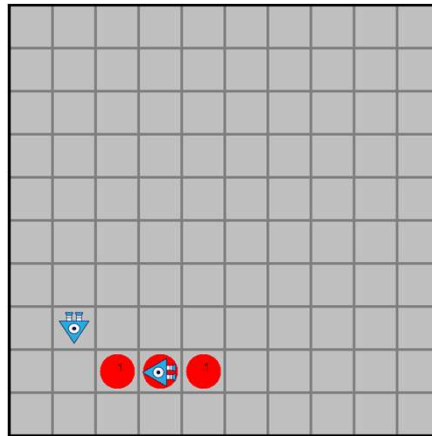
Münze aufheben



```
myRobot2.pickCoin();
```

Eine neue Methode: pickCoin, ebenfalls ohne Parameter. Der Robot myRobot2 nimmt eine der beiden Münzen auf dieser Position wieder auf, hat jetzt also wieder zwei Münzen.

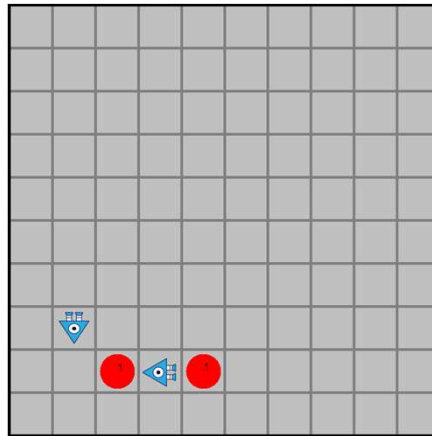
Vorwärtsschritt



```
myRobot2.move();
```

Und durch diesen Vorwärtsschritt sehen wir, dass nicht mehr *zwei* Münzen auf der letzten Position von myRobot2 sind, sondern nur noch *eine*. Auf der jetzigen Position von myRobot2 war vorher und ist weiterhin nur *eine* Münze, hier hat sich nichts geändert.

Münze aufheben



```
myRobot2.pickCoin();
```

Nach einem weiteren Aufruf der Methode `pickCoin` ist an der momentanen Position *keine* Münze mehr, und `myRobot2` hat jetzt *drei* Münzen.

Allgemein: Programmablauf

Bevor es mit den Robotern weitergeht, müssen wir uns noch ein paar grundsätzliche Gedanken machen, was es eigentlich bedeutet, wenn ein Programm aufgerufen wird und abläuft.

Programme und Prozesse



- Unter *Programm* versteht man dummerweise zwei verschiedene Texte: Quelltext und übersetzter Code
- Quelltext: Was wir auf den letzten Folien in Java geschrieben haben bzw. was wir übernommen haben
- Java-Quelltext wird in Java Bytecode übersetzt
- Die Ausführung des übersetzten Programms nennt man Prozess
 - Mehrere Prozesse zeitgleich auf demselben Programm möglich

Zuerst eine wichtige begriffliche Unterscheidung, nämlich die zwischen einem *Programm* und einem *Prozess*.

Programme und Prozesse



- Unter *Programm* versteht man dummerweise zwei verschiedene Texte: Quelltext und übersetzter Code
- Quelltext: Was wir auf den letzten Folien in Java geschrieben haben bzw. was wir übernommen haben
- Java-Quelltext wird in Java Bytecode übersetzt
- Die Ausführung des übersetzten Programms nennt man Prozess
 - Mehrere Prozesse zeitgleich auf demselben Programm möglich

Allerdings wird das Wort *Programm* durchaus problematisch verwendet. Zumindest überall da, wo es missverständlich sein könnte, ist es besser, konkret vom Quelltext beziehungsweise von der Übersetzung des Quelltextes zu sprechen.

Programme und Prozesse



- Unter *Programm* versteht man dummerweise zwei verschiedene Texte: Quelltext und übersetzter Code
- Quelltext: Was wir auf den letzten Folien in Java geschrieben haben bzw. was wir übernommen haben
- Java-Quelltext wird in Java Bytecode übersetzt
- Die Ausführung des übersetzten Programms nennt man Prozess
 - Mehrere Prozesse zeitgleich auf demselben Programm möglich

Aber egal ob Quelltext oder Übersetzung: Das Programm ist einfach eine Sequenz von Informationen. Aufruf eines Programms bedeutet, dass auf dem Computer ein Prozess gestartet wird, der die Anweisungen aus diesem Programm abarbeitet.

Programme und Prozesse



- Unter *Programm* versteht man dummerweise zwei verschiedene Texte: Quelltext und übersetzter Code
- Quelltext: Was wir auf den letzten Folien in Java geschrieben haben bzw. was wir übernommen haben
- Java-Quelltext wird in Java Bytecode übersetzt
- Die Ausführung des übersetzten Programms nennt man Prozess
 - Mehrere Prozesse zeitgleich auf demselben Programm möglich

Ein Programm kann man auch mehrmals starten, das ergibt mehrere Prozesse, die völlig unabhängig voneinander laufen.

Programme und Prozesse



- Ein Computer enthält einen oder mehrere Prozessorkerne
- Mehrere Prozesse können auf einem Computer gleichzeitig laufen
 - Webbrowser, Emailprogramm, Kalendertool usw.
- Tatsächlich läuft auf jedem Prozessorkern zu jeder Zeit nur ein Prozess
 - Die anderen Prozesse warten derweil
 - Reihum bekommt jeder Prozess einen Prozessorkern für sehr kurze Zeit zugeteilt
 - Illusion von Multitasking

Wir schauen noch ganz kurz und oberflächlich, was passiert, wenn Prozesse auf dem Computer abgearbeitet werden.

Programme und Prozesse



- Ein Computer enthält einen oder mehrere Prozessorkerne
- Mehrere Prozesse können auf einem Computer gleichzeitig laufen
 - Webbrowser, Emailprogramm, Kalendertool usw.
- Tatsächlich läuft auf jedem Prozessorkern zu jeder Zeit nur ein Prozess
 - Die anderen Prozesse warten derweil
 - Reihum bekommt jeder Prozess einen Prozessorkern für sehr kurze Zeit zugeteilt
 - Illusion von Multitasking

Das Herzstück eines Computers ist die Zentralprozessoreinheit oder englisch Central Processing Unit, kurz CPU. Moderne CPUs enthalten in der Regel mehrere Prozessorkerne.

Programme und Prozesse



- Ein Computer enthält einen oder mehrere Prozessorkerne
- Mehrere Prozesse können auf einem Computer gleichzeitig laufen
 - Webbrowser, Emailprogramm, Kalendertool usw.
- Tatsächlich läuft auf jedem Prozessorkern zu jeder Zeit nur ein Prozess
 - Die anderen Prozesse warten derweil
 - Reihum bekommt jeder Prozess einen Prozessorkern für sehr kurze Zeit zugeteilt
 - Illusion von Multitasking

Sie sehen ja selbst in Ihrer täglichen Arbeit mit dem Computer, dass mehrere Prozesse parallel laufen. Jedes Betriebssystem bietet eine Möglichkeit, sich die momentan laufenden Prozesse auflisten zu lassen.

Programme und Prozesse



- Ein Computer enthält einen oder mehrere Prozessorkerne
- Mehrere Prozesse können auf einem Computer gleichzeitig laufen
 - Webbrowser, Emailprogramm, Kalendertool usw.
- Tatsächlich läuft auf jedem Prozessorkern zu jeder Zeit nur ein Prozess
 - Die anderen Prozesse warten derweil
 - Reihum bekommt jeder Prozess einen Prozessorkern für sehr kurze Zeit zugeteilt
 - Illusion von Multitasking

Allerdings ist diese Parallelität zum Großteil nur vorgegaukelt, denn mehr als einen Prozess kann ein Prozessorkern zu keinem Zeitpunkt abarbeiten.

Programme und Prozesse



- Ein Computer enthält einen oder mehrere Prozessorkerne
- Mehrere Prozesse können auf einem Computer gleichzeitig laufen
 - Webbrowser, Emailprogramm, Kalendertool usw.
- Tatsächlich läuft auf jedem Prozessorkern zu jeder Zeit nur ein Prozess
 - Die anderen Prozesse warten derweil
 - Reihum bekommt jeder Prozess einen Prozessorkern für sehr kurze Zeit zugeteilt
 - Illusion von Multitasking

Deshalb gibt es einfach gesprochen eine Warteschlange. Alle Prozesse warten darin, bis sie drankommen, abgesehen von denen, die gerade auf einem der Prozessorkerne eine kurze Zeit lang ein Stück weiter abgearbeitet werden. Nach dieser kurzen Zeit wird jeder Prozess wieder in die Warteschlange eingereiht.

Programme und Prozesse



- Ein Computer enthält einen oder mehrere Prozessorkerne
- Mehrere Prozesse können auf einem Computer gleichzeitig laufen
 - Webbrowser, Emailprogramm, Kalendertool usw.
- Tatsächlich läuft auf jedem Prozessorkern zu jeder Zeit nur ein Prozess
 - Die anderen Prozesse warten derweil
 - Reihum bekommt jeder Prozess einen Prozessorkern für sehr kurze Zeit zugeteilt
 - Illusion von Multitasking

Dadurch, dass die Prozesse auf den Prozessorkernen so schnell wechseln, sieht es für den Langsammerker Homo Sapiens so aus, als würden alle Prozesse tatsächlich parallel laufen. Dass dem nicht so ist, davon bekommen Sie einen Eindruck, wenn die CPU durch zu viele Prozesse überlastet ist und bei den einzelnen Prozessen nicht mehr hinterherkommt.

Anweisungen abarbeiten



- Jede Anweisung in Java wird in eine Folge von elementaren Anweisungen in Java Bytecode übersetzt
- Ohne Schleifen o.ä. wird stupide eine Anweisung nach der anderen ausgeführt
 - In der Reihenfolge, wie sie in der Java-Datei vorkommen

```
myRobot2.putCoin();  
myRobot2.move();  
myRobot2.turnLeft();  
myRobot2.move();  
myRobot2.pickCoin();  
myRobot2.move();
```

Bisher haben wir uns die Abarbeitung von Prozessen als Ganzes angeschaut, nun gehen wir auf einzelne Anweisungen herunter.

Anweisungen abarbeiten



- Jede Anweisung in Java wird in eine Folge von elementaren Anweisungen in Java Bytecode übersetzt
- Ohne Schleifen o.ä. wird stupide eine Anweisung nach der anderen ausgeführt
 - In der Reihenfolge, wie sie in der Java-Datei vorkommen

```
myRobot2.putCoin();  
myRobot2.move();  
myRobot2.turnLeft();  
myRobot2.move();  
myRobot2.pickCoin();  
myRobot2.move();
```

Der Quelltext, den Sie in Java schreiben, wird in eine Sprache übersetzt, die Java Bytecode heißt. Anweisungen in dieser Sprache sind einfacher und elementarer als in Java, so dass jede einzelne Java-Anweisung nicht in eine einzelne Anweisung in Java Bytecode übersetzt wird, sondern in eine Abfolge von Anweisungen. Selbstverständlich soll der übersetzte Quelltext exakt das tun, was Sie im Quelltext festgelegt haben. Mit anderen Worten: Der übersetzte Quelltext soll dasselbe bedeuten wie der Quelltext selbst, was durch das Wort „Übersetzen“ ausgedrückt wird.

Anweisungen abarbeiten



- Jede Anweisung in Java wird in eine Folge von elementaren Anweisungen in Java Bytecode übersetzt
- Ohne Schleifen o.ä. wird **stupid** eine Anweisung nach der anderen ausgeführt
 - In der Reihenfolge, wie sie in der Java-Datei vorkommen

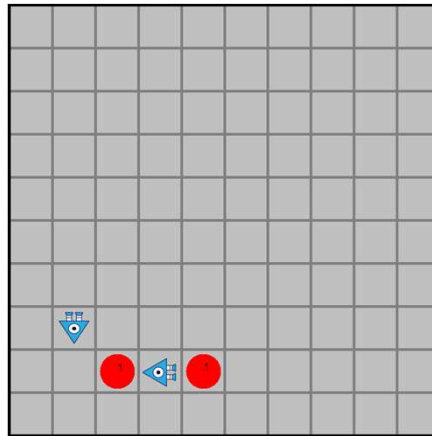
```
myRobot2.putCoin();  
myRobot2.move();  
myRobot2.turnLeft();  
myRobot2.move();  
myRobot2.pickCoin();  
myRobot2.move();
```

Schleifen werden wir gleich erstmals sehen; ohne sie geht der Prozess einfach durch alle Anweisungen von oben nach unten durch.

Weiter: Erste Schritte in Java mit FopBot

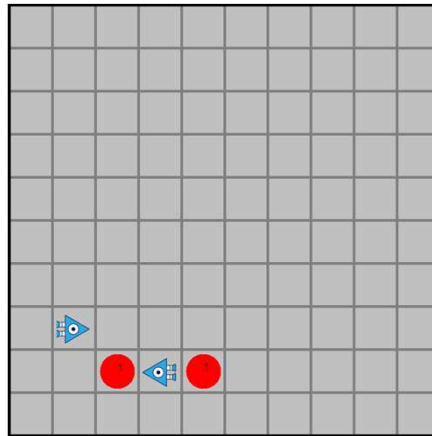
Nach diesen allgemeinen Überlegungen machen wir weiter mit FopBot.

Münze aufheben



Das ist die Situation, wie wir sie eben verlassen hatten.

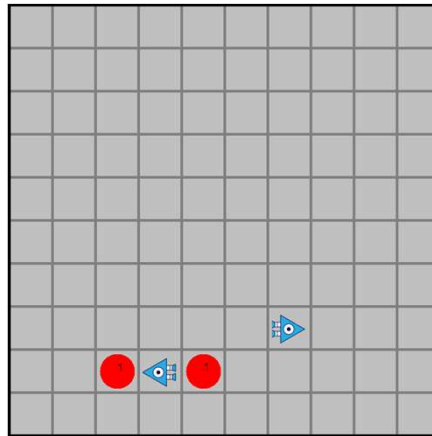
Linksdrehung



```
myRobot.turnLeft();
```

Zuerst lassen wir myRobot noch eine Linksdrehung machen.

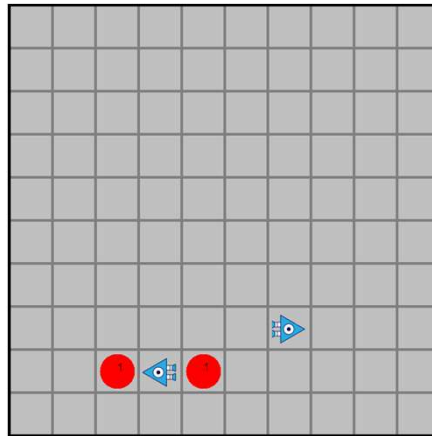
Vorwärtsschritte in Schleife



```
for ( int i = 0; i < 5; i++ ) { myRobot.move(); }
```

Jetzt etwas völlig Neues, fünf Vorwärtsschritte durch eine einzige Anweisung, eine *Schleife*, genauer: eine *for*-Schleife.

Vorwärtsschritte in Schleife

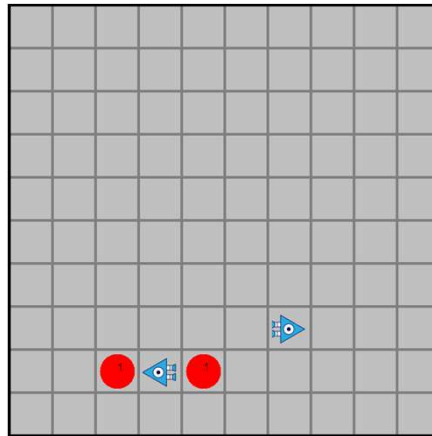


```
for ( int i = 0; i < 5; i++ ) { myRobot.move(); }
```

Wir werden später in diesem Kapitel genauer betrachten, warum der Kopf einer for-Schleife so kompliziert aufgebaut ist.

Hier reicht erst einmal ein erstes Grundverständnis: Mit dem farblich unterlegten Baustein wird dafür gesorgt, dass die nachfolgende Anweisung mehrfach ausgeführt wird.

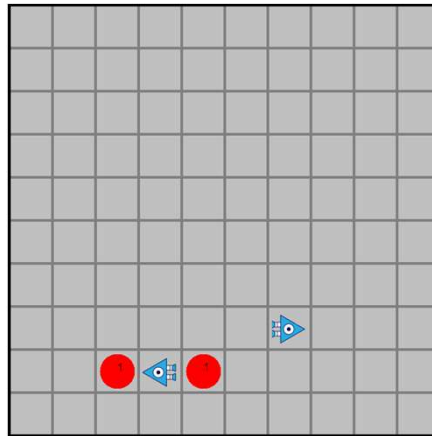
Vorwärtsschritte in Schleife



```
for ( int i = 0; i < 5; i++ ) { myRobot.move(); }
```

Und an dieser Stelle steht, wie oft die Anweisung ausgeführt werden soll, nämlich fünfmal. Alles andere in den runden Klammern ist bis auf weiteres immer identisch. Wir nehmen es erst einmal so hin, wie es da steht, ohne uns darum zu kümmern, was das alles bedeuten soll und warum es so kompliziert ist.

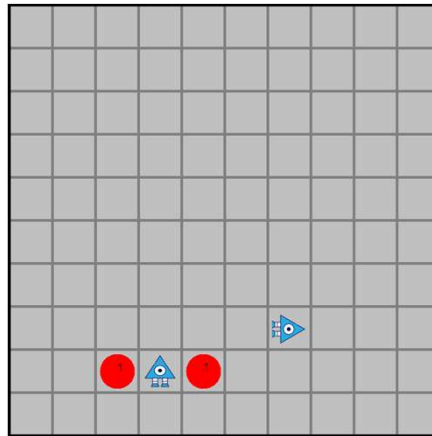
Vorwärtsschritte in Schleife



```
for ( int i = 0; i < 5; i++ ) { myRobot.move(); }
```

Das, was fünfmal auszuführen ist, ist in geschweifte Klammern gepackt. Bei einer einzelnen Anweisung wie hier dem move könnte man die geschweiften Klammern auch weglassen. Wir schreiben sie trotzdem hin nach dem Motto: erst die allgemeine Regel systematisch klären und einüben, danach irgendwann später auch die Ausnahmen betrachten.

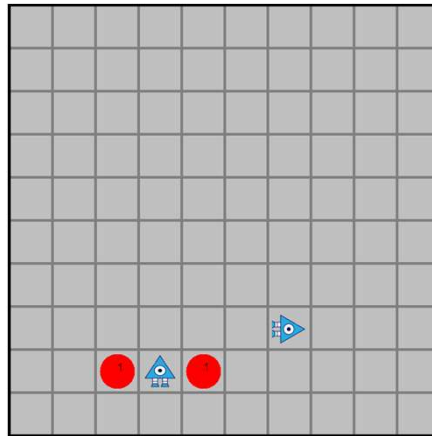
Linksdrehungen in Schleife



```
for ( int i = 0; i < 3; i++ ) { myRobot2.turnLeft(); }
```

Eine zweite for-Schleife. Dreimal hat myRobot2 sich durch diese Schleife um 90 Grad nach links gedreht. Vorher hat myRobot2 nach links gezeigt, jetzt zeigt myRobot2 also nach oben.

Linksdrehungen in Schleife



```
for ( int i = 0; i < 3; i++ ) { myRobot2.turnLeft(); }
```

Der wesentliche Unterschied zur ersten Schleife ist: Wo vorher eine 5 stand, steht jetzt eine 3, also *dreimal* wird turnLeft aufgerufen.

Abgekürzte Notation

```
for ( int i = 0; i < 3; i++ ) { myRobot2.turnLeft(); }
```

```
for ( ... 3 ... ) { myRobot2.turnLeft(); }
```

Wir erlauben uns, for-Schleifen auf den nächsten Folien abgekürzt zu schreiben, so wie in der zweiten Zeile. Die ganzen Details in der ersten Zeile, die in der zweiten Zeile ausgelassen werden, sind bis auf weiteres immer identisch. Daher können wir sie auf den Folien auslassen. In den Java-Programmen muss es natürlich immer so wie in der ersten Zeile aussehen.

While-Schleife

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

Eine andere Art von Schleife, die while-Schleife. Mit der while-Schleife führt man Anweisungen solange aus, bis eine bestimmte Bedingung nicht mehr erfüllt ist.

While-Schleife

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

Diese Bedingung steht immer gleich hinter dem Schlüsselwort while in runden Klammern. In diesem Fall der Aufruf einer Methode von Klasse Robot, die einen Wahrheitswert zurückliefert, also wahr oder falsch, englisch true oder false.

Methode hasAnyCoins liefert genau dann true zurück, wenn der Roboter, mit dem die Methode aufgerufen wird, im Moment des Aufrufs mindestens eine Münze hat. Die Schleife ist also zu Ende genau in dem Moment, wenn myRobot2 keine Münzen mehr hat.

While-Schleife

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

In jedem Durchlauf soll dasselbe passieren: Eine Münze wird an der momentanen Position platziert, danach geht myRobot2 einen Schritt vorwärts. Da myRobot2 momentan drei Münzen hat und nach oben zeigt, können wir also vorhersagen, dass myRobot2 dreimal eine Münze ablegen und danach jedes Mal einen Schritt nach oben gehen wird.

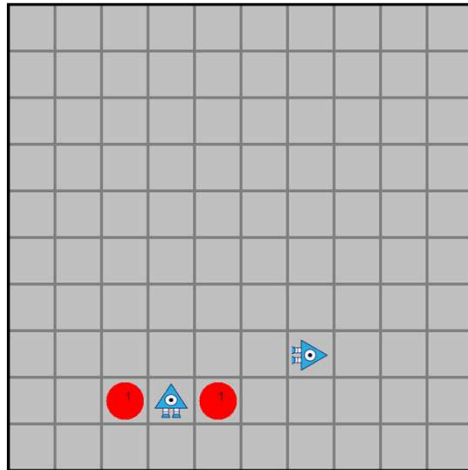
While-Schleife

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

In dieser Schleife sollen also *zwei* Anweisungen wiederholt ausgeführt werden.

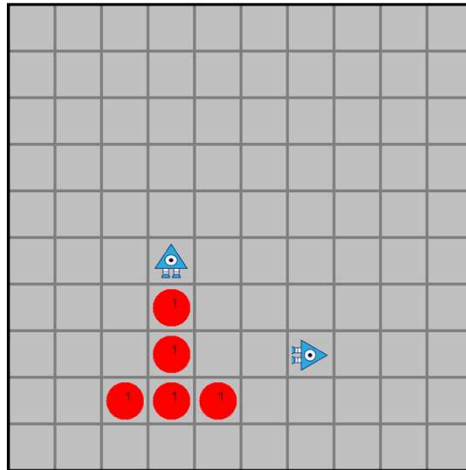
Wir hatten vorher gesagt, dass man bei *einer* Anweisung die geschweiften Klammern auch weglassen dürfte. Bei *mehr* als einer Anweisung, so wie hier, haben wir keine Wahl, wir können die geschweiften Klammern, die die Anweisungen zu einem Anweisungsblock zusammenfassen, *nicht* weglassen.

While-Schleife



Hier ist noch einmal der Ausgangszustand, *bevor* die while-Schleife ausgeführt wird.

While-Schleife



Und das ist das Ergebnis *nach* Ausführung der while-Schleife. Dreimal hatte myRobot2 noch mindestens eine Münze, das heißt, dreimal liefert die Fortsetzungsbedingung true zurück, so dass myRobot2 dreimal eine Münze ablegt und dann jeweils einen Schritt vorwärtsgeht. Beim vierten Mal hat myRobot2 keine Münzen mehr, daher liefert die Methode hasAnyCoins im vierten Schritt false zurück, und die Schleife bricht ab, bevor die beiden Anweisungen im Schleifenrumpf ein viertes Mal ausgeführt werden können.

Das ist so ja auch sinnvoll, denn wenn myRobot2 keine Münzen mehr *hat*, kann auch keine Münze mehr *abgelegt* werden.

Das Gesamtprogramm für die vorangegangenen ersten Schritte

Diese Anweisungen, die wir auf den ersten Folien gesehen haben, bilden für sich genommen noch kein vollständiges Java-Programm. Wir schauen uns jetzt kurz den Rahmen an, den wir um die Anweisungen schreiben müssen, um aus ihnen ein vollständiges Java-Programm zu machen.

Das Gesamtprogramm



```
import fopbot.*;
import static Direction.*;

public class FirstSteps {
    public static void main ( String [ ] args ) {
        World.setVisible ( true );
```

```
        .....
```

```
        .....
```

```
        .....
```

```
    }
```

```
}
```

Alles, was jetzt farbig unterlegt ist, ist dieser notwendige Rahmen für unsere Anweisungen.

Die Details dieses Rahmens werden Ihnen in den Übungsaufgaben zu FopBot vorgegeben sein, Sie müssen sich bei der Bearbeitung der Übungsaufgaben bis auf weiteres überhaupt nicht um diesen Rahmen kümmern. Ignorieren Sie ihn einfach.

In späteren Kapiteln werden wir die Bestandteile des Rahmens systematisch durchgehen und dann auch richtig verstehen.

Das Gesamtprogramm



```
import fopbot.*;
import static Direction.*;

public class FirstSteps {
    public static void main ( String [ ] args ) {
        World.setVisible ( true );
        .....
        .....
        .....
    }
}
```

Nur ein einziges Detail in diesem Rahmen ist für uns jetzt schon wichtig: Hier, unmittelbar nach dem Wort „class“ für Englisch „Klasse“, steht das Wort, das Sie auch im Dateinamen finden.

Das Gesamtprogramm, das wir auf dieser und den folgenden Folien schrittweise zusammenstellen, finden Sie also in den Übungsunterlagen in einer Datei namens FirstSteps.java: FirstSteps in genau dieser Schreibweise mit genau dieser Groß- und Kleinschreibung. Bei der Dateiendung java sind immer alle Buchstaben klein.

Das Gesamtprogramm



```
import fopbot.*;
import static Direction.*;

public class FirstSteps {
    public static void main ( String [ ] args ) {
        World.setVisible ( true );
        .....
        .....
        .....
    }
}
```

Dies ist die Stelle, in die die Anweisungen zu schreiben sind. Diese Stelle wird in den Übungsaufgaben zu FopBot von Ihnen zu füllen sein.

Die Punkte hier gehören nicht zur Programmiersprache, sondern zeigen nur auf der Folie an, dass wir an dieser Stelle die ganzen Anweisungen einfügen werden. Eine einzelne Folie reicht halt nicht, um das ganze Programm in vernünftiger Schriftgröße zu zeigen.

Das Gesamtprogramm



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );  
myRobot.move();  
myRobot.move();  
myRobot.turnLeft();  
myRobot.move();  
myRobot.move();  
myRobot.turnLeft();  
.....
```

Das sind die ersten sieben Anweisungen, die wir eingangs betrachtet hatten: Einrichtung von Roboter myRobot und ein paar Methodenaufrufe mit myRobot.

Das Gesamtprogramm



```
Robot myRobot = new Robot ( 3, 1, UP, 0 );  
myRobot.move();  
myRobot.move();  
myRobot.turnLeft();  
myRobot.move();  
myRobot.move();  
myRobot.turnLeft();
```

```
.....
```

Wir haben eben gesehen, dass wir Auslassungen im Programm durch zehn Punkte wie hier darstellen. Aus Platzgründen kommt das, was wir hier ausgelassen haben, erst auf den nächsten Folien.

Das Gesamtprogramm



.....

```
myRobot.move();  
Robot myRobot2 = new Robot ( 2, 1, RIGHT, 5 );  
myRobot2.putCoin();  
myRobot2.move();  
myRobot2.putCoin();  
myRobot2.move();  
myRobot2.putCoin();
```

.....

**Und die nächsten sieben Anweisungen, darunter auch die
Einrichtung von myRobot2.**

Das Gesamtprogramm



```
.....  
myRobot2.putCoin();  
myRobot2.move();  
myRobot2.turnLeft();  
myRobot2.turnLeft();  
myRobot2.move();  
myRobot2.pickCoin();  
myRobot2.move();  
.....
```

Die nächsten sieben Anweisungen danach.

Das Gesamtprogramm



```
.....  
myRobot2.pickCoin();  
myRobot2.turnLeft();  
for ( int i = 0; i < 5; i++ ) {  
    myRobot2.move();  
}  
.....
```

Nach noch zwei Anweisungen kommen wir schon zur ersten Schleife, die wir hatten ausführen lassen.

Das Gesamtprogramm

```
.....  
for ( int i = 0; i < 3; i++ ) {  
    myRobot2.turnLeft();  
}  
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

Und die weiteren beiden Schleifen.

Das waren alle Anweisungen, die wir betrachtet haben, und damit sind alle diese Anweisungen im Gesamtquelltext eingefügt.

Ein einfaches FopBot-Programm zur Demonstration von arithmetischen Operationen

Java bietet eine ganze Palette von arithmetischen Operationen, die Sie auch aus der Schulmathematik kennen.

Arithmetische Operationen



```
import fopbot.*;
import static Direction.*;

public class ArithmeticOperations {
    public static void main(String[] args) {
        World.setVisible ( true );
        Robot ari = new Robot ( 0, 0, RIGHT, 357 );

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

Das hier besprochene Beispielpogramm ist in der Datei `ArithmeticOperations.java` zu finden.

Arithmetische Operationen



```
import fopbot.*;
import static Direction.*;

public class ArithmeticOperations {
    public static void main(String[] args) {
        World.setVisible ( true );
        Robot ari = new Robot ( 0, 0, RIGHT, 357 );
        .....
        .....
        .....
    }
}
```

Wir nennen den Roboter jetzt ari für Arithmetik und geben ihm so viele Münzen mit, dass er auf jeden Fall ausreichend viele für die Demonstration haben wird.

Arithmetische Operationen



```
for ( int i = 0; i < 3 - 5 + 6; i++ ) { ..... }  
for ( ... 3 - 5 + 6 ... ) { ..... }  
for ( int i = 0; i < 2 * ( 1 + 3 ); i++ ) { ..... }  
for ( ... 2 * ( 1 + 3 ) ... ) { ..... }  
for ( int i = 0; i < 2 + 7 / 3; i++ ) { ..... }  
for ( ... 2 + 7 / 3 ... ) { ..... }  
for ( int i = 0; i < ( 15 - 4 ) % 3 ... ) { ..... }  
for ( ... i < ( 15 - 4 ) % 3 ... ) { ..... }
```

Sie sehen vier verschiedene arithmetische Ausdrücke farblich unterlegt, jeweils mit einer voll ausgeschriebenen for-Schleife und mit derselben for-Schleife in der verkürzten Schreibweise, die wir hier auf diesen Folien verwenden.

Wir werden diese arithmetischen Ausdrücke nacheinander verwenden, um die Anzahl Vorwärtsschritte des Roboters zu spezifizieren. Die for-Schleifen werden wir wieder so abkürzen wie schon besprochen.

Am Ende sehen wir uns an, was insgesamt herausgekommen ist.

Arithmetische Operationen



```
for ( ... 3 - 5 + 6 ... ) {  
    ari.putCoin();  
    ari.move();  
}
```

Wir legen wieder in jedem Schleifendurchlauf eine Münze ab und gehen dann einen Schritt vorwärts.

Die Auswertung des arithmetischen Ausdrucks ist analog zur Schulmathematik: Von der 3 werden zuerst 5 abgezogen, und auf dieses Zwischenergebnis werden dann noch 6 addiert.

Das Endergebnis ist also 4, die Schleife wird viermal durchlaufen.

Arithmetische Operationen



```
for ( ... 3 – 5 + 6 ... ) {  
    ari.putCoin();  
    ari.move();  
}  
ari.turnLeft();
```

Um die Ergebnisse der einzelnen Schleifen bildhaft auseinanderzuhalten, machen wir nach jeder Schleife eine Biege mit Methode turnLeft. Das werden wir auch im Folgenden zwischen je zwei Schleifen so machen, ohne das noch einmal zu erwähnen.

Arithmetische Operationen



```
for ( ... 3 - 5 + 6 ... ) {  
    ari.putCoin();  
    ari.move();  
}  
ari.turnLeft();  
for ( ... 2 * ( 1 + 3 ) ... ) {  
    ari.putCoin();  
    ari.move();  
}
```

Wie in der Schulmathematik, gilt auch in Java Punkt- vor Strichrechnung. Will man die Auswertungsreihenfolge ändern, muss man Klammern setzen.

Hier werden also zuerst 1 und 3 addiert, das Zwischenergebnis 4 wird mit 2 multipliziert, das Ergebnis ist 8, also acht Schleifendurchläufe.

Arithmetische Operationen



```
for ( ... 3 - 5 + 6 ... ) {  
    ari.putCoin();  
    ari.move();  
}  
ari.turnLeft();  
for ( ... 2 * ( 1 + 3 ) ... ) {  
    ari.putCoin();  
    ari.move();  
}  
ari.turnLeft();
```

```
for ( ... 2 + 7 / 3 ... ) {  
    ari.putCoin();  
    ari.move();  
}
```

Hier kommt jetzt Punkt- vor Strichrechnung zum Tragen: Die Division 7 durch 3 wird vor der Addition ausgeführt. Sind so wie hier sowohl der Dividend als auch der Divisor von ganzzahligem Typ, dann wird ganzzahlige Division mit Rest durchgeführt. Das Ergebnis der Division ist also nicht 2 Komma 3 Periode, sondern nur 2. Die Nachkommastellen werden einfach abgeschnitten.

Das Endergebnis nach der Addition ist 4, also vier Schleifendurchläufe.

Arithmetische Operationen



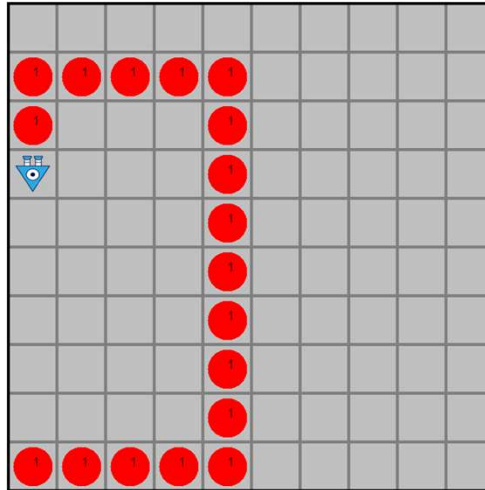
<pre>for (... 3 - 5 + 6 ...) { ari.putCoin(); ari.move(); } ari.turnLeft(); for (... 2 * (1 + 3) ...) { ari.putCoin(); ari.move(); } ari.turnLeft();</pre>	<pre>for (... 2 + 7 / 3 ...) { ari.putCoin(); ari.move(); } ari.turnLeft(); for (... (15 - 4) % 3 ...) { ari.putCoin(); ari.move(); }</pre>
--	---

Durch die Klammerung wird die Regel „Punkt- vor Strichrechnung“ durchbrochen, also wieder zuerst 4 von 15 abgezogen, Zwischenergebnis 11.

Das Prozentzeichen als arithmetischer Operator kommt so in der Regel nicht in der Schulmathematik vor. Es wird „modulo“ ausgesprochen und liefert den Rest bei ganzzahliger Division.

Konkret hier wird der Rest von 11 durch 3 berechnet. Die 3 geht dreimal in die 11, und als Rest bleibt 2. Das Endergebnis ist also 2, zwei Schleifendurchläufe.

Arithmetische Operationen



Da wir den Roboter ari nach jeder Schleife um 90 Grad gedreht haben, können wir die Endergebnisse der vier arithmetischen Operationen direkt vom Ergebnisbild ablesen: 4, 8, 4 und 2.

Anweisungen abarbeiten

Oracle Java Tutorials: Control Flow Statements

Bei den allgemeinen Betrachtungen zum Programmablauf weiter vorne hatten wir noch nicht Schleifen einbezogen. Das holen wir jetzt nach.

Anweisungen abarbeiten

Oracle Java Tutorials: Control Flow Statements

Auf den Titelfolien einzelner Abschnitte werden Sie hin und wieder Verweise auf Dokumentationen finden. Die Links zu den Einstiegsseiten der verschiedenen Dokumentationsquellen finden Sie in moodle.

Anweisungen abarbeiten



Bei einer while-Schleife wird wiederholt ausgeführt:

- zuerst der Test der Fortsetzungsbedingung
- dann die Anweisungen im Rumpf

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

**Schleifen steuern die Abarbeitungsreihenfolge in spezifischer Weise.
Wir betrachten als erstes die while-Schleife.**

Anweisungen abarbeiten



Bei einer while-Schleife wird wiederholt ausgeführt:

- zuerst der Test der Fortsetzungsbedingung
- dann die Anweisungen im Rumpf

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

Der Sinn und Zweck von Schleifen ist, dass dieselben Anweisungen mehr als einmal ausgeführt werden.

Anweisungen abarbeiten

Bei einer while-Schleife wird wiederholt ausgeführt:

- zuerst der Test der Fortsetzungsbedingung
- dann die Anweisungen im Rumpf

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

Bei der while-Schleife steht hinter dem Schlüsselwort while in runden Klammern die Fortsetzungsbedingung. Das ist ein Ausdruck, der einen binären Wahrheitswert zurückliefert, also wahr oder falsch, englisch true oder false.

Anweisungen abarbeiten



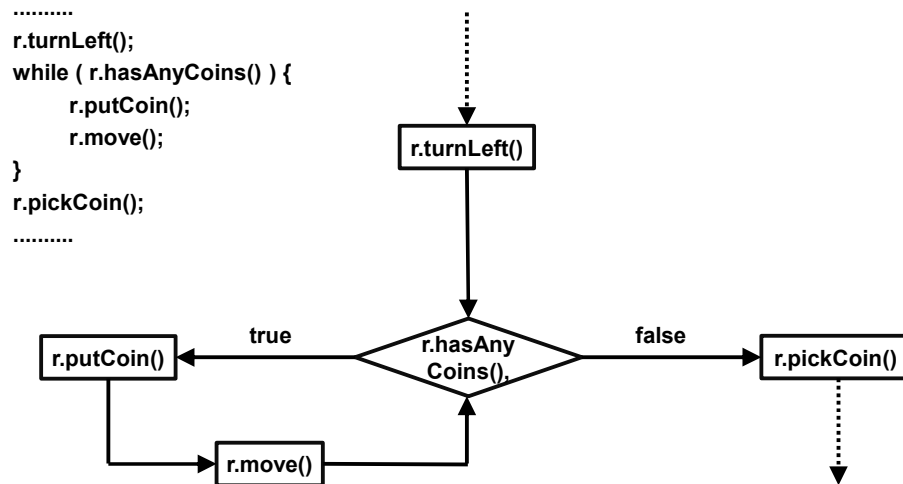
Bei einer while-Schleife wird wiederholt ausgeführt:

- zuerst der Test der Fortsetzungsbedingung
- dann die Anweisungen im Rumpf

```
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.putCoin();  
    myRobot2.move();  
}
```

Falls die Fortsetzungsbedingung true ist, werden die Anweisungen in den geschweiften Klammern einmal abgearbeitet. Diese Anweisungen bilden den *Rumpf* der Schleife. Danach wird wieder die Fortsetzungsbedingung ausgewertet.

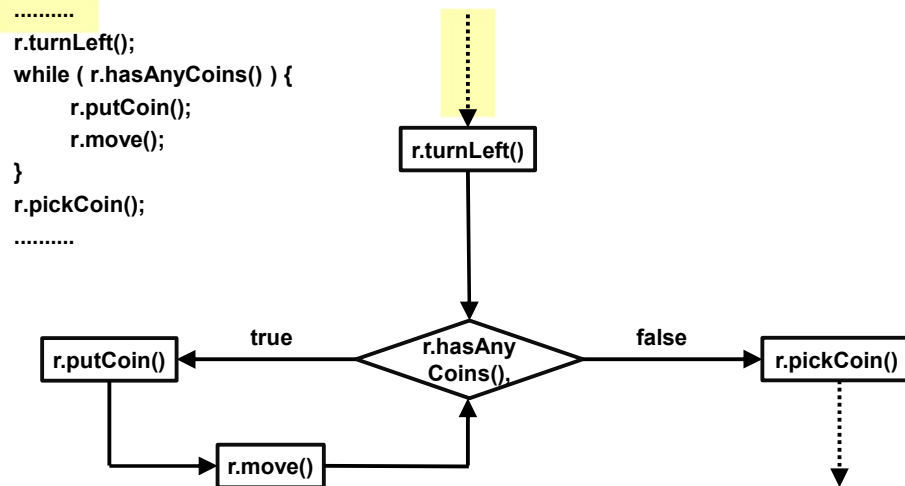
Anweisungen abarbeiten



Hier sehen Sie eine visuelle Darstellung einer solchen Schleife. Um die Darstellung übersichtlich zu halten, wird hier einfach der Kleinbuchstabe `r` als Name des Roboters gewählt, nicht `myRobot2` wie auf der letzten Folie.

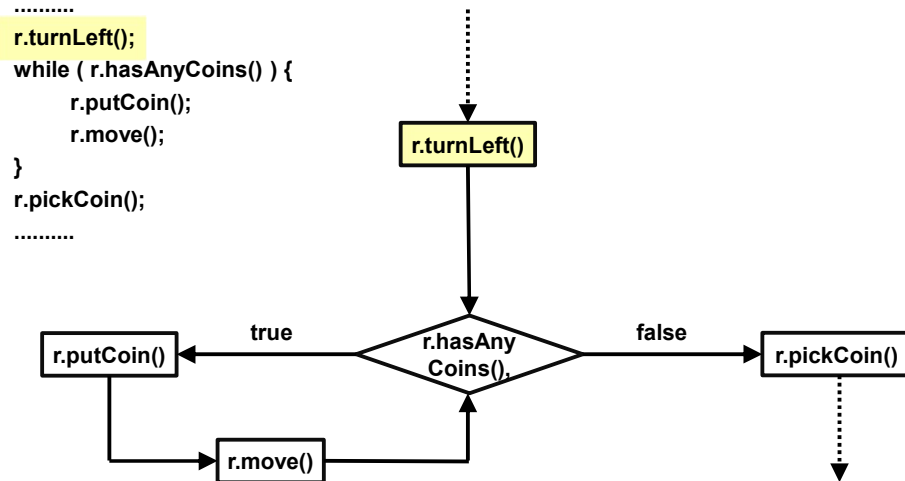
Eine solche Darstellung nennt man ein *Flussdiagramm*: Der Prozess fließt sozusagen durch die Anweisungen, und ein Pfeil bedeutet einen möglichen Schritt von der Ausführung einer Anweisung zu einer unmittelbar danach ausgeführten Anweisung.

Anweisungen abarbeiten



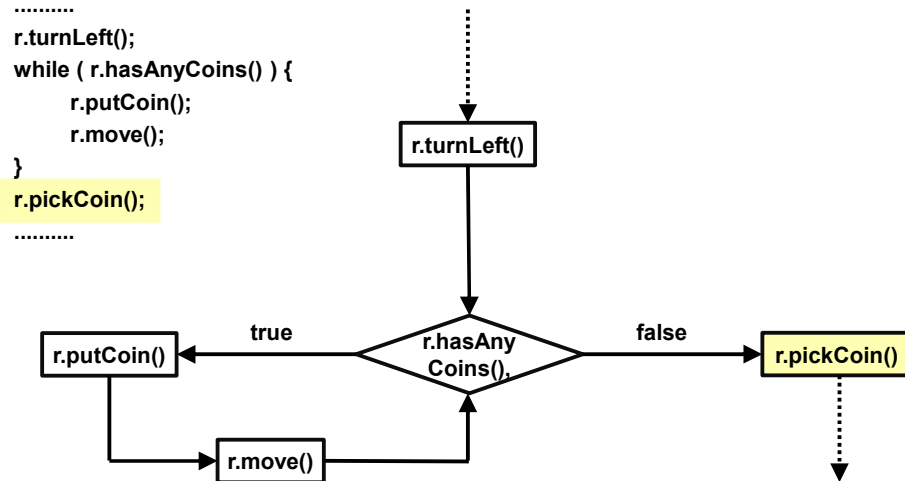
Irgendwelche andere Anweisungen, die vor diesem kleinen Ausschnitt des Gesamtprogramms stehen, werden hier ausgeblendet und nur durch Punkte angedeutet.

Anweisungen abarbeiten



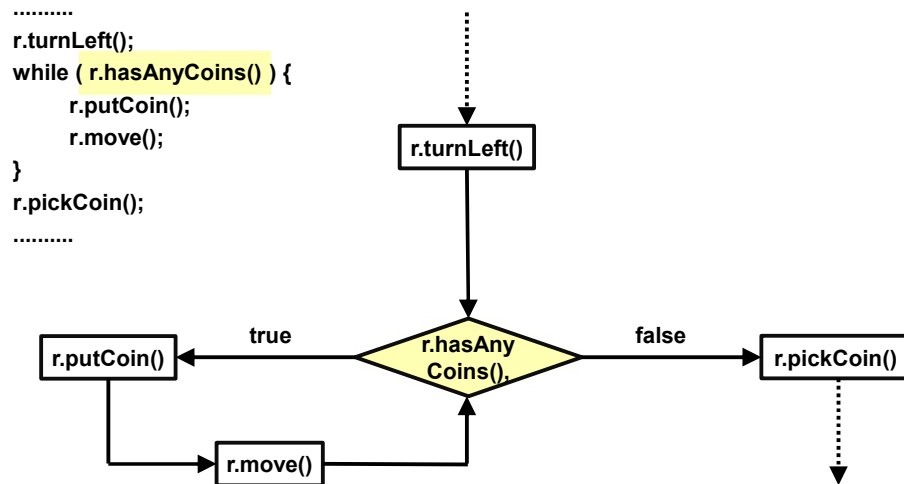
Um die Schleife inklusive ihrem Kontext zu verstehen, nehmen wir aber die unmittelbar der Schleife vorangehende Anweisung in das Bild auf ...

Anweisungen abarbeiten



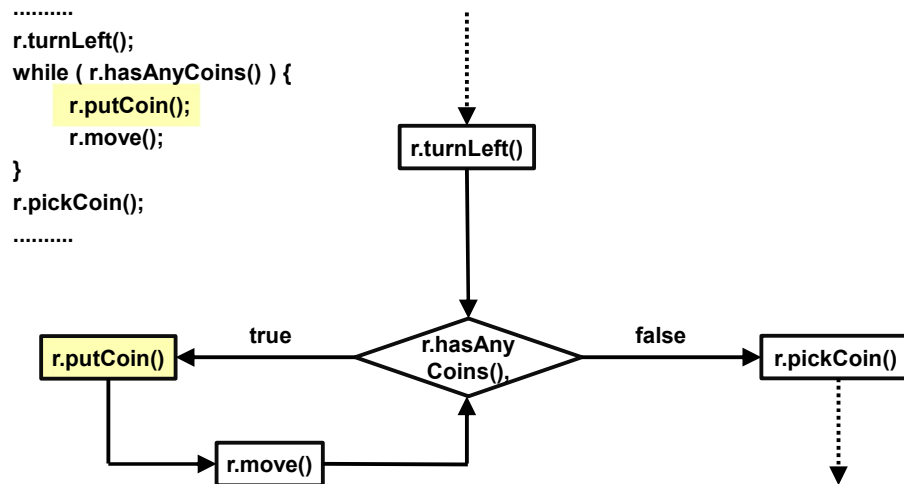
... und auch die unmittelbar nachfolgende.

Anweisungen abarbeiten



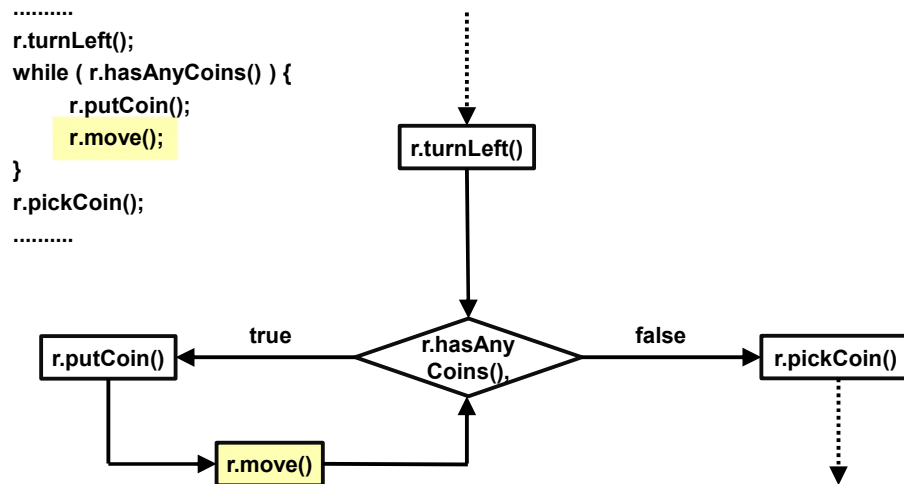
Der Test, ob eine Bedingung true oder false ergibt, wird typischerweise so wie hier in Flussdiagrammen dargestellt. Je nachdem, ob true oder false herauskommt, geht es dann mit unterschiedlichen Anweisungen weiter, daher zwei herausgehende Pfeile, die zweckmäßigerweise mit true und false annotiert sind. Jedes Mal, wenn der Prozess an die farblich markierte Stelle im Flussdiagramm kommt, wird also getestet, ob der Roboter noch Münzen hat. Falls ja, geht es mit der ersten Anweisung im Schleifenrumpf weiter. Falls nein, ist die Schleife beendet, das heißt, es geht mit der Anweisung unmittelbar nach der Schleife weiter.

Anweisungen abarbeiten



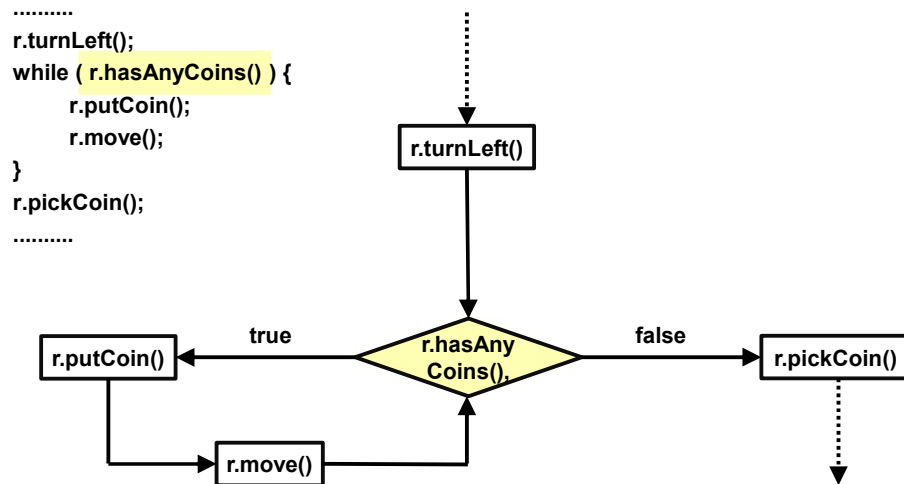
Falls der Roboter noch Münzen hat, geht es wie gesagt mit der ersten Anweisung im Schleifenrumpf weiter ...

Anweisungen abarbeiten



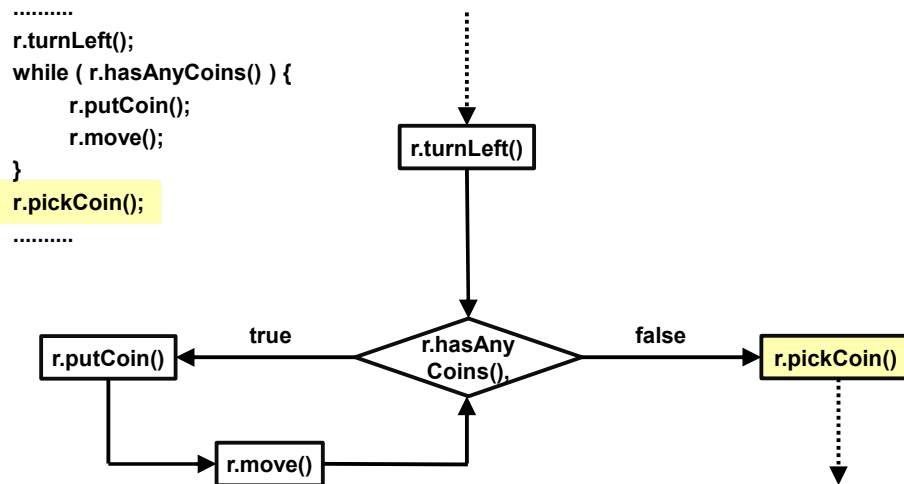
... und dann mit der zweiten Anweisung.

Anweisungen abarbeiten



Nach Abarbeitung des Schleifenrumpfs kehrt der Prozess wieder zur Fortsetzungsbedingung zurück.

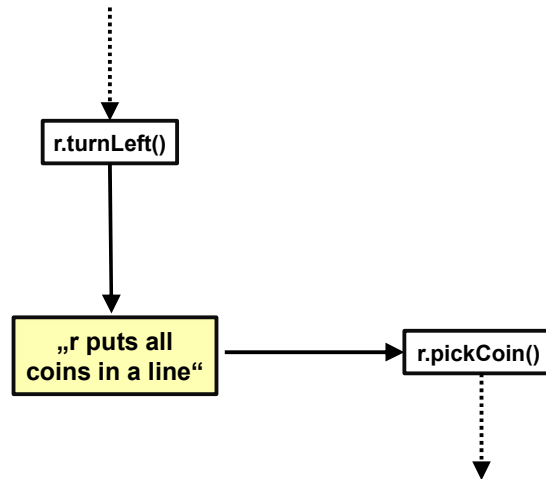
Anweisungen abarbeiten



Falls die Fortsetzungsbedingung true ist, geht es wieder nach links einmal durch den Schleifenrumpf. Ist die Fortsetzungsbedingung hingegen false, springt der Prozess wie hier gezeigt aus der Schleife heraus, also zur nächsten Anweisung unmittelbar *nach* der Schleife.

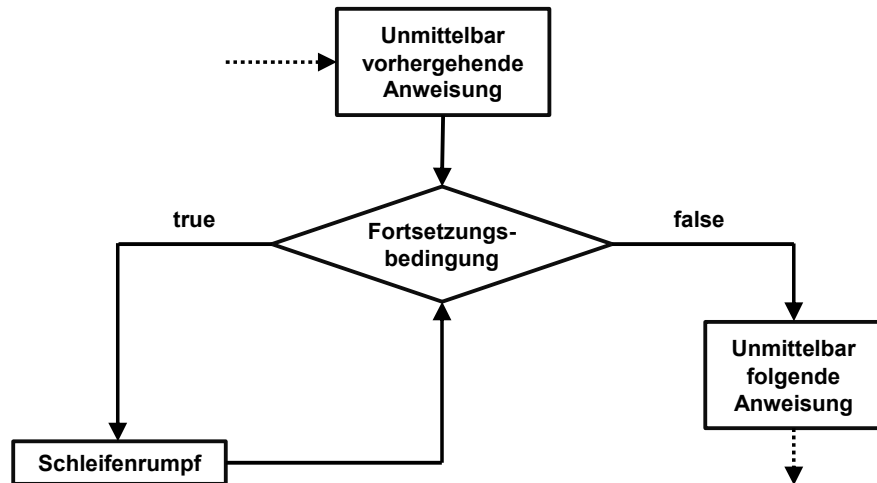
Anweisungen abarbeiten

```
.....  
r.turnLeft();  
while ( r.hasAnyCoins() ) {  
    r.putCoin();  
    r.move();  
}  
r.pickCoin();  
.....
```



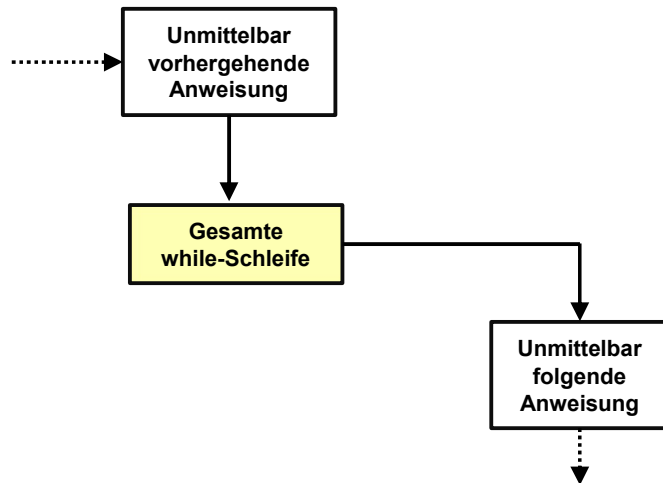
Eine zentrale Einsicht für das Programmieren allgemein ist, dass jede while-Schleife (genauso die gleich noch zu besprechenden weiteren Schleifen und Verzweigungen) als eine einzelne Anweisung gesehen werden kann, die etwas Bestimmtes macht. Tatsächlich subsumiert man Schleifen und ähnliches unter dem Oberbegriff „Anweisung“, um diese Gleichartigkeit zu betonen.

Anweisungen abarbeiten



Noch einmal das Flussdiagramm der allgemeinen while-Schleife, wieder mit vorangehender und folgender Anweisung als zusätzlichen Kontext, nun aber losgelöst von einem konkreten Beispiel, mit den allgemeinen Begrifflichkeiten.

Anweisungen abarbeiten



Nicht nur in Beispielen wie vor zwei Folien, sondern natürlich auch ganz allgemein kann man so eine Schleife als Ganzes wieder als eine Anweisung sehen.

Anweisungen abarbeiten



```
myRobot.turnLeft();  
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.move();  
    myRobot2.putCoin();  
}  
myRobot.turnLeft();
```

1. Führe myRobot.turnLeft() aus
2. Führe myRobot2.
hasAnyCoins() aus
3. Falls das Ergebnis false ist,
springe zu 7
4. Führe myRobot2.move() aus
5. Führe myRobot2.putCoin() aus
6. Springe zu 2
7. Führe myRobot.turnLeft() aus

Bei Schleifen kann es leicht zu Missverständnissen kommen, was die exakten Details der Abarbeitung angeht. Es ist für viele hilfreich, diese Details nicht nur durch ein Flussdiagramm, sondern auch durch ein alternatives Prozessmodell zu klären, das heißt, in diesem Modell formulieren wir Code, der identisch zur Abarbeitung der Schleife ist. Aber die Art der Anweisungen ist primitiver, hardwarenäher und (hoffentlich) weniger missverständlich.

Um die Verständlichkeit zu verbessern, enthält auch dieses Beispiel noch etwas Kontext in Form einer Anweisung unmittelbar *vor* und einer Anweisung unmittelbar *nach* der Schleife.

Anweisungen abarbeiten



```
myRobot.turnLeft();  
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.move();  
    myRobot2.putCoin();  
}  
myRobot.turnLeft();
```

1. Führe myRobot.turnLeft() aus
2. Führe myRobot2.
hasAnyCoins() aus
3. Falls das Ergebnis false ist,
springe zu 7
4. Führe myRobot2.move() aus
5. Führe myRobot2.putCoin() aus
6. Springe zu 2
7. Führe myRobot.turnLeft() aus

Eine einzelne Anweisung sieht in beiden Modellen eigentlich gleich aus.

Anweisungen abarbeiten



```
myRobot.turnLeft();  
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.move();  
    myRobot2.putCoin();  
}  
myRobot.turnLeft();
```

1. Führe myRobot.turnLeft() aus
2. Führe myRobot2.
hasAnyCoins() aus
3. Falls das Ergebnis false ist,
springe zu 7
4. Führe myRobot2.move() aus
5. Führe myRobot2.putCoin() aus
6. Springe zu 2
7. Führe myRobot.turnLeft() aus

Falls die Fortsetzungsbedingung erfüllt ist, werden die Anweisungen im Schleifenrumpf abgearbeitet.

Anweisungen abarbeiten



```
myRobot.turnLeft();  
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.move();  
    myRobot2.putCoin();  
}  
myRobot.turnLeft();
```

1. Führe myRobot.turnLeft() aus
2. Führe myRobot2.
hasAnyCoins() aus
3. Falls das Ergebnis false ist,
springe zu 7
4. Führe myRobot2.move() aus
5. Führe myRobot2.putCoin() aus
6. Springe zu 2
7. Führe myRobot.turnLeft() aus

Rechts sehen Sie unmissverständlich, dass es danach weitergeht mit dem nächsten Test der Fortsetzungsbedingung.

Anweisungen abarbeiten



```
myRobot.turnLeft();  
while ( myRobot2.hasAnyCoins() ) {  
    myRobot2.move();  
    myRobot2.putCoin();  
}  
myRobot.turnLeft();
```

1. Führe myRobot.turnLeft() aus
2. Führe myRobot2.
hasAnyCoins() aus
3. Falls das Ergebnis false ist,
springe zu 7
4. Führe myRobot2.move() aus
5. Führe myRobot2.putCoin() aus
6. Springe zu 2
7. Führe myRobot.turnLeft() aus

Hierhin kommt man nur durch den bedingten Sprung in 3, denn in 6 steht ja ein unbedingter Sprung zurück. Gibt es keine solche Anweisung nach der Schleife, ist die Ausführung des Programms mit der Beendigung der Schleife ebenfalls beendet.

Anweisungen abarbeiten



```
while ( <<Fortsetzungsbedingung>> ) {  
    << Anweisung 1 >>  
    << Anweisung 2>>  
    << Anweisung 3>>  
}  
<<Anweisung 4>>
```

1. Falls die Fortsetzungsbedingung nicht stimmt, springe zu 6
2. Führe <<Anweisung1>> aus
3. Führe <<Anweisung2>> aus
4. Führe <<Anweisung3>> aus
5. Springe zu 1
6. Führe <<Anweisung 4>> aus

So sieht das dann allgemein aus, mit beliebiger Fortsetzungsbedingung und beliebigen Anweisungen.

Es hat sich eingebürgert, solche Platzhalter in doppelte eckige Klammern zu setzen, also Kleiner- und Größer-Zeichen. Wir folgen dieser Konvention in der FOP.

Anweisungen abarbeiten



```
Robot r = new Robot ( 1, 2, UP, 9 );
```

```
do {
```

```
    r.move();
```

```
    r.putCoin();
```

```
} while ( r.hasAnyCoins() );
```

```
r.turnLeft();
```

```
Robot r = new Robot ( 1, 2, UP, 9 );
```

```
r.move();
```

```
r.putCoin();
```

```
while ( r.hasAnyCoins() ) {
```

```
    r.move();
```

```
    r.putCoin();
```

```
}
```

```
r.turnLeft();
```

Es gibt eine kleine Variation der while-Schleife, die do-while-Schleife, die wir uns bei dieser Gelegenheit ebenfalls kurz anschauen. Links oben sehen Sie ein kurzes Quelltextfragment mit einer do-while-Schleife; rechts unten sehen Sie ein Quelltextfragment mit einer while-Schleife, das exakt dasselbe macht.

Anweisungen abarbeiten



```
Robot r = new Robot ( 1, 2, UP, 9 );
```

```
do {
```

```
    r.move();
```

```
    r.putCoin();
```

```
} while ( r.hasAnyCoins() );
```

```
r.turnLeft();
```

```
Robot r = new Robot ( 1, 2, UP, 9 );
```

```
r.move();
```

```
r.putCoin();
```

```
while ( r.hasAnyCoins() ) {
```

```
    r.move();
```

```
    r.putCoin();
```

```
}
```

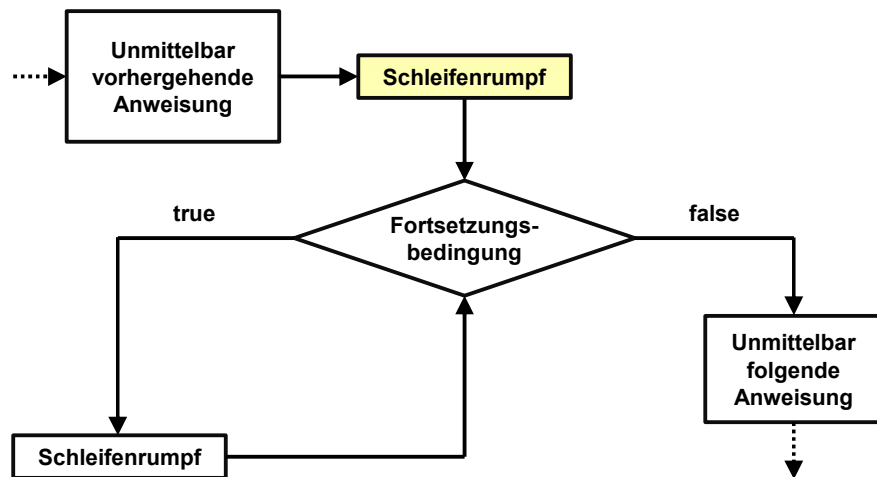
```
r.turnLeft();
```

An der farblich unterlegten Stelle sehen Sie den Unterschied zwischen do-while-Schleife und while-Schleife: Der Rumpf der do-while-Schleife wird auf jeden Fall einmal ausgeführt, denn erst nach dieser ersten Ausführung des Rumpfes wird die Fortsetzungsbedingung erstmals abgeprüft.

Anders formuliert: Bei der while-Schleife kann es sein, dass der Rumpf kein einziges Mal ausgeführt wird, nämlich wenn die Fortsetzungsbedingung gleich beim ersten Test false ergibt; bei der do-while-Schleife kann das hingegen nicht passieren.

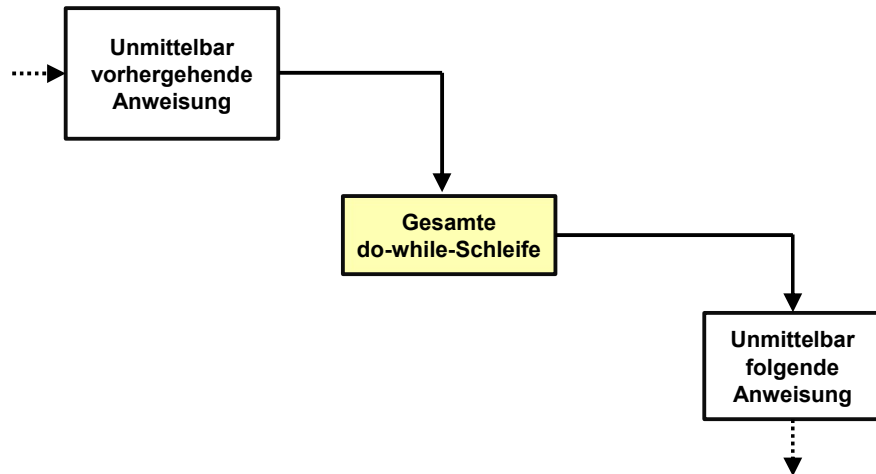
Manchmal ist das eine, in anderen Fällen das andere sinnvoll. Generell ist die while-Schleife häufiger sinnvoll als die do-while-Schleife und wird daher auch häufiger verwendet.

Anweisungen abarbeiten



Hier sehen Sie den Unterschied graphisch: Eine Kopie des Schleifenrumpfes ist eingefügt zwischen der vorangehenden Anweisung und der Fortsetzungsbedingung.

Anweisungen abarbeiten



Beachten Sie, dass das Bild mit wegabstrahierter Schleife exakt dasselbe ist wie bei der while-Schleife vor ein paar Folien (abgesehen vom Label der abstrahierten Anweisung).

Anweisungen abarbeiten



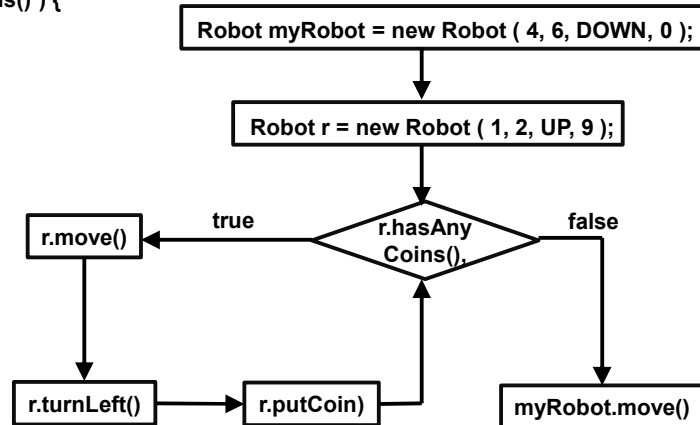
```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );
for ( Robot r = new Robot ( 1, 2, UP, 9 );
      r.hasAnyCoins(); r.putCoin() ) {
    r.move();
    r.turnLeft();
}
myRobot.move();
```

```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );
Robot r = new Robot ( 1, 2, UP, 9 );
while ( r.hasAnyCoins() ) {
    r.move();
    r.turnLeft();
    r.putCoin();
}
myRobot.move();
```

Bei der for-Schleife kann es eher zu Missverständnissen kommen als bei der einfacher gestrickten while-Schleife. Interessanterweise kann man die for-Schleife in Java auf die while-Schleife zurückführen, um diese Missverständnisse von vornherein zu vermeiden. Die Codefragmente oben links mit for-Schleife und unten rechts mit while-Schleife führen exakt dieselben Schritte aus und produzieren daher auch exakt dasselbe Ergebnis.

Anweisungen abarbeiten

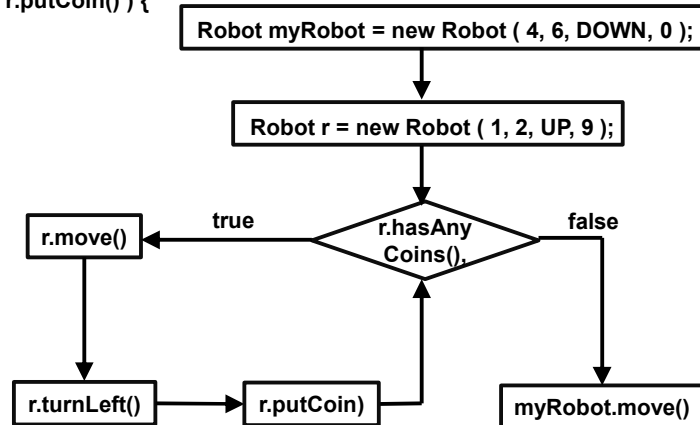
```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
Robot r = new Robot ( 1, 2, UP, 9 );  
while ( r.hasAnyCoins() ) {  
    r.move();  
    r.turnLeft();  
    r.putCoin();  
}  
myRobot.move();
```



Das Codefragment rechts unten auf der letzten Folie sehen Sie nochmals hier oben links. Mit dem, was wir über while-Schleifen und ihre Flussdiagramme wissen, können wir das Flussdiagramm für dieses Codefragment leicht erstellen.

Anweisungen abarbeiten

```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
for ( Robot r = new Robot ( 1, 2, UP, 9 );  
      r.hasAnyCoins(); r.putCoin() ) {  
    r.move();  
    r.turnLeft();  
}  
myRobot.move();
```



Und da wir gesagt haben, dass das Codefragment mit for-Schleife exakt dieselben Schritte macht, ist dieses Flussdiagramm automatisch auch das Flussdiagramm des Codefragments mit for-Schleife auf derselben Folie.

Anweisungen abarbeiten



```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
for ( Robot r = new Robot ( 1, 2, UP, 9 );  
    r.hasAnyCoins(); r.putCoin() ) {  
    r.move();  
    r.turnLeft();  
}  
myRobot.move();
```

```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
Robot r = new Robot ( 1, 2, UP, 9 );  
while ( r.hasAnyCoins() ) {  
    r.move();  
    r.turnLeft();  
    r.putCoin();  
}  
myRobot.move();
```

Nun die Details in Java im Einzelnen. Die runden Klammern nach dem Schlüsselwort for enthalten drei Teile, jeweils separiert voneinander durch Semikolon. Der erste Teil ist im Prinzip äquivalent zu einer Anweisung unmittelbar vor der Schleife.

Vorgriff: Wir werden später sehen, dass es *doch* einen Unterschied zwischen links und rechts gibt, nämlich darin, wo überall r sichtbar ist, Stichwort *Scope* in Kapitel 03a.

Anweisungen abarbeiten



```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );
```

```
for ( Robot r = new Robot ( 1, 2, UP, 9 );
```

```
    r.hasAnyCoins(); r.putCoin() ) {
```

```
    r.move();
```

```
    r.turnLeft();
```

```
}
```

```
myRobot.move();
```

```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );
```

```
Robot r = new Robot ( 1, 2, UP, 9 );
```

```
while ( r.hasAnyCoins() ) {
```

```
    r.move();
```

```
    r.turnLeft();
```

```
    r.putCoin();
```

```
}
```

```
myRobot.move();
```

Diesmal sind wir bequem bei der Benennung des Roboters und nennen ihn einfach nur r.

Anweisungen abarbeiten



```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
for ( Robot r = new Robot ( 1, 2, UP, 9 );  
    r.hasAnyCoins(); r.putCoin() ) {  
    r.move();  
    r.turnLeft();  
}  
myRobot.move();
```

```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
Robot r = new Robot ( 1, 2, UP, 9 );  
while ( r.hasAnyCoins() ) {  
    r.move();  
    r.turnLeft();  
    r.putCoin();  
}  
myRobot.move();
```

Wie der Vergleich mit rechts zeigt, ist der zweite Teil eine Fortsetzungsbedingung, keine Abbruchbedingung, das wird häufig verwechselt. Diese Fortsetzungsbedingung wird wie bei der while-Schleife immer am Anfang des Durchlaufs getestet. Ist die Fortsetzungsbedingung false, geht der Prozess nicht noch einmal in den Schleifenrumpf. Auch bei der for-Schleife kann es wie bei der while-Schleife sein, dass der Schleifenrumpf nicht ein einziges Mal ausgeführt wird, nämlich wenn die Fortsetzungsbedingung schon beim ersten Mal false ist.

Anweisungen abarbeiten



```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
for ( Robot r = new Robot ( 1, 2, UP, 9 );  
    r.hasAnyCoins(); r.putCoin() ) {  
    r.move();  
    r.turnLeft();  
}  
myRobot.move();
```

```
Robot myRobot = new Robot ( 4, 6, DOWN, 0 );  
Robot r = new Robot ( 1, 2, UP, 9 );  
while ( r.hasAnyCoins() ) {  
    r.move();  
    r.turnLeft();  
    r.putCoin();  
}  
myRobot.move();
```

An der dritten Stelle kann im Prinzip eine beliebige Anweisung stehen. Der Vergleich zeigt, dass diese Anweisung in jedem Durchlauf einmal ausgeführt wird, nämlich unmittelbar nach den Anweisungen, die im Rumpf der for-Schleife stehen.

Anweisungen abarbeiten

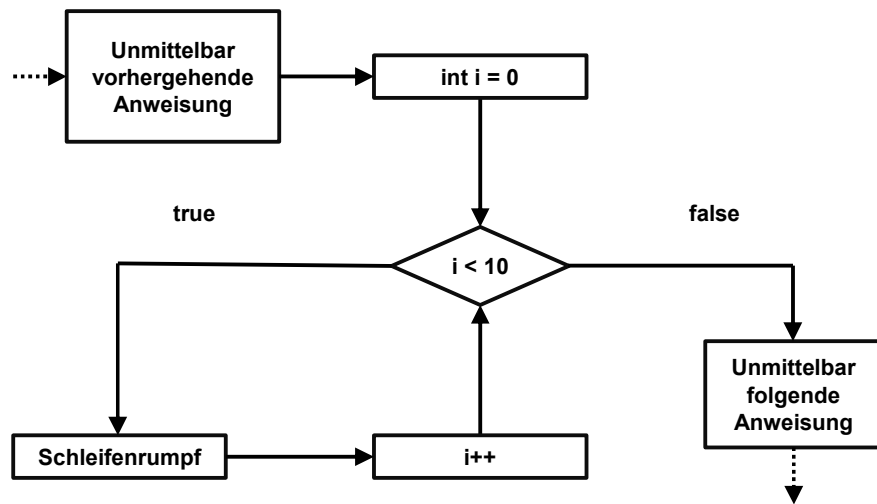


```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

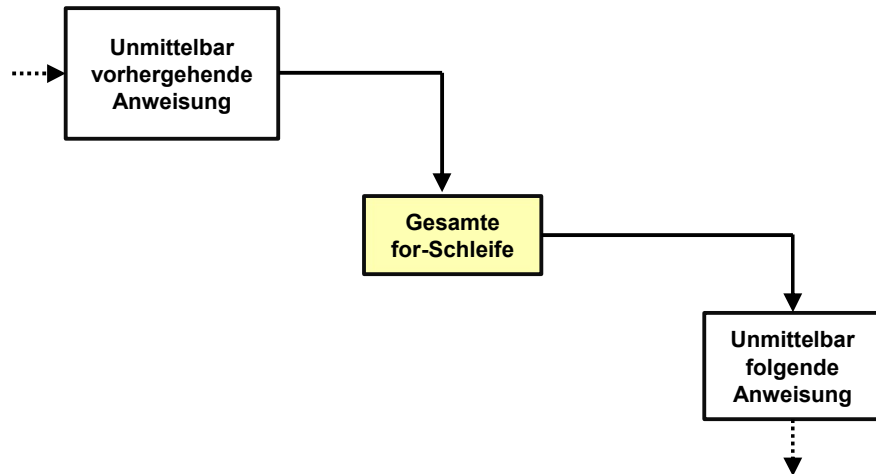
Nachdem wir jetzt ganz allgemein gesehen haben, wie for-Schleifen funktionieren, können wir jetzt auch die spezielle Form der for-Schleife, die wir öfters schon verwendet hatten, vollständig verstehen.

Anweisungen abarbeiten



Zuerst das Flussdiagramm dieser for-Schleife.

Anweisungen abarbeiten



Auch bei der for-Schleife sieht das abstrahierte Bild wieder exakt genauso aus wie bei der while-Schleife und der do-while-Schleife.

Anweisungen abarbeiten



```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

Nun die Details im Einzelnen. Diese Anweisung wird ja nur einmal ausgeführt, unmittelbar vor der eigentlichen Schleife. Sie richtet eine Variable namens *i* vom Typ *int* ein. Im Gegensatz zur Klasse *Robot*, die in *FopBot* hinzudefiniert wurde, ist *int* in Java eingebaut. Eine Variable vom Typ *int* enthält als Wert eine ganze Zahl. Dieser Wert kann positiv, negativ oder 0 sein. Hier wird der Wert von *i* gleich bei der Einrichtung auf 0 gesetzt.

Das Wort „*int*“ steht für *integer* oder *integral*, was im Englischen unter anderem *ganzzahlig* heißt.

Etwas ungenau sagen wir einfach, *i* ist 0, statt genauer zu sagen, dass *i* den Wert 0 hat.

Anweisungen abarbeiten



```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

Die Fortsetzungsbedingung besagt, dass i kleiner 10 sein muss.
Anders herum gesagt: Wenn i größer oder gleich 10 ist in dem
Moment, in dem die Fortsetzungsbedingung geprüft wird, dann wird
die Schleife beendet.

Anweisungen abarbeiten



```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

Das doppelte Pluszeichen hinter dem Namen der Variable sorgt dafür, dass der Wert in i um 1 hochgezählt wird. Am Ende jedes Schleifendurchlaufs – nach Ausführung des Schleifenrumpfs und vor Abtesten der Fortsetzungsbedingung – wird der Wert in i also jeweils um 1 hochgezählt.

Anweisungen abarbeiten



```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

So sieht das dann in den einzelnen Schleifendurchläufen aus.

Zum Zählen der Schleifendurchläufe denken wir uns einen Zähler *k* hinzu, der im Programm nicht auftaucht, sondern eben nur gedacht ist. Naheliegender wäre, *i* zur Zählung zu nehmen, aber da *i* bei 0 anfängt, könnte das verwirrend sein, denn im ersten Durchlauf hat *i* den Wert 0, im zweiten den Wert 1 und so weiter.

Anweisungen abarbeiten



```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

Wir haben schon des Öfteren zehn Punkte zum Anzeigen einer Auslassung verwendet. Bisher waren das Bestandteile des Programms, die aus Platzgründen auf späteren oder vorhergehenden Folien gezeigt wurden. Hier lassen wir zum ersten – natürlich nicht zum letzten – Mal Details aus, weil sie uns schlicht und einfach egal sind: Die Erkenntnisse, die diese Folie vermitteln soll, hängen in keinster Weise vom Inhalt des Schleifenrumpfes ab.

Anweisungen abarbeiten

```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

Zu Beginn des ersten Schleifendurchlaufs ist i noch kein einziges Mal hochgezählt worden, hat also noch den initialen Wert 0.

Anweisungen abarbeiten

```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

Und durch das Hochzählen durchläuft i eben alle ganzen Zahlen aufsteigend mit 0 beginnend.

Anweisungen abarbeiten

```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also 0 < 10	true
2	i ist 1, also 1 < 10	true
3	i ist 2, also 2 < 10	true
...
9	i ist 8, also 8 < 10	true
10	i ist 9, also 9 < 10	true
11	i ist 10, also 10 < 10	false

Zu Beginn jedes Schleifendurchlaufs wird ja die Fortsetzungsbedingung getestet. Wenn wir in Gedanken jeweils den momentanen Wert von i in die Fortsetzungsbedingung einsetzen, sehen wir, dass diese beiden Zahlen jeweils miteinander verglichen werden.

Anweisungen abarbeiten

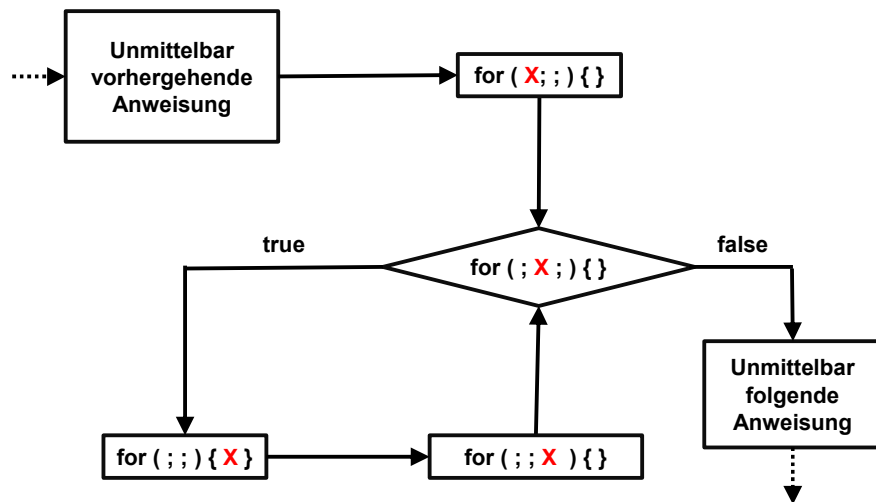


```
for ( int i = 0; i < 10; i++ ) { ..... }
```

k	Vergleich zu Beginn des k-ten Durchlaufs	Fortsetzungs- bedingung
1	i ist 0, also $0 < 10$	true
2	i ist 1, also $1 < 10$	true
3	i ist 2, also $2 < 10$	true
...
9	i ist 8, also $8 < 10$	true
10	i ist 9, also $9 < 10$	true
11	i ist 10, also $10 < 10$	false

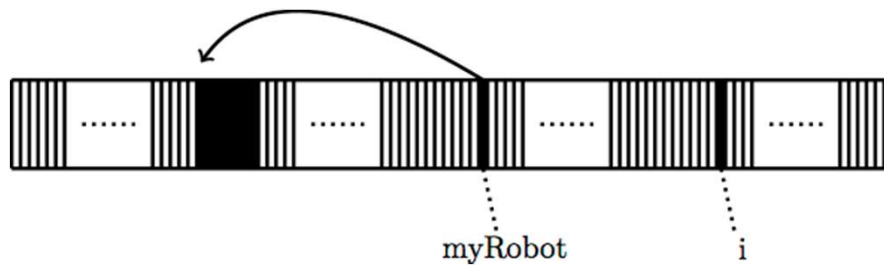
Die Fortsetzungsbedingung ist also zehnmal erfüllt, der Schleifenrumpf wird zehnmal durchlaufen, und nach dem zehnten Durchlauf ist Schluss, da die Fortsetzungsbedingung nicht mehr erfüllt ist.

Anweisungen abarbeiten



Allgemein sieht das Flussdiagramm natürlich genauso aus wie in den Beispielen. Zur Bezugnahme auf die einzelnen Bestandteile der for-Schleife sind diese hier jeweils mit einem roten X markiert und der Rest der for-Schleife ist der Kürze halbe jeweils weggelassen.

Anweisungen abarbeiten



Es wird später immer wieder wichtig sein, genau zwischen Klassen wie Robot und primitiven Datentypen wie int zu unterscheiden, auch was die Ablage im Computerspeicher angeht. Sie sehen hier, dass die Trennung zwischen Objekt und Referenz bei primitiven Datentypen nicht besteht, sondern der Name `i` verweist tatsächlich auf die Speicherstelle, an der die Zahl abgespeichert wird.

Anweisungen abarbeiten



```
for ( ; 9 < 10; ) {  
    myRobot.putCoin();  
    myRobot.move();  
}
```

```
while ( 9 < 10 ) {  
    myRobot.putCoin();  
    myRobot.move();  
}
```

Noch ein Detail zur for-Schleife: An diesem Beispiel sehen Sie, dass die einzelnen Teile in runden Klammern hinter dem Schlüsselwort **for** auch leer sein können, aber die Semikolons müssen trotzdem sein. Unten rechts sehen Sie die äquivalente **while**-Schleife, das heißt, die Leerstellen bei der **for**-Schleife fehlen auch dort ersatzlos.

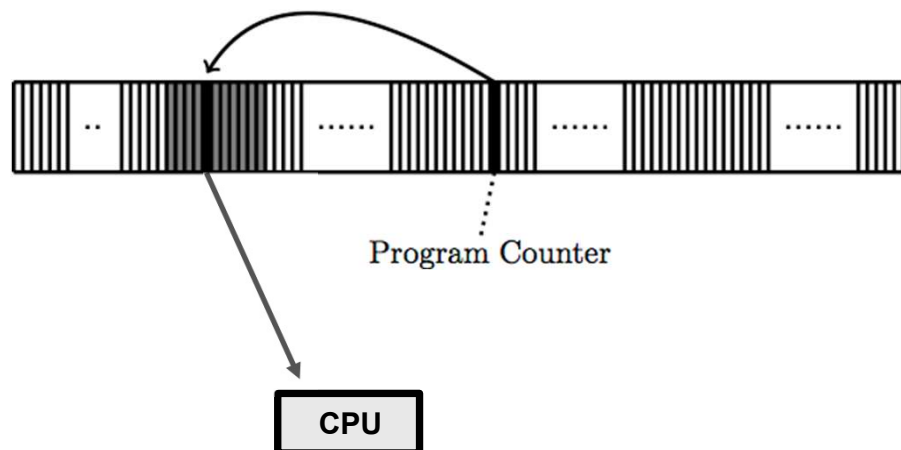
Anweisungen abarbeiten

```
for ( ; ; ) {  
    myRobot.putCoin();  
    myRobot.move();  
}
```

```
while ( true ) {  
    myRobot.putCoin();  
    myRobot.move();  
}
```

Auch der mittlere Teil, die Fortsetzungsbedingung könnte bei der for-Schleife leer bleiben; die leere Fortsetzungsbedingung ist immer erfüllt, also eine Endlosschleife. Ein Prozess auf diesem Programm wird ohne Intervention von außen nicht enden.

Anweisungen abarbeiten

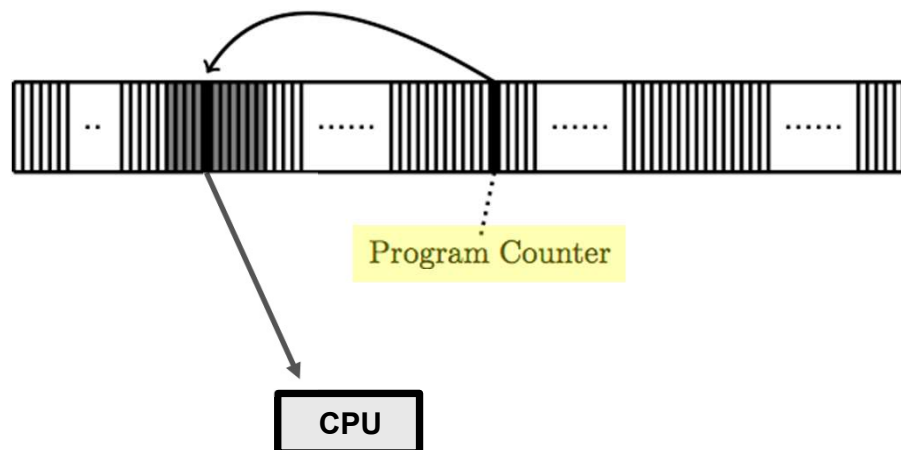


Zum Ende dieses Abschnitts versuchen wir, uns die Abarbeitung von Anweisungen im Computer zu veranschaulichen – realistisch, aber sehr knapp und stark vereinfachend.

Ein Prozess, der vom Computer ausgeführt wird, basiert auf einer Kopie des zugrundeliegenden Programms, also des übersetzten Quelltextes, im Speicher, hier als grauer Adressbereich versinnbildlicht. Das Programm besteht aus einzelnen Anweisungen.

In unserer starken Vereinfachung belegt jede Anweisung genau ein Maschinenwort. Darin ist erst einmal die Art der Anweisung kodiert, also zum Beispiel Addition oder Subtraktion zweier Zahlen oder auch eine Sprunganweisung. Ebenfalls in der Anweisung enthalten sind die Speicheradressen der Operanden, also wo im Speicher die beiden Summanden bei einer Addition zu finden sind oder zu welcher Anweisung – gleich zu welcher Speicheradresse – das Programm springen soll.

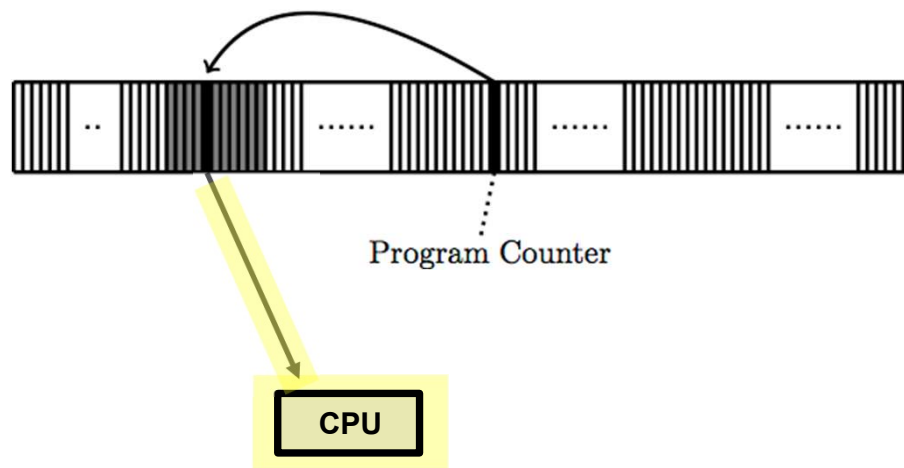
Anweisungen abarbeiten



Der Program Counter ist ein speziell reservierter Speicherplatz, der immer die Adresse der nächsten auszuführenden Anweisung enthält. Nach jeder Anweisung, die keine Sprunganweisung ist, wird der Program Counter einfach hochgezählt und verweist damit auf das jeweils nächste Maschinenwort. Bei einer Sprunganweisung wird der Inhalt des Program Counter überschrieben mit der in der Sprunganweisung bereitgestellten Zieladresse.

Nebenbemerkung: In der Regel sieht es in der Realität so aus, dass die Hardware ein Register für den Program Counter des momentan ablaufenden Programms bereitstellt. Register sind spezielle Speicherplätze separat vom eigentlichen Speicher. Wird ein Programm unterbrochen, weil andere Programme auf diesem Prozessor eine Zeitlang laufen sollen, dann wird der Program Counter des unterbrochenen Programms in der Regel in einer ganz normalen Variable zwischengespeichert, bis das Programm wieder fortgesetzt und der Program Counter ins Register geladen wird.

Anweisungen abarbeiten



Weiterhin stark vereinfachend, wird in jedem Taktzyklus die Anweisung, deren Adresse der Program Counter momentan enthält, in den Zentralprozessor des Computers geladen und dort dann abgearbeitet.

Erinnerung: Die gängige englische Bezeichnung für den Zentralprozessor lautet Central Processing Unit, abgekürzt CPU.

Schlüsselwörter (Keywords)

Oracle Java Spezifikation: 3.9 Keywords

Bevor wir uns im nächsten Kapitel weitere Programme mit FopBot anschauen, sehen wir uns als erstes die Bestandteile von Java-Programmen an, die uns bei den ersten beiden Programmen schon begegnet sind. In Java-Programmen gibt es verschiedene Arten von Bestandteilen. Wir beginnen mit Schlüsselwörtern.

Schlüsselwörter (Keywords)



Beispiele:

**abstract, boolean, break, byte, catch, char,
class, continue, do, double, else, extends,
final, finally, float, for, if, implements, import,
instanceof, int, interface, long, new, package,
private, protected, public, return, short, static,
throw, throws, try, void, while**

Dies hier sind die wichtigsten Schlüsselwörter in Java – die, die auch in den verschiedenen Kapiteln dieser Vorlesung immer wieder auftreten werden, teilweise auch schon in den ersten Abschnitten dieses Kapitels aufgetreten sind.

Schlüsselwörter (Keywords)



Beispiele:

**abstract, boolean, break, byte, catch, char,
class, continue, do, double, else, extends,
final, finally, float, for, if, implements, import,
instanceof, int, interface, long, new, package,
private, protected, public, return, short, static,
throw, throws, try, void, while**

Diese vier Schlüsselwörter haben wir schon verwendet. Schon an diesen fünf Beispielen kann man ganz gut erkennen, was Schlüsselwörter sind und wofür sie da sind: Ein Schlüsselwort kann nur an bestimmten Stellen stehen. An einer solchen Stelle hat das Schlüsselwort eine ganz bestimmte Funktion.

Ein Beispiel: Das Schlüsselwort for beziehungsweise while kann nur am Beginn einer for- beziehungsweise while-Schleife stehen, und die Funktion ist anzuzeigen, dass jetzt eine for- beziehungsweise while-Schleife kommt.

Schlüsselwörter (Keywords)



Beispiele:

**abstract, boolean, break, byte, catch, char,
class, continue, do, double, else, extends,
final, finally, float, for, if, implements, import,
instanceof, int, interface, long, new, package,
private, protected, public, return, short, static,
throw, throws, try, void, while**

Diese fünf Schlüsselwörter hatten wir zwar auch schon gesehen, aber uns noch nicht genauer angeschaut, das machen wir später.

Identifizier (Bezeichner)

Oracle Java Spezifikation: 3.8 Identifiers

Die zweite Art von Bestandteilen von Java-Programmen sind Bezeichner, englisch Identifier. Das sind Namen, die wir selbst wählen können beziehungsweise selbst wählen müssen, wann immer wir eine Entität in ein Java-Programm einführen, die einen Namen bekommen soll.

Im Folgenden sprechen wir immer von Identifiern.

Identifier (Bezeichner)



Namen von

- **Klassen wie Robot**
 - **Variablen wie myRobot, myRobot2, ari und i**
 - **Methoden wie move, turnLeft und hasAnyCoins**
- Später noch viele weitere Beispiele**

Konkret wählen wir also Identifier für verschiedene Entitäten in einem Java-Quelltext.

Identifizier (Bezeichner)



Namen von

- **Klassen wie Robot**

- **Variablen wie myRobot, myRobot2, ari und i**

- **Methoden wie move, turnLeft und hasAnyCoins**

→ **Später noch viele weitere Beispiele**

Wir hatten schon erwähnt, dass wir Entitäten wie Robot *Klassen* nennen. Der Begriff Klasse trifft es sicher ganz gut: myRobot und myRobot2 lassen sich als Roboter *klassifizieren*.

Identifizier (Bezeichner)



Namen von

- **Klassen wie Robot**

- **Variablen wie myRobot, myRobot2, ari und i**

- **Methoden wie move, turnLeft und hasAnyCoins**

→ Später noch viele weitere Beispiele

Solche Entitäten wie myRobot und myRobot2 nennen wir *Variable*.

Vorgriff: Im Kapitel 01b, Abschnitt „Begriffsunterscheidung: Variable, Konstante, Referenz und Objekt“, werden wir den Begriff *Variable* und verwandte Begriffe systematisch betrachten.

Identifizier (Bezeichner)

Namen von

- **Klassen wie Robot**
- **Variablen wie myRobot, myRobot2, ari und i**
- **Methoden wie move, turnLeft und hasAnyCoins**

→ Später noch viele weitere Beispiele

Wir hatten auch schon gesagt, dass wir solche Entitäten wie move und turnLeft und hasAnyCoins generell *Methoden* nennen. Auch die Namen von Methoden sind Identifizier.

Identifizier (Bezeichner)



Namen von

- **Klassen wie Robot**
- **Variablen wie myRobot, myRobot2, ari und i**
- **Methoden wie move, turnLeft und hasAnyCoins**

→ Später noch viele weitere Beispiele

Klassen, Variable und Methoden sind nicht die einzigen Entitäten in Java, für die man Identifier festlegt, aber die einzigen, die wir bisher betrachtet haben.

Identifizier (Bezeichner)



■ Bestehen aus

- 'a' – 'z'
- 'A' – 'Z'
- '0' – '9'
- '_' und '\$'
- Weitere (hier nicht betrachtet)

■ Erstes Zeichen darf keine Ziffer sein

Nach den festen Regeln, die der Java-Compiler erzwingt, darf ein Identifizier neben Groß- und Kleinbuchstaben und Ziffern noch Unterstriche und Dollarzeichen enthalten. Die Konvention sagt aber, dass man die beiden Sonderzeichen nur in bestimmten Ausnahmefällen verwenden sollte. Einen dieser Ausnahmefälle sehen wir gleich.

Identifizier (Bezeichner)



■ Bestehen aus

- 'a' – 'z'
- 'A' – 'Z'
- '0' – '9'
- '_' und '\$'
- Weitere (hier nicht betrachtet)

■ Erstes Zeichen darf keine Ziffer sein

Abgesehen von *einer* festen Regel ist man frei darin, Identifizier beliebig aus den erlaubten Zeichen zu bilden: Es darf nur das erste Zeichen keine Ziffer sein, um Verwechslungen mit ganzen Zahlen zu vermeiden.

Das erste Zeichen kann also prinzipiell ein Klein- oder Großbuchstabe oder eines der beiden Sonderzeichen sein. Jedes weitere Zeichen darf dann auch eine Ziffer sein.

Identifizier (Bezeichner)



Nicht wählbar als Identifizier:

- **Schlüsselwörter**
 - **false, true, null**
-

Wörter mit Sonderbedeutungen sind nicht als Identifizier wählbar.

Identifizier (Bezeichner)

Nicht wählbar als Identifizier:

- **Schlüsselwörter**

- **false, true, null**

Schlüsselwörter sind nicht möglich als Identifizier.

Identifizier (Bezeichner)



Nicht wählbar als Identifizier:

- **Schlüsselwörter**

- **false, true, null**

Hinzu kommen drei Wörter, die nicht Schlüsselwörter sind, aber ebenfalls reserviert. Wir werden auch diese drei Wörter später kennenlernen.

Identifizier (Bezeichner)

While

Bei Identifizieren wird strikt zwischen Groß- und Kleinschreibung unterschieden. *Dies* wäre ein gültiger Identifizier, da das *Schlüsselwort* *while* aus Kleinbuchstaben besteht.

Identifizier (Bezeichner)

While

WHILE

Das wäre aus demselben Grund ein weiterer gültiger Identifizier.

Wegen der Unterscheidung zwischen Groß- und Kleinschreibung sind das zudem zwei *verschiedene* Identifizier. Wir könnten die beiden ohne Namenskonflikte zwei verschiedenen Entitäten zuordnen.

Identifizier (Bezeichner)

While

WHILE

wHiLe

whiLe

whi1e

Und noch drei weitere. Zwischen diesen fünf Identifiern gibt es absolut keine Verknüpfung, sie hätten auch a b c d e heißen können, das macht keinen Unterschied.

Identifizier (Bezeichner)

While

WHILE

wHiLe

whileE

whi1e

while

Was aber definitiv *kein* erlaubter Identifizier ist, ist das Wort **while** ganz in Kleinbuchstaben, denn das ist ein Schlüsselwort.

Identifizier (Bezeichner)



Konventionen:

- **Namen von Variablen, Attributen, Parametern, Methoden u.a. beginnen mit Kleinbuchstaben**
 - myRobot, myRobot2, move, getNumberOfCoins
- **Namen von Klassen u.a. beginnen mit Großbuchstaben**
 - Robot
- **Wortanfänge im Innern: Großbuchstaben**
 - turnLeft, hasAnyCoins, putCoin

Wie schon angedeutet, haben sich gewisse Konventionen für die Wahl von Identifiern in Java entwickelt. Solche Konventionen sehen in verschiedenen Programmiersprachen durchaus unterschiedlich aus. Zum Beispiel sehen sie in der eng mit Java verwandten Sprache C++ anders aus.

Identifizier (Bezeichner)



Konventionen:

- **Namen von Variablen, Attributen, Parametern, Methoden u.a. beginnen mit Kleinbuchstaben**

- `myRobot`, `myRobot2`, `move`, `getNumberOfCoins`

- **Namen von Klassen u.a. beginnen mit Großbuchstaben**

- `Robot`

- **Wortanfänge im Innern: Großbuchstaben**

- `turnLeft`, `hasAnyCoins`, `putCoin`

Für diese Konvention haben wir schon einige Beispiele gesehen, unter anderem die hier aufgeführten.

Identifizier (Bezeichner)



Konventionen:

- Namen von Variablen, Attributen, Parametern, Methoden u.a. beginnen mit Kleinbuchstaben

- myRobot, myRobot2, move, getNumberOfCoins

- Namen von Klassen u.a. beginnen mit Großbuchstaben

- Robot

- Wortanfänge im Innern: Großbuchstaben

- turnLeft, hasAnyCoins, putCoin

Für Namen von Klassen hingegen haben wir bisher nur dieses eine Beispiel gesehen. Weitere werden folgen.

Identifizier (Bezeichner)



Konventionen:

- **Namen von Variablen, Attributen, Parametern, Methoden u.a. beginnen mit Kleinbuchstaben**
 - myRobot, myRobot2, move, getNumberOfCoins
- **Namen von Klassen u.a. beginnen mit Großbuchstaben**
 - Robot
- **Wortanfänge im Inneren: Großbuchstaben**
 - turnLeft, hasAnyCoins, putCoin

Auch für diese Konvention haben wir schon Beispiele gesehen. Auch in anderen Bereichen der Informatik und inzwischen sogar außerhalb der Informatik finden Sie inzwischen Beispiele für diese sicherlich gewöhnungsbedürftige Schreibweise.

Identifizier (Bezeichner)



nameEinerVariablen

nameEinesAttributs

nameEinesParameters

nameEinerMethode

NameEinerKlasse

Nun je noch ein schematisches Beispiel für jede Art von zu bezeichnenden Entitäten. Die Namen sind sicherlich selbsterklärend.

Identifizier (Bezeichner)

nameEinerVariablen

nameEinesAttributs

nameEinesParameters

nameEinerMethode

NameEinerKlasse

Bei den einen ist das erste Zeichen gemäß Konvention ein Kleinbuchstabe.

Identifizier (Bezeichner)



nameEinerVariablen

nameEinesAttributs

nameEinesParameters

nameEinerMethode

NameEinerKlasse

Bei den anderen ein Großbuchstabe.

Identifizier (Bezeichner)

nameEinerVariablen

nameEinesAttributs

nameEinesParameters

nameEinerMethode

NameEinerKlasse

Und jeder Wortanfang im Innern ist ein Großbuchstabe.

Identifizier (Bezeichner)



Ausnahme: Konstanten

**UP
DOWN
LEFT
RIGHT**

Wir haben aber auch schon Identifizier gesehen, die nicht in dieses Schema passen, konkret diese vier.

***Vorgriff:* In Kapitel 01b im Abschnitt zu „Begriffsunterscheidung: Variable, Konstante, Referenz und Objekt“, werden wir solchen Entitäten wie diesen vier systematischer unter dem Namen *Konstanten* begegnen. Im Namen einer Konstante sind per Konvention alle Buchstaben großgeschrieben so wie hier.**

Klammerpaare

Damit verabschieden wir uns insgesamt von den Identifiern und wenden uns einer anderen Art von lexikalischen Elementen zu: den Klammern.

Klammerpaare

()

{ }

[]

< >

Es gibt in Java vier Arten von Klammerpaaren: runde, geschweifte, eckige und spitze.

Klammerpaare

()

{ }

[]

< >

Runde Klammerpaare umfassen zum Beispiel Parameterlisten von Methoden. Aber es gibt auch noch andere Anwendungsfälle.

Klammerpaare

()

{ }

[]

< >

**Geschweifte Klammerpaare umfassen etwa einen Schleifenrumpf,
aber das ist nur ein Beispiel.**

Klammerpaare

()

{ }

[]

< >

Anwendungsfälle für eckige Klammerpaare werden wir in Kapitel 01d erstmals sehen, Stichwort Arrays.

Klammerpaare

()

{ }

[]

< >

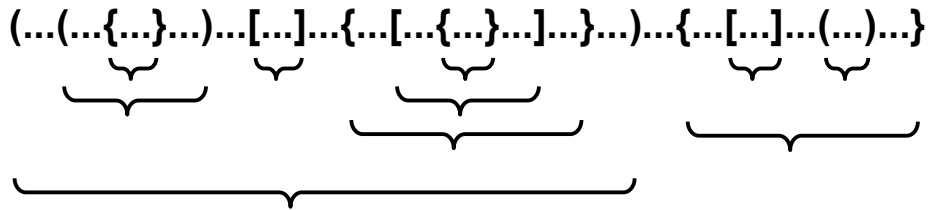
Den Anwendungsfall für spitze Klammern sehen wir hingegen erst in Kapitel 06.

Klammerpaare

(...(...{...}...)...[...]...{...[...{...}...]...}...)...{...[...]...(...)}...

Klammerpaare müssen in bestimmter Relation zueinander stehen, damit das Ganze korrekt ist, das heißt, damit das Programm vom Compiler übersetzt wird, also der Compiler die Übersetzung nicht mit einer Fehlermeldung abbricht.

Klammerpaare



Konkret heißt korrekte Klammerung: Entweder folgt ein Klammerpaar strikt nach dem anderen, oder eines umfasst das andere.

Das ist ja sicherlich auch das, was man intuitiv als korrekte Klammerung ansehen würde.

Klammerpaare

Inkorrekte Klammersetzung

$(\dots[\dots])\dots\{\dots\}\dots\{\dots\}\dots(\dots[\dots])$

Um zu verstehen, was korrekte Klammerung genau ist, ist es instruktiv, sich anhand von ein paar Beispielen anzuschauen, wie *inkorrekte* Klammerung aussieht.

Klammerpaare

Inkorrekte Klammersetzung

(...[...])...{...}{...}{...}(...[...])

Zwei Klammerpaare, die sich teilweise überlappen, sind weder aufeinanderfolgend noch ineinander geschachtelt, also inkorrekt.

Klammerpaare

Inkorrekte Klammersetzung

(...[...])...]{...}...}{...}[...]

Hier fehlt eine öffnende Klammer ...

Klammerpaare

Inkorrekte Klammersetzung

(...[...])...]{...}{...}{...}(... [...]

... und hier eine schließende.

Kommentare

Oracle Java Spezifikation: 3.7 Comments

Kommentare sind ebenfalls lexikalische Einheiten, auch wenn sie für den Compiler keine Funktion haben und von ihm übersprungen werden.

Kommentare

```
int i = 0;    //
```

```
/*
```

```
*/
```

```
/**
```

```
*/
```

Es gibt prinzipiell drei lexikalisch unterschiedliche Arten von Kommentaren in Java.

Kommentare

```
int i = 0; //
```

```
/*
```

```
*/
```

```
/**
```

```
*/
```

Die erste Art wird durch zwei Slashes eingeleitet und kommentiert alles aus von den zwei Slashes bis zum Ende der Zeile.

Kommentare

```
int i = 0;    //
```

```
/*
```

```
*/
```

```
/**
```

```
*/
```

Will man über mehrere Zeilen hinweg etwas auskommentieren oder nur einen Teil einer Zeile, geht das von Slash-Stern bis Stern-Slash.

Kommentare

```
int i = 0;    //
```

```
/*
```

```
*/
```

```
/**
```

```
*/
```

Viele Formatierungsprogramme für Java-Quelltexte, insbesondere das Standardwerkzeug Javadoc, interpretieren einen doppelten Stern nach dem Slash als Aufforderung, diesen Kommentar nach ihren Regeln zu formatieren.

Kommentare

```
/**  
 * This is but an example.  
 * @author Karsten Weihe  
 * @version 3.14159  
 */
```

So sieht beispielhaft ein Kommentar in Javadoc aus. Das Programm javadoc übersetzt Kommentare wie diesen zusammen mit dem eigentlichen Inhalt des Quelltextes in eine recht übersichtlich gestaltete Webseite im Format HTML.

Kommentare

```
/**
```

```
* This is but an example.
```

```
* @author Karsten Weihe
```

```
* @version 3.14159
```

```
*/
```

Hier noch einmal der Anfang und das Ende eines Kommentars, wie eben schon gesehen.

Kommentare

```
/**  
 * This is but an example.  
 * @author Karsten Weihe  
 * @version 3.14159  
 */
```

Jede Zeile des Kommentars wird durch einen Stern eingeleitet.

Kommentare

```
/**  
 * This is but an example.  
 * @author Karsten Weihe  
 * @version 3.14159  
 */
```

Die einzelnen Kommentarzeilen werden durch Javadoc übersichtlich in die HTML-Seite platziert und in geeigneter Schriftart und Schriftgröße dargestellt.

Kommentare

```
/**  
 * This is but an example.  
 * @author Karsten Weihe  
 * @version 3.14159  
 */
```

Eine Reihe von *Tags* sind in Javadoc vordefiniert, zum Beispiel diese beiden. Ihre Bedeutung ist offensichtlich.

Einen Kommentar mit den Tags `@author` und `@version` würde man in der Regel an den Anfang der Datei schreiben.

Whitespaces

Oracle Java Spezifikation: 3.6 White Space

Schließlich noch eine letzte Art von lexikalischen Elementen, die man leicht übersieht, weil man sie auf dem Bildschirm oder auf einem Ausdruck nicht wirklich sieht, die aber zur Trennung anderer lexikalischer Elemente voneinander an manchen Stellen unerlässlich und an anderen Stellen für die Formatierung des Java-Quelltextes zumindest sehr empfehlenswert sind.

Whitespaces

Beispiele:

- Leerzeichen (blank)
 - Zeilenumbruch (newline)
 - Tabulator (tab)
-

Hier geht es jetzt darum, was wir an Leerzeichen, Zeilenumbrüchen und Tabulatorschritten per Tastatur im Editor in den Quelltext einfügen, um Sinneinheiten zu trennen und den Quelltext übersichtlicher zu gestalten.

Whitespaces

- **Mindestens ein Whitespace zwischen:**
 - **Zwei Schlüsselwörtern**
 - **Zwei Identifiern**
 - **Schlüsselwort und Identifier**
- **Ansonsten optional beliebig viele *zwischen* lexikalischen Elementen**
- **Niemals *innerhalb* eines lexikalischen Elements**

Die Frage ist, wo *dürfen* und wo *müssen* Whitespaces sein.

Whitespaces

- **Mindestens ein Whitespace zwischen:**

- Zwei Schlüsselwörtern
- Zwei Identifiern
- Schlüsselwort und Identifier

- **Ansonsten optional beliebig viele *zwischen* lexikalischen Elementen**

- **Niemals *innerhalb* eines lexikalischen Elements**

Wenn Wörter unmittelbar aufeinanderfolgen, die zusammengesetzt ein Identifier sein könnten, dann müssen sie durch mindestens ein Whitespace voneinander getrennt sein, damit es keine Missverständnisse für den Compiler gibt.

Whitespaces

```
int n=0;  
Robot myRobot=new Robot(1,2,UP,3);  
public static void main
```

In diesen drei Zeilen sehen Sie jeweils ein einzelnes Leerzeichen an jeder Stelle, an der mindestens ein Whitespace zwingend stehen muss: ...

Whitespaces

```
int n=0;
```

```
Robot myRobot=new Robot(1,2,UP,3);
```

```
public static void main
```

... zwischen Schlüsselwort und Identifier, ...

Whitespaces

```
int n=0;
```

```
Robot myRobot=new Robot(1,2,UP,3);
```

```
public static void main
```

... sowie zwischen zwei Identifiern (Robot und myRobot) und wieder zwischen Schlüsselwort und Identifier.

Whitespaces

```
int n=0;
```

```
Robot myRobot=new Robot(1,2,UP,3);
```

```
public static void main
```

Diese Abfolge von Wörtern findet sich in den bereitgestellten Dateien, die einzelnen Wörter werden wir aber erst später betrachten. Hier können wir aber schon einmal feststellen, dass public, static und void Schlüsselwörter sind und main ein Identifier ist, so dass dies ein Beispiel für den noch fehlenden Fall ist, nämlich dass auch zwischen zwei Schlüsselwörtern mindestens ein Whitespace sein muss.

Whitespaces

- **Mindestens ein Whitespace zwischen:**

- **Zwei Schlüsselwörtern**
- **Zwei Identifiern**
- **Schlüsselwort und Identifier**

- **Ansonsten optional beliebig viele *zwischen* lexikalischen Elementen**

- **Niemals *innerhalb* eines lexikalischen Elements**

Zwischen zwei aufeinanderfolgenden lexikalischen Elementen dürfen immer Whitespaces sein, und das dürfen auch beliebig viele sein. Diese Regel ist die Grundlage dafür, dass Java-Quelltexte lesbar formatiert werden können mit Leerzeichen, beispielsweise Tabulatoren zur Einrückung einer Anweisung und beliebig vielen Zeilenumbrüchen zwischen zwei Zeilen des Quelltextes.

Whitespaces

- **Mindestens ein Whitespace zwischen:**
 - **Zwei Schlüsselwörtern**
 - **Zwei Identifiern**
 - **Schlüsselwort und Identifier**
- **Ansonsten optional beliebig viele *zwischen* lexikalischen Elementen**
- **Niemals *innerhalb* eines lexikalischen Elements**

Um Missverständnisse für den Compiler zu vermeiden, muss jedes lexikalische Element immer sozusagen am Stück zusammengeschrieben werden.

Damit sind wir mit der letzten Art von lexikalischem Bestandteil und mit lexikalischen Bestandteilen von Java-Quelltexten insgesamt fertig.